Hybrid CPU-GPU Solver for Gradient Domain Processing of Massive Images

Sujin Philip, Brian Summa, Valerio Pascucci
Scientific Computing and Imaging Institute
Salt Lake City, USA
sujin, bsumma, pascucci@sci.utah.edu

Peer-Timo Bremer
Lawrence Livermore National Laboratory
Livermore, USA
bremer5@llnl.gov

***Abstract*—Gradient domain processing is a computationally expensive image processing technique. Its use for processing massive images, giga or terapixels in size, can take several hours with serial techniques. To address this challenge, parallel algorithms are being developed to make this class of techniques applicable to the largest images available with running times that are more acceptable to the users. To this end we target the most ubiquitous form of computing power available today, which is small or medium scale clusters of commodity hardware. Such clusters are continuously increasing in scale, not only in the number of nodes, but also in the amount of parallelism available within each node in the form of multicore CPUs and GPUs. In this paper we present a hybrid parallel implementation of gradient domain processing for seamless stitching of gigapixel panoramas that utilizes MPI, threading and a CUDA based GPU component. We demonstrate the performance and scalability of our implementation by presenting results from two GPU clusters processing two large data sets.**

***Keywords*-hybrid programming; gpgpu; cluster; gradient domain; image processing**

I. Introduction

Massive panoramic images are increasing in popularity due to the availability of inexpensive robots to automate the capturing process [1]. Extreme resolution imagery is also available from sources such as the United States Geological Survey (USGS) [2] for satellite imagery and the High Resolution Imaging Science Experiment (HiRISE) [3] for high resolution planetary images. Processing such images requires significant computational resources and time. As these images continue to grow, new methods must be developed to make their processing feasible in practice.

A common problem with such images is that they are composed of hundreds of individual images taken over a course of hours or even days. This results in the images having greatly varying lighting conditions and/or exposure and the composed image is an unappealing patchwork. Much work has been done in removing the seams from these images and currently gradient domain based approaches provide the best solution.

Two methods have been proposed for gradient domain processing of massive images: the streaming multigrid [4] and progressive Poisson [5] techniques. These are sequential methods and though efficient, still take several hours to compute solutions for gigapixel images. Hence, some work has been done recently to extend these methods to work under a distributed environment in [6] and [7] respectively. The implementation described in [7] has been shown to be portable across different systems and has good scalability in both the strong and weak sense. It processes the images in a streaming, out-of-core fashion and provides strict control over the required resources.

In this paper we extend the framework described in [7] to also support GPUs, providing a hybrid implementation that can efficiently utilize both CPUs and GPUs to speedup the computation of the solution. In the following sections we will provide an overview of our framework and will detail the GPU implementation of the solver. We will provide several variations of GPU implementations and compare their performance with the base line CPU implementation. Finally, strong and weak scaling of the framework is demonstrated with results from runs on two different cluster and on two large image datasets.

II. Related Work

Gradient Domain Image Processing. Gradient domain image processing is a technique where images are manipulated in the gradient space, not on the direct pixel values. After processing an image's gradient field for the desired effect, these methods attempt to find a smooth image that is closest to the guiding gradient with a minimal least squared error. Gradient domain processing has been used for applications such as seamless cloning [8], drag-and-drop pasting [9], matting [10] and seamless panorama stitching [4], [6]–[8], [11], [12]. Other examples include compressing HDR (High Dynamic Range) images for display on standard monitors [13] and applications such as detection of lighting [14] or shapes [15] from images, shadow removal [16], reflections [17], and artistic editing in the gradient domain [18].

Poisson Solution. The problem of finding this smooth image can be expressed as a 2D Poisson equation. Computing the solution to Poisson equations efficiently has been the focus of a large body of work. Direct solution of Poisson equations can be computed using Fast Fourier Transform

Figure 1. Although the result is a seamless, smooth image, without coarse upsampling the independent processing of the tiles will fail to account for large trends that span beyond a single overlap and can lead to unwanted, unappealing shifts in color.

(FFT) methods [18]–[21]. These methods are global in nature and require special formulations for parallelization. Furthermore, they have not yet shown to adapt well to an out-of-core framework which is mandatory for extreme scale images.

The accuracy of the solution is not crucial for some applications and a coarse approximation may be sufficient to achieve the desired results. For example, it has been shown that Bilateral upsampling [22] produces good results for tone-mapping of HDR images.

Often a Poisson solution is found by discretizing the equations into a large linear system. These systems can then be solved by finding a direct solution or through iterative methods. An extensive review of direct solution methods can be found in [23] and Heath et al. [24] provide a survey of parallel algorithms for direct solution. Iterative methods for solving linear systems include conjugate gradient methods and Successive Over-Relaxation (SOR) [25]. Conjugate gradient methods have fast convergence but for large linear systems memory consumption can sometimes be a limiting factor and SOR is then the preferred method. Szeliski et al present a fast method of Poisson blending [26] that takes advantage of the smoothness of each individual image's offset map to represent them using low-dimensional splines. This reformulation results in a much smaller linear system than the original and has lesser memory requirements and faster solving time.

Multigrid iterative methods can be used to provide faster convergence. These techniques include preconditioners [27], [28] and multigrid solvers [29], [30]. Multigrid methods could use either adaptive [31]–[35] or non-adaptive meshes [4]–[6], [36]. Chow et al. [37] provide a survey of current methods for distributed multigrid solver.

Summa et al. [5] provide a method where upsampling and the coarse-to-fine half of a multigrid cycle are combined to produce high quality results for imaging. In this approach, it was shown that an initial coarse solution, when upsampled and used as the initialization of an iterative solver, produces results visually indistinguishable from a direct solution.

Out-of-Core Methods. Toledo [38] provides a survey of general out-of-core algorithms for solving linear systems. Most of them require that at least the solution be kept in the main memory which is not possible for gigapixel imagery. The streaming multigrid method [4], [6] and the progressive Poisson method [5], [7] are the only methods that have been shown to be able to handle images at this scale.

Tile-Based Methods. Large images are often stored as multi-resolution tiles, therefore methods which operate directly on this format would be ideal. Stookey et al. [39] use a tiled based approach to compute an over-determined Laplacian PDE. The method uses overlapping tiles and solves the PDE on these tiles independently. It then blends the solutions, using a weighted average, to get the final solution. Since the tiles are solved locally and independently, the final solutions do not account for large scale trends in the image and may introduce unwanted shifts as shown in Figure 1. Philip et al.'s tile based approach [7] uses the corresponding portion of an upsampled, coarse, full solution as the initial values for the tiles and therefore can account for these trends.

III. Gradient Domain Review

Gradient based techniques manipulate an image's gradient field instead of its color values. The resulting gradient field is then solved to obtain a smooth image that fits the gradient field with minimal least squared error. That is - given a gradient field $\vec{G}(x,y)$, defined over a domain $\Omega \subset \Re^2$, its corresponding image $P(x,y)$ is to be found such that its gradient ∇P fits $\vec{G}(x,y)$.

In order to minimize $\|\nabla P - \vec{G}\|$ in a least squares sense, one has to solve the following optimization problem:

$$\min_P \iint_\Omega \|\nabla P - \vec{G}\|^2 \quad (1)$$

Minimizing equation (1) is equivalent to solving the Poisson equation $\Delta P = div\ \vec{G}(x,y)$ where Δ denotes the Laplace operator $\Delta P = \frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2}$ and $div\ \vec{G}(x,y)$ denotes the divergence of $\vec{G}$. For discrete images, we adapt the above equations using a standard finite difference approach which approximates the Laplacian as:

$$\begin{aligned} \Delta P(x,y) &= P(x+1,y) + P(x-1,y) + \\ &\quad P(x,y+1) + P(x,y-1) - 4P(x,y) \end{aligned} \quad (2)$$

and the divergence of $\vec{G}(x,y) = (G^x(x,y), G^y(x,y))$ as:

$$\begin{aligned} div\ \vec{G}(x,y) &= G^x(x,y) - G^x(x-1,y) + \\ &\quad G^y(x,y) - G^y(x,y-1) \end{aligned}$$

We then discretize the differential form $\Delta P = div\ \vec{G}(x,y)$ using finite differences into a sparse linear system: $L\mathbf{p} = \mathbf{b}$. Each row of the matrix L stores the weights of the standard five point Laplacian stencil, $\mathbf{p}$ is the vector of pixel colors, and $\mathbf{b}$ encodes the guiding gradient field. Both L and $\mathbf{b}$ encode boundary conditions as well.

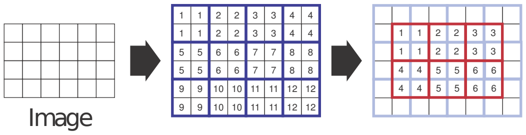


Figure 2. Our tile-based approach: (left) An input image is divided into equally spaced sub-tiles. (center) In the first phase after a symbolic padding by a column and row in all dimensions, a solver is run on a tile denoted by a collection of 4 labeled sub-tiles. (right) Data is sent and collected for the next phase to create new data tiles with 50% overlap.

The guiding gradient field and the boundary conditions are chosen based on the application. For example, seamless cloning uses Dirichlet boundaries set to the color values of the background image and uses the foreground image's gradient as the guiding field [8]. Panorama stitching uses Neumann boundary conditions and the guiding gradient field is computed as a composite of the gradients from the source images. At image boundaries, where an unwanted, large gradient exists (the seams), the gradient is averaged or set to zero [4], [5], [8], [11], [12]. Another interesting example is gradient domain painting [18] which uses artistic input as the guiding field.

IV. Hybrid Gradient Domain Processing

To provide the necessary flexibility to exploit all available resources we have chosen a tile based approach for our implementation. The input image is divided into tiles and distributed among the various nodes and processing units within a node. These tiles are then processed independently and in parallel. Traditionally, tile based solvers have been used only when global trends could be ignored. Otherwise, as discussed above, independent processing of tiles may introduce unwanted shifts (Figure 1). To avoid these shifts, we employ the method described in [5], [7], where it has been shown that using an upsampled coarse solution as the initial value for an iterative solver produces visually pleasing results. Independent processing of the tiles may also introduce inconsistencies at the boundaries of the tiles, resulting in new seams in the solution. We smooth out these seams by running a second pass with tiles that significantly overlap the tiles from the first pass, as shown in Figure 2. The second pass is constrained to work only in the interior pixels of the tiles so as to not introduce any more seams. In this section, we will provide an overview of our framework for seamless panorama stitching and describe our implementation of the iterative solver on the GPU.

Seamless Panorama Stitching. In the initialization phase, the image is logically divided into a set of tiles and distributed among the various nodes of the cluster. If the cluster contains heterogeneous nodes, the tiles can be distributed based on the amount of computing resources on each node [7]. The nodes then load a (very) coarse version of the image and compute its full solution using a direct FFT solver.

An important component in seamless panorama stitching is the map or label image. Each pixel in the map stores an identifier to the source image which gives the color value or stores a flag indicating that the pixel falls outside the boundary of the panorama. This map is necessary to identify the boundary of the panorama, which is typically irregular, and also the boundaries between the source images. Furthermore, the information provided by the map is used for computing the gradient field.

Each node streams its domain of tiles from the disk to a pipeline which computes the solution. After the first pass the image is re-tiled for the second pass. The re-tiling is done in such a way that there is a 50% overlap in processing between the two passes (see Figure 2). This will remove any seams that may be produced in the first pass of the solution. We chose an overlap of 50% because it simplifies the re-tiling logic. A node may have to exchange overlapping portions of tiles in the boundary of its domain with its nieghbors during the re-tiling process. After the data exchange and re-tiling, the new tiles are again passed to the pipeline for the second pass and the final solution is written to disk.

The solver pipeline runs on every node and consists of two main stages. One computes the divergence of a tile, using the input image and map file. The other solves the tile with an iterative solver using the divergence and the upsampled coarse solution, which it uses as the initial value. For the iterative solver we have chosen to implement Successive Over-Relaxation (SOR) for its simplicity and fast convergence. The stages of the pipeline are dynamically scheduled to execute on the available resources (CPUs and GPUs). For our implementation we have chosen to use the scheduler describe in [40], [41] as it has been shown to be an efficient single-node scheduler. This scheduler allows the programmer to specify execution pipelines in the form of directed graphs consisting of various processing modules. These modules can have multiple implementations for different types of resources. The programmer only has to provide these different implementations and the scheduler takes care of managing the resources and load balancing by dynamically choosing the appropriate implementation at runtime.

A detailed explanation of the framework can be found in [7]. In this work, we have extended the implementation described in that paper to include support for processing on the GPUs in addition to multi-threaded CPUs on each node.

GPU SOR Implementation. We tested several implementations of SOR solver for the GPU, starting with a simple adaptation of the CPU implementation. This base implementation was then incrementally modified with different optimizations and its performance was recorded. Figure 3

Kernel Performance	
Implementation	Avg. Time (sec)
CPU	52.171
Kernel #1	3.810
Kernel #2	2.103
Kernel #3	1.529
Kernel #4	1.421
Kernel #5	2.016

Figure 3. The performance results for the kernels we tested. Kernel #1 is the base kernel, Kernel #2 processes the color channels separately, Kernel #3 uses shared memory for solution and divergence, Kernel #4 uses shared memory for solution and texture for divergence, and Kernel #5 uses red-black iterations.

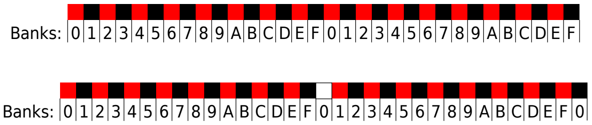

Banks: 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Banks: 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F 0

Figure 4. Avoiding bank conflict in red-black iterations: The figures show one row of a block's window. In the default configuration (top), two pixels of the same color are mapped to each bank resulting in bank conflicts when the threads try to simultaneously access them. (bottom) by adding a buffer of one pixel at column 16, the bank conflict is avoided since now each pixel of the same color occupy different banks.

shows a comparison of the performance of the various implementations tested. The kernels were developed and tested on NVIDIA devices with compute capability 1.3, on machines running cuda version 3.2. The baseline GPU performance tests were run on image tiles of dimension 1024×1024 pixels for a fixed 1000 iterations and the average time taken was recorded.

In SOR, the map image is only used to identify whether the corresponding pixels fall outside the boundary of the panorama. To save both memory and transfer cost to the GPU, we chose to encode this information in the divergence itself. The divergence at a pixel is set to a special, very large value (floating point NaN), if it falls outside the boundary of the panorama. The kernels refer the divergence buffer, instead of the map, to check if a particular pixel is outside the boundary.

An important factor in kernel performance is the thread-block dimension. Each iteration of SOR has relatively little computation compared to the amount of memory accessed. Global memory accesses are very expensive as they reside in the off-chip graphics RAM. The GPU's warp-scheduler (a warp is a set of 8 threads executing in SIMD manner) tries to hide memory access latencies by scheduling other warps while the memory request is being processed [42]. Therefore, blocks should contain sufficient number of threads for the warp scheduler to hide memory access latencies. The devices tested support a maximum block dimension of 512 threads but for our implementation we settled on a block dimension on 16×16 or 256 threads to avoid register spillage.

Initially, we started with a straight forward implementation of SOR for the GPU. In this first incarnation, the data was stored with interleaved RGB values in $float3$ variables. This gave poor performance due to bad memory coalescing and locality. The first optimization applied was to split the channels into different buffers and process them independently. This significantly improved the performance of the kernels as shown in Figure 3 for kernel #2.

The next optimization was to copy each block's portion of solution and divergence to the on-chip, per-block shared memory buffers. This was based on the observation that there is a significant amount of reuse for these values. Each pixel is referred by up to four of its neighbors and moving these values into the fast shared memory can improve the performance significantly. The values were loaded into the shared memory in parallel and in a coalesced manner. The outer-neighbors of the boundary pixels of a block were still accessed from global memory since they were only accessed once per launch. Next we noted that there are only read-only accesses to the divergence. By using read-only texture memory to store the divergence, we could free important shared memory space. Furthermore, texture memory accesses are cached and their use can provide performance on par with, or better than, shared memory depending upon the access patterns. As the graph shows, there is a slight performance improvement between kernel #3 and kernel #4.

Finally, we implemented red-black iterations for the solver to increase its convergence speed. In red-black iterations, pixels are assigned to either red or black groups so as to form a red-black checkerboard pattern. First the red pixels are solved using the values of the black pixels and then the black pixels are solved using the updated values of red pixels. Implementing red-black iterations efficiently has some challenges because the red-black access pattern is bad for memory coalescing. The block dimension was set to 16×16 and each block worked on a window of 32×32 pixels. At each iteration, the 16 threads of a thread row would first operate on the red pixels of two rows and then black pixels of the two rows. The solution pixels were stored in a shared memory buffer of dimensions 32×32 which is copied to and from the global memory in parallel and in a coalesced manner. There are 16 banks in the shared memory of devices with compute capability 1.3. The shared memory stores consecutive four-bytes values in consecutive banks. Due to these banks, shared memory accesses to 16 consecutive floats can be performed in parallel. Bank conflicts occur when two or more threads try to access the same bank. The default shared memory configuration resulted in bank conflicts because the interleaved access pattern of the red-black iterations caused two threads to access common banks. To avoid bank conflicts, the configuration was modified to 33×32 and the image columns $16 - 31$ were mapped to

$17 - 32$ columns of the shared memory buffer. Figure 4 details this shared memory configuration. There are still some bank conflicts in cases where the pixels of columns 15 and 16 need to access their right and left neighbors respectively. For the red-black implementation the above and bellow neighbors of the first and last rows were also loaded into shared memory. This is due to the fact that even though they are accessed only once, they could be loaded into the shared memory beforehand in coalesced manner, slightly improving the performance.

Figure 3 for kernel #5 shows the performance of the red-black kernel. The performance of this kernel is slower than the previous implementation because of the higher number of global memory accesses per thread, but our tests have shown that it converges faster than the previous implementation. Even with this faster convergence, we have found that the overall performance of the kernel #4 is faster than the performance of the red-black kernel. Therefore, we have chosen to use kernel #4 for the GPU solver.

V. Results

We have tested our implementation using two gigapixel-sized panorama datasets. We have also run the tests on two clusters: an *HP Cluster* and a *Dell Cluster*. The *HP Cluster* consists of nodes configured with 2.67GHz Xeon X5550 Processors (8 cores), 24GB of RAM and 2 NVIDIA Tesla T10 GPUs. The *Dell Cluster* is comprised of nodes configured with 2.5GHz Nehalem Processors (8 cores), 48GB of RAM and 2 NVIDIA Quadro FX 5800 GPUs. We have tested both strong and weak scalability of our implementation running from 2-60 nodes on the *HP Cluster* and 2-64 nodes on the *Dell Cluster*. To discount the effects due to other jobs running and accessing the shared resources on the cluster, the timings were taken as the best over several runs. The datasets used are:

- **Fall Panorama.** $126,826 \times 29,633$, 3.27-gigapixel. When tiled, this dataset is composed of 124×29 1024^2 sized windows. See Figure 5 for image results from a *HP Cluster* 32 node test run.
- **Winter Panorama.** $92,570 \times 28,600$, 2.65-gigapixel. When tiled, this dataset is composed of 91×28 1024^2 sized windows. See Figure 6 for image results from a *HP Cluster* 32 node test run.

HP Cluster. Strong and weak scaling tests were performed for 2-60 nodes. The strong scaling results are shown in Figures 7 and 8. We can see good strong scaling results for up to 16 nodes. Beyond 16 nodes the efficiency of the system falls to 50% due to the relatively small size of the image which is not able to keep the processors busy and hide the IO and message passing latencies. To see if the efficiency improves with a larger data set, we scaled the fall dataset to 12-gigapixels and ran strong scaling tests. As Figure 12 shows, we are getting a good efficiency of 64% even for 60 nodes. For weak scaling, the size of the image to be solved increases with the number of nodes. The image region to be solved was expanded from the center of the scaled dataset and the number of iterations for the solver were locked to 1000 for testing to discount variations in timings due to slower converging regions. As Figure 9 (left) shows, our implementation has good weak scalability and the efficency never falls bellow 80%.

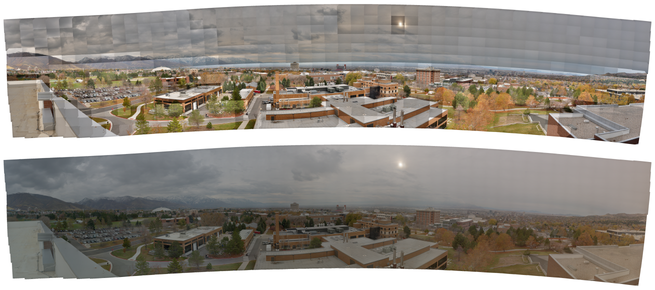


Figure 5. Fall Panorama - $126,826 \times 29,633$, 3.27-gigapixel. (top) The panorama before seamless blending and (bottom) the result of the parallel Poisson solver run on 32 nodes with 124×29 windows.

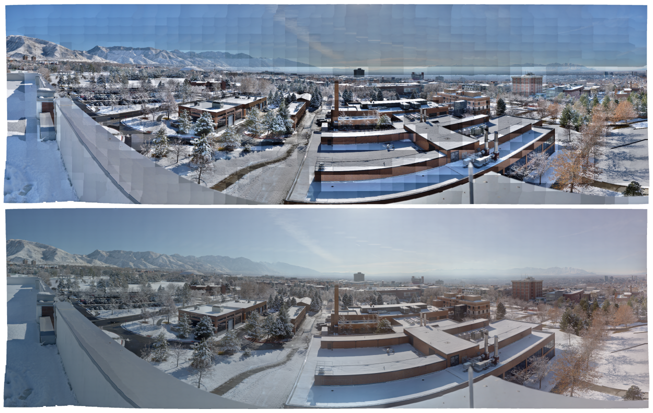


Figure 6. Winter Panorama - $92,570 \times 28,600$, 2.65-gigapixel. (top) The panorama before seamless blending, (bottom) the result of the parallel Poisson solver run on 32 nodes with 91×28 windows.

Dell Cluster. Both strong and weak scalability tests were also run on this system for 2-64 nodes. Figures 10 and 11 show the strong scaling results. Similar to the *HP Cluster*, good efficiency is maintained only up to 16 nodes. To test that this is due primarily to the size of the image, we ran the strong scaling tests on a up-scaled version of the data set and as expected we maintain good efficiency of, above 69% even for 64 nodes (see Figure 13). Figure 9 (right) shows the weak scaling results and similar to the *HP Cluster* we are getting good weak scalability.

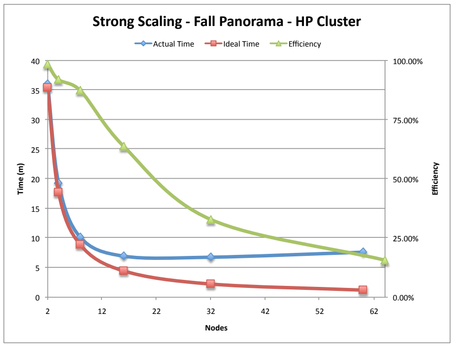
VI. Conclusions

This paper extends the framework described in [7] to utilize the processing power of GPUs, that are increasingly becoming common on high-end clusters. We have

Weak Scaling - *HP Cluster*					
Nodes	Cores	GPUs	Size (MP)	Time (m)	Efficiency
2	16	4	75.5	5.07	100.00%
4	32	8	150.99	6.20	81.77%
8	64	16	301.99	5.52	91.85%
16	128	32	603.98	5.63	90.05%
32	256	64	1,207.96	5.50	92.18%
60	480	120	2,264.92	5.72	88.64%

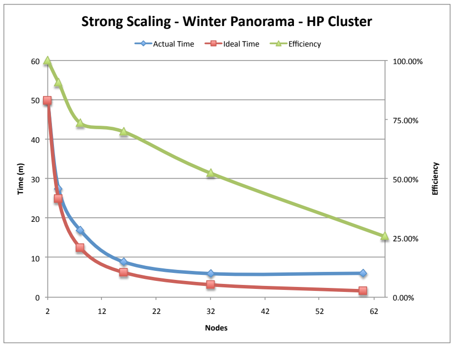
Weak Scaling - *Dell Cluster*					
Nodes	Cores	GPUs	Size (MP)	Time (m)	Efficiency
2	16	4	75.5	4.95	100.00%
4	32	8	150.99	5.35	92.52%
8	64	16	301.99	6.08	81.41%
16	128	32	603.98	5.50	90.00%
32	256	64	1,207.96	6.23	79.45%
64	512	128	2,415.92	5.97	82.91%

Figure 9. Weak scaling tests run on the *HP Cluster* (left) and *Dell Cluster* (right) for the Fall Panorama dataset. Our implementation shows good efficiency even when running on a large number of cores with many GPUs.



Strong Scaling - Fall Panorama - *HP Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
2	16	4	36.00	35.38	100.00%
4	32	8	19.27	17.69	93.41%
8	64	16	10.13	8.85	88.85%
16	128	32	6.95	4.42	64.75%
32	256	64	6.77	2.21	33.23%
60	480	120	7.62	1.18	14.76%

Figure 7. The strong scaling results for the Fall Panorama run on the *HP Cluster* from 2-60 nodes up to a total of 480 cores and 120 GPUs.



Strong Scaling - Winter Panorama - *HP Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
2	16	4	49.8	49.80	100.0%
4	32	8	27.45	24.90	90.71%
8	64	16	16.92	12.45	73.58%
16	128	32	8.90	6.23	69.94%
32	256	64	5.93	3.11	52.49%
60	480	120	10.28	1.56	25.85%

Figure 8. The strong scaling results for the Winter Panorama run on the *HP Cluster* from 2-60 nodes up to a total of 480 cores and 120 GPUs.

provided an efficient implementation of the Successive Over-Relaxation method for GPU which has increased the performance of the framework while maintaining its good strong and weak scalability for sufficiently large data sets. We have demonstrated the performance and scalability of our framework with results from runs on two clusters and two data sets.

Overall, this paper presents an efficient implementation of a hybrid, distributed gradient domain image processing framework that is capable of handling massive images.

Acknowledgment

This work is supported in part by the National Science Foundation awards OCI-0904631, OCI-0906379, IIS-1045032, and CCF-0702817. This work was also performed under the auspices of the U.S. Department of Energy by the University of Utah under contract DE-SC0001922 and DE-FC02-06ER25781 and by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

References

[1] GigaPan,, http://www.gigapan.org/about.php.

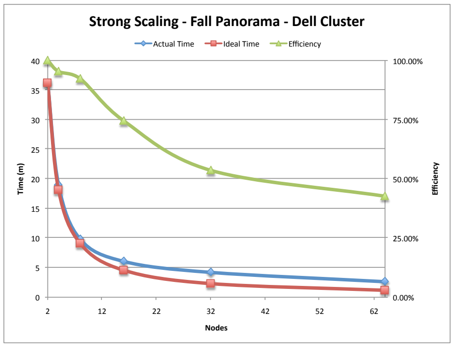
[2] USGS, united States Geological Survey http://www.usgs.gov/.

[3] HiRISE, , high Resolution Imaging Science Experiment http://hirise.lpl.arizona.edu/.

[4] M. Kazhdan and H. Hoppe, "Streaming multigrid for gradient-domain operations on large images," *ACM ToG.*, vol. 27, no. 3, 2008.

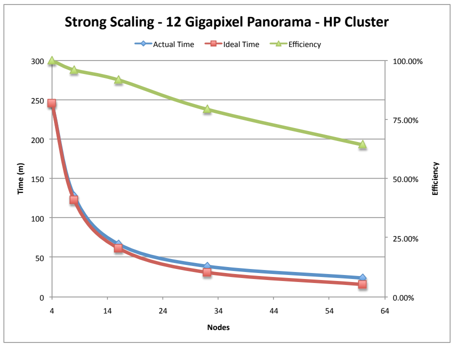
[5] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci, "Interactive editing of massive imagery made simple: Turning atlanta into atlantis," *ACM Trans. Graph.*, vol. 30, pp. 7:1–7:13, April 2011. [Online]. Available: http://doi.acm.org/10.1145/1944846.1944847

[6] M. Kazhdan, D. Surendran, and H. Hoppe, "Distributed gradient-domain processing of planar and spherical images," *ACM ToG. to appear*, 2010.



Strong Scaling - Fall Panorama - *Dell Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
2	16	4	36.17	36.17	100.00%
4	32	8	18.97	18.09	95.33%
8	64	16	9.80	9.04	92.27%
16	128	32	6.07	4.52	74.49%
32	256	64	4.22	2.26	53.57%
64	512	128	2.65	1.13	42.65%

Figure 10. The strong scaling results for the Fall Panorama run on the *Dell Cluster* from 2-64 nodes up to a total of 512 cores and 128 GPUs.



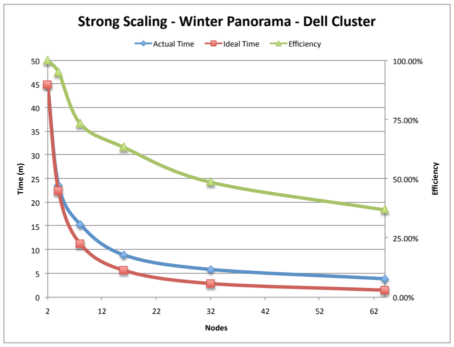
Strong Scaling - Scaled Fall Panorama - *HP Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
4	32	8	245.45	245.45	100.00%
8	64	16	127.83	122.73	96.00%
16	128	32	66.82	61.36	91.84%
32	256	64	38.62	30.68	79.45%
60	480	120	23.78	15.34	64.50%

Figure 12. The strong scaling results for the Scaled Fall Panorama run on the *HP Cluster* from 4-60 nodes.

[7] S. Philip, B. Summa, P.-T. Bremer, and V. Pascucci, "Parallel Gradient Domain Processing of Massive Images," in *Eurographics Symposium on Parallel Graphics and Visualization*, T. Kuhlen, R. Pajarola, and K. Zhou, Eds. Llandudno, Wales, UK: Eurographics Association, 2011, pp. 11–19.

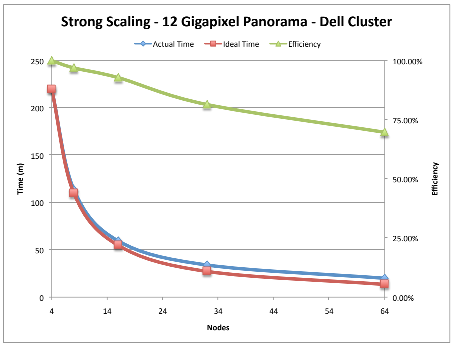
[8] P. Pérez, M. Gangnet, and A. Blake, "Poisson image editing," *ACM ToG.*, vol. 22, no. 3, pp. 313–318, 2003.

[9] J. Jia, J. Sun, C.-K. Tang, and H.-Y. Shum, "Drag-and-drop pasting," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*. New York, NY: ACM, 2006, pp. 631–637.



Strong Scaling - Winter Panorama - *Dell Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
2	16	4	44.80	44.80	100.00%
4	32	8	23.58	22.40	95.00%
8	64	16	15.27	11.20	73.35%
16	128	32	8.83	5.60	63.42%
32	256	64	5.75	2.80	48.70%
64	512	128	3.78	1.40	37.04%

Figure 11. The strong scaling results for the Winter Panorama run on the *Dell Cluster* from 2-64 nodes up to a total of 512 cores and 128 GPUs.



Strong Scaling - Scaled Fall Panorama - *Dell Cluster*					
Nodes	Cores	GPUs	Actual (m)	Ideal (m)	Efficiency
4	32	8	219.77	219.77	100.00%
8	64	16	113.35	109.88	96.94%
16	128	32	59.18	54.94	92.83%
32	256	64	33.73	27.47	81.44%
64	512	128	19.70	13.74	69.72%

Figure 13. The strong scaling results for the Scaled Fall Panorama run on the *Dell Cluster* from 4-64 nodes.

[10] J. Sun, J. Jia, C.-K. Tang, and H.-Y. Shum, "Poisson matting," *ACM ToG.*, vol. 23, no. 3, pp. 315–321, 2004.

[11] A. Levin, A. Zomet, S. Peleg, and Y. Weiss, "Seamless image stitching in the gradient domain," in *In Eighth European Conference on Computer Vision (ECCV 2004*. Springer, 2004, pp. 377–389.

[12] A. Agarwala, M. Dontcheva, M. Agrawala, S. Drucker, A. Colburn, B. Curless, D. Salesin, and M. Cohen, "Interactive digital photomontage," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004.

[13] R. Fattal, D. Lischinski, and M. Werman, "Gradient domain high dynamic range compression," in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 2002.

[14] B. K. P. Horn, "Determining lightness from an image," *Comput. Graphics Image Processing*, vol. 3, no. 1, pp. 277–299, Dec. 1974.

[15] Y. Weiss, "Deriving intrinsic images from image sequences," in *International Conference on Computer Vision*, 2001, pp. 68–75.

[16] G. D. Finlayson, S. D. Hordley, and M. S. Drew, "Removing shadows from images," in *ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part IV*. London, UK: Springer-Verlag, 2002, pp. 823–836.

[17] A. Agrawal, R. Raskar, S. K. Nayar, and Y. Li, "Removing photography artifacts using gradient projection and flash-exposure sampling," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*. New York, NY, USA: ACM, 2005, pp. 828–835.

[18] J. McCann and N. S. Pollard, "Real-time gradient-domain painting," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*. New York, NY, USA: ACM, 2008, pp. 1–7.

[19] R. W. Hockney, "A fast direct solution of Poisson's equation using Fourier analysis," *Journal of the ACM*, vol. 12, no. 1, pp. 95–113, Jan. 1965.

[20] A. K. Agrawal, R. Chellappa, and R. Raskar, "An algebraic approach to surface reconstruction from gradient fields," in *ICCV*, 2005, pp. I: 174–181.

[21] A. K. Agrawal, R. Raskar, and R. Chellappa, "What is the range of surface reconstructions from a gradient field?" in *ECCV*, 2006, pp. I: 578–591.

[22] J. Kopf, M. F. Cohen, D. Lischinski, and M. Uyttendaele, "Joint bilateral upsampling," *ACM ToG*, vol. 26, no. 3, p. 96, 2007.

[23] F. W. Dorr, "The direct solution of the discrete poisson equation on a rectangle," *SIAM Review*, vol. 12, no. 2, pp. 248–263, April 1970.

[24] Heath, Ng, and Peyton, "Parallel algorithms for sparse linear systems," *SIREV: SIAM Review*, vol. 33, 1991.

[25] O. Axelsson, *Iterative Solution Methods*. New York, NY: Cambridge Universty Press, 1994.

[26] R. Szeliski, M. Uyttendaele, and D. Steedly, "Fast poisson blending using multi-splines," in *Computational Photography (ICCP), 2011 IEEE International Conference on*, april 2011, pp. 1 –8.

[27] S. Gortler and M. Cohen, "Variational modeling with wavelets," in *Symposium on Interactive 3D graphics*, 1995, pp. 35–42.

[28] R. Szeliski, "Locally adapted hierarchical basis preconditioning," *ACM ToG.*, vol. 27, no. 3, pp. 1135–1143, 2008.

[29] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[30] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed. SIAM, 2000.

[31] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal Computational Physics*, vol. 82, pp. 64–84, 1989.

[32] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Eurographics Symposium on Geometry Processing*, 2006, pp. 61–70.

[33] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe, "Multilevel streaming for out-of-core surface reconstruction," in *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 69–78.

[34] A. Agarwala, "Efficient gradient-domain compositing using quadtrees," in *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*. New York, NY, USA: ACM, 2007, p. 94.

[35] P. M. Ricker, "A direct multigrid poisson solver for oct-tree adaptive meshes," *The Astrophysical Journal Supplement Series*, vol. 176, pp. 293–300, 2008.

[36] M. Kazhdan, "Reconstruction of solid models from oriented point sets," in *Eurographics Symposium on Geometry Processing*, 2005, pp. 73–82.

[37] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A survey of parallelization techniques for multigrid solvers," in *Parallel Processing for Scientific Computing*, ser. Software, Environments, and Tools, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds. Philadelphia: SIAM, Nov. 2006, vol. 20, pp. 179–201, ch. 10,.

[38] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," in *External memory algorithms*, ser. Dimacs Series In Discrete Mathematics And Theoretical Computer Science. Boston, MA: American Mathematical Society, 1999, pp. 161–179.

[39] J. Stookey, Z. Xie, B. Cutler, W. R. Franklin, D. M. Tracy, and M. V. A. Andrade, "Parallel ODETLAP for terrain compression and reconstruction," in *GIS*, W. G. Aref, M. F. Mokbel, and M. Schneider, Eds. ACM, 2008, p. 17.

[40] H. T. Vo, D. K. Osmari, B. Summa, J. L. D. Comba, V. Pascucci, and C. T. Silva, "Streaming-enabled parallel dataflow architecture for multicore systems," *Comput. Graph. Forum*, vol. 29, no. 3, pp. 1073–1082, 2010.

[41] H. T. Vo, D. Osmari, L. Scheidegger, J. Comba, J. Shepherd, and C. Silva, "Poster i06 - hyperflow: An efficient dataflow architecture for multi cpu-gpu systems," in *NVIDIA Research Summit*, 2010.

[42] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, 2010, version 3.2.