The Quasar Computation System: Quick Reference Manual

Jan 6th, 2016

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 and gives access to OpenCL devices.

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] script.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).
- `-nogl`: disables OpenGL support (used for visualization, e.g. the function `imshow`).
- `script.q`: a script 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 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.

1.3 Quasar Programming Language

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. The presence of special parallelizable functions (called kernel functions and device functions). Implementation note: parallelizable functions are compiled natively for the current platform (i.e., using CUDA NVCC, GCC, MSVC or any other C++ compiler).
4. Transparent use of CPU / GPU resources. Essentially, no knowledge on GPU programming is required. GPU functionality is even completely hidden. However, knowledge on *parallel* programming is a must!
5. Minimal runtime overhead. By design, the size of the compiler and runtime system is minimized (to a binary of about 1 MB), as well as the involved runtime overhead.
6. Some improved syntax: lambda expressions, indexing of the results of a function call (like `imread(file)[0..100,0..100]`), ...

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

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, matrices, cubes etc.* Currently 1D, 2D and 3D matrix structures are implemented. 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 2D-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)
```

8. *Operators*: see table 2.1.

   Notes:

   - There are no bit-wise integer operators. For bit-wise integers operations, use the functions `and` (bitwise conjunction), `or` (bitwise disjunction), `xor` (exclusive or), `not` (bitwise negation), `shl` (bit-wise left shift), `shr` (bitwise right shift) instead.

[1] This is in contrast to MATLAB, which uses column-major order (i.e. FORTRAN order).

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

- Within kernel or device functions, the operators +=, -= have a special meaning: they specify *atomic* operations (i.e. these operations are free from 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:

```
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)]
```

[2]Except when the step size is too large, such as `0..2..3 = [0, 2]` or `0..100..10 = [0]`.

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
end

if a==2
endif

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

while expr
        break
        continue
end

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
end
```


Non-uniform ranges can be specified as follows:

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

or more conveniently as:

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

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"
end
```

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 4.8). 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
end
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
end
[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]
        end
    end
    if cname=="hsv2rgb"
       y = hsv2rgb(x)
    else
       error "the specified color transform ",cname, " is not supported!"
    endif
end
```

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
end
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)
```


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 §4.6).

20. *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 its meaning is the same as the C/C++ #include pragma: the content of the referenced .q file is processed in the current .q file at the position of the import keyword. 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 10.1). 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 top of the program!

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 a special 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 compilers, 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 (often denoted by `'??'` in warning/error messages). 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 (see further in section §6). Also the shorthands are listed in the table. Additional primitive types are given in table 2.3.

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

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

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

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

Hence, every vector is a matrix, and every matrix is 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`. E.g., for digital images, `cube` can both represent color images (with dimensions $M \times N \times 3$) and grayscale images (with dimensions $M \times N \times 1$).

Explicitly annotating the types of variables can bring performance benefits in certain cases, although for code simplicity it is advised to only specify the type when necessary. Exceptions are kernel and device functions (see section 2.4.1 and section 2.4.2), which often require explicit typing.

It is possible to check at run-time whether a variable (or intermediate result) has a certain type, using the function `type(A:typename)`. Additionally, the check can be performed at compile-time at any point in the code using the `assert` function, for example:

```
assert(type(variable,"icube"))
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 check fail, a compiler error will be generated. 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")
```

This operation is only performed at runtime, 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.
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

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(.)`	-

Table 2.5: Overview of floating point representations

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

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`.

Lambda expressions (especially those that use closures, see section 4.2) are very powerful in Quasar.

2.2.1 Floating point representation

In Quasar, the internal representation of scalar numbers "`scalar`" (or complex scalar numbers "`cscalar`"), is 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
```

For performance reasons, 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. 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).
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 all approaches that perform parallel computations using floating point numbers. The example "Accurate sum" gives more information in this issue (see section 11.7).
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 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]
        end
    end
end
```

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.

2.2.3 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 the highest possible dimension that can currently be specified in Quasar is `cube{n}`. This is useful for generic specialization purposes (see further in section §6).

2.2.4 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
end
```

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
end
```

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
end
```

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
end
```

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 : ??
end

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
end
```

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]
            end
        end
        im_out[m,n,0..2] = a
    end
end
```

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]
        end
    end
    im_out[pos[0],pos[1],0..2] = a
end
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. 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. through `assert(type(B,"vec"))` statements. 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
    end
end
```

There are some scenarios that currently cannot be solved by the auto-parallelizer (for example situations in which shared memory is necessary). 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 Custom writing of parallel code

Any user should read this section, as this section is quite fundamental! Here, we describe two usages of writing parallel code:

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

Most algorithms can be efficiently implemented using the "basic usage" approach. The advanced usage generally brings no further savings in terms of computation time (except for algorithms that can benefit from storing a lot

of intermediate values). The advanced usage consists of synchronization, dealing with data races, sharing memory across multiprocessors.
For beginning users, it is advised to get acquainted first with the basic usage techniques, before considering the advanced usage.

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 than host functions, not only due to their parallelization. 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
end
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 arguments need to be *explicitly* typed. Recall that untyped arguments (or arguments without type) in Quasar are denoted by `??`. Untyped arguments are not allowed because these arguments can not be mapped onto a device architecture.
Kernel functions differ from "host" functions, in the sense that in addition to the standard data types, some special data type specifiers are allowed:

- `vec`x: with x=1,2,...,32 corresponds to a vector of length x.
- `ivec`x: with x=1,2,...,32 corresponds to an integer vector of length x.
- `cvec`x: with x=1,2,...,32 corresponds to a complex-valued vector of length x.
- `int`: integer data type

The value x should be seen as an extra length constraint on the corresponding vector, and this length information is used by the compiler for type inference purposes.
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.

When declaring parameters, for vectors of length ≤ 32, it is more efficient to add the length explicitly in the type, as in `vecx`, `ivecx`, `cvecx`, with x=1,2,...,32. This is because vectors, with a length that is known at compile-time, are treated in a special way: they are grouped together requiring less memory read/write requests, or they are implemented using SIMD instructions if the underlying back-end compiler supports them. Note that inside `__kernel__` functions, it is recommended to use integers instead of scalars (when possible). This may yield a speed-up of about 30% using the CUDA computation engine. 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 considered to be 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 4.5.3, but this feature is only for specific use-cases). Instead, 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`:

- `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 (advanced users only, see section 2.4.4)
- `blkidx` (of type `int`, `(i)vecx`): the block index (advanced users only, see section 2.4.4)
- `blkdim` (of type `int`, `(i)vecx`): the current dimensions a block (advanced users only, 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 (advanced users only, see section 2.4.4)
- `warpsize` (of type `int`): the warp size of the device (advanced users only, see section 2.4.4)

The `parallel_do` function performs 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])
        end
    end
end
```


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]`×`dimensions[1]`×`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 computer will explode if not used properly): specifies no bounds checking on the input/output data. In case of access outside the data boundaries, a runtime error will be generated (or the program may crash). 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.

The default access modes are currently `'safe` (inside kernel/device functions) and `'checked` outside of kernel/device functions (for performance reasons). In case the output of the program depends on the access mode, it is best to explicitly indicate the access mode in the code.
Finally, there are some rules w.r.t. the calling conventions for kernel functions:

- Kernel functions can not have optional arguments.
- A kernel function *cannot* call a "host" function.
- A kernel function *cannot* evaluate a lambda expression.
- 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 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)`: 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
```

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. For the above definitions:

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

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

```
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]])
end

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])
end

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

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*. Therefore, the functions `ones` and `zeros`, can only be used for allocating vectors (not cubes or matrices), and the maximum length of a vector is 32. Second, the length of the vector must be constant. It is hence not possible to allocate dynamic memory (e.g. `zeros(some_variable)`). This may however change in a future version of Quasar.

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 can (currently) only be passed using the kernel function arguments. For example:

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

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:

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

   Global memory can reside either in CPU memory, GPU memory or both. However, 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 (do not expect 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 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. It is generally

believed that CUDA uses some sort of space filling curves[3] for optimizing the data layout. See section 9.2 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! When Quasar notes that the amount of local memory of a kernel function exceeds 64 (or 128) bytes, the number of threads spawned is decreased (resulting in a computational cost if some of the multi-processors get unemployed by this measure). 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.

2.4.4 Advanced usage: shared memory and synchronization

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
end
```

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. Not properly using `syncthreads` may result in a deadlock.

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

Shared Memory Shared memory, which is visible to all threads of a kernel function within one block, can be allocated using the function `shared(.)` or `shared_zeros(.)`. 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
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: when using `shared_zeros`, always put a `syncthreads` command at the end of the allocations (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
    end
    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!
end
```

There are however two caveats when using shared memory:

1. For the CUDA computation engine, the amount of shared memory is limited to either 16K or 48K. 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.

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 chapter 5):

```
3. 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: $8 \times 20 \times 4 \times 4$ bytes $= 2560$ bytes. The compiler then gives the following message:

```
Information: shared_mem_test.q - line 17: 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]
        end
    end
    bins[blkpos] = val
    % step 2 - synchronize all threads using this barrier
    syncthreads
    % Now it is safe to read from the variable bins
end
```

Specifying the block size The block size (`blkdim`) can be manually specified, through the function `parallel_do`. This is done as follows:

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

where `dims` and `sz` are both vectors of length 1, 2 or 3. 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 even lower, due to register usage and other internal details.

- 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.
- Also take into account that the actual maximum number of threads per block can be significantly lower than 1024, because this number is also affected by the amount of shared memory that is used by the kernel function. In case you guess and the number is too high, CUDA will generate an error like `cudaErrorUnknown` or `cudaLaunchOutOfResources`.

Note that 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.

Block size not specified: what happens? In case the block size is not specified, it can be accessed from the kernel function through the blkdim argument (at least, this argument should 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 or 32×16. 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 amount of shared memory that is used by the kernel function, as well as the number of registers. Furthermore, it always returns a block size that the hardware can deal with.

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. The following example illustrates a scenario in which this is not the case:

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

If Quasar would run the above program directly, the GPU would return `errorInvalidValue`. To ensure proper functioning of the program, Quasar will internally pad 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
end
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 "unemployed" by this measure. The algorithm for computing the data padding sizes is quite sophisticated. This algorithm attempts to minimize the number of "unemployed" threads.

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`. In contrast to other function parameters like `blkdim`, `warpsize` is always an integer.

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.
Types in Quasar can be classified into the following categories:

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

Class #2 types can be parametric, 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.
There also some derived types, which can be expressed directly in terms of the above types. Note that the following definitions are already recognized by Quasar, 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). Therefore 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 other types than class #3.
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
end
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 4.5.2).

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

3.3 Class / user defined type (UDT) definitions

As already explained, it is possible to define user-defined types. Although Quasar currently does not support class member functions, inheritance or interfaces, there are some ways to simulate this behavior.

Class member functions To define class member functions, the special `reduction` keyword (see section 4.7) 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.

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_ref`	reference	reference	N/A	N/A
`??`	reference*	reference*	N/A	N/A

*: unless the underlying value is scalar

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



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 : [?? -> ??]
end

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])
```

3.4 Passed by reference / Passed by value

In the current implementation of Quasar, the semantics of whether a given variable is passed to a function by reference or by value, depends not only on the type of the variable, but also on whether it is passed to a host function, device function or kernel function. This is due to some complications in the run-time system (the current scheme implements the option that is computationally the fastest). The same also applies to reference copies of the object, for example through the assignment `a = b`. An overview of the variable passing conventions is given in table 3.1.

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
for k=0..numel(v)-1
        a += v[k]
end
```

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]
end
```

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.

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" ->
        ...
    end
end
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
    end
    y = uninit(size(x))
    parallel_do(size(x), my_kernel)
end
```

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)
end
```

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]))
end

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
end

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!

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 some extent) to balance the work between CPU/GPU. As programmer, you know whether the code will be run on GPU/CPU, in the following way:

- 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.

4.4 Nested parallelism

A new feature (since Jan 2014) is that `__kernel__` and `__device__` functions can now also use the `parallel_do` (and `serial_do`) functions. The top-level host function may for example spawn 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 (or will be) mapped onto CUDA dynamic parallelism on Kepler devices such as the GTX 780, GTX Titan. (Note: requires one of these cards to be effective).
- 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.
- The high-level matrix operations from the previous section are automatically taking advantage of the nested parallelism.

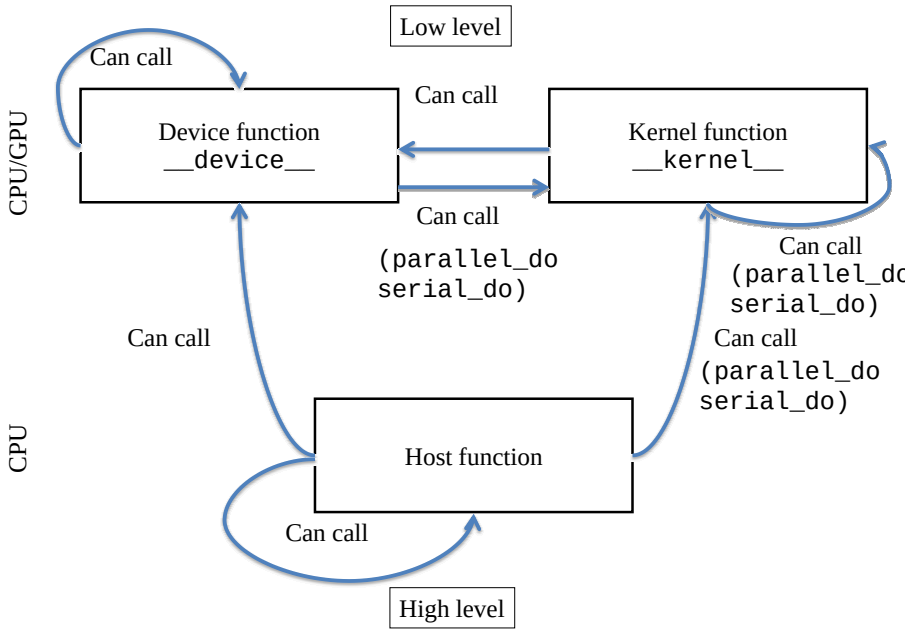

Figure 4.1: Relationship between the different function types in Quasar.

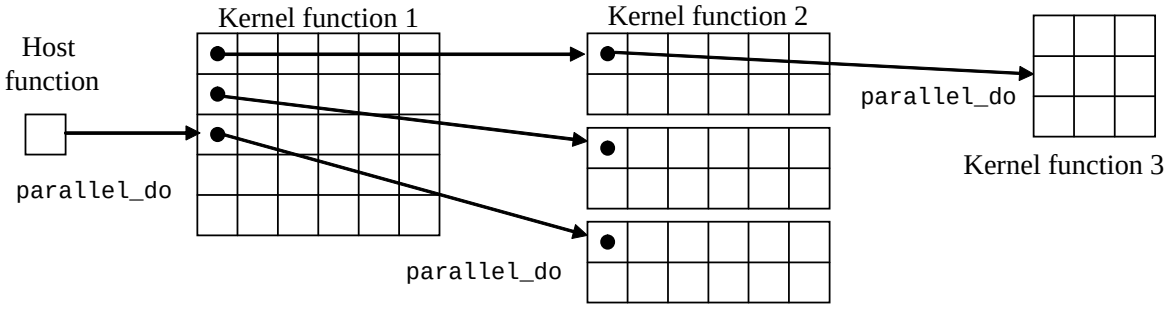

Figure 4.2: Illustration of nested parallelism.

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.

Limitations:

- Nested kernel functions may not use shared memory (they can access the shared memory through the calling function however), 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`).

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)
    if isscalar(x)
        y = conj(x)
    elseif isreal(x)
        y = transpose(x)
    elseif iscomplex(x)
        y = conj(transpose(x))
    else
        error "(herm_transpose) invalid type:",type(x),"!"
    endif
end
```

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

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 4.7. Note that the function overloading has the following restrictions:

- all function overloads must be defined inside the same module.
- all function overloads must be defined at the global scope (i.e. not inside another functions).
- only "host" functions can be overloaded, not `__kernel__`, `__device__` 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.

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 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.

4.5.2 Functions vs. lambda expressions

In Quasar, a function is defined as follows:

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

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:

- Functions can have *optional* arguments, whereas lambda expressions cannot.

Table 4.1: Comparison of functions and lambda expressions

	Function	Lambda expression
Optional arguments	Yes	-
Multiple output arguments	Yes	-
Supports overloading	Yes	-
`__kernel__`, `__device__`	Yes	Yes
Function closures	Yes	Yes
Function handles (`(??,??)->??`)	Yes	Yes
First-class citizen	Yes	Yes
Can contain nested functions	Yes	-
Can contain nested lambda expressions	Yes	Yes

- 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.
- Functions can have multiple output arguments, while lambda expressions only have one output argument (note: this may change in a future version of Quasar).

On the other hand, the definition of lambda expressions is more compact, and lambda expressions are more suitable for inlining 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.

4.5.3 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
end
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
end
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 11.6) is more efficient!

4.6 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]
    end
end
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)
end
```

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

4.6.1 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]
    end
end

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

Note that variadic *kernel* functions are currently not supported.

4.6.2 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)
end

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

helper(myprint)
```

4.6.3 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])
end
```

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]
end
```

And we wish to pass both output values to one function

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

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 fixed-length vectors `vecX`, `cvecX`, `ivecX` (this means: the compiler should be able to determine the length of the vector statically).
- 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.

4.6.4 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] = ...
end
```

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())
```

4.7 Reductions

Quasar implements a very generic compile-time graph reduction scheme, that is - as far as we are aware of - not yet found in other programming languages. 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)
```


After 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 done 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.

4.7.1 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)'
```

4.7.2 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 4.7.3.
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)
```

4.7.3 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 13.1). 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.
end
```

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 4.7.2), 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.

4.7.4 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.

4.8 Partial evaluation and recursive lambda expressions

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])
end
```

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])
end
```

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, 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).

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)
end
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)
end
```

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"`.

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

- '`$assump(variable)`' lists all assertions that are currently known about a variable, including the implicit type predicates that are obtained through type inference. Note that the result of '`$assump`' is an expression, so for visualization it may be necessary to convert it to a textual representation using '`$str(.)`' (to avoid the expression from being evaluated).
- '`$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])
end
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
            ])
end
```

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])
end
```

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])
end
```

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 Type classes

Type classes allow the type range of the input parameters to be narrowed. 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 that exist in Quasar. Type classes are also useful for defining reductions:

```
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 4 times, one for each element.

6.2 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)
    ...
end
```

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)
    ...
end
```

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
    end
    new_list[pos] = item
end
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)
end
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
end
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`'.

6.3 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])
    ...
end

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`.

6.4 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
end
```

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 '`pointer`'. This problem can be solved by using parametrized types:

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

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
end
```

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.

6.5 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 chapter 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
end
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
end
```

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 9.2) in a more general way. Below, the implementation 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]
    end
    y[pos] = total
end

% 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

```
function [] = __kernel__ denoising(x : mat, y : mat)
    assert(x[pos]>0)
    y[pos] = soft_thresholding(x[pos], 10)
end
```

6.7 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]
        end
    end
    y[pos] = sum
end
```

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]]
    end
    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]]
        end
        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]]
    end
    y[pos] = sum*(1.0/9)
end
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]
        end
    end
end
```

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)
end
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])
        end
    end
end
```

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]
        end
    end
end
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)
end
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
end
```


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
end
```

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)
end
```

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)
end
```

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)
end

% The method point.setLocation
function [] = setLocation(self : ^point, x, y)
    self.x = x
    self.y = y
end

% The method point.translate
function [] = translate(self : ^point, dx, dy)
    self.x += dx
    self.y += dy
end

% The method point.tostring
function y = tostring(self : point)
    y = sprintf("(%f,%f)",self.x,self.y)
end
```

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
end
```

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
end
```

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
end
```

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
end
```

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)
end

% Constructor with int parameter
function y = stack[T](capacity : int)
    y = stack[T](tab:=vec[T](capacity), pointer:=0)
end

% Constructor with vec[T] parameter
function y = stack[T](items : vec[T])
    y = stack[T](tab:=copy(items), pointer:=0)
end
```

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
end
```

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
end

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
end

type duck : bird
    ...
end
```

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
end

type point : geometry
    x : scalar
    y : scalar
end

type line : geometry
    p1 : point
    p2 : point
    x1 : scalar
    y1 : scalar
    x2 : scalar
    y2 : scalar
end

type circle : point
    radius : scalar
end

function y = distance_from_origin(p : ^point)
    y = sqrt(p.x^2 + p.y^2)
end

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. They 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) -> ??]
end

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. 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]))
    end
    _out = uninit(size(x))
    parallel_do(size(x),_out,x,w,opt__auto_optimize1_kernel)
end
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 Serializable and parallelizable loops

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. There are however a number of restrictions to the Quasar code:

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. The for-loops must be *ideal* (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
    end
end
```

   This is an example of a non-ideal loop:

```
for m=0..size(x,0)-1
    for n=m..size(y,0)-m
    end
end
```

3. Only the `for` keyword is recognized (not `repeat` or `while`, unless these keywords are used to mimick a for-loop).

4. When host (i.e. non kernel/device) functions are called from a for loop, the for loop is not eligible for parallelization/serialization.

5. 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, for the simple reason that these functions often require low-level access to the block position (`blkpos`) and dimensions (`blkdim`).

8. In case (non-fixed length) vectors or matrices are *constructed* inside the for-loop, it is required that the compiler setting "enable dynamic kernel memory" is enabled (see further in section §8.3).

In case one of these conditions are violated, a warning message is generated (see section 13.1.1).
The ALP can be configured using the pragmas/code attributes (see section 13.2):

```
#pragma loop_parallelizer (on|off)
{!parallel for}    % or...
{!serial for}      % or...
{!interpreted for}
```

The loop parallelizer pragma allows to completely disable the ALP (which is not recommended!). `{!parallel for}` forces the next loop to be parallelized. In case of failure, a detailed compiler error will be generated, giving the user the opportunity to check the conditions of the ALP. `{!serial for}` forces the next loop to be *serialized* (i.e. executed in single-threaded mode on the CPU), even when the loop is parallelizable. Finally, `{!interpreted for}` forces the loop to be interpreted. This comes at a computational cost, but can still be useful for e.g., debugging purposes
The Quasar compiler will generate warnings when the automatic for-loop parallelizer (serializer) is turned off.

```
For-loop will be interpreted. This may cause performance degradation. In case no matrices
    are created inside the for-loop, you may consider automatic for-loop parallelization/
    serialization (which is now turned off).
```

Consider the following example:

```
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
        end
    end
end
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). This way, the programmer can improve the code.
In the next section, dynamic kernel memory will be discussed, which greatly improves the ALP by allowing functions such as `zeros`, `uninit`, `transpose`, `reshape`, to be used from within parallel loops or kernel/device functions.

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 at the Quasar compiler and runtime sides. 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 small-sized (`3x3`) to medium-sized (e.g. `64x64`) 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.
- 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..9, 2]`) in order to extract a sub-matrix. The slicing operations then take the current boundary access mode (e.g. mirroring, circular) into account.

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
end

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

#pragma force_parallel
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
    end
end
```

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**

   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 minimum size is 1024 bytes (everything smaller is rounded up to 1024 bytes)
   - The maximum size is 32768 bytes

   For larger allocations, please see the *cooperative memory model*. The minimum size also limits the number of objects that can be allocated.

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.

   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:

   - The Quasar memory allocator uses locking (to limited extend), so simultaneous memory allocations on all processor cores may be expensive.

- The memory is disposed only when the kernel function exists. This is to internally avoid the number of callbacks from kernel function code to host code. Suppose that you have a `1024x1024` grayscale image that allocates 256 bytes per thread. Then this would require `1GB` of RAM! In this case, you should use the cooperative memory model (which does not have this problem).

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 now 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
```

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` types for vectors of length 2 to 16 whenever your algorithm allows it. 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 32 threads. A group of 32 threads must execute (every instruction) together. 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 block position (`blkpos`), the controlling condition should be written so as to minimize the number of divergent warps.

8.4 Atomic operations inside parallel loops

An often recurring programming idiom is the use of atomic operations for data aggregation (e.g. to calculate a sum). In the most simple form, this idiom is as follows (called the *JDV variant*):

```
total = 0.0
#pragma force_parallel
for m=0..511
    for n=0..511
        total += im[m,n]
    end
end
```

However, it could also be more sophisticated as well (called the *HQL variant*):

```
A = zeros(2,2)
#pragma force_parallel
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]
end
```

Here, the accumulator variables are matrix elements, also multiple accumulators are used inside a for loop.
Even though this code is correct, the atomic add (+=) may result in 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 is a spin-lock that basically serializes all the memory write accesses). The performance is often much worse than performing all operations in *serial*!
The obvious solution is the use of shared memory, thread synchronization in combination with parallel reduction patterns (see section 11.6). 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, under the following conditions:

- All accumulator expressions (e.g. `total`, `A[0,0]`) should be 1) variables, 2) expressions with constant numeric indices or 3) expressions with indices whose value does not change during the for-loop.
- The accumulator variables should be **scalar numbers**. Complex-valued numbers and fixed-length vectors are currently not (yet) supported.
- Only **full dimensional** parallel reductions are currently supported. A sum along the rows or columns can not be handled yet.
- 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. This, together with the impact on the occupancy (due to high number of registers being used) might lead to a performance degradation.

8.5 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
- `$inline(lambda(...))`: performs inlining of lambda expressions/functions
- `$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.

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 4.7.4), 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

Advanced GPU concepts

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:

- '`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 needs to be stored in the *texture memory* (see further, in section 9.2).

- '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 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]
    end
    y[pos] = sum
end
```

Version with constant memory:

```
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]
    end
    y[pos] = sum
end
```

Version with constant texture memory for f:

[1] For more information, see Kepler tuning guide.
[2] For more information, see Maxwell tuning guide.


```
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]
    end
    y[pos] = sum
end
```

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]
    end
    y[pos] = sum
end
```

Version with HW textures (see section 9.2):

```
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]
    end
    y[pos] = sum
end
```

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 §11.5), the performance can even further be improved to 85 ms.

9.2 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

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 9.1: Texture memory limitations

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.

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 9.1 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]
end

% Kernel function, using hardware textures
function [] = __kernel__ interpolate_hwtex (y:mat, x:mat'hwtex_linear,
          scale:scalar, pos:ivec2)
    y[pos] = x[scale * pos]
end
```

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 §9.5), 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.

9.3 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. 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).

9.4 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]
        end
        y[pos[0],pos[1],0..2] = sum
    end

    y = uninit(size(x))
    parallel_do (size(y,0..1), x, y, fc, n, kernel)
end
```

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).

9.5 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
end
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
    end
    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
end

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 (!)

9.6 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.
Originally, the Quasar compiler either reserved 0K or 32K shared memory per block, depending on whether the kernel function allocated shared memory. Shared memory is dynamically allocated from within the kernel function. This actually deteriorates the performance, because N*32K < 48K requires N=1. So there is only one block that can be launched simultaneously.
In the latest version, the compiler is able to infer the total amount of shared memory that is being used through the logic system (see chapter 5). For example, 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. Often the arguments of the function `shared` are non-constant. In this case you can use assertions.

```
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
```

The assertion also allows 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.
Note: the compiler does not recognize yet all possible assertions that restrict the amount of shared memory. For example `assert(numel(blkdim)<=1024); x = shared(blkdim)` *will not work yet. In the future, more use cases like this will be accepted.*

9.7 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).

CHAPTER 10

Best practices

10.1 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)
end
```

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
end
% 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 §4.6):

```
function [] = main(arg1, arg2, ...other_args)
    print arg1
    print arg2
    for i=0..numel(other_args)-1
        print other_args[i]
    end
end
```

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.

10.2 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.

10.3 Loop parallelization

Suppose you want to parallelize several nested loops, such as:

```
for m=0..M-1
    for n=0..N-1
        for k=0..K-1
            for l=0..L-1
                ...
            end
        end
    end
end
```


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
        ...
    end
end
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. The auto parallelizer is active by default, but can be toggled on/off using the pragma:

```
#pragma loop_parallelizer(on|off)
```

For more info, see section 13.1.

10.4 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
end
```

Alternatively, the matrices are passed by reference (see 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
end
```

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 4.6.3, the output arguments of functions can be chained using the spread operator `...` For example,

```
function [band1_out, band2_out] = process(band1, band2)
    ...
end
process(...subband_decomposition(input))
```

This way, it becomes unnecessary to store the output arguments in intermediate variables.

10.5 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 $-1,175\times10^{-38}$ -1,175×10^{-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 13.1). 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)
end
```

In this example, the assertion obviously fails. Quasar breaks with the following error message:

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


CHAPTER 11

Parallel programming examples

This section contains a number of useful parallel programming examples together with an explanation.

11.1 Gamma correction [basic]

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.

11.2 Fractals [basic]

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
    end
    im[pos] = n-log2(log(abs(z))/log(N))
end

x = zeros(768,768)
parallel_do(size(x),x,10,complex(-1.42),512,mandelbrot_fractal)
imshow(x,[])
```

11.3 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
    end

    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)
    end

    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)
end
```

11.4 2D Haar inplace wavelet transform using lifting [basic]

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
    end
    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
    end

    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)
    end
end
```

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
    end
    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
    end

    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)
    end
end
```

11.5 Convolution [advanced]

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]]
    end
    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]]
        end
        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]]
    end
    y[pos] = sum*(1.0/9)
end
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 13.2) 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 9.6).

11.6 Parallel sum [advanced]

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
    end
    bins[blkpos] = val
    % step 2 - reduction
    syncthreads
    bit = 1
    while bit < blkdim
        if mod(blkpos, bit*2) == 0 && blkpos+bit<blkdim
            bins[blkpos] += bins[blkpos + bit]
        endif
        syncthreads
        bit *= 2
    end
    % write output
    if blkpos == 0
       y = bins[0]
    endif
end
```

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 stored in the kernel output argument `y` (see section 4.5.3).
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
    end
    bins[blkpos] = val
    syncthreads % step 2 - reduction
    bit = 1
    while bit < blkdim
        if mod(blkpos,bit*2) == 0
            bins[blkpos] = acc(bins[blkpos], bins[blkpos + bit])
        endif
        syncthreads
        bit *= 2
    end
    y = bins[0] % write output
end
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).

11.7 A more accurate parallel sum [advanced]

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 [] = __kernel__ r_sum_kernel(x : vec, y : 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
        end
        s[blkpos] = sum
        % step 2 - reduction
        syncthreads

        % now sort all bins from large to small magnitudes
        bit = 1
        % use regular summing
        while bit<blkdim
            if and(blkpos,2*bit-1) == 0 && blkpos+bit<blkdim
                t = s[blkpos] + s[blkpos+bit]
                s[blkpos] = t
                syncthreads
            endif
            bit *= 2
        end
        if blkpos==0
            y[0] = s[0]
        endif
    end
    y = r_aggregator(x, r_sum_kernel)
end

% 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)
    if iscomplex(x)
        y = complex(uninit(1))
    else
        y = uninit(1)
    endif
    parallel_do([[1,BLOCK_SIZE],[1,BLOCK_SIZE]],x,y,nblocks,kernel)
    z = y[0]
end

% Define a reduction to replace the summing function by our
% "improved" implementation.
reduction (x : cube)    -> sum(x) = r_sum(x)
```

11.8 Parallel sort [advanced]

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
        end
        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
                end
                syncthreads
                j /= 2
            end
            k *= 2
        end

        % 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
        end
    end

    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)
end
```

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.

11.9 Matrix multiplication [advanced]

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 - v2.0
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
        end
        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
        end

        % write output
        if blkpos[0] == 0
            c[blkpos[1],blkpos[2]] = bins[0,blkpos[1],blkpos[2]]
        endif
    end

    % 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
            end
            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
            end
            syncthreads
            % Compute the product of the two submatrices
            for n = 0..BLOCK_SIZE-1
                sum += sA[blkpos[0],n] * sB[n,blkpos[1]]
            end
            syncthreads
        end
        if pos[0] < size(C,0) && pos[1] < size(C,1)
            C[pos] = sum % Write the result
        endif
    end
```

CHAPTER 12

Built-in function quick reference

Some built-in functions are listed in table 12.1. More runtime library functions are given in table 12.2. For a detailed explanation of the functions, we refer to the Documentation Browser (F1 in Redshift).

Table 12.1: 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 12.2: Runtime library functions.

CHAPTER 13

The Quasar compiler/optimizer

The Quasar compiler/optimizer often significantly improves the computation time of Quasar programs. Different strategies are employed:

1. *Constant folding*: when programs contain constant expressions such as `(pi*4)/3-4/5`, some performance gain can be obtained by computing the result during compile-time. In Quasar, the constant folding optimization technique is implemented as follows:

   - First, the meta-function `$eval(...)` performs compile-time evaluation of its input arguments. For example, `$eval((pi*4)/3-4/5)` results in a constant `3.388790204786`.
   - Then, constant propagation of scalar values is by the following reductions:

```
reduction x:scalar'const -> -x = $eval(-x)
reduction (x:scalar'const, y:scalar'const) -> x+y = $eval(x+y)
reduction (x:scalar'const, y:scalar'const) -> x-y = $eval(x-y)
reduction (x:scalar'const, y:scalar'const) -> x*y = $eval(x*y)
reduction (x:scalar'const, y:scalar'const) -> x/y = $eval(x/y)
reduction (x:scalar'const, y:scalar'const) -> x^y = $eval(x^y)
```

   Here, the modifier `'const` expresses that the corresponding variable has a constant (and known value).

2. *Type inference*: this allows the compiler to determine the types of the variables used in a Quasar. Correspondingly, the compiler can generate more optimal code for the given input variables (e.g. using specialized reductions). The following example demonstrates how the type inference works:

```
A = ones(1,3,4)  % A is of type 'mat'
sz = size(A)     % sz is of type 'ivec3', with sz[0]==1,sz[1]!=1,sz[2]!=1
B = randn(sz)    % B is of type 'mat', with size "1,?,?"
f = x -> 2*x     % f is of type '[??->??]'
g = x : scalar -> 2*x  % f is of type '[scalar->scalar]'
C = f(B)         % C is of type 'mat' with size "1,?,?"
D = g(B)         % compiler error, B has type 'mat' and not 'scalar'!
```


3. *Reductions of expressions*: details on reductions can be found in section 4.7. Reductions can be used to significantly improve the computational performance of certain algorithms. For example, using the complex-to-real (C2R) 2D-iFFT:

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

a speed-up of approximately a factor 2 is obtained compared to the unoptimized version. There are a number options to configure the reductions in Quasar:

- Reductions can be temporarily disabled or enabled using the following `#pragma`:

```
#pragma reductions(on|off)
```

- By default, the Quasar compiler reports when a specific reduction is being used. This can be toggled on/off:

```
#pragma show_reductions(on|off)
```

- These settings can also be modified on a global level, using the config settings `COMPILER_PERFORM_REDUCTIONS` and `COMPILER_SHOW_REDUCTIONS` (see table 13.1).

4. *Expression optimization*: operations on large matrices are grouped and automatically converted into a kernel function (see section §8.1). For example:

```
x = randn(512,512,64)
y = 0.1 + (0.8 * 255 * sin(x/255)) + 10 * w
```

The expression optimization can be configured using the following pragma:

```
#pragma expression_optimizer (on|off)
```

Alternatively, this setting can also be changed on a global level, using the config settings `COMPILER_EXPRESSION_OPTIMIZER` (see table 13.1).

5. *Automatic loop parallelization* (ALP): a general overview of the automatic loop parallelizer can be found in section 2.3. Some implementation details of this technique will be discussed in the next section.

13.1 Automatic loop parallelization (ALP)

The ALP attempts to parallelize for-loops in Quasar programs, starting the outside loops. The ALP program automatically recognizes one, two and three dimenional for-loops, and also maximizes the dimensionality of the loops subject to parallelization. There are however a number of restrictions to the Quasar code:

- The types of all variables must be *known* (either explicitly as function argument types, or through type inference, see section 2.2).
- 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).
- Currently, user functions and lambda expressions are not supported. This may change in a future version of Quasar.
- Only a limited number of built-in functions are allowed. These are the functions that are also accessible from within `__kernel__` functions (see table 12.1).
- Only types that can be used inside `__kernel__` or `__device__` functions are allowed.
- Data dependencies/conflicts between different iterations are detected and not allowed.
- Advanced kernel function features such as shared memory and thread synchronization (see section 2.4.4) are currently not supported.

In case one of these conditions are violated, a warning message is generated (see section 13.1.1), and the code is *not* parallelized (often resulting in a much slower sequential program).
The ALP can be configured using the pragmas (see section 13.2):

```
#pragma loop_parallelizer (on|off)
#pragma force_parallel
```

and the global configuration setting `COMPILER_AUTO_FORLOOP_PARALLELIZATION` (see table 13.1).

13.1.1 Auto-parallelization warning messages

In case the auto-parallelization does not succeed, warning messages are outputted, to help fixing the problem. Different messages are listed and discussed below:

1. *Operator ':' can not be parallelized. Consider using [a..b] instead*

   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]`.

2. *Parallelization of the sequence a..b not possible: all operands should be constant!*

   Without dynamic kernel memory (see section §8.3), the compiler needs to know the size of the sequence, therefore, a and b should be constants. See point 1.

3. *Function `zeros` requires exactly one constant argument for parallelization!*

   Without dynamic kernel memory (see section §8.3), the function `zeros` (or `uninit`, `ones`) can only be used to allocate vectors of a fixed length. Note that the memory can still be allocated outside the loop and used inside the loop, which does not have this restriction. However, small vectors of fixed length ≤ 32, can be allocated in local device memory. Hence `zeros(8)` allocates a vector of length 8 in parallel.

4. *Parallelization of 'xxx' not possible: need to know the size of the result of type 'yyy'*

   The compiler needs to know the types and sizes of the variables. Here, the size of one of the variables is not known.

5. *Parallelization not possible due to the type of the variable*

   The variable type is either not supported or not known. Currently supported types for automatic parallelization are: `int`, `scalar`, `cscalar`, `ivecx`, `vecx`, cvec*x*, `vec`, `mat`, `cube`, `cvec`, `cmat`, `ccube`.

6. *Maximum vector length (32) for local memory is exceeded!*

   On GPU devices, local memory is very scarse (see section 2.4.3). Hence there is a limit on the maximum length of vectors.

7. *Operations involving strings can currently not be parallelized!*

   At present, there is no support for processing string operations in parallel.

8. *Function call can not be parallelized!*

   Host function calls inside parallel for-loops are currently not supported, for the simple reason that the compiler does not know yet how to handle them. There are a couple of exceptions: most of the built-in functions listed in table 12.1 can still be used. The solution is often to turn the called function into a `__device__` function.

9. *Statement (print etc.) can not be parallelized. Consider placing this outside the loop.*

   It is not possible to parallelize prints, the computation engine (e.g. CUDA) does not support this.

10. *Operations involving objects can currently not be parallelized!*

    There is no support yet for manipulation of objects inside parallel for loops.

11. *Construction of cell arrays can not be parallelized!*

    Without dynamic kernel memory (see section §8.3), there is no support yet for cell arrays inside parallel for loops.

12. *Possible data dependency detected for variable 'xxx'!*

    In case of a detected data dependency (e.g. read after write, write after read, or write after write), the compiler prints this message and refuses to perform the parallelization. The problem can often be solved by creating auxiliary variables, restructuring the loops, or splitting the loops. Also see section 2.4.4 for a more detailed explanation on how data races can efficiently be tackled in Quasar.

13.2 Compilation settings

Compilation settings can be configured in the config file `Quasar.config.xml`. A number of global settings are listed in table 13.1. 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 force_parallel`	Forces the next for-loop to be parallelized.
`#pragma force_serial`	Forces the next for-loop to be serialized.
`#pragma reductions (on|off)`	Turns off/enables reductions defined using the reduction keyword
`#pragma show_reductions (on|off)`	Enables/disables messages when reductions are applied.
`#pragma expression_optimizer (on|off)`	Enables/disables the expression optimizer.

Table 13.1: 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 (see section XX)
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
COMPILER_SHOW_MISSED_OPT_OPPORTUNITIES	True/False	Displays additional possibilities for optimization
COMPILER_AUTO_FORLOOP_PARALLELIZATION	True/False	Enables automatic parallelisation of for loops
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)

