

Fast Multipole Methods on Graphics Processors

Nail A. Gumerov¹ and Ramani Duraiswami¹

ABSTRACT/ INTRODUCTION

Graphics Processors (GPUs) provide access to significant computational processing resources. They contain a large number of processing units with access to local and shared memory, and achieve significant speedups vis-à-vis CPUs on problems that can be mapped to their SPMD architecture. Many applications in molecular dynamics, astrophysics and other areas require the $O(N^2)$ computation of mutual Coulombic potentials and forces among N particles. The FMM provides a hierarchical approximate algorithm, to compute these quantities to a specified error ε at $O(M \log N)$ cost and memory. More generally FMM like algorithms are used to accelerate matrix vector products in applications such as the solution of integral equations, radial basis function interpolation/evaluation and machine learning. On an NVIDIA 8800 GTX installed on a PC, our FMM algorithm achieves timings that if computed using an $O(N^2)$ algorithm correspond to speeds of 25-45 Tflops (for achieved L_2 errors of $\sim 10^{-6} - 2 \times 10^{-4}$).

Keywords

Fast Multipole Method, GPU, GPGPU, Personal Supercomputing, NVIDIA CUDA, GPU/Multicore Development Environment.

1. FAST MULTIPOLE METHODS

The fast multipole method provides an approximate computation of the following sum (or matrix vector product)

$$\phi_j = \phi(\mathbf{y}_j) = \sum_{i=1}^N \Phi(\mathbf{y}_j, \mathbf{x}_i) q_i, \quad j=1, \dots, M, \quad \mathbf{x}_i, \mathbf{y}_j \in R^d. \quad (1)$$

where $\{\mathbf{x}_i\}$ are sources, $\{\mathbf{y}_j\}$ receivers, q_i strengths, and d the dimensionality of the problem. The kernel function $\Phi(\mathbf{y}_j, \mathbf{x}_i)$, for the case of monopoles and dipoles of the Laplace equation are

$$\Phi(\mathbf{y}_j, \mathbf{x}_i) = |\mathbf{y}_j - \mathbf{x}_i|^{-1}, \quad \Phi(\mathbf{y}_j, \mathbf{x}_i) = \frac{\mathbf{x}_i - \mathbf{y}_j}{|\mathbf{y}_j - \mathbf{x}_i|^3}, \quad \mathbf{x}_i \neq \mathbf{y}_j.$$

The FMM was introduced by Rokhlin & Greengard in 1987 [1]. It can be summarized as follows. Assume that the computational domain including all sources and receivers is included in a large cube (which by way the scaling can be made to provide the unit cube). This cube then is subdivided by an octree by dividing this box into 8 sub-cubes, and going down to level l_{\max} , so that at each level we have 8^l boxes ($l = 0, \dots, l_{\max}$). Each box containing sources can be referred as a source box and each box containing receivers can be referred as a receiver box. Obviously, depending on the distribution the boxes can contain different number of points, and some boxes may even be empty. The empty boxes at all levels are skipped, which provides basic adaptivity of the FMM for non-uniform distributions. The source boxes constitute the source hierarchy (parent-child relationships)

and the receiver boxes form the receiver-hierarchy. The structuring of the initial source and receiver boxes can be done using spatial ordering (bit-interleaving technique, e.g. see [2]). This technique also allows one to determine neighbors for each box, which is important for the FMM. Also some computations necessary for the FMM run part can be done before the run of the FMM (for example, computation of elements of the translation matrices). This is the precomputation part of the FMM.

The run part of the FMM computes $\phi(\mathbf{y}_j) \{ \}$ for an arbitrary input vector \mathbf{q} . This part can be repeated for the same source and receiver locations many times (e.g. for iterative solution of linear system (1), when the solution $\{q_i\}$ must be found. It consists of three steps: *Upward pass*, *Downward pass*,

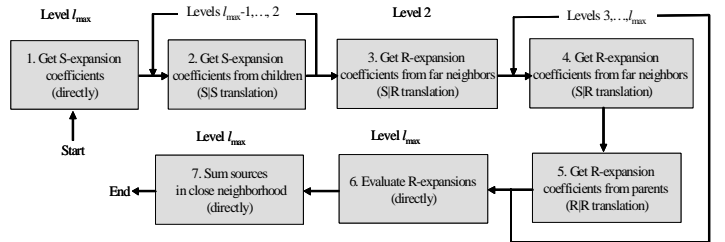


Fig. 1: Outline of the FMM algorithm for evaluating the sum (1) and Final summation.

It effectively decomposes the sum in to a near part, that directly computes for a receiver the sum due to sources contained its box at the finest level, and due to those in neighboring boxes, while the sum due to all the other points are estimated via a sequence of expansions and translations that result in a consolidated local function expansion at the center of each box, that may be evaluated at each receiver to obtain the sum to a given accuracy ε , which is controlled by the number of terms retained in the expansions. Several variants of the algorithm differing in the type of translation used, the data structures considered, and others are available; and discussed and compared in [3].

When both force and potential calculations are involved, in 3D, we have to evaluate 4 sums of the type (1). When using the FMM, a substantial savings can be achieved, since only 4 local sums need to be evaluated, and the force evaluation corresponding to the far field can be obtained by differentiating the consolidated local function expansion, which corresponds to applying a sparse matrix to a p^2 vector of coefficients [5].

2. FMM on the GPU

The graphical processor we used was the NVIDIA 8800 GTX, with the CUDA programming environment [4]. Since the bulk of our software is developed with Fortran 95, we also developed a

¹ University of Maryland, College Park. NASA support is gratefully acknowledged.

middleware environment [5] that allowed processing of data resident on the GPU via function calls, and restricted the number of device functions in CU [4]. We had to consider peculiarities of the GPU architecture; namely a hierarchical memory layout, with significant penalties for accessing data from main memory, single precision computations, the need for high intensity computations, and the need to parallelize individual stages of the FMM [4]. For reasons of space, a longer discussion of the steps taken will be presented elsewhere, though the novel contributions include:

- i) a new set of basis functions which avoided the need to use spherical harmonics;
- ii) their use with rotation-coaxial translation operators;
- iii) individual error bounds for each expansion/translation;
- iv) novel FMM stencils and exploitation of symmetries in the multipole-to-local translation stage, reducing translations needed by $2/3$.

In addition, similar to some other groups, we developed fast routines for performing direct kernel sums on the GPU, and used that to accelerate the local summation part. The speedups vis-à-vis a standard serial CPU implementation (on a QX6700 Intel CPU) achieved on the GPU were (for $N=10^6$):

- i) speed up of local sums by $500\times$ for potential calculation and by $750\times$ for potential + forces
- ii) Effective speed up of multipole-expansions by $\sim 40\times$
- iii) Multipole to multipole translations by up to $40\times$
- iv) M2L was speeded up by $30\times$
- v) Local evaluation speedup was in the range $20-50\times$.

The FMM achieves optimal speedup when the wall clock times of the local summations and the far-field summations are approximately balanced, and to achieve this on the GPU (with a very efficient local summator) we had to reduce the number of levels, leading to larger local sums than on the CPU. In the above comparisons, we present results for the CPU version workings at its optimal settings, and likewise for the GPU.

3. RESULTS

Fig. 2 below summarizes the results from evaluation of the potential and force between N randomly placed sources and N different randomly placed evaluation points in the unit cube.

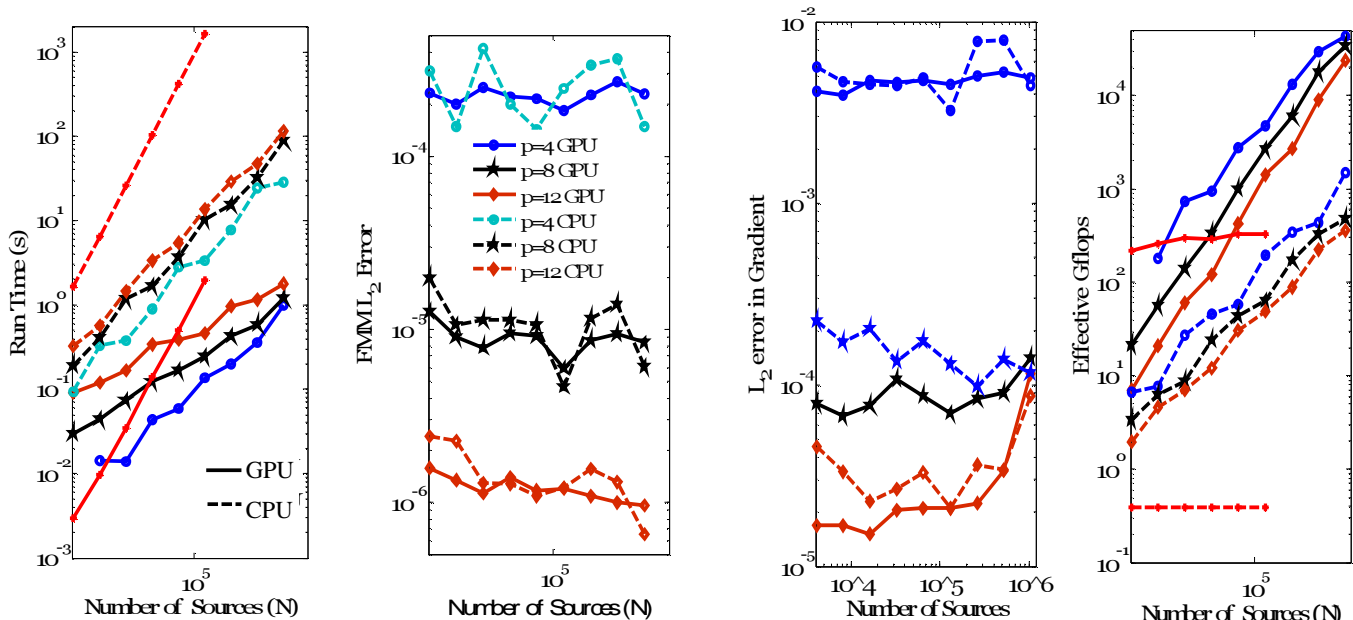


Fig. 2: Results achieved on our FMM code on the CPU and GPU for $N=4096, \dots, 1048576$. **Left:** Wall-clock time for the FMM run portion for different accuracy settings on the CPU and GPU. A roughly linear scaling is observed. Also shown are the direct product results for the GPU and CPU (the lines with slope =2). **Left Middle:** Error achieved on the CPU and GPU versions of the FMM for different truncation numbers. **Right Middle:** Error achieved in force calculation on the CPU and GPU versions of the FMM for different truncation numbers. **Right:** Effective number of Gflops achieved (following [6]). While the peak performance of the direct potential/force evaluation is about 200 GFlops, the FMM achieves speeds of up to 50 TFlops. Sum help used are consistent across

First, the direct summation on the GPU achieves significant speedup (these results are similar to the potential calculations in [7]). Next, the error with the GPU FMM is observed to be comparable to that on the CPU, both for gradient and potential evaluation. Interestingly, we observed that the errors achieved using the approximate FMM (at even the medium accuracy setting) were better than those achieved by the direct sum – presumably because the FMM has fewer operations, and thus smaller roundoff. Gflops are reported using a formula presented in [6].

4. CONCLUSIONS

In future work we are extending the algorithm to use multiple GPUs, and to different kernels such as the biharmonic. We believe availability of the FMM like algorithms on commodity graphics hardware will allow scientists to simulate much larger problems on their desktops, making them personal supercomputers.

5. REFERENCES

- [1] L. Greengard & V. Rokhlin, “A fast algorithm for particle simulations,” J. Comput. Phys., 73, 1987, 325-348.
- [2] N.A. Gumerov & R. Duraiswami. *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*, Elsevier, 2005.
- [3] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, V1.0, NVIDIA Corp. 06/2007.
- [4] N. Gumerov, R. Duraiswami & W. Dorland. “Middleware for programming NVIDIA GPUs from Fortran 9X,” submitted.
- [5] N.A. Gumerov & R. Duraiswami, “Comparison of the efficiency of translation operators used in the fast multipole method for the 3D Laplace equation,” Univ. of Maryland, Dept. Comp. Sci. Tech. Report CS-TR-4701
- [6] M.S. Warren, J.K. Salmon, D.J. Becker, M.P. Goda, T. Sterling, & G.S. Winckelmans. “Pentium Pro inside: I. a treecode at 430 Gigaflops on ASCI Red,” Bell price winning paper at SC’97, 1997.
- [7] J. Stone. Performance Case Studies: Ion Placement Tool VMD, Univ. Illinois. www.ks.uiuc.edu/Research/vmd/projects/ece498/