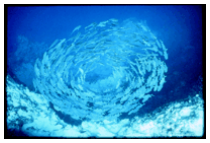


Accelerated Simulation of an Individual-Based Fish Schooling Model on a Graphics Processing Unit

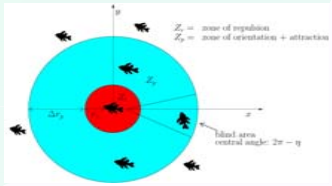
Hong Li[1], Allison Kolpas[2], Linda R. Petzold[1,3], J. Moehlis[3]

[1]Department of Computer Science; [2]Department of Ecology, Evolution, and Marine Biology; [3]Department of Mechanical Engineering
University of California Santa Barbara

Abstract

Many organisms move collectively in groups, such as schools of fish, flocks of birds, and herds of wildebeest. Individual-based models for group formation help us understand how patterns of motion at the population level emerge from interactions at the individual level. The usefulness of such models, which can incorporate detailed experimental observations of the behavior of individual organisms, is limited by the ability to simulate them in a reasonable amount of time. As a result of recent advances in graphics processing units (GPUs) hardware and software architecture, tremendous computational resources are available at a very low cost. We briefly review an individual-based model for fish schooling and demonstrate the use of GPUs for parallelization of the model within one realization and across multiple realizations. Our GPU-based simulations are typically 200 times faster than optimized CPU-based simulations. This technology has enabled us to better understand how schools form, generate patterns, and transfer information, by allowing the simulation of more detailed models for larger group sizes over longer periods of time.

Fish Schooling Model



Interaction Rules

- Each agent travels at constant speed s
- Each time step τ, agents determine a new direction of travel based on neighbors in behavioral zones
- If agents in zone of repulsion,

$$v_i(t+\tau) = -\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|}$$

- if agents in zone of orientation + attraction

$$v_i(t+\tau) = \omega_0 \frac{\sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|}}{\left| \sum_{j \neq i} \frac{p_j(t) - p_i(t)}{|p_j(t) - p_i(t)|} \right|} + \omega_0 \frac{\sum_j \hat{v}_j(t)}{\left| \sum_j \hat{v}_j(t) \right|}$$

- if contributions sum to zero or no influences

$$v_i(t+\tau) = v_i(t)$$

- Normalize:

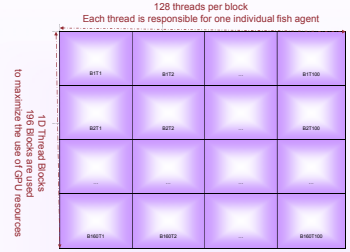
$$\hat{v}_i(t+\tau) = \frac{v_i(t+\tau)}{|v_i(t+\tau)|}$$

- Add noise
- Update simultaneously:

$$p_i(t+\tau) = p_i(t) + s\hat{v}_i(t+\tau)\tau$$

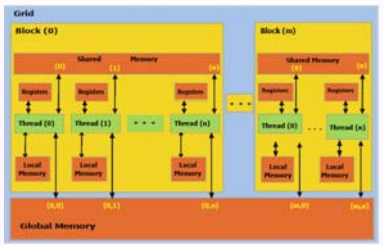
- Explore effects of varying $r = \omega_r/\omega_a$

Problem Decomposition



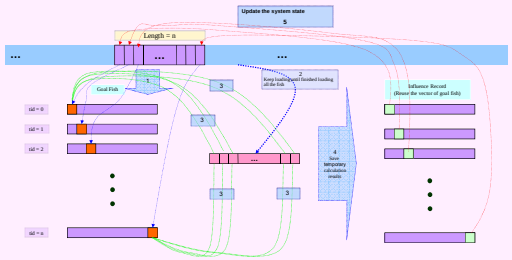
To accurately determine statistical properties of the collective motion of fish, many realizations of the fish schooling model are typically required for a given set of parameters. This can be very computationally intensive. There are essentially two ways to improve the performance: parallelize the simulation across the realizations, and parallelize the simulation within one realization. We do both.

Parallelization Across Realizations



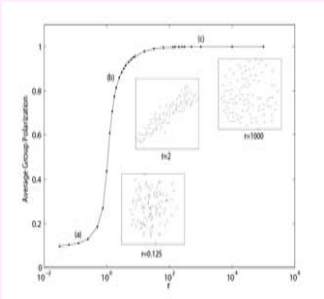
Parallelizing the independent realizations is an effective way to improve the performance. Ensembles of fish schooling runs are very well-suited for implementation on the GPU through the CUDA.

Parallelization Within One Realization



Memory layout of the fish schooling model. There are n threads to simulate one realization with thread id tid identifying the thread. In process 1, each thread loads one fish as its "goal fish" from the device memory to shared memory; In process 2, each thread loads fish to another array in shared memory; In process 3, each thread uses the full data from process 2 to compute the influences on its own "goal fish"; Processes 2 and 3 continue until all of the influences to the "goal fish" have been computed; In process 4, each thread saves its influence to the influence record array (this reuses the previous array to save shared memory); In process 5, each thread updates its "goal fish" to the system state vector on the device.

Simulation Results



Group polarization as a function of r averaged over 1120 steady-state replicate simulations run in parallel on the GPU. (a) A realization of the swarm state (r = 0.125), (b) a realization of the dynamic parallel state (r = 2), (c) a realization of the highly parallel state (r = 1000)

r	Sequential simulation time	GPU time	Speed-up
0.03125	7924067.2	33750.6	234.8
0.0625	7925788.6	33767.7	234.7
0.125	7908017.5	33750.6	234.3
0.25	7938803.4	33790.0	235.0
0.5	7934238.6	33774.8	235.0
1	7962735.7	33855.3	235.0
2	7969067.5	33755.0	233.4
4	8008521.9	33859.2	231.0
8	8091104.2	33681.4	236.6
16	8091787.8	33706.8	240.2

Performance comparison for different parameters. on the three test problems.

References

[1] Camazine S, Deneubourg JL, Franks NR, Sneyd J, Theraulaz G, Bonabeau E. Self-organization in Biological Systems. Princeton University Press: Princeton, NJ, 2003.
[2] Couzin ID, Krause J, James R, Ruxton GD, Franks NR. Collective memory and spatial sorting in animal groups. Journal of Theoretical Biology 2002; 218: 1-11.
[3] Couzin ID, Krause J, Franks NR, Levin SA. Effective leadership and decision making in animal groups on the move. Nature 2005; 433: 513-516.
[4] Kolpas, A., Coarse-Grained Analysis of Collective Motion in Animal Groups, Ph.D. Dissertation, University of California, Santa Barbara, 2008.
[5] Partridge BL. The structure and function of fish schools. Scientific American 1982; 245: 90-99.
[6] GPGPU-Home. GPGPU homepage, 2007. http://www.gpgpu.org/.
[7] H. Li and L. Petzold. Stochastic simulation of biochemical systems on the graphics processing unit. Department of Computer Science, University of California, Santa Barbara, 2007. Submitted.
[8] H. Li, A. Kolpas, L. Petzold, and J. Moehlis. Parallel simulation for a fish schooling model on a general-purpose graphics processing unit. Concurrency and Computation: Practice and Experience, 2008. to appear.
[9]] H. Li, A. Kolpas, L. Petzold, and J. Moehlis. Efficient Parallel Simulation of an Individual-Based Fish Schooling Model on a Graphics Processing Unit, Grace Hopper 2008.
[10] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2007.
[11] NVIDIA Forums members. NVIDIA forums, 2007. http://forums.nvidia.com.

Acknowledgements:
We thank Iain Couzin for helpful discussions related to the model, and Brent Oster for introducing us to the NVIDIA GPU. This work was supported in part by the U.S. Department of Energy underDOE award No. DE-FG02- 04ER25621, by NIH Grant EB007511, by the Institute for Collaborative Biotechnologies through grant DAAD19-03-D004 from the U.S. Army Research Office, and by National Science Foundation Grant NSF-0434328. J.M. also was supported by an Alfred P. Sloan Research Fellowship in Mathematics.

Nvision 08
August 25-27, 2008
San Jose, CA

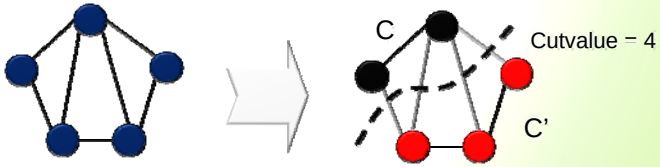
Accelerating Local Search Procedures on Graph Cut Optimization Problems using CUDA

Antonio S. Motemayor, Raúl Cabido, Abraham Duarte, Juan José Pantrigo
{antonio.sanz, raul.cabido, abraham.duarte, juanjose.pantrigo}@urjc.es

nVISION 08
TECHNOLOGY. ART. PASSION.
THE WORLD OF VISUAL COMPUTING.

Graph Cuts

Graph-Cut optimization problems consist of finding a bipartition of a graph into two subsets, in such a way that a given objective function is optimized (maximized or minimized).

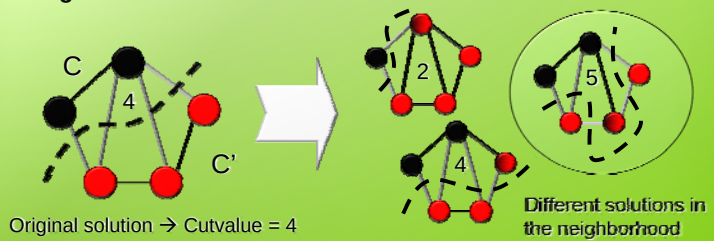


Some Graph-Cut examples are Max-Cut, Mix-Cut, N-Cut or Max-Min-Cut, among others. Most of them are NP-Hard optimization problems. Therefore, it is convenient to devise algorithms for finding an approximate solution to this problem in a reasonable time.

Local Search Procedures

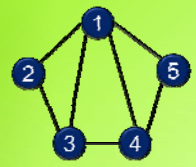
Some of the most successful methods for solving this kind of problem are called ***metaheuristics***. These procedures are high level strategies for exploring search spaces, with dynamic balance between diversification and intensification. The term diversification generally refers to the exploration of the search space, whereas the term intensification refers to the exploitation of the accumulated search experience. Intensification strategies usually are implemented as local search methods, where a given neighborhood of a solution is systematically examined, looking for the best solution.

Local Search procedure: movement of nodes between C and C'
Neighborhood: 1 node at a time



Vertex-based vs. Edge-based cut solution representation

Graph representation



	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	0	0
3	1	1	0	1	0
4	1	0	1	0	1
5	1	0	0	1	0

Adjacency matrix

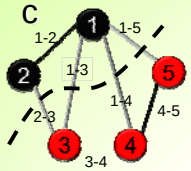
	1-2	1-3	1-4	1-5	2-3	3-4	4-5
1	1	1	1	1	0	0	0
2	1	0	1	0	0	0	0
3	0	1	0	0	1	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1

Cocycle matrix

Cut solution representation

1	2	3	4	5
1	1	0	0	0

Node-based

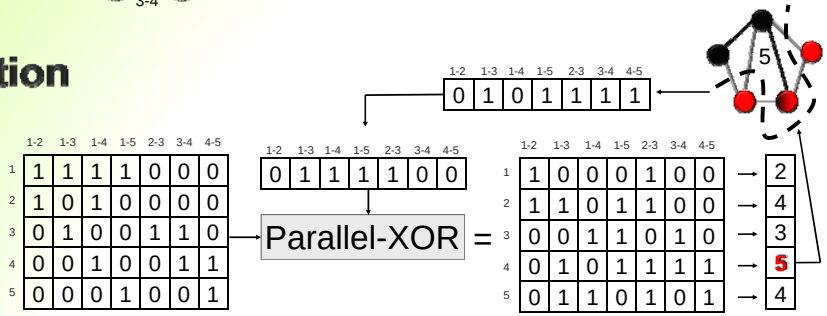


Edge-based

1-2	1-3	1-4	1-5	2-3	3-4	4-5
0	1	1	1	1	0	0

Edge-based parallelization

Our proposal is to use the edge-based representation for an easy and efficient streaming implementation. In a node-based approach, each node movement would lead to direct accesses to its matrix for the extraction of edge weights.



pScan

Results and conclusions

For the edge-based approach, the number of evaluations per second (eps) for a graph with 800 nodes and 1600 edges is about 91000 on CPU and 690000 on GPU (~x7.5), for the same number of nodes and 8000 edges the GPU is ~12 times faster. For the first configuration, a CPU node based implementation would offer ~40000 eps as it is a sparse case in which the cocycle matrix is not so large in comparison to its corresponding adjacency matrix.

We present a promising work-in-progress GPU local search adaptation for the graph Max-Cut problem using the Nvidia CUDA architecture. Preliminary results show good performance for the GPU approach for medium sized and sparse graphs. Even on CPU, the exploitation of the cocycle matrix instead of adjacency structures could be interesting compared to the node based implementation for sparse configurations.

CUDA Computing in Digital Breast Tomosynthesis – Reconstruction and Computer Aided Detection

Xiangwei Zhang[a], Rivera Diego[b], Chukka Srinivas[a], Haili Chui[a], and Kevin Kreeger[a]

[a]Hologic/R2, 2585 Augustine Dr., Santa Clara, CA; [b]Hologic Inc., 35 Crosby Drive , Bedford, MA

Abstract

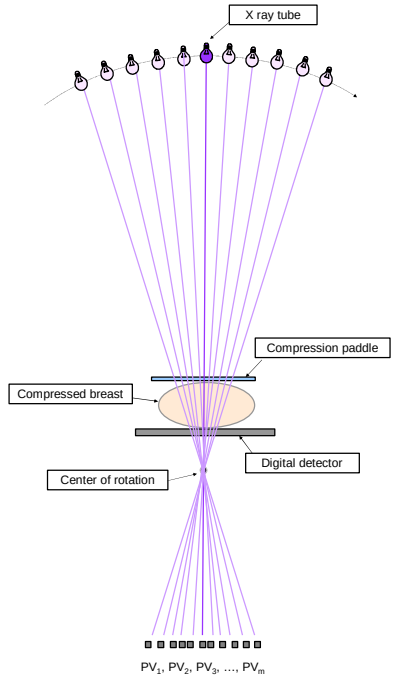
CUDA computing has been introduced into 3D image volume reconstruction and computer aided detection (CAD) of breast cancer in digital breast tomo-synthesis (DBT). Compared to CPU implementation of the same algorithms, GPU CUDA computing has shown a considerable degree of speedup for both reconstruction and CAD.

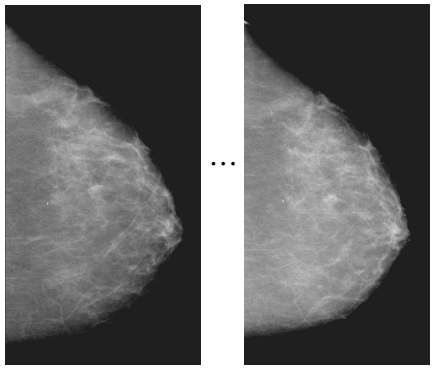
Introduction

- Digital mammography (DM) and CAD have been proven very efficacious for early detection of breast cancer.
- DBT is able to reveal 3D anatomical structures hidden in DM by acquiring multiple X-ray projections (PVs) at different angles.
- In DBT reconstruction of each view, a 3D volume with 50+ slices (5M pixels each) is generated from 11+ PVs (3M pixels each);
- CUDA GPU computing can be helpful for handling the large amount of data.

DBT scan

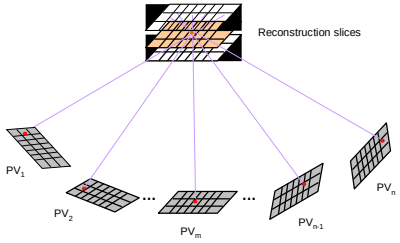
- DBT obtains multiple X-ray projection views (PVs) at different angles.
- To maintain similar dosage as DM, usually each PV uses much lower dosage.
- To get high in-plane resolution, DBT uses a very limited scanning angle (15° to 30°).
- The number of PVs usually ranges from 11 to 15.





DBT reconstruction

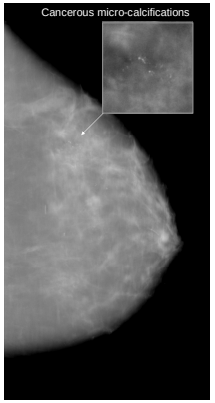
- In back projection (BP) method, for each voxel in the reconstruction volume, the corresponding pixels in PVs can be determined by imaging geometry.
- The corresponding pixel values in each PV are added to give the pixel value in reconstruction volume.



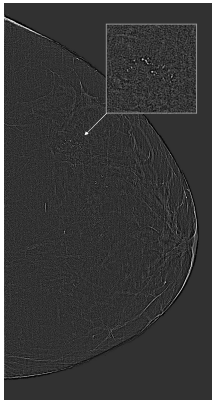
- BP computationally maps to image filtering and ray-tracing at each voxel from multiple projection views.
- DBT reconstruction naturally maps to a single instruction/multiple data (SIMD) architecture, thus making the application amenable for parallel computing and can be implemented on GPU.
- CUDA implementation -- our CUDA implementation of BP is based on the use of the GPU constant memory and the bilinear interpolation operation available for CUDA arrays.
- Speed up by CUDA is around 25X on the DBT reconstruction, with the processing time per view being reduced from ~100 seconds to ~4 seconds.

DBT CAD

- many image features (contrast, edge, gradient …) are calculated by image convolution on the whole volume.



DBT Reconstruction slice



Contrast feature image

- Image convolution can be implemented using CUDA to attain data parallelism.
- Shared memory is used to improve memory access efficiency.
- Speed up by CUDA is around 15X on the image filtering steps, with the processing time per view being reduced from 5-8 minutes to 20-30 seconds.

Conclusion

In digital breast tomo-synthesis, CUDA computing is able to achieve a considerable degree of speedup. Furthermore, CUDA computing is flexible and generic enough to provide practical solutions for two completely different types of tasks -- DBT reconstruction and DBT micro-calcification detection CAD.

FASTRA: FAST TOMOGRAPHIC RECONSTRUCTIONS USING AN EIGHT GPU DESKTOP SUPER COMPUTER

S. van der Maar, K. J. Batenburg, T. Huysmans, J. Sijbers

IBBT-Vision Lab, Dept. of Physics, University of Antwerp.

http://fastra.ua.ac.be

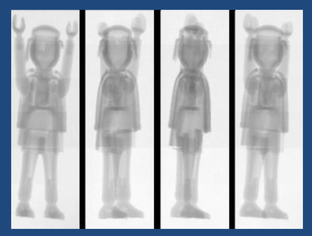
Tomography

Bob

Tomography is a technique for reconstructing a three-dimensional image of an object from its projections, sampled along a range of angles.

A large number of calculations must be performed during a reconstruction and a huge block of data needs to be moved from and to global memory. This application turned out to be perfect for NVIDIA™ GPUs, with their large number of computing elements and wide memory bus.

Bob goes into the scanner

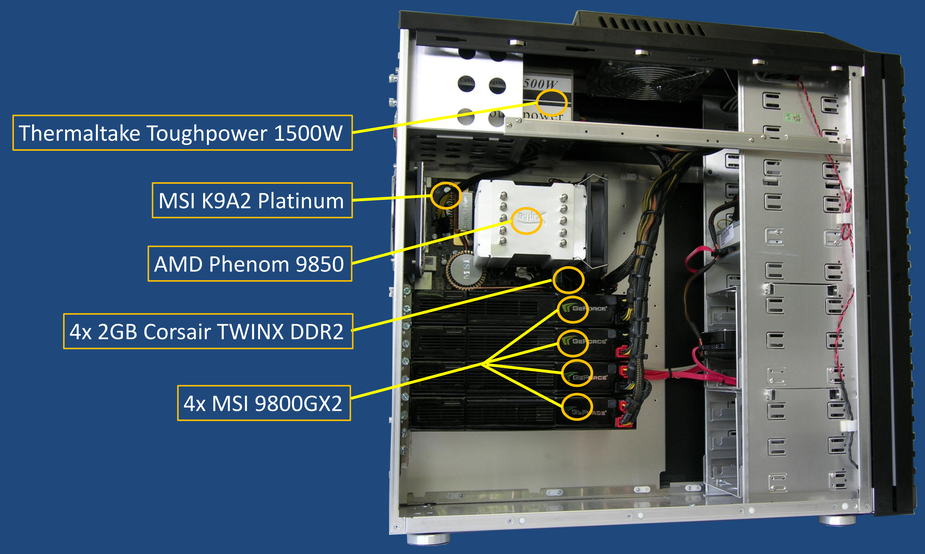


A large number of projections are required.

Tomography: from the set of projections, a 3D image is reconstructed

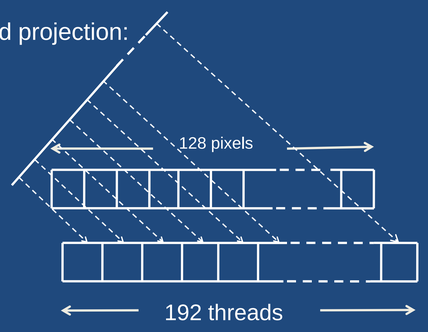
FASTRA

Our system is composed of off-the-shelf components, for a total cost of less than 5000 dollars.

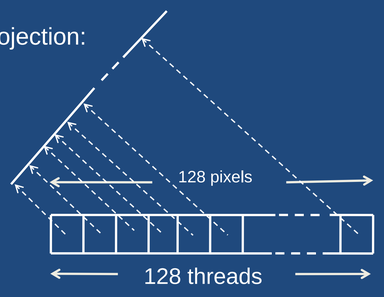


Implementation

Forward projection:



Back projection:



Achieving coalescing of global memory accesses was an important aspect of this project. An image is divided into 128x128 pixel sub-blocks. In that way it is possible to have all threads in one CUDA thread-block working on a row stored in the faster on-chip memory. The memory needed by one block is the sum of its respective image pixels, projection values, and their total weights. These 640 floats require less than 2.6 kB, allowing for a 100% occupancy.

All write-operations are targeted at separate memory locations, avoiding write-collisions. For each sub-block, 192 detector pixels are used, ensuring that all image pixels are fully covered. Back projection is performed with a dedicated thread for each column of pixels.

Block sizes of 128 and 192 allow for a full use of the calculation units. Both numbers are proper denominators of 768, the total number of threads running on the SIMD processors.

As the performed calculations are independent of the processed data, the code contains no branches, eliminating stalls. The extremely large number of available worker threads makes sure the occupancy will be virtually 100%.

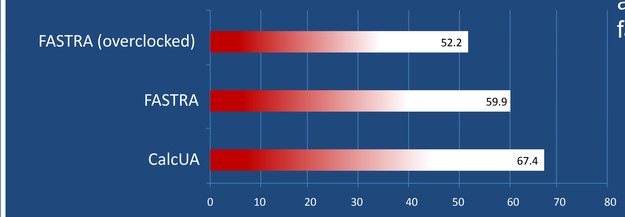
Results

CalcUA

When the results of our NVIDIA™ powered system are compared with an actual supercomputer, it becomes clear just how powerful our desktop set-up is.

We compared the performance of the Fastra with that of the CalcUA (www.calcua.ua.ac.be), the main workhorse of the University of Antwerp. It cost 5.5 million dollars when acquired in March 2005. Containing 512 Opteron nodes, it has a theoretical peak performance of two TerraFLOPS. CalcUA was able to reconstruct 512 slices, each containing 1024^2 pixels, in 67.4 seconds.

Our FASTRA system can reconstruct these 512 slices in 59.9 seconds. By overclocking the shader cores, this can be reduced to 52.2 seconds. This means that for our application, the FASTRA system is 1.13 to 1.29 times faster than Antwerp University's supercomputer.

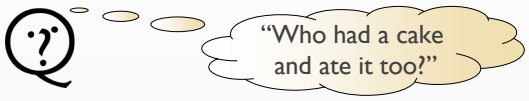


Comparison of running times of FASTRA and CalcUA (in seconds)

Powerset (a Microsoft company)

Combinatorial Set Matching Using GPUs

Anand Madhavan and Eitan Frachtenberg
{anand, eitan}@powerset.com

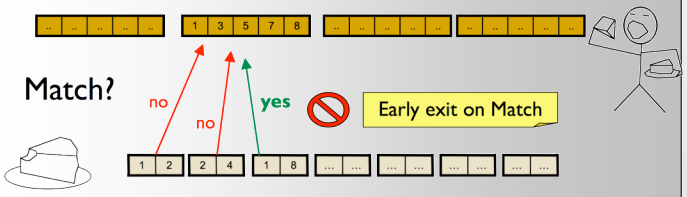


(Refer to the right sidebar titled 'Motivating example' for more information)

The Problem

Given a number of E_{max} tuples, we are interested in finding the ones for which there exists at least one E_{min} tuple, as a subset

E_{max} tuples (where entity *having* and entity *eating* cake are the same)



E_{min} tuples (facts where cake being *had* and cake being *eaten* are the same)

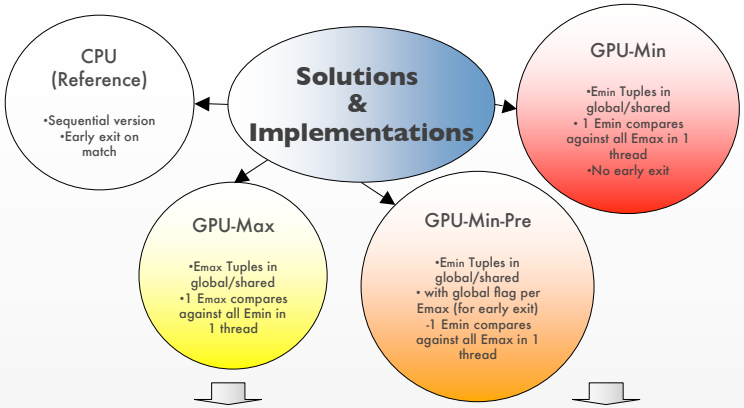
Challenges

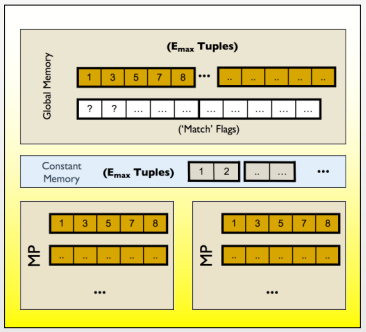
Low arithmetic intensity
`…(t1[i]==t2[i])..&&..(t1[j]==t2[j])…`
mostly boolean operations and comparisons

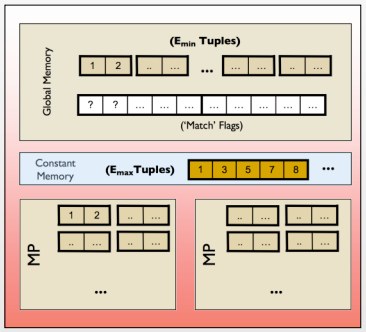
Typical size does not fit in constant memory
100's of kB of tuples

Sequential version very effective when early matches occur

Early exit on Match

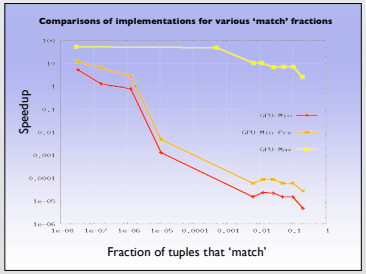


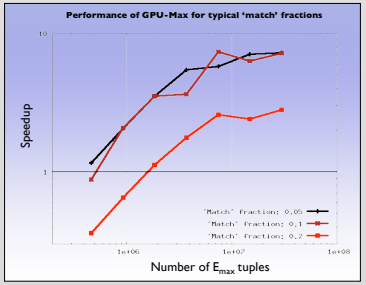




Results and Conclusions

- Speedups are measured against the CPU (Reference) implementation
- Speedups are sensitive to the fraction of all tuples that 'match'. The Sequential algorithm takes advantage of 'early exit' at higher fractions.
- Comparison of GPU-Min-Pre and GPU-Min illustrate that it might be worthwhile doing 'pre-check's (device global memory access) for every E_{max} tuple to facilitate early exit
- Although low 'match' ratios are not typical, we are able to see speedups of 50x **despite** the low arithmetic intensity





Since the **GPU-Max** implementation outperforms the others, we study its performance for other sizes of inputs and typical hit-ratios…

•GPU-Max with single E_{max} in a thread matched against all other E_{min} takes advantage of 'early exit' just like the sequential algorithm

• For the typical 0.1 'match' fraction, we see speedups of upto 8x. With increasing problem size, we see better speedups (as expected) as the cost of data transfer to card gets amortized

Motivating example

"Who had a cake and ate it too?"

Mary and **John** were participants in a **cake**-making competition. Making a **cake** was a piece of **cake** to **Mary** and **John**. As the competition got started, **Mary** started making a chocolate **cake**. **She** put **it** in the oven, whilst **John** was busy making his **cheesecake**. Towards the end of the competition, **many** of them **had** some delicious **cakes** displayed on their tables. **Mary** gave a piece of her **cake** to **John** for tasting. **John** took the piece of **cake**. **He had it** in his hand, while eyeing his own **cheesecake**. **He** ended up **eating** a piece of the **cheesecake** instead. **Mary** on other hand, **had** her chocolate **cake** in her hand and was waiting for **John** to try it. While waiting **she ate** some of her chocolate **cake**.

Who *ate* cakes?

Who *had* cakes?

Who *ate* the exact same cake they *had*?

The problem can be described as combinatorial set generation and matching.

The Emin tuples are generated as part of a linguistic constraint. The Emax tuples are generated as part of another linguistic constraint and the combinatorial set matching is required as part of a combined constraint.

Although the number of cakes or entities in a document is bounded, the number of set combinations can grow exponentially.

For more information contact
{anand, eitan}@powerset.com

FIR and Fast Givens Complex QR Benchmark Performance on NVIDIA GPUs

Michael P. McGraw-Herdeg[1], Douglas P. Enright[1], Michael A. AuYeung[1], B. Scott Michel[1]
[1]The Aerospace Corporation, El Segundo, CA
(E-mail: *mherdeg@rush.aero.org, Douglas.P.Enright@aero.org, Michael.C.AuYeung@aero.org, scottm@aero.org*)

Abstract
If scientific computing users are to harness the power of a multithreaded manycore GPU efficiently, they must first understand what practical applications can be accelerated. To this end we implemented two computationally intensive signal and image processing High-Performance Embedded Computing Challenge Benchmark kernels [1] on the Tesla C870 and 8800GTX GPUs. For a highly parallel algorithm, a GPU was up to 16.9x faster than a CPU on sufficiently large data sets; for an algorithm with more data dependencies, a GPU was up to 1.6x faster.

The HPEC Challenge finite impulse response and QR decomposition benchmarks were implemented in NVIDIA's C extension, the Compute Unified Device Architecture. For the FIR filter bank benchmark, a fast convolution FFT-based frequency-domain finite impulse response (FDFIR) kernel on the GPU performed up to 16.9 times as fast as the reference CPU implementation. A non-transform time-domain finite impulse response (TDFIR) kernel was up to 7.6 times as fast as the CPU. For the QR decomposition of a complex matrix, the GPU implementation was up to 1.6 times faster than the CPU: parallelization is possible even in the highly data-interdependent QR kernel. With a few exceptions, we found that the 8800GTX performs QR decomposition at about the same speed as the Tesla C870 and performs FIR about 5% faster than the C870.

Hardware and Software
The Tesla C870 and 8800GTX each have 16 multithreaded streaming multiprocessors. Each contains 8 scalar processor cores and two special function units for transcendentals; the cores are scheduled via the single-instruction, multiple-thread (SIMT) scheduler within each multiprocessor [2]. In addition, each multiprocessor has 8192 registers, a 16KB parallel data cache of fast "shared memory," and access to 1536MB of GDDR3 "global memory" clocked at 800MHz, resulting in a peak bandwidth of 76.8GB/s. The architecturally similar 8800GTX differs only in that it has 768MB of GDDR3 "global memory" clocked at 900MHz, resulting in a peak bandwidth of 86.4 GB/s. This higher clock rate explains why the 8800GTX is, experimentally, slightly faster than the C870.

The test system, running Ubuntu Linux 7.10, contains two quad-core 64-bit Xeon E5430 processors running at 2.66GHz, 4GB of memory, and an Intel D5400XS motherboard a PCI Express x16 1.1a bus. Each core within the processor has separate 32kB instruction and data caches. The processor has two Level 2 6MB caches (one per dual-core). CPU code was executed in a single thread on one core of one processor.

The benchmarks were implemented using NVIDIA's CUDA SDK version 2.0 beta 2. CUDA provides general-purpose C computation on a GPU with device-specific extensions and libraries. The code was compiled with gcc v4.1.3 and nvcc v0.2.1221.

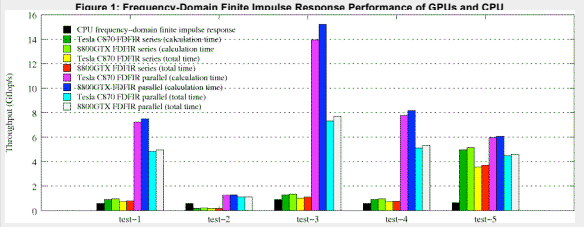
CUDA Programmability
Once a programmer has learned SIMT processing, the CUDA paradigm allows efficient access to processing power that greatly exceeds the performance of a CPU. Embarrassingly parallel problems can see substantial speedup. More complicated applications may entail further code specialization. Arranging threads at the block level introduces little additional complexity: the FIR and QR GPU benchmarks have roughly as many lines of code as their HPEC Challenge counterparts. But a developer may need to dramatically restructure code to hide memory access latency, because CUDA programs benefit greatly from having a high ratio of computations to memory access.

CUDA does most of the thread management for developers. A single line of code invokes a device function and tightly interleaves many threads' computations and memory accesses. Two NVIDIA-supplied libraries – CUFFT, for Fast Fourier Transforms, and CUBLAS, a set of basic linear algebra subroutines – implement commonly used functions that can be called from the host system directly. The FDFIR benchmark uses CUFFT; the TDFIR and QR benchmarks use CUBLAS. Structuring algorithms to fit these libraries' interfaces is not difficult.

Finite Impulse Response Kernel

Frequency Domain: Up To 16.9x FDFIR GPU Performance Compared With CPU

Figure 1: Frequency-Domain Finite Impulse Response Performance of GPUs and CPU



Above: Parallel computations consistently outperform the CPU, although by a smaller factor in test 2, the "short filter" case, where most of the data fits in CPU cache.

Series GPU computations underperform the CPU only in test 2. In test 5, where the FFT size is very large, the series algorithm has only slightly lower throughput than the parallel algorithm.

"Total" throughput includes time spent copying data to and from the device, a cost that developers may hide by reusing data throughout multiple kernel executions in a larger application. "Calculation" throughput accounts for just the time required to perform the computation with the data already resident in global memory. In test 3, 8800GTX parallel calculation throughput is 15.2 Gflop/s, 16.9 times as fast as CPU throughput. Its total parallel throughput is 7.7 Gflop/s, just 8.58 times as fast as CPU throughput.

In these tests, 8800GTX exceeds C870 throughput in every case. The mean percentage improvement is 5.34 percent for the series total, and 3.2 percent for the parallel total. In test 3 parallel calculation, the C870 performs at 13.9Gflop/s, and the 8800GTX performs at 15.2 Gflop/s.

Table 1: FIR Test Parameters and Workload Size

Test	1	2	3	4	5
N	4096	1024	4096	4096	32768
K	128	12	4096	128	4096
M	64	20	128	512	128
FDFIR Workload (Mflop)	34.1383	2.2265	105.2636	273.1062	708.0165
TDFIR Workload (Mflop)	268.4355	1.9961	17179.87	2147.484	137439

Left: Five sets of test data were generated. In test 2, frequency-domain FIR requires more floating-point operations than time-domain FIR. The filters in test 2 contain only 12 elements, and it is called a "short filter" [4] test. TDFIR may sometimes be more efficient than FDFIR on short filter tests, even on the GPU.

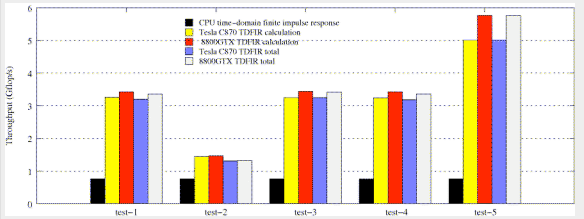
The FIR benchmark models a set of M filters which operate on a set of M distinct input signals of length N. Each filter has K coefficients. Signal and filter elements are complex, single-precision floating point numbers. The output of filter *m* is given by convolution in the *time domain* [4]:

$$\sum_{k=0}^{K-1} x_m[i-k]w_m[k] \quad \text{for } i = 0,1,\ldots,N-1.$$

The *frequency-domain* approach is often preferred because its computation time does not depend on filter size. Frequency-domain FIR computes the FFT of the signal and filter, multiplies the transformed signal and filter, then inverts the transformation.
The FIR filter was implemented on the GPU in three ways. The *series filter* performs FDFIR one signal at a time. The *parallel filter* uses CUFFT's "batch mode" to process all the signals at once. The *time-domain filter* performs the convolution directly and uses the Level 1 BLAS *caxpy* operation from NVIDIA's CUBLAS library.

Time Domain: Up To 7.6x TDFIR GPU Performance Compared With CPU

Figure 2: Time-Domain Finite Impulse Response Performance of GPUs and CPU

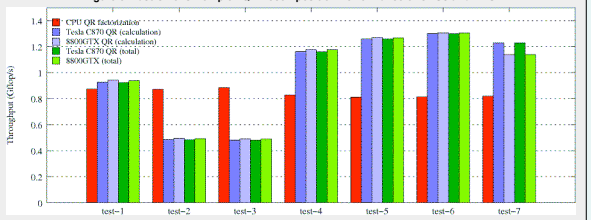


Above: CPU throughput is constant in all five tests. GPU throughput in the "short filter" test case 2. In the largest test case, test 5, which has 128 distinct signals, GPU throughput is highest. the largest test 5. In test 5, the 8800GTX outperforms the CPU by a factor of 7.59 and outperforms the C870 by a factor of 1.15. In the other tests, 8800GTX throughput exceeds C870 throughput by smaller margins: 1.1 percent in test 2, 4.9 percent in tests 1, 5.3 percent in test 3, and 5.6 percent in test 4. GPU TDFIR outperforms GPU parallel FDFIR in the short test 2. In test 5, GPU TDFIR has higher total throughput than FDFIR. (But its calculation time is also much higher.)

Complex QR Factorization Kernel

Up to 1.6x GPU Performance Compared With CPU

Figure 3: Fast Givens Complex QR Decomposition Performance of GPUs and CPU



Above: CPU QR factorization throughput is roughly identical for each test case. On large workloads, GPU throughput is consistently faster than CPU throughput; GPU throughput is 1.4-1.6 times as fast as CPU throughput in tests 4-7. The two GPUs are mostly comparable: on tests 1-6, the 8800GTX has throughput just 0.39-1.98 percent higher than the C870's throughput. But on test 7, the 8800GTX's throughput is 7.2 percent lower the C870 throughput.

Table 2: QR Test Parameters and Workload Size

Test	1	2	3	4	5	6	7
M	500	180	150	1000	1000	2000	2000
N	100	60	150	500	1000	1000	2000
Workload (Mflop)	397.3	30.5	45	13333	33333	61333	106667

Left: Seven sets of test data were generated. The workload figures were taken from the qrWorkload.m script in the HPEC Challenge suite [1].

QR Factorization
In the QR benchmark, an *mxn* complex matrix A is factorized into an *mxm* unitary matrix Q and an upper triangular matrix R. A, Q, and R contain complex, single-precision floating point numbers. After QR factorization, the following properties hold:

$$A = QR; Q^H Q = I$$

The HPEC reference implementation performs QR via Givens rotations. A Givens rotation selectively zeroes an element of the target matrix A by updating two of its rows. In the Fast Givens QR algorithm [4], the rotations necessary to triangularize A into R are directly computed into Q.
The parallel approach used on the GPU also uses fast Givens QR but performs more than one rotation at once, in the standard Sameh-Kuck concurrency strategy [6]. This pattern concurrently zeroes elements that are a knight's move apart; see figure 1 of [6]. A pipelining approach could perform twice as many rotations simultaneously [7].

Conclusion: CUDA Performance Depends Upon Amount Of Data Parallelism Available
Given our current results, Fast Givens QR factorization as implemented entirely on the GPU is a proabably poor factorization method for QR. The most frequently called functions, row updates, have a very low ratio of calculations to memory accesses, a problem which could only partially be solved with vectorization and parallel reduction. A QR algorithm which performs panel factorizations on the CPU and uses the GPU heavily for BLAS-3 matrix multiplications has achieved 192 Gflop/s and has shown a speedup of 8.3x over a 2.67GHz Core2 Duo E6700 [3] . The Tesla C870 and 8800GTX GPU can achieve at most 346 Gflop/s of sustained throughput [8]. But our fastest result, from FDFIR, is 15.2Gflop/s, 4.4% of that maximum. Effective CUDA programs must perform many computations between memory accesses and should exploit the GPU's plentiful register space and fast shared memory banks. Vector programming may also improve performance.
We have previously shown that the 8800GTX is 15x faster at FDFIR (total time) and 2.5x faster at Fast Givens QR than an Athlon-64 4200+ running at 2.21 GHz [9]. The GPU's speedup over newer CPUs is no longer as large. We expect performance will improve with the new GT200 which has almost twice as many multiprocessors and registers along with greater memory bandwidth as compared to the 8800GTX and Tesla C870 GPUs [2].

References
[1] MIT Lincoln Laboratories, "HPEC Challenge," available online at *http://www.ll.mit.edu/HPECchallenge/*, accessed 5 August 2008.
[2] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide", Version 2.0, 7 June 2008.
[3] V. Volkov, J. Demmel, "LU, QR, and Cholesky Factorizations using Vector Capabilities of GPUs," EECS Department, University of California, Berkeley, technical report UCB/EECS-2008-49, available online at *http://eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.pdf*, accessed 5 August 2008.
[4] J. Lebak, A. Reuther, and E. Wong, "Polymorphous Computing Architecture (PCA) Kernel-Level Benchmarks," MIT Lincoln Laboratory project report PCA-KERNEL-1, 13 June 2005.
[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Third Edition, Johns Hopkins University Press, 1996.
[6] A. H. Sameh and D. J. Kuck, "On Stable Parallel Linear System Solvers," *Journal of the ACM*, Vol. 25 No. 1, Jan. 1978 p. 81-91.
[7] M. Hofmann and E. J. Kontoghiorghes, "Pipeline Givens Sequences for computing the QR decomposition on an EREW PRAM," *Parallel Computing*, Vol. 32 No. 3, March 2006.
[8] NVIDIA Corporation, "CUDA 1.0 FAQ," available online at *http://forums.nvidia.com/index.php?showtopic=36286*, accessed 5 August 2008.
[9] M.P. McGraw-Herdeg, D. P. Enright, and B. S. Michel, "Benchmarking the NVIDIA 8800GTX with the CUDA Development Platform," High Performance Embedded Computing Conference 2007, 18-20 September 2007, Lexington, MA, United States. Available online at *http://www.ll.mit.edu/HPEC/agendas/proc07/Day2/29_D.Enright_Abstract.pdf*, accessed 5 August 2008.

General Purpose Molecular Dynamics on Graphic Processing Units (GPUs)

AMES LABORATORY
United States Department of Energy

Joshua Anderson
(Iowa State University and Ames Laboratory)

Chris Lorenz
(Kings College London)

Alex Travesset
(Iowa State University and Ames Laboratory)

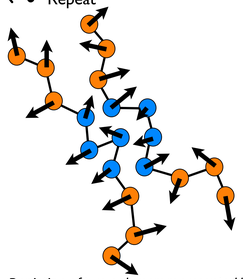
IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

HOOMD-blue
Highly Optimized Object Oriented Molecular Dynamics

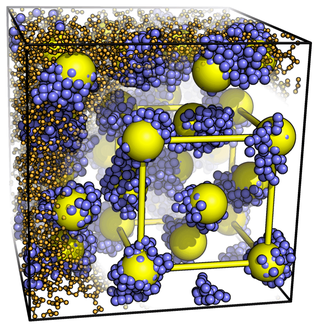
www.ameslab.gov/hoomd

Molecular Dynamics (MD) ①

- Solves the equations of motion of N particles
- Can model a huge variety of systems in and out of thermal equilibrium from coarse-grained polymer solutions to all-atomic silica interfaces, large biomolecules and more
- A single run requires days of computer time on a large computer cluster
- Iterative method
  - Generate neighbor list
  - Calculate pair forces
  - Calculate bond forces
  - Step forward in time
  - Repeat



Depiction of two polymers represented by beads with positions and velocities in the simulation.



Results from a simulation where 500 polymers self-assembled into an fcc ordered phase of micelles. Blue beads form hydrophobic cores while the orange beads make up the hydrophilic corona.

GPU Hardware ②

- Inexpensive, yet extremely fast hardware originally designed for use in real-time rendering in computer games and CAD software
- The latest models are programmable enough for use in scientific computing, including MD
- Executing data-parallel algorithms, the GPU can process tens of thousands of threads at once
- NVIDIA® CUDA™ provides a familiar C language environment for writing GPU code

CPU Core
5 GFLOP/s peak

- **Lightning**
  - Fastest cluster at ISU
  - 592 processor cores (148 nodes)
  - Opteron 280 (2.4 GHz) dual core

GPU
933 GFLOP/s peak

gcc

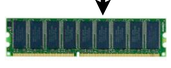
CUDA

6.4 GiB/s

8 GiB/s

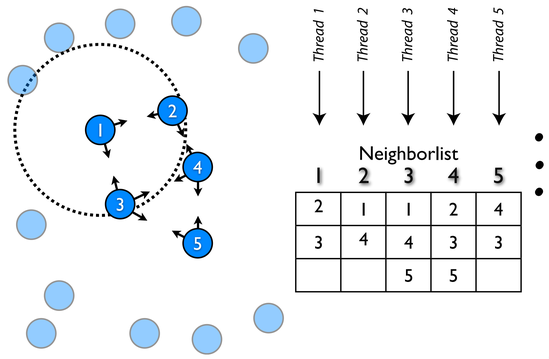
PCI-Express

141.2 GiB/s



Steps in the algorithm ③

- Short range pair force summation
  - Uses an already generated neighbor list (see below)
  - Each thread on the GPU sums the force of a single particle, so all forces are summed simultaneously in parallel



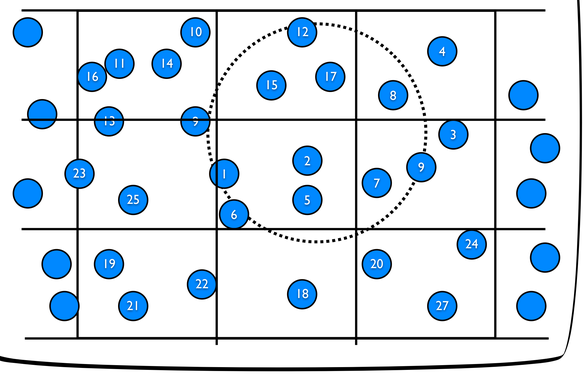
Overall Performance

Software	Hardware	TPS (time steps per second)
HOOMD	GTX 280	256.0 (very preliminary)
HOOMD	Tesla C870	194.4
LAMMPS	32x Opteron 280 CPU cores	185.5
HOOMD	8800 GTS (G92)	163.4

HOOMD performance compared vs. LAMMPS on a fast cluster. TPS lists the number of time steps per second that are calculated for a standard benchmark run of a system with 64,000 particles.

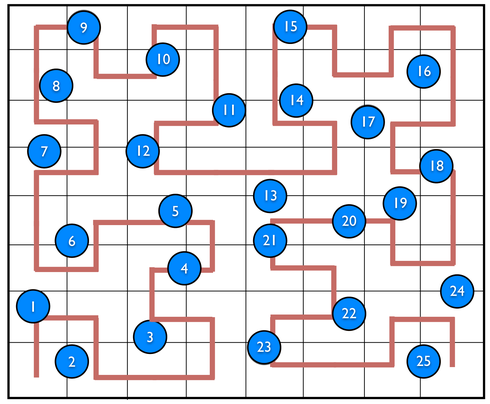
Neighbor list generation ④

- Each thread builds the neighbor list for a single particle so that all lists are built in parallel
- See our paper for the full details of the algorithm

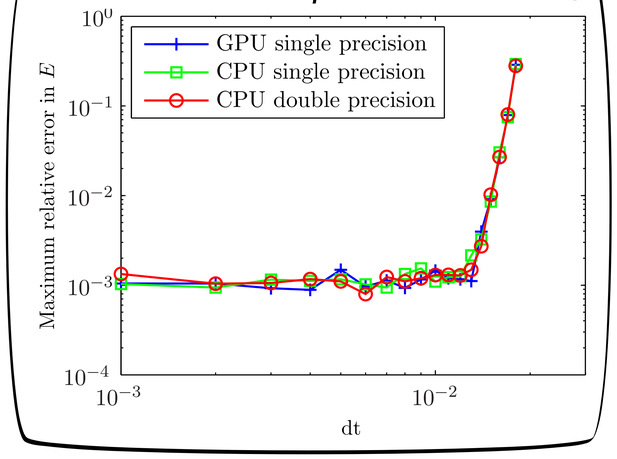


Particle reordering ⑤

- In large systems, memory access patterns can lead to a high percentage of cache misses
- Particles are reordered in memory following a Hilbert curve to improve the cache hit rate, increasing the performance by a factor of 5

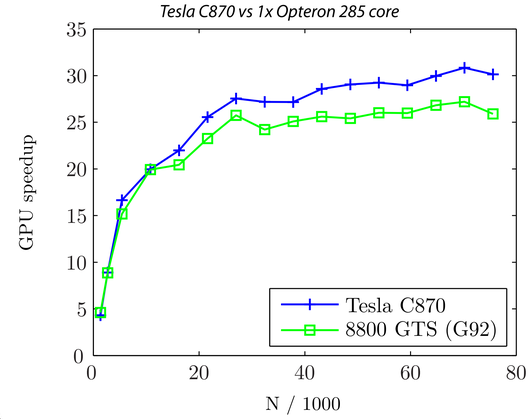


No loss of precision ⑥



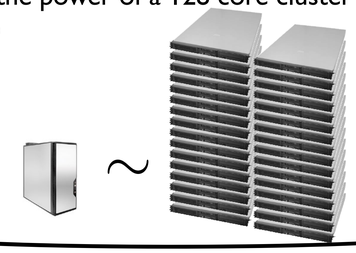
Speedup vs. single CPU core ⑦

Tesla C870 vs 1x Opteron 285 core



Cluster Performance ⑧

- A benchmark simulation is run in LAMMPS, developed by Sandia National Lab, on Lightning for comparison
- HOOMD running on a **single** Tesla C870 reaches a performance equivalent to **32** processor cores
- Four GPUs can be hosted in a single machine putting the power of a 128 core cluster on the desktop



HOOMD software ⑨

- HOOMD is **already** mature enough to perform simulations for research. **Download it now!** http://www.ameslab.gov/hoomd
- We are constantly adding new features

References

[1] General purpose molecular dynamics simulations fully implemented on graphics processing units
Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset
Journal of Computational Physics 227 (2008) 5342-5359

[2] Cuda programming guide 1.1. <http://developer.nvidia.com/object/cuda.html>

[3] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comp. Phys. 117 (1995) 1–19

[4] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecul ar modeling applications with graphics processors, J. Comp. Chem. 28 (2007) 2618–2640.

Acknowledgements

This work is funded by NSF through Grant DMR-0426597 and by DOE through the Ames lab under Contract No. DE-AC02-07CH11358.

Thank you, NVIDIA, for your generous hardware donations.

GPU Accelerated Algebraic Multigrid Methods for the Virtual Heart Model

Manfred Liebmann[†], Gernot Plank[‡], Anton Prassl[‡] and Gundolf Haase[†]

Overview of the Virtual Heart Model

The Bidomain Equations:

The bidomain equations are a set of coupled partial differential equations which describe the current flow in the myocardium (region I). Optionally, the current flow in a surrounding medium (torso, fluid bath, blood-filled cavities of the heart) are included in the formulation (regions II). The bidomain equations are written as follows:

$$\begin{aligned} -\nabla \cdot (\sigma_i \nabla \phi_i) &= -\beta I_m \\ -\nabla \cdot (\bar{\sigma}_e \nabla \phi_e) &= \beta I_m \\ -\nabla \cdot (\bar{\sigma}_b \nabla \phi_e) &= I_e \end{aligned}$$

where

$$\begin{aligned} I_m &= C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, \vec{\eta}) - I_{tr} \\ \frac{d\vec{\eta}}{dt} &= g(V_m, \vec{\eta}) \\ V_m &= \phi_i - \phi_e \end{aligned}$$

The bidomain solver of the Cardiac Arrhythmia Research Package CARP, one of the most efficient solvers for this type of problem, implements several solution methods. Most frequently, however, based on an operator splitting technique the following scheme is employed [7, 8, 1]. The bidomain equations are decoupled into

an Elliptic PDE

$$(A_i + A_e)\Phi_e^{k+1} = A_i V^{k+1} + I_e$$

a Parabolic PDE

$$\begin{cases} V^{k*} = (1 - \Delta t A_i) V^k - \Delta t A_e \phi_e^k & \Delta x > 100 \mu m \\ \left[1 + \frac{1}{2}\Delta t A_i\right] V^{k*} = \left[1 - \frac{1}{2}\Delta t A_i\right] V^k - \Delta t A_e \phi_e^k & \Delta x < 100 \mu m \end{cases}$$

and a set of ODE's

$$\begin{aligned} V^{k+1} &= V^{k*} + \frac{\Delta t}{C_m} i_{ion}(V^{k*}, \vec{\eta}^k) \\ \vec{\eta}^{k+1} &= \vec{\eta}^k + \Delta t\, g(V^{k+1}, \vec{\eta}^k) \end{aligned}$$

where

$$A_i = -\frac{\nabla \cdot (\bar{\sigma}_i \nabla)}{\beta C_m};\ A_e = -\frac{\nabla \cdot (\bar{\sigma}_i \nabla)}{\beta C_m};\ t = k\Delta t$$

Further, besides the bidomain solver, CARP consists of the ionic model library IMP, the visualization tool Meshalyzer, and codes for the computation of ECG (ϕ_e recovery) and MCG (cooperation with Dr. Weber dos Santos and the Physikalisch Technische Bundesanstalt in Berlin).

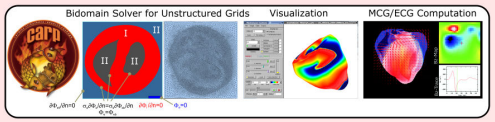
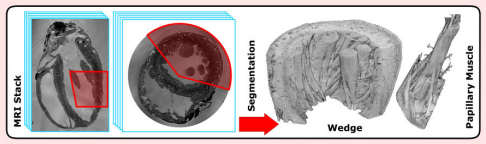


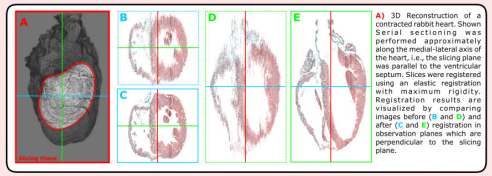
Image Processing and Mesh Generation:

A first attempt to segment high-resolution MRI image stacks and histological image stacks in a semi-automatic fashion has been undertaken [2]. In trichrome-stained histological sections, interesting tissue types and the extracellular space occupy areas of the color histogram with minimal overlapping and thus can be segmented directly using color thresholding. Segmentation of the 3D MRI datasets constitutes a more complicated challenge. This process required several steps including the estimation and removal of the bias field and a set of ad-hoc morphological operations.



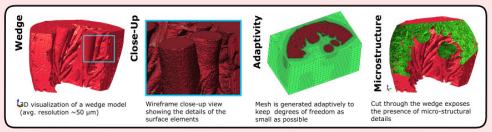
Registration

In the process of slicing the specimen, preparing the section and acquiring histological images, considerable rigid and non-rigid distortion is introduced. We correct for this distortion by using slice-to-slice registration, in two steps. In the first step, an initial coarse rigid registration alignment is realized. In the second step, non-rigid registration between adjacent slices is performed following the technique developed by Keelling and Ring [3].



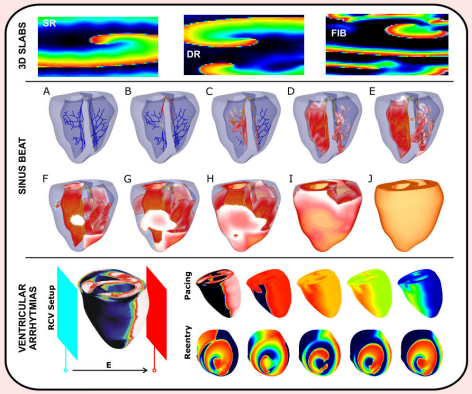
Mesh Generation

Within the framework of a national cooperation, the Octree-based meshing software TARANTULA has been developed which produces boundary-fitted, locally-refined and conformal multi-element meshes on unstructured grids to allow 1) smooth representation of organ boundaries, and 2) to reduce the degrees of freedom by using adaptive methods when discretizing the non-myocardial volume.



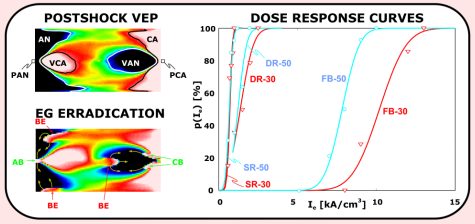
Arrhythmogenesis:

The group of the applicant has gathered experience in carrying out computer simulations dealing with mechanisms underlying the formation of arrhythmias using both simple slab-like geometries and more realistic setups using anatomically realistic representations of the ventricular geometry. Further, a novel method to simulate to incorporate the Purkinje system has been developed recently to allow the simulation of sinus beats and the interaction of the Purkinje system with arrhythmic acitivity.



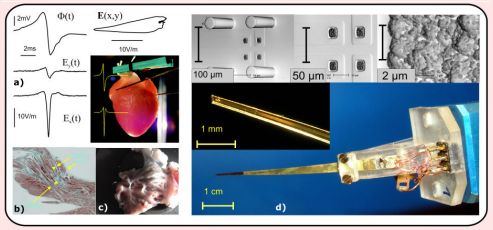
Defibrillation:

Electrical defibrillation is the application of a large electrical shock to the heart to terminate otherwise life-threatening arrhythmias. The applicant has been carrying out simulation studies [6] which investigated the relationship between shock energy requirements for successfull defibrillation and 1) the degree of organization of an arrhythmia, and 2) the effect of microscopic heterogeneities.



Experimental Validation:

A new flexible sensor for in-vitro experiments has been developed [11, 10, 9, 5] to measure the surface potential Φ, and its gradient, $\mathbf{E}$ (electric near field), at given sites of the heart. During depolarisation, $\mathbf{E}$ describes a vector loop from which direction and magnitude of local conduction velocity ϑ can be computed. Four recording silver electrodes separated by 50 μm, conducting leads, and solderable pads were patterned on a 50 μm thick polyimide film. Combined with high-resolution data acquisition (sampling rate 100-800 kHz at 16- 24-bit) we are able to discriminate signal latencies of just a few microseconds and to monitor the computed time-course of $\mathbf{E}$ as well as the magnitude and direction of local conduction velocity ϑ on-line from beat-to-beat. Thus, the capability to measure accurately activation time as well as direction and velocity of propagation at the microscopic size scale makes the method ideally suited for the validation of computer simulations using micro-anatomical grids, as suggested in this SFB proposal. A map of $\mathbf{E}$ combined with the corresponding histographs forms the fundament for a micro-structure related computer mode. (a) Guinea-Pig heart and signals Φ and $\mathbf{E}$. (b) histograph with myocytes (red) and barriers from connective tissue (blue). (c) right atrial endocard. (d) sensor.



References

[1] *G. Plank, M. Liebmann, R. Weber dos Santos, E. Vigmond and G. Haase.* **Algebraic Multigrid Preconditioner for the Cardiac Bidomain Model**, IEEE Trans. Biomed. Eng., accepted pending minor revision.

[2] *Burton* A.B.R., G. Plank*, J. Schreiber*, V. Grau Colomer, H. Aksamner, S.L. Keeling, J.L. Lee, N. Smith, N.A. Trayanova and P. Kohl. (* shared first authors)* **3-Dimensional Models of Individual Cardiac Histo-Anatomy: Tools and Challenges.** Ann. N.Y. Acad. Sci., in press.

[3] *S. Keeling, W. Ring.* **Medical image registration and interpolation by optical flow with maximal rigidity.** J. Math. Imag. Vision. 23:4765, 2005.

[4] *N. Trayanova, G. Plank, and B. Rodriguez.* **What have we learned from Mathematical Models of Defibrillation and Postshock Arrhythmogenesis?, Application of Bidomain Simulations.** E Heart Rhythm, in press.

[5] *Hofer, E., F. Keppilinger, T. Thurner, T. Wiener, and G. Plank.* **A novel floating sensor array to detect electric nearfields of beating heart preparations.**, Biosensors and Bioelectronics, 21(12), 2232-2239, 2006.

[6] *Plank G., L.J. Leon, S. Kimber, E.J. Vigmond,* **Defibrillation depends on conductivity fluctuations and the degree of disorganization in reentry patterns.**, J. Cardiovasc. Electrophysiol., 16(2):205-216, 2005.

[7] *Weber dos Santos, R., G. Plank, S. Bauer, E.J. Vigmond,* **Parallel multigrid preconditioner for the cardiac bidomain model.**, IEEE Trans. Biomed. Eng., 51(11):1960-8, 2004.

[8] *Weber dos Santos, R., G. Plank, S. Bauer, E.J. Vigmond,* **Preconditioning techniques for the bidomain equations.** Lecture Notes in Computational Science and Engineering (LNCSE), ISBN/ISSN 1439758, 40:571-580, 2004.

[9] *Plank, G. E.J. Vigmond, L.J. Leon, E. Hofer.* **Cardiac near-field morphology during conduction around a microscopic obstacle - a computer simulation study.** Ann. Biomed. Eng., 31:1-7, 2003.

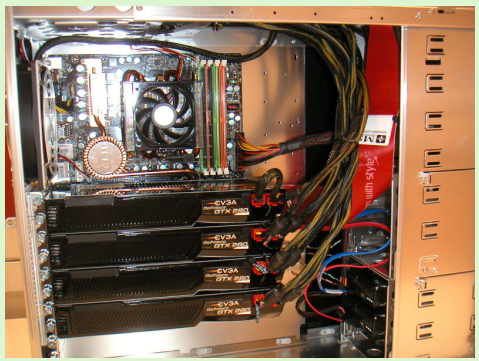
[10] *Plank, G. E. Hofer.* **The use of cardiac near-field measurements to determine activation times.** Ann. Biomed. Eng., 31:1066-1076, 2003.

[11] *Plank, G. and E. Hofer.* **Model study of vector-loop morphology during electrical mapping of microscopic conduction in cardiac tissue.** Ann. Biomed. Eng., 28(10):1244-1252, 2000.

Multi-GPU Accelerated Algebraic Multigrid Methods

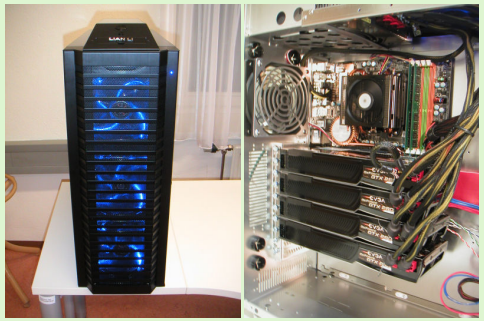
CARP Simulator:

The capabilities of the CARP simulator are extended to include a multi-GPU algebraic multigrid solver for the elliptic PDE implemented using the Nvidia CUDA Toolkit and the Parallel Toolbox software package.



Quad-GPU Compute Server:

A test platform for the multi-GPU PCG-AMG solver has been set up using a MSI K9A2 Platinum mainboard, a quadcore Phenom 9950 processor, and four Nvidia GTX 280 boards. Early tests show good scaling of the parallel multigrid solver on the test platform. Giving a 50-fold performance advantage over a typical single CPU configuration. With these results a tremendous speedup of the Virtual Heart simulation will be possible in near future.



PCG-AMG Benchmark

The algebraic multigrid method (AMG) provides an efficient preconditioner for the preconditioned conjugate gradient (PCG) algorithm. Currently the complete multigrid iteration within the PCG solver is executed on the GPU while the setup of the algebraic multigrid method is handled by the CPU due to high complexity of the algorithm. Table 1 shows the comparison of a single PCG-AMG iteration on different cluster computers: Kepler (32 Opteron processors, Infiniband interconnect), Boltzmann (16 Opteron processors, Gigabit Ethernet network) and Liebmann (4 Quadcore Barcelona processors, Shared memory architecture). The finite element matrix of the Virtual Heart simulation has dimension 862.515 with $12,795.209$ nonzero elements. All computations use double precision arithmetics.

N	$\#A$	Kepler 32P (IB)	Boltzmann 16P (GBE)	Liebmann 16P (SHM)	Single GTX 280	Dual GTX 280	Quad GTX 280
862.515	12,795.209	0.021094	0.058125	0.053750	0.022178	0.012828	0.008277

Table 1: Timing of PCG-AMG iteration in seconds (double precision)

Considering the cost of a typical cluster computer compared to a single Nvidia GTX 280 board we see two to three orders of magnitude in price/performance advantage.

[†] Institute for Mathematics and Scientific Computing, Univerisity of Graz, Austria [‡] Institute of Biophysics, Center for Physiological Medicine, Medical University of Graz, Austria

GPU Radio Astronomy Signal Convolution

Chris Harris, Karen Haines, Lister Staveley-Smith
chris@wasp.uwa.edu.au

The University of Western Australia

Abstract

The next generation of radio telescopes, such as Square Kilometer Array and the associated Pathfinder arrays, require vast amounts of computation due to the extremely large number of interferometers and the imaging requirements. The hardware for this computation is becoming a significant consideration in array design, both in terms of initial cost and power consumption. The Graphics Processing Unit (GPU) provides power efficiency and affordability as well as the flexibility of general purpose hardware. This work implements a GPU-based FX spectrometer, which processes up to 128 streams of 8-bit interferometer data, for a variable number of frequency channels. The GPU signal convolution outperforms a traditional CPU implementation by up to two orders of magnitude. Current research is examining the GPU implementation of more sophisticated filters, such as polyphase filterbanks.

Background

Interferometry combines the signals of multiple telescopes to obtain higher angular resolution than that which could be produced by a single telescope. The processing involved in combining these streams, correlation, is computationally intensive. Correlation can consist of a number of stages starting with the packed data streams from the telescopes through to obtaining the final image. The GPU FX Spectrometer presented here starts with the packed data streams, and produces accumulated output, ready for gridding.

Figure 1: A Murchison Widefield Array (MWA) tile similar to those used to produce each stream of test data. The tile consists of 16 antenna elements in a 4x4 configuration. The signal from these elements is combined in hardware to produce two streams of data per tile, consisting of orthogonal polarisations.

Algorithm

The algorithm performs three main steps on the GPU: unpacking, the fast Fourier transform, and signal convolution. These steps are shown in Figure 2. Unpacking the data consists of converting the data from its 8 bit representation to a 32 bit floating point. The fast Fourier transform is applied in segments to all streams, the length of the transform is set at the beginning of the algorithm. In the final step, the data is cross-multiplied per frequency channel for each pair of streams (including self pairings) and then accumulated. The second step utilises a vendor supplied GPU FFT library. The first and third stages are specifically written for the GPU, in particular to be optimal for $NL > 512$, where N is the number of interferometer signal streams and L is the number of output spectral channels.

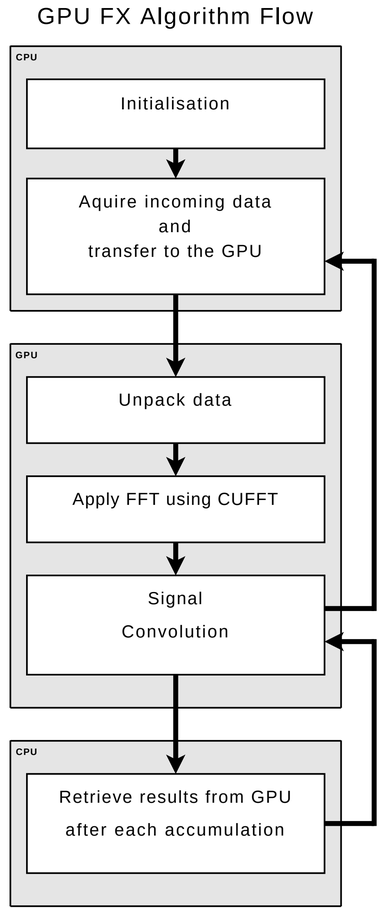


Figure 2: An overview of the algorithm flow of the GPU FX Spectrometer. Aside from data uploading, result retrieval, and algorithm flow control; the computation from the raw sampled stream to the final output is processed in its entirety on the GPU.

Results

The GPU FX algorithm was implemented in CUDA. Testing was performed on 1 GB of data obtained from the MWA as described in Figure 1. Results were obtained for a range of transform lengths. These are shown in Figure 3, as the bandwidth from each individual tile stream that the GPU can keep up with for the given parameters. A tile stream is a single polarisation signal from a tile.

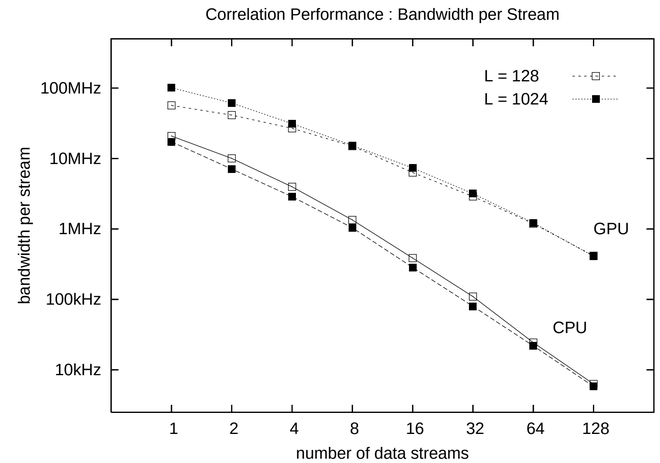


Figure 3: Performance results for testing for a number of tile streams. Shown are two alternate transform lengths L, which corresponds to the number of spectral channels in the output.

Discussion

We found the optimal performance of the GPU to be dependant on thread loading and topology. In particular the convolution stage required a thread to baseline ratio of 1:1 or 1:4 depending on the GPU capabilities and the correlation parameters. The lower performance for a small transform length and number of streams is caused by *thread starvation*, in which there is insufficient parallelism for the GPU to be running at full capacity. In general the GPU outperforms a comparative CPU implementation by one to two orders of magnitude.

As this algorithm has the capacity to be made parallel in time using time-slicing, additional performance may be attained by using multiple GPUs. The total number of streams is limited by the total global memory available on the GPU. Beyond this limit, the algorithm could computed in parallel between multiple GPUs, with each processing a subset of pairs. Alternate optimisations could be written to improve performance for a low number of streams and output spectral channels.

Acknowledgements

The authors thank Frank Briggs of the Research School of Astronomy and Astrophysics at the Australian National University for providing the MWA data and comparative CPU code used in this research.

Parallel maximal matching algorithm for arbitrary graphs using GPU

Piotr Stańczyk, University of Warsaw, Poland

Definitions

Matching – a subset of edges such that no two edges have a common endpoint.

Maximal matching – a matching of maximal possible cardinality.

Perfect matching – a matching that uses all vertices.

Allowed edge – an edge which belongs to some maximal matching

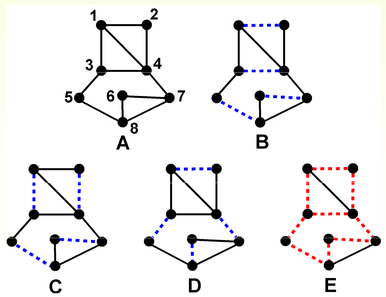


Figure 1. An example graph G=(V,E) with n=|V| vertices (A), its three perfect matchings (B, C and D) and all allowed edges (E)

Perfect matching, cycle covers and matrix determinant

For any graph G it is possible to construct an antisymmetric, symbolic adjacency matrix A. It turns out that:

$$\det(A) \neq 0 \text{ IFF } G \text{ has a perfect matching} \quad (1)$$

This follows from the fact that each product of det(A) formula can be mapped to some cycle cover. All products corresponding to non-even cycle covers get reduced, while even cycle covers can be further mapped to perfect matchings.

	1	2	3	4	5	6	7	8
1		A	B	C				
2	-A			D				
3	-B			E	F			
4	-C	-D	-E				H	
5			-F					-G
6							K	-J
7				-H		-K		-I
8					G	J	I	

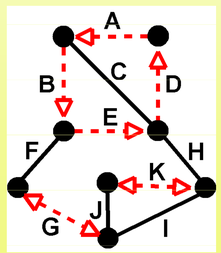


Figure 2. Symbolic adjacency matrix for a graph from Figure 1 with a sample even cycle cover

Randomization

By computing *det(A)* it is possible to check if a graph *G* has a perfect matching, but symbolic computation is very expensive (exponential cost). It is possible however to substitute symbols from *A* with random elements of Z_p. For $P = O(n^2)$ (1) still holds with high probability (one sided error possible).

	1	2	3	4	5	6	7	8
1		A	B	C				
2	-A			D				
3	-B			E	F			
4	-C	-D	-E				H	
5			-F					-G
6							K	-J
7				-H		-K		-I
8					G	J	I	

⇨

	1	2	3	4	5	6	7	8
1		13	7	3				
2	4			9				
3	5			14	6			
4	6	2	11				1	
5			5					8
6							10	16
7				1		12		9
8					15	16	2	

Figure 3. Sample substitution for P=17

In order to find a perfect matching in G, it is possible to iteratively remove each edge from a graph and compute determinant for the reminding graph. If it is not equal to 0, the edge is allowed… This leads to $O(n^5)$ algorithm.

Minors of a matrix

Minor A_{kl} of a matrix *A* is a sub matrix of *A* obtained by removing *l'*th row and *k'*th column. There exists a relation between minors and matrix inverse:

$$\det(A) * A^{-1}[k,l] = (-1)^{k+l}\det(A_{lk}) \quad (2)$$

By computing inverse of A, it is possible to find all allowed edges of a graph. By repeatedly computing an inverse matrix and removing a single allowed edge, we obtain $O(n^4)$ algorithm.

Gaussian elimination

Instead of computing many inverse matrices for sub matrices of A, it is possible to perform Gaussian elimination:

$$M_{kl}^{-1} = M^{-1} - M^{-1}[k,*] * M^{-1}[*,l] / M^{-1}[k,l] \quad (3)$$

This leads to $O(n^3)$ time complexity.

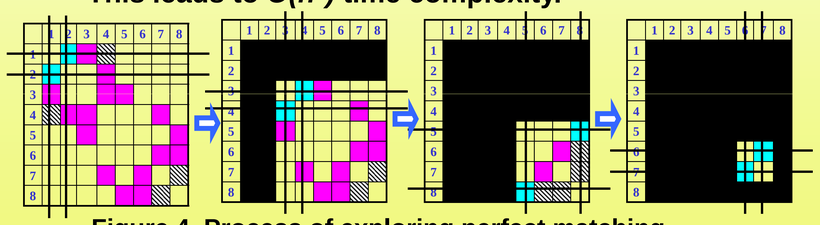


Figure 4. Process of exploring perfect matching via Gaussian-elimination steps of A^{-1} matrix. Found edges: (1,2), (3,4), (5,8) and (6,7)

From perfect matching to maximal matching

In order for the algorithm to support maximal matching, it is sufficient to find a maximal non-singular sub matrix of A and use it as input for the perfect matching algorithm. The matrix can be obtained via Gaussian elimination.

Parallel approach

Parallel implementation ($O(n^2)$ time with $O(n)$ processors) of this algorithm requires following parallel building blocks:

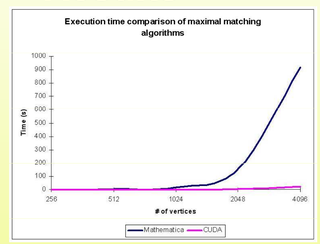
- Gaussian elimination
- Matrix Inverse
- Vector-Matrix multiplication

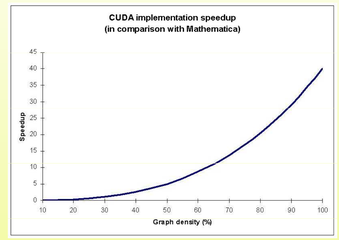
As all of these operations are matrix based, they can be implemented efficiently with GPU.

Performance boost with GPU

To further increase performance on GPU, it is possible to introduce lazy Gaussian elimination. It updates a matrix only after a number of elimination steps in order to save global memory bandwith and exploit shared memory.

Results





References

[1] Algebraic Algorithms for Matching and Matroid Problems, Nicholas J. A. Harvey, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

[2] Maximum matchings via Gaussian elimination, M. Mucha, P. Sankowski, Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium.

Predicting Near-Space Radiation Hazards in Real Time

A collaboration between

University of Washington & Eagle Harbor Technologies

Michele Cash
Robert Winglee
Erika Harnett

Tim Ziemba
John Carscadden
Dan O'Donnell

Background

Lethal doses of radiation dominate Earth's near-space environment at altitudes between 700 km and 50,000 km. In this region, highly energetic particles orbit our planet in two concentric toroids known as the Van Allen Radiation Belts. These particles, which are trapped by the dipole structure of the Earth's magnetic field, can remain in orbit for many months or even years and pose a significant threat to both humans and robotic missions in Earth orbit - degrading electronics and materials in spacecraft systems and potentially causing biological damage to astronauts. While these particles have always been there, increased human and satellite presence in Earth orbit has made understanding these energetic ions and electrons a priority. In particular, the dynamic nature of the radiation belts and their response to solar storms and activity makes it difficult to ensure the safety of satellites and astronauts in space through positioning alone. Thus, there is a need for space weather modeling and forecasting in order to develop hazard prediction and assessment for space missions.

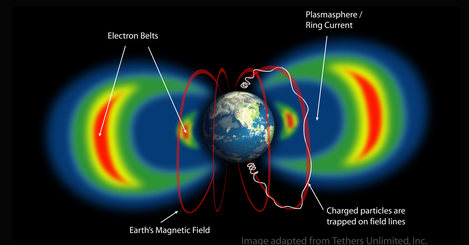
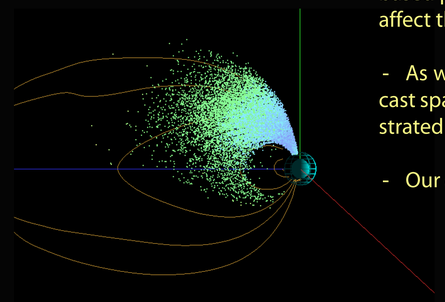


Image adapted from Tethers Unlimited, Inc. www.tethers.com



Multi-fluid / Multi-Scale Model

- University of Washington has developed a cutting edge CPU-based multi-fluid/multi-scale space environment model, which includes the local particle effects that govern the acceleration of energetic particle populations

- In a collaboration between UW and Eagle Harbor Technologies, we are creating a GPU-based particle tracking code to explore the ways in which changing solar wind conditions affect the radiation belts and the near-Earth environment

- As with current atmospheric weather forecasting, speed is essential to effectively forecast space weather in a useful time frame; with the particle tracking code we have demonstrated our ability to move more than 500,000 particles in faster than real time

- Our current research direction for real-time tracking of radiation belt ions is three fold:

1) Validate and verify the current GPU particle tracking code
2) Develop visual diagnostic tools that will allow investigation of the Earth's radiation belts and their dynamics
3) Convert the CPU-based multi-fluid space environment code to run on a GPU

Space Weather Modeling

- Is important for predicting potentially damaging radiation environments that will be encountered by satellites, spacecraft, and astronauts

- Warning that a solar storm is heading toward Earth is available only three days in advance

- Accuracy of the forecasted effects remains low since models that simulate the interaction of solar storms with the Earth's radiation belts typically take weeks or even months to run

- The potential space hazards that develop around other planets during such storms are even less well known

- Particular uncertainties include the environments that astronauts and robotic spacecraft might encounter on an extended journey to the Moon, Mars or elsewhere in our solar system

- Radiation environments are in a constant state of modulation, due to their interactions with a perpetual stream of energetic particles from the Sun called the solar wind. This solar wind carries with it the interplanetary magnetic field, which modifies the terrestrial magnetic field. This in turn leads to the energization of particles within our space environment generating auroras and the radiation belts.

Plasma Dynamics

$$\frac{\partial \rho_\alpha}{\partial t} + \nabla \cdot (\rho_\alpha \vec{v}_\alpha) = 0$$

$$\rho_\alpha \frac{d\vec{v}_\alpha}{dt} = n_\alpha q_\alpha (\vec{E} + \vec{v}_\alpha \times \vec{B}) - \nabla P_\alpha$$

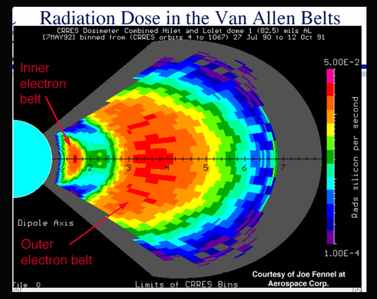
$$\frac{\partial P_\alpha}{\partial t} = -\gamma \nabla \cdot (P_\alpha \vec{v}_\alpha) + (\gamma - 1)\vec{v}_\alpha \cdot \nabla P_\alpha$$

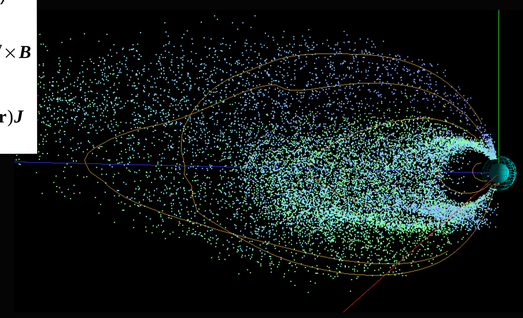
Electro Dynamics

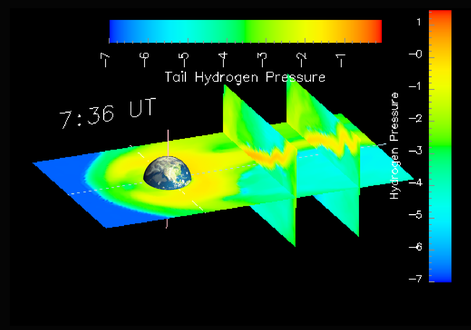
$$\boldsymbol{E} + \boldsymbol{V}_{de} \times \boldsymbol{B} + \frac{1}{en_e} \nabla P_e = 0 \text{ (Ohm's Law)}$$

$$n_e = \sum_i n_i, \quad \boldsymbol{V}_{de} = \sum_i \frac{n_i}{n_e} \boldsymbol{V}_i - \frac{\boldsymbol{J}}{en_e}, \quad \boldsymbol{J} = \frac{1}{\mu_0} \nabla \times \boldsymbol{B}$$

$$\boldsymbol{E} = -\sum_i \frac{n_i}{n_e} \boldsymbol{V}_i \times \boldsymbol{B} + \frac{\boldsymbol{J} \times \boldsymbol{B}}{en_e} - \frac{1}{en_e} \nabla P_e + h(\mathbf{r}) \boldsymbol{J}$$







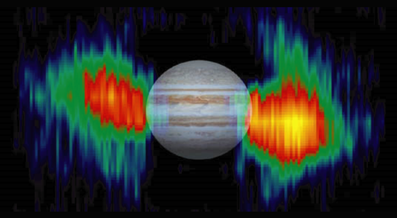
Benefits of GPU-based code

- Faster computations at higher grid resolution with enhanced visualizations

- Two-way communication between the particle tracking code and the global space environment model, improving the speed and efficacy of information interchange

- Obtain results in real-time when it normally takes many weeks to run the code

- Important implications for space weather forecasting, fusion research, trestle weather and climate change, ocean dynamics, and modeling volcano eruptions



*Send email to: mcash@u.washington.edu

Work was supported in part by Air Force STTR Phase I, Contract #FA9550-07-C-0162

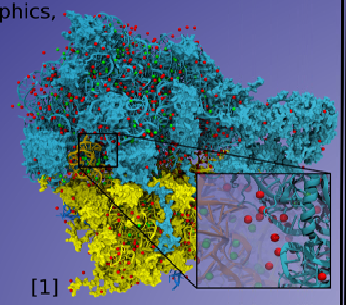
Harmony

Managing Heterogeneity and Parallelism in Multi-Processor Systems

Gregory Diamos and Sudhakar Yalamanchili
Georgia Institute of Technology
{gtg250v@mail.gatech.edu, sudha@ece.gatech.edu}

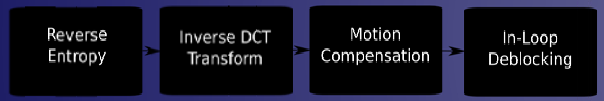
Heterogeneous Systems

* **Have high performance potential**
  - 10-100x speedup for media, graphics, and scientific codes
* **Have high complexity**
  - Different programming models
  - Best architecture can be data dependent
* **Inhibit compatibility**
  - Typically tie an application to a specific system configuration

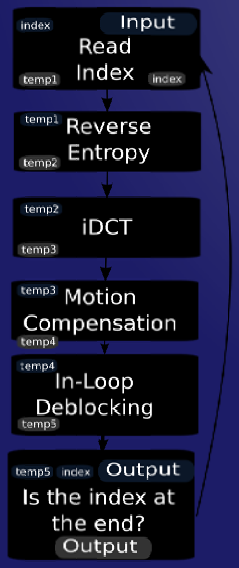


[1]

H.264 Decode



Harmony Programming Model

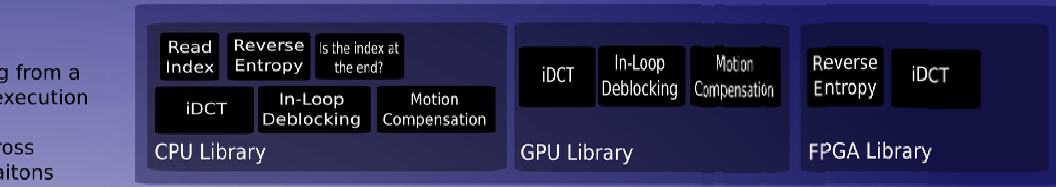


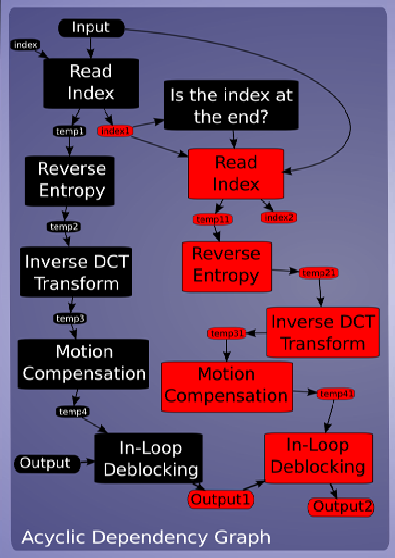
Harmony Program

* Applications are composed of compute kernels, control decisions, and global variables
* Compute kernels are analogous to function calls, but must explicitly declare input and output variables
* Control decisions are identical in function to branches
* Global variables are declared as inputs or outputs of kernels, allowing runtime management of data dependencies and relaxed coherence and consistency memory models
* A program is stepped through sequentially, but kernel calls are not blocking
* When a branch is encountered, its target can be speculatively predicted

Kernel Libraries

* Create a dynamic mapping from a kernel's invocation to its execution
* Facilitate compatibility across different system configuraitons

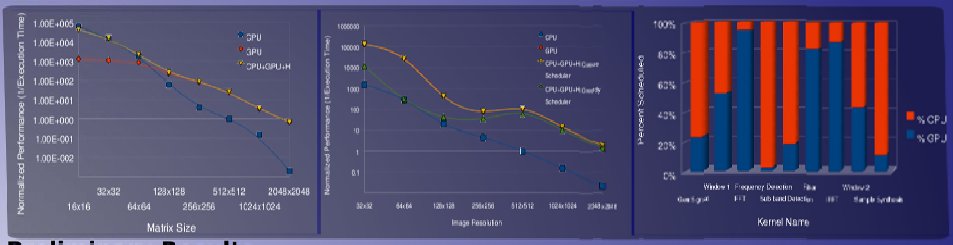




Acyclic Dependency Graph

Inference of Parallelism

* As kernels are encountered, they are added to a dependency graph that captures data parallelism
* Input and output variables are used to infer producer/consumer dependencies
* Speculation is used to increase parallelism (red kernels in the figure are speculative)
* Variable renaming trades memory footprint for parallelism (red variables in the figure have been renamed)

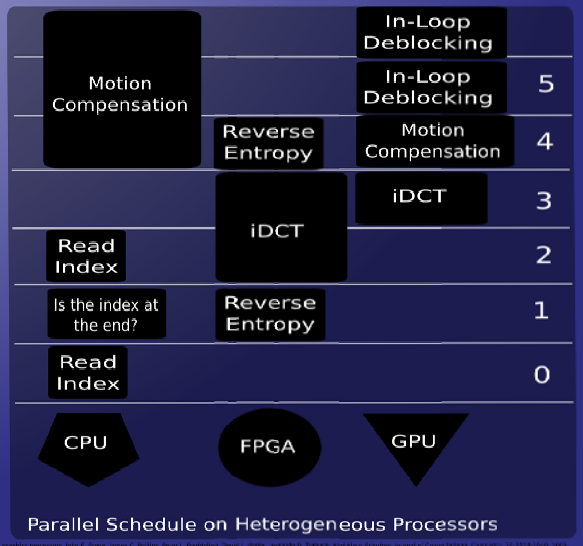


Preliminary Results

* For applications where the best architecture is data dependent, a Harmony program can track the performance of the best architecture across all data sizes
* A poor implementation of a scheduler can negatively impact performance by up to 80x for some applictions
* An optimal partitioning is often unintuitive, data dependent, and system configuration dependent, making it more suited to runtime mapping than static partitioning

Resource Constrained Scheduling

* A scheduler attempts to minmize the execution time of the application while satisfying all data dependencies using available processors
* Different architectures can have different performance characteristics
* An optimization phase correlates input variables to execution time for individual kernels on specific architectures
* An analytic model is used to predict the execution time of kernels and create a better schedule



Parallel Schedule on Heterogeneous Processors

[1] Accelerating molecular modeling applications with graphics processors, John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten, Journal of Computational Chemistry, 28:2618-2640, 2007.

Performance Prediction Targeting Multi-GPU Execution

Dana Schaa and David Kaeli, Department of Electrical and Computer Engineering, Northeastern University

Introduction

- GPGPU is currently focused on enhancing performance on a single GPU (usually yielding 10X-40X speedup)
- Although larger speedups are obtainable from a single GPU, they require abandoning general purpose code in favor of graphics-based, architecture-specific algorithms.
- Our alternative methodology is a technique for **execution prediction and enhancement using multiple GPUs.**

Goals

- Allow developers to predict the most appropriate GPU configuration for an application without having to purchase hardware or create a full software implementation.
- Alleviate some of the need for hardware-specific tuning, increasing portability and decreasing complexity.
- Provide additional speedup by utilizing more resources.

Assumptions for Prediction Performance

- Applications work with streaming data.
- CPU paging algorithm resembles an LRU scheme.
- Execution times on a single GPU are obtained empirically, as are significant computations on the CPU.
- An abstract specification of the parallel application.

Extended Parallel Model

Considerations:

- PCI-e bandwidth
- Network bandwidth
- Pinned vs. pageable memory
- Disk access time
- RAM bandwidth

$$t_{total} = t_{cpu} + t_{cpu\_comm} + t_{gpu} + t_{gpu\_comm}$$

$$t_{cpu\_comm} = \begin{cases} t_{memcpy} & \text{for shared configurations} \\ t_{network} & \text{for distributed configurations} \end{cases}$$

$$t_{gpu\_comm} = \begin{cases} t_{pinned\_alloc} + t_{pci-e} & \text{for pinned memory} \\ t_{disk} + t_{pci-e} & \text{for pageable memory} \end{cases}$$

$$t_{disk} = \begin{cases} 0 & \text{dataset} < \text{RAM} \\ 2 \cdot \dfrac{(B_{dataset} - B_{RAM})}{T_{disk}} & \text{RAM} < \text{dataset} < \text{RAM} + \text{transfers ize} \\ 2 \cdot \dfrac{B_{transfersize}}{T_{disk}} & \text{dataset} > \text{RAM} + \text{transfers ize} \end{cases}$$

Predicting Performance

Using our equations, along with empirical bandwidth and execution values, we created vectorized scripts that calculate expected execution time for applications with various communication requirements.

We are able to vary the number of processors and the input data size to provide complete coverage of the problem set.

Configurations Handled by this Model

Distributed GPUs

Shared-System GPUs

Sample Calculation

```
%----------Distributed Ray Tracing----------
P = 1:16
XDIM = 1024; YDIM = 768;
GLOBAL_STATE_SIZE = 16384;

for SCALE = 1:4
   ROWS = XDIM*SCALE./P;
   COLS = YDIM*SCALE;
   DATA_PER_GPU = ROWS.*COLS.*PIXEL_SIZE;

   GPU_TIME = .314*SCALE^2./P;

   NETWORK_TIME = step(P - 1) .* ...
      ((GLOBAL_STATE_SIZE.*(P-1))/NET_BAND + ...
      (DATA_PER_GPU.*(P-1))/NET_BAND )

   PCI_TIME = DATA_PER_GPU./GPU_BAND;

   FPS(:,SCALE) = 1./ ...
      (GPU_TIME + NETWORK_TIME + PCI_TIME)
end;
%-------------------------------------------
```

Applications

- Average 11% difference in prediction versus actual execution time over all applications (max 40%)

Convolution	Independent data
Least-Squares	Independent data
Ray Tracing	Sync/update each iteration
Image Reconstruction	Sync/update each iteration
2D FFT	Large communication
Matrix Multiplication	Large communication

Results

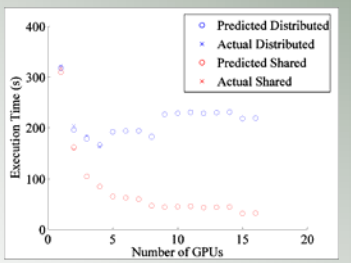


Figure 1: Results for Image Reconstruction

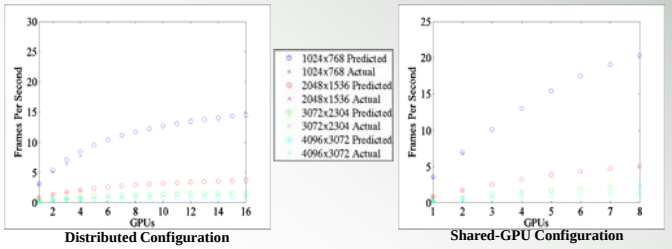


Figure 2: Results for Ray Tracing

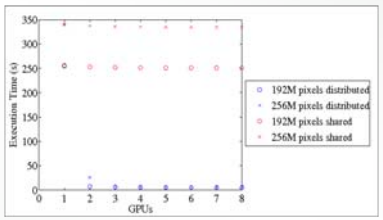


Figure 3: Predicted Results for Convolution

Acknowledgement

This work was supported in part by the Institute for Complex Scientific Software, and Gordon-CenSSIS, The Bernard M. Gordon Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (Award Number EEC-9986821).

GPU Accelerated C-Arm CT and Fluoroscopy: A Pilot Study

Dmitri Riabkov, Todd Brown, Arvi Cheryauka, and Alexander Tokhtuev

GE Healthcare-Surgery, 384 Wright Brothers Drive, Salt Lake City, 84116, Utah, USA

1. INTRODUCTION

Performance characteristics for cone-beam computed tomography (CBCT) on a mobile C-arm are dictated by the clinical needs of interventional imaging and image-guided procedures. Large image datasets have to be processed by computationally expensive algorithms fast enough to make reconstructed 3-D image a relevant part of clinical workflow. C-arm mobility presents engineering challenges, associated with power and cost constraints, in achieving the necessary level of performance.

Fluoroscopy applications may not be as computationally intensive as a CT reconstruction, but they impose a different set of constraints on image processing implementation[1]. X-ray image detector produces a video stream that must be processed without skipped frames or noticeable lag. Quality of incoming images may also be sub-optimal due to the desire of minimizing the radiation dose delivered to the patient and the clinician. Fluoroscopy images may not use detectors full frame rate or full dynamic range and may contain excessive amounts of quantum noise. Until recently, good fluoroscopic image quality was only attainable on highly specialized proprietary image processing hardware, which cannot be easily reprogrammed for acceleration of other computing tasks.

It has been shown that Graphics Processing Units (GPUs) can provide outstanding levels of computational power[6-7]. Their widespread use outside of the core market was limited by inflexible architectures, explicitly optimized for rendering applications. Latest generation of GPU designs from major manufacturers are better suited for general-purpose computing. For example, NVIDIA Tesla products are being introduced as high-performance computing solutions[4]. To facilitate use of this technology by those not familiar with graphic programming, NVIDIA provides CUDA (Compute Unified Device Architecture) Toolkit consisting of a C-language development environment, driver, runtime and numerical libraries[5]. The present study explores the feasibility of fluoroscopy image processing and CBCT acceleration on this novel computational platform.

2. METHODS

The fluoroscopy and CBCT algorithms are applied to grayscale images of 1024 by 1024 pixels with 16-bit depth, which is representative of a video frame size found in today's high-end fluoroscopy systems[1].

Fig. GE-OEC 9900 Elite Mobile X-ray C-arm for interventional and minimally-invasive surgery[3].

The experimental CBCT acquisition setup includes the prototype of a mobile C-arm, the programmable rotating table, and the anthropomorphic phantoms[2]. A scan performed with the turntable rotational sweep is functionally equivalent to a scan with orbital sweep using the C-arm gantry.

Processing performance is considered adequate for fluoroscopy application if a sustained rate of 30 frames per second is achievable. Two SIMD general-purpose computing models for GPUs (GPGPU), Cg/OpenGL and CUDA, are considered for CBCT reconstruction in this study.

2.1. Hardware setup

Two experimental setups based on a 90nm G80 GPU architecture are used in this study. The GeForce 8800 GTX is one of the latest NVIDIA consumer graphic cards. The Tesla C870 is NVIDIA GPU-based add-on board targeting high-performance computing[4]. Both cards have 128 thread processors operating at 1.35 GHz with listed peak performance of 518 GFlops.

2.2. Computed tomography algorithm

Modified Feldkamp-Davis-Kress (FDK) CBCT reconstruction algorithm was used for a quasi-circular scanning trajectory[6]. The backprojection step is the most resource-consuming portion of the algorithm. In this study we use the same approach as in the earlier work, but implement it in different computing environment.

2.3. Fluoroscopy algorithms

A fluoroscopy image processing chain is simulated for benchmarking purposes by executing following operations commonly found in modern C-Arms:

- Frame transfer into image processing device memory.
- Temporal averaging to reduce the amount of quantum noise.
- Grayscale conversion that can be used for gamma correction or histogram equalization
- Spatial averaging to replace / supplement temporal averaging when blurring artifacts become noticeable.
- Additional spatial filtration for algorithms such as edge detection or other image enhancements.
- Histogram accumulation for calculation of an image metric that can be used to make other operations adaptive

Temporal averaging is implemented in a typical recursive fashion by adding the currently acquired frame F_a with the previously averaged frame F_p using appropriate weights $\mathbf{a}$ and (1 - $\mathbf{a}$), so that the current temporal average F_c is the superposition of the new and previous frames:

$$F_c = F_a \cdot (1-\alpha) + F_p \cdot \alpha.$$

Spatial averaging and filtration are implemented by the application of convolution kernels, of various symmetric geometries, across an image frame. The general solution to a 2D convolution, g_{con}, is given as:

$$g_{con}(x_1, x_2) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f_1(x_1, x_2) \cdot f_2(x_1 - k, x_2 - l).$$

Most practical convolutions are local and can be implemented for 1024 by 1024 images with kernels measuring less than 16 pixels in size, thus greatly simplifying calculations. Some widely used 2-D kernels, such as Gaussian blur, are also separable and can be replaced with the application of two 1-D kernels, further reducing computational load.

2.4. Cg/OpenGL Computing Model

The Cg/OpenGL computing model uses an OpenGL interface to manage GPU-CPU interactions while the Cg (C for graphics) runtime API (Application Programming Interface) executes the fragment kernel program on the GPU. In a typical execution under this model, the data is loaded to texture memory using OpenGL settings. To invoke GPU pipeline execution, a quadrilateral, mapped as another texture and sized to cover a rectangular region of pixels matching the desired size of the output array, is drawn. The Cg fragment program running on the GPU has read-only access to the data in the texture memory and that data is processed in parallel. Processing is performed on each pixel/data element of the quad texture independently. The result of the fragment program execution is written either to the display or the off-screen frame buffer. The frame buffer can then be attached as a predefined texture for another round of GPU processing; however, the fragment program cannot modify the data values. A fixed one-directional program pipeline structure, such as this, allows high-speed parallel execution. However, interaction between different processing threads is not possible.

2.5. CUDA Computing Model

The Compute Unified Device Architecture (CUDA) was developed as a higher-level abstraction for general purpose computing on current and future NVIDIA GPUs[5]. CUDA presents the GPU as a data-parallel computing device connected to a host CPU. Each device has several multiprocessors capable of simultaneously executing parallel threads of the same program on different data elements. A multiprocessor program, traditionally called a kernel, can be written in an extended version of C programming language. When a kernel is scheduled for execution on a multiprocessor, a thread block is created. Thread blocks are organized into a grid to run the same kernel on more than one multiprocessor. coalesced into contiguous aligned accesses. Three distinct types of device memory spaces can be defined: read-write non-cashed global memory, read-only cached constant memory, and read-only texture memory. Texture units perform 2-D caching, address conversion, return value normalization and interpolation for texture fetches.

3. RESULTS

3.1. Fluoroscopy

All operations in the simulated fluoroscopy chain are defined on a pixel-by-pixel basis and can be effectively parallelized in CUDA. The execution time for each operation is defined as the period between the host CPU calling a CUDA kernel and the point when the GPU has signaled completion to the host.

Data transfer time for an image frame is close to **3 *ms*** in our experimental setup. This time is highly dependent on the host hardware (CPU, chipset) and is not indicative of GPU performance.

Recursive temporal averaging takes approximately **0.3 *ms*** per frame, when pixel values are loaded through texture fetches. If straightforward reads of 16-bit integers from global memory are used instead, execution time increases to **0.8 *ms*** per frame.

A grayscale conversion does not require execution of a separate kernel, but may be combined with another operation to avoid storing an intermediate result. Time required to perform temporal averaging of a frame is increased by **0.04 *ms*** when gamma correction is applied to input pixel values by taking their square root. LUT-based grayscale conversion can be efficiently implemented by using the caching and interpolation capabilities of texture unit. Using an input pixel value as a coordinate to perform a texture fetch from a 1024-value LUT adds **0.06 *ms*** to a timing estimate.

Number of memory reads and arithmetical instructions required to apply a spatial N-by-N kernel is $O(N^2)$. For a separable 2-D kernel the processing is completed in two 1-D stages. It's execution time scales linearly with kernel size. Naïve implementations of 2-D convolutions perform all calculations directly on image data in global device memory. For performance-optimized implementation of separable convolution, all pixel values are loaded into shared memory using coalesced memory accesses. After synchronization of the load instructions, each thread in the thread block calculates a single result by application of convolution kernel for the given dimension. The results are then written back to global memory, again in a coalesced pattern. The results for different kernel geometries are summarized in Table below.

Kernel size	Convolution time for naïve implemen-tation of non-separable kernel (ms)	Convolution time for naïve implemen-tation of separable kernel (ms)	Convolution time for optimized imple-mentation of separable kernel (ms)
5x5	14.02	2.67	1.11
7x7	22.57	3.86	1.11
9x9	38.31	5.04	1.17
11x11	57.71	6.21	1.44
13x13	81.01	7.40	1.45
15x15	108.06	8.58	1.60

It is interesting to note that in border situations, it is preferable to keep some of the threads idle in order to preserve memory alignment requirements for global memory coalescing. Computational resources are plentiful on the GPU, making memory bandwidth a limiting factor in this and many other fluoroscopy operations.

Unlike other operations, histogram accumulation has a memory access pattern that depends on pixel values rather than coordinates. If more than one thread needs to increment an accumulated value in the same histogram bin, thread execution has to be serialized. Memory access collisions become more likely when input pixel values are distributed unevenly over dynamic range, which will be the case for many imaging situations. Reduction in entropy of simulated input image data increased execution time by an order of magnitude (see Table below). NVIDIA's white paper[5] suggest a variation of an algorithm that achieves deterministic performance by placing multiple copies of histogram into shared memory and then merging partial results. Due to the limitations on the size of shared memory (16 KB on G80) this approach completely eliminates collisions only for histograms of 64 or less bins.

Size of random data pattern	Run time for 1024 bin histogram (ms)	Run time for 64 bin histogram (ms)
14-bit	0.56	0.26
12-bit	0.64	0.26
10-bit	0.85	0.26
9-bit	1.21	0.26
8-bit	1.81	0.26
7-bit	3.12	0.26
6-bit and less	6.41	0.26

Overall, a practically useful fluoroscopy image processing chain could be assembled to fit within **33 *ms*** frame time limits. Convolutions in the frequency domain may be also considered for inclusion. CUFFT library supplied with CUDA[5] executes forward and inverse Fast Fourier transforms on a full 1024 by 1024 frame in **7 *ms*** in our experimental setup.

3.2. Computed tomography

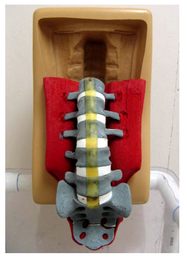
The cone beam voxel-driven backprojection was implemented with the projection-wise approach using CUDA and benchmarked against the Cg/OpenGL results.

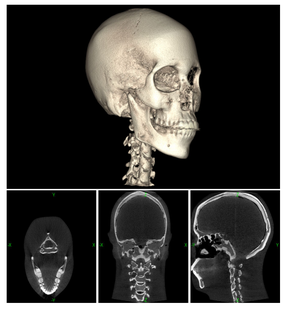
The reconstructed volume of 512^3 voxels resides in global memory of the GPU, in 32-bit floating-point precision. The host CPU loads each projection into GPU texture memory and launches a GPU kernel, which runs a thread for each voxel coordinate. A thread queries the projection at the calculated coordinates using hardware-based bi-linear interpolation, calculates the backprojection value and adds it to the voxel in the global memory. After all projections have been processed, the reconstructed volume is uploaded to the host CPU memory. Backprojection times for CUDA implementation are compared in Table below with timing results for Cg/OpenGL calculations reported in our earlier paper[6]

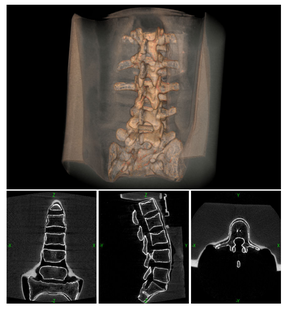
GPU	Computing model	Time (sec)
GeForce 8800 GTX	Cg/OpenGL	5.05
	CUDA	10.6
Tesla C870	CUDA	10.6

Analysis of the CUDA implementation results show that almost half of the run time is spent on the reads and writes to global memory. This suggests a performance optimization strategy of doing larger number of calculation per access to the global memory. The texture memory size allows all the projection data be loaded at once. A thread may then accumulate values from all projections before writing the final result for each voxel into global memory.

The examples of GPU-assisted image reconstruction are shown in Figures below. These experimental C-arm CT results demonstrate CT-like image quality and well suited for interventional and minimally invasive surgical procedures.







DISCUSSION

CUDA-enabled NVIDIA GPUs have an excellent potential for cost-effective implementation of both fluoroscopic and tomographic imaging applications on a mobile C-arm. The simulated fluoroscopy image chain has deterministic performance sufficient for real-time processing of 1024 by 1024 frames with 16-bit pixels depths.

The performance advantages of CUDA technology are best observed for the algorithms that are computationally expensive, but not memory intensive. The number of read/write accesses to global memory has to be substantially smaller than the number of GPU computing instructions. According to the results highlighted in this work, a Cg/OpenGL implementation has advantages over a CUDA implementation when computations are scattered over large datasets, as is the case for the projection-wise approach to FDK reconstruction algorithm.

CUDA Toolkit could make general-purpose computing on GPU more flexible by not requiring graphics-specific programming for non-graphics tasks. For example, extensive OpenGL feature setups are no longer necessary in some cases. In other situations, Cg/OpenGL or explicit graphics constructs in CUDA, such as texture fetch, may be used to simplify numerical code and increase its performance. Architectural limitations of NVIDIA G80 GPU should be also taken into account during software design in order to achieve optimal and deterministic performance in CUDA 1.1.

REFERENCES

[1] Belanger B., Betraoui F., Dhawale P., Gopinath P., Tegzes P., and. Vagvolgyi B., 2006, Development of next generation digital flat panel catheterization system: Design principles and validation methodology, *Proc.of SPIE Medical Imaging*, **6142**, 61421D.

[2] Cheryauka A., Tubbs D., Langille V., Kalya P., Smith B., and R. Cherone, 2008, CT Imaging with a Mobile C-arm Prototype, *Accepted for SPIE'08 Medical Imaging.*

[3] GE Healthcare mobile C-arm products for surgical applications, 2008, http://www.gehealthcare.com/usen/xr/surgery/products/oec9900elite.html

[4] NVIDIA Tesla GPU Computing Solutions for HPC. http://www.nvidia.com/object/tesla_computing_solutions.html

[5] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home.html

[6] Riabkov D., Xue X., Tubbs D., and A. Cheryauka, 2007, GPU-Accelerated Cone-Beam Reconstruction on a Mobile C-arm, *Proc. of High-Performance Image Reconstruction Workshop*, Lindau, July 9-16, 68-71.

[7] Xu F., and K. Muller, 2007, GPU-Acceleration of Attenuation and Scattering Compensation in Emission Computed Tomography, *Proc.of High-Performance Image Reconstruction Workshop*, Lindau, July 9-16, 33-36.

Experience Porting MATLAB Systems Biology Applications to CUDA

LAVA: Laboratory for Computer Architecture at Virginia
University of Virginia, Charlottesville, VA 22904

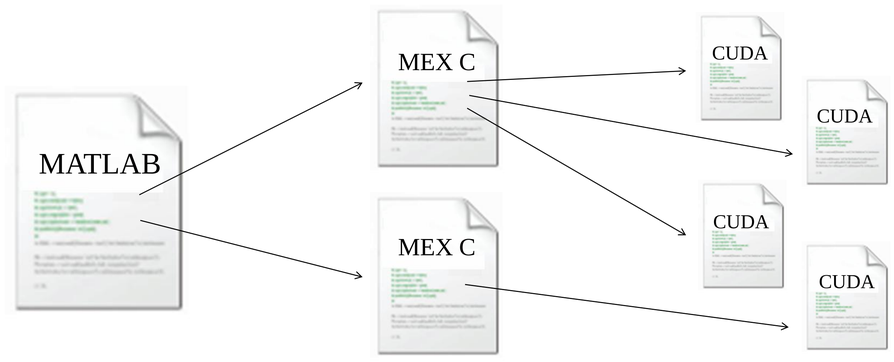
Lukasz Szafaryn, Michael Boyer, Kevin Skadron

http://lava.cs.virginia.edu

This work is supported by a grant from NVIDIA Research and NSF grant no. CNS-0615277.

Introduction

Systems biology seeks to develop an understanding of the myriad interacting components of biological systems. Various modes of biomedical imaging provide a rich source of data for building and testing models. However, accurate modeling requires massive parameterization, which in turn requires image extraction, tracking, and mining for relationships. GPU computing and CUDA offer the potential for substantial speedups. However, direct porting to CUDA is not always possible due to the preference of many systems biologists to use MATLAB. We are investigating how to best obtain the benefits of GPU computing while allowing the biologist to continue using MATLAB. There are usually two conflicting goals involved in accelerating MATLAB applications with CUDA: convenience of porting and performance.



Structure of MATLAB application accelerated with CUDA: interpreted MATLAB code (a) offloads tasks to compiled C code (b) for faster execution, which is further accelerated by offloading parts of work to parallel GPU kernels (c).

Tradeoffs

Developing modular code

- replacing each MATLAB function with equivalent parallelized GPU function
- common routines can be possibly reused in other applications

Hiding specific aspects of GPU programming inside each module

- each module sets up its own GPU execution parameters
- each module performs its own I/O with GPU transparently

Convenience

Performance

Restructuring and combining code

- writing code based on algorithm tasks rather than MATLAB statements
- overlap parallel (often unrelated) tasks by executing them in the same kernel call

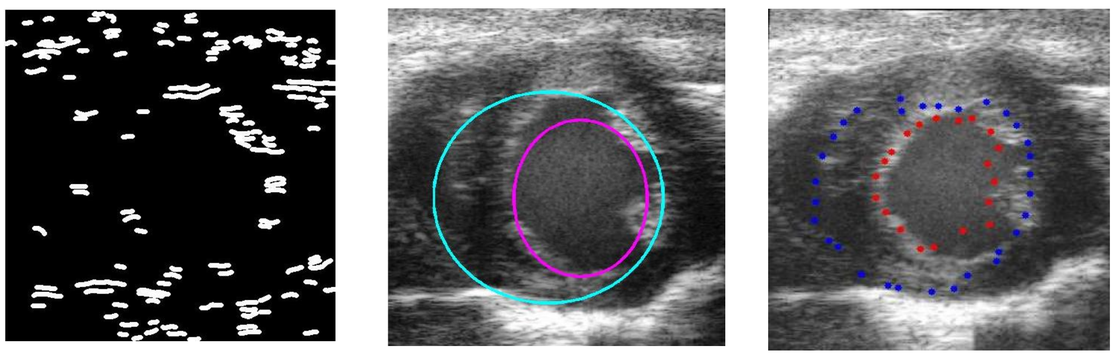
Exposing specific aspects of GPU programming

- doing more work inside each GPU kernel call to fully exploit parallelism and avoid overhead
- performing I/O with GPU manually to eliminate redundant data transfers

Heart Wall Tracking

An Example Application

A particularly challenging application we are working with tracks the movement of the inner and outer walls of a mouse heart over a sequence of 100 640x480 ultrasound images. First, the program performs several image processing operations on the first image to detect initial, partial shapes of heart walls. In order to reconstruct approximated full shapes, the program generates ellipses that best match the partially detected shapes. Ellipses are then superimposed over the image and sampled to mark points on the heart walls. Finally, the program tracks movement of the heart walls by detecting the movement of image areas under sample points as the shapes of the heart walls change throughout the sequence of images.



Stages in Heart Wall Tracking application: heart wall shape detection (a), ellipse matching (b), heart wall shape tracking (c).

Performance Results

The Heart Wall Tracking application illustrates tradeoffs in offloading computation to the GPU. Clearly, data parallelism and sufficient work per kernel are pre-requisites. However, even when the underlying algorithm is not limited by Amdahl's Law, extracting the parallelism from MATLAB may require restructuring and sacrifices in the modularity of the offloaded CUDA computation.

Major Application Part	Parallelizable Part of Code [%]	Original MATLAB Run Time [s]	Convenience Porting, Run Time [s] and Speedup [x]	Performance Porting, Run Time [s] and Speedup [x]
SRAD	89	8.71	1.26 / 6.92	1.17/ 7.42
Hough Search	87	15.87	2.97 / 5.34	2.57 / 6.17
Tracking	37	129.28	115.43 / 1.12	89.78 / 1.44
All	41	187.39	160.16 / 1.17	125.77 / 1.49

Performance numbers were obtained by running application on NVIDIA GeForce 8800GTX.

Future Work

Interesting directions for future work include automated compiler analysis within the MATLAB runtime to perform the necessary restructuring transparently, automatic analysis of whether to offload a CUDA kernel or run it locally on a CPU, and techniques to cope with tightly coupled serial-parallel steps while preserving the overall MATLAB programming "look and feel".

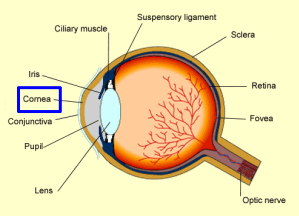
Molecular Dynamics Simulation Using CUDA
Hydrogel for Tissue Engineering:

Seung Soon Jang
Materials Science and Engineering
Georgia Institute of Technology

Jaekyu Lee
Hyesoon Kim
College of Computing
Georgia Institute of Technology

When people get hurt in their cornea.....

We need to develop artificial cornea to treat them.



Hydrogel: Single Network v.s. Double Network

What do we expect from DN hydrogel? → Stronger Mechanical Properties

PAMPS SN PAMPS/PAAm DN

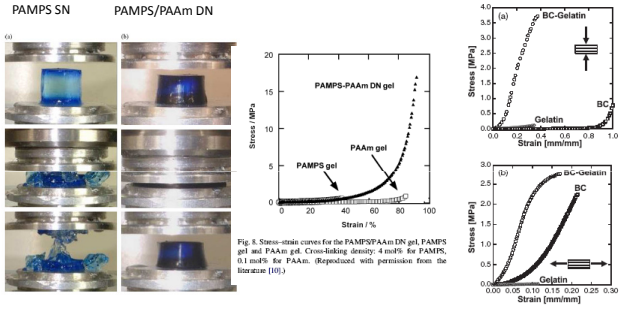


Fig. 8. Stress–strain curves for the PAMPS/PAAm DN gel, PAMPS gel and PAAm gel. Cross-linking density: 4 mol% for PAMPS, 0.1 mol% for PAAm. (Reproduced with permission from the literature [10].)

J. P. Gong et al., Adv. Mater. 2003, 15, 1155

PAMPS: poly (2-acrylamido-2-methylpropanesulfonic acid)
PAAm: poly (acrylamide)

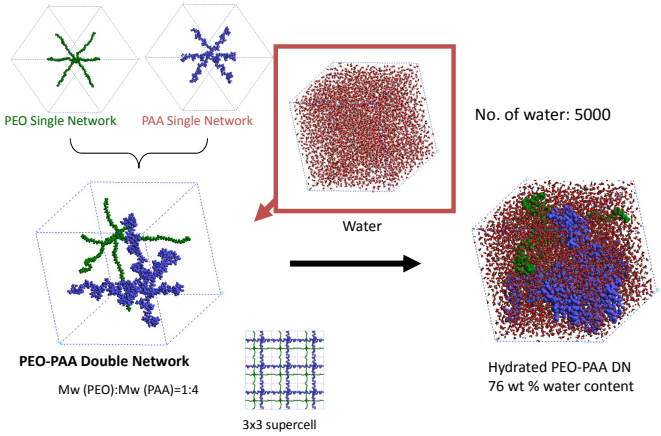
Figure 2. a) Compressive and b) elongational stress–strain curves of BC–gelatin DN, gelatin, and BC gels. The compression and elongation were performed perpendicular to and parallel to the stratified direction of the BC and BC–gelatin DN gels, respectively. The concentration of gelatin in the feed was 30 wt.-%, while the EDC concentration was 1 M.

A. Nakayama et al., Adv. Funct. Mater. 2004, 14, 1124.

BC: Bacterial Cellulose

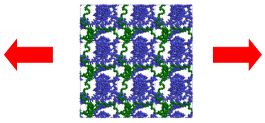
Simulation Study: Double Network (DN) Hydrogel

To understand why the hydrogel gets higher mechanical strength

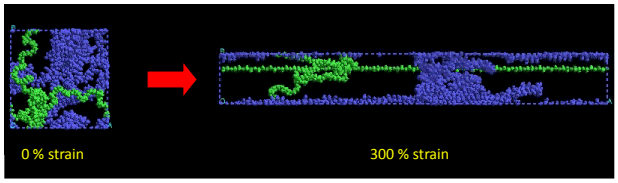


Hydrogel: Mechanical Property

Uniaxial Extension



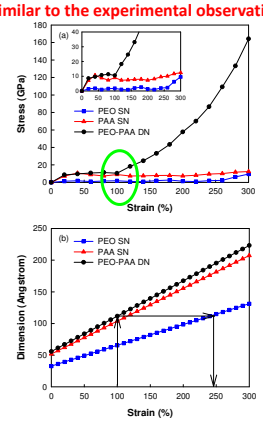
Using NVT MD simulation at 300 K,
the hydrogel dimension in X-axis direction is extended up to 300 % of its original dimension with a constant strain rate while the other two dimensions in Y- and Z-axis direction are shrunk according to the poison ratio (0.5).

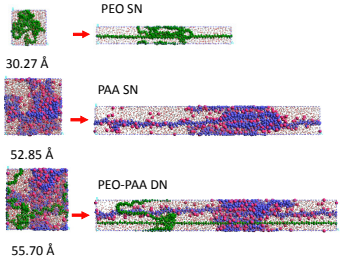


For a clear view of polymer chains, the water molecules are made invisible.

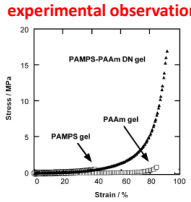
Hydrogel: Mechanical Property

Simulated Stress-Strain Relationship
Similar to the experimental observation

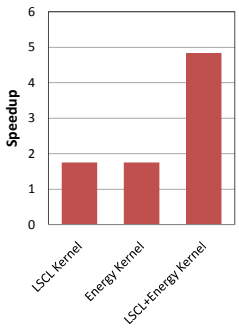


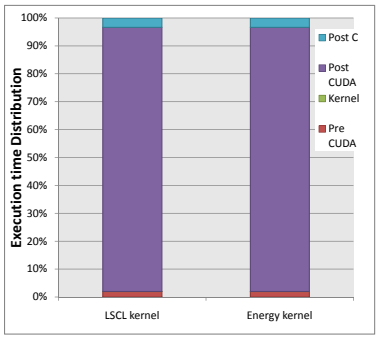


experimental observation



Simulation Using CUDA





Experience Porting General-Purpose Applications to GPUs using CUDA

LAVA: Laboratory for Computer Architecture at Virginia
University of Virginia, Charlottesville, VA 22904
Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Kevin Skadron
http://lava.cs.virginia.edu

This work is supported by a grant from NVIDIA Research and NSF grant nos. IIS-0612049 and CNS-0615277.

Introduction

Graphics processors (GPUs) have become increasingly attractive for general purpose computation as they provide massive parallelism, high memory bandwidth, and general purpose instruction sets. They often contain hundreds of cores and are the best commercially-available example of "many core" processors today.

Motivation

CUDA and GPUs allow study of massively parallel, shared memory programs on commodity hardware. CUDA also offers a novel programming model. Our group is studying scaling bottlenecks in manycore architectures, and this poster summarizes our experiences using CUDA for a variety of applications, as reported in [1]. Instead of evaluating a single application, we explore a set of applications with different parallelism and data-sharing characteristics.

Rodinia Benchmark Suite

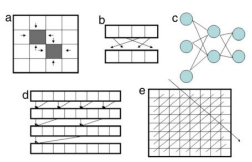
We are releasing our codes as the Rodinia benchmark suite for research in heterogeneous computing, including OpenMP and some FPGA implementations. Other codes and implementations will be added as they are completed. http://lava.cs.virginia.edu/wiki/rodinia

[1] S. Che, et al., A performance study of general-purpose applications on graphics processors using CUDA, *J. Parallel and Distributed Computing,* Elsevier, 2008.

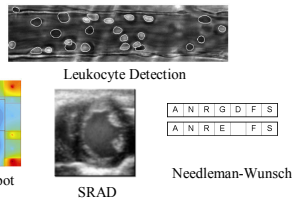
Application Domains

We use Berkeley's "dwarf" taxonomy to select a representative range of application behaviors. Each dwarf represents a set of algorithms with similar computation and data movement patterns.

Structured Grid	**– Leukocyte, SRAD and HotSpot**
Unstructured Grid	**– Back Propagation**
Dynamic Programming	**– Needleman-Wunsch**
Dense Linear Algebra	**– Kmeans**
Combinational Logic	**– DES**



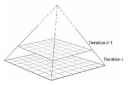
Applications with various sharing patterns



Applications from different domains

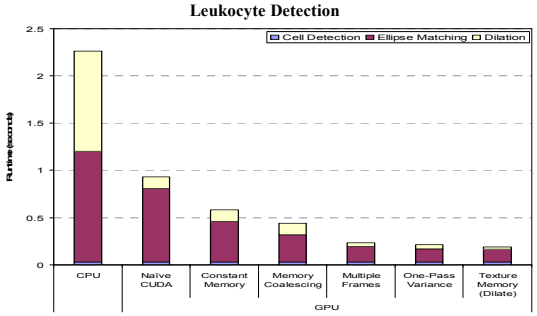
Communication patterns of different applications: (a) in SRAD and HotSpot, the value of each point depends on its neighboring points; (b) DES involves many bit-level permutations; (c) Back Propagation works on a layered neural network; (d) in SRAD and Back Propagation, parallel reductions can be performed using multiple threads; (e) the parallel Needleman–Wunsch algorithm processes the score matrix in diagonal strips

Common Optimization Techniques



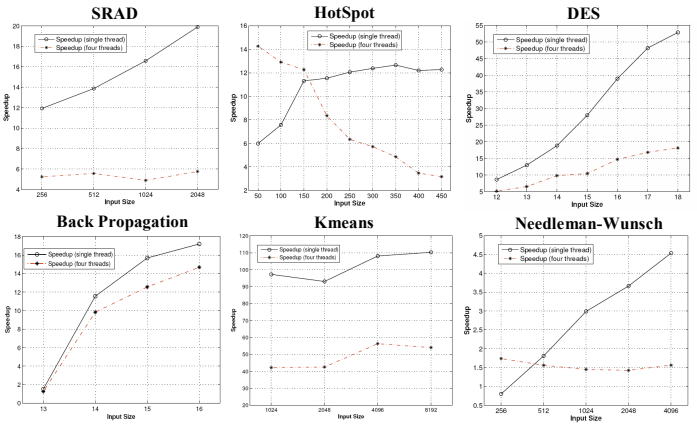
- Iterative solvers, such as our HotSpot thermal solver, benefit from a pyramidal data structure, trading off redundant computation to avoid the synchronization between thread blocks
- Lookup tables are useful for avoiding a large penalty due to branch divergence within SIMT groups (e.g., in the irregular data permutations of DES)
- Localizing data access patterns and inter-thread communication within thread blocks takes advantage of the SM's per-block shared memory (useful in most applications)
- Frequently accessed, read-only values shared across a warp should be placed in cached constant memory (e.g., in Leukocyte).
- Large, read-only data structures with temporal and spatial locality should be accessed as textures to exploit texture caching (e.g., in Kmeans)
- Data access patterns should be organized so that warps access contiguous blocks of memory (e.g., in Leukocyte).

Leukocyte Detection



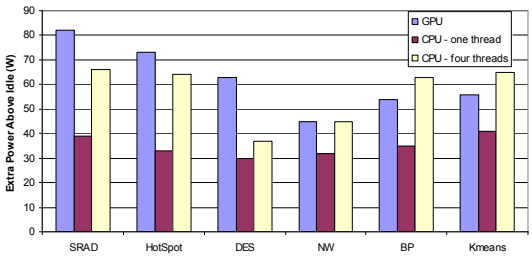
For example, performance of Leukocyte Detection improves by about 30% with constant memory and Kmeans by about 70% with textures.

Application Results



The GPU implementations are developed using CUDA and the multithreaded CPU versions using OpenMP. We compare application performance between an NVIDIA Geforce GTX 280 and an Intel Xeon CPU with two hyperthreaded dual-core processors (3.2 GHz, 2 MB L2 and 4GB main memory). The power result is obtained by subtracting the system power consumed when system is idling (213W) from the system power consumed during execution, and uses an early engineering sample of a GTX 260.

Power Consumption



A Multi-View Stereo Implementation on Massively Parallel Hardware

Mate Beljan, Ronny Klowsky, Michael Goesele
GRIS, TU Darmstadt, Germany

Region-Growing Multi-View Stereo

massively parallel implementation of a recent multi-view stereo approach [1] using the CUDA framework

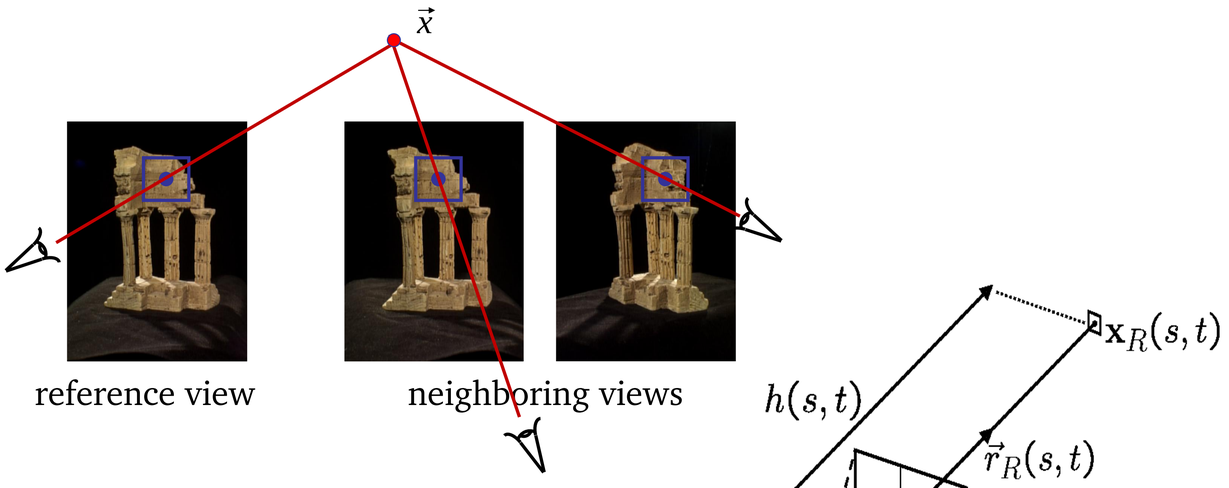
for each image
    select views to match (global)
    find initial matches
    for each pixel in prioritized order
        select views to match (local)
        optimize depth, normal, and color scale
merge resulting depth maps

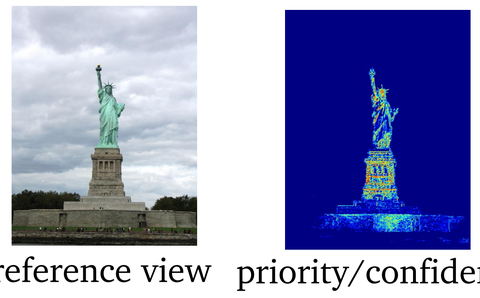
- inner loop (almost) embarrassingly parallel
- some dependencies caused by prioritized processing order
- no inherent rendering process
- large number of image/texture accesses using bilinear interpolation

CUDA Implementation

- use single thread per pixel and reference/neighboring view pair
- align neigborhoods to CUDA warps
  - eight neighbors/threads per reference view pixel
  - four pixels to fill 32 threads in warp
- embarrassingly parallel in kernel execution, no intra-block synchronization

- start kernel on batch of samples
  - front of priority queue
- assume reordering effect negligible in comparison to CPU version
- update priority queue after kernel execution



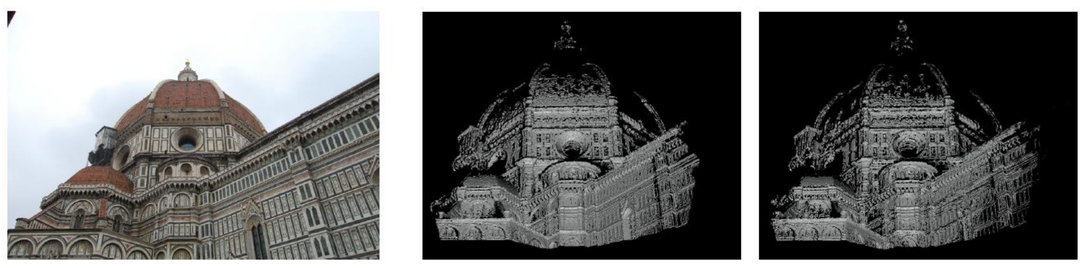

reference view priority/confidence

Analysis of Implementation

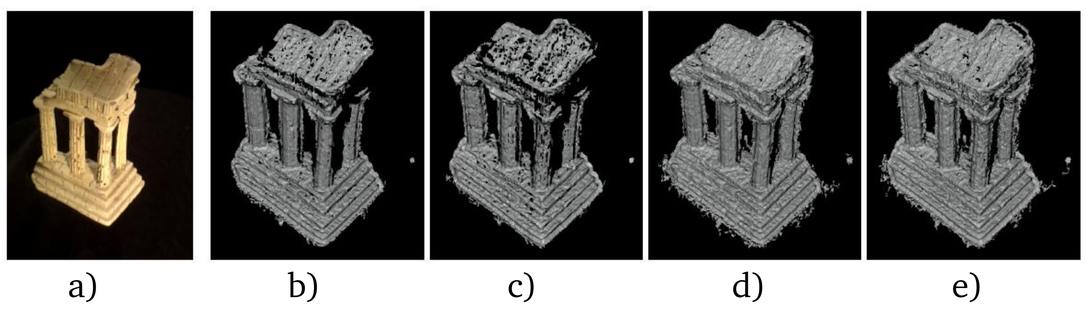
- currently128 threads per block, limited by register usage (on G92)
- non-linear optimization requires significant amount of memory
  - reference view and neighboring views stored as textures
  - intermediate results cached in global memory (yields noticeable increase in execution speed)
- sequential updating of priority queue necessary after each kernel run
  - requires memory transfers between Host and Device

- overall good match to CUDA architecture
- (not yet) full occupancy

Results


a) b) c)

View from the Florence dataset and reconstructed depth maps. b) Optimized CPU version with 8 neighbors and view selection. c) GPU version with 8 neighbors.


a) b) c) d) e)

View 123 from the templeFull dataset and reconstructed depth maps. b) Optimized CPU version with 4 neighbors. c) GPU version with 4 neighbors. d) Optimized CPU version with 8 neighbors and view selection. e) GPU version with 8 neighbors.

configuration (pixels×neighbors)	registers per thread	registers per block	local memory per thread	shared memory per block	total GPU time	recon-structed samples
8×8	83	5312	120 byte	2876 byte	0.592037 s	2191
16×8	62	7936	120 byte	5700 byte	0.496031 s	2191
16×8	48	6144	200 byte	5700 byte	0.516032 s	2191
32×8	32	8192	716 byte	11348 byte	0.464029 s	1843
44×8	16	5632	760 byte	15584 byte	0.864054 s	2191

Table 1: Run times for processing the initial SfM features of View 123 from the templeFull dataset using different configurations of threads (created using the compile flag *maxrregcount*). The number of threads per block is always a multiple of the warp size (32) to avoid partially filled warps.

templeFull dataset	execution time	GPU time	reconstructed samples	time per sample
optimized CPU with 4 neighbors	23.26 s	—	83673	0.278 ms
GPU with 4 neighbors (32×4)	20.81 s	12.36 s	80972	0.257 ms
original CPU with view selection	335.38 s	—	92823	3.613 ms
optimized CPU with view selection	30.27 s	—	99511	0.304 ms
optimized CPU with 8 neighbors	53.40 s	—	99371	0.537 ms
GPU with 8 neighbors (16×8)	49.06 s	34.53 s	99623	0.492 ms

Table 2: Run times and number of reconstructed samples for the MVS algorithms operating on View 123 of the templeFull dataset. Execution time gives the overall process time of the MVS algorithm without file access. GPU time only measures the kernel time and memory transfer between host and GPU. Reconstructed samples gives the number of samples in the computed depth map. Time per sample is the ratio of execution time to number of reconstructed samples.

TECHNISCHE UNIVERSITÄT DARMSTADT

References

[1] M. Goesele, N. Snavely, B. Curless, H. Hoppe, S. Seitz. Multi-View Stereo for Community Photo Collections. ICCV 2007.

Practical Implementation of Helical Cone-Beam CT Imaging using Multiple GPUs

Gábor Jakab[1] Balázs Domonkos[1,2] Tamás Bükki[1]

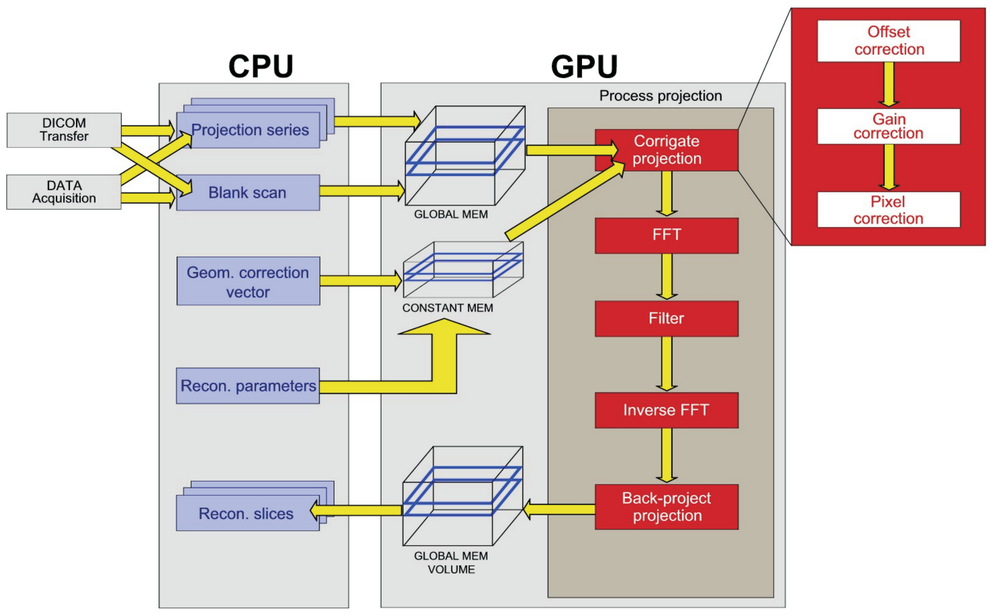
(1) Mediso Medical Imaging Systems (2) Budapest University of Technology and Economics

Introduction

During the last years, small animal imaging devices such as Computed Tomography (CT), Single Photon Emission Computed Tomography (SPECT), and Positron Emission Tomography (PET) became widely used tools for preclinical research. The rapidly growing amount of acquired raw data requires faster and more effective solutions for image reconstruction [3] [2]. Small animal CT systems manufactured by Mediso Medical Imaging Systems based on cone-beam micro-CT scanners belong to the primary domains of our research and development activity. These systems have distinctive requirements compared to the human CT devices. In order to achieve sufficient slice thickness of 20–50 μms for whole body examinations using projection images of 2048×1024 pixels, the reconstructed volume can consist up to 1024^3 voxels. On the other hand, the computer which performs the corrections and the reconstruction have to be compact enough to fit into the hull of the scanner.

Implementation

Former aims cannot be achieved using conventional CPU-based implementations, since either the reconstruction of a volume with such dimensions lasts for several hours or a cluster of computing nodes is required. Inspired by the attractive flops per price ratio the application of high-end commodity graphics hardware can be a promising approach for compute intensive reconstruction algorithms. Thus we have developed a GPU adaptation of the complete practical circular and helical CT reconstruction scheme with projection image preprocessing, filtering, and back-projection (see the block-diagram below). Our CT reconstruction algorithm is based on the well-known FDK method [1], which solves the three-dimensional reconstruction for cone-beam CT scanners. Not only the filtering and the back-projection step of the FDK method, but also preprocessing of the scanned raw projections such as offset, gain, bad pixel, and geometrical corrections are decomposed to parallel processes performed on the GPU.

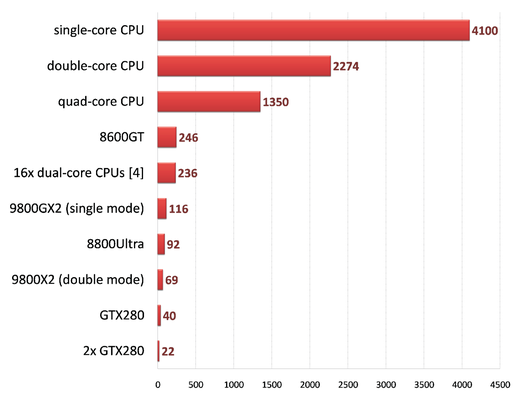


Block-diagram of the reconstruction

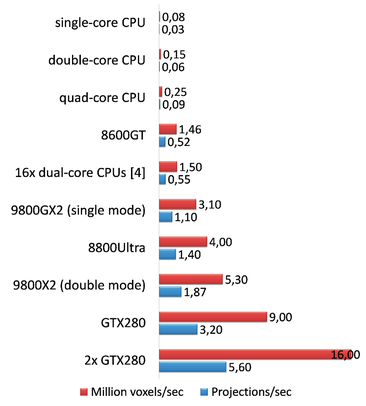
Furthermore, our implementation performs on-the-fly reconstruction, which means that the reconstruction process starts right after the first projection image has been scanned. This enables presenting real time images to the user who can examine the partially reconstructed slices during the data acquisition. The reconstruction is finished immediately after the last projection has been scanned. Moreover, the performance of the system can be improved almost linearly by increasing number of GPUs settled in the same device, since the distribution scheme of the preprocessing-reconstruction pipeline does not require any inter-GPU communication. This can involve up to four GPUs with 4 GBs of total graphics memory, considering the latest high-end graphics hardware and motherboards.

Results

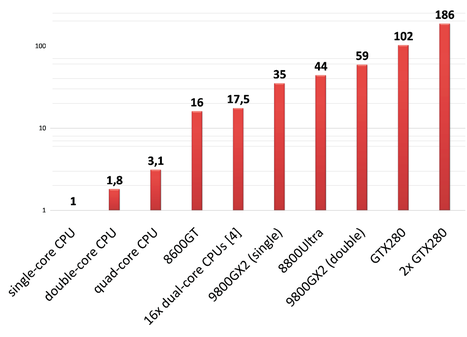
For our experiences we used Mediso NanoScan PET-CT scanner, Core2 6700 (dual-core) and Core2 Q9450 (quad-core) CPUs, NVDIA 8600GT, 8800ULTRA, 9800GX2, and GTX280 GPUs. The volumes were reconstructed on a 512^3 Cartesian Cubic lattice from 360 projection images each consisting of 1024×512 pixels. The resolution of the projections has very small influence on the overall reconstruction time. We have investigated the execution of the reconstruction using ten different hardware configurations. The following charts show the reconstruction times, the voxel and projection procession speeds, and the comparison of the overall performance respectively. Finally, the renderings of the reconstructed objects are presented.



Reconstruction time (seconds)



Procession Speed (1/second)



Performance Speed Up

Mediso NanoScan PET-CT device

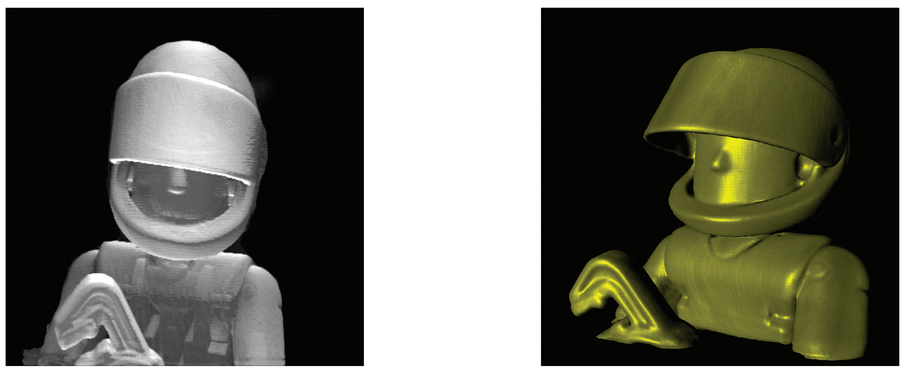
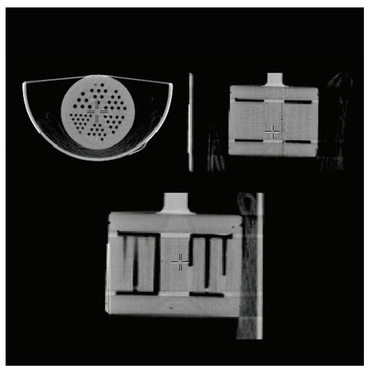


Image of a Lego-man rendered using volume ray casting and first-hit ray casting respectively



Orthogonal slices of a reconstructed Jaszczak phantom

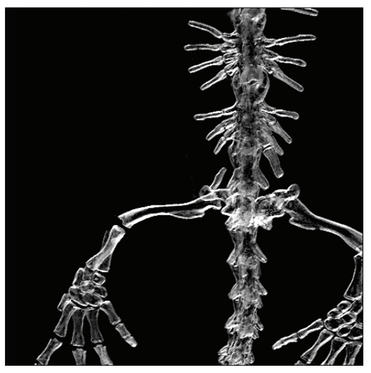


Image of spinal cord of a newt using volume ray casting

Conclusion

We have implemented a medical industry level micro-CT reconstruction application. Using *Common Unified Device Architecture (CUDA)* coupled with *OpenMP*, according to our measured results, 60 times speed-up can be reached compared to the quad-core CPU-based execution.

References

[1] Feldkamp, L. A., Davis, L. C., and Kress, J. W. Practical cone-beam algorithm. Journal of the Optical Society of America 1 (June 1984), 612–619.

[2] Knaup, M., Steckmann, S., Bockenbach, O., and Kachelrieb, M. Tomographic Image Reconstruction using the Cell Broadband Engine (CBE) General Purpose Hardware. In Presentations at Computational Imaging (2007).

[3] Scherl, H., Keck, B., Kowarschik, M., and Hornegger, J. Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). Nuclear Science Symposium Conference Record (2007), 4464–4466.

[4] Fang Xu et al. Real-time 3D computed tomographic reconstruction using commodity graphics hardware. Phys. Med. Biol. 52 3405-3419. (2007)