XIX International Conference on Water Resources
CMWR 2012
University of Illinois at Urbana-Champaign
June 17-22,2012

EXPLICIT SHALLOW WATER SIMULATIONS ON GPUS: GUIDELINES AND BEST PRACTICES

André R. Brodtkorb*, Martin L. Sætra†

*SINTEF ICT, Department of Applied Mathematics, NO-0314 Oslo, Norway.
†Centre of Mathematics for Applications, University of Oslo, NO-0316 Oslo, Norway.
e-mail:*Andre.Brodtkorb@sintef.no, †m.l.satra@cma.uio.no

Key words: Shallow Water Equations, GPUs, Best Practices

Summary. Graphics processing units have now been used for scientific calculations for over a decade, going from early proof-of-concepts to industrial use today. The inherent reason is that graphics processors are far more powerful than CPUs when it comes to both floating point operations and memory bandwidth, illustrated by the fact that three of the top 500 supercomputers in the world now use GPU acceleration. In this paper, we present guidelines and best practices for harvesting the power of graphics processing units for shallow water simulations through stencil computations.

1 Introduction

The shallow water equations are a set of hyperbolic partial differential equations, and as such can be solved using explicit finite difference and finite volume schemes with compact stencils; schemes that map very well to the hardware architecture of graphics processing units (GPUs). In fact, GPUs have been used successfully for solving the shallow water equations since the very beginning of GPU computing (see, e.g., [5]). The most powerful processor in everything from laptops to supercomputers today is typically the GPU, and this is the single most important reason for using GPUs for general purpose computations. In this paper, we present guidelines and best practices for implementing explicit stencil based shallow water solvers on GPUs. Our main focus is on NVIDIA hardware, as this is the most used platform in academia, but most of the techniques described also apply to GPUs from AMD. We briefly introduce the GPU and the shallow water equations in this section, before we in Section 2 describe strategies for mapping the shallow water equations to the GPU, and summarize in Section 3.

Graphics processing units: Dedicated processors that accelerated graphics operations were introduced in the 80'ies to offload demanding graphics from the CPU, and in 1999 NVIDIA coined the term GPU with the release of the GeForce 256 graphics card. Around the same time we also saw the first use of GPUs for non-graphics applications [10]. These early graphics cards accelerated a fixed set of graphics operations such as vertex

transformations, lighting and texturing, later the fixed functionality has gradually been replaced with fully programmable *shading*. In 2007, NVIDIA released CUDA [4], a language dedicated to GPU computing, which sparked a huge interest in the use of GPUs for scientific applications [9]. Today, we see an ever increasing trend of using the GPU to accelerate high-performance computing, and three of the top five supercomputers on the top 500 list [8] now utilize GPUs.

The major difference between CPUs and GPUs is their architectural design. Current multi-core CPUs are designed for simultaneous execution of multiple applications, and use complex logic and a high clock frequency to execute each application in the shortest possible time. GPUs, on the other hand, are designed for calculating the color of millions of screen pixels in parallel from a complex 3D game world. This essentially means that while CPUs are designed to minimize single thread latency, GPUs are designed for throughput. This is also reflected in how the transistors are used. CPUs use most of their *transistor budget* on huge caches and complex logic to minimize latencies, leaving a very small percentage of transistors for actual computations. GPUs, on the other hand, spend most of their transistor budget on computational units, and have very limited caches and very little of the complex logic found in CPUs. If we compare the state of the art, CPUs such as the Intel Core i7-3960X can have up-to 6 cores × 8-way SIMD = 48 floating point units, whilst GPUs such as the NVIDIA GeForce 580 GTX can have up-to 16 cores × 32-way SIMD = 512, an increase of an order of magnitude[1].

The shallow water equations: The shallow water equations are applicable for a wide range of problems, such as dam breaks, inundations, oceanographic currents, avalanches, and other phenomena and scenarios in which the governing flow is horizontal. This system of equations is also representative of the wider class of hyperbolic partial differential equations, and techniques developed for the shallow water equations are also often applicable to other hyperbolic conservation laws. In the simplest case, the homogeneous shallow water equations in two spatial dimensions can be written

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \tag{1}$$

or in vector form, $Q_t + F(Q)_x + G(Q)_y = 0$. Here, Q is our vector of conserved variables, and F and G are flux functions that give rise to gravity waves. In the case of water, h will be the water depth, hu and hv is the momentum along the abscissa and ordinate, respectively, and g is the gravitational acceleration. In this paper, we focus on modern explicit schemes with compact stencils that solve these equations (see, e.g., [13, 7]), as these schemes are often highly suitable for implementation on GPUs.

[1]This comparison assumes single precision operations. It should also be mentioned that a GPU core is quite different from a CPU core, and we refer the reader to [4] for a full overview.

2 Mapping the Shallow Water Equations to the GPU

Explicit schemes with compact stencils map well to the GPU architecture, since each output element can be computed independently of all other elements, giving rise to a high level of parallelism. In this section, we will illustrate how the classical Lax-Friedrichs finite volume scheme can be mapped to the GPU as an example. It should be noted that the presented code is for illustrative purposes only, and disregard many important optimization parameters. Let us start by writing up the classical Lax-Friedrichs scheme for a volume (i, j):

$$\begin{aligned} Q_{ij}^{n+1} = \tfrac{1}{4}\left(Q_{i,j+1}^n + Q_{i,j-1}^n + Q_{i+1,j}^n + Q_{i-1,j}^n\right) \\ - \tfrac{\Delta t}{2\Delta x}\left[F(Q_{i+1,j}^n) - F(Q_{i-1,j}^n)\right] - \tfrac{\Delta t}{2\Delta y}\left[G(Q_{i,j+1}^n) - G(Q_{i,j-1}^n)\right]. \end{aligned} \tag{2}$$

Here, we explicitly calculate the vector Q at the next time step, $(n+1)\Delta t$, using the stencil containing our four nearest neighbors. A traditional CPU algorithm that evolves the solution one time step can often be similar to the following:

```
for (int j=1; j<ny-1; ++j) {
  for (int i=1; i<nx-1; ++i) {
    int n=(j+1)*nx+i, s=(j-1)*nx+i, e=j*nx+i+1, w=j*nx+i-1;
    h_new[j*nx+i] = 0.25*(h[n]+h[s]+h[e]+h[w])
        - 0.5*dt/dx*(hu[e]-hu[w]) - 0.5*dt/dy*(hv[n]-hv[s]);
  }
}
```

Here we have shown the code for computing h^{n+1} for each internal volume in the discretization, and hu^{n+1} and hv^{n+1} would be computed similarly. Because the computation of volume (i, j) is independent, we may solve for all volumes in parallel, and this is what we exploit when mapping the computations to the GPU. The first thing we do is to identify the independent parallel section of our code, and write it as a GPU *kernel*, shown in the following example:

```
__global__ void LaxFriedrichs(float* h_new,
        float* h, float* hu, float* hv, int nx, int ny) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;
  if (i > 0 && i < nx-1 && j > 0 && j < ny-1) {
    int n=(j+1)*nx+i, s=(j-1)*nx+i, e=j*nx+i+1, w=j*nx+i-1;
    h_new[j*nx+i] = 0.25*(h[n]+h[s]+h[e]+h[w])
        - 0.5*dt/dx*(hu[e]-hu[w]) - 0.5*dt/dy*(hv[n]-hv[s]);
  }
}
```

In this kernel, the variables `h_new`, `h` etc. are physically located in GPU memory. We use the variables i and j to index this memory. We *launch* this kernel for each finite volume on the GPU using a *grid* of *blocks* to execute it on the GPU (see Figure 1):

```
dim3 block(16, 12);
dim3 grid(ceil(nx/16.0), ceil(ny/12.0));
LaxFriedrichs<<<grid, block>>>(h_new, h, hu, hv, nx, ny);
```

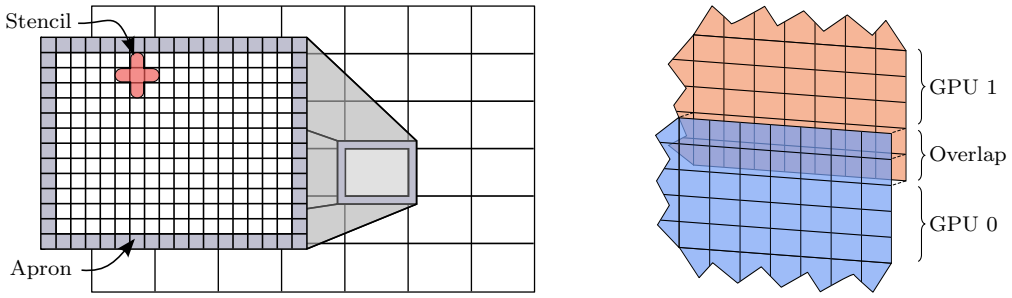

Figure 1: (Left) Illustration of the block and grid concepts with one block hilighted. The *apron* (also called local ghost cells) is used to fulfill the requirements of the stencil, so that all blocks may be executed independently and in parallel by the GPU. (Right) Multi-GPU domain decomposition using overlapping ghost cell regions. The two GPUs exchange the overlapping ghost cell region after every simulation step.

This creates one thread per volume which the GPU executes in parallel. In addition to the code shown above, we also need to allocate memory on the GPU, copy initial conditions from the CPU to the GPU, and copy results back after the desired simulation time has been reached. In both the CPU and the GPU code, we also need to implement boundary conditions, for example using global ghost cells that in general must be updated before every simulation stage.

The presented GPU code launches nx $\times$ ny threads organized into blocks of 16×12. On the GPU, each block is assigned to one of many processors, and one processor can hold multiple blocks, which is used to hide memory latencies. We typically achieve best performance with a large number of blocks, but determining the optimal block size is often a very difficult task. One important parameter here is the 32-way SIMD nature of GPUs, that is, 32 consecutive threads must execute the same instruction on different data for full performance. Another key optimization parameter is *shared memory*. In the code above, each output element is computed from five input elements which have to be read from *global* GPU memory. On the CPU, the input variables would automatically be *cached* for performance. On the GPU, however, we must manually place data in shared memory, a type of programmable cache available to all threads within the same block. We can do this by loading one data element per thread, in addition to the *apron*, into shared memory (see Figure 1). Such a strategy gives us the classical block domain decomposition, in which each CUDA block has an input domain which overlaps with its neighboring blocks, allowing it to execute independently. This means that for a large block size we on average read just over *one* element per thread, compared to eight without the use of shared memory. However, the shared memory is limited in size, thereby limiting our block size. There are several other important optimization parameters, and we refer the reader to [3] for a more thorough discussion of these.

Multi-GPU: The technique presented above for executing CUDA blocks in parallel through the use of domain decomposition can also be used to enable parallel simulations

on multiple GPUs. Multi-GPU simulations are highly attractive for simulating very large domains or when performance requirements are very high. Modern computers can be equipped with up-to four dual-GPUs on a single chassis, effectively creating a desktop supercomputer. However, whilst the block decomposition is highly suitable within one GPU because of the limited shared memory size, it can often be better to use a row decomposition for multi-GPU simulations within one node (see Figure 1). The row decomposition minimizes the overlapping areas of the domain, and also communication costs, as each node has at most two neighbors (as opposed to four for the block decomposition). Before each simulation step, one simply exchanges the overlapping regions between the two GPUs, thereby coupling the two otherwise independent simulations.

The GPU is located on the PCI-express bus, and all communication between different GPUs is therefore slow, especially in terms of latency. Simulations on multiple GPUs may therefore suffer from this slow communication between the GPUs. One way of alleviating this is to use *ghost cell expansion*, in which the size of the overlap between two GPUs is increased. By doing this, we can simulate more than one time step before exchanging information between GPUs, as disturbances will at most travel one cell per time step. Thus, for an overlap of six cells, we would be able to run three time steps before having to exchange the overlapping region. However, the cost is that we must now calculate the flow in the overlapping cells on both GPUs, meaning there is a trade-off. Our experience shows that an overlap of on the order of tens of cells yields highest performance, allowing near-perfect weak *and* strong scaling for representative domain sizes [12]. An extension to this technique is to overlap data transfers with computation, which can be done by solving for the internal cells of the domain simultaneously as the overlap region is being exchanged.

Sparse Simulations: Many real-world scenarios will often have large areas without water, such as in simulation of dam breaks and inundations near riverbanks and coastal regions. These dry areas do not require any computation, as the compact stencil ensures that water will travel at most one cell per time step. Traditional techniques, however, often perform some calculations on these cells before discarding the results if the cell is dry, effectively wasting both memory bandwidth and floating point operations. One particularly effective approach to address this shortcoming on the GPU is to use sparse simulations (see Figure 2) [11]. Sparse simulations are based around the grid concept of CUDA, whereby only blocks requiring computation are launched. At the end of each time step, each block stores -1 in a buffer if it is dry and no water may flow into it in the next time step, or its grid position otherwise. At the next time-step, this buffer is sorted so that the non-negative indices come first, and a grid with of one block per non-negative index is launched. The extra sorting of the buffer incurs a small performance penalty, but being able to skip dry regions altogether yields a great performance increase on GPUs (see Figure 3).

The technique of skipping dry parts of the domain can also be extended to the data representation, whereby dry parts of the domain are not represented on the GPU at all.

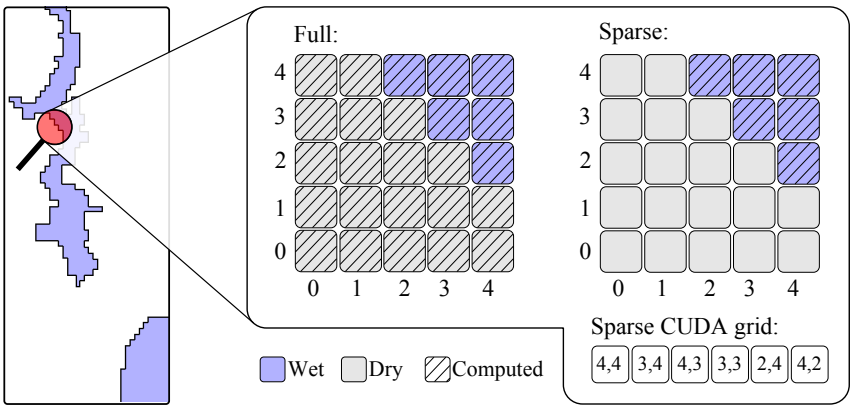

Figure 2: Dry map of a real-world dam break case, with blocks containing water marked. By storing whether or not a block contains water, we can create a list of blocks that require computation before each time step, and launch the *kernel* only on these blocks. This saves both computation and bandwidth, yielding a significant performance improvement on typical domains (see also Figure 3).

This essentially follows the same procedure, but the data layout is changed so that wet blocks are stored after one another in GPU global memory. Such a sparse memory layout is especially attractive for extremely large domains with little water, and even problems where the full domain would otherwise not fit in GPU global memory. However, the altered data layout makes the process of reading the apron slightly more complicated, as neighboring blocks are no longer neighbors in the physical memory layout.

Accuracy and Performance: A requirement for developing numerical codes on the GPU that accurately captures the physical reality is to choose good verification and validation cases first, and then to optimize the code only after it gives the correct results. A classical problem one soon encounters in this process is that floating point arithmetic is not commutative due to round off errors, that is $a_f + b_f \neq b_f + a_f$. This is important to note when executing parallel code, as the sequence of floating point operations between different execution units often will be non-deterministic.

A further difficulty with numerical codes is that the inevitable floating point errors can blow up, and one often tends to use double precision calculations instead of single precision to counter this. However, single precision may in many cases still be the best choice. First of all, using single precision gives you roughly double the performance, because the size of your data is halved and single precision operations are twice as fast. Furthermore, it is often the case that modeling errors, measurement errors, and other factors shadow the errors imposed by using single precision. For example in [2], the handling of dry states completely masks the errors introduced by using single precision arithmetic for the target scenarios, meaning single precision is sufficiently accurate. However, it is still important to keep floating point errors in mind when implementing all numerical codes. As another example, let us consider a numerical scheme based on the water elevation instead of the water depth (e.g., the Kurganov-Petrova scheme [6]). In such a scheme, it

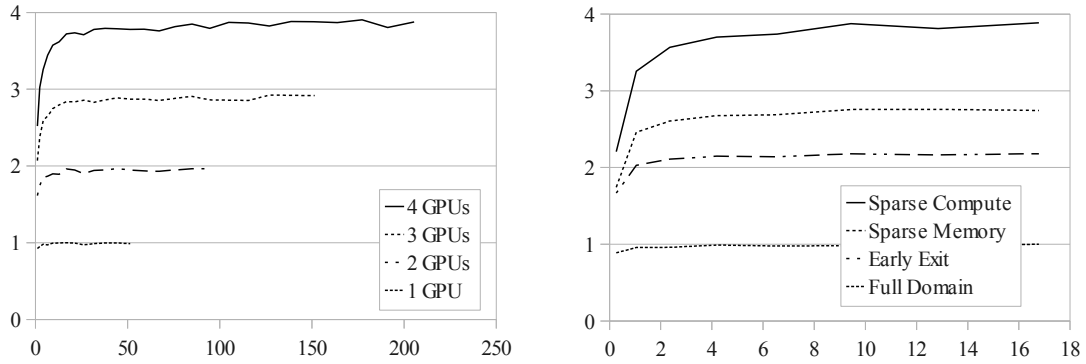

Figure 3: (Left) Performance as a function of domain size in millions of cells for 1 to 4 GPUs. The simulation is a wet bed circular dam break, and shows near-perfect weak and strong scaling for sufficiently large domains. The performance is normalized with respect to the fastest 1 GPU run. (Right) Performance as a function of domain size in millions of cells for different algorithms used to increase the computational throughput for a dry bed circular dam break. The average number of wet cells is approximately 26%. Full domain computes all cells, early exit launches and then exits exits blocks without water, sparse memory stores and computes only on wet blocks, and sparse compute computes only on wet blocks. All graphs are normalized with the fastest full domain run as the reference.

is often tempting to store the water elevation in memory. This, however, will give rise to large relative errors when the water depth is small compared to the elevation. The reason is that floating point numbers are most accurately represented when close to zero. Thus, for a small water depth at a large elevation, the round off errors will often lead to large floating point errors in the results. Therefore, it may be important to store the quantity of interest as a number close to zero in memory, and then reconstruct derived quantities on demand.

Thorough performance assessment for GPUs is often difficult and somewhat neglected. Many papers that are published unfortunately relay overly optimistic speedup figures over CPU codes. This is problematic because when one examines the theoretical performance gap between the architectures, which currently lies at around 7 times [1], it is clear that speedup claims of hundreds or more require a thorough explanation. Thus, our view is that a good performance assessment focuses on identifying bottlenecks of the algorithm and reporting the attained percentage of peak performance through careful profiling. This will give the viewer a much more balanced view of the algorithm, and more importantly, clearly identify directions for further research. Profiling can be performed using Parallel NSight, which is a superb tool for profiling of GPU codes directly in Visual Studio, and similar tools such as the CUDA Profiler also exist for Linux and OS X.

3 Summary

We have presented general guidelines and best practices for mapping explicit shallow water schemes with compact stencils to graphics processing units. These schemes are

naturally suited for the execution model of modern GPUs, and can give unprecedented simulation speeds. We have furthermore discussed strategies for expanding to multi-GPU simulations and strategies for avoiding computing dry areas of the simulation domain. Finally, we have presented floating point considerations with respect to accuracy versus performance, and best practices for performance assessment.

Acknowledgement: Part of this work is supported by the Research Council of Norways project number 180023 (Parallel3D) and the Norwegian Meteorological Institute. The authors acknowledge the continued support from NVIDIA.

REFERENCES

[1] A. R. Brodtkorb. *Scientific Computing on Heterogeneous Architectures*. PhD thesis, University of Oslo, 2010.

[2] A. R. Brodtkorb, T. R. Hagen, K.-A. Lie, and J. R. Natvig. Simulation and visualization of the saint-venant system using GPUs. *Computing and Visualization in Science*, 13:341–353, 2011.

[3] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55:1–12, 2012.

[4] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann. Elsevier Science, 2011.

[5] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, and M. Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[6] A. Kurganov and G. Petrova. A second-order well-balanced positivity preserving central-upwind scheme for the Saint-Venant system. *Communications in Mathematical Sciences*, 5:133–160, 2007.

[7] R. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[8] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 supercomputer sites. `http://www.top500.org/`, December 2011.

[9] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[10] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005*, pages 21–51, 2005.

[11] M. L. Sætra. Sparse grid shallow water simulation on GPUs. In *Proceedings of ENUMATH 2011*, Leicester, UK, 2012.

[12] M. L. Sætra and A. R. Brodtkorb. Shallow water simulations on multiple GPUs. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer Berlin / Heidelberg, 2012.

[13] E. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. John Wiley & Sons, Ltd., 2001.