ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/271444531

A fast parallel implementation of queue-based morphological reconstruction using gpus

Conference Paper · October 2012

CITATIONS
9

READS
495

1 author:

George Teodoro
Federal University of Minas Gerais

150 PUBLICATIONS **1,816** CITATIONS

SEE PROFILE

All content following this page was uploaded by George Teodoro on 27 January 2015.

The user has requested enhancement of the downloaded file.

A Fast Parallel Implementation of Queue-based Morphological Reconstruction using GPUs

George Teodoro, Tony Pan, Tahsin M. Kurc,
Lee Cooper, Jun Kong, and Joel H. Saltz
{gteodor,tony.pan,tkurc,lee.cooper,jun.kong,jhsaltz}@emory.edu

Center for Comprehensive Informatics, Emory University, Atlanta, GA 30322

Abstract. In this paper we develop and experimentally evaluate a novel GPU-based implementation of the morphological reconstruction operation. This operation is commonly used in the segmentation and feature computation steps of image analysis pipelines, and often serves as a component in other image processing operations. Our implementation builds on a fast hybrid CPU algorithm, which employs a queue structure for efficient execution, and is the first GPU-enabled version of the queue-based hybrid algorithm. We evaluate our implementation using state-of-the-art GPU accelerators and images obtained by high resolution microscopy scanners from whole tissue slides. The experimental results show that our GPU version achieves up to 20× speedup as compared to the sequential CPU version. Additionally, our implementation's performance is superior to the previously published GPU-based morphological reconstruction, which is built on top of slower baseline version of the operation.

1 Introduction

Image data and image analysis have applications in a wide range of domains, including satellite data analyses, astronomy, computer vision, and biomedical research. In biomedical research, for instance, digitized microscopy imaging plays a crucial role in quantitative characterization of biological structures in great detail at cellular and sub-cellular levels. Instruments to capture high resolution images from tissue slides and tissue microarrays have advanced significantly in the past decade. A state-of-the-art slide scanner can capture a 50K×50K pixel image in a few minutes from a tissue sectioned at 3-5 micron thickness, enabling the application of this technology in research and healthcare delivery. Systems equipped with auto-focus mechanisms can process batches of slides with minimal manual intervention and are making large databases of high resolution slides feasible.

In a brain tumor studies, for instance, images created using these devices may contain upto 20 billion pixels (with digitization at 40X objective magnification), or approximately 56 GB in size when represented as 8 bit uncompressed RGB format. To extract meaningful information from these images, they usually are processed through a series of steps such as color normalization, segmentation, feature computation, and classification in order to extract anatomic structures at cellular and sub-cellular levels and classify them [1]. The analysis steps consist

of multiple data and compute intensive operations. The processing of a large image can take tens of hours.

On the computational hardware front, GPGPUs have become a popular implementation platform for science applications. There is a shift towards heterogeneous high performance computing architectures consisting of multi-core CPUs and multiple general-purpose graphics processing units (GPGPUs). Image analysis algorithms and underlying operations, nevertheless, need to take advantage of GPGPUs to fully exploit the processing of such systems.

In this paper, we develop and experimentally evaluate an efficient GPU-based implementation of the morphological reconstruction operation. Morphological reconstruction is commonly used in the segmentation, filtering, and feature computation steps of image analysis pipelines. Morphological reconstruction is essentially an iterative dilation of a marker image, constrained by the mask image, until stability is reached (the details of the operation are described in Section 2). The processing structure of morphological reconstruction is also common in other application domains. The techniques for morphological reconstruction can be extended to such applications as the determination of Euclidean skeletons [8] and skeletons by influence zones [5]. Additionally, Delaunay triangulations [10], Gabriel graphs [2] and relative neighborhood graphs [11] can be derived from these methods, and obtained in arbitrary binary pictures [12].

2 Morphological Reconstruction Algorithms

A number of morphological reconstruction algorithms have been developed and evaluated by Vincent [13]. We refer the reader to his paper for a more formal definition of the operation and the details of the algorithms. In this section we briefly present the basics of morphological reconstruction, including definitions and algorithm versions.

Morphological reconstruction algorithms were developed to both binary and gray scale images. In gray scale images the value of a pixel p, $I(p)$, comes from a set $\{0, ..., L-1\}$ of gray levels in a discrete or continuous domain, while in binary images there are only two levels. Figure 1 illustrates the process of gray scale morphological reconstruction in 1-dimension. The marker intensity profile (red) is propagated spatially but is bounded by the mask image's intensity profile (blue). In a simplified form, the reconstruction $\rho_I(J)$ of mask I from marker image J is done by performing elementary dilations (i.e., dilations of size 1) in J by a structuring element G. Here, G is a discrete grid, which provides the neighborhood relationships between pixels. A pixel p is a neighbor of pixel q if and only if $(p, q) \in G$. G is usually a 4-connected or 8-connected grid. An elementary dilation from a pixel p corresponds to propagation from p to its immediate neighbors in G. The basic algorithm carries out elementary dilations successively over the entire image J, updates each pixel in J with the pixelwise minimum of the dilation's result and the corresponding pixel in I (i.e., $J(p) \leftarrow (max\{J(q), q \in N_G(p) \cup \{p\}\}) \wedge I(p)$; where $N_G(p)$ is the neighborhood of a pixel p on grid G and $\wedge$ is the pixelwise minimum operator), and stops when *stability* is reached, i.e., when no more pixel values are modified.

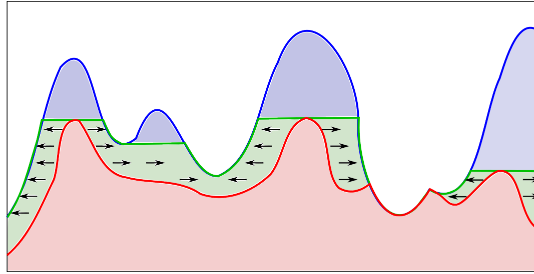

Fig. 1. Gray scale morphological reconstruction in 1-dimension. The marker image intensity profile is represented as the red line, and the mask image intensity profile is represented as the blue line. The final image intensity profile is represented as the green line. The arrows show the direction of propagation from the marker intensity profile to the mask intensity profile. The green region shows the changes introduced by the morphological reconstruction process.

Vincent has presented several morphological reconstruction algorithms, which are based on this core technique [13]:

Sequential Reconstruction (SR): Pixel value propagation in the marker image is computed by alternating raster and anti-raster scans. A raster scan starts from the pixel at $(0, 0)$ and proceeds to the pixel at $(N - 1, M - 1)$ in a row-wise manner. An anti-raster scan starts from the pixel at $(N - 1, M - 1)$ and moves to the pixel at $(0, 0)$ in a row-wise manner. Here, N and M are the resolutions of the image in x and y dimensions, respectively. In each scan, values from pixels in the upper left or the lower right half neighborhood are propagated to the current pixel in raster or anti-raster fashion, respectively. The raster and anti-raster scans allow for changes in a pixel to be propagated in the current iteration. The SR method iterates until stability is reached, i.e., no more changes in pixels are computed.

Queue-based Reconstruction (QB): In this method, a first-in first-out (FIFO) queue is initialized with pixels in the regional maxima. The computation then proceeds by removing a pixel from the queue, scanning the pixel's neighborhood, and queuing the neighbor pixels that changed. The overall process continues until the queue is empty. The regional maxima needed to initialize the queue requires significant computational cost to generate.

Fast Hybrid Reconstruction (FH): This approach incorporates the characteristics of SR and QB algorithms, and is about one order of magnitude faster than the others. It first makes one pass using the raster and anti-raster scans as in SR. After that pass, it continues the computation using a FIFO queue as in QB.

A pseudo-code implementation of FH is presented in Algorithm 1, N^+ and N^- denote the set of neighbors in $N_G(p)$ that are reached before and after touching pixel p during a raster scan.

Algorithm 1 Fast Hybrid gray scale reconstruction

Input
I**:** *mask image*
J**:** *marker image, defined on domain* $D_I, J \leq I$.

1: Scan D_I in raster order.
2: Let p be the current pixel
3: $J(p) \leftarrow (max\{J(q), q \in N_G^+(p) \cup \{p\}\}) \wedge I(p)$
4: Scan D_I in anti-raster order.
5: Let p be the current pixel
6: $J(p) \leftarrow (max\{J(q), q \in N_G^-(p) \cup \{p\}\}) \wedge I(p)$
7: **if** $\exists q \in N_G^-(p) \mid J(q) < J(p)$ *and* $J(q) < I(q)$
8: fifo_add(p)
9: {**Queue-based propagation step**}
10: **while** fifo_empty() = false **do**
11: $p \leftarrow$ fifo_first()
12: **for all** $q \in N_G(p)$ **do**
13: **if** $J(q) < J(p)$ and $I(q) \neq J(q)$ **then**
14: $J(q) \leftarrow min\{J(p), I(q)\}$
15: fifo_add(q)

3 Fast Parallel Hybrid Reconstruction Using GPUs

To the best of our knowledge, the only existing GPU implementation of morphological reconstruction is a modified version of SR [4]: SR_GPU. As discussed before, SR is about an order of magnitude slower than FH. Therefore, SR_GPU is built on top of a slow base line algorithm, limiting its gains when compared to the fastest CPU algorithm — FH. This section describes our GPU parallelization of the Fast Hybrid Reconstruction algorithm, referred to in this paper as FH_GPU.

The FH_GPU, as its CPU based counterpart, consists of a raster and anti-raster scan phase that is followed by a queue-based phase. To implement the raster and anti-raster scanning in our algorithm we extended the approach of Karas [4]. We have made several changes to SR_GPU: (1) we have templated our implementation to support binary images as well as gray scale images with integer and floating point data types; (2) we have optimized the implementation and multi-threaded execution for 2-dimensional images — the original implementation was designed for 3-dimensional images. This modification allows us to specialize the Y-direction propagation during the scans; and (3) we have tuned the shared memory utilization to minimize communication between memory hierarchies. For a detailed description of the GPU-based raster scan phase we direct the reader to the paper [4]. The rest of this section discusses the queue-based phase, which is the key aspect to achieve efficiency on execution of morphological reconstruction and is the focus of this work.

3.1 Queue-based Phase Parallelization Strategy

After the raster scan phase, *active pixels*, those that satisfy the propagation condition to a neighbor pixels, are inserted into a global queue for computation.

The global queued is then equally partitioned into a number of smaller queues. Each of these queues is assigned to a GPU thread block. Each block can carry out computations independently of the other blocks.

Two levels of parallelism can be implemented in the queue-based computation performed by each thread of block: (i) pixel level parallelism which allows pixels queued for computation to be independently processed; and (ii) the neighborhood level parallelism that performs the concurrent evaluation of a pixel neighborhood. The parallelism in both cases, however, is available at the cost of dealing with potential race conditions that may occur when a pixel is updated concurrently.

Algorithm 2 GPU-based queue propagation phase

```
 1: {Split initial queue equally among thread blocks}
 2: while queue_empty() = false do
 3:   while (p = dequeue(...))! = EMPTY do in parallel
 4:     for all q ∈ N_G(p) do
 5:       if J(q) < J(p) and I(q) ≠ J(q) then
 6:         oldval = atomicMax(&J(q), min{J(p), I(q)})
 7:         if oldval < min{J(p), I(q)} then
 8:           queue_add(q)
 9:   if tid = 0 then
10:     queue_swap_in_out()
11:   __syncthreads()
12:   __threadfence_block()
```

During execution, a *value max* operation is performed on the value of the current pixel being processed (p) and that of each pixel in the neighborhood ($q \in N_G(p)$) — limited by a fixed mask value $I(q)$. To correctly perform this computation, atomicity is required in the maximum and update operations. This can only be achieved at the cost of extra synchronization overheads. The use of atomic CAS operations is sufficient to solve this race condition. The efficiency of atomic operations in GPUs have been significantly improved in the last generation of NVidia GPUs (Fermi) [9] because of the use of cache memory. Atomic operations, however, still are more efficient in cases where threads do not concurrently try to update the same memory address. When multiple threads attempt to update the same memory location, they are serialized in order to ensure correct execution. As a result, the number of operations successfully performed per cycle is reduced [9].

To lessen the impact of the potential serial execution, our GPU parallelization employs the pixel level parallelism only. The neighborhood level parallelism is problematic in this case because the concurrent processing and updating of pixels in a given neighborhood likely affect one another, since the pixels are located in the same region of the image. Unlike traditional graph computing algorithms (e.g., Breadth-First Search [3, 6]), the lack of parallelism in neighborhood processing does not result in load imbalance in the overall execution, because all pixels have the same number of neighbors which is defined by the structuring element G.

The GPU-based implementation of the queue propagation operation is presented in Algorithm 2. After splitting the initial queue, each block of threads enters into a loop in which pixels are dequeued in parallel and processed, and new pixels may be added to the local queue as needed. This process continues until the queue is empty. Within each loop, the pixels queued in last iteration are uniformly divided among the threads in the block and processed in parallel (Lines 3—8 in Algorithm 2). The value of each queued pixel p is compared to every pixel q in the neighborhood of pixel p. An atomic maximum operation is performed when a neighbor pixel q should be updated. The value of pixel q before the maximum operation is returned (*oldval*), and used to determine whether the pixel's value has really been changed (Line 7 in Algorithm 2). This step is necessary because there is a chance that another thread might have changed the value of q between the time the pixel's value is read to perform the update test and the time that the atomic operation is performed (Lines 5 and 6 in Algorithm 2). If the maximum operation performed by the current thread has changed the value of pixel q, q is added to the queue for processing in the next iteration of the loop. Even with this control, it may happen that between the test in line 7 and the addition to the queue, the pixel q may have been modified again. In this case, q is added multiple times to the queue, and although it may impact the performance, the correctness of the algorithm is not affected.

After computing pixels from the last iteration, pixels that are added to the queue in the current iteration are made available for the next iteration, and all writes performed by threads in a block are guarantee to be consistent (Lines 9 to 12 in the algorithm). As described in the next section, the choice to process pixels in rounds, instead of making each pixel inserted into the queue immediately accessible, is made in our design to implement a queue with very efficient read performance (dequeue performance) and with low synchronization costs to provide consistency among threads when multiple threads read from and write to the queue.

3.2 Parallel Queue Implementation and Management

The parallel queue is a core data structure employed by FH_GPU. It is used to store information about pixels that are active in the computation and should be processed. An efficient parallel queue for GPUs is a challenging problem [3, 4, 6] due to the sequential and irregular nature of accesses.

A straight forward implementation of a queue (named Naïve here), as presented by Hong et al. [3], could be done by employing an array to store items in sequence and using atomic additions to calculate the position where each item should be inserted. Hong et al. stated that this solution worked well for their use case. The use of atomic operations, however, is inefficient when the queue is heavily employed as in our algorithm. Moreover, a single queue that is shared among all thread blocks introduces additional overheads to guarantee consistency across threads in the entire device.

We have designed a parallel queue that operates independently in a per thread block basis to avoid inter-block communication, and is built with a cascade of storage levels in order to exploit the GPU fast memories for efficient

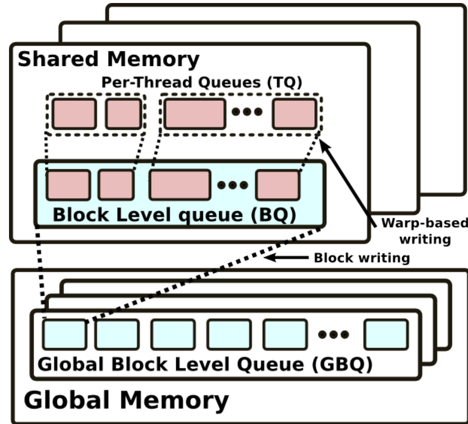

Fig. 2. Multi-level parallel queue.

write and read accesses. The parallel queue (depicted in Figure 2) is constructed in multiple-levels:(i) Per-Thread queues (TQ) which are very small queues private to each thread, residing in the shared memory; (ii) Block Level Queue (BQ) which is also in the shared memory, but is larger than TQ. Write operations to the block level queue are performed in thread warp-basis; and (iii) Global Block Level Queue (GBQ) which is the largest queue and uses the global memory of the GPU to accumulate data stored in BQ when the size of BQ exceeds the shared memory size.

The use of multiple levels of queues improves the scalability of the implementation, since portions of the parallel queue are utilized independently and the fast memories are employed more effectively. In our queue implementation, each thread maintains an independent queue (TQ) that does not require any synchronization to be performed. In this way threads can efficiently store pixels from a given neighborhood in the queue for computation. Whenever this level of queue is full, it is necessary to perform a warp level communication to aggregate the items stored in local queues and write them to BQ. In this phase, a parallel thread warp prefix-sum is performed to compute the total number of items queued in individual TQs. A single shared memory atomic operation is performed at the Block Level Queue to identify the position where the warp of threads should write their data. This position is returned, and the warp of threads write their local queues to BQ.

Whenever a BQ is full, the threads in the corresponding thread block are synchronized. The current size of the queue is used in a single operation to calculate the offset in GBQ from which the BQ should be stored. After this step, all the threads in the thread block work collaboratively to copy in parallel the contents of QB to GBQ. This data transfer operation is able to achieve high throughput, because the memory accesses are coalesced. It should be noted that, in all levels of the parallel queue, an array is used to hold the queue content. While the contents of TQ and BQ are copied to the lower level queues when full, GBQ does not have a lower level queue to which its content can be copied. In the current implementation, the size of GBQ is initialized with a

fixed tunable memory space. If the limit of GBQ is reached during an iteration of the algorithm, excess pixels are dropped and not stored. The GPU method returns a boolean value to CPU to indicate that some pixels have been dropped during the execution of the method kernel. In that case, the algorithm has to be re-executed, but using the output of the previous execution as input, because this output already holds a partial solution of the problem. Hence, recomputing using the output of the previous step as input will result in the same solution as if the execution was carried out in a single step. The operation of reading items from the queue for computation is embarrassingly parallel, as we statically partitioning pixels queued in the beginning of each iteration.

4 Experimental Results

We evaluate the performance of the GPU-enabled fast hybrid morphological reconstruction algorithm (FH_GPU), comparing it to that of existing fastest CPU and GPU implementations under different configurations. Test images used in the experiments were extracted from the set of images collected from a brain tumor research study. Each test image had been scanned at 20× magnification resulting in roughly 50K×50K pixels. The images used in this studies are partitioned into *tiles* to perform image analyses on a parallel machine. A tile is processed by a single GPU or a CPU core. Hence, the performance numbers presented in this paper are per individual tiles.

We used a system with a contemporary CPU (Intel i7 2.66 GHz) and two NVIDIA GPUs (C2070 and GTX580). Codes used in our evaluation were compiled using using "gcc 4.2.1", "-O3" optimization flag, and NVidia CUDA SDK 4.0. The experiments were repeated 3 times, and, unless stated, the standard deviation was not observed to be higher than 1.3%.

4.1 Results

The evaluation of morphological reconstruction, as the input data size varies, using 4-connected and 8-connected grids is presented in Table 1. The speedups are calculated using the single core CPU version as the baseline.

The results show that both SR_GPU and FH_GPU achieve higher speedups with 8-connected grids. The performance differences are primarily a consequence of the GPU's ability to accommodate the higher bandwidth demands of morphological reconstruction with 8-connected grid, thanks to the higher random memory access bandwidth of the GPU. For example, when the input tile is 4K×4K, the CPU throughput, measured as a function of the number of pixels visited and compared per second, increases from 67 to 85 million pixels per second as the grid changes from 4-connected to 8-connected. For the same scenario, the FH_GPU has a much better throughput improvement, going from 746 to 1227 million pixels per second. This difference in computing rates highlights that morphological reconstruction has abundant parallelism primarily in memory operations (like other graph algorithms [3, 7]), which limits the performance of the CPU for the 8-connected case. The use of an 8-connected grid is usually

Input Size	FH CPU(ms)	SR_GPU (ms) C2070	SR_GPU (ms) GTX 580	FH_GPU (ms) C2070	FH_GPU (ms) GTX 580	SR_GPU speedup C2070	SR_GPU speedup GTX 580	FH_GPU speedup C2070	FH_GPU speedup GTX 580
4-connected grid									
4K×4K	1990	1578	1007	281	181	1.26	1.97	7.08	10.99
8K×8K	8491	5393	3404	1034	646	1.57	2.49	8.21	13.14
12K×12K	19174	11773	6688	2334	1350	1.62	2.86	8.21	14.2
16K×16K	34818	19278	10990	3960	2256	1.80	3.16	8.79	15.43
8-connected grid									
4K×4K	1314	599	401	169	115	2.19	3.27	7.77	11.42
8K×8K	5267	1646	1089	502	316	3.19	4.83	10.49	16.66
12K×12K	11767	3156	1952	1023	656	3.72	6.02	11.5	17.93
16K×16K	20965	5076	3088	1736	1078	4.13	6.78	12.07	19.44

Table 1. Impact of the input tile size on the performance of (i) FH: the fastest CPU version, (ii) SR_GPU: the previous GPU implementation [4], and (iii) FH_GPU: our queue-based GPU version.

more beneficial for SR_GPU, since the propagation of pixel values is more effective for larger grids and, as a result, fewer raster scans are needed to achieve stability.

The performance analysis for FH_GPU under the scenario with input data size variation is also interesting. As larger input tiles are processed, maintaining the number of threads fixed, the algorithm increases its efficiency. The change from 4K×4K to 8K×8K tiles improved the throughput from about 1227 to 1533 millions of pixels visited and compared per second — using 8-connected grid and the GTX 580. The throughput increase is consequence of a better utilization of the GPU computing power, because of the smaller amortized synchronization costs at the end of each iteration. The speedup on top of the CPU version is also higher, since there is no throughput improvement for the CPU as the tile size increases.

For tiles larger than 8K×8K, no improvement in throughput is observed for FH_GPU either. Small increases in speedup values compared to the CPU version are observed for the 12K×12K and 16K×16K image tiles. These gains are consequence of the better performance achieved by the multiple iterations of the raster scan phase, which is the initial phase in FH_GPU before the queue-based execution phase. The raster scans have a better performance for larger image tiles because the overhead of launching the 4 GPU *kernels* used by the raster scan phase is better amortized since more pixels are modified per pass.

5 Conclusions

We have presented an implementation of a fast hybrid morphological reconstruction algorithm for GPUs. Unlike a previous implementation, the new GPU algorithm is based on an efficient sequential algorithm and employs a queue-based computation stage to speed up processing. Our experimental evaluation on two state-of-the-art GPUs and using high resolution microscopy images show that (1) multiple levels of parallelism can be leveraged to implement an efficient queue and (2) a multi-level queue implementation allows for better utilization of fast memory hierarchies in a GPU and reduces synchronization overheads.

Acknowledgments. This research was funded, in part, by grants from the National Institutes of Health through contract HHSN261200800001E by the National Cancer Institute; and contracts 5R01LM009239-04 and 1R01LM011119-01 from the National Library of Medicine, R24HL085343 from the National Heart Lung and Blood Institute, NIH NIBIB BISTI P20EB000591, RC4MD005964 from National Institutes of Health, and PHS Grant UL1TR000454 from the Clinical and Translational Science Award Program, National Institutes of Health, National Center for Advancing Translational Sciences. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH. We also want to thank Pavel Karas for releasing the SR_GPU implementation used in our comparative evaluation.

References

1. L. Cooper, J. Kong, D. Gutman, F. Wang, S. Cholleti, T. Pan, P. Widener, A. Sharma, T. Mikkelsen, A. Flanders, D. Rubin, E. Van Meir, T. Kurc, C. Moreno, D. Brat, and J. Saltz. An Integrative Approach for In Silico Glioma Research. *IEEE Transactions on Biomedical Engineering*, oct. 2010.
2. R. K. Gabriel and R. R. Sokal. A New Statistical Approach to Geographic Variation Analysis. *Systematic Zoology*, 18(3):259–278, Sept. 1969.
3. S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 2011.
4. P. Karas. Efficient Computation of Morphological Greyscale Reconstruction. In *MEMICS*, volume 16 of *OASICS*, pages 54–61. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.
5. C. Lantuéjoul. Skeletonization in quantitative metallography. In R. Haralick and J.-C. Simon, editors, *Issues in Digital Image Processing*, volume 34 of *NATO ASI Series E*, pages 107–135, 1980.
6. L. Luo, M. Wong, and W.-m. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, 2010.
7. A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.
8. F. Meyer. Digital Euclidean Skeletons. In *SPIE Vol. 1360, Visual Communications and Image Processing*, pages 251–262, 1990.
9. Nvidia. *Fermi Compute Architecture Whitepaper*.
10. F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
11. G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.
12. L. Vincent. Exact Euclidean Distance Function By Chain Propagations. In *IEEE Int. Computer Vision and Pattern Recog. Conference*, pages 520–525, 1991.
13. L. Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Transactions on Image Processing*, 2:176–201, 1993.

View publication stats