Impact of GPU Memory Access Patterns on FDTD

Matthew Livesey*, James F. Stack, Jr.†, Fumie Costen‡, Takeshi Nanri§, Norimasa Nakashima§, Seiji Fujino§
* Accenture, Bailey Lane, Manchester, M21 9HE, UK
† Remcom, Inc. , 315 S. Allen Street, Suite #416, State College, PA, 16823, USA
‡ School of Electrical and Electronic Engineering, the University of Manchester, M13 9PL, U.K. and RIKEN, Saitama, Japan
§ Kyushu University, Japan

***Abstract*—The application of General Purpose computing on a GPU is an effective way to accelerate the FDTD method. This work explores the different domain decomposition techniques from the literature and extends the theoretically best approach with additional flexibility. We examine the performance on both Tesla and Fermi architecture GPUs and identify the best way to determine the GPU parameters for the proposed method.**

I. Introduction

The use of General Purpose computing on Graphics Processing Units (GPGPU) to execute scientific computations is becoming increasingly prevalent. GPGPU is particularly suitable for executing problems with a high degree of data parallelism, in order to make use of the many processing units present on a typical GPU.

The Finite-Difference Time-Domain (FDTD) method is popular because it directly solves Maxwell's curl equations with a minimal set of assumptions, thus providing a robust, straightforward method. The FDTD method is characterised by long FDTD iterations over large amounts of data organised into multidimensional arrays. Due to significant data independence between the calculations performed for each element in each array, the FDTD method exhibits a large degree of parallelism and is suitable for execution on a GPU.

II. General Purpose GPU computing with CUDA

NVIDIA's Tesla Architecture introduced a GPU hardware design consisting of an array of general-purpose, streaming multi-processors. The architecture of a Tesla GPU is described in detail in [1].

In conjunction, NVIDIA introduced the Compute Unified Device Architecture programming model (CUDA) [1] [2]. CUDA allows a programmer to specify which sections of a computation (described as 'kernels') should be executed on the GPU [1]. A kernel operates over a grid of data, where a grid is divided into blocks in either one or two dimensions. Each block is further divided into threads in one, two or three dimensions [2]. Within one kernel, a thread is identified by its indices at both the block and thread level.

The global memory of a Tesla architecture GPU is currently upto 4 GB and accessible to all threads in all blocks in a grid. It has limited bandwidth and no cache so will limit performance if used too extensively [2]. It is possible to optimise the performance of access to global memory through "memory coalescing" [2]. If simultaneously executing threads within a block access consecutive memory locations the requests are combined into a single larger memory fetch. Fetching a large, consecutive block of memory results in more efficient use of the memory bandwidth [2].

A major revision to the architecture, named Fermi, introduced improvements in the memory system. With Fermi the shared memory can be configured as partially user programmable and partially a level 1 cache [3]. A level 2 cache to global memory, unified across all SMs, is also introduced [3].

III. Existing implementations of the FDTD method on GPU hardware

The FDTD method is a common algorithm for modelling electro-magnetic behaviour through computation. It is covered in detail in [4]. Several implementations of the three-dimensional FDTD method using CUDA have been published. Each of these exhibits one of three domain decomposition approaches to mapping the x,y and z dimensions of the three dimensional FDTD space to the allocation of blocks and threads.

In the first approach, the FDTD space is broken down into two-dimensional planes, with the number of planes equal to the number of elements along one axis. Within each plane, the elements are divided into blocks and threads applied to the other two axes. The FDTD equations for a particular plane are solved in parallel using a kernel on the GPU, but each plane is computed by its own kernel invocation, sequentially to the others within a single time-step. While this method allows very fine-grained decomposition to expose all of the parallelism within a single plane, it does not exploit the full concurrency of the algorithm since all the calculations in all planes are independent from each other. This approach is used in [5] and [6].

The second approach addresses the entire three-dimensional space within a single kernel invocation. It also allocates blocks and threads in two dimensions, however in this case each thread executes all of the elements in the z direction in a `for` loop. This means that each thread does much more work than in the first approach and only one kernel instance is required to address the full 3D space, but it does not alter the amount of parallelism exposed overall. This approach is demonstrated in [7] and also used in [8].

The third approach takes a fundamentally different approach to domain decomposition. Blocks are allocated in two dimen-

978-1-4673-0462-7/12/$31.00 ©2012 IEEE

sions and the x and y indices of each block are mapped to two of the x, y and z dimensions of the FDTD space. Threads within each block are allocated in one dimension, and the x index of the thread is mapped to the remaining dimension of the FDTD space. Each thread has three indices (two inherited from the block and one for its position within the block) which are mapped to the dimensions of 3D space. One thread can be allocated to every cell in the FDTD space, exposing the maximum available parallelism in the algorithm. This method of decomposition is demonstrated in [9] and [10].

We use a similar method to the third approach, but with flexibility in how much work is done by each thread. The number of cells executed by each thread is determined by the ratio of threads per block to N_x (number of elements in the x dimension). If N_x is 6 , but the number of threads per block is 3, each thread is responsible for two cells in the x dimension.

IV. Numerical experiments

Our GPU implementation was executed on a Tesla T10 GPU and on a Fermi M2050 GPU. On each GPU, the implementation was executed for 5000 time-steps. The calculation was performed in both single and double precision using both the coalesced and uncoalesced configuration.

The results of our experiments are shown in Fig. 1.

It is clear that the coalesced approach provides better performance than the uncoalesced approach, and the coalesced approach performs best when the ratio between the number of threads per block and the number of cores available in a GPU board is around 0.15.

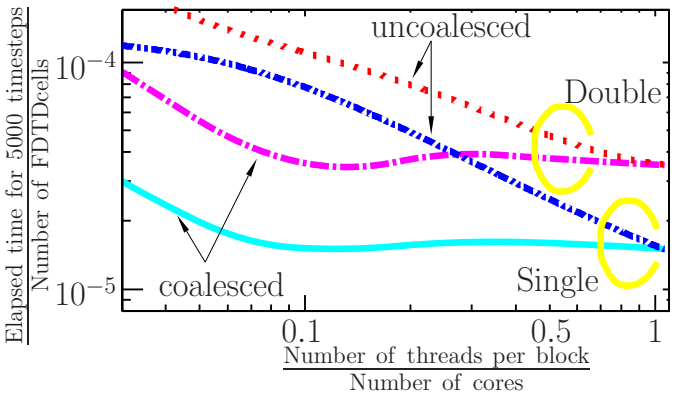
In case of the coalesced approach with single precision, the T10 GPU takes about 1.5 times as much time as the M2050 GPU. This suggests the improvement of the performance by the M2050 GPU in single precision mainly comes from improved memory bandwith on the M2050, which is 1.4 times wider than that of the T10 GPU.

In case of double precision, our coaleased approach performs better on the M2050 GPU than the T10 GPU, reducing 60 % of the elapsed time of the T10 GPU. Although the superior double precision performance in the M2050 GPU contributes to this improved overall performance, we find the major contribution to be the increased memory bandwidth for double precision data on the M2050 GPU.

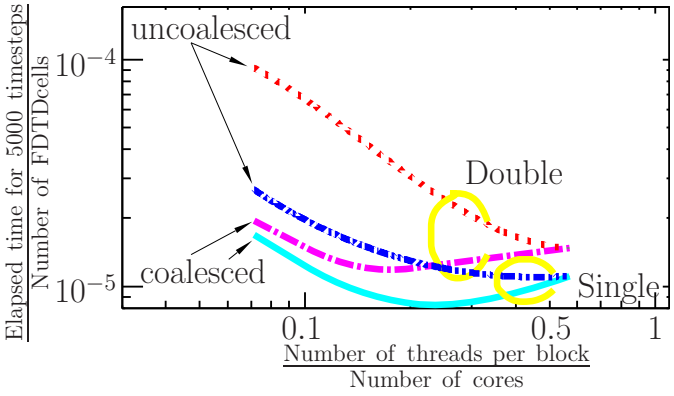
V. Conclusion

We find that there are three approaches to domain decomposition in the literature when applying GPGPU to the FDTD method in three dimensions. One of these gives the opportunity to exploit the full parallelism in the algorithm. We show through a flexible implementation of this approach that coalesced memory access provides beneficial performance, and that the amount of work performed by each thread also influences performance.

We also find that the newer Fermi architecture provides performance improvements over the Tesla architecture and attribute this improvement to the improved memory subsystem on Fermi GPUs.



(a) T10



(b) M2050

Fig. 1. Performance of our implementation on the T10 GPU and the M2050 GPU.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39 –55, March-April 2008.

[2] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[3] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi." Whitepaper, 2009.

[4] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. New York: Artech House Publishers, 3 ed., 2005.

[5] J. Chi, F. Liu, E. Weber, Y. Li, and S. Crozier, "GPU-accelerated FDTD modeling of radio-frequency field-tissue interactions in high-field MRI," *IEEE Trans. Biomed. Eng.*, vol. 58, pp. 1789 –1796, june 2011.

[6] Z. Bo, X. Zheng-hui, R. Wu, L. Wei-ming, and S. Xin-qing, "Accelerating FDTD algorithm using GPU computing," in *IEEE Int. Conf. Microw. Tech. Comput. Electromagn.*, pp. 410 –413, may 2011.

[7] T. Nagaoka and S. Watanabe, "A gpu-based calculation using the three-dimensional fdtd method for electromagnetic field analysis," in *IEEE Int. Conf. Eng. Med. Biol. Soc.*, pp. 327 –330, 31 2010-sept. 4 2010.

[8] J. F. Stack, "Accelerating the finite difference time domain (FDTD) method with CUDA," in *Appl. Comput. Electromagn. Soc. Conf.*, 2011.

[9] C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, and M. Okoniewski, "Speed it up," *IEEE Microw. Mag.*, vol. 11, pp. 70 –78, april 2010.

[10] P. Sypek, A. Dziekonski, and M. Mrozowski, "How to render FDTD computations more effective using a graphics accelerator," *IEEE Tran. Magn.*, vol. 45, pp. 1324 –1327, march 2009.