Shallow Water Simulations on Multiple GPUs

Martin Lilleeng Sætra[1] and André Rigland Brodtkorb[2]

[1] Center of Mathematics for Applications, University of Oslo,
P.O. Box 1053 Blindern, NO-0316 Oslo, Norway
[2] SINTEF, Dept. Appl. Math., P.O. Box 124, Blindern, NO-0314 Oslo, Norway
Email: m.l.satra@cma.uio.no, Andre.Brodtkorb@sintef.no

This is a draft version of the following article: Proceedings of the Para 2010 Conference Part II, Lecture Notes in Computer Science 7134 (2012), pp 56–66, DOI: 10.1007/978-3-642-28145-7_6. The full article can be found on `http://dx.doi.org/10.1007/978-3-642-28145-7_6`.

Abstract

We present a state-of-the-art shallow water simulator running on multiple GPUs. Our implementation is based on an explicit high-resolution finite volume scheme suitable for modeling dam breaks and flooding. We use row domain decomposition to enable multi-GPU computations, and perform traditional CUDA block decomposition within each GPU for further parallelism. Our implementation shows near perfect weak and strong scaling, and enables simulation of domains consisting of up-to 235 million cells at a rate of over 1.2 gigacells per second using four Fermi-generation GPUs. The code is thoroughly benchmarked using three different systems, both high-performance and commodity-level systems.

1 Introduction

Predictions of floods and dam breaks require accurate simulations with rapid results. Faster than real-time performance is of the utmost importance when simulating these events, and traditional CPU-based solutions often fall short of this goal. We address the performance of shallow water simulations in this paper through the use of multiple graphics processing units (GPUs), and present a state-of-the-art implementation of a second-order accurate explicit high-resolution finite volume scheme.

There has been a dramatic shift in commodity-level computer architecture over the last five years. The steady increase in performance does no longer come from higher clock frequencies, but from parallelism through more arithmetic units: The newest CPU from Intel, for example, contains 24 single precision arithmetic units (Core i7-980X). The GPU takes this parallelism even further with up-to 512 single precision arithmetic units (GeForce GTX 580). While the GPU originally was designed to offload a predetermined set of demanding graphics operations from the CPU, modern

GPUs are now fully programmable. This makes them suitable for general purpose computations, and the use of GPUs has shown large speed-ups over the CPU in many application areas [1, 2]. The GPU is connected to the rest of the computer through the PCI Express bus, and commodity-level computers can have up-to two GPUs connected at full data speed. Such solutions offer the compute performance comparable to a small CPU cluster, and this motivates the use of multiple GPUs. In fact, three of the five fastest supercomputers use GPUs as a major source of computational power [3]. However, the extra floating-point performance comes at a price, as it is nontrivial to develop efficient algorithms for GPUs, especially when targeting multiple GPUs. It requires both different programming models and different optimization techniques compared to traditional CPUs.

Related Work: The shallow water equations belong to a wider class of problems known as hyperbolic conservation laws, and many papers have been published on GPU-acceleration of both conservation and balance laws [4, 5, 6, 7, 8, 9, 10]. There have been multiple publications on the shallow water equations as well [11, 12, 13, 14, 15, 16], illustrating that these problems can be efficiently mapped to modern graphics hardware. The use of multiple GPUs has also become a subject of active research. Micikevicius [17] describes some of the benefits of using multiple GPUs for explicit finite-difference simulation of 3D reverse time-migration (the linear wave equation), and reports super-linear speedup when using four GPUs. Overlapping computation and communication for explicit stencil computations has also been presented for both single nodes [18] and clusters [19] with near-perfect weak scaling. Perfect weak scaling was shown by Acuña and Aoki [20] for shallow water simulations on a cluster of 32 GPU nodes, by overlaping computations and communication. Rostrup and De Sterck [21] further present detailed optimization and benchmarking of shallow water simulations on clusters of multi-core CPUs, the Cell processor, and GPUs. Comparing the three, the GPUs offer the highest performance.

In this work, we focus on single-node systems with multiple GPUs. By utilizing more than one GPU it becomes feasible to run simulations with significantly larger domains, or to increase the spatial resolution. Our target architecture is both commodity-level computers with up-to two GPUs, as well as high-end and server solutions with up-to four GPUs at full data speed per node. We present a multi-GPU implementation of a second-order well-balanced positivity preserving central-upwind scheme [22]. Furthermore, we offer detailed performance benchmarks on three different machine setups, tests of a latency-hiding technique called ghost cell expansion, and analyzes of benchmark results.

2 Mathematical Model and Discretization

In this section, we give a brief outline of the major parts of the implemented numerical scheme. For a detailed overview of the scheme, we refer the reader to [22, 23]. The shallow water equations are derived by depth-integrating the Navier-Stokes equations, and describe fluid motion under a pressure surface where the governing flow is horizontal. To correctly model phenomena such as tsunamis, dam breaks, and flooding over realistic terrain, we need to include source terms for bed slope and friction:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} + \begin{bmatrix} 0 \\ -gu\sqrt{u^2+v^2}/C_z^2 \\ -gv\sqrt{u^2+v^2}/C_z^2 \end{bmatrix}. \tag{1}$$

Here h is the water depth and u and v are velocities along the abscissa and ordinate, respectively. Furthermore, g is the gravitational constant, B is the bottom topography, and C_z is the Chézy friction coefficient.

To be able to simulate dam breaks and flooding, we require that our numerical scheme handles wetting and drying of cells, a numerically challenging task. However, we also want other properties, such as well-balancedness, accurate shock-capturing without oscillations, at least second order accurate flux calculations, and that the computations map well to the architecture of the GPU. A scheme that fits well with the above criteria is the explicit Kurganov-Petrova scheme [22], which is based on a standard finite volume grid. In this scheme, the physical variables are given as cell averages, the bathymetry as a piecewise bilinear function (represented by the values at the cell corners), and fluxes are computed across cell interfaces (see also Figure 1). Using vectorized notation, in which $Q = [h, hu, hv]^T$ is the vector of conserved variables, the spatial discretization can be written,

$$\begin{aligned} \frac{dQ_{ij}}{dt} &= H_f(Q_{ij}) + H_B(Q_{ij}, \nabla B) - \big[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\big] \\ &\qquad\qquad\qquad\qquad\qquad\qquad - \big[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\big] \\ &= H_f(Q_{ij}) + R(Q)_{ij}. \end{aligned} \tag{2}$$

Here $H_f(Q_{ij})$ is the friction source term, $H_B(Q_{ij}, \nabla B)$ is the bed slope source term, and F and G are the fluxes across interfaces along the abscissa and ordinate, respectively. We first calculate $R(Q)_{ij}$ in (2) explicitly, and as in [23], we use a semi-implicit discretization of the friction source term,

$$\tilde{H}_f(Q_{ij}^k) = \begin{bmatrix} 0 \\ -g\sqrt{{u_{ij}^k}^2 + {v_{ij}^k}^2}/h_{ij}^k {C_z}_{ij}^2 \\ -g\sqrt{{u_{ij}^k}^2 + {v_{ij}^k}^2}/h_{ij}^k {C_z}_{ij}^2 \end{bmatrix}. \tag{3}$$

This yields one ordinary differential equation in time per cell, which is then solved using a standard second-order accurate total variation diminishing Runge-Kutta scheme [24],

$$\begin{aligned} Q_{ij}^* &= \big[Q_{ij}^n + \Delta t R(Q^n)_{ij}\big] / \big[1 + \Delta t \tilde{H}_f(Q_{ij}^n)\big] \\ Q_{ij}^{n+1} &= \Big[\tfrac{1}{2}Q_{ij}^n + \tfrac{1}{2}\big[Q_{ij}^* + \Delta t R(Q^*)_{ij}\big]\Big] / \big[1 + \tfrac{1}{2}\Delta t \tilde{H}_f(Q_{ij}^*)\big], \end{aligned} \tag{4}$$

or a first-order accurate Euler scheme, which simply amounts to setting $Q^{n+1} = Q^*$. The timestep, Δt, is limited by a CFL condition,

$$\Delta t \leq \tfrac{1}{4}\min_\Omega \Big\{ \big|\Delta x/\lambda_x\big|, \big|\Delta y/\lambda_y\big| \Big\}, \qquad \begin{aligned} \lambda_x &= u \pm \sqrt{gh}, \\ \lambda_y &= v \pm \sqrt{gh} \end{aligned} \tag{5}$$

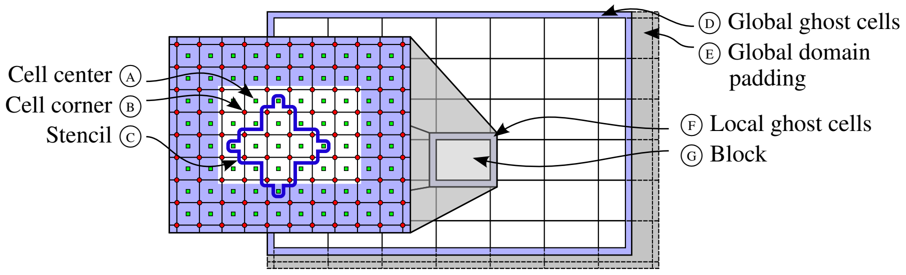


Figure 1: Domain decomposition and variable locations for the single-GPU simulator. The global domain is padded Ⓔ to fit an integer number of CUDA blocks, and global ghost cells Ⓓ are used for boundary conditions. Each block Ⓖ has local ghost cells Ⓕ that overlap with other blocks to satisfy the data dependencies dictated by the stencil Ⓒ. Our data variables Q, R, H_B, and H_f are given at grid cell centers Ⓐ, and B is given at grid cell corners Ⓑ.

that ensures that the fastest numerical propagation speed is at most one quarter grid cell per timestep.

In summary, the scheme consists of three parts: First fluxes and explicit source terms are calculated in (2), before we calculate the maximum timestep according to the CFL condition, and finally evolve the solution in time using (4). The second-order accurate Runge-Kutta scheme for the time integration is a two-step process, where we first perform the above operations to compute Q^*, and then repeat the process to compute Q^{n+1}.

3 Implementation

Solving partial differential equations using explicit schemes implies the use of stencil computations. Stencil computations are embarrassingly parallel and therefore ideal for the parallel execution model of GPUs. Herein, the core idea is to use more than one GPU to allow faster simulation, or simulations with larger domains or higher resolution. Our simulator runs on a single node, enabling the use of multithreading, and we use one global *control* thread in addition to one *worker* thread per GPU. The control thread manages the worker threads and facilitates domain decomposition, synchronization, and communication. Each worker thread uses a modified version of our previously presented single-GPU simulator [23] to compute on its part of the domain.

Single-GPU Simulator: The single-GPU simulator implements the Kurganov-Petrova scheme on a single GPU using CUDA [25], and the following gives a brief overview of its implementation. The simulator first allocates and initializes data according to the initial conditions of the problem. After initialization, we repeatedly call a step function to advance the solution in time. The step function executes four CUDA *kernels*

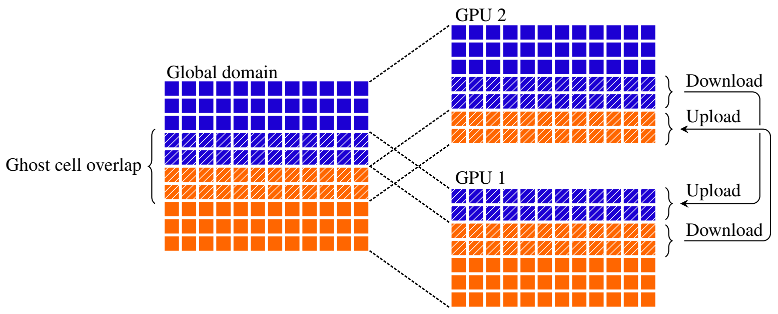


Figure 2: Row decomposition and exchange of two rows of ghost cells. The shaded cells are a part of the overlapping ghost cell region between subdomains.

in order, that together implement the numerical scheme. The first kernel computes the fluxes across all interfaces, and is essentially a complex stencil computation. This kernel reads four values from global memory, performs hundreds of floating point operations, and writes out three values to global memory again. It is also the most time consuming kernel, with over 87% of the runtime. The next kernel finds the maximum wave speed in the domain, and then computes the timestep size according to the CFL condition. The third kernel simply solves the ordinary differential equations in time to evolve the solution. Finally, the fourth kernel applies boundary conditions by setting the values of global *ghost cells* (see Figure 1).

Threaded Multi-GPU Framework: When initializing our simulator, the control thread starts by partitioning the global domain, and continues by initializing one worker thread per subdomain, which attaches to a separate GPU. We can then perform simulation steps, where the control thread manages synchronization and communication between GPUs. An important thing to note about this strategy is that the control thread handles all multi-GPU aspects, and that each GPU is oblivious to other GPUs, running the simulation on its subdomain similar to a single-GPU simulation.

We use a row domain decomposition, in which each subdomain consists of several rows of cells (see Figure 2). The subdomains form overlapping regions, called ghost cells, which function as boundary conditions that connect the neighbouring subdomains. By exchanging the overlapping cells before every timestep, we ensure that the solution can propagate properly between subdomains. There are several benefits to the row decomposition strategy. First of all, it enables the transfer of continuous parts of memory between GPUs, thus maximizing bandwidth utilization. A second benefit is that we can minimize the number of data transfers, as each subdomain has at most two neighbours. To correctly exchange ghost cells, the control thread starts by instructing each GPU to download its ghost cells to *pinned* CPU memory, as direct GPU to GPU-transfers are currently not possible. The size of the ghost cell overlap is dictated by the stencil, which in our case uses two values in each direction (see Figure 1). This means that we need to have an overlap of four rows of cells, two from each of the subdomains.

After having downloaded the ghost cells to the CPU, we need to synchronize to guarantee that all downloads have completed, before each GPU can continue by uploading the ghost cells coming from neighbouring subdomains. Note that for the second-order accurate Runge-Kutta time integration scheme, we have to perform the ghost cell exchange both when computing Q^* and when computing Q^{n+1}, thus two times per full timestep.

The multi-GPU simulator is based on our existing single-GPU simulator, which made certain assumptions that made it unsafe to execute from separate threads. This required us to redesign parts of the code to guarantee thread safety. A further difficulty related to multi-GPU simulation is that the computed timestep, Δt, will typically differ between subdomains. There are two main strategies to handle this problem, and we have investigated both. The simplest is to use a globally fixed timestep throughout the simulation. This, however, requires that the timestep is less than or equal to the smallest timestep allowed by the CFL condition for the full simulation period, which again implies that our simulation will not propagate as fast as it could have. The second strategy is to synchronize the timestep between subdomains for each timestep, and choose the smallest. This strategy requires that we split the step function into two parts, where the first computes fluxes and the maximum timestep, and the second performs time integration and applies boundary conditions. Inbetween these two substeps we can find the smallest global timestep, and redistribute it to all GPUs. This strategy ensures that the simulation propagates at the fastest possible rate, but at the expense of potentially expensive synchronization and more complex code.

Ghost Cell Expansion: Synchronization and overheads related to data transfer can often be a bottleneck when dealing with distributed memory systems, and a lot of research has been invested in, e.g., latency hiding techniques. In our work, we have implemented a technique called *ghost cell expansion* (GCE), which has yielded a significant performance increase for cluster simulations [26, 27]. The main idea of GCE is to trade more computation for smaller overheads by increasing the level of overlap between subdomains, so that they may run more than one timestep per ghost cell exchange. For example, by extending the region of overlap from four to eight cells, we can run two timesteps before having to exchange data. When exchanging ghost cells for every timestep, we can write the time it takes to perform one timestep as

$$w_1 = T(m) + c_T + C(m, n) + c,$$

in which m and n are the domain dimensions, $T(m)$ is the ghost cell transfer time, c_T represents transfer overheads, $C(m, n)$ is the time it takes to compute on the subdomain, and c represents other overheads. Using GCE to exchange ghost cells only every kth timestep, the average time per timestep becomes

$$w_k = T(m) + c_T/k + C(m, n + \mathcal{O}(k)) + c,$$

in which we divide the transfer overheads by k, but increase the overlap, and thus the size of each subdomain. This means that each worker thread computes on a slightly larger domain, and we have larger but fewer data transfers.

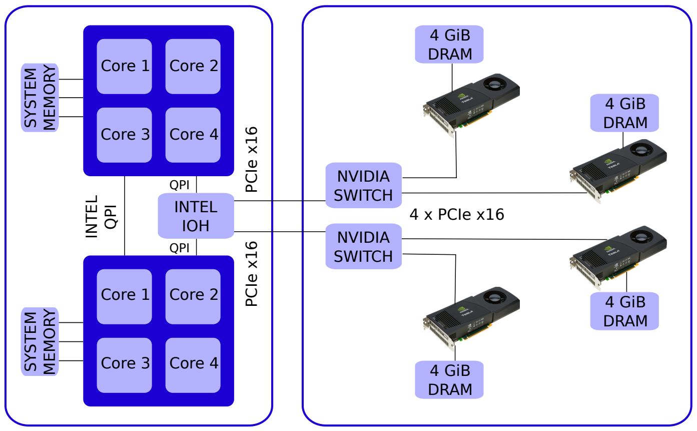


Figure 3: Hardware setup of the Tesla S1070 GPU Computing Server with four Tesla C1060 GPUs (right) connected to an IBM X3550 M2 server (left).

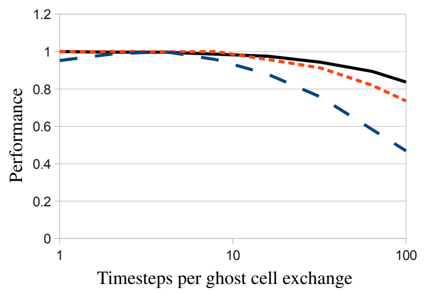
4 Results and Analysis

To validate our implementation, we have compared the multi-GPU results with the original single-GPU simulator [23], which has been both verified against analytical solutions and validated against experiments. Our multi-GPU results are identical to those produced by the single-GPU implementation, which means that the multi-GPU implementation is also capable of reproducing both analytical and real-world cases.

We have used three different systems for benchmarking our implementation. The first system is a Tesla S1070 GPU Computing Server consisting of four Tesla C1060 GPUs with 4 GiB memory each[1], connected to an IBM X3550 M2 server with two 2.0 GHz Intel Xeon X5550 CPUs and 32 GiB main memory (see Figure 3). The second system is a SuperMicro SuperServer consisting of four Tesla C2050 GPUs with 3 GiB memory each (2.6 available when ECC is enabled)[2], and two 2.53 GHz Intel Xeon E5630 CPUs with 32 GiB main memory. The third system is a standard desktop PC consisting of two GeForce 480 GTX cards with 1.5 GiB memory each and a 2.67 GHz Intel Core i7 CPU with 6 GiB main memory. The first two machine setups represent previous and current generation server GPU nodes, and the third machine represents a commodity-level desktop PC.

As our performance benchmark, we have used a synthetic circular dam break over a flat bathymetry, consisting of a square 4000-by-4000 meters domain with a water column placed in the center. The water column is 250 meters high with a radius of 250 meters, and the water elevation in the rest of the domain is 50 meters. At time $t = 0$, the dam surrounding the water column is instantaneously removed, creating an outgoing circular wave. We have used the first-order accurate Euler time integrator in

[1]Connected through two PCIe $\times16$ slots.
[2]Connected through four PCIe $\times16$ slots.



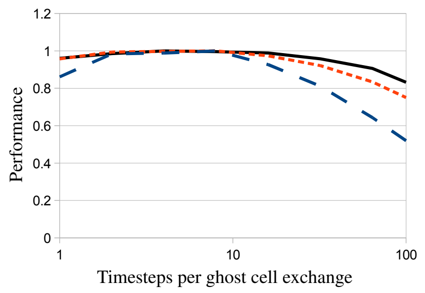


Figure 4: Testing the impact of different numbers of ghost cell expansion rows with four GPUs. Tested on both The Tesla S1070 system (see Figure 3) (left), and the Tesla C2050-based system (right). The domains tested consists of 1024^2 (dashed), 4096^2 (densely dashed) or 8192^2 (solid) cells. The graphs have been normalized relative to the peak achieved performance for each domain size.

all benchmarks, and the friction coefficient C_z is set to zero. The bed slope and friction coefficient do not affect the performance in this benchmark.

Ghost Cell Expansion We have implemented ghost cell expansion so that we can vary the level of overlap, and benchmarked three different domain sizes to determine the effect. Figure 4 shows that there is a very small overhead related to transferring data for sufficiently large domain sizes, and performing only one timestep before exchanging overlapping ghost cells actually yields the best overall results for the Tesla S1070 system. Expanding with more than eight cells, the performance of the simulator starts decreasing noticeably. From this, we reason that the overhead connected with data transfers between subdomains in these tests is negligible, compared to the transfer and computational time. Increasing the level of GCE only had a positive impact on the smallest domain for the Tesla S1070 system, where the transferred data volume is so small that the overheads become noticeable. On the Tesla C2050-based system, however, we see that the positive impact of GCE is more visible. We expect this is because this GPU is much faster, making the communication overheads relatively larger.

Our results show that ghost cell expansion had only a small impact on the shared-memory architectures we are targeting for reasonably sized grids, but gave a slight performance increase for the Tesla C2050 GPUs. This is due to the negligible transfer overheads. We thus expect GCE to have a greater effect when performing ghost cell exchange across multiple nodes, since the overheads here will be significantly larger, and we consider this a future research direction.

Since our results show that it is most efficient to have a small level of GCE for the Tesla S1070 system, we choose to exchange ghost cells after every timestep in all of our other benchmarks for this system. For the Tesla C2050-based system we exchange data after eight timesteps, as this gave the overall best results. Last, for the GeForce 480 GTX cards, which displayed equivalent behaviour to that of the Tesla C2050-based system, we also exchange ghost cells after performing eight timesteps.

Timestep Synchronization: We have implemented both the use of a global fixed timestep, as well as exchange of the minimum global timestep in our code, and benchmarked on our three test systems to determine the penalty of synchronization. In the tests we compared simulation runs with a fixed $\Delta t = 0.001$ in each subdomain, and runs with global synchronization of Δt. When looking at the results we see that the cost of synchronizing Δt globally has a negligible impact on the performance of the Tesla S1070 system, with an average 0.36% difference for domain sizes larger than five million cells on four GPUs. As expected, the cost is also roughly halved when synchronizing two GPUs compared to four (0.17%). For smaller domain sizes, however, the impact becomes noticeable, but these domains are typically not candidates for multi-GPU simulations. The Tesla C2050- and GeForce 480 GTX-based systems also display similar results, meaning that global synchronization of Δt is a viable strategy for reasonably sized domains.

Weak and Strong Scaling: Weak and strong scaling are two important performance metrics that are used for parallel execution. While varying the number of GPUs, weak scaling keeps the domain size per GPU fixed, and strong scaling keeps the global domain size fixed. As we see from Figure 5, we have close to linear scaling from one to four GPUs. For domains larger than 25 million cells the simulator displays near perfect weak and strong scaling on all three systems. Running simulations on small domains is less efficient when using multiple GPUs for two reasons: First of all, as the global domain is partitioned between more GPUs, we get a smaller size of each subdomain. When these subdomains become sufficiently small, we are unable to fully occupy a single GPU, and thus do not reach peak performance. Second, we also experience larger effects of overheads. However, we quickly get close-to linear scaling as the domain size increases.

The Tesla C1060 GPUs have 4.0 GiB of memory each, which enables very large scale simulations: When using all four GPUs, domains can have up to 379 million cells, computing at 396 megacells per second. Because the most recent Tesla C2050 GPUs from NVIDIA have only 3.0 GiB memory per GPU, our maximum domain size is smaller (235 million cells), but our simulation speed is dramatically faster. Using four GPUs, we achieve over 1.2 gigacells per second. The fastest system per GPU, however, was the commodity-level desktop machine with two GeForce 480 GTX cards. These cards have the highest clock frequency, and we achieve over 400 megacells per second

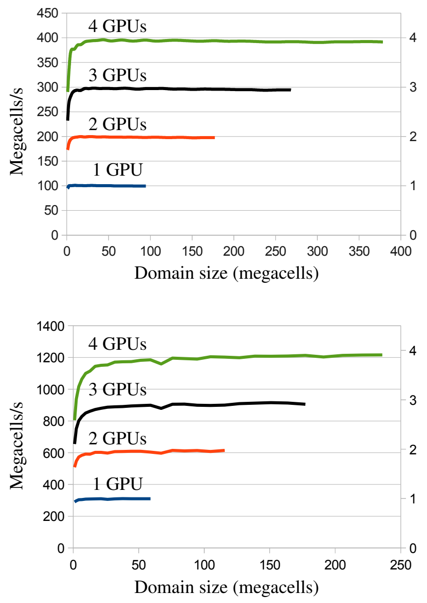


Figure 5: (Left) Performance experiment on a Tesla S1070 system (see Figure 3) with up-to four GPUs. (Right) Performance experiment on a Tesla C2050-based system, using up-to four GPUs. The secondary y-axis on the right-hand side shows scaling relative to the peak achieved performance of a single GPU.

per GPU.

5 Summary and Future Work

We have presented an efficient multi-GPU implementation of a modern finite volume scheme for the shallow water equations. We have further presented detailed benchmarking of our implementation on three hardware setups, displaying near-perfect weak *and* strong scaling on all three. Our benchmarks also show that communication between GPUs within a single node is very efficient, which enables tight cooperation between subdomains.

A possible further research direction is to explore different strategies for domain decomposition, and especially to consider techniques for adaptive domain decompositions.

Acknowledgements: Part of this work is supported by the Research Council of Norway's project number 180023 (Parallel3D), and the authors are grateful for the continued support from NVIDIA. Thanks also to Ole W. Saastad for access to the Teflon GPU compute cluster at the University of Oslo.

References

[1] Brodtkorb, A., Dyken, C., Hagen, T., Hjelmervik, J., Storaasli, O.: State-of-the-art in heterogeneous computing. Journal of Scientific Programming **18**(1) (2010) 1–33

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

[3] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top 500 supercomputer sites. `http://www.top500.org/` (November 2010)

[4] Hagen, T., Henriksen, M., Hjelmervik, J., Lie, K.A.: How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine. In Hasle, G., Lie, K.A., Quak, E., eds.: Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF. Springer Verlag (2007) 211–264

[5] Hagen, T., Lie, K.A., Natvig, J.: Solving the Euler equations on graphics processing units. In: Proceedings of the 6th International Conference on Computational Science. Volume 3994 of Lecture Notes in Computer Science., Berlin/Heidelberg, Springer Verlag (2006) 220–227

[6] Brandvik, T., Pullan, G.: Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. Journal of Mechanical Engineering Science **221**(12) (2007) 1745–1748

[7] Brandvik, T., Pullan, G.: Acceleration of a 3D Euler solver using commodity graphics hardware. In: Proceedings of the 46th AIAA Aerospace Sciences Meeting. Number 2008-607 (2008)

[8] Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.: Nodal discontinuous Galerkin methods on graphics processors. Journal of Computational Physics **228**(21) (2009) 7863–7882

[9] Wang, P., Abel, T., Kaehler, R.: Adaptive mesh fluid simulations on GPU. New Astronomy **15**(7) (2010) 581–589

[10] Antoniou, A., Karantasis, K., Polychronopoulos, E., Ekaterinaris, J.: Acceleration of a finite-difference weno scheme for large-scale simulations on many-core architectures. In: Proceedings of the 48th AIAA Aerospace Sciences Meeting. (2010)

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

[12] Liang, W.Y., Hsieh, T.J., Satria, M., Chang, Y.L., Fang, J.P., Chen, C.C., Han, C.C.: A GPU-based simulation of tsunami propagation and inundation. In: Algorithms and Architectures for Parallel Processing. Volume 5574 of Lecture Notes in Computer Science., Berlin/Heidelberg, Springer Verlag (2009) 593–603

[13] Lastra, M., Mantas, J., Ureña, C., Castro, M., García-Rodríguez, J.: Simulation of shallow-water systems using graphics processing units. Mathematics and Computers in Simulation **80**(3) (2009) 598–618

[14] de la Asunción, M., Mantas, J., Castro, M.: Simulation of one-layer shallow water systems on multicore and CUDA architectures. The Journal of Supercomputing (2010) 1–9 [published online].

[15] de la Asunción, M., Mantas, J., Castro, M.: Programming CUDA-based GPUs to simulate two-layer shallow water flows. In: Euro-Par 2010 - Parallel Processing. Volume 6272 of Lecture Notes in Computer Science., Berlin/Heidelberg, Springer Verlag (2010) 353–364

[16] Brodtkorb, A., Hagen, T.R., Lie, K.A., Natvig, J.R.: Simulation and visualization of the Saint-Venant system using GPUs. Computing and Visualization in Science (2010) [forthcoming].

[17] Micikevicius, P.: 3D finite difference computation on GPUs using CUDA. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, New York, NY, USA, ACM (2009) 79–84

[18] Playne, D., Hawick, K.: Asynchronous communication schemes for finite difference methods on multiple GPUs. Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on (May 2010) 763–768

[19] Komatitsch, D., Göddeke, D., Erlebacher, G., Michéa, D.: Modeling the propagation of elastic waves using spectral elements on a cluster of 192 GPUs. Computer Science - Research and Development **25** (2010) 75–82

[20] Acuña, M., Aoki, T.: Real-time tsunami simulation on multi-node GPU cluster. ACM/IEEE conference on Supercomputing (2009) [poster].

[21] Rostrup, S., De Sterck, H.: Parallel hyperbolic PDE simulation on clusters: Cell versus GPU. Computer Physics Communications **181**(12) (2010) 2164–2179

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

[23] Brodtkorb, A., Sætra, M., Altinakar, M.: Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. [preprint].

[24] Shu, C.W.: Total-variation-diminishing time discretizations. SIAM Journal of Scientific and Statistical Computing **9**(6) (1988) 1073–1084

[25] NVIDIA: NVIDIA CUDA reference manual 3.1 (2010)

[26] Ding, C., He, Y.: A ghost cell expansion method for reducing communications in solving PDE problems. In: ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society (2001) 50–50

[27] Palmer, B., Nieplocha, J.: Efficient algorithms for ghost cell updates on two classes of MPP architectures. In Akl, S., Gonzalez, T., eds.: Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA, IASTED, ACTA Press (November 2002) 192–197