

Asynchronous Communication Schemes for Finite Difference Methods on Multiple GPUs

D.P. Playne and K.A. Hawick

Computer Science

Massey University

Albany, North Shore 102-904, Auckland, New Zealand

{ d.p.playne, k.a.hawick } @massey.ac.nz

Abstract

Finite difference methods continue to provide an important and parallelisable approach to many numerical simulations problems. Iterative multigrid and multilevel algorithms can converge faster than ordinary finite difference methods but can be more difficult to parallelise. Data parallel paradigms tend to lend themselves particularly well to solving regular mesh PDEs whereby low latency communications and high compute to communications ratios can yield high levels of computational efficiency and raw performance. We report on some practical algorithmic and data layout approaches and on performance data on a range of Graphical Processing Units (GPUs) with CUDA. We focus on the use of multiple GPU devices with a single CPU host.

1. Introduction

Graphical Processing Units (GPUs) have given data-parallel computing a new lease of life through their highly-efficient tightly-coupled computational capabilities within a single compute device. While further improvement in chip fabrication techniques will undoubtedly lead to yet more cores on a single device it is already also quite feasible to consider application programs that make use of multiple devices. This architectural notion of a CPU serviced by a small to medium number of separate GPU or similar accelerator devices is a particularly attractive way of increasing the power of individual nodes of a cluster computer or supercomputer[1] using off-the shelf technological components.

In this paper we look at software and ideas for managing multiple GPUs from a single CPU host – that may itself have one or more likely several

conventional cores. We look at the asynchronicity issues and latency hiding potential for employing multiple host level threads within a single application, whereby each host thread runs a separate GPU device. We use and compare a pair of NVIDIA's GTX260 GPU device cards within a single host as well as their GTX295 dual-GPU combination card.

A classic application problem in computational science[2], [3] is that of regular-structured mesh of floating point field values upon which a partial differential equation (PDE) is solved iteratively using spatial stencils which provide discretised finite-difference operators.

Key issues that arise are how to arrange the problem data domain in the memory of the computational device(s) to minimise overall communications of cell halos or where to asynchronously overlap communications with computations[4]. Work has been reported elsewhere on these issues for a single host thread and a single GPU accelerator device[5]. In this present paper we focus on multiple device effects and the particular problems that arise from yet another hierarchical layer in memory access speed.

Halo issues in memory[6] and stencil-based problems[7] are not new and have been widely experimented upon for many different software architectures and problems in parallel computing. Many applications in science and engineering can be formulated in this way[3] and can therefore make use of cluster supercomputers with nodes that have been accelerated with multiple devices. Applications range from those that involve simple stencil problems[8] in financial modelling[9] and in fluid mechanics[10], through complex stencils coding involving non trivial data types such as vectors or complex numbers[11], [12], to those problems that couple multiple simple models together such as weather[13], environmental[14] and climate simulation codes[15].

We focus on regular-structured mesh with direct storage, but these ideas relate to other problems that can be formulated in terms of sparse matrices[16]. We illustrate our approach and results using code implemented in NVIDIA's Compute Unified Device Architecture (CUDA) – which is a C-based syntactic language highly suited to NVIDIA's Tesla GPU devices[17]. Our host code running on conventional CPU core(s) employs straight C code and the pThreads library for multi-core functionality[18].

This approach targets applications that can be decomposed into asynchronously coupled individual computational kernels that are themselves intrinsically tightly coupled data-parallel computations. The latency balance points are unusual however because of the memory decompositions possible between the CPU and main memory and the various sorts of memory available on the GPU device itself.

In Section 4 we summarise the GPU architecture and describe the application problem in Section 5. In Section 7 we present some performance results which are discussed in terms of the various asynchronous tradeoff possibilities and tradeoffs possible in Section 8 where we also offer some conclusions and areas for further work experimenting with asynchronous support devices.

2. Finite Difference Equations

Finite-difference methods for solving partial differential equations on regular structured meshes generally lend themselves well to implementation on GPUs with CUDA[8]. The general method involves approximating spatial calculus operators such as derivatives ($\frac{\partial}{\partial x}$, $\frac{\partial}{\partial y}$, ...) or the Laplacian ($\nabla^2 \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$) with explicit difference formulae discretised on the spatial (regular) mesh. So for example (in two dimensions) the Laplacian can be approximated by: $\frac{1}{\Delta^2} (u_{x-1,y} + u_{x+1,y} - 4u_{x,y} + u_{x,y-1} + u_{x,y+1})$, where Δ is the spatial distance between cells. This direct method extends to some quite complicated operators such as the biharmonic operator (∇^4), and other situations where operators can be composited or otherwise combined[12]. Equations of the general form: $\frac{\phi(\mathbf{r},t)}{dt} = \mathcal{F}(\phi, \mathbf{r}, t)$, where the time dependence is a first derivative and the spatial dependence in the right hand side is expressed in terms of an additive list of partial spatial derivatives, can be time-integrated using finite-difference methods. Although for explanatory purposes it is often easiest to use the first-order Euler time-integration method, in practice it is straightforward to apply other second-, fourth- or

higher -order methods such as Runge-Kutta[19], once the spatial operators can be coded.

3. Memory Halo

Field equations using finite-differencing solvers almost always access the values of neighbouring cells to determine how to change the value of a cell. The neighbours that the cell must access is the memory halo of the equation. Figure 1 shows some common memory halos.

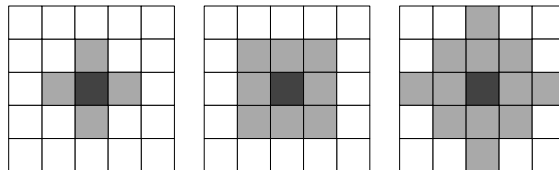


Figure 1. Some common memory halos. The grey cells denote cells that used in the calculation of the equation.

The size of an equation memory halo has an effect on how well the equation can be split across multiple GPUs. When a field is split, the cells on the edge of each sub-field must be communicated to the other GPU for each step of the simulation. An equation with a large memory halo will require more cells to be communicated each step than an equation with a small memory halo. Results showing the impact of the memory halo size on performance results are presented in Section 7.

4. GPU Parallelism

Graphical Processing Units or GPUs have been developed for rendering real-time computer graphics. To meet the increasing demands of games market, these GPUs have developed into highly-parallel many-core architectures. NVIDIA GPUs are based on the Tesla architecture which contain a scalable array of multi-processors[20]. These multi-processors each contain 8 scalar processors (SP).

The programming model of CUDA GPUs is Single Instruction Multiple Thread or SIMT, this paradigm is similar to the well known SIMD. All the SPs within the same multi-processor must execute the same instruction, however different multi-processors can execute different instructions. GPUs are capable of managing thousands of threads (known in CUDA as kernels) which are scheduled and executed on the multi-processors. As this thread management is performed in hardware, it presents little overhead.

The main performance factor that must be considered when designing a CUDA program is the memory access. As GPUs are designed to have a very high computational throughput, they do not have the high-levels of memory caching common to CPUs. Instead GPUs contain a number of specialised memory caches that must be explicitly used.

Global Memory is the main device memory that must be used for all GPU programs. This is the only memory that can be written to or read from by the host. This is also the slowest memory to access from the multi-processors. Access times to this type of memory can be improved, when 16 sequential threads access 16 sequential and aligned memory address, the transactions will be coalesced into a single transaction.

Shared Memory is shared between threads executing on the same multi-processor. Threads can read and write to this memory and share common information between threads.

Texture Memory Cache is not a separate memory type but rather a cached method of accessing global memory. This cache will automatically cache values spatially surrounding those accessed. This spatiality can be defined in 1-, 2- or 3-dimensions. This memory type is best used when neighbouring threads access memory addresses in the same area.

Constant Memory Cache is like texture memory in that it is also a cached method of accessing global memory. Instead of caching neighbouring values, constant memory cache allows all threads in a multi-processor to access the same memory location in a single transaction.

Proper use of memory and access patterns can greatly improve the performance of GPU programs. In the following section we describe how to implement a field-equation simulation across multiple GPU devices.

5. Field-Equation Simulation on GPUs

In previous work [5] we have described implementations for field-equations using finite-differencing on GPUs with CUDA. The field equation used as an example is the Cahn-Hilliard equation, we use the same example equation for this paper. The basic design of the best implementation is as follows. Each cell has one kernel allocated to it, this kernel will read the value of the cell and surrounding neighbours from memory, compute the new value of this cell according to the equation and write the result to an output array. Texture memory is used for reading values from memory as it automatically caches neighbouring values, this was found to significantly improve the performance of the simulation.

In this we aim to improve this simulation by spreading the computation across two GPUs. The implementation we describe is designed to execute on two GPUs, however the same techniques can be applied to systems with more GPUs. First we must discuss how the field is split between the GPUs.

6. Field Decomposition

To decompose a field-equation simulation across two GPUs we must separate the field between the two. The field is split into two halves and one half is loaded into memory on each GPU and with two sets of bordering cells from the other half. These borders are the cells from the other half of the field that are required by the cells on the edge of half. The width of these borders depend on the memory halo of the model.

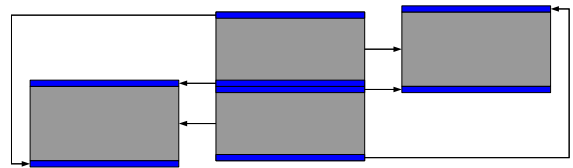


Figure 2. The main field (centre) being split into two separate field (left and right) with the appropriate border cells.

The field should be split in the highest-dimension, this means that the cells in the borders will be in sequential addresses. This allows the copy procedures to operate with maximum efficiency. If the field is split in the lower dimensions, multiple copy calls will be required to copy the data between the devices. These multiple copy calls will perform slower than one larger call.

6.1. Synchronous Memory Copy

This method is the simplest method of decomposing the field equation simulation across two GPUs. To interact with multiple GPUs, multiple CPU threads are required. For these implementations we use the pThreads library to manage the multi-threading on the CPU. For this implementation two threads are created to interact with one GPU each. For every step, each thread performs the following simple algorithm:

- 1) Compute the simulation for the cells in the field
- 2) Synchronise with the other thread
- 3) Copy bordering cells from GPU
- 4) Exchange bordering cells
- 5) Load new bordering cells into GPU

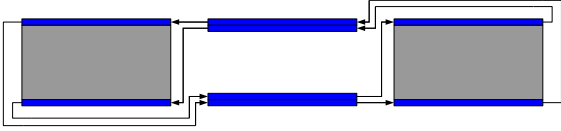


Figure 3. The communication of border cells, this process must be performed after each simulation time-step.

6) Repeat

This method is simple to implement and provides a way to quickly utilise multiple GPUs. However, the downside of this approach is that the each GPU must calculate the simulation for the field then stop and sit idle while the CPU threads exchange data. When the field length and memory halo are both small, this memory exchange time does not take long and is does not have a large impact on performance. However, as the field length and/or memory halo increases in size, this memory exchange has a larger impact on performance.

Any time the GPU sits idle is a waste of resources, to minimise this idle-time we make use of the asynchronous memory copy/execution capabilities of the GPU.

6.2. Asynchronous Memory Copy

This implementation uses asynchronous memory access to reduce idle GPU time. CUDA supports asynchronous host-device memory access and execution for GPUs with compute capability 1.1 or higher can split device execution and memory copies into separate streams. Execution in one stream can be performed at the same time as a memory copy from another. This can be extremely useful for reducing GPU idle time.

The basic idea behind this implementation is to use asynchronous copies to exchange the border cell values between the two threads while the GPU is still computing the simulation for the rest of the field. We decompose the field into two halves as described in Section 6. Each thread in the asynchronous memory copy implementation performs the following algorithm:

- 1) Compute the border cells (Stream 1)
- 2) Compute the other cells in the field (Stream 2)
- 3) Copy bordering cells from GPU to the host (Stream1)
- 4) Exchange bordering cells (CPU)
- 5) Load new bordering cells into GPU (Stream 1)
- 6) Repeat

There are some programatic challenges that must be overcome when implementing this design that are worth noting. For the GPU to copy data from the device memory to the host memory asynchronously, the host memory must be allocated by CUDA to ensure that it is page locked. This is normally performed using the CUDA function `cudaMallocHost(void **ptr, size_t size)`. However, memory declared using this function will only be accessible to the thread that declared it. This means that exchanging the bordering cells requires an extra CPU memory copy to copy the data between the page-locked memory for each thread. This extra memory copy is obviously undesired.

This problem can be overcome by allocating the memory using the function `cudaHostAlloc(void **ptr, size_t size, unsigned int flags)` with the flag `cudaHostAllocPortable`. This flag tells the compile to make the memory available to all CUDA contexts rather than only the one used to declare it. In this way both threads can use the memory to exchange border cells.

7. Results

To test the multi-GPU designs for field equation simulations, we have implemented a sample field equation using the two designs. We selected the Cahn-Hilliard equation because we have previously reported on its single-GPU performance and believe it to be a good representative for finite-differencing simulations. We have implemented this simulation using both multi-GPU designs and compared their performance to previously recorded single-GPU results. All implementations make use of the Euler integration method, this method is not used for actual simulations due to its poor accuracy, however for ease of description and performance comparison it is suitable.

The details of the platform the implementations have been tested on are as follows. The machine is running Ubuntu 9.10 64-bit with an 2.66Ghz Intel® Core™2 Quad CPU and 8GB of DDR2-800 memory. The GPUs used for the simulations is one NVIDIA® GeForce® GTX 295 which contains two GPUs each access to 896MB of memory. We also executed the simulations on a machine containing two GTX 260+ overlocked graphics cards to determine if there was any performance difference when using separate graphics cards (with one PCI-E 16 bus each). It was found that both configurations produced very similar results with no major difference in performance.

The performance of both implementations are compared to a single-GPU implementation across a range of field-lengths and memory halos. Equations with larger memory halos must access more values and are generally slower which distorts the performance comparison. Therefore the same equation is used but the number of bordering cells copied each iteration is increased (as appropriate to the stated memory halo). This shows a comparison of the performance of the implementation at providing the necessary data rather than a comparison between the performance of equations with varying sizes of memory halo.

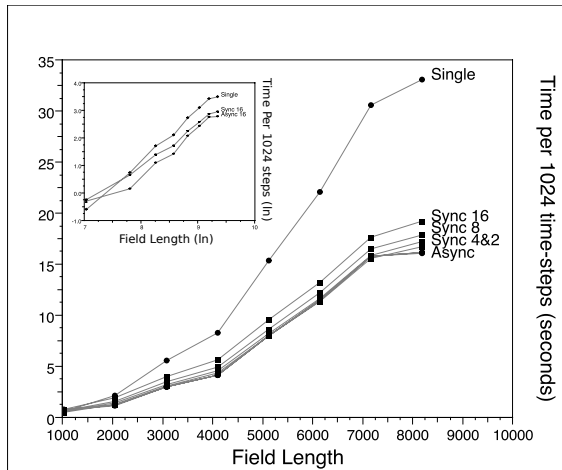


Figure 4. Performance results showing the seconds per 1024 simulation time-steps for the two multi-GPU implementations, results are shown for field lengths of 1024 to 8192 with memory halo sizes 2, 4, 8 and 16. Note that the Asynchronous implementations have almost indistinguishable timings for all memory halos and are simply labeled as ‘Async’.

The results shown in Figure 4 show some interesting and unexpected effects. In most cases the Asynchronous memory copy implementation provides the best performance, the performance improvement generally increases as the field length and memory-halo increase in size. This can be seen more clearly in Figure 5.

The most unexpected effect seen in these results is the loss in performance seen when the field lengths are not powers of two. This effect can be seen in Figure 4 where the drop in performance appears as kinks in the plot. This effect is most apparent in the Asynchronous memory copy implementation which shows in Figure 5 where the performance increase is noticeably lower for field lengths that are not a power of two.

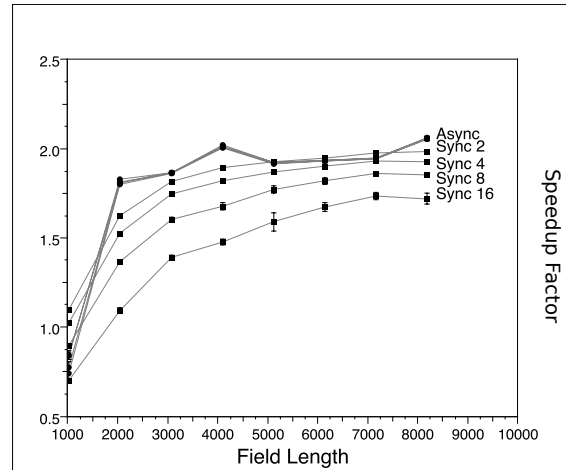


Figure 5. The speedup factors both implementations provide over the single-GPU implementation.

8. Discussion and Conclusions

We have presented two methods for implementing finite-differencing field-equation simulations on multiple GPUs. The implementations we have discussed and presented performance data on operate on a single machine containing two GPU devices. The results presented show that the correct use of asynchronous memory communication can provide almost linear speed up over a single-GPU implementation for larger system sizes. Synchronised memory copies can still be useful in some cases for providing a simple method for achieving performance improvements.

While GPGPU has opened the door to cheap and powerful data-parallel architectures, they do not have the same scalability of Cluster machines. This scalability can be achieved by extending from the described dual-GPU system to a many-GPU cluster. In such a cluster each node is in fact a highly parallel machine containing one or more GPU devices. The system described in this work performs communication between devices using CPU memory, thus the communication latency is relatively low. In a GPU accelerated cluster, proper use of Asynchronous communication becomes even more vital to fully utilise computing resources.

We intend to continue this work by examining approaches for decomposing such simulations across clusters with non-homogenous, GPU-accelerated nodes along with possible methods for pipelining communications for such systems to improve compute/communication ratios.

References

- [1] R. Rabenseifner and G. Wellein, *Modeling, Simulation and Optimization of Complex Processes*. Springer, 2005, ch. Comparison of Parallel Programming Models on Clusters of SMP Nodes, pp. 409–425.
- [2] K. A. Hawick, E. A. Bogucz, A. T. Degani, G. C. Fox, and G. Robinson, “Computational fluid dynamics algorithms in high-performance fortran,” in *Proc. AIAA 25th Computational Fluid Dynamics Conf*, June 1995.
- [3] L. C. McInnes, B. A. Allan, R. Armstrong, S. J. Benson, D. E. Bernholdt, T. L. Dahlgren, L. F. Diachin, M. Krishnan, J. A. Kohl, J. W. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, and S. Zhou, *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2006, ch. Parallel PDE-Based Simulations Using the Common Component Architecture, pp. 327–381.
- [4] M. E. Hayder, C. S. Ierotheou, and D. E. Keyes, *Progress in Computer Research*. Nova Science, 2001, no. ISBN:1-59033-011-0, ch. Three parallel programming paradigms: comparisons on an archetypal PDE computation, pp. 17–38.
- [5] A. Leist, D. Playne, and K. Hawick, “Exploiting Graphical Processing Units for Data-Parallel Scientific Applications,” *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009, cSTN-065.
- [6] M. Grote and H. D. Simon, “Parallel preconditioning and approximate inverses on the connection machine,” in *Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing*, R. Sincovec, Ed., vol. 2. SIAM, July 1993, pp. 519–523.
- [7] R. F. Barrett, P. C. Roth, and S. W. Poole, “Finite difference stencils implemented using chapel,” Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122, 2007.
- [8] P. Micikevicius, “3D finite difference computation on GPUs using CUDA,” in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, no. ISBN:978-1-60558-517-8, 2009.
- [9] F. O. Bunnin, Y. Guo, Y. Ren, and J. Darlington, “Design of high performance financial modelling environment,” *Parallel Computing*, vol. 26, pp. 601–622, 2000.
- [10] B. A. Tanyi and R. W. Thatcher, “Iterative Solution of the Incompressible Navier-Stokes Equations on the Meiko Computing Surface,” *Int. Journal for Numerical Methods in Fluids*, vol. 22, pp. 225–240, 1996.
- [11] K. A. Hawick and D. P. Playne, “Modelling, Simulating and Visualizing the Cahn-Hilliard-Cook Field Equation,” *International Journal of Computer Aided Engineering and Technology (IJCAET)*, vol. 2, no. CSTN-075, pp. 78–93, 2010.
- [12] K. Hawick and D. Playne, “Automated and parallel code generation for finite-differencing stencils with arbitrary data types,” Computer Science, Massey University, Tech. Rep. CSTN-106, December 2009, submitted to ICCS, Workshop on Automated Program Generation for Computational Science, Amsterdam 2010.
- [13] V. T. Vu, G. Cats, and L. Wolters, “Asynchronous Communication in the HIRLAM Weather Forecast Model,” in *Proc. 14th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*, 2008.
- [14] M. Ashworth, F. Foelkel, V. Gulzow, K. Kleese, D. Eppel, H. Kapitza, and S. Unger, “Parallelisation of the GESIMA Mesoscale Atmospheric Model,” *Parallel Computing*, vol. 23, pp. 2201–2213, 1997.
- [15] M. Mineter and N. Hulton, “Parallel processing for finite-difference modelling of ice-sheets,” *Computers & Geosciences*, vol. 27, pp. 829–838, 2001.
- [16] A. Filippone and M. Colajanni, “PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices,” *ACM Trans. Mathematical Software*, vol. 26, no. 4, pp. 527–550, December 2000.
- [17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March-April 2008.
- [18] IEEE, *IEEE Std. 1003.1c-1995 thread extensions*, 1995.
- [19] L. F. Shampine, “Some practical Runge-Kutta Formulas,” *Mathematics of Computation*, vol. 46, no. 173, pp. 135–150, January 1986, ISSN: 0025-5718.
- [20] NVIDIA CUDA™ Programming Guide Version 2.3, NVIDIA® Corporation, 2009, last accessed August 2009. [Online]. Available: <http://www.nvidia.com/>