Cell-based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units

David Nicholaeff[1]*, Neal Davis[2], Dennis Trujillo[3], Robert Robey[1]*

[1]*XCP-2 Eulerian Codes, Los Alamos National Laboratory, Los Alamos, NM.*
[2]*Dept. of Nuclear, Plasma, & Radiological Engineering, University of Illinois at Urbana–Champaign, Urbana, IL.*
[3]*Dept. of Physics, New Mexico State University, Las Cruces, NM.*

SUMMARY

Presented in this paper is an OpenCL implementation of a cell-based adaptive mesh refinement (AMR) scheme modeling the shallow water equations using general-purpose graphics processing units (GPGPUs). The challenges associated with ensuring locality of computation in order to fully exploit the throughput and massive data parallelism of the GPU are discussed along with solutions. In particular, data ordering is taken as a free parameter. A stencil-based space-filling curve method allows for optimal load-balancing, while the resulting nontrivial arrangement of cells is addressed by a Cartesian-indexed hash mapping which allows for efficient parallel neighbor accesses. This in turn presents a homogeneous interface to the data across the multiple levels present in heterogeneous architectures.
The relative speed-up of the GPU-enabled AMR code is compared to the respective serial implementation, both with and without the message-passing interface (MPI), providing evidence for the need to design and implement numerical methods on heterogeneous architectures exploiting GPGPUs.

KEY WORDS: Cell-based Adaptive Mesh Refinement; AMR; GPGPU; GPU; OpenCL; heterogeneous architecture; exascale computing; parallel physics simulations

1. INTRODUCTION

We present an OpenCL implementation of a cell-based adaptive mesh refinement (AMR) scheme modeling the shallow water equations with the intent of exploring the dynamics of AMR across heterogeneous architectures. In particular, our AMR framework is extensible to a variety of governing equations. We model a cylindrically symmetric full dam break in the shallow water equations merely to act as a representative test case for several of the advantageous properties of cell-based adaptive meshes, such as clean symmetry preservation.

We focus on and discuss algorithm architecture, highlighting the importance of locality, as optimal partitioning of the computational domain to ensure maximized locality of data allows better exploitation of the massive number of parallel threads on the general-purpose graphics processing unit (GPGPU). The result is a proof of concept that a cell-based AMR code can be effectively implemented in the memory and threading model provided by OpenCL. The program requires dynamic memory in order to properly implement the mesh, which while unsupported in the OpenCL 1.1 standard [18], is effectuated by a combination of CPU memory management and GPU computation. Load-balancing is achieved through a stencil-based space-filling curve, eliminating the need for a complete recalculation of the indexing on the mesh. And a Cartesian-indexed hash

*Correspondence to: David Nicholaeff: dnic@lanl.gov, Robert Robey: brobey@lanl.gov

Contract/grant sponsor: Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

mapping scheme to allow fast parallel neighbor accesses at $O(n)$ is discussed, superseding the use of a k-D tree at $O(n \log n)$ (which itself supersedes $O(n^2)$ algorithms). Griebel et al. [13] explored a similar synthesis of hashing with space-filling curves, albeit in a multigrid setting. The relative speed-up of the GPU-enabled AMR code over the original CPU-only serial version, implemented both with and without the message-passing interface (MPI), shows an order of magnitude speed-up.

The problems of domain decomposition and load-balancing on GPGPU clusters appeared in the literature early in the stages of GPGPU computing and remains an active area of research. See for example Fan et al. [9] and Göddeke et al. [11]. Dynamic data structures in GPGPU computing also has recently become a very active area of research. See Merrill et al. [23] and the references therein. Furthermore, AMR has been explored on GPUs previously. Schive et al. [31] developed a GPU accelerated AMR code for an astrophysical simulation which uses a hierarchy of grid patches with an octree data structure. However, the aforementioned works as well as the general research direction of AMR on GPGPU clusters, to the best of the authors' knowledge, have all focused on structures similar or identical to those found in multigrid solvers; octrees are common data structures organizing computational cells, and the AMR uses block-based or patch-based adaptive meshes. In contrast, our body of work focuses on cell-based AMR, an inherently thread-centric approach as each cell is treated independently. Additionally, as a cell is the basic data element, the hierarchy imposed by partitioning the cells across the heterogeneous architecture has a homogeneous, uniform interface as each level in the hierarchy is simply a collection of cells of varying quantity.

The results of our cell-based AMR scheme, aptly named CLAMR, provide evidence that parallelization using the GPU delivers significant speed-up for typical numerical simulations and is feasible for scientific applications in the next generation of supercomputing. In particular, to reach the exascale computing paradigm by 2018, a hundredfold reduction in power consumption per operation is required. Today's petascale systems consume roughly seven megawatts of power, while the US Department of Energy's goal for an upper limit on exascale systems will be around twenty megawatts of power [35]. Unfortunately, scaling current technologies will not realize this goal due to physical limitations – ranging from voltage scaling issues and defect sizes to quantum mechanical effects.[†] However, the primary power demand of current architectures arises from the memory hierarchy and associated memory transfers of the compute nodes. GPU-based computing hides intranode memory latency with increased throughput and hence provides one of the most encouraging paths to exascale computing [32].

The techniques implemented in CLAMR highlight the need for a paradigm shift in programming methodologies that are mindful of parallelism, as analyzed in Davis et al. [7]; locality is one such important consideration, and its proper application demands careful reflection on the problem decomposition.

The paper is partitioned into five sections. Following the introduction, Section 2 analyzes the motivation for the architecture decisions behind CLAMR with a review of the adaptive mesh scheme, both logistically and heuristically, as well as a discussion of heterogeneous platforms. The implementation is presented in Section 3. This consists of an overview of the numerical method, an overview of the physical model for the shallow water equations, and an analysis of the major challenges of implementation for the adaptive mesh on the GPU – including the difficulties of ensuring proper partitioning of work load and of ensuring locality of memory accesses arising from neighbor searches. Section 4 presents our results. The timings between explicit CPU use and a hybrid implementation of the CPU and GPU are compared, both with a single compute node and with the use of MPI. Ultimately our results suggest that while future numerical physics codes need heterogeneous architectures to see significant speed-up, they cannot do so without devising better algorithms in lieu of ever greater computational power.

[†]The Bohr radius is roughly 10^{-11}m and current processor fabrication processes are roughly at 10^{-8}m. Hence, the next generation of chips will be a factor of 100 larger than the characteristic size of the atom, implying that quantum effects will start to play a role in circuit design. For more fun facts about the nanoscale, see http://www.nano.gov/nanotech-101/special.

2. CELL-BASED ADAPTIVE MESH REFINEMENT ON HETEROGENEOUS PLATFORMS

2.1. The Cell-based AMR Scheme

Numerical models are heavily influenced by their choice of discretization; for example, symmetry preservation is one important feature which can be adversely effected by the structure of the mesh. Hence, a careful selection of the mesh type is critical to properly approximate a continuous space and produce physically legitimate results. We implement a cell-based mesh, a discretization of space into a grid of square cells. The continuous functions of interest, such as mass and momentum, are all taken to be at the center of a cell, which is to say that in the discretization the state variables are averaged over the cell and assigned a Cartesian coordinate existing at the cell center. When the mesh is described as adaptive, that is to say that cells across the mesh can have variable levels of refinement (size) predicated by certain rules.

The motivation for an adaptive mesh is two-fold. First, regions of physical interest are superimposed onto an area of the mesh where the refinement is higher, thus allowing for higher precision. Further, this allows different physical scales to be simultaneously analyzed (e.g. in wave phenomenon long wavelength regions can be placed on coarser cells whereas high frequency waves require greater resolution to discern individual wave peaks)[‡]. The second advantage, which directly returns to the motivation to reduce power consumption, is memory frugality. Simply put, the physical model will require far less memory as it will use far less cells for the discretization.

To put the cell-based scheme into perspective, we discuss the types of AMR. The most common AMR in the literature is a structured AMR which superimposes blocks or patches of cells with a finer regular structure over regions of greater physical interest. The technique is described in a series of articles by Berger-Colella-Oliger ([3] and [2]). Perhaps the greatest advantage of this method is that the regular grids in refined regions are just treated like another mesh, making some parts of the implementation much the same as the regular grid. However, it induces additional refinement on cells which could have been left unrefined, thereby not efficiently fulfilling a primary goal of AMR: memory frugality.

The cell-based scheme, on the other hand, refines individual cells, thereby maximizing memory efficiency. As a further consequence, it precisely refines regions near important physical processes such as shocks or steep wavefronts. And lastly, it has less mesh refinement imprinting for curved or spherical shocks where the regular refined grid in structured AMR can impact the spherical symmetry of a problem.

On the implementation side of the cell-based AMR scheme, there are several rules dictating the adaptive refinement of the mesh. First, two neighboring cells must have no more than one level of refinement difference. This stems from both ease of implementation and reducing the small error that occurs at refinement steps (the frequency content of the simulated waves will be reduced in half, causing a minor reflection of the wave at the interface). Second, a cell is refined symmetrically, which is to say that it is bisected along all axes. Third, regions of physical interest are refined – this equates to steep gradients in both pressure and material interfaces. Fourth, refinement leads the event, which is to say that refinement should precede before the regions of physical interest arrive. Fifth, indexing is done using a standard Cartesian grid.

2.2. Considerations of Heterogeneous Platforms

From a data structures and algorithms perspective, cell-based adaptive mesh refinement for supercomputing applications requires software development methods which consider the dynamics of heterogeneous platforms. For both an overview and references to multiple sources of this argument, see Davis et al. [7] The model platform presented here is a simple configuration where a single CPU communicates with a single GPU, thus defining a single node. These nodes then

[‡]More precisely, we're mentioning different spatial scales. For multi-physics models requiring variable timesteps across the mesh, cell-based AMR can be used, but new challenges arise to match fluxed quantities across cell boundaries. Quirk [28] mentions specifically the issues of spurious reflections, but also provides an important discussion of grid efficiency and vorticity generation in adaptive meshes.

communicate using MPI. As the numerical calculations are performed on the GPU, we consider this primarily a GPGPU platform. The dynamics of such a platform dictate new considerations in algorithm architecture, which ultimately motivate the design behind the architecture of CLAMR. Note that real architectures will likely have multi-core nodes and may have more than one GPU. The nuances created by these variations to the primary design configuration are too varied to completely consider in this effort.

2.2.1. Intranode Implementation GPU computing, as currently implemented, relies on massive data parallelism to realize speed-up in code run and compute times. Implementation is not without its challenges, as the programmer is restricted to problem domains which only allow for the data element to be treated in isolation or with minimal coupling to other data elements[§]. In particular, GPUs have emphasized high bandwidth local memory of limited size but at the cost of data transfers both across the PCI bus and from global GPU memory to fast local memory. Additionally, many of the tools and optimizations available to the CPU such as $O(n^2)$ algorithms optimized at $O(n \log n)$ [19] are not available on the GPU[¶].

Nevertheless, due to the extremely parallel nature of the GPU, high memory bandwidth, and tremendous computational abilities, tasks such as numerically intensive calculations can be executed in a significantly reduced period of time as compared to the same calculation performed on the CPU.

This speed-up is due to the large number of threads brought to bear by the GPU along with the essentially zero context switching time between the thread groups. However, given the evolving nature of an adaptive mesh, dynamical data structures are required which consequently cannot be solely handled by the GPU. More correctly, memory cannot be dynamically allocated on the GPU as of OpenCL specification 1.1 [18]. But this is easily solved; memory management on the CPU and numerical calculation on the GPU are combined to form a heterogeneous computing environment. It is here where the necessity of locality becomes apparent: there is a 20 to $40\times$ factor increase in clock cycles due to memory latency when comparing reads from global memory to reads from local memory on the GPU [26, Ch. 3], with an even larger factor slowdown resulting from writes back to the CPU [25, Ch. 3]. It is perhaps best to view the local memory on the GPU as a programmable cache where the speed-up is highly dependent on the effectiveness of the programmer's reuse of data while it is in the local memory.

2.2.2. Internode Implementation For our single node computation, all calculations, including physics calculations, global reductions, neighbor calculations, and cell refinements, were successfully moved over to the GPU, with the CPU merely acting as a mechanism to reallocate memory. Namely, the state variables of the cells are resident on the GPU. As we move to MPI, however, our simple one node platform of one CPU core communicating with one GPU device needs to be expanded. The memory on the GPU must be retrieved by the CPU to communicate among nodes. Future enhancements to OpenCL will likely allow device buffers to be sent directly by MPI, but the complexity of fully implementing this technique will be challenging and require dynamic memory lists on the GPU.

The move to MPI sees an increase in the number of nodes, with each node still defined as a single CPU core controlling a single GPU device. We prefer a single core since our algorithm relies on the full use of the GPU for the numerical methods and additional CPU cores are a power drain that don't contribute much to our performance. The GPU is so much faster than the CPU that it makes more sense to do all possible computation on the GPU. Still, most future CPU architectures will

[§]This is not to say that problems of irregular and data-dependent parallelism have not been addressed. A review of some of the works at the GPU Tech Conference, `http://www.gputechconf.com/gtcnew/on-demand-gtc.php`, such as S0600, S0314, and S0042, show how graph representations and efficient implementations of scan/reduction operations can be utilized to see impressive performance. The hash-based algorithms developed by two of the current authors in Robey et al. [29] exploits a customized scan operation for speed-up. However, to get the best performance from the GPU, a high level of data independence is imperative.

[¶]This is in reference to many of the tree-based algorithms which attain the $O(n \log n)$ optimizations. Many tree-based algorithms are being ported to the GPU, but code complexity begs the question is the product worth the effort when a careful consideration of the problem decomposition might reveal underlying structure of a more independent nature.

likely be multicore. While the additional CPU cores give us no advantage for a single node, since all the data is resident on the GPU and very little is done host side on the CPU, data will need to be pulled off the GPU for MPI communication and writing out to files. The additional CPU cores can then be used to speed-up the operations and provide for another level of data locality with some benefits of better cache use. To take advantage of this additional level of data hierarchy, we would assign a rank to each CPU core. This implies that multiple CPU cores control a single GPU. Hence, each CPU core has its own command queue which will be serially passed down to the GPU. We ran this test on a single node with an AMD Opteron 6168; 48 CPU cores, each with their own MPI rank, drove a single GPU or two GPUs and no problems were encountered with the small test runs.

There are alternatives to this compute model. One would be to add an OpenMP or thread layer and have a single MPI rank for each node and use the CPU threading to gain access to the additional CPUs on the node. While technically viable, the essentially three different levels of parallel coding adds more complexity than we wanted to tackle. Another approach would be to run OpenCL kernels on the multicore level to access the additional CPUs. This option might be attractive in the future, but currently the GPU is so much faster than the CPU that it is hard to find work that would be reasonable to put on the multicore CPUs. The most attractive alternative at this time is to have one MPI rank per GPU and if there are additional CPU cores, just not use them in the computation. There is some waste in this compute model, but most large production systems will have a small number of CPU cores per node to conserve power. The last alternative would be to use CUDA on the GPUs instead of OpenCL. We have chosen OpenCL to allow a wider range of system architectures with some loss of additional CUDA functionality that could give higher performance in the near-term. Future hardware developments along the lines of the Intel MIC and AMD Fusion approaches may require a reassessment of our compute model, but it is clear that no single compute model will be able to run efficiently on every system architecture in the near future as hardware designs proliferate.

Our considerations in decomposing the mesh are focused on a viable scheme for a two-level partition. With the addition of the MPI internode layer we need to maximize locality in order to reduce data transfers across compute elements. This is the crux of CLAMR's design. Moving to MPI challenges the versatility of our choice of partitioning. A standard method for partitioning in MPI, using a recursive bisection, then combined with a standard lexicographic data order aligned with cache on individual nodes, vastly complicates code complexity in load-balancing and MPI communication in general. Maintaining a data order to be used across all levels of the hierarchy, however, simplifies implementation while simultaneously giving the algorithm designer freedom to choose a data order which is most beneficial to the problem at hand. By using space-filling curves, global arrays can be partitioned across MPI compute elements by simply dividing the arrays by a fixed stride, ensuring balanced load balance; within the same data order, each MPI compute node can repeat this process of dividing the arrays by a fixed stride in order to best achieve load balance across the workgroups on the GPU.

3. ARCHITECTURE & ALGORITHMS

The examination of heterogeneous platforms, discussed in Section 2, led to several key architecture decisions for CLAMR. Our implementation is designed to efficiently create a dynamic memory space by using the CPU to manage memory transfers while the GPU performs the operations on the cells; memory transfers are reduced to maximize the time for which the control flow stays with the GPU. Calculating global reductions on the GPU is one technique used to accomplish this feat.

This section progresses as follows. First an overview of the control flow of CLAMR is given. Second, the physical model and numerical method are analyzed, pointing out the issues involved in finite differencing across adaptive meshes. In the last three subsections, partitioning and locality, neighbor searching, and enhanced-precision sums, we look at the tools required to put cell-based AMR on the GPU. Namely, data ordering is treated as a free parameter, allowing a surface area to volume ratio minimization and thus a minimization in the amount of data needed to be transferred.

As a consequence, neighbor searching becomes nontrivial and global calculations vary, hence the necessity of clever neighbor searching methods and consistency checking enhanced-precision sums.

CLAMR is available through a BSD license from `github.com/losalamos/CLAMR`.

3.1. *CLAMR: Control Flow*

The initialization of CLAMR begins by creating global objects. The mesh is built with each cell's initial state variables set to the problem specification. This is followed by a preliminary global space partitioning accompanied by a calculation of cell neighbors. Next, the compute context is established, which includes a command queue containing the commands to be sent to the compute device. In particular, all kernel objects are declared. (For a review of OpenCL terminology, see the OpenCL Specification [18].) As part of establishing the compute context, memory is allocated on the GPU. Then the state variables for each cell are written to the GPU's global memory space, and all kernel arguments are set.

When setup completes, control flow is transferred to the GPU as the size of the timestep for the numerical scheme is computed. This is calculated based on the wave speed which requires a global reduction as the maximum wave speed needs to be established. Then, in preparation for execution of the workgroups, the local tile (workgroup memory space variables) is set. In addition, the conditions on the outer boundary of the mesh are created as needed.

At this point, the state variables are updated as determined by the governing equations and numerical scheme. Section 3.2 provides the details. We note here that calculations are done in double precision. Following the update, the gradients are used, depending on the magnitude and sign change across cells, to refine or coarsen the cells of the mesh. A device-global reduction is done to indicate the new number of cells, both globally and on each tile.

With the new number of cells in the mesh determined, memory on the GPU needs to be reallocated. Control flow is returned back to the CPU; the new variable arrays are first resized before the rezone call and the old arrays are resized afterwards in a technique reminiscent of the double-buffering commonly done in graphics applications. Control flow is once again returned to the GPU as a rezoning of the cells is done; cells are refined, coarsened, or unchanged as necessary. The space-filling stencil is applied (which is highly parallel, namely a global space-filling curve call is no longer needed), and the new cell neighbors are set.

Finally, CLAMR proceeds to the next iteration while not at the maximum simulation time.

3.2. *The Physics Model & The Numerical Method*

Currently, CLAMR is implemented with the shallow water wave equations because of their relative simplicity as well as the high degree of symmetry present in our particular problem setup: a cylindrical shock impacts the center of the mesh at the first timestep, i.e. we are running a full circular dam break problem. In their conservative form, the equations are:

$$\begin{aligned}
\frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} &= 0 \quad \text{(Conservation of mass)} \\
\frac{\partial (hu)}{\partial t} + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) &= 0 \quad \text{(Conservation of } x\text{-momentum)} \\
\frac{\partial (hv)}{\partial t} + \frac{\partial}{\partial x}(hvu) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) &= 0 \quad \text{(Conservation of } y\text{-momentum)}
\end{aligned}$$

where h is the height of a column of water, g is the acceleration due to gravity, and u and v are the wave velocities in the x and y directions, respectively. More precisely, they are the velocities of the water molecules. The speed of propagation, the phase velocity, is $\sqrt{gh}$. For a nice discussion which mentions this, see Tao [34]. Note that mass equals height (times a constant) because water is incompressible. In particular, the incompressibility causes all pressures to only vary the height of the water while the width and length of a differential column remain constant, and the density of water remains constant. Also note that the pressure term is $gh^2/2$. For a more rigorous presentation, see the sections in Landau & Lifshitz [20] on long gravity waves and shallow-water theory. If the reader is particularly interested in the shallow water equations on the GPU, we suggest Castro et al. [5] and Brodtkorb et al. [4].

For the discretization, we use a total variation diminishing (TVD) finite difference scheme based on a two-step Lax-Wendroff method [21] in conjunction with a minmod symmetric flux limiter to provide an upwind weighted artificial viscosity term. The Lax-Wendroff method is second-order accurate in space and time, hence it is a suitable choice for smooth regions. Around steep shocks, oscillations produced by the second-order method necessitate damping by switching to a first-order upwind method, which is accomplished by the flux limiter. For a solid foundation on the numerical methods mentioned, as well as modeling the shallow water equations in general, see LeVeque's book on finite difference methods [22] and George's Master's thesis [10]. For more information about the complete numerical method as implemented, see Davis [8], Sweby [33], and Yee [36].

While the references above provide the background for the numerical method used in CLAMR, the adaptive mesh complicates the equations. The half-timestep equations are:

$$\begin{aligned} U^{n+1/2}_{i+1/2,\,j} &= \frac{r_i U^n_{i+1,\,j} + r_{i+1} U^n_{i,\,j}}{r_{i+1} + r_i} - \Delta t \left(\frac{F^n_{i+1,\,j} A_{i+1} a_{i+1} - F^n_{i,\,j} A_i a_i}{V_{i+1} v_{i+1} + V_i v_i} \right) \\ U^{n+1/2}_{i,\,j+1/2} &= \frac{r_j U^n_{i,\,j+1} + r_{j+1} U^n_{i,\,j}}{r_{j+1} + r_j} - \Delta t \left(\frac{G^n_{i,\,j+1} A_{j+1} a_{j+1} - G^n_{i,\,j} A_j a_j}{V_{j+1} v_{j+1} + V_j v_j} \right). \end{aligned}$$

Following a standard notation, subscripts i and j are spatial coordinates, while the superscript n is a temporal coordinate. Here U represents a general state variable, which for the shallow water equations are the mass, the x-momentum, and the y-momentum. F and G are the x and y flux terms for the state variable U, respectively.

The adaptation from the regular grid equations,

$$\begin{aligned} U^{n+1/2}_{i+1/2,\,j} &= \frac{U^n_{i+1,\,j} + U^n_{i,\,j}}{2} - \frac{\Delta t}{2\Delta x} \left(F^n_{i+1,\,j} - F^n_{i,\,j} \right) \\ U^{n+1/2}_{i,\,j+1/2} &= \frac{U^n_{i,\,j+1} + U^n_{i,\,j}}{2} - \frac{\Delta t}{2\Delta y} \left(G^n_{i,\,j+1} - G^n_{i,\,j} \right), \end{aligned}$$

arises from the inclusion of spatial scaling variables which are necessary to account for the adaptive mesh. In the adaptive mesh half-timestep equations, r, A, and V are radius, area, and volume, respectively, along with the scaling variables a and v for the area and volume, respectively. The first expression with the r terms is a linear interpolation to find the state variable at the face between cells (the large $\times$ between cells 5 and 7 in Figure 1). In the regular grid equations, this interpolation is merely the average of the two neighboring state variables. The second expression which includes the timestep is the flux term; it weights the fluxes F and G by the area of the cell interface multiplied by a scale factor, which to allow for arbitrary dimension is taken in the form

$$a_i = \min\left(1,\ \frac{a_{i+1}}{a_i}\right), \qquad a_{i+1} = \min\left(1,\ \frac{a_i}{a_{i+1}}\right).$$

This total flux, expressed in the numerator of the second term of the adaptive mesh half-timestep equations, is then divided over the volume of the staggered compute cell (the colored region in Figure 1), which appears in the denominator. In order to allow for arbitrary dimension, the volume contributions by neighboring cells are multiplied by the scale factors

$$v_i = \min\left(\frac{1}{2},\ \frac{v_{i+1}}{v_i}\right), \qquad v_{i+1} = \min\left(\frac{1}{2},\ \frac{v_i}{v_{i+1}}\right).$$

In the regular grid equations, no scaling is necessary, and consequently A/V is simply $1/\Delta x$ or $1/\Delta y$.

Efficient use of the GPU requires that the equations be consolidated into as few as possible so that all the cells/threads execute the same code block. The same equation expressed in multiple conditional blocks will suffer performance degradation due to all threads in a workgroup having

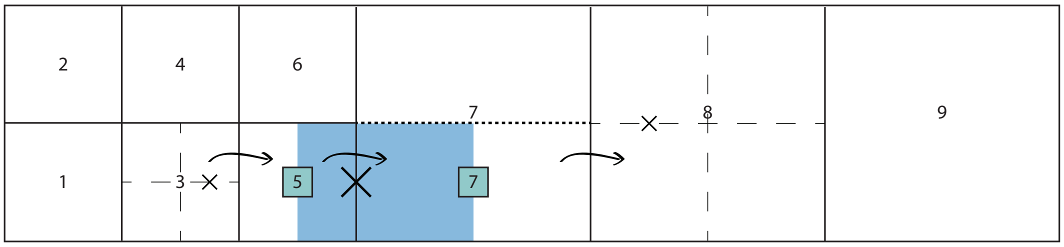

Figure 1. The finite difference stencil shows the half-timestep calculation, as well as the comparison of gradients for the flux correction to remove oscillations (applied at the full-timestep). The value of a state variable is linearly interpolated at the face (linear interpolations are shown with an $\times$), and the fluxes for the staggered compute cell are computed from the stored values (green boxes). Notice that the value of cell 7 is taken along a half cell with the same characteristic width as cell 5. That is, the neighbor of greater refinement sets the characteristic scale. When flux limiting is applied, a five-point stencil is used to find sequential gradients (shown by the arrows).

to execute each block. Namely, breaking an equation into two similar blocks controlled by a conditional, such as if the cells differ in refinement level, will effectively double the run-time since the threads execute both blocks. This characteristic of GPU performance has been termed "lock-step". Hence consolidating the equations into one with additional factors to scale or turn on/off terms is crucial for the GPU's performance, as these extra factors are calculated at the start of the kernel in short conditional blocks thereby minimizing the performance reduction. See Coutinho et al. for a nice discussion of divergence and lock-step on the GPU [6].

The full-timestep calculation is:

$$U^{n+1}_{i,\,j} = U^{n}_{i,\,j} - \Delta t \left(\frac{\overline{F}^{n+1/2}_{i+1/2,\,j} - \overline{F}^{n+1/2}_{i-1/2,\,j}}{\Delta x} + \frac{\overline{G}^{n+1/2}_{i,\,j+1/2} - \overline{G}^{n+1/2}_{i,\,j-1/2}}{\Delta y} \right).$$

The important point to note here is that the calculation from the perspective of a coarse cell with refined neighbors requires averaged fluxes, denoted by the F and G terms with bar overhead. This averaging, however, is complicated by the flux terms for one state variable not being of the same form as another. For example, the mass flux term in the x direction is hu, while the corresponding term for the x-momentum is $hu^2 + \frac{1}{2}gh^2$. Looking again at Figure 1, we see that the flux across cell 7's left face is the sum of the fluxes across its interfaces between cell 5 and cell 6. Numerically the most important consideration is conserving the state variables, and this equates to computing the fluxes identically from the perspectives of cells 5, 6, and 7. The major ramification is the necessity of averaging the fluxes at the full-timestep, as opposed to averaging at the half-timestep.

Finally, to correct for oscillations near shocks, a minmod symmetric flux limiter is used to impose the TVD property. This requires a five-point stencil in order to take state information from the neighboring cells' neighbors, thereby allowing the ratios of the gradients to be examined. This of course is in contrast to the above equations for the Lax-Wendroff method, which are compact in the sense that they are only using a cell's nearest neighbors' state information. Then, once the gradients are compared, and if there is a sign change, the flux limiter term is applied. The corrections are:

$$\begin{aligned} U^{n+1}_{i,\,j} &\pm \frac{\nu(1-\nu)}{2}\big[1 - \phi(r^+,\, r^-)\big]\Delta U^n \\ \phi(r^+,\, r^-) &= \max\big(0,\, \min(1,\, r^+,\, r^-)\big) \end{aligned}$$

where ϕ is the flux limiter, and ΔU is an upwind difference in the state variable. The Courant number, ν, is the timestep divided by the grid spacing and multiplied by an eigenvalue of the system of equations corresponding to the state variable. The ratios r^+ and r^- are dimensionless values quantifying the change in the gradient across the five-point stencil. Referring to Figure 1, where the flux correction is being computed at the interface between cells 5 and 7, r^+ and r^- are determined by taking the inner product of sequential finite differences and dividing by the finite difference at the interface. For example, r^+ takes the inner product of the gradient across the interface between cells 7 and 8, as shown with the arrow (and which may or may not require an interpolation as shown by the small $\times$), with the gradient across the interface between cells 5 and 7. This is then divided by the gradient across the interface between cells 5 and 7, squared, thereby effectively capturing the sign change in the upwind direction. The value of r^- is computed in the same manner, only replacing the gradient across the interface between cells 7 and 8 with that between cells 3 and 5.

3.3. Partitioning & Locality

The numerical method presented above was formulated in such a way as to take full advantage of the maximum throughput of the GPU. The physical calculation is compressed into a few equations with minimal conditional branching, allowing all processing elements on the GPU to work concurrently. But this only ensures that a thread processing for a single cell can maximize its concurrency with other threads.

An efficient scheme for partitioning cells, and therefore computations, homogeneously across compute cores is still absolutely imperative. Taking an arbitrary k-dimensional mesh and mapping it into a 1-D array provides a direct, easily applied method for apportioning data elements across the GPU's cores. Elements of the 1-D array can be taken in sequence in blocks whose size matches that of the workgroup size on the GPU.

Computer data representation of a multidimensional array is, of course, actually linear, with data offsets calculated as a function of the row and column of the index. However, a GPU workgroup with limited memory and expensive calls to global memory provides the incentive for keeping data local. This is also true for a CPU despite a large L3 cache. There is still a motivation for ordering the array such that locality is largely preserved, as this eliminates cache and page misses. Accordingly, a key goal of the data structure used for CLAMR is to decompose the two-dimensional grid of state variables into a linear array while minimizing the number of out-of-workgroup neighbor accesses that must be made.

Figure 2 shows two contiguous load divisions of a section of a two-dimensional grid of workunits between several workgroups, each capable of processing N workunits. In the naïve contiguous case, ordering the workunits linearly by column and row requires $2N + 2$ off-tile accesses per workgroup (where here we take all N elements to be in a single row as the characteristic case), while the optimal case for a perfect square tile only necessitates $4\sqrt{N}$ off-tile accesses. This is a realization of a surface area to volume ratio minimization. Additionally, the naïve linear ordering progressively fragments workunits in a workgroup when local refinement occurs during solution of a problem.

To best achieve surface area to volume minimization, the use of appropriate space-filling curves for filling a k-dimensional space is necessary. Peano [27] explores the theory behind mapping a higher dimensional space to a one dimensional space when they have the same cardinality using a self-recursive stencil; see Figure 2 of Haverkort et al. [14] for a visual survey of several space-filling curves and Jin et al. [17] for a discussion of space-filling curves as a means for computational reordering. For the purposes of the current exposition, a comparison of the Hilbert curve [15] and Z-order curve [24] suggest that the Hilbert space-filling curve maximizes spatial locality. This is shown, with the use of CLAMR, by implementing partition measures; see Section 4 for a quantitative analysis.

Now the Hilbert space-filling curve, strictly speaking, requires a recalculation of the full index every time refinement occurs; this can lead to some unusual dead ends and consequent backtracking in the index ordering (Figure 3). Consequently, a local stencil preserving the entry and exit neighbors before refinement has been implemented, guaranteeing locality while avoiding fragmentation and backtracking. The relative orientation and direction of the required coarse curve through the cell is calculated, and by axiomatically matching the Hilbert curve of the refined mesh, a unique ordering is selected.

Load-balancing, one of the important considerations in distributed-memory message-passing systems, is cleanly solved by this implementation on the GPU. Namely, it can spawn a number of workunits not restricted by the number of physical processing elements on the device. Coupled with an index ordering minimizing the surface area of the tile, this scheme distributes the processing and memory access load equitably and near-optimally. For further internode discussion, see Section 2.2.2.

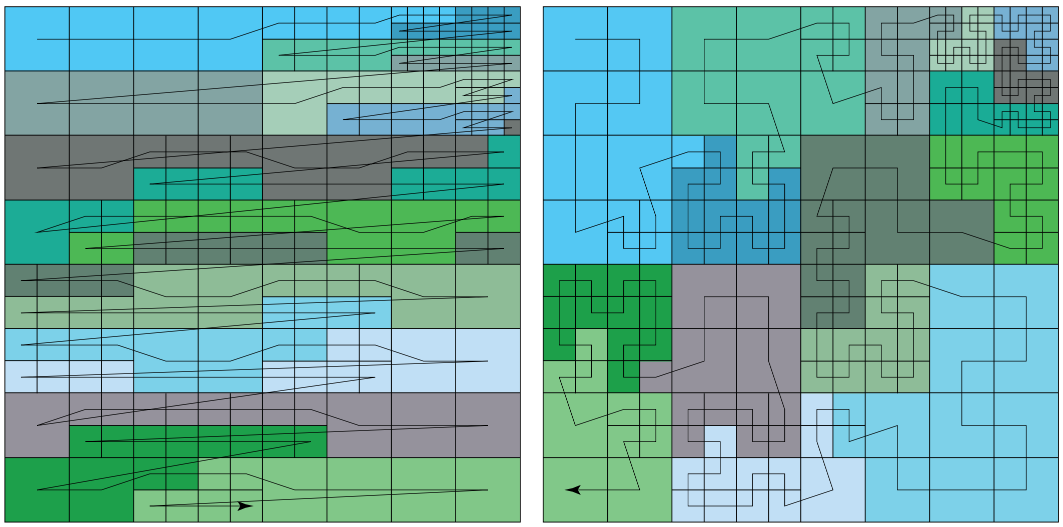

Figure 2. Here are two contiguous load divisions of a two-dimensional array between several workgroups (indicated by color). On the left is a naïve division which begins to fragment; it requires $2N + 2$ off-tile accesses for a tile size of N. On the right is an optimal division done by the Hilbert curve, which minimizes the surface area to volume ratio, consequently requiring only $4\sqrt{N}$ off-tile accesses for a tile size of N.

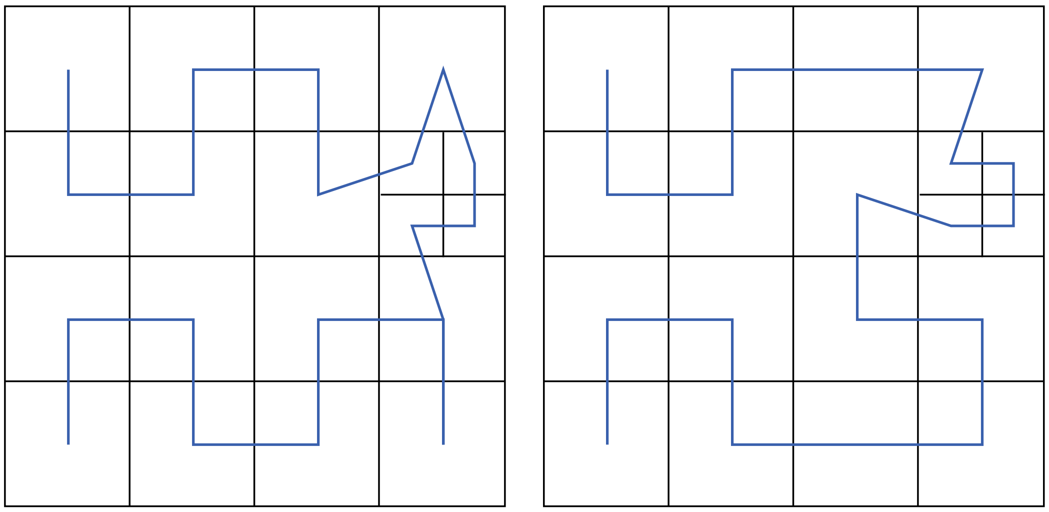

Figure 3. The Hilbert space-filling curve for a coarse mesh is shown locally refining with a global curve calculation on the left and locally refining with a local curve calculation on the right.

3.4. Neighbor Searching

The cost of the optimal spatial partitioning of the mesh is a nontrivial order of the data elements. Specifically, calculating neighbors becomes nontrivial. To make the problem worse, the basic stencil from the numerical method above requires consideration. For the Lax-Wendroff method, only a cell's most immediate neighbors' state information is needed. However, correcting to ensure TVD requires accessing two neighbors away.

From the hardware perspective of the GPU, we know that workgroups cannot communicate with each other and that the access times to global memory are much slower than the access times to local memory. So to take advantage of the spatial locality achieved by the partitioning of the last section, efficient search and retrieval of neighbor data is essential. It allows all necessary state information to be stored in local variables quickly at the start of the physics calculation kernel.

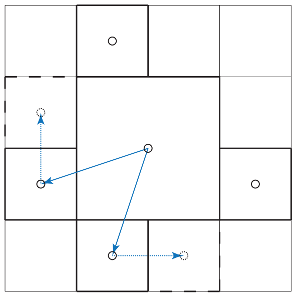

Figure 4. The neighbor accessing scheme illustrates the need to access the neighbor of a neighbor given certain levels of relative refinement.

In CLAMR, neighbor information was originally passed onto the GPU in the form of arrays of left neighbors, right neighbors, top neighbors, and bottom neighbors. While a significant portion of the cells would make local GPU memory accesses for neighbor information, given the surface area to volume ratio minimization achieved by using a space-filling Hilbert curve, global GPU memory accesses were still required. This in turn induced branching in the main computation kernel. Consider further the nature of neighbor accesses, as exemplified by Figure 4. In order to compute the flux across a face where the bordering cells are at a level of greater refinement, two neighbor accesses are required for a single face. However, the current cell doesn't have knowledge of one of those neighbors, and so it must first query the neighbor it does know to get the index in the Hilbert-curve-ordered array of the unknown neighbor.

Constructing the arrays of neighbors could proceed in several ways. A very naïve algorithm would simply take the current cell and search through every other cell of the mesh to check whether it's a neighbor, i.e. an $O(n^2)$ algorithm. A more clever approach uses a k-D tree as seen in Figure 5. A k-D tree recursively bisects the mesh using a weight function (which in this case is the number of cells) while alternating axes. Construction of the arrays of neighbors then requires $O(n \log n)$ time. A skip-list is a further optimization utilizing a hierarchy of linked-lists (effectively a hierarchy of trees with various levels of sparsity), thereby introducing speed-up as a result of random accesses. Unfortunately, the worst case can still give $O(n \log n)$ time complexity. Nevertheless, there is active research in construction of k-D trees. See section 4.1 of Godiyal et al. [12] for building a k-D tree on the GPU.

Originally CLAMR used a k-D tree constructed at every timestep and introduced a significant performance bottleneck. This was then replaced with a k-D tree construction on the initial timestep, followed by an update of the neighbor indexing based on tracking refinements and calculating relative offsets. This scheme introduced significant complexity into the code as well as implicitly assumed a Z-order space-filling curve.

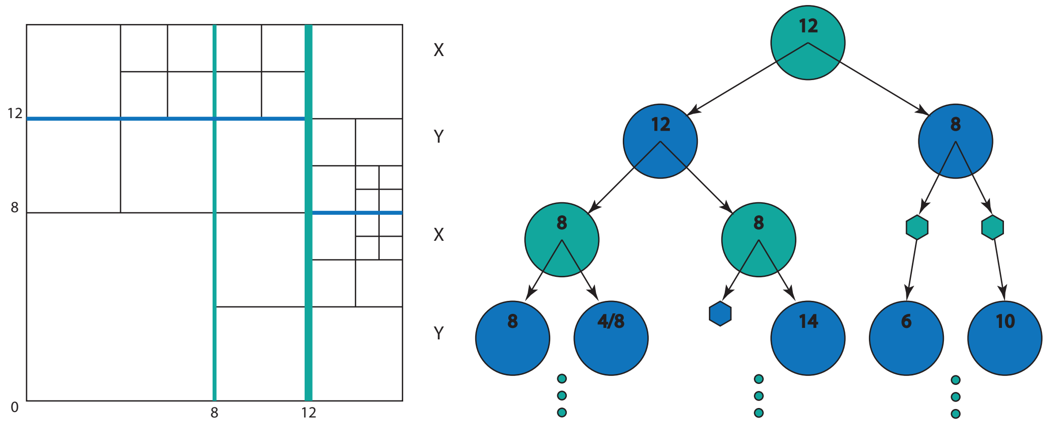

Figure 5. The mesh is recursively bisected using a k-D tree (i.e. a k-dimensional tree).

In the current version of CLAMR, we implemented a scheme which uses an analytic function to map cells into a hash table; the construction of a hash table only requires $O(n)$ time while look-up is $O(1)$. Another advantage is that the hash table construction is done on the GPU. Our design target was to find a perfect (collision-free) hash for simplicity and performance reasons. Exploiting the underlying structure of the AMR grid and that no point in the mesh could exist at two levels at one time, the minimal memory for a perfect hash table would be the size of the grid at the finest level. The use of the finest level grid allows the mapping of all possible cells with no extra space incorporated. As the maximum number of levels of refinement is increased, however, the hash table size can grow quickly. This is discussed in Section 4.4. Figure 7 shows two hash schemes, with Figure 6 providing a reference for the indexing. For more in-depth work on parallel hashing with the GPU, including collision handling algorithms, see Alcantara et al. [1]

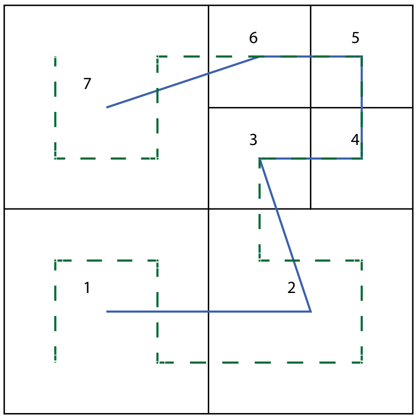

Figure 6. This is the reference grid for the hashing done in Figure 7. The blue curve fills the actual mesh, while the dotted green curve shows how the mesh would be filled if it were entirely refined at the finest level.

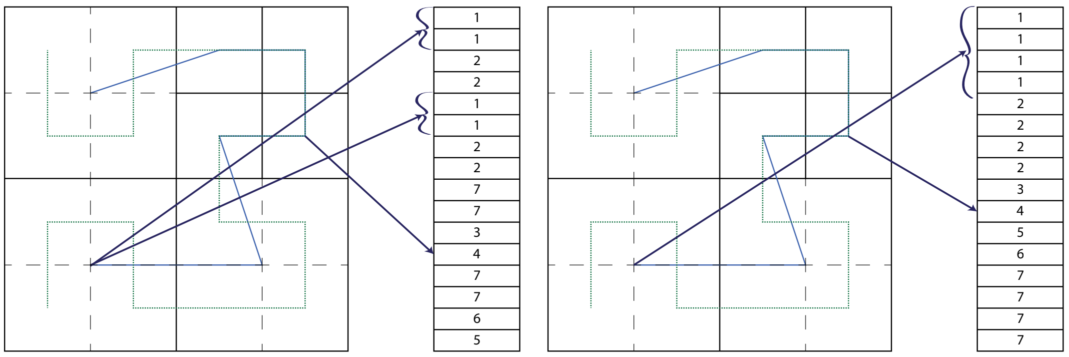

Figure 7. The first hash mapping follows the standard indexing used for arrays, spanning the x-axis first and then incrementing the y coordinate after one stride has finished. This is done on a superimposed grid of the finest level of refinement. The second hash mapping macroscopically matches the indexing induced by the Hilbert space-filling curve for a superimposed grid of the finest level of refinement.

3.4.1. Hash Implementation and Key Functions The hash indexing currently implemented uses a standard Cartesian coordinate defined key. A cell, regardless of its refinement, calculates its position as if it were at the finest level of refinement, and then it can make a constant time access to the hash table. Its position in the hash table is indexed as $w \cdot y + x$, where w is the width of the mesh, and y and x are its Cartesian coordinates, all taken at the finest level of refinement.

In the paper presenting CSAMR [16], a clever oct-based data structure is presented for an adaptive mesh. They too use a hash function to access parts of the mesh, but they store each level of refinement in the hash table, which requires more memory than the method implemented here. For the GPU we need to minimize the amount of memory required. Thus the hash table is structured at the greatest level of refinement, and coarser cells merely fill in multiple elements in the array. In CLAMR, there is actually an optimization which guarantees a cell never writes to more than 7 hash buckets. This is a byproduct of the rule that neighboring cells can only differ in scale by one level of refinement. See Figure 7 for the visualization.

On the other hand, given the standard indexing, there is significant divergence from the Hilbert curve in a cell's location in the arrays. As an interesting avenue of exploration, the hash mapping could tandemly structure the hash table to match the partition of mesh cells by using the Hilbert space-filling curve at the finest level of refinement. The gains here are more likely to be seen in the use of MPI, as it will reduce transfers of the pieces of the global hash table.

3.4.2. Algorithm Block for Neighbor Calculation Using MPI The biggest change associated with adding the MPI layer is in the neighbor calculation routine. The construction of the hash table necessitates focus on the local region of the mesh in order to achieve satisfactory memory scalability. The algorithm becomes more complicated as follows:

1. Determine $\min/\max$ i and j of local mesh region
2. Add 2-cell extra buffer at the coarsest level at the min & max to allocate hash table
3. Calculate local hash table
4. Calculate local neighbors
5. Determine all cells on the boundary of the local tile by searching for unsatisfied neighbors
6. Look inward from boundary cells for index data needed by other processors and gather to all processors
7. Each processor fills in off-tile extra buffer with gathered values
8. Recalculate neighbors for ghost regions with new hash
9. Find all off-processor data needed from new neighbors
10. Set up communication pattern for ghost regions
11. Start filling ghost region in arrays

This is the most complicated and error-prone portion of the MPI/OpenCL code. However, once this is done, the rest of the code simply uses the ghost update calls with little additional complexity over the serial code.

3.5. Enhanced Precision Sums

Modifying the data order has the undesirable side effect of changing the calculation of the total mass in a problem by a small amount. This is just due to the finite precision arithmetic of adding up the elements of a large array where addition is not truly associative. Though well within the error bounds of the simulation, it makes it difficult to verify the correctness of the programming. The problem grows worse as the problem size increases and the range in the data increases. So to properly allow the arbitrary data reordering for performance and locality, this error should be dealt with.

Enhanced precision sums can be used to reduce or even eliminate global sum errors. The basic concept is to use a second variable to carry the truncation part of the sum thereby increasing the digits of accuracy. Prior studies (Robey et al. [30]) showed that the Kahan sum reduces the error in calculation, as well as for MPI-distributed methods.

A demonstration problem was developed to stress the enhanced precision sums with a million cells, six orders of magnitude range in the initial conditions, and 3 levels of refinement. Every time the mesh was refined, a global sum was done for the Z-order and Hilbert data orderings. The maximum difference between the global sums for the two data orders in 100 iterations was measured at 448 times the machine epsilon. This is high enough to mask small errors in programming such as using old boundary conditions. Using the Kahan sum, the maximum difference in the sums for the two data orders was reduced to zero, thus vastly improving the detection of programming errors.

The cost of this technique is so small that it should be standard practice, particularly when data orderings are changing as is commonly the case in parallel calculations. The cost of performing the sum on the CPU is about 66% over that of the standard sum. Since the sum is less than one percent of the run-time cost, the addition to the run-time is negligible.

4. PERFORMANCE AND SCALABILITY

With the use of data ordering as a free parameter, the GPU-enabled code achieved an overall speed-up factor of one order of magnitude over the CPU-only code for our single node runs. Our tests were run on the Moonlight cluster at Los Alamos National Laboratory. A node on the cluster is comprised of two eight-core Intel Xeon E5-2670 CPUs rated at 2.6 GHz, each core with 0.5 MB of secondary cache and each chip with a 2MB tertiary cache between its eight cores. The two Xeon sockets share 32 GBytes of RAM on each node. Each compute node has two GPGPU NVIDIA Tesla M2090 cards connected to PCIe-2.0 x16 slots. Practical maximum bandwidth to these GPGPU cards is between 6.0 GB/s to 6.6 GB/s[‖].

For the test runs, however, a single node is defined as using a single core of one of the Xeon chips, and using a single Tesla M2090. Our code used GCC 4.4.6 as the C compiler, and the NVIDIA OpenCL SDK from the Cuda toolkit, version 4.1, provided the OpenCL 1.1 libraries. Shown below in Tables I and II is a detailed breakdown of the dominant routines in the various parts of CLAMR on a 256^2 coarse mesh and a 512^2 coarse mesh, respectively, with 2 and 4 levels of refinement. Explicit CPU use is compared to the hybrid implementation of the CPU combined with the GPU.

The importance of the results stems not from the absolute performance gain but from the realization that getting on the GPU performance curve is vital to future performance increases. For example, earlier runs were done on a cluster where each node was comprised of an AMD Opteron 6168 processor for the CPU and an NVIDIA Tesla C2050 for the GPU. The C compiler was GCC 4.5.1, and the NVIDIA OpenCL SDK, version 4.0.13, provided the OpenCL libraries. Those runs showed an overall speed-up factor of roughly $35\times$. Likewise, as the NVIDIA Kepler cards are released, the speed-up factors should increase again. Hence an order of magnitude speed-up is well worth the effort, especially as the GPU performance curve promises to drive this factor higher. Nevertheless, the numbers show the authors that there are still further optimizations desired in the finite difference kernel, as well in the hash setup of the neighbor calculation.

4.1. Partitioning Data

Demonstrating quantitatively the effectiveness of various partitioning via space-filling curves, Figure 8 shows that the Hilbert space-filling curve achieves the near optimal surface area to volume ratio minimization, where the partition measure used to evaluate the ratio is a straightforward calculation of the number of unique off-tile accesses divided by the number of unique off-tile accesses which would be made by the optimal partitioning. This effectively compares surface areas directly, thus producing a measure independent of tile size. A deeper analysis of partition measures is done in [37].

The measure of the Hilbert curve is in stark contrast to the measure of the Z-order curve, where fragmenting actually leads to an increase in the ratio as the maximum level of refinement is taken higher. And while the original order partitioning slowly minimizes the ratio further with increasing refinement, it is clear that the Hilbert curve achieves the best spatial partitioning.

The effect of the local stencil on partitioning is also seen in Figure 8, where a global Hilbert curve filling the space on every iteration is compared to an initial Hilbert curve followed by two separate local stencils for refinement – a fixed Z-order stencil and the Hilbert stencil. The surface area to

[‖]Specifications courtesy of `http://hpc.lanl.gov/tlcc2_home`.

volume ratio changes negligibly with the local stencil, but positively. The use of the local stencil for the Z-order actually counters the fragmentation associated with the global Z-order. Given the ease of implementation for a local stencil, and the inherently parallel nature of its application, these results provide good evidence for their use.

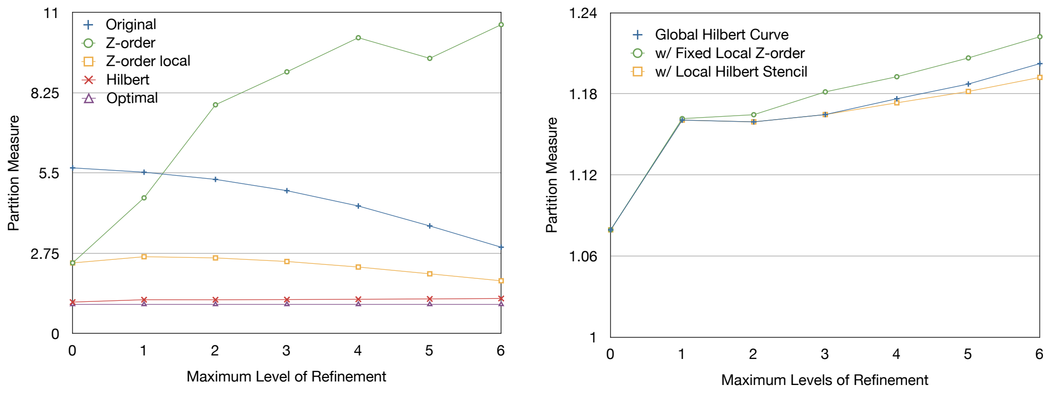

Figure 8. The graph on the left shows that the Hilbert curve achieves a near optimal minimization of the surface area to volume ratio. Hence, fewer off-tile neighbor accesses are made. The graph on the right shows that the use of a local stencil has a negligible, yet positive, effect on the surface area to volume ratio.

4.2. *Single-Node Speed-up of Partitioning*

Of further consequence, the partition measure acts as an indicator of the probable effect a partitioning method has on run-time. Our analysis shows increasing correlation as the maximum level of refinement is increased. The Pearson product-moment correlation coefficient between the partition measure and the measured run-time for the various space-filling curves shown in Figure 8 ranges from 0.869 for three levels of refinement up to 0.983 for six levels of refinement. The correlation is a low 0.224 when there is no refinement, but this is expected as the grid remains fixed and there is no need to rezone the mesh. Still, given the small mesh size of 256^2 for the partition measure comparison runs, and given the small statistical sample, the data show a strong but not perfect correlation which is likely clearer for larger meshes.

On the other hand, our partition measure is first-order in the sense that we have only considered spatial locality, i.e. the amount of data which is to be transferred. We have not provided a secondary measure which fully accounts for the effects of hardware, such as exploiting quad-loads from cache on the GPU. Consequently, the original order still has impressive performance on some hardware when comparing to the best space-filling curve, the Hilbert curve with the local stencil, since a single off-tile access has the tendency to grab a neighboring cell which will be needed by another cell in the workgroup. Shown below in Figure 9 is a collection of normalized runs on the aforementioned cluster comprised of single nodes with the AMD CPU and NVIDIA Tesla C2050 GPU as a function of partition measure. As the graph shows, partition measure based on spatial metrics is a solid indicator of runtime, except in the original order case where cache accessing is not being taken into account. Further exploration of a more accurate partition measure is beyond the scope of this work.

Nevertheless, the performance of the Hilbert curve with local stencil matches the original order. And the uniform interface of the space-filling curve approach makes coding much simpler, especially in the context of MPI. The effects on the MPI performance are presented next.

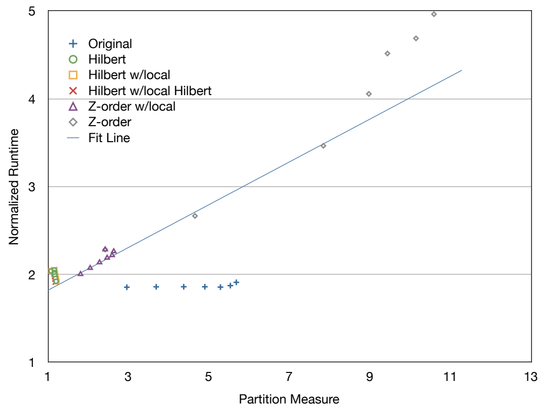

Figure 9. Here is a collection of normalized runs as a function of partition measure. We see that the partition measure, based on spatial metrics, is a solid indicator of runtime except for the original order since cache accessing effects are not accounted for in the metrics.

4.3. MPI Performance: Strong Scaling, Weak Scaling, and Partition Measure

We tested the effect of our Hilbert space-filling curve scheme on both strong scaling and weak scaling, comparing it to a scheme which uses the original data order both intranode and internode. All data presented in this section was obtained from runs on the aforementioned Moonlight cluster. The strong scaling tests were run with a fixed coarse mesh size of 512^2 at both 2 levels and 4 levels of refinement for 500 iterations. Figure 10 only shows the data for the Hilbert curve method, as the original order produced nearly identical data and hence for clarity we plot fewer curves. We see that even at a few nodes the MPI only scheme quickly outperforms the single node CPU, as expected. However, the hybrid MPI/GPU scheme suffers from the overhead of pulling data off the GPU for MPI communication. Further optimizations are needed to better overlap GPU memory accesses and MPI communication calls. Still, by 16 nodes we see convergence.

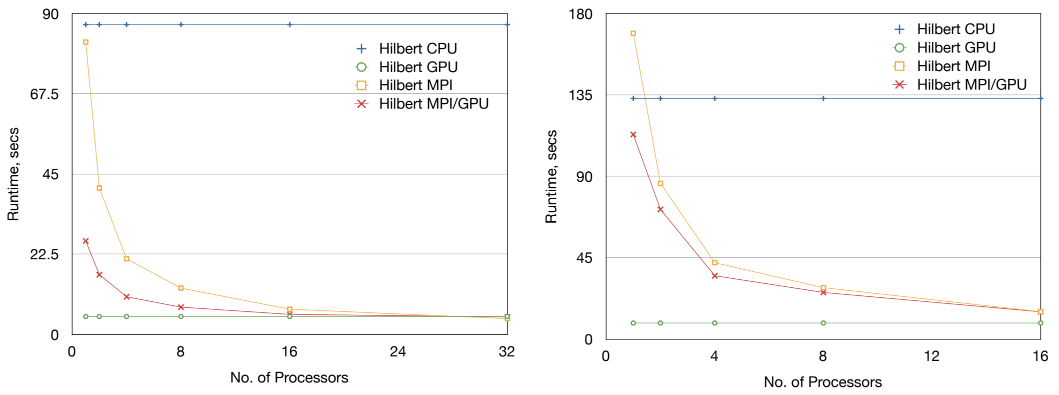

Figure 10. The left graph shows strong scaling for a problem run on a 512^2 coarse mesh with a maximum of 2 levels of refinement for 500 iterations, and the right graph shows the same problem but with a maximum of 4 levels of refinement.

The weak scaling tests were run with a base coarse mesh of 256^2, so that the problem size is $256^2 N$ where N is the number of nodes. Again the runs were done for 500 iterations at both 2 and 4 levels of refinement. We only show the case for 2 levels of refinement in Figure 10. We see in the left plot that, for both the Hilbert scheme and the original order scheme, the scaling efficiency is better for MPI only versus the MPI/GPU hybrid. This is true for 4 levels of refinement as well. We also see that the scaling efficiency curves for both MPI and the MPI/GPU hybrid exhibit similarity;

of particular interest is that the similarity is mirrored in the right plot of Figure 10 which shows the MPI partition measure as a function of the number of nodes. Here the partition measure is the surface area to volume ratio of a node's region of the global mesh.

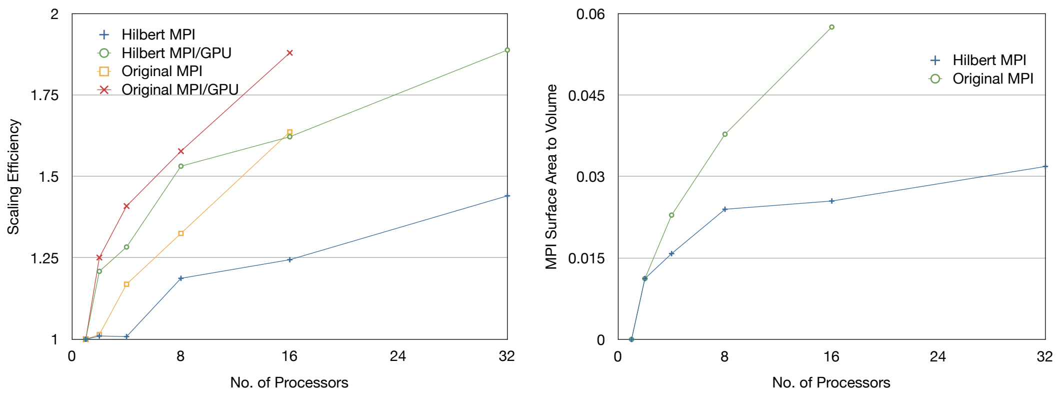

Figure 11. The left graph shows weak scaling for a problem run on a 256^2N coarse mesh (N equal to the number of nodes) with a maximum of 2 levels of refinement for 500 iterations. The right graph shows the MPI partition measure as a function of the number of nodes for the problem run.

Figure 11 also shows that our Hilbert scheme obtains better scaling efficiency then the original order at 2 levels of refinement. For 4 levels of refinement, however, we weren't able to run out the original order beyond 8 nodes; that limited data seems to show that the original order obtains slightly better scaling than the Hilbert scheme, but again it simply was not run with enough nodes.

Our conclusion is that the focus on locality is the right choice, given the similarity of the scaling efficiency curves to the MPI partition measure curves. Ultimately our partition measure needs to be expanded beyond solely spatial metrics and include some kind of cache effect metric.

4.4. *Neighbor Calculation Data*

On a separate note, the hash table implementation for neighbor calculations shows an impressive speed-up over the k-D tree, as seen in Figure 12. The CPU hash achieves an order of magnitude speed-up over the k-D tree, while the GPU hash achieves a speed-up of three orders of magnitude. By increasing the memory complexity of the neighbor algorithm, a large reduction from $O(n \log n)$ to $O(n)$ is achieved in the time complexity. And it is important to note two aspects regarding the additional memory space: it is used only temporarily, and each bucket in the hash table only stores a single integer for perfect hashing. In particular, production level codes can have hundreds of state variables per cell, each stored as a float or double, so the relative space taken in global memory for the hash table is not unreasonable.

Figure 12 does show that there is still an issue with the growth of the hash table as the maximum number of allowable refinement levels ($l_{\max}$) increases. The worst case spatial complexity for our 2-D problem goes as $O(4^{l_{\max}}n)$. However, this can be resolved by "local" hashing, which partially constructs the hash table in local regions of the mesh, or by "iterative" hashing, which constructs the hash table iteratively by level of refinement, thus exploiting the fact that the entire hash table need not be constructed at once for our numerical methods. Implementing these hashes is an exciting avenue of research.

The impressive speed-ups prompted further research into the uses of hashing for numerical methods on GPUs and CPUs. Robey et al. [29] extends beyond neighbor calculation; it examines sorting, remapping, and table look-up as well. An important point discussed there is that hash-based algorithms benefit from a two-fold performance boost. They see speed-up both in algorithm design and in the ease of parallelization without recourse to scan operations (at least in the neighbor calculation algorithm).

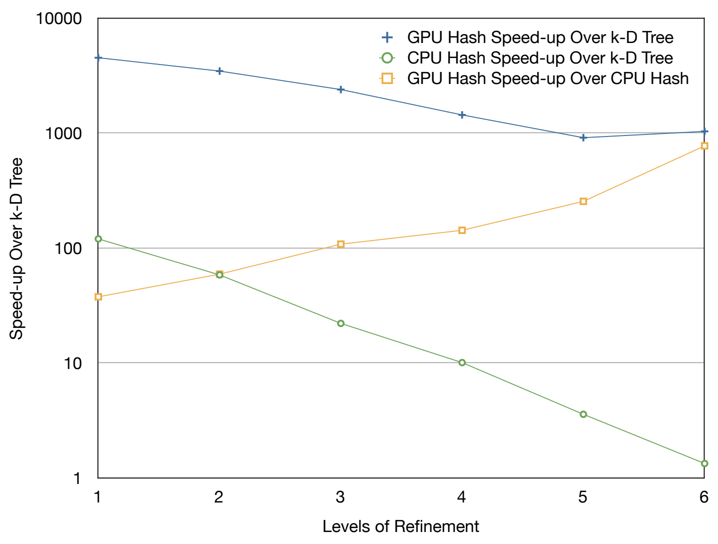

Figure 12. The speed-up over a k-D tree for the hash table implementation on both the CPU and GPU.

5. CONCLUSION

We've shown that it is viable to implement cell-based adaptive mesh refinement on clusters of general purpose graphics processing units. We've shown that finite differencing in this scheme requires careful consideration of the fluxes in order to ensure the conservation of the state variables. We've also stressed the necessity of thinking locally. We took data ordering as the free parameter to establish a partitioning of the mesh which is the most efficient, and then we addressed the issues it created. Namely, the nontrivial neighbor accessing was resolved via a hash-based method of neighbor indexing. Ultimately, we've shown that GPGPUs can be used for intense numerical calculations while making memory and not FLOPs the primary focus in algorithm architecture. The order of magnitude overall speed-up is indicative of a methodology worth the effort.

ACKNOWLEDGEMENTS

We would like to thank Rachel Robey for the real-time OpenGL graphics code, H. C. Edwards of Sandia National Laboratories for the global Hilbert space-filling curve code, and Richard Barrett for the sparse MPI communication package. Discussions with the Applied Math Department at the University of Washington, colleagues at Lawrence Livermore National Lab, the University of California at Davis, and fellow colleagues at LANL, helped refine our work as it was developing. The lead author would also like to thank James Quirk for invaluable discussions on "computational classics".

REFERENCES

1. Dan A. Alcantara, Andri Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)*, 28(5), Dec. 2009.
2. M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.
3. Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

4. Andr R. Brodtkorb, Martin L. Stra, and Mustafa Altinakar. Efficient shallow water simulations on gpus: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1 – 12, 2012.
5. Manuel J. Castro Díaz, Sergio Ortega Acosta, Marc de la Asunción, José Miguel Mantas, and José M. Gallardo. GPU computing for shallow water flow simulation based on finite volume schemes. *Comptes Rendus Mécanique*, 339(2):165–184, 2011.
6. B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Performance debugging of gpgpu applications with the divergence map. In *22nd International Symposium on Computer Architecture and High Performance Computing*, pages 33 –40, Oct. 2010.
7. Neal E. Davis, Robert W. Robey, Charles R. Ferenbaugh, David Nicholaeff, and Dennis P. Trujillo. Paradigmatic shifts for exascale supercomputing. *The Journal of Supercomputing*, pages 1–22, 2012.
8. Stephen F. Davis. A simplified tvd finite difference scheme via artificial viscosity. *SIAM Journal on Scientific and Statistical Computing*, 8(1):1–18, 1987.
9. Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 47–, Washington, DC, USA, 2004. IEEE Computer Society.
10. D.L. George. *Numerical approximation of the nonlinear shallow water equations with topography and dry beds: a Godunov-type scheme*. University of Washington, 2004.
11. Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering (IJCSE)*, 4(1):36–55, 2008.
12. Apeksha Godiyal, Jared Hoberock, Michael Garland, and John Hart. Rapid multipole graph drawing on the gpu. In *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 90–101. Springer Berlin / Heidelberg, 2009.
13. M. Griebel and G. W. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25:827–843, 1999.
14. Herman J. Haverkort and Freek van Walderveen. Locality and bounding-box quality of two-dimensional space-filling curves. *CoRR*, abs/0806.4787, 2008.
15. David Hilbert. Ueber die stetige abbildung einer line auf ein flchenstck. *Mathematische Annalen*, 38:459–460, 1891. 10.1007/BF01199431.
16. Hua Ji, Fue-Sang Lien, and Eugene Yee. A new adaptive mesh refinement data structure with an application to detonation. *Journal of Computational Physics*, 229(23):8981–8993, 2010.
17. Guohua Jin and John Mellor-crummey. Using space-filling curves for computation reordering. *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, 2005.
18. Khronos Group. *The OpenCL Specification*, version 1.1, revision 44 edition, June 2011.
19. Donald E. Knuth. *The art of computer programming. Vol. 1, Fundamental algorithms*. Addison-Wesley Pub. Co., 3 edition, July 1997.
20. L. D. Landau and E. M. Lifshitz. *Fluid Mechanics, Second Edition: Volume 6*. Course of theoretical physics. Butterworth-Heinemann, 2 edition, January 1987.
21. Peter Lax and Burton Wendroff. Systems of conservation laws. *Communications on Pure and Applied Mathematics*, 13(2):217–237, 1960.
22. R.J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge texts in applied mathematics. Cambridge University Press, 2002.
23. Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.
24. G.M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., 1966.
25. NVIDIA. Opencl best practices guide, May 2010.
26. NVIDIA. Opencl programming guide for the cuda architecture, version 3.2, Aug. 2010.
27. G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890. 10.1007/BF01199438.
28. James J. Quirk. Amr_sol: Design principles and practice. *In 29th Computational Fluid Dynamics, VKI Lecture Series, Chapter 5*, 1998.
29. Rachel N. Robey, David Nicholaeff, and Robert W. Robey. Hash-based algorithms for discretized data. *In Review, SIAM J. of Sci. Com.*, 2012.
30. Robert W. Robey, Jonathan M. Robey, and Rob Aulwes. In search of numerical consistency in parallel programming. *Parallel Comput.*, 37:217–229, April 2011.
31. Hsi-Yu Schive, Yu-Chih Tsai, and Tzihong Chiueh. Gamer: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457, 2010.
32. Gilad Shainer, Ali Ayoub, Pak Lui, and Tong Liu. Raising the speedlimit: New gpu-to-gpu communications model increases cluster efficiency, January 2011.
33. P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM J. Numer. Anal.*, 21(5):995–1011, 1984.
34. Terry Tao. The shallow water wave equation and tsunami propagation. `http://terrytao.wordpress.com/2011/03/13/`, 2011.
35. Alvin Trivelpiece. Exascale Workshop Panel Meeting Report. `science.energy.gov/`, January 2010.
36. H.C Yee. Construction of explicit and implicit symmetric tvd schemes and their applications. *Journal of Computational Physics*, 68(1):151 – 179, 1987.
37. G. W. Zumbusch. On the quality of space-filling curve induced partitions. *Z. Angew. Math. Mech.*, 81:25–28, 2001. Suppl. 1, also as report SFB 256, University Bonn, no. 674, 2000.

Table I. Performance on a Single Node[⌝] – 200 iterations on a 256^2 Coarse Mesh

	2 Levels of Refinement			4 Levels of Refinement		
Function	CPU (s)	GPU (s)	Speed-up	CPU (s)	GPU (s)	Speed-up
Total	6.478	0.895	7.238	11.769	1.223	9.623
Finite Difference	5.096	0.707	7.208	6.197	0.813	7.622
Refine Potential	0.251	0.045	5.578	0.407	0.060	6.783
Rezone All	0.064	0.053	1.208	0.165	0.052	3.173
Neighbor Calc	0.653	0.051	12.804	4.512	0.250	18.048
Write Device	0	0.002	—	0	0.002	—
Read Device	0	0.000	—	0	0.000	—

[⌝] A single node is defined as one core of an Intel Xeon E5-2670 for the CPU, and an NVIDIA Tesla M2090 for the GPU.

Table II. Performance on a Single Node[⌝] – 200 iterations on a 512^2 Coarse Mesh

	2 Levels of Refinement			4 Levels of Refinement		
Function	CPU (s)	GPU (s)	Speed-up	CPU (s)	GPU (s)	Speed-up
Total	26.001	3.210	8.100	45.155	4.211	10.723
Finite Difference	19.310	2.619	7.373	21.719	2.810	7.729
Refine Potential	1.070	0.133	8.045	1.681	0.175	9.606
Rezone All	0.480	0.175	2.743	1.000	0.142	7.042
Neighbor Calc	3.504	0.198	17.697	18.943	0.980	19.330
Write Device	0	0.005	—	0	0.005	—
Read Device	0	0.000	—	0	0.000	—

[⌝] A single node is defined as one core of an Intel Xeon E5-2670 for the CPU, and an NVIDIA Tesla M2090 for the GPU.