CELL-BASED ADAPTIVE MESH REFINEMENT IMPLEMENTED WITH GENERAL PURPOSE GRAPHICS PROCESSING UNITS*

DAVID NICHOLAEFF[†‡‖], NEAL DAVIS[†§], DENNIS TRUJILLO[†¶], AND ROBERT ROBEY[†‖]

Abstract.
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; the resulting nontrivial arrangement of cells is addressed by a Cartesian-indexed hash mapping which allows for efficient parallel neighbor accesses; and, an enhanced-precision sum ensures consistency in the calculation regardless of data order.

The relative speed-up of the GPU enabled AMR code is compared to the respective serial implementation, providing strong evidence for the need to implement numerical algorithms for physics applications on heterogeneous architectures exploiting GPGPUs.

Key words. Cell-based Adaptive Mesh Refinement, AMR, GPGPU, GPU, OpenCL, heterogeneous architecture, exascale computing, parallel physics simulations

AMS subject classifications. 65M08, 65M50, 76B15, 76M12

1. Introduction. At the present, numerical applications on graphics processing units (GPUs) remain scarce. We present an OpenCL implementation of a cell-based adaptive mesh refinement (AMR) scheme modeling the shallow water equations, although the framework is extensible to a variety of governing equations. We then discuss algorithm architecture, highlighting the importance of locality, as optimal partitioning of the computational domain to ensure maximized locality of data allows the full exploitation of the massive number of parallel threads on the GPU. 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 [16], is effectuated by a combination of CPU memory management and GPU computation. Load balancing is achieved through a stencil-based implementation of a space-filling curve, eliminating the need for a complete recalculation of the indexing on the mesh. A Cartesian-indexed hash 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). And the relative speed-up of the GPU-enabled AMR code over the original CPU-only serial version shows speed-up factors of 30-40×, obtained over a wide range of problem sizes. Griebel et al. [10] explored a similar synthesis of hashing with space-filling curves in a multigrid setting.

The results of our cell-based AMR scheme, aptly named CLAMR, provide strong evidence that parallelization using the GPU delivers significant speed-up for typical

*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.

[†]XCP-2 Eulerian Codes, Los Alamos National Laboratory, Los Alamos, NM.

[‡]Department of Physics & Astronomy, University of California at Los Angeles, Los Angeles, CA.

[§]Department of Nuclear, Plasma, & Radiological Engineering, University of Illinois at Urbana–Champaign, Urbana, IL.

[¶]Department of Physics, New Mexico State University, Las Cruces, NM.

[‖]Corresponding authors: dnic@lanl.gov, brobey@lanl.gov

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 [30]. 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 [28].

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. [6]; locality is markedly critical, and its proper application demands careful consideration of 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 the message passing interface (MPI). Ultimately our results suggest that future numerical physics codes must use heterogeneous architectures to see significant speed-up, but they cannot do so without devising better algorithms in lieu of ever greater computational power.

2. Cell-based AMR 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

*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`.

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 ([20] ,[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, such as incorporating a hierarchical data-processing structure of heterogeneous substructures. For both an overview and references to multiple sources of this argument, see Davis et al. [6] 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 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.

†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. This is not discussed here, but the shallow water modeling by LeVeque [20] specifically addresses this issue.

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 $O(n \log n)$ [17], or even simple debugging and profiling, 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 specifically, memory cannot be dynamically allocated on the GPU as of OpenCL specification 1.1 [16]. Therefore, 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 [24, Ch. 3], with an even larger factor slowdown resulting from writes back to the CPU [23, 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, the simple platform of one CPU core communicating with the GPU 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.

Our algorithm makes full use of the GPU for the numerical methods, but it does not exploit the multicore functionality of modern CPUs. Nevertheless, to allow for future use of this added level in the hierarchy of compute elements, we assign a rank to each core of all the CPUs. This implies that multiple cores control a single GPU, or more generally there will be a larger number of CPU cores controlling GPUs. Hence, each CPU core has its own command queue which will be serially passed down to the GPU.

The considerations in decomposing the mesh remain with the implementation of MPI: locality is the highest priority – in order to reduce data transfers across compute elements – hence moving to MPI challenges the versatility of our choice of partitioning.

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; as seen in the control flow of CLAMR in Figure 3.1, 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 new 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 forcing enhanced-precision sums.

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 [16].) 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. 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, and the new cell neighbors are set.

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

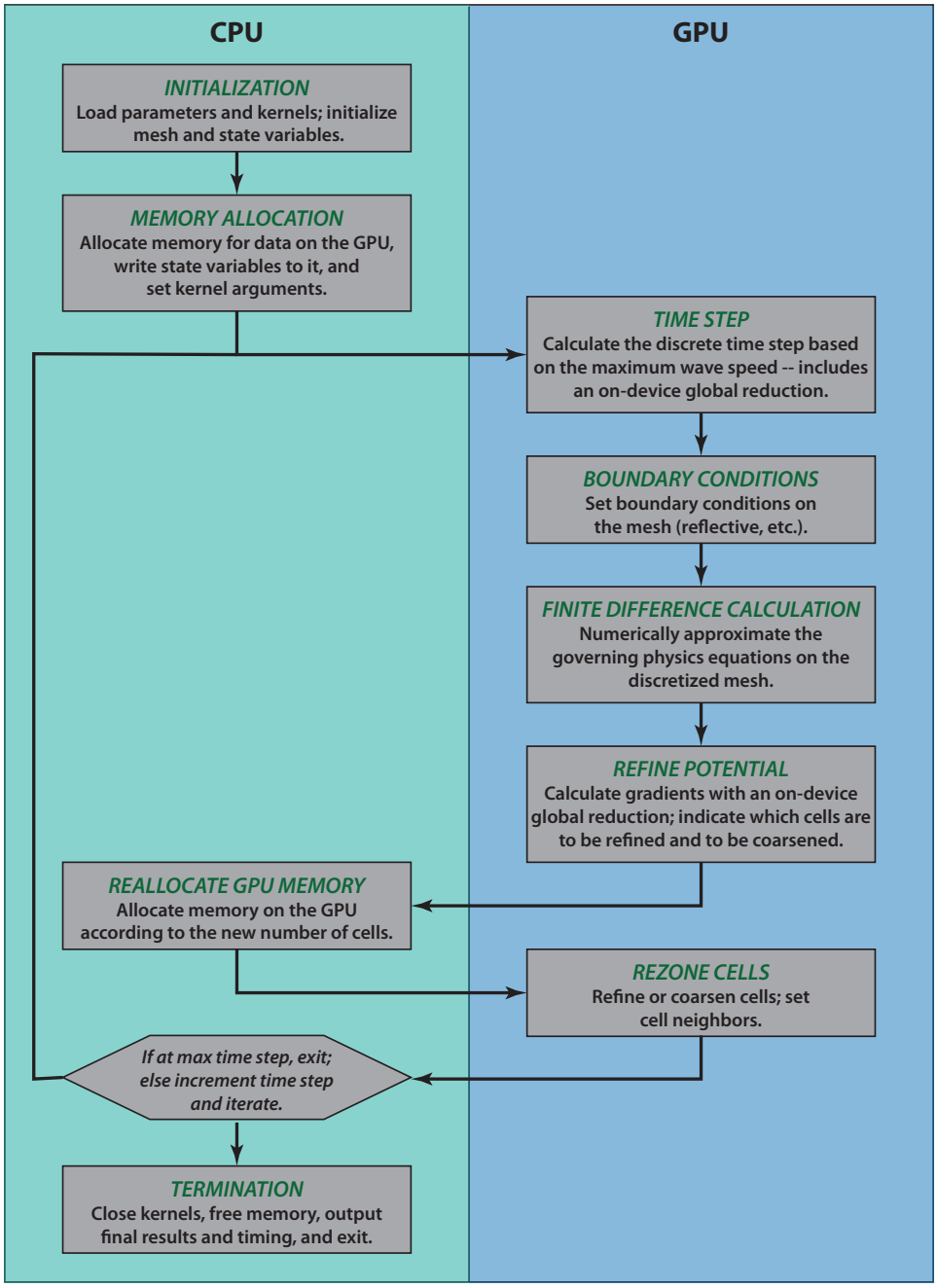

Fig. 3.1: The control flow of CLAMR.

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. In their conservative form, the equations are:

$$\frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0 \qquad \text{(Conservation of mass)}$$

$$\frac{\partial (hu)}{\partial t} + \frac{\partial}{\partial x}\left(hu^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial y}(huv) = 0 \qquad \text{(Conservation of } x\text{-momentum)}$$

$$\frac{\partial (hv)}{\partial t} + \frac{\partial}{\partial x}(hvu) + \frac{\partial}{\partial y}\left(hv^2 + \frac{1}{2}gh^2\right) = 0 \qquad \text{(Conservation of } y\text{-momentum)}$$

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.[‡] 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 [18] on long gravity waves and shallow-water theory.

For the discretization, we use a total variation diminishing (TVD) finite difference scheme based on a two-step Lax-Wendroff method [19] 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 [21] and George's Master's thesis [8]. For more information about the complete numerical method as implemented, see Davis [7], Sweby [29], and Yee [12].

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:

$$U^{n+1/2}_{i+1/2,\ j} = \frac{r_i U^n_{i+1,\ j} + r_{i+1} U^n_{i,\ j}}{r_{i+1} + r_i} - \Delta t \left(\frac{F^n_{i+1,\ j} A_{i+1} a_{i+1} - F^n_{i,\ j} A_i a_i}{V_{i+1} v_{i+1} + V_i v_i} \right)$$

$$U^{n+1/2}_{i,\ j+1/2} = \frac{r_j U^n_{i,\ j+1} + r_{j+1} U^n_{i,\ j}}{r_{j+1} + r_j} - \Delta t \left(\frac{G^n_{i,\ j+1} A_{j+1} a_{j+1} - G^n_{i,\ j} A_j a_j}{V_{j+1} v_{j+1} + V_j v_j} \right).$$

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.

[‡]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 `http://terrytao.wordpress.com/2011/03/13/the-shallow-water-wave-equation-and-tsunami-propagation/`.

The adaptation from the regular grid equations,

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

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 3.2). 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

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

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 3.2), which appears in the denominator. In order to allow for arbitrary dimension, the volume contributions by neighboring cells are multiplied by the scale factors

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

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

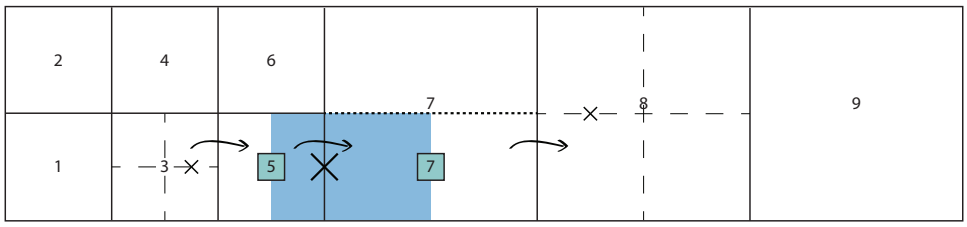

Fig. 3.2: The finite difference stencil shows the half-timestep calculation, as well as the comparison of gradients for the flux correction to remove oscillations (applied at the full-timestep). The value of a state variable is linearly interpolated at the face (linear interpolations are shown with an $\times$), 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).

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 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 [4].

The full-timestep calculation is:

$$U_{i,\,j}^{n+1} = U_{i,\,j}^{n} - \Delta t \left(\frac{\overline{F_{i+1/2,\,j}^{n+1/2}} - \overline{F_{i-1/2,\,j}^{n+1/2}}}{\Delta x} + \frac{\overline{G_{i,\,j+1/2}^{n+1/2}} - \overline{G_{i,\,j-1/2}^{n+1/2}}}{\Delta y} \right).$$

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 3.2, 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:

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

where ϕ is the flux limiter, and ΔU is an upwind difference in the state variable. The Courant number, ν, is an eigenvalue of the system of equations corresponding to the state variable multiplied by the timestep and divided by the cell width. The ratios r^+ and r^- are dimensionless values quantifying the change in the gradient across the five-point stencil. Referring to Figure 3.2, 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. And while for a CPU a large L3 cache can hold most of the data in cache memory, 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 3.3 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 [25] 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. [11] for a visual survey of several space-filling curves and Jin et al. [15] 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 [13] and Z-order curve [22] 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 3.4). Consequently, a local stencil preserving the entry and exit neighbors 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. See Davis et al. [5] for a visualization of the stencil cases and a more in-depth treatment.

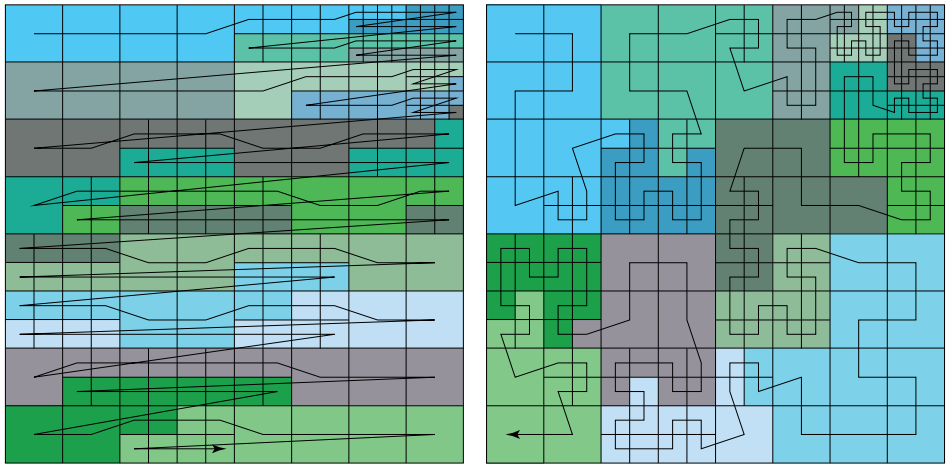

Fig. 3.3: Here are two contiguous load divisions of a two-dimensional array between several workgroups (indicated by color). On the left is a naïve division which begins to fragment; it requires $2N + 2$ off-tile accesses for a tile size of N. On the right is an optimal division done by the Hilbert curve, which minimizes the surface area to volume ratio, consequently requiring only $4\sqrt{N}$ off-tile accesses for a tile size of N.

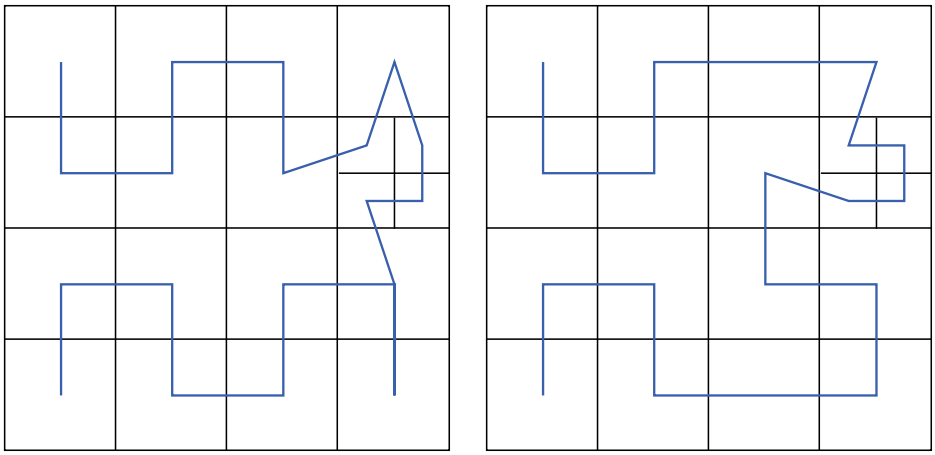

Fig. 3.4: The Hilbert space-filling curve for a coarse mesh shown locally refining with a global curve calculation on the left, and locally refining with a local curve calculation on the right.

The load balancing, which dominates considerations in distributed-memory message-passing systems, is cleanly solved by the implementation on the GPU, which 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.

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.

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 3.5. 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.

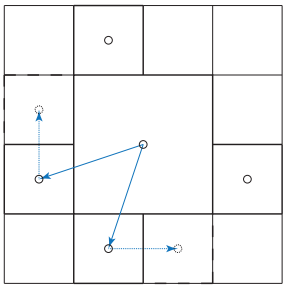

Fig. 3.5: The neighbor accessing scheme illustrates the need to access the neighbor of a neighbor given certain levels of relative refinement.

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 3.6. 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. [9] 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.

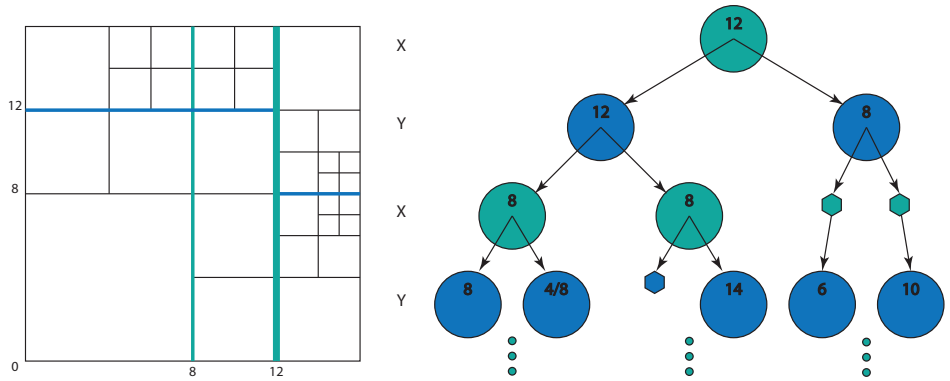

Fig. 3.6: Recursively bisecting the mesh using a k-D tree (i.e. a k-dimensional tree).

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 lookup 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. Figure 3.8 shows two hash schemes, with Figure 3.7 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]

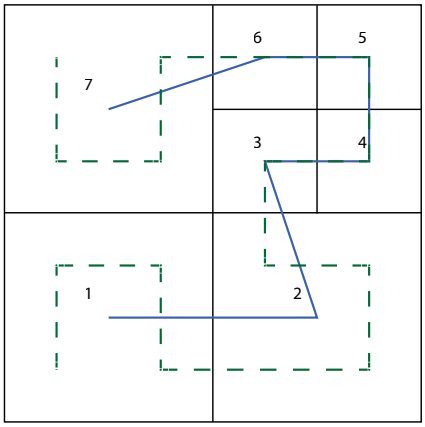

Fig. 3.7: This is the reference grid for the hashing done in Figure 3.8. The blue curve fills the actual mesh, while the dotted green curve shows how the mesh would be filled if it were entirely refined at the finest level.

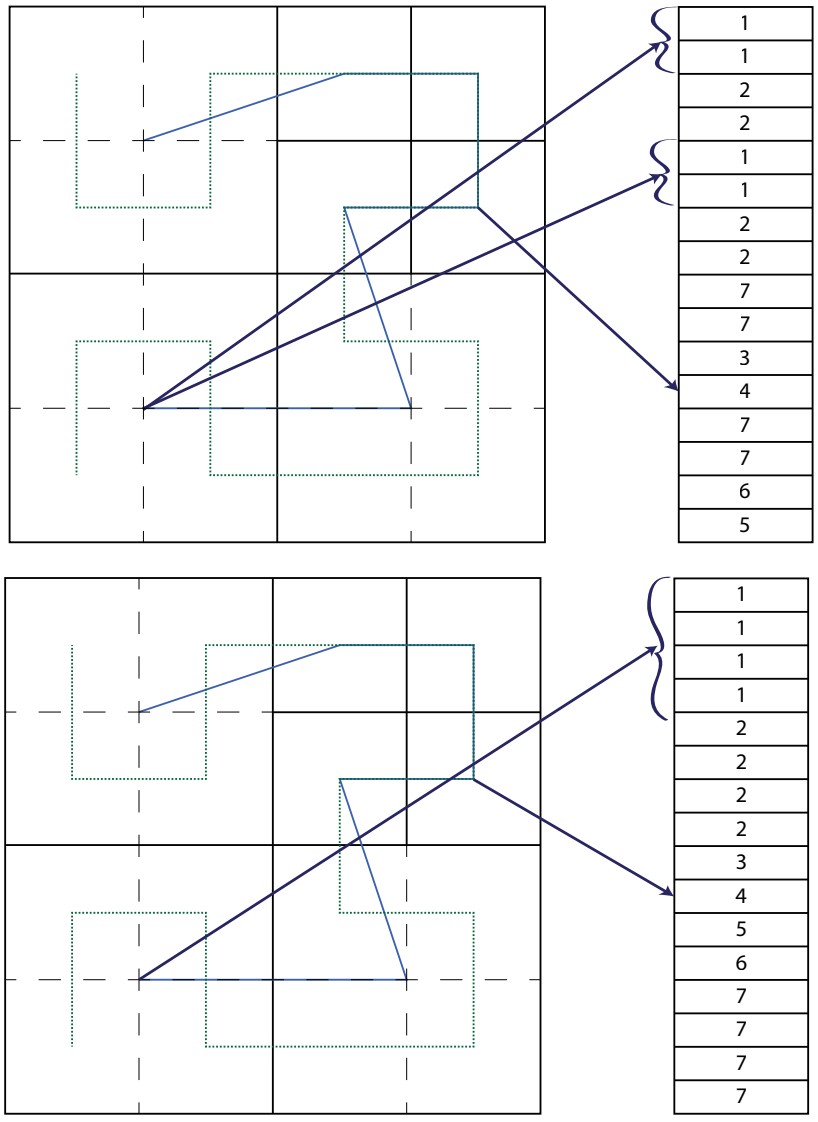

Fig. 3.8: The first hash mapping follows the standard indexing used for arrays, spanning the x-axis first and then incrementing the y coordinate after one stride has finished. This is done on a superimposed grid of the finest level of refinement. The second hash mapping macroscopically matches the indexing 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 [14], 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. See Figure 3.8 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
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

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. [27]) 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, an overall speed-up of 30-40× over the CPU using the GPU is reported. Our tests were run with 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. Shown below in Table 4.1 is a detailed breakdown of the various timings for parts of CLAMR for the specific example of a 450×450 mesh, comparing explicit CPU use with the heterogeneous platform utilizing the GPU. The overall speed-up for the test run in the table is 38.9×.

Table 4.1: Performance for CLAMR on a 450×450 coarse grid run on an NVIDIA Tesla C2050 and AMD Opteron 6168.

Function	CPU (s)	GPU (s)	Speed-up
Timestep Calc	10.576	0.143	73.958
Apply BCs	1.518	7.182	41.401
Finite Difference (TVD)	295.826		
Refinement Potential	11.630	0.397	29.295
Rezoning	7.115	0.684	10.402
Partitioning	2.258	0	—
Mass Sum (Kahan)	2.346	0.095	24.695
Write to Device	0	0.0072	—
Read from Device	0	0.0129	—

As a demonstration of scalability, the statistics of many runs shows a consistent speed-up across varying grid sizes. Run on regular increments of grid size from 128×128 to 512×512, an average speed-up of 36.1× with a standard deviation of 4.9× is achieved. The importance of this result stems not from the absolute performance

gain but from the realization that getting on the GPU performance curve is vital to future performance increases.

4.1. Partitioning Data. Demonstrating quantitatively the effectiveness of various partitioning via space-filling curves, Figure 4.1 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 [31].

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 naïve partitioning slowly minimizes the ratio further with increasing refinement, it is clear that the Hilbert curve achieves the best spatial partitioning.

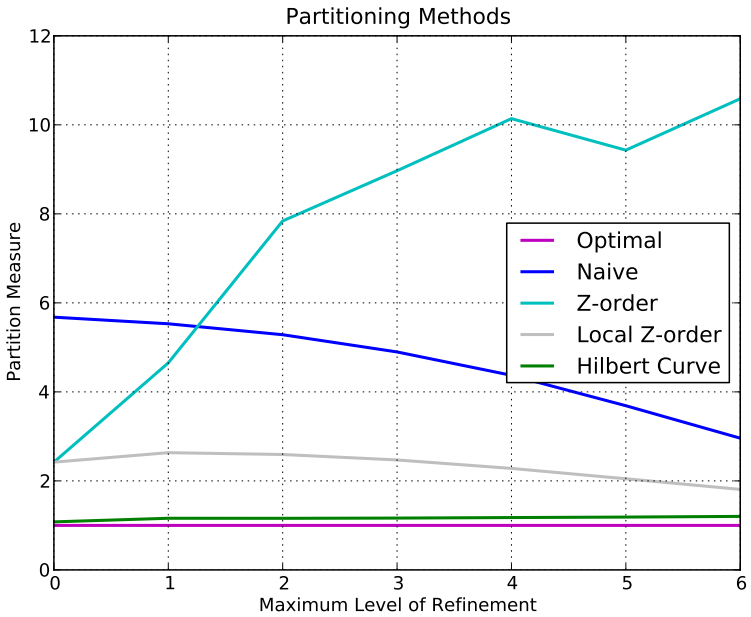

Fig. 4.1: The graph shows that the Hilbert curve achieves a near optimal minimization of the surface area to volume ratio. Hence, fewer off-tile neighbor accesses are made.

The effect of the local stencil on partitioning is seen in Figure 4.2, 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 volume ratio changes negligibly with the local stencil, but positively. In Figure 4.1 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.

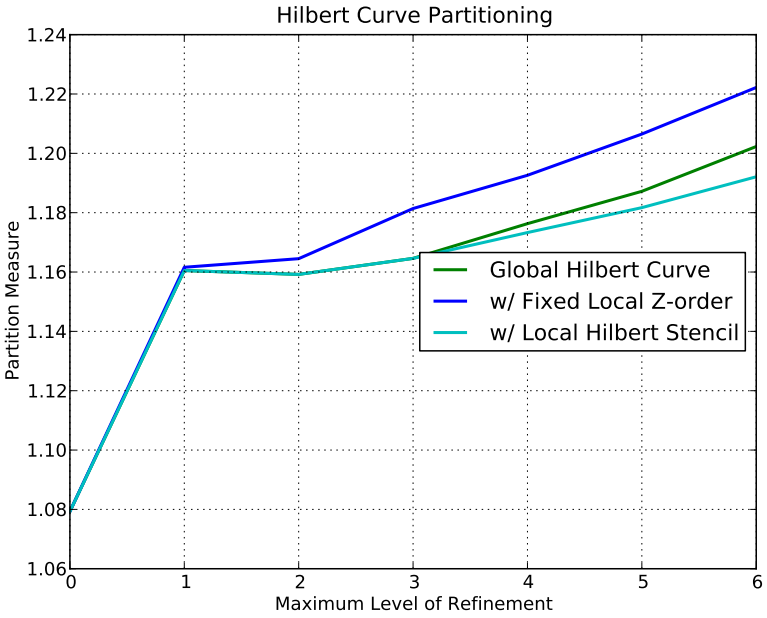

Fig. 4.2: The graph shows that the use of a local stencil has a negligible, yet positive, effect on the surface area to volume ratio.

4.2. 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 Figures 4.1 and 4.2 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 256×256 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 naïve case 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.

Further exploration of a more accurate partition measure is beyond the scope of this work. Davis et al. [5] goes into greater depth, examining space-filling curves and their use with numerical models run on the GPU. Nevertheless, the performance of

the Hilbert curve with local stencil shows a reduction in run-time significant enough to warrant the efforts to find better partitioning methods. A simple regression fit and analysis shows a roughly 50% reduction in run-time from the worst performing Z-order curve to the ideal performing Hilbert curve.

4.3. 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 4.3. The CPU version of the hash table shows a $(354.5 \pm 68.8)\times$ speed-up, while the GPU version shows a $(21901.6 \pm 5869.4)\times$ speed-up. This has prompted further research into the uses of hashing for numerical methods on GPUs and CPUs. See Robey et al. [26] for some of that work.

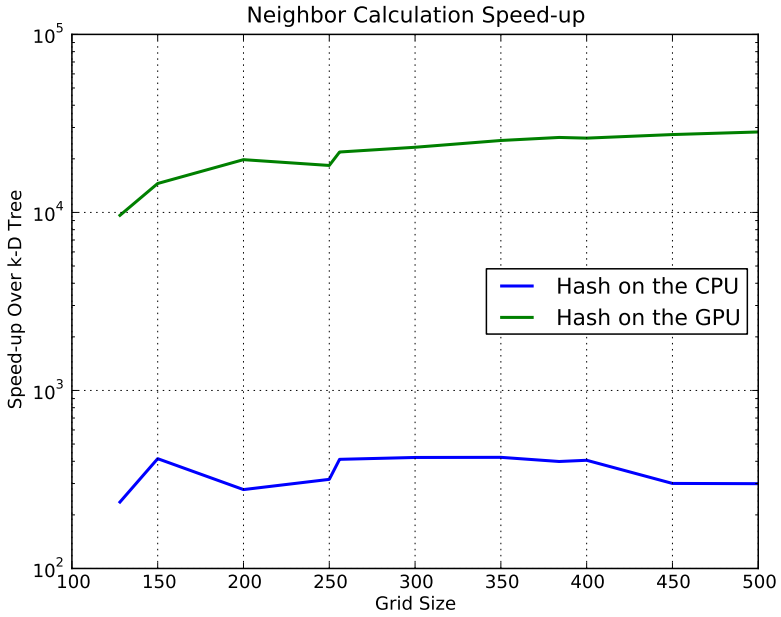

Fig. 4.3: The speed-up over a k-D tree for the hash table implementation on both the CPU and GPU.

5. Conclusion. We've shown that it is viable to implement cell-based adaptive mesh refinement on general purpose graphics processing units. We've also shown that finite differencing in this scheme requires careful consideration of the fluxes in order to ensure the conservation of the state variables. Most importantly, however, we've 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-key based method of neighbor indexing. Consistency in global reductions was implemented with the use of enhanced-precision sums. Ultimately, we've shown that GPUs can be used for intense numerical calculations while making memory and not FLOPs the primary focus in algorithm architecture. The 30-40$\times$ 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 Hilbert space-filling curve code, and Richard Barrett for the sparse MPI communication package. Also, discussions with the Applied Math Department at 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.

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] 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.

[5] Neal Davis, David Nicholaeff, and Robert Robey. Efficient locality-preserving array ordering on the gpu. *In Preparation*, 2012.

[6] Neal Davis, David Nicholaeff, Dennis Trujillo, Charles Ferenbaugh, and Robert Robey. Paradigms of exascale computing. *In Review*, August 2011.

[7] Stephen F. Davis. A simplified tvd finite difference scheme via artificial viscosity. *SIAM Journal on Scientific and Statistical Computing*, 8(1):1–18, 1987.

[8] D.L. George. *Numerical approximation of the nonlinear shallow water equations with topography and dry beds: a Godunov-type scheme*. University of Washington, 2004.

[9] 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.

[10] 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.

[11] Herman J. Haverkort and Freek van Walderveen. Locality and bounding-box quality of two-dimensional space-filling curves. *CoRR*, abs/0806.4787, 2008.

[12] H.C and Yee. Construction of explicit and implicit symmetric tvd schemes and their applications. *Journal of Computational Physics*, 68(1):151 – 179, 1987.

[13] David Hilbert. Ueber die stetige abbildung einer line auf ein flchenstck. *Mathematische Annalen*, 38:459–460, 1891. 10.1007/BF01199431.

[14] 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.

[15] 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.

[16] Khronos Group. *The OpenCL Specification*, version 1.1, revision 44 edition, June 2011.

[17] Donald E. Knuth. *The art of computer programming. Vol. 1, Fundamental algorithms*. Addison-Wesley Pub. Co., 3 edition, July 1997.

[18] L. D. Landau and E. M. Lifshitz. *Fluid Mechanics, Second Edition: Volume 6*. Course of theoretical physics. Butterworth-Heinemann, 2 edition, January 1987.

[19] Peter Lax and Burton Wendroff. Systems of conservation laws. *Communications on Pure and Applied Mathematics*, 13(2):217–237, 1960.

[20] Randall LeVeque. Clawpack v4.6.1 documentation: Amr refinement strategy. `http://depts.washington.edu/clawpack/users/amrclaw/amr_strategy.html`, 2009.

[21] R.J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge texts in applied mathematics. Cambridge University Press, 2002.

[22] G.M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., 1966.

[23] NVIDIA. Opencl best practices guide, May 2010.

[24] NVIDIA. Opencl programming guide for the cuda architecture, version 3.2, Aug. 2010.

[25] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890. 10.1007/BF01199438.

[26] Rachel N. Robey, David Nicholaeff, and Robert W. Robey. Hash-based algorithms for dis-

cretized data. *In Preparation*, 2012.

[27] Robert W. Robey, Jonathan M. Robey, and Rob Aulwes. In search of numerical consistency in parallel programming. *Parallel Comput.*, 37:217–229, April 2011.

[28] Gilad Shainer, Ali Ayoub, Pak Lui, and Tong Liu. Raising the speedlimit: New gpu-to-gpu communications model increases cluster efficiency. `http://www.scientificcomputing.com/articles-HPC-GPU-Raising-the-Speed-Limit-010711.aspx`, January 2011.

[29] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM J. Numer. Anal.*, 21(5):995–1011, 1984.

[30] Alvin Trivelpiece. Exascale Workshop Panel Meeting Report. `http://www.er.doe.gov/ascr/Misc/GrandChallenges.html`, January 2010.

[31] 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.