Potential Uses of GPGPU-Enhanced Cluster Computing in MCAE Simulations for Non-linear Mechanical Dynamics

Robert F. Lucas, Gene Wagenbreth, Dan M. Davis, and Ke-Thia Yao
Information Sciences Institute, Viterbi School of Engineering, University of Southern California
{ rflucas, genew, ddavis, kyao) @isi.edu

Abstract

Newly emerging heterogeneous computing is now making itself useful in Linux clusters in the form of systems with General Purpose Graphics Processing Units (GPGPUs) such as nVidia 8800s, 9800s and Teslas. Experience is showing these are effective in addressing many issues that have been problematic for some time. Sparse systems of linear equations have long been computational bottlenecks in applications ranging from science to optimization. For many problems, including Mechanical Computer Aided Engineering (MCAE), iterative methods are unreliable and the performance of sparse matrix factorization is preferable. Multi-frontal sparse matrix factorization is often favored and, by representing the sparse problem as a tree of dense systems, it maps well to modern memory hierarchies. This allows effective use of BLAS-3 dense matrix arithmetic kernels. Graphics Processing Units (GPUs) are architected differently than their general-purpose hosts and have an order-of-magnitude more floating point processing power. These units were largely single-precision when first released, but contain increasing portions of dual precision registers. This paper explores the hypothesis that GPUs can accelerate the speed of a multi-frontal linear solver, even when only processing a small number of the largest frontal matrices. The authors show that GPUs can more than double the throughput of the sparse matrix factorization, when measured in a realistic "end-to-end" performance test. This in turn promises to offer a very cost-effective speedup to many problems in disciplines such as MCAE. This cost effectiveness appears in computer power per watt, reduced hardware footprint, reduced power and cooling costs and sustainable programming investments. Performance data are presented, as well as future research needs and goals.

1. Introduction

High Performance Computing (HPC) capabilities have become a critical tool in the Forces Modeling and Simulation (FMS) experiments required by DoD planners and commanders. (Messina *et al*., 1997 Lucas *et al*., 2003 & 2009). This has enabled the Joint Concept Development and Experimentation Directorate (J9) at the U.S. Joint Forces Command (JFCOM) to effectively "field" up to 10,000,000 Agent-Based-Model (ABM) entities on a synthetic urban battlespace. (Davis, *et al*., 2008). These large scale simulations typically utilize empirical data, most often lodged in a "look-up table" to control entity behaviors, *e.g*. casualty results from a explosion, and environment changes like the area covered by a chemical weapons plume. While ostensibly effective at providing an acceptably realistic scenario to the experiment participants, who are most often uniformed service members, it often raises questions as to validity. These arise due to the necessity of constricting the range of reactions to a limited and general set of rules.

One of the grand-challenge research issues in FMS could be stated as "Can sufficient computing power be brought to bear on large-scale battlespace simulations to allow for their simultaneous synthesis of ABM and equation-based modeling?" Concomitant issues for legacy simulation programs would include:

- the incorporation of the equation-based model algorithms
- the resolution of new validation concerns
- the selection of the optimal target phenomena
  for conversion from rule-based emulation
  to equation-based simulation.

Heretofore, the major constraint has been computational sufficiency. FMS has been relying on the benefits of Moore's law, which has lasted for decades beyond its original limiting date of 1975 (Moore, 1965). Even so, every advance in computing power has been almost entirely devoted to increasing the number of entities, as has been required when major conflict areas moved from the open fields of Europe to the teeming cities of the Middle East. Attempts at heterogeneous computing using devices like Field Programmable Field Arrays (FPGAs) were interesting, (Abu-Ghazaleh, 2004) but have not been widely adopted. With the advent of the concept of using General Purpose Graphics Processing Units (GPGPUs) as computational accelerators, a new approach seemed possible. This was marginally successful, even when early attempts were made by laboriously porting legacy codes using the color pixel registers of the GPUs and implementing them in the languages developed for screen display. The onerous nature of this activity hampered the use of GPUs, as it reduced overall productivity, as that concept had been developed and defined by the HPCS Project (Kepner, 2003).

The authors believe it now is time to revisit the original question, so below, they report on their results obtained in implementing code from the discipline of Mechanical Computer Aided Engineering (MCAE). They then discuss how it could be used, with GPGPU acceleration, to provide real-time equation-based simulations within the legacy ABM synthetic environments.

2. Forays into GPGPU Acceleration in MCAE

The ISI team first investigated the use of GPGPU acceleration by implementing a series of legacy codes on single and dual processor platforms with from two to eight cores. The basic problem was the solving of a system of linear equations, $Ax = b$, where A is both large and sparse, as this was observed to be a computational bottleneck in many scientific and engineering applications. Current CPU designs were not particularly optimized for the solution of this set of problems, but the architecture of the GPUs appears to be more so. Because users did used to have the easy access to GPGPUs capability, for the past forty years a tremendous amount of research has gone into this problem. That research has explored both direct and iterative methods (Heath *et al*., 1991). This paper focuses on a subset of this large space of numerical algorithms: factoring large sparse symmetric indefinite matrices. Such problems often arise in Mechanical Computer Aided Engineering (MCAE) applications. These applications would find direct utility if incorporated into FMS simulations, modeling blast damage, plume dispersal, electro-magnetic propagation, *etc*. For decades, researchers have sought to exploit novel computing systems to accelerate the performance of sparse matrix factorization algorithms. This paper continues that trend, exploring whether or not one can accelerate the factorization of large sparse matrices, which is already parallelized on a modern multi-core microprocessor, by additionally exploiting graphics processing units (GPUs).

Amelioration of computational bottlenecks which are obviously disruptive, such as sparse matrix factorization, could make good use of GPGPU computing. Previous generations of accelerators, *e.g.* those from Floating Point Systems (Charlesworth *et al*., 1986), were designed for the relatively small market of scientific and engineering applications. Now, a new group of accelerators like GPUs are designed for a mass-market arena; in the present case that is often gaming. There are other acceleration chips, *e.g*. the Sony, Toshiba, and IBM's (STI) Cell (Pham et al,, 2006), and they also are one of the new generation of devices whose market share is growing rapidly, independent of science and engineering. The floating point performance of these new commodity components is outstanding (Harris, 2004). That raises the question as to whether or not any of these can be exploited for scientific computing. The quest to explore broader use of GPUs is often called GPGPU, which stands for General Purpose computation on GPUs (Lastra *et al*., 2004), a term that is adopted for use in this paper.

.

Of the many algorithms that are available, the multifrontal method is particularly attractive, as it transforms the sparse matrix factorization into a hierarchy of dense matrix factorizations. Multifrontal codes can effectively exploit the memory hierarchies of cache-based microprocessors, routinely going out-of-core to disk as needed. With the right data structures, the vast majority of the floating point operations can be performed with calls to highly tuned BLAS3 routines, such as the SGEMM matrix-matrix multiplication routine (Dongarra, 1990), and near peak throughput can be expected. Not surprisingly, all of the major commercial MCAE applications use multifrontal solvers.

Recent GPGPU work has demonstrated that dense, single-precision linear algebra computations, *e.g.*, SGEMM, can achieve very high levels of performance on GPUs (Larson *et al*., 2001; Fatahalian *et al*., 2004; Govindaraju *et al*., 2007).

This in turn led to early efforts to exploit GPUs in multifrontal linear solvers by investigators at USC (Lucas, 2008), ANSYS (Poole, 2008), and AAI (Pierce, 2009). These early efforts compared the performance of early model NVIDIA G80 GPUs to that of single CPU hosts. In the work reported herein, the previous work was extended and a report is made on the performance of a multifrontal linear solver exploiting both a state-of-the-art NVIDIA Tesla C1060 GPU as well as shared memory concurrency on its dual-socket, quad-core Intel Nehalem host microprocessor.

3. Overview of a Multifrontal Sparse Solver

Figure 1 below, depicts the non-zero structure of a small sparse matrix. Coefficients that are initially non-zero are represented by an ‘x’, while those that fill-in during factorization are represented by a ‘*’. Choosing an optimal order in which to eliminate these equations is, in general, an NP-complete problem, so heuristics are used to try to reduce the storage and operations necessary. The multifrontal method treats the factorization of the sparse matrix as a hierarchy of dense sub-problems. Figure.2 depicts the multifrontal view of the matrix in Figure 1. The directed acyclic graph of the order in which the equations are eliminated is called the elimination tree. When each equation is eliminated, a small dense matrix called the frontal matrix is assembled. In Figure 1, the numbers to the left of each frontal matrix are its row indices. Frontal matrix assembly proceeds in the following fashion: the frontal matrix is cleared, it is loaded with the initial values from the pivot column (and row if it’s asymmetric), then any updates generated when factoring the pivot equation’s children in the elimination tree are accumulated. Once the frontal matrix has been assembled, the variable is eliminated. Its Schur complement, represented by the shaded area in Figure 2, is computed as the outer product of the pivot row and pivot column from the frontal matrix. Finally, the pivot equation’s factor (a column of L) is stored and its Schur complement placed where it can be retrieved when needed for the assembly of its parent’s frontal matrix. If a post-order traversal of the elimination tree is used, the Schur complement matrix can be placed on a stack of real values.

$$\begin{array}{c|ccccccccc|}
1 & X & & X & & & & X & & \\
3 & & X & X & & & & & & X \\
2 & X & X & X & & & & * & X & * \\
7 & & & & X & & X & X & & \\
9 & & & & & X & X & & & X \\
8 & & & & X & X & X & * & X & * \\
4 & X & & * & X & & * & X & X & * \\
5 & & & X & & & X & X & X & X \\
6 & & X & * & & X & * & * & X & X
\end{array}$$

Figure 1

Sparse matrix with symmetric non-zero structure

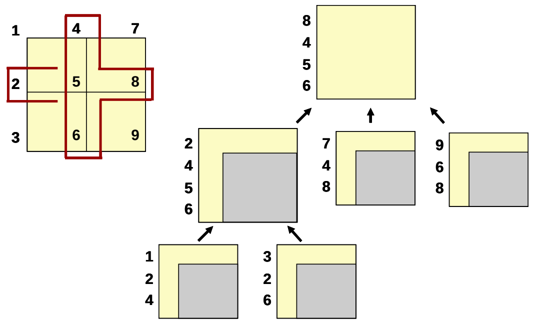

Figure 2

Multifrontal view of sparse matrix from Fig.1

The computational cost of assembling frontal matrices is reduced by exploiting supernodes. A supernode is a group of equations whose non-zero structures in the factored matrix are indistinguishable. For example, zeros filled-in during the factorization of the matrix in Figure 1 convert its last four equations into a supernode. The cost of assembling one frontal matrix for the entire supernode is amortized over the factorization of all the constituent equations, reducing the multifrontal matrices overhead. Furthermore, when multiple equations are eliminated from within the same frontal matrix, their Schur complement can be computed very efficiently as the product of two dense matrices.

The multifrontal code discussed in this paper has two strategies for exploiting the shared-memory, multithreaded concurrency. The frontal matrices at the leaves of the elimination tree can all be assembled and factored independently. At the lower levels in the tree, there can be thousands of such leaves, dwarfing the number of processors, and hence each supernode is assigned to an individual processor. This leads to a breadth-first traversal of the elimination tree, and a real stack can no longer be used to manage the storage of the update matrices (Ashcraft & Lucas, 2001). Near the top of the elimination tree, the number of supernodes drops to less than the number of processors. Fortunately, for the finite element matrices considered in this work, these few remaining supernodes are large, and a right-looking code can be sped up by dividing the matrix into panels and assigning them to different processors.

The objective of the work reported here is to attempt to use GPUs as inexpensive accelerators to factor the large supernodes near the root of the elimination tree, while processing the smaller supernodes near the bottom of the tree by exploiting shared-memory concurrency on the multicore host. This should lead to a significant increase in the throughput of sparse matrix factorization compared to a single CPU. The next section gives a brief description of the NVIDIA Tesla C1060 and its CUDA programming language, highlighting just those features used in this work to factor individual frontal matrices.

4. Graphics Processing Units

The NVIDIA Tesla GPU architecture consists of a set of multiprocessors. Each of the C1060's thirty multiprocessors has eight Single Instruction, Multiple Data (SIMD) processors. This GPU supports single precision (32 bit) IEEE 754 (Arnold *et al*., 1992) formatted floating-point operations. It also supports double precision, but currently at a significantly lower performance. Each SIMD processor can perform two single precision multiplies and one add at every clock cycle. The clock rate on the C1060 card is 1.3 GHz. Therefore, the peak performance is:

$$\textit{1.3 GHz * 3 results/cycle * 8 SIMD/mp * 30 mp = 936 GFlops/s}$$

The ratio of multiplies to adds in matrix factorization is one, so for a linear solver, the effective peak performance is 624 GFlop/s. In practice, the NVIDIA CuBLAS SGEMM routine delivers just over half of that performance.

Memory on the Tesla GPU is organized into device memory, shared memory and local memory. Device memory is large (4 GBytes), is shared by all multiprocessors, is accessible from both host and GPU, and has high latency (over 100 clock cycles). Each multiprocessor has a small (16 KBytes) shared memory that is accessible by all of its SIMD processors. Shared memory is divided into banks and, if accessed so as to avoid bank conflicts, has a one cycle latency. Shared memory should be thought of a user-managed cache or buffer between device memory and the SIMD processors. Local memory is allocated for each thread. It is small and can be used for loop variables and temporary scalars, much as registers would be used. The constant memory and texture memory were not used in this effort.

In the experience of the authors, there are two primary issues that must be addressed to use the GPU efficiently:

- code must use many threads, without conditionals, operating on separate data to keep the SIMD processors busy
- code must divide data into small sets, which can be cached in the shared memory. Once in shared memory, data must be used in many operations (10 – 100) to mask the time spent transferring between shared and device memory.

It is not yet feasible to convert a large code to execute on the GPU. Instead, compute-bound subsets of the code should be identified that use a large percentage of the execution time. Only those subsets should be converted to run on the GPU. Their input data is transferred from the host to the GPU's device memory before initiating computation on the GPU. After the GPU computation is complete, the results are transferred back to the host from the GPU's device memory.

To facilitate general-purpose computations on their GPU, NVIDIA developed the Compute Unified Device Architecture (CUDA) programming language (Buck *et al*., 2008). CUDA is a minimal extension of the C language and is loosely type-checked by the NVIDIA compiler (and preprocessor), *nvcc*, which translates CUDA programs (.cu) into C programs. These are then compiled with the *gcc* compiler and linked as an NVIDIA provided library. Within a CUDA program, all functions have qualifiers to assist the compiler with identifying whether the function belongs on the host of the GPU. For variables, the types have qualifiers to indicate where the variable lives, *e.g*., __device__ or __shared__. CUDA does not support recursion, static variables, functions with arbitrary numbers of arguments, or aggregate data types.

5. Algorithm for Factoring Individual Frontal Matrices on the GPU

It had already been determined that, in order to get meaningful performance using the GPU, it was necessary to both maximize use of the NVIDIA supplied SGEMM arithmetic kernel and to minimize data transferred between the host and the GPU. See the author's work, previously published at other conferences (Lucas et al., 2007). It was decided to adopt the following strategy for factoring individual frontal matrices on the GPU:

- Download the factor panel of a frontal matrix to the GPU. Store symmetric data in a square matrix, rather than a compressed triangular.
- Use a left-looking factorization, proceeding over panels from left to right:
    - Update a panel with SGEMM
    - Factor the diagonal block of the panel
    - Eliminate the off-diagonal entries from the panel
- Update the Schur complement of this frontal matrix with SGEMM
- Return the entire frontal matrix to the host, converting back from square to triangular storage
- Return an error if the pivot threshold was exceeded or a diagonal entry was equal to zero

Table 1.
Log of time spent factoring a model frontal matrix

Method Name	GPU msec	%GPU time
Copy data to and from GPU	201.0	32.9%
Factor 32x32 diagonal blocks	42.6	7.0%
Eliminate off diagonal panels	37.0	6.1%
Update with SGEMM	330.6	54.1%
Total time	611.4	100.0%

Table 1 represents the time log for factoring a large simulated frontal matrix with the fully optimized CUDA factorization code. This timing was taken when the GPU was eliminating 3072 equations from 4096. Approximately half of the execution time on the GPU is spent in SGEMM. Eliminating off-diagonals and factoring diagonal blocks takes only 13% of the time. The remaining third of the time is spent realigning the matrices and copying data to and from the host. A further 0.029 seconds are spent on the host, and not reflected in Table 1. The computation rate for the entire dense symmetric factorization is 163 GFlops/s. In contrast, four cores of the Intel Xeon Nehalem host achieve 29 GFlop/s when factoring the same sized frontal matrix and using the same 32-column panel width. Performance results using the GPU to factor a variety of model frontal matrices is presented in Table 2. These range in the number of equations eliminated from the frontal matrix (size) as well as the number of equations left in the frontal matrix, *i.e.*, its external degree (degree). As expected, the larger the frontal matrix gets, the more operations one has to perform to factor it, and the higher the performance of the GPU.

Table 2.
Performance of the GPU frontal matrix factorization kernel

Size	Degree	Seconds	GFLOPS
1024	1024	0.048	51.9
1536	1024	0.079	66.3
2048	1024	0.117	79.7
512	2048	0.045	60.2
1024	2048	0.079	86.5
1536	2048	0.123	101.3
2048	2048	0.179	112.2
512	3072	0.076	74.7
1024	3072	0.128	103.9
1536	3072	0.188	122.4
2048	3072	0.258	136.0
512	4096	0.116	84.0
1024	4096	0.185	118.3
2048	4096	0.361	150.9

6. Performance of the Accelerated Multifrontal Solver

In this section the performance impact of the GPU on overall multifrontal sparse matrix factorization is examined. The matrix extracted from the LS-DYNA MCAE application is used. It is derived from a three dimensional problems composed of three cylinders nested within each other, and connected with constraints. The rank of this symmetric matrix is 760320 and its diagonal and lower triangle contain 29213357 non-zero entries. After reordering with Metis, it takes 7.104E+12 operations to factor the matrix. The resulting factored matrix contains 1.28E+09 entries. Figure 3 displays the number of supernodes per level in the elimination tree for the three cylinder matrix, along with the number of operations required to factor the supernodes at each level. Notice that the vast majority of the operations are in the top few levels of the tree, and these are processed by the GPU.

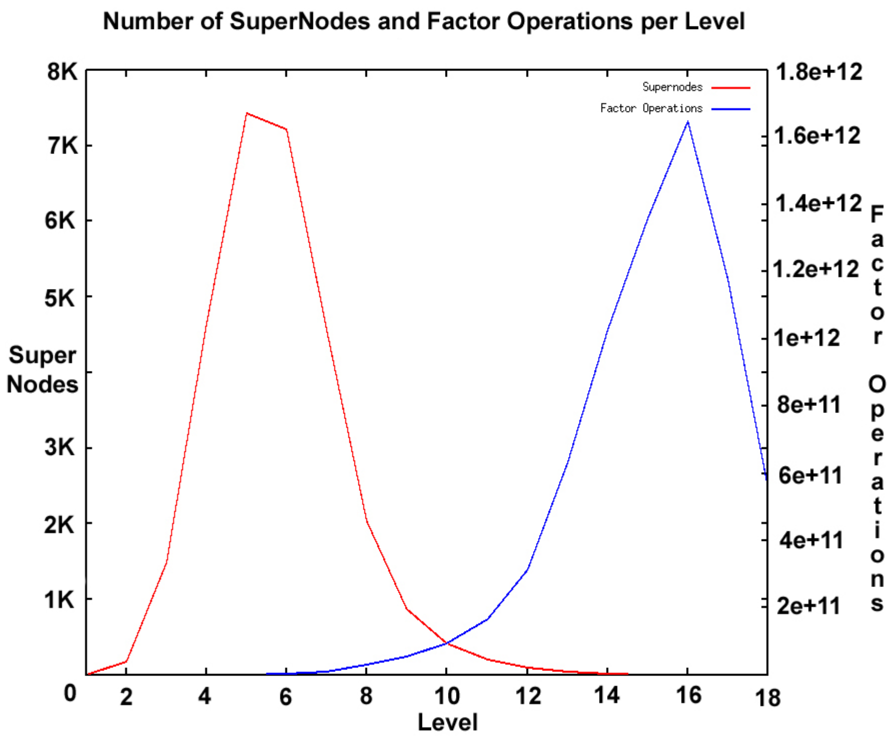

Figure 3.
Number of supernodes and factor operations per level in the tree.

Figure 4 plots the performance achieved by the multicore host when factoring the supernodes at each level of the tree. Note, near the leaves, performance is nowhere near the peak. This is true even for one core, as the supernodes are too small to facilitate peak SGEMM performance. As multiple cores are used, relatively little speedup is observed, which is likely due to the relatively low ratio of floating point operations to memory loads and stores for these small supernodes, leaving them memory bound on the multicore processor.

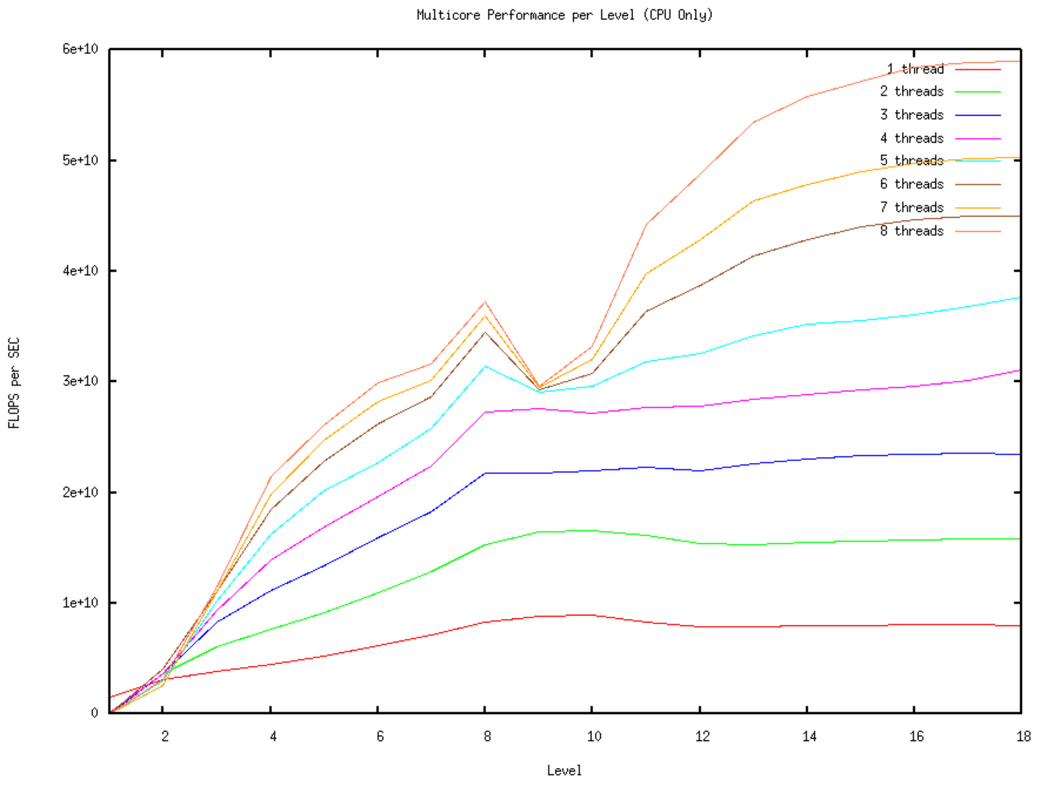

Figure 4.
Multicore performance per level in the elimination tree.

7. Summary and Conclusions

These sections have demonstrated that a GPU can in fact be used to significantly accelerate the throughput of a multi-frontal sparse symmetric factorization code, even when exploiting shared memory concurrency on the host multicore microprocessor. It has been demonstrated that there is a factorization speed-up of 5.91 relative to one core on the host, and 1.34 when using eight cores. This was done by designing and implementing a symmetric factorization algorithm for the NVIDIA C1060 in the CUDA language and then offloading a small number of large frontal matrices, containing over 90% the total factor operations, to the GPU.

However, more work needs to be done before the use of GPUs will be common for the numerical aspects of such applications. The GPU frontal matrix factorization code implemented for this experiment should be revisited to make it more efficient in its use of memory on the GPU. It should be modified to implement pivoting so that indefinite problems can be factored entirely on the GPU. Further, it should be extended to work on frontal matrices that are bigger than the relatively small device memory on the GPU, much as the multifrontal code goes out-of-core when the size of a sparse matrix exceeds the memory of the host processor.

Finally, the question arises: “If one GPU helps, why not more?”. Researchers have been implementing parallel multifrontal codes for over two decades (Duff, 1993). In fact, the multifrontal code used in these experiments has both MPI constructs. Therefore exploiting multiple GPUs is not an unreasonable thing to consider. However, when it is considered that an implementation would have to simultaneously overcome both the overhead of accessing the GPU as well as the costs associated with communicating amongst multiple processors; it may be very challenging to efficiently factor one frontal matrix with multiple GPUs.

In examining the use of this technology in FMS, a number of important simulation concerns come to mind.

- Bomb damage assessment

- Chemical weapons plume dispersal
- Electromagnetic propagation
- Armor damage from IED's or anti-tank rounds
- Blast effects on personnel

All of these have characteristics that are optimally exploited by Single Instruction Multiple Data (SIMD) algorithms. Most extensively use Fast Fourier Transforms (FFT) or Matrix Multiply functions. These types of algorithms are particularly amenable to GPGPU heterogeneous computing. The potential benefits of extending FMS into this realm of computing are manifold, but un-quantified at this time. It should be possible to determine the optimal ratio of GPGPUs to CPU cores that would provide the extra computing power to handle these SIMD compute intensive calculations and do this without unduly disrupting the CPU's calculations for the ABM simulation and synthetic environment.

Acknowledgments

The authors would like thank Norbert Juffa, Stan Posey, and Peng Wang of NVIDIA for their encouragement and support for this work. This has included guidance in performance optimization as well as access to the latest NVIDIA GPUs. Thanks are also due to the excellent staffs at JFCOM, ASC-MSRC and MHPCC. Some of this material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-05-2-0204. Other work is based on research sponsored by the U.S. Joint Forces Command via a contract with the Lockheed Martin Corporation and SimIS, Inc. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations.

References

Abu-Ghazaleh, N., Linderman, R., Hillman, R., & Hanna, J., (2004), Exploiting HHPC for parallel discrete event simulation, in the Proceedings of the HPCMP Users Group Conference,

Arnold, M.G., T.A. Bailey, J.R. Cowles & M.D. Winkel, (1992), Applying Features of IEEE 754 to Sign/Logarithm Arithmetic, IEEE Transactions on Computers, August 1992, Vol. 41, No. 8, pp. 1040-1050

Ashcraft, C. and Robert Lucas, (2001) A Stackless Multifrontal Method, Tenth SIAM Conference on Parallel Processing for Scientific Computing, March, 2001

Buck, I. (2008), GPU Computing: Programming a Massively Parallel Processor, International Symposium on Code Generation and Optimization, San Jose, California

Charlesworth, A., and J. Gustafson, (1986), Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer, in IEEE Computer, vol. 19, issue 3, pp 10-23, March 1986

Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, (1990) A Set of Level 3 Basic Linear Algebra Subprograms, , ACM Transactions on Mathematical Software 16(1):1-17, March 1990

Duff, Ian and John Reid, (1983), The Multifrontal Solution of Indefinite Sparse Symmetric Linear Systems, ACM Transactions on Mathematical Software, 9 (1983), pp 302-335

Duff, Ian, (1986) Parallel Implementation of Multifrontal Schemes, Parallel Computing, 3 (1986), pp 193-204.

Fatahalian, K., J. Sugarman, and P. Hanrahan, (2004), Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, In Proceedings of the ACM Sigraph/Eurographics Conference on Graphics hardware, pages 133-138, Eurographics Association, 2004

Govindaraju, N. and D. Manocha, (2007), Cache-Efficient Numerical Algorithms Using Graphics Hardware, University of North Carolina Technical Report, 2007.

Harris., M. J.(2004), GPGPU: Beyond graphics., Eurographics Tutorial, Grenoble, France August 2004.

Heath, M., E. Ng, and B. Peyton, (1991), Parallel algorithms for sparse linear systems, Society for Industrial and Applied Mathematics Review. 33 (1991), pp. 420-460

Karypis G. and V. Kumar, (1995) A fast and high quality multilevel scheme for partitioning irregular graphs, International Conference on Parallel Processing, pp. 113-122, 1995

Kepner, J., (2004), HPC Productivity: An Overarching View, International Journal of High Performance Computing Applications, Volume 18, Issue 4, Pages: 393 - 397

Larson, S.E. and David McAllister, (2001) Fast matrix multiplies using graphics hardware, In Proceedings of the 2001 ACM/IEEE conference on Supercomputing, pages 55-55, ACM Press, 2001

Lastra, M., A. Lin, and D. Minocha, (2004) ACM Workshop on General Purpose Computations on Graphics Processors. 2004

Lucas, R., & Davis, D., (2003), Joint Experimentation on Scalable Parallel Processors, 2003 I/ITSEC Conference, Orlando, FL

Lucas, R.F., Davis, D.M. & Wagenbreth, G., (2007), "Implementing a GPU-Enhanced Cluster for Large-Scale Simulations," in the Proceedings of the Interservice/Industry Simulation, Training and Education Conference, Orlando, Florida, 2007

Lucas, R.F., (2008), GPU-Enhanced Linear Solver Results, in the proceedings of Parallel Processing for Scientific Computing, SIAM, 2008

Lucas, R.F., Wagenbreth, G., Tran, J.J., & Davis, D. M., (2007), Multifrontal Computations on GPUs, unpublished ISI White Paper, on line at: www.isi.edu/~ddavis/JESPP/2007_Papers/SC07/mf2_gpu_v0.19a-nms.doc

Messina, P. C., Brunett, S., Davis, D. M., Gottschalk, T. D., (1997) Distributed Interactive Simulation for Synthetic Forces, In Mapping and Scheduling Systems, International Parallel Processing Symposium, Geneva

Moore, G.E., (1965), Cramming more components onto integrated circuits, *Electronics Magazine*, Vol 38, Nr. 8, April 19, 1965

Pham, D. C., T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J.
Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J.
Warnock, Stephen Weitzel, Dieter Wendel, and K. Yazawa, (2006) Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor, IEEE Journal of Solid State Circuits, Vol. 41, No. 1, January 2006

Pierce, D., (2009), retrieved from the Internet on 10 May 2010, from: cqse.ntu.edu.tw/cqse/download_file/DPierce_20090116.pdf

Poole, G., (2005), RFLucas in private communication with Gene Poole, ANSYS Inc., at SC|08, Nov 2008, Austin, TX