Finite Element Method for Computational Electromagnetics on Exascale Heterogeneous Computing Platforms

Olivier M. Cotte[1] and Dennis D. Giannacopoulos[1,2]

[1]Department of Electrical Engineering, McGill University, Montreal, QC H3A 0E9 Canada,
olivier.moussavoucotte@mail.mcgill.ca

[2]Computational Electromagnetics Lab, Department of Electrical & Computer Engineering, McGill University,
dennis.giannacopoulos@mcgill.ca

This paper demonstrates a strategy to solve large-scale parallel finite-element time-domain (FETD) with a performance analysis on heterogeneous computer architectures. Scalability is demonstrated with speedup studies of a 3D metallic body illuminated by an electromagnetics plane wave with more than $(2^{31}-1)$ degrees of freedom on a computer system with an Intel® Xeon™ CPU and NVIDIA® RTX™ GPU. This work is significant because in the domain of finite element analysis in computational electromagnetics (CEM) with complex geometry we demonstrate on a parallel large-scale computing platform local machine—the HP Z8 G5 workstation an approach to address principal constraints in CEM involving memory bandwidth and capacity, which affect FETD efficiency and precision.

Index Terms—Exascale computing, Finite element analysis, Numerical models, Parallel architectures, Sparse matrices

I. Introduction

ACCORDING to the U.S. Department of Energy (DOE), the exascale-class machines (capable of 10^{18} floating-point operations per second or more) will be a heterogeneous mix of CPUs and specialized processing units such as Graphical Processing Unit (GPU) and Field programmable gate arrays (FPGA) [1]. Memory and bandwidth will not increase and data transfer latencies will be further exposed [2].

The finite element (FE) method is used in CEM simulations and requires that linearized systems of algebraic equations (the stiffness matrix) be solved at each time step. These linear systems of equations solution are one of the computational bottleneck as the scale of problems increases. Direct solvers (eg, Lower Upper (LU) decomposition [3]), iterative-solver (eg, preconditioned conjugated gradient (CG) [4]) are popular but the size of some problems may renders these methods impractical.

Suppose we have a large positive-definite sparse matrix in compressed sparse row (CSR) format [5], [6] with dimension $n \times n$, denoted as A. The partitioning procedure is accomplished through METIS [7] using graph-based methods, whereas the matrix algebra on GPU and multicore architectures (MAGMA) [8] library handles submatrix factorization and solves.

The parallelization occurs at the vectorization level using AVX/AVX2/AVX-512 instructions on two distinct Intel architectures—Knights Landing (Intel Xeon Phi 7120p, also referred to as MIC or Xeon Phi) and Xeon Gold 6138P—and on an NVIDIA CUDA GPU (RTX 2080 + RTX 4080). Intel CPUs support AVX, AVX2, and AVX-512 instruction sets (with AVX-512 first integrated in Knights Landing).

Additionally, we incorporate the variable band matrix storage as format for the sparse matrix representation with optimum storage requirements for positive-definite matrices. Specifically, for symmetric matrices, only one triangular portion of the system is kept [9]. Consequently, skyline matrix storage (SKS) is compatible with Cholesky decomposition [10]. We are then able to verify the claim that CSR is more suitable than SKS format for large problems.

II. Methodology

Heterogeneous computer architectures, where CPUs co-exist with accelerators such as vector coprocessors, GPUs and FPGAs, are evolving for exascale computing. The software Gmsh [11] provides access to CAD boundaries, mesh generation in 1D, 2D, and 3D, interpolation with hierarchical basis functions for both H^1 and $H(\text{curl})$ spaces, and integration for all element shapes. GmshFem [12], an open-source C++ finite element framework built onto the Gmsh API, handles mesh-based sparse matrix assembly. This framework also allows external solvers (e.g., PETSc [13]) as built-in and storing large sparse matrices in CSR format. The system software component of the Intel Xeon Phi system, known as the Intel Many Integrated Core (MIC) Platform Software Stack (MPSS) also provides this functionality. We then exploit auto-vectorization features on Intel compilers to accelerate linear algebra routines. A comparable evaluation by [14] showed Intel's oneAPI porting a CUDA GEMM routine from MAGMA to Data Parallel C++ (DPC++).

Fig. 1 presents a comparative overview of various memory management types, including traditional CPU/GPU shared memory, GPU-specific memory, CPU-pinned memory, and the abstraction provided by Thrust's memory management interfaces.

Table I summarize these issues. One of the reasons to choose Thrust is in contrast to CUDA APIs such as

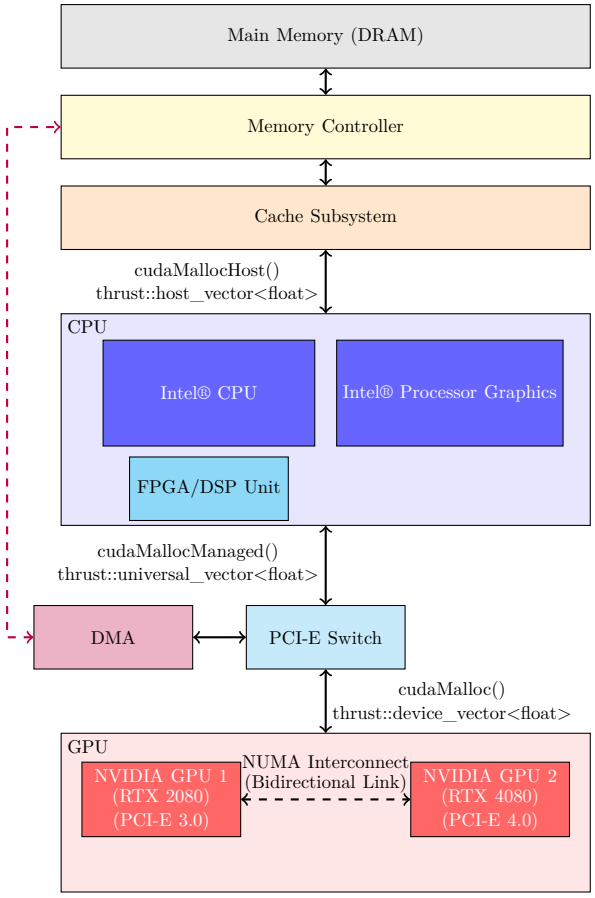


Fig. 1: HP Z8 G5 Workstation

TABLE I: Comparison of Memory Management Types for CPU, GPU, Shared, and Thrust Memory Management

Memory Location	Allocation Interface	PTE Init	Cache Coherent	Byte Alignment
CPU/GPU Shared Memory				
CPU	malloc() (C), new() (C++)	CPU	Yes	8/16 bytes
CPU/GPU	cudaMallocManaged()	CPU	Yes	256 bytes
GPU Memory				
GPU	cudaMalloc()	GPU	No	256 bytes
CPU-Pinned Memory				
CPU	cudaMallocHost(), cudaHostAlloc()	CPU	No	256 bytes
Thrust Abstraction (Unified CPU/GPU Access)				
CPU	thrust::host_vector<> (Host Memory)	CPU	Yes	8/16 bytes (host alignment)
GPU	thrust::device_vector<> (Device Memory)	GPU	No	256 bytes
CPU/GPU	thrust::universal_vector<> (Unified Memory)	CPU	Yes	256 bytes

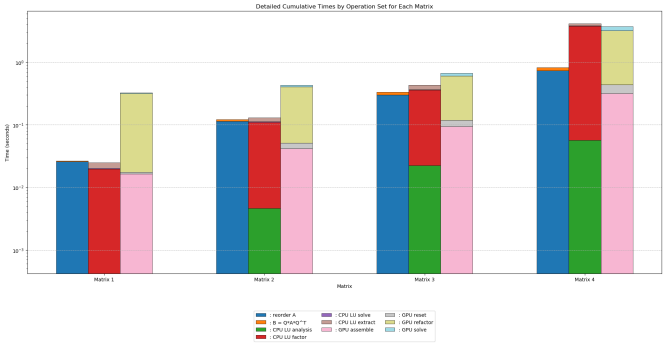


Fig. 2: Detailed cumulative times across different matrices

cudaMalloc() and cudaMemcpy() that require low-level memory handling, Thrust is integrated with unified memory which provides automatic memory coherence and makes it an excellent choice for heterogeneous computing. Compared to malloc() or new() on the CPU, which only supports host memory, Thrust abstracts both host and device memory management.

Fig. 2 highligh the bottleneck with timing of each operations involve in solving the linear system. GPU-specific memory allocated using cudaMalloc() or thrust::device_vector<> and are not cache coherent thus requires explicit memory transfers for consistency.

III. Conclusion

Cache coherence ensures that changes made by the CPU are visible to the GPU and vice versa without explicit synchronization. This makes cudaMallocManaged() ideal for heterogeneous workloads. We ought to precisely test that hypothesis. Our ongoing efforts is also considering discontinuous Galerkin formulations on GPUs.

References

[1] J. Dongarra, J. Hittinger, J. Bell, L. Chacon, R. Falgout, M. Heroux, P. Hovland, E. Ng, C. Webster, and S. Wild, "Applied mathematics research for exascale computing," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2014.

[2] D. Kothe, S. Lee, and I. Qualters, "Exascale computing in the united states," Computing in Science & Engineering, vol. 21, no. 1, pp. 17–29, 2018.

[3] K. He, S. X.-D. Tan, H. Wang, and G. Shi, "Gpu-accelerated parallel sparse lu factorization method for fast circuit analysis," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 3, pp. 1140–1150, 2015.

[4] A. Dziekonski and M. Mrozowski, "Block conjugate-gradient method with multilevel preconditioning and gpu acceleration for fem problems in electromagnetics," IEEE Antennas and Wireless Propagation Letters, vol. 17, no. 6, pp. 1039–1042, 2018.

[5] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations," 1994.

[6] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," IEEE Transactions on parallel and distributed systems, vol. 27, no. 2, pp. 428–440, 2015.

[7] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[8] S. Tomov and J. Dongarra, "Matrix algebra on gpu and multicore architectures," in Proc. Workshop Electronic Structure Calculation Methods Accelerators, 2012, pp. 37–55.

[9] H. Samukawa, "A proposal of level 3 interface for band and skyline matrix factorization subroutine," in Proceedings of the 7th international conference on Supercomputing, 1993, pp. 397–406.

[10] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched cholesky factorization on a gpu," in 2014 43rd International Conference on Parallel Processing. IEEE, 2014, pp. 432–440.

[11] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities," International journal for numerical methods in engineering, vol. 79, no. 11, pp. 1309–1331, 2009.

[12] A. Royer, E. Béchet, and C. Geuzaine, "Gmsh-fem: an efficient finite element library based on gmsh," in 14th World Congress on Computational Mechanics (WCCM), ECCOMAS Congress 2020. Scipedia, 2021.

[13] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp et al., "Petsc users manual," 2019.

[14] C. Rosales, "Porting to the intel xeon phi: Opportunities and challenges," in 2013 Extreme Scaling Workshop (xsw 2013). IEEE, 2013, pp. 1–7.