CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–14
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Comparison of GPU Strategies for Unstructured Mesh Physics

Charles R. Ferenbaugh*

Los Alamos National Laboratory, Los Alamos, NM 87544

SUMMARY

There have been few efforts to date to write physics algorithms for general unstructured meshes (meshes composed of arbitrary polygons/polyhedra) on graphics processing units (GPUs). Typical strategies for GPU memory management, such as double-buffering and coalescing memory accesses, are difficult to apply to the irregular memory storage patterns of unstructured meshes. This paper presents results from an initial GPU version of a typical unstructured mesh kernel. Three different memory management strategies are described and implemented. Timing results for all three strategies are presented, in some cases showing speedups of over 20x compared to the original CPU code. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: advanced architectures; GPU; unstructured mesh

1. INTRODUCTION

The high-performance computing world is experiencing a major transition in its computer architectures. New architectures such as the IBM Cell Broadband Engine (CBE) and graphics processing units (GPUs) are becoming common in supercomputer clusters, continuing the trend started by systems such as Roadrunner at Los Alamos National Laboratory (the first system to break the petaflop barrier [1, 2]); and Tianhe-1A in Tianjin, China (the #1 supercomputer on the Top 500 list as of November 2010, and still at #2 a year later [3, 4]). Architectures such as these provide high computational performance combined with low power usage, and are likely to be used in future systems such as the "exascale" systems being discussed in the international HPC community [5].

These systems will pose significant challenges to all scientific software developers, introducing new programming models to manage the increased hardware complexity, and requiring major rewrites of existing software. They will be particularly challenging for developers of algorithms for general unstructured meshes, that is, meshes containing arbitrary polygons (in 2D) or polyhedra (in 3D). Common memory management techniques such as double-buffering and data prefetching [6, 7] cannot be easily applied to the irregular memory storage patterns of unstructured meshes. Also, the

*Correspondence to: Los Alamos National Laboratory, Mail Stop B295, Los Alamos, NM 87544. Email: `cferenba@lanl.gov`.

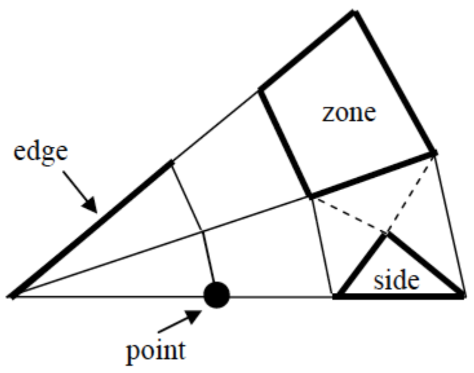

Figure 1. FLAG terminology for mesh entities.

indirect addressing instructions needed by connectivity array lookups are often very inefficient. As a result, unstructured mesh methods have had relatively few implementations on advanced architectures to date.

Many physics applications at Los Alamos and elsewhere are based on general unstructured mesh algorithms. Often these are also staggered-mesh algorithms: they compute some mesh variables on points, and others on zones, with frequent lookups done between the two. Because these algorithms are so common and so useful, it makes sense to ask whether they can be implemented efficiently on advanced architectures. I presented initial results for a CBE implementation of one LANL physics kernel in [8]. In this paper I will apply similar ideas to implementing the same kernel on a GPU.

There has been other work done on unstructured mesh physics on GPUs, but in most cases attention is limited to a specific subset of possible cell types. For instance, in [9] and [10], only tetrahedral mesh cells are used, while in [11] only so-called “zoo meshes” (tetrahedra, pyramids, prisms, and hexahedra) are supported. In [12], a structured mesh is used as a base, but degenerate points and zones are allowed. In approaches such as these, the implementation usually relies on the connectivity list for an element either having fixed size (e.g., four vertices per zone for a tet mesh) or having a fixed upper bound (e.g., at most eight vertices per zone for a zoo mesh in 3-D). As a result these approaches cannot be directly extended to the general case.

I am aware of only two instances of work applicable to general unstructured meshes. In [13], unstructured meshes are handled by casting each physics update step as a single sparse matrix-vector solve. And in [14], a domain-specific language (DSL) is developed for unstructured mesh algorithms on a variety of platforms, including GPUs. This paper presents a more direct approach.

2. THE FLAG PHYSICS APPLICATION

FLAG is a radiation-hydrodynamics application used extensively at Los Alamos for simulations in materials science, high explosives, and plasma physics [15]. It runs on unstructured finite-volume meshes with arbitrary polygons (or polyhedra) as zones. The general FLAG code runs problems in one, two and three dimensions, and uses MPI to run in parallel on networked computing clusters. In this study, I restricted my attention to two-dimensional, serial problems.

Table I. Basic data flow for FLAG hydrodynamics.

step	main inputs	main outputs
Update mesh, predictor step	point velocity, position	point position (half-advanced)
Update mesh geometry	point position	**side surface vector; side and zone volume**
Update zone state variables	zone volume	zone density, pressure, sound speed
Compute basic hydro forces	side surface vector; zone pressure	side force
Sum forces to points	side force	**point force**
Compute artificial viscosity	point velocity, position; zone density, sound speed	**side force**
Sum forces to points	side force	**point force**
Update mesh motion	point force	point acceleration, velocity
Update mesh, corrector step	point velocity, position	point position (fully advanced)
Update mesh geometry	point position	**side and zone volume**
Update zone state variables	zone volume	zone density, energy, pressure, temperature

The terminology for the FLAG mesh data structure in two dimensions is shown in figure 1. Points, edges, and zones are straightforward. Each zone is split into *sides*; these are triangles formed from the centroid of the zone and the two endpoints of one edge. The data structure includes mapping arrays for side-to-zone and side-to-edge (one-to-one), and for side-to-points and edge-to-points (one-to-two). To support arbitrary polygons as zones, there are no mappings from points or zones to any other entity, since such mappings would need to return a variable number of results.

The FLAG hydrodynamics algorithms use a staggered-mesh scheme, in which positions and velocities are defined on mesh points, while most state variables such as density, internal energy, and pressure are defined on zones. A few variables are defined on edges. For ease of computation, many variables defined on points or zones are first computed on sides, and then summed to the points or zones as appropriate.

Because of the staggered-mesh scheme, FLAG algorithms frequently need to compute a variable defined on one type of mesh entity, using variables defined on different entities. For example, a basic subset of the hydro steps in FLAG is shown in Table I. Note that, of the eleven steps listed in this simplified outline, five involve lookups of point data to compute zone/side data, or vice-versa (these are marked in **bold** type). As more details of the physics are added, many more steps with lookups take place.

For this study, I isolated the FLAG *artificial viscosity* (AV) kernel and implemented it on the GPU. Artificial viscosity is a fictitious term commonly introduced into fluid flow equations to correctly handle shock regions with large discontinuities in the problem state variables. Several different artificial viscosity treatments are implemented in FLAG; for this study, only the Barton [16] and TTS [17, 18] methods are used. These routines account for about 30-40% of the total runtime in the test problems profiled in this study. In the FLAG code base, these routines contain about 900 lines of Fortran source code; I reduced this number to about 500 by eliminating some code paths that were not needed for this set of problems. (Note that the full FLAG code base contains about 470,000 total lines of source code.)

In the Barton treatment in its simplest form, a force is computed for each side based on the velocity difference between the two mesh points on that side. The TTS treatment is more complex: the force is calculated by computing a separate density, pressure, and force for each side within the zone. Both of these methods include side (or zone) variables together with point variables in the formula for viscosity q_s of a side s:

$$q_s^{\text{Barton}} = \rho_z \frac{\mathbf{u}_{12} \cdot \mathbf{x}_{12}}{|\mathbf{x}_{12}|} \times (\text{other terms} \ldots) \tag{1}$$

$$q_s^{\text{TTS}} = \rho_s \frac{\mathbf{x}_{12} \cdot \mathbf{s}_s}{|\mathbf{s}_s|} \times (\text{other terms} \ldots) \tag{2}$$

Here ρ_z is the zone density; ρ_s is the side density; $\mathbf{u}_{12}$ and $\mathbf{x}_{12}$ are the differences in point velocity and position, respectively, between the two points in a side; and $\mathbf{s}_s$ is the side surface area vector. Several other terms in each equation are omitted here, since they are complicated and the details are not relevant to this paper. The given terms are enough to illustrate the main point: in order to compute new values for side variables, it is necessary to do some lookups of point variable values.

3. GPU-SPECIFIC OPTIMIZATIONS

3.1. Target Platform and Programming Model

The GPU versions of the AV kernel were written in the NVIDIA CUDA language [7], a C-like language with extensions to efficiently utilize the parallel compute engine of a GPU. Because CUDA is built on C, it was not difficult to write the kernels in CUDA and call them from the main FLAG code, written in Fortran and C.

The CUDA programming model makes a distinction between the *device*, a coprocessor running compute-intensive calculations (the GPU, in our case), and the *host* running the main code (the CPU). The host/device terminology will be used frequently in this paper for describing how algorithms are implemented and data is moved.

The target platform was the *Darwin* cluster at LANL. The nodes of this cluster have AMD Opteron 6168 CPUs running at 1.90 GHz, and the nodes used for this study have attached NVIDIA Tesla C2050 GPUs (CUDA compute capability 2.0). GPUs of this model contain 14 *streaming multiprocessors* (SMs) to do the compute work. Each SM has hardware to process a *warp*, or 32 threads, in parallel, and with fast context switching can keep up to 1536 threads in progress at once. (The thread limit for any specific kernel may be lower than 1536 due to register pressure or memory limits). The full GPU has a global memory of about 2.6 Gb, and each SM has a fast-access shared memory space of up to 48 Kb. The C2050 also has two features not found in earlier NVIDIA architectures: two separate DMA engines to support overlapped writes to and from the device, and L1 and L2 hardware caches to cache global memory accesses.

3.2. Parallelism

Since the GPU requires a large number of small threads to run in parallel, the first major design decision for the GPU implementation must be a choice of the basic computational unit to assign to a thread. Most of the computation in the AV algorithms consists of loops over sides, so a

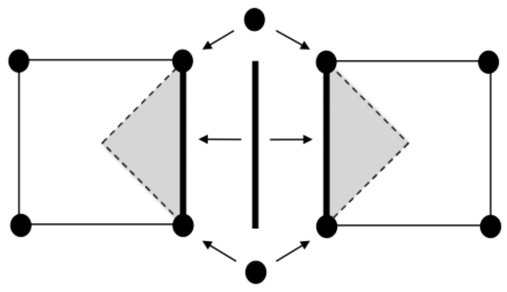

Figure 2. Data replication of point and edge data onto sides, to support chunk-by-chunk processing on the GPU.

natural choice is to assign one thread per side. This choice makes loops over sides straightforward. Loops over zones are slightly more complicated: the threads representing different sides in the zone must coordinate their computations. Coordination between sides was made easier by imposing the requirement that all sides in one zone must be assigned to the same CUDA block. This allows the use of the CUDA `__syncthreads()` command and similar synchronization mechanisms within each zone.

Loops over points and edges can be handled by replicating the point and edge data onto sides, and rewriting the loops as side loops. The data replication is described in more detail in the following sections.

An alternate option for threading would have been to assign one thread per zone. This would make both zone and side loops easier to implement. However, it would also reduce the amount of available parallelism by a significant factor; in a typical FLAG problem, most zones are four-sided, so the total numbers of CUDA threads and blocks would decrease by a factor of four. For typical problem sizes, this would lead to a significant performance difference (see section 4.2 for more details); so the thread-per-zone option was not pursued.

3.3. Data Movement

Data movement between the CPU and GPU can be quite expensive; to minimize this cost, I use the common technique of dividing the problem into chunks and overlapping data movement with computation. Normally this is done by moving a chunk of input data to the GPU, computing on it, and moving the corresponding chunk of output data back to the CPU. For side data, this is straightforward, since each thread processes exactly one side. Zone data can also be processed easily: since each side is in exactly one zone, the data can be ordered so that each zone has all of its sides consecutive in the side list. As a result, each contiguous chunk of zones in memory corresponds to a contiguous chunk of sides.

Point data is more complicated to process: since each side contains two points, and each point may be part of more than one side, there is no way in general to make contiguous chunks of points correspond exactly to chunks of sides. Edge data poses similar difficulties. Two different solutions are used here. The first is to have the host replicate the point and edge data so that each side has its own copy, as shown in figure 2. The side-based replicated data can then be coalesced into chunks that correspond exactly to side chunks and can be moved as needed between host and device. This operation is called a *scatter* when applied to input data that is moved to the device before it is used for computation, or a *gather* when applied to output data that is retrieved from the device after the computation is complete. (This is the same strategy used on the CBE in [8].)

A second possible approach to point and edge data is to relax the condition on when it is moved between host and device. In this approach, input point data is moved to the device before computation begins on the *first* side chunk that reads it, while output point data is moved back to the host after processing completes on the *last* side chunk that writes it. Note that this strategy only works because the GPU global memory is large enough ($>$2 Gb) to hold entire point arrays; it would not be viable on an architecture such as the CBE with limited data storage (256 Kb) on the accelerator.

For either of these strategies, the basic structure of the main processing loop is similar. The main computation kernel runs on the device, and this computation overlaps with data transfers between host and device. In addition to all of this, some computation is performed on the host: managing the data movement and kernel execution of different chunks; reordering data between the host code format and a more GPU-friendly layout; and performing gathers and scatters of point- and edge-based data, if applicable.

The CUDA programming model allows for *streams* of computation and data movement steps to run asynchronously and overlap with one another. Various synchronization commands can be used to ensure that operations in any single stream are done in the correct order, without disrupting operations in other streams. The expanded functionality of newer NVIDIA GPUs with higher compute capabilities is particularly useful here. All NVIDIA GPUs support concurrent execution on host and device. With compute capability 1.1 and higher, both of these may be overlapped with data movement in one direction (host to device, or device to host). With compute capability 2.0 and higher (as found on the Tesla C2050 used for this study), data movement in both directions at once may be overlapped with the host and device execution. For the AV kernels, it has proved most efficient to run with four separate streams, one for each of the four types of processing (host compute, device compute, and both directions of data movement) that can run at the same time.

3.4. Memory Locality

The GPU memory systems are designed for efficient access of contiguous locations in memory: the threads in a warp can read or write to contiguous addresses much faster than to addresses spread out over the memory space. This is unfortunate for unstructured mesh kernels like the AV kernel, since the indirect addressing needed to use point and edge connectivity arrays generally leads to spread-out, inefficient access patterns. This is not an issue if the data has been gathered or scattered on the host, as described in the preceding section; but if the data is in its original ordering, something else must be done to improve the efficiency.

One simple solution is to reorder the data on the device, so that the reordered data can always be accessed in sequence (except during the reordering step itself). This is done using the same scatter and gather operations described above, but now performed on the device instead of the host. Another solution is to take advantage of the GPU hardware caches on the C2050; these can provide fast access to a global memory location if it (or another location in its cache line) has already been accessed recently. The caches can be particularly effective when the zone and point numbering schemes have at least some correlation between them, which is usually the case in typical FLAG meshes. The GPU texture cache could also be used for this purpose, at least for read-only data; however this provides only about 6-8 Kb of storage per SM, as opposed to the L1 and L2 caches which can respectively provide 48 Kb per SM and 768 Kb globally. Because of its smaller size, the texture cache was not used in this study.

To test the various solutions proposed in this section and the preceding one, three different strategies for handling input point data have been implemented and tested:

1. **CPU coalescing**: Point data arrays are scattered to side-based arrays on the CPU; the side-based arrays are then transferred to the GPU and loaded in shared memory.
2. **GPU coalescing**: Point data arrays are transferred from CPU to GPU; the GPU then scatters them into contiguous locations in shared memory.
3. **Hardware cache**: Point data arrays are transferred from CPU to GPU; the GPU then accesses them directly from global memory, using values from the hardware L1 and L2 caches whenever possible.

Output point data is mostly handled in the same way, but with an extra complication: a reduction operation must be used to combine the data from multiple sides to a single point. (In the AV kernels, this is always a summation, but other reduction operators would work equally well.) For the CPU-coalesce strategy, this is easy to incorporate in the gather step. But for the others, different side threads must often write to the same point array element, and these writes must be synchronized in some way.

For the GPU-coalesce strategy, CUDA does provide atomic memory operations that could be used to avoid synchronization issues; but these were tried and found to be quite expensive. So instead, a new CUDA kernel was written to do the gather operation by itself, using one thread per point instead of one per side. Each thread gathers all of the side information for its point; since no other thread is working on the same point, no synchronization between threads is needed. Finally, for the hardware cache strategy, the caches are not coherent, so they cannot be used to gather output point data (unless atomic operations are used, which are too expensive). Instead, the same point-based gather kernel described above was re-used.

4. RESULTS

Table II describes the test problems used for this study. All of these simulate a shock wave moving through a volume of ideal gas, with various mesh shapes and initial conditions. They are adapted from standard problems that are run regularly in the FLAG regression test suite.

All tests were run on the LANL *darwin* cluster, described earlier.

4.1. Basic Timing Results

The first tests done were simple timing tests comparing the three GPU kernels with the baseline CPU-only kernel. Results are shown in Table III and Figure 3.

For all of the problems tested, the CPU-coalesce strategy provided some speedup, but not as much as the other two strategies. Some of the reasons for this difference will be explored further in

Table II. Test problems used in performance studies

test name	# zones	# sides	# timesteps	mesh type
sedov	32400	129600	2065	square, all quad zones
tube	36864	147456	2123	rectangular, all quad zones
noh	6000	23940	27265	radial, tri and quad zones
sedovpoly	22801	136202	3127	square, mostly hexagonal zones

Table III. Timing data for the three GPU AV kernels, and the serial CPU baseline, on various test problems.

	runtime (s)			
test name	baseline CPU	CPU coalesce	GPU coalesce	GPU cache
sedov	134.52	12.47	6.10	6.10
tube	161.14	15.23	7.56	7.59
noh	315.06	46.21	32.35	32.65
sedovpoly	222.36	22.35	11.08	11.24

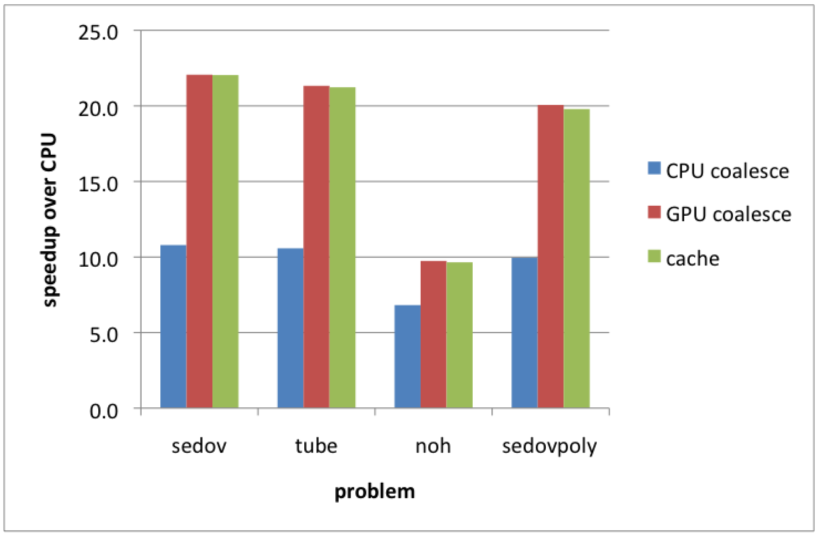

Figure 3. Speedups for the three GPU AV kernels, computed relative to the baseline CPU kernel.

section 4.4 below. The GPU-coalesce and cache strategies, meanwhile, had nearly indistinguishable timing results. This suggests that the cache strategy provides, in hardware and with minimal code change, similar performance benefits to the GPU-coalesce strategy which requires hand-coded software to perform the coalescing.

The *sedov*, *tube*, and *sedovpoly* problems had speedups of over 20x when run with either of the latter two strategies. The *noh* problem had a smaller speedup, closer to 10x. This is probably because *noh* has a smaller number of zones and sides than the others. As shown in the next section, the GPUs were most efficient when processing chunks of about 20-30K sides (usually about 5-7.5K zones) at a time. So the *noh* problem, with only 23K sides and a single chunk, was not able to perform as well on the GPU as the others with >120K sides and multiple chunks.

Note also that the *sedovpoly* problem uses a truly general unstructured mesh, unlike the other test problems which are based on structured meshes. This problem showed a speedup of about 20x

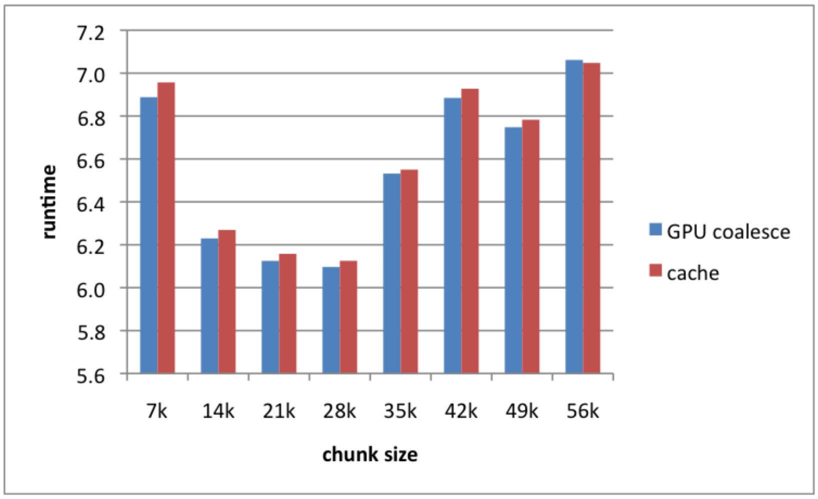

Figure 4. Timings of GPU AV kernels with varying values for processing chunk size. Runtimes are in seconds. Chunk sizes are computed as multiples of 7K (number of threads needed to fill the GPU).

over the CPU baseline when run using the GPU-coalesce and cache strategies. The speedup is only slightly smaller than that of the *sedov* and *tube*, problems that are based on structured meshes and have similar numbers of sides to *sedovpoly*.

4.2. Optimal Chunk Size

As described in section 3.3, the AV kernels use the common approach of dividing problem data into chunks to allow for overlapping of data movement and computation. The chunk size must be chosen carefully for best performance. A small chunk size may not be sufficient to keep the GPU fully occupied, and may also lead to inefficient data transfers because of the overhead of sending many small messages. Meanwhile, a large chunk size leads to longer startup (and shutdown) times as the chunk pipeline is filled (and emptied). To find the best chunk size for the AV kernels, the *sedov* problem was run using several different chunk sizes; the resulting timings are shown in Figure 4.

The C2050 contains 14 SMs, each of which can run up to 512 threads of an AV kernel at once. (This is lower than the general limit of 1536 threads/SM, due to heavy register usage by the AV kernel.) So the entire GPU can support $14 \times 512 = 7168$, or 7K, threads running simultaneously. Clearly, in order to use the GPU efficiently, there should be at least this many threads in flight for as much of the runtime as possible; so only multiples of 7K were tested as chunk sizes.

The results show that the most efficient chunk size was 28K, with 21K a close second. These chunk sizes correspond to enough threads to fill the GPU four (for 28K) or three (for 21K) times. Apparently this number of threads gives fairly efficient data transfers, while keeping all of the elements of the pipeline busy enough for good performance. Note also that, since processing on four different chunks can be overlapped, the code is most efficient when running a problem with at least four chunks (and preferably more). The *sedov* problem tested here contained either five or seven chunks (for chunk size 28K or 21K, respectively), resulting in a good overlap and good performance.

Based on these results, the 28K chunk size was used for all of the other timing results shown in this section.

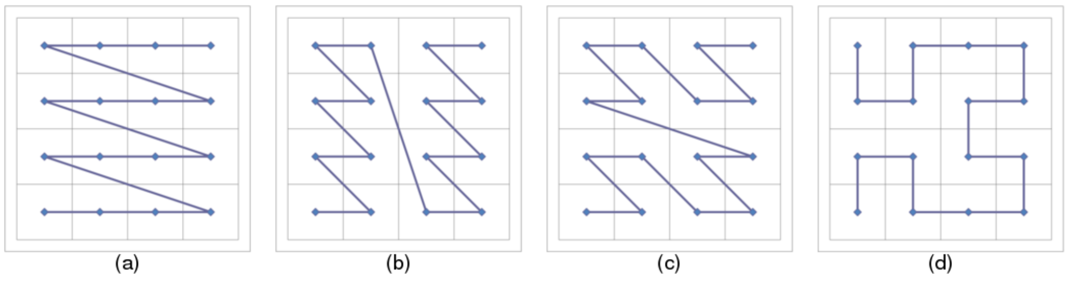

Figure 5. Examples of mesh numbering schemes used to test memory locality: (a) simple row/column; (b) column blocks with size 2; (c) z-curve; (d) Hilbert curve.

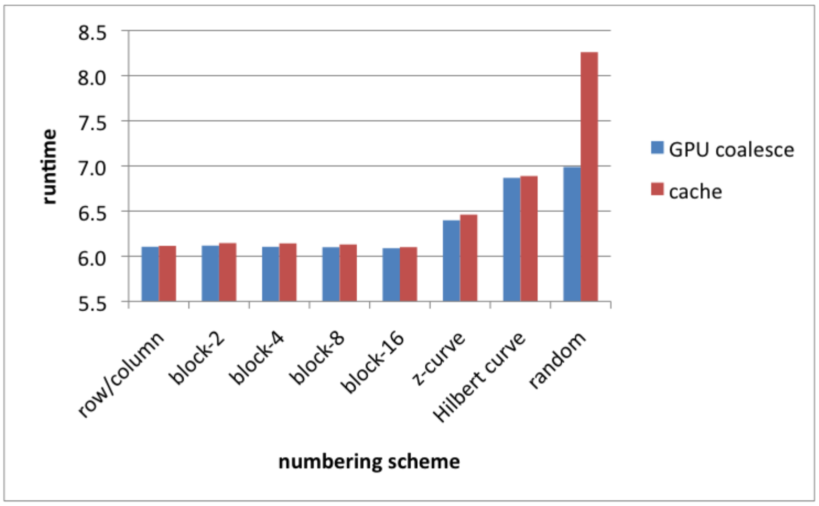

Figure 6. Timings of GPU AV kernels, using various mesh numbering schemes. Runtimes are in seconds.

4.3. *Memory Locality and Mesh Numbering*

As noted earlier, the memory system for NVIDIA GPUs is designed so that the threads in any warp have fast access to contiguous memory locations; if the memory locations are non-contiguous, access is still possible but much slower. To better understand the effects of memory locality on the AV kernel, I modified the FLAG code to allow for several different mesh numbering schemes with varying degrees of locality. For simplicity, these were implemented only on square meshes. The schemes used were:

- simple row/column numbering (the default for rectangular meshes in FLAG)
- numbering columns by blocks, with various user-selected block sizes
- space-filling curves
- random numbering

Figure 5 shows some examples of these.

I used only the *sedov* problem for this experiment, since it is the only one of the test problems which uses a regular square mesh. I ran this problem using each of the numbering schemes, and using both the GPU-coalesce and hardware cache strategies. Timing results for these runs are shown in Figure 6.

The timings for the various blocked numbering schemes were not noticeably different from the default row/column scheme. More surprisingly, the two schemes using space-filling curves ran roughly 5-10% slower than the default scheme. This is somewhat unexpected, since other kinds of GPU codes have used space-filling curves for similar purposes with good results.

However, for the AV kernel, the chunking algorithm used to determine chunks of points is dependent on the *maximum* difference between numbers of adjacent zones and points. Recall from section 3.3 that, when a point is adjacent to multiple sides, its data is moved to the GPU with the lowest-numbered adjacent side, and moved back to the CPU with the highest-numbered adjacent side. So this algorithm would work best with a numbering scheme that minimizes the maximum difference between adjacent sides. The space-filling curves are good for minimizing the *typical* difference, but not the maximum.

Note for instance, in Figures 5(c) and (d), that the two center zones on the leftmost boundary are separated by distance 6 and 13, respectively. As the mesh grows from 16 zones to larger sizes with n zones, this difference will grow as $O(n)$. By contrast, with the row/column and column block schemes, the maximum difference between adjacent zones anywhere in the mesh will be at most $O(\sqrt{n})$, leading to a more efficient division of the problem into point data chunks.

Finally, the random numbering gave the slowest runtimes (as expected), and was the only one for which the GPU-coalesce and cache strategies showed a significant performance difference. This is probably because the cache strategy reads many global memory values multiple times, as opposed to the GPU-coalesce strategy which reads each global memory value only once. With a random numbering scheme, there would be a significant amount of cache thrashing, repeated many times in each kernel call.

Since this experiment was done only on square meshes, it is natural to ask how these numbering schemes might be extended to more general unstructured meshes. It would not be difficult to construct a numbering for arbitrary meshes with properties analogous to the row-column numbering scheme for square meshes. The geometric domain of the mesh would be split into horizontal strips approximately one zone high. Then for each strip in turn, its zones would be enumerated in something approximating a left-to-right order. In a similar fashion, analogues to the column block scheme could be constructed. Given the timing results in this section, I would expect these schemes to give good performance.

It would be much more complicated to extend the definitions of the space-filling curves to handle arbitrary meshes. Fortunately, the timing results suggest that this effort is not necessary.

4.4. Isolating Kernel Functions

As described in section 3.3, the GPU AV kernels normally run with four different steps running in parallel: CPU computation, GPU computation, and data movement to and from the GPU. In order to understand which of these steps were the limiting factor for run times, I ran the *sedov* problem on the three GPU kernels with the asynchronous processing disabled, so that the steps were run serially instead of overlapping. Results of these runs are shown in Figure 7.

For all three of the kernels, the limiting factor proved to be the CPU computation time. This is particularly true when using the CPU-coalesce strategy, since a signficant amount of time is spent on the CPU scatter and gather steps. Since these steps run exclusively on the CPU, and do not

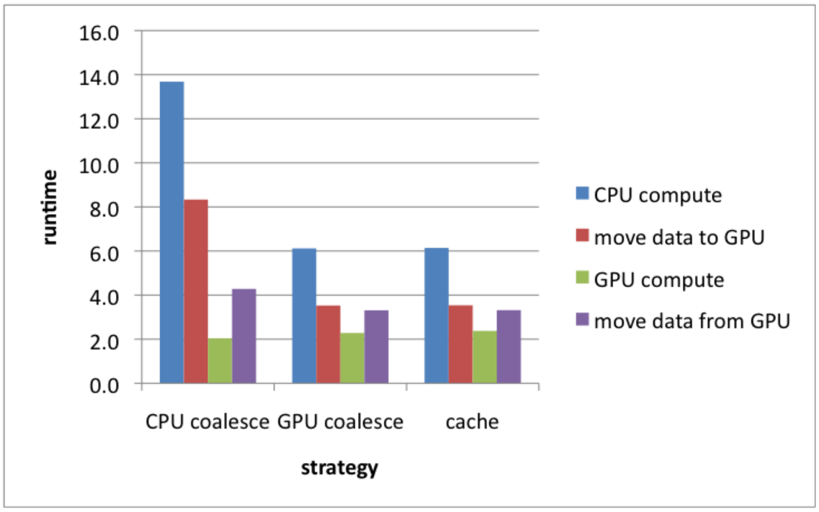

Figure 7. Timings of GPU AV kernels, with different kernel steps isolated and run serially. Runtimes are in seconds.

scale with increasing numbers of accelerator cores, this strategy is not likely to be useful on future architectures.

For the other two strategies, the amount of CPU computation was smaller but still the dominant factor. The CPU computation in these cases consisted of managing the GPU and the data movement, and reordering data between the host code format and a more GPU-friendly layout. In the current code, it would not be feasible to move these steps to the GPU; and it would be difficult to make them run much faster on the CPU. However, some steps could be taken to reduce runtimes on future architectures and for other unstructured mesh codes; these will be discussed in the final section.

4.5. Implementation Notes

The original CPU version of the AV kernel contained about 500 lines of Fortran source code, as noted earlier. The three GPU versions contain between 750 and 880 CUDA source lines apiece, and about 30 lines of utility code were shared between all three versions. This represents an increase of about 50-80% in length over the CPU kernel. This increase was mainly due to added CPU functionality needed for GPU computing, such as device memory allocation, data transfer, and computation of auxiliary data structures to support chunk processing. If multiple kernels were running on the GPU, it is likely that much of this code could be shared, and the 50-80% figure would be reduced.

5. CONCLUSIONS AND FUTURE DIRECTIONS

This paper has described and compared some initial strategies for implementing general unstructured mesh algorithms on a GPU. One of the strategies, CPU-coalesce, has performance that lags the other two strategies, and with its heavy CPU usage is unlikely to scale with the processing power of future GPU architectures. The other two strategies, GPU-coalesce and hardware cache, show promising initial performance, with a speedup of over 20x over the original serial CPU kernel.

These are likely to be useful for other algorithms and architectures in the future, if some of the remaining scaling issues can be resolved.

In the case of the current AV kernels, a major contributor to host compute time is data reformatting between the FLAG format and a layout optimal for the GPU. Changing the FLAG data structures is beyond the scope of the current study; but in another context where the host data structures are device-friendly (or can be made so), data conversion would be unnecessary and host compute time would be correspondingly lower.

Alternately, if the algorithm structure can be changed from an accelerator-offload model to a more GPU-centric approach, more optimizations are possible. This was not an option for the current study, given the large size of the FLAG code base; but with a smaller code base, or a longer timeframe, such a rewrite could be feasible. When most of the computation is done on the device, the time spent transforming data between host and device formats can be reduced or even eliminated. This approach can also reduce the need for data movement between host and device, leading to further performance improvements.

It may also be possible to improve algorithm performance by taking advantage of future hardware and software developments. For instance, data movement times could perhaps be improved if the GPU had a software cache similar to that found on the CBE [6]. Implementing this would probably require hardware improvements, since data transfers would need to be initiated on the device side, and current GPUs do not support this. Also, some GPU designers have mentioned the idea of an integrated memory space shared by host and device. If such a memory space were available, the need for data movement would be eliminated entirely; in addition, some steps that are currently done by the host, such as data reordering, could be moved to the device instead.

These are several possible ways to implement general unstructured mesh efficiently on GPUs and other similar architectures, taking advantage of the great performance gains that are possible. Strategies such as these can allow the existing base of unstructured mesh codes to be carried forward, and also allow for development of new codes, as the next generation of architectures becomes more widely available.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the US Department of Energy National Nuclear Security Administration by Los Alamos National Security, LLC, at Los Alamos National Laboratory, under contract DE-AC52-06NA25396. The author gratefully acknowledges the support of the US DOE NNSA Advanced Simulation and Computing (ASC) Program.

I would also like to thank Konstantin Lipnikov, Eric Nelson, Mikhail Shashkov, and the anonymous referees for providing comments on earlier drafts of this paper; and Rao Garimella for help in generating polygonal meshes.

References

1. Barker KJ, Davis K, Hoisie A, Kerbyson D, Lang M, *et al.*. Entering the petaflop era: The architecture and performance of Roadrunner. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
2. Grice D, Brandt H, Wright C, McCarthy P, Emerich A, *et al.*. Breaking the petaflops barrier. *IBM Journal of Research and Development* 2009; **53**(5):1:1–1:16.
3. Raman M. China makes world's fastest supercomputer. *International Business Times*, October 28, 2010.

4. Meuer H, Strohmaier E, Dongarra J, Simon H. Top 500 supercomputing sites. `http://www.top500.org`.
5. Messina P, Brown D, *et al.*. Crosscutting technologies for computing at the exascale. Report from DOE "Crosscutting Technologies for Computing at the Exascale" workshop, Washington, DC February 2010. Available at `extremecomputing.labworks.org/crosscut/report.stm`.
6. Arevalo A, Matinata RM, Pandian M, Peri E, Ruby K, *et al.*. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM International Technical Support Organization, 2008. Available at `www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf`.
7. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 2011. Available at `http://developer.nvidia.com/nvidia-gpu-computing-documentation`.
8. Ferenbaugh C. An efficient approach to unstructured mesh hydrodynamics on the Cell Broadband Engine. *Proceedings of the Nuclear Explosives Code Developers' Conference 2010*, to appear.
9. Klöckner A. High-performance high-order simulation of wave and plasma phenomena. PhD Thesis, Brown University 2010. Available at `http://mathema.tician.de/academic/research`.
10. Corrigan A, Camelli FF, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 2011; **66**(2):221–229.
11. Komatitsch D, Erlebacher G, Goddeke D, Michea D. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics* 2010; **229**:7692–7714.
12. Singh G. Mimetic finite difference method on GPU: Application in reservoir simulation and well modeling. Master's Thesis, Norwegian University of Science and Technology, Department of Mathematical Sciences 2010. Available at `http://daim.idi.ntnu.no/masteroppgave?id=5668`.
13. Markall G. Accelerating unstructured mesh computational fluid dynamics on the NVidia Tesla GPU architecture. Master's Thesis, Imperial College London 2008. Available at `www.doc.ic.ac.uk/~grm08/gmarkall_isol.pdf`.
14. DeVito Z, Joubert N, Palacios F, Oakley S, Medina M, *et al.*. Liszt: A domain specific language for building portable mesh-based PDE solvers. *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, 2011.
15. Burton D. Consistent finite-volume discretization of hydrodynamics conservation laws for unstructured grids. *Technical Report UCRL-JC-118788*, Lawrence Livermore National Laboratory, Livermore, CA 1994.
16. Margolin L. A centered artificial viscosity for cells with large aspect ratios. *Technical Report UCRL-53882*, Lawrence Livermore National Laboratory, Livermore, CA 1988.
17. Wallick KB. Temporary triangular subzoning (TTS), in REZONE: A method for automatic rezoning in two-dimensional Lagrangian hydrodynamics problems. *Technical Report LA-10829-MS*, Los Alamos National Laboratory, Los Alamos, NM 1987.
18. Caramana E, Shashkov M. Elimination of artificial grid distortion and hourglass-type motions by means of Lagarangian subzonal masses and pressures. *Journal of Computational Physics* 1998; **142**:521–561.