The Quasar Computation System: Quick Reference Manual

October 30, 2023

Contents

Chapter 1

Introduction

The Quasar Computation System is optimized to deal with "astronomical" numbers of data values or operations, massively performed in parallel and/or distributed along several processors, hence its name. In the first place, the system is intended to be used for processing of 2D or 3D images, and excels in iterative algorithms that allow for a lot of parallelism. The system consists of three major components:

- Quasar compiler: compiles input code (.q files written in the Quasar scripting language) to an intermediate format, which can either be directly interpreted or translated to Common Intermediate Language (CIL) code (managed executable files). These managed executable files can then be run under Windows (.Net or MONO), Linux (MONO) or Mac (MONO).
- Quasar interpreter: mostly used for debugging code.
- Quasar computation engine: a computation engine performs general (high-level) computations, such as multiplication of real-valued matrices, taking the imaginary part of a complex number, performing FFTs and various built-in functions. Computation engines are substitutable, which means that one engine can take over the work of another engine.[1]

1.1 Computation Engines

Different computation engines exists which take advantage of certain technology present on the system.

1. *Generic CPU computation engine*: makes use of an optimizing C++ compiler (such as GCC, Intel Compiler, ...) in the background and automatically uses OpenMP for multi-threading. This gives a speed up of typically 2x-8x compared to sequential execution.
2. *CUDA computation engine*: uses the CPU for small number of computations (e.g. operations with small matrices), and dynamically switches to GPU computation for larger amount of data, and depending on whether the data currently already resides in GPU/CPU memory.

[1] For GPU computation engines vs. Generic CPU computation engine (see section 1.1), this is done automatically and at any time. For other computation engines, this is only possible by specifying command-line flags, in future versions this may be possible at runtime as well.

3. *Hyperion computation engine*: provides multi-GPU support (see section §11) and allows using OpenCL devices.

4. *Helios computation engine*: a light computation engine, developed in C++, intended for embedded platforms.

The specific details and implementation of the computation engine are completely transparent to the user. More concretely, the user can specify by command line which computation engine to use. For example `-cpu` specifies to use the generic CPU engine, `-gpu` will give the "best" GPU engine for the given system (at least if CUDA/OpenCL is installed). The computation engines perform automatic memory management, i.e. the user is relieved from allocating/freeing memory, and copying memory from/to the GPU. The CPU computation engine (currently) uses a garbage collector, while the CUDA computation engine has a custom fast memory allocator.
The Quasar compiler automatically invokes the NVidia CUDA compiler (CUDA computation engine) or the configured C/C++ compiler (CPU computation engine) for compiling critical parts of the code (so-called device and kernel functions, see further).

1.2 How to use?

One single executable program performs all the work (both compiling and running the code). The usage is as follows:

```
./Quasar.exe [-debug] [-cpu|-gpu] [-profile] [-double] [-nogl] [-make_exe] program.q
```

where the parameters have the following meaning:

- `-debug`: use the interpreter for running the code. In case of failure, exact information on the lines which triggered the error will be given (useful for debugging).
- `-cpu`: uses the generic CPU computation engine for running the code (default=-gpu)
- `-gpu`: uses a GPU computation engine (default choice)
- `-profile`: runs the code in interpreted mode, and collects profiling information. The profiling information is then printed to the console at the end of the program.
- `-double`: instructs the computation engine to use the double precision floating point by default (see section 2.2.1).
- `-make_exe`: builds a managed executable (.exe). The executable can be run using .Net/MONO.
- `-make_lib`: builds a managed library (.qlib), that can be used in other Quasar programs.
- `-nogl`: disables OpenGL support (used for visualization, e.g. the function `imshow`).
- `program.q`: a source code file written in the Quasar programming language, containing the program to run.

When the `-debug` switch is not specified, the compiler produces an executable binary (.exe) which allows the program to be run directly without compilation. The compiler is relatively fast, most (simple) algorithms take a couple of milliseconds to compile.
Note that the GPU computation engine is often 10x to 100x faster then the CPU computation engine. Nevertheless, it is useful to occasionally run the program on the CPU as well, to check the numerical accuracy/precision of the results.

Architecture: 32-bit/64-bit CPU or GPU

Quasar has been designed to operate correctly in the following conditions:

- 32-bit CPU (x86) - the CPU uses a 32-bit address space.
- 64-bit CPU (x64) - the CPU uses a 64-bit address space (useful for addressing more than 2GB of RAM).
- 32-bit GPU - the GPU uses a 32-bit address space.
- 64-bit GPU - the GPU uses a 64-bit address space (when the GPU has more than 2GB RAM, although devices with less than 1GB RAM support it).

By default, the choice of 32-bit/64-bit CPU depends on the OS. If a 64-bit OS is installed, the 64-bit CPU version of Quasar will be used. The mode in which the GPU is run, depends on the installed version of the GPU runtime (e.g., 64-bit or 32-bit CUDA Runtime). The normal practice is to run the GPU in the same mode as the CPU. Under some circumstances, some GPU devices do not support 64-bit yet. For CUDA, this can be solved by using a special 32-bit version of the CUDA interoperability DLL (CUDA.Net.dll), instead of the default cross-architecture DLL.

Important note: since CUDA 7.0 (released in 2015), the 32-bit mode is not supported anymore. Quasar still supports 32-bit modes for backward compatibility (e.g., in combination with CUDA 6.5). However, it is highly recommended to switch to 64-bit versions of Quasar whenever possible.

Supported libraries

A number of libraries have builtin support. These include: OpenGL, FFTW, cuFFT, cuBLAS, cuSolver and cuDNN. It suffices to use the Quasar functions designed to use these libraries in a user-friendly way. For more information, see the CUDA guide.

Distributing Quasar programs

Quasar programs need to be distributed together with the Quasar runtime library. For this purpose, portable Quasar runtime installers are available for Windows and Linux. The portable Quasar runtime installer can e.g. be integrated in your product installer.

1.3 Quasar Programming Language

Motivation for a new programming language for heterogeneous computing From the principle, the right tool for the right job, Quasar aims at simplicity (a low barrier of entry) while aiming at a high performance that is similar to handwritten C++/CUDA/OpenCL code.
Additionally, the Quasar language unifies CPU and GPU programming: one single code path is sufficient to generate optimized versions for both CPU and GPU. This considerably reduces programming effort. In fact, the Quasar compiler can recognize and optimize sophisticated programming patterns (such as parallel reductions, prefix sums, stencil operations etc.) To be able to do so, higher-level information is extracted from the Quasar program. In other programming languages this information is often lost (e.g., because there are no built-in dynamically sized multi-dimensional arrays, the presence of pointers and aliasing conditions hamper compiler analysis, array/vector sizes cannot be statically determined etc.). Quasar then uses target-specific source-to-source optimizations to generate efficient C++/CUDA or OpenCL code. For the final translation to binary, commercial or open-source compilers are used in the background. This also allows benefitting from the low-level optimizations in these compilers.

Furthermore, Quasar offers the low-level flexibility and optimization possibilities of C/C++ together with the high-level rapid testing/development of Octave/Matlab. Additionally, Quasar is user/programmer-friendly and is easy to learn. Of course, there are always certain compromises to be made (e.g., flexibility of programming versus computational cost), but this is where the compiler research and the various tools kick in.

Syntax features The emphasis of the Quasar programming language is on simplicity and practical usefulness. The syntax is similar to MATLAB/Octave (this is mainly to keep the transition from Matlab to Quasar easy), although there are a number of differences which encourage efficient programming:

1. Objects (such as matrices, cell matrices etc) are passed by *reference* rather than by *value*. This means that a simple assignment a=b has negligible computation cost, since it only involves copying pointers. However, one has to be careful with function calls: when passing a matrix as an input argument, the function is allowed to modify the input parameter.[2] This is mainly for efficiency reasons. On the other hand, scalar numbers (real or complex) are passed by value at any time.

2. Zero-based indexing. All indices start with 0, similar to C/C++, Java, C#, ...

3. Some improved syntax (similar to GNU OCTAVE): lambda expressions, indexing of the results of a function call (like `imread(file)[0..100,0..100]`), ...

Advantages In general, Quasar has the following advantages:

1. *Uniform programming model for CPU and GPU.* Unlike some other programming models, in Quasar it is not necessary to implement separate functions for different target devices (for example, a CPU implementation and a GPU implementation). In fact, the same code is targetted toward heterogeneous compute devices. For this purpose, compiler transformations specialize the code for the target architecture. When desired, it is possible to write target-specific code, but in practice this is rarely needed.

2. *Compact and easy to learn programming language.* Quasar code is simple to develop using the Quasar Redshift IDE. An easy learning curve, together with various integrated debugging and visualization tools allow a novice to get started really quickly. Compiler errors and warnings have been optimized to be as informative and helpful as possible.

3. *Access to low-level parallelization primitives* (through kernel and device functions). Kernel and device functions are compiled natively using existing C++/CUDA compilers (e.g., CUDA NVCC, GCC, MSVC or any other C++ compiler). Quasar code can therefore be seen as a thin layer on top of C++ or CUDA.

4. *High-level programming.* Loop parallelization and kernel generation convert the code to low-level kernels. The high-level programming approach not only increases productivity, but it also stimulates writing concise and readable code. This simultaneously reduces the chance for bugs.

5. *Transparent use of CPU / GPU resources.* Essentially, no knowledge on GPU programming is required. However, knowledge on *parallel* programming is a must!

6. *Automatic concurrent kernel execution.* Automatic assignment of CUDA streams is generally a tedious task. The runtime system automates this task, as a result, kernel launches and memory copies can overlap whenever the compute device resources and data dependencies allow it.

[2] If the intention is to copy the values of objects, the function copy(.) can be used to perform a deep copy of objects.

7. *Lightweight runtime system* with minimal runtime overhead. As long as the bulk of the computations is done within kernel/device functions, the runtime overhead is negligible. The execution time is often similar to handwritten C++ or CUDA code.

8. *Dynamic runtime scheduling.* The runtime system offloads computations to be best suitable (and available) device.

9. *Automatic memory management* and memory transfers: there is no need to worry about deallocation, dangling pointers. Additionally, the runtime system makes sure that the memory is transferred to the right device at the right time.

10. *Hardware agnostic programming.* In general, the Quasar programming model is hardware-agnostic, so that the code does not depend much on the features of the (GPU) hardware. When desired, low level primitives (e.g., shared memory, thread synchronization, textures, ...) can be accessed.

11. *Easy access to low-level CUDA features.* such as textures, surfaces, cooperative threading, warp shuffling, shared memory and thread synchronization.

12. *Builtin OpenGL interoperability* and visualization features. OpenGL interoperability allows access to data allocated in CUDA, which is useful for efficient visualization. There is the possibility of generating both texture and vertex data from Quasar, which allows creating e.g., advanced 3D plots.

13. *Future-proofness*: older Quasar programs automatically use GPU feature of newer GPU architectures. You only need to update to the latest Quasar version

1.4 Integration with foreign programming languages

Many existing code bases exist, therefore Quasar can be seamlessly integrated with several programming languages. In this section, we provide a quick overview of integration techniques. For more details, see the external interface reference.

1. *.Net languages* (C#, F#, IronPython, ...): the Quasar .Net host API can be used to access Quasar features (including running Quasar libraries or binaries generated using Quasar).

2. *Java*: a Java bridge has been developed by the Flemish Institute for Biotechnology (VIB) and will soon be available *open source.*

3. *C++*: the Quasar C++ host and DSL APIs offer an extensive set of runtime features to allow either Quasar programs to be used as libraries in C++ projects, or C++ libraries to be called from Quasar. Furthermore, the Helios system allows transpiling Quasar code to C++ which can be linked with existing C++ modules.

4. *Python:* the *pyQuasar* Python-Quasar bridge acts as both a library to Quasar and a class extension to Python, allowing Quasar functions to be called from Python, and vice versa. pyQuasar also maps NumPy arrays onto Quasar arrays.

Chapter 2

Getting started

In this section, we will give you a little tutorial, to give you an idea of the Quasar programming language. First, a number of high-level concepts are listed. These concepts are included mainly to ease programming in Quasar. The most interesting parts are discussed in Section "writing parallel code" (section 2.4).

2.1 Quasar high-level programming concepts

1. *Variables*: Quasar variables are (by default) weakly-typed, although some mechanisms exist to enforce strong-typing. Variable names and function names are case sensitive.
2. *Data types*: some of the built-in data types are listed here:
   - `scalar`: specifies a floating-point number. The used precision depends on the settings of the computation engine.
   - `cscalar`: specifies a complex-valued scalar number
   - `vec`: a dense vector (1D array) of arbitrary length (the size is limited by the system resources)
   - `mat`: a dense matrix (2D array) of arbitrary size (the size is limited by the system resources)
   - `cube`: a dense cube (3D array) of arbitrary size (the size is limited by the system resources)
   - `cvec`: a complex valued dense vector
   - `cmat`: a complex valued dense matrix
   - `ccube`: a complex valued dense cube
   - `string`: a string expression
   - `cell`: a cell matrix object
   - `kernel_function`: represents a reference to a kernel or device function (see further).
   - `function`: a reference to a Quasar (user) function or lambda expression
   - `object`: a user-defined object (see function `object`())

Note that specific functions (see further) need to be used to create variables of a given type. There are also some special built-in datatypes: `vec`x and `cvec`x, with x=1,...,32 specify a vector of length x.

3. *Scalar numbers*: scalar numbers can be entered in decimal notation (`5.678`) as well as in scientific notation (`-1.9e-4`). Imaginary numbers are defined by adding the suffix `j` (or `i`), hence `1+1j` or `1-1j` represent complex numbers. Non-decimal numbers are also supported: for example, binary numbers `1011011b` (suffix `b` or `B`), octal numbers `123456o` (suffix `o` or `O`) and hexadecimal numbers `1Fh` or `0ECD3Fh` (suffix `h` or `H`).

4. *Integer numbers*: Quasar supports integer numbers (type `int`). The bit length of the `int` type depends on the computation engine, but is typically 32-bit. Also integer types with specified bit length exist: `int8`, `uint8`, `int16`, `uint16`, `uint32`.

5. *% Comments are cool*
   However, note that multi-line comments are currently not (yet) supported.

6. *Assignment expressions*:

```
cool = 1
quasar = cool
```

   The separation of lines using ";" is *optional*, and only mandatory when multiple statements are placed on the same line. One can assign to multiple variables at once, similar to C/C++:

```
a = b = 1
```

   also, the result of an assignment is a value (in this case, 1). Multiple variable assignment is also possible (i.e. assigning multiple values to multiple variables at once). For example:

```
[a, b] = [1, 2]
```

   will assign 1 to a and 2 to b. It is equivalent to:

```
a=1; b=2
```

   The multiple variable assignment is mostly useful for 1) assigning multiple return values from functions, for interchanging values:

```
[a, b]=[b, a]
```

   will swap the values of a and b. In some cases, it may be useful to neglect a certain return value. This can be done using the placeholder `_`:

```
[a, _] = [1,2]
[_, b] = [1,2]
```

7. *Arrays (vectors, matrices, cubes etc.)* The data type used may depend on the settings of the computation engine (currently, only 32-bit floating point is allowed, for efficiency). The following program illustrates how to create vectors and perform operations:


```
a = [0, 1, 2, 3] + 4
b = [3, 3, 3, 3]
print "a = ", a, "a .* b = ", a .* b, "sum(a.*b) = ", sum(a .* b)
print "a^2 = ", a.^2
```

String expressions are defined by double quotes (“”), the function “print” allows to print several comma separated values to the console. The “sum” function computes the sum of all components of the vector. The first line evaluates to

```
a = [4, 5, 6, 7]
```

i.e., 4 is added to every component of the vector. `[4, 5, 6, 7]` then represents a row vector. A matrix can be defined as follows:

```
a = [[1,2],[2,1]]
```

Statements and expressions can be split across multiple code lines. The following is also valid:

```
a = [[1,2],
     [2,1]]
```

However, for readability, it is adviced to put an underscore at the line break:

```
a = [[1,2], _
     [2,1]]
```

Similarly, a 3D matrix can be defined by:

```
a = [[[1,2],[3,4]],[[5,6],[7,8]]]
```

An alternative way of defining matrices is using the function `zeros`(.) or `ones`(.), which will initialize the values of the matrix to 0 and 1, respectively:

```
a = zeros(5)
b = zeros(6, 4)
c = zeros(8, 6, 4)

d = ones(5)
e = ones(6, 4)
f = ones(8, 6, 4)
```

Alternatively, a vector with the dimensions can be used:

```
dims = [4, 5, 6]
a = ones(dims)
```

The function `size`(.) returns the size of a vector/matrix/cube:


```
dims = size(a)
dim_y = size(a,0)
dim_x = size(a,1)
dim_z = size(a,2)
[dim_y,dim_x] = size(a,0..1)
[dim_y,dim_x,dim_z] = size(a)
```

Note that the dimensions are zero-based. By convention, y is the first dimension (corresponding to index 0), x is the second dimension and z is the third dimension. Internally, matrices are stored in row-major order.[1] An $n \times n$ identity matrix can be created by using the function eye(.):

```
Q = eye(n)
```

Another example:

```
a = [[1,2],[2,1]]
b = [[3],[4]]
a[0,0] = 2
print a * b, ",", eye(3)
print a[0,0]
print "size(a)=", size(a), "size(a,1)=", size(a,1)
```

The function `numel(.)` gives the number of elements of a vector/matrix/cube. Practically: `numel(X) = prod(size(X))`.

8. *Operators*: see table 2.1.

   Notes:

   - There are no bit-wise integer *operators*. Instead, use the functions `and` (bitwise conjunction), `or` (bitwise disjunction), `xor` (exclusive or), `not` (bitwise negation), `shl` (bit-wise left shift), `shr` (bitwise right shift).
   - Within kernel or device functions, the operators +=, `-=` have a special meaning: they specify *atomic* operations (i.e. these operations are free of data races). There are currently 13 atomic operators (see table below).

9. *Sequences*: a sequence defines a row vector:

```
a=0..9
b=0..2..6
```

The middle argument defines the step size. Generally, the sequence *includes* the specified lower and upper bounds.[2] Hence, the above statements are equivalent to:

```
a=[0,1,2,3,4,5,6,7,8,9]
b=[0,2,4,6]
```

The sequences can subsequently be used for matrix indexing:

[1] This is in contrast to MATLAB, which uses column-major order (i.e. FORTRAN order).
[2] Except when the step size is too large, such as `0..2..3 = [0, 2]` or `0..100..10 = [0]`.

Table 2.1: Some operators

=	assignment	!	inversion (of Boolean values)
+	add	&&	Boolean AND
-	subtract / negation	||	Boolean OR
*	matrix multiplication (or multiplication of scalar values)	? :	Conditional expression (similar to C/C++)
/	division of scalar values	+=	a += b is a shorthand for a = a + b
.*	point-wise multiplication (vec, mat, cube data types)	-=	a -= b is a shorthand for a = a - b
./	point-wise division (vec, mat, cube data types)	*=	a *= b is a shorthand for a = a * b
^	exponentiation (scalar values currently)	/=	a /= b is a shorthand for a = a / b
.^	point-wise exponentiation	^=	a ^= b is a shorthand for a = a ^ b
<	smaller than	.*=	a .*= b is a shorthand for a = a .* b
<=	smaller than or equal	./=	a ./= b is a shorthand for a = a ./ b
>	greater than	.^=	a .^= b is a shorthand for a = a .^ b
>=	greater than or equal	^^=	Atomic maximum
==	equality	__=	Atomic minimum
!=	inequality	~=	Atomic bitwise exclusive or (XOR)
..	Defines a sequence (see further)	|=	Atomic bitwise OR
		&=	Atomic bitwise AND

```
A=randn(64,64)
A_sub = A[a,b]
```

Example:

```
a = 1..2..10
b = sum(a)
c = linspace(1, 2, 5)
print "a = ", a, "c = ", c
print sum = ", [b, sum(c)]
```

The linspace function creates an uniformly spaced row vector of 5 values between 1 and 2, hence c = [1,1.25,1.5,1.75,2]. Implicit sequences (`:`) can be used to quickly index matrices:

```
print A[:,0], A[0,:]
```

This statement prints the first column of A, followed by the first row of A. Note that for the Matlab keyword "end", there is no Quasar equivalent. However, it is still possible to use `A[0..size(A,0)-1,0]`.

10. *Control structures*: Quasar supports several control structures:

```
for a=0..2..4
        break
        continue
endfor

if a==2
endif

if a==2
   ...
elseif a==3
   ...
else
   ...
```



```
endif

while expr
       break
       continue
endwhile

repeat
    break
    continue
until expr
```

Note that "`if`" is ended with "`endif`". Also "`if`", "`endif`" statements *must* be spread along several lines of code. This is to improve readability of the code. The following is NOT allowed:

```
if a==2; do_something(); endif % Not allowed!
```

An example of a for-loop:

```
for i=1..2..100
    j=i+1
    print i, " ", j
    if i==1
        print "i is one"
    elseif i==3
        print "i is three"
    endif
endfor
```

Non-uniform ranges can be specified as follows:

```
for powerOfTwo=[1,2,4,8,16,32,64,128]
    print powerOfTwo
endfor
```

or more conveniently as:

```
for powerOfTwo=2.^(1..7)
    print powerOfTwo
endfor
```

11. Switches are also possible, the syntax is a little different, for example:

```
match a with
| 1 ->
    print "a=1"
| 2 ->
    print "a=2"
| (3, 4) ->
    print "a=3 or a=4"
| "String" ->
    print "a=String"
```



```
| _ -> print
    "a is something else"
endmatch
```

Note that different data types (i.e. strings and scalar numbers) can be mixed. Multiple case values can be specified (grouped by parentheses).

12. *Ternary operators*: an inline if is also available, just like in C/C++:

```
y = condition ? true_value : false_value
y = (x > T) ? x - T : 0
```

When the condition is true, only the value for true is evaluated. Conversely, when the condition is false, only the false-part is executed.

13. *Lambda expressions*: simply speaking, lambda expressions define inline functions, for example:

```
v = (x,y) -> 2*x+y
u = x -> 2*x
w = x -> y -> x + y
z = w(10)
print v(1,2), " ", z(5)
a = [[1,2],[2,1]]
print v(a,a)
print w(4)(5)
```

Note that here, `w` is a lambda expression that returns another lambda expression (`y -> x + y`) when evaluated. As such, partial evaluation is possible, e.g., `z=w(10)` (see further in section 4.10). Lambda expressions can contain several sub-expressions and can be spread over several lines, as follows:

```
print_sum = (a, b) -> (sum=a+b;
                       print(sum); sum)
```

The different expressions are separated using semicolons (';'). The return value of the lambda expression is always the last expression (in the above example, `sum`). Using ternary operators, it is fairly simple to define recursive lambda expressions:

```
factorial = x -> x > 0 ? x * factorial(x - 1) : 1
```

14. *Functions*: the syntax for functions is different from the syntax for lambda expressions:

```
function [outarg1, ..., outargM] = name (inarg1, ..., inargN)
```

Here there are M output arguments (outarg1, ..., outargM) and N input arguments (inarg1, ..., inargN). "name" is the name of the function. Note that all output arguments must be assigned, otherwise the function call fails. An example:


```
function y = do_something(x)
    y = x * 2
endfunction
a = [[1,2],[2,1]]
b = do_something(a)
print b
```

Calling a function with multiple output arguments requires multiple variable assignment:

```
function [x, y] = compute(a, b)
    x = a + b
    y = a * b
endfunction
[u, v] = compute(2,3)
print u, " ", v
```

Functions can contain inner functions (up to arbitrary nest depths). The inner functions (direct childs, not siblings) can then only be accessed from the outer function. For example:

```
function y = colortransform (x : vec3, cname)
    function a = hsv2rgb (c)
        h = floor(c / 60)
        f = frac(c / 60)
        v = 255 * c
        p = v * (1 - c)
        q = v * (1 - f * c)
        t = v * (1 - (1 - f) * c)
        match h with
        | 0 -> a = [v, t, p]
        | 1 -> a = [q, v, p]
        | 2 -> a = [p, v, t]
        | 3 -> a = [p, q, v]
        | 4 -> a = [t, p, v]
        | _ -> a = [v, p, q]
        endmatch
    endfunction
    if cname=="hsv2rgb"
       y = hsv2rgb(x)
    else
       error "the specified color transform ",cname, " is not supported!"
    endif
endfunction
```

Argument types can optionally be specified, as shown above in `x : vec3`. Quasar will check at compile-time (and run-time) if the arguments are of the correct type, otherwise an error will be raised. The presence or absence of argument types has no further influence on the execution and end result of the program (except when types do not match and an error is generated). However, specifying argument types can help the Quasar optimizer to generate more efficient code.

Function handles can also be used, for example:


```
my_func = colortransform
print my_func([0.2, 0.2, 0.3])
```

15. *Optional* function arguments: functions can have optional arguments. In case an argument is missing, the default value is used. For example:

```
function [y, k] = my_func(b : mat, a : scalar = 4)
    print a + b
    y = k = 0
endfunction
my_func(2)
```

Since `my_func` is called with one argument, the default value for the second argument will be used (4 in this case).
Hence, functions can have multiple outputs and optional arguments, whereas lambda expressions can not. Note that the optional function arguments can - on their turn - be expressions and even function calls:

```
function [y, k] = my_func(b : mat, a : mat = eye(4))
function [y, k] = my_func(b : mat, a : mat = A .* B)
function [y, k] = my_func(b : mat, a : mat = 2 * b)
```

Note that by default, variable references (if the name does not correspond to another input argument) refer to the outer context in which the function is defined. They capture the value at the time the function is defined. The variables are defined in the order that they are put as argument. The following would lead to an error:

```
function [y, k] = my_func(a : mat = 2 * b, b : mat)
```

Here, `b` is not defined at the time `a = 2 * b` is evaluated.

16. *Cell matrices*: vectors, matrices and cubes can be grouped in a cell-structure. Cell matrices are either created using the function `cell` or using the special designated quotes ''. `copy`(.) performs a deep copy of a cell matrix (i.e. the function recursively applies `copy`(.) to all its elements). Some examples are given below:

```
A = cell(2,2)
G = cell(3)
A[0,0] = eye(4)
A[1,1] = 3
A[0,1] = cell(1,4)
A[0,1][1] = ones(3,3)
A[0,1][1][0,0] = 2
print A[0,1][1]*3
print size(A)*2
B = copy(A)
C = B-A
print C[0,0]
D = {A,B,C}
D_names = {"A","B","C"}
print D_names[1]
```



```
print size('')
```

One important special feature is that operations on cell matrices are supported when the different operands have the same structure. It is possible to compute the sum of two cell matrices using:

```
C = B+A
```

Althernatively, we can multiply all elements of a cell matrix by a constant:

```
C = B*4
```

Or, we can use cell matrices in function calls (note that this is only allowed with built-in functions).

```
C = max(B,4)
```

Cell matrices are convenient structures especially for rapid prototyping. However, because cell matrices can store *any* data, type inference that is required for efficient parallelization may fail on cell matrices. For this purpose, it is useful to use *fully typed* cell matrices (see section §3.5).

17. *Dynamic evaluation*: string expressions can be parsed and evaluated at runtime using the `eval`(.) function:

```
val = eval("(x) -> 3*eye(x)")(8)
```

Here, the eval function parses the string expression "`(x) -> 3*eye(x)`´´ and returns a corresponding lambda expression. This lambda expression can then be evaluated at the same speed as "regular" lambda expressions. This can be useful for simulations (e.g. passing functions through the command line).

18. Reading an input image:

```
img_in = imread("lena_big.tif")
```

Grayscale images return a two-dimensional matrix, color images return a three-dimensional cube, in which the length of the third dimension is either 3 (RGB) or 4 (RGBA - RGB with an alpha channel).

19. The spread operator: the spread operator "...´´ allows to unpack vectors to arbitrary indices or function parameters. Using the spread operator, the following lines of code can be simplified:

```
pos = [0,1,2]
y = im[pos[0],pos[1],pos[2],0] % Before
y = im[...pos, 0] % After

luminance = (R,G,B) -> 0.2126 * R + 0.7152 * G + 0.0722 * B
c = [128, 42, 96]
lum = luminance(c[0],c[1],c[2]) % Before
lum = luminance(...c) % After
```

The spread operator is in particularly useful in combination with variadic functions (see section §4.8).

Importing .q files: .q files can contain multiple variable and function definitions which can be accessed from other .q programs. To do so, the import keyword can be used. The import keyword should be used only at the global scope (hence not within functions or control structures) and at the beginning of the Quasar module. For example:

```
import "system.q"
import "imfilter.q"

% all definitions from system.q and imfilter.q are now available.

im = imfilter(imread("img.tif"),ones(7,7))
```

There is one exception: "main" functions are completely skipped and hence not imported (see section 13.1). Also, .q files are only to be imported once (multiple imports will have no effect and will be ignored by the compiler), and the import definitions must be placed on the beginning of the program.

Syntax notice: the control structure keywords are as follows: `if` → `endif`, `for` → `endfor`, `type` → `endtype`, `while` → `endwhile`, `match` → `endmatch`, `function` → `endfunction`, `try` → `endtry`. In older versions of Quasar, the keywords were: `if` → `endif`, `for` → `end`, `type` → `end`, `while` → `end`, `match` → `end`, `function` → `end`,`try` → `end`. We have found experimentally that matching the control structure endings with the beginnings enhances not only the readability of the code but also prevents certain types of bugs (especially with nested control structures). For legacy code, the Quasar compiler still accepts the old endings. This is done on a per-file basis. When both control structure endings are mixed, the compiler generates an error. So the user is encouraged to use the new control structure endings!

Table 2.2: Quasar main primitive types. Note: to use the types with asterisk(*), it is required to import the module "`inttypes.q`"

type	0-dim	1-dim	2-dim	3-dim	n-dim
integer number	`int`	`ivec`(*)	`imat`(*)	`icube`(*)	`icube{n}`(*)
shorthand for		`vec[int]`	`mat[int]`	`cube[int]`	`cube{n}[int]`
scalar number	`scalar`	`vec`	`mat`	`cube`	`cube{n}`
shorthand for		`vec[scalar]`	`mat[scalar]`	`cube[scalar]`	`cube{n}[scalar]`
complex scalar number	`cscalar`	`cvec`	`cmat`	`ccube`	`ccube{n}`
shorthand for		`vec[cscalar]`	`mat[cscalar]`	`cube[cscalar]`	`cube{n}[cscalar]`

2.2 A brief introduction of the type system

Note: a full depth explanation on the Quasar user-defined types will be given in section §3. Here we only give a brief introduction.

Quasar has an array-based type system, that facilitates working with multi-dimensional data, which includes for example conversions between vectors and matrices. In general, variable types are *implicit* (hence do in general not need to be specified by the user). In contrast to the MATLAB/Octave, the Quasar compiler obtains the types of the variables through *type inference*. The type inference is not *strict*: if the compiler is not able to figure out the type of a variable, this variable will considered to be of an *unknown* type (denoted by the type `'??'`). The main primitive types of Quasar are summarized in table 2.2. The types `vec`, `mat`, `cube`, `cvec`, `cmat`, `ccube`, ... are actually shorthands for their corresponding generic versions with explicit type parameters (see further in section §6). Also the shorthands are listed in the table. Additional primitive types are given in table 2.3.

The dimensionality can be specified via the brace syntax: e.g., `cube{4}` denotes a four-dimensional array. Quasar defines "infinite" dimensional data types: `cube{:}` and `ccube{:}`, although in practice, the current implementation only supports up to 16-dimensional data structures.

The relation between the different "dimensional" types is defined as follows:

$$\texttt{vec} \subset \texttt{mat} \subset \texttt{cube} \subset \texttt{cube\{:\}}$$

$$\texttt{ivec} \subset \texttt{imat} \subset \texttt{icube} \subset \texttt{icube\{:\}}$$

$$\texttt{cvec} \subset \texttt{cmat} \subset \texttt{ccube} \subset \texttt{ccube\{:\}}$$

Hence, every vector can be passed to a function requiring a matrix, and every matrix can be passed to a function requiring a cube. Whether a value `A` is vector, matrix, or cube, depends on the number of dimensions of `A`:

$$\texttt{A} \text{ has type } \begin{cases} \texttt{vec} & \text{if } \texttt{ndims(A)==1} \\ \texttt{mat} & \text{if } \texttt{ndims(A)==2} \\ \texttt{cube} & \text{if } \texttt{ndims(A)==3} \\ \texttt{cube\{n\}} & \text{if } \texttt{ndims(A)==n} \end{cases}$$

where `ndims` returns the total number of dimensions. Note that scalar numbers are not part of the relationship (hence $\texttt{scalar} \not\subset \texttt{vec}$). This is mainly for implementation efficiency.

The consequence is that, functions defined for arguments of type `cube` can also accept arguments of type `vec` and `mat`. For example, for digital images, `cube` can both represent color images (with dimensions $M \times N \times 3$) and grayscale images (with dimensions $M \times N \times 1$). In some cases, it is useful to indicate size information in the type.

This can be done by adding size parameters to the type: e.g., `cube(:,:,3)` denotes a 3D array for which the size in the last dimension is always 3. Size parameters give invaluable information to the compiler allowing specialized code to be generated.
Explicitly annotating the types of variables can bring performance benefits, although for code simplicity it is advised to only specify the type when necessary: in many cases the type is obtained and propagated using type inference. For example, when using the function `im=imread("image.png", "rgb")`, the compiler will infer that the type of `im` is `cube(:,:,3)`.
It is possible to check at run-time whether a variable (or intermediate result) has a certain type, using the function `type(A:typename)`. Moreover, the check can be performed using type checks and/or assertions, for example:

```
print(variable:cube[int])
print(1:cube) % Error: Type check failed: `int'const` is not `cube`
assert(type(1,"scalar"))
assert(type(1i,"cscalar"))
assert(type(zeros(2,2),"mat"))
assert(type("Quasar","string"))
```

In case one of the above the type checks fail, a compiler error will be generated. Additionally, the file `system.q` defines a number of lambda expressions for checking types:

```
isreal     = x -> type(x, "scalar")  || type(x,"vec")   || type(x,"mat")
                || type(x, "cube")
iscomplex  = x -> type(x, "cscalar") || type(x,"cvec") || type(x,"cmat")
                || type(x, "ccube")
isscalar   = x -> type(x, "scalar")  || type(x, "cscalar")
isvector   = x -> type(x, "vec")     || type(x, "cvec")
ismatrix   = x -> type(x, "mat")     || type(x, "cmat")  || isvector(x)
iscube     = x -> type(x, "cube")    || type(x, "ccube") || ismatrix(x)
```

Under some circumstances, the Quasar compiler is not able to figure out the types of the variables through inference. One example is the `load` function, which reads data from a file (through a process called deserialization) and stores them into variables.

```
[A, B] = load("myfile.dat")
```

The file load operation is only performed at runtime, the result depends on the content of the file being loaded and correspondingly the compiler can not predict the types of the variables. Then it makes sense to give the compiler some type information, such that it can perform some smart optimizations when needed:

```
assert(type(A,"ccube"))
assert(type(B,"vec"))
```

The `assert` function then has a two-fold purpose: 1) it gives the compiler information about the types of A and B and 2) it performs a runtime check to validate the data read from“myfile.dat”.
An alternative (and perhaps cleaner) way to check the type of the variable is by using type annotations. The above example then becomes:

```
[A : ccube, B : vec] = load("myfile.dat")
```

In case the types do not match, the runtime system will generate an error message. Type annotations need to be declared only the first time the variable is used.

Table 2.3: Additional primitive types "*first-class citizens*"

Type	Purpose
`string`	Sequences of characters
`lambda_expr`	Lambda expressions
`function`	Function handles
`kernel_function`	Kernel functions
`object`	Objects
`??`	Unspecified type (i.e. determined at run-time)

Table 2.4: Type conversion table

From/To	`int/ivec/imat/icube`	`scalar/vec/mat/cube`	`cscalar/cvec/cmat/ccube`
`int/ivec/imat/icube`	-	`float(.)`	`complex(.)` / `complex(re,im)`
`scalar/vec/mat/cube`	`int(.)`	-	`complex(.)` / `complex(re,im)`
`cscalar/cvec/cmat/ccube`	`int(real(.))` / `int(imag(.))`	`real(.)` / `imag(.)`	-

Finally, type conversion is generally not needed in Quasar (avoided for computational performance reasons), although a conversion table is given in table 2.4. Only for generic programming purposes (see section §6), an overridable type conversion function `cast(x, new_type)` is available.
Lambda expressions can also be explicitly typed. Quasar follows typing conventions similar to the Haskell and ML programming languages. For example:

- `[int -> int]`: a function that takes "`int`" as input and gives "`int`" as output
- `[(mat, scalar) -> (mat, mat)]`: a function that takes two input arguments (of type `mat` and `scalar`) and that has two output arguments (both of type `mat`)
- `[int -> int -> int]`: a function that projects an integer input onto a lambda expression of type `int -> int`.
- `[...scalar -> scalar]`: a variadic function that takes an arbitrary number of scalar values as input and that returns a scalar number.

Lambda expressions (especially those that use closures, see section 4.2) are very powerful in Quasar. To reduce the overhead associated with calling lambda expressions on computation devices, the Quasar compiler attempts to inline lambda expressions and functions whenever possible or beneficial.

2.2.1 Floating point representation

In Quasar, the internal representation of scalar numbers "`scalar`" (or complex scalar numbers "`cscalar`"), is *usually* not specified at the code level. This allows the floating point representation to be changed on a global level. By default, Quasar will use single precision floating point numbers (see table 2.5). However, it is possible to compile and run the programs using double precision as well, by passing the `-double` command line option to Quasar, e.g.:

```
./Quasar.exe -debug -double script.q
```

When numerical precision is not of uttermost importance, it is recommended to use single precision. Note that some older GPUs have limited double precision FP support. CUDA devices before compute capability 1.3 even do not have double precision FP support. Moreover, using double precision FP numbers doubles the memory bandwidth. Consequently, programs using double precision may run up to 2x slower than programs with single precision FP. Additionally, several consumer GPUs (e.g., Geforce series) have a double precision throughput that is much lower

Table 2.5: Overview of floating point representations

Quasar type	IEEE 32-bit (single precision) `scalar'single`	IEEE 16-bit (half precision) `scalar'half`	IEEE 64-bit (double precision) `scalar'double`
Significand	23 bits	10 bits	52 bits
Exponent	8 bits	5 bits	11 bits
Minimum pos. value	$1.17549435 \times 10^{-38}$	5.9605×10^{-8}	$2.225073858507201 \times 10^{-308}$
Maximum pos. value	$3.40282347 \times 10^{38}$	65504.0	$1.797693134862316 \times 10^{308}$
Exact integer repr.	$-2^{24}+1$ to $2^{24}-1$ $(16,777,215)$	$-2^{11}+1$ to $2^{11}-1$ $(2,048)$	$-2^{53}-1$ to $2^{53}-1$

than the floating point precision throughput. However, there are some reasons to enable double precision in Quasar programs:

- When numerical accuracy is an issue: remark that results obtained using double-precision arithmetic may differ from the same operations obtained using single-precision arithmetic, due to the greater precision and due to rounding errors. Therefore, it is important to compare and express the results within a certain tolerance rather than expecting them to be exact. Moreover, GPU devices typically flush numbers smaller than the minimum representable value (in absolute sense) to zero. Correspondingly, by using double precision FP numbers it may be possible to reduce some of the error introduced by underflow, as the minimal representable value is of the order 10^{-308} for double, while 10^{-38} for single precision (see table 2.5).
- For comparing the results of the algorithms to MATLAB/C++ implementations using double precision FP numbers.

Often it is useful to check whether the program is not suffering from floating point inaccuracies. This can simply be done by running the program once in *double precision mode*.
Note: NVidia GPU's GTX 260, 275, 280, 285, 295 chips (with compute capability 1.3) have a low performance in double precision computations (about 1/8 of single precision performance). Devices of the NVidia Fermi architecture (compute capability 2.0+) have 1/2 the performance of single precision operations. Performance is greatly improved with either NVidia Tesla cards or the NVidia Titan (which is based on the Kepler architecture). For full details, see https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions.
Finally, recall that FP math is not associative, i.e. the sum $(A+B)+C$ is not guaranteed to be equal to $A+(B+C)$. When parallelizing computations, the order of the operations is often changed (and may be even unspecified), leading to results which may differ each time the technique runs even with the same input data. This limitation is not inherent to Quasar, but applies to the specific approach used to perform parallel computations using floating point numbers. The example "Accurate sum" gives more information in this issue (see section 10.7); also in section 13.6 we explain how parallel code can be written that does not depend on this "non-deterministic" behavior.
The global constant "`eps`" is available for determining the machine precision (similar to `FLT_EPSILON`/`DBL_EPSILON` in C or `eps` in Octave/Matlab). The functions `maxvalue(scalar)` and `minvalue(scalar)` can be used to determine the maximum and minimum values representable in floating point format.

2.2.2 Mixed precision floating point computations

It is recommended to use the default scalar data type (with globally defined precision) as much as possible. However, in some cases, it is desirable to perform certain parts of the computation in a higher precision (e.g., when numerical accuracy is important) or even in a lower precision (e.g., when memory and/or computation time counts). Therefore Quasar allows specifying the required precision as part of the scalar type:

- `scalar'single`: represents a 32-bit IEEE single precision floating point type (FP32)

- `scalar'double`: represents a 64-bit IEEE double precision floating point type (FP64)
- `scalar'half`: represents a 16-bit IEEE half precision floating point type (FP16)

The half precision floating point representation (FP16) cannot be set globally (due to its inherent precision and range restrictions), however, it is possible to write functions that mix scalar types of different precision. Mixing FP16 and FP32 appears to be common in machine learning (e.g., convolutional neural networks). Pascal, Volta and Turing GPUs also allow support FP16 computations natively. Volta's tensor cores provide mixed matrix multiply-accumulate operations, in which the matrix is stored in FP16, the result is represented in FP32.

Remark: Quasar currently does not support *bfloat16*, the half-precision floating point format supported by Google's Tensor Processing Units (TPUs).

2.2.3 Integer types

Next to floating point numbers, Quasar has also (limited) support for integer types. The default integer type is "`int`" (signed integer). Its bit length depends on the computation engine, but is guaranteed to be at least 32-bit. There are also integer types with a pre-defined bit length, these are mainly provided 1) to enable more efficient input/output handling (e.g. reading/writing of images in integer format), or 2) to write certain algorithm in which memory usage/memory bandwidth should be as low as possible. Generally, the use of the integers with pre-defined bit length should be avoided. For completeness, these types are listed below:

- `int8`: a signed 8-bit integer (with range -128..127)
- `int16`: a signed 16-bit integer (with range -32768..32767)
- `int32`: a signed 32-bit integer (with range $-2^{31}..2^{31}-1$)
- `int64`: (not fully implemented yet)
- `uint8`: an unsigned 8-bit integer (with range 0..255)
- `uint16`: an unsigned 16-bit integer (with range 0..65535)
- `uint32`: an unsigned 32-bit integer (with range $0..2^{32}-1$)
- `uint64`: an unsigned 64-bit integer (with range $0..2^{64}-1$)

A matrix containing 8-bit integers can be obtained as follows:

```
A = mat[int8](rows,cols)
```

Note that, by default, arithmetic operations for integer matrices are disabled (e.g. summing, subtracting, conversion to floating point etc.). These operations can be included by importing the inttypes library (`import "inttypes.q`"). Integer types can have special modifiers (the modifier can be added by writing a apostrophe ' directly after the type name). These modifiers indicate how the conversion from a floating point number / integer number with larger bit depth to the considered integer type takes place.

- `int'checked` (default): generates an error when the integer can not be represented using the current type (note: not implemented yet)
- `int'sat`: in case of overflow, the integer is saturated (clipped) to the highest (or lowest) possible value that can be represented.
- `int'unchecked`: performs no integer overflow checking. This may often be the fastest.

The following function, which sums two 8-bit unsigned integer matrices, illustrates the usage of integer modifiers:


```
function y : mat[uint8'sat] = add(a : mat[uint8], b : mat[uint8])
    for m=0..size(a,0)-1
        for n=0..size(a,1)-1
            y[m,n] = a[m,n] + b[m,n]
        endfor
    endfor
endfunction
```

Here, integer saturation is used in case the sum of `a[m,n]` and `b[m,n]` does not fall in the range 0..255.

Important note

Variables are implicitly integer when defined by a constant with no decimal sign. This may have some unexpected consequences as in the following example:

```
a = 0 % a is integer
for n=0..N-1
   a += x[n] % x[n] is implicitly converted to integer
endfor
```

Here, `x[n]` is automatically converted to integer. To warn the programmer, a type conversion warning message will be shown. If it is desired to declare `a` as a scalar value, the first line needs to be replaced by `a = 0.0`.

2.2.4 Fixed sized datatypes

As already mentioned in section 2.2, vectors and matrices can optionally have their size specified in any dimension. For example, `a : cube(:,4,:)` always has `size(a,1) == 4`. The size acts as a constraint on the specialized type; the constraint is checked by both the compiler and the runtime. ':' indicates that the length is unspecified (not known at compile-time).
Using this technique, it is easy to define single instruction multiple data (SIMD) operations. For x86/64 CPU targets in Quasar, vectors of length 4 (`vec(4)`) may be[3] mapped onto SSE datatypes while vectors of length 8 (`vec(8)`) may be mapped onto AVX datatypes. See section §12 for more information.
In generic functions, it is possible to use type parameters for this purpose (for example `cube(:,:,P)`). For the following function:

```
function [] = color_transform[P : int](im_in : cube(:,:,P), im_out : cube(:,:,P))
   ...
endfunction
```

array size errors can be caught early in the development process: when the function `color_transform` is called and the compiler cannot guarantee that `im_in` and `im_out` have the same size in the third dimension, a compiler error will result.

2.2.5 Higher dimensional matrices

Higher-dimensional matrices (with dimension > 3) need to be specified using an explicit dimension parameter. For example `cube{4}` denotes a 4-dimensional array. The array with unspecified (infinite) dimension in Quasar is `cube{:}`. This is useful for generic specialization purposes (see further in section §6). To express higher-dimensional

[3] depending on the configuration settings and depending on whether the specific operation is accelerated by the processor.

loops for which the dimensionality is not known in advance, the functions `ind2pos` and `pos2ind` convert between position coordinates and linear indices, as illustrated by the following example:

```
function y:'unchecked = multidim_subsampling(x : cube{:}, factor : int)
    y = uninit(int(size(x)/factor))

    {!parallel for}
    for i = 0..numel(y)-1
        p = ind2pos(size(y), i)
        y[p] = x[p*factor]
    endfor
endfunction
```

Here, a higher dimensional cube is subsampled by `factor` along all dimensions of the cube. This technique is particularly useful for implementing operations with unknown dimensionality of the input parameters (as e.g., in Kronecker products).

2.2.6 User-defined types, type definitions and pointers

Quasar supports user-defined types (UDTs) and pointers: the user-defined types are defined as classes, as illustrated below:

```
type point : class
    x : scalar
    y : scalar
endtype
```

The `type` keyword is always followed by a type definition. The class `point` can be instantiated using its default constructor:

```
p = point()
```

or:

```
p = point(x:=4, y:=5)
```

Remark that the arguments of the constructor are *named*. The order of the arguments can then also be changed:

```
p = point(y:=5, x:=4)
```

By default, classes in Quasar are *immutable*. This means that, once initialized, the value of the class cannot be changed (or a compiler error will be generated)! Classes can also be made mutable, as follows:

```
type point : mutable class
    x : scalar
    y : scalar
endtype
```

Immutable classes allow for some optimizations to be applied. For example, they can be stored in *constant* device memory, some memory transfers are eliminated, moreover, the Quasar runtime does not need to check if the value of this class has been changed in device memory. For these reasons, it is recommended to use immutable classes whenever possible.

Additionally, a UDT can contain other UDTs:

```
type rectangle : class
    p1 : point
    p2 : point
endtype
```

Remark that the fields of the rectangle (`p1`, `p2`) are stored *in-place*. This means that the internal storage size of the UDT is the sum of the storage sizes of its fields. In this case, using single precision FP, elements of the `point` class will take 8 bytes and consequently elements of the rectangle class will contain 16 bytes.
Like in other programming languages (e.g. C/C++, Pascal), it is also possible to define a rectangle that stores *references* to the `point` class. Therefore, Quasar supports Pascal-type *pointers*:

```
type pt_rectangle : class
    p1 : ^point
    p2 : ^point
endtype
```

Remark that in many programming languages, pointers can be a source of programming errors (e.g. dangling pointers, uninitialized pointers etc). For this reason, the pointers in Quasar have special properties, that allow them to be safe in usage:

- Multiple indirections (`^^point`) are not allowed.
- All pointer values should be initialized, either used a constructor of the class, or using a null pointer (`nullptr`). For example, the above class can be initialized using:

  ```
  r = pt_rectangle(p1:=nullptr, p2:=point(1,2))
  ```

- Pointers are only allowed to be used for UDTs, not for scalars (`scalar`, `cscalar`, ...) or matrices (`vec`, `mat`, `cube`, ...).
- All pointer values are *typed*. It is for example *not* allowed to declare a pointer to an unknown type (`^??`).
- Pointer arithmetic is also not allowed.

Internal detail: the pointers in Quasar rely on customized form of reference counting to help track allocation of memory, including a technique to solve memory leaks caused by potential circular references. Moreover, the pointers make an abstraction from the particular *device*: the object can reside either in CPU memory, GPU memory, or both.
It is also possible to define (multi-dimensional) arrays of UDTs, using parametric types:

```
type point_vec : vec[point]
type point_mat : mat[point]
type point_cube : cube[point]
type rectangle_vec : vec[rectangle]
type pt_rectangle_vec : vec[^pt_rectangle]
```

Using UDT arrays is often more efficient than storing the individual elements of the UDT in separate matrices. This is because 1) the indexing often only needs to be performed once and 2) because better memory coalescing and caching. The UDT arrays can be initialized by zero (or using `nullptr`'s), in the following way:


```
a = point_vec(10)
b = point_mat(4, 5)
c = point_cube([1, 2, 3])
```

Note that a type definition (`type x : y`) is required for this construction. The following is (currently) not supported:

```
a = vec[point](10)
```

Moreover, the multi-dimensional arrays and UDTs may contain variables with *unspecified* types:

```
type point : class
       x : ??
       y : ??
endtype

type cell_vec : vec[??]
type cell_mat : mat[??]
type cell_cube : cube[??]
```

One caveat is: variables with unspecified types do not support automatic parallelization (see further in section 2.3) and can not be passed to kernel functions (see section 2.4.1).
UDTs can also contain vectors/matrices:

```
type wavelet_bands : mutable class
    LL : ^wavelet_bands
    HL : mat
    LH : mat
    HH : mat
endtype
```

The premise is that this class does not have a default constructor (`wavelet_bands()`), because there are no default values for matrices. Also `nullptr`'s are not allowed. Hence, it is necessary to explicitly specify the value of `wavelet_bands`:

```
bands = wavelet_bands(LL:=nullptr,
                      HL:=ones(64,64),
                      LH:=ones(64,64),
                      HH:=ones(64,64))
```

2.3 Automatic parallelization

The Quasar compiler automatically attempts to parallelize for-loops, depending on the matrix indexing scheme, input/output variables, constants and data dependencies. For example, the sequential code fragment, demonstrating a spatial filtering using a box filter (`mask`):

```
im = imread("image_big.tif")
im_out = zeros(size(im))
N = 5
mask = ones(2*N+1,2*N+1)/(2*N+1)^2
```



```
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
        a = [0.,0.,0.]
        for k=-N..N
            for l=-N..N
                a += mask[N+k,N+l] * im[m+k,n+l,0..2]
            endfor
        endfor
        im_out[m,n,0..2] = a
    endfor
endfor
```

automatically expands to the following equivalent parallel program:

```
im = imread("image_big.tif")
im_out = zeros(size(im))
N = 5
mask = ones(2*N+1,2*N+1)/(2*N+1)^2

function []=__kernel__ parallel_func(im:cube,im_out:cube,mask:mat,N:int,pos:ivec2)
        a = [0.,0.,0.]
    for k=-N..N
        for l=-N..N
            a += mask[N+k,N+l] * im[pos[0]+k,pos[1]+l,0..2]
        endfor
    endfor
    im_out[pos[0],pos[1],0..2] = a
endfunction
parallel_do(size(im,0..1),im,im_out,mask,N,parallel_func)
```

In this program, first a kernel function is defined. Next, the `parallel_do` function launches the kernel function in parallel for every pixel in the image `im`. The kernel function processes exactly one pixel intensity, and is called repetitively by the function `parallel_do`. When compiling the Quasar program, the kernel functions and automatically parallelized loops are compiled, depending on the computation engine being used, to CUDA or native C++ code (using OpenMP). This ensures optimal usage of the computational resources.

The Quasar optimizer may fail to extract a parallel program, for example because the type of certain variables is not known or because certain dependencies between variables have been detected (the latter causing the loop to be executed serially - "serialized"). For mapping algorithms onto hardware, variable types need to be well defined. As explained in section 2.2, when the variable type is not specified, Quasar uses type inference to derive the exact type from the context. When this fails, warning messages are displayed on the console during compilation that can help to make the program parallel, e.g., by explicitly declaring the type (`B : vec = ...`). Quite often, it may be the intention of the programmer to have a parallel loop. In this case, it is possible to interrupt the program compilation when the loop parallelization fails (thereby generating a compiler error). This is possible by putting `{!parallel for}` directly before the for-loop to be parallelized:

```
{!parallel for}
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
    endfor
endfor
```


In case the compiler then detects some dependencies between the variables, a warning message will be displayed reporting these dependencies and the loop will be parallelized despite the warnings (possibly causing data races). There are some scenarios that currently cannot be handled by the auto-parallelizer (for example polymorphic variables which change type during the loop). Additionally, certain features cannot be used from inside loops, such as shared memory functions, thread synchronization... Therefore, and also for full flexibility, it is also possible to perform the parallelization completely manually. This is described in the following section.

2.4 Writing parallel code using kernel functions

In this section, we describe two ways of writing parallel code:

- *basic usage* (does not require any prior knowledge on GPU programming) - see section 2.4.2.
- *advanced usage* (for "experienced" GPU users) - see section 2.4.4.

Most algorithms can be efficiently implemented using the "basic" approach. The advanced usage consists of synchronization, dealing with data races, sharing memory across multiprocessors and other GPU programming techniques, which may lead to increased performance taking more advantage of the available features.
For beginning users, it is advised to get acquainted first with the basic usage techniques, before considering the advanced usage. Additionally, kernels written using the "basic usage" approach are often further optimized by the Quasar compiler to use some more advanced features (see 17).

2.4.1 Basic usage: kernel functions

A kernel function is a Quasar function with a special attribute `__kernel__`, that can be parallelized. Kernel functions are launched in parallel on every element of a certain matrix, using the "`parallel_do`" built-in function. The `__kernel__` attribute specifies that the function should be *natively* compiled for the targeted computation engine (e.g. CUDA, CPU). Consequently, `__kernel__` functions are *considerably* faster in execution than host functions thanks to their parallelization and native code generation. As example, consider the following algorithm:

```
function [] = __kernel__ color_temperature(x : cube, y : cube, temp,
    cold : vec3, hot : vec3, pos : ivec2)

    input = x[pos[0],pos[1],0..2]
    if temp<0
       output = lerp(input,cold,(-0.25)*temp)
    else
       output = lerp(input,hot,0.25*temp)
    endif
    y[pos[0],pos[1],0..2] = output
endfunction
hot = [1.0,0.2,0.0]*255
cold = [0.3,0.4,1]*255
img_out = zeros(size(img_in))
parallel_do(size(img_out,0..1),img_in,img_out,temp,cold,hot,color_temperature)
```

The kernel function is launched on a grid of dimensions "`size(img_out,0..1)`" using the `parallel_do` construct. This means that every pixel in `img_out` will be addressed individually by `parallel_do`, and correspondingly the function `color_temperature` will be called for every pixel position.

For a kernel function, all parameters need to be *explicitly* typed. Recall that untyped parameters in Quasar are denoted by `??`. Code using untyped parameters cannot be mapped onto an efficient implementation, therefore compiler will first try to deduce the type from the context. The kernel function is then treated as a generic function (see section §6). If the type deduction fails, a compiler error will result.
To write efficient kernel functions it is recommended to use vector and matrix types with size constraints. For example:

- `vec(X)` (or shortcut[4] `vecX`): corresponds to a vector of length X.
- `ivec(X)` (or shortcut `ivecX`): corresponds to an integer vector of length X.
- `cvec(X)` (or shortcut `ivecX`): corresponds to a complex-valued vector of length X.
- `mat(X,Y)` (or shortcut `matXxY`): a matrix of size $X \times Y$.
- `cube(X,Y,Z)`(or shortcut `cubeXxYxZ`): a cube of size $X \times Y \times Z$.
- `int`: integer data type

The explicit size of the vector and matrix parameters allows calculations involving the variables to be mapped onto an efficient implementation. Additionally, it also helps the type inference: for example, the product of a vector of length 4 (`vec(4)`) and a 4×4 matrix (`mat(4,4)`) results in a vector of length 4.
However, some datatypes can not be passed as arguments to kernel functions: cell matrices containing unknown types (`??`) and strings. To pass cell matrices, use parametrized matrix types (`vec[cube]`, `mat[cube]`, `mat[vec]`, etc., see section §3). Strings need to be converted to vectors (using the functions `fromascii`, `fromunicode`). Device functions (`__device__`) possibly containing closure variables (see section 4.2), can also be passed.
For vectors of length ≤ 64, it is most efficient to add the length explicitly in the type as above. Vectors with length known at compile-time are treated in a special way: the components of the vector are grouped together requiring less memory read/write requests and may be implemented using SIMD instructions if the underlying back-end compiler supports them. At the very least, these vectors are allocated on the stack (or registers), which is significantly faster than in the kernel dynamic memory (see section §8.3). Matrices with size constraints (such as `matXxY`, `cubeXxYxZ`) are also treated as fixed-length vectors internally.

Remark: the current Quasar implementation places a limit on the maximum length of the constraint, or the product of the dimensions (e.g., $X \times Y$ for `matXxY`, $X \times Y \times Z$ for `cubeXxYxZ`. This limit is 64). When the vector length is longer, the specification of the value will not have an effect (apart from type inference purposes).

Inside `__kernel__` and `__device__` functions, it is recommended to use integers instead of scalars (when possible). This may yield a speed-up of about 30% for CUDA targets and even more for CPU targets. When a scalar constant contains a decimal point (e.g., 1.2), the compiler will consider this constant to be a floating point number, otherwise it will be treated as an integer.
The syntax of the `parallel_do` function is as follows:

```
parallel_do(dimensions, inarg1, ..., inargN, kernel_function)
```

where `dimensions` is a vector. Note that *normally* kernel function cannot have output arguments (there is a special advanced feature that allows kernel functions to return values of certain types, see section 4.7, but this feature is only for specific use-cases). Instead, in most use cases the return values should be written to the input arguments passed by reference, i.e. arguments of types `vec`, `mat`, `cube`, `cvec`, `cmat`, `ccube`.
There are some *special* arguments that can be defined in the kernel function declaration, but that do *not* need to be passed to `parallel_do`:

[4] replace `X` by the exact value, for example `vec2`, `vec3`, ...

- `pos` (of type `int`, (`i`)`vecX`): the current position of the work item being processed. Note that a "work item" can be either an individual pixel, or a "group of pixels", depending on how you specify the "dimensions" argument.
- `blkpos` (of type `int`, (`i`)`vecX`): the current position within the block (for advanced users, see section 2.4.4)
- `blkidx` (of type `int`, (`i`)`vecX`): the block index (for advanced users, see section 2.4.4)
- `blkdim` (of type `int`, (`i`)`vecX`): the current dimensions a block (for advanced users, see section 2.4.4). Internally, the data is processed on a block-by-block basis. The dimensions of a block depend on the computation engine in use. For example, for the CUDA computation engine (with CUDA compute capability 2.0), the block dimensions can be as large as 16×32 or 32×16. For the CPU computation engine, the block dimensions will rather be $1 \times$#num_processors.
- `blkcnt` (of type `int`, (`i`)`vecx`): the number of blocks in each dimension (for advanced users, see section 2.4.4)
- `warpsize` (of type `int`): the warp size of the device (for advanced users, see section 2.4.4)

The `parallel_do` function basically executes the following sequential program in parallel:

```
blkdim = choose_optimal_block_size(kernel_function) % done automatically
for m=0..dimensions[0]-1
    for n=0..dimensions[1]-1
        for p=0..dimensions[2]-1
            pos = [m,n,p]
            blkpos = mod(pos, blkdim)
            blkidx = floor(pos/blkdim)
            kernel_function(inarg1, ..., inargN, [pos], [blkpos], [blkidx], [blkdim])
        endfor
    endfor
endfor
```

Here, first optimal block dimensions (`blkdim`) for the given kernel function are being selected. Then, `kernel_function` is run inside the three loops, `prod(dimensions)`=`dimensions[0]`$\times$`dimensions[1]`$\times$`dimensions[2]` times.
Special *modifiers* are available for kernel function arguments. The modifiers are specified using the apostrophe-symbol:

```
function [] = __kernel__ imfilter_kernel_nonsep_mirror_ext(y : cube'unchecked,
    x : cube'unchecked, mask : mat'unchecked'const, center : ivec2, pos : ivec3)
```

These modifiers specify how vector/matrix/cube elements are accessed, and in particular enable efficient boundary handling in image processing:

- `'safe`: disregards writes outside the data boundaries, reads outside the data boundaries evaluate to *zero*.
- `'circular`: performs circular boundary handling
- `'mirror`: mirrors when accessing outside the data boundaries.
- `'clamped`: clamps (saturates) to the data boundaries (`y[0] = y[-1] = y[-2] = ...` and `y[N-1] = y[N] = y[N+1] =` ...)

- `'unchecked` (warning: dangerous usage - your program may crash if not used properly): specifies no bounds checking on the input/output data. In case of access outside the data boundaries, a runtime error may or may not be generated. Specify this modifier in case you are sure your kernel/device function is 100% correct, and when you want to enjoy a modest extra code speedup.
- `'checked`: the opposite of `'unchecked`: generates an error when indices are out of the data boundaries. Quasar will give information on which matrices are the prime suspect.
- `'const`: indicates that the matrix variable is constant and will not be changed inside the kernel function.

The default access modes are currently `'safe` (inside kernel/device functions) and `'checked` outside of kernel/device functions (for performance reasons). In case the program behavior depends on the access mode, it is best to explicitly indicate the access mode.
Finally, there are some rules with respect to the calling conventions for kernel functions:

- Kernel functions can not have optional arguments.
- A kernel function *can not* call a "host" function.
- A kernel function *can* call a "device" function (see section 2.4.2)
- A kernel function can call a lambda expression declared with the `__device__` attribute (see section 2.4.2).
- A kernel function can call a lambda expression declared without the `__device__` attribute, on the premise that the lambda expression can be *inlined*.
- A kernel function can call other kernel functions, through `parallel_do` (see further in section §4.4). A kernel function cannot directly call another kernel function using a standard function call.

There are some special functions that can be used within kernel functions:

- `periodize(x, N)`: periodizes the input coordinate, i.e. $k + a \cdot N$, with $0 \leq k < N$ becomes k. This function is used automatically when the modifier `'circular` is specified.
- `mirror_ext(x, N)`: mirrors the input coordinate between $[0, N-1]$. This function is used automatically when the modifier `'mirror` is specified.
- `clamp(x, N)`: clamps the input coordinates to [0, N]. This function is used automatically when the modifier `'clamped` is specified.
- `int(x)`: converts the input argument to integer (using type casting)
- `float(x)`: converts the input argument to floating-point (using type casting)
- `shared(dims)`, `shared_zeros(dims)`, `shared[T](dims)`: the function has a special meaning - allocation of *shared* memory (see section 2.4.4).

2.4.2 Device functions

In the example in the previous section, the linear interpolation function `lerp` is defined as:

```
lerp = __device__ (a : scalar, b : scalar, d : scalar) -> a + (b - a) * d
```


Table 2.6: Quasar: which function types can call ...?

From/To	"host"	`__device__`	`__kernel__`
"host"	Yes	Yes	
`__device__`	No	Yes	`parallel_do`/`serial_do` only
`__kernel__`	No	Yes	

Device functions are the only functions (next to kernel functions) that can be called from a kernel/device function. The `__device__` function specifies that the function should be *natively* compiled for the targeted computation engine (e.g. CUDA, CPU), however, in contrast to kernel functions, they can not be used as argument to a call of the `parallel_do` function. Device functions are hence useful to aid the writing of kernel functions. For example, if one often needs a 2D vector that is orthogonal to a given 2D vector, one can define:

```
orth = __device__ (x : vec2) -> [-x[1], x[0]]
```

The function *orth* can then be used from other functions (also outside kernel/device functions).
table 2.6 lists whether functions of different types can call each other. Note that there are a number of combinations that are not supported:

- A device function cannot call a host function. This is simply because "default" functions are, by default, not natively compiled. However, in many cases, it is possible to convert the host function to a device function, by adding the `__device__` modifier.
- A device function cannot call a kernel function *directly*, nor can a kernel function call another kernel function (unless `parallel_do` is used, see further in section §4.4). This is because kernel functions have special facilities for parallelization (e.g. they can use OpenMP etc).
- However, a host function can call a device function. This is useful for declaring functions that can be used both from host code as from kernel code. An example is the *sinc* function:

```
sinc = __device__ (x:scalar) -> x == 0 ? 1.0 : sin(x)/x

print sinc(0) % call the device function
```

Remark: kernel and device functions have dedicated types, containing respectively `__kernel__` and `__device__` type modifiers. For the above definitions:

```
imfilter_kernel_nonsep_mirror_ext : _
    [__kernel__(cube,cube,mat,ivec2,ivec3) -> ()]
lerp : [__device__(scalar,scalar,scalar) -> scalar]
orth : [__device__(vec2) -> vec2]
```

These types can be used for defining more general functions that use device/kernel functions as input argument. For example:

```
add = __device__ (x : scalar, y : scalar) -> x + y
sub = __device__ (x : scalar, y : scalar) -> x - y
mul = __device__ (x : scalar, y : scalar) -> x * y
orth = __device__ (x : vec2) -> [-x[1], x[0]]
ident = __device__ (x : scalar) -> sub(add(x, 2*x), 2*x)

function [] = __kernel__ my_kernel (X : mat, Y : mat, Z : mat, pos : ivec2)
```



```
    Z[pos] = add(X[pos], Y[pos])
    v = orth([X[pos], Y[pos]])
endfunction

X = ones(4,4)
Y = eye(4)
Z = zeros(size(X))
parallel_do(size(Z),X,Y,Z,my_kernel)
```

One special feature of device functions, is that they can be used as function pointers and passed to kernel functions. This can be used to reduce the number of kernel functions, or as an alternative to dynamic code generation:

```
% Definition of a __device__ function type
type binary_function : [__device__ (scalar, scalar) -> scalar]

add = __device__ (x : scalar, y : scalar) -> x + y
sub = __device__ (x : scalar, y : scalar) -> x - y
mul = __device__ (x : scalar, y : scalar) -> x * y

function [] = __kernel__ arithmetic_op(Y : cube, _
    A : cube, B : cube, fn : binary_function, pos : ivec3)
    Y[pos] = fn(A[pos], B[pos])
endfunction

A = ones(50,50,3)
B = rand(size(A))
Y = zeros(size(A))
parallel_do(size(Y),Y,A,B,add,arithmetic_op)
```

Unfortunately, there is a performance penalty associated to function pointer calls: the extra indirection avoids the compiler to inline the function. For this reason, when possible the Quasar compiler will attempt to avoid function pointer calls (by substituting the exact function).

2.4.3 Memory usage inside kernel or device functions

There are three types of memory that can be used inside kernel or device functions:

1. *local memory*: this is memory that is local to the function, and each parallel run of the kernel function (called 'thread') contains a private copy of this memory. Below are a few examples of the creation of local memory:

   ```
   A = [0, 1, 2, 3] % Generates a variable of type 'ivec4'
   B = [0., 1., 2., 3.] % Generates a variable of type 'vec4'
   C = [1 + 1j, 2 - 2j] % Generates a complex-valued variable of type 'cvec2'
   D = ones(6) % Generate a vector of length 6, filled with 1.
   E = zeros(8) % Generates a vector of length 8, filled with 0.
   F = complex(zeros(4)) % Generates a complex-valued vector of length 4
   ```

   For the GPU computation engine, there are however a few limitations: first, local memory is internally stored in device registers, is hence very fast, but also *scarse*. When the maximum number of device registers is exceeded, global memory is used instead (which has a much larger memory access latency).

2. *shared memory*: this type of memory is shared across threads, and allocated using the functions `shared` and `shared_zeros`. Its usage is discussed in section 2.4.4.

3. *global memory*: this type of memory is used for storing vectors and matrices with either large dimensions or dimensions that cannot be determined at compile-time. Global memory is also used for dynamically allocated objects (see section §8.3). For example:

```
function [] = __kernel__ my_kernel (X : mat, pos : ivec2)
  % X[pos] is stored in global memory
endfunction

X = ones(4096,4096)
parallel_do(size(X), my_kernel)
```

   Here, `X` is allocated outside a kernel function. The values of X, in total $4 \times 4096 \times 4096$ bytes (in case of 32-bit floating point), are stored automatically in global memory in a *linear way*. The following formula is used for translating the 3D index to a linear index:

$$\text{index}(\text{dim}_1, \text{dim}_2, \text{dim}_3) = (\text{dim}_1 \text{Ndims}_2 + \text{dim}_2)\,\text{Ndims}_3 + \text{dim}_3$$

   The `ind2pos(size, index)` function performs exactly this calculation. Global memory can reside either in CPU memory, GPU memory or both. When calling a kernel function using `parallel_do` in the GPU computation engine, the global memory will automatically be transferred to the GPU. Because the maximum amount of local memory and shared memory that can be used is limited by the hardware (e.g., not more than 48K), global memory is the only way to pass large amounts of data to a kernel function. The only premise is: a kernel/device function cannot allocate global memory, the memory should *best* be allocated in advance and passed to the function.
   In some cases, a Quasar program may run out of global GPU memory. In that case, Quasar will automatically transfer a non-frequently used memory buffer back to the CPU. This memory buffer can be later transferred back to the GPU. By this technique, Quasar programs can use all the available memory in the system (both CPU and GPU).

4. *texture memory*: texture memory is *read-only* global memory that is internally optimized for *spatial access patterns* (whereas the global memory is more optimal for linear accesses). In particular, the data layout is optimized for texture sampling using nearest neighbor interpolation or linear interpolation. CUDA uses space filling curves[5] for optimizing the data layout. See section 9.3 for more information.

Finally, it is important to mention that local memory should be *scarcely* used (or at least: with care), because for the GPU, the local memory is mapped directly onto the device registers. In CUDA, the total size of the device registers is 32K for compute capability 2.0 and 64K for compute capability 64K. However, the device registers are shared across all computing threads: hence, when invoking 512 threads in parallel, the total amount of local memory available to a kernel/device function is respectively 64 bytes and 128 bytes! If more registers are used, the kernel will execute, but global memory is used instead (called *register spilling*). To avoid performance impact, the Quasar runtime decreases the number of threads being spawned. The maximum number of threads that a given kernel function uses, can be determined using the function `prod(max_block_size(my_kernel))`. Also see section 2.4.4 for more information.

[5] http://en.wikipedia.org/wiki/Space-filling_curve

2.4.4 Advanced usage: shared memory and synchronization

For this section, a basic familiarity with the GPU architecture is required. Internally, chunks of data are processed in blocks, as follows:

```
pos = [m,n,p]
blkpos = mod(pos, blkdim)
blkidx = floor(pos/blkdim)
blkcnt = ceil(size(y)./blkdim)
```

Within one block, a kernel function can access data from kernel functions running in parallel on this block. This is very useful for implementing some special parallel algorithms, such as parallel sum, parallel sort, spatially recursive filters etc. However, read/write operations can interfere (data races), so special care is needed.
Advanced usage consists of 1) using thread synchronization, 2) using shared memory, 3) dealing with data races.

Thread Synchronization The number of threads that run in parallel over one block can be calculated using `prod(blkdim)`, i.e., the product of the block dimensions. Sometimes, it is necessary that each threads wait until a given operation is completed, by means of a thread barrier. All threads (within one block!) then wait until completion of the operation. In Quasar, this is done using the `syncthreads` keyword:

```
function [] = __kernel__ my_kernel (y : mat, z : mat, pos : ivec2, idx : ivec2)
    y[pos] = 10*idx
    syncthreads % all threads wait here until the above operation has been completed.
    z[pos] = y[pos]*2
endfunction
```

It is important to mention that the thread synchronization is performed on a *block level*, rather than on the full grid. In the above example, when the `syncthreads` is first encountered, only values `y[pos]` with `pos`$\in$`[0..blkdim[0]-1]` $\times$ `[0..blkdim[1]-1]` will have been computed, and not the complete matrix y!
Finally, the correct usage of `syncthreads` is that all threads effectively meet the barrier. It is for example not allowed to put a synchronization barrier inside a conditional `if...else...` clause, unless it is sure that each thread encounters the same number of barriers while running the kernel function. `syncthreads` may also have a parameter to indicate the synchronization granularity, for example `syncthreads(block)`, `syncthreads(warp)`, `syncthreads(grid)`, `syncthreads(host)`. For more details, see section §9.8.1.

Remark: note that `syncthreads` does not ensure that global memory writes performed by the block are completed and can be seen by other threads. If this is required, consider using `memfence` (see section 9.8.3).

Shared Memory The GPU has several memory types (global memory, texture memory, shared memory, registers etc.). Therefore, to achieve the best performance it is best to use the right memory type for each task. Global memory is used by default and registers are used for local calculations within kernel/device functions. Shared memory is visible to all threads of a kernel function within one block and can be allocated using the function `shared`(.) or `shared_zeros`(.) from kernel functions. It's usage is as follows:

```
var1 = shared(dim); % vector
var2 = shared(dim1,dim2); % matrix
var3 = shared(dim1,dim2,dim3); % cube

var4 = shared_zeros(dim); % vector initialized with 0's
var5 = shared_zeros(dim1,dim2); % matrix initialized with 0's
```



```
var6 = shared_zeros(dim1,dim2,dim3); % cube initialized with 0's

var7 = shared[uint8](dims) % generic memory allocation
var8 = shared_zeros[uint64](dims)
      % generic memory allocation initialized with 0's
syncthreads % REQUIRED in case of shared_zeros!!!
```

Shared memory is visible and shared within *one block*. That means that, when going to another block (e.g. when `blkidx` changes), the content of the shared memory cannot be relied on. Use `shared_zeros` only when you want to initialize the memory with zeros. The shared memory allocated with `shared` is not initialized (like in C/C++). This is often faster.

Important: one or multiple `shared_zeros` calls always need to be followed by a `syncthreads` statement (as shown in the example below). This is because the memory initialization by `shared_zeros` is performed in *parallel*. Hence, when all threads randomly start using the allocated memory it is necessary to wait until the zero initialization operation has fully been completed. In fact, the (internal) implementation of `shared_zeros` is as follows:

```
function [] = __kernel__ shared_mem_example(blkpos : ivec3, blkdim : ivec3)
    A = shared(100) % One vector of 100 elements
    % Compute the index of the current thread
    threadId = (blkpos[0] * blkdim[1] + blkpos[1]) * blkdim[2] + blkpos[0]
    nThreads = prod(blkdim) % Number of threads within one block

    for i=threadId..nThreads..numel(A)-1 % Parallel initialization
        A[i] = 0.0
    endfor
    syncthreads % Make sure all threads have finished before continuing!

    % Is equivalent to
    B = shared_zeros(100)
    syncthreads % Make sure all threads have finished before continuing!
endfunction
```

There are however two caveats when using shared memory:

1. For the CUDA computation engine, the amount of shared memory per block is typically 32 KB. On CUDA architectures, shared memory is on-chip and much faster than other off-chip memory. Consequently the amount of shared memory is limited. Taking into account that a (single precision) floating point value takes 4 bytes, the maximum dimensions of a square block of shared memory that you can allocate are 64×64. The more shared memory a kernel uses, the less blocks that can be executed in parallel on the GPU.

2. To obtain maximal performance benefits when using shared memory, it is important to make sure that the compiler can determine statically the amount of memory that will be used by the kernel function. If not, the compiler will assume that the kernel function will take all of the available shared memory on the GPU, which prevents the hardware from processing multiple blocks in parallel. For example, if you request:

   ```
   x = shared(20,3,6)
   ```

   the compiler will reserve $20 \times 3 \times 6 \times 4$ bytes $= 1440$ bytes for the kernel function. However, often the arguments of the function `shared` are non-constant. In this case you can use assertions (see further in 5):


```
assert(M<8 && N<20 && K<4)
x = shared(M,N,K)
```

With this assertion the compiler is able to infer the amount of required shared memory. In this case: 8×20×4×4 bytes = 2560 bytes. The compiler then gives the following message:

```
Information: Calculated an upper bound for the amount of shared memory: 2560 bytes
```

Due to these restrictions, shared memory should be used in a "smart" way and with care.

Dealing with data races To solve data races, one can either use atomic operations (e.g., `+=`, `-=`,`/=`,`*=`,`^=`, ...). Atomic operations are serialized, so the end result of the computation will always be correct. Atomic operations are often used in combination with synchronization barriers (see above). For example:

```
function [] = __kernel__ my_kernel(x : mat, y : vec, blkpos : ivec2, blkdim : ivec2)
    bins = zeros(blkdim) % allocates shared memory
    nblocks = (size(x)+blkdim-1)./blkdim

    % step 1 - do some computations
    val = 0.0
    for m=0..nblocks[0]-1
        for n=0..nblocks[1]-1
            val += x[blkpos + [m,n] .* blkdim]
        endfor
    endfor
    bins[blkpos] = val
    % step 2 - synchronize all threads using this barrier
    syncthreads
    % Now it is safe to read from the variable bins
endfunction
```

Hint: only use atomic operations when necessary. If there is no possibility for a data race, it is more efficient to use non-atomic counterparts (e.g. `y[pos] = y[pos] + 1`). For GPUs, atomic operations in shared memory are also more efficient than atomic operations in global memory. Often, atomic operations can be avoided using the parallel reduction algorithm (see section §8.4).

Warp size The warp size is the number of threads in a warp, a subdivision that is used in GPU hardware implementation for memory coalescing and instruction dispatch. The warp size is important for branching: branch divergence occurs when not all threads within a warp follow the same execution path; this should be avoided as much as possible. For recent GPUs, the warp size is typically 32. The warp size is also important to know when accessing constant memory (see section §9.1): constant memory works the most efficient when all threads within one warp access the same memory location at the same time.
In Quasar, the warpsize can be requested using the special kernel function parameter `warpsize`.

Specifying the GPU block size By default, the GPU block size is determined automatically by the runtime system. For situations in which explicit control of the block size is required, it is also possible to manually specify the block size (`blkdim`) using the function `parallel_do`. For example:


```
sz = max_block_size(my_kernel, my_block_size)
parallel_do([dims,sz],...,my_kernel)
```

where `dims` and `sz` are both vectors of equal length. `my_block_size` then typically depends on the amount of shared memory you want to use within the kernel. The built-in function `max_block_size` computes the *maximally allowed* block size for the given kernel function. Note that

- For the CUDA computation engine, the maximum block size is limited by the maximum number of threads per block. For CUDA compute capability 2.0, we should have that the maximum number of elements ≤ 1024. In practice, this number is often even lower, due to the resources used by the kernel (registers, shared memory, ...).
- For optimal performance, the number of elements in each block should be a multiple of the warp size (typically 32). To optimize the *memory access pattern* it might be even desirable to set the block width as a multiple of 32.
- For the CPU computation engine, the maximum block size is *unlimited*, unless synchronization (`syncthreads`) is used. In this case, the block size is limited depending on the number of multi-processors in the system.
- Performance is not necessarily proportional to the number of threads per block. In some cases, the optimal launch configuration has a block size that is as small as 64 or 128.

The above behavior is handled transparently by the function `max_block_size(.)`. Hence one should always call `max_block_size`, to determine the maximal block size for a given kernel function.
Warning: the specification of the block size (especially without using `max_block_size`) should be done with care, because when the block size is too small, the performance of the kernel function may be severly impacted. Additionally, the code may fail or work less optimally on future computation devices.

Block size not specified: what happens? In case the block size is not specified, it can be accessed from the kernel function through `blkdim` (this parameter should then be added to the argument list). The Quasar runtime system computes the block size that is estimated to be the most optimal for the given kernel, according to some heuristics. Quite often, this will be 16×32. Note that the block size is always a divisor of the dimensions `dims`. When necessary, the block size for a given kernel function can be retrieved programmatically using `opt_block_size`:

```
sz = opt_block_size(my_kernel)
```

Note that `opt_block_size` uses an internal optimization method for determining the best possible block size for the given data dimensions, taking into account the resources used by the kernel function (e.g., registers, shared memory, ...). Furthermore, it always returns a block size that the hardware can handle.

Large vector/matrix dimensions that are not a power/multiple of 2. It is best to specify dimensions to `parallel_do` that are a multiple of the maximal block size (e.g. 16×32 or 32×16). GPU computation engines best work with input data dimensions that are a multiple of (a power of) two. In the following example, this is not the case:

```
function [] = __kernel__ my_kernel(y : vec'unchecked, pos : int)
   y[pos] = 1.0
endfunction
y = zeros(65535)
parallel_do(size(y),y,my_kernel) % errorInvalidValue
```


To ensure proper functioning of the program, the runtime system internally pads the input dimensions to be a multiple of two, as follows:

```
function [] = __kernel__ my_kernel(y : vec'unchecked, pos : int)
   if pos >= 0 && pos < numel(y)
       y[pos] = 1.0
   endif
endfunction
y = zeros(65535)
pad = x -> ceil(x / BLOCK_SIZE) * BLOCK_SIZE % Block size is determined automatically
parallel_do(pad(numel(y)),y,my_kernel) % Success!
```

Note that this is performed completely transparently to the user, but comes at a slight performance cost: 1) the position checking `if pos >= 0 && pos < numel(y)`, which is performed by all threads and 2) some threads (the ones for which the if-test fails) may be "inactive" by this measure.

Chapter 3

Type system

3.1 Type definitions

Although variable types in Quasar often do not need to be specified (the types are either determined at compile time by type inference, or at runtime), it is always recommended to use strong typing. Strong typing has the immediate advantage that the compiler can generate some more optimal code for the typed variables. In this section, a more detailed overview of the Quasar type system is provided.
The following type categories exist in Quasar:

1. *Class #1*: Primitive types (`scalar`, `cscalar`, `int`, `int`x`,uint`x`,string`)
2. *Class #2*: vector/matrix/cube types (`vec`, `mat`, `cube`, `cube{:}`)
3. *Class #3*: Classes / user-defined types (`class`)
4. *Class #4*: Function types (`[?? -> ??]`, `[(??, ??) -> (??, ??)]`, `[__device__ scalar -> scalar]`, `[__kernel__ () -> ()]` ...)
5. *Class #5*: Type references (`type`).

Class #2 types are also container types and can embed all other types. For example, `mat[scalar]` represents a matrix type for scalar numbers, `cube[[??->??]]` represents a 3D array of functions with one input argument and one output argument. The parameters can be nested: `cube[cube[^T]]` denotes a 3D array of 3D arrays of pointers to objects of type T. By default, the default type arguments of `vec`, `mat` and `cube` is `scalar`. The `^`-prefix cannot be used on vectors/matrices: these objects are already passed by reference. In fact, `^` is only used for classes.
There also some derived types, which can be expressed directly in terms of the above types. Note that the following definitions are already defined as shorthand, so you do not need to define them yourself:

```
% Complex-valued matrices
type cvec : vec[cscalar]
type cmat : mat[cscalar]
type ccube : cube[cscalar]

% Integer matrices
```

```
type ivec : vec[int]
type imat : mat[int]
type icube : cube[int]

% By default: the argument type of vec[.],mat[.],cube[.] is scalar:
type vec : vec[scalar]
type mat : mat[scalar]
type cube : cube[scalar]

% Cell matrices
type cellvec : vec[??]
type cellmat : mat[??]
type cellcube : cube[??]
```

Class #3 types are passed *by value*. It is possible to declare pointers to these types (e.g. in order to pass them by reference). For this purpose, Pascal-style pointers can by used (e.g. `^T`). There is only one level of indirection possible (in contrast to C/C++) and also pointer values need to be explicitly initialized. It is not possible to declare pointers to types other than classes.
Class #2 types can contain parameters of class #3: for example `mat[T]`, `cube[T]`, `vec[^T]`. Especially vectors/matrices/cubes of UDTs containing only primitive types are very efficient, because they use a sequential layout scheme (i.e. they are stored contiguously in memory, with appropriate alignment depending on the machine/GPU).
Recall that cell matrix types containing unspecified sub-types `??`, such as `vec[vec[??]]`, `mat[??]`, cannot be passed to kernel or device functions. This is mainly for performance reasons: when the types of all variables are specified, the compiler can generate more optimal code. On the other hand, not specifying types can be an advantage for rapid prototyping.

3.2 Variable construction

The construction of variables of a specified type depends on the type class:

1. Class #1: variables of class #1 are constructed using symbols: a number containing a decimal point (e.g. `1.4e3`) will have type `scalar`. When the symbol contains the imaginary unit (`1i` or `1j`) it will be a complex scalar `cscalar` (e.g. `1+3i`). Strings (`string`) can be defined using "quotation marks". Note that it is not possible currently to construct variable of type `intx` or `uintx`: these types are mainly intended to be used for storage, and because for most computation engines default integer type (`int`) offer a better performance, the types cannot be used for calculations.

2. Class #2: variables of vector, matrix, cube types can be created using the `[]` constructor. The type of the result depends of the types of the operands (which should be the same for all operands, otherwise a compiler error is generated). For example, `[1,2,3,4]` has type ivec, `[[1.0,2.0],[3.0,4.0]]` has type mat. If `a,b,c` have a user-defined type `T`, then `[a,b,c]` will have type `vec[T]`. Similarly `[[a],[b]]` has type `mat[T]`. Real-valued vectors, cubes and matrices of arbitrary dimenions can be constructed using the functions `uninit`, `zeros`, and `ones`:

   ```
   A = uninit(2)
   B = zeros(3,4)
   C = ones(1,2,3)
   ```


Here, the function `uninit` allocates a vector of length 2, without initializing the data. `zeros` creates a matrix of 3 rows and 4 columns, and initializes each element to 0. `ones` creates a cube of dimensions $1 \times 2 \times 3$ and initializes each element to 1. Complex-valued versions can be obtained using the function `complex`, combined with `uninit`, `zeros` or `ones`:

```
A = complex(uninit(2))
B = complex(zeros(3,4))
C = complex(ones(1,2,3))
```

Variables of parametric vectors, matrices and cubes can also be constructed, however they require a type alias:

```
type my_cell : mat[cube]
A = my_cell(1,2)
A[0,0] = uninit(4,2)
A[0,1] = uninit(4,2)
```

3. Class #3: user-defined types are constructed either using the type name followed by (), or by explicitly assigning values to all fields, as shown below:

```
type point : class
   x : scalar
   y : scalar
endtype
p = point()
q = point(x:=1,y:=2)
```

4. Class #4: variables of these types are created using either a lambda expression, or a function definition (see section 4.6).

5. Class #5: use the `type` keyword to define types.

3.3 Size constraints

Often, more efficient code can be generated when the array (e.g., vector, matrix, cube, ...) dimensions are known at compile-time. For example, when dimensions are known, certain types of for-loops can be efficiently parallelized, without using dynamic kernel memory. For this purpose, array types can be annotated with their dimensions, such as in the following table:

Data type	Example	Purpose
`vecX` or `vec(X)`	`vec4`	A vector of length X
`matXxY` or `mat(X,Y)`	`mat2x3`	A matrix of size $X \times Y$
`cubeXxYxW` or `cube(X,Y,W)`	`cube2x2x3`	A cube of size $X \times Y \times W$
`cubeXxYxWxZ` or `cube(X,Y,W,Z)`	`cube2x2x2x2`	A hypercube of size $X \times Y \times W \times Z$

There are two equivalent ways to annotate the size: either `vecX` or `vec(X)`. The former notation can only be used when X (or Y or Z) are numeric. A type parameter can be used as well as long as the dimensions can be determined statically (e.g., as part of a generic function, see section §6.7).
There are several advantages of annotating types with size constraints:

- A known dimension becomes a constant rather than a parameter that must be passed to a kernel/device function. Constants can be propagated and simplify indexing expressions.
- By specifying the array dimensions through the type, the compiler and runtime can automatically check the dimensions. Out-of-bounds array accesses and dimensioning problems can easily be detected at compile-time.
- Additionally, the code generator may map the types onto stack memory or even registers, which also avoids dynamic allocation overhead.
- In generic functions, generic size parameters allows coupling arguments of different matrices. For example:

```
function C = generic_matrix_vector_mult[M,N](A : mat(M,N), B : vec(N))
   ...
endfunction
```

  Here, the size constraints will be checked by the compiler (and/or runtime) and an error will be reported when the dimensions of the input matrix/vector are not correct.

Variables of fixed-size array types are always passed by reference, just like array types without dimension constraint. Hence, the dimension constraint acts as a contract for the type, but omitting the contract will not change the program behavior.
The type inference will automatically determine that the type of an expression `zeros(2,2)`, `[[0,1],[1,0]]` is `mat(2,2)`.
Also, incompatibilities in matrix/vector operations can be detected at compile-time in some cases, for example:

```
y = [1,2,3] * eye(4)
```

The use of array size constraints is currently the best approach to avoid the use of dynamic kernel memory (which incurs a performance penalty), see section §8.3.
When array dimensions are only *partially* known, this can be annotated using `:`, as in `cube(:,:,3)`. This is useful for example to indicate the number of color channels of an image. Both the width and height are then unspecified (this may depend on an input file). To make code more easily extendible and also to facilitate compiler analysis (for loop parallelization, type inference), partially parametrized array types can be used.
Example:

```
img : cube(:,:,3) = imread("lena_big.tif")
```

indicates that `A` is a cube with three dimensions, for which the first two dimensions have unspecified length (`:`) and the last dimension has length 3. When a for-loop is written over the third dimension, either explicitly or implicitly via matrix slices:

```
img[m,n,:]
```

the compiler can determine the length `numel(img[m,n,:])==3` so that the vector type `vec3` can be assigned.
Implementation note: *for arrays with number of elements <= 64, the stack memory (or registers) will be used, while for >64, the arrays will be allocated in dynamic kernel memory, which has some computational overhead (see section §8.3). To maximize performance, it is best to keep the arrays short. In future versions of Quasar, the constant 64 may be increased or be made more adaptable.*

3.4 Dimension constraints

In some cases it is useful to indicate the dimensionality of an array type directly through a parameter. This can be achieved using dimension constraints, using the notation `cube{n}`. The following table gives an overview of some common types:

Data type	Equivalent to	Meaning
`cube{1}`	`vec`	vector
`cube{2}`	`mat`	matrix
`cube{3}`	`cube`	3 dimensional array (cube)
`cube{4}`	-	4 dimensional array (hypercube)
`cube{N}`	-	`N` dimensional array (`N`=constant)
`cube{:}`	-	Infinite dimensional array

The type cube{:} is useful in function definitions. It indicates that the function accepts arrays of arbitrary dimension. It is currently not possible to construct infinite dimensional arrays (in fact the maximum dimensionality is currently set internally to 16, which should be enough for most practical purposes).
Dimension-parametrized arrays such as `cube{N}` are useful for describing algorithms that do not depend on a certain dimensionality of the input data. Because the dimensionality is known at compile-time, the compiler can still generate efficient indexing code. For more information, see section §6.8.

3.5 Cell array types

Cell arrays are structured but unnamed array types. In Quasar, each element of an array can have a different type. For example, a 2×2 cell matrix type with vectors in the first row and scalar values in the second row is given by:

```
mat[{{vec, vec},
     {scalar, scalar}}]
```

Cell array types (CATs) can therefore be seen as a generalization of tuple types to multiple dimensions. Arbitrary numbers of dimensions can be defined by nesting `{}`. CATs mostly occur during type inference and are not often used in user-code. In particular, CATs allow the type inference to succeed (and correspondingly efficient parallelization and other optimizations) in places where inhomogeneous data is stored in a cell matrix.
As a multidimensional generalization of tuples, cell array types can also be used to define lambda expressions (see section 4.6) with multiple return values:

```
get_string_and_value : [scalar -> vec[{string,scalar}]] =
     value -> {"String", value}
[str, val] = get_string_and_value(1.234)
```

Note that the right handed side `{"String", value}` constructs a cell vector.

3.6 Type constructors and the typename function

By calling the type as a function, for many types such as scalars and vectors/matrices, ..., an instance of the type can be constructed. This is particularly useful for generic code (see section §6) but also for constructing values of types for which no default creation function exists (for example `zeros` can be used to create a vector value of type

vec[scalar], but the function cannot create an integer vector of type vec[int]). Below are a number of examples of type constructors:

```
integer_vec = vec[int](200) % integer vector of length 200
uint8_vec4 = vec[uint8](4) % a 32-bit value containing four 8-bit unsigned integers
integer_val = typename(scalar)(1) % a scalar with value 1
typed_cell = typename(mat[{{vec,vec},{scalar,scalar}}])(2,2) % typed cell matrix
hyper_cube = typename(cube{4}[int16])(2,2,2,2) % 4-D array of int16
half_vec = typename(vec[scalar'half])(4) % half precision float vector of length 4
```

Quasar types are parsed using a type parser which is separate from the main parser. Therefore, with exception of the type annotation operator :, type names cannot be directly placed inside regular statements. As an alternative Quasar now provides the builtin function typename, so that type names can be given inline and so that instances of these types can be constructed more easily (in the past a type alias was required for this purpose).
For example, typename(vec[scalar'half])(4) creates a half-precision vector value of length 4. This of practical relevance for SIMD processing (see further in section §12).

3.7 Type classes

Type classes allow the type range of the input parameters to be narrowed. For example:

```
[int|cube]
```

denotes a type for a variable that can either be int or cube. The compiler will always attempt to simplify the type class definition. For example, [int'const|int] becomes int, because the const modifier does not make much sense here.
When such simplifications are not possible, the variables of these types are automatically polymorphic (see section 4.1). Nevertheless, type classes allow restricting the possible types that a function parameter can have (which is better than not specifying any type at all!), allowing the compiler to still proceed with the type inference. For example:

```
function y = diag(x : [mat|cmat])
```

This construction only allows variables of the type 'mat' and 'cmat' to be passed to the function. This is useful when it is already known in advance which types are relevant (in this case a real-valued or complex-valued matrix). Equivalently, type class aliases can be defined. The type:

```
type AllInt : [int|int8|int16|int32|uint8|uint32|uint64]
```

groups all integer types. Several functions of the Quasar standard library use type classes to allow the functions to be applicable to a wide range of input types.

3.8 Class / user defined type (UDT) definitions

User-defined types are used for storing structured data. Although Quasar currently does not support class member functions, inheritance or interfaces, there are some ways to simulate this behavior and obtain the same functionality.

Class member functions To define class member functions, the special `reduction` keyword (see section 4.9) can be used. First, a general function should be defined using a lambda expression or function definition (e.g. `point_distance` in the example below). Next, a reduction can be used to redirect the member function `p.distanceto` to the `point_distance` function.

```
type point : class
    x : scalar
    y : scalar
endtype

reduction (p : point, q : point) -> p.distanceto(q) = point_distance(p, q)

p = point(x:=4, y:=5);
q = point(x:=2, y:=1)
print p.distanceto(q)
```

Interfaces Defining an interface can be achieved by defining a user-defined class of function variables:

```
type my_interface : class
    times2_function : [scalar -> scalar]
    sum_function : [?? -> ??]
endtype

obj = my_interface(
    times2_function := (x : scalar) -> 2*x,
    sum_function := (x) -> sum(x)
)

print obj.times2_function(2)
print obj.sum_function([1,2,3])
```

Interface initialization requires all fields to be set. This enforces that are interface methods are implemented.

3.9 Function types

Function types have the following form:

```
[__modifier__ (inArgType1,...,inArgTypeN)->(outArgType1,...,outArgTypeN)]
```

where __modifier__ is either `__kernel__` (indicating a kernel function type), `__device__` (indicating a device function type) or entirely omitted (host function types). Below is an example of a function taking a matrix and a vector, with no output parameters:

```
[(mat,vec)->()]
```

A device function with no input arguments and one output argument (with unspecified type) has the following type:

```
[__device__()->??]
```

A variadic function taking a cube followed by an arbitrary number of scalar values can have the type:

```
[(cube,...scalar)->()]
```


Only kernel/device function types can be passed to kernel/device functions. If function parameters have a function type set, the compiler and/or runtime system will check the function variable that is passed, and generate an error if there is a type mismatch.

3.10 Enumerations

Enumeration types, which contain an ordering of named values, are defined as in the following example:

```
type color : enum
    Red
    Green
    Blue = 2
    Cyan
    Magenta
    Yellow
endtype
```

Enumerations are treated as integer types. To each named value (key) an integer value is assigned, that starts counting from 0, unless a value is explicitly assigned (like 2 in the above example). The individual values can be accessed with the dot-syntax:

```
my_color = color.Magenta
```

One exception are matches, which do not require the enum class name to be specified, because it is clear from the context:

```
match a with
| Red -> print ("The color is red")
| Green -> print ("The color is green")
| _ -> print ("Unknown color")
endmatch
```

3.11 Passed by reference / Passed by value

The semantics of whether a given variable is passed to a function by reference or by value, generally depends on the type of the variable. Scalar types are passed by value, while vectors, matrices and cubes are passed by reference. In the first case (pass by value), a function can modify the variable *locally*, but the effects of the change are not visible outside the function. In the second case (pass by reference), the function can modify the vector or matrix elements of the variable, and these modifications are visible to the calling function.[1] This way, some run-time overhead involved with copying matrices can be avoided.
An overview of the variable passing conventions is given in table 3.1. Some dynamic types (`string`, `object`, `type`) are not supported by kernel/device functions, therefore the table lists not applicable (N/A) for these types.

3.12 Constants

To enforce that a by reference variable cannot be modified by the function, it is possible to add the `'const` modifier to the type definition (see section 16.6), as in the following example:

[1] This is in contrast to some other programming languages, like MATLAB.

Table 3.1: Overview of the variable passing conventions.

Type	host function	device function (host call)	kernel function	device function (device call)
`scalar`, `cscalar`	value	value	value	value
`vecx`, `ivecx`, `cvecx`	reference	reference	reference	reference
`vec`, `mat`, `cube`	reference	reference	reference	reference
`string`	reference	reference	N/A	N/A
`function`	reference	reference	reference	reference
`class`	value	value	value	value
`^class`	reference	reference	reference	reference
`object`, `type`	reference	reference	N/A	N/A
`??`	reference*	reference*	N/A	N/A

*: unless the underlying value is scalar

```
function [] = compute(A : mat'const)
...
endfunction
```

Explicitly declaring `'const` is mostly useful for preventing accidental writes to the `A` matrix. For other purposes, it is not needed to manually annotate types with the `'const` modifier because the compiler will add this modifier automatically.
In general, constant variable types have the following benefits:

- By using constant values, redundant memory transfer operations between the different computation devices can be avoided.
- Constant values may propagate to functions, avoiding use of closures and function pointers, leading to a better execution performance.
- Quasar may use GPU features (e.g., non-coherent texture cache) to accelerate access to constant memory.
- Constant values may be used as size and dimension parameters of types. For example `cube{N}`.

Below an example is given of such constant value propagation. Here, by declaring the variable `half_wnd_size` as a constant, the value of `half_wnd_size` can be substituted into the two for-loops (with iterators `k` and `l`), leading to additional optimizations such as loop unrolling.

```
half_wnd_size : 'const = 4
function y = mean_filter(x : mat'circular)
    y = zeros(size(x))
    tmp : mat'circular = zeros(size(x))
    for m=0..size(y,0)-1
        for n=0..size(y,1)-1
            sum = 0.0
            for k=-half_wnd_size..half_wnd_size
                sum += x[m,n+k]
            endfor
            tmp[m,n] = sum
        endfor
    endfor
    for m=0..size(y,0)-1
        for n=0..size(y,1)-1
            sum = 0.0
```



```
            for l=-half_wnd_size..half_wnd_size
                sum += tmp[m+l,n]
            endfor
            y[m,n] = sum
        endfor
    endfor
endfunction
```


Chapter 4

Programming concepts

This section covers some extra advanced concepts that can help in writing efficient and easily readable Quasar programs.

4.1 Polymorphic variables

In Quasar, the variable data types are usually deduced from the context. The data type of a variable usually does not change. Polymorphic variables are variables for which the data type changes throughout the program. A common example is the calculation of the sum of a set of matrices:

```
v = vec[cube](10)
a = 0.0
for k=0..numel(v)-1
        a += v[k]
endfor
```

Here, the type of `a` is initially `scalar`, however, inside the for-loop the type becomes `cube` (because the sum of variables of type `scalar` and `cube` has type `cube`). Polymorphic variables are particularly useful for rapid prototyping. Note that for maximal efficiency, polymorphic variables should rather be *avoided*. When the compiler knows that a variable is not-polymorphic, type-static code can often be generated (i.e. as if you have declared the types of all the variables). The above example can be replaced by:

```
v = vec[cube](10)
a = v[0]
for k=1..numel(v)-1
        a += v[k]
endfor
```

Another side issue of polymorphic variables is that the automatic loop parallelizer may have more difficulties making assumptions with respect to the type of the variable at a given time. Therefore, the code fragment with the polymorphic variable may not be parallelized/serialized (even though a warning will be generated by the compiler). Of course, it is up to the programmer to decide whether a variable is allowed to be polymorphic or not.

However, when a variable is assigned with values of *different* array types (for example, with different sizes, different dimensionalities), the type of the variable will be *widened* as follows:

- The dimenionality of the resulting type is the maximum of the dimensionality of the different array types (`(vec, mat) -> mat`)
- When arrays have different size constraints (see section §3.3), the corresponding size constraint for the specified dimension will be *dropped* (`(vec(4), vec(8) -> vec`, `(mat(2,2), mat(3,2)) -> mat(:,2)`).

Type widening allows the compiler to statically determine the type of the variable.

4.2 Closures

A closure allows a function or lambda expression to access those non-local variables even when invoked outside of its immediate scope. In Quasar, its immediate use lies in the pre-computation of certain data, that is then used repeatedly, for example in an iterative method. Consider the following example:

```
function f : [cube->cube] = filter(name)
    match name with
    | "Laplacian" ->
        mask = [[0,-1,0],[-1,4,-1],[0,-1,0]]
        f = x -> imfilter(x, mask)
    | "Gaussian" ->
        ...
    endmatch
endfunction
im = imread("image.tif")
y = filter("Laplacian")(im)
```

What happens: the filter mask "`mask`" is pre-computed inside the function `filter`, but is seen as a non-local variable to the lambda expression `f = x -> imfilter(x, mask)`. Now, when this lambda expression is initialized, it stores a reference to the data of `mask` with it. The lambda expression is then returned as an output of the function filter, which then appears as a generic function of type `cube -> cube`. This way, even when the function `filter` is called repeatedly, `mask` only needs to be initialized once.
An even more interesting usage pattern, is to use closure variables inside kernel functions:

```
function y : mat = gamma_correction(x : mat, gamma : scalar)
    function [] = __kernel__ my_kernel(pos : ivec2)
       y[pos] = x[pos] ^ gamma
    endfunction
    y = uninit(size(x))
    parallel_do(size(x), my_kernel)
endfunction
```

Here, the kernel function `my_kernel` can access the variables `x`, `y`, `gamma` defined in the *outer* scope! The above function can be written more compactly using a lambda expression:

```
function y : mat = gamma_correction(x : mat, gamma : scalar)
    y = uninit(size(x))
    parallel_do(size(x), __kernel__ (pos : ivec2) -> y[pos] = x[pos]^gamma)
endfunction
```


Device functions also support closure variables. Practically, this means that they can have a *memory*:

```
gamma = 2.4
lut = ((0..255)/255).^gamma*255
gamma_correction = __device__ (x : scalar) -> lut[x]
function y : cube = pointwise_op(x : cube, fn : [__device__ scalar->scalar])
    y = uninit(size(x))
    parallel_do(size(x), __kernel__ (pos : ivec3) -> y[pos] = fn(x[pos]))
endfunction

im=imread("lena_big.tif")
y = pointwise_op(im, gamma_correction)
imshow(y)
```

Here, the lookup table `lut` is initialized, the `gamma_correction` function has type `[__device__ scalar->scalar]`, and performs the gamma correction using the specified lookup table. The advantage of this technique, is that the function `lut` does not need to be passed separately to the function `pointwise_op`, which makes is somewhat simpler to write generic code.
Correspondingly, a function definition defines the signature of a single function (which is similar to an interface with one member function in other programming languages such as Java/C++):

```
type binary_function : [__device__ (scalar, scalar) -> scalar]
```

The implementation can then still use internal or private variables, defined using function closures.

Closure variables are read-only An important remark: to avoid side effects, closure variables in Quasar are read-only! When attempts are made to change the value of a closure variable, a compiler error will be raised. The reason is illustrated in the following example:

```
a = 1
function [] = __device__ accumulate(x : scalar)
    a += x % COMPILER ERROR: a is READ-ONLY
    a = a + x % COMPILER WARNING: a is a "new" copy,
                changes are only visible locally
endfunction

accumulate(4)
print a % The result is 1
```

In this example, the variable `a`, which is defined outside the function accumulate, is changed using the operator `+=`, every time the function `accumulate` is called. This is not desirable, as this side effect can be very easily overlooked by the programmer: firstly, all variable definitions in Quasar are implicit, making it even more difficult to detect where the variable is actually declared. Secondly, the function accumulate may be passed as a return value to another function, and then the variable `a` may not exist anymore (apart from its reference).
The second syntax (`a = a + x`), however, is legal, but will generate a compiler warning, suggesting the programmer to choose another name for the variable `a`. In this case, the statement has to be interpreted as $a_{\text{inner}} \triangleq a_{\text{outer}} + x$, where '$\triangleq$' defines a variable declaration, a_{inner} is the local variable of the function, and a_{outer} refers to the non-local variable. This way, changes to `a` only happen locally, without causing side effects to the outer context.
Another benefit of the constant-ness of closure variables for GPU computation devices, is that the closure variable values needs to be transferred to the device memory, but not back!

Closure variables are bound at the function definition In contrary to some other programming languages, closure variables are bound at the moment the function/lambda expression is defined. This means that their values are stored within the function variable. Therefore, the following example will result in the value 18 being printed to the console:

```
A = ones(4,4)
D = __device__ (x : scalar) -> sum(A) + x % A is bound to D

A = ones(2,2)
print D(0) % prints 16
```

Note however, that vectors and matrices are passed by reference, therefore, modifying the vector and matrix elements of the bound closure variables *will* be visible by the device function.

4.3 Device functions, kernel functions, host functions

As already mentioned in section 2.4, there are three types of functions in Quasar: device functions, kernel functions and host functions. There are strict rules about how functions of a different type can call each other:

- Both `__kernel__` and `__device__` functions are low-level functions, they are natively compiled for CPU and/or GPU. This has the practical consequence that the functionality available for these functions is *restricted*. It is for example not possible to `load` or `save` information inside kernel or device functions. On the contrary, the `print` function *is* supported, but only for `string`, `scalar`, `int`, `cscalar`, `vecX`, `ivecX` and `cvecX` datatypes.
- Host functions are high-level functions, typically they are interpreted (or Quasar EXE's, compiled using the just-in-time compiler).
- A kernel function is normally repeated for every element of a matrix. Kernel functions can only be called using the `parallel_do`/`serial_do` functions.
- A device function can be called from host code or from other device/kernel functions.
- Kernel and device functions can call other kernel functions, through `parallel_do`/`serial_do` (nested parallelism, see section §4.4).

The distinction between these three types of functions is necessary to allow GPU programming. Furthermore, it provides a mechanism to balance the work between CPU/GPU. To find out whether the code will be run on GPU/CPU, the following recipe can be used:

- Kernel functions are a candidate to run on the GPU. When the kernel function is sufficiently "heavy" (i.e. data dimensions $\geq$ 1024, branches, thread synchronization), there is a high likelihood that the function will be executed on the GPU.
- Device functions run on the GPU (when called from a kernel function that is launched on the GPU) or on the CPU.
- Host functions run exclusively on the CPU.

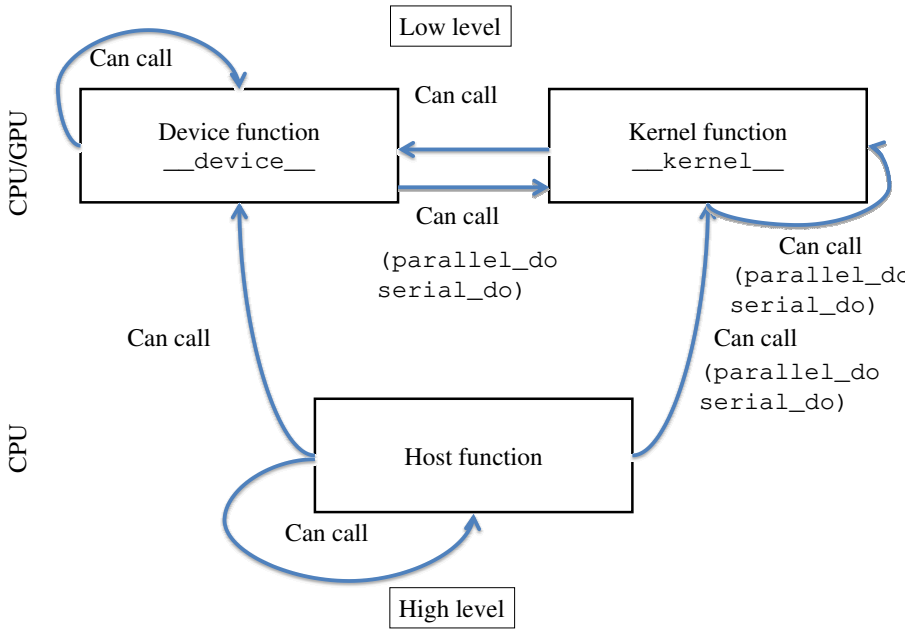


Figure 4.1: Relationship between the different function types in Quasar.

4.4 Nested parallelism

`__kernel__` and `__device__` functions can also launch nested kernel functions using the `parallel_do` (and `serial_do`) functions. The top-level host function may for example launch 30 threads (see figure 4.2), from which every of these 30 threads spans again 12 threads (after some algorithm-specific initialization steps). There are several advantages of this approach:

- More flexibility in expressing the algorithms
- The nested kernel functions are mapped onto CUDA dynamic parallelism on Kepler devices such as the Geforce GTX 780, Geforce GTX Titan or newer.
- When a `parallel_do` is placed inside a `__device__` function that is called directly from the host code (CPU computation device), the `parallel_do` will be accelerated using OpenMP.

Notes:

- There is no guarantee that the CPU/GPU will effectively perform the nested operations in parallel. However, future GPUs may be expected to become more efficient in handling parallelism on different levels.
- The setting "dynamic parallelism" needs to be enabled. By default, inner kernels are executed sequentially.

Limitations:

- Nested kernel functions may not use shared memory (they can access the shared memory through the calling function however), texture memory, and they may also not use thread synchronization.
- Currently only one built-in parameter for the nested kernel functions is supported: `pos` (and not `blkpos`, `blkidx` or `blkdim`).

In fact, when nesting `parallel_do` and `serial_do` and considering two levels of nesting, there are four possibilities:

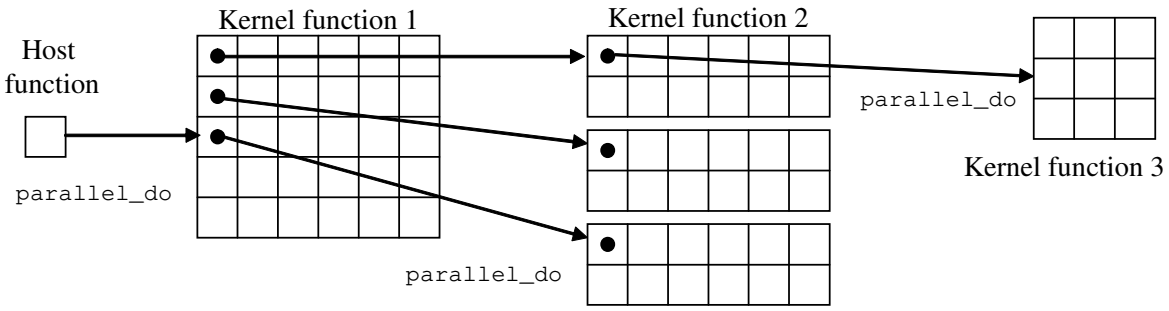


Figure 4.2: Illustration of nested parallelism.

Outer operation	Inner operation	Result
`serial_do`	`serial_do`	Sequential execution on CPU
`serial_do`	`parallel_do`	Parallel execution on CPU via OpenMP
`parallel_do`	`serial_do`	Execution on GPU, inner kernel is executed sequentially
`parallel_do`	`parallel_do`	Execution on GPU, sequentially (default) or CUDA dynamic parallelism (when enabled)

4.5 Function overloading

To implement functions taking different argument with different types, the most simple approach is to check the types of the function at runtime. Consider for example the following function that computes the Hermitian transpose of a matrix:

```
function y = herm_transpose(x:[scalar|cscalar|mat|cmat])
    if isscalar(x)
        y = conj(x)
    elseif isreal(x)
        y = transpose(x)
    elseif iscomplex(x)
        y = conj(transpose(x))
    endif
endfunction
```

Although this technique is legal in Quasar, it has two important disadvantages:

1. Type inference is difficult: the compiler cannot uniquely determine the type of the result of `herm_transpose(x)`, because the type depends on the type of `x` (and the conditions used in the `if` clauses). Instead, the compiler will assume that the resulting type is unknown (`'??'`). Hence, several optimizations (such as loop parallelization) that apply to code blocks that make use of the result of `herm_transpose(x)`, will be disabled.

2. Runtime checking of variable types creates some additional overhead, while in this case this could be handled perfectly by the compiler.

For these reasons, Quasar supports function overloading. The above function could be implemented as follows:

```
function y = herm_transpose(x : cscalar)
    y = conj(x)
endfunction
function y = herm_transpose(x : scalar)
```



```
    y = x
endfunction
function y = herm_transpose(x : mat)
    y = transpose(x)
endfunction
function y = herm_transpose(x : cmat)
    y = conj(transpose(x))
endfunction
```

In case none of the definitions apply, the compiler will generate an error stating that the function `herm_transpose` is not defined for the given input variable types. The overload resolution (i.e. the method the compiler uses for selecting the correct overload), follows the rules of the reduction resolution, which will be discussed in section 4.9. Note that the function overloading has the following restrictions:

- all function overloads must reside within the same module.
- all function overloads must be defined at the global scope (i.e. not inside another functions).
- only host and device functions can be overloaded, not `__kernel__` functions or lambda expressions.
- function overloads must differ in number of input arguments (or input argument types). Thereby, differences in output arguments are ignored.
- it is *not possible* to obtain a function handle of an overloaded function. Internally, overloaded functions are converted by the compiler to reductions (see section 4.9).

Finally, when the type of the input variables is not known to the compiler, the overload resolution will be performed at runtime.

4.5.1 Device function overloading

Some functions, especially functions from the standard libraries, are often used in different circumstances:

1. within a host function, e.g., to process one single huge matrix
2. from a kernel/device function, where the function is to be executed with many threads in parallel.

One notable example is the `svd` (singular value decomposition) function: when called from a device function, `svd` needs to be specialized for small matrices, while in a host function, `svd` may need to deal with large matrices (resulting in entirely different parallel implementations). Therefore, the host function can be overloaded using a device function with exactly the same signature (apart from the `__device__` specifier):

```
function [p:mat,d:mat,q:mat] = svd(a:mat)
   ...
endfunction
function [p:mat,d:mat,q:mat] = __device__ svd(a:mat)
   ...
endfunction
```

when `svd` is then user from a kernel/device function, the device overload is used. On the other hand, when called from a host function, the host overload will be used. In case only a device function is provided (and no host function overload), then the device function can also still be called from the host function (see figure 4.1) but then it is not possible to have different implementations.

Overloading is possible for size and dimension constraints (see 3.4). This allows the svd function to be overloaded for 2×2 matrices:

```
function [p:mat2x2,d:mat2x2,q:mat2x2] = __device__ svd(a:mat2x2)
   ...
endfunction
```

The compiler then exploits the available type information to find the most efficient implementation for the given type.

4.5.2 Optional function parameters

It is possible to declare values for optional function parameters. When the parameter is not used, the specified default value is used. For example,

```
function y = func1(x = eye(4))
function y = func2(x = eye(4), y = [[1,2],[3,4]])

func1() % same as func1(eye(4))
func1(eye(5))
func2(x:=eye(3), y:=randn(6))
func2(y:=4)
```

Named optional parameters can be specified through the `x:=value` syntax. This is mainly useful when for example the first optional argument will be omitted, but not the second.
As indicated in the above example, the optional values can be expressions. These expressions are evaluated when the function is called and when no argument is used. It is recommended to only use functions with no other side effects other than calculating the value of the optional parameter. The expressions may refer to other parameters, but *only* in the order that the parameters are passed:

```
function y = func3(x, y = 3*x)

func3(eye(4))
```

Here, by default `y = 3*eye(4)`, will be used.

4.6 Functions versus lambda expressions

In Quasar, a function is defined as follows:

```
function y = fused_multiply_add(a, b, c)
    y = a * b + c
endfunction
```

On the other hand, a lambda expression can be defined to compute the same result:

```
fused_multiply_add = (a, b, c) -> a * b + c
```

The question is then: what is the difference between functions and lambda expressions apart from their syntax? From a run-time perspective, lambda expressions and functions are treated in the same way in Quasar: both are functions of type `(??,??,??) -> ??`. The difference is only visible at compile-time:

Table 4.1: Comparison of functions and lambda expressions

	Function	Lambda expression
Optional arguments	Yes	No
Multiple output arguments	Yes	Yes
Supports overloading	Yes	No
`__kernel__`, `__device__`	Yes	Yes
Function closures	Yes	Yes
Function handles (`(??,??)->??`)	Yes	Yes
First-class citizen	Yes	Yes
Can contain control structures	Yes	No
Can contain nested functions	Yes	No
Can contain nested lambda expressions	Yes	Yes

- Functions can have *optional* arguments, whereas lambda expressions cannot.
- Functions are *named*, while lambda expressions are often *anonymous*.
- Functions can be *overloaded* (see section 4.5), in contrast to lambda expressions, which can not be overloaded. A second definition with the same name simply overwrites the first definition; moreover, the function variable may become *polymorphic*.
- Both functions and lambda expressions can have multiple output arguments. For lambda expressions, this requires the variable to be *explicitly typed* (see section 4.6.1).
- Both functions and lambda expressions can have closures (see section 4.2).
- Lambda expressions can have block statements `(aa; bb; cc)`; although they cannot contain control structures (`if`, `for`, `while`, etc.)

On the other hand, the definition of lambda expressions is more compact, and lambda expressions are often inlined by default by the compiler.
Hence, the programmer may choose whether a function is preferable for a given situation, or a lambda expression. A summary of the resemblances and differences between functions and lambda expressions is given in table 4.1.

4.6.1 Explicitly typed lambda expressions

A lambda expression can either be partially typed or explicitly typed. In the former case, the type of the lambda expression parameters are typed, which causes the return value of the lambda expression to be determined by type inference:

```
sincos = (x : scalar) -> [sin(x), cos(x)]
```

In the latter case, the variable capturing the lambda expression is explicitly typed:

```
sincos : [scalar -> (scalar, scalar)] =
    x -> {sin(x), cos(x)}
```

This syntax has two benefits:

- Lambda expressions with multiple output arguments can be defined. Whereas in the first example, the lambda expression returns a single output argument, i.e., vector of length two, in the second example, the lambda expression itself has two output arguments.

- The explicit type is applied recursively to inner lambda expressions. For example, the higher-level lambda expression (see section 4.10):

```
add : [scalar->scalar->scalar] = x -> y -> x + y
```

is equivalent to:

```
add : (x : scalar) -> (y : scalar) -> x + y
```

When the explicit type is subsequently replaced by type alias, multiple lambda expressions can share the same signature:

```
type binary_op : [scalar -> scalar -> scalar]
add : binary_op = x -> y -> x + y
sub : binary_op = x -> y -> x - y
```

Explicitly typed lambda expressions will be further used in the section on functional programming in Quasar (see 15).

4.7 Kernel function output arguments

To improve the syntax for kernel functions that have scalar output variables (e.g., sum, mean, standard deviation, ...), kernel output arguments are added as a *special* language feature to Quasar. The feature is *special*, because kernel functions intrinsically generate multiple output values, as they are applied to a typically large number of elements, while here there is only one value per output argument. The kernel output arguments are *shared* between all threads and all blocks. Moreover, the kernel output arguments are restricted to be of the type `scalar`, `cscalar`, `int`, `ivecX`, `vecX` and `cvecX`. The following example illustrates the use of kernel function output arguments:

```
function [y : int] = __kernel__ any(A : mat, pos : ivec2)
    if A[pos] != 0
        y = 1
    endif
endfunction
if parallel_do(size(A),A,any)
    print "At least one element of A is non-zero!"
endif
```

The function `any` returns 1 when at least one element of the input matrix `A` is nonzero. The variable y is initialized by zero by the `parallel_do` function, before the first call to `any` is made.
Kernel function output arguments are also subject to data races (see section 2.4.4), therefore atomic operations should be used! Remark that atomic operations also cause some overhead, and are only useful when there are only a small number of writes to the output arguments. In the following example, the sum of the elements of a sparse matrix `A` is computed.

```
function [sum : scalar] =
     __kernel__ stats(A : mat, pos : ivec2)

    if A[pos] != 0
       sum += A[pos]
```



```
    endif
endfunction
sum = parallel_do(size(A),A,stats)
```

This output argument accumulation approach is only recommended when the number of nonzero elements of `A` is small compared to the total number of elements of `A` (lets say, less than 1%). In other cases, implementation of the sum using parallel reductions (see section 10.6) is more efficient!

Important note After completion of the kernel function, the output arguments of the kernel function are directly copied back to the CPU memory. This may not be desirable because this creates an implicit device-host synchronization point. For multi-GPU processing (see 11), this may cause one or multiple GPUs to become idle. The solution in this case is to use input parameters instead of output parameters:

```
function [] = __kernel__ stats(sum : vec(1), A : mat, pos : ivec2)
    if A[pos] != 0
       sum[0] += A[pos]
    endif
endfunction
sum = zeros(1)
parallel_do(size(A),sum,A,stats)
```

The result is here stored in the vector `sum`. Only at the moment that the exact value of sum is needed (e.g., when evaluating `sum[0]`), a memory transfer and synchronization between GPU and CPU. This can be avoided in case a subsequently launched kernel function directly reads `sum[0]` via the device memory.

4.8 Variadic functions

Variadic functions are functions that can have a variable number of arguments. For example:

```
function [] = func(... args)
    for i=0..numel(args)-1
        print args[i]
    endfor
endfunction
func(1, 2, "hello")
```

Here, `args` is called a *rest* parameter (which is similar to ECMAScript 6). How does this work: when the function `func` is called, all arguments are packed in a cell vector which is passed to the function. Optionally, it is possible to specify the types of the arguments:

```
function [] = func(... args:vec[string])
```

which indicates that every argument must be a string, so that the resulting cell vector is a vector of strings.
Several library functions in Quasar already support variadic arguments (e.g. `print`, `plot`, ...), although now it is possible to define your own functions with variadic arguments.
Moreover, a function may have a set of fixed function parameters, optional function parameters and variadic parameters. The variadic parameters should *always* appear at the end of the function list (otherwise a compiler error will be generated)


```
function [] = func(a, b, opt1=1.0, opt2=2.0, ...args)
endfunction
```

This way, the caller of `func` can specify extra arguments when desired. This allows adding extra options for e.g., solvers.

4.8.1 Variadic device functions

It is also possible to define device functions supporting variadic arguments. These functions will be translated by the back-end compilers to use cell vectors with dynamically allocated memory (it is useful to consider that this may have a small performance cost).
An example:

```
function sum = __device__ mysum(... args:vec)
    sum = 0.0
    for i=0..numel(args)-1
        sum += args[i]
    endfor
endfunction

function [] = __kernel__ mykernel(y : vec, pos : int)
    y[pos]= mysum(11.0, 2.0, 3.0, 4.0)
endfunction
```

Note that variadic *kernel* functions are not supported.

4.8.2 Variadic function types

Variadic function types can be specified as follows:

```
fn : [(...??) -> ()]
fn2 : [(scalar, ...vec[scalar]) -> ()]
```

This way, functions can be declared that expect variadic functions:

```
function [] = helper(custom_print : [(...??) -> ()])
    custom_print("Stage", 1)
    ...
    custom_print("Stage", 2)
endfunction

function [] = myprint(...args)
    for i=0..numel(args)-1
        fprintf(f, "%s", args[i])
    endfor
endfunction

helper(myprint)
```


4.8.3 The spread operator

Unpacking vectors The spread operator unpacks one-dimensional vectors, allowing them to be used as function arguments or array indexers. For example:

```
pos = [1, 2]
x = im[...pos, 0]
```

In the last line, the vector `pos` is unpacked to `[pos[0], pos[1]]`, so that the last line is in fact equivalent with

```
x = im[pos[0], pos[1], 0]
```

Note that the spread syntax `...` makes the writing of the indexing operation a lot more convenient. An additional advantage is that the spread operator can be used, without knowing the length of the vector `pos`. Assume that you have a kernel function in which the dimension is not specified:

```
function [] = __kernel__ colortransform (X, Y, pos)
    Y[...pos, 0..2] = RGB2YUV(Y[...pos, 0..2])
endfunction
```

This way, the colortransform can be applied to a 2D RGB image, as well as a 3D RGB image. Similarly, if you have a function taking three arguments, such as:

```
luminance = (R,G,B) -> 0.2126 * R + 0.7152 * G + 0.0722 * B
```

Then, typically, to pass an RGB vector `c` to the function `luminance`, you would use:

```
c = [128, 42, 96]
luminance(c[0],c[1],c[2])
```

Using the spread operator, this can simply be done as follows:

```
luminance(...c)
```

Passing variadic arguments The spread operator also has a role when passing arguments to functions. Consider the following function which returns two output values:

```
function [a,b] = swap(A,B)
    [a,b] = [B,A]
endfunction
```

And we wish to pass both output values to one function

```
function [] = process(a, b)
    ...
endfunction
```

Then using the spread operator, this can be done in one line:

```
process(...swap(A,B))
```


Here, the multiple values `[a,b]` are unpacked before they are passed to the function `process`. This feature is particularly useful in combination with variadic functions.
Notes:

- Only vectors (i.e., with dimension 1) can currently be unpacked using the spread operator. This may change in the future.
- Within kernel/device functions, the spread operator is currently supported on vector types `vecX`, `cvecX`, `ivecX` (this means: the compiler should be able to statically determine the length of the vector).
- Within host functions, cell vectors can be unpacked as well.
- The spread operator can be used for concatenating vectors and scalars:

```
a = [1,2,3,4]
b = [6,7,8]
c = [...a, 4, ...b]
```

  where `c` will be a vector of length 8. For small vectors, this is certainly a good approach. For long vectors, this technique may have a poor performance, due to the concatenation being performed on the CPU. In the future, the automatic kernel generator may be extended, to generate efficient kernel functions for the concatenation.

4.8.4 Variadic output parameters

The output parameter list does not support the variadic syntax `...`. Instead, it is possible to return a cell vector of a variable length.

```
function [args] = func_returning_variadicargs()
    args = vec[??](10)
    args[0] = ...
endfunction
```

The resulting values can then be captured in the standard way as output parameters:

```
a = func_returning_variadicargs() % Captures the cell vector
[a] = func_variadicargs() % Captures the first element, and generates an
                          % error if more than one element is returned
[a, b] = func_variadicargs() % Captures the first and second elements and
                             % generates an error if more than one element
                             % is returned
```

Additionally, using the spread operator, the output parameter list can be unpacked and passed to any function:

```
myprint(...func_variadicargs())
```

4.9 Reductions

Quasar implements a very flexible compile-time graph reduction scheme to allow expression transformations. Reductions are defined inside Quasar programs through a special syntax and allow the compiler to "reason" about the operations being performed in the program, without having to evaluate these operations. The syntax is as follows:


```
reduction (var1:t1, ..., varN:tN) -> expr(var1,...,varN) = substitute(var1,...,varN)
```

Once the reduction has been defined, the compiler will attempt to apply the reduction each time an expression that matches with `expr` has been found. Expressions can be regular Quasar expressions and are not restricted to functions. For example, suppose that we have an efficient implementation for the fused multiply-add operation a+b*c, called `fmad(a,b,c)`, we can use this implementation for all combinations `a+b*c` that occur in the program. This is achieved by defining the following reduction:

```
reduction (a:cube, b:cube, c:scalar) -> a+b*c = fmad(a,b,c) _
    where size(a) == size(b)
```

Remark that we explicitly indicated the types of the variables `a`,`b` and `c` for which this reduction is applicable, together with a restriction on the sizes of `a` and `b` (`size(a) == size(b)`).
Reductions can also be used to define an alternative implementation for a cascade of functions:

```
reduction (x) -> real(ifft2(x)) = irealfft2(x)
```

Here, a complex->real (C2R) 2D FFT algorithm (implemented by `irealfft2(x)`) will be used to compute `real(ifft2(x))`. Because the C2R FFT operates on half the amount of memory of a complex->complex (C2C) FFT, the performance will be increased by roughly a factor of two!
Reductions are also ideal for some clever "trivial" optimizations:

```
reduction (x:mat) -> real(x) = x
reduction (x:mat) -> imag(x) = zeros(size(x))
reduction (x:mat) -> transpose(transpose(x)) = x
reduction (x:mat) -> x[:,:] = x
reduction (x:mat) -> real(transpose(x)) = transpose(real(x))
```

Using the above reductions, the compiler will simplify the following expression:

```
f = (x : mat) -> transpose(real(ifft2(fft2(transpose(x)))))
```

as follows:

```
Applied reduction ifft2(fft2(transpose(x))) -> transpose(x)
Applied reduction real(transpose(x)) -> transpose(x)
Applied reduction (x:mat) -> transpose(transpose(x)) -> x
Result after 3 reductions: f=(x:mat) -> x
```

Hence, the compiler finds that the operation `f(x)` is an identity operation! In this trivial example, we can assume that the programmer would have found the same result, however there are some situations that we will describe later in this section, in which the reduction technique can save a lot of time for the programmer.
Clearly, reductions bring the following benefits:

- Define once, optimal everywhere!
- More readable and clean optimized code compared to other programming languages that do not use reductions.
- The compiler can indicate some places in the code suited for optimization, but where e.g., some of the types of the variables is not known.

To avoid typing the same reduction with different types over and over again, type classes can be used (see section §3.7):

```
type RealNumber: [scalar|cube|AllInt|cube[AllInt]]
type ComplexNumber: [cscalar|ccube]
reduction (x : RealNumber) -> real(x) = x
reduction (x : ComplexNumber) -> complex(x) = x
```

Without type classes, the reduction would need to be written four times.

4.9.1 Symbolic variables and reductions

A special subset of the reductions are the *symbolic* reductions. Symbolic reductions often operate on variables that are "not defined" using the regular variable semantics. An example is given below:

```
reduction (x : scalar, a : scalar) -> diff(a, x) = 0
reduction (x : scalar, a : int) -> diff(x^a,x) = a*x^(a-1)
reduction (x, y, z : scalar) -> diff(x + y, z) = diff(x, z) + diff(y, z)
reduction (x : scalar, y : scalar) -> diff(x, y) = 0
reduction (x, y : scalar) -> diff(sin(x),y) = cos(x) * diff(x, y)

f = x -> diff(sin(x^4)+2,x) % Simplifies to 4*cos(x^4)*x^3
```

To be able to calculate derivatives with respect to variables that have not been defined/initialized, symbolic variables can be used, using the `symbolic` keyword:

```
symbolic x : int, y : scalar
```

These variables have no further meaning during the execution of the program. As such, during runtime, they do not exist. However, they help writing symbolic expressions:

```
reduction (f, x : scalar) -> argmin(f, x) = solve(diff(f, x) = 0, x)
symbolic x : scalar
print argmin((x-2)^2, x)
```

Here, the definition of `x` as a symbolic scalar is required, otherwise the compiler would not have any type information about `x`. Then, in case the compiler is not able to determine the minimum `argmin`, an error will be generated:

```
Line 3: Symbolic operation failed - no reductions available for 'argmin((x-2)^2, x)'
```

4.9.2 Reduction resolution

This subsection describes how Quasar decides which reduction to use at a particular time, and also in which order several reductions need to be applied. Suppose that we have an expression like:

```
reduction (A : mat, x : vec'col) -> A*x = f(x) % RED #1
reduction (A : mat, B : mat) -> norm(A, B) = sum((A-B).^2) % RED #2
g = (x : vec'col, b : vec'col) -> norm(A*x, b)
```

Then, both RED#1 and RED#2 can be applied. Quasar will prioritize reductions that have larger number of input variables (in this case RED#2, with input variables `A` and `B`). Reductions that having more variables are

generally more difficult to match (because they contain more conditions that need to be satisfied than reductions with for example 1 variable). Moreover, it is assumed that, in terms of expression optimization, reductions with more variables are designed to be more efficient. Therefore, the reduction will proceed as follows:

```
g = x -> sum((A*x-b).^2)
g = x -> sum((f(x)-b).^2)
```

In this case, the result is actually independent of the order of reduction application. However, there are cases where the order make play a role, such that the end result may differ. This is called a *reduction conflict*. Reduction conflicts will be further treated in section 4.9.3.
When the number of variables of two reductions is *equal*. Another criterion is needed to decide which reduction needs to be applied first. Quasar currently uses a three-level decision rule:

1. Prioritize reductions with the largest numbers of variables.
2. Prioritize *exact* matches. For example, `A:vec` may match a reduction with variables (`x:mat`), because `vec` $\subset$ `mat` (see section 2.2). However, when a reduction exists that has as input `x:vec`, this reduction will be prioritized.
3. Prioritize application to expressions with a higher depth in the expression tree representation. Sometimes the same reduction may be applied twice within the same expression. For example, in

   ```
   reduction x -> sum(x) = my_sum(x)
   f = x -> sum(sum(x))
   ```

   the sum reduction can be applied *twice*. The order is then from *right* to *left*, which enables correct type inference (the reduction to apply for the second step may depend on the type of `my_sum(x)`. In terms of an expression tree representation, this comes down to prioritizing applications with a higher depth in the expression tree (root=depth 0, children=depth 1, ...). Hence, the reduction proceeds as follows:

   ```
   f = x -> sum(my_sum(x))
   f = x -> my_sum(my_sum(x))
   ```

By these rules, the reduction application will work as "expected", and also for function applications (see section 4.5). Overloaded functions are in fact internally implemented in Quasar using reductions:

```
reduction (x : cscalar) -> herm_transpose(x) = herm_transpose_cscalar(x)
reduction (x : scalar) -> herm_transpose(x) = herm_transpose_scalar(x)
reduction (x : mat) -> herm_transpose(x) = herm_transpose_mat(x)
reduction (x : cmat) -> herm_transpose(x) = herm_transpose_cmat(x)
```

4.9.3 Ensuring safe reductions

If not used correctly, reductions may introduce errors (bugs) in the Quasar program that may be difficult to spot. To prevent this from happening, the Quasar compiler detects a number of situations in which the application of a reduction is considered to be *unsafe*. The reduction safety level can be configured using the `COMPILER_REDUCTION_SAFETYLEVEL` variable (see table 17.3). This variable can take the following values:

- `NONE`: perform no safety checks
- `SAFE`: perform safety checks and report a warning in case of a problem
- `STRICT`: generate an error in case "unsafe" reductions have been detected.

There are five situations in which a reduction is considered to be *unsafe*:

1. *Free variables in reduction*: the right handed side of the reduction contains a variable that is not present in the left handed side. For example:

```
reduction x -> f(x) + y
```

   Here the variable `y` causes a problem because the compiler does not have any information on this variable. It is hence unbound. The problem can be fixed in this case:

```
reduction (x, y) -> f(x) + y
```

2. *Undefined functions in reductions*: all functions in the right handed side of the reduction need to be defined in Quasar, either through standard definitions, or through other reductions.

3. *Reduction operands defined in non-local scope*: when some of the operands to which a reduction is applied to, are defined in a non-local context, side-effects maybe created in case these non-local variables are modified afterwards. For example, a change of type may cause the reduction application to be invalid at run-time, even though it seemed valid at compile-time. For example:

```
reduction x:mat -> ifft2(fft2(x)) = x
A = ones(4,4)
for k=1..10
    y = x:mat -> ifft2(fft2(x + A))
    A = load("myfile.dat") # may cause the reduction
                    # ifft2(fft2(x))=x to be invalid.
endfor
```

4. *Reduction conflicts*: sometimes, the result of the application of several reductions may depend on the order of the reductions. Usually, this is a result of poor definitions of the reductions, as demonstrated in the following example:

```
reduction (A : mat, B : mat, x : vec'row) -> norm(A*x, B) = f(A, B, x) % RED #1
reduction (A : mat, B : mat) -> norm(A, B) = sum((A-B).^2) % RED #2
g = (x : vec'row, b : vec'row) -> norm(A*x, b)
```

   In this example, we could either apply reduction #1 or reduction #2. According to the reduction resolution results (see section 4.9.2), the Quasar compiler will choose reduction #1 because it has three variables, `A`, `B` and `x`. However, applying reduction #2 would result in a completely different result `sum((A*x-B).^2)` and it is not guaranteed that `f(A, B, x)=sum((A*x-B).^2)`. The compiler detects this automatically and raises the reduction conflict error/warning whenever there is a problem. Here, the reduction conflict can be solved by defining:


```
reduction (A : mat, B : mat, x : vec'row) -> f(A, B, x) = sum((A*x-B).^2)
```

5. *Reduction cross-references*: circular dependencies may be created between reductions:

```
reduction (A, B) -> f(A, B) = g(A, B)
reduction (A, B) -> g(A, B) = f(A, B)
```

   This obviously is also not allowed and will generate a compiler error.

These rules allow to write safe Quasar reductions which cause no undesired side-effects.

4.9.4 Reduction where clauses

Reductions can also be applied in a conditional way. This is achieved by specifying a where clause. The where clause determines at compile time (or at runtime) whether a given reduction may be applied. There are two main use cases for where clauses:

1. To avoid invalid results: In some circumstances, applying certain reductions may lead to invalid results (for example a real-valued sqrt function applied to a complex-valued input, derivative of $\tan(x)$ in $\pi/2\ldots$)
2. For optimization purposes (e.g. allowing alternative calculation paths).

For example:

```
reduction (x : scalar) -> abs(x) = x where x >= 0 reduction (x : scalar) -> abs(x) = -x where x < 0
```

In case the compiler has no information on the sign of x, the following mapping is applied:

```
abs(x) -> x >= 0 ? x : (x < 0 ? -x : abs(x))
```

And the evaluation of the where clauses of the reduction is performed at runtime.
However, when the compiler has information on x (e.g. `assert(x = -1)`), the mapping will be much simpler:

```
abs(x) -> -x
```

Note that the `abs(.)` function is a trivial example, in practice this could be more complicated:

```
reduction (x : scalar) -> someop(x, a) = superfastop(x, a) where 0 <= a && a < 1
reduction (x : scalar) -> someop(x, a) = accurateop(x, a) where 1 <= a
```

There are also three special conditions that can be used inside reductions. These conditions are mainly used internally by the Quasar compiler, but can also be useful for certain user optimizations:

- `$ftype("__host__")`: is true only when the outer function is a host (i.e. non-kernel/device function)
- `$ftype("__device__")`: is true only when the outer function is a device function
- `$ftype("__kernel__")`: is true only when the outer function is a device function

These conditions reduce the applicability of the reduction depending on the outer function scope in which the reduction is to be applied. For example, it is possible to specify reductions that can only be used inside device functions, reductions for host functions etc.

4.9.5 Variadic reductions

In analogy to variadic functions (see 4.8), it is possible to define reductions with a variable number of arguments. The main use of variadic reductions is two-fold:

1. Providing a calling interface to variadic functions:

```
reduction (a : string, ...args : vec[??]) -> myprint(a, args) = handler(a, ...args)
```

   This avoids defining several reductions (one reduction for n number of parameters) in order to call variadic functions.

2. Capturing symbolic expressions with variable number of terms. Suppose that we want to define a reduction for the following expression:

$$\lambda_1 \left(f_1(x)\right)^2 + \lambda_2 \left(f_2(x)\right)^2 + ... + \lambda_K \left(f_K(x)\right)^2$$

   where $f_1, f_2, ..., f_K$ are functions to be captured, $\lambda_1, \lambda_2, ..., \lambda_K$ are scaling parameters and where the number of terms K is variable. A reduction for this expression can be defined as follows:

```
reduction (x, ...lambda, ...f) -> sum([lambda*f(x).^2]) = mysum(f,lambda)
```

   Here, `lambda` and `f` are variadic captures, their type is a cell vector of respectively scalars and functions (i.e., `vec` and `vec[[??->??]]`). Such reduction can match expressions like:

```
2*(x.^2) + cos(3)*((2*x).^2) + 2*((-x).^2)
```

4.10 Partial evaluation

Quasar has a complete implementation of lambda expressions, and also allows partial evaluation:

```
f = (x, y) -> x + y
g = y -> x -> f(x, y)
print g(4)(5) % Will return 9
```

Here, the partial evaluation `x -> f(x, y)`, returns a lambda expression that adds the free variable y to its input, x. Consider for example a linear solver, that solves $Ax = y$, using the function `x=lsolve(A, y)`. Suppose that we have a large number of linear systems that need to be solved. Then we can define the partial evaluation of `lsolve`:

```
lsolver = A -> y -> lsolve(A, y)
solver = lsolver(A)
for k=1..100
    x[k] = solver(y[k])
endfor
```

Similarly, we can have a lambda expression that solves a quadratic equation $Ax^2 + Bx + C = y$: `x=qsolve(A, B, C - y)`, by returning for example the largest solution:

```
qsolver = (A, B, C) -> y -> qsolve(A, B, C - y)
solver = qsolver(A, B, C)
```



```
for k=1..100
    x[k] = solver(y[k])
endfor
```

Now, `solver(y)` is a generic solver that can be used in other numerical techniques, while `lsolver` and `qsolver` can be used to create the desired solver.
It is also possible to define lambda expressions that have another lambda expression as input. The syntax is not `f = (x -> y) -> g(x)`, but:

```
h = (x : lambda_expr) -> f(x(y))
```

or, more type-safe versions:

```
h = (x : [?? -> ??]) -> f(x(y))
h = (x : [cube -> cube]) -> f(x(y))
```

For example, we can define a lambda expression that sums the output of two other lambda expressions `f1` and `f2`, again as another lambda expression:

```
f = (f1 : [?? -> ??], f2 : [?? -> ??]) = x -> f1(x) + f2(x)
```

or, even more generally, using reductions:

```
reduction (f1 : [?? -> ??], f2 : [?? -> ??], x) -> f1 + f2 = x -> f1(x) + f2(x)
f1 = x -> x * 2
f2 = x -> x / 2
f3 = f1 + f2 % Result: f3 = x -> x * 2 + x / 2
```

Recursive lambda expressions can be defined simply as:

```
factorial = (x : scalar) -> (x > 0) ? x * factorial(x - 1) : 1
```

One only needs to be careful that the recursion stops at a given point (otherwise a stack overflow error will be generated[1]).

4.11 Code attributes

Code attributes provide a means to incorporate meta-data in Quasar source code files. Additionally, code attributes also allow passing information to the compiler and runtime systems.
The general syntax of a code attribute is as follows:

```
{!attribute prop1=value1, ..., propN=valueN}
```

where `attribute` is the name of the code attribute, `prop1` and `propN` are parameter names and `value1` and `valueN` are the respective values.
The following code attributes are currently available:

[1] Note: tail recursion optimization is currently not supported, however, for device lambda functions the underlying back-end compiler may perform this optimization.

- Annotation data:

Code attribute	Purpose
`{!author name="" }`	Adds the name of the author to the source file. Multiple authors can be specified using a comma-separated list
`{!doc category=""}`	The category of the module in the documentation system
`{!copyright text=""}`	A copyright text for the current source code file
`{!description text=""}`	A brief description of the purpose of the source code file
`{!region name=""}`	Begins a region in the source code (which is automatically folded by the editor)
`{!endregion}`	Ends a region in the source code

- Compiler code attributes:

Code attribute	Purpose
`{!auto_inline}`	Automatically inlines the current function
`{!auto_vectorize}`	Automatically vectorizes code within the current block
`{!specialize args=(arglist)}`	Automatically specializes a function based on the specified argument list
`{!kernel target="cpu|gpu|..."}`	Sets the target of the kernel function (see section 17.2.5)
`{!kernel_transform enable="(transform)"}`	Enables a given kernel transform
`{!kernel_arg name=arg}`	Annotates kernel arguments with data that can be used during the optimization
`{!kernel_accumulator name=arg; type="+="}`	Annotates a variable as an accumulator (for parallel reductions). This code attribute is automatic and does not need to be used from user-code.
`{!kernel_shuffle dims=[1,0]}`	Shuffles the dimensions of a kernel function
`{!kernel_tiling dims=[1,4,1]; mode="global|local"; target="cpu|gpu"}`	Performs kernel tiling
`{!unroll times=4; multi_device =true; interleave=true}`	Unrolls the current loop

- Loop parallelization attributes (see section §8.2):

Code attribute	Purpose
`{!parallel for, dim=N}`	Parallelizes the following N-dimensional for-loop
`{!serial for, dim=N}`	Serializes the following N-dimensional for-loop
`{!interpreted for}`	Interprets the following for-loop

- Runtime attributes:

Code attribute	Purpose
`{!sched mode=cpu|gpu|auto`	Sets the current scheduling mode
`{!sched gpu_index=1}`	Sets the GPU index (for multi-GPU processing, see section §11)
`{!alloc mode=auto|cpu|gpu }`	Sets the memory allocation mode
`{!alloc type=auto| pinnedhostmem|texturemem|unifiedmem}`	Sets the memory allocation type
`{!transfer vars=a,b,c; target=cpu|gpu}`	Transfers variables `a,b,c` to the CPU or GPU

Code attributes that are not recognized by the compiler/runtime are simply ignored: no error or warning is generated!.

4.12 Macros

Macros allow to define *custom* code attributes, that have a specific effect at compile-time or at runtime. For example, a custom code attribute may be defined to combine different code attributes, with or without conditionals. Macros can be seen as functions that are expanded at compile-time:

```
macro [] = time(op) % timing macro
    tic()
    op()
    toc()
endmacro

function [] = operation()
    im = imread("lena_big.tif")[:,:,1]

    for m=0..size(im,0)-1
        for n=0..size(im,1)-1
            im[m,n] = 255 - im[m,n]
        endfor
    endfor
endfunction

{!time operation}
```

Macros can be defined to implement optimization profiles:

```
macro [] = backconv_optimization_profile(radius, channels_in, channels_out, dzdw, input)
    !specialize args=(radius==1 && channels_out==3 && channels_in==3 || _
                    radius==2 && channels_out==3 && channels_in==3 || _
                    radius==3 && channels_out==3 && channels_in==3 || _
                    radius==4 && channels_out==3 && channels_in==3 )

    if $target("gpu")
        % Small radii: apply some shared memory caching
        if radius <= 3
            % Calculate the amount of shared memory required by this transform
            shared_size = (2*radius+1)^2*channels_in*channels_out

            {!kernel_transform enable="localwindow"}
            {!kernel_transform enable="sharedmemcaching"}
            {!kernel_arg name=dzdw, type="inout", access="shared", op="+=", cache_slices=dzdw[:,:,pos
                [2],:], numel=shared_size
            !kernel_arg name=input, type="in", access="localwindow", min=[-radius,-radius,0], max=[
                radius,radius,0], numel=2048}
        endif
        {!kernel_tiling dims=[128,128,1], mode="global"}
    elseif $target("cpu")
                {!kernel_tiling dims=[128,128,1], mode="local"}
    endif
endmacro
```


The above macro instructs the compiler to perform certain optimizations depending on the provided parameter values. In the above example, depending on the value of `radius`, certain operations are enabled/disabled. Macros provide a mechanism to share optimizations between different algorithms, separating the algorithm specification from the implementation details.

4.13 Exception handling

Quasar supports a basic form of exception handling, similar to programming languages such as Java, C#, C++ etc. Currently the exception support is quite elementary, although it may be extended in the future.
A try-catch statement consists of a try-block followed by one more catch blocks. Whenever an exception generated within the try block, the handler in the catch block is immediately executed. The catch block received an exception object (`ex`) which has by default the type `qexception`.

```
try
    error "Fail again"
catch ex
    print "sigh: ", ex
endtry
```

In the future, it will be possible to derive classes from `qexception` and catch specific exceptions, although for the moment this is not possible yet.

4.14 Documentation conventions

Quasar uses Natural Docs (https://www.naturaldocs.org/) documentation conventions. This allows Natural Docs tools to be used in combination with Quasar code. Quasar Redshift (see section §18.1) also contains a builtin documentation viewer based on Natural Docs conventions. An example of Natural Docs for documenting a function is given below.

```
% Function: harris_cornerdetector
%
% Implementation of the harris corner detector
%
% :function y = harris_cornerdetector(x : cube, sigma : scalar = 1, k : scalar = 0.04, T : scalar =
    200)
%
% Parameters:
% x - input image (RGB or grayscale)
% sigma - the parameter of the Gaussian smoothing kernel
% k - parameter for the Harris corner metric (default value: 0.04)
% T - corner detection threshold (default value: 200)
function y = harris_cornerdetector(x : cube, sigma : scalar = 1, k : scalar = 0.05, T : scalar = 1)
```

Additionally, it is recommended to set `{!author name="" }` and `{!doc category="" }` code attributes at the beginning of a Quasar file. In particular, documentation category contains the path where the Quasar module is placed in the documentation browser. For example:

```
{!doc category="Image Processing/Multiresolution"}
```


See the `Library` folder for examples of selecting documentation categories. The Quasar libraries extensively use NaturalDocs conventions for documentation generation.

Chapter 5

The logic system

As many other programming languages, Quasar has an `assert` function. The `assert` function will evaluate the specified condition and will result in an error message when the condition is false. The `assert` function can be called with either one or two arguments:

```
assert(condition)
assert(condition, "condition is false")
```

In the second case, the error message is specified, which makes it easier for the user to resolve the issue. The `assert` function also gives hints to the compiler system (see section 5.3). When the compiler is able to figure out that the condition will *never* be true, a compiler error will be generated! Note that this is in contrast to most existing programming languages, for which `assert` is simply a run-time function. The algorithm for evaluating assertions is then as follows:

1. The compiler checks the condition of the assertion.
2. There are three possible outcomes: *valid*, *satisfiable* or *unsatisfiable:*
   a) If the result is *unsatisfiable*, a compile-time error will be generated.
   b) If the result is *satisfiable*, the compiler will take the condition as a hint.
   c) If the result is *valid*, the compiler is certain that the condition will be met in all situations. Therefore, the compiler may remove the assertion instruction from the program.

   The compiler is either able to recognize the condition (see section 5.3) or not able to do so. In the former case, a logic evaluation will be performed. In the latter case, the result is always *satisfiable*.
3. In case the result is *satisfiable*, the condition will still be checked at run-time.

The compiler is free to decide which assertions to take into account and also how to propagate information through the various compilation phases. The exact behavior may be controlled using compiler settings. For example, the following program may result in a compile-time error:

```
function [] = __kernel__ kernel (b : scalar, pos : ivec3)
    assert(b==3)
endfunction
parallel_do(size(im), 2, kernel)
```

Here, the constant parameter value for `b`, is passed through the `parallel_do` function to the kernel function `kernel`. Through automatic specialization techniques (see section 6.6), the Quasar compiler will know in this case that the value for `b` is 2, resulting in a compiler error. In the future, this behavior may be extended to even more complex scenarios.

5.1 Kernel function assertions

It is possible to call the `assert` function from a kernel or device function:

```
function [] = __kernel__ kernel (pos : ivec3)
    b = 2
    assert(b==3)
endfunction
```

Obviously, the above assertion fails. Quasar breaks with the following error message:

```
(parallel_do) test_kernel - assertion failed: line 23
```

Recall that the kernel function is typically called by many threads in parallel. Therefore, the following rules apply:

1. When the user program catches an assertion failure from a kernel function, there is *at least one* thread (or position `pos`) for which the condition failed.
2. It is currently not possible to retrieve the position that corresponds to assertion failure.[1]
3. The output of the kernel function is *undetermined*. Some threads may have completely finished, others may not have started. The order in which this happens is completely unspecified. In other words, when an assertion fails, the output of the kernel function should be ignored.

Kernel function assertions provide a very useful mechanism for directly debugging and verifying code on a CPU or GPU. The assertion system is also used internally by Quasar to perform vector and matrix boundary checking.

5.2 Built-in compiler functions

There are three meta functions that help with assertions. These functions are evaluated at compile-time (as indicated by the `$`-prefix)

- '`$check(proposition)`' checks whether the specified proposition can be satisfied, given the previous set of assertions, resulting in three possible values: `"Valid"`, `"Satisfiable"` or `"Unsatisfiable"`.
- '`$assump(variable)`' lists all assertions that are currently known about a variable, including the implicit type predicates that are obtained through type inference. Note that the result of '`$assump`' is an expression, so for visualization it may be necessary to convert it to a textual representation using '`$str(.)`' (to avoid the expression from being evaluated).

[1] Note that the kernel function debugger in Quasar Redshift can bring a solution here.

- '`$simplify(expr)`' simplifies logic expressions based on the knowledge that is inserted through assertions.

Usually, you will not need to call these functions directly from your code. Nevertheless, they can be useful for testing (for example in interactive mode).

5.3 Assertion types recognized by the compiler

There are different types of assertions recognized by the Quasar compiler. These assertions can be combined in a transparent way using the Boolean operators `!` (inversion), `&&` (and) and `||` (or).

5.3.1 Equalities

The most simple cases of assertions are the equality assertions `a==b`. For example:

```
symbolic a, b
assert(a==4 && b==6)
assert($check(a==5)=="Unsatisfiable")
assert($check(a==4)=="Valid")
assert($check(a!=4)=="Unsatisfiable")
assert($check(b==6)=="Valid")
assert($check(b==3)=="Unsatisfiable")
assert($check(b!=6)=="Unsatisfiable")
assert($check(a==4 && b==6)=="Valid")
assert($check(a==4 && b==5)=="Unsatisfiable")
assert($check(a==4 && b!=6)=="Unsatisfiable")
assert($check(a==4 || b==6)=="Valid")
assert($check(a==4 || b==7)=="Valid")
assert($check(a==3 || b==6)=="Valid")
assert($check(a==3 || b==5)=="Unsatisfiable")
assert($check(a!=4 || b==6)=="Valid")
print $str($assump(a)),",",$str($assump(b)) % prints (a==4),(b==6)
```

Here, we use `symbolic` to declare symbolic variables (variables that are not to be "evaluated", i.e. translated into their actual value since they don't have a specific value). Next, the function *assert* tests whether the `$check(.)` function works correctly (=self-checking).

5.3.2 Inequalities

The propositional logic system can also work with *inequalities*:

```
symbolic a
assert(a>2 && a<4)
assert($check(a>1)=="Valid")
assert($check(a>3)=="Satisfiable")
assert($check(a<3)=="Satisfiable")
assert($check(a<2)=="Unsatisfiable")
assert($check(a>4)=="Unsatisfiable")
assert($check(a<=2)=="Unsatisfiable")
assert($check(a>=2)=="Valid")
assert($check(a<=3)=="Satisfiable")
assert($check(!(a>3))=="Satisfiable")
```

The idea is here that the inequality assertions can help the simplification of if conditions. For example,


```
assert(x > 10)
if x > 0
   y = x
else
   y = -x
endif
```

In this case, the if-test can be completely eliminated thereby ignoring the else-block, because it is certain that `x` is positive.

5.3.3 Type assertions

Type assertions are useful for 1) checking whether a variable has a given type and 2) for giving hints to the compiler. For example, as mentioned in section 2.2, we may use a type assertion to make sure that data read from a file has the right type:

```
[A, B] = load("myfile.dat")
assert(type(A,"ccube") && type(B,"vec"))
```

Please note that assertions should *not* be used with the intention of variable type declaration. To declare the type of certain variables type annotations can be used:

```
[A : ccube, B : vec] = load("myfile.dat")
```

Type annotations should be used on the *first* occurrence of a variable. In this case, the type annotation prevents `A` and `B` from becoming a polymorphic variable (see section 4.1). For type assertions, there is no such requirement (they can be used in combination with a polymorphic variable).

5.4 User-defined properties

It is also possible to define "properties" of variables, using a symbolic declaration. For example:

```
symbolic is_a_hero, Jan_Aelterman
```

Then you can assert:

```
assert(is_a_hero(Jan_Aelterman))
```

Correspondingly, if you perform the test:

```
print $check(is_a_hero(Jan_Aelterman)) % Prints: Valid
print $check(!is_a_hero(Jan_Aelterman)) % Prints: Unsatisfiable
```

If you then try to assert the opposite:

```
assert(!is_a_hero(Jan_Aelterman))
```

The compiler will complain:

```
assert.q - Line 119: NO NO NO I don't believe this, can't be true!
Assertion '!(is_a_hero(Jan_Aelterman))' is contradictory with 'is_a_hero(Jan_Aelterman)'
```


5.5 Unassert

In some cases, it is neccesary to undo certain assertions that were previously made. For this task, the function '`unassert`' can be used:

```
unassert(propositions)
```

This function only has a meaning at compile-time; at run-time nothing needs to be done. For example, if you wish to reconsider the assertion '`is_a_hero(Jan_Aelterman)`' you can write:

```
unassert(is_a_hero(Jan_Aelterman))
print $check(is_a_hero(Jan_Aelterman)) % Prints: Satisfiable
print $check(!is_a_hero(Jan_Aelterman)) % Prints: Satisfiable
```

Alternatively you could have written:

```
unassert(!is_a_hero(Jan_Aelterman))
print $check(is_a_hero(Jan_Aelterman)) % Prints: Valid
print $check(!is_a_hero(Jan_Aelterman)) % Prints: Unsatisfiable
```

5.6 The role of assertions

In Quasar, the role of assertions is two-fold:

- It helps to early detect logical errors (mistakes by the programmer)
- It serves as a technique used for optimization. Firstly, assertions can specified upper bounds for variables, which help the compiler / code generator for the specific back-ends to generate more efficient code. Secondly, assertions can help eliminating branches in the code that are never used, as in the following example:
- ```
  assert(x > 0)
  if x > 20
      y = x - 20
  elseif x < -20
      y = x + 20
  else
      y = 0
  endif
  ```

  In this case, the branch `x < -20` can be completely eliminated, because it is known that `x > 20`.


  ```

Chapter 6

# Generic programming

Often, functions need to be duplicated for different container types (e.g. '`vec[int8]`', '`vec[scalar]`', '`vec[cscalar]`'). To avoid this duplication there is support for generic programming in Quasar.
Consider the following program that extracts the diagonal elements of a matrix and that is supposed to deal with arguments of either type '`mat`' or type '`cmat`':

```
function y : vec = diag(x : mat)
        assert(size(x,0)==size(x,1))
        N = size(x,0)
        y = zeros(N)
        parallel_do(size(y), __kernel__ (x:mat, y:vec, pos:int) -> y[pos] = x[pos,pos])
endfunction
function y : cvec = diag(x : cmat)
        assert(size(x,0)==size(x,1))
        N = size(x,0)
        y = czeros(N)
        parallel_do(size(y), __kernel__ (x:cmat, y:cvec, pos : int) -> y[pos] = x[pos,pos])
endfunction
```

Although function overloading here greatly solves part of the problem (at least from the user's perspective), there is still duplication of the function '`diag`'. In general, we would like to specify functions that can "work"irrespective of their underlying type.In Quasar, this is fairly easy to do:

```
function y = diag[T](x : mat[T])
        assert(size(x,0)==size(x,1))
        N = size(x,0)
    y = vec[T](N)
        parallel_do(size(y), __kernel__ (pos) -> y[pos] = x[pos,pos])
endfunction
```

As you can see, the types of the function signature have simply be omitted. The same holds for the '`__kernel__`' function.

In this example, the type parameter 'T' is required because it is needed for the construction of vector 'y' (through the 'vec[T]' constructor). If 'T==scalar', 'vec[T]' reduces to 'zeros', while if 'T==cscalar', 'vec[T]' reduces to 'czeros' (complex-valued zero matrix). In case the type parameter is not required, it can be dropped, as in the following example:

```
function [] = copy_mat(x, y)
        assert(size(x)==size(y))
        parallel_do(size(y), __kernel__ (pos) -> y[pos] = x[pos])
endfunction
```

Remarkably, this is still a generic function in Quasar; no special syntax is needed here.
Note that in previous versions of Quasar, all kernel function parameters needed to be explicitly *typed*. This is now no longer the case: the compiler will deduce the parameter types by calls to 'diag' and by applying the internal type inference mechanism. The same holds for the '__device__' functions.
When calling 'diag' with two different types of parameters (for example once with 'x:mat' and a second time with 'x:cmat'), the compiler will make two generic instantiations of 'diag'. Internally, the compiler may either:

1. Keep the generic definition (*type erasion*)

   ```
   function y = diag(x)
   ```

2. Make two instances of 'diag' (*reification*):

   ```
   function y : vec = diag(x : mat)
   function y : cvec = diag(x : cmat)
   ```

The compiler will combine these two techniques in a transparent way, such that: 1) for kernel-functions explicit code is generated for the specific data types and 2) for less performance-critical host code type erasion is used (to avoid code duplication).
The selection of the code to run is made at *compile-time*, so correspondingly the Quasar Spectroscope debugger has special support for this. Of course, when calling the 'diag' function with a variable of type that cannot be determined at compile-time, a compiler error is generated:

```
The type of the arguments ('op') needs to be fully defined for this function call!
```

6.1 Parametrized functions

As already mentioned, generic functions can be defined by just omitting the type declarations for the function parameters. For example, consider adding an item to a list (represented by a vector) at a given position.

```
function new_list = add_item(list, item, pos)
    ...
endfunction
```

However, very often it is desirable that the type relation between list and item is specified. For example, the type of list is 'vec[T]' where T is some type. This can be achieved using parametrized functions:


```
function new_list = add_item[T](list : vec[T], item : T, pos)
    ...
endfunction
```

then, when the function `add_item` is called, the compiler (or the runtime system) will check whether the types of `list` and `item` match. The type variable is available within the context of `add_item`. This easily allows variables to be defined with the same type as `item` or `list`:

```
function new_list = add_item[T](list : vec[T], pos, item : T)
    if pos > numel(list)
        % extend the list with new items
        new_list = vec[T](pos+1)
        new_list[0..numel(list)-1] = list
    else
        new_list = list
    endif
    new_list[pos] = item
endfunction
list1 = vec[int](10)
list2 = vec[string](8)
add_item(list1, 5, 4) %OK
add_item(list1, 4, "text") %Type mismatch
add_item(list2, 2, "let's try again") % OK
```

In some cases (for example when `T` only determines the output parameters), we wish to select the "version" of `add_item` that will be called. This can be done by filling in `T` explicitly (through a technique called generic function instantiation):

```
add_item[int](list1, 5, 4)
add_item[string](list2, 2, "let's try again")
```

This is particularly useful when defining functions that return generic objects:

```
function list = create_list[T](initial_length : int)
       list = vec[T](initial_length)
endfunction
my_list = create_list[int](10)
```

Parametric function themselves are variables and they have a certain type. In the above examples, the types of `add_item` and `create_list` are:

```
add_item : [(vec[??], ??, ??) -> vec[??]]
add_item[int] : [(vec[int], ??, int) -> vec[int]]
add_item[string] : [(vec[string], ??, string) -> vec[string]]
create_list[int] : [int -> vec[int]]
```

Remarks:

- Kernel and device functions can also be parametric. For the device-specific code, only the reification technique is used. The compiler will therefore rely on its type inference techniques to determine the types of all function parameters.

- Functions can have multiple type parameters. When one of the type parameters is not used, a compiler warning is given.
- In essence, generic programming in Quasar allows the programmer to write programs in which none of the data types needs to be specified. Consider the following example:

```
x = imread("lena_big.tif")
function [] = __kernel__ gamma_correction(x, pos)
       x[pos] = 255*(x[pos]/255)^0.5
endfunction
parallel_do(size(x), x, gamma_correction)
```

  Here, the compiler is able to determine the type of `x` (`'cube'`) and from this information the compiler finds that the type of the parameter 'pos' in 'gamma_correction' is 'ivec3'. When the function 'gamma_correction' is later used in combination with a matrix of a different type (e.g. '`mat[uint8]`'), the compiler will create a second version of 'gamma_correction' where 'pos' will be of type 'ivec2'.

6.2 Parametrized reductions

Similar to functions, reductions can also be parametrized. This relieves the programmer from the extra work in replicating reductions for different data types. Parametrized reductions frequently occur in combination with parametrized functions.
Suppose that we have a highly efficient multiplication function that works on a matrix (with an arbitrary data type) and vectors (with the same element data type as the matrix). Then we want to define an operator * in order to map expressions `A*B` onto this generic function. This can be achieved as follows:

```
function y = matrix_multiplication(A : mat[T], B : vec[T])
    ...
endfunction

reduction (T, A : mat[T], B : vec[T]) -> A*B = matrix_multiplication(A,B)
```

I.e., the type parameter acts as nothing more than an extra input parameter for the reduction. Optionally, the reduction may include a where clause to impose additional constraints on `T`.

6.3 Parametrized types

In a type erasure approach, generic types can be obtained by not specifying the types of the members of a class:

```
type stack : mutable class
    tab
    pointer
endtype
```

However, this limits the type inference, because the compiler cannot make any assumptions w.r.t. the type of '`tab`' or '`pointer`'. When objects of the type '`stack`' are used within a for-loop, the automatic loop parallelizer will complain that insufficient information is available on the types of '`tab`' and '`po`(like type)`inter`'. This problem can be solved by using parametrized types:


```
type stack[T] : mutable class
    tab : vec[T]
    pointer : int
endtype
```

An object of the type '`stack`' can then be constructed as follows:

```
obj = stack[int] % or even:
obj = stack[stack[cscalar]]
```

Parametric classes are similar to template classes in C++.
It is also possible to define methods for parametric classes:

```
function [] = __device__ push[T](self : stack[T], item : T)
    cnt = (self.pointer += 1) % atomic add for thread safety
    self.tab[cnt - 1] = item
endfunction
```

Methods for parametric classes can be '`__device__`' functions as well, so that they can be used on both the CPU and the GPU. This allow us to create thread-safe and lock-free implementations of common data types, such as sets, lists, stacks, dictionaries etc. within Quasar.

6.4 Generic memory allocation functions and casting

When programming generic functions, it is often useful to allocate vectors or matrices of generic types. This can be accomplished using generic extensions of the memory allocation functions: `uninit[T]()`, `zeros[T]()`, `shared[T]()`, `shared_zeros[T]()`, as in the following example:

```
function y:'unchecked = generic_multiply[T](a : mat[T]'unchecked, b : mat[T]'unchecked) concealed
    assert(size(a,1)==size(b,0),"Matrix multiplication: dimensions do not match:",size(a,0..1),"x",
        size(b,0..1),"!")
    y = mat[T](size(a,0),size(b,1))
    for m = 0..size(y,0)-1
        for n = 0..size(y,1)-1
            result = cast(0.0, T)
            for k = 0..size(a,1)-1
                result += a[m,k] * b[k,n]
            endfor
            y[m,n] = result
        endfor
    endfor
endfor
```

Here, the type `T` is only known when either the function is specialized (i.e., at compile-time) or at runtime.
Additionally, often it is required to initialize scalar variables based on the generic type. This is achieved using a type cast:

```
result = cast(0.0, T)
```

Note that type casts should be avoided as much as possible and should only be used when dealing with generic functions.

6.5 Explicit specialization through meta-functions

Normally, generic functions are automatically specialized (which is called *implicit specialization*). This is a compiler-decision that relies on a number of heuristics. However, there is also the possibility of explicitly indicating that a given function need to be specialized and also in which way. This can be achieved using the meta-function `$specialize`:

```
$specialize(function_name, constraint1 && ... && constraintN)
```

In Quasar, there are three levels of genericity (for which specialization can be done):

1. *Type constraints*: a type constraint binds the type of an input argument of the function.
2. *Value constraints*: gives an explicit value to the value of an input argument
3. *Logic predicates*: additional assumptions on the input arguments (see 5) that are not type or value constraints

Example 1 As an example, consider the following generic function:

```
function y = __device__ soft_thresholding(x, T)
    if abs(x)>=T
        y = (abs(x) - T) * (x / abs(x))
    else
        y = 0
    endif
endfunction
reduction x : scalar -> abs(x) = x where x >= 0
```

Now, we can make a specialization of this function to a specific type:

```
soft_thresholding_real = $specialize(soft_thresholding, type(x,"scalar") && type(T, "scalar"))
```

But also for a fixed threshold:

```
soft_thresholding_T = $specialize(soft_thresholding,T==10)
```

We can even go one step further and specify that ‘x>0‘:

```
soft_thresholding_P = $specialize(soft_thresholding,x>0)
```

Everything combined, we get:

```
soft_thresholding_E = $specialize(soft_thresholding, type(x,"scalar") && type(T,"scalar") && T==10 &&
    x>0)
```

Based on this knowledge (and the above reduction), the compiler will then generate the following function:

```
function y = __device__ soft_thresholding_E(x : scalar, T : scalar)
    if x >= 10
        y = x - 10
    else
        y = 0
    endif
endfunction
```


It can be noted that the function has now significantly been simplified.

Example 2 Explicit specialization can also be used to change the types of function parameters at compile-time. This is legal as long as the parameter types are always *narrowed* by this operation. This is useful for example to address the GPU hardware texturing units (see section 9.3) in a more general way. Below, the implemenuninittation of a 1D spatial filter with variable directionality is given.

```
function [] = __kernel__ filter_kernel(x : mat, y : mat'unchecked, a : vec, dir : ivec2, pos : ivec2)
    offset = int(numel(a)/2)
    total = 0.
    for k=0..numel(a)-1
        total += x[pos + (k - offset).* dir] * a[k]
    endfor
    y[pos] = total
endfunction

% Default implementation - simply pass all parameters to the kernel function
parallel_do(size(im),im,im_out,a,[0,1],filter_kernel)

% Implementation II - use the GPU hardware hardware texturing units (HTUs)
parallel_do(size(im),im,im_out,a,[0,1],$specialize(filter_kernel,type(x,mat'hwtex_nearest)))

% Implementation III - perform constant substitution + use the HTUs
parallel_do(size(im),im,im_out,$specialize(filter_kernel,type(x,mat'hwtex_nearest) && a==[1,2,3,2,1]/9
     && dir==[0,1]))
```

On an NVidia Geforce 435M GPU, the third implementation is about two times faster than the first implementation and 10% faster than the second implementation.

6.6 Implicit specialization

When generic device functions are called from kernel/device functions, the generic device function will be implicitly specialized (i.e., specialized automatically by the compiler).

```
function y = __device__ soft_thresholding(x, T)
    if abs(x)>=T
        y = (abs(x) - T) * (x / abs(x))
    else
        y = 0
    endif
endfunction

function [] = __kernel__ denoising(x : mat, y : mat)
    assert(x[pos]>0)
    y[pos] = soft_thresholding(x[pos], 10)
endfunction
```

For execution on a computation device, all types must be statically determined at compile-time. Therefore, the implicit specialization allows defining generic functions without bothering about the type to which the function will be applied. Note that during the implicit specialization phase, the compiler may generate an error when a type mismatch is encountered.

6.7 Generic size-parametrized arrays

In some cases, it is useful to integrate knowledge on the size of arrays into their types. This can be achieved using generic size-parametrized arrays. Consider the following example of a color transform on an image with `N` channels (where `N` is a generic parameter):

```
function img_out = colortransform[N](C : mat(N,N), img : cube(:,:,N))
    img_out = zeros(size(img,0),size(img,1),N)
    {!parallel for}
    for m=0..size(img,0)-1
        for n=0..size(img,1)-1
            img_out[m,n,:] = (C * squeeze(img[m,n,:]))
        endfor
    endfor
endfunction
```

The parameter `N` constraints the arguments `C` and `img` to match in dimensions in the following way: the third dimension of img should match the first and second dimensions of `C` (if not, a compiler error or (in some rare cases) a runtime error will be generated). Due to the implicit specialization (section 6.6), calling the function with a `mat3x3` color transform matrix will ensure that the parameter `N=3` is substituted during compile-time, leading to efficient calculation of the vector-matrix product

```
C * squeeze(img[m,n,:])
```

. The `squeeze` function is required here, because `img[m,n,:]` returns a vector of dimensions `1x1xN` which is multiplied with matrix `C`. The `squeeze` function removes the singleton dimensions and converts the vector to `Nx1x1` (or simply `N`, since singleton dimensions at the end do not matter in Quasar). For the assignment, this conversion is done automatically. In practice the `squeeze` function will have zero-overhead; it is only required here to keep the compiler happy.
Because with this technique, the dimensions of `img_out[m,n,:]` are known at compile-time, the implementation of the above function will not use dynamic kernel memory (section §8.3), which is a big performance benefit.
The type parametrization allows for linear arithmetic. Therefore, it is possible to declare types like `vec(2*N)` and `mat(2+N,2*N)`. For example, defining a matrix of type `mat(2,N)`:

```
A = mat(2,N)
B = cube(N,2*N,3)
Y = A[0,:]
C = B[:,:,1]
```

The compiler can now infer that `Y` has type `vec(N)` and that `C` has type `mat(N,2*N)`. After specialization with `N=2`, `C` will have type `mat(2,4)` and the resulting variables enjoy the benefits of fixed sized data types.

6.8 Generic dimension-parametrized arrays

Often, it is useful to define functions that can handle arrays with arbitrary dimensions. For this purpose, the type `cube{:}` can be used (see 2.2.5). This type specifies a cube with an arbitrary (or to speak, infinite) dimension. Because currently Quasar does not support multidimensional for-loops directly, the easiest way is to linearize the index and use two conversion functions:

- `ind2pos(sz, index)`: converts a linear (scalar) index to an n-dimensional position vector

- `pos2ind(sz, pos)`: converts an n-dimensional position vector to a linear (scalar) index

The following example illustrates a subsampling operation on an n-dimensional array:

```
function y:'unchecked = subsample(x : cube{:}, D : int)
    y = uninit(int(size(x)/D))
    {!parallel for}
    for v = 0..numel(y)-1
        p = ind2pos(size(y), v)
        y[p] = x[p*D]
    endfor
endfunction
```

Then, it might be desirable to know the dimension of the array at the moment that the function is executed. The dimension of an array can be obtained using the `ndims()` function, however, in case the function has two arrays as input, it may be desired to express that both arrays have the same dimensionality. For this purpose, dimension-parametrized array types is available.
An example of such a function is the Kronecker product, for which the implementation is different for each value of the dimensionality. Nevertheless, the function can be implemented in a generic fashion, as follows:

```
function z = kron[N](x : cube{N}, y : cube{N})
    function [] = __kernel__ kernel(x : vec, y : vec, z : vec,
        x_dim : ivec(N), y_dim : ivec(N), pos : int, blkdim : int, blkcnt : int)
        stepsize = blkdim * blkcnt
        total = prod(x_dim)
        rep = prod(y_dim)
        z_dim = int(x_dim .* y_dim)

        for i=pos..stepsize..total-1
            x_pos = ind2pos(x_dim, i)
            val = x[i]
            for j=0..rep-1
                y_pos = ind2pos(int(y_dim), j)
                z_pos = x_pos + x_dim .* y_pos
                z[pos2ind(z_dim, z_pos)] = val * y[pos2ind(y_dim, y_pos)]
            endfor
        endfor
    endfunction

    z = uninit(size(x,0..N-1).*size(y,0..N-1))
    nextpow2 = n -> int(2^ceil(log2(n)))
    parallel_do(min(nextpow2(numel(x)), 16384), x, y, z, size(x,0..N-1), size(y,0..N-1), kernel)
endfunction
```

When called, the compiler will specialize the generic function `kron` for each value of `N` that occurs in the calling code. When the dimensionalities of `x` and `y` do not match, the maximum of their dimensionalities will be used. Note that the above implementation combines size-parametrized and dimension-parametrized arrays in a natural way: the length of the dimension vectors `x_dim` and `y_dim` matches the dimensionality of the input data.

6.9 Example of generic programming: linear filtering

A linear filter computes a weighted average of a local neighborhood of pixel intensities, and the weights are determined by the so-called filter mask.
In essence, 2D linear filtering formula can be implemented in Quasar using a 6 line `__kernel__` function:

```
function [] = __kernel__ filter(x : cube, y : cube, mask : mat, ctr : ivec3, pos : ivec3)
    sum = 0.0
    for m=0..size(mask,0)-1
        for n=0..size(mask,1)-1
            sum += x[pos+[m,n,0]-ctr] * mask[m,n]
        endfor
    endfor
    y[pos] = sum
endfunction
```

However, this may not be the *fastest* implementation, for two reasons:

- The above kernel function performs several read accesses to `x` (e.g. for 3x3 masks it requires 9 read accesses per pixel!). As outlined in the Quick optimization guide, the implementation should use shared memory as much as possible.
- In case the filter kernel is separable (i.e. `mask = transpose(mask_y) * mask_x`), a faster implementation can be obtained by performing the filtering in two passes: a horizontal pass and a vertical pass. However, a naive implementation of this approach may have a bad data locality and depending on the size of the filter mask, it may even do more worse than good.

The best approach is therefore to combine the above techniques (i.e. shared memory + separable filtering). For illustrational purposes, we will consider only the mean filter (with `mask=ones(3,3)/9`) in the following.

```
function [] = __kernel__ filter3x3kernelseparable(
    x:cube,y:cube,pos:ivec3, blkpos:ivec3,blkdim:ivec3)
    vals = shared(blkdim+[2,0,0])
    sum = 0.
    for i=pos[1]-1..pos[1]+1
        sum += x[pos[0],i,blkpos[2]]
    endfor
    vals[blkpos] = sum
    if blkpos[0]<2
        sum = 0.
        for i=pos[1]-1..pos[1]+1
            sum += x[pos[0]+blkdim[0],i,blkpos[2]]
        endfor
        vals[blkpos+[blkdim[0],0,0]] = sum
    endif
    syncthreads
    sum = 0.
    for i=blkpos[0]..blkpos[0]+2
        sum += vals[i,blkpos[1],blkpos[2]]
    endfor
    y[pos] = sum*(1.0/9)
endfunction
x = imread("image.png")
```



```
y = zeros(size(x))
parallel_do(size(y),x,y,filter3x3kernelseparable)
imshow(y)
```

Remark that the above implementation is rather complicated, especially the block boundary handling code is excessive. Through generic programming, it is possible to extend this code fragment, in order to be used in a wider context. Quasar has two programming techniques:

1. **Function variables and closure variables**

   Suppose that we express a filtering operation in a general way:

   ```
   type f : [__device__ (cube, ivec2) -> vec3]
   ```

   This is a type declaration of a function that takes a cube and a 2D position as input, and computes a 3D color value.

   Then, a linear filter can be constructed simply as follows:

   ```
   mask = ones(3,3)/9
   ctr = [1,1]
   function y : vec3 = __device__ linearfilter(x : cube, pos : ivec2)
       y = [0.0,0.0,0.0]
       for m=0..size(mask,0)-1
           for n=0..size(mask,1)-1
               y += x[pos+[m,n,0]-ctr] * mask[m,n]
           endfor
       endfor
   endfunction
   ```

   Note that the body of this function is essentially the body of the kernel function at the top of this page.

   Next, we can define a kernel function that performs filtering for *any* filtering operation of type `f`:

   ```
   function [] = __kernel__ genericfilterkernelnonseparable(
       x:cube,y:cube, masksz:op:f,ivec2,pos:ivec3, blkpos:ivec3,blkdim:ivec3)
       vals = shared(blkdim+[masksz[0]-1,masksz[1]-1,0])
       vals[blkpos] = x[pos-[1,1,0]]
       if blkpos[0]<masksz[0]-1
           vals[blkpos+[blkdim[0]-1,-1,0]] = x[pos+[blkdim[0]-1,-1,0]]
       endif
       if blkpos[1]<masksz[0]-1
           vals[blkpos+[blkdim[1]-1,-1,0]] = x[pos+[blkdim[1]-1,-1,0]]
       endif
       syncthreads
       y[pos] = op(vals, blkpos)
   endfunction
   x = imread("image.png")
   y = zeros(size(x))
   parallel_do(size(y),x,y,size(mask,0..1),linearfilter,genericfilterkernelnonseparable)
   imshow(y)
   ```


Here, `masksz = size(mask,0..1)` (the size of the filter mask). Now we have written a generic kernel function, that can take any filtering operation and compute the result in an efficient way. For example, the filtering operation can also be used for mathematical morphology or for computing local maxima:

```
function y : vec3 = __device__ maxfilter(x : cube, pos : ivec2)
    y = [0.0,0.0,0.0]
    for m=0..size(mask,0)-1
        for n=0..size(mask,1)-1
            y = max(y, x[pos+[m,n,0]-ctr])
        endfor
    endfor
endfunction
```

The magic here, is to implicit use of closure variables: the function `linear_filter` and `max_filter` hold references to non-local variables (i.e. variables that are declared outside this function). Here these variables are `mask` and `ctr`. This way, the function signature is still `[__device__ (cube, ivec2) -> vec3]`.

2. **Explicit/implicit specialization**

   Previous point (1) is demonstrates a simple generic programming approach through function pointers. Some people believe that generic programming leads to a loss in efficiency. One of their arguments is that by the dynamic function call `y[pos] = op(vals, blkpos)`, where `op` is actually a function pointer, efficiency is lost: the compiler is for example not able to inline `op` and has to emit very general code to deal with this case.

   In Quasar, this is not necessarily true - being a true domain-specific language, the compiler has a lot of information. In fact, the optimization of the generic function `generic_filter_kernel_nonseparable` can be made explicit, using the `$specialize` meta function:

```
linearfilterkernel = $specialize(genericfilterkernelnonseparable, op==maxfilter)
x = imread("image.png")
y = zeros(size(x))
paralleldo(size(y),x,y,size(mask,0..1),linearfilterkernel)
imshow(y)
```

   The function `$specialize` is evaluated at compile-time and will substitute `op` with respectively `linear_filter` and `max_filter`. Correspondingly these two functions can be inlined and the resulting code is equivalent to the `linear_filter_kernel` function being completely written by hand. Now, in Quasar, function pointers are *avoided* by default (through the compilation setting "enable function pointers in generated code"). This is achieved exactly using this technique.

3. **Datatype-independent implementation**

   We can also go one step further and generalize the data types of the above kernel function:

```
mask = ones(3,3)/9
ctr = [1,1]
function y : vec3 = __device__ linearfilter[T](x : cube[T], pos : ivec2)
    y = [0.0,0.0,0.0]
    for m=0..size(mask,0)-1
        for n=0..size(mask,1)-1
            y += x[pos+[m,n,0]-ctr] * mask[m,n]
```



```
        endfor
    endfor
endfor
function [] = __kernel__ genericfilterkernelnonseparable[T](
    x:cube[T],y:cube[T], masksz,op:[__device__ (cube[T], ivec2) -> vec3],ivec2,pos:ivec3, blkpos:
        ivec3,blkdim:ivec3)
    vals = shared[T](blkdim+[masksz[0]-1,masksz[1]-1,0])
    vals[blkpos] = x[pos-[1,1,0]]
    if blkpos[0]<masksz[0]-1
        vals[blkpos+[blkdim[0]-1,-1,0]] = x[pos+[blkdim[0]-1,-1,0]]
    endif
    if blkpos[1]<masksz[0]-1
        vals[blkpos+[blkdim[1]-1,-1,0]] = x[pos+[blkdim[1]-1,-1,0]]
    endif
    syncthreads
    y[pos] = op(vals, blkpos)
endfunction
x = imread("image.png")
y = zeros(size(x))
parallel_do(size(y),x,y,size(mask,0..1),linearfilter,genericfilterkernelnonseparable)
imshow(y)
```

Here, the compiler will specialize the function `genericfilterkernelnonseparable`, as follows:

```
$specialize(genericfilterkernelnonseparable,op==linearfilter,T==scalar)
```

Functions with closure variables are building blocks for larger algorithms. Functions can have arguments that are functions themselves. Function specialization is a compiler operation that can be used to generate explicit code for fixed argument values. In the future, function specialization may be done automatically in some circumstances.

Chapter 7

Object-oriented programming

In Quasar, there are three types of classes:

- `class`: for creating constant objects with a fixed layout that can be marshalled to the target device (e.g., GPU)
- `mutable class`: for non-constant objects with a fixed layout that can be marshalled to the target device (e.g., GPU)
- `dynamic class`: Python-like classes, for which members can be added to the object at run-time

The distinction between `class` and `mutable class` enables the compiler and run-time to make stronger assumptions on the constantness of the corresponding objects, potentially resulting in a more efficient execution.
Furthermore, classes of the type `class` and `mutable class` can be used from host, device and kernel functions. Dynamic classes can *only* be used from host functions.
Another difference between `class` and `dynamic class` is in the null values. For `dynamic class`, a null reference `null` is used. For `class` and `mutable class`, a null pointer `nullptr` needs to be used.

7.1 Mutable/non-mutable classes

Mutable/non-mutable classes require all members to be statically typed. Dynamically typed members are not supported, because they can not be mapped onto static types on the computation device.
However, it is possible to define parametric types, in which the dynamically typed members are replaced by a parameter type (see further). Then the parametric type needs to be instantiated (either directly, or via function specialization), to be used on the computation device.
A example of a mutable class, with a few member functions is given below:

```
type point : mutable class
    x : scalar
    y : scalar
endtype
```

Recursive types can also be defined, although, the recursive member needs to be a pointer type (^). For example, the definition of a linked list of points can be as follows:

```
type point : mutable class
    x : scalar
    y : scalar
    next : ^point
endtype
```

7.2 Constructors

A constructor can be added to the `point` class. The following constructor uses the default constructor `point(x:=xval, y:=yval)` to initialize all object members.

```
function y = point(px : scalar, py : scalar)
    y = point(x:=px, y:=py)
endfunction
```

Constructors can be overloaded. A constructor that is intended to be used from kernel/device functions should have the `__device__` modifier:

```
function y = __device__ point(px : scalar, py : scalar)
    y = point(x:=px, y:=py)
endfunction
```

7.3 Destructors

Due to the automatic memory management, there are no destructors. Destructors may be added in a future version of Quasar.

7.3.1 Methods

To define methods, Quasar uses a pattern similar to Google Go. A method is a function for which the first parameter is `self`, referring to the object on which the method is called. The `self` object can be passed by-value (without a pointer ^), or by reference (using the pointer ^).

```
function y = scale(self : point, b)
    y = point(x:=b*self.x,y:=b*self.y)
endfunction

% The method point.setLocation
function [] = setLocation(self : ^point, x, y)
    self.x = x
    self.y = y
endfunction

% The method point.translate
function [] = translate(self : ^point, dx, dy)
    self.x += dx
    self.y += dy
```



```
endfunction

% The method point.tostring
function y = tostring(self : point)
    y = sprintf("(%f,%f)",self.x,self.y)
endfunction
```

Methods can be called in the same way as in other object-oriented languages. For example:

```
p = point(1.0, 2.0) % constructor
print p.scale(2) % method
```

Finally, methods can be overloaded. A method that is intended to be used from kernel/device functions should have the `__device__` modifier.

7.3.2 Properties

Properties can be added to the class, using reductions. The following reductions define a getter and setter for the property `length`:

```
reduction (a : point) -> a.length = sqrt(a.x ^ 2 + a.y ^ 2)
reduction (a : ^point, b : scalar) ->
    (a.length = b) = (a = point(x:=b/a.length*a.x,y:=b/a.length*a.y))
```

7.3.3 Operators

Similarly, operators can be defined. For example, to calculate the difference between two points, one could define:

```
reduction (a : point, b : point) -> a - b = point(a.x-b.x,a.y-b.y)
```

7.4 Dynamic classes

Dynamic classes are very useful for scripting. Consider the following dynamic class definition:

```
type Bird : dynamic class
    name : string
    color : vec3
endtype
```

At run-time, it is possible to add fields or methods:

```
bird = Bird()
bird.position = [0, 0, 10]
bird.speed = [1, 1, 0]
bird.is_flying = false
bird.start_flying = () -> bird.is_flying = true
```

Dynamic classes are also enable easy interoperability with other languages (e.g., C#, Visual Basic). Dynamic classes are also frequently used by the UI library (`Quasar.UI.dll`).

Despite the fact that dynamic classes can have properties that are added at run-time, the compiler still performs type inference on them, resulting in efficient code.
One limitation is that dynamic classes cannot be used from within `__kernel__` or `__device__` functions. As a compensation, the dynamic classes are also a bit lighter (in terms of run-time overhead), because there is no multi-device (CPU/GPU/...) management overhead. It is known a priori that the dynamic objects will "exist" in the CPU memory.

7.5 Parametric types

A disadvantage of non-static types is that the compiler may not be able to determine the types of the members of the class.

```
type stack : mutable class
    tab
    pointer
endtype
```

In this case, the compiler cannot make any assumptions w.r.t. the type of `tab` or `pointer`. When objects of the type `stack` are used within a for-loop, the automatic loop parallelizer will complain that insufficient information is available on the types of `tab` and `pointer`.
Parametric types can be used to solve this issue:

```
type stack[T] : mutable class
    tab : vec[T]
    pointer : int
endtype
```

An object of the type `stack` can then be instantiated as follows:

```
obj = stack[int]()
obj = stack[stack[cscalar]]()
```

It is also possible to define methods for parametric classes:

```
function [] = __device__ push[T](self : stack[T], item : T)
    cnt = (self.pointer += 1) % atomic add for thread safety
    self.tab[cnt - 1] = item
endfunction
```

Methods for parametric classes can be `__device__` functions as well, so that they can be used on both the CPU and the GPU.
The internal implementation of parametric types and methods in Quasar (i.e. the runtime) uses a combination of erasure and reification.
Defining a constructor is based on the same pattern that we used to define methods. For the above stack class, we have:

```
function y = stack[T]()
    y = stack[T](tab:=vec[T](100), pointer:=0)
endfunction

% Constructor with int parameter
```



```
function y = stack[T](capacity : int)
    y = stack[T](tab:=vec[T](capacity), pointer:=0)
endfunction

% Constructor with vec[T] parameter
function y = stack[T](items : vec[T])
    y = stack[T](tab:=copy(items), pointer:=0)
endfunction
```

Note that the constructor itself creates an instance of the type, rather than that it is done automatically. Consequently, it is possible (although it should be avoided) to return a `nullptr` value as well.

```
function y : ^stack[T] = stack[T](capacity : int)
    if capacity > 1024
        y = nullptr % Capacity too large, no can do...
    else
        y = stack[T](tab:=vec[T](capacity), pointer:=0)
    endif
endfunction
```

Operators/properties on parametric classes can be defined using parametric reductions. In a parametric reduction, the type parameter itself is part of the parameter list of the reduction.

```
type point[T] : mutable class
    x : T
    y : T
endtype

reduction (T, a : point[T], b : point[T]) -> a - b = point[T](a.x-b.x,a.y-b.y)
```

Note: it is currently not possible to define constraints on the type parameters. This functionality may be added in a future version of Quasar.

7.6 Inheritance

Inherited classes can be defined as follows:

```
type bird : class
    name : string
    color : vec3
endtype

type duck : bird
    ...
endtype
```

Inheritance is allowed on all three class types (mutable, immutable and dynamic).
Note: multiple inheritance is currently not supported.
As an example, consider the following `point`, `line` and `circle` classes:

```
type geometry : mutable class
    color : scalar
endtype
```



```
type point : geometry
    x : scalar
    y : scalar
endtype

type line : geometry
    p1 : point
    p2 : point
    x1 : scalar
    y1 : scalar
    x2 : scalar
    y2 : scalar
endtype

type circle : point
    radius : scalar
endtype

function y = distance_from_origin(p : ^point)
    y = sqrt(p.x^2 + p.y^2)
endfunction

c = circle(color:=0, radius:=4, x:=12, y:=5)
g = geometry(color:=1)
p = point(x:=4, y:=3, color:=1)

print "point distance from origin: ", distance_from_origin(p) % result=5
print "circle center distance from origin: ", distance_from_origin(c) % result=13
```

7.7 Virtual functions, interfaces, abstract classes

Virtual functions, interfaces, abstract classes are currently not supported by Quasar. These concepts may be implemented in a future version.
As a simple alternative of an interface, function types can be used. This way, it is possible to 'emulate' interfaces in Quasar:

```
type my_interface : mutable class
    times2_function : [__device__ scalar -> scalar]
    sum_function : [__device__ vec -> scalar]
    do_something : [(^my_interface) -> ??]
endtype

obj = my_interface(
    times2_function := (__device__ (x : scalar) -> 2*x),
    sum_function := (__device__ (x : vec) -> sum(x)),
    do_something := (self : ^my_interface) -> print(self)
)

print obj.times2_function(2)
print obj.sum_function([1,2,3])
obj.do_something(obj)
```


In the same way, abstract classes and virtual functions can be emulated. An advantage is that this technique works across computation devices, with no additional compiler support.

Chapter 8

Special programming patterns

8.1 Matrix/vector expressions

Operations on large matrices are grouped and automatically converted into a kernel function (sometimes called *broadcasting*). For example:

```
x = randn(512,512,64)
y = 0.1 + (0.8 * 255 * sin(x/255)) + 10 * w
```

will automatically be translated to:

```
function [_out:cube]=opt__auto_optimize1(x:cube,w:cube)
    function [] = __kernel__ opt__auto_optimize1_kernel _
        (_out:cube'unchecked,x:cube'unchecked,w:cube'unchecked,pos:ivec3)
        _out[pos]=((0.1+(204*sin((x[pos]/255))))+(10*w[pos]))
    endfunction
    _out = uninit(size(x))
    parallel_do(size(x),_out,x,w,opt__auto_optimize1_kernel)
endfunction
reduction (x:cube, w:cube) -> (((0.1+(204*sin((x/255))))+(10*w))= _
    opt__auto_optimize1(x,w)

x = randn(512,512,64)
y = opt__auto_optimize1(x, w)
```

which is faster, because intermediate results are directly computed in local memory, without accessing the global memory (see section 2.4.3). Remark that this procedure depends on the success of the type inference. In some cases, it may be necessary to give a hint to the compiler about the types of certain variables, through `assert(type(var,"typename"))` (see section 2.2). Also, the expression optimizer generates a reduction to deal with expressions of the form `(((0.1+(204*sin((x/255))))+(10*w))`. When the same expression appears several times in the code, even in slightly modified version (e.g. `sin(sinc(x)/255)` instead of `sin(x/255)`), the generated `__kernel__` function will be re-used.
The expression optimization can be configured using the following pragma:


```
#pragma expression_optimizer (on|off)
```

8.2 Loop parallelization/serialization

In Quasar, loop parallelization consists of 1) the detection of parallelizable (or serializable) loops and 2) the automatic generation of kernel functions for these loops. The automatic loop parallelizer (ALP) attempts to parallelize for-loops, starting with the outside loops first. The ALP automatically recognizes one, two and three dimenional for-loops, and also maximizes the dimensionality of the loops subject to parallelization. The ALP extracts the loops and converts them to kernel functions (see section 2.4.1), which are subsequently optimized using target-specific code transformations (see 17). Howver, not every for-loop can be parallelized and executed on a GPU: a number of restrictions exist:

1. All variables that are being used inside the loop must have a *static type* (i.e. explicitly typed as `vec`, `mat`, `cube`, see section 2.2) or a static type can be inferred from the context (type inference or through explicit/implicit specialization, see section §6). Practically speaking: only types that can be used inside `__kernel__` or `__device__` functions are allowed.

2. For the best performance, for slicing operations `A[a..b,2]`, the dimensions `a` and `b` must be constant and known at compile time (either specified explicitly, or obtained through constant propagation). When these constants are not known, a nested kernel function will be generated. Depending on the context, this may or may not involve dynamic kernel memory (see section §8.3)

3. The for-loop nest must be *perfect* (no code in between subsequent for statements, no dependencies between the loop boundaries, no `break`, no `continue`) for example:

   ```
   for m=0..size(x,0)-1
          for n=5..size(y,0)-1
       endfor
   endfor
   ```

   This is an example of a imperfect loop:

   ```
   for m=0..size(x,0)-1
       for n=m..size(y,0)-m
       endfor
   endfor
   ```

   (Note: certain types of imperfect loops can also be handled using the imperfect loop function transform, this requires placing

   ```
   {!function_transform enable="imperfectloop"}
   ```

   inside the for-loop)

4. Only types that can be used inside **kernel** or **device** functions are allowed (e.g., types based on `class` and `mutable class`, but not `dynamic class`).

5. When host (i.e. non kernel/device) functions are called from a loop, the loop is not eligible for parallelization/serialization. In particular, only a limited number of built-in functions are supported. Functions that interact with/take variables with unspecified types (such as `print`, `load`, `save`, ...) are not supported.

6. Data dependencies/conflicts between different iterations are detected and not allowed. In case a dependency is detected, the loop can be serialized. In this case, the for-loop will be natively compiled (in C++) and executed on the CPU in single-threaded mode.

7. Advanced kernel function features such as shared memory and thread synchronization (see section 2.4.4) are not supported, because these functions often require low-level access to the block position (`blkpos`) and dimensions (`blkdim`).

In case one of these conditions are violated, an error message is generated, and the code is not parallelized. In default form, this leads coded to be interpreted by the Quasar interpreter (often resulting in a significantly slower execution). It is therefore recommended to resolve issues raised by the automatic for-loop parallelizer.
The ALP can be configured using the pragmas (see section 17.3):

```
#pragma loop_parallelizer (on|off)
```

and the global configuration setting

```
COMPILER_AUTO_FORLOOP_PARALLELIZATION
```

(see table 17.3).
Individual for-loop parallelization can also be controlled by placing a code attribute (see section §4.11) in front of the for-loop definition. The following code attributes are available:

- `{!parallel for}`: forces the loop to be parallelized, despite dependencies (i.e., potential data races) detected between the variables. In the latter case, a warning message is shown.
- `{!parallel for; dim=n}`: forces the following `n` loops to be parallelized jointly. Note that this operation requires that the for-loop openings are placed directly after each other, without any intermediate statements (apart from comments). Explicitly specifying the dimension allows the outer n loops to be parallelized even in cases that there are some additional inner loops. When the `dims` parameter is omitted, the compiler attempt to select a maximal value for `dims` depending on the inner for-loops.
- `{!parallel for; multi_device=true}`: causes the for-loop to be parallelized over multiple devices (at least, for runtime engines that support multiple devices, see section §11).
- `{!serial for}`: serializes the for-loop, for execution on devices that support serial execution (e.g., the CPU).
- `{!interpreted for}`: forces the loop to be interpreted. This comes at a computational cost, but can still be useful for e.g., debugging purposes

An example of ALP code attributes is given below:

```
im = imread("image.tif","rgb")
{!parallel for}
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
        im[m,n,0..2] = 255-im[m,n,0..2]
    endfor
endfor
```


Most of the time, `{!parallel for}` is not required because the compiler is able to detect the parallelism automatically. However, `{!parallel for}` *ensures* that the subsequent loop will be parallelized, generating a compiler error when this fails.

8.2.1 While-loop serialization

The ALP is also able to serialize while loops, as in the following example:

```
{!serial for}
while !finished
        x = a[0,index]
        y = a[1,index]
        index += 1
        if x.^2 + y.^2 < 1
            finished = true
        endif
endwhile
```

8.2.2 Example: gamma correction

The following example illustrates a gamma correction operation:

```
im = imread("image.tif")
im_out = zeros(size(im))
gamma = 1.1
tic()
{!parallel for}
for i=0..size(im,0)-1
    for j=0..size(im,1)-1
        for k=0..size(im,2)-1
            im_out[i,j,k] = im[i,j,k]^gamma
        endfor
    endfor
endfor
toc()
fig1 = imshow(im)
fig2 = imshow(im_out)
fig1.connect(fig2)
```

When no code attribute (`{!parallel for}`, `{!serial for}`) is used, the compiler will analyze the code, inspect the variable dependencies and decide whether the loop can be parallelized or serialized. In fact, it is not necessary to specify these code attributes, however, when it is done, the compiler will generate an error in case the parallelization/serialization fails (e.g., due to some data dependency). Resolving these errors may allow the code to be parallelized.

Warning: when `{!parallel for}` is placed in front of a loop, the loop will be parallelized irrespective of the dependencies. Consequently, when using `{!parallel for}` uncarefully, the program behavior is affected and the wrong results may be obtained!

8.3 Dynamic kernel memory

Very often, it is desirable to construct (non-fixed length) vector or matrix expressions within a for-loop (or a kernel function). Before Jan. 2014, this resulted in a compilation error *"function cannot be used within the context of a kernel function"* or *"loop parallelization not possible because of function XX"*. The transparent handling of vector or matrix expressions with in kernel functions requires some special (and sophisticated) handling by the Quasar compiler and runtime. In particular: what is needed is dynamic kernel memory. This is memory that is allocated on the GPU (or CPU) during the operation of the kernel. The dynamic memory is disposed (freed) either when the kernel function terminates or at a later point.
There are a few use cases for dynamic kernel memory:

- When the algorithm requires to process several medium-sized (`16x16`) to large-sized (e.g. `128x128`) matrices. For example, a kernel function that performs matrix operations for every pixel in the image. The size of the matrices may or may not be known in advance.[1]
- Efficient handling of multivariate functions that are applied to (non-overlapping or overlapping) image *blocks*.
- When the algorithm works with dynamic data structures such as linked lists, trees, it is also often necessary to allocate "nodes" on the fly.
- To use some sort of "/scratch" memory that does not fit into the GPU shared memory (note: the GPU shared memory is 32K, but this needs to be shared between all threads - for 1024 threads this is 32 bytes private memory per thread). Dynamic memory does not have such a stringent limitation. Moreover, dynamic memory is not shared and disposed either 1) immediately when the memory is not needed anymore or 2) when a GPU/CPU thread exists. Correspondingly, when 1024 threads would use 32K each, this will require less than 32MB, because the threads are *logically* in parallel, but not *physically*.

In all these cases, dynamic memory can be used, simply by calling the `zeros`, `ones`, `eye` or `uninit` functions. One may also use slicing operators (`A[0..255, 2]`) in order to extract a sub-matrix. The slicing operations then take the current boundary access mode (e.g. mirroring, circular) into account. Dynamic kernel memory has the following advantages:

- Memory can be allocated and released on the fly (for example, scratch pads within a kernel that exceed the shared memory size)
- Functions can be written in a generic way so that they can handle any vector/matrix dimensions

However, there also some drawbacks:

- Despite the internal parallel memory allocator being quite efficient, code that uses dynamic kernel memory is generally 10x-25x slower than code that does not rely on dynamic kernel memory. This is due to 1) the execution cost of allocation and disposal operations (which are performed behind the scenes) and 2) the storage of the data in the global memory which is slower than, e.g., register memory.
- The restrictions of the dynamic memory subsystem may cause runtime errors to be generated, for example when insufficient dynamic kernel memory is available. To alleviate this problem, it is possible to increase the amount of dynamic kernel memory by a configuration setting (in Redshift: Program Settings/runtime/kernel dynamic memory reserve), although this memory is then not available for other purposes.

[1] Note that to avoid use of dynamic kernel memory in some cases, types can be annotated with size information (see section §3.3).

Because of these drawbacks, an optimization warning is generated by the compiler. It is recommended to use dynamic kernel memory sparely and only when there is no direct alternative. Different approaches exist to avoid that dynamic kernel memory is used:

- Use of array size constraints (see section §3.3). The resulting variables have the advantages that the memory is stored in the registers or on the stack (which is highly efficient).
- Use of size-parametrized generic types (see section §6.7).
- Memory preallocation: allocate one large matrix (using `zeros`, `uninit` functions) outside the kernel function and pass the matrix to the kernel function

8.3.1 Examples

The following program transposes `16x16` blocks of an image, creating a cool tiling effect. Firstly, a kernel function version is given and secondly a loop version. Both versions are equivalent: in fact, the second version is internally converted to the first version.

Kernel version

```
function [] = __kernel__ kernel (x : mat, y : mat, B : int, pos : ivec2)
    r1 = pos[0]*B..pos[0]*B+B-1 % creates a dynamically allocated vector
    r2 = pos[1]*B..pos[1]*B+B-1 % creates a dynamically allocated vector

    y[r1, r2] = transpose(x[r1, r2]) % matrix transpose
        % creates a dynamically allocated vector
endfunction

x = imread("lena_big.tif")[:,:,1]
y = zeros(size(x))
B = 16 % block size
parallel_do(size(x,0..1) / B,x,y,B,kernel)
```

Loop version

```
x = imread("lena_big.tif")[:,:,1]
y = zeros(size(x))
B = 16 % block size

{!parallel for}
for m = 0..B..size(x,0)-1
    for n = 0..B..size(x,1)-1
        A = x[m..m+B-1,n..n+B-1] % creates a dynamically allocated vector
        y[m..m+B-1,n..n+B-1] = transpose(A) % matrix transpose
    endfor
endfor
```

8.3.2 Memory models

To acommodate the widest range of algorithms, two memory models are currently provided (some more may be added in the future).

1. **Concurrent memory model** (default)

   In the concurrent memory model, the computation device (e.g. GPU) autonomously manages a separate memory heap that is reserved for dynamic objects. The size of the heap can be configured in Quasar and is typically 32MB.

   The concurrent memory model is extremely efficient when all threads (e.g. ≥ 512) request dynamic memory at the *same time*. The memory allocation is done by a specialized parallel allocation algorithm that significantly differs from traditional sequential allocators.

   For efficiency, there are some internal limitations on the size of the allocated blocks:

   - The total amount of kernel dynamic memory is by default 128 MB, but can be configured (see *Redshift/program settings/Runtime/Memory management*)
   - The maximum size is by default 64 KiB (65536 bytes), but can be configured (see *Redshift/program settings/Runtime/Memory management*).
   - The minimum size is by default 64 bytes and scales linearly with the setting of maximum size. When the allocation request size is smaller than this minimum size, the request size is rounded up to the minimum size.
   - All Quasar modules need to use the same kernel dynamic memory settings: this is necessary so that dynamic memory blocks can be accessed correctly (allocated, deallocated) in between different modules

   Also note that the maximum size also incorporates the management overhead (reference counting, block sizes etc.), so in case you intend to allocate a 128×128 matrix in 32-bit precision float mode (64K) it is required to select 128 KiB.

   The concurrent memory model is primarily designed to enable a large number of concurrent allocations of relatively small memory blocks (for example to perform a local matrix calculation). It is generally not memory-efficient to allocate large memory blocks from a kernel function in the concurrent memory model. This is because often thousands of threads may be launched on the GPU, and correspondingly, thousands of dynamically allocated memory blocks may be active for a certain local calculation. However, the temporary memory blocks may easily consume several megabytes of GPU memory, and this may restrict the amount of GPU memory available for other purposes. For example, consider 2800 concurrent threads using each 128KB of data. This corresponds to 377MB of GPU memory!

   Therefore, for large dynamic memory allocations, consider using the *cooperative memory model*.

2. **Cooperative memory model**

   In the cooperative memory model, the kernel function requests memory directly to the Quasar allocator. This way, there are no limitations on the size of the allocated memory. Also, the allocated memory is automatically garbage collected. To enable the cooperative memory model from a kernel function, use:

   ```
   {!kernel memorymodel="cooperative"}
   ```

   inside the kernel function.

   Because the GPU cannot launch callbacks to the CPU, this memory model requires the kernel function to be executed on the CPU.

   Advantages:

- The maximum block size and the total amount of allocated memory only depend on the available system resources.

Limitations:

- Can only be used from CPU kernel functions.
- The Quasar memory allocator uses locking (to limited extend), so simultaneous memory allocations on all processor cores may be expensive.

8.3.3 Features

- Device functions can also use dynamic memory. The functions may even return objects that are dynamically allocated.
- **The following built-in functions are supported and can be used from within kernel and device functions**:

```
zeros, czeros, ones, uninit, eye,
copy, reshape, repmat, shuffledims,
seq, linspace, real, imag, complex,
mathematical functions matrix/matrix
multiplication matrix/vector multiplication
```

Note: when the above functions are applied to arrays with size constraints (see section §3.3), no dynamic kernel memory will be used.

8.3.4 Performance considerations

Dynamic kernel memory can greatly improve the expressibility of Quasar programs, however there are also a number of downsides that need to be taken into account.

- *Global memory access*: code relying on dynamic memory may be slow (for linear filters on GPU: 4x-8x slower), not because of the allocation algorithms, but because of the global memory accesses. However, it all depends on what you want to do: for example, for non-overlapping block-based processing (e.g., blocks of a fixed size), the dynamic kernel memory is an excellent choice.
- *Static vs. dynamic allocation*: when the size of the matrices is known in advanced, static allocation (e.g. outside the kernel function may be used as well). The dynamic allocation approach relieves the programmer from writing code to pre-allocate memory and calculating the size as a function of the size of the data dimensions. The cost of calling the functions `uninit`, `zeros` is negligible to the global memory access times (one memory allocation is comparable to 4-8 memory accesses on average - 16-32 bytes is still small compared to the typical sizes of allocated memory blocks). Because dynamic memory is disposed whenever possible when a particular threads exists, the maximum amount of dynamic memory that is in use at any time is much smaller than the amount of memory required for pre-allocation.
- Use `vecX`, `matXxY`, `cubeXxYxZ`, ... types (with size constraint) whenever your algorithm allows it (see section §3.3). This completely avoids using global memory, by using the registers instead. Once a vector of length 17 is created, the vector is allocated as dynamic kernel memory.

- Avoid writing code that leads to thread divergence: in CUDA, instructions execute in warps of (typically) 32 threads. Within a warp, instructions of different threads are executed at the same time (called *implicit SIMD*). Control flow instructions (`if`, `match`, `repeat`, `while`) can negatively affect the performance by causing threads of the same warp to diverge; that is, to follow different execution paths. Then,the different execution paths must be serialized, because all of the threads of a warp share a program counter. Consequently, the total number of instructions executed for this warp is increased. When all the different execution paths have completed, the threads converge back to the same execution path.
- To obtain best performance in cases where the control flow depends on the position (i.e., `pos` or `blkpos`), the controlling condition should be written so as to minimize the number of divergent warps.

8.4 Map and Reduce pattern

A popular parallel programming model is the Map and reduce model. In this model, a Map() function is applied to the data and subsequently, the data is aggregated using a Reduce() function. In its most simple for, a sequential implementation would be as follows:

```
total = 0.0
for m=0..511
    for n=0..511
        total = total + map(im[m,n])
    endfor
endfor
```

If this loop would be parallelized directly, this would lead to data races, because the summation (`total = total + ...`) is not atomic. Luckily the Quasar compiler *atomizes* this summation operation behind the screens, leading to the following parallel loop:

```
total = 0.0
{!parallel for}
for m=0..511
    for n=0..511
        total += map(im[m,n])
    endfor
endfor
```

For an overview how the compiler atomizes operations, see table 8.1. Note that this programming pattern may occur with multiple result values: the result can be a scalar value, a vector or even a matrix:

```
A = zeros(2,2)
{!parallel for}
for i=0..255
    A[0,0] += x[i,0]*y[i,0]
    A[0,1] += x[i,0]*y[i,1]
    A[1,0] += x[i,1]*y[i,0]
    A[1,1] += x[i,1]*y[i,1]
endfor
```

Direct implementation of this loop in OpenCL and CUDA would give a **poor performance** on GPU devices, due to all adds being serialized in the hardware (all threads need to write to the same location in memory, so there

Table 8.1: Atomization table of operators

Non-atomized	Atomized	Description
`a = a + b`	`a += b`	Addition
`a = a - b`	`a -= b`	Subtraction
`a = a * b`	`a *= b`	Multiplication
`a = max(a, b)`	`a ^^= b`	Maximum
`a = min(a, b)`	`a __= b`	Minimum
`a = and(a, b)`	`a &= b`	Bitwise AND
`a = or(a, b)`	`a |= b`	Bitwise OR
`a = xor(a, b)`	`a ~= b`	Bitwise XOR

is a spin-lock that basically serializes all the memory write accesses). The performance is often much worse than performing all operations sequentially on a CPU!
The obvious solution is the use of shared memory, thread synchronization in combination with parallel reduction (see section 10.6). In general it is quite hard to write these kind of algorithms, taking all side-effects in consideration, such as register pressure, shared memory pressure. Therefore, the Quasar compiler now detects the above pattern and converts it into an efficient parallel reduction algorithm. There are some considerations:

- The following operators are supported: addition (`+=`), subtraction (`-=`), multiplication (`*=`), minimum (`__=`), maximum (`^^=`), bitwise AND (`&=`), bitwise OR (`|=`) and bitwise XOR (`~=`).
- The compiler allows maximal flexibility in defining the body of the loop: i.e., it is possible to use control structures (if, loop, while, ...). Also, other intermediate values can be calculated, hence it is not a prerequisite that the loop only consists of summation statements such as in the above example.
- The reduction may also perform a dimension reduction (for example, summing a matrix along the rows). Currently, the Quasar compiler supports dimension reductions for which the number of dimensions is reduced with 1 at the time (dimension reduction in multiple dimensions at the time may be supported in tfuture versions).
- There is an **upper limit** on the number of accumulators (due to the size limit of the shared memory). For 32-bit floating point, up to 32 accumulators and for 64-bit floating point, up to 32 accumulators are supported. When the upper limit is exceeded, the generated code will still work, but the block size will *silently* be reduced (see also section §9.9). This, together with the impact on the occupancy (due to high number of registers being used) might lead to a performance degradation.

8.5 Cumulative maps (prefix sum)

A related frequently occurring programming patterns are cumulative maps, for example:

```
total = 0.0
{!parallel for}
for m=0..511
    for n=0..511
        im[m,n] = im[m-1,n] + map(im[m,n])
    endfor
endfor
```

Here, we assume that the `'safe` access modifier is used (see section 2.4.1), so that `im[m-1,n]` for `m=0` is equal to 0. Some applications of the cumulative map patterns are cumulative sums (e.g., integral images), infinite impulse

response (IIR) filters, etc. The cumulative map pattern is closely related to the map and reduce pattern, with the difference that the intermediate calculated values also need to be stored in memory.
The parallel algorithm for efficiently calculating a cumulative map is the *prefix sum*. By static analysis, the Quasar compiler is able to recognize the above pattern and convert it into a parallel prefix sum algorithm. The conditions are mostly the same as in case of the parallel reduction pattern:

- The following operators are supported: addition (`+=`), subtraction (`-=`), multiplication (`*=`), minimum (`__=`), maximum (`^^=`), bitwise AND (`&=`), bitwise OR (`|=`) and bitwise XOR (`~=`).
- Again, the compiler allows maximal flexibility in defining the body of the loop: i.e., it is possible to use control structures (if, loop, while, ...).
- One restriction is that the dependency should be `im[m,n]` ← `im[m-k,n]` where `k` is always 1. Higher order cumulative patterns (e.g., as in higher-order IIR filters) are not supported yet. In this case, it is recommended to decompose the filter into a cascade of first order IIR filters. Each filter of the cascade can then by individually parallelized using the cumulative map pattern.

8.6 Meta functions

Note: this section gives more advanced info about how internal routines of the compiler can be accessed from user code. Normally these functions do not need to be used directly, however this information can still be useful for certain operations.
Quasar has a special set of built-in functions, that are aimed at manipulating expressions at compile-time (although in the future the implementation may also allow them to be used at run-time). The functions are special, because actually, they do not follow the regular evaluation order (i.e. they can be evaluated from the outside to the inside of the expression, depending on the context). To make the difference clear with the host functions, these functions start with prefix `$`.
For example, `x` is an expression, as well as `x+2` or `(x+y)*(3*x)^99`. A string can be converted (at runtime) to an expression using the function `eval`. This is useful for runtime processing of expressions for example entered by the user. However, the opposite is also possible:

```
print $str((x+y)*(3*x)^99) % Prints "(x+y)*(3*x)^99"
```

This is similar to the string-izer macro symbol in C:

```
#define str(x) #x
```

However, there are a lot of other things that can be done using meta functions. For example, an expression can be evaluated at compile-time using the function `$eval` (which differs from `eval`)

```
print $eval(log(pi/2)) % Prints 0.45158273311699, but the result is computed at compile-time.
```

The `$eval` function also works when there are constant variables being referred (i.e. variables whose values are known at compile-time). Although this seems quite trivial, this technique opens new doors for compile-time manipulation of expressions that are completely different from C/C++ but somewhat similar to Maple or LISP macros).
Below is a small overview of the meta functions in Quasar:

- `$eval(.)`: compile-time evaluation of expressions
- `$str(.)`: conversion of an expression to string

- `$subs(a=b,.)`: substitution of a variable by another variable or expression
- `$check(.)`: checks the satisfiability of a given condition (the result is either valid, satisfiable or unsatisfiable), based on the information that the compiler has at this point.
- `$assump(.)`: returns an expression with the assertions of a given variable
- `$simplify(.)`: simplifies boolean expressions (based on the information of the compiler, for example constant values etc.)
- `$args[in](.)`: returns an expression with the input arguments of a given function.
- `$args[out](.)`: returns an expression with the input arguments of a given function.
- `$nops(.)`: returns the number of operands in the expression
- `$op(.,n)`: returns the n-th operand of the expression
- `$ubound(.)`: calculates an upper bound for the given expression
- `$specialize(func,.)`: performs function specialization (see section §6.5)
- `$inline(lambda)(...)`: performs inlining of lambda expressions/functions
- `$ftype(x)` with `x="__host__"/"__device__"/"__kernel__"`: determines whether we are inside a host, device or kernel function.
- `$typerecon(x,y)` : reconstructs the type of the specified function with a given set of (specialized) input parameters. For example, given a function `f = x -> x`, `$typerecon(f,type(x,scalar))` will return `[scalar->scalar]`. This meta function is mainly used internally in the compiler.
- `$target(.)`: indicates whether we are compiling for the given target platform (see section 17.2.6).

Notes:

- Most of these functions (and in particular `$eval`, `$check`, `$specialize`, `$typerecon` and `$inline`) are only provided for testing and should not be used from user-code.
- The function `$ftype` is useful in combination with reductions with where clause (section 4.9.4), to express that the reduction may only be applied in a device/kernel or host function (also see [functions](Functions-in-Quasar)). For example:

```
reduction x -> log(x) = x - 1 where abs(x - 1) < 1e-1 && $ftype("__device__")
```

means that the reduction for `log(x)` may only be applied *inside* `__device__` functions, when the condition `abs(x - 1) < 1e-1` is met. Here, this is simply a linear approximation of the logarithm around `x==1`.

Example: copying the type and assumptions from one variable to another

It is possible to write statements such as "assume the same about variable 'a' as what is assumed on 'b'". This includes the type of the variable (as in Quasar, the type specification is nothing more than a predicate).

```
a : int
assert(0 <= a && a < 1)
b : ??
```



```
assert($subs(a=b,$assump(a)))
print $str($assump(b)) % Prints "type(b,"int")) && 0 <= b && b < 1"
```


Chapter 9

GPU hardware features

The GPU was originally designed for computer graphics and there are a lot of other facilities available to speed up GPU applications. In this Section, we describe a number of advanced GPU techniques from which Quasar programs can also potentially benefit. In this section, we describe several GPU specific optimization techniques that can easily be used from Quasar programs.

9.1 Constant memory and texture memory

The GPU hardware provides several caches or memory types that are designed for dealing with (partially) constant data:

- **Constant memory**: NVIDIA GPUs provide 64KB of constant memory that is treated differently from standard global memory. In some situations, using constant memory instead of global memory may reduce the memory bandwidth (which is beneficial for kernels). Constant memory is also most effective when all threads access the same value at the same time (i.e. the array index is not a function of the position).
- **Texture memory**: texture memory is yet another type of read-only memory. Like constant memory, texture memory is cached on chip, so it may provide higher effective bandwidth than obtained when accessing the off-chip DRAM. In particular, texture caches are designed for memory access patterns exhibiting a great deal of spatial locality.

For practical purposes, the size of the constant memory is rather small, so it is mostly useful for storing filter/weight coefficients that do not change while the kernel is executed. On the other hand, the texture memory is quite large, has its own cache, and can be used for storing constant input signals/images.
In Quasar, constant/texture memory can be utilized by adding modifiers to the kernel function parameter types. The following modifiers are available (see table 9.1):

- `'hwconst`: the vector/matrix needs to be stored in the *constant memory*. Note: if there is not enough constant memory available, a run-time error is generated!
- `'hwtex_nearest` or `'hwtex_linear`: the vector/matrix/cube needs to be stored in the *texture memory* (see further, in section 9.3). Up to 3-dimensional data structures (cubes) are supported.

Table 9.1: Overview of access modifiers controlling the GPU hardware texturing unit

Modifier	Purpose
'hwconst	The data needs to be stored in constant memory (max 64KB). Implies 'const
'hwtex_nearest	Store the vector/matrix/cube in the GPU texture memory, use nearest neighbor lookup
'hwtex_linear	Store the vector/matrix/cube in the GPU texture memory, use linear interpolation
'hwtex_const	Does not store the data in the GPU texture memory, but instead uses the non-coherent L2-cache of the GPU when accessing the data

- `'hwtex_const` - non-coherent texture cache. This option requires CUDA compute architecture 3.5 or higher - as in Geforce GPUs of the 900 series, and allows the data still be stored in the global memory, will utilizing the texture cache for load operations. This combines the advantages of the texture memory cache with the flexibility (ability to read/write) of the global memory.

Note that because of the different access mechanism, these modifiers cannot be combined.
For Fermi and later devices, global memory accesses (i.e., without `'hw*` modifiers) are cached in the L2-cache of the GPU. For Kepler GPU devices, using `'hwtex_const` the *texture* cache is utilized directly, bypassing the L2 cache. The texture cache is a separate cache with a separate memory pipeline and relaxed memory coalescing rules, which may bring advantages to bandwidth-limited kernels.[1]
Starting with Maxwell GPU devices, the L1 cache and the texture caches are unified. The unified L1/texture cache coalesces the memory accesses, gathering up the data requested by the threads in a warp, before delivering the data to the warp.[2]
For using constant memory, we give the following guidelines:

- When your kernel function is using some constant vectors (weight vectors with relatively small length), and when all threads (or more specifically, all threads within one warp) access the same value of the vector at the same time (the index is not a function of the position!), you should definitely consider using `'hwconst`. In case different constant vector elements are accessed from different threads, the constant cache must be accessed multiple times, which degrades the performance.

- When your kernel function is accessing constant images (`vec`, `mat` or `cube`) on Kepler/Maxwell devices with compute architecture >= 3.5, it may be worthful to use `'hwtex_const`.

However, the best is to investigate whether the modifier improves the performance (e.g. using the Quasar profiler).

Example Consider the following convolution program:
Default version with no constant memory being used:

```
function [] = __kernel__ kernel(x : vec, y : vec, f : vec, pos : int)
    sum = 0.0
    for i=0..numel(f)-1
        sum += x[pos+i] * f[i]
    endfor
    y[pos] = sum
endfunction
```

Version with constant memory:

[1] For more information, see Kepler tuning guide.
[2] For more information, see Maxwell tuning guide.


```
function [] = __kernel__ kernel_hwconst(x : vec, y : vec, f : vec'hwconst, pos : int)
    sum = 0.0
    for i=0..numel(f)-1
        sum += x[pos+i] * f[i]
    endfor
    y[pos] = sum
endfunction
```

Version with constant texture memory for `f`:

```
function [] = __kernel__ kernel_hwtex_const(x : vec, y : vec, f : vec'hwtex_const, pos : int)
    sum = 0.0
    for i=0..numel(f)-1
        sum += x[pos+i] * f[i]
    endfor
    y[pos] = sum
endfunction
```

Version with constant texture memory for `x` and `f`:

```
function [] = __kernel__ kernel_hwtex_const2(x : vec'hwtex_const, y : vec, f : vec'hwtex_const, pos :
     int)
    sum = 0.0
    for i=0..numel(f)-1
        sum += x[pos+i] * f[i]
    endfor
    y[pos] = sum
endfunction
```

Version with HW textures (see section 9.3):

```
function [] = __kernel__ kernel_tex(x : vec, y : vec, f : vec'hwtex_nearest, pos : int)
    sum = 0.0
    for i=0..numel(f)-1
        sum += x[pos+i] * f[i]
    endfor
    y[pos] = sum
endfunction
```

For 100 runs on vectors of size 2048^2, with 32 filter coefficients, we obtain the following results for the NVidia Geforce 980 (Maxwell architecture):

```
Default: 513.0294 ms
f: 'hwconst: 132.0075 ms
f: 'hwtex_const: 128.0074 ms
x,f: 'hwtex_const: 95.005 ms
f: 'hwtex_nearest: 169.0096 ms
```

It can be seen that using constant memory (`'hwconst`) alone yields a speed-up of almost a factor 5 in this case. The best performance is obtained with `hwtex_const`. Moreover, using shared memory (see section §10.5), the performance can even further be improved to 85 ms.

Performance tip: use `'hwconst` when all threads in a kernel access the same memory location simultaneously (for example, filter coefficients in a linear filter). When to use `'hwtex_nearest` versus `'hwtex_const`, depends

on the way the data is reused (whether the matrix is readonly or needs to be copied often between the global memory and the texture memory): when the matrix is readonly and the access pattern is localized (e.g., a local window operation) or in combination with boundary access methods as `'safe`, `'mirror`, `'circular`, `'clamped`, it may be beneficial to use `'hwtex_nearest`. When the matrix access pattern is random, a better choice might be `'hwtex_const`. But as advice gives a worse performance in the above example, it is best to profile in order to be sure.

9.2 Shared memory designators

Shared memory designators provide a convenient approach to fetch blocks of data from global memory into shared memory of the GPU as well as a mechanism to write back the data.
As mentioned before in section 2.4.4, shared memory is best used whenever possible to relieve stress on the global memory and the most common use is in a memory mapping technique (similar to DMA): a portion of the data is mapped onto the shared memory. Local updates are later to be written back to the global memory. Different from a cache is that the layout of the shared memory is fully controlled: each array element is mapped onto a specified array element in the global memory. To facilitate the use of shared memory (in particular, the error-prone copying of data from and to shared memory), shared memory designators have been added to Quasar.
Shared memory designators thus provide a novel complementary technique to the `shared()` function to allocate and update shared memory. A shared memory designator is specified using the 'shared access modifier as follows:

```
function [] = __kernel__ kernel(A : mat, a : scalar, b : scalar, pos : ivec3)
    B : 'shared = transpose(a*A[0..9, 0..9]+b) % fetch

    % ... calculations using B (e.g., directly with the indices)

    A[0..9, 0..9] = transpose(B) % store
endfunction
```

The designator 'shared tells the compiler that this variable is intended to be stored in shared memory of the GPU. However, rather than one thread calculating eye(17), the compiler will generate code such that the calculations are *distributed* over the threads within the block. For this purpose, the compiler detects "thread-invariant" code related to the designated shared memory variables and modifies the code such that it is distributed over the threads, followed by the necessary thread synchronization.
For the above example, this will generate the following code:

```
function [] = __kernel__ kernel(A:mat,a:scalar,b:scalar,pos:ivec3,this_thread_block:thread_block)
    B=shared(10,10)
    for i=this_thread_block.thread_idx..this_thread_block.size..99
        [k01,k11]=ind2pos([10,10],i)
        B[k01,k11]=a*A[k11,k01]+b
    endfor
    this_thread_block.sync()
    for $i=this_thread_block.thread_idx..this_thread_block.size..99
        [k00,k10]=ind2pos([10,10],i)
        A[k00,k10]=B[k10,k00]
    endfor
endfunction
```


Note that the code using shared memory designators is significantly easier to understand, compared to the above code with low-level threading and synchronization primitives. It becomes then straightforward to speed up existing algorithms.
The shared memory designator technique relies on:

1. *vector/matrix size inference*: the compiler can determine that size(transpose(a∗A[0..9, 0..9]+b))==[10,10], so that the appropriate amount of shared memory can be allocated
2. *cooperative groups* (see 9.8): this_thread_block allows access to the low-level block functionality (size, position, thread index etc.)

The shared() and shared_zeros() functions exist as alternative, as a low-level interface to the shared memory. Summarizing, the differences are:

	shared()	: 'shared
Type	"Low" level	"High" level
Syntax	S=shared(sz)	S:'shared=uninit(sz)
Thread distribution	manual	automatic
Use in kernel function	Yes	Yes
Use in device function	No	No
Use in for-loop	No	Yes

See the matrix example below for an example of how to use the shared memory designators from a for-loop.

9.2.1 How to use

When a variable is declared with the shared designator : 'shared, the compiler will scan for several patterns related to the variable

1. Initialization uninit(): the standard way to initialize shared variables is

```
S : 'shared = uninit(M,N,K)
```

   The full type information of S (e.g., cube'shared) is omitted here since the compiler can obtained it via type inference. The above is the equivalent of S = shared(M,N,K), however S : 'shared = uninit(M,N,K) allows to compiler to manage the shared memory accesses.

   As is the case with shared() it is best that the parameters M,N,K are either constant (declared using : int'const, or a type parameter of a generic function), or that upper bounds on M∗N∗K are given via an assertion (e.g.,) assert(M∗N∗K<=512)). This way, the compiler can calculate the amount of shared memory that is required for the kernel function execution.

2. *Fetch and broadbast*:

   Instead of initializing with uninit() it is possible to initialize directly with an expression, for example:

```
S1 : 'shared = transpose(a*A[0..9, 0..5]+b)
S2 : 'shared = img[p[0]+(-c..16+c),p[1]+(-c..16+c),:]
S3 : 'shared = sum(reshape(img,[M,numel(img)/M]),1)
```


Instead of every thread calculating duplicate results (as would have been the case without using 'shared), the calculations are distributed over the threads within the thread block. In other words, the compiler will do the heavy work and generate code using the block parameters blkpos, blkdim (see before). After the operation, a thread synchronization (syncthreads) will implicitly be performed.

This initialization-by-expression can also be seen as a fetch and broadcast: first the memory is copied from global memory to shared memory (with possibly some intermediate calculations), next once in shared memory, the data is available to all threads (after syncthreads).

Through type inference, the compiler can determine the dimensions of S1, S2 and S3. For example, in the first case, the compiler will determine S1 : mat'shared(6,10).

3. *Gather and store*: process and copy back to global memory

   Using the same technique as with *fetch and broadcast*, the data stored in shared memory can be written back to the global memory:

```
A[0..9, 0..9] = transpose(S1)
B[0..sz[0]-1, (0..sz[1]-1)] = S2
```

Again, the calculations are distributed over the individual threads.

1. *Shared calculation*

   This pattern incurs a loop over the shared variable, as in the following example:

```
Sb : 'shared = uninit(M)
for L=0..M-1
    D = diagonal(cholesky(Sa[L,0..3,0..3]))
    Sb[L] = log_PP + 2*sum(log(D))
endfor
```

   Again, instead of every thread performing the entire loop, the loop will be distributed over the thread block. This allows for some calculation of temporary variables for which the results are shared over the entire thread block. The approach is similar to *fetch and broadcast* with the difference that the loop to initialize the shared variable is explicitly written out.

2. *Parallel reduction*

   This pattern is currently experimental, but the idea is to expand aggregation operations into a parallel reduction algorithm, as in the following example:

```
{!parallel for; blkdim=M}
for n=0..N-1
    B : 'shared = uninit(M)
    B[n] = ...
    syncthreads

    total = sum(B)
endfor
```

   Notice the calculation of total, via the sum function. Rather than every thread computing total independently, the calculation can again be distributed over the thread block. This technique provides a simple parallel

reduction primitive.

Currently, only the functions sum, prod, mean, min and max with one parameter are supported.

9.2.2 Virtual blocks and overriding the dependency analysis

To distinguish calculations that are position-dependent from calculations that can be shared within the thread block, the compiler performs a dependency analysis: it starts from the pos kernel function parameter and then determines the variables that are dependent of pos. Once the calculation of a variable depends on pos, the calculation can not be distributed any more over the thread block - the result of the calculation needs to be different for each thread.
In some cases, it is desirable to override the dependency analysis. For example, the *virtual block* technique defines tiles with size that is independent of the GPU architecture. The mapping onto GPU blocks is then implicitly handled by the compiler.

```
N : int'const = 16 % virtual block size
{!parallel for; blkdim=[N,N]}
for m=0..size(A,0)-1
    for n=0..size(A,1)-1
        mp = int(m/N)
        np = int(n/N)
        Ap = A[mp,Np]
    endfor
endfor
```

Here, the values mp and np are constant within each thread block, however by the specific way that the variables mp and np are calculated via modulo operation on the position indices m and n, the compiler cannot (yet) determine the constantness within the thread block. The following code attribute (to be placed inside the inner for loop or kernel function) indicates that, irrespective of the indexing, the variable Ap are constant over the thread block:

```
{!shared_mem_promotion assume_blk_constant={Ap}}
```

In the above example, the virtual block size and the GPU block size are fixed via {! parallel for ; blkdim=[N,N]}, but this does not necessarily have to be this way. It is up to the programmer to indicate that variables are constant within the thread block.
In the future, "virtual blocks" which size differs from the GPU blocks may be implemented using collaborative thread groups.

9.2.3 Examples

In this section, we discuss three example use cases of shared memory designators: image histograms, image separable filtering and parallel reduction.

9.2.3.1 Histogram

The calculation of a histogram is a good use case of the shared memory designators:

```
function [] = __kernel__ hist_kernel[Bins](im : vec, hist : vec(Bins), pos : int)
    hist_sh : 'shared = zeros(Bins)

    for m=pos..16384..numel(im)-1
        hist_sh[int(im[m])] += 1
    endfor
```



```
    hist[:] += hist_sh % add the result
endfunction
```

First, the histogram "scratchpad" is allocated in shared memory and initialized with zeros. Here the size of the histogram is a generic parameter, which guarantees that the compiler always has the exact size of the histogram. In the second step, the image is traversed using a so called "grid-strided loop". This is to ensure that the histogram can be updated several times by the same thread before the results are written to the histogram in global memory, thereby reducing the number of shared to global memory copies. As a final step, the local histogram is added to the global histogram.
A more simple way to implement hist_kernel would have been to not use shared memory at all,
as in the following kernel:

```
function [] = __kernel__ hist_kernel_global(im : vec, hist : vec, pos : int)
    hist[int(im[pos])] += 1
endfunction
```

A previous implementation technique in Quasar, via the sharedmemcaching kernel transform, is now deprecated in favor of designated shared memory, which yields not only more easily readable code but is also more flexible in its use.

```
function y : vec'unchecked = hist_sharedmemcaching(im)
    y = zeros(256)

    {!parallel for}
    for m=0..size(im,0)-1
        for n=0..size(im,1)-1
            for k=0..size(im,2)-1
                {!kernel_transform enable="sharedmemcaching"}
                {!kernel_arg name=y; type="inout"; access="shared"; op="+="; cache_slices=y[:]; numel
                    =256}
                v = im[m,n,k]
                y[v] += 1
            endfor
        endfor
    endfor
endfunction
```

On a Geforce GTX 980 GPU, the results are as follows:

Kernel function	Execution time for 100 runs on a 512x512 image
hist_kernel	**12.765 ms**
hist_kernel_global	73.7107 ms
hist_sharedmemcaching	28.5985 ms

The shared memory technique gives a speed-up of a factor 5.6x! It is even outperforms the shared mem caching transform by a factor 2x!.

9.2.3.2 Separable linear filtering

In this example, an implementation of separable linear filtering is given. Each block of the input image, including some border that depends on the filter length, is transferred to the shared memory.

```
function img_out = separable_linear_filtering[c : int](img_in : cube(:,:,3), fc : vec'hwconst(2*c+1))
    function [] = __kernel__ kernel(img_out : cube(:,:,3), pos : ivec2)

        p = pos - mod(pos, 16)
        s1 : 'shared = img_in[p[0]+(-c..16+c),p[1]+(-c..16+c),:]
        s2 : 'shared = uninit(16+2*c+1,16,3)

        % work shared along the threads
        for m=0..16+2*c
            for n=0..15
                total = zeros(3)
                for k=0..numel(fc)-1
                    total += fc[k] * s1[m,n+k,:]
                endfor

                s2[m,n,:] = total
            endfor
        endfor

        total1 = zeros(3)
        [m1,n1] = pos-p
        for k=0..numel(fc)-1
            total1 += fc[k] * s2[m1+k,n1,:]
        endfor
        img_out[pos[0],pos[1],:] = total1

    endfunction
    img_out = uninit(size(img_in))
    parallel_do([size(img_in,0..1),[16,16]],img_out, kernel)
endfunction
```

For the first, horizontal filtering stage, there are more output pixels to be computed than there are threads in each block (in particular, for filter length 2∗c+1 and block size 16x16, there are 16x(16+2∗c+1) output pixels). Using the shared memory designators, these calculations are also distributed over the thread block. For example, for c=4, this will yield 256+144=384, corresponding to 12.5 warps. The result of the first filtering stage is stored in shared memory.

The second, vertical filtering stage takes inputs from the previous stage stored in shared memory. The result is written back to the global memory.

Note that the function is parametrized on the half-filter length c. This way, the number of loop iterations for 0..16+2∗c, as well as numel(fc) can be computed at compile-time. The compiler also manages to calculate the amount of shared memory required to execute this kernel (6348 bytes for s1 and 4416 bytes for s2 in single precision floating point mode).

9.2.3.3 Parallel reduction (sum of NxN matrices)

Below is an example of a parallel reduction algorithm rewritten to take advantage of shared memory designators. Although the advantages of this particular implementation are limited compared to the low-level shared() function, the implementation is given for completeness.


```
function [y] = red(im : cube)

    M : int'const = 512 % block size
    N : int'const = 8*M % number of blocks times block size
    y = 0.0

    {!parallel for; blkdim=M} % Explicitly set the block size
    for n=0..N-1
        B : 'shared = uninit(M) % Shared entry for each thread

        % Calculate partial sum and store in B
        total = 0.0
        for k=n..N..numel(im)-1
            total += im[ind2pos(size(im),k)]
        endfor
        B[mod(k,M)] = total
        syncthreads

        % Sum over the shared memory - parallel reduction
        total = sum(B)

        if mod(n,M)==0
            y += total % Sum intermediate results
        endif
    endfor
endfunction
```

9.3 Speeding up spatial data access using Hardware Texturing Units

The hardware texturing units are a part of the graphics-accelerating heritage of the GPU. Originally, texture mapping was designed to enable realistically looking objects by letting the applications "paint" onto the geometry. From the rendered triangles, texture coordinates were interpolated along the X, Y and Z coordinates, such that for every output pixel, a texture value could be fetched (e.g. using nearest-neighbor or linear/trilinear interpolation). Later, programmable graphics and non-color like texture data (e.g. bump maps, shadow maps) were introduced and also the graphics hardware became more sophisticated. The hardware performance was improved by using dedicated hardware for transforming texture coordinates into hardware addresses, by adding texture caches and by using memory layouts optimized for spatial locality.
There is also hardware support for some of the type modifiers explained in section 2.4, in particular "`safe`", "`circular`", "`mirror`" and "`clamped`".
More generally, in Quasar, there are two main use cases for textures:

- The first is to use the texture for more optimized spatial data access: as an alternative for coalescing, to use the texture cache to reduce bandwidth requirements, ...
- The second is to make use of the fixed-function hardware that was originally intended for graphics applications:
  - The use of boundary conditions ("`safe`", "`circular`", "`mirror`" and "`clamped`")
  - The automatic conversion of integer values to floating point
  - The automatic conversion of 2D and 3D indices to addresses
  - Linear interpolation of 2D and 3D data.

Limitation	CUDA 2.x value
Maximum length for 1D texture	134217728
Maximum size for 2D texture	65536×65536
Maximum size for 3D texture	2048 × 2048 × 2048
Allowed element types	`scalar`, `int`, `int8`, `int16`, `int32` `uint8`, `uint16`, `uint32`
Access type	locally read-only, changes visible in next kernel function call
Access modifiers	`safe`, `circular`, `mirror` and `clamped` (no `checked`/`unchecked`)
Maximum number of textures/Quasar module	128 (or 256)

Table 9.4: Texture memory limitations

The hardware texture units can only be used in combination with texture memory. Texture memory is a read-only part of the global memory (see section 2.4.3), that is cached on-chip (e.g. 6-8 KB per multi-processor) and ordered using a space-filling curve optimized for spatial locality.
In table 9.4 there are a number of limitations listed for texture memory.
Using the hardware texture units in Quasar is quite simple: it suffices to add the following special modifiers to the types of arguments of kernel functions:

- `'hwtex_nearest`: use the hardware texturing unit in nearest interpolation mode for the specified argument
- `'hwtex_linear`: use the hardware texturing unit in linear interpolation mode for the specified argument

Note that these modifiers are only permitted to `vec`, `mat` or `cube` types. Complex-valued data or higher dimensional matrices are currently not yet supported.
The following image scaling example illustrates the use of hardware textures:

```
% Kernel function, not using hardware textures
function [] = __kernel__ interpolate_nonhwtex (y:mat, x:mat, scale:scalar, pos:ivec2)
    scaled_pos = scale * pos
    f = frac(scaled_pos)
    i = int(floor(scaled_pos))

    y[pos] = (1 - f[0]) * (1 - f[1]) * x[i[0], i[1]] +
                f[0] * (1 - f[1]) * x[i[0]+1, i[1]] +
            (1 - f[0]) * f[1] * x[i[0], i[1]+1] +
                f[0] * f[1] * x[i[0]+1, i[1]+1]
endfunction

% Kernel function, using hardware textures
function [] = __kernel__ interpolate_hwtex (y:mat, x:mat'hwtex_linear,
          scale:scalar, pos:ivec2)
    y[pos] = x[scale * pos]
endfunction
```

Note that the use of the hardware textures (and in particular the linear interpolation) is quite simple. However, it is important to stress that the `hwtex` modifiers can only be used for kernel function arguments. It is for example not possible to declare variables using these modifiers (if you try so, the modifiers will not have any effect).
The hardware textures enable some performance benefit. For example, on a Geforce 435M, for the above program the following results were obtained:


```
2D nearest neighbor interpolation without hardware texturing: 109.2002 msec
2D nearest neighbor interpolation with hardware texturing: 93.6002 msec
3D nearest neighbor interpolation without hardware texturing: 421.2007 msec
3D nearest neighbor interpolation with hardware texturing: 312.0006 msec

2D Linear interpolation without hardware texturing: 156.0003 msec
2D Linear interpolation with hardware texturing: 109.2002 msec
3D Linear interpolation without hardware texturing: 873.6015 msec
3D Linear interpolation with hardware texturing: 312.0006 msec
```

Especially, in 3D with linear interpolation, the performance is almost 3x higher than the regular approach. Textures have also a number of limitations:

- For non-floating point textures, the texture width should be a multiple of 32. Otherwise a run-time error will be generated. Note: for regular floating point textures there is no such limitation.
- The maximum size of the texture is limited (but increasing with newer GPU generations). The maximum size is typically 65536×65536 (2D) or $4096 \times 4096 \times 4096$ (3D).
- The element types are restricted.
- It is possible to write to texture memory from a kernel function (see section §9.6), but the effects are only visible in a next kernel function call.
- Textures cannot be used inside nested kernel functions (see section §4.4).
- The boundary condition `'checked` cannot be used in combination with hardware textures.

Summarizing, hardware textures have the following advantages:

1. Texture memory is *cached*, this is helpful when global memory is the main bottleneck.
2. Texture memory is efficient also for *less regular* access patterns
3. Supports *linear/bilinear* and *trilinear* interpolation in hardware
4. Supports boundary accessing modes (*mirror*, *circular*, *clamped* and *safe*) in hardware.

9.4 16-bit (half-precision) floating point textures

To reduce the bandwidth in computation heavy applications (e.g. real-time video processing), it is possible to specify that the GPU texturing unit should use 16-bit floating point formats. This can be configured on a global level in Redshift / Program Settings / Runtime / Use CUDA 16-bit floating point textures. Obviously, this will reduce the memory bandwidth by a factor of 2 in 32-bit float precision mode, and by a factor of 4 in 64-bit float precision mode. The option is also particularly useful when visualizing multiple large images.
Note that 16-bit floating point numbers have some limitations. The minimal positive non-zero value is 5.96046448e-08. The maximal value is 65504. Integers between -2047 and 2047 can be exactly represented. The machine precision (eps) value is 0.00097656. For these reasons, 16-bit floating point textures should not be used for accuracy sensitive parts of the algorithm. They are useful for rendering and visualization purposes (e.g., real-time video processing). Currently, 16-bit precision is only available for textures (and not for non-texture arrays), however, the support for 16-bit floating point arrays will be added in a future version of Quasar.

9.5 Multi-component Hardware Textures

Very often, kernel functions access RGB color data using slicing operations, such as:

```
x[m,n,0..2]
```

When the accesses `m` and/or `n` are irregular compared to the kernel function position variable `pos`, it may be useful to consider the use of multi-component hardware textures. These textures allow fetches of 2, 3 or 4 color components in one single operation, which is very efficient. A multi-component hardware texture can be declared by adding `'hwtex_nearest(4)` to the access modifier of the cube type. The modifier is only permitted to `mat`, `cube` or `cube{4}` types. Complex-valued data or higher dimensional matrices are currently not yet supported. An example of a Gaussian filter employing multi-component textures is given below:

```
function y = gaussian_filter_hor(x, fc, n)

    function [] = __kernel__ kernel(x : cube'hwtex_nearest(4), y : cube'unchecked, fc : vec'unchecked,
          n : int, pos : vec2)
        sum = [0.,0.,0.]
        for i=0..numel(fc)-1
            sum = sum + x[pos[0],pos[1]+i-n,0..2] * fc[i]
        endfor
        y[pos[0],pos[1],0..2] = sum
    endfunction

    y = uninit(size(x))
    parallel_do (size(y,0..1), x, y, fc, n, kernel)
endfunction
```

In parentheses, the number of components is indicated. Note that the hardware only supports 1, 2 or 4 components. In this mode, the Quasar compiler *will* support the texture fetching operation

```
x[pos[0],pos[1]+i-n,0..2]
```

and will translate the slice indexer into a 4-component texture fetch.
In combination with 16-bit floating point formats, the texture fetch even only requires a transfer of 64 bits (8 bytes) from the texture memory. On average, this will reduce the memory bandwidth by a factor 2 and at the same time reduces the stress on the global memory.
Finally, it is best to not use the same matrix value in `'hwtex_nearest(4)` mode and later in `'hwtex_nearest` mode (or vice versa) in another kernel function, because a mode change requires the texture memory to be reallocated and recopied (which affects the performance).

9.6 Texture/surface writes

For CUDA devices with compute capability 2.0 or higher, it is possible to write to the texture memory from a kernel function. In CUDA terminology, this is called a surface write. In Quasar, it suffices to declare the kernel function parameter using the modifier `'hwtex_nearest` (or `'hwtex_nearest(n)`) and to write to the corresponding matrix. One caveat is that the texture write is only visible starting from the **next** kernel function call. Consider the following example:

```
function [] = __kernel__ kernel (y: mat'hwtex_nearest, pos : ivec2)
    y[pos] = y[pos] + 1
```



```
    y[pos] = y[pos] + 1 % unseen change
endfunction
y = zeros(64,64)
parallel_do(size(y),y,kernel)
parallel_do(size(y),y,kernel)
print mean(y) % Result is 2 (instead of 4) because the surface writes
            % are not visible until the next call
```

This may be counterintuitive, but this allows the texture cache to work properly.
An example with 4 component surface writes is given below (one stage of a wavelet transform in the vertical direction):

```
function [] = __kernel__ dwt_dim0_hwtex4(x : cube'hwtex_nearest(4), y : cube'hwtex_nearest(4), wc :
    mat'hwconst, ctd : int, n : int, pos : ivec2)
    K = 16*n + ctd
    a = [0.0,0.0,0.0,0.0]
    b = [0.0,0.0,0.0,0.0]
    tilepos = int((2*pos[0])/n)
    j0 = tilepos*n
    for k=0..15
        j = j0+mod(2*pos[0]+k+K,n)
        u = x[j,pos[1],0..3]
        a = a + wc[0,k] * u
        b = b + wc[1,k] * u
    endfor
    y[j0+mod(pos[0],int(n/2)), pos[1],0..3]=a
    y[j0+int(n/2)+mod(pos[0],int(n/2)),pos[1],0..3]=b
endfunction

im = imread("lena_big.tif")
im_out = uninit(size(im))
parallel_do([size(im_out,0)/2,size(im_out,1)],im2,im_out,sym8,4,size(im_out,0), dwt_dim0_hwtex4)
```

On a Geforce GTX 780M, the computation times for 1000 runs are as follows:

```
without 'hwtex_nearest(4): 513 ms
with 'hwtex_nearest(4): 176 ms
```

Here this optimization resulted in a speedup of approx. a factor 3 (!)

9.7 Maximizing occupancy through shared memory assertions

Kernel functions that explicitly use shared memory can be optimized by specifying the amount of memory that a kernel function will actually use.
The maximum amount of shared memory that Quasar kernel functions can currently use is 32K (32768 bytes). Actually, the maximum amount of shared memory of the device is 48K (16K is reserved for internal purposes). The GPU may process several blocks at the same time, however there is one important restriction:

> *"The total number of blocks that can be processed at the same time also depends on the amount of shared memory that is used by each block."*

For example, if one block uses 32K, then it is not possible to launch a second block at the same time, because 2 x 32K>48K. In practice, your kernel function may only use e.g. 4K instead of 32K. This would then allow 48K/4K = 12 blocks to be processed at the same time.
When the amount of shared memory used by a kernel can not be deduced from the code, the Quasar runtime will assume that the kernel uses 32K shared memory per block. However, because N*32K < 48K requires N=1, only one block can be launched simultaneously. This significantly degrades the performance. Therefore it is recommended that you give some hints to the compiler about the amount of shared memory that the kernel function will use. This can be done as follows:

1. *Explicitly giving the dimensions of the shared memory arrays*

   When you request:

   ```
   x = shared(20,3,6)
   ```

   the compiler will reserve 20 x 3 x 6 x 4 bytes = 1440 bytes for the kernel function.

2. *Using assertions*

   Often the arguments of the function `shared` are non-constant. In this case you can use assertions. The Quasar compiler can then infer the total amount of shared memory that is being used through the logic system (see 5).

   ```
   assert(M<8 && N<20 && K<4)
   x = shared(M,N,K)
   ```

   Due to the above assertion, the compiler is able to infer the amount of required shared memory. In this case: 8 x 20 x 4 x 4 bytes = 2560 bytes. The compiler then gives the following message:

   ```
   Information: sharedmemtest.q - line 17: Calculated an upper bound for the amount of shared
       memory: 2560 bytes
   ```

Assertions have also an additional benefit: they allow the runtime system to check whether not too much shared memory will be allocated. In case `N` would exceed 20, the runtime system will give an error message.

9.8 Cooperative groups and warp shuffling functions

The following special kernel function parameters give fine grain control over GPU threads.

Parameter	Type	Description
`coalesced_threads`	`thread_block`	a thread block of coalesced threads
`this_thread_block`	`thread_block`	describes the current thread block
`this_grid`	`thread_block`	describes the current grid
`this_multi_grid`	`thread_block`	describes the current multi-GPU grid

The `thread_block` class has the following properties:

Property	Description

Property	Description
`thread_idx`	Gives the index of the current thread within a thread block
`size`	Indicates the size (number of threads) of the thread block
`active_mask`	Gives the mask of the threads that are currently active

The `thread_block` class has the following methods.

Method	Description
`sync()`	Synchronizes all threads within the thread block
`partition(size : int)`	Allows partitioning a thread block into smaller blocks
`shfl(var, src_thread_idx : int)`	Direct copy from another thread
`shfl_up(var, delta : int)`	Direct copy from another thread, with index specified relatively
`shfl_down(var, delta : int)`	Direct copy from another thread, with index specified relatively
`shfl_xor(var, mask : int)`	Direct copy from another thread, with index specified by a XOR relative to the current thread index
`all(predicate)`	Returns true if the predicate for *all threads* within the thread block evaluates to non-zero
`any(predicate)`	Returns true if the predicate for *any thread* within the thread block evaluates to non-zero
`ballot(predicate)`	Evaluates the predicate for all threads within the thread block and returns a mask where every bit corresponds to one predicate from one thread
`match_any(value)`	Returns a mask of all threads that have the same value
`match_all(value)`	Returns a mask only if all threads that share the same value, otherwise returns 0.

In principle, the above functions allow threads to communicate with each other, without relying on, e.g., shared memory. The warp shuffle operations allow taking values from other *active* threads (active means not disabled due to thread divergence). `all`, `any`, `ballot`, `match_any` and `match_all` allow to determine whether the threads have reached a given state.

The warp shuffle operations require a Kepler GPU (or higher) and allow the use of shared memory to be avoided (register access is faster than shared memory). This may bring again performance benefits for computationally intensive kernels such as convolutions and parallel reductions (`sum`, `min`, `max`, `prod` etc.).

Using this functionality will require the CUDA target to be specified explicitly (i.e., the functionality cannot be easily simulated by the CPU). This may be obtained by placing the following code attribute (see section §4.11) inside the kernel:

```
{!kernel target="nvidia_cuda"}
```

. For CPU execution a separate kernel needs to be written.

Below an example is given for a parallel reduction algorithm based on warp shuffling functions. This is an alternative to the parallel reduction algorithm using shared memory (section 10.6) and has the advantage that no shared memory is used.

```
function y : scalar = __kernel__ reduce_sum(coalesced_threads : thread_block, x : vec, pos : int)
    {!kernel target="nvidia_cuda"}
    val = x[pos]
    lane = coalesced_threads.thread_idx
    i = int(coalesced_threads.size / 2)
    val = x[pos]
```



```
    % accumulate within the warp
    while i > 0
        val += coalesced_threads.shfl_down(val, i)
        i /= 2
    endwhile

    if lane==0
        y += val
    endif

endfunction
```

In case the warp size is known to be 32, this can be even written as follows:

```
function y : scalar = __kernel__ reduce_sum(coalesced_threads : thread_block, x : vec, pos : int)
    {!kernel target="nvidia_cuda"}
    val = x[pos]
    lane = coalesced_threads.thread_idx

    % accumulate within the warp (of fixed size 32)
    val += coalesced_threads.shfl_down(val, 16)
    val += coalesced_threads.shfl_down(val, 8)
    val += coalesced_threads.shfl_down(val, 4)
    val += coalesced_threads.shfl_down(val, 2)
    val += coalesced_threads.shfl_down(val, 1)

    if lane==0
        y += val
    endif
endfunction
```

9.8.1 Fine synchronization granularity

As an extension of the cooperative groups, the keyword `syncthreads` accepts a parameter that indicates which threads are being synchronized. This allows more fine grain control on the synchronization.

Keyword	Description
`syncthreads(warp)`	performs synchronization across the current (possibly diverged) warp (32 threads)
`syncthreads(block)`	performs synchronization across the current block
`syncthreads(grid)`	performs synchronization across the entire grid
`syncthreads(multi_grid)`	performs synchronization across the entire multi-grid (multi-GPU)
`syncthreads(host)`	synchronizes all host (CPU and GPU threads)

The first statement `syncthreads(warp)` allows divergent threads to synchronize at any time (it is also useful in the context of Volta's independent scheduling). `syncthreads(block)` is equivalent to `syncthreads`. The grid synchronization primitive `syncthreads(grid)` is allows to place barriers inside kernel function that synchronize all blocks. The following function:

```
function y = gaussian_filter_separable(x, fc, n)
    function [] = __kernel__ gaussian_filter_hor(x : cube, y : cube, fc : vec, n : int, pos : vec3)
        sum = 0.
```



```
        for i=0..numel(fc)-1
            sum = sum + x[pos + [0,i-n,0]] * fc[i]
        endfor
        y[pos] = sum
    endfunction
    function [] = __kernel__ gaussian_filter_ver(x : cube, y : cube, fc : vec, n : int, pos : vec3)
        sum = 0.
        for i=0..numel(fc)-1
            sum = sum + x[pos + [i-n,0,0]] * fc[i]
        endfor
        y[pos] = sum
    endfunction

    z = uninit(size(x))
    y = uninit(size(x))
    parallel_do (size(y), x, z, fc, n, gaussian_filter_hor)
    parallel_do (size(y), z, y, fc, n, gaussian_filter_ver)
endfunction
```

Can now be simplified to:

```
function y = gaussian_filter_separable(x, fc, n)
    function [] = __kernel__ gaussian_filter_separable(x : cube, y : cube, z : cube, fc : vec, n : int
        , pos : vec3)
        sum = 0.
        for i=0..numel(fc)-1
            sum = sum + x[pos + [0,i-n,0]] * fc[i]
        endfor
        z[pos] = sum
        syncthreads(grid)
        sum = 0.
        for i=0..numel(fc)-1
            sum = sum + z[pos + [i-n,0,0]] * fc[i]
        endfor
        y[pos] = sum
    endfunction
    z = uninit(size(x))
    y = uninit(size(x))
    parallel_do (size(y), x, y, z, fc, n, gaussian_filter_separable)
endfunction
```

The advantage is not only in the improved readability of the code, but the number of kernel function calls can be reduced which further increases the performance. *Note: this feature requires at least an NVIDIA Pascal GPU.* There are a few caveats with grid synchronization:

- The number of active blocks should be less or equal than the total number of blocks that the GPU can process in parallel. This is to ensure that all active blocks can reach the grid barrier *at the same time*.
- The values of locally declared variables (such as `sum` in the above example) are *not* preserved across grid barriers and need to be reinitialized.

The Quasar compiler enforces these conditions automatically. However, in case manual control of the block size and/or block count is desirable, the function `opt_block_cnt` can be used (see following Section).

Important note: using cooperative group features such as grid and multi-grid synchronization requires cooperative kernel launches, which is only available for Pascal, Volta (or newer) GPUs and requires either Windows or Linux with the GPU device operating in **TCC driver mode**. Also see https://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/tesla_compute_cluster.htm. As an alternative, when cooperative kernel launching is not available, Quasar will *emulate* grid synchronization. This is achieved by disabling the configuration setting `CUDA_BACKEND_COOPERATIVEGROUPS_TCCDRIVER` in `Quasar.config.xml`.

9.8.2 Optimizing block count for grid synchronization

Whereas `opt_block_size` calculates the best suited block size for a kernel function, `opt_block_cnt` can be used to calculate the block count, in order to ensure that all blocks can be active at the same time. This condition is required when performing grid synchronization, but may also occur in a few kernel tiling schemes.

```
block_size = opt_block_size(kernel,data_dims)
grid_size = opt_block_cnt(kernel,block_size,data_dims)
parallel_do([grid_size.*block_size, block_size], kernel)
```

9.8.3 Memory fences

CUDA follows a weak consistence memory model, which means that shared/global memory writes are not necessarily performed in order. This may result in unexpected results in case the programming code assumes a fixed order of memory operations.
The keyword `memfence` can be used to place memory barriers in the code; this is useful when threads need to wait for a global memory operation to be completed. Currently `memfence` only has effect inside kernel/device functions. Note that these instructions in addition prevent the compiler from performing optimizations across the fence (e.g., caching a value in the registers).

Keyword	Description
`memfence(block)`	Suspends the current thread until its global/shared memory writes are visible by all threads in the current block
`memfence(grid)`	Suspends the current thread until its global memory writes are visible by all threads in the grid
`memfence(system)`	Suspends the current thread until its global memory writes are visible by all threads in the system (CPU, GPU)

9.9 Kernel launch bounds

By default, the runtime system decides on how many threads use when launching a given kernel. The back-end compiler (e.g., NVCC) however, must ensure that the kernel is compiled for a flexible number of threads. For this purpose, an upper limit (called launch bounds) for the possible number of threads is selected. The launch bounds are in turn used to calculate the number of registers used. The smaller the number of registers that are being used, the more thread blocks can be scheduled in parallel and the higher the efficiency. On the other hand, for large kernels with a large number of registers, the back-end compiler places less-used variables in the global memory to avoid to many registers to be used (called register spilling). When variables are stored in global memory, the cost can be tremendous.
However, often an approximate idea of the number of threads per block is known at compile-time. For example,

the technique from section 9.7 may assume that the number of threads per block is fixed to a given number. For practical purposes, let us assume that this number is 128, where the maximum number of threads the kernel permits is 1024. For 128 threads, the amount of shared memory used by the kernel (i.e., 512 bytes in single precision mode and 1024 bytes in double precision mode) is much lower than in case the kernel would be executed with 1024 threads per block. This allows the GPU to schedule multiple threads in parallel, which increases the computational performance of the kernel. Now, since we know that only 128 threads are used, the compiler has actually more registers available than in the case of 1024 threads, potentially avoiding the register spilling. Hence the compiler can tune the number of registers with this extra piece of information.
For this purpose, CUDA provides a `__launch_bounds__` directive, and similarly this directive is also available in Quasar, For examples:

```
{!cuda_launch_bounds max_threads_per_block=128}
```

where the maximal number of threads per block obviously needs to be smaller or equal than the maximal supported number of threads by the GPU (typically, 1024). It suffices to place this code attribute directly inside a kernel function (for example at the top). It is also possible to specify the minimal number of blocks per streaming processor (SM) that needs to be executed:

```
{!cuda_launch_bounds max_threads_per_block=128; min_blocks_per_sm = 4}
```

The number of registers used by a kernel function can be inspected in the Quasar Redshift IDE by hovering with the mouse over the left margin in the source code editor, at the position of a kernel function. We have seen performance improvements of 20-30% or more by tuning the number of registers and number of threads per blocks.
The kernel launch attribute is also generated by several automatic code transformations present in the Quasar compiler.

9.10 Memory management

There are some problems operating on large images that do not fit into the GPU memory. The solution is to provide a FAULT-TOLERANT mode, in which the operations are completely performed on the CPU (we assume that the CPU has more memory than the GPU). Of course, running on the CPU comes at a performance hit. Therefore I will add some new configurable settings in this enhancement.
Please note that GPU memory problems can only occur when the total amount of memory used by one single kernel function > (max GPU memory - reserved mem) * (1 - fragmented mem%). For a GPU with 1 GB, this might be around 600 MB. Quasar automatically transfers memory buffers back to the CPU memory when it is running out of GPU space. Nevertheless, this may not be sufficient, as some very large images can take all the space of the GPU memory (for example 3D datasets).
Therefore, three configurable settings are added to the runtime system (see `Quasar.Redshift.config.xml`):

1. `RUNTIME_GPU_MEMORYMODEL` with possible values:
   - *SmallFootPrint* - A small memory footprint - opts for conservative memory allocation leaving a lot of GPU memory available for other programs in the system
   - *MediumFootprint* (default) - A medium memory footprint - the default mode
   - *LargeFootprint* - chooses aggressive memory allocation, consuming a lot of available GPU memory quickly. This option is recommended for GPU memory intensive applications.
2. `RUNTIME_GPU_SCHEDULINGMODE` with possible values:

- *MaximizePerformance* - Attempts to perform as many operations as possible on the GPU (potentially leading to memory failure if there is not sufficient memory available. Recommended for systems with a lot of GPU memory).

- *MaximizeStability* (default) - Performs operations on the CPU if there is not GPU memory available. For example, processing 512 MB images when the GPU only has 1 GB memory available. The resulting program may be slower. (FAULT-TOLERANT mode)

3. `RUNTIME_GPU_RESERVEDMEM`

   - The amount of GPU memory reserved for the system (in MB). The Quasar runtime system will not use the reserved memory (so that other desktop programs can still run correctly). Default value = 160 MB. This value can be decreased at the user's risk to obtain more GPU memory for processing (desktop applications such as Firefox may complain...)

Please note that the "`imshow`" function also makes use of the reserved system GPU memory (the CUDA data is copied to an OpenGL texture).

9.11 Querying GPU hardware features

In general, GPU hardware features will be unlocked when the hardware supports it. However, for target-dependent kernels (e.g., see section 17.2.5), it might be useful to retrieve hardware related parameters at runtime. This can be achieved using the function `gpu_feature`:

```
feature = gpu_feature(feature_name)
```

where `feature_name` can be any of the following:

feature_name	description
`num_sm`	The number of streaming processors for the primary GPU
`cuda_arch`	The CUDA architecture (`major*100+minor*10`). For example: 700 (Volta)
`max_reg_per_block`	The max. number of registers available per GPU block
`max_shared_mem_per_block`	The max. amount of shared memory available per GPU block
`max_shared_mem_per_sm`	The max. amount of shared memory available per streaming processor
`max_constant_memory`	The max. amount of constant memory available
`warp_size`	The size of one warp (typically: 32)
`max_threads_per_block`	The maximum number of threads available per block
`max_warps_per_sm`	The maximum number of warps per streaming processor
`max_blocks_per_sm`	The maximum number of blocks per streaming processor

These functions are mostly available in order to avoid hard-coded constants in the code. Quasar kernel functions should be written in such a way that they work across different GPU architectures and not relying on certain particular hardware constants. These functions, together with `max_block_size` and `opt_block_size` (see section 2.4.4) help to achieve this.

Chapter 10

Parallel programming examples

This section contains a number of useful parallel programming examples together with an explanation.

10.1 Gamma correction

As a first example, we demonstrate how a gamma correction can be programmed in Quasar.

```
x = imread("image.png")
y = copy(x)
gamma = 0.22
parallel_do(size(y),y,gamma,__kernel__ (y:cube'unchecked, gamma:scalar, pos:ivec3) -> _
        y[pos] = 255*(y[pos]*(1.0/255))^gamma)
imshow(y)
```

The above approach makes use of `__kernel__` lambda expressions, which allows to define `__kernel__` functions in just one line of code. Note that it is possible to put multiple statements inside a lambda expression, this is done as follows:

```
kernel_lambda = __kernel__ (y:cube'unchecked) -> (statement1; statement2; ...)
```

Sometimes, it is useful to share functionality between different kernel functions. This can be achieved using a `__device__` function:

```
gamma_correction = __device__ (x:scalar,gamma:scalar) -> _
    255*(y*(1.0/255))^gamma
gamma_correction_kernel = __kernel__ (y:cube'unchecked, gamma:scalar, pos:ivec3) -> _
    y[pos] = gamma_correction(x[pos], gamma)
```

Device functions are defined in the same way as kernel functions, but they can *not* be directly executed using the `parallel_do` function.

10.2 Fractals

As a second example, we consider the calculation of the Mandelbrot fractal. In Quasar, this can be obtained using quite simple code, by using complex arithmetic.

```
% Mandelbrot fractal with Normalized Iteration Count algorithm
function [] = __kernel__ mandelbrot_fractal(im : mat'unchecked, s : scalar, _
    t : cscalar, num_it : int, pos : ivec2)
    p = (float(pos) ./ size(im,0..1))-0.5
    c = t+s*complex(p[1],p[0])
    z = 0i
    N = 2.0
    for n=1..num_it
        if abs(z)>N
            break
        endif
        z = z*z + c
    endfor
    im[pos] = n-log2(log(abs(z))/log(N))
endfunction

x = zeros(768,768)
parallel_do(size(x),x,10,complex(-1.42),512,mandelbrot_fractal)
imshow(x,[])
```

10.3 Image rotation, translation and scaling [basic]

The example below uses a `__device__` function to perform linear interpolation. The main kernel function then performs an affine transform on its position argument, `pos`. Boundary checking in the function `linear_interpolate` is only performed once, using the test `min(i) >= 0 && max(i-size(img_in,0..1)) < -1`. Alternatively, the modifier `'unchecked` in `img_in:cube'unchecked` can be omitted, which would give the same result, but this would result in 4 boundary checks (one for each `img_in[...]` access) instead of 1.

```
function [] = rotatescaletranslate(img_in, img_out, theta, s, tx, ty)
    % Device function for performing linear interpolation
    function [q:vec3] = __device__ linear_interpolate(img_in:cube'unchecked, p:vec2)
        i = floor(p)
        f = frac(p)
        if min(i) >= 0 && max(i-size(img_in,0..1)) < -1
            q = img_in[i[0],i[1],0..2] * (1 - f[0]) * (1 - f[1]) + _
                img_in[i[0],i[1]+1,0..2] * (1 - f[0]) * f[1] + _
                img_in[i[0]+1,i[1],0..2] * f[0] * (1 - f[1]) + _
                img_in[i[0]+1,i[1]+1,0..2] * f[0] * f[1]
        else
            q = [0.,0.,0.]
        endif
    endfunction

    function [] = __kernel__ tf_kernel(img_out : cube'unchecked, _
                img_in:cube'unchecked, A:mat'unchecked'const, t:vec2, pos:ivec2)
        center = size(img_in,0..1)/2
```



```
        p = pos - center
        p = [A[0,0]*p[0] + A[0,1]*p[1], A[1,0]*p[0] + A[1,1]*p[1]] + center + t
        img_out[pos[0],pos[1],0..2] = linear_interpolate(img_in, p)
    endfunction

    degrees_to_radians = theta -> theta*pi/180
    theta = degrees_to_radians(theta)
    A = [[cos(theta), -sin(theta)],
         [sin(theta), cos(theta)]] * 2^s

    parallel_do(size(img_out,0..1),img_out,img_in,A,-[ty,tx],tf_kernel)
endfunction
```

10.4 2D Haar inplace wavelet transform using lifting

The following code demonstrates an inplace Haar wavelet transform, implemented using the lifting scheme (but without normalization). The forward and backward transform respectively use the 2×2 transform matrices:

$$\vec{A} = \begin{pmatrix} 1/2 & 1/2 \\ 1 & -1 \end{pmatrix} \quad \text{and} \quad \vec{A}^{-1} = \begin{pmatrix} 1 & 1/2 \\ 1 & -1/2 \end{pmatrix}.$$

The main advantages of the Haar wavelet transform in the context of Quasar programs, is that the transform is very fast (takes less than 2 ms to compute for a $512 \times 512 \times 3$ input image on a NVidia Geforce 435M using the CUDA computation engine). Moreover, for integer input data within the range $[0, 255]$, this unnormalized transform does not suffer from floating point rounding errors, hence the reconstruction (backward transform applied after the forward transform) is exact.

Forward transform:

```
function [] = haar_fw(x, num_scales)
    function [] = __kernel__ hor_haar_fw_kernel(x : cube'unchecked, _
                      y : cube'unchecked, j : int, pos : ivec3)
        n = size(x,1)/2^(j+1)
        if mod(pos[1],2)==0
            [a, b] = [x[pos], x[pos+[0,1,0]]]
            y[pos[0],pos[1]/2,pos[2]]=0.5*(a+b)
            y[pos[0],pos[1]/2+n,pos[2]]=a-b
        endif
    endfunction
    function [] = __kernel__ ver_haar_fw_kernel(x : cube'unchecked, _
                      y : cube'unchecked, j : int, pos : ivec3)
        m = size(x,0)/2^(j+1)
        if mod(pos[0],2)==0
            [a, b] = [x[pos], x[pos+[1,0,0]]]
            y[pos[0]/2,pos[1],pos[2]]=0.5*(a+b)
            y[pos[0]/2+m,pos[1],pos[2]]=a-b
        endif
    endfunction

    tmp = zeros(size(x))
    for j=0..num_scales-1
        sz = [size(x,0)/2^j,size(x,1)/2^j,size(x,2)]
```

```
        parallel_do(sz,x,tmp,j,hor_haar_fw_kernel)
        parallel_do(sz,tmp,x,j,ver_haar_fw_kernel)
    endfor
endfunction
```

Backward transform:

```
function [] = haar_bw(x, num_scales)
    function [] = __kernel__ hor_haar_bw_kernel(x : cube'unchecked, _
                             y : cube'unchecked, j : int, pos : ivec3)
        n = size(x,1)/2^(j+1)
        if mod(pos[1],2)==0
            a = x[pos[0],pos[1]/2,pos[2]]
            b = x[pos[0],pos[1]/2+n,pos[2]]
            y[pos]=a+0.5*b
            y[pos+[0,1,0]]=a-0.5*b
        endif
    endfunction
    function [] = __kernel__ ver_haar_bw_kernel(x : cube'unchecked, _
                             y : cube'unchecked, j : int, pos : ivec3)
        m = size(x,0)/2^(j+1)
        if mod(pos[0],2)==0
            a = x[pos[0]/2,pos[1],pos[2]]
            b = x[pos[0]/2+m,pos[1],pos[2]]
            y[pos]=a+0.5*b
            y[pos+[1,0,0]]=a-0.5*b
        endif
    endfunction

    tmp = zeros(size(x))
    for j=num_scales-1..-1..0
        sz = [size(x,0)/2^j,size(x,1)/2^j,size(x,2)]
        parallel_do(sz,x,tmp,j,hor_haar_bw_kernel)
        parallel_do(sz,tmp,x,j,ver_haar_bw_kernel)
    endfor
endfunction
```

10.5 Convolution

As a fifth example, we will illustrate how a 3×3 local means filter can be implemented. There are different possibilities: 1) using a non-separable filtering, 2) using separable filtering (but requiring extra memory to store the intermediate values), or 3) using shared memory (see section 2.4.4).

1. Non-separable implementation

```
x = imread("image.png")
y = zeros(size(x))
parallel_do(size(y),x,y,__kernel__ (x:cube,y:cube,pos:ivec3) -> _
    y[pos] = (x[pos+[-1,-1,0]]+x[pos+[-1,0,0]]+x[pos+[-1,1,0]] + _
```



```
            x[pos+[ 0,-1,0]]+x[pos ]+x[pos+[0,1,0]] + _
            x[pos+[ 1,-1,0]]+x[pos+[ 1,0,0]]+x[pos+[1,1,0]])*(1.0/9))
imshow(y)
```

2. Separable implementation:

```
x = imread("image.png")
y = zeros(size(x))
tmp = zeros(size(x))
parallel_do(size(y),x,tmp,__kernel__ (x:cube,y:cube,pos:ivec3) -> _
    y[pos] = x[pos+[-1,0,0]]+x[pos]+x[pos+[1,0,0]])
parallel_do(size(x),tmp,y,__kernel__ (x:cube,y:cube,pos:ivec3) -> _
    y[pos] = x[pos+[0,-1,0]]+x[pos]+x[pos+[0,1,0]]*(1.0/9))
imshow(x)
```

3. Separable implementation, using shared memory:

```
function [] = __kernel__ filter3x3_kernel_separable(x:cube,y:cube,pos:ivec3,
    blkpos:ivec3,blkdim:ivec3)
    [M,N,P] = blkdim+[2,0,0]
    assert(M<=10 && N<=16 && P<=3) % specify upper bounds for the amount of shared memory
    vals = shared(M, N, P) % shared memory

    sum = 0.
    for i=pos[1]-1..pos[1]+1 % step 1 - horizontal filter
        sum += x[pos[0],i,blkpos[2]]
    endfor
    vals[blkpos] = sum % store the result
    if blkpos[0]<2 % filter two extra rows (needed for vertical filtering)
        sum = 0.
        for i=pos[1]-1..pos[1]+1
            sum += x[pos[0]+blkdim[0],i,blkpos[2]]
        endfor
        vals[blkpos+[blkdim[0],0,0]] = sum
    endif
    syncthreads
    sum = 0.
    for i=blkpos[0]..blkpos[0]+2 % step 2 - vertical filter
        sum += vals[i,blkpos[1],blkpos[2]]
    endfor
    y[pos] = sum*(1.0/9)
endfunction
x = imread("image.png")
y = zeros(size(x))
parallel_do(size(y),x,y,filter3x3_kernel_separable)
imshow(y)
```

Comparison of the computation times:

Implementation	Time/run (NVidia Geforce 435M)
Non-separable	3.70 msec
Separable	4.24 msec
Separable, w. shared memory	3.51 msec

It can be noted that a separable implementation for a 3×3 filter kernel, only brings a benefit when shared memory is used.
Remarks:

- The out of bounds checking compilation (see section 17.3) option needs to be turned off in order to have this benefit.
- Also important is that the upper bounds for using shared memory are specified. This can be done using the assertion system. The compiler is then able to compute the maximal amount of shared memory that will be needed by the kernel function (see section 9.7).

10.6 Parallel reduction sum

Note that the Quasar compiler will generate automatically code that performs a parallel sum (see section §8.4). This section is mainly for educational purposes, for understanding the shared memory and thread synchronization.
A parallel sum can be implemented in Quasar using a logarithmic algoritm of complexity $\log_2 N$. This consists of first computing "partial" sums of groups of elements, stored in shared memory, followed by recursively adding of the shared memory partial sums. A lot of information on this kind of algorithm can be found in literature. In Quasar, the implementation for vectors is as follows:

```
function [y : scalar] = __kernel__ my_sum(x : vec'unchecked,
    blkpos : int, blkdim : int)

    bins = shared(blkdim) % Note - we assume that blkdim is a power of two!
    nblocks = (numel(x)+blkdim-1)/blkdim
    % step 1 - parallel sum
    val = 0.0
    for m=0..nblocks-1
        if blkpos + m*blkdim < numel(x)
            val += x[blkpos + m*blkdim]
        endif
    endfor
    bins[blkpos] = val
    % step 2 - reduction
    syncthreads
    bit = 1
    while bit < blkdim
        index = 2*bit*blkpos
        if blkpos+index<blkdim
            bins[index] = bins[index] + bins[index + bit]
        endif
        syncthreads
        bit *= 2
    endwhile
    % write output
```



```
    if blkpos == 0
       y += bins[0]
    endif
endfunction
```

In step 1, the input is split in a number of blocks, where each block has size "blockdim". Then all blocks are summed in parallel, the results are stored in "bins" (has one entry per block element). In step 2, all elements of bins are added together, using an FFT-like butterfly. When $blkdim = 16$, the algorithm is as follows:

```
% iteration 1 (subsequent steps are performed in parallel)
bins[0] += bins[1]
bins[2] += bins[3]
bins[4] += bins[5]
bins[6] += bins[7]
bins[8] += bins[9]
bins[10] += bins[11]
bins[12] += bins[13]
bins[14] += bins[15]
syncthreads
% iteration 2 (subsequent steps are performed in parallel)
bins[0] += bins[2]
bins[4] += bins[6]
bins[8] += bins[10]
bins[12] += bins[14]
% iteration 3
bins[0] += bins[4]
bins[8] += bins[12]
% iteration 4
bins[0] += bins[8]
```

Finally, the end result (`bins[0]`) is accumulated in the kernel output argument `y` (see section 4.7).
Remark: due to the atomic operation `+=`, the result is not deterministic: floating point rounding errors depend on the order of the operations. For an atomic add, the order of operations is not specified. This can be solved by storing the intermediate results in a vector, and summing this vector independently.
The above example can be used to write a more generic parallel reduction, that can be used for multiplication, maximization, minimization:

```
type accumulator : [__device__ (scalar, scalar) -> scalar]
function y : scalar = __kernel__ parallel_reduction(x : vec'unchecked,
     acc : accumulator, val : scalar, blkpos : int, blkdim : int)
    bins = shared(blkdim) % Note - we assume that blkdim is a power of two!
    nblocks = (numel(x)+blkdim-1)/blkdim)
    for m=0..nblocks-1 % step 1 - parallel sum
        if blkpos + m*blkdim < numel(x)
            val = acc(val, x[blkpos + m*blkdim])
        endif
    endfor
    bins[blkpos] = val
    syncthreads % step 2 - reduction
    bit = 1
```



```
    while bit < blkdim
        index = 2*bit*blkpos
        if index + bit < blkdim
            bins[index] = acc(bins[index], bins[index + bit])
        endif
        syncthreads
        bit *= 2
    endwhile
    y += bins[0] % write output
endfunction
device_sum = __device__ (x : scalar, y : scalar) -> x + y
device_prod = __device__ (x : scalar, y : scalar) -> x * y
reduction (x : cube) -> sum(x) = parallel_do(512, x, device_sum,0,parallel_reduction)
reduction (x : cube) -> prod(x) = parallel_do(512, x, device_prod,0,parallel_reduction)
```

Here, we define the accumulation functions (`device_sum` and `device_prod`), and we pass the functions dynamically to the `parallel_reduction` function.
Note that the Quasar compiler is also able to recognize for-loops that could benefit from the parallel reduction algorithm. In this case, the for-loop is automatically transformed to the above algorithm (see section §8.4).

10.7 A more accurate parallel sum

As mentioned in section 2.2.1, floating point math is not associative, and the order of the summations may depend on the GPU architecture (the used block dimensions, etc.). The code below illustrates a more accurate parallel summation algorithm than in the previous section, combining Kahan's algorithm, with the parallel sum reduction reduction. The main idea of Kahan's algorithm, is to accumulate small errors in a separate variable. Because the operations do not require any extra global or shared memory, all operations are performed in local memory (see section 2.4.3), yielding minimal overhead compared to the direct algorithm.

```
% Sum of all elements in the specified cube.
function y : scalar = r_sum(x : cube) concealed
    function [y : scalar] = __kernel__ r_sum_kernel(x : vec,
        nblocks : int, blkdim : int, blkpos : int)

        s = shared(blkdim)
        % step 1 - parallel sum
        sum = 0.0
        c = 0.0
        for n=0..nblocks-1
            if blkpos + n * blkdim < numel(x)
                % Kahan's sum reduction
                u = x[blkpos + n * blkdim] - c
                t = sum + u
                c = (t - sum) - u
                sum = t
            endif
        endfor
        s[blkpos] = sum
        % step 2 - reduction
        syncthreads
```



```
        % now sort all bins from large to small magnitudes
        bit = 1
        % use regular summing
        while bit<blkdim
            index=2*bit*blkpos
            if index+bit<blkdim
                s[index] = s[index] + s[index+bit]
                syncthreads
            endif
            bit *= 2
        endwhile
        if blkpos==0
            y += s[0]
        endif
    endfunction
    y = r_aggregator(x, r_sum_kernel)
endfunction

% Aggregator helper function (deals with the computation
% of the block sizes)
function z = r_aggregator(x, kernel) concealed
    N = numel(x)
    BLOCK_SIZE = prod(max_block_size(kernel, N))
    nblocks = int((N + BLOCK_SIZE-1) / BLOCK_SIZE)
    z = parallel_do([[1,BLOCK_SIZE],[1,BLOCK_SIZE]],x,nblocks,kernel)
endfunction

% Define a reduction to replace the summing function by our
% "improved" implementation.
reduction (x : cube) -> sum(x) = r_sum(x)
```

10.8 Parallel sort

To implement a parallel sorting algorithm, several algorithms exist. For the bitonic sort algorithm, the Quasar implementation is as follows:

```
function [] = sort(x)
    function [] = __kernel__ bitsort(x : mat, n : int, blkdim : ivec2,
        blkpos : ivec2, pos : ivec2)
        k = 2
        % copy the row to the shared memory...
        s = shared(blkdim[0], n)
        for l = 0..blkdim[1]..n-1
            tid = blkpos[1] + l
            if tid < size(x,1)
                s[blkpos[0], tid] = x[pos[0], tid]
            else
                s[blkpos[0], tid] = 1e37 % maximum floating point value
            endif
        endfor
        syncthreads
```



```
        % parallel bitonic sort
        while k <= n
            % bitonic merge
            j = int(k / 2)
            while j > 0
                for l = 0..blkdim[1]..n-1
                    tid = blkpos[1] + l % thread id
                    ixj = xor(tid, j)
                    if tid < ixj
                        if and(tid, k) == 0
                            v = [blkpos[0], tid]
                            w = [blkpos[0], ixj]
                        else
                            v = [blkpos[0], ixj]
                            w = [blkpos[0], tid]
                        endif
                        if s[v] > s[w]
                            [s[v], s[w]] = [s[w], s[v]]
                        endif
                    endif
                endfor
                syncthreads
                j /= 2
            endwhile
            k *= 2
        endwhile

        % Copy back the results
        for l = 0..blkdim[1]..n-1
            tid = blkpos[1] + l
            if tid < size(x,1)
                x[pos[0], tid] = s[blkpos[0], tid]
            endif
        endfor
    endfunction

    nextpow2 = x -> 2^ceil(log2(x))
    n = nextpow2(size(x,1))
    sz = max_block_size(bitsort, [size(x,0),min(n,256),1])
    parallel_do([[size(x,0),sz[1],1],sz],x,n,bitsort)
endfunction
```

A complete explanation of the bitonic sort algorithm can be found on http://en.wikipedia.org/wiki/Bitonic_sorter. Here, bitonic sorting is applied along the *rows* of the matrix.
The function handles input sizes that are not a multiple of two.

10.9 Matrix multiplication

Matrix multiplication in CUDA is so much fun that some people write books on this topic (see http://www.shodor.org/media/content//petascale/materials/UPModules/matrixMultiplication/moduleDocument.pdf). The following is the block-based solution proposed by NVidia. The solution exploits shared memory to reduce the number

of accesses to global memory.

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        // Block row and column
        int blockRow = blockIdx.y, blockCol = blockIdx.x;
        // Each thread block computes one sub-matrix Csub of C
        Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
        // Each thread computes 1 element of Csub accumulating results into Cvalue
        float Cvalue = 0.0;
        // Thread row and column within Csub
        int row = threadIdx.y, col = threadIdx.x;
        // Loop over all the sub-matrices of A and B required to compute Csub
        for (int m = 0; m < (A.width / BLOCK_SIZE); ++m)
        {
                // Get sub-matrices Asub of A and Bsub of B
                Matrix Asub = GetSubMatrix(A, blockRow, m);
                Matrix Bsub = GetSubMatrix(B, m, blockCol);
                // Shared memory used to store Asub and Bsub respectively
                __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
                // Load Asub and Bsub from device memory to shared memory
                // Each thread loads one element of each sub-matrix
                As[row][col] = GetElement(Asub, row, col);
                Bs[row][col] = GetElement(Bsub, row, col);
                __syncthreads();
                // Multiply Asub and Bsub together
                for (int e = 0; e < BLOCK_SIZE; ++e)
                Cvalue += As[row][e] * Bs[e][col];
                __syncthreads();
        }
        // Each thread writes one element of Csub to memory
        SetElement(Csub, row, col, Cvalue);
}
```

(Note: some functions are omitted for clarity)
However, this implementation is only efficient when the number of rows of matrix A is about the same as the number of cols of A. In other cases, performance is not optimal. Second, there is the issue that this version expects that the matrix dimensions are a multiple of `BLOCK_SIZE`. Why use a 3x3 matrix if we can have a 16x16?
In fact, there are 3 cases that need to be considered (let $n < N$):

1. $(n \times N) \times (N \times n)$: The resulting matrix is small: in this case, it is best to use the parallel sum algorithm.
2. $(N \times N) \times (N \times N)$: The number of rows/cols of A are more or less equal: use the above block-based algorithm.
3. $(N \times n) \times (n \times N)$: The resulting matrix is large: it is not beneficial to use shared memory.

The following example illustrates this approach in Quasar:

```
% Dense matrix multiplication
function C = dense_multiply(A : mat, B : mat)
    % Algorithm 1 - is well suited for calculating products of
```



```
% large matrices that have a small matrix as end result.
function [] = __kernel__ kernel1(a : mat'unchecked, b : mat'unchecked,
    c : mat'unchecked, blkdim : ivec3, blkpos : ivec3)
        n = size(a,1)
        bins = shared(blkdim)
        nblocks = int(ceil(n/blkdim[0]))
        % step 1 - parallel sum
            val = 0.0
            for m=0..nblocks-1
            if blkpos[0] + m*blkdim[0] < n % Note - omitting [0] gives error
                   d = blkpos[0] + m*blkdim[0]
                   val += a[blkpos[1],d] * b[d,blkpos[2]]
            endif
            endfor
            bins[blkpos] = val
            % step 2 - reduction
            syncthreads
            bit = 1
            while bit < blkdim[0]
            if mod(blkpos[0],bit*2) == 0
                          bins[blkpos] += bins[blkpos + [bit,0,0]]
            endif
            syncthreads
            bit *= 2
            endwhile
            % write output
            if blkpos[0] == 0
            c[blkpos[1],blkpos[2]] = bins[0,blkpos[1],blkpos[2]]
            endif
endfunction

% Algorithm 2 - the block-based algorithm, as described in the CUDA manual
function [] = __kernel__ kernel2(A : mat'unchecked, B : mat'unchecked,
C : mat'unchecked, BLOCK_SIZE : int, pos : ivec2, blkpos : ivec2, blkdim : ivec2)
        % A[pos[0],m] * B[m,pos[1]]
            sA = shared(blkdim[0],BLOCK_SIZE)
            sB = shared(BLOCK_SIZE,blkdim[1])
            sum = 0.0
            for m = 0..BLOCK_SIZE..size(A,1)-1
            % Copy submatrix
            for n = blkpos[1]..blkdim[1]..BLOCK_SIZE-1
                          sA[blkpos[0],n] = pos[0] < size(A,0) && m+n < size(A,1) ? A[pos[0],m+n] :
                               0.0
            endfor
            for n = blkpos[0]..blkdim[0]..BLOCK_SIZE-1
                          sB[n,blkpos[1]] = m+n < size(B,0) && pos[1] < size(B,1) ? B[m+n,pos[1]] :
                               0.0
                endfor
                syncthreads
                % Compute the product of the two submatrices
            for n = 0..BLOCK_SIZE-1
                          sum += sA[blkpos[0],n] * sB[n,blkpos[1]]
            endfor
```

```
                syncthreads
            endfor
            if pos[0] < size(C,0) && pos[1] < size(C,1)
            C[pos] = sum % Write the result
            endif
    endfunction

    % Algorithm 3 - the most straightforward algorithm
    function [] = __kernel__ kernel3(A : mat'unchecked, B : mat'unchecked,
        C : mat'unchecked, pos : ivec2)
            sum = 0.0
            for m=0..size(A,1)-1
            sum += A[pos[0],m]*B[m,pos[1]]
            endfor
            C[pos] = sum
    endfunction

    [M,N] = [size(A,0),size(B,1)]
    C = zeros(M,N)
    if M <= 4
           P = prevpow2(max_block_size(kernel1,[size(A,1),M*N])[0])
           parallel_do([P,M,N],A,B,C,kernel1)
    elseif size(A,1)>=8 && M >= 8
           P = min(32, prevpow2(size(A,1)))
           blk_size = max_block_size(kernel2,[32,32])
           sz = ceil(size(C,0..1) ./ blk_size) .* blk_size
           parallel_do([sz,blk_size],A,B,C,P,kernel2)
    else
           parallel_do(size(C),A,B,C,kernel3)
    endif
endfunction
```


Chapter 11

Multi-GPU programming

Quasar supports multi-device configurations, which allows several GPUs to be combined with a CPU. For the programmer, outside kernel/device functions, the programming model is sequential in nature, irrespective of whether one or multiple GPUs are being used. The Quasar multi-GPU feature allows a program to be executed on multiple GPUs (let say 2), without any/very little changes (see below) to the code, while benefitting from a 2x acceleration. To achieve this, the load balancing is entirely automatic and will take advantage of the available GPUs, when possible. The run-time system supports peer-to-peer memory transfers (when available) and transfers via host pinned memory. Here, host pinned memory is used to make sure that the memory copies from the GPU to the host are entirely asynchronous.

Each of the GPU devices has its own command queue, this is a queue on which the load balancer places individual commands that needs to be processed by the respective devices. The load balancer takes several factors into account, such as memory transfer times, load of the GPU, dependencies of the kernel function, ...

The multi-GPU functionality relies on the scheduler and load balancer in the Hyperion (v2) runtime system; therefore, it is only available for Hyperion devices (v2).

Systems with multiple GPUs contain either multi-GPU boards with PCI Express bridge chip or multiple PCI Express slots. In case of multiple PCI express slots, memory transfers from GPU A to GPU B need to pass the host (CPU) memory. Therefore, there are huge differences between the memory transfer times in the system (e.g. local device transfers, between GPU and the host and between GPU peers). Due to passing the CPU memory, memory transfers between GPUs may not be as efficient as expected, causing overhead and non-linear multi-GPU performance scaling.

It is therefore necessary to have a good understanding of the different factors that affect performance in a multi-GPU system.

11.1 A quick glance

All the memory transfers between the GPUs and between host and GPU, are managed automatically (and reduced as much as possible). In some cases it is useful to have more control about which GPU is used for which task. This can be achieved by explicitly setting the GPU device via a scheduling instruction:


```
{!sched gpu_index=0}
or
{!sched gpu_index=1}

{!sched} to reset to automatic scheduling
```

This overrides the default decision of the load balancer. For for-loops this can be done as follows:

```
for k=0..num_tasks-1
    {!sched gpu_index=mod(k,2)}
    parallel_do(..., kernel1)
    parallel_do(..., kernel2)
endfor
```

This way, each GPU will take care of one iteration of the loop. To enhance the load balancing over the GPUs, it may also be more beneficial to use the following technique

```
{!parallel for; multi_device=true}
for k=0..num_tasks-1
    parallel_do(..., k, kernel1)
    parallel_do(..., k, kernel2)
endfor
```

Here,

```
{!parallel for; multi_device=true}
```

will essentially unroll the for-loop twice, where each `parallel_do` function is launched on a different device. Internally, the following code is generated:

```
for k=0..2..num_tasks-1
    {!sched gpu_index=0}
    parallel_do(..., k, kernel1)
    {!sched gpu_index=1}
    parallel_do(..., k+1, kernel1)

    {!sched gpu_index=0}
    parallel_do(..., k, kernel2)
    {!sched gpu_index=1}
    parallel_do(..., k+1, kernel2)
endfor
```

11.2 Setting up the device configuration

A Hyperion device configuration is stored in the application user folder (evaluate quasardir("app_dir") in Redshift to know the location):

```
<quasar>
    <computation-engine name="v2 dual CUDA/CPU engine" short-name="CUDA - v2">
        <cpu-device num-threads="2" max-cmdqueue-size="32" cuda-hostpinnable-memory="true" />
        <cuda-device max-concurrency="4" max-cmdqueue-size="64" ordinal="0" />
```

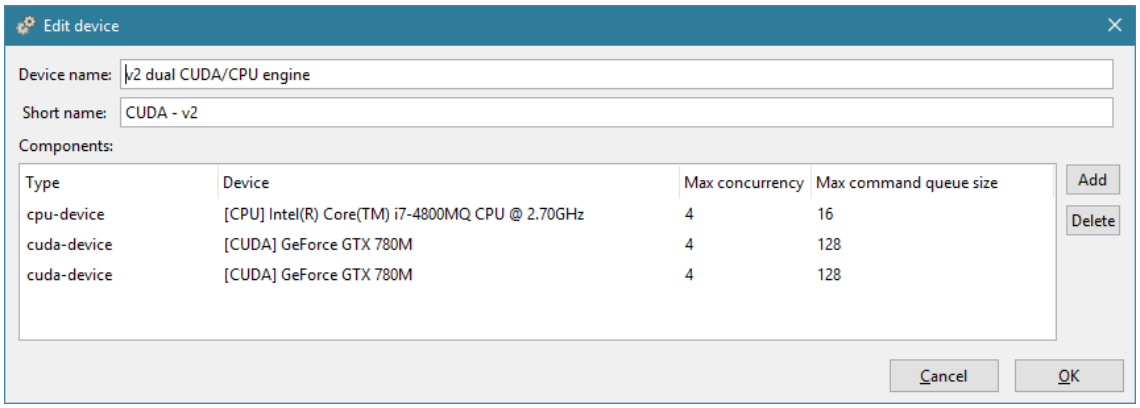


```
        <cuda-device max-concurrency="4" max-cmdqueue-size="64" ordinal="1" />
    </computation-engine>
</quasar>
```

A multi-GPU configuration has one CPU device and at least two CUDA devices. The following parameters are available:

Parameter	explanation
ordinal	the GPU index (typically 0, 1, 2, 3, ...)
num−threads	the number of CPU threads used for launching parallel kernels (note: each kernel can spawn some more OpenMP threads, this number is determined by dividing the number of logical processor cores by num−threads).
max−cmdqueue−size	the maximum length of the device command queue (concurrent kernel execution mode only).
max−concurrency	the number of CUDA streams associated to each device (concurrent kernel execution mode only)
cuda−hostpinnable−memory	if true, memory transfers between host CPU and the GPU(s) will be accelerated using host pinnable (non-pageable) memory. This is required for asynchronous memory copies.

A typical maximum size for the command queue is 64. This value strikes a balance between sufficient concurrency on the one hand and buffering and scheduling overheads on the other hand. A good value for max-concurrency is 4, higher values often don't impact the GPU performance (neither positively or negatively), due to the limited number of kernels that the hardware can run concurrently.
Note that a default Hyperion device configuration is automatically generated by the Quasar installer. Additionally, the Hyperion computation engine can be configured in Redshift via "Configure Devices":



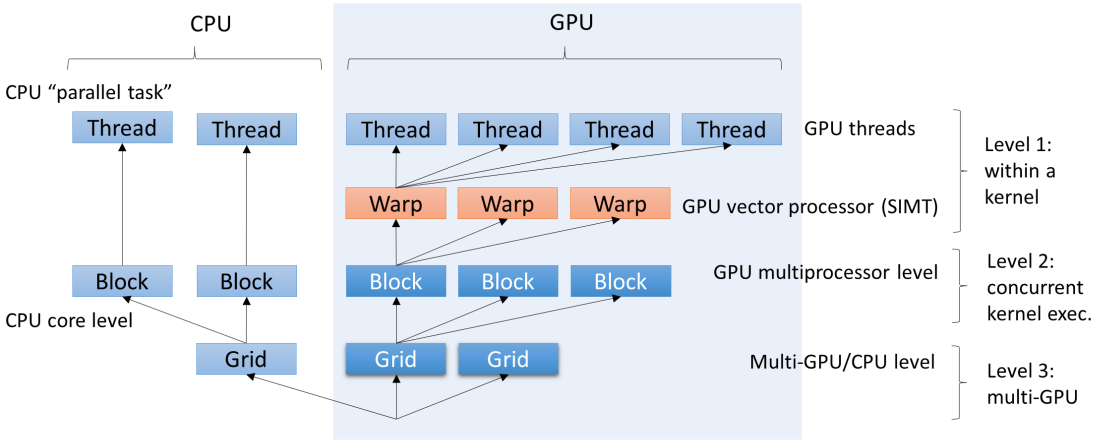
11.3 Three levels of concurrency

Quasar features three levels of concurrency:

- *Level 1: within a kernel*: GPU threads are executed in parallel, via parallel_do() or via automatically parallelized for-loops.
- *Level 2: concurrent kernel execution*: Quasar will concurrently launch kernels (e.g., by automatically assigning CUDA streams). This requires a runtime option ("concurrent kernel execution") to be enabled. When one GPU is not fully occupied (e.g., due to a low occupancy), the GPU may execute subsequent kernels concurrently.
- *Level 3: multi-GPU processing*: kernels can be launched on different GPUs in the system.

This requires:

- load balancing, to ensure that each GPU is sufficiently (or maximally) utilized
- scheduling: kernel launches may be reordered in order to obtain a better device utilization
- automatic memory transfers (peer-to-peer) between GPU devices. As mentioned before, the peer-to-peer copies may pass the CPU host memory. This is to be avoided, so this issue is also taken into account by the runtime system when making scheduling and load balancing decisions.



For each of the three concurrency levels, Quasar has an automatic mode. It some cases it may be beneficial to switch to the manual mode as well, e.g., to take control in own hands or to further optimize the program.

11.4 Manual vs. automatic multi-GPU scheduling

Quasar supports two multi-GPU scheduling modes:

1. *Automatic scheduling*: the scheduling and load-balancing algorithm decides fully autonomously on which GPU (or CPU) to execute the given kernels. This is a fairly sophisticated algorithm that not only takes kernel/task dependencies into account, but also the memory state, the required memory transfers (e.g., peer-to-peer copies) and synchronization between the GPUs.
2. *Manual scheduling*: here the user specifies which sections of the code run on which GPU. This is mostly useful when the code lends itself for logical separation onto multiple GPU.

Both modes can be used interchangedly during the program execution: the automatic scheduling takes the manual scheduling rules into account. Therefore, it is perfectly possible that both techniques complement each other. In the following, we explain in more detail how this is done.

In Quasar, the scheduler can be controlled via code attributes (Quasar's equivalent for pragmas and attributes in other programming languages).

Code attribute	explanation
{!sched mode=auto}	sets the scheduling mode to automatic
{!sched mode=cpu}	sets the scheduling mode to CPU, meaning that the following kernel functions will be executed on the CPU
{!sched mode=gpu}	sets the scheduling mode to GPU, meaning that the following kernel functions will be executed on the GPU
{!sched mode=gpu; gpu_index=n}	sets the scheduling mode to GPU n, meaning that the following kernel functions will be executed on GPU n
{!sched gpu_index=n}	retains the scheduling mode, but sets GPU n as the active GPU. The following kernel functions scheduled to the GPU will be executed on GPU n

When the GPU scheduling mode is not specified from within the code, the scheduler will automatically assign a GPU depending on the current load of the GPUs (*automatic scheduling*) and the associated memory transfer costs. Switching to manual mode can be performed by one of the above scheduling code attributes. At any point in time, it is possible to switch back to automatic scheduling, by means of {!sched mode=auto}.
During scheduling, it may occur that some kernel functions have preference for a certain GPU, while other kernel functions have not. The scheduler takes the preferences into account, so that unassigned kernel functions may be scheduled to the other GPU, if available.
Additionally, it is possible to manually copy variables to a specified GPU target:

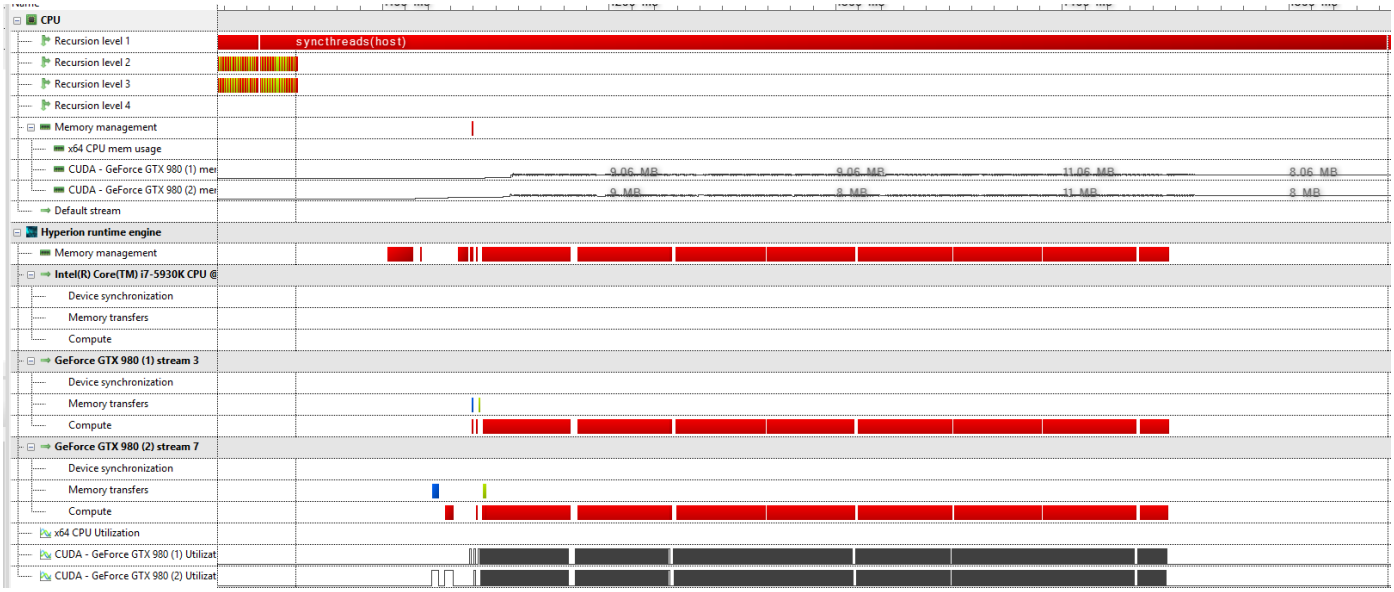
Code attribute	explanation
{!transfer vars=A; target=gpu; gpu_index =0}	Copies the variable A to GPU 0 memory
{!transfer vars=A; target=gpu; gpu_index =1}	Copies the variable A to GPU 1 memory
{!transfer vars=A; target=cpu}	Copies the variable A to the CPU memory

Important to know is that the above code attributes should be used only when necessary (e.g., when profiling has indicated that it is advantageous):

- Using these code attributes incorrectly may lead to unnecessary memory transfers
- Additionally, a transfer operation involves a scheduling synchronization, leading all the pending kernels for variable A to be launched immediately. This may interfere with the scheduling algorithm leading to a degraded performance.

11.5 Host Synchronization

The global scheduling algorithm generally only launches kernels at the moment that the results are needed (e.g., for further processing on the host CPU or for visualization). To enforce all pending kernels to be launched, a host-wide barrier can be used using syncthreads(host). This function works similar to syncthreads(block), but with the difference that all computation devices in the system will be synchronized. Below is a screenshot of the Redshift profiler that shows that syncthreads(block) performs a lot of work: in fact a whole batch of kernel functions is launched.



One subtle issue to be aware of is that measuring time differences in a multi-GPU environment using tic() and toc() may not give the desired results: it is possible that all kernel functions in between tic() and toc() are postponed for execution and in that case toc() will result a very small time difference (e.g., 1 microsecond). To correct the timing, it is best to synchronize the devices:

```
tic()
    parallel_do(...)
    syncthreads(host)
toc()
```

The reason that toc() does not implicitly imply syncthreads(host), is that host synchronization breaks concurrency: suppose that we would have the following code fragment:

```
tic()
    {!sched mode=gpu; gpu_index=0}
    parallel_do(...)
    syncthreads(host)
toc()
```



```
tic()
    {!sched mode=gpu; gpu_index=1}
    parallel_do(...)
    syncthreads(host)
toc()
```

Here, the host barrier would cause GPU 2 to be idle in the first tic() ... toc() block and correspondingly, GPU 1 is also idle in the second tic() ... toc() block. To measure independently how long work takes on GPUs 1 and 2. The execution times of a kernel on a GPU can therefore not be measured *independently*, at least not when other kernels are running on the other GPU. To optimize execution performance, it is nevertheless useful to have these independent measurements. These can be achieved using the built-in profiling tools.

11.6 Key principles for efficient multi-GPU processing

There are 5 key principles to be applied to enable efficient multi-GPU processing:

1. *Compute intensive kernels*: the program contains enough kernels that are compute-bound (i.e., not limited by memory or register restrictions). In case a program does not fully utilize a single GPU, its performance will most likely not be improved by using multiple GPUs. It is therefore recommended to optimize performance first on a single GPU, before attempting to get performance benefits by switching to multiple GPUs.

2. *Inter-kernel concurrency*: there must exist concurrency between kernels within a window of N subsequent launches. Typically, N is quite large (e.g., N=32768) to detect sufficient concurrency. Often, dependencies between kernel functions exist, which means that a kernel function needs to wait for the result of another kernel function. This is fine, as long as a kernel does not need its results from different GPUs. If this is the case, a peer-to-peer copy between the GPUs is performed, which may have one subtle drawback: during the peer-to-peer copy, both GPUs are involved and the copy performs synchronization between the two devices. Luckily, the scheduler can detect this situation and work around it. Correspondingly, the programs that benefit the most from multi-GPU acceleration and that can enjoy linear scaling, are the programs in which logical separation is possible between the GPUs and for which limited data transfer between the GPUs is required.

3. *Aggregation variables impose device synchronization*: whenever a scalar result is obtained (for example, by calling the sum() function), the CPU synchronizes with the GPU device. The corresponding kernel function (together with its dependencies) needs to be executed *immediately*. This significantly reduces the freedom of the scheduler in reordering the operations. To avoid this problem, it is beneficial to use vectors of length 1:

   ```
   function [] = __kernel__ calc_sum(result : vec(1), data : cube, pos : vec3)
       result += data[pos]
   endfunction
   ```

   This approach is more efficient than using the return parameters of the kernel function (which are currently immediately read by the CPU when the kernel function is complete). Only when directly accessing the content of the vector result:

   ```
   print result[0]
   ```

   a device synchronization will be performed.

A related subtle side effect is when passing vectors/matrices with small dimensions (numel(x)<=64) to a kernel function. Because these objects are passed directly to the registers of the kernel function, the values are read-out by the CPU. To avoid this problem, it is best to omit the length of the vector or size of the matrix in the kernel function definition.

```
function [] = __kernel__ process(mean_location:vec, pos:int) % vec instead of vec(3)
endfunction
```

Note that this problem only occurs for objects that are being written in one kernel and read in a subsequent kernel. For read-only data, there is no problem.

4. *Memory transfers between host and GPU* should be avoided as much as possible, for the same reason as above. In concurrent kernel execution mode, the runtime performs asynchronous memory transfers, therefore the issue from point 3 does not apply. Because memory transfers are *implicit*, a program may perform more memory transfers to intended. It is then useful to investigate the profiling results (e.g., the profiling timeline in Redshift) to find the origins of the memory transfers.

5. *Avoid duplicate calculation within loops*: when a constant intermediate value (vector, matrix or cube) is required, this value should not be recalculated over and over again, requiring repeated GPU transfers. Instead it is better to compute the value once and reuse it. The runtime keeps the memory resident in multiple GPUs, so that no GPU transfers are requires for kernel functions using the value.

In order to reach linear multi-GPU scaling, it is necessary to take the above principles into account.

11.7 Supported Libraries

Currently all built-in Quasar functions support multi-GPU processing. Furthermore, the following CUDA libraries have been enabled:

- cuFFT: Fast Fourier transforms
- cuBLAS: Basic Linear Algebra subprograms(*)
- cuSolver: Solvers for linear systems
- cuDNN: Deep neural networks

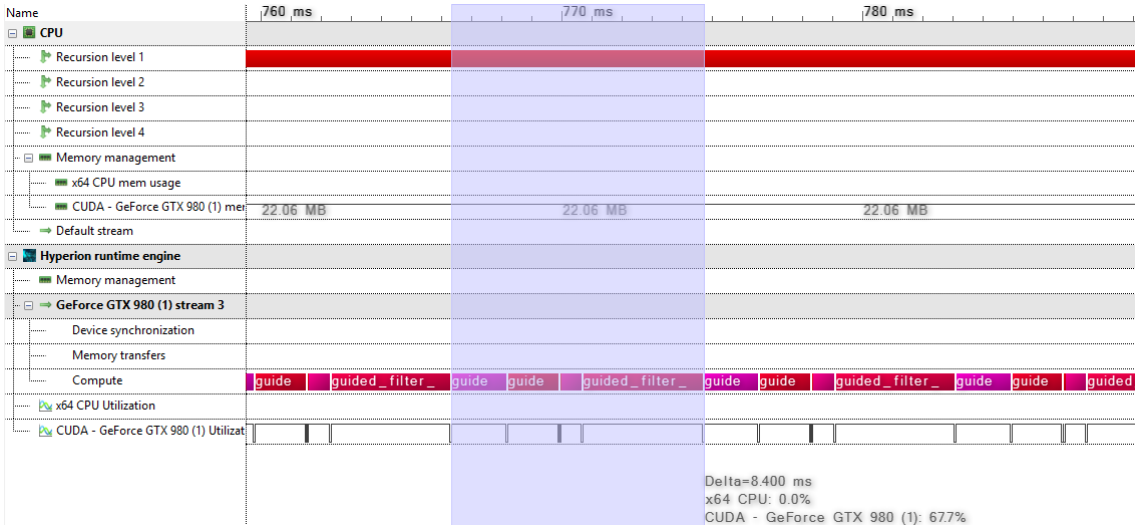
In automatic scheduling mode, the Quasar runtime will automatically dispatch the CUDA library functions to the available GPUs.
(*) Some of the cuBLAS functions, in particular those that result a scalar value, perform implicit device synchronization by default. Therefore these functions do not offer a lot of multi-GPU benefits for the moment. In a future version, this issue will be mitigated.

11.8 Profiling techniques

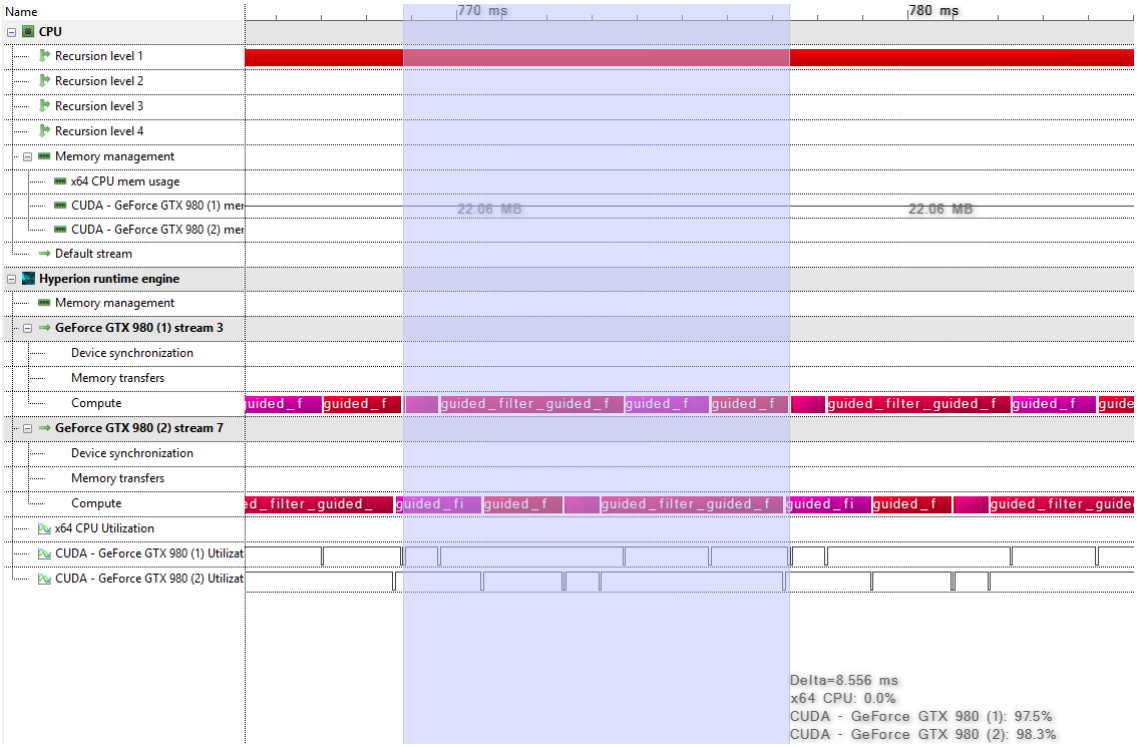
To speed up Quasar programs that do not take optimally advantage of the available GPUs, it is necessary to determine which key principle(s) that is/are violated. For this purpose, using the Redshift profiler, several techniques can be used to analyze the behavior of multi-GPU programsThe profiler incorporates the CUDA Profiling tools (CUPTI), which are also used by NVidia NSight and NVidia Visual Profiler. In contrast to the NVidia NSight and Visual Profiler, the Redshift profiler links the kernel functions directly to the host functions and the source code;

therefore, it is possible to obtain extra information, such as which code initiated a particular memory allocation or memory transfer, which parameters are passed to a given kernel function etc. The GPU events view gives the specific order of the commands that are being sent to the GPU, together with the relevant source code links.
In this section we discuss several profiling techniques that are useful to identify potential multi-GPU execution issues.

1. *Identify whether compute intensive kernels are present*: compute intensive kernels can easily be spotted in the timeline view. Ideally, subsequent kernels should start immediately, with no gap in between. So for example the single GPU execution of the guided_filter .q test program:



Here, all kernels are contiguously executed. Due to the low number of dependencies between the kernels, automatic multi-GPU scheduling will yield linear scaling, as demonstrated by the following multi-GPU execution:



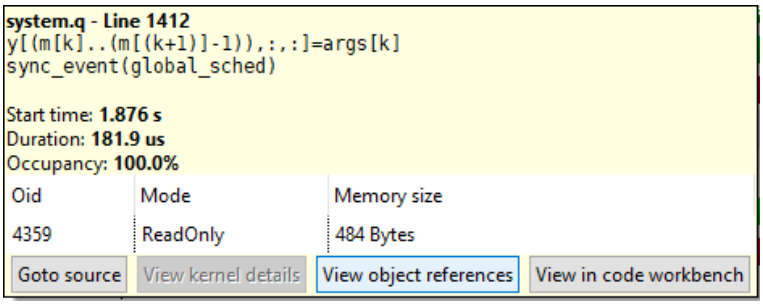
No code changes were required to obtain the multi-GPU execution. It sufficed to switch from the CUDA to the dual CUDA device.

1. *Check inter-kernel concurrency*: dependencies between kernels can be inspected in the Redshift profiler view.

2. *Check device synchronization*: calls to the global scheduler can be found in the device synchronization track with label sync_event(global_sched). The device that contains sync_event is the device that is being synchronized.

Host functions are indicated in red, memory allocations in yellow, executed kernel functions in magenta and memory transfers between GPUs in green.

In this case, it is useful to check whether the scheduler invocation can be avoided. By inspecting the tooltip of sync_event(global_sched) and clicking onto "view in code workbench", it is possible to track down the variable that causes the device synchronization.



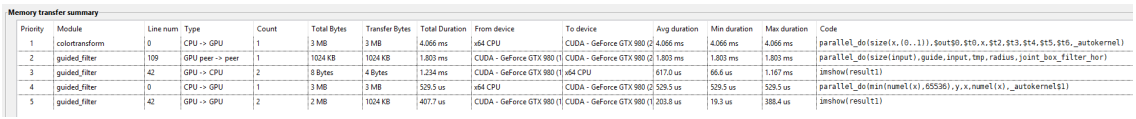
The tooltip also shows a table containing the object identifier (oid), access mode and memory size. In Quasar, each object (e.g., vector, matrix) has a unique identifier. The oid is displayed in the GPU events view and in

the memory profiler. By clicking onto "View object references", all operations on this object can be visualized. In this case, the CPU is accessing a variable with size 484 bytes. This requires the global scheduler to be invoked. Note however that the global scheduler may also work without any particular device synchronization being required: this is the result of the global scheduling queue being full.

Additionally, in the above screenshot, note the high number of peer to peer memory transfers indicates a poor usage of both GPUs: peer to peer memory transfers block both involved GPUs and therefore severely impacts the multi-GPU scaling.

3. *Check for memory transfer bottlenecks*

   Memory bottlenecks can be identified in the memory transfer summary view. Displayed are the type of transfer (e.g. from CPU to GPU, from GPU to CPU, between GPU peers), the number of times that this memory transfer took place, minimum, average and maximum duration of each transfer, etc.

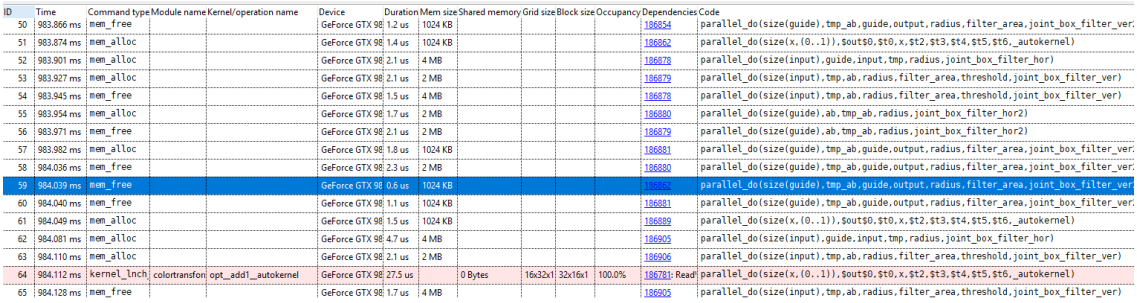

Memory transfer summary

Priority	Module	Line num	Type	Count	Total Bytes	Transfer Bytes	Total Duration	From device	To device	Avg duration	Min duration	Max duration	Code
1	colortransform	0	CPU -> GPU	1	3 MB	3 MB	4.066 ms	x64 CPU	CUDA - GeForce GTX 980 (2	4.066 ms	4.066 ms	4.066 ms	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
2	guided_filter	109	GPU peer -> peer	1	1024 KB	1024 KB	1.803 ms	CUDA - GeForce GTX 980 (1	CUDA - GeForce GTX 980 (2	1.803 ms	1.803 ms	1.803 ms	parallel_do(size(input),guide,input,tmp,radius,joint_box_filter_hor)
3	guided_filter	42	GPU -> CPU	2	8 Bytes	4 Bytes	1.234 ms	CUDA - GeForce GTX 980 (1	x64 CPU	617.0 us	66.6 us	1.167 ms	imshow(result1)
4	guided_filter	0	CPU -> GPU	1	3 MB	3 MB	529.5 us	x64 CPU	CUDA - GeForce GTX 980 (2	529.5 us	529.5 us	529.5 us	parallel_do(min(numel(x),65536),y,x,numel(x),_autokernel$1)
5	guided_filter	42	GPU -> GPU	2	2 MB	1024 KB	407.7 us	CUDA - GeForce GTX 980 (1	CUDA - GeForce GTX 980 (1	203.8 us	19.3 us	388.4 us	imshow(result1)

   The tooltip of this report allows to directly browse to the source code that caused this memory transfer. Additionally, object identifiers (oid) can be tracked via the GPU events view.

4. *Check for duplicate calculations/operations in the GPU events view*

   The GPU events view gives a listing of all the commands executed on the GPU. See the documentation on the enhanced profiler for the details.



ID	Time	Command type	Module name	Kernel/operation name	Device	Duration	Mem size	Shared memory	Grid size	Block size	Occupancy	Dependencies	Code
50	983.866 ms	mem_free			GeForce GTX 98	1.2 us	1024 KB					186854	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
51	983.874 ms	mem_alloc			GeForce GTX 98	1.4 us	1024 KB					186862	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
52	983.901 ms	mem_alloc			GeForce GTX 98	2.1 us	4 MB					186878	parallel_do(size(input),guide,input,tmp,radius,joint_box_filter_hor)
53	983.927 ms	mem_alloc			GeForce GTX 98	2.1 us	2 MB					186879	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
54	983.945 ms	mem_free			GeForce GTX 98	1.5 us	4 MB					186878	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
55	983.954 ms	mem_alloc			GeForce GTX 98	1.7 us	2 MB					186880	parallel_do(size(guide),ab,tmp_ab,radius,joint_box_filter_hor2)
56	983.971 ms	mem_free			GeForce GTX 98	2.1 us	2 MB					186879	parallel_do(size(guide),ab,tmp_ab,radius,joint_box_filter_hor2)
57	983.982 ms	mem_alloc			GeForce GTX 98	1.8 us	1024 KB					186881	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
58	984.036 ms	mem_free			GeForce GTX 98	2.3 us	2 MB					186880	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
59	984.039 ms	mem_free			GeForce GTX 98	0.6 us	1024 KB					186882	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
60	984.040 ms	mem_free			GeForce GTX 98	1.1 us	1024 KB					186881	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
61	984.049 ms	mem_alloc			GeForce GTX 98	1.5 us	1024 KB					186889	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
62	984.081 ms	mem_alloc			GeForce GTX 98	4.7 us	4 MB					186905	parallel_do(size(input),guide,input,tmp,radius,joint_box_filter_hor)
63	984.110 ms	mem_alloc			GeForce GTX 98	2.1 us	2 MB					186906	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
64	984.112 ms	kernel_lnch	colortransfor	opt__add1__autokernel	GeForce GTX 98	27.5 us		0 Bytes	16x32x1	32x16x1	100.0%	186781: Read	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
65	984.128 ms	mem_free			GeForce GTX 98	1.7 us	4 MB					186905	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)

   In particular, it is important that a result is only calculated once, when it is used several times. This way, the runtime can keep a copy of the data resident in the device memory of each of the GPUs.

11.9 Automatic GPU scheduling

When the GPU scheduling mode is not specified from within the code, the scheduler will automatically assign a GPU depending on the current load of the GPUs and the associated memory transfer costs. For this technique to be effective, it is currently required that the code is structured in such a way that subsequent kernel calls can be parallelized over the different GPUs, with limited memory transfer. In case the scheduler detects large overheads due to memory transfers, it is very likely that the code will be executed on only one GPU. However, when the code (mostly) contains parallel operations on separate memory blocks (matrices etc.) with only few synchronizations between the GPUs, the automatic GPU scheduling will be able to detect the coarse grain parallelism and the program will be able to use multiple GPUs.

11.10 Developing multi-GPU applications

A practical workflow for developing multi-GPU applications is then as follows:

1. Start from a Quasar program that is multi-GPU agnostic (i.e., that runs on a single GPU).
2. Profile the program and determine the code regions in which the automatic GPU scheduling (section 11.9) is ineffective.
3. For these regions, insert manual scheduling commands as outlined in section 11.1

Note that even after adding manual scheduling commands, the program can still work in single-GPU mode, without any changes (no error is generated when `gpu_index` is larger or equal than the number of available GPUs!) However, improper use of `{!sched}` may lead to performance degradations, so it is recommended to profile the multi-GPU application regularly.

Chapter 12

SIMD processing on CPU and GPU

In SIMD, multiple processing elements process the data simultaneously, driven by a single instruction stream. SIMD is supported by x86/x64 architectures, as well as ARM and refers to special 'multimedia' instructions that were originally added to allow real-time video encoding/decoding operations as well as 3D games to run the CPU.
CUDA supports SIMD processing via its warp-based execution model, also known as single instruction multiple threads (SIMT). However, CUDA supports a limited set of SIMD intrinsics on half precision floating point formats and 8-bit/16-bit integer types that can be used within a *single* thread, allowing essentially the vector length to be extended above the warp size (for example vector length 128 for 8-bit integers).
In essence, when performing calculations using the vectors of appropriate length (see below), the operations are automatically mapped onto a SIMD implementation whenever possible. For example im[m,n,0..3] might (depending on the settings and machine architectures) load into an SSE register (four 32-bit floating point numbers) in x86/x64 code, while im[m,n,0..7] might lead to an AVX load (eight 32-bit floating point numbers).
Some type aliases are defined in inttypes.q and floattypes.q to simplify the SIMD processing:

Type	Defined in	Meaning	Alias for
i8vec4	inttypes.q	8-bit signed integer vector of length 4	vec[int8](4)
u8vec4	inttypes.q	8-bit unsigned integer vector of length 4	vec[uint8](4)
i16vec4	inttypes.q	16-bit signed integer vector of length 4	vec[int16](4)
u16vec4	inttypes.q	16-bit unsigned integer vector of length 4	vec[uint16](4)
hvec2	floattypes.q	16-bit (half precision) floating point vector of length 2	vec[scalar'half](4)

By using vector types such as above, the back-end compiler (e.g., MSVC, GCC, ...) can choose the appropriate SIMD instructions. Because not all compilers have the best code generation in this respect (as a sidenote, the Intel C/C++ compiler currently offers the best results in our experiments), several low-level operations have been

manually vectorized in the header file quasar_simd.h.
The Quasar compiler supports automatic vectorization of kernel functions. This relieves the programmer from unfolding loops and writing manually vectorized expressions. A kernel can automatically be enabled for SIMD processing by the following code attribute:

```
{!kernel_transform enable="simdprocessing"; target="cpu"}
```

where target specifies cpu (in case of x86/x64 SIMD) or cuda (in case of CUDA SIMD). When the target is ommitted, SIMD is applied to any target architecture (when possible). Using Quasar SIMD processing, the SIMD acceleration depends much less on the back-end compiler that is being used.
Because it is not guaranteed that SIMD processing improves the performance (especially when the used vector types or operations are natively supported), the SIMD processing is not enabled by default. Therefore, the following subsections list the operations and vector types that are accelerated.

12.1 Storage versus computation types

It is important to distinguish between types used for *storage* and types used for *calculation*. Storage types represent the data in a more compressed format allowing the bandwidth to the memory to be reduced whenever the precision requirements allow it. For example, an image may be stored in uint8 format, whereas the processing is performed in int32 (or even scalar) format. This combines the storage benefits with the calculation benefits associated with the computation units of the device.
Going one step further, the computations may also be performed in a lower precision, allowing, e.g., in CUDA 4 uint8 integers to be processed simultaneously. This is useful to reduce the pressure on the computation units of the GPU multi-processor. However, one should take into account that:

- intermediate calculations may suffer from *integer overflows* or *lack of precision*
- not all operations are supported by the hardware (for example, mathematical functions such as sin, cos on integer data), and using these operations will reduce the performance (because the compiler needs to convert to an integer or floating-point format, generate code to call the function and convert back to the initial integer format).

Therefore, by default, calculations are performed in a suited type for which all operations are fully supported by the device (typically, 32-bit integers or 32-bit floating point operations). Conversion operations between storage types and computation types (e.g., from u8vec4 to ivec4 and back) can be implemented using SIMD instructions depending on the device.
Storage modifiers:
Integer types have storage modifiers. These determine how integer overflows are handled when the results of a computation are written back to memory. When using SIMD vector types, the overflow detection can also be accelerated using SIMD instructions. The following table gives an overview of which integer overflow checks are currently accelerated.

Modifier	Meaning	x86/x64 SIMD	CUDA SIMD
'unchecked	No overflow checking is performed	Yes	Yes

Modifier	Meaning	x86/x64 SIMD	CUDA SIMD
'checked	Overflow checking is performed, an overflow leads to a runtime error	No	No
'sat	Values are clipped (saturated) to the value range of the target type	Yes	No

12.2 x86/x64 SIMD accelerated operations

Supported vector lengths: 4 and 8 (see below).
x86 and x64 processors support various vector processing extensions: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2. It is important that the selected CPU target architecture matches the machine, otherwise an exception will be generated. The target architecture can be configured in the Program Settings dialog box in Redshift (or alternatively in the Quasar.config.xml configuration file):
Target architecture specifies the architecture to tune for. Vector extensions sets the highest level of vector processing instructions that the processor supports.
The following operations are guaranteed to result in SIMD instructions, independent of the back-end compiler.
Floating point operations:

Operation	Type	Requirements (32-bit float)	Requirements (64-bit float)
Operators +, −, .∗, ./	scalar4	SSE	SSE2 / AVX
Operators +, −, .∗, ./	scalar8	AVX	AVX
Comparison operators ==, !=, >, <, >=, <=	scalar4	SSE	SSE2 / AVX
Functions min, max, floor, ceil, sqrt, rsqrt, round, frac, sign	scalar4	SSE2	SSE2 / AVX
Functions sum, prod, dotprod	scalar4	SSE2	SSE2 / AVX

Integer operations:

Operation	Type	Requirements
Operators +, −	int4	SSE2
Operators .∗	int4	SSE4.1
Comparison operators ==, !=, >, <, >=, <=	int4	SSE2
Functions abs, min, max, sign, sum, prod, any	int4	SSE2
Functions and, or, xor,	int4	SSE2

Type conversions:

Operation	Type	Requirements (32-bit float)	Requirements (64-bit float)
Conversion	vec4 –> ivec4	SSE2	SSE2 / AVX
Conversion	ivec4 -> vec4	SSE2	SSE2 / AVX

Operation	Type	Requirements (32-bit float)	Requirements (64-bit float)
Conversion	ivec4 -> i8vec4, i16vec4, u8vec4, u16v	SSE2	
Conversion	i8vec4, i16vec4, u8vec4, u16vec4 -> ivec4	SSE2	

12.2.1 Example: AVX image filtering on CPU

The following code demonstrates how to use SIMD processing on the CPU.

```
{!parallel for}
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
        {!kernel_transform enable="simdprocessing"; target="cpu"; numel=8}
        r = 0.0
        for x=0..7
            r += im[m,n+x]
        end
        im_out[m,n] = r/(2*K+1)
    endfor
endfor
```

The simdprocessing kernel transform causes the loop to be automatically parallelized and vectorized. The parameter numel specifies the desired vector length (8 for AVX, 4 for SSE). If numel is omitted, the vector length is determined depending on the available SIMD extensions of the CPU (e.g., AVX, SSE).
Whenever suited, the vectorizer will convert branches to branch-free expressions that can be vectorized. In addition, *cooldown* code is generated for cases that the image dimensions (in the above example `size(im,1)`) are not a multiple of the SIMD width.

12.3 CUDA SIMD accelerated operations

The following SIMD vector types are supported:

Type name	Alias for	Meaning
u8vec4	vec[uint8](4)	Four unsigned integers (0-255)
i8vec4	vec[int8](4)	Four signed integers (-128..127)
u16vec2	vec[uint16](2)	Two unsigned integers (0-65536)
i16vec2	vec[int16](2)	Two signed integers (-32768-32767)
hvec2	vec[scalar'half](2)	Two half-precision floating point numbers

CUDA vector instructions always interact with 32-bit words. Depending on the precision (8-bit integer, 16-bit integer or 16-bit floating point), the vector length is either 2 or 4.
Floating point operations:
The half-precision floating point format (FP16) is useful to reduce the device memory bandwidth for algorithms which are not sensitive to the reduced precision. For half, only integers between -2048 and 2048 can exactly be represented. Integers larger than 65520 or smaller than -65520 are rounded toward respectively positive and negative infinity.
Next to the reduced bandwidth, starting with the Pascal architecture, the GPU also offers hardware support for computations using this type. To gain maximal performance benefits, it is best to use the 32-bit length 2 SIMD half type (hvec2). Use of hvec2 typically results in two numbers being calculated in parallel, leading to a performance that is similar to one single precision floating point (FP32) operation. However, a Volta or Turing GPU is required

to obtain performance benefits from using calculation in half precision format. For Kepler, Maxwell and Pascal GPUs, hardware support for operations in half precision is notably *slow*, therefore it is best to use the half type only for storage purposes (i.e., calculations are performed in the single-precision floating point type).
The following table shows the operations that have been accelerated using half types. For unsupported operations, the computation will be performed in FP32 format, leading to extra conversions between FP32 and FP16.

Operation	Type
Operators +, −, .∗	hvec2
Comparison operators ==, !=, >, <, >=, <=	hvec2
Functions ceil, cos, exp, floor, log, log2, log10, sin, rsqrt, sqrt, round	hvec2

Integer operations:
CUDA supports SIMD operations for 8-bit and 16-bit integer vectors that fit into a 32-bit words. Below is a table of the operations that are accelerated. For unsupported operations, the computation will be performed in 32-bit integer format, leading to extra conversions between 32-bit and 8/16-bit integer formats.

Operation	Type	Requirements
Operators +, −	i8vec4, i16vec2, u8vec4, u16vec2	Any CUDA version
Comparison operators ==, !=, >, <, >=, <=	i8vec4, i16vec2, u8vec4, u16vec2	Any CUDA version
abs	i8vec4, i16vec2	Any CUDA version
min, max	i8vec4, i16vec2, u8vec4, u16vec2	Any CUDA version

When reading lower precision values such as `uint8(4)`, `int16(2)`, ... from an array, the Quasar compiler will upcast the value to the best suited machine precision value (usually, `vec[int]`). This avoids unintended integer overflows in the computations, but may also prevent some operations from being SIMD accelerated.
As a workaround, Quasar provides the `{!simd_noupcast}` code attribute. The following box filtering example illustrates the use of this code attribute:

```
function [] = boxfilter8(im8 : mat[uint8], im_out : mat[uint8])
  {!parallel for}
  for m=0..size(im,0)-1
    for n=0..16..size(im,1)-1
      {!simd_noupcast enable=im8}
      r = vec[uint8](16)
      for x=0..K-1
        r += im8[m,n+x+(0..15)]
      endfor
      im_out[m,n+(0..15)] = int(r/(2*K))
    endfor
  endfor
endfunction
```

This technique ensures that the accumulation in `r` is performed in `uint8` precision. Be aware that there are no integer overflow checks in this code.

12.3.1 Example: 8-bit image filtering

The following example illustrates how to make use of CUDA SIMD integer operations. The calculations are performed in 32-bit integer format, while 8-bit unsigned integers are used for storage.

```
import "inttypes.q"
im8 = uint8(imread("image.jpg"))
im_out = mat[uint8'sat](size(im8))
{!parallel for}
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
       {!kernel_transform enable="simdprocessing"; target="cuda"; numel=4}
        r = 0
        for x=0..K-1
            r += im8[m,n+x]
        endfor
        im_out[m,n] = int(r/(2*K)) % integer division requires int() function
    endfor
endfor
```

Compared to the previous example, the vectorization is performed automatically.

12.3.2 Example: 16-bit half float image filtering

```
import "floattypes.q"
im_half = scalar_to_half(imread("image.jpg"))
for m=0..size(im_half,0)-1
    for n=0..size(im_half,1)-1
        {!kernel_transform enable="simdprocessing"; target="cuda"; numel=2}
        r = 0.0
        for x=0..K-1
            r += im_half[m,n+2*x]
        endfor
        im_out2[m,n] = r/(2*K+1)
    endfor
endfor
```

12.4 ARM Neon accelerated operations

ARM Neon SIMD operations are currently not supported but may be added in the future. However, by using the vector-types, the back-end compiler may generate code relying on SIMD instructions.

12.5 Automatic alignment

To maximally benefit from SIMD operations, it is essential that the memory load operations are aligned (which means that the memory address modulo a given constant is zero, this constant is often equal to the cache line size, e.g., 16 bytes).
The Quasar runtime ensures that all vectors and matrices have aligned memory addresses. However, this is not sufficient to ensure that all memory loads are aligned, for example the memory load A[n∗4+(0..3)+1] can never be aligned. In this case, an unaligned read will be generated which may reduce the performance.

12.6 Automatic SIMD code generation

There are already several situations in which the Quasar compiler automatically generates SIMD code:

1. when fixed-length vectors are used, where the length is suitable chosen (see *supported vector lengths* for CPU and CUDA above).
2. during expression optimization (see section §8.1).
3. in code using shared memory designators (see 9.2).

In the future, the compiler may rely more and more on SIMD code generation when it is appropriate. For now, SIMD processing can be activated by enabling the `simdprocessing` transform.

Chapter 13

Best practices

13.1 Use "main" functions

Quasar programs are executed from the top to the bottom. This means that, if there are statement in between function definitions, these statements will also be executed. This can be handy to define symbols at the global level, such as constants, lambda expressions etc. However, it is advisable to put the main program logic in one function, the function "`main`". The function "`main`" will be called automatically by the runtime when the .q file is loaded. An example of a main function is as follows:

```
function [] = main()
    img = imread("lena_big.tif")
    imshow(img)
endfunction
```

The main function may contain fixed and optional parameters:

```
function [] = main(required_param1, opt_param1=4.0)
```

The required parameters must then be specified via the command-line (or via the set command line arguments dialog box in Redshift). For example:

```
Quasar.exe myprog.q 1 2
```

When not enough parameters are specified (or too many), a run-time error will be generated. Practically, there are only two types that are allowed: `scalar` and `string`. In other to pass values of other types (e.g. matrices), it is currently best to wrap them in a string, and to convert the string to the right data type using the function `eval`. The following example illustrates this:

```
function [] = main(matrix_string : string)
    matrix : mat = eval(matrix_string)
    print matrix
endfunction
% Command line
```

```
Quasar.exe "[[1,0],[1,-1]]"
% Runtime system calls the main function as:
main("[[1,0],[1,-1]]")
```

Additionally, the main function can be made to accept a variable number of parameters, by defining it as a variadic function (see section §4.8):

```
function [] = main(arg1, arg2, ...other_args)
    print arg1
    print arg2
    for i=0..numel(other_args)-1
        print other_args[i]
    endfor
endfunction
```

This permits great flexibility when passing various parameters to Quasar programs.
Important remark: function "main" has a special behavior when the .q file is imported (using the *import* keyword, see earlier): in particular, the function definition is completely skipped, as if no "main" function was present in the file. Hence, for .q modules that are only intended to be imported, the "main" function can contain some testing code.

13.2 Shared memory usage

Shared memory (see section 2.4.4) is on-chip and fast, however, for the CUDA computation engine, recent GPU devices use a global memory cache that has about the same efficiency as the shared memory. Consequently, the best practice is to only use shared memory when it is *needed*, for example when there is communication needed between the different kernel functions that are running in parallel on the same block. The reason is: copying from global memory to shared memory also has a performance cost, and because shared memory is limited, kernel functions often need to be restructured so that everything can fit into the shared memory. This "restructuring cost" often outweighs the benefits of using shared memory. So only use shared memory when it is really necessary.

13.3 Loop parallelization

Often, you may want to parallelize nested loops, such as:

```
for m=0..M-1
    for n=0..N-1
        for k=0..K-1
            for l=0..L-1
                ...
            endfor
        endfor
    endfor
endfor
```

The question is: which loops to parallelize? The answer is actually problem-specific (depends on the dimensions of the variables and their dependencies), but in general, it is recommended to parallelize the outer loops as much as possible, because this minimizes communication and synchronization with the computing device (e.g. GPU). For example, the above loops would be best parallelized as follows:


```
function [] = __kernel__ my_kernel(...)
   for l=0..L-1
      ...
   endfor
endfunction
parallel_do([M,N,K],...,my_kernel)
```

However, in many cases it is not necessary to perform this parallelization yourself: the Quasar compiler has an efficient built-in auto parallelization routine, which checks variables and their dependencies, and chooses a parallelization strategy that has the most benefit for the particular problem. For more info, see section 17.1.1.
Using the `{!parallel for; dim=N}` code attribute, it is also possible how many of the outer for-loops need to be parallelized. Special care needs to be taken when `N` is selected. When N is too small, the kernel may end up being executed on the CPU because the data dimensions of the parallel loops are too small to match well with the GPU requirements. See also section §8.2.

13.4 Output arguments

Functions can have multiple arguments, as shown in the following example:

```
function [band1 : mat, band2 : mat]=subband_decomposition(input : mat)
    band1 = input .* G
    band2 = output .* H
endfunction
```

Alternatively, the matrices are passed by reference (see Consequently, the best practice is to only use shared memory when it is *needed*, for example when there is communication needed between the different kernel functions that are running in parallel on the same block. The reason is: copying from global memory to shared memory also has a performance cost, and because shared memory is limited, kernel functions often need to be restructured so that everything can fit into the shared memory. This "restructuring cost" often outweighs the benefits of using shared memory. So only use shared memory when it is really necessary.section 1.3), and this can also be exploited for returning processing results:

```
band1 = input % copy reference
band2 = zeros(size(input))
function [] = subband_decomposition(band1 : mat, band2 : mat)
    band2[:,:] = band1 .* G
    band1 = band1 .* H
endfunction
```

It is preferable to use the first approach (for readability of the code), however the second approach is also useful in some cases: the difference is in the memory usage: in the first approach: memory needs to be allocated for `input`, `band1` and `band2`, while in the second approach, only memory is needed to store `band1` and `band2` (hence one memory allocation is eliminated). For applications relying on huge matrix sizes (for example applications working with digital camera images, or for real-time video applications), it is recommended to use the second approach.
Remark that simply using "`band2 = band1 .* G`" in the second approach would not give the correct result, because, even though `band2` contains a pointer to the matrix memory, the value of `band2` itself is still passed by value. Instead, adding `[:,:]` ensures that no new memory is allocated for `band2`.
In case after a call, one output argument is not necessary in the subsequent code, the output argument can be captured using a placeholder:


```
[band1,_] = subband_decomposition(input)
```

This way, in future versions, the compiler may optionally specialize the function `subband_decomposition`, by generating a version in which the second output parameter is not being calculated.
As mentioned in section 4.8.3, the output arguments of functions can be chained using the spread operator `...` For example,

```
function [band1_out, band2_out] = process(band1, band2)
    ...
endfunction
process(...subband_decomposition(input))
```

This way, it becomes unnecessary to store the output arguments in intermediate variables.

13.5 Writing numerically stable programs

Here, we consider numerical stability and software program stability. To ensure *numerical stability*, programs may need to make use of the following functions:

- `isfinite(x)`: checks whether variable x is finite (i.e. not infinite and not NaN "not a number").
- `isinf(x)`: returns true only if the variable x is infinite. Infinities are used to represent overflow and divide-by-zero.
- `isnan(x)`: returns true only if the variable x is "not a number". The NaN encoded floating point numbers have no numerical value. They are produced by operations that have no meaningful result, like infinity minus infinity.

Remark that, by default, GPU computation engines, flush denormal floating point values to 0. Practically, this means that if the result of a single-precision floating-point operation, before rounding, is in the range -2^{-126} to $+2^{-126}$ (or $-1,175\times10^{-38}$ -1,175$\times10^{-38}$), it is replaced by 0 (also see section 2.2.1). To avoid potential underflows, it maybe necessary to pre-scale the input data to a good "working" range, before numerical operations are performed. CPU computation engines may allow for denormal numbers (depending on the setting of the compiler, and whether SIMD instructions are used etc.), yielding more accurate numerical results, but at a decreased performance: working with denormal floating point numbers can be up to 100 times slower than in case of normalized numbers. Hence, in case numerical problems are an issue, it may be good to compare the results of the CPU and GPU computation engines.
Software stability: there are four causes for a Quasar program to be interrupted:

1. *Errors* (generated using the `error` statement or by Quasar runtime functions). Currently, error handling (e.g. try-catch blocks) are not supported yet. Hence when an error is generated, the program is automatically terminated.
2. *Out-of-memory*: when the system (or GPU) has not enough memory, the program will be halted. By default, Quasar attempts to move memory from the GPU to the system memory when it detects that a memory allocation may result in an out-of-memory error. In some cases, this may not be possible (e.g., a `__kernel__` function that uses more memory than available on the GPU).
3. *Stack overflow*: usually when a recursive function calls itself in an endless loop. For example, the function:


```
f = x -> f(x)
```

will result in a stack overflow error.

4. *Abusing 'unchecked modifiers*: the `'unchecked` modifier (see section 2.4.1) is introduced for memory accesses where it is completely certain that a kernel function will not go out of bounds of the vectors/matrices/etc. This gives a performance benefit of up to 30% or more for certain functions. When the kernel function breaches the boundaries, the program may either result an error (e.g. `cudaUnknownError`), or crash. To prevent this kind of problems, one can 1) either remove the `'unchecked` modifiers from the kernel function arguments, or 2) run the program using the CPU computation engine, with the flag `COMPILER_PERFORM_BOUNDSCHECKS=true` (see table 17.3). In the second case, Quasar will report an error and some information on the variables that violate the boundary conditions, so that abuses of the `'unchecked` modifier can be fixed.
   An alternative solution is to temporarily replace `'unchecked` by `'checked`, this will instruct Quasar to perform bounds checking at any time for the specified variable, irrespective of the `COMPILER_PERFORM_BOUNDSCHECKS` variable.

To catch errors, it may be useful to place assertions inside kernel or device functions:

```
function [] = __kernel__ kernel (pos : ivec3)
    b = 2
    assert(b==3)
endfunction
```

In this example, the assertion obviously fails. Quasar breaks with the following error message:

```
(parallel_do) testkernel - assertion failed: line 23
```

13.6 Writing deterministic kernels

When writing kernels involving atomic operations, the outcome may depend on the order of the instructions being executed. This is due to the floating point arithmetic often not being associative: in general $(A + B) + C! = A + (B + C)$. Due to the parallelism, the order of the instructions cannot be controlled. The nondeterministic nature may not be a desired property of the algorithm. The following parallel reduction implementation exhibits this problem.

```
function y : scalar = __kernel__ parallel_reduction(x : vec'unchecked,
     acc : accumulator, val0 : scalar, pos : int, blkpos : int, blkdim : int)
    val = val0
    bins = shared(blkdim)
    for m=pos..32*blkdim..numel(x)-1 % step 1 - parallel sum
        val = acc(val, x[m])
    endfor
    bins[blkpos] = val
    syncthreads % step 2 - reduction
    bit = 1
    while bit < blkdim
        index = 2*bit*blkpos
        if index + bit < blkdim
            bins[index] = acc(bins[index], bins[index + bit])
        endif
```



```
        syncthreads
        bit *= 2
    endwhile
    % write output
    if blkpos == 0
        y += bins[0]
    endif
endfunction
```

The problem is here that `y += bins[0]` is an atomic update operation. It is performed only once per block, however multiple blocks may interfere causing the order of the addition operations to be altered. The solution is here to avoid the atomic update altogether, and store the result per block in an output vector (with as length the number of blocks in the input vector). Then, in a separate kernel launch of `parallel_reduction`, the block sums are summed separately. To reduce the size of the blocksums vector, grid-strided loops can be used (see section §9.2.3.1).
Thanks to the specific way that updates are performed in shared memory and due to the thread synchronization, the updates of `bins` are guaranteed to be deterministic! This shows that it is possible to remedy a kernel function and ensure that its output is deterministic.
When accumulation in shared memory with thread synchronization (or similarly, warp shuffling) is not one of the options for a specific kernel, an alternative is to use integer or fixedpoint representations. This way, we have successfully solved some depth map determinism problems during depth map calculation in computer vision applications.

Chapter 14

Built-in function quick reference

Some built-in functions are listed in table 14.1. More runtime library functions are given in table 14.2. For a detailed explanation of the functions, we refer to the Documentation Browser (F1 in Redshift).

Table 14.1: Common built-in functions (most functions are self-explanatory). Functions with asterisk (*) are accessible from `__kernel__` and `__device__` functions.

abs (*)	absolute value/modulus	sum (*)	sum of the elements
acos (*)		cumsum	cumulative sum
atan (*)		prod	product of the elements
atan2 (*)		cumprod	cumulative product
ceil (*)		mean	
round (*)		linspace	
cos (*)		lerp (*)	linear interpolation
sin (*)		dotprod (*)	vector dot product
exp (*)		zeros (*)	
exp2 (*)	power of two	ones (*)	
floor (*)		rand	uniformly distributed
mod (*)	modulo	randn	normal distributed
frac (*)	fractional part	cell	cell matrix
log (*)		eye	identity matrix
log2 (*)	logarithm base 2	size (*)	dimensions of object
log10 (*)	logarithm base 10	numel (*)	number of elements =`prod(size(x))`
max (*)		complex (*)	complex value
min (*)		real (*)	real part
saturate (*)	clamps to [0,1]	imag (*)	imaginary part
sign (*)	sign of the number	float (*)	conversion to float (kernel function)
sqrt (*)		int (*)	take integer part
tan (*)		isnan (*)	value is NaN (not a number)
angle (*)	angle of a complex number	isinf (*)	value is infinite
transpose	matrix transpose	isfinite (*)	value is finite
herm_transpose	Hermitian transpose	maxvalue (*)	maximum value for the specified type
conj (*)	conjugate	minvalue (*)	minimum value for the specified type
copy	performs a shallow copy	repmat	repeat matrix
deepcopy	performs a deep copy	reshape	reshape matrix
squeeze	removes singleton dimensions	shuffledims	swaps dimensions
fft1 / ifft1	1-dimensional (I)FFT	type	returns data type of object
fft2 / ifft2	2-dimensional (I)FFT	object	creates an empty structure
fft3 / ifft3	3-dimensional (I)FFT	sprintf	build a C-style format string
shared (*)	allocation of shared mem.	printf	print a C-style format string
shared_zeros (*)	shared mem with zero init.	strcat	string concatenation
and (*)	bitwise AND	sscanf	parses using a C-style format string
or (*)	bitwise OR	factorial	the factorial function
xor (*)	bitwise XOR	inv	matrix inverse
shl (*)	bitwise left shift	svd	singular value decomposition
shr (*)	bitwise right shift	serial_do	serial execution
not (*)	bitwise inversion	parallel_do	parallel execution
mirror_ext (*)	mirroring extension	max_block_size	see section 2.4.4
periodize (*)	periodic extension	assert	runtime assertion
tounicode	converts `vec` to a UNICODE string	schedule	manual run-time scheduling function
toascii	converts `vec` to an ASCII string	mat2cell	converts from matrix to cell matrix
fromunicode	converts UNICODE string to `vec`	cell2mat	converts from cell matrix to matrix
fromascii	converts ASCII string to `vec`		
ind2pos	converts linear index to n-D coords		

imread	reads an image	`img=imread("filename.png")`
imwrite	writes an image	`imwrite("filename.png", data)` `imwrite("filename.png", data, [minval,maxval])`
imshow	shows an image	`imshow(img)` `imshow(img, [minval,maxval])`
eval	Quasar expression evaluation	`y=eval("x->2*cos(x)")`
save	Save variables to file	`save("out.dat",A,B,C)`
load	Load variables from file	`[A,B,C]=load("out.dat")`
dir	Lists files in a directory	`files=dir("/home/*.png")`
tic	start timer	`tic()`
toc	stop timer and print elapsed time	`toc()`
fopen	opens file for reading/writing	`f=fopen("out.dat","wb")`
fread	reads from a file	`data=fread(f,[24,8],"float32")`
fwrite	writes to a file	`fwrite(f,data,"float32")`
fclose	closes a file	`fclose(f)`
fgets	reads one line in text modus from a file	`y=fgets(f)`
plot	generates a plot	`plot(x,y)`
title	set title of the plot	`title("text")`
xlim	sets ranges for the x-axis	`xlim([0, 10])`
ylim	sets ranges for the y-axis	`ylim([-pi, pi])`
xlabel	sets the x-axis label	`xlabel("x")`
ylabel	sets the y-axis label	`ylabel("y")`
legend	displays a legend	`legend("serie 1", "serie 2")`
disp	display a matrix	`disp(A)`
print	print text to the console	`print A,...`
error	generate an error	`error A,...`
pause	pauses program execution for n msec.	`pause(0.5)`

Table 14.2: Runtime library functions.

Chapter 15

Functional image processing in Quasar

In this section we will highlight some functional image processing abilities of Quasar. Through the combination of generics, automatic function inlining and avoidance of function pointers, Quasar permits practical functional programming of image processing algorithms, while at the same time exploiting the parallelism and generating efficient CPU/GPU code. As a starting point, we will consider all values (e.g. scalars, integers, vectors) to be application of functions. These functions can be constant functions, functions with no parameters etc. Rather than passing values to functions, functions themselves are used as parameters for other functions. This allows the caller to choose whether a constant or a non-constant input is used. Moreover, intermediate results can be calculated only when needed, rather than precalculated. Additionally, in some cases, it is more efficient to evaluate functions more than once (e.g., because the memory cost involved with storing the intermediate results is much higher than the calculation time of these results).
The concept of a (2D) image can be made explicit by a function type definition:

```
type image : [__device__ ivec2'const -> vec3]
```

Here we say that an image is a function that maps from a two-dimensional integer (`ivec2`) domain to a three-dimensional (say, RGB) domain. Note that the types `ivec2` and `vec3` are short-hand notation for `vec[int](2)` and `vec[scalar](3)`.
Similarly, the concept of an algorithm can be defined as:

```
type algorithm : [__device__ (image, ivec2'const) -> vec3]
```

The algorithm takes an image and a 2D position as an input and calculates the components of the output at that position. Note that it would be more easy to define

```
type algorithm : [__device__ (image) -> image]
```

, which is also viable. However, the algorithm definition incorporating a position vector allows to separate the computational approach (e.g., pixel-based parallel, block-based parallel, etc.) from the algorithm definition. We can then design additional infrastructure for this task, by the following abstraction:


```
type application : [(image, algorithm) -> image]
```

This can be read as: an application uses an algorithm to convert an image to another image.

```
type color_function : [__device__ vec3 -> vec3]
```

A rasterizer samples an image and converts it back to a cube:

```
type rasterizer : [(image, ivec3) -> cube]
```

The function `make_image` converts an image (of type `cube`) into a function variable (of type `image`):

```
make_image : [cube -> image] =
    im_data -> __device__ (pos) -> im_data[pos[0], pos[1], 0..2]
```

By declaring the resulting function as a device function (`__device__`) we can ensure that native device-specific code is generated for the function. But there is more: the Quasar compiler performs automatic function inlining and function pointer reduction, so that there is often no performance loss experienced due to the use of function variables. This allows algorithms to be specified in a flexible and generic way.
We can define a 'casting' operator to call the function `make_image` directly.

```
reduction (x : cube) -> image(x) = make_image(x)
```

Similarly, we can write a function to perform a point-wise operation on an image:

```
pointwise_op : [color_function -> algorithm] =
    fn -> __device__ (x, pos) -> fn(x(pos))
```

The pointwise operation will apply a given color processing function (`color_function`) to every pixel in the image. Similarly, we can define an operation that translates the image:

```
translate : [vec2 -> algorithm] =
    shift -> __device__ (x, pos) -> x(pos + int(shift))
```

The `translate` function takes a 2-dimensional offset vector and returns an `algorithm` (which can then be applied to the image). In functional programming, lazy evaluation is preferred over eager evaluation, and therefore we will "concatenate" several image processing algorithms in a chain, and at the very end perform the computation. In this way, the Quasar compiler will obtain several degrees of freedom in optimizing the code. For example, it is not necessary to store intermediate results in memory.
As a next algorithm, we define a horizontal mean filter (with as input parameter the size of the local window):

```
function algo : algorithm = mean_filter_hor(N : int)
    norm_factor = 1.0 / N % normalization factor
    function y : vec3 = __device__ algo(input : image, pos : ivec2)
        y = [0., 0., 0.]
        for n=-int(N/2)..int(N/2)
            y = y + input(pos + [0,n])
        endfor
        y = y * norm_factor
    endfunction
endfunction
```


Together with the corresponding vertical mean filter:

```
function algo : algorithm = mean_filter_ver(N : int)
    norm_factor = 1.0 / N % normalization factor
    function y : vec3 = __device__ algo(input : image, pos : ivec2)
        y = [0., 0., 0.]
        for n=-int(N/2)..int(N/2)
            y = y + input(pos + [n,0])
        endfor
        y = y * norm_factor
    endfunction
endfunction
```

we can then easily calculate mean filters with `N x N`-masks, because the mean filters are separable. Next, the `compute` function converts the image back to its pixel representation (also called rasterization):

```
compute : rasterizer =
    (x, sz) -> (y = zeros(sz); parallel_do(sz, __kernel__ (pos : ivec2) -> y[pos[0],pos[1],0..2] = x(
        pos)); y)
```

Essentially, the function evaluates the input function for all positions within the domain.
Next, we define two operations that apply a specified processing algorithm to an image. In the first algorithm, we "defer" the processing, which means that the processing is only applied when the compute function is applied.

```
apply_deferred : application =
    (x, p) -> __device__ (pos) -> p(x, pos)
```

In a second operation, we calculate the intermediate result:

```
apply_immediate : [vec3 -> application] =
    sz -> (x, p) -> make_image(compute(apply_deferred(x, p), sz)
```

Then, our goal is to be able to subsequently apply several algorithms. To simplify the calculation of the end result, we define the definition-assignment operator `:=`

```
reduction (y : cube, x : image) -> (y := x) = y = compute(x, size(y))
```

This assignment will lead to the result being evaluated, resulting in a variable of type `cube`. Finally, to concatenate several independent operations, we define the pipelining operator `|>`:

```
symbolic immediate, deferred
reduction (x : cube, y) -> (x |> y) = make_image(x) |> y
reduction (x : image, a : algorithm, sz : ivec3) -> (x |> immediate(sz, a)) = apply_immediate(sz)(x, a
    )
reduction (x : image, a : algorithm) -> (x |> deferred(a)) = apply_deferred(x, a)
```

Here, symbolic introduces some symbols that will be defined later using reductions.

15.1 Example: tranlation and filtering

As an example, we demonstrate the functional image processing framework in applying a translation and a mean filter to an input image. The algorithm specification is then:


```
im = imread("image.tif")
im_out := im |> deferred(translate([64, 64])) _
           |> deferred(mean_filter_ver(7)) _
           |> mean_filter_hor(7)
imshow (im_out)
```

This approach allows to easily describe image processing pipelines, in which different filtering/processing building blocks are connected to each other. Additionally, it allows the Quasar compiler to optimize the entire pipeline *at once*.

Chapter 16

The Quasar runtime system

The Quasar environment contains both a compiler (see 17) and a runtime system. The runtime system manages the computation devices and also makes sure that Quasar code is efficiently executed on the selected/available devices. Note that the compiler and runtime system are not entirely separated: the runtime system exploits information (metadata) generated at compile-time, e.g., to improve scheduling options.
Most data structures (e.g., primitive types, user-defined types, functions, arrays, matrices, ...) support automatic transfer to the GPU. For the run-time system, this implies some "bookkeeping" (overhead) at runtime. For example, to keep track of the most "recent" version of a data structure, both a CPU pointer and a GPU pointer may be required, plus a set of dirty bits.
Two versions of the runtime system exist and are both in use

- Standard engine (v1 runtime): offers single GPU CUDA and OpenMP CPU support.
- Hyperion engine (v2 runtime): offers OpenCL, multi-GPU CUDA/OpenCL and two-layer threaded OpenMP CPU support. For the Hyperion engine, most built-in core routines (such as `sum`, `prod` functions and implementation of matrix operators) are implemented in Quasar itself. The Hyperion engine is therefore easily extensible to future accelerator devices. The Hyperion engine supports multi-GPU and allows combining OpenCL GPUs with CUDA GPUs (for example on NVidia Optimus laptops, Intel HD Graphics with the NVidia Geforce GPU).

Compared to the Standard engine, the Hyperion engine has a more efficient object graph management system, allowing clustering objects into subgraphs which allow simultaneous transfers of objects to the GPU, with better management and support of GPU memory models (pinned memory, unified memory etc).
Apart from implementation differences, there are no functional differences between the two runtime engines, i.e., a program developed on runtime v1 works on runtime v2 (and vice versa).
The runtime system can be selected through the device selector, for example in the Redshift IDE. Hyperion devices are listed explicitly as *v2*, e.g. CUDA v2, OpenCL v2, ...
In general, the runtime system performs several tasks:

- Program interpretation and execution
- Providing an abstraction layer for computation device (e.g., CUDA, OpenCL, OpenMP)

- Performing object management (reference counting, grouping memory transfers etc.)
- Performing memory management (allocation, transfer from CPU ->device, device ->CPU, memory pinning)
- Load balancing/runtime scheduling of parallel tasks

The above components will now be discussed in more detail in the following subsections.

16.1 Program interpretation and execution

A Quasar program consists of a sequence of high-level "instructions", for which an instruction often involves an expression evaluation. These instructions are either compiled to .Net byte-code or interpreted by the Quasar interpreter.
The actual time-consuming calculations are performed by invoking kernel or device functions. In particular, a kernel function is a function that acts on one element of the data, and that is repeated (in parallel) to all other elements. Consider the point-wise multiplication of matrices A .* B. Then a kernel function computes the product of two elements at a given position 'pos': `Y[pos] = A[pos] * B[pos]`. This kernel function is internally defined in Quasar as follows:

```
pw_multiply = __kernel__ (A:mat,B:mat,Y:mat,pos:ivec2) -> Y[pos]=A[pos]*B[pos]
```

Now, when the interpreter asks the computation engine to calculate the product of two matrices, the kernel function 'pw_multiply' is scheduled. The same occurs for the assignment `Y[pos]=...`: this operation is also performed in a kernel function. Eventually, the runtime system obtains a "stream" of kernel function invocations:

```
kernel1 -> kernel2 -> kernel3
```

The runtime system is then responsible for launching these kernels on the data.

16.2 Abstraction layer for computation devices

The run-time system has several back-ends: for CUDA, OpenCL and OpenMP. For every kernel function, the Quasar compiler typically generates code for different targets. However, the high-level instructions of a Quasar program are target-independent (even when compiled to .Net bytecode), which allows a Quasar program to be easily retargetted during runtime to other devices.
There are a set of run-time parameters that allow controlling the different run-time back-ends:

- *Enable concurrent kernel execution (CKE).* This mode has specific meaning depending on the back-end:
  - For CUDA back-ends, the CKE mode uses CUDA streams and the CUDA asynchronous programming model. The streaming is performed completely automatically by the run-time system (by tracking dependencies). When necessary, inter-stream event synchronization is also performed.
  - For OpenCL back-ends, CKE relies on the OpenCL event system to specify that kernel functions that need to executed in parallel.
  - For CPU back-ends (Runtime v1), the CKE mode has no effect.
  - For CPU back-ends (Runtime v2 - hyperion engine), the CKE mode causes kernel functions to be executed in different CPU threads, in a two-level concurrency model: kernel functions are executed in parallel on different threads, however, each invididual kernel function may use a fixed number of OpenMP threads.

For example, for an 8-core processor with hyperthreading enabled, 2 kernel functions can be executed in parallel, running the calculations each on 4 OpenMP threads, enabling efficient use of the CPU. These configuration parameters can be set in the hyperion device configuration file.

- *Enable CUDA pinned host memory*: if enabled, pinned host memory is used automatically for critical memory transfers.
- *Enable CUDA 16-bit floating point textures*: the use of 16-bit floating point textures reduces storage space. Note that this setting requires the use of `hwtex_*` modifier (See section §9.4).
- *Enable OpenMP multi-threading with N CPU threads*: enables OpenMP multi-threading on multicore processors, with the specified number of OpenMP threads
- *GPU scheduling mode*: controls the runtime behavior in circumstances of GPU sharing and/or high memory usage. The following modes are available:
  - *Maximize Stability*: in this mode, certain operations may be slowed down when the GPU is running out of memory (e.g. by automatically copying memory back to the CPU in order)
  - *Maximize Performance*: in this mode, the slow down is prevented and instead a run-time error is generated
- *GPU memory model*: controls how aggressively the runtime allocates memory. The following options are available.
  - *Small footprint*: the run-time memory manager is extremely conservative in memory allocations
  - *Medium footprint*: default mode for the memory manager
  - *Large footprint*: allocates device memory aggressively, leaving limited GPU memory to other processes. Use this mode when you intend to allocate many very large memory blocks.
- *Default image codec provider*: selects between GDI+ and GTK+ for coding and decoding images. GDI+ and GTK+ support different sets of image codecs. Some image formats are better supported by GDI+, other formats are better supported by GTK+ (using libjpeg, libtiff, libpng etc).
- *Use OpenGL for visualization*: when enabled, all rendering is accelerated using OpenGL. Note that OpenGL is currently not enabled in most terminal sessions (e.g. Windows remote desktop, Linux SSH -X)
- *OpenGL anti-aliasing mode*: specifies the anti-aliasing mode to be used for the OpenGL context (mainly useful when rendering lines and points).

16.3 Object management

Quasar has a custom object system to represent the internal data. There are different built-in object types:

- Scalar types (scalar, int)
- Matrix types (real-valued, complex-valued)
- Function types (functions are first-class variables and can also store data through function closures).
- Cell matrix types: a structured way for storing variables of all other types (similar to Matlab/Octave cell matrices)

- Untyped objects: user-defined data types, useful for rapid prototyping
- Typed objects: user-defined data types, same but more powerful
- String types: limited support currently.

Objects can contain references to other objects. Circular references are also allowed (although the handling of them is not fully implemented at this time). An important feature is that all object types are transparently accessible from all computation devices. For example, in contrast to many other GPU libraries, Quasar makes no distinction between CPU data types and GPU data types. For efficiency, restrictions apply, for example untyped objects cannot be accessed from the device because runtime type checking would cause a significant amount of overhead. Instead, the runtime is able to decide whether a given object should reside in CPU or GPU memory or both. This means that data needs to be transferred from/to the device without the user even knowing it (of course this information is still available through profiling - and the run-time system can be controlled programatically, see section 16.7). The object and memory management systems are responsible for these transfers. The object system then has to track the references between the objects. Typically, when one object is accessed in device memory, all objects that are referenced also need to be copied to the same device memory. To make sure that this is performed efficiently, the objects are "clustered" so that the complete object graph can be transferred in one memory copy operation.

16.4 Memory management

When executing kernels on a given data set, the operands of the kernel function may or not reside in the correct memory for the device (e.g., GPU memory). This requires *memory management*.
The memory manager is responsible for allocating memory in device/CPU memory and for transferring data from the CPU to the device. When the device is running out of memory, it may also be necessary to transfer data back from the device to the CPU.
The runtime scheduler then has to ensure that 1) the memory manager is doing the correct task (so that all required data is in the correct memory before a given kernel is launched) and 2) that the kernels are properly executed according to the data dependencies.
Both tasks of the runtime scheduler actually interfere - for example, it is possible that the CPU requests to release memory that is still used in one of the asynchronously running kernels. The same applies to data that needs to be transferred from or to the GPU. The runtime scheduler then has to ensure that both the CPU and the different kernels have a consistent view on the data.
Therefore, Quasar has a distributed memory system: an object may reside in the memory of different computation devices at the same time. As long as read accesses are considered, this does not pose any problems on its own. The main difficulties are rather: 1) dealing with memory allocation requests when there is not sufficient device memory available, 2) avoiding memory fragmentation and 3) making sure that no memory blocks are released that are still in use by an asynchronous kernel.
Different memory allocation algorithms are automatically used by the runtime system. The CPU engine uses a garbage collector with 3 generations (short term, mid term and long term). Because of the (relatively) scarce GPU device memory, the CUDA computation engine has a new memory allocation algorithm that relying on reference counting instead of garbage collection. This allocation algorithm is also optimized for middle (>1024) to large (>100MB) data allocations.
In some cases, the device may run out of memory. If this happens, some (idle) memory blocks are selected to be transferred back to the CPU memory. Therefore, the least-recently-used (LRU) strategy is adopted (taking into account which memory blocks are currently in use by kernels) and an optimization takes place to minimize the amount of memory that needs to be copied to the CPU.

However, in some cases, the memory fragmentation and size of the working set for a kernel are too large so that moving memory becomes impractical. In this case, an error report is currently displayed:

```
Amount of pinned memory: 398,131,200 bytes
Freelist size: 6 memory blocks
Largest free block: 67,665,920 bytes, insufficient to hold request size of 132,710,400 bytes
Process total: 736,493,568, Inuse: 580,608,000 bytes, Free: 155,885,568 bytes;
Device total: 1,010,368,512, device free: 167,772,160
Chunk 0 size 67,108,864 bytes: Fragmentation: 47.8%, pinned: 0 bytes (0.0%), free: 17,342,464 bytes
    (25.0%) Chunk 1 size 134,217,728 bytes: Fragmentation: 0.0%, pinned: 132,710,400 bytes (98.0%),
    free: 1,507,328 bytes (1.0%)
Chunk 2 size 268,435,456 bytes: Fragmentation: 0.0%, pinned: 132,710,400 bytes (49.0%), free:
    36,192,256 bytes (13.0%)
Chunk 3 size 266,731,520 bytes: Fragmentation: 32.9%, pinned: 132,710,400 bytes (49.0%), free:
    100,843,520 bytes (37.0%)
```

The solution is then to modify the Quasar program such that smaller (or less) memory blocks are used (for example, by splitting images in smaller tiles). An alternative is to change the memory allocation model to "huge memory footprint". This way, Quasar will allocate large memory chunks and the possibility that a block does not fit in the chunks is decreased.

16.5 Load balancing and runtime scheduling

The runtime scheduler defines the order of the kernel launches and whether kernels can be executed concurrently. Therefore, the scheduler also tracks the different data dependencies between subsequent kernels. At the core of the runtime scheduler is the command queue. The command queue is a first-in first-out queue that either contains an individual kernel launch or a memory transfer operation. When kernel and device functions are scheduled, load balancing is performed an an appropriate computation device is selected based on several parameters.
The runtime scheduler aims at distributing the kernel functions over the available computation devices, taking full advantage of their possibilities.

16.6 Optimizing memory transfers with const and nocopy

To suppress the need to unnecessary memory copies, kernel and device function parameters can be annotated with the modifiers `'const` and `'nocopy`. The meaning of these modifiers is as follows:

- `'const`: Indicates that the parameter is constant and will not be changed inside the kernel function.
- `'nocopy`: Indicates that the input values of the parameter (for example of a vector or matrix type) are not used. This is useful when implementing functions that completely overwrite the content of a vector/matrix without reading the original values of this vector/matrx.

The `'const` avoids that, after execution of the kernel/device function, function parameters are being copied back to the host (or equivalently, to another GPU). On the other hand, `'nocopy` avoids that the memory is being copied from the host to the device (or equivalently between GPUs).
The compiler automatically detects parameters that can be annotated with `'const` and `'nocopy`, so therefore there is no direct need to use these modifiers in user code. However, the modifiers may still be of use for indicating the calling interface of the function: the calling function then knows that e.g. the given function does not change the value of one of the parameters, or that the input value of one of the parameters is not used directly.

Finally, the reader may have noted that it could also be useful to have a nocopy at input and output modifier; this could serve as a local "scratch memory" to be used by the kernel function (e.g., as used in the implementation of various linear algebra routines). In Quasar, there is no special modifier for this purpose, it suffices to not declare any access modifier - as long as the scratchpad memory will not be used by another function, the memory transfer will not be performed.

16.7 Controlling the runtime system programmatically

Sometimes it is useful to control the runtime system from Quasar code. This can be achieved using the following code attributes:

- `{!sched mode=cpu|gpu|fpga}`: sets the scheduling mode manually, instructing the scheduler to run the following kernels on the specified device.
- `{!sched gpu_index=1}`: sets the GPU index (within the range `0..num_gpus-1`) for the next kernel function. This feature is useful for multi-GPU programming (see section §11).
- `{!sched}`: switches back to automatic scheduling
- `{!alloc mode=auto|cpu|gpu}`: sets the memory allocation mode for the next allocation request.
- `{!alloc type=auto|pinnedhostmem|unifiedmem|texturemem}`: sets the allocation type for the next allocation request. This function has only effect when the allocation mode is GPU. Pinned host memory is memory that avoids paging and that can more efficiently been transferred to/from the GPU. Unified memory is automatically page-mapped between CPU and GPU (requires CUDA 6.0 or higher). Texture memory (to be used with access modifiers `'hwtex_*` (see section 9.3), often allows more efficient data access patterns.
- `{!transfer vars=a; target=cpu|gpu}`: instructs the runtime system to transfer the specified variable to the CPU (or GPU). vars may refer to an expression (for example, `a[index]`, in which case the specified element of the cell array is transferred).
- `{!transfer vars=a; target=cpu|gpu; gpu_index=1}`: instructs the runtime system to transfer the specified variable to the GPU with the specified index.

Note that the above functions are provided for fine-tuning implementations. However, the code attributes should be used with care, because inappropriate usage may degrade the performance and cause unexpected errors in some cases. Several aspects, such as the memory location of a variable, can be visualized and analyzed in the variables window in the Redshift IDE.

Chapter 17

The Quasar compiler/optimizer

An overview of the Quasar compilation process is given in the figure below. After parsing, type inference is performed in order to determine the type of all the variables and expressions in the program. Next, a function transform pass is performed, with several optimization passes (see section 17.1). Then, kernel and device functions are processed using a kernel transform pipeline which allows the generation of target-specific code and which enables target-specific optimizations (either built-in or user-controlled). After the kernel transformations, the generated target-specific code is translated to intermediate C++, CUDA, OpenCL or LLVM code, in order to be compiled using one of the back-end compilers. The output of the function transform pass is intermediate Quasar code that can further be translated to CIL bytecodes (resulting in a standalone executable that can executed by MONO or .Net).
The function and kernel transforms will be discussed in the following sections in more detail.

17.1 Function Transforms

The Quasar compiler supports several function transforms:

- *Automatic loop parallelization/serialization*: performs automatic parallelization or serialization of loops within a function. Both for-loops and while-loops can be handled.
- *Automatic kernel generator*: vectorizes matrix expressions and generates corresponding kernel functions.
- *Function optimization*: performs 'generic' function optimization steps, such as constant folding, index packing, loop unrolling and elimination of dead branches etc.
- *Function data transfer optimizer*: annotates arguments of functions in such a way that the run-time system has less work (e.g., less memory copies required). Typically, when a matrix is only read from, its dirty bit does not need to be changed (possibly reducing the required number of data transfers)
- *Boundary accessor transform*: adds boundary accessor functions (see above) for newly generated array indexers
- *Lambda to regular function converter*: converts lambda expressions to regular functions that can be further optimized using the kernel transform pipeline, allowing the lambda expressions to be treated and optimized in the same way as regular functions

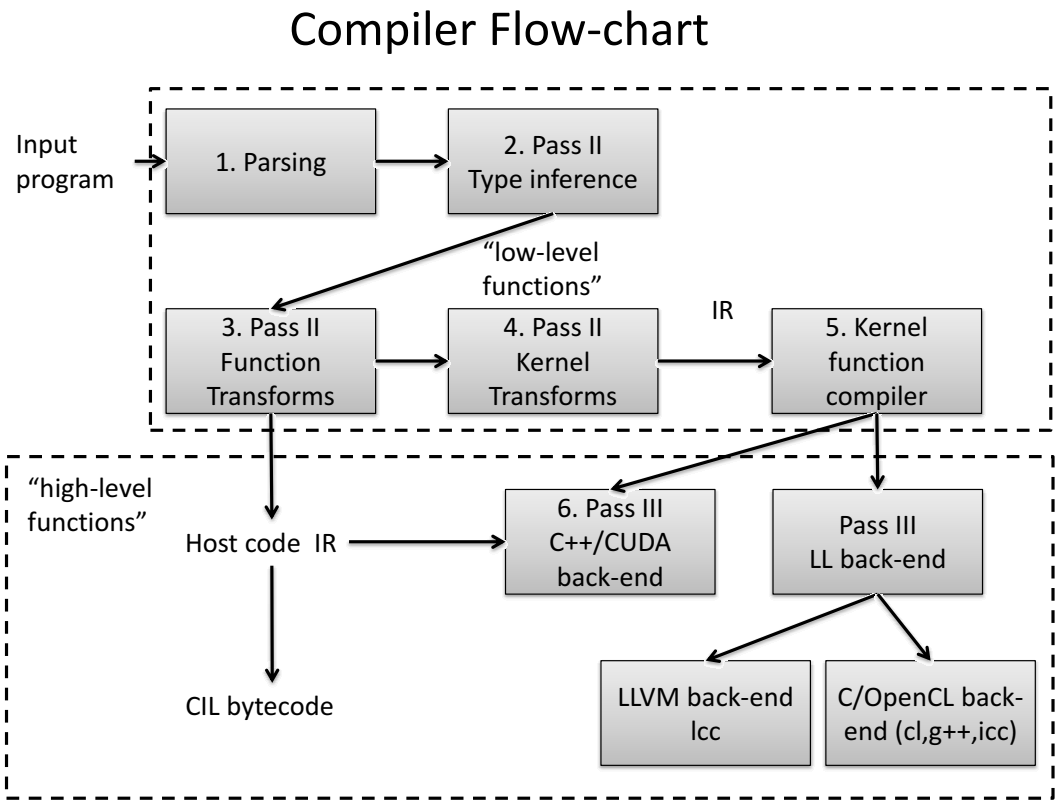


Figure 17.1: Overview of the Quasar compiler

- *High level inference*: performs high-level inference of certain operations.
- *Automatic function instantiation*: automatic instantiation/specialization of functions
- *Automatic kernel specialization*: automatically specializes kernel functions
- *Function inlining transform*: inlines function calls annotated with the `$inline` meta-function (see section §8.6), uses heuristics to inline functions automatically depending on the optimization level
- *Dynamic memory handler*: manages and checks the use of dynamic kernel memory inside kernel functions.
- *Generic function predicator*: detects functions (typically device/kernel functions) that are generic and that need to be specialized
- *Imperfect loop optimization*: converts certain types of imperfect loops to perfect loops that can be parallelized using the automatic loop parallelizer.
- *Device function parallelizable test*: checks whether a device function can be executed in parallel (useful for automatic loop parallelization)
- *Shared memoy promotion*: modifies kernel functions to use shared memory as a cache, typically due to the use of shared memory designators (see 9.2).
- *Modular optimization framework*: a group of interprocedural optimizations, such as schedule reordering optimizations, memory resource optimization and kernel fusion (see section 17.1.6).

- *Kernel post-processor*: invoking the kernel transform pipelines and generating target-specific code.

Some of the function transforms are highlighted below.

17.1.1 Automatic For-Loop Parallelizer (ALP)

The ALP automatically parallelizes multi-dimensional loops (of arbitrary dimension) in the code, generating a kernel function and corresponding `parallel_do` call. In case of dependencies between the iterations of the for-loop, it is also possible that a `serial_do` call is generated. In that case, the loop will be executed serially on the CPU. The ALP attempts to parallelize starting with the outside loops. The ALP program automatically recognizes one, two or higher dimenional for-loops, and also maximizes the dimensionality of the loops subject to parallelization.

Automatic Parallelization Error Messages During parallelization of for-loops, several compiler errors may be generated. These compiler errors can be classified in four categories: 1) errors related to dependency problems, 2) errors related to features that are currently not supported by the ALP, 3) issues related to the use of dynamic kernel memory and 4) attempts to call host functions from the loop. The errors are now discussed in more detail:

1. *Dependency problems*:

   In case of dependency problems, the loop cannot be parallelized but will rather be serialized. In some cases, it is useful to modify the algorithm to get rid of the dependency.

   - Parallelization not possible: dependency on iterator variable!

   It is also possible to force a loop to be parallelized irrespective of the warnings, using

   ```
   {!parallel for}
   ```

   . Note that this is at the responsibility of the user - the result may be incorrect if there are data races.
2. *Unsupported features*:
   Some features are currently unsupported by the loop parallelizer. For example, the types of the variables must be determined statically.

   - Operations involving strings can currently not be parallelized/serialized.
   - Operations involving objects can currently not be parallelized/serialized.
   - Construction of cell arrays can not be parallelized/serialized.
   - Statement can not be parallelized/serialized. Consider placing this outside the loop.
   - Cannot use reserved variable name in this context.
   - Parallelization/serialization not possible because the variable is polymorphic (not a unique type).
   - Inner loop break detected. Loop cannot be parallelized/serialized.
   - continue detected. Loop cannot be parallelized/serialized.
   - Parallelization not possible because variable is not locally defined.
3. *Kernel dynamic memory*:
   Other parallelization operations result in dynamic memory use inside the kernel (which may degrade the performance on, e.g., a GPU). In particular, without dynamic kernel memory (see section §8.3), matrix expressions such as `A[:,2]`, `B[3,4,:]` can not be parallelized. This is because the size of the result is unknown to the compiler. When the result is a small vector (of length <= 32), the problem can be solved using the sequence syntax, with constant lower and upper bounds. For example: `A[0..4,2]`, `B[3,4,0..2]`.

- Operator : can not be parallelized. Consider using `[a..b]` instead!
- Parallelization of the function requires kernel dynamic memory to be enabled!
- Operator requires dynamic kernel memory.
- Parallelization of the operator not possible: need to know the size of the result of type!

4. *Calling host functions*:
   Host functions cannot be called from a device/kernel function. Consequently, when a host function is directly called from a loop (and the host function cannot be inlined), the parallelization fails.
   - Function call can not be parallelized! Check if the function is declared with the `__device__` modifier!
   - Method is not a device function and can therefore not be parallelized/serialized!
   - Parallelization/serialization not possible because the specified function is not a device or kernel function!

17.1.2 Automatic Kernel Generator

The automatic kernel generator extracts expression patterns that are used once, or multiple times throughout the code. For example:

```
function Z = f(A:mat,X:mat,Y:mat)
    Z = A.*X+Y+4
endfunction
```

In this case, the expression `A.*X*Y+T` will be extracted and converted to a kernel function that can be evaluated with one single `parallel_do` function (instead of multiple kernels being called by the run-time). The generated kernel function will be:

```
function Z = opt1(A:mat,X:mat,Y:mat,T:scalar)
    function [] = __kernel__ kernel(Z:mat,A:mat,X:mat,Y:mat,T:scalar,pos:ivec2)
        Z[pos]=A[pos].*X[pos]+Y[pos]+T
    endfunction
    Z = uninit(size(A))
    parallel_do(sizeof(Z),Z,A,X,Y,T,kernel)
endfunction
```

Additionally, a reduction is generated internally, allowing the reduction pattern to be reused for later optimizations:

```
reduction (A:mat,X:mat,Y:mat,T:scalar)->A.*X+Y+T = opt1(A,X,Y,T)
```

The code is then transformed into:

```
function Z = f(A:mat,X:mat,Y:mat)
    Z = opt1(A,X,Y,4)
endfunction
```

Another purpose of the automatic kernel generator: the back-ends typically do not know how to handle high-level expressions such as `(A:mat)+(B:mat)`, therefore this transform converts the operations to low-level calls that can be handled by the back-end. This may lead to nested kernel function calls, utilizing CUDA dynamic parallelism or OpenMP nested parallelism (see section §4.4).
The automatic kernel generator is in a certain sense similar to the automatic loop parallelizer, with the main difference that the kernel generator starts from expressions involving matrices (but no indices) while the automatic loop parallelizer starts from loops involving matrices with indices (hence scalar values).

The automatic kernel generator also supports certain aggregation functions, such as `sum`, `prod`, `max`, `min`. When such a function is used, for example in expressions as `sum(A.*B+C)` the compiler will generate a parallel reduction algorithm to calculate the final result.

17.1.3 Automatic Function Instantiation

The automatic function instantiation step checks for certain conditions in which a (generic) function can be specialized:

- The function to be called is generic and requires specialization (e.g., because of a contained kernel function with generic arguments).
- Some of the arguments of the function are generic device functions. In this case, a type deduction is performed for obtaining the generic parameters. For example, consider calling the following function:

```
function y = apply[T](x : T, y : T, fn : [__device__ (T,T)->T])
    y = fn(x,y)
endfunction
print apply(1,2,(__device__(x,y)->x+y))
```

  Here, the type deduction will find that `T==int`, so that the function `fn` ha type `[__device__ (int,int)->int]`. Correspondingly, the function `apply` can be specialized.

17.1.4 High Level Inference

The high-level inference transform infers high-level information from Quasar programs. High-level here refers to information on high-level data structure level (like vector/matrix level). The information can then be used in several later optimization stages.
Consider the following function:

```
function [output:cube] = overlay_labels(image:cube, nlabels:mat)
    output = zeros(size(image))

    function [] = __kernel__ draw_outlines(nlabelsk:mat'clamped,pos)
        if nlabelsk[pos] == nlabelsk[pos+[0,-1]] && nlabelsk[pos] == nlabelsk[pos+[-1,0]]
            %interior pixel
        else
            %edge pixel
            output[pos[0],pos[1],0] = 1
            output[pos[0],pos[1],1] = 1
            output[pos[0],pos[1],2] = 1
        endif
    endfunction
    parallel_do(size(input,0..1), nlabels, draw_outlines)
endfunction
```

The program contains some logic for converting a label image (where every segment has a constant value) into an image where the segment boundaries are all assigned the value 1.
Perhaps without knowing so, the programmer actually specifies a lot of extra information that the compiler can exploit. A first approach is *type inference*, which allows the compiler to generate code with "optimal" data types that can be mapped onto x86/64/GPU instructions.

However, there is actually a lot more high-level information available. For example, after the assignment `output = zeros(size(image))`, we know that `size(output) == size(image)`. Combined with the `parallel_do` construct, we can determine that the boundary checks for `output` are actually unnecessary!
In fact, every statement has a number of pre-constraints, and after processing the statement, some post-constraints holds. For example, for the `transpose` function:

```
y = transpose(x)
% pre: ndims(x) <= 2
% post: size(y) = size(x,[1,0])
```

For the matrix multiplication:

```
y = A * x
% pre: ndims(A) <= 2 && ndims(x) <= 2
% post: size(y) = [size(A,0), size(x,1)]
```

The constraints do not only help the compiler to find out mistakes (such as incompatible matrix dimensions), but can also be used for controlling the later optimization stages. For example:

```
for m=0..size(z,0)-1
    for n=0..size(z,1)-1
        z[m,n] = x[m,n] + y[m,n]
    endfor
endfor
```

If it is known that `size(z)==size(x) && size(z) == size(y)`, then not only the boundary checks can be omitted, but also the loop can be flattened to a one dimensional loop, resulting in performance benefits on both CPU and GPU.

17.1.5 Function inlining

Functions can be inlined by annotating the function itself (C style inlining) or by annotating the function call with the `$inline` meta function (section §8.6). Marking a function to be inlined can be done using `{!auto_inline}`:

```
function [x,y] = __device__ sincos(theta)
    {!auto_inline}
        [x,y] = [sin(theta),cos(theta)]
endfunction
```

Alternative, the function can be inlined on a case by case basis (using `$inline`):

```
[a,b] = $inline(sincos)(0.3)
```

In this example, the compile-time expansion will even lead to the result being calculated at compile-time.
The Quasar compiler also uses its own heuristics to automatically inline functions. By default lambda expressions are inlined differently than functions (for lambda functions `{!auto_inline}` can not be used). The lambda expression inlining mode depends on the `COMPILER_LAMBDAEXPRESSION_INLINING` setting (see section 17.3), which has the following values:

- *Never*: never inline any lambda expressions
- *Always*: automatically inline every lambda expression

- *OnlySuitable (default)*: only inlines "simple" lambda expressions that do not have closures.

Note that the back-end compilers may aggressively inline device functions, even without inlining being specified on the Quasar language level.

17.1.6 Kernel fusion

Nowadays, it is not uncommon that a convolution operation on a Full HD color image takes around 10 microseconds. However, with execution times so low, for many GPU libraries this has the consequence that the bottleneck moves back from the GPU to the CPU: the CPU must ensure that the GPU is busy at all times. This turns out to be quite a challenge on its own: when invoking a kernel function, there is often a comined runtime and driver overhead in the order of 5-10 microseconds. That means that all launched kernel functions must provide sufficient workload. Because just a filtering operation on a Full HD image is already in the 5-10 microsecond range of execution time, many smaller parallel operations (e.g., operations with data dimensions$< 512 \cdot 1024$) are often not suited anymore for the GPU, unless the computation is sufficiently complex. In the new features of CUDA 10 it is mentioned:

> *"The overhead of a kernel launch can be a significant fraction of the overall end-to-end execution time."*

CUDA 10 introduces a graph API, which allows kernel launches to be prerecorded in order to be played back many times later. This technique reduces CPU kernel launch cost, often leading to significant performance improvements. We notice however that CUDA graphs are a runtime technique; its compile-time equivalent is kernel fusion. In kernel fusion, several subsequent kernel launches are fused into one big kernel launch (called a *mega*kernel). In older versions of CUDA, dependent kernels could not easily be fused, because every kernel execution imposes an implicit grid synchronization at the end of execution. This grid synchronization could only be avoided by using CUDA streams which allows independent kernels to be processed concurrently. More recently, grid barriers (which synchronizes "all threads" in the grid of a kernel function) have become available, either via cooperative groups (in CUDA 9) or via emulation techniques. These grid barriers open the way to kernel fusion of dependent kernels.
Obviously, any synchronization point such as a grid barrier involves a certain overhead, a time that GPU threads spend waiting for other threads to complete. The total overhead can be minimized by minimizing the number of synchronization points. On the other hand, grid barriers are entirely avoided when there are no dependencies between the kernels. This automatically means that reordering of kernel launches is an essential step of the automatic kernel fusion procedure.
The application of compile-time kernel fusion also has several other performance related benefits: when multiple kernels become one kernel often temporary data stored in global memory can be entirely moved to the GPU registers. Since accessing GPU registers is significantly faster than reading from/writing to global memory, the execution performance of kernels can be vastly improved. In addition, the compiler can reuse memory resources and eliminate memory allocations, essentially leading a static allocation scheme, in which all temporary buffers are preallocated, prior to launching the megakernel.
The kernel fusion also has a number of complicating factors:

- It is neccessary to determine at compile-time that arrays (vectors, matrices, ...) have the same size. Luckily, Quasar's high level inference engine allows to achieve this.
- Kernels are often launched with different data (i.e., grid and block) sizes, while launching a megakernel requires one size to be passed. In Quasar, this is achieved by performing automatic kernel tiling.
- Kernels often operate on data of different dimensionalities (vector, matrix, ...). In Quasar, this is solved by performing a kernel flattening transform in combination with a grid-strided loop.

The remedies for different data dimensions each involve a separate overhead, usually in the form of more registers used by the megakernel. This may lead to register-limited kernel launches, in which some GPU multiprocessors are

underutilized because of insufficient register memory. Therefore, Quasar includes a dynamic programming based optimization algorithm that takes all these factors into account.
In short, kernel fusion in Quasar is achieved by placing:

```
{!kernel_fusion strategy="smart"}
```

inside the parent function of the kernel functions. Important to realize is that all functions called by this parent function are inlined, so that if the callees launch kernel functions on their own, these kernel fusions can also be fused into the mega kernel. The compiler therefore sees a sequence of kernel launches and has to determine 1) a suitable order to launch the kernels and 2) whether the kernels are suited to perform kernel fusion.
When performing kernel fusion, a strategy needs to be passed, which controls the cost function used by the optimization algorithm:

Kernel fusion strategy	Purpose
`smart`	Let the Quasar compiler decide what is the best strategy
`noreordering`	Performs kernel fusion, *without* reordering the kernel launches
`minsyncgrid`	Minimizes the number of required grid synchronization barriers
`minworkingmemory`	Minimizes the total amount of (global) memory required for executing the fused kernel
`manual`	Kernel fusion barriers placed in the code determine which kernels are fused

Kernel fusion barriers `{!kernel_fusion barrier}` may be added to prevent kernels from being fused. In this case, M kernels are fused into N kernels with $1 < N \leq M$.
Under some circumstances (which also depend on the kernel fusion strategy), the compiler may waive fusion of certain kernels. The reasons can be inspected in the kernel fusion code transform log, which is accessible through the code workbench window in Redshift.

17.2 Kernel transforms

Various kernel transforms are available, that are specialized to parallel or serial loops:

- *Closure variable promotion pass*: promotes closure variables of kernel functions to fully fledges function parameters, at least whenever possible/advantageous.
- *Auto vectorization*: automatically vectorizes code, making it more suitable for execution on SIMD processors.
- *Branch divergence reducer*: converts conditional computations into equivalent branch-free expressions, with the goal to reduce branch divergence on a GPU
- *Kernel flattening*: reduces the dimensionality of loops whenever possible. Loops of lower dimensionality are often more efficient due to the smaller number of registers being used by the resulting kernel function.
- *Kernel data access scout*: performs some scanning and annotating operations necessary for later optimization steps.
- *Kernel dimension interchange*: changes the order of the loops in a multi-dimensional for loop
- *Memory coalescing transform*: automatically detects lack of memory coalescing and suggests an appropriate dimension interchange to improve memory coalescing.
- *Demultiplexer*: splits up the compilation streams, allowing target-dependent code to be generated

- *Kernel tiling*: tiles the iterations of a loop; useful for loop unrolling and vectorization. Additional benefits on CUDA architectures.
- *Parallel reduction transform*: converts loops with `+=`, `*=` etc. accumulation patterns to the parallel reduction algorithm.
- *Parallel dimension reduction transform*: same as the parallel reduction transform, but also works for aggregation along one dimension and for parallel prefix sum patterns.
- *Local windowing transform*: Caches global memory into shared memory, in order to reduce memory access times. Mainly useful for targetting GPUs
- *CPU skeleton generation*: Generates skeleton code for the CPU so that the kernel function can run efficiently on x86/x64 and ARM architectures. The pass may take information generated during previous passes into account (e.g. to enable vectorization of instructions)
- *Boundary accessor transform*: adds boundary accessor functions (see above) for newly generated array indexers
- *Shared memory caching transform*: caches some of the input/output matrices in shared memory. Useful for, e.g., histogram calculation.
- *Remove singleton matrix dimensions*: replaces matrices or cubes with singular dimensions by lower-dimensional versions of those. This allows the indexing to be performed more efficiently (eliminating multiplication operations with dimension 1).
- *Target-specific optimization transform*: performs some user-defined target-specific optimizations (usually based on the `$target()` meta function).
- *Avanced post optimization*: scans for several suboptimal patterns generated by other kernel transforms and replaces these patterns by more efficient versions.
- *SIMD processing*: automatically vectorizes code, depending on the default vector length for the target architecture

Some of the transforms are described in more detail below.

17.2.1 Parallel Reduction Transform

The parallel reduction transform (PRT) is specifically useful for GPU target platforms (CUDA/OpenCL), exploiting both the thread synchronization and shared memory capabilities of the GPU. The PRT detects a variety of accumulation patterns in output variables, such as `result += 1.0`. The accumulation operator is always an atomic operator (e.g., atomic add `+=`, atomic multiply `*=`, atomic minimum `__=`, atomic maximum `^^=`).
The following is an example of an accumulation patterns where the PRT can be applied:

```
y = y2 = 0.
for m=0..size(img,0)-1
    for n=0..size(img,1)-1
        y += img[m,n]
        y2 += img[m,n].^2
    endfor
endfor
```


This loop can be executed in parallel using atomic operations, however this may cause a poor performance. The parallel reduction transform converts the above pattern to:[1]

```
function [y:scalar,y2:scalar] = __kernel__ kernel(img:vec'const,$datadims:int,blkpos:int,blkdim:int)
    $bins=shared(blkdim,2)
    $accum0=$accum1=0
    $m=blkpos

    while $m<$datadims
        $accum1+=$getsafe(img,$m)
        $accum0+=($getsafe(img,$m).^2)
        $m+=blkdim
    endwhile

    $bins[blkpos,0]=$accum0
    $bins[blkpos,1]=$accum1
    syncthreads
    $bit=1

    while $bit<blkdim
        if mod(blkpos,(2*$bit))==0
            $bins[blkpos,(0..1)]=($bins[blkpos,(0..1)]+$getsafe($bins,(blkpos+$bit),(0..1)))
        endif

        syncthreads
        $bit*=2
    endwhile

    if sum(blkpos)==0
        y2+=$bins[0,0]
        y+=$bins[0,1]
    endif
endfunction
```

The parallel reduction transform relieves the user from writing complicated parallel reduction algorithms. Accumulator variables can have `scalar`, `cscalar`, `int`, `vecX`, `cvecX` and `ivecX` datatypes. The transform calculates the required amount of shared memory and ensures that the kernel is not performance limited due to shared memory pressure. For CUDA 9 (or newer) backends, the parallel reduction transform switches to a warp-shuffling based algorithm when too many accumulator variables are present. It is even possible to specify per accumulator variable which parallel reduction algorithm needs to be used:

```
y = y2 = 0.
for m=0..size(img,0)-1
    for n=0..size(img,1)-1
        {!kernel_accumulator name=y; algorithm="warpshuffle"}
        {!kernel_accumulator name=y2; algorithm="sharedmemory"}
        y += img[m,n]
        y2 += img[m,n].^2
    endfor
endfor
```

[1] Actual implementation details may vary for different versions of Quasar.

Table 17.1: Available parallel reduction algorithms

Algorithm	Purpose
`default`	Lets the compiler choose the most suitable parallel reduction algorithm
`sharedmemory`	Performs the parallel reduction in shared memory
`warpshuffle`	Performs the parallel reduction using warp shuffling operations (CUDA 9 or higher)
`atomicop`	Uses a grid-strided loop in combination with an atomic operation

table 17.1 lists available parallel reduction algorithms. `atomicop` leads possibly to the least efficient algorithm but is available as alternative (e.g., for platforms on which warp shuffling is not supported).

17.2.2 Local Windowing Transform

The local windowing transform is designed to improve local windowing operations (also known as *stencil operations*), such as in convolutions, morphological operations. The transform is again useful for code that has to run on the GPU. Currently, the transform needs to be enabled with `{!kernel_transform enable="localwindow"}`.

```
function [] = __kernel__ joint_box_filter_hor(g : mat'mirror, x : mat'mirror, tmp_g : mat'unchecked,
    tmp_x : mat'unchecked, tmp_gx : mat'unchecked, tmp_gg : mat'unchecked, r : int, pos : vec2)
    {!kernel_transform enable="localwindow"}
    s_g = 0.
    s_x = 0.
    s_gx = 0.
    s_gg = 0.
    for i=-r..r
        t_g = g[pos + [0,i]]
        t_x = x[pos + [0,i]]
        s_g = s_g + t_g
        s_x = s_x + t_x
        s_gx = s_gx + t_g*t_x
        s_gg = s_gg + t_g*t_g
    endfor
    tmp_g[pos] = s_g
    tmp_x[pos] = s_x
    tmp_gx[pos] = s_gx
    tmp_gg[pos] = s_gg
endfunction
```

This is translated into:

```
function [] = __kernel__ joint_box_filter_hor(g:mat'const'mirror,x:mat'const'mirror,tmp_g:mat'
    unchecked,tmp_x:mat'unchecked,tmp_gx:mat'unchecked,tmp_gg:mat'unchecked,r:int,pos:ivec2,blkpos:
    ivec2,blkdim:ivec2)
    {!kernel name="joint_box_filter_hor"; target="gpu"}
    sh$x=shared((blkdim+[1,((r+r)+1)]))
    sh$x[blkpos]=x[(pos+[0,-(r)])]

    if (blkpos[1]<(r+r))
        sh$x[(blkpos+[0,blkdim[1]])]=x[(pos+[0,-(r)]+[0,blkdim[1]])]
    endif

    blkof$x=((blkpos-pos)-[0,-(r)])
    sh$g=shared((blkdim+[1,((r+r)+1)]))
    sh$g[blkpos]=g[(pos+[0,-(r)])]
```



```
    if (blkpos[1]<(r+r))
        sh$g[(blkpos+[0,blkdim[1]])]=g[(pos+[0,-(r)]+[0,blkdim[1]])]
    endif

    blkof$g=((blkpos-pos)-[0,-(r)])
    syncthreads
    s_g=0.
    s_x=0.
    s_gx=0.
    s_gg=0.
    for i=-(r)..r
        t_g=sh$g[(pos+[0,i]+blkof$g)]
        t_x=sh$x[(pos+[0,i]+blkof$x)]
        s_g=(s_g+t_g)
        s_x=(s_x+t_x)
        s_gx=(s_gx+(t_g*t_x))
        s_gg=(s_gg+(t_g*t_g))
    endfor
    tmp_g[pos]=s_g
    tmp_x[pos]=s_x
    tmp_gx[pos]=s_gx
    tmp_gg[pos]=s_gg
endfunction
```

17.2.3 Kernel Tiling Transform

Kernel tiling is a kernel function code transformation that divides the data in blocks of a fixed size. There are three possible modes:

- **Global kernel tiling**: the data is subdivided in blocks of, e.g., '32x32x1' or '32x16x1'. All threads in the grid collaborate to first calculate the first block, then the second block, and so on. The loop that traverses all these blocks is placed inside the kernel function. Note that the blocks normally don't corresponds to the GPU (CUDA) blocks. It is very common that a 1D block index traverses 2D or 3D blocks (also called grid-strided loop). Global kernel tiling has the following benefits:
  - Threads are reused; the maximum number of GPU blocks can be reduced, saving thread initialization and destruction overhead
  - Calculations independent of the block can be placed outside of the tiling loop
  - Less dependency on the GPU block dimensions, resulting in more portable code

  A disadvantages of global kernel tiling is that the resulting kernels are more complex and typically use more registers.

  Global kernel tiling can be activated by placing one of the following code attributes in the kernel function:

  ```
  {!kernel_tiling dims=[32,16,1]; mode="global"; target="gpu"}
  {!kernel_tiling dims=auto; mode="global"; target="gpu"}
  ```

  The target (e.g., `cpu`, `gpu`) specifies for which platform the kernel tiling will be performed. It is possible to enable tiling for one platform but not for the other. By combining multiple code attributes it is also possible

to specify different block sizes for different platforms. In case of automatic tiling dimensions, the runtime will search for suitable block dimensions that optimize the occupancy.

For certain kernels (e.g., involving parallel reduction, shared memory accumulation, ...), global kernel tiling is performed automatically. Kernels that use grid-level synchronization primitives (without TCC driver mode cooperative grouping) are also automatically tiled globally. This is also required in order for kernel fusion (see section 17.1.6) to work correctly.

- **Local kernel tiling**: here, each thread performs the work of N consecutive work elements. Instead of having 1024 threads process a block of size 32×32, we have 256 threads processing the same block. Each thread then processes data in a single instruction multiple data (SIMD) fashion. Moreover, the resulting instructions can be even mapped onto SIMD instructions (e.g., CUDA SIMD/SSE/AVX/ARM Neon, see also section §12), if the hardware supports them. In the following example, a simple box filter is applied to a matrix with element type `uint8`.

```
function [] = __kernel__ conv(im8 : mat[uint8], im_out : mat[uint8], K : int, pos : ivec(2))
    [m,n] = pos
    r2 = vec[int](4)
    for x=0..K-1
       r2 += im8[m,n+x+(0..3)]
    endfor
    im_out[m,n+(0..3)] = int(r2/(2*K))
endfunction
```

Local kernel tiling can be activated by placing the following code attribute in the kernel function:

```
{!kernel_tiling dims=[1,1,4]; mode="local"; target="gpu"}
```

In case purely SIMD processing is intended, it is better to use the following code attribute (see below):

```
{!kernel_transform enable="simdprocessing"}
```

Advantages of local kernel tiling:

  - Less threads, so less thread initialization and destruction overhead
  - mapping onto SIMD possible (when the block dimensions are chosen according to the GPU platform).
  - For some operations, recomputation of values can be avoided

Disadvantages:

  - The resulting kernels use more registers, which may negatively impact the performance in some cases (e.g. register limited kernels)
  - Not all operations may be accelerated by hardware SIMD operations (for example, division operations).
  - The compiler needs to ensure that the data dimensions are a multiple of the block size. If this is not the case, extra ”cool down” code is added.

- **Hybrid tiling**: combines the advantages of global and local kernel tiling. To activate hybrid tiling, code attributes for both local and global tiling can be placed inside the kernel function:


```
{!kernel_tiling dims=auto; mode="global"; target="gpu"}
{!kernel_tiling dims=[1,1,4]; mode="local"; target="gpu"}
```

Hybrid kernel tiling usually occurs as a result of several compiler optimizations.

Example

Consider the following convolution example:

```
for m=0..size(im,0)-1
    for n=0..size(im,1)-1
        {! kernel_tiling dims=[1,32]; mode="local"; target="cpu"}
        r = 0.0
        for x=-K..K
            r += im[m,n+x]
        endfor
        im_out[m,n] = r/(2*K+1)
    endfor
endfor
```

This is translated into:

```
function [] = __kernel__ opt___kernel$2(K:int,im:mat'const,im_out:mat'unchecked,$datadim:ivec2)
    $nMax=$cpu_gridDim().*$cpu_blockDim()
    {!parallel for; private=r; private=x}
    for $p1=0..$nMax[1]-1
        for $p2=0..$nMax[0]-1
            pos=[$p1,$p2]
            r=zeros(32)
            for x=-K..K
                r+=im[pos[0],32.*pos[1]+(0..31)+x)]
            endfor
            im_out[pos[0],32.*pos[1]+(0..31)]=r./((2.*K)+1)
        endfor
    endfor
endfunction
```

The operations `im[pos[0],32.*pos[1]+(0..31)]` are translated to vector operations.

17.2.4 Kernel Boundary Checks

The Kernel Boundary Check transform detects index-out-of-bounds at compile-time, given the available information (i.e., through the truth-value system).

In certain cases, the transform can detect that the index will always be inside the bounds. In this case, the kernel argument type can have the modifier `'unchecked` (omitting the boundary checks).

For example, the code fragment:

```
assert(size(A) == size(B))
parallel_do(size(A),A,B,
    __kernel__ (y : mat,x : mat,pos : vec2) -> y[pos] += 4.0 * x[pos])
```

Can be translated into:

Table 17.2: Supported target platform identifiers

Target platform identifier	Description
`cpu`	Generic CPU target
`gpu`	Generic GPU target
`nvidia_cuda`	NVidia CUDA target (NVidia GPUs supporting CUDA)
`nvidia_opencl`	NVidia OpenCL target
`nvidia`	All NVidia targets
`amd_opencl`	AMD OpenCL target
`amd`	All AMD targets
`generic_opencl`	All other OpenCL targets
`opencl`	All OpenCL targets

```
parallel_do(size(A),A,B,
    __kernel__ (y : mat'unchecked,x : mat'unchecked,pos : vec2) -> y[pos] += 4.0 * x[pos])
```

17.2.5 Target-specific programming and manually invoking the runtime scheduler

By default, the Quasar compiler takes a single kernel function as input and specializes the kernel function to multiple devices (e.g., CPU, GPU). In some circumstances, it may be desirable to manually write implementations for certain performance-critical functions for several targets. This can be achieved by either 1) writing multiple kernel functions and by indicating the compilation target within each kernel function (this section) or 2) using the `$target()` meta-function (section 17.2.6). The compilation target of a kernel function can be specified using:

```
{!kernel name="gpu_kernel"; target="gpu"}
```

Here, `gpu_kernel` indicates the name of the kernel function (which should correspond to the function definition). Several target identifiers are listed in table 17.2. Depending on the compilation mode, it may happen that no code is generated for a given kernel function (for example, a GPU kernel when compiling in CPU mode). To decide which kernel function implementation is used, the `schedule` function needs to be used which has mostly the same arguments as the `parallel_do` function, with exception of the last parameter which takes a cell vector of the kernel function implementation. The `schedule` function returns a zero-based index of the selected kernel function passed via the cell vector.

Because `schedule` on its own does not trigger `parallel_do`, the schedule function is best used within a `match` control structure. Within the control structure, also additional launching code could be placed (like calculation of the optimal launch configuration, e.g., using the `max_block_size`, `opt_block_size` and `opt_block_cnt` functions). An example is given below for a sum algorithm using the parallel reduction pattern. Actually, for illustrational purposes: this is the parallel reduction code that is generated for the map-reduce pattern from section §8.4.

Example: a multi-target Reduce and Map algorithm with automatic scheduling

```
function y = manual_map_sum(x : vec, map : [__device__ (scalar)->scalar])
    % GPU implementation
    function y = __kernel__ gpu_kernel(x : vec'unchecked, nblocks : int,
        map : [__device__ (scalar) -> scalar], blkdim : int, blkpos : int)
        {!kernel name="gpu_kernel"; target="gpu"}
        bins = shared(blkdim)
```



```
        % step 1 - parallel sum
        val = 0.0
        for n=0..nblocks-1
            if blkpos + n * blkdim < numel(x)
                val += map(x[blkpos + n * blkdim])
            endif
        endfor
        bins[blkpos] = val
        % step 2 - reduction
        syncthreads
        bit = 1
        while bit<blkdim
            index = 2*bit*blkpos
            if index+bit<blkdim
                t = bins[index] + bins[index+bit]
                bins[index] = t
                syncthreads
            endif
            bit *= 2
        endwhile
        if blkpos==0
            y = bins[0]
        endif
    endfunction

    % Implementation using OpenMP
    function y = __kernel__ cpu_kernel(x : vec'unchecked,
        map : [__device__ (scalar) -> scalar])
        {!kernel name="cpu_kernel"; target="cpu"}
        val = 0.0
        % The following special directive activates OpenMP to perform
        % a parallel reduction. {!parallel for} is indended to
        % eliminate #pragma force_parallel in the long term; since
        % it provides a means to specify special attributes.
        {!parallel for; reduction={val,"+="}}
        for n=0..numel(x)-1
            val += map(x[n])
        endfor
        y = val
    endfunction

    % Manually call the scheduler with two implementations
    % In each case, we need to do some additional work
    match schedule(numel(x), x, map, {gpu_kernel, cpu_kernel}) with
    | 0 ->
        N = numel(x)
        BLOCK_SIZE = prod(max_block_size(gpu_kernel, N))
        nblocks = int((N + BLOCK_SIZE-1) / BLOCK_SIZE)
        y = parallel_do([[1,BLOCK_SIZE],[1,BLOCK_SIZE]],x,nblocks,map,gpu_kernel)
    | 1 ->
        y = serial_do(1,x,map,cpu_kernel)
    endmatch
endfunction
```


In this example, two kernel functions have been implemented: one for respectively the GPU and the CPU. The GPU implementation uses the parallel reduction algorithm based on shared GPU memory (see section 9.7), while the CPU implementation relies on the OpenMP parallel reduction pragma. The scheduling function is then called to determine the best suitable target for this operation. Subsequently, when the scheduler selects the GPU, additional launch configuration code is executed and the resulting parameters are passed to the `parallel_do` function.
Note that the parameter lists of `cpu_kernel` and `gpu_kernel` do not need to be equal. For `schedule`, it suffices to pass the relevant parameters that affect the scheduling decision.

17.2.6 Compile-time specialization through the $target() meta function

Optionally, kernel functions can use the `$target()` meta function for the implementation of target-specific code, for example:

```
if $target("nvidia_cuda")
    % We are compiling for CUDA targets
else
    % All other targets
endif
```

The `$target` function is evaluated at compile-time during a target-specific optimization step. The function returns 1 in case we are compiling for the specified target and 0 otherwise. For the possible values accepted by the `$target` function, see table 17.2.
Such approach is similar to preprocessor tests often used in C or C++ code. To keep the code readibility high, the technique can best be used when the amount of target-specific code is reasonably small (for example at most a few lines of code), otherwise it is recommended to write multiple kernel implementations (seesection 17.2.5).

17.3 Common compilation settings

Compilation settings can be configured in the config file `Quasar.config.xml`. This file is stored in `%APPDATA%\..\Local\Quasar` (windows) or `~/.config/Quasar` (linux). A number of global settings are listed in table 17.3. Some of the global settings can also be modified in the program, using the `#pragma` directive. The following pragmas are available:

`#pragma loop_parallelizer (on|off)`	Turns off/activates the automatic loop parallelizer
`#pragma show_reductions (on|off)`	Enables/disables messages when reductions are applied.
`#pragma expression_optimizer (on|off)`	Enables/disables the expression optimizer.

These settings can also be set directly using the Redshift IDE (see section §18.1).

17.4 CUDA target architecture

The CUDA target architecture (compiler option `CUDA_BACKEND_TARGETARCHITECTURE`) specifies the name of the NIVDIA virtual GPU architecture to generate code for. Possible values are Default, sm_13, sm_20, sm_21, sm_30, sm_35, sm_37, sm_50, sm_52, sm_53, sm_50, sm_60, sm_61, sm_62, sm_70, sm_71, sm_72, sm_75. table 17.4 gives an overview of common GPU architectures:
A complete list can be found on the NVIDIA website: https://developer.nvidia.com/cuda-gpus.
When the option *'Default'* is selected, the target architecture is automatically determined based on the installed graphics card. However, in some cases, it is useful to set a target architecture that differs from the default value:

Table 17.3: Global compilation settings

Setting	Value	Description
NVCC_PATH	path	Contains the path of the NVCC compiler shell script (nvcc_script.bat or nvcc_script.sh)
CC_PATH	path	Contains the path of the native C/C++ compiler shell script
MODULE_DIR	directory	';' separated list of directories to search for .q files (when using import)
INTERMEDIATE_DIR	directory	Intermediate directory to be used for compilation. If none specified, the directory of the input file is used
COMPILER_PERFORM_REDUCTIONS	True/False	Enables reductions (reduction keyword)
COMPILER_REDUCTION_SAFETYLEVEL	off, safe, strict	Compiler safety setting for performing reductions
COMPILER_DISPLAY_REDUCTIONS	True/False	Displays the reductions that have been performed
COMPILER_DISPLAY_WARNINGS	True/False	Displays warnings during the compilation process
COMPILER_OUTPUT_OPTIMIZED_FILE	True/False	If true, the optimized .q file is written to disk (for verification)
COMPILER_EXPRESSION_OPTIMIZER	True/False	Enables automatic extraction and generation of __kernel__ functions (see section 17.1.2)
COMPILER_SHOW_MISSED_OPT_OPPORTUNITIES	True/False	Displays additional possibilities for optimization
COMPILER_AUTO_FORLOOP_PARALLELIZATION	True/False	Enables automatic parallelisation of for-loops (see section 17.1.1)
COMPILER_SIMPLIFY_EXPRESSIONS	True/False	Simplifies branch expressions whenever possible
COMPILER_PERFORM_BOUNDSCHECKS	True/False	Performs boundary checking of unchecked variables (CPU engine only) Useful for debugging mistakes in the use of the 'unchecked modifier.
COMPILER_PERFORM_NAN_INF_CHECKS	True/False	Generates code that automatically checks for NaN or Inf values (experimental feature)
COMPILER_PERFORM_NAN_CHECKS	True/False	Generates code that automatically checks for NaN values (experimental feature)
COMPILER_LAMBDAEXPRESSION_INLINING	Off,OnlySuitable,Always	Allows/disallows the inlining of lambda expressions
COMPILER_USEFUNCTIONPOINTERS	Always, SmartlyAvoid,Error	Controls the generation of kernel/device functions using function pointers
COMPILER_OPTIMIZATIONLEVEL	Basic/Moderate/Full	Specifies the optimization level used for compiling Quasar programs
COMPILER_BACKEND_CPU	Default/...	CPU back-end compiler to be used
COMPILER_BACKEND_CUDA	NVidiaC, NVidiaRTC	CUDA back-end compiler to be used
COMPILER_BACKEND_OPENCL	OpenCLDriver	OpenCL back-end compiler to be used
COMPILER_BACKEND_GENERATE_HUMAN_READABLE_CODE	True/False	Generates human readable code in the back-end stages (at the cost of longer code)
COMPILER_BACKEND_OUTPUTLINEINFO	True/False	Generate line number information in the binary files (.ptx). Useful for debugging and profiling.
CUDA_BACKEND_TARGETARCHITECTURE	Default, sm_20, ...	NVIDIA Target compile architecture (corresponding to the NVCC -arch option). See section §17.4.
CUDA_BACKEND_FLUSHTOZERO	True/False	Flushes denormalized numbers to zero
CUDA_BACKEND_PRECISIONDIVISION	True/False	Use precision division function
CUDA_BACKEND_PRECISIONSQRT	True/False	Use precision square root function
CUDA_BACKEND_FASTMATH	True/False	Use faster mathemetical functions (at the cost of lower numerical accuracy)
CUDA_BACKEND_OPTIMIZATIONMODE	O0/O1/O2	Sets the optimization level for compiling CUDA code
CUDA_BACKEND_DYNAMICPARALLELISM	True/False	Generates code that uses dynamic parallelism (requires target architecture sm_35 or higher)
CUDA_BACKEND_COOPERATIVEGROUPS	True/False	Generates code that uses cooperative groups (requires CUDA 9.1 or higher)
CUDA_BACKEND_WARPSHUFFLING	True/False	Generates code that uses warp shuffling (requires CUDA 9.1 or higher)

Name	Architecture	CUDA versions	Example cards
Fermi	sm_20	3.2-8.0	GeForce 400, 500, 600
Kepler	sm_30	5.0+	GeForce 700
	sm_35	5.0+	Tesla K40
	sm_37	5.0+	Telsa K80
Maxwell	sm_50	6.0+	Quadro M6000
	sm_52	6.0+	Geforce 900, Titan X
	sm_53	6.0+	Jetson TX1
Pascal	sm_60	8.0+	Tesla P100
	sm_61	8.0+	Geforce 1000, Titan Xp
	sm_62	8.0+	Jetson TX2
Volta	sm_70	9.0+	Tesla V100, TITAN V
Turing	sm_75	10.0+	Geforce RTX 2080 Ti

Table 17.4: Overview of CUDA target architectures

- When generating CUDA code on a computer without NVIDIA GPU (for example, when building a library or binary to be deployed later on a system with NVIDIA GPU).
- When targetting older GPU cards.

GPU architectures are (generally) backward compatible. When compiling for sm_30, one can use the Geforce GTX 770, Titan X and even Titan Xp, but it is not possible to use older cards as the GeForce GTX 480. However, take into account that the sm_30 may not take advantage of features in newer generations of GPUs. Therefore, only when backward compatibility is an issue, it is useful to select a lower architecture than what your GPU supports. On the other hand, some (very) old versions are not supported by the latest CUDA compiler. Therefore, it is recommended to use the option 'Default', unless there is a specific reason to generate code for a lower GPU architecture.

Chapter 18

Development tools

Quasar comes with two main development tools: Redshift (the main IDE) and Spectroscope (a commandline debugger tool).

18.1 Redshift - integrated development environment

Redshift is the main IDE for Quasar. It is built on top of GTK and runs on Windows, Linux and MAC. Redshift has the following main features:

- Multiple document editing with syntax highlighting and completion lists.
- Built-in Quasar Spectroscope (see section §18.2) for debugging and interactive programming
- Call stack, variable definition, variable watch, breakpoints windows, ...
- Incremental compilation, keeping binary modules (e.g., PTX) in memory: significantly decreases overall compilation time
- Source code parsing, function browsing, comment parsing
- Ability to break and continue running programs.
- Integrated documentation system and help system
- Integrated OpenGL functionality for fast visualization
- 2D and 3D image viewer
- Advanced profiler tool with timeline view and special profiler code editor margin.
- Interactive interpretation of Quasar commands
- Data/image debug tooltips
- Background compilation with source code error annotations

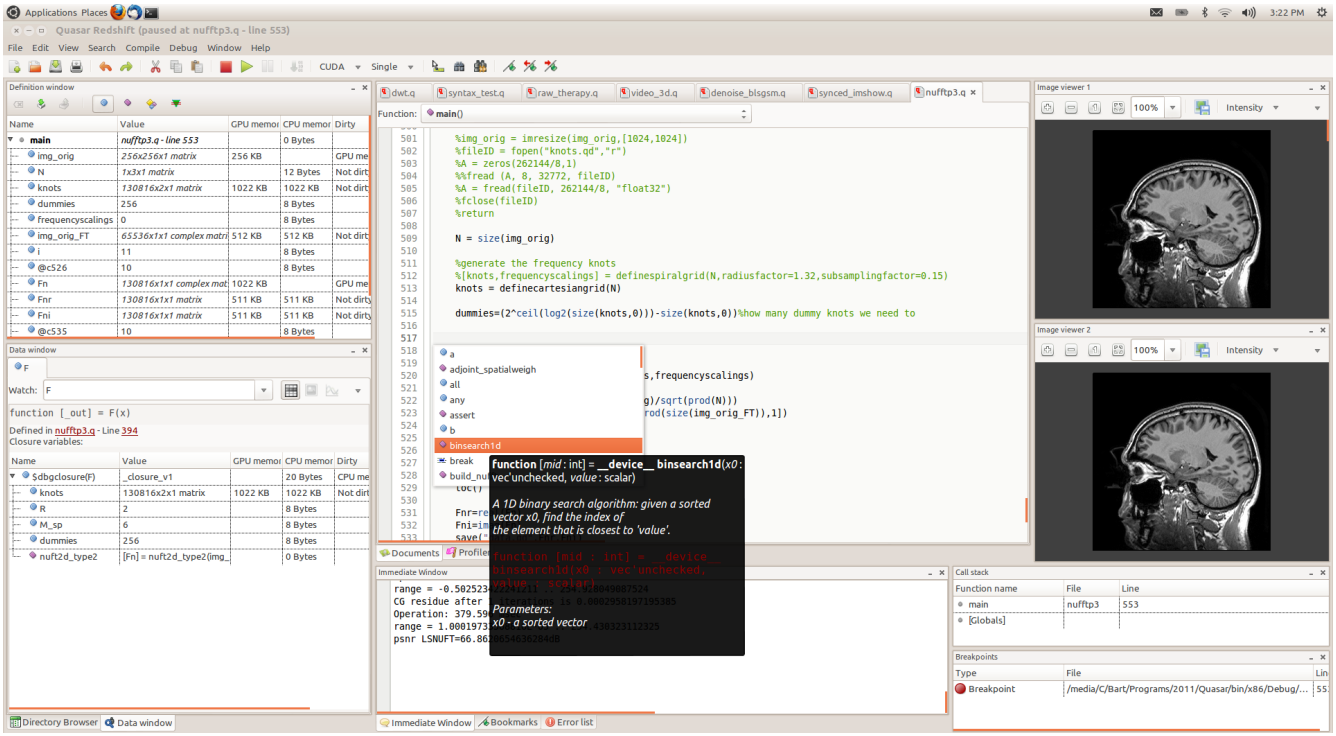


Figure 18.1: Screenshot of the Redshift IDE for Quasar.

- Optimization pipeline visualization

In the IDE, it is also possible to select the GPU devices on which the program needs to be executed in the main toolbar. Additionally, the precision mode (e.g., 32-bit floating point or 64-bit floating point) can be selected. The flame icon toggles concurrent kernel execution, a technique in which CUDA/OpenCL kernels are launched asynchronously; in many circumstances speeding up the execution.

18.2 Spectroscope - command line debugger

Spectroscope is a command line debugger for Quasar. Its functionality is actually integrated in Redshift (section §18.1), although in some circumstances it is still useful to use the debugging tools from a terminal in circumstances where a graphical environment is not available (e.g., over SSH on a remote server). Spectroscope provides an interactive environment where Quasar statements can be entered on interpreted on the fly. In addition, different commands are available:

- compile [program.q]: compiles a specified program while checking for compilation errors
- load [program.q]: compiles the specified program and loads the definitions (global variables and functions)
- run: runs the loaded program
- list: lists all variables within the current scope/context
- trace: prints the current stack trace
- step: performs a debugger step into a function
- stepover: let's the debugger step over the current statement

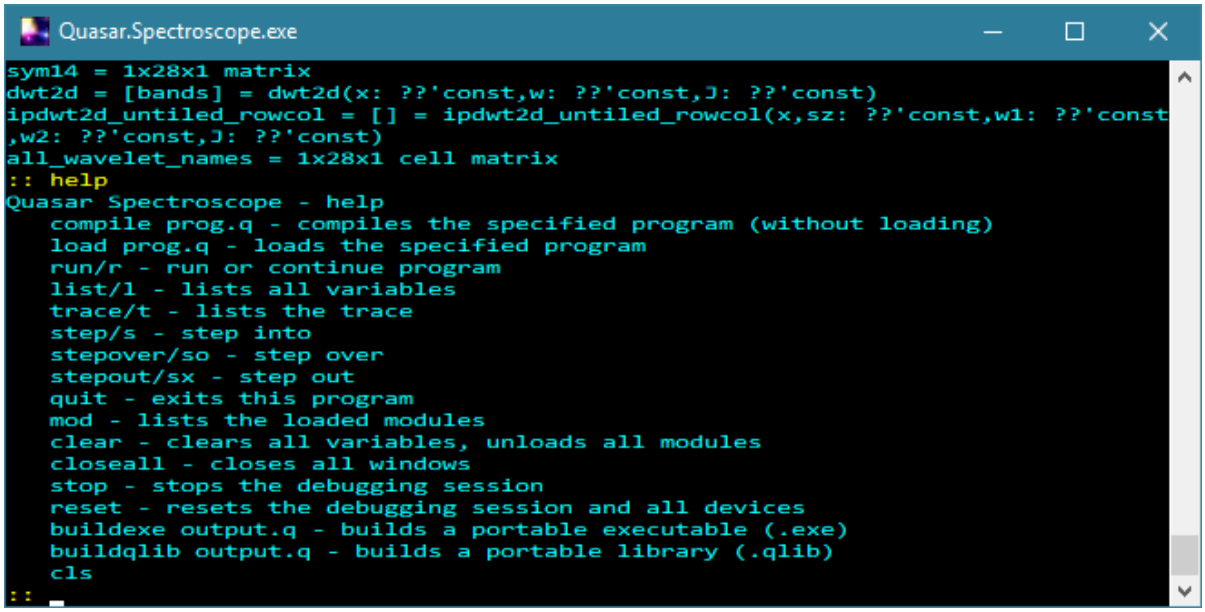


Figure 18.2: Screenshot of Quasar Spectroscope.

- stepout: let's the debugger step out of the current function
- mod: lists all loaded Quasar modules
- clear: clears all variables, unloads all loaded modules
- path: prints the current module search directory
- stop: terminates the current debugging session
- reset: debugger hard reset - resets the debugging session and all devices
- buildexe [program.q]: builds a portable executable file (.exe)
- buildqlib [program.q]: builds a portable library (.qlib)
- cls: clears the screen

18.3 Redshift Profiler

To analyze the performance of Quasar programs, a profiler has been integrated in Redshift. The profiler uses the NVIDIA CUDA Profiling tools interface (CUPTI) to obtain accurate time measurements. Using the Redshift Profiler, it is not necessary to use the tools nvprof, NVIDIA Visual Profiler and NVIDIA nSight separately. The Redshift Profiler offers the following features:

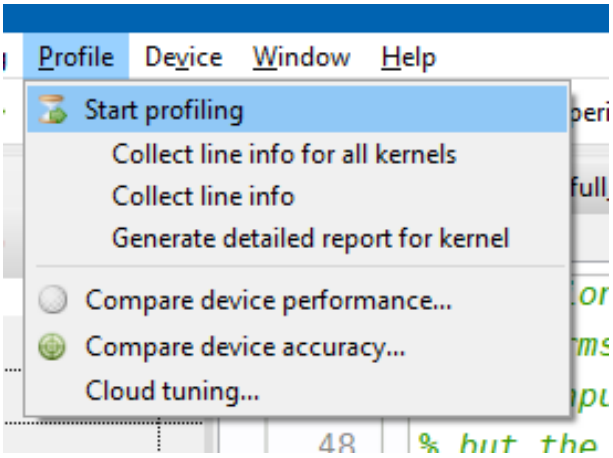
- Accurate kernel timing measurements with less profiling overhead
- Integrated source correlation, to see instruction counts per code line
- Detailed timeline view, linking kernel launches to the corresponding CPU activity
- GPU events are linked with the corresponding CPU events (e.g. parallel_do or function call on the CPU).

- Several detailed metrics are available for analyzing the performance of an individual kernel. Metrics include FLOPs/sec, integer operations/sec, branch efficiency, achieved occupancy, average number of instructions per second and many more.
- Detailed GPU event view, to see individual operations performed on the GPU
- Support for Multi-GPU profiling (e.g., peer to peer memory transfers)
- Memory profiling: sub-tree view of memory allocations at any point in time

To take advantage of CUPTI, it is necessary to enable "Use NVIDIA CUDA profiling tools" in the program settings (done by default). Without this option set, the Quasar Profiler reverts to CUDA events (which is less accurate and degrades the performance during profiling).
In Windows, `cupti.dll` is bundled with the Quasar installation. In Linux, it may be necessary to adjust the LD_LIBRARY_PATH to include `libcupti.so`, depending on the installed version of CUDA. This can be done by modifying the `.bashrc` file (for example, for cuda 10.2):

```
export LD_LIBRARY_PATH=/usr/local/cuda-10.2/extras/CUPTI/lib64/:$LD_LIBRARY_PATH
```

The profiling menu in Redshift has been updated with the new features, as can be seen in the following screenshot:



The following features are available:

- *Start profiling*: executes the program with the profiler attached. This will collect information about the CPU and GPU kernels, memory allocations, memory transfers...
- *Collect line information for all kernels*: executes all kernel functions in the current module with an execution tracer attached. This slows down the execution of the kernel functions. The results are displayed in the source code editor (see further).
- *Collect line information for a specific kernel*: this is useful when you are optimizing one particular kernel
- *Generate detailed report for kernel*: executes the program, collecting an extensive sets of metrics for the selected kernel function and generates a report (see below)
- *Compare device performance*: useful to compare the execution performance of two devices (e.g. CUDA vs. OpenCL)
- *Compare device accuracy*: checks the accuracy/correctness of the program by gathering statistics (e.g. minimum value, maximum value, average value of a matrix) during the execution of the program.

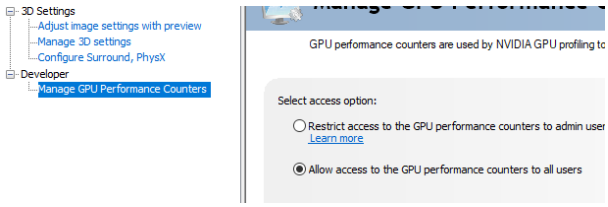


Figure 18.3: NVIDIA control panel: allow access to the GPU performance counters to all users

- *Cloud tuning*: this feature is used to optimize the runtime scheduler and load balancer based on runtime measurements of kernels. The results (kernel execution times on CPU and GPU) are sent to the Gepura/Quasar server.

As a result of CUPTI, the profiler may now list kernel functions that are not visible to Quasar but that are launched by library calls (e.g., CuDNN, CuBLAS, CuFFT, ...). An example is given below:

Kernel summary

Priority	Kernel	Module	Device	Launches	Total time	Avg duration
1	convolve_biased_deriv_bias	cnn_full_multigpu	GeForce G	198	312.705 ms	1.579 ms
2	maxwell_scudnn_128x32_lar	(unknown)	GeForce G	1192	169.915 ms	142.5 us
3	_Z15fft2d_c2r_32x32IfLb0EL	(unknown)	GeForce G	6336	84.737 ms	13.3 us
4	_Z28fermiPlusCgemmLDS1	(unknown)	GeForce G	6336	83.657 ms	13.2 us
5	_Z15fft2d_r2c_32x32IfLb1EE	(unknown)	GeForce G	6435	74.749 ms	11.6 us

18.3.1 Security settings

In CUDA 10.1 or newer, using cuPTI profiling requires an additional setting to be made. In *Windows*, open the NVIDIA control panel. Click on the desktop menu and enable the developer settings. Then, select "Manage GPU Performance Counters" and click on "Allow access to the GPU performance counters to all users" (see figure 18.3).
In *Linux desktop systems,* create a file `/etc/modprobe.d` with the following content:

```
options nvidia "NVreg_RestrictProfilingToAdminUsers=0"
```

where on some Ubuntu systems, nvidia may need to be replaced by `nvidia-xxx` where `xxx` is the version of the display driver (use `lsmod | grep nvidia` to find the number). In addition, it may be necessary to rebuild the ram FS:

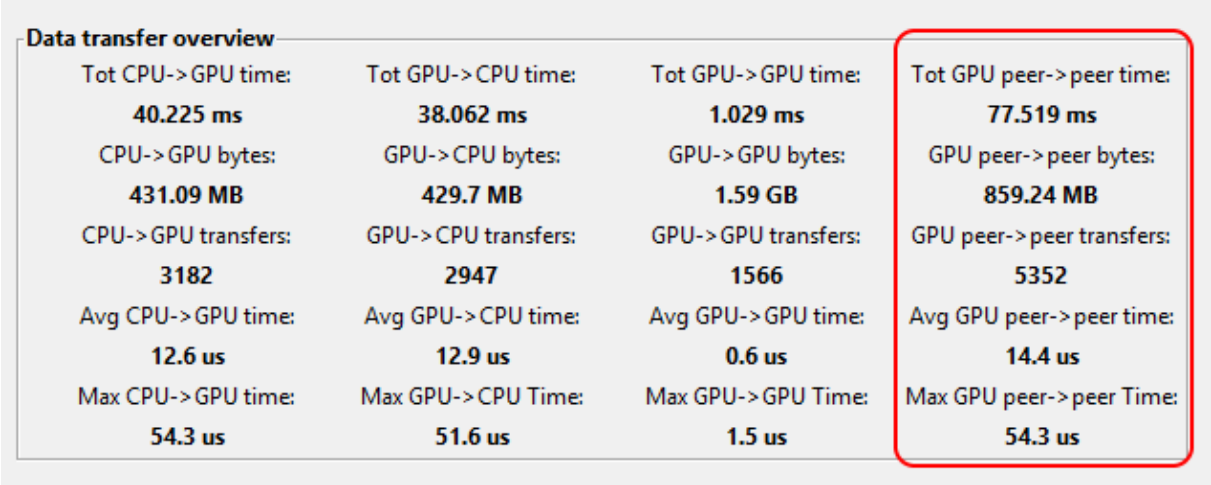
```
update-initramfs -u
```

We recommend creating a backup of the `initrd` image first (see `/boot` directory).
For NVIDIA Jetson development boards, the modprobe approach is not available. The only way to get the cuPTI profiling to work is by executing Quasar or Redshift with admin rights, for example, using `sudo`.
For more information, see https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-perm SolnAdminTag.

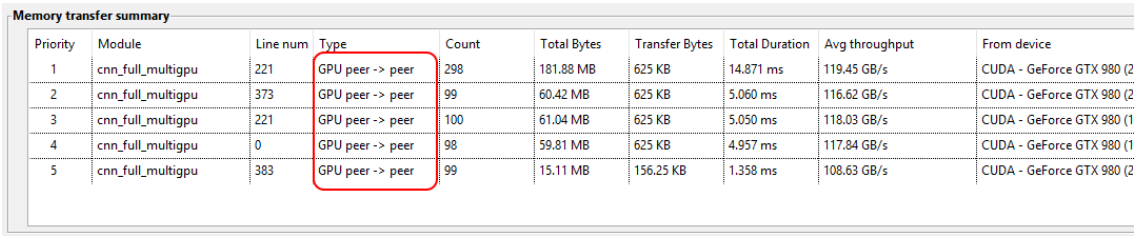
18.3.2 Peer to peer transfers

In multi-GPU configurations, the profiler now displays memory statistics for peer to peer memory copies (i.e., transfers between two GPUs). See the multi-GPU programming documentation for an explanation of the performance

implications related to these peer to peer transfers.



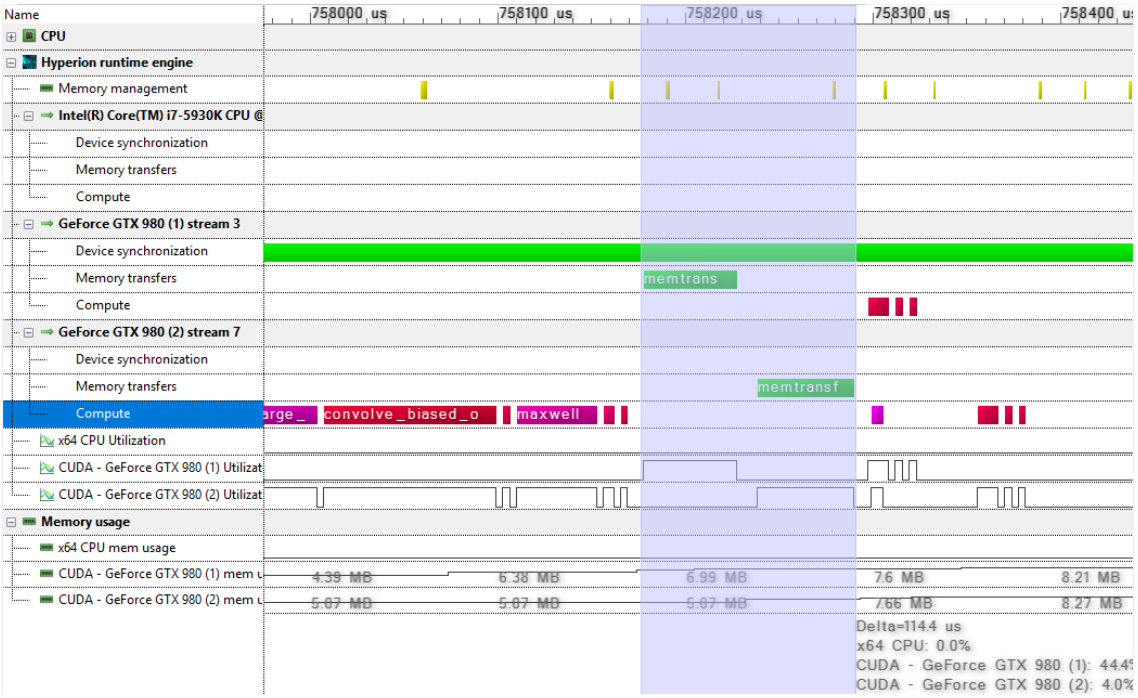
When the profiler indicates memory transfer performance bottlenecks, it is possible to investigate every bottleneck individually, via the memory transfer summary.



Per line of code, the memory transfers are listed, including the following information:

- *Count*: the number of transfers that were measured
- *Type*: the type of transfer: CPU to GPU, GPU to CPU, GPU to GPU or peer to peer
- *Total Bytes*: the total number of bytes for all memory transfers of this type
- *Transfer Bytes*: the number of bytes transferred per transaction (*Total bytes* = *Count* * *Transfer Bytes*)
- *Total Duration*: the total amount of time taken by the memory transfers of this type
- *Average throughput*: the measured transfer speed.
- *From device*: the device from which the transfer originates
- *To device*: the device to which the transfer is done

In systems in which multiple GPUs are not connected to the same PCIe slot, the peer to peer copy between GPUs generally passes the CPU memory. Correspondingly, these peer-to-peer copies cause two transfers: 1) from the source GPU to the CPU host and 2) from the CPU host to the target GPU. In the memory transfer view, the peer to peer copies are listed as one operation, for clarity. In the timeline view however, such peer to peer copies are displayed as dual operations (in green in the screenshot below):



In the screenshot, it can be seen that the peer to peer copy blocks all operations on both GPUs, which is degrades the runtime performance.

18.3.3 GPU event view

The GPU event view shows each operation performed on the GPU(s). This is useful to analyze whether e.g., memory copies and recomputation can be avoided.

ID	Time	Command type	Module name	Kernel/operation name	Device	Duration	Mem size	Shared memory	Grid size	Block size	Occupancy	Dependencies	Code
50	983.866 ms	mem_free			GeForce GTX 98	1.2 us	1024 KB					186854	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
51	983.874 ms	mem_alloc			GeForce GTX 98	1.4 us	1024 KB					186862	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
52	983.901 ms	mem_alloc			GeForce GTX 98	2.1 us	4 MB					186878	parallel_do(size(input),guide,input,tmp,radius,joint_box_filter_hor)
53	983.927 ms	mem_alloc			GeForce GTX 98	2.1 us	2 MB					186879	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
54	983.945 ms	mem_free			GeForce GTX 98	1.5 us	4 MB					186878	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
55	983.954 ms	mem_alloc			GeForce GTX 98	1.7 us	2 MB					186880	parallel_do(size(guide),ab,tmp_ab,radius,joint_box_filter_hor2)
56	983.971 ms	mem_free			GeForce GTX 98	2.1 us	2 MB					186879	parallel_do(size(guide),ab,tmp_ab,radius,joint_box_filter_hor2)
57	983.982 ms	mem_alloc			GeForce GTX 98	1.8 us	1024 KB					186881	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
58	984.036 ms	mem_free			GeForce GTX 98	2.3 us	2 MB					186880	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
59	984.039 ms	mem_free			GeForce GTX 98	0.6 us	1024 KB					186862	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
60	984.040 ms	mem_free			GeForce GTX 98	1.1 us	1024 KB					186881	parallel_do(size(guide),tmp_ab,guide,output,radius,filter_area,joint_box_filter_ver
61	984.049 ms	mem_alloc			GeForce GTX 98	1.5 us	1024 KB					186889	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
62	984.081 ms	mem_alloc			GeForce GTX 98	4.7 us	4 MB					186905	parallel_do(size(input),guide,input,tmp,radius,joint_box_filter_hor)
63	984.110 ms	mem_alloc			GeForce GTX 98	2.1 us	2 MB					186906	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)
64	984.112 ms	kernel_lnch	colortransfor	opt_add1_autokernel	GeForce GTX 98	27.5 us		0 Bytes	16x32x1	32x16x1	100.0%	186781; Read	parallel_do(size(x,(0..1)),out0,$t0,x,$t2,$t3,$t4,$t5,$t6,_autokernel)
65	984.128 ms	mem_free			GeForce GTX 98	1.7 us	4 MB					186905	parallel_do(size(input),tmp,ab,radius,filter_area,threshold,joint_box_filter_ver)

The following types of operations are listed:

- mem_alloc: a memory allocation operation
- mem_free: a memory deallocation operation
- mem_transfer: memory transfer operation
- kernel_lnch_sync: a synchronous kernel launch operation
- kernel_lnch_async: an synchronous kernel launch operation
- device_func_call_sync: a synchronous device function call

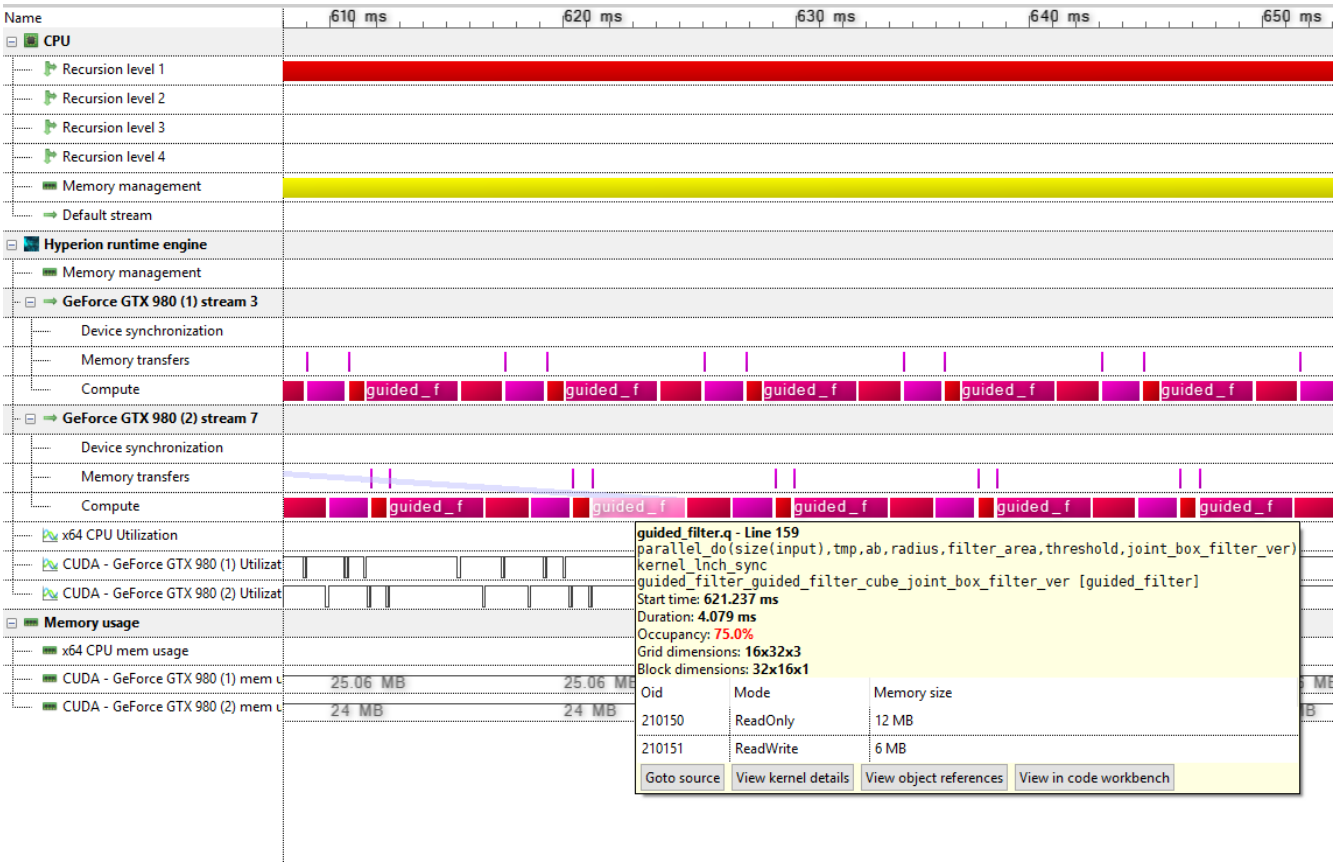
- device_func_call_async: an asynchronous device function call
- sync_event: a synchronization event
- sync_event(global_sched) (Hyperion engine only): indicates when the global scheduling algorithm was run (see multi-GPU programming guide for more information)

Notes:

- In multi-GPU configurations, memory allocations of a single object may be listed multiple times, because an object may use memory of multiple GPUs.
- Both synchronous and asynchronous kernel launches may be listed *out of order* with respect to other events. This is because the kernel start and duration times on the GPU are listed. Due to the nature of CUDA streams, the runtime may assume that an operation is finished at the moment a kernel function is launched, therefore a memory deallocation may occur even before the memory is used in a kernel function.

18.3.4 Timeline view

The timeline view now accurately depicts the kernel execution times and duration. In the following screenshot, it can be seen that both GPUs are (almost) fully utilized:

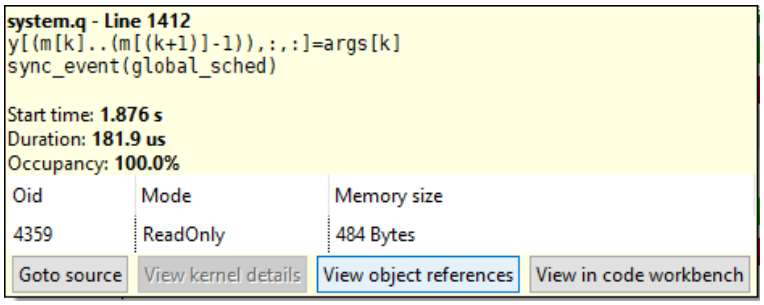


The mouse tooltips now also show a table containing the memory access information of the kernel function.

- *Oid*: a unique object identifier (for a vector, matrix, user-defined object etc.)
- *Mode*: the object access mode (ReadOnly indicates that the kernel function only reads the data, WriteOnly indicates that the kernel function only writes to the data, ReadWrite indicates both reading and writing).

- *Memory size*: the amount of memory taken by this object.

This can be used to track down memory transfers, check the access mode (ReadOnly, WriteOnly) etc. By double-clicking on the object references, the operations to an individual object can be visualized in the GPU events view. For sync_event(global) (which triggers a run of the global scheduling algorithm), the object reference that triggered the scheduling operation can be inspected:

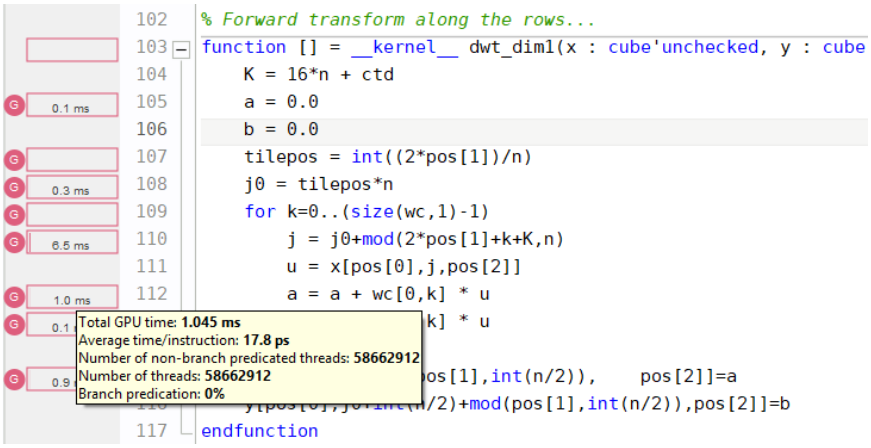


Note that a global scheduling operation can occur:

- when the global command queue reaches its maximal capacity
- when the CPU requires the result of an operation (for example, a kernel function returning a scalar value).

18.3.5 Kernel line information

When profiling a kernel with "collecting line information" enabled, execution information is displayed in the code editor:



Shown is the total execution time of running each line of the specified kernel on the GPU, as well as:

- *Average time per instruction*: the total duration divided by the number of instructions (ignoring the parallelism)
- *Number of threads*: the total number of threads that executed this instruction
- *Number of non-branch predicated threads*: the total number of active threads (i.e., threads that are not disabled due to a false branch condition)
- *Branch predication*: the percentage of threads that were disabled due to a false branch condition.

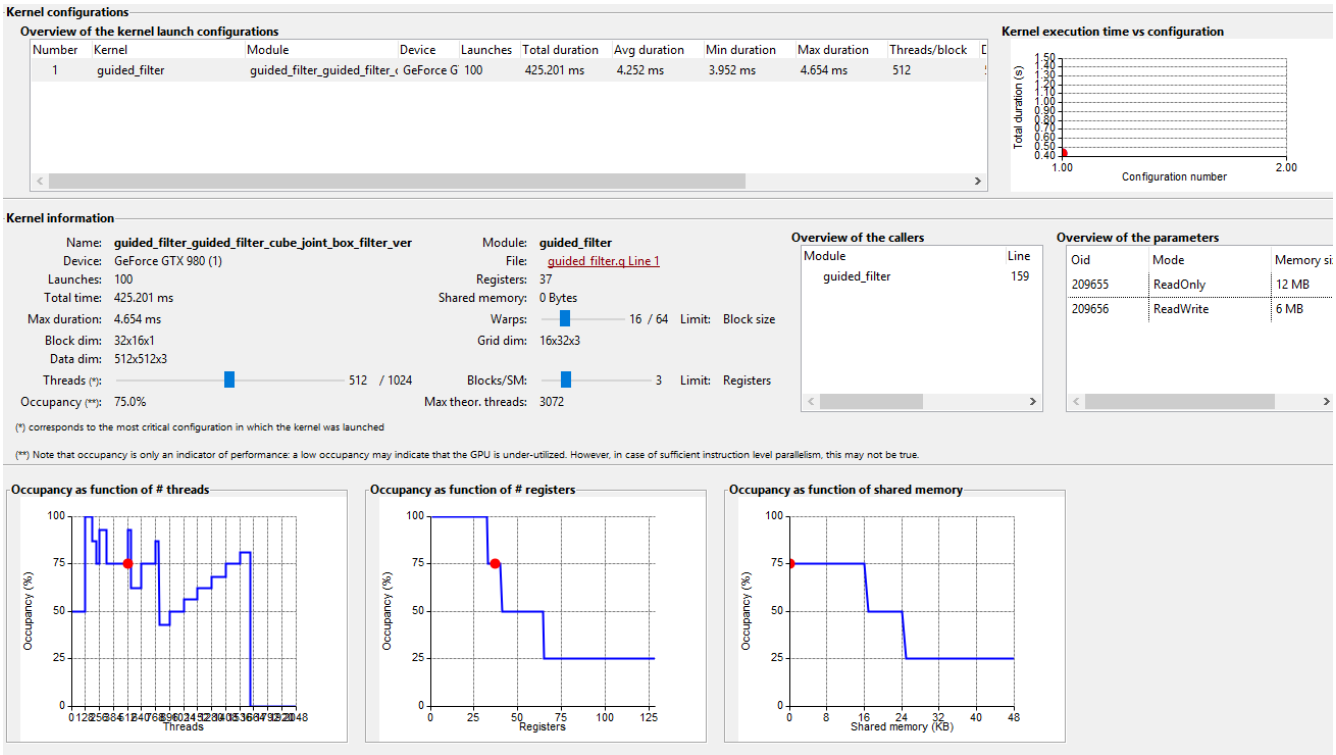
The kernel line information now allows to accurately identify bottlenecks within kernel functions, based on PTX to Quasar source code correlation. Note that due to compiler optimizations, the mapping from PTX to Quasar is not one-to-one. Therefore, the line information may not always correspond to the exact operation that was executed. To improve the correlation, the CUDA optimizations in the Program Settings dialog box can be disabled.

18.3.6 Kernel metric reports

Occupancy report

The occupancy report lists several parameters of the kernel function (block dimensions, data dimensions, amount of shared memory) and displays the calculated occupancy. Occupancy is a metric for the degree of "utilization" of the multiprocessors of the GPU.
Notes:

- The calculated occupancy is not necessarily the real occupancy. By clicking the "single launch configuration" or "multiple launch configuration" buttons, the achieved occupancy can be calculated based on hardware metrics.
- Occupancy is an indicator of performance, but a low occupancy does not necessarily results in a poor performance. It may happen for example that the function units of the GPUs are underutilized (e.g., due to a memory bandwidth bottleneck)
  whereas the occupancy is maximal.



The report displays the kernel execution time and subsequent analysis per launch configuration. A launch configuration is a set of parameter values, such as the grid dimensions, the block dimensions, the amount of shared memory being used). Because the performance of the kernel depends on the launch configuration, it is necessary to separate the measured profiling information according to the launch configuration.
In the bottom, the (theoretical) occupancy as function of respectively the number of threads/block, the number registers and the amount of shared memory is displayed. This indicates how occupancy can be improved:

- The lower the amount of shared memory used by the kernel, the higher the occupancy will be. Of course, shared memory is required to mitigate the lower device memory bandwidth, therefore in practice a trade-off is always necessary.

- Because the set of registers are shared between the different blocks, a lower number of registers may allow more blocks to run concurrently. Therefore, a lower number of registers generally leads to a higher occupancy.

However, the occupancy value is mostly indicates whether the launch configuration is selected to be "efficient" for the particular GPU. Unless the block dimensions are manually specified (e.g. via parallel_do, {!cuda_launch_bounds max_threads_per_block=X}, or {! parallel_for blkdim=X}), the runtime system uses in internal optimization algorithm to select the launch configuration that maximizes the occupancy.
In practice, an occupancy value of 50% (or even in many cases 25%) is sufficient to obtain maximal performance for the selected launch configuration. To gain more insights about the performance of a particular kernel, additional analysis is required.
Max. theoretical threads: calculated as the number of assigned warps × warp size × max active blocks, is the number of threads that is active on the GPU, after the warm-up phase of the kernel. If the occupancy value is smaller than 100%, check if the product of the data dimensions is smaller than max. theoretical threads. If this is the case, GPU multiprocessors are idle because the data dimensions of the kernel are too small.
Overview compute vs. memory and function unit utilization
The compute vs. memory metrics are obtained by reading the hardware counters of the GPU. The following metrics are given:

- *Kernel duration*: Indicates the duration of one single run of the kernel function in the selected launch configuration

- *Achieved occupancy*: Shows the achieved occupancy of the kernel. The achieved occupancy is calculated based on the number of warps (and correspondingly threads) that have *effectively* been executed on the GPU. In the best case, the achieved occupancy is very close to the theoretical occupancy. When the achieved occupancy is significantly lower than the theoretical occupancy, the instruction scheduling issues may have occurred.

- *Warp execution efficiency*: number of eligable warps per cycle compared to the total number of warps. An eligable warp is a warp that can be executed in the next cycle.

- *SM efficiency*: The SM efficiency metric provides information about the effectiveness in issuing instructions.

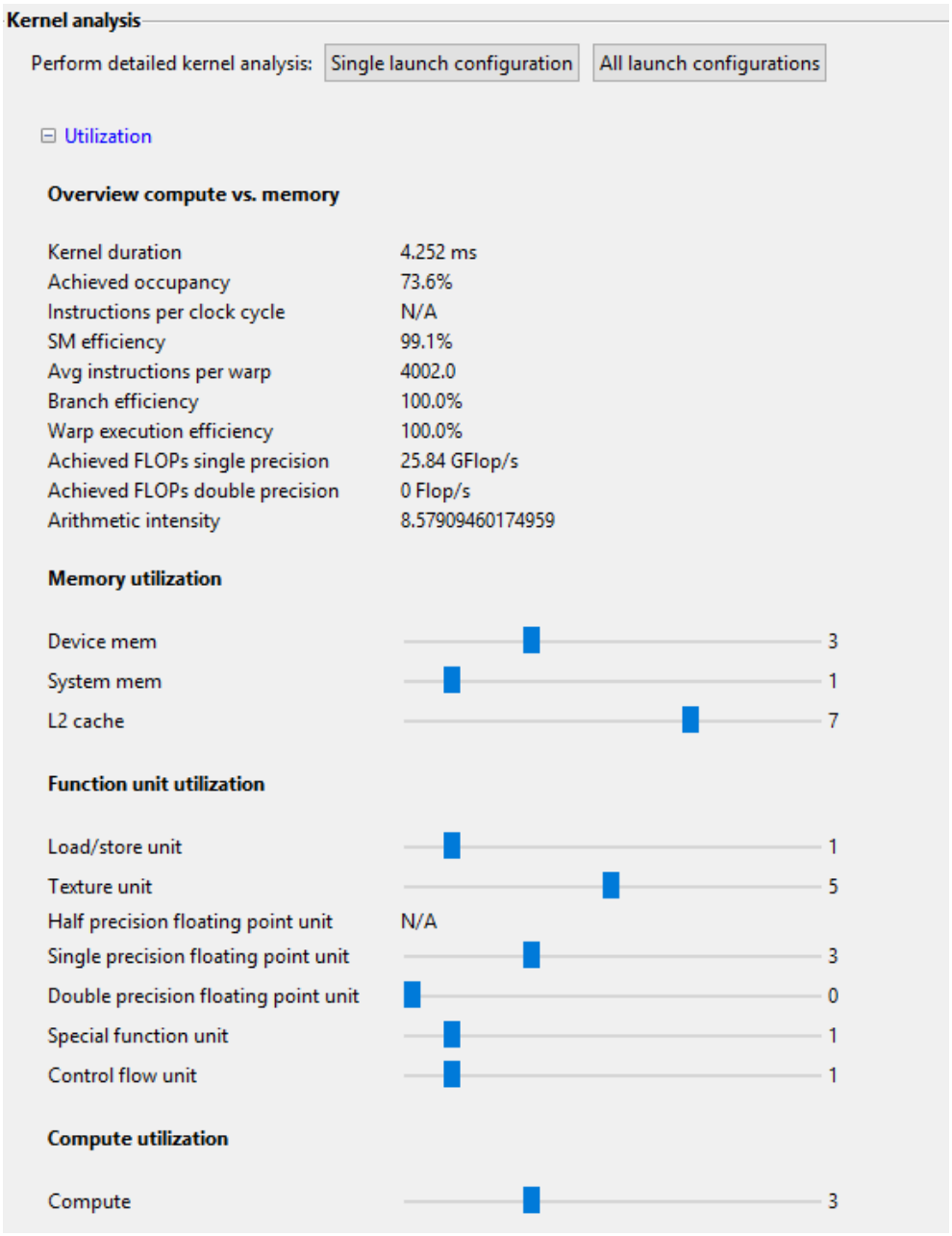
- *Branch efficiency*: The ratio of uniform control flow decisions over all executed branch instructions. When the branch efficiency is maximal, all threads within each warp take the same control path. A low branch efficiency indicates a poor performance due to branch divergence.

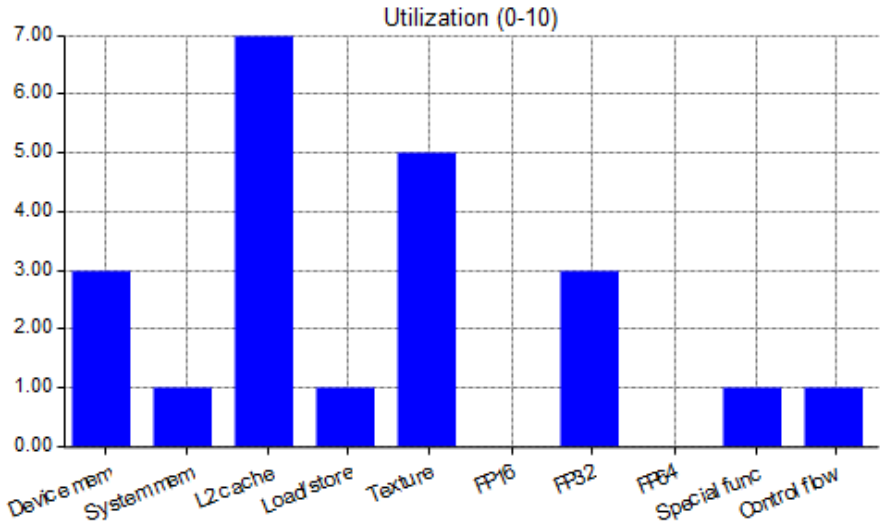
- *Average number of instructions/warp*: Indicates the average number of instructions that were executed per warp.

- *Achieved FLOPs single/double precision*: Gives the number of floating point operations per second (either single or double precision) that this kernel function achieves. It is useful to compare the achieved FLOPs with the maximally achievable FLOPs for the specific GPU (typically 5 TFlops or higher), although memory dependencies typically lead to significantly lower FLOPs.

- *Arithmetic intensity*: a large value indicates that the kernel is compute-bound, while a small values signifies that the kernel is memory-bound (or stalls occur).

In addition, the utilization of the individual function units on a scale from 0 to 10 is displayed (the higher, the better).



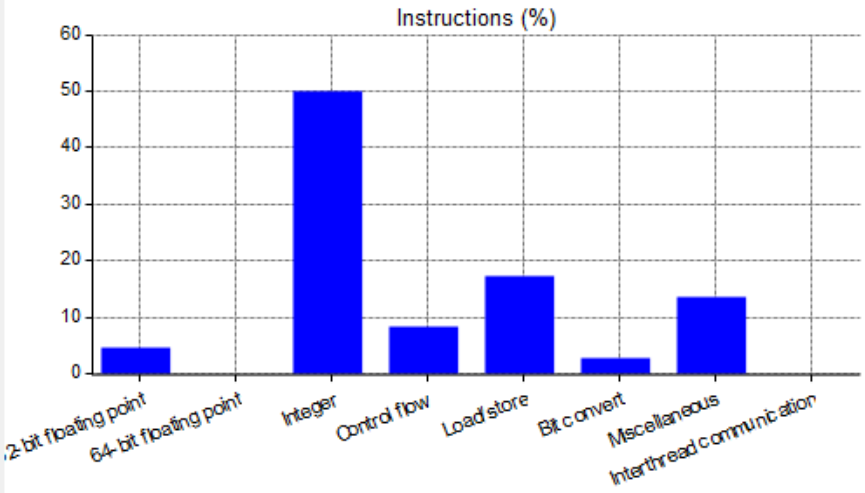


Instruction execution count

The instruction execution count report shows the total number of instructions per type that were executed. Due to the presence of different function and computation units, maximal utilization can be achieved by balancing the operations. For example, when the number of floating point operations is much higher than the number of integer operations, it is useful to investigate whether some parts of the calculations can be done in integer precision.
Note: miscellaneous instructions are warp voting and shuffling operations.

Instruction execution count

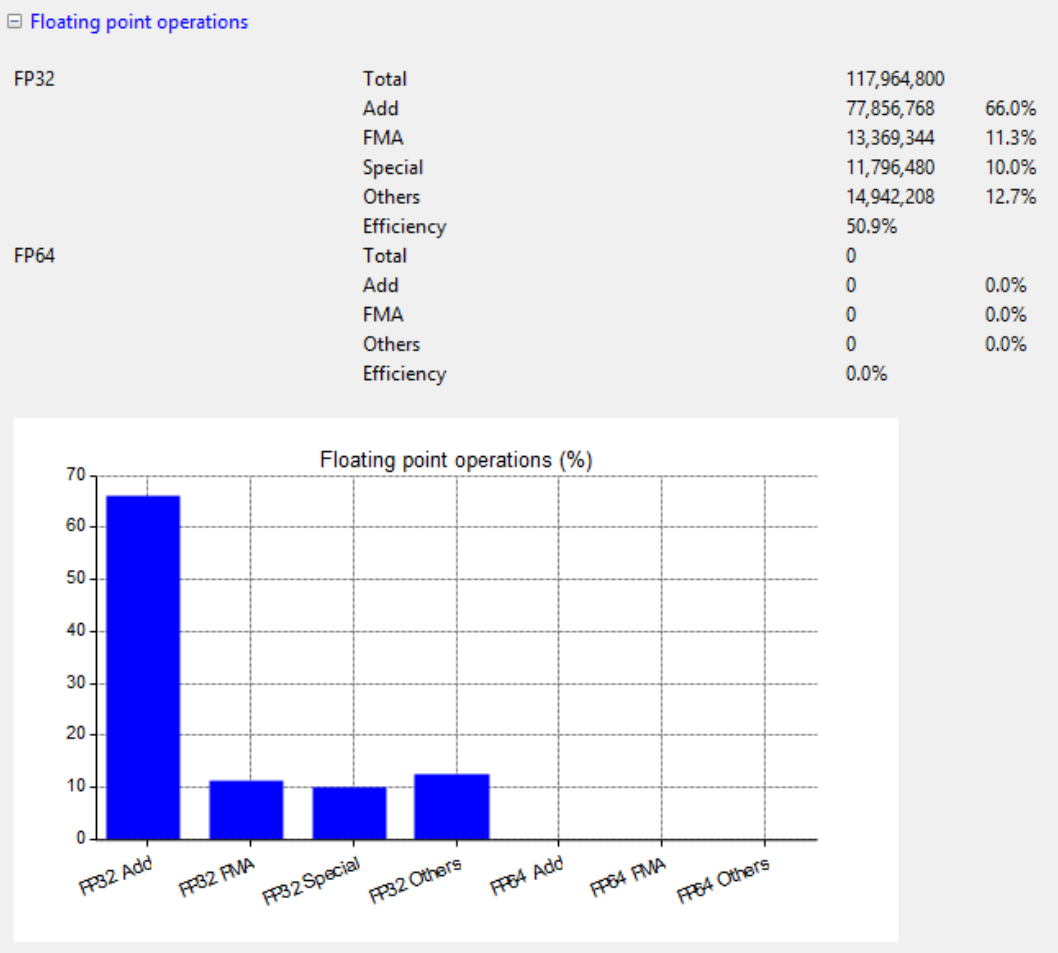
Total instructions	3,147,300,864	
32-bit floating point	148,635,648	4.7%
64-bit floating point	0	0.0%
Integer	1,574,436,864	50.0%
Control-flow	263,454,720	8.4%
Load/store	542,638,080	17.2%
Bit-convert	86,507,520	2.7%
Miscellaneous	424,673,280	13.5%
Interthread com.	0	0.0%



Floating point operations

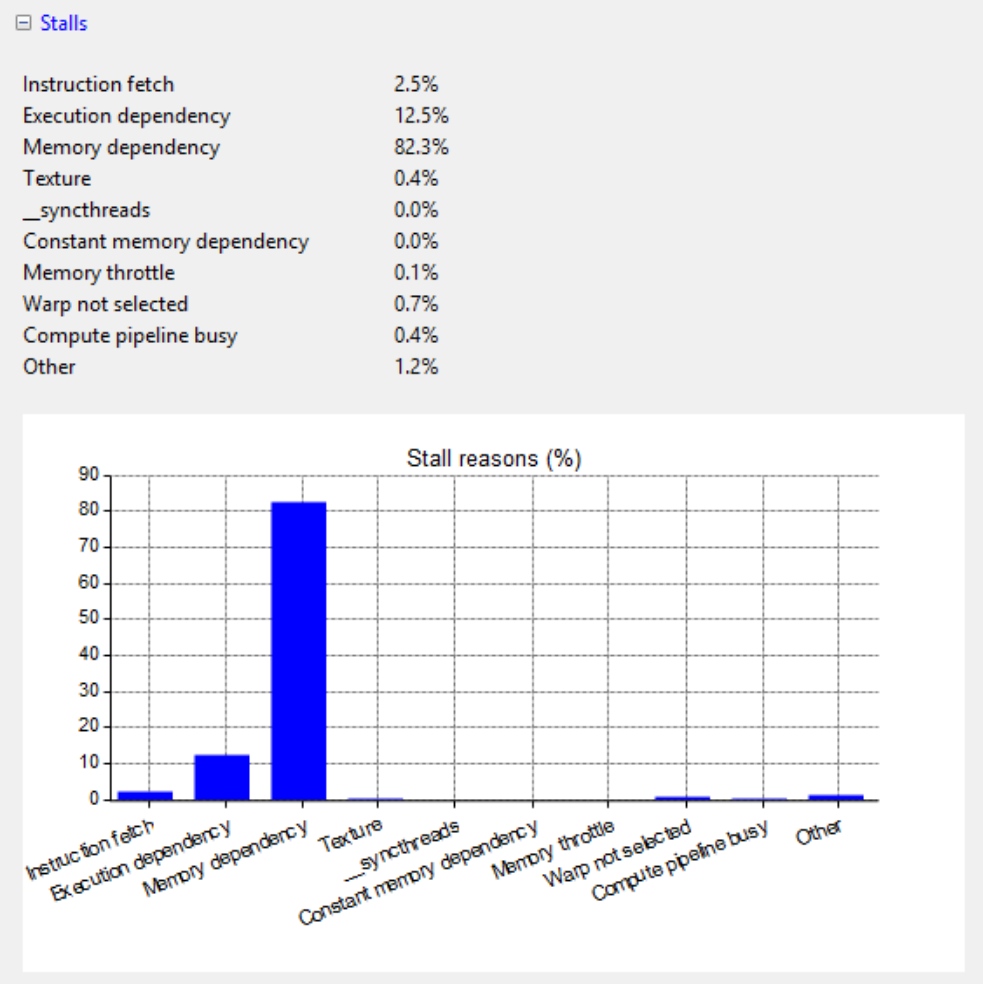
The floating point operations can further be categorized into:

- Add operations
- Multiply operations
- FMA (fused multiply and add): the CUDA compiler combines add and multiply operations to improve the performance
- Special: special mathematical functions (sin, cos etc.)
- Others: other floating point operations (e.g. conversion between integer and floating point).



Stall reasons

Issue stall reasons indicate why an active warp is not eligable.

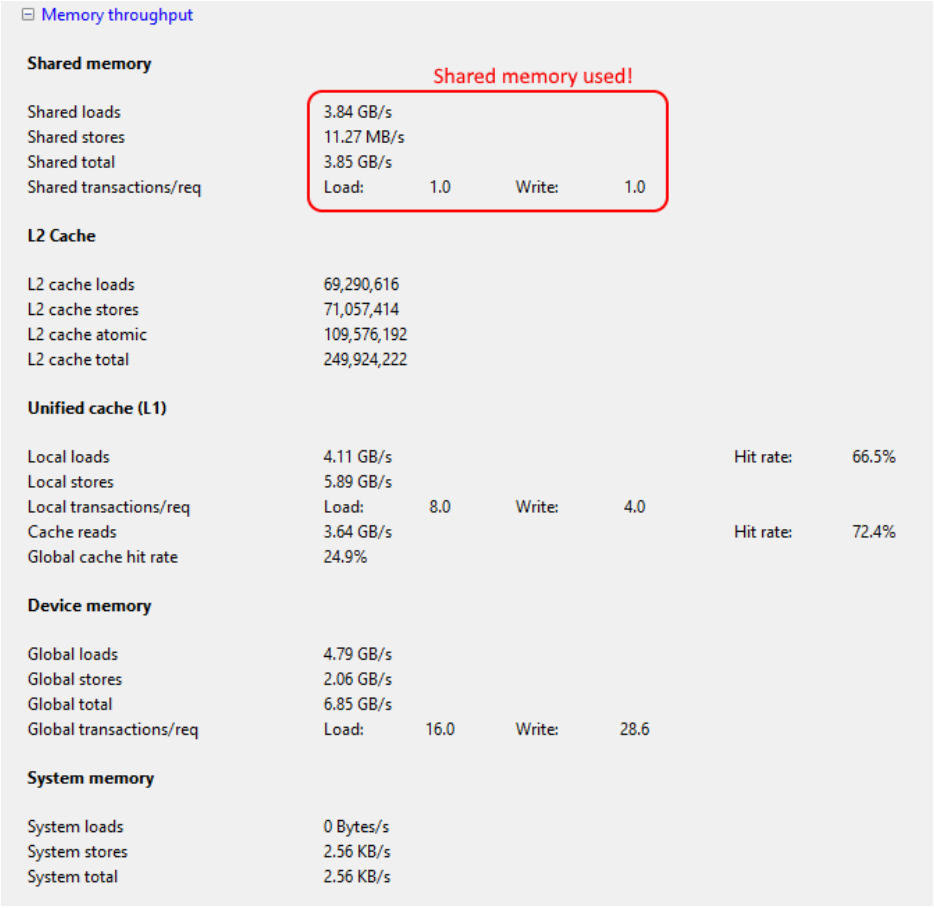


The following possibilities occur:

- *Instruction Fetch*: The next instruction has not yet been fetched.
- *Execution Dependency*: An input is not yet available. By increasing the instruction-level parallelism, execution dependency stalls can be avoided.
- *Memory Dependency*: too many load/store requires are pending. Memory dependencies can be reduced by optimizing memory requests (e.g., memory coalescing, caching in shared memory, improving memory alignment and access patterns).
- *Texture*: too many texture fetches are pending
- *__syncthreads*: too many threads are blocked by thread synchronization
- *Constant*: A constant load leads to a constant cache miss.
- *Compute pipeline busy*: Insufficient computation resources are available during the execution of the kernel.
- *Memory Throttle*: Too many individual memory operations are pending. This can be improved by grouping memory operations together (for example, via the vector data types such as vec(4)).

Memory throughput analysis

The memory throughput analysis indicates the load/store and total throughputs (in bytes/second) achieved for the different memory units (shared memory, unified L1 cache, device memory and system memory), as well as the number of L2 cache operations and the hit rates of the L1 cache.



Also shown are the average number of transactions per memory request. When the number of transactions per request is high, it may be beneficial to group the transactions (e.g., using vector data types such as vec(4)).
The best kernel performance is generally achieved by a good balance between operations using the different memory units. For many kernel functions, this practically means:

- Use of shared memory whenever applicable: see the documentation on shared memory designators in Quasar
- Use of texture memory, especially for readonly memory with a random access pattern.

