Automatic GPU Code Generation of Modelica Functions

Hilding Elmqvist[1], Hans Olsson[1], Axel Goteman[1,2], Vilhelm Roxling[1,2],
Dirk Zimmer[3], Alexander Pollok[3]

[1]Dassault Systemes, Lund, Sweden, {Hilding.Elmqvist, Hans.Olsson}@3ds.com
[2]Lund Institute of Technology, Lund, Sweden, {axel.goteman, vilhelm.roxling}@gmail.com
[3]Institute of System Dynamics and Control, DLR, Germany, {Dirk.Zimmer, Alexander.Pollok}@dlr.de

Abstract

Modelica users can and want to build more realistic and complex models. This typically means slower simulations. In the past, the speed of single CPUs has increased significantly to partly compensate, but more recently, there has been a shift to multi-core architectures. This is taken to the extreme in Graphics Processing Units (GPUs).

This paper discusses code generation for GPU cores. This is important when the model has regular structure, for example, discretization of PDEs. The behavior of each cell can then be partly described by a function call. The evaluation of such calls can then be made in parallel on the GPU cores. The same function is thus executed on every GPU core, but operates on different data; the data of its cell.

Our GPU code generator automatically generates code for Modelica functions, i.e. no additional language constructs are needed. The function is just annotated as suitable for execution on a GPU.

Keywords: Modelica functions, Multi-core, GPU, CFD

1 Introduction

Modelica users can and want to build more realistic and complex models. This typically means slower simulations. The speed of CPUs has of course increased enormously to partly compensate. But now it's important to utilize the many cores available in modern computer architectures.

The paper (Elmqvist, et al., 2014) presents an algorithm for automatic partitioning of model equations onto CPU cores. This technique is now available in Dymola 2016 (Dassault Systemes, 2015).

This paper discusses code generation for GPU (Graphics Processing Unit) cores. This is important when the model has regular structure, for example, discretization of PDEs. The behavior of each cell can then be partly described by a function call. The evaluation of such calls can then be made in parallel on the GPU cores. The same function is thus executed on every GPU core, but operates on different data; the data of its cell.

We believe GPU code generation should be transparent for the user. The user only needs to give a hint that a certain function is suitable for GPU execution. In addition to the simplification for the user, it enables better portability of Modelica code, since if a tool does not support GPU code generation, it can simply ignore the annotation. The drawback might be that the user does not have full control of how the GPU and its memory are utilized; i.e. might not be able to get optimal speed.

Another important advantage with automatic GPU code generation is that built-in operators such as matrix multiplication and overloaded operators can also benefit. In addition, it allows reuse of normal Modelica library functions and automatically generating GPU code for them.

(Gebremedhin, et al., 2012) proposes an extension to Modelica called ParModelica, which introduces special **kernel function** and **parallel function** declarations and special variable declaration prefixes to indicate what memory to use for the variables: **parglobal**, **parlocal**, etc.

The outline of this paper is as follows. First a general introduction to GPU architecture and programming is given. Then follows principles of automatic GPU code generation from Modelica functions. Finally several examples are presented.

The speed-up factor varies for the different problem formulations. The best speed-up so far for this early Dymola prototype is about 5 times. It should be noted that this was achieved on a laptop with NVIDIA's Quadro K2100M GPU chip and an Intel Core i7-4800 MQ processor.

2 GPU Architectures and Programming Models

In this section, a short introduction to GPU architectures is made, bringing up some of the fundamentals and the aspects that are considered most important for this paper, concerning performance.

After that follows a short introduction to CUDA, the programming model used in this project, where some more performance considerations are brought up, ending with a hands-on example and some practical details for the code generation. Some of the aspects are directed to the user, to get an idea of the kind of code that could be accelerated by the GPU, and some aspects are rather for the auto generation.

A thorough introduction to the topics is made by (Kirk, 2013) and (Goteman, et al., 2015).

2.1 GPU Architectures

The most fundamental difference between CPU's and GPU's, is that the CPU's are general purpose processors, designed to perform well on sequential code, whereas a GPU is designed to process massively parallel tasks, only. The large cache memories and control logics of a CPU are not provided for the cores of a GPU. Instead, the manufacturers focus on putting as many cores as possible on the chip, letting the threads share cache and control logic between them. A GPU today may have thousands of cores. That is possible, since GPU's only have to work on SIMD (singe instruction, multiple data) instructions. That means that, when feeding the GPU a task, that task is the same for every thread. The only difference is which data the task is to be performed on. Now, a task is often a sequence of instructions that, as we will come to later, in code is expressed as a function. This function may contain e.g. an *if*-statement that causes divergence in terms of execution among the threads. The reason that threads may take different paths at such branching points is that the conditions may depend on thread specific data (such as position in a grid, temperature at that position, etc.). We'll come back to this in a bit.

A GPU is designed to process a large amount of threads as fast as possible, and it is important to see the distinction between that and to process every single thread as fast as possible. The GPU is not supposed to work on all threads at once, which generally is not possible, as the number of cores is too low for it. It is designed to have multiple times more threads loaded into registers, than it can execute. This allows for efficient *thread scheduling*, which means that whenever a thread is idle because of a long latency operation, such as a global memory access, the cores can switch to work on other threads that are ready and waiting for execution.

GPU's are good at thread scheduling, but all time spent on long latency operations cannot be hidden. To fully utilize a GPU, you'll want to let the cores work with floating point operations as much as possible. And even if today's chips have a bandwidth to global memory (RAM) of more than 200 GB/s, global memory accesses has to be considered for good performance. A good way to analyze this is to consider the number of floating point operations per global memory access for a thread, often called the CGMA (Compute to Global Memory Access) ratio. It should obviously be kept as high as possible. If it is too low, there is no way to keep the cores busy, independently how the threads are scheduled. That is because, if the limited amount of data that can be delivered to the cores per time unit is lower than the rate at which the operations on the data can be executed, the data transferring has become a limiting factor. So already on a high level, as a user, it can be advantageous to have the CGMA ratio-thinking in mind when considering letting a function be computed on the GPU.

The threads are partitioned on many levels, and this partitioning can differ between different hardware architectures. But most architecture has a lowest partitioning level, at which the parts are called *warps*. On warp level, no divisions between threads are made. That means that if one thread in a warp has a long latency operation (or just any operation, for that matter) in an *if*-statement, but not the others, all threads in the warp will have to wait for that one thread. But the waiting is at least limited to the warp, which on most current architectures consists of 32 threads. That means that having *if*-statements does not necessarily mean a big difference in performance. The divergence can be organized to be minimized within warps, but as it can be hard to know how warps are arranged, and where the divergence will appear, divergent code should generally be avoided.

The last, and most important, aspect to bring up, is the process of transferring data between the CPU memory and the GPU memory. It is the main bottleneck of a GPU. The transfer speed is relatively low, and the transfer is often related to a lot of overhead work. That implies that there is no point to send work to the GPU, unless there is a lot of it. So if it is possible to avoid memory transfers of this kind, e.g. by not repeatedly transferring constant data, it should be done.

2.2 The NVIDIA CUDA Programming Model

CUDA is a parallel computing platform and programming model invented by NVIDIA. In this project, the extension CUDA C/C++ has been used, making it possible to program CUDA enabled GPU's in a C/C++ environment, with a few extensions.

In CUDA, a function that should be executed, or *launched*, in parallel is called a *kernel*. The kernel is launched for a number of threads, which are divided into *blocks*, creating a *grid* of blocks. The blocks are the level on which threads are loaded to registers for execution, meaning that when a block is loaded, it will not unload before all its threads are executed. Thus threads can only be synchronized within a block. Synchronization here means putting points in the code where the threads should wait and synchronize with other threads before continuing execution. Recall that all threads are not executed at once.

Because all the threads in a block are loaded and unloaded into registers at once, there are limitations on the number of threads in a block. E.g. NVIDIA Quadro K2100M, the chip used for most experiments in this project, has a maximum block size of 1024 threads. The blocks are executed on *Streaming Multiprocessors* (SM's), and an SM can usually accommodate a number of blocks. It is on the SM level that warps are scheduled for execution, and it is therefore important to load as many threads as possible to each SM. On K2100M, each SM can have a maximum of 2048 threads loaded at once, and there are three SM's, giving a total of 6144 thread slots. So in order to fully utilize this chip, considering the scheduling of warps, at least 6144 threads should be launched. It may be interesting to know that each SM has 192 cores, giving a total of 576 cores, which equals the number of threads that can actually execute in parallel.

If all 6144 slots for threads are loaded during an entire kernel execution, the kernel is said to have 100 % *occupancy*. Of course this rarely happens since some blocks are bound to finish sooner than others. For good performance, the occupancy should be kept as high as possible, to allow for as much warp scheduling as possible. If a kernel on K2100M would be launched with blocks of 768 threads, the SM's would still only have place for two whole blocks, resulting in a lower occupancy. Or if a kernel is complex, each thread may require more registers, forcing down the number of threads loaded into an SM, thus decreasing the occupancy. There are more aspects that can affect the occupancy, and it is definitely a term that is good to know when considering GPU performance in general.

2.2.1 Example: vector addition

It may be of interest to see how all this could look in practice. First a few notes about CUDA C/C++:

1.) Generally, when inside a kernel, i.e. when code is executed on the GPU, no data that is not allocated on GPU memory can be accessed.
2.) Built-in primitives such as *int* and *float*, pointers, and structs can be copied to the GPU as arguments to the kernel (without deep copy). Large sets of data, like arrays, have to be copied using some CUDA API function.
3.) A kernel's return type must be of type void.
4.) A kernel may call other functions on the GPU, called *device* functions.
5.) Only a subset of C/C++ is supported.
6.) Thrust is a C++ STL based library for CUDA.

Two arrays can be added on the CPU in the following function:

```
void vectorAddCPU(const double *a, size_t n,
  const double *b, double *c){
        for(size_t i=0; i<n; ++i){
                c[i] = a[i]+b[i];
        }
}
```

It is clear that this is a very parallel task. *n* threads could be launched, where each thread should have an individual variable *i* in some way. The simplest way to recognize that something is parallelizable is usually when it is placed in a for-loop, or in nested for-loops, and it does not depend on previous iterations. Below is a kernel for vector addition:

```
__global__
void vectorAddGPU_kernel(const double *a, size_t n,
  const double *b, double *c){
        size_t i=threadIdx.x+ blockIdx.x*blockDim.x;
        if(i<n){
                c[i]=a[i]+b[i];
        }
}
```

Note the keyword *__global__* needed before the return type. First the thread is identifying itself using the thread specific variable *threadIdx*, the variable *blockDim*, and the block specific variable *blockIdx*. Those are variables of the simple type *dim3*, having three members: *x, y* and *z*. This helps to arrange threads according to your problem in up to three dimensions. In this case the problem is obviously one dimensional. The *if*-statement is needed to prevent memory access violations in cases where more threads are launched than needed. That is usually the case when many blocks are launched, since all blocks have the same size.

However, some operations are needed to call the kernel:

```
void vectorAddGPU(const double *a, size_t n, const
double *b, double *c){
 // Allocate GPU memory.
 double *a_d, *b_d, *c_d;
 cudaMalloc(&a_d, n*sizeof(double));
 cudaMalloc(&b_d, n*sizeof(double));
 cudaMalloc(&c_d, n*sizeof(double));

 // Copy a and b to GPU.
 cudaMemcpy(a_d, a, n*sizeof(double),
   cudaMemcpyHostToDevice);
 cudaMemcpy(b_d, b, n*sizeof(double),
   cudaMemcpyHostToDevice);

 // Define grid and block dimensions
 dim3 block = dim3(1024,1,1);
 dim3 grid = dim3((n+1023)/1024,1,1);

 // Launch kernel
 vectorAddGPU_kernel
```

```
  <<<grid,block>>>(a_d, n, b_d, c_d);

 // Copy result back to the CPU.
 cudaMemcpy(c, c_d, n*sizeof(double),
  cudaMemcpyDeviceToHost);

 // Free GPU memory
 cudaFree(a_d);
 cudaFree(b_d);
 cudaFree(c_d);
}
```

First memory is allocated on the GPU for the three vectors, using the CUDA API function cudaMalloc(). Then *a* and *b* are copied to the allocated memory, using the CUDA API function cudaMemcpy(). Then the blocks and the grid are defined, and the kernel is launched. Note the CUDA syntax to specify the kernel launch settings. When the addition is completed on the GPU, the result in *c* has to be copied back, again using cudaMemcpy(). Last of all the GPU memory has to be freed with the CUDA API function cudaFree().

3 GPU Code Generation

The basis for our code generator for Modelica functions is that we recognize that specially marked functions (**annotation**(gpuFunction=true)) satisfy certain for-loop patterns, and for those functions automatically generate GPU-code. The GPU-code consist of a wrapper that allocates variables, copies default values, executes the main body (automatically calling generated CUDA kernel-functions), and then copy back outputs. Any functions called in a kernel function are mapped to device functions.

3.1 Variable allocations

All array variables of the function must either have unknown size (only allowed for inputs), or a size given as a simple arithmetic function of other sizes and integer literals. The unknown array sizes are also propagated to the kernel function. (For performance reasons – and to catch errors – it is good to have as few unknown sizes as possible.) The arrays are allocated on the GPU and existing values copied to the GPU; this allows non-input variables to be assigned a default value in a binding expression. Protected arrays are treated as outputs of the kernel function.

3.2 Loop patterns

The first pattern for the algorithm is that the entire algorithm is a (possibly nested) for-loop with range 1:size(array, literal) and inside the loop any algorithmic code satisfying certain assumptions. The code inside the for loop(s) is mapped to a kernel function; and the (nested) for-loop(s) are replaced by a parallel launch of the kernel function – and checking the index in the kernel function. As an example, consider the function:

```
function vectorAdd
 input Real a[:];
 input Real b[size(a,1)];
 output Real c[size(a,1)];
algorithm
  for i in 1:size(a,1) loop
    // Kernel part
    c[i]:=a[i]+b[i];
  end for;
 annotation(gpuFunction=true);
end vectorAdd;
```

This Modelica function is translated to the previously given vectorAddGPU and vectorAddGPU_kernel code.

3.3 Time integration on the GPU

The second pattern (intended to handle time-integration on the GPU) is a for-loop (with arbitrary index) that contains one or more instances of the first pattern. Each instance of the first pattern is then mapped to a kernel function and called at the appropriate place. The rest of the body may contain assignments; and any array assignment is mapped to a device-to-device copy. The main benefit of this pattern is that we do not need to copy back outputs from the GPU until the end of the function, and device-to-device copy is normally a lot faster.

As an example, consider the following partial differential equation with v(0,0)=0:

$$\frac{\partial v(x,t)}{\partial x} = F\big(v(x,t)\big)$$
$$\frac{\partial v(x,t)}{\partial t} = v(x,t)$$

One way of solving such PDEs is to discretize in the x direction and use an ODE solver for the resulting equations. However, fine grained spatial discretization requires short time increments and it might then be better to use a fixed step size Euler method for time integration.

The function IntegrateF below implements such a solution.

```
function IntegrateF
 input Real v[:];
 output Real next_v[size(v,1)]:=v;
 input Real dt;
 input Integer nSteps;
protected
  Real temp_v[size(v,1)];
algorithm
  // Loop in wrapper-code
  for step in 1:nSteps loop
    // Handled using device-to-device copy:
    temp_v:=next_v;
    for i in 1:size(v, 1) loop
```

```
      // Kernel part
       next_v[i]:=temp_v[i]+dt*(if i>1 then
         F(temp_v[i-1]) else 0);
    end for;
  end for;
end IntegrateF;
```

For the function IntegrateF the main code is similar to vectorAddGPU. The difference is that the called GPU function part is replaced by a loop as follows:

```
dim3 block0=dim3(1024, 1, 1);
dim3 grid0=dim3((x0_0dim0+1023)/1024,1,1);
  {
    int st;
    for(st=1; st<=nSteps; st+=1) {
      cudaMemcpy(tem_vGPU, next_vGPU,
        temp_vGPUs*sizeof(double),
        cudaMemcpyDeviceToDevice);
      IntegrateFcuda<<<grid0,block0>>>(
        vGPU, vdim0,next_vGPU,
        dt,nSteps, temp_vGPU);
    }
  }
```

i.e. the time integration loop is executed on the CPU. There is synchronization between each iteration, i.e. all launched kernel calls must have completed. Note that the statement temp_v:=next_v is translated to a copy call on the GPU memory.

The copying can be avoided by swapping arguments to the kernel calls. It is possible to manually avoid it (note: next_v is initialized to v) – assuming an even number of steps:

```
algorithm
  // Loop in wrapper-code
  for step in 1:2:nSteps loop
   for i in 1:size(v, 1) loop
      // Kernel  part: first kernel function
      temp_v[i]:=next_v[i]+dt*(if i>1 then
        F(next_v[i-1]) else 0);
    end for;
    for i in 1:size(v, 1) loop
      // Kernel  part: second kernel function
      next_v[i]:=temp_v[i]+dt*(if i>1 then
        F(temp_v[i-1]) else 0);
    end for;
  end for;
```

Automating the entire generation of time integration code from the model code would be a possibility for the future, by using synchronous partitions and specifying a solver method associated with the clock.

3.4 For-expressions

An alternative pattern to nested for-loops would be arrays assigned in for-expressions; it would simplify some of the assumptions below, but for performance reasons we would likely need to fuse the loops from multiple for-expressions.

3.5 Assumptions for kernel code

The assumptions on the inner code are (these could be automatically verified, but this is not yet included in the prototype):

- All array indices are valid; based on the array sizes.
- Any right-hand-side variable is not assigned in the inner code. (An exception can be made for scalar temporaries that are initialized in the inner code.) This explains why we need two arrays next_v and temp_v in the example above.
- Each left-hand-side array element is only assigned once.
- Currently only access to scalar variables, and scalar element of arrays in the right hand side, i.e. slices are not supported.

4 Application examples

4.1 Matrix Operations

(Gebremedhin, et al., 2012) uses matrix multiplication as one bench mark example for an extension to Modelica called ParModelica which introduces special **kernel function** and **parallel function** declarations and special variable declaration prefixes to indicate what memory to use for the variables: **parglobal**, **parlocal**, etc.

In our approach, such a matrix multiplication function can be coded in Modelica as follows. Note that the only new element compared to Modelica version 3.3 (Modelica, 2014) is the annotation.

```
function Multiply
  input Real A[:,:];
  input Real B[size(A,2),:];
  output Real C[size(A,1),size(B,2)];
protected
  Real temp;
algorithm
  for i in 1:size(A,1) loop
    for j in 1:size(B,2) loop
      temp := 0;
      for k in 1:size(A,2) loop
        temp := temp + A[i, k]*B[k, j];
      end for;
      C[i, j] := temp;
    end for;
  end for;
  annotation(gpuFunction=true);
end Multiply;
```

It is translated to two functions. The kernel function which runs on the GPU is shown below:

```
#include <stddef.h>
__global__
void Multiply_cuda(
  double const * A, size_t Adim0,
    size_t Adim1,
  double const * B, size_t Bdim1,
  double * C)
{
  double temp;
  temp=0;
  int i = 1+threadIdx.x +
    blockDim.x*blockIdx.x;
  int j = 1+threadIdx.y +
    blockDim.y*blockIdx.y;
  if ((i<=Adim0) && (j<=Bdim1)) {
    temp = 0;
    {
      int end_ = Adim1;
      int k;
      for(k = 1; k <= end_; k += 1) {
        temp = temp +
          A[(i-1)*Adim1+(k-1)] *
          B[(k-1)*Bdim1+(j-1)];
      }
    }
    C[(i-1)*Bdim1+(j-1)] = temp;
  }
  return;
}
```

The other function runs on the CPU to allocate memory for the GPU, copy data to and from the GPU and to invoke the kernel function:

```
extern "C"
void Multiply(
  double const * A, size_t Adim0,
    size_t Adim1,
  double const * B, size_t Bdim1,
  double * C)
{
  /* GPU Memory declaration */
  static double  * AGPU=0;
  static size_t AGPUS=0;
  size_t AGPUs;
  static double  * BGPU=0;
  static size_t BGPUS=0;
  size_t BGPUs;
  static double  * CGPU=0;
  static size_t CGPUS=0;
  size_t CGPUs;
  /* GPU Memory size */
  AGPUs=Adim0*Adim1;
  BGPUs=Adim1*Bdim1;
  CGPUs=Adim0*Bdim1;

  /* GPU Memory allocation */
  if (AGPU&&(AGPUS<AGPUs))
    {AGPUS=0; cudaFree(AGPU); AGPU=0;}
  if (!AGPU)
    {AGPUS=AGPUs;cudaMalloc((void**)&AGPU,
      AGPUS*sizeof(double));}

  if (BGPU&&(BGPUS<BGPUs))
    {BGPUS=0; cudaFree(BGPU); BGPU=0;}
  if (!BGPU)
    {BGPUS=BGPUs;cudaMalloc((void**)&BGPU,
      BGPUS*sizeof(double));}

  if (CGPU&&(CGPUS<CGPUs))
    {CGPUS=0;cudaFree(CGPU);CGPU=0;}
  if (!CGPU)
    {CGPUS=CGPUs;cudaMalloc((void**)&CGPU,
      CGPUS*sizeof(double));}

  /* GPU Memory copy to */
  cudaMemcpy(AGPU, A,AGPUs*sizeof(double),
    cudaMemcpyHostToDevice);
  cudaMemcpy(BGPU, B,BGPUs*sizeof(double),
    cudaMemcpyHostToDevice);
  cudaMemcpy(CGPU, C,CGPUs*sizeof(double),
    cudaMemcpyHostToDevice);
  /* Call GPU function */
  dim3 block=dim3(32, 32, 1);
  dim3 grid=dim3((Adim0+31)/32,
    (Bdim1+31)/32, 1);
  GPUfunction_cuda<<<grid,block>>>(AGPU,
    Adim0, Adim1,BGPU, Bdim1,CGPU);
  /* GPU Memory copy from */
  cudaMemcpy(C, CGPU,CGPUs*sizeof(double),
    cudaMemcpyDeviceToHost);
}
```

4.1.1 Timing

Timing of the function Multiply for matrix multiplication was done for different sizes (n) of square matrices. Table 1 summarizes the execution times on CPU and on GPU and the speedup factor.

Table 1: GPU Speed-up for matrix multiplication. The speedup values show the CPU/GPU time ratio.

N	CPU[s]	GPU[s]	Speedup
50	0.000453	0.000438	1.03
100	0.00362	0.0018	2.01
200	0.0275	0.00996	2.76
500	0.506	0.142	3.56
1000	6.37	1.11	5.74

Note that the CPU-performance for large matrices is sensitive to caches; which for n being a power of 2 can increase the CPU time by up to a factor of 3; the chosen dimensions avoid that effect.

4.2 Cold Plate

As a second application example, a cold-plate is modeled. These are, for instance, used to cool power-electronics. The dissipated heat is transported away from the source by conduction and convection. In this two-dimensional example, a single fluid pipe is surrounded by two rectangular conducting plates. For the sake of clarity, we created a simple monolithic model that can mentally be split up into three kinds of cells: thermal conduction cells, fluid volume cells, and fluid flow cells. This is illustrated in Figure 1.

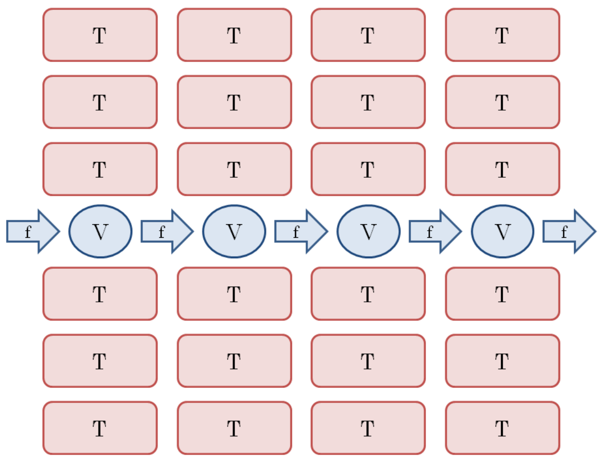

Figure 1: Illustration of a simple cold plate model, split up into thermal (T), fluid volume (V), and fluid flow (f)-cells.

Overall, the behavior of the cells is similar to models in the Modelica.Fluid and Modelica.Thermal. HeatTransfer domains. The number of cells in both dimensions is configurable, tuning the number of variables and states. In the thermal conduction cells, heat is stored and conducted to the four neighboring thermal cells or fluid volume cells. This is a slight simplification, as the resulting dynamic is anisotropic. In the fluid volume cells, balance equations for mass and energy are established.

Fluid is transported between the volume cells by flow-cells. These calculate the mass flow based on the pressure values of the neighboring volume cells by the function Modelica.Fluid.Pipes.BaseClasses.Wall-Friction.Detailed.massFlowRate_dp(). This calculation is quite involved and is therefore done in parallel by the GPU as shown in the following Modelica code, i.e. since the kernel function calls massFlowRate_dp, CUDA code is generated as a device function.

```
function WallFriction
  input Real dp[:];
  input Real d[size(dp,1)-1];
  input Real da;
  input Real db;
  input Real v[size(dp,1)-1];
  input Real va;
  input Real vb;
  input Real celllength;
  input Real diameter;
  input Real roughness;
  input Real m_flow_small;
  output Real m_flow[size(dp,1)];
algorithm
  for i in 1:size(dp,1) loop
   m_flow[i] := if i == 1 then
     massFlowRate_dp(dp[1], da, d[1],
       va, v[1], celllength, diameter, roughness, m_flow_small)
   else if i == size(dp,1) then
    massFlowRate_dp(dp[i], d[i-1], db,
      v[nX], vb, celllength, diameter, roughness, m_flow_small)
   else
     massFlowRate_dp(dp[i], d[i-1], d[i],
       v[i-1], v[i], celllength, diameter, roughness, m_flow_small);
  end for;
  annotation(gpuFunction=true);
end WallFriction;
```

As initial conditions, the temperature of all cells was set to 295K. A constant pressure gradient of 0.01bar was applied and the inlet temperature was set to 373.15K, resulting in a heating transient. Since the fluid transport is rather stiff, a small step-size has to be applied when using the RK2 method for integration.

Table 2: GPU Timing [s] and speed-up for cold plate model

nx	ny	CPU	GPU	CPU/GPU
256	256	17.2	12.1	1.42
512	512	91.7	42.9	2.13
500	200	26.0	18.1	1.43
1000	100	28.6	20.2	1.41
2000	200	103.0	65.2	1.57

As a result of the parallelization, a speed-up by a factor of 2 was achieved in one case. Note, that the step size and simulated time are different for the different grids.

The presented example only showed a very simple model of a cold plate with a straight flow of cooling liquid. Nevertheless, we think that the measured performance gains can be roughly transferred to more complex designs.

4.3 Shallow Water

For wave power plants or off-shore constructions such as wind-turbines and oil-platforms, as well as for free floating objects such as ships, the interaction with the water surface is a key component for system simulations. For this purpose, the shallow water equations represent a set of partial differential equations (PDEs) that enable an efficient approximation of the surface dynamics (Vreugdenhil, 1994). The PDE for a 2D-surface in its simplest form is shown below:

$$\frac{\partial h}{\partial t} = -H\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right)$$

$$\frac{\partial v_x}{\partial t} = -g\frac{\partial h}{\partial x}$$

$$\frac{\partial v_y}{\partial t} = -g\frac{\partial h}{\partial y}$$

In this model, the velocity of the water flowing within the 2D surface is described by v_x and v_y. A gradient (i.e. a difference between inflow and outflow) then causes the surface height *h* to raise or fall. Spatial gradients in the surface height then cause a counteracting acceleration of the water flow. *H* and *g* are parameters of the model. *H* is chosen as *L/4*, where *L* is the length of the side of the surface area, and *g* is set to 9.81 m/s^2 to model gravity realistically.

The PDE is transformed into an ODE by discretizing the space using finite differences and a staggered grid for the velocity and surface height, as depicted in Figure 2. The resolution of this grid can be set by using the parameter n.

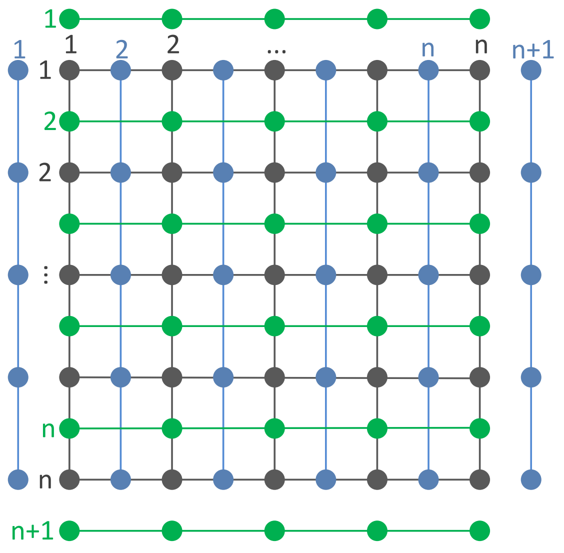

Figure 2: A staggered grid: black points symbolize the height grid $\boldsymbol{h}$. Blue represents $\boldsymbol{v_x}$ and green $\boldsymbol{v_y}$. For a size of *n*, the grid contains in total $3n^2+2n$ points.

Using this discretization, the computation can be written in form of a GPU function that performs a loop over the staggered grid:

```
function ShallowWater
  input Real h[:,:];
  input Real vx[size(h, 1) + 1,size(h, 2)];
  input Real vy[size(h, 1),size(h, 2) + 1];
  input Real dx;
  input Real L;
  input Real g;
  output Real der_h[size(h, 1),size(h, 2)];
  output Real der_vx[size(vx, 1),size(vx, 2)];
  output Real der_vy[size(vy, 1),size(vy, 2)];
protected
  Real H=L/4;
algorithm
  for iy in 1:size(h, 2) loop
    for ix in 1:size(h, 1) loop
      der_h[ix, iy] := H*(vx[ix, iy] - vx[ix + 1, iy] +
        vy[ix, iy] – vy[ix, iy + 1])/dx;
      der_vx[ix, iy] := if ix > 1 then
        g*(h[ix - 1, iy] – h[ix, iy])/dx else 0;
      der_vy[ix, iy] := if iy > 1 then
        g*(h[ix, iy - 1] – h[ix, iy])/dx else 0;
    end for;
  end for;
  annotation(gpuFunction=true);
end ShallowWater;
```

This Modelica function is then included in a complete Modelica model. This model also describes the boundary conditions (closed boundary with zero velocity) and the initial state (zero velocity with a surface height that forms a Gaussian bell curve in the center). In addition, the model generates data for visualization.

The model can be simulated using, for example, a Runge-Kutta method of second order (RK2) with fixed step size of 40ms. Figure 3 shows the simulation result for *n*=64 after the Gaussian bell has “dropped” and created a typical circular wave front that grows until it is reflected at the boundaries.

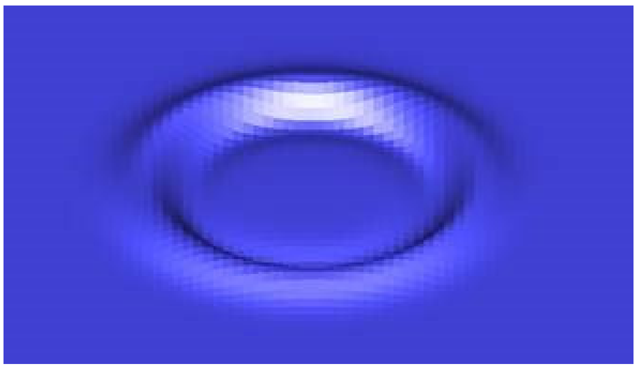

Figure 3: Dymola animation of a circular wave front in the shallow water model.

Given its large number of states, the model is well-suited for parallel computation on a GPU. On the other side, the actual computation of an element is relatively cheap in comparison to the required communication overhead with its neighbor cells.

To improve performance, the time integration was performed directly on the GPU, using the explicit Euler method (using code similar to the previously described function integrateF). The states could then be sampled from the GPU memory at a desired rate. Table 3 presents the results of our performance measurements. For larger models it becomes greater than a factor of 5. In the table, *nSteps* is the number of iterations between each sample, and the results shows how important it can be to avoid unnecessary copying.

Table 3: GPU Speed-up for the shallow water simulation, using inlined integration. The values show the CPU/GPU time ratio.

nSteps\n	32	64	128	256
1	0.36	0.77	0.91	1.00
10	0.46	0.94	1.19	1.42
100	0.77	1.89	3.03	3.19
1000	1.01	3.06	5.30	5.12

The values in the table show the CPU/GPU time ratio of the simulations. *n* is the number of cells in one dimension. The CPU simulations are made by the same Modelica code, doing inline integration on the CPU instead.

Using GPU parallelization, PDEs for shallow water simulation can be better performed in combination with classic system simulation. In this way, the practical application range of Modelica can be extended.

5 Conclusions

Modelica models are getting more and more complex which means that simulations must be performed more efficiently. This paper demonstrates a technique to generate GPU code for Modelica functions in order to speed-up simulations by parallel execution on many GPU cores. No Modelica extensions are needed, only an annotation indicating that a certain function might be suited for execution on the GPU.

In our prototype implementation, and using the GPU of a laptop, a speed-up of 5 could be achieved in some cases.

Acknowledgements

This work has partly been performed as a master thesis project at Lund Institute of Technology. The first author served as an industrial advisor and Michael Doggett as the formal supervisor.

This paper is partly based on research performed within the ITEA2 project MODRIO. Partial financial support of the Swedish VINNOVA is highly appreciated.

Helpful discussions with Sven Erik Mattsson are appreciated. The cold plate model originated from input of Daniel Bender.

References

Dassault Systèmes (2015): Dymola 2016. http://www.Dymola.com

Elmqvist H., Mattsson S.E., Olsson H. (2014): Parallel Model Execution on Many Cores. Proceedings of the 10th International Modelica Conference March 10-12, 2014, Lund, Sweden.

Gebremedhin M., Hemmati Moghadam A., Fritzson F., Stavåker K. (2012): A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms. Proceedings 9th Modelica Conference, Munich, Germany, September 3-5, pp. 393-404. Download: http://www.ep.liu.se/ecp/076/041/ecp12076041.pdf

Goteman A., Roxling V. (2015): GPU Usage for Parallel Funcions and Contacts in Modelica, master's thesis Lund Institute of Technology, Lund, Sweden. (To be published)

Kirk D.B., Hwu W. (2013): Programming Massively Parallel Processors, 2nd edition.

Modelica (2014): Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.3, Revision 1, June 11, 2014. https://www.modelica.org/documents/ModelicaSpec33Revision1.pdf

Vreugdenhil C.B. (1994), Numerical Methods for Shallow-Water Flow, Kluwer Academic Publishers, ISBN 0792331648