HPC for Legacy EM Code, a Mixed Language Approach using CUDA

Srijith Rajamohan[1], **Kyle Anderson**[2]

Simcenter, Department of Computational Engineering
University of Tennessee at Chattanooga, TN 37405, USA
tny954@mocs.utc.edu[1], kyle-anderson@utc.edu[2]

Abstract: Integrating CUDA into an existing Fortran framework is investigated and its benefits and pitfalls are discussed. The suitability of the Graphics Processing Unit (GPU) architecture to an unstructured finite element EM solver is discussed. Test cases are run with various kernel parameters on an initial coarse mesh and their resulting runtimes are compared with the corresponding Fortran subroutines to examine the acceleration benefits attained by using CUDA. It is then demonstrated using a larger mesh that the algorithm is scalable, within the limitations of the GPU's memory.

Keywords: GPU, Computational Electromagnetics, HPC, Fortran, CUDA

1. Introduction

GPU Computing is defined as the use of Graphics Processing Units to perform scientific and engineering work, which by definition implies a hybrid computing model consisting of a GPU and a CPU. The sequential portion of the application executes on the CPU, which is mostly optimized for non computational tasks such as branching [10] while the computationally intensive sections are dispatched to the GPU. In the late 1990's scientists began to adopt Graphics Processing Units (GPUs) for scientific applications, with particular emphasis in the fields of electromagnetics and medical imaging. However, because GPUs had initially been designed as dedicated fixed function graphics pipelines, doing so necessitated mapping the scientific applications to resemble pixel shaders [11].

With support for GPU programming provided by high-level Application Programming Interfaces (APIs) such as CUDA, it has became possible to use high level languages such as C, C++ and Fortran for this purpose. CUDA allows the application developer to write device code in C functions known as kernels. A kernel differs from a regular function in that it is executed by many GPU threads in a Single-Instruction Multiple Data (SIMD) fashion. Several mainstream applications such as Adobe Photoshop, Mathworks Matlab, Wolfram Mathematica, and NI Labview routinely exploit the advantages of the GPU's computing capabilities.

While variations of C and Fortran are commonly used for scientific applications, many of the codes in use today have been developed over a period of many years and are not structured appropriately to benefit from GPU programming. Among these codes, there is an abundance of scientific legacy code written in Fortran [1], which continues to be the language of choice for high performance scientific applications [13]. As mentioned previously, with the evolution of cost effective hardware accelerators [11] it has become easier to pursue increasingly complex applications without resorting to supercomputing resources. CUDA is one of the more popular approaches, although active development is also being pursued using OpenCL [2], DirectCompute [3], X10 [4], and FPGAs [15]. This last option deserves special mention as it is a hardware solution and although it can be quite flexible, its use often results in greater-than-average turn-around time [11].

For the current work, CUDA has been chosen because of its widespread acceptance, ready availability, and the fact that its use and performance is well documented in the open literature. Currently the only CUDA version provided by Nvidia requires a kernel written in C/C++ (although there are other commercial versions available). Therefore, one of two options must be chosen for using CUDA with Fortran codes. In the first option, the entire code is ported to C/C++ and then accelerated using

CUDA. A more efficient development path is to profile the code and determine processor-intensive routines that are then ported to a CUDA kernel in C/C++ for acceleration. In this work the second approach has been adopted because porting the entire code is not always pragmatic or even desirable [13].

2. Finite Element Discretization

The time dependent Maxwell's curl equations [5] are given by

$$\nabla \times E = -\frac{\partial B}{\partial t} \tag{1}$$

$$\nabla \times H = \frac{\partial D}{\partial t} + J \tag{2}$$

$$\nabla \cdot B = 0 \tag{3}$$

$$\nabla \cdot D = \rho_c \tag{4}$$

where E and H are electric and magnetic field intensities whereas D and B are electric and magnetic flux densities which will be the fundamental variables in this formulation. This can be written in a divergence form given by

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = 0 \tag{5}$$

where Q is the list of electric (D) and magnetic (B) flux densities

$$Q = (D_x, D_y, D_z, B_x, B_y, B_z) \tag{6}$$

The solution to the governing equations is obtained using a Petrov Galerkin scheme, which is a weighted residual method and it can be written in the following form

$$\iiint_{\Omega} [\phi] \left(\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) \right) d\Omega = 0 \tag{7}$$

where ϕ is a weight function given as

$$[\phi] = N[I] + \left(\frac{\partial N}{\partial x}[A] + \frac{\partial N}{\partial y}[B] + \frac{\partial N}{\partial z}[C] \right) [\tau] \tag{8}$$

where

$$N = \sum_{i}^{n} N_i c_i \tag{9}$$

Here N_i represents a basis function, c_i is an arbitrary constant, and [A], [B], [C] are flux Jacobian matrices. The τ matrix is given as

$$[\tau]^{-1} = \sum_{k=1}^{n} \left| \frac{\partial N}{\partial x}[A] + \frac{\partial N}{\partial y}[B] + \frac{\partial N}{\partial z}[C] \right| \tag{10}$$

3. Implementation

An EM solver presents ample opportunities for parallelization as the mesh elements can be spatially partitioned and allocated to the compute resources. Because a finite element code tends to be quite cumbersome compared to its alternative, the FDTD method, there exists opportunities and hurdles for a CUDA implementation. However adapting an unstructured solver to take advantage of the GPU architecture can be challenging because no predetermined memory access pattern exists for the nodes that make up the unstructured mesh. In accordance with Amdahl's law the code is profiled to determine the most time intensive subroutines that can benefit from acceleration. As a result, the residual calculation has been determined to dominate the solver runtime implying that high priority is placed on porting these calculations to the GPU. These routines have consequently been rewritten in C to execute as a kernel on the GPU. Because the majority of the code is written in Fortran, wrapper code has been developed to provide the interface with the CUDA kernel. This involves an additional data transfer, copying from Fortran code to C wrapper and from there to the kernel. Compatible data structures are created in C for the purpose of message passing from Fortran. Every effort has been made to maintain flexibility in the code and make it as generic as possible.

Mesh associated parameters that are read only are stored in shared memory, however the residuals are stored in global memory. Access conflicts for a node residual can result in delays that can severely affect performance. There are two ways to deal with this problem. One option involves renumbering the nodes to minimize conflicts between threads, although this still does not eliminate contention at nodes shared by different threads. This method uses atomic operations to ensure thread safe operation, but it is an operation that can severely affect performance. In the second method, memory locations for the residual are allocated for the nodes associated with each mesh element assigned to a thread, regardless of whether or not the same node is accessed in another thread. Each thread writes it's residual contribution to a separate memory location associated with a node and the final residual at each node is accumulated using a reduction operator across the threads. This comes at the price of increased memory usage, however all memory accesses are uncontested. As a result peak performance can be obtained as long as there is sufficient memory in the GPU global memory to hold the duplicated residual contributions. The second approach is chosen in this work to minimize access delays.

The residual computation for this kernel is distributed approximately equally over the blocks to maximize the occupancy of each block. It must be pointed out that determining the ideal load conditions is not a trivial task as this usually involves finding the right balance between the number of blocks, threads and nodes per thread. Although there are hard limits on most of these parameters, identifying the optimal set of parameters is often dependent on the solver. This is a result of the fact that each application tends to have differing memory access patterns, memory usage, processing needs etc. It is noted that, for this study both the baseline Fortran code and CUDA enabled version use single precision arithmetic.

In the flowchart in Fig.1 the boxes in yellow are code sections that are relevant to the porting. Before the beginning of the iteration data related to the mesh is copied from the main memory space to the GPU memory space. In this implementation only the residual routines are ported onto the GPU. The green boxes indicate Fortran code sections or routines whose function is to pack data into data structures that can be passed to C wrapper code. The blue boxes are C wrapper code routines that receives packed data from Fortran. The yellow boxes represent CUDA kernels that are invoked from the C wrapper. The boxes in orange can be implemented in a similar fashion, however this is left for future work.

CUDA needs to be initialized with a call from the Fortran solver code to a dummy C function, since the first CUDA invocation incurs a startup initialization overhead. This overhead is proportional to the size of the data allocated, hence a large memory allocation can slow down code execution. From the Fortran code, in the driver routine the function 'initcuda' is called to initialize CUDA. This subroutine makes a single memory allocation for a single integer, which also implicitly initializes the GPU.

The solver parameters are copied from the global memory to shared memory variables. Global memory has a latency of about 400 to 800 cycles [9], depending on the instruction, while the shared

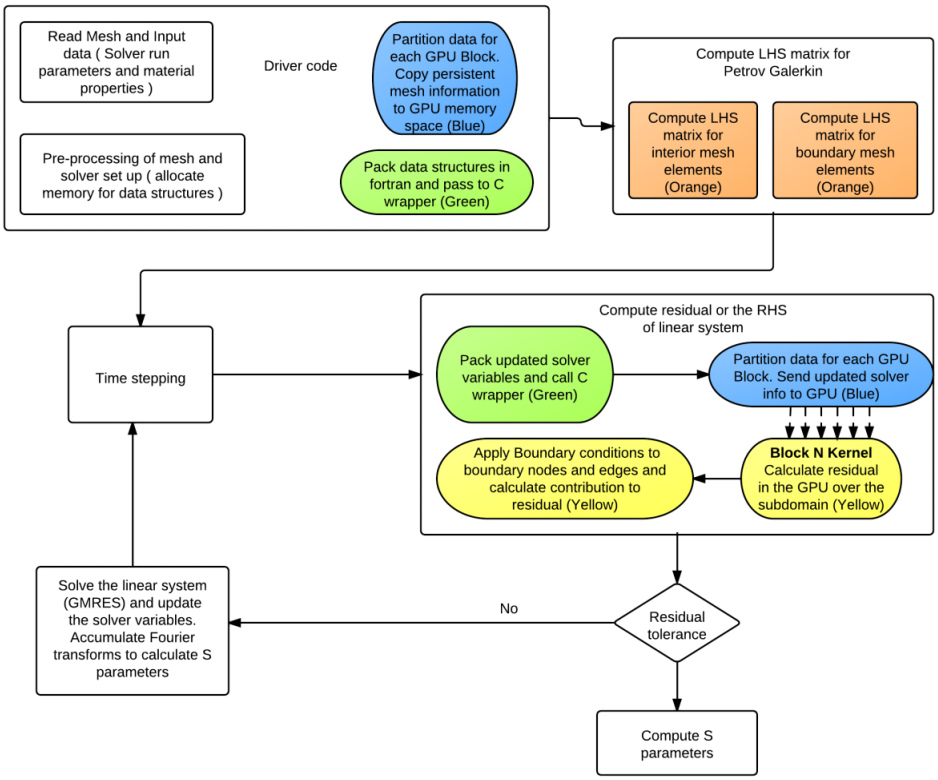

Figure 1: Proposed algorithm.

memory is on-chip and has a latency of about 20 cycles. Considering that these parameters will be repeatedly accessed in an iteration loop, significant performance improvements can be achieved by doing so. To maintain data integrity and to provide synchronization only the first thread makes the data copy. Since this resides in shared memory space, all the threads within the block have access to these variables. A "syncthread" command is then issued to ensure data integrity, since the order in which the threads are spawned is indeterminate and threads within a block have no other synchronization method. The code snippet below displays how this is accomplished.

```
__shared__ iter_info_common info_device;
    ......

 if(threadIdx.x == 0)
 {
 info_device.stepNo = info->stepNo;
 info_device.timeAccuracy = info->timeAccuracy;
 info_device.irestart = info->irestart;
 info_device.ngauss = info->ngauss;
 info_device.elementType = info->elementType;
 info_device.nq = info->nq;
 condition1 = (info->stepNo >= 3 && info->timeAccuracy == 2) || ( info->irestart == 1 && ↩
     info->timeAccuracy == 2);
 }
 __syncthreads();
    ......
    ......
```

4. Results

The implementation has been tested on a single GTX470 desktop graphics card from Nvidia using a CUDA compute capability version 2.0 and a CUDA runtime driver version 3.20. The results obtained using the GPU are compared to those obtained using the original code and the GPU results are found to be of sufficient accuracy. The graphics card has 14 multiprocessors(clocked at 1.22Ghz), each comprising 32 cores for a total of 448 cores. Wall clock times are recorded during each run for both the original Fortran version and for the code ported to the GPU.

The figure below compares the execution on Fortran with a GPU implementation using a 2930 node tetrahedral mesh.

Total threads	Blocks	Threads per block	Mesh per element	Kernel execution time	Fortran execution time
800	25	32	140	0.679496	3.249844
1600	25	64	70	0.383718	3.250823
1600	50	32	70	0.617468	3.269889
3200	50	64	35	0.620021	3.256026
3200	25	128	35	0.404993	3.300707
2560	20	128	40	0.279704	3.255644
5120	20	256	20	0.301782	3.250507
6400	50	128	18	0.617136	3.25056

Figure 2: Runtime statistics of Fortran code vs CUDA kernel.

It can be seen from the figure that load balancing is key to performance on the GPU. In the Fermi architecture each Streaming Processor can have up to 8 active blocks and 48 active warps (or 1536 active threads). The maximum number of threads per block is 1024 and the warp size is 32. Ideally at least 128 threads are recommended for optimum performance, however increasing the number of threads per block decreases the number of mesh elements proportionately; as a result the blocks are not fully utilized. Blocks have an associated context-switch cost as a result of having to save registers and shared memory, hence increasing block size indiscriminately will, in fact, slow down execution. Threads in a single block are executed on a single multiprocessor hence large block sizes can cause shared memory (software-managed data cache) register spillage, which causes some rather unfortunate results. Specifically there is no determinate means to ensure that arrays assigned to shared memory will necessarily reside there, those that are deemed too large are moved into the global memory which can result in severe performance degradation. Inspection of the generated ptx file is recommended to ensure that such an event has not occurred.

Table 1 compares average execution times of Fortran code and CUDA kernel (optimal parameters and load conditions) for two different meshes to ensure scalability. Note that the CUDA runtimes include memory transfers to the wrapper and the CUDA kernel since this metric provides a more realistic estimate of achievable runtimes in real world applications. Runtimes for the larger mesh indicate that the algorithm scales well with the increased load. In fact the speedup increased from a factor of 9 for the smaller mesh to slightly over 11 for the larger mesh, which is consistent with the notion that GPU startup overhead and data transfer latencies are amortized by larger workloads. The size of the mesh that can be used is only limited by the GPU local memory, which in this case is 1.2 GB.

Table 1: Average execution times

Mesh	Fortran execution time	CUDA Kernel Execution time
Unstructured mesh with 2930 nodes and 14181 tetrahedron elements	0.45s	0.05s
Unstructured mesh with 18676 nodes and 99428 tetrahedron elements	3.25s	0.28s

5. Conclusion

CUDA kernels have been integrated into a Fortran code for a FEM EM solver and an 11x speedup over the corresponding Fortran subroutine has been obtained. It is shown that the CUDA API can be utilized to augment a Fortran electromagnetic solver through selective acceleration of code using Amdahl's law. Future work entails investigating a double precision code as well as an MPI Fortran framework with CUDA acceleration for scalable heterogeneous computing.

References

[1] http://www.fortran.com/tools.html

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

[3] http://developer.nvidia.com/directcompute

[4] http://x10−lang.org/home.html

[5] W.K. Anderson et al., "Petrov-Galerkin and discontinuous-Galerkin methods for time-domain and frequency-domain electromagnetic simulations", J. Comput. Phys., Vol 230 Issue 23, pp 8360-8385, 2011

[6] K.S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media", IEEE Trans. Anten. Propag. 14, pp 302-307, 1966

[7] J.S. Shang, R.M. Fithen, "A comparative study of characteristic-based algorithms for the Maxwell equations", J. Comput. Phys. 125 , pp 378-394, 1996

[8] C. Fumeaux, D. Baumann, R. Vahldieck, "Advanced FVTD simulation of dielectric resonator antennas and feed structures", Appl. Comput. Electromagn., Vol 19, pp 155-164, 2004

[9] Nvidia CUDA Programming Guide, http://developer.nvidia.com/cuda-toolkit-41

[10] D.D. Donno, A.Esposito, L.Tarricone and L.Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD, IEEE Antennas and Propagation Magazine", Vol. 52, No.3, June 2010

[11] R.Weber, A.Gothandaraman, R.J. Hinde, and G.D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study", IEEE Transactions on Parallel and Distributed Systems, Vol. 22, p58-68, Jan 2011

[12] W.B. Langdon, Simon Harding, "Performing with CUDA", CIGPU 2011 workshop, p423-430, 2011

[13] V.K. Decyk, C.D Norton, H.J Gardner, "Why Fortran, Computing in Science and Engineering", Vol 9, Issue: 4, p68-71, 2007

[14] M.Ujaldon, "Using GPU's for accelerating electromagnetic simulations", ACES Journal, Vol 25, No 4, April 2010

[15] K.Compton, S Hauck, "An introduction to Reconfigurable Computing", April 2000