GPU Based TLM Algorithms in CUDA and OpenCL

Filippo Rossi, Colter McQuay, and Poman So

Computational Electromagnetics Research Laboratory
Department of Electrical and Computer Engineering
University of Victoria, Victoria, BC, V8W 3P6, Canada

***Abstract*—** Recent advancements in graphics computing technology has brought highly parallel processing power to desktop computers. As multi-core multi-processor computing technology becomes mature, a new front in parallel computing technology based on graphics processing units has emerged. This paper reports a highly parallel symmetrical condensed node TLM procedure for the NVIDIA graphics processing units. The algorithm has been tested on three NVIDIA processors, from low-end laptop graphics card to high-end workstation graphics processors.

***Index Terms*—** TLM, FDTD, GPU, SIMD, time-domain, parallel computing, stream computing.

I. INTRODUCTION

Graphics processing unit (GPU) based parallel computing has been an important topic for the computing industry for over a decade. Macedonia addressed this topic in a computing magazine article in 2003 [1]. Most of the papers on GPU computing were related to signal and image processing [2–6]. Krakiwsky *et al.* and Inman *et al*. applied the technique to accelerate the FDTD algorithm [7, 8]. Takizawa *et al.* applied GPU computing to heat transfer simulation [9]. Z. Luo *et al.* and Harding *et al.* applied the paradigm to artificial neural network [10] and genetic algorithm [11], respectively. Furthermore, a cluster of GPU based computers can be created to execute grand challenge problems [12]. Researchers at Stanford [13] have been using this technique for years in protein folding computation.

Developing general purpose numerical modules for GPU was made easy by NVIDIA. The company released its Compute Unified Device Architecture (CUDA) Software Development Kit (SDK) in early 2007. The SDK enables programmers to develop GPU code in a high level language, *C-for-CUDA*. Rossi *et al*. reported the first implementations of a two- and a three-dimensional transmission line matrix (TLM) [14-17] program using the CUDA SDK [18]. This highly parallel TLM code has been ported to the new released OpenCL [19] environment. This makes it possible to run the program on non-NVIDIA GPUs and on heterogeneous computing hardware (for instance, GPU based computers with multiple multi-core CPUs). This paper addresses the algorithm design, programming techniques, and performance issues for implementing GPU based programs; in particular, the pros and cons of choosing CUDA and OpenCL will be discussed.

II. GPU COMPUTING

Modern GPU designs architectures are based on the Single Instruction Multiple Data (SIMD) computing paradigm. This hardware architecture utilizes multiple processors to perform similar tasks on vast quantities of data. The appeal for GPUs exists not only because of their computational ability, but also given that they are relatively inexpensive and can be installed on existing workstations. The NVIDIA GPUs used in this project are GeForce 8800 Ultra, Quadro FX 570M and Quadro FX5600 graphics cards [20]; these GPUs have 4 to 16 multi-processors with 8 processors each for a total of 32 to 128 processors. The GPUs have a maximum of 1.5 GB of GDDR3 global memory. A schematic that depicts the computing model of the NVIDIA GPU using a layer of a TLM mesh is shown in Fig. 1. The figure illustrates a typical iteration cycle. The data structure to be processed (called a mesh) is defined in both the CPU and the GPU. After seeding a data structure with initial conditions, the host transfers the data to the GPU's global memory and constant memory. A GPU function (called a kernel) would then be invoked which would execute on all multi-processors (4 to 16). This computing paradigm is scalable by utilizing GPU clusters internal or even external to a workstation [21]. Adaptation to GPUs is suitable for many science and engineering applications. However, the parallelization of existing algorithms may require intricate and complex adaption efforts.

The driving forces behind the computing framework depicted in Fig. 1 are the thread-blocks that control the GPU executions, Fig. 2. A thread block is defined as a grouping of threads that executes concurrently on the GPU multi-processors. Multiple data elements could be

assigned per thread (data block). A maximum of 512 threads per thread block are available for the GPUs, therefore it is then necessary to partition the mesh into data blocks. Each multi-processor would, in turn, execute on a data block utilizing its thread-block and then transfer the results back to the global memory. After which the multiprocessor would download the next available data block. The 16 multi-processors on the graphics card used in this project worked concurrently and are coordinated to process all data blocks in the global memory.

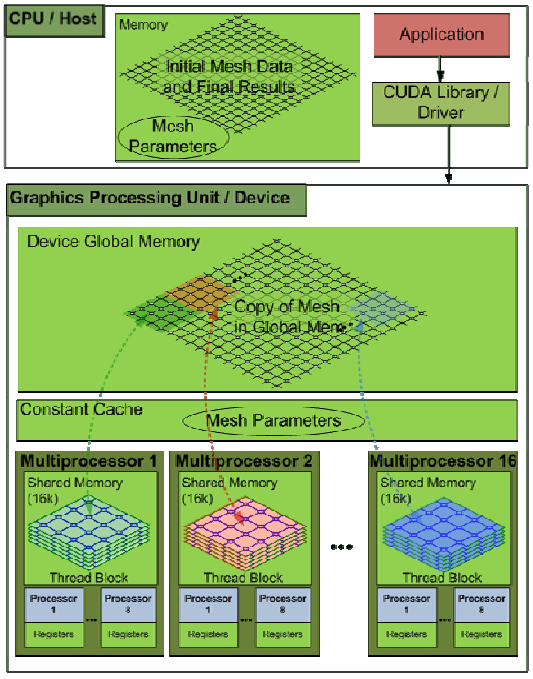


Fig. 1. NVIDIA CUDA based GPU computing framework.

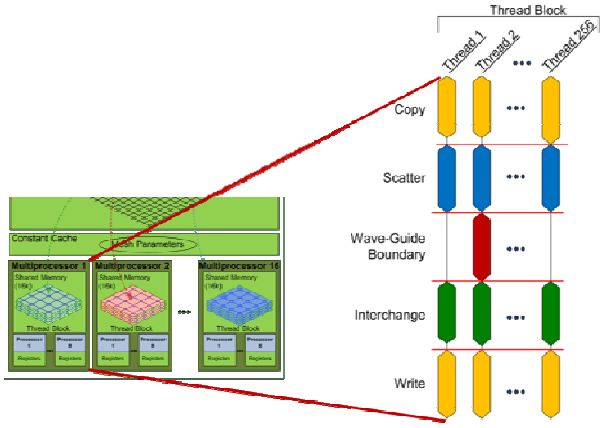
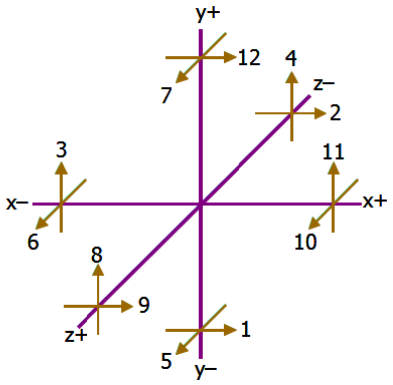


Fig. 2. NVIDIA CUDA based GPU computing framework.

III. SYMMETRIC CONDENSED NODE TLM

The three-dimensional symmetrical condensed node (SCN) is depicted in Fig. 3. The fundamental procedures in a TLM algorithm are the scattering, transfer and reflection of voltage impulses, [17]. There are two orthogonal transmission link lines in each port of the TLM node. Voltage impulses travelling along the *x*-axis are polarized in the *y*- or *z*-direction; similarly voltage impulses on the *y*- and *z*-axis are polarized in the other two orthogonal directions on the plane transverse to the direction of propagation. Hence, there are a total of 12 voltage impulses in each symmetrical condensed node. The scattering matrix for the symmetrical condensed node TLM method is a 12×12 matrix [17]. Therefore, a matrix multiplication operation can be used to obtain the reflected voltage vector from the incident voltage vector. The other two TLM operations are transfer of impulses to the neighboring link lines and reflection of impulses at material boundaries, Fig. 4. These two operations are applied to the two orthogonally polarized voltage impulses. All three TLM operations — scattering, transfer and reflection of voltage impulses — are localized operations which may be executed in parallel to reduce computing time. A quad-core processor may execute each operation concurrently for four TLM



$$[V]_{k+1}^{r} = [S] \times [V]_{k}^{i}$$

$$[S] = \begin{bmatrix}
a & a & & & & & a & -a & & & & \\
a & & & & a & & & -a & a & & & \\
a & & a & & & a & & & -a & & & \\
 & a & & a & & -a & & & & a & & \\
 & & a & & a & -a & a & & & & & \\
 & a & & a & & a & & -a & & & & \\
 & & -a & & a & & a & & a & & & \\
 & a & & -a & & a & & & & a & & \\
a & & & & -a & & & & a & & a & \\
 & -a & & & & a & & a & & & & \\
-a & & a & & & & a & & & & & a \\
 & a & -a & & & & & & a & & a &
\end{bmatrix}$$

Fig. 3. SCN scattering algorithm.

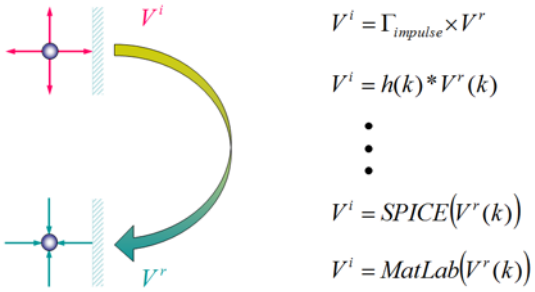


Fig. 4. TLM boundary operation.

nodes. A traditional serial TLM program can be easily parallelized by using OpenMP [22] compiler directives. However, the number of cores on a single CPU is small and the gain in performance by using OpenMP is therefore still limited. With GPUs, numerical procedures such those described above can be executed in parallel on a much larger scale.

IV. IMPLEMENTATION

Efficient use of multiprocessor resources, especially global memory transfer strategies, can help to achieve close to the maximum theoretical operating speeds of GPUs. Memory transfer rates between the global memory and multiprocessors can be used as a benchmark for GPU performance since much of the kernel execution time (70% to 80%) may be spent in accessing global memory. In the case of the Quadro FX 5600, the maximum theoretical memory bandwidth to global memory is 76.8 GB/sec [21] or expressed as read+write round trip: 38.4 GB/sec.

Memory coalescing is a performance enhancement technique whereby access to global memory by multiprocessors can be accelerated [20]. Global memory (GDDR3 memory) consists of physical banks of memory. Access to global memory by any of the multiprocessors results in 400-600 clock cycles of latency. In other words, each four byte float or integer copied from or written to global memory takes 400-600 clock cycles. Since the GDDR3 global memory exists physically as banks of memory, reads/writes can be organized such that [20]:

1. The starting address of each half-warp (16 threads) falls on a 64 byte interval
2. Each thread of a half-warp reads/writes 4, 8 or 16 bytes consecutively
3. The threads of each half-warp must be spaced at 4, 8 or 16 byte intervals.

Figure 5 illustrates the differences in memory access speed (GB/sec read-write round trip) between coalesced code (~25 GB/sec) and non-coalesced code (~3 GB/sec). A speed-up of over 8 times for coalesced kernel code warranted developing TLM kernel that adhered to coalescing coding strategies.

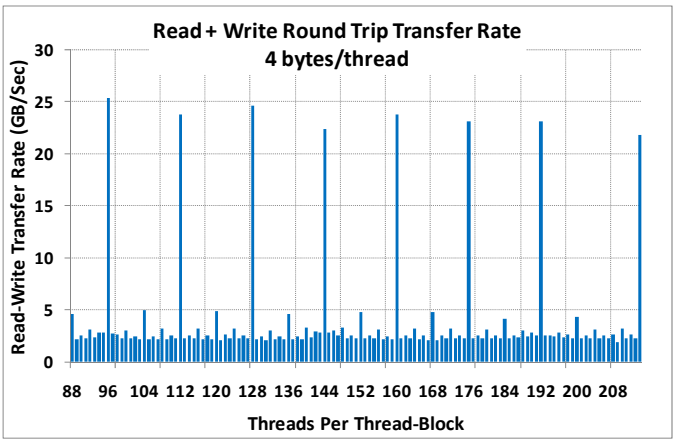


Fig. 5. GPU performance differences between coalesced a non-coalesced memory configurations.

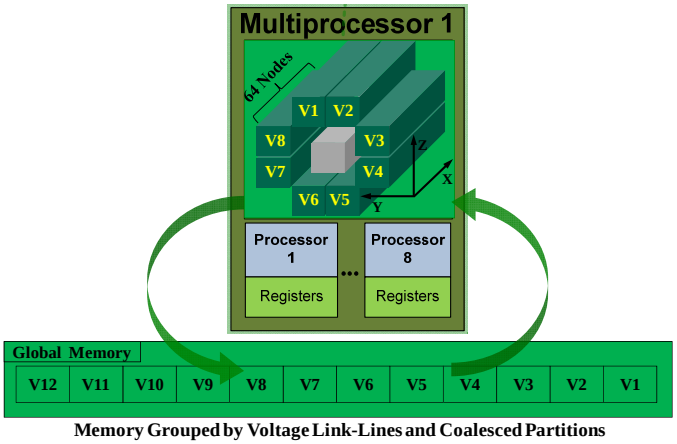


Fig. 6. Coalesced global memory access by thread-blocks of multiprocessors.

The TLM kernel is designed such that global memory is grouped by voltage link lines (12 per TLM node), and accessed by the multiprocessors in a coalesced manner to take maximum advantage of the GPUs speed performance, Fig. 6. The resolution of the Y and Z dimensions of a mesh is each one node wide. However, the X dimension is partitioned into 64 node segments. The addressing is thus contiguous, first in the x-direction, then the y-direction and finally the z-direction. The thread-block dimension is defined at 64 threads, where the kernel would read a voltage link line for 64 nodes at a time in the x-direction. For example, 64 values of V1 would be read for 64 nodes, then for V2 would be read for the same 64 nodes and so on until all 12 voltagess have been read so that the scattering

procedure may commence on 64 nodes. Writing results back to global memory is done in a similar manner. Fig 7 showcases the performance enhancement of configuring the TLM kernel in a coalesced fashion.

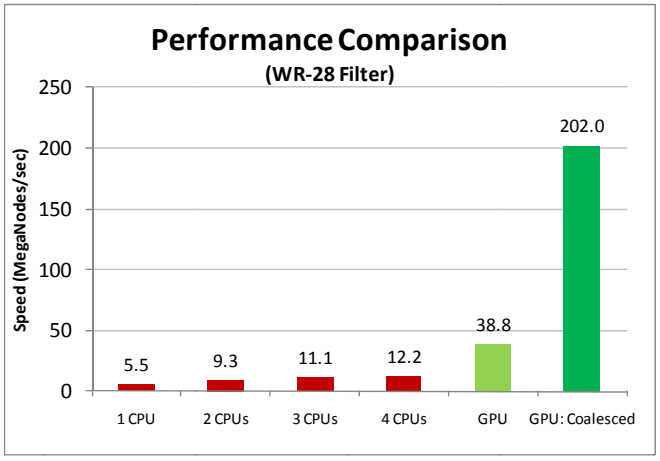


Fig. 7. TLM codes performance on 1 to 4 CPUs. The codes were used to analyze a WR-28 filter (150,000 nodes). The GPU codes have a non-coalesced GPU kernel and a coalesced GPU Kernel.

V. OPENCL IMPLEMENTATION

The newly released OpenCL application programming interface (API) has many features found in the CUDA paradigm. In fact, once a program has been developed using the CUDA framework, it is not difficult to implement an equivalent version in OpenCL. This is especially true if the NVIDIA's "Driver API" method is used instead of "C for CUDA" [19]. Both OpenCL and the Driver API standards are handle-based hence objects to memory and functions are required. However, OpenCL differs in its ability to utilize a heterogeneous mixture of parallel hardware. An OpenCL application can utilize a mixture of GPUs, CPUs, and other processors. Therefore OpenCL is a powerful API for integrating multiple parallel platforms under one cohesive programming paradigm.

Figure 8 depicts some equivalent programming constructs in the two APIs. As shown in the figure most of the equivalent constructs have meaningful mapping from CUDA to OpenCL; the only exception in the table is the "Local Memory" construct.

In addition to systemically replacing the programming constructs, it is also necessary to substitute CUDA kernel code attribute modifiers with equivalent OpenCL methods, Fig. 9. A short code segment that illustrates the code transformation concept is shown in Fig. 10. When compared to CUDA, OpenCL is a relatively new API hence it is less efficient than CUDA especially when the problem size is small. Figure 11 compares the performance of our CUDA and OpenCL TLM code. When the structure size reaches about 10 million nodes, the two programs have similar computation speed.

CUDA	OpenCL
Thread	Work-item
Thread Block	Work Group
Shared Memory	Local Memory
Local Memory	Private Memory

Fig. 8. Equivalent programming constructs in CUDA and OpenCL.

CUDA Kernel Code	OpenCL Kernel Code Replacement
gridDim	**get_num_groups(*n*)**
blockDim	**get_local_size(*n*)**
blockIdx	**get_group_id(*n*)**
threadIdx	**get_local_id(*n*)**
__global__ function (callable from host, not callable from device)	**__kernel** function (callable from host or device)
__device__ function (not callable from host)	
__constant__ variable declaration	**__constant** variable declaration
__device__ variable declaration	**__global** variable declaration
__shared__ variable declaration	**__local** variable declaration
__syncthreads()	**barrier()**

Fig. 9. Attribute transformation table.

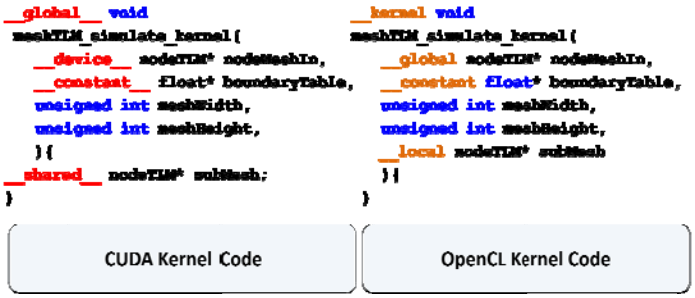


Fig. 10. Kernel code transformation.

VI. CONCLUSION

We have successfully designed and implemented a highly parallel SCN TLM algorithm for the CUDA/OpenCL enabled NVIDIA GPU. It is found that both CUDA and OpenCL are good programming APIs for implementing computational intensive applications such as TLM on GPU based hardware. Our latest CUDA implementation of the 3D SCN TLM routine has achieved a 293 MNodes/sec performance of an empty

structure and 288 MNodes/sec with the filter implemented. Ongoing research activities are focusing on improving the speed of execution and adapting the

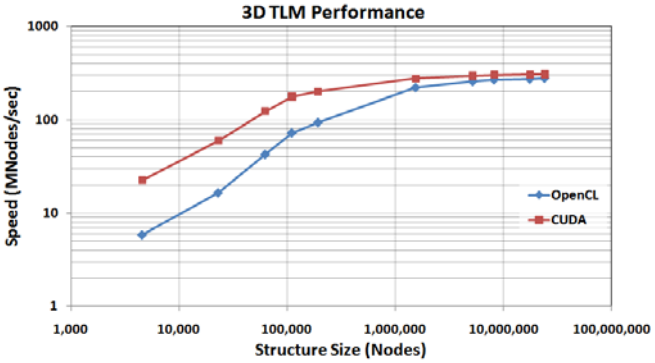


Fig. 11. Performance comparison — CUDA versus OpenCL.

algorithms to solve various structures and configurations. An investigation in utilizing a cluster of 4 NVIDIA GPUs on an Acceleware ClusterInABox™ Quad Q30 workstation is being conducted. The implementations described above can be modified to handle the generalized symmetrical condensed node (GSCN) TLM algorithm developed by Trenkic *et al.* [23, 24]. The total number of voltage impulses to be stored per node would thus increases from 12 to 18. This would reduce the number of nodes each multi-processor thread-block can handle. However the GSCN scattering procedure would not cause any significant reduction on the overall performance as the bottleneck is in the data transfer, not in number of floating point operations. Hence, the performance results depicted in figures 8 and 12 are still valid but the code would reach the maximum acceleration at a smaller structure size.

ACKNOWLEDGMENT

The authors wish to acknowledge the financial supports from the Canada Foundation for Innovation (CFI) and the Natural Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] M. Macedonia, "The GPU Enters Computing's Mainstream", *IEEE Computer*, vol. 36, no. 10, pp. 106–108, October 2003.

[2] G. Shen, G. P. Gao, S. Li, H. Y. Shum and Y. Q. Zhang, "Accelerating Video Decoding Using GPU", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693, May 2005.

[3] J. Y. Hong and M. D. Wang, "High speed processing of biomedical images using programmable GPU", *International Conference on Image Processing*, vol. 4, pp. 2455–2458, October 2004.

[4] Y. Heng and L. Gu, "GPU-based Volume Rendering for Medical Image Visualization", *27th Annual International Conference on Engineering in Medicine and Biology*, pp. 5145–5148, 2005.

[5] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU", *IEEE Proceedings of the Information Visualization*, pp. 609–614, July 2006.

[6] J. S. Meredith, S. R. Alam and J. S. Vetter, "Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures", *IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, March 2007.

[7] S. E. Krakiwsky, L. E. Turner and M. M. Okoniewski, "Graphics Processor Unit Acceleration of Finite-Difference Time-Domain Algorithm", *Proceedings of IEEE International Symposium on Circuits and Systems*, vol. 5, pp. V265 – V268, May 2004.

[8] M. J. Inman, and A. Z. Elsherbeni, "Programming video cards for computational electromagnetics applications", *IEEE Antennas and Propagation Magazine*, vol. 47, no. 6, pp. 71–78, December 2005.

[9] H. Takizawa, N. Yamada, S. Sakai, and H. Kobayashi, "Radiative Heat Transfer Simulation Using Programmable Graphics Hardware", *5th IEEE/ACIS International Conference on Computer and Information Science*, pp. 29–37, July 2006.

[10] Z. Luo, H. Liu, and X. Wu, "Artificial Neural Network Computation on Graphic Process Unit", *Proceedings of IEEE International Joint Conference on Neural Networks*, vol. 1, pp. 622–626, August 2005.

[11] S. Harding, W. Banzhaf, "Fast Genetic Programming and Artificial Developmental Systems on GPUs", *21st International Symposium on High Performance Computing Systems and Applications*, p. 2, May 2007.

[12] F. Zhe, Q. Feng, A. Kaufman and S. Yoakum-Stover, "GPU Cluster for High Performance Computing", *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 47, 2004.

[13] folding.stanford.edu/FAQ-ATI.html

[14] F. V. Rossi, "Massively Parallel Two-Dimensional TLM Algorithm on Graphics Processing Units," *IEEE International Microwave Symposium*, June 2008.

[15] F. Rossi and P. P. M. So, "Parallelized three-dimensional TLM algorithms on a graphics processing unit", *25th International Review of Progress in Applied Computational Electromagnetics Symposium*, pp. 110–114, March 2009.

[16] W. J. R. Hoefer, "The Transmission-Line Matrix Method – Theory and Applications", *IEEE Transactions on Microwave Theory and Techniques*, vol. MTT-33. No. 10, pp.882-893, October 1995.

[17] P. B. Johns, "A symmetrical condensed node for the TLM method," *IEEE Transactions on Microwave Theory and Technique*, vol-35, no. 4, pp. 370–377, April 1987.

[18] http://www.nvidia.com/object/cuda_home_new.html, April 2010.

[19] http://www.khronos.org/opencl/

[20] http://www.nvidia.com

[21] ClusterInABox Quad (Q30) Product Info, http://www.acceleware.com/default/index.cfm/our-products/clusterinabox-quad, November 2008.

[22] http://OpenMP.org/wp/

[23] V. Trenkic, C. Christopoulos, and T. M. Benson, "Development of a general symmetrical condensed node for the TLM method", *IEEE Trans. on Microwave Theory and Techniques*, vol. MTT-44, no. 12, pp. 2129–2135, December 1996.

[24] V. Trenkic, C. Christopoulos, and T. M. Benson, "Advanced node formulations in TLM — the adaptable symmetrical condensed node", *IEEE Trans. on Microwave Theory and Techniques*, vol. MTT-44, no. 12, pp. 2473–2478, December 1996.

Filippo Rossi received the B.Eng. degree in Electrical Engineering in 2008 from the University of Victoria, Victoria, British Columbia, Canada. Currently he is completing a Master of Applied Science at the University of Victoria. He is working at the Computational Electromagnetics Research Laboratory (CERL) at the University of Victoria in GPU computing, as well as working with the Millimeter Instrumentation team at the Herzberg Institute of Astrophysics, Victoria, B.C., Canada.

Colter McQuay is an undergraduate student at the University of Victoria, B.C. in Electrical Engineering with a specialization in Signal Processing and Computer Music. Colter was born in Kamloops B.C. in 1987. In 2009, his research focused on implementing TLM algorithms on GPU hardware using OpenCL , presenting a paper at the USRI Conference in Boulder Colorado in Jan 2010. Currently Colter is involved in writing an open source electromagnetic simulation application using the code developed in previous research.

Poman So is an Assistant Professor at the University of Victoria. He received the B.Sc. degree in Computer Science and Physics from the University of Toronto, Toronto, Ontario, Canada, in 1985; the B.A.Sc. and M.A.Sc. degrees in Electrical Engineering from the University of Ottawa, Ottawa, Ontario, Canada, in 1985 and 1987, respectively; and the Ph.D. degree in Electrical Engineering from the University of Victoria, Victoria, BC, Canada, in 1996.

Dr. So possesses twenty years of hands-on object-oriented software engineering experience in time-domain computational electromagnetics. He developed a number of electromagnetic wave simulators based on the Transmission Line Matrix (TLM) method. Dr. So is a co-founder of the Faustus Scientific Corporation and is the creator and chief software architect of MEFiSTo, a general purpose time-domain electromagnetic field solver based on the Transmission Line Matrix method. From July 1998 to June 2005, Dr. So was the Principal Software Engineer at Faustus Scientific Corporation. In July 2005, He joined the Department of Electrical Engineering at the University of Victoria. His research interests include object-oriented computational electromagnetics, graphics processing unit (GPU) based massively parallel TLM algorithms, time domain modeling of advanced electromagnetic structures, and modeling of bio-electromagnetic systems.

Dr. So is a Registered Professional Engineer in the Province of British Columbia, Canada. He is a senior member of The Institute of Electrical and Electronics Engineers (IEEE), a member of Applied Computational Electromagnetics Society (ACES), and a member of the Canadian Medical and Biological Engineering Society (CMBES). He has published over 100 refereed journal and conference papers. Dr. So serves regularly a reviewer for the IEEE Transactions on Microwave Theory and Techniques, the IEEE Microwave and Wireless Components Letters, the Applied Computational Electromagnetics Society Journal, International Journal of RF and Microwave Computer Aided Engineering, the International Journal of Numerical Modeling – Electronic Networks, Devices and Fields by John Wiley and Sons Ltd. He is a member of the Editorial Advisory Board for the International Journal of Numerical Modeling – Electronic Networks, Devices and Fields by John Wiley and Sons.