Electromagnetic Device Optimization: The Forking of Already Parallelized Threads on Graphics Processing Units

S. Ratnajeevan H. Hoole [1], Sivamayam Sivasuthan [1], Victor U. Karthik [1], Arunasalam Rahunanthan [2], Ravi S. Thyagarajan [3], and Paramsothy Jayakumar [3]

[1] Department of Electrical and Computer Engineering
Michigan State University, East Lansing, MI 48824, USA
srhhoole@gmail.com, sivasuth@msu.edu, uthayaku@msu.edu

[2] Department of Mathematics and Computer Science
Edinboro University, Edinboro, PA 16444, USA.
rahunanthana@gmail.com

[3] U.S. Army Tank Automotive Research, Development & Engineering Center, Warren, MI 48397, USA
ravi.s.thyagarajan.civ@mail.mil, paramsothy.jayakumar.civ@mail.mil

Abstract ─ In light of the new capability to fork an already parallelized kernel on a GPU, this paper shows how the use of the parallelization capabilities of a PC's Graphics Processing Unit (GPU) makes the finite element design of coupled problems (such as the electroheat shape optimization problems we work with) realistic and practicable in terms of computational time.

Index Terms - Finite elements, GPU computing, inverse problems, parallelization.

I. INTRODUCTION: INVERSE PROBLEMS

In contrast to the forward problem (Fig. 1) that we normally solve, inverse problems are more realistic in device design going from the bottom to the top of that figure, in such design tasks as, say, compute the size and other descriptions of a motor that can produce so much torque. Figure 2 shows the design cycle for an inverse problem as a repeating cycle of forward problems. In the first step, the design parameter set $\bar{h}$ is randomly selected (or estimated by a subject expert), and thereupon we generate the parameter based mesh, get the finite element solution, measure the object value (often conveniently defined as a least square difference between design objects desired and those computed) and check whether it is minimum or not. If this is minimum, we terminate the loop; otherwise we change the design parameters and do the same procedure again.

This procedure repeats until the object value goes to its minimum. This solution process however, is computationally intensive. To address this problem, parallelization on GPU threads has been proposed [1-2]. Each finite element solution in its matrix solution part is computationally intensive [3,4] and GPU parallelization significantly reduces solution time. But in genetic algorithm optimization [5,6], several copies of the matrix are held on the GPU and the corresponding solutions attempted. This runs into the memory limits of GPUs, newly at 12 GB from around the time of the initial submission of this paper [7].

In this paper therefore, we look more deeply at using the GPU to do the optimization in parallel. We examine memory limits and use the recently revived element-by-element finite element method for speedy finite element matrix solutions on the GPU [8,9] to address memory concerns and exploit such matrix solution speedups to obtain a speedup

Submitted On: March 13, 2014
Accepted On: September 6, 2014

of 28 for genetic algorithm based coupled field optimization. Where we are allowed to fork only one computational kernel and not allowed to fork that kernel into further parallelizable processes on a GPU, we delve into important considerations for choosing which one kernel to fork.

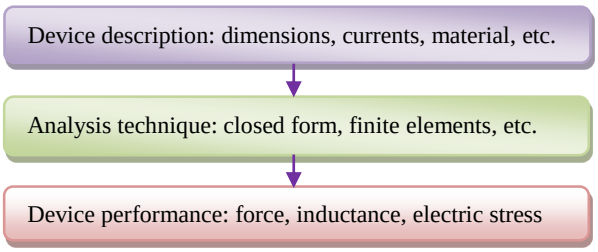

Fig. 1. The typical forward problem.

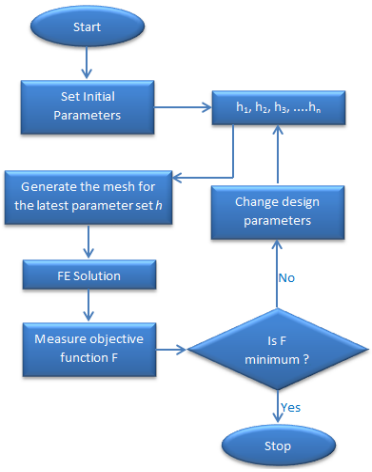

Fig. 2. The design cycle for inverse problem.

II. GPU COMPUTATION: MEMORY STUDY

In the CUDA programming model (see Fig. 3), a kernel is executed by a grid of thread blocks. A thread block is a batch of threads that can cooperate sharing data through shared memory and synchronizing their execution. Threads from different blocks operate independently.

Figure 4 shows the anatomy of the CUDA C/C++ program. Serial code executes on a CPU thread. Parallel code executes in many concurrent GPU threads across multiple parallel processing elements. The main limit with GPU computing is memory [7]. We worked with the 4 GB NVIDIA system, the best available till recently. Despite this limit, we have shown that for a single matrix equation, sizes up to 32768 x 37268 can be broached (for the first order triangular finite element magnetostatic and temperature field devices we were working with [10]) without running into the limit [1]. This is quite a large problem and that is why seminal papers on GPU computation for finite elements do not mention this limit [2]-simply because they did not run into the limit. In parallelized genetic algorithm based optimization in inverse problems however [5,6], several finite element solutions have to be performed simultaneously. Memory limits therefore are critical. In the following sections we examine these limits with a view to establishing the practicality of parallelizing finite element optimization on the GPU for coupled field problems where the memory load is doubled by the two-stage finite element problem and exploded when several two-stage kernels are launched on parallel GPU threads in genetic algorithm optimization, because gradient methods of optimization run into problems of mesh discontinuity and programming complexities in keeping track of shape changes [10].

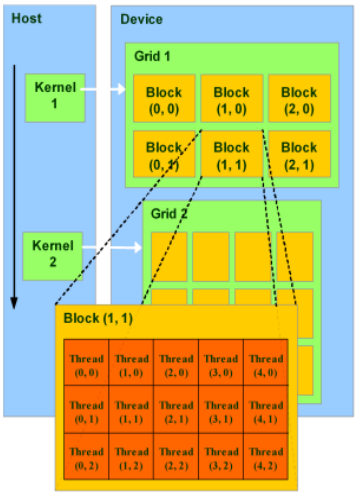

Fig. 3. The CUDA programming model.

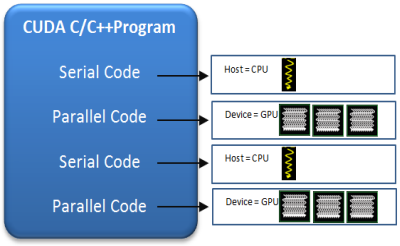

Fig. 4. Anatomy of the CUDA C/C++ program.

The test problem we finally take up is that of reshaping an originally square conductor which is heated by eddy currents (Fig. 5). The object is to have a constant temperature along a straight line. This paper being on parallelizing already forked kernels, the actual description of the geometry and analysis by first order triangular finite elements for the first stage problem from eddy current magnetics and the second stage by thermal analysis of the Poissonian temperature system (also with first order triangles), is left to references [10, 11]. In [11], the shape is optimized by gradient techniques, and [10] elaborates on the details of genetic algorithm optimization which are not taken up here but rather are left to [10].

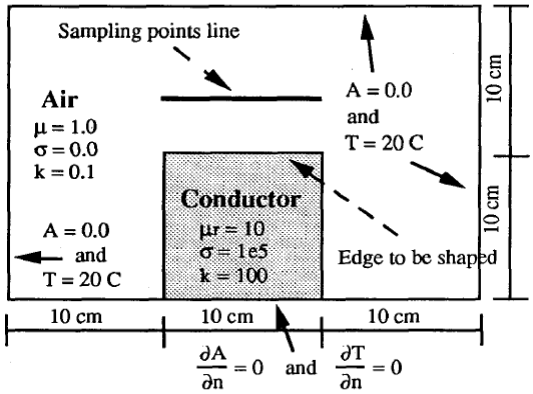

Fig. 5. The electro-thermal shape optimization by two-stage finite element analysis.

In this section, we investigate the standard sparse and profile matrix storage methods [3,4], in-order to reduce the matrix storage requirement and to use those storage scheme representations to get the solution.

We began this study looking at the largest single precision matrix sizes we can store on a single GPU. Besides full matrix storage and even symmetric matrix storage, which we do not consider because of the memory need running into order n^2 for an nxn matrix, we looked particularly at profile storage and sparse storage [3,4]. Our findings are shown in Table 1. Clearly, neither sparse storage nor profile storage runs even close to the 4 GB memory limit (superseded today by the 12 GB limit [7]) at the practically large matrix size of 10,000. However, they could if we were launching several threads, each with a matrix solution, as required with the GPU implementation of the genetic algorithm [5,6,8,10]. Therefore, we will confine ourselves to the sparse storage scheme, the better of the storage schemes as seen from Table 1.

Table 1: Storage demand with matrix size for different storage schemes

Matrix Size	Storage (MB)		
	Regular	Profile	Sparse
100	0.0400	0.0044	0.0065
400	0.0686	0.0413	0.0169
900	3.1070	0.1271	0.0363
1,600	9.7961	0.2870	0.0703
2,500	23.8895	0.5438	0.1137
6,000	137.4435	1.5428	0.2734
8,000	244.2932	2.6614	0.3594
10,000	381.66046	4.0821	0.4502

A further study was done to compare the performance of different methods with sparse storage. As seen from Table 1 for sparse storage, the memory requirement is approximately 0.45 MB, even for the unlikely large matrix size of 10,000 x 10,000. For the two-stage problem therefore, we still need only 0.45 MB since the eddy current and thermal problems are solved in sequence, because the thermal solution needs the thermal Specific Absorption Rate (SAR) from the eddy current solution. With 4 GB available, 8000+parallel threads are allowed, corresponding to a genetic algorithm population of 8000+. Now that 8 GB is available to us [7], memory we conclude is not an issue, except for very large problems or when full storage is used.

III. ELEMENT-BY-ELEMENT FINITE ELEMENTS

In the mid-1980s, the then new IBM PC 286 had a memory limit of 612 KB, which could not hold even a trivial matrix in memory. To overcome this, researchers used the Jacobi method of matrix solution (also known as Gauss-Seidel by power systems engineers) in a modified form [3,4]. Practically, the Element-by-Element Finite Element Method (EbEFEM) does not need a large amount of memory because it never stores or forms the global matrix except the diagonal. Generally, iterative algorithms such as the Jaccobi method, Conjugate Gradient method, etc., are used to get the solution of the problem [3]. During the 1980s, researchers could not represent big problems in

very limited memory so they used the EbEFEM method with an iterative method to represent very large problems [12]. Mahinthakumar and Hoole [13] used parallel implementation of the Jacobi conjugate gradients algorithm for field problems. In order to reduce the cost of memory, they used EbEFEM with the Jacobi Conjugate Gradients algorithm (JEBECG), which is very fast [13].

Figure 6 shows the sequential execution time against matrix size under for Incomplete Cholesky Conjugate Gradients (ICCG), Jacobi Conjugate Gradients (JCG) and Jacobi EbECG (JEbECG) on a SEQUENT SYMMETRY parallel computer for matrices from magnetic product design using first order traingles. Until matrix size 750, the ICCG method dominates; between 750 and 2500, JCG dominates rather than the other two methods; and for matrix size greater than 2500, JEbECG dominates.

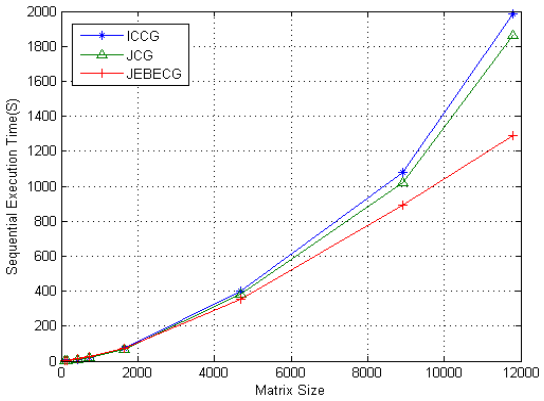

Fig. 6. Sequential execution times for ICCG, ICG, and JEBECG methods for matrix sizes.

Figure 7 for parallel implementation on the same shared memory machine using all its 4 processors shows similar findings. One processor does book-keeping, and with the 3 remaining processors working in parallel, the speedup is 2 or less (and not 3 because of communication bottlenecks). It is critical to note that more processors are not available for faster computation; nor to parallelize the genetic algorithm, and in one such genetic algorithm thread to parallelize matrix solution.

For matrix size under 500, ICCG dominates; for matrix size between 500 and 1800, JCG dominates; and for matrix size greater than 1800, JEbECG dominates. For simple problems, conjugate gradient schemes with sparsity computation or renumbering are suitable. It is not widely recognized that although renumbering: a) is necessary only for reducing storage in ICCG and Cholesky schemes of solution, and b) speeds up Cholesky by reducing fill-in, we are able to show that in ICCG it also unintentionally speeds up computations because the approximate Cholesky preconditioner gets to be more accurate [3]. For large problems, the element-by-element scheme is very profitable because it does not need matrix formation computation and storage capacity for the global matrix.

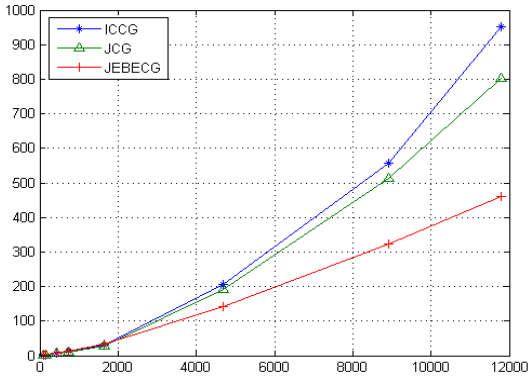

Fig. 7. Parallel execution times for ICCG, ICG, and JEBECG methods for matrix sizes.

IV. ELEMENT-BY-ELEMENT GAUSS-SEIDEL METHOD ON THE GPU

First ,we will describe the element-by-element scheme [8-13] which we are going to exploit for parallelizing already parallelized kernels. In solving the finite element matrix equation:

$$[P]\{\phi\} = \{Q\}, \quad (1)$$

far more powerful methods exist like the Incomplete Cholesky-preconditioned Conjugate Gradients algorithm (ICCG) than the older Gauss-Seidel iterations. The Gauss-Seidel iterations, commonly used by power engineers, are an improvement on the even older Gauss iterations. In Gauss-Seidel in each iteration m+1 we use the latest available values of the unknowns ϕ, using equation i of (1) to compute ϕ_i, treating only ϕ_i as the unknown and all the other variables as known and given by their latest values, some from the present iteration m+1 and the rest from the previous iteration m:

$$\phi_i^{m+1} = \frac{1}{P_{ii}}\left(Q_i - \sum_{k=1}^{i-1} P_{ik}\phi_k^{m+1} - \sum_{k=i+1}^{n} P_{ik}\phi_k^{m}\right), \quad (2)$$

with obvious modifications for i=1 and i=n. In this algorithm, ϕ_{i-1} must be computed before ϕ_i. Here at iteration m+1, computing ϕ_i in the order i=1 to n, ϕ is at values of iteration m+1 up to the (i-1)th component of $\{\phi\}$ and at the value of the previous iteration m for values after i. The original Gauss iterations (improved by Gauss-Seidel) uses the old iteration m's values for computing all ϕ_i in iteration m+1 according to:

$$\phi_i^{m+1} = \frac{1}{P_{ii}}\left(Q_i - \sum_{k=1}^{i-1} P_{ik}\phi_k^m - \sum_{k=i+1}^{n} P_{ik}\phi_k^m\right). \quad (3)$$

This is inefficient in the context of sequential computations. But in this case of parallelization, if we can resort to this conventionally inefficient method, we need not form the matrix [P]. If [D] is the matrix [P] with all off diagonal elements eliminated, then the Gauss iterations in this modified form gives:

$$[D]\{\phi\}^{m+1} = Q - [P - D]\{\phi\}^m. \quad (4)$$

Thus, without forming [P], the operations of the right hand side of (3) can be effected by taking each first order triangular finite element in turn, computing its local 3x3 Dirichlet matrix $[P]^L$ and using that because,

$$[P] = \sum_{elements}[P]^L. \quad (5)$$

So as each $[P]^L$ is formed, the three values of $\{\phi\}^m$ may be taken and subtracted as in the right hand side of (3) or (4) as justified by (5). Only the diagonal elements of [P] are stored so as to be able to divide by $P_{ii} = D_i$ as required in (3) and (4) quickly. Figure 8 shows the speedup of the element-by-element Gauss iterations. The speedup keeps increasing, seemingly endlessly, as matrix size goes up.

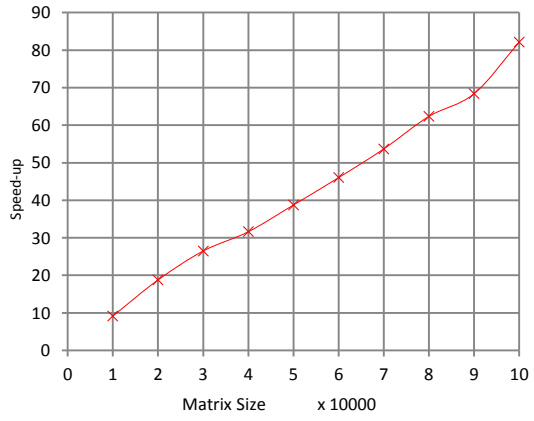

Fig. 8. Speedup for element-by-element Gauss iterations with matrix size.

For comparison we also parallelized on the GPU the ICCG algorithm with matrix storage-Incomplete Cholesky preconditioning requires [P]. From the results (Fig. 9), it is seen that the speedup is much lower than by element-by-element Gauss iterations, and saturates with matrix size because of the increased communications in forming and dealing with the matrix that is stored. But these figures are much faster than the speedup from 6 to 90-something reported by Kiss, *et al.* [7], presumably because of programming efficiency.

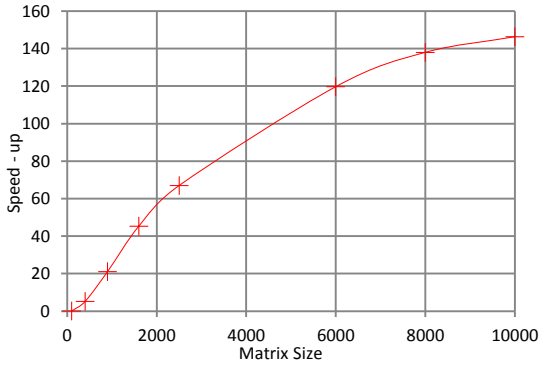

Fig. 9. Incomplete Cholesky conjugate gradients algorithm: matrix size vs. CPU time/GPU time.

V. NEW DEVELOPMENT IN CUDA

Thus far, parallelization in CUDA has not allowed parallelism within parallelism. Although it is allowed in multiprocessor machines, it was not very useful in finite element analysis, because shared memory machines with 4, 8, 16 or rarely 32 processors did not have spare processors to devote to parallel threads branching off from an already parallelized thread. (Supercomputers with more processors are not considered in this discussion because they are not readily available).

But CUDA 5.0 recently introduced support for forking into branches an already parallelized stream. This feature is a major breakthrough of the CUDA programming paradigm because CUDA allows many threads to be supported. This in turn allows a kernel to be launched and synchronized with new grids directly from the GPU using CUDA's standard<<< >>>syntax. A broad subset of the CUDA runtime API is now available on the device, allowing launch, synchronization, streams, events, and more. CUDA Dynamic Parallelism is available only on SM 3.5 architecture GPUs [14].

Since SM3.5 still has not come to PCs, we merely stick here to the single forking approach in determining what part of a kernel on a forked thread is to be further forked. In our work, we use the genetic algorithm where the object function corresponding to every member $\bar{h}$ of a population has to be computed many times to find the minimum. The many members $\bar{h}$ form the genetic search space. Since $\bar{h}$ consists of dimensions and materials of a particular design being examined for its goodness [15], for those dimensions a mesh is constructed, the finite element problem solved and the object function evaluated. The object function itself is computed from a finite element solution involving a matrix equation. Thus, we may treat the object function computation as a kernel and launch it on multiple threads, each for a different member of the population. Then, within that thread, as things are now on a PC, we can parallelize the matrix equation solution at a speedup of 147 and more by ICCG (Fig. 9) and even more by Gauss (Fig. 8). Alternatively, we may do the object function evaluation for each member of the population in sequence and in that process parallelize the matrix computations. Let the population number be n. Say the object function evaluation for each member of the population takes $t_0 + t_m$ in time where t_m is the time for the matrix solution and t_0 the time for other operations. Therefore, if we parallelize the operations for different members of the population, evaluating time for all object functions corresponding to the entire population would still be, neglecting coordination time,

$$t = t_o + t_m \,, \tag{6}$$

since these are done simultaneously. Here, we assume that the work for each member of the population being done in parallel, the time for combining results and other communications is negligible.

On the other hand, if we parallelized the matrix computation, the evaluation of the object function has to be in sequence since we cannot have forking from a parallelized kernel. The total time would then be the number of members in the population multiplied by the time for computing the object function for each member of the population:

$$t = n\left(t_0 + \frac{t_m}{147}\right). \tag{7}$$

Here, we have assumed that the speedup of 147 we have obtained for matrix solution by ICCG (Fig. 9) for matrix size upward of 10,000 would be achievable. A decision on which of the processes is to be parallelized would depend on considerations like this. However, we have not seen such considerations in the literature. On this basis, we found it better with a population size of 512 we were dealing with [10] to parallelize the population evaluation. The results are in Fig. 10, where the speedup saturates around 28 because of communication issues as the population rises.

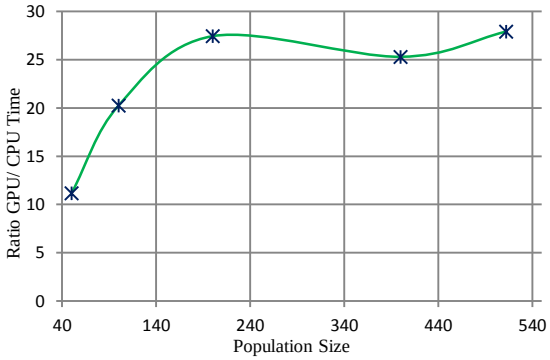

Fig. 10. Speedup: GA optimization GPU time/CPU time with population size.

VI. CONCLUSIONS

GPU parallelization is far superior to using multiprocessor machines because unlimited threads can launch computational kernels in parallel. While multiprocessor machines can fork a thread already running in parallel, they lack the processors that can be allocated. Although GPU cards till now did not allow a forked process to be further parallelized, this is being addressed by new architectures, such as the SM 3.5 architecture GPU [14]. For the vast majority of PCs with a GPU card but with dynamic parallelism not available, we have presented the methodology for deciding which one of the processes should be implemented in parallel to obtain the best speedup.

In GPU computing, the memory of the NVIDIA GPU is limited and this affects optimization work rather than the direct problem, because of the need to keep several copies of the matrix of coefficients in each genetic algorithm thread. The sparse storage scheme is the most efficient way to represent the matrix for finite element optimization. With it, only very large problems will find memory an obstacle, and for that class of problems, the element-by-element method

can be used.

If we use element-by-element FEM, practically unlimited size of problems can be solved without storing any matrix. GPU computation for finite element optimization by the genetic algorithm affords significant speedup. Element-by-element GPU matrix solution has even better speedup without saturating.

ACKNOWLEDGEMENTS

This work was funded in part by the US Army's Tank, Automotive Research, Development and Engineering Center (TARDEC), under contract number W911NF-11-D-0001. This paper has been approved by the US Army's Tank, Automotive Research, Development and Engineering Center (TARDEC) with the statement: "UNCLASSIFIED: Distribution Statement A. Approved for public release."

REFERENCES

[1] S. Sivasuthan, V. U. Karthik, and S. R. H. Hoole, "CUDA memory limitation in finite element optimization to reconstruct cracks," pp. 1967-1974 in D. E. Chimenti, L. J. Bond, and D. O. Thompson (eds.), *40th Annual Review of Progress in Quantitative Nondestructive Evaluation, AIP Conference Proceedings 1581, American Institute of Physics*, Melville, NY, 2014.

[2] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *IJNME*, vol. 85, no. 5, pp. 640-669, 2011.

[3] S. R. H. Hoole, "Computer aided analysis and design of electromagnetic devices," Elsevier, NY, 1989.

[4] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes, "Algorithm 586: ITPACK 2C: a FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods," *ACM Trans. Math. Software*, vol. 8, no. 3, pp. 302-322, 1982.

[5] M. L. Wong and T. T. Wong, "Implementation of parallel genetic algorithms on graphics processing units," pp. 197-216 in M. Gen, D. Green, O. Katai, B. McKay, A. Namatame, R. A. Sarkar, and B. T. Zhang (eds.), *Intelligent and Evolutionary Systems, Book Series: Studies in Computationl Intelligence*, vol. 187, Springer, 2009.

[6] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Genetic programming on graphics processing units," *Genetic Programming and Evolvable Machines*, vol. 10, no. 4, pp. 447-471, 2009.

[7] http://www.newegg.com/Product/Product.aspx?Item=N82E16814133494, Downloaded July 1, 2014.

[8] S. Sivasuthan, V. U. Karthik, A. Rahunanthan, P. Jayakumar, R. Thyagarajan, L. Udpa, and S. R. H. Hoole, "GPU computation: why element by element conjugate gradients?," *Sixteenth Biennial IEEE Conference on Electromagnetic Field Computation*, Annecy France, May 25-28, 2014.

[9] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, "Parallel realization of element-by-element FEM technique by CUDA," *IEEE Trans. Magnetics*, vol. 48, no. 2, pp. 507-510, 2012.

[10] V. U. Karthik, S. Sivasuthan, A. Rahunanthan, R. S. Thiyagarajan, P. Jeyakumar, L. Udpa, and S. R. H. Hoole, "Faster, more accurate parallelized inversion for shape optimization in electroheat problems on a graphics processing unit (GPU) with the real-coded genetic algorithm," *ECE Dept., Michigan State University, available from authors*, (also COMPEL-Paper under advanced stage of review).

[11] T. Pham, S. Ratnajeevan, and H. Hoole, "Unconstrained optimization of coupled magneto-thermal problems," *IEEE Trans. Magnetics*, vol. 31, no. 3, pp. 1988-1991, 1994.

[12] J. T. Hughes, I. Levit, and J. Winget, "An element-by-element solution algorithm for problems of structural and solid mechanics," *Comp. Meth. in App. Mech. & Eng*., vol. 36, no. 2, pp. 241-254, 1983.

[13] G. Mahinthakumar and S. R. H. Hoole, "A parallelized element by element jacobi conjugate gradients algorithm for field problems and a comparison with other schemes," *Int. J. App. Electromag. in Matl*., vol. 1, no. 1, pp. 15-28, 1990.

[14] http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html.

[15] E. R. Laithwaite, "The goodness of a machine," *Electron. & Power*, vol. 11, no. 3, pp. 101-103, 1965.

S. Ratnajeevan H. Hoole is a Professor of Electrical and Computer Engineering at Michigan State University. He has previously served as Member of the University Grants Commission of Sri Lanka where with six others he regulated the administration of all 15 Universities in that country. He also briefly was Vice Chancellor, University of Jaffna. A Fellow of the IEEE, he holds a Ph.D. degree from Carnegie Mellon University and a higher doctorate, the D.Sc. (Eng.) degree, from London. Besides Engineering, he has contributed much to the learned literature in the humanities and social sciences.

Sivamayam Sivasuthan was born in Sri Lanka, and received his B.Sc. degree First Class Hons. in Computer Science from the University of Jaffna. He has been reading for a doctoral degree in Electrical and Computer Engineering at Michigan State University since January 2012. His thesis focuses on exploiting graphics processing units to parallelize finite element optimization.

Victor U. Karthik was born in Sri Lanka, and received his B.Sc. Hons in Electrical and Electronics Engineering from the University of Peradeniya, Sri Lanka. He has been reading for a doctoral degree in Electrical and Computer Engineering at Michigan State University since January 2012. His thesis focuses on the genetic algorithm to optimize electro-heat problems for application in machine design and hyperthermia.

Arunasalam Rahunanthan has a Ph.D. degree in Mathematics from the University of Wyoming and a B.Sc. Engineering degree from the University of Peradeniya in Sri Lanka. He is currently an Instructor in the Department of Mathematics and Computer Science at the Edinboro University of Pennsylvania. His research interests include numerical methods for ODEs and PDEs, mathematical modeling of multiphase flows in multiscale porous media using Graphics Processing Units (GPUs) and Bayesian inference for quantifying uncertainty in problems related to subsurface flows.

Ravi Thyagarajan currently serves as Deputy Chief Scientist at the U.S. Army Tank Automotive Research, Development and Engineering Center (TARDEC), and was selected to the Researcher Review Board as a Senior Technical Specialist in June 2012.

His research pursuits are in the areas of underbody blast modeling and design, occupant protection and fast-running modeling methodologies. He received his Ph. D. degree in Applied Mechanics from Caltech, and has over 15 years of prior experience in the automotive industry at Ford and Visteon, where he was involved in all aspects of product development of automotive interiors. His automotive experience includes leadership roles in concept-to-launch product design, human factors/ergonomics development, as well as in the standardized application of CAE tools during the overall design engineering process, for which he won several awards, including the President's level Customer-Driven Quality award.

He is a past recipient of the Forest R McFarland Award from SAE, holds two patents and has co-authored over 40 technical papers. He received the Army Materiel Command (AMC) Systems Analysis awards in 2010 and 2012, for pioneering application of modeling and simulation methodologies in underbody blasts. He is also a Member of the Army Acquisition Corps, and is a Certified Acquisition Professional in Systems Engineering (SE), Program Management (PM) and Science & Technology (S&T) Management.

Paramsothy Jayakumar is a Senior Research Scientist, SAE Fellow, and a member of the Analytics Team at the U.S. Army Tank Automotive Research, Development, & Engineering Center (TARDEC) in Warren, Michigan. Prior to joining U.S. Army TARDEC, he worked for BAE Systems, Ford Motor Company, Altair Engineering, and Engineering Mechanics Research Corporation in the areas of multibody dynamics software development, vehicle dynamics modeling & simulation consulting, simulation technology development, durability load simulation, vehicle instrumentation & loads measurement, and road load engineering.

Jayakumar has written more than 100 journal and conference papers. His research in terramechanics and multibody dynamics won the best paper awards at the NDIA's Ground Vehicle Systems Engineering and Technology Symposium in 2011 and 2012. He holds a U.S. patent for a system for virtual prediction of road loads and tire modeling. He was also instrumental in developing seven SAE standards for tire testing for the purpose of tire modeling for which he received the SAE 2014 James M. Crawford Technical Standards Board Outstanding Achievement Award.

Jayakumar is a member of the U.S. Army Acquisition Corps, an Honorary Fellow of the Department of Mechanical Engineering at the University of Wisconsin-Madison, and an Associate Editor for the ASME Journal of Computational and Nonlinear Dynamics. He received his M.S. and Ph.D. degrees in Structural Dynamics from Caltech, and B.Sc. Eng. (Hons, First Class) from the University of Peradeniya, Sri Lanka.