Multifrontal Computations on GPUs and Their Multi-core Hosts

Robert F. Lucas[1], Gene Wagenbreth[1], Dan M. Davis[1], and Roger Grimes[2]

[1] Information Sciences Institute, University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90230
{rflucas, genew, ddavis}@isi.edu
[2] Livermore Software Technology Corporation
7374 Las Positas Rd
Livermore, California 94551
grimes@lstc.com

Abstract. The use of GPUs to accelerate the factoring of large sparse symmetric indefinite matrices shows the potential of yielding important benefits to a large group of widely used applications. This paper examines how a multifrontal sparse solver performs when exploiting both the GPU and its multi-core host. It demonstrates that the GPU can dramatically accelerate the solver relative to one host CPU. Furthermore, the solver can profitably exploit both the GPU to factor its larger frontal matrices and multiple threads on the host to handle the smaller frontal matrices.

Keywords: GPU acceleration, GPGPU, multifrontal algorithms, MCAE.

1 Introduction

Solving the system of linear equations Ax = b, where A is both large and sparse, is a computational bottleneck in many scientific and engineering applications. Therefore, over the past forty years, a tremendous amount of research has gone into this problem, exploring both direct and iterative methods [1]. 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. 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).

The GPU is a very attractive candidate as an accelerator to ameliorate a computational bottleneck such as sparse matrix factorization. Unlike previous generations of accelerators, such as those designed by Floating Point Systems [2] for the relatively small market of scientific and engineering applications, current GPUs are designed to improve the end-user experience in mass-market arenas such as gaming. Together with other niche chips, such as Sony, Toshiba, and IBM's (STI) Cell [3], they are a new generation of devices whose market share is growing rapidly, independently of science and engineering. The extremely high peak floating point performance of these new commodity components begs the question as to whether or not they can be exploited to increase the throughput and/or reduce the cost of applications beyond the markets for which they are targeted. The quest to explore broader use of GPUs is often called GPGPU, which stands for General Purpose computation on GPUs [4].

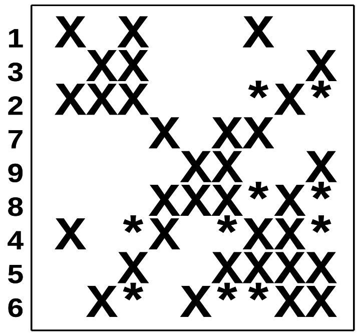
There are many algorithms for factoring large sparse linear systems. The multifrontal method [5] is particularly attractive, as it transforms the sparse matrix factorization into a hierarchy of dense matrix factorizations. Multifrontal codes can effectively exploit the memory hierarchies of cache-based 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 [6], and near peak throughput is 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 [7][8][9]. This in turn led to early efforts to exploit GPUs in multifrontal linear solvers by investigators at USC [10], ANSYS [11], and AAI [12]. These early efforts compared the performance of early model NVIDIA G80 GPUs to that of single CPU hosts. In the work reported herein, we extend the previous work and report 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.

The remainder of the paper is organized as follows. The next section provides a brief overview of the multifrontal method and illustrates how it turns a sparse problem into a tree of dense ones. This is followed by a brief overview of the NVIDIA Tesla C1060 GPU used in this experiment. We discuss both the unique nature of its architecture as well as its CUDA programming language. Section IV presents our strategy for factoring individual frontal matrices on the GPU and provides performance results on the GPU. Section V presents the impact on the overall performance of the multifrontal sparse solver of utilizing both shared memory parallelism and the GPU. Finally, we summarize the results of our experiment and suggest directions for future research.

2. Overview of a Multifrontal Sparse Solver

Fig.1 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, such as METIS [13], 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. **Fig.2** depicts the multifrontal view of the matrix in **Fig.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 (the shaded area in **Fig.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.



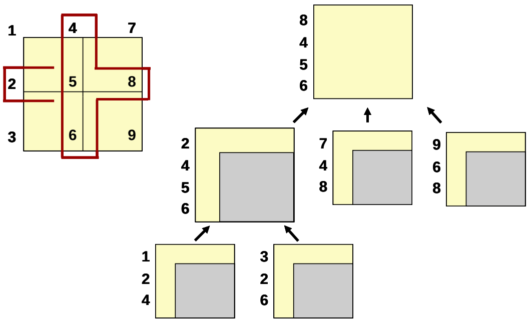


Fig.1 Sparse matrix with symmetric non-zero structure

Fig.2 Multifrontal view of sparse matrix from Fig.1

The 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 **Fig.1** turn 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.

Fig.3 depicts a finite element grid generated by the LS-DYNA MCAE code (www.lstc.com). The matrix for the grid in Fig 3 is relatively small, having only 235,962 equations. Matrices with two orders-of-magnitude more equations are routinely factored today. Factoring such large problems can take many hours, a time that is painfully apparent to the scientists and engineers waiting for the solution.

Fig.4 illustrates the elimination tree for the matrix corresponding to the grid in Fig 3, as ordered by METIS. This particular elimination tree has 12,268 relaxed [14] supernodes in it. There are thousands of leaves and one root. The leaves are relative small, O(10) equations being eliminated from O(100). The supernodes near the root are much bigger. Hundreds of equations are eliminated from over a thousand.

Because dense factor operations scale as order N^3, approximately two-dozen supernodes at the top of the tree contain half of the total factor operations.

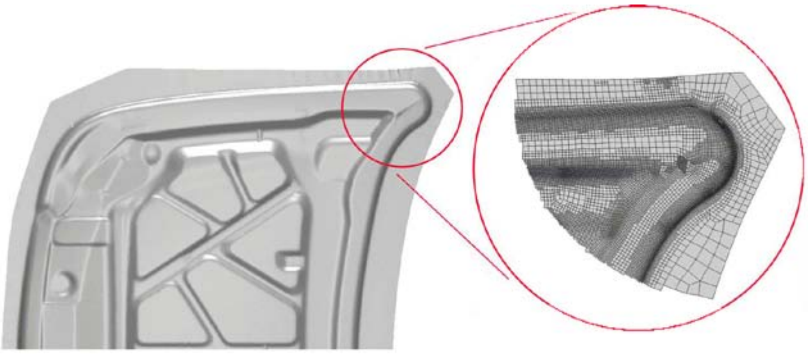


Fig.3: Example of an MCAE Finite Element Problem and Grid (courtesy LSTC)

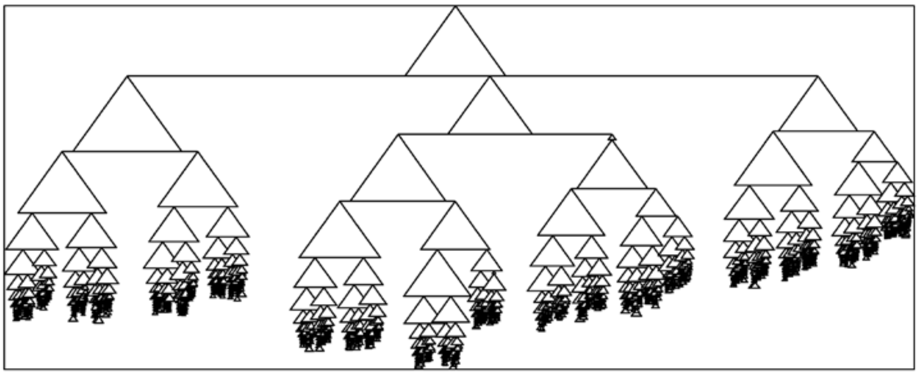


Fig. 4 Supernodal elimination tree for problem in Figure 3 (courtesy Cleve Ashcraft)

The multifrontal code discussed in this paper has two strategies for exploiting 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 [15]. 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.

3. 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 [16] formatted floating-point operations. It also supports double precision, but 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:

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 our experience, 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 [17]. 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.

4. Algorithm for Factoring Individual Frontal Matrices on the GPU

In earlier work, we determined that, in order to get meaningful performance using the GPU, we had to both maximize use of the NVIDIA supplied SGEMM arithmetic kernel and minimize data transferred between the host and the GPU. We 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. This wastes storage, but is easy to implement.
- 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%

The time log for factoring a large, simulated frontal matrix with the fully optimized CUDA factorization code is in Table 1. 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	Secs	GFlop/s
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
1536	4096	0.267	137.3
2048	4096	0.361	150.9

5. Performance of the Accelerated Multifrontal Solver

In this section we examine the performance impact of the GPU on overall multifrontal sparse matrix factorization. We will use a matrix extracted from the LS-DYNA MCAE application. 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 5 plots the time is takes to factor the matrix, as a function of the number of cores employed, both with and without the GPU. The dual socket Nehalem host sustains 10.3 GFlop/s when using one core, and 59.7 GFlop/s when using all eight. When the GPU is employed, it performs 6.57E+12 operations, 92% of the total, and sustains 98.1 GFlop/s in doing so. The overall performance with the GPU improves to 61.2 GFlop/s when one host core is used, and 79.8 GFlop/s with all eight. For perspective, reordering and symbolic factorization take 7.9 seconds, permuting the input matrix takes 2.64 seconds, and the triangular solvers take 1.51 seconds.

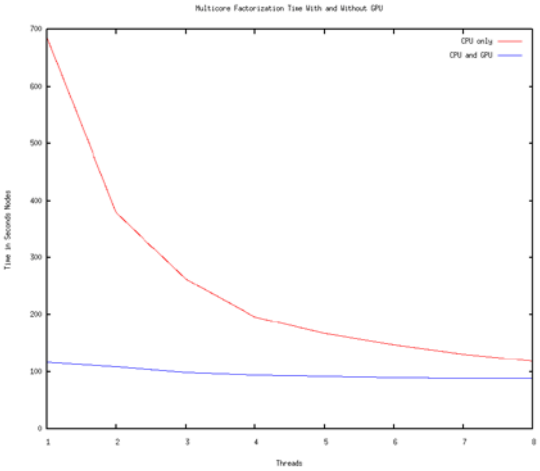


Fig. 5 Multicore factorization time, with and without the GPU.

To understand why there seems to be so little speedup when the GPU-enhanced solver goes from one core to eight, consider Figure 6. It 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 7 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.

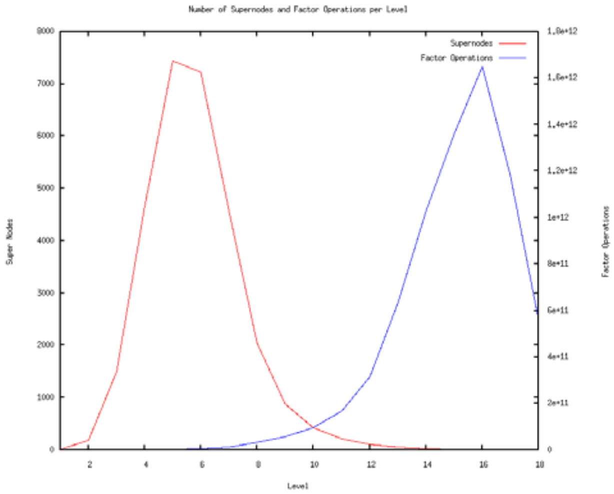


Fig. 6 Number of supernodes and factor operations per level in the tree.

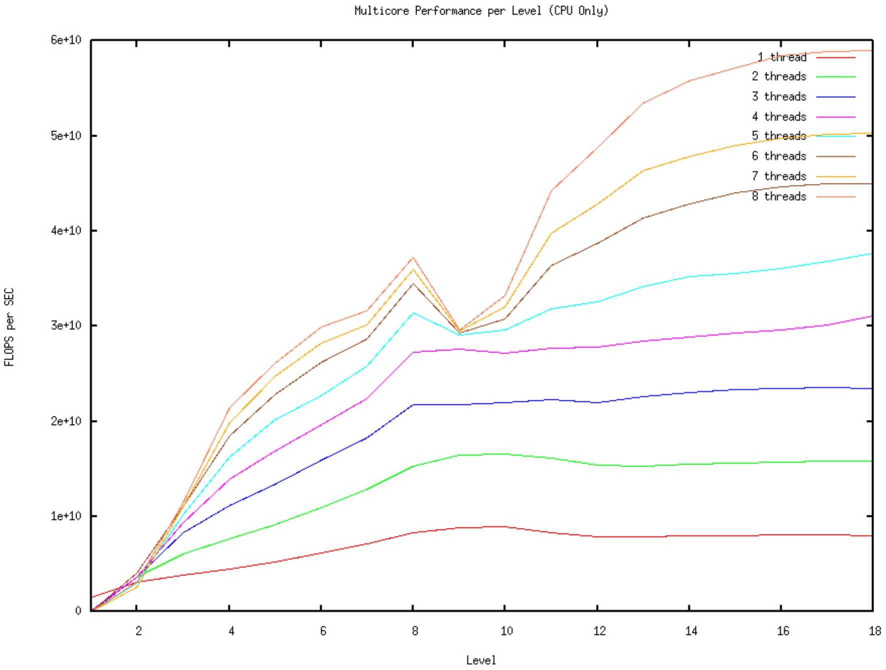


Fig. 7 Multicore performance per level in the elimination tree.

6. Summary

This paper has 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. We have demonstrated 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.

The authors have recently received a preproduction NVIDIA C2050 (Fermi) GPU, which provides double precision at one half the performance of single precision, much like the Pentium host's SSE function units. We will rerun the experiments reported above in double precision and report the results at VecPar 2010. We expect no change to our overall conclusion that GPUs can accelerate the shared memory multifrontal code.

We believe that by demonstrating that the GPU can be successfully exploited in a production quality, double precision multifrontal code, we will have taken the next logical step towards integrating GPUs into MCAE applications. 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, if one GPU helps, why not more? Researchers have been implementing parallel multifrontal codes for over two decades [18]. 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 one considers that one 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.

Acknowledgement

We 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.

References

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

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

[3] 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, 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

[4] A. Lastra, M. Lin, and D. Minocha, *ACM Workshop on General Purpose Computations on Graphics Processors*. 2004

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

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

[7] E. Scott Larson and David McAllister, Fast matrix multiplies using graphics hardware, In Proceedings of the *2001 ACM/IEEE conference on Supercomputing*, pages 55-55, ACM Press, 2001

[8] Fatahalian, K., J. Sugarman, and P. Hanrahan, 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

[9] Govindaraju, N. and D. Manocha, Cache-Efficient Numerical Algorithms Using Graphics Hardware, *University of North Carolina Technical Report*, 2007.

[10] Lucas, R.F., GPU-Enhanced Linear Solver Results, in the proceedings of *Parallel Processing for Scientific Computing, SIAM,* 2008

[11] Private communication with Gene Poole, ANSYS Inc., at SC|08, Nov 2008, Austin, TX

[12] cqse.ntu.edu.tw/cqse/download_file/DPierce_20090116.pdf

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

[14], Cleve Ashcraft and Roger Grimes, The Influence of Relaxed Supernode Partitions on the Multifrontal Method, *ACM Transactions in Mathematical Software*, 15 (1989), pp. 291-309

[15] Cleve Ashcraft and Robert Lucas, A Stackless Multifrontal Method, Tenth SIAM Conference on Parallel Processing for Scientific Computing, March, 2001

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

[17] Buck, I. GPU Computing: Programming a Massively Parallel Processor, *International Symposium on Code Generation and Optimization*, San Jose, California

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