Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Load-Balanced Isosurfacing on Multi-GPU Clusters

Steven Martin[1], Han-Wei Shen[1], Patrick McCormick[2]

[1] The Ohio State University
[2] Los Alamos National Laboratory

Abstract

Isosurface extraction is a common technique applied in scientific visualization. Increasing sizes of volumes over which isosurfacing is to be applied combined with increasingly hierarchical parallel architectures present challenges for efficiently distributing isosurfacing work loads. We propose a technique that, with a modest amount of preprocessing, efficiently distributes isosurfacing load to GPU compute resources within a cluster. Load uniformity is maximized over a set of user-defined isovalues, enabling improved scalability over naive, non-data-centric, work distribution approaches.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics—Isosurface generation

1. Introduction

Isosurface extraction is a common technique applied in scientific visualization. Isosurfaces are often rendered to show structures indicated by surfaces over which a particular value is uniform. Additionally, it is often of use to have the triangle data of these surfaces available for the computation of quantities such as surface area. In many cases, a user has an idea of what isovalue ranges may be reasonable for the extraction of features of interest, but may not know exactly what isovalues should be used. Thus, providing fast isosurfacing of a particular subset of potential isovalues can be of particular utility.

As scientists have sought to increase simulation accuracy, the quantity of data produced has increased commensurately. Analysis tools, including those that provide isosurfacing, must scale to support this increased volume of data.

Over recent years, a transition has been seen toward hierarchical parallelism, both in terms of memory and processors. Even single PCs often contain multiple CPUs and GPUs, with each GPU containing multiple stream processors. Clusters add an additional level within the hierarchy. Challenges are introduced not only by the hierarchical nature of the compute resources, but also by the diversity of interconnects between them.

Making the most of these compute resources requires deciding the levels within the hierarchy at which subdivision of work and data-dependent distribution of work is appropriate. Several considerations must be made:

- Hardware constraints: Limits are often imposed on the local memory available in different elements of the compute resources, and there are often substantial disparities between processor speed, available local memory, and interconnect speed.
- Required result constraints: Results from an isosurfacing algorithm should be in a format appropriate for how they will be used. For example, triangles from an isosurfacing algorithm should be stored in a buffer with an appropriate format for rendering, if rendering is required.
- Preprocessing cost: The resources consumed, both in time and space, by preprocessing must be warranted by the expected gains in usability as a result.
- Efficient scalability: Algorithms must scale well with increased data size and compute resources, while also having reasonable absolute speeds for the range of expected target data sizes and systems.

We propose an approach, exhibited in figure 1, that evenly distributes isosurfacing work to multiple GPUs in a cluster, taking into consideration user-defined salient isovalue ranges. The approach then applies our efficient parallel isosurfacing algorithm on each GPU. A modest amount of preprocessing enables efficient distribution of work.

This paper is organized as follows. Section 2 describes re-

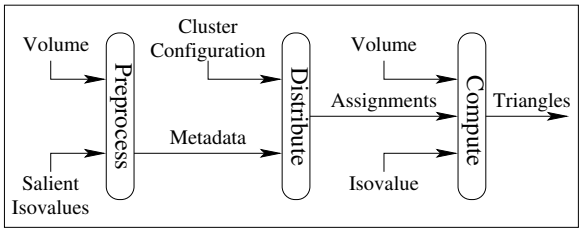

Figure 1: *Our approach preprocesses the volume data for a range of salient isovalues to estimate the amount of work required to perform isosurfacing for blocks of the input volume. The blocks are subsequently assigned to GPUs such that the isosurfacing work is more evenly distributed.*

lated work. Details of the isosurfacing cost heuristic are discussed in section 3.1. Then, details of the work distribution and isosurfacing algorithms are discussed in sections 3 and 4 respectively. Finally, results and conclusions are discussed in sections 5 and 6.

2. Related Work

A commonly applied tool in scientific visualization, isosurfacing has been well explored in research literature. Two broad groups of isosurfacing techniques exist: those that explicitly generate geometric primitives for the surfaces, and those that provide for direct rendering of the surfaces without necessarily generating geometric primitives for the entire isosurface. The former has advantages in cases where the geometric primitives are necessary or when the same surface is to be viewed from many different views. The latter has advantages in situations where the surface geometry is not needed or there are a limited number of views of interest and there is significant occlusion exhibited in those views. Our technique is among those in the former category, explicitly generating geometric primitives for isosurfaces.

Among techniques in the former category, the marching cubes technique, introduced by Lorensen, et al. [LC87], has become the ubiquitous solution. Further improvements on the core technique have been proposed by Nielson, et al. [Nie04]. An in-depth discussion is made on potential improvements on the marching cubes algorithm by Lopes, et al. [LB03].

In the original marching cubes algorithm, even cells without an isosurface in them are scanned. One approach used in avoidance of this is the use of hierarchical spatial data structures. Wilhelms, et al. [WVG92] propose using octrees, Livnat, et al. [LSJ96] propose a kd-tree-based method, and Dyken, et al. [DZTS08] extends the concept to a hierarchy of histograms to assist in efficient isosurface extraction. Itoh, et al. [IYK01] propose another method using contour trees to accelerate isosurfacing for unstructured volumes, skipping empty cells. Shen, et al. [SJ95] apply an algorithm utilizing the minimum and maximum values for groups of cells, in the context of unstructured data, to reduce unnecessary empty cell scanning. Another approach is a technique introduced by Gallagher [Gal91] in which values are bucketized to facilitate faster searching.

Several techniques have been developed to explicitly generate isosurface geometry using GPUs. Tatarchuk, et al. [TSD07] describe a technique using GPU geometry shaders to generate triangle geometry for tetrahedral volumes and tetrahedralized hexahedral volumes. Dyken, et al. [DZTS08] apply histopyramids [ZTTS06] to accelerating marching cubes isosurfacing on GPUs. Marching cubes are implemented directly in vertex shaders by Goetz, et al. [GJD05] and further enhanced with span-space acceleration techniques by Johansson, et al. [JC06]. Pascucci [Pas04] and Klein, et al. [KSE04] propose implementations of the marching tetrahedra algorithm on GPUs.

Many techniques have been developed that do not explicitly generate isosurface geometry. One of the simplest methods is to perform volume rendering with a transfer function that exposes the isovalues. Further refinements upon that are applying volume ray casting where the intersections with the surfaces in the interpolated cells are computed, then illuminated using common illumination models such as Phong's illumination model [Pho75]. One such example of a technique using ray tracing to render isosurfaces is proposed by Parker, et al. [PSL*98]. Point splatting based techniques such as those proposed by Co, et al. [CHJ03] and Livnat et al. [LT04] can also be applied. Röottger, et al. [RKE00] describe how cell projection, a technique often used for volume rendering, can be applied to isosurfacing. Another common approach is to generate view-dependent geometry that does not necessarily include the entire isosurface, taking into account occlusion. Gao, et al. [GS03] proposes one such technique where triangular geometry is directly generated in areas that pass a GPU-accelerated occlusion test.

While the fundamental marching cubes algorithms can easily map to data-parallel architectures under limited circumstances, a naive mapping can be very inefficient if the distribution of the isosurfaces throughout the volume is nonuniform. Additionally, many of the above techniques introduce acceleration data structures which add an additional degree of complexity to parallelization of the isosurfacing algorithms. Due to these concerns, and the ever increasing sizes of datasets to be analyzed, parallel isosurface extraction has been widely explored.

Gao, et al. [GS01] propose a parallel view-dependent isosurfacing algorithm using occlusion culling, combining hierarchical data structures with image space partitioning. Hansen, et al. [HH92] propose an algorithm that assigns individual cells in the volume to Connection Machine virtual processors – a concept that exists in a similar sense in the context of OpenCL-capable GPUs. Shen, et al. [SHLJ96] extend the span space isosurfacing acceleration algorithm

to a MIMD system by using a lattice-based search structure distributed to different processing elements. Zhang, et al. [ZBB01] and Chiang, et al. [CFSW01] both seek to provide an infrastructure for out-of-core rendering of isosurfaces on clusters. Gerstner, et al. [GR00] provides a strategy for distributing work for their hierarchical tetrahedral grid isosurfacing technique to processors in a SMP system.

Zhang, et al. [ZN03] uses a similar cost heuristic, in the context of out-of-core isosurfacing, to what is applied in our technique. However their technique considers active cells rather than triangle counts, and their technique uses hard-coded coefficients while ours profiles the target system and uses linear regression to estimate the coefficients. Isosurface statistics, as discussed by Scheidegger et al. [SSD*08], could be applied in the computation of a cost heuristic. However, we found a sampling of triangle counts to provide sufficient information for cost determination in our application while being substantially less complex.

Our technique directly generates triangular geometry for isosurfaces using marching cubes. With clusters having multiple nodes, each with multiple GPUs, with each GPU having multiple stream processors, we operate on two levels of parallelism: node-level and GPU-level. A data-parallel model is used for work distribution. To distribute load at the node level we use a cost heuristic based on profiling information to assign large blocks of cells from the volume to GPUs. To distribute load at the GPU level we apply data-parallel algorithms to each block [HSJ86], subdividing the block into rows. Our algorithm combines the simplicity of marching cubes with data-parallel algorithms to enable balanced fine-grained parallelism at the GPU level. Simultaneously, coarse-grained parallelism is applied at the node level using heuristics to provide for effective load balancing with minimal overhead.

3. Block Distribution Algorithm

The input data is treated as an array of cuboid blocks of cells, and it is assumed that the input data is too large to be fit entirely on any one node in the cluster. The goal of a block distribution algorithm is to assign these blocks to different GPUs in the cluster such that the load will be balanced for subsequent isosurfacing operations. The blocks need to be assigned to GPUs, without having to load the blocks explicitly on every node.

The block distribution algorithm consists of three phases: preprocessing, profiling, and assignment. The preprocessing phase collects data-centric information needed to compute the cost heuristic such as the triangle counts for different isovalues in different blocks. The profiling phase collects machine-centric information needed to compute the cost heuristic for the target machine. The assignment phase assigns blocks to GPUs across the cluster, given a user-defined range of salient isovalues and the cost heuristic.

3.1. Isosurfacing Cost Heuristic

An isosurfacing cost heuristic is required to estimate the amount of time it will take to compute isosurfaces for a block of cells in the volume. Some critical design requirements for such a heuristic are:

- It must enable estimation of the amount of time a block will take to compute. Even blocks with very few triangles may take substantial time.
- Not all of the data can be loaded every time we want to evaluate the heuristic. Instead, the heuristic must be computable with a value extracted from a simple, compact metadata representation produced by preprocessing.
- The heuristic should reflect the hardware platforms being used. The relationship of the overhead associated with starting the isosurfacing of a block to the actual isosurfacing work for a block may vary from platform to platform.
- It must be well-conditioned. We cannot have heuristic that will produce unreasonably large changes in its estimates for relatively small changes in the input metadata.

In our experiments we found a linear correlation between triangle count and isosurfacing time as exhibited in figure 2. Preprocessing can easily be performed to estimate triangle counts for different isovalues in different blocks of an input volume, which can then be subsequently stored as metadata. Requiring only the generated metadata rather than the entire volume, this enables fast and accurate estimation of a cost heuristic for isosurfacing a given block for a given isovalue.

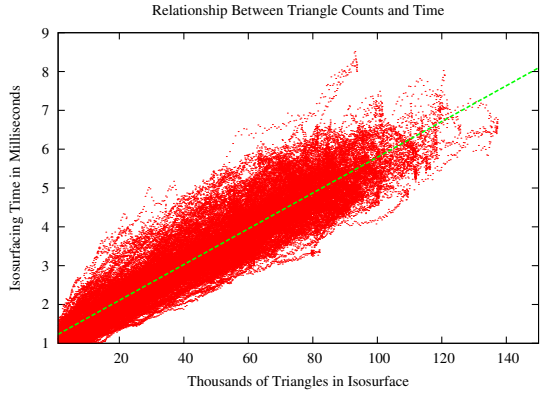

Figure 2: *The time required for isosurfacing a single block of a volume varies approximately linearly with the triangle count in the isosurface. The constant factor in the fit line is reduced by applying the optimizations discussed in section 4.3*

3.2. Preprocessing

Preprocessing is performed once per data set, in a standalone cluster-aware program. The preprocessing phase determines the triangle count for a range of isovalues for each block of cells. The probe isovalues used for determining the triangle

counts should be chosen so that they evenly cover the histogram of data values. This provides a representative sampling of potential isovalues. Our approach is to uniformly distribute the blocks across the cluster, with one process per CPU. For each block, the data values in the block are sorted in ascending order. To find M different probe isovalues for N sample values, we choose an isovalue to be the value at every $N/(M-1)$'th value in the sorted list.

For each of these probe isovalues we iterate through the data cells, on the CPU, to find the number of triangles that would be returned from the marching cubes algorithm. This is accomplished by classifying the cells into different marching cubes cases then using those classifications, per cell, to lookup triangle counts from a table. It is not necessary to explicitly compute the isosurfaces as the triangle counts are sufficient. The resulting mapping of isovalues to isosurface triangle counts is aggregated then written to a file, with a set of M entries for each block.

3.3. Profiling

The goal of the profiling phase is to determine the unknowns in the linear function mapping triangle count to the cost. The approach to do this needs to be reasonably inexpensive, but at the same time able to come up with reasonably confident estimates for the heuristic. Additionally, the approach must be appropriate for the block sizes used when subdividing the data for distribution to GPUs.

Our system generates a test volume of a size similar to that of a block. In our test cases a block size of 128^3 was used, but others could be used subject to the compromise discussed in §5.1.3. This synthetic test volume is sufficient so long as it provides for a diversity of triangle counts for different isovalues. The volume samples are generated by superimposing sinusoidal waves with random frequencies, directions, and amplitudes. This results in a reasonably complex volume for isosurfacing. We then compute the isosurfaces for isovalues ranging from the minimum to the maximum value in this generated field, estimating the time it takes for each. A linear least squares fitting is used to fit a linear function to these results, mapping triangle counts to expected times.

The resulting expected time from this equation, when evaluated for a particular triangle count, is the cost heuristic value for that triangle count. For clusters with more than one kind of GPU, the cost heuristic can be computed independently on the different kinds of GPUs. This provides a consistent basis for comparison of potential costs for isosurfacing across the different GPUs.

3.4. Assignment

Blocks are assigned to GPUs when the isosurfacing program is started, or when the user changes the set of salient isovalue ranges. From the preprocessing stage we have a table, one for each block, mapping a set of sample isovalues to triangle counts. From the profiling stage we have an equation mapping triangle counts to a cost heuristic. Using these tables, blocks need to be assigned to GPUs such that the variance is minimized between the sums of the cost heuristics of the blocks assigned to each GPU.

For every block, the cost heuristic is estimated using the set of salient isovalue ranges defined by the user. Because these ranges will not, in general, match the exact sample isovalues from the preprocessing stage, linear interpolation is applied between sample isovalues as necessary. The mean of the triangle count within the ranges specified by the user is computed to find the expected triangle count for a given block. With this triangle count, the cost heuristic can be evaluated, resulting in a single cost heuristic value for each block.

The blocks are then sorted in order of descending cost heuristic value. With this list, the blocks are then assigned to GPUs in a round-robin fashion. This results in an assignment of blocks to GPUs that is not necessarily optimal, but still is a good starting point.

To further refine the block assignments, they are randomly exchanged between GPUs, subject to the constraint that all exchanges must decrease the variance of the sums of the cost heuristic values assigned to each GPU. This is accomplished in a three step iterative process:

1. Pick a random pair of block assignments, with each element of the pair on a different GPU. This pair defines a potential exchange of block assignments.
2. If the variance is decreased by performing this exchange, the exchange is said to be successful. If the exchange is successful then we apply the exchange and return to step 1. Otherwise, we continue through this process.
3. If the number of unsuccessful exchanges since the last successful exchange exceeds a limit or the variance decreases below a threshold, break from this process, else return to step 1.

After this process is complete, each GPU has a list of blocks assigned to it. The block isosurfacing algorithm can then be applied independently on each GPU, where each GPU is responsible for processing the blocks assigned to it.

4. Block Isosurfacing Algorithm

When the block isosurfacing algorithm is applied, each GPU will have been assigned a set of blocks of the volume and the user will have selected a particular isovalue that they would like to visualize. The block isosurfacing algorithm needs to generate triangles for the isosurfaces, populating vertex buffers on the GPU. An algorithm for this needs to produce packed triangle buffers without wasted space in a format amenable to GPU rendering. Because the number of triangles produced for isosurfaces will vary substantially within

and between blocks, pre-allocating buffers to store triangles may be unacceptably wasteful in terms of memory consumption. Additionally, because GPUs are fundamentally parallel, such an algorithm needs to map well to the GPU parallel programming model.

One CPU thread controls each GPU, keeping each GPU busy processing the blocks assigned to it, resulting in one triangle buffer per block. We perform a marching cubes algorithm in two passes. The first pass counts the number of triangles and the offsets of the triangles into the vertex buffers. It does not directly compute the spatial positions of the triangles. The second pass creates the triangles, writing their spatial positions and normals into the vertex buffers according to the vertex buffer offsets found in the first pass. Figure 3 exhibits this process.

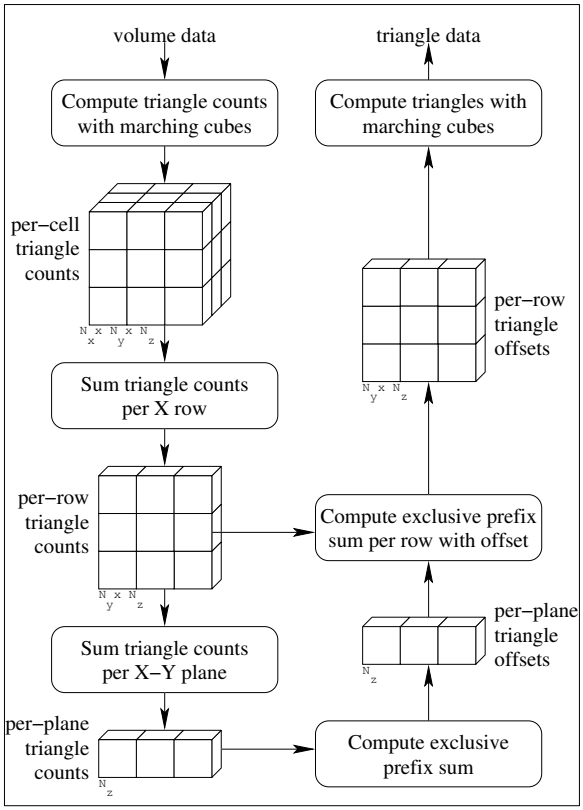

Figure 3: *The triangle counting and creation process computes vertex buffer offsets for rows of the block of cells being isosurfaced then applies marching cubes to fill the vertex buffer.*

4.1. Triangle Counting

The triangle counting phase takes a block of cells as input, and generates two outputs: a count of the total number of triangles in the isosurface in the block, and the offset of triangles within the vertex buffer for each X row of cells in the input volume. The triangle counting algorithm is local to each GPU, with one GPU operating on one block at a time.

Our approach applies exclusive prefix sums to compute the exact indices within the output vertex buffer for output triangles associated with each row of cells, resulting in a packed vertex buffer. The prefix sums could be implemented in parallel using techniques similar to those introduced by Harris [HSO07]. However, because we are computing prefix sums over many small distinct lists of numbers rather than one large list of numbers, it is more efficient to simply perform the many independent serial prefix sums in parallel. This maps well to GPUs because the individual sums are of nearly uniform length.

4.2. Triangle Creation

With the buffers resulting from the triangle counting pass, we now have the information needed to know where to store the triangles created by the marching cubes algorithm. The triangle creation phase computes these triangles and their normals.

Each X row of cells is assigned to a GPU thread. Each GPU thread then computes the isosurface triangles for its assigned row of cells. The resulting triangles for each row are placed into the target vertex buffer using the offsets computed in the triangle counting phase. This results in a packed vertex buffer on each GPU.

The packed vertex buffer contains positions of the vertices of the triangles. With these positions the normals for each vertex of the triangles can be computed by using finite differences to compute the gradient at each vertex. To obtain consistent normals, ghost cells are required around blocks. We found that using texture hardware and finite differences was substantially more efficient than attempting to compute normals directly using triangle connectivity and triangle geometry.

4.3. Optimizations

Some elements of the computation within the triangle counting and triangle creation phases is redundant. With a naive implementation, the blocks of cells will be sampled twice. Optimizations can be made to reduce the amount of redundant work. We apply two such optimizations: a minimum-maximum table for empty space skipping, and an isosurface crossing table to cache results from the triangle counting phase for use in the triangle creation phase.

4.3.1. Minimum-Maximum Table

Minimum and maximum values of the set of values within X rows of cells are computed at load time. Each row is subdivided into contiguous spans, with the minimum and maximum values being computed and stored for each span. This

data lets the triangle counting and triangle creation phases skip spans of cells that do not contain the isovalue, thus potentially reducing the number of required memory reads.

Trade-offs are present in terms of how large the spans in the minimum-maximum table should be. If the spans are too large, then it may be that fewer opportunities will be encountered to skip spans that do not contain isovalues. If spans are too small, then too much memory may be required to store the tables. Additionally, the minimum-maximum table needs to be read once per span to determine if the span contains the isovalue. This implies that, in addition to high memory consumption, span lengths that are too small may also result in excessive memory reads. We found span lengths in the range of 10 to 20 cells to be reasonable for the test datasets.

4.3.2. Isosurface Crossing Table

When we perform triangle counting, we are identifying the active cells. Rather than scanning all cells a second time in the triangle creation phase, we can record the indices of active cells within each X row in an isosurface crossing table. Then, when we perform the triangle creation we can iterate through this table instead of data values to apply marching cubes only to the cells that are active.

As with the minimum-maximum table, a compromise is present between performance and memory consumption. Large tables supporting a large number of isosurface crossings per X row can permit greater performance in cases where a large number of isosurface crossings per X row occur. Also, because only one of these tables needs to be stored per-GPU, rather than per-block with the minimum-maximum table, memory limitations are less restrictive. We implement this table as a byte per cell, at the full resolution of a block, because our block sizes are reasonably small.

5. Results

The test platform was a cluster of 12 Linux nodes. Each node had 16GiB of memory, two NVIDIA Quadro FX5600s each with 1.5GiB of memory, two quad core AMD Opteron 2350 CPUs at 2GHz, and an Infiniband interface. The algorithm was implemented using OpenCL for the GPU elements, MPI for inter-node communication, and Intel Threading Building Blocks [Rei07] for CPU multithreading. An important aspect of this configuration is the hierarchical nature of the parallelism – load must be balanced between nodes, amongst CPUs, and amongst GPUs.

We conducted four experiments on our test platform to explore:

- the relationship between isosurface triangle counts and isosurfacing time
- strong scalability: speedup in terms of a varying number of GPUs for a fixed data size
- volume size scalability: performance in terms of varying data size for a fixed number of GPUs

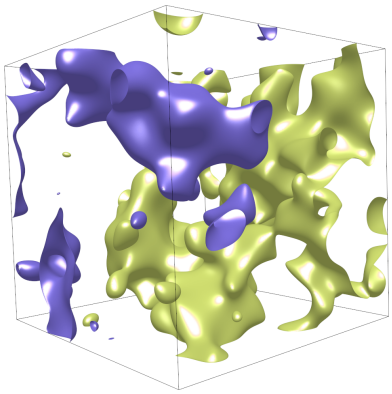

Figure 4: *The blue (dark) surface is isovalue -1.0 within the test volume used for the subsequent graphs and the yellow (light) surface is isovalue +3.0 within the same volume. At a volume resolution of 384x256x256 the gold surface contains 298858 triangles and the blue surface contains 916337 triangles.*

- the relationship between salient isovalue ranges, isovalues, and speedup

It was found that our cost heuristic yielded substantial performance improvements over a naive round robin distribution of blocks without a cost heuristic.

The test dataset was constructed from a sum of sine waves with random amplitude, frequency, and phase. The isosurfaces for isovalues -1.00 and 3.00 are exhibited in figure 4. This dataset was chosen because it offers sufficient complexity and variation to be interesting, and is easy to reproduce at any resolution. The base dataset size we use is 1536x1024x1024 resulting in 6 gigabytes of IEEE754 single precision floating point samples. To maintain consistency, the dataset was downscaled from this base size as necessary for the different experiments.

5.1. Triangle Counts versus Isosurfacing Time

The time required to isosurface each block of cells within a volume was recorded, along with the number of triangles in the blocks, resulting in a mapping of triangle counts to times as in figure 2. This experiment directly examines the performance of the block isosurfacing algorithm from section 4. For a fixed block size, a linear relationship was found between the isosurfacing time for a single block, and the number of triangles within the isosurface in the block. Different elements contribute to the constant and linear factors.

5.1.1. Constant factor

Several elements contribute to the constant term in the linear relationship. Fundamentally, they are of two types: those that are related to the size of the block being isosurfaced and

those that are not. In our algorithm, GPU kernel execution startup times and OpenCL API overhead are independent of the size of the volume blocks being considered. Additionally, the GPU to CPU and CPU to GPU transfer times from within the triangle counting algorithm are dominated by the startup cost of the transfers rather than the size of the transfers because the transfer sizes are intentionally small, on the order of 128 bytes for a 64^3 block and 512 bytes for a 128^3 block.

However, other elements of the triangle counting algorithm that contribute to the constant factor do exhibit dependence on the size of the block. Time is required to perform the exclusive prefix sums on the tables for the block as seen in section 4.1. Additionally, time is required to perform marching cubes table lookups and volume lookups to count the number of triangles in each cell. The minimum-maximum table optimization from section 4.3 seeks to reduce these contributors to the constant factor by reducing the number of cells whose triangle counts must be checked, at the cost of requiring some additional table lookups.

Typical constant time factors seen on our test platform for a 128^3 block on a single GPU were around 1.2ms. This time is dominated by the API and kernel startup time overhead. Further exploration may be worth consideration when NVIDIA Fermi-class GPUs become available, which may reduce kernel context switching time. Additionally, the drivers for OpenCL are still relatively new, so additional optimizations should be expected in the future to reduce API-related overhead. Such hardware and software improvements would further increase the benefits seen from our algorithm by reducing this constant factor.

5.1.2. Linear factor

The linear term in the triangle count to time relationship also has multiple contributing factors. Marching cubes triangle construction requires interpolations and table lookups, per triangle, with up to five triangles per cell. For the vertices of each triangle, finite differencing using the GPU texturing hardware is used to compute gradients that are normalized to form triangle vertex normals. This requires 18 texture lookups per triangle for central differencing, hence its contribution to the linear factor. While the isosurface crossing table from section 4.3 can reduce the constant factor substantially by eliminating the need for a second scan of the volume cells for triangles in the triangle creation phase, it does introduce a linear factor because it requires a write for each non-empty cell in the triangle counting phase (§4.1) and a read for each non-empty cell in the triangle creation phase (§4.2). The write is of lesser consequence from a performance standpoint because there are no read-after-write hazards associated with it in the triangle counting phase.

Typical times seen on our test platform for the linear factor were around 40ns per triangle, on a single GPU. Faster GPU memory and better GPU caches would reduce this factor substantially, so it is expected that with new NVIDIA GPUs such as Fermi this linear factor may see a substantial improvement, though not to the same extent that would be expected of the constant factor.

5.1.3. Block size compromise

With some of the constant factor contributors depending on the number of cells in the block, and some not depending on the number of cells in the block, it is clear that choosing an appropriate block size is a trade-off. As the block size is made smaller, the overall performance for isosurfacing in terms of single blocks will decrease because the net overhead for isosurfacing will be higher, but the load balancing between different GPUs may be more accurate because of decreased load balancing data granularity. We found block sizes of around 128^3 to be a good compromise, with larger blocks offering insufficient flexibility for load balancing thus reducing multi-GPU speedup, and smaller blocks having too much overhead.

In our algorithm the triangle counting phase (§4.1) contributes primarily to the constant factor while the triangle creation phase (§4.2) contributes primarily to the linear factor. As architectures change, the algorithm can be adapted by moving complexity from one phase to the other.

The linear relationship between single block isosurfacing time and the number of triangles enables the transformation of predicted triangle counts into a cost heuristic as discussed in section 3.1.

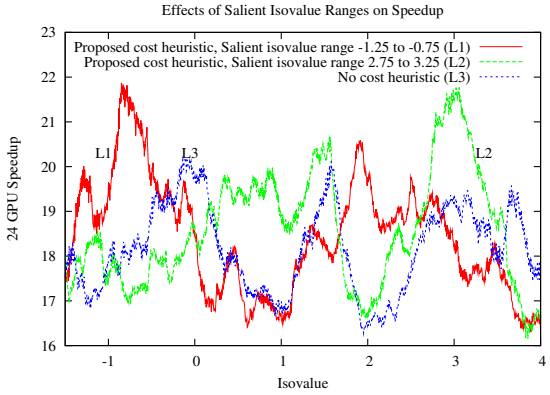

Figure 5: *The salient isovalue ranges substantially affect the performance. In this figure it can be seen that the speedup is improved over ranges of isovalues that are specified as salient. When no cost heuristic is used, the distribution of performance over the isovalue range is not well defined because the effective cost value of the work for each block is equal. Each line has 1100 sample isovalues, computed over a 1536x1024x1024 test volume on 24 GPUs.*

5.2. Effects of salient isovalue ranges on speedup

This experiment was conducted to explore how the user selected salient isovalue range and the isovalue being isosurfaced affect the speedup. Fixed size (1536x1024x1024) data was broken into roughly uniformly sized 128^3 cell blocks, with some variation at the edges of the volume. Blocks were distributed to the different GPUs using the algorithm discussed in section 3. Isosurfaces were then computed using the algorithm in section 4.

Three different runs were performed, each sweeping 2000 isovalues from -3.00 to 7.00, with the 1100 isovalues ranging from -1.50 to 4.00 exhibited in figure 5 with a different line for each run:

- red line (line L1): uses our cost heuristic in distributing the blocks, with a salient isovalue range of -1.25 to -0.75 selected.
- green line (line L2): uses our cost heuristic in distributing the blocks, with a salient isovalue range of 2.75 to 3.25 selected.
- blue line (line L3): uses no cost heuristic, distributing the blocks in an arbitrary order.

Speedup varies based on how evenly isosurfacing work is distributed across the nodes. The uniformity of isosurfacing work distribution at the node level is a function of both the isovalue and the dataset, because the work is linearly proportional to the number of triangles in the isosurface. Our algorithm assigns blocks to nodes to minimize the variance between the sums of the work assigned to each node, analogous to the variance of the sums of the cost heuristic values of the blocks assigned to each node. The salient isovalue range determines the range of isovalues that are considered when computing the cost heuristic.

The green line (line L2) in figure 5 exhibits a strong peak in the range of 2.75 to 3.25 because that is the salient isovalue range that was selected for that run, which implies that the work was assigned to nodes to maximize work uniformity only for those ranges of isovalues. However, other peaks are visible in locations like 1.5 because there is likely a similar spatial distribution of the isosurfaces for values around 1.5 as there is for isovalues around 3.0, in this data. Similarly to the green line, the red line (line L1) exhibits the same phenomenon for a different salient range, -0.75 to -1.25, with different similar regions for the same reason.

The blue line (line L3) is drawn for the naive no-cost-heuristic method. It shows varied performance because the arbitrary block assignments can create different work distribution uniformities, and thus different speedups, for different isovalues. Both the red line and the green line demonstrate substantially improved speedup over the naive method in their salient ranges.

The results of this exhibit that selecting salient isovalue ranges does offer the potential for improved speedup within those regions. At the same time, isovalues outside of those ranges do not suffer unacceptable penalties in speedup, sometimes even receiving improved speedup.

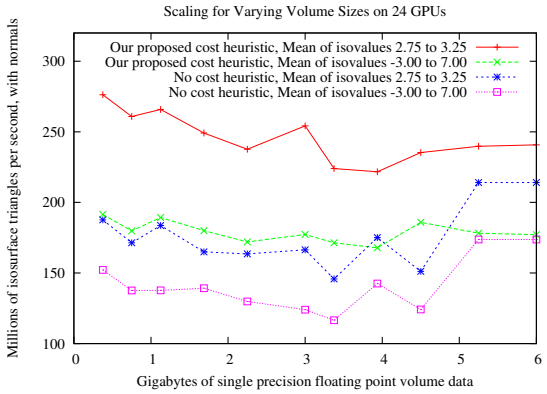

Figure 6: *The performance advantage of using our cost heuristic over using no cost heuristic is maintained over the range of loadable volume sizes on a cluster of 24 GPUs. The salient isovalue range used for the cost heuristic is 2.75 to 3.25, resulting in a mean isosurfacing performance on the order of 250 million triangles per second over that range of isovalues. Using no cost heuristic over that same range yields performance on the order of 175 million triangles per second.*

5.3. Volume size scalability

This experiment examined volume size scalability of our algorithm; that is, it ran trials for varying data sizes, with a fixed number of processing elements. The data was scaled from 1536x1024x1024 down to the appropriate sizes. Data was divided into roughly uniformly sized 128^3 cell blocks. 32 blocks were assigned per GPU for the largest resolution and 2 blocks were assigned per GPU for the smallest resolution.

Two different runs were performed, one with our proposed cost heuristic, with a salient isovalue range of 2.75 to 3.25, the other with no cost heuristic and arbitrary block assignments. From each of those runs, data was collected for two different ranges of isovalues:

- 100 isovalues in 2.75 to 3.25, the salient range
- 2000 isovalues in -3.00 to 7.00, the entire range

Mean and maximum times were recorded per isovalue, yielding the four lines in figure 6.

Small scale variation occurs within the lines of figure 6 primarily because there is a small degree of noise present in the isosurfacing times and in the cost heuristic accuracy, thus, that noise can manifest itself in the results. In the case of the run done with no cost herustic there is a second source of variation. With no cost heuristic, the block assignments are arbitrary, thus changing the data size completely rearranges the block assignments which results in substantial

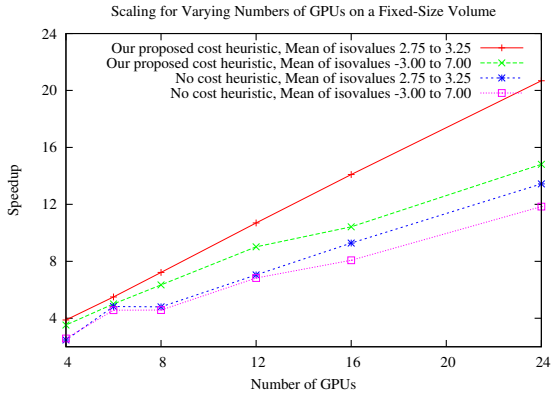

Figure 7: *Using our proposed cost heuristic improves scalability, especially when the isovalues for which isosurfaces are being computed are within the salient range. In this figure, the salient range of isovalues used for the computation of the cost heuristic is 2.75 to 3.25 and the volume is 768x512x512 samples.*

variation in the resulting times. In the case of our cost heuristic based algorithms, such variation does not occur to the same extent because block assignment is done based upon the cost heuristic.

The red line (the top line in the key) shows the performance of our method when the salient range matches the range being isosurfaced. It substantially outperforms the other test cases, with triangle rates on the order of 250 million per second. Comparing the other lines it can be seen that the performance can still be better than using no heuristic at all even when the isosurfacing is done in ranges outside of the salient isovalue range.

All four lines exhibit good volume size scalability, with the performance not substantially decreasing for an increasing data size on the same set of processing elements. In the next section it will be shown that the algorithm delivers strong scalability as well.

5.4. Strong scalability

This experiment was conducted on a fixed size 768x512x512 test data set, breaking the data into blocks of approximately 128^3 cells. The experiment was run on 4, 6, 8, 12, 16, and 24 GPUs to examine strong scalability; that is performance scaling for a fixed data size and varying numbers of processing elements. The time to isosurface every block was recorded, and the results for every block were stored, including triangles with normal data. Two different runs were performed for each GPU count:

- using our cost heuristic, with a salient isovalue range of 2.75 to 3.25 selected.
- using no cost heuristic

From each of those runs, we took the mean and maximum times for isosurfacing two ranges of isovalues, 2.75 to 3.25 (the salient range), and -3.00 to 7.00 (the entire range.) For the former range 100 isovalues were sampled and for the latter range 2000 isovalues were sampled. This resulted in the 4 lines in figure 7.

The dependence on triangle counts for isosurfacing on the GPUs means that load may be distributed unevenly between nodes depending on the isovalue and the data. Our block distribution technique seeks to reduce this disparity, and the results in figure 7 exhibit its success in achieving this.

Over the salient isovalue range of 2.75 to 3.25, our technique has substantially better speedup (21x on 24 GPUs) versus the naive technique with no cost heuristic over the same range (13x on 24 GPUs). Even over the entire range of values, -3.00 to 7.00, a modest benefit in speedup was seen versus the naive technique. When scaled to a larger number of GPUs, with appropriately larger data, we expect that scalability would continue similar trends.

If the block size could be made smaller, then we could possibly further improve the performance. However, this would be unlikely to increase performance because decreasing the block size would decrease the absolute performance per GPU. If interconnect, bus, and/or disk speeds were higher relative to the speed of the GPUs an attempt could be made to dynamically load blocks on demand. However, we already attain 86% efficiency over the salient range of isovalues with 24 GPUs, so it is unlikely that such an approach could yield further performance improvement.

6. Conclusion

We have presented an efficient, load-balanced multi-node, multi-CPU, multi-GPU method for computing triangular isosurfaces on volume data. A preprocessing stage computes metadata, permitting the efficient computation of a cost heuristic. The cost heuristic is computed for blocks using the preprocessed data and user-specified hints on isosurface saliency, then blocks are distributed to GPUs to maximize work uniformity. An efficient parallel isosurfacing algorithm is then applied on each GPU with the assistance of the CPUs to produce triangles in packed arrays that may subsequently be used for rendering or other computations.

Our implementation is able to deliver isosurfacing performance in excess of 250 million triangles per second on 24 GPUs. Strong scalability is exhibited with 90% utilization with 8 GPUs and 86% utilization with 24 GPUs. Our algorithm enables the leveraging of contemporary hybrid-architecture clusters with CPU and GPU resources for more efficient exploration of large scale volume data.

References

[CFSW01] CHIANG Y., FARIAS R., SILVA C., WEI B.: A unified infrastructure for parallel out-of-core isosurface extraction and

volume rendering of unstructured grids. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (2001), IEEE Press Piscataway, NJ, USA, pp. 59–66. 3

[CHJ03] CO C., HAMANN B., JOY K.: Iso-splatting: A point-based alternative to isosurface visualization. In *Proceedings of the Eleventh Pacific Conference on Computer Graphics and Applications-Pacific Graphics 2003* (2003), Citeseer, pp. 325–334. 2

[DZTS08] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.: High-speed marching cubes using histopyramids. In *Computer Graphics Forum* (2008), vol. 27, Blackwell Publishing, pp. 2028–2039. 2

[Gal91] GALLAGHER R.: Span filtering: an optimization scheme for volume visualization of large finite element models. In *Proceedings of the 2nd conference on Visualization'91* (1991), IEEE Computer Society Press Los Alamitos, CA, USA, pp. 68–75. 2

[GJD05] GOETZ F., JUNKLEWITZ T., DOMIK G.: Real-time marching cubes on the vertex shader. In *Proceedings of Eurographics* (2005), vol. 2005. 2

[GR00] GERSTNER T., RUMPF M.: Multiresolutional parallel isosurface extraction based on tetrahedral bisection. In *Volume Graphics* (2000), vol. 278. 3

[GS01] GAO J., SHEN H.: Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (2001), IEEE Press, p. 74. 2

[GS03] GAO J., SHEN H.: Hardware-assisted view-dependent isosurface extraction using spherical partition. In *Proceedings of the symposium on Data visualisation 2003* (2003), Eurographics Association, p. 276. 2

[HH92] HANSEN C., HINKER P.: Massively parallel isosurface extraction. In *Proceedings of the 3rd conference on Visualization'92* (1992), IEEE Computer Society Press, p. 83. 2

[HSJ86] HILLIS W., STEELE JR G.: Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1183. 3

[HSO07] HARRIS M., SENGUPTA S., OWENS J.: Parallel prefix sum (scan) with CUDA. *GPU Gems 3*, 39 (2007), 851–876. 5

[IYK01] ITOH T., YAMAGUCHI Y., KOYAMADA K.: Fast isosurface generation using the volume thinning algorithm. *IEEE Transactions on Visualization and Computer Graphics* (2001), 32–46. 2

[JC06] JOHANSSON G., CARR H.: Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, October* (2006), Citeseer, pp. 16–19. 2

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proceedings of Pacific Graphics 04* (2004), pp. 186–195. 2

[LB03] LOPES A., BRODLIE K.: Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics* (2003), 16–29. 2

[LC87] LORENSEN W., CLINE H.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), ACM, p. 169. 2

[LSJ96] LIVNAT Y., SHEN H., JOHNSON C.: A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84. 2

[LT04] LIVNAT Y., TRICOCHE X.: Interactive point-based isosurface extraction. In *Proceedings of the conference on Visualization'04* (2004), IEEE Computer Society Washington, DC, USA, pp. 457–464. 2

[Nie04] NIELSON G.: Dual marching cubes. In *Proceedings of the conference on Visualization'04* (2004), IEEE Computer Society Washington, DC, USA, pp. 489–496. 2

[Pas04] PASCUCCI V.: Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. UCRL-CONF-202459, Lawrence Livermore National Laboratory (LLNL), Livermore, CA. 2

[Pho75] PHONG B.: Illumination for computer generated pictures. *Communications of the ACM* (1975). 2

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.: Interactive ray tracing for isosurface rendering. In *Proceedings of the conference on Visualization'98* (1998), IEEE Computer Society Press Los Alamitos, CA, USA, pp. 233–238. 2

[Rei07] REINDERS J.: *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007. 6

[RKE00] RÖTTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of the conference on Visualization'00* (2000), IEEE Computer Society Press Los Alamitos, CA, USA, pp. 109–116. 2

[SHLJ96] SHEN H., HANSEN C., LIVNAT Y., JOHNSON C.: Isosurfacing in span space with utmost efficiency. In *Proceedings of the 7th conference on Visualization'96* (1996), IEEE Computer Society Press Los Alamitos, CA, USA. 2

[SJ95] SHEN H., JOHNSON C.: Sweeping simplices: A fast isosurface extraction algorithm for unstructured grids. In *IEEE Visualization: Proceedings of the 6 th conference on Visualization'95* (1995), Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,. 2

[SSD*08] SCHEIDEGGER C., SCHREINER J., DUFFY B., CARR H., SILVA C.: Revisiting histograms and isosurface statistics. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1659–1666. 3

[TSD07] TATARCHUK N., SHOPF J., DECORO C.: Real-Time isosurface extraction using the GPU programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses* (2007), ACM, p. 137. 2

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG) 11*, 3 (1992), 201–227. 2

[ZBB01] ZHANG X., BAJAJ C., BLANKE W.: Scalable isosurface visualization of massive datasets on COTS clusters. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (2001), IEEE Press, p. 58. 3

[ZN03] ZHANG H., NEWMAN T.: Efficient parallel out-of-core isosurface extraction. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), IEEE Computer Society, p. 3. 3

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.: GPU point list generation through histogram pyramids. *Technical Reports of the MPI for Informatics* (2006), 2006–4. 2