Quasar - CUDA Support GUIDE

Bart Goossens

Dec 9th, 2015

Contents

1 Preface

This document contains information on the CUDA features that are currently supported in Quasar, how they can be accessed and used from within Quasar.

The goal is to make a wide set of CUDA features available to a user group that has no (or limited) experience with programming in CUDA, or to teams that do not have the available time resources available to do low-level CUDA programming.

The CUDA back-ends in Quasar closely follow the features new CUDA releases so that 1) the specific performance improving CUDA features are accessible from Quasar and so that 2) the end-users can benefit from buying new NVidia GPUs. The final goal is that when switching to newer GPUs, they see an acceleration of their algorithms, without having to do any efforts or changing their programs.

Roughly speaking, we can distinguish the CUDA features in two classes:

1. CUDA features that are used automatically by Quasar. Examples are: shared memory, CUFFT, dynamic parallelism, OpenGL interoperability, CUDA streams and synchronization, stream callbacks...
2. CUDA features in which the user has to make small modifications to the program in order to see the effects. Examples are: the use of hardware texturing units, the use of 16-bit floating point formats, the use of the non-coherent constant cache...

Some features impact the code generation, while other features impact the run-time. Because host/kernel and device functions are all described from within Quasar, the Quasar compiler has a good view on the intentions of the programmer and can take appropriate action (e.g., applying various code transformations, or adding meta information that may help the run-time).

This document also discusses several techniques that are currently available in the Quasar compiler/run-time.

2 CUDA features

2.1 Running kernels on the GPU

A simple example of a kernel that can be executed on the GPU is given below:

```
im = imread("image.png")
im_out = zeros(size(im))

function [] = __kernel__ filter_kernel(x, y : 'unchecked, mask, dir, pos)
    offset = int(numel(mask)/2)
    total = 0.
    for k=0..numel(a)-1
        total += x[pos + (k - offset).* dir] * mask[k]
    end
    y[pos] = total
end

parallel_do(size(im),im,im_out,[1, 2, 3, 2, 1] / 9,[0,1],filter_kernel)
```

First, the kernel function filter_kernel is defined. Then, an image is read from the hard disk (**imread**). Next, the output image is allocated an initialized with zeros (**zeros**). Finally, the **parallel_do** function runs the kernel filter_kernel , which filters the image using the specified filter mask [1,2,3,2,1]/9 .

Kernel functions need to be declared using the **__kernel__** modifier. Although, technically, this modifier could even be omitted from the Quasar language specification, it brings transparency to the user about which functions eventually will be executed on the GPU device.

When running kernel functions, the run-time system automatically adapts the block dimensions, shared memory size, to the GPU parameters obtained using the CUDA run-time API. The run-time also makes sure that the data dimensions are a multiple of the block dimensions, thereby maximizing the occupancy of the resulting kernel. If necessary, the run-time system will run a version of the kernel that can process data dimensions that are not a multiple of the block dimensions.

Depending on the characteristics of the kernel (determined at compile-time), the run-time system also chooses the cache configuration of the kernel: to trade-off shared memory vs. data caching.

To define kernels in Quasar, lambda expressions can also be used, if that is more convenient. In combination with (automatic) closure variables, we can define the above filter simply as:

```
parallel_do(size(im),__kernel__ (pos) -> y[pos] =
    (x[pos+[0,-2]]+x[pos[0,2]]+2*(x[pos+[0,-1]]+x[pos+[0,1]]+3*x[pos])/9)
```

Note that in this example, the filter mask [1,2,3,2,1]/9 is substituted in the kernel function, which is a manual optimization. However, the compiler is able to do this optimization *automatically*. Even in the first example, the variables **dir** and mask are determined to be constants by the compiler, and a specialization of the kernel filter_kernel is generated automatically that utilizes this constantness.

Alternatively, in many cases, kernel functions may be generated automatically by the Quasar compiler. For example, for matrix expressions A+B.*C+D[:,:,2] the compiler may generate a kernel function. Idem for the automatic loop parallelizer. Consider the loop:

```
x = imread("image.png")
y = zeros(size(x))
for m=0..size(y,0)-1
    for n=0..size(y,1)-1
        for k=0..size(y,2)-1
            y[m,n,k] = (x[m,n-2,k]+x[m,n+2,k] + 2*(x[m,n-1,k]+x[m,n+1,k])+3*x[m,n,k])/9
        end
    end
end
```

For the above loop, the compiler will perform a dependency analysis and determine that the loop is parallelizable. Subsequently, a kernel function and **parallel_do** call will automatically be generated. This relieves the user from thinking in terms of kernel functions and parallel loops.

2.2 Device functions

To enable kernel functions to share functionally, **__device__** functions can be defined. **__device__** functions can be called from either host functions (i.e. without **__kernel__**/**__device__**) or other kernel/device functions. An example is given below:

```
function y = __device__ hsv2rgb (c : vec3)
    h = int(c[0] * 6.0)
    f = frac(c[0] * 6.0)
    v = 255.0 * c[2]
    p = v * (1 - c[1])
    q = v * (1 - f * c[1])
    t = v * (1 - (1 - f) * c[1])
    match h with
    | 0 -> y = [v, t, p]
    | 1 -> y = [q, v, p]
    | 2 -> y = [p, v, t]
    | 3 -> y = [p, q, v]
    | 4 -> y = [t, p, v]
    | _ -> y = [v, p, q]
    end
end
```

Note the compact syntax notation in which vectors can be handled.

Device functions can be generic (like template functions in C++). In Quasar, it suffices to not specify any type of the function arguments. The following lambda expression

```
norm = __device__ (x) -> sqrt(sum(x.^2))
```

will then be specialized for every usage. For example, one can call the function with a scalar number (**norm**(−2)=2) or using a vector (**norm**([3,4])=5).

2.3 Block position, block dimensions, block index

Several GPU working parameters can be accessed via special kernel function parameters (with a fixed name). For example:

```
function [] = __kernel__ traverse(pos, blkpos, blkdim, blkidx)
    ...
end
```

The meaning of the special kernel function parameters is as follows:

Parameter name	Meaning
pos	Position
blkpos	Position within the current block
blkdim	Block dimensions
blkidx	Block index
blkcnt	Block count (grid dimensions)
warpsize	Size of a warp

Correspondingly, the pos argument is calculated (internally) as follows:

```
pos = blkidx .* blkdim + blkpos
```

The type of the parameters can either be specified by the user (e.g., **ivec2**: an integer vector of length 2), or determined automatically through type inference.

By default, the block dimensions are determined automatically by the Quasar runtime system, but optionally, the user can specify the block dimensions manually, using the **parallel_do** function.

The maximum block size for a given kernel function can be requested using the function max_block_size. An optimal block size can be calculated with the function opt_block_size.

2.4 Datatypes and working precision

Quasar supports both integer, scalar (floating point) and complex scalar data data types. Typically, the global floating point working precision is specified at a global level (inside the Quasar Redshift IDE). This allows the user to change the precision of his program at any point, which allows him/her to investigate the accuracy or performance benefits of a different working precision. Two precision modes are supported (single precision and double precision). In the future, half-precision mode (which is supported by CUDA 7.5) may also be added.

The impact of the working precision is global, and leads to different sets of functions to be called internally. For example, cuFFT can be used in single or in double precision mode; this is done automatically (the user does not need to change the code, as it suffices to call the fft1, **fft2** or fft3 functions).

2.5 Atomic operations

Atomic operations have a special syntax in Quasar. Whenever a compound assignment operator is used (e.g., += ,−=,. . .) on a vector/matrix variable that is stored in the global/shared memory of the GPU, an atomic operation is performed. Atomic operations are supported for integers and floating point numbers.
The following table lists the types of supported atomic operations.

Operator	Meaning
+=	Atomic add
-=	Atomic subtract
=	Atomic multiply
/=	Atomic divide
ˆ=	Atomic power
.*=	Atomic pointwise multiplication (e.g. for vectors)
./=	Atomic pointwise division (e.g. for vectors)
.ˆ=	Atomic pointwise power (e.g. for vectors)
&=	Atomic bitwise AND
|=	Atomic bitwise OR
˜=	Atomic bitwise XOR
ˆˆ=	Atomic maximum
_=	Atomic minimum

The result of the atomic operation is the value obtained after the operation. For example, for a=0; b=(a+=1), we will have that b=1.

The use of atomic operations is very convenient, especially for writing functions that aggregate the data, like:

```
x = imread("image.tif")
[sum1,sum2] = [0.0,0.0]
for m=0..size(x,0)-1
    for n=0..size(x,1)-1
        sum1 += x[m,n]
        sum2 += x[m,n]^2
    end
end
```

A direct translation to atomic operations on the GPU does not lead to the most optimal performance (because the operations will be serialized internally). Therefore, to be able to benefit from the simplicity in implementing summing/other aggregation algorithms, the Quasar compiler automatically recognizes aggregation variables, and translates the resulting loop to a more efficient parallel sum reduction algorithm using shared memory. This often improves the performance by a factor of ten!

2.6 Shared memory and thread synchronization

As mentioned in the previous section, the compiler may generate code that uses the shared memory of the GPU automatically. In fact, the compiler is able to detect certain programming patterns and to replace them by algorithms that make use of shared memory. Some examples are:

- Convolution algorithms
- Parallel reduction algorithms

This alleviates the user from writting algorithms making use of the shared memory. However, in some cases, it is also useful to manually write algorithms that store intermediate values in the shared memory. Therefore, shared memory can be allocated using the function **shared** (or optionally using **shared_zeros**, in case the memory needs to be initialized with zeros). Thread synchronization is performed using the keyword **syncthreads**. An example is given below:

```
function y = gaussian_filter(x, fc, n)
    function [] = __kernel__ kernel(x:cube,y:cube'unchecked,fc:vec'unchecked'hw_const,n:
        int,pos:ivec3,blkpos:ivec3,blkdim:ivec3)
        [M,N,P] = blkdim+[numel(fc),0,0]
        assert(M*N*P<=1024)  % put an upper bound on the amount of shared memory
        vals = shared(M,N,P) % allocate shared memory

        sum = 0.
        for i=0..numel(fc)-1   % step 1 - horizontal filter
            sum += x[pos[0]-n,pos[1]+i-n,blkpos[2]] * fc[i]
        end
        vals[blkpos] = sum  % store the result

        if blkpos[0]<numel(fc) % filter two extra rows (needed for vertical filtering
            sum = 0.
            for i=0..numel(fc)-1
                sum += x[pos[0]+blkdim[0]-n,pos[1]+i-n,blkpos[2]] * fc[i]
            end
            vals[blkpos+[blkdim[0],0,0]] = sum
        endif

        syncthreads  % thread synchronization

        sum = 0.
        for i=0..numel(fc)-1   % step 2 - vertical filter
            sum += vals[blkpos[0]+i,blkpos[1],blkpos[2]] * fc[i]
        end
        y[pos] = sum
    end
    y = uninit(size(x))
    parallel_do (size(y), x, y, fc, n, kernel)
end
```

Here, it is worth mentioning that the assertion M∗N∗P<=1024 helps the compiler, to determine an upper bound for the amount of shared memory that needs to be reserved for the kernel function. This allows for multiple blocks to be processed in parallel on the GPU (maximizing the occupancy).

2.6.1 Hardware textures

In Quasar, several built-in type modifiers define what happens when data is accessed outside the array/matrix boundaries:

These type modifiers need to be added to the types of the kernel function parameters.

When using GPU hardware textures, the boundary conditions `'circular`, `'mirror`, `'clamped` and `'safe` are directly supported by the hardware, leading to an extra acceleration. In Quasar, it is possible to indicate that a vector/matrix (with width that is a multiple of the minimal hardware texture pitch) needs to be stored as a hardware texture, using special type modifiers that can be used for kernel function argument types:

An example is the following interpolation kernel function:

```
function [] = __kernel__ interpolate_hwtex (y : mat, x : mat'hwtex_linear'clamped, scale :
     scalar, pos : ivec2)
    y[pos] = x[scale * pos]
end
```

Here, the type modifiers hwtex_linear and clamped are combined.

As an extension, *multi-component* hardware textures, can also be defined, using the following type modifiers:

The advantage is that complete RGBA-values can be loaded with only one texture lookup.

For devices with compute architecture 2.0 of higher, writes to hardware textures are also supported, via CUDA surface writes. This is done fully automatically and requires no changes to the Quasar code.

Correspondingly, a matrix can reside in (global) device memory, in texture memory, in CPU host memory, or even all of the above. The run-time system keeps track of the status bits of the variables.

2.7 Memory optimizations

To optimize memory transfer and access, the Quasar compiler/run-time employ a variety of techniques, many of them relying on underlying CUDA features.

When a kernel function is started (using the **parallel_do** function), it is made sure that all kernel function arguments are copied to the GPU, and that the arguments are up to date. This is the automatic memory management function, that is performed by the Quasar run-time. The automatic memory management works the best with large blocks of memory. Therefore, in case of small structures, such as nodes of a linked lists, several connected structures are grouped in a graph and the graph is transferred at once to the device. However, in some cases it is useful to perform some extra memory optimizations.

2.7.1 Modifiers 'const and 'nocopy

The types of kernel function parameters can have the special modifiers `'const` and `'nocopy`. These modifiers are added automatically by the compiler after analyzing the kernel function. The meaning of this modifiers is as follows:

The advantage of these type modifiers is that it permits the run-time to avoid unnecessary memory copies

1. between host and the device
2. between multiple devices (in multi-GPU processing modes)
3. between linear device memory and texture memory

In case of parameters with the `'const` modifier, the dirty bit of the vector/matrix does not need to be set. For `'nocopy`, it suffices to allocate data in device memory *without* initializing it.

Furthermore, the modifiers are exploited in later optimization and code generation passes, e.g. to take automatic advantage of caching capabilities of the GPU (see below).

An example of the modifiers is given below:

```
function [] __kernel__ copy_kernel(x:vec'const,y:vec'nocopy,pos:int)
    y[pos] = x[pos]
end
x = ones(2^16)
y = zeros(size(x))
parallel_do(numel(x),x,y,kernel)
```

Here `'nocopy` is added because the entire content of the matrix y is overwritten.

2.7.2 Constant memory

NVIDIA GPUs typically provide 64KB of constant memory that is treaded 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). Kernel function parameters can be declared to reside in constant memory by adding the hw_const modifier:

```
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
```

The use of constant memory may have a dramatic impact on the performance. In the above example, the computation time was improved by a factor 4x, by only adding `'hwconst` modifier to the kernel function.

2.7.3 Texture memory

With the advent of CUDA, the GPU's sophisticated texture memory can also be used for general-purpose computing. Although originally designed for OpenGL and DirectX rendering, texture memory has properties that make it very useful for computing purposes. 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.

The Quasar run-time allocates texture memory (via CUDA arrays) and transparently copies data to the arrays when necessary. Texture memory has the advantage that it can efficiently be accessed using more irregular access patterns, due to the texture cache.

The layout of the CUDA array is generally different from the memory layout in global memory. For example, there is a global option that the user can choose to use 16-bit float textures (instead of the regular 32-bit float textures). This not only reduces the amount of texture memory but also offers performance enhancements due to the reduced memory bandwidths (e.g. for real-time video processing). When necessary, the Quasar run-time performs the necessary conversions.

2.7.4 Non-coherent Texture Cache

As an alternative for devices with compute capability 3.5 (or higher), the non-coherent texture (NCT) cache can be used. The NCT cache 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.

In Quasar, kernel function parameters can be declared to be accessed via the NCT cache, by adding the modifiers `'hwtex_const`. Internally, hwtex_const generates CUDA code with the __ldg() intrinsic.

Optionally, depending on the compiler settings, the compiler may also add hwtex_const modifiers automatically, when determined to be beneficial.

2.7.5 Unified memory

CUDA unified memory is a feature introduced in CUDA 6.0. For unified memory, one single pointer is shared between CPU and GPU. The device driver and the hardware make sure that the data is migrated automatically between host and device. This facilitates the writing of CUDA code, since no longer pointers for both CPU and GPU need to be allocated and also because the memory is transferred automatically.

In Quasar, unified memory can be used by setting RUNTIME_CUDA_UNIFIEDMEMORYMODE to Always in Quasar .Runtime.Config.xml.

2.8 Streams and Concurrency

Quasar has two execution modes that can be specified by the user on a global level (typically from within the Quasar Redshift IDE):

- *non-turbo*: perform only one kernel at the time (synchronous kernel execution)
- *turbo*: overlap kernels/memory transfers when possible (asynchronous kernel execution)

The turbo execution mode is fully automatic and internally relies on the asynchronous CUDA API functions. In the background, dependencies between matrix variables, kernels etc. are being tracked and the individual memory copy operations and kernel calls are automatically dispatched to one of the available CUDA streams. This way, operations can be performed in parallel on the available GPU hardware. Often, the turbo mode enhances the performance by 10%-30%. Also, automatic multi-GPU processing can be obtained using this concurrency model (see further).

Moreover, in Quasar, the streaming mechanism is expanded to the CPUs, which allows configurations in which for example, 4 CPU streams (threads) are allocated, 2 GPUs with each 4 streams. A load balancer and scheduler will divide the workload over the different CPU and GPU streams. It is possible to configure the number of CPU streams and the number of GPU streams. For example, on an 8-core CPU, a two parallel loops may run concurrently each on 4 cores. Idem for a system with two GPUs, each GPU may have 4 streams and kernel functions can be scheduled to in total 8 streams.

For inter-device synchronization purposes (e.g., between CPU and GPU), stream callbacks (introduced in CUDA 5.0) are used automatically.

The run-time system can be configured to use host-pinnable memory by default. This the advantage that kernels can directly access memory on the CPU, but also in *turbo* mode, the asynchronous memory copy operations from GPU to the host are non-blocking.

2.9 CuFFT: CUDA FFT library

The CUDA fft library is natively supported using the functions fft1, **fft2**, fft3, ifft1, **ifft2**, ifft3 and real-valued versions **real**(ifft1(x)), **real**(**ifft2**(x)), **real**(ifft3(x)). Often, one is interested in the real part of the inverse Fourier transform. Via a reduction (which is defined in Quasar),

```
reduction x -> real(ifft3(x)) -> irealfft3(x)
```

all occurences of **real**(ifft3(x)) will be converted to the real-valued version of the ifft3. Internally Quasar uses the CUDA FFT plan with C2R option for implementing **real**(ifft1(x)). The CuFFT module also supports automatic streaming. For small data sizes, or on CPU target platforms, the FFTW library is used as a fallback.

In future versions, other CUDA libraries may also be supported by Quasar, such as cuRand, cuBLAS, cuDNN, . . .

2.10 OpenGL interoperability

For visualization purposes, OpenGL is used automatically. To avoid copying the data to the system (CPU) memory, Quasar uses the OpenGL interoperability features of CUDA, via global memory and/or texture memory (CUDA array) mappings. This way, visualization is extremely fast! To display an image using OpenGL, it suffices to call the **imshow**() function:

```
im = imread("image.png")
imshow(im)
```

Similarly, a video sequence can be displayed, using the **hold** function, which will make sure that the image is updated in the current display window.

```
stream = vidopen("movie.mp4")
sync_framerate(stream.avg_frame_rate)  % Sets the display frame rate

hold("on")
while vidreadframe(stream)
    frame = float(stream.rgb_data)
    imshow(frame)
end
```

This short fragment of code is enough to show a video sequence in a display window! Moreover, the video is shown using real-time 2D OpenGL rendering. In particular, the **imshow** function creates an OpenGL context, transfers the content of the frame to CUDA texture memory (if it is not already there) and renders the result using CUDA-OpenGL interoperability.

During the visualization, Quasar datatypes are automatically mapped onto corresponding OpenGL types. Therefore matrices of different data types can easily be visualized (e.g. 16-bit integer, 32-bit float etc.)

2.11 Dynamic parallelism

CUDA dynamic parallelism allows a CUDA thread to spawn its own sub-thread. This is particularly useful for tasks with mixed coarse grain and fine grain parallelism. In Quasar, nested parallelism (obtained by calling the **parallel_do** function from a kernel/device function) is mapped automatically onto the CUDA dynamic parallelism. Thereby, the resulting program is linked with the CUDA device runtime.

There are several advantages of the dynamic parallelism:

- More flexibility in expressing the algorithms
- The nested kernel functions are (or will be) mapped onto CUDA dynamic parallelism on Maxwell/Kepler devices.
- The high-level matrix operations from the previous section are automatically taking advantage of the nested parallelism.

In several cases, the mapping onto dynamic parallelism is implicit. Consider the following loop:

```
for m=0..size(y,0)-1
    for c=0..size(y,2)-1
        row = y[m,:,c]
        y[m,:,c] = row[numel(row)-1..-1..0]
    end
end
```

Inside the loop, the vector operations will be expanded by the compiler to a kernel function. When subsequently the two-dimensional loop is parallelized, nested parallelism is obtained.

2.12 Multi-GPU

Quasar supports multi-device configurations, which allows several GPUs to be combined with a CPU. For the programmer, outside kernel/device functions, the programming model is sequential in nature, irrespective of whether one or multiple GPUs are being used. The Quasar multi-GPU feature allows a program to be executed on multiple GPUs (let say 2), without any/very little changes (see below) to the code, while benefitting from a 2x acceleration.

To achieve this, the load balancing is entirely automatic and will take advantage of the available GPUs, when possible. The run-time system supports peer-to-peer memory transfers (when available) and transfers via host pinned memory. Here, host pinned memory is used to make sure that the memory copies from the GPU to the host are entirely asynchronous.

Each of the GPU devices has its own command queue, this is a queue on which the load balancer places individual commands that needs to be processed by the respective devices. The load balancer takes several factors into account, such as memory transfer times, load of the GPU, dependencies of the kernel function, ...

Hence all the memory transfers between the GPUs and between host and GPU, are managed automatically (and reduced as much as possible). In some cases it is useful to have more control about which GPU is used for which task. This can be achieved by explicitly setting the GPU device via a scheduling instruction:

```
{!sched gpu_index=0}
or
{!sched gpu_index=1}
```

This overrides the default decision of the load balancer. For for-loops this can be done as follows:

```
for k=0..num_tasks-1
    {!sched gpu_index=mod(k,2)}
    parallel_do(..., kernel1)
    parallel_do(..., kernel2)
end
```

This way, each GPU will take care of one iteration of the loop. To enhance the load balancing over the GPUs, it may also be more beneficial to use the following technique

```
for k=0..num_tasks-1
    {!unroll times=2; multi_device=true}
    parallel_do(..., k, kernel1)
    parallel_do(..., k, kernel2)
end
```

Here, {!unroll times=2; multi_device=true} unrolls the for loop twice, where each **parallel_do** function is launched on a different device. Internally, the following code is generated:

```
for k=0..2..num_tasks-1
    {!sched gpu_index=0}
    parallel_do(..., k, kernel1)
    {!sched gpu_index=1}
    parallel_do(..., k+1, kernel1)

    {!sched gpu_index=0}
    parallel_do(..., k, kernel2)
    {!sched gpu_index=1}
    parallel_do(..., k+1, kernel2)
end
```

Due to the asynchronous nature of the calls, the kernel functions will effectively be executed in parallel on two GPUs.

3 Example

To illustrate the mapping of Quasar code onto CUDA code, we give a more elaborated example in this section. We consider a generic morphological operation, defined by the following functions:

```
op_min = __device__ (x,y) -> min(x,y)

function y = morph_filter[T](x : cube[T], mask : mat[T], ctr : ivec2, init : T, morph_op :
     [__device__ (T, T) -> T])
    function [] = __kernel__ kernel (x : cube[T]'mirror, y, mask : mat[T]'unchecked'
        hwconst, ctr, init, morph_op, pos)
        {!kernel_transform enable="localwindow"}
        res = init
        for m=0..size(mask,0)-1
            for n=0..size(mask,1)-1
                if mask[m,n] != 0
                    res = morph_op(res,x[pos+[m-ctr[0],n-ctr[1],0]])
                endif
            end
        end
        y[pos] = res
    end

    y = cube[T](size(x))
    parallel_do(size(x),x,y,mask,ctr,init,morph_op,kernel)
end
```

The morphological filter is generic in the sense that the data types for the operation are not specified, and also not the operation morph_op. Initially, the code purely works in the global memory and requires several accesses to x (in practice, depending on the size of the mask).

Using the following compiler transform steps/optimizations, the above code transformed to code that makes use of the shared memory. In particular, the following steps are being performed:

1. specialization of the generic functions op_min and morph_filter
2. inlining of op_min
3. constancy analysis of the kernel function parameters
4. dependency analysis of the kernel function parameters, determining a size for the local processing window
5. local windowing transform, generating code using shared memory and thread synchronization.
6. array/matrix boundary access handling.

The resulting Quasar code is:

```
function [y:cube] = morph_filter(x:cube,mask:mat,ctr:ivec2,init:scalar)=static
    function [] __kernel__ kernel(x:cube'const'mirror,y:cube,mask:mat'const'unchecked'
        hwconst,ctr:ivec2'const,init:scalar'const,pos:ivec3,blkpos:ivec3,blkdim:ivec3)=
        static

        {!kernel name="kernel"; target="gpu"}
        sh$x=shared((blkdim+[((((size(mask,0)-1)-ctr[0])+ctr[0])+1),((((size(mask,1)-1)-
            ctr[1])+ctr[1])+1),1]))
        sh$x[blkpos]=x[(pos+[-(ctr[0]),-(ctr[1]),0])]

        if (blkpos[1]<(((size(mask,1)-1)-ctr[1])+ctr[1]))
            sh$x[(blkpos+[0,blkdim[1],0])]=x[(pos+[-(ctr[0]),-(ctr[1]),0]+[0,blkdim[1],0])
                ]
        endif
        if (blkpos[0]<(((size(mask,0)-1)-ctr[0])+ctr[0]))
            sh$x[(blkpos+[blkdim[0],0,0])]=x[(pos+[-(ctr[0]),-(ctr[1]),0]+[blkdim[0],0,0])
                ]
        endif
        if ((blkpos[1]<(((size(mask,1)-1)-ctr[1])+ctr[1]))&&(blkpos[0]<(((size(mask,0)-1)-
            ctr[0])+ctr[0])))
            sh$x[(blkpos+[blkdim[0],blkdim[1],0])]=x[(pos+[-(ctr[0]),-(ctr[1]),0]+[blkdim
                [0],blkdim[1],0])]
        endif

        blkof$x=((blkpos-pos)-[-(ctr[0]),-(ctr[1]),0])
        syncthreads

        res=init
        for m=0..(size(mask,0)-1)
            for n=0..(size(mask,1)-1)
                if ($getunchk(mask,m,n)!=0)
                    res=min(res,sh$x[(pos+[(m-ctr[0]),(n-ctr[1]),0]+blkof$x)])
                endif
            end
        end
        y[pos,res]
    end

    y=cube(size(x))
    parallel_do(size(x),x,y,mask,ctr,init,kernel)
end
```

Equivalently, the user could have written this kernel function from the first time. However, many users do not prefer to do this, because 1) it requires more work, 2) it is prone to errors, 3) the code readability is reduced.

Finally, the generated kernel function can straightforwardly be translated to CUDA code.