The Fast Multipole Method on NVIDIA GPUs and Multicore Processors

Toru Takahashi,[a] Cris Cecka,[b] Eric Darve[c]

[a] Department of Mechanical Science and Engineering, Nagoya University
[b] Institute for Applied Computational Science, Harvard University
[c] Institute for Computational and Mathematical Engineering, Stanford University

May 17 2012

Outline

1. Fast multipole method
2. Multipole-to-local operator: blocking and data access pattern
3. Particle-particle interactions
4. Numerical results: multicore CPU and NVIDIA GPU

Fast Multipole Method

Applications:

- Gravitational simulations, molecular dynamics, electrostatic forces
- Integral equations for Laplace, Poisson, Stokes, Navier. Both fluid and solid mechanics.
- Interpolation using radial basis functions: meshless methods, graphics (data smoothing)
- Imaging and inverse problems: subsurface characterization and monitoring, imaging
- $O(N)$ fast linear algebra: fast matrix-vector and matrix-matrix products, matrix inverse, LU factorization, for dense and sparse matrices, "FastLAPACK"

Matrix-vector products

FMM: a method to calculate

$$\phi(\mathbf{x}_i) = \sum_{j=1}^{N} K(\mathbf{x}_i, \mathbf{x}_j)\, \sigma_j, \quad 1 \leq i \leq N$$

in $\mathcal{O}(N)$ or $\mathcal{O}(N \ln^{\alpha} N)$ arithmetic operations.

There are many different FMM formulations.

Most of them rely on low-rank approximations of the kernel function K in the form:

$$K(\mathbf{x}, \mathbf{y}) = \sum_{q=1}^{p} u_q(\mathbf{x}) \sum_{s=1}^{p} T_{qs}\, v_s(\mathbf{y}) + \varepsilon$$

FMM low-rank approximation

Many formulas are possible to obtain a low-rank approximation.

We use an approach based on interpolation formulas for smooth functions. This allows building an FMM that is:

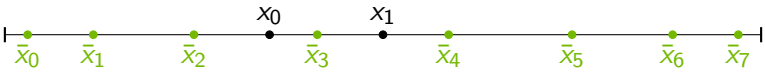
- Very general: any smooth (even oscillatory) kernels can be used.
- Simple to apply: we only need a routine that evaluates K.
- Converges rapidly and has optimal rank given an L^2 error.

It is however less efficient than specialized methods for $K(\mathbf{x}, \mathbf{y}) = 1/|\mathbf{x} - \mathbf{y}|$.

Interpolation scheme

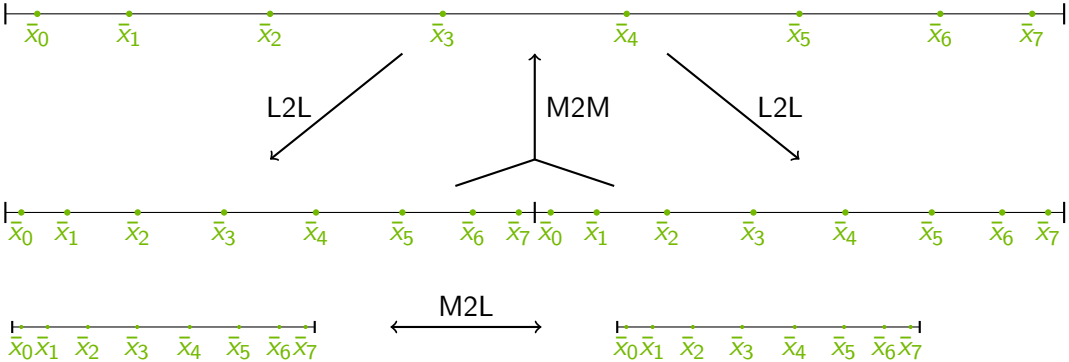
Low-rank approximation is obtained through a Chebyshev interpolation scheme:

$$K(\mathbf{x},\mathbf{y}) \approx \left(\frac{\ell}{2^{k+1}}\right)^{\lambda} \sum_{\mathbf{l}} \sum_{\mathbf{m}} R_n(\bar{\mathbf{x}}_{\mathbf{l}},\mathbf{x})\; K(\bar{\mathbf{x}}_{\mathbf{l}},\bar{\mathbf{y}}_{\mathbf{m}})\; R_n(\bar{\mathbf{y}}_{\mathbf{m}},\mathbf{y})$$



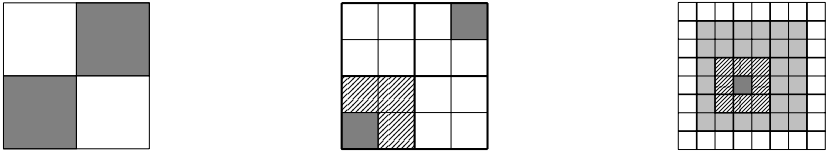
W. Fong, E. Darve, The black-box fast multipole method, *J. Comput. Phys.*, 228(23) (2009) 8712 8725.

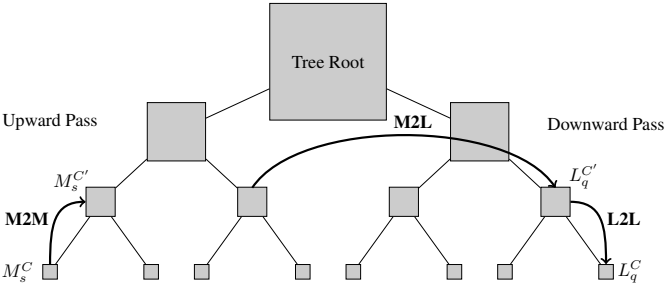
FMM operators



M2M and L2L operators are transpose of each other. The M2L operator depends only on the relative position of each cluster. It is independent of the location of particles inside each cluster.

Data structures and well-separated clusters

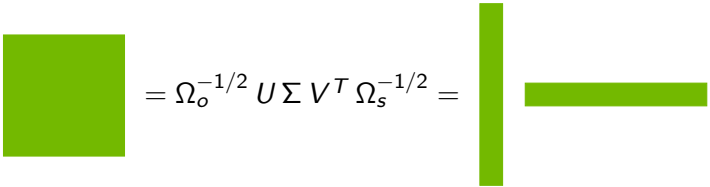




E. Darve, C. Cecka, and T. Takahashi. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. *Comptes Rendus Mécanique*, 339(2–3):185–193, 2011.

M2L operator

- The operator $K(\bar{\mathbf{x}}_\mathbf{l}, \bar{\mathbf{y}}_\mathbf{m})$ needs to be applied many times. It is independent of the particles positions $\mathbf{x}_i$. This allows certain optimizations to be computed off-line.
- A singular value decomposition is used to compress the operator $K(\bar{\mathbf{x}}_\mathbf{l}, \bar{\mathbf{y}}_\mathbf{m})$ to an optimal rank:

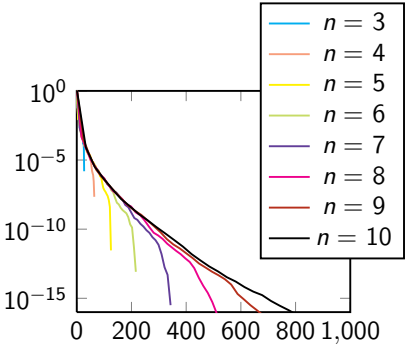
$\mathbf{K}_{IJ} =$  $= \Omega_o^{-1/2}\, U\, \Sigma\, V^T\, \Omega_s^{-1/2} =$

$\mathbf{K}_{IJ}\, \sigma_J =$ 

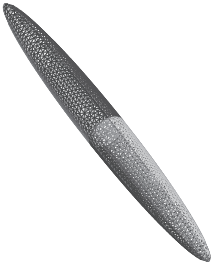
σ_J

Cost $= 2rN$ instead of N^2

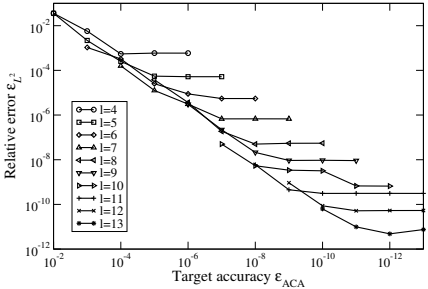
Convergence of Chebyshev interpolation and SVD compression



Error vs SVD cutoff for $K = 1/r$



2D prolate mesh



Error vs target accuracy for $K = e^{ikr}/r$

M. Messner, M. Schanz, and E. Darve. Fast directional multilevel summation for oscillatory kernels based on Chebyshev interpolation. *J. Comp. Phys.*, 2011.

M2L operator

The M2L operator, inside a given level, leads to a set of nested loops:

```
For all cluster C in this level {
  For all cluster C' in the interaction list of C {
    For all q=1 to p {
      For all s=1 to p {
        L(q)(C) += T(q,s)(C,C') M(s)(C');
      } } } }
```

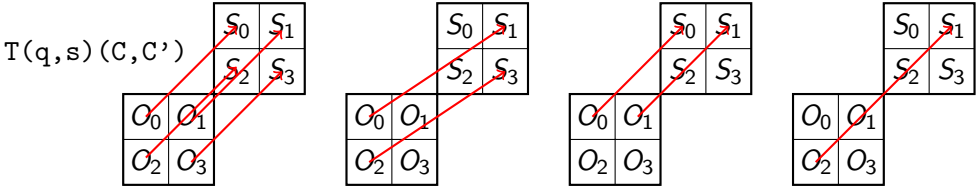
`C` and `C'` are clusters that are well separated. The matrix `T(q,s)` is the M2L operator. The vector `M` has the multipole coefficients and `L` the local coefficients.

The core operation is therefore a matrix-vector product.

Blocking strategies

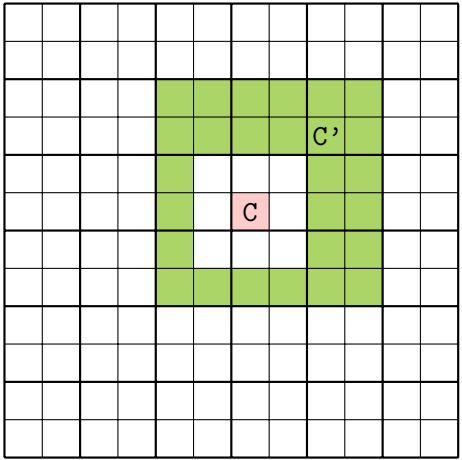
Four different strategies:

1. Basic approach: matrix-vector products.
2. Sibling blocking. Reuse `T(q,s)(C,C')` for all `(C,C')` that share the same `T`. This allows using matrix-matrix products instead of matrix-vector products.
   64 MV $\rightarrow$ 27 MM



3. Further block siblings into larger chunks. This is possible if the number of clusters is sufficient.

Fourth blocking strategy



4. Move the `q s` loop outside. This allows the greatest degree of reuse of `T(q,s)(C,C')` across all 316 pairs of clusters.

Downside: calculation is slightly more irregular due to the different treatment of each sibling. This only works at level 3 and below.

T. Takahashi, C. Cecka, and E. Darve. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *Int. J. Num. Meth. Engr.*, 2011.

Arithmetic intensity

Algorithm variant	Arithmetic intensity
Algorithm 1	1.9–2.0*
Algorithm 2	4.2–4.3*
Algorithm 3	21–32*
Algorithm 4	108–133†

Platform and algorithm	Gflops
CPU	125.2
GPU A1	72.0
GPU A2	146.9
GPU A3	257.3
GPU A4	358.8

*: memory bound; † arithmetic bound

Left: ratio of number of flops performed by kernel divided by number of words read from memory. Arithmetic intensity should be around 30–35 for peak efficiency.

Right: Performance of M2L in single-precision. Tech. details: medium accuracy, rank $= 32$, $N = 10^7$, $K = 1/r$, cube with random uniform dist., DELL Poweredge 1950 with two quad-core Intel Xeon E5345 2.33 GHz CPUs (OpenMP), Tesla C2050.

Comparison of algorithms on CPU (under development)

	Alg.	P	n	d	$N = 10^4$	10^5	10^6	10^7	10^8
CPU-6x	2	s	4	0	56(3)	64(4)	65(5)	65(6)	65(7)
CPU-6x	4	s	4	0	5(3)	12(4)	22(5)	25(6)	27(7)
CPU-6x	2	s	8	-1	24(2)	28(3)	31(4)	31(5)	32(6)
CPU-6x	4	s	8	-1	1(2)	5(3)	13(4)	22(5)	27(6)

Gflops in M2L computation (tree depth); algo. 2 and 4 on CPU

6-core Xeon X5680 3.33GHz

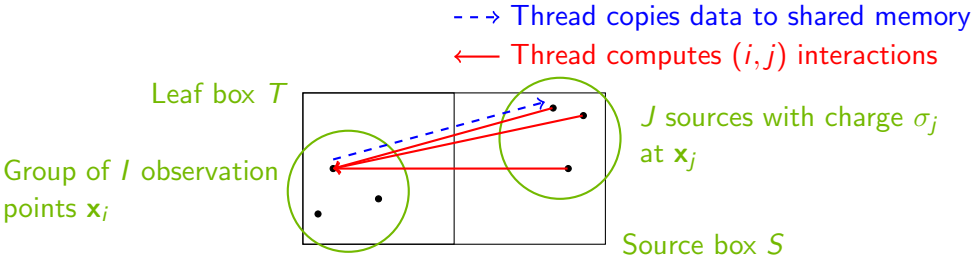
P: precision [s|d]; n: parameter that determines the accuracy [4|8]

d: parameter to adjust the depth of the tree; N: number of particles

Short-range interactions

- Assign one thread-block to one leaf box T. I: number of threads per thread-block.
- I threads process I observation points in T each iteration.
- For every source box S in $\mathcal{N}(T)$, all the threads load data in shared memory ($\mathbf{x}_j$ and σ_j) for J sources in S.
- The loop over sources (summation over j) is unrolled to improve performance.

With this approach a high-arithmetic intensity can be reached.



Validation of both codes

	P	n	d	10^4	10^5	10^6
CPU	s	4	0	7.39e-4	6.65e-4	7.89e-4
GPU	s	4	0	7.39e-4	6.65e-4	7.89e-4
CPU	s	8	-1	3.74e-6	8.05e-6	1.92e-5
GPU	s	8	-1	3.72e-6	8.07e-6	1.92e-5
CPU	d	4	0	7.32e-4	6.67e-4	7.80e-4
GPU	d	4	0	7.32e-4	6.67e-4	7.80e-4
CPU	d	8	-1	2.13e-7	2.25e-7	2.84e-7
GPU	d	8	-1	2.13e-7	2.25e-7	2.84e-7

Relative l^2 error of the CPU and GPU codes versus N^2 code.

$K(\mathbf{x}, \mathbf{y}) = 1/|\mathbf{x} - \mathbf{y}|$. Points are randomly uniformly distributed in a box.

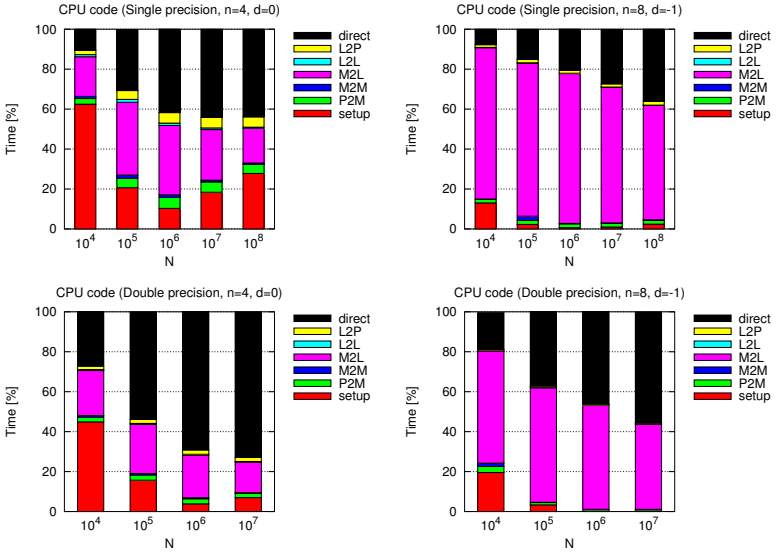
Hardware used for tests

Peak performance: top is single precision while bottom is double precision.

Processor	Xeon X5680 six-core, 3.33GHz two sockets†	Tesla C2050
# of cores in total	12	448
Peak performance	319.68	1030‡
[Gflop/s]	159.84	515‡
Bandwidth [GB/s]	64*	115♯
Flop-to-word ratio	20	36

† Hyper-threading is not used. ‡ Performance for MAD operation.
* 32 GB/s per socket. ♯ ECC is enabled.

Benchmark results: serial CPU breakdown

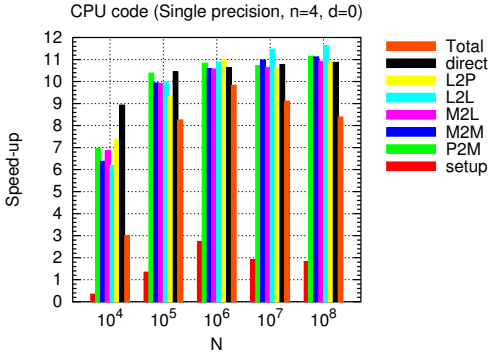


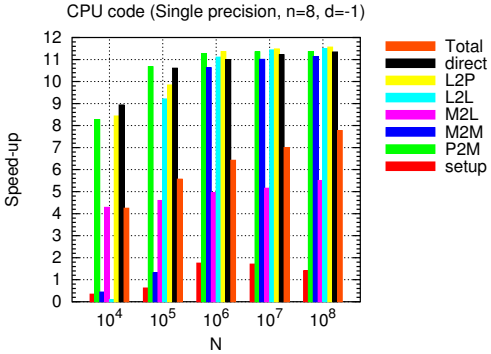
P = particle
M = multipole exp.
L = local exp.

Direct (P2P for leaf clusters) and M2L occupy 60–98% of the total running time.

Benchmark results: CPU speedup using 12 cores (OpenMP)

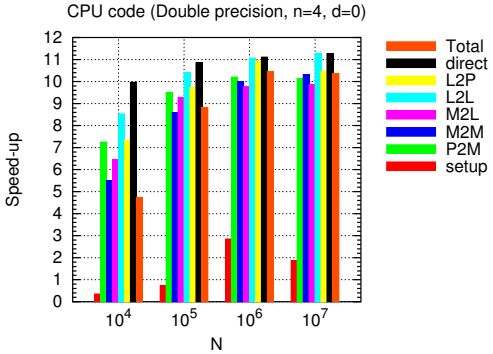
Single precision

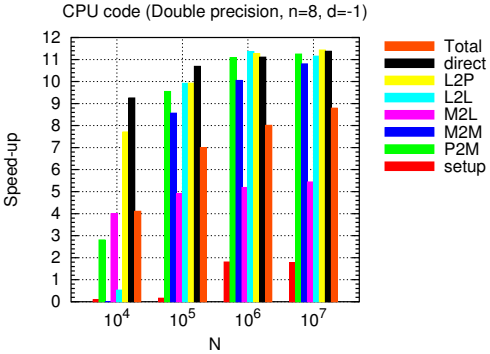




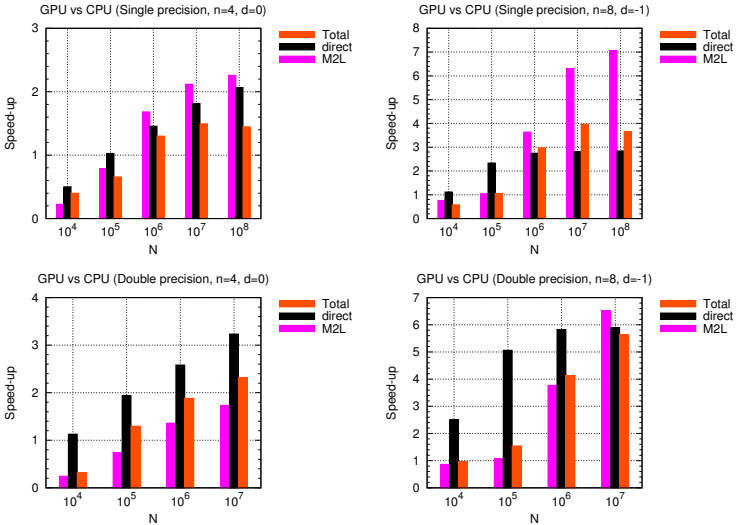
Benchmark results: CPU speedup using 12 cores (OpenMP)

Double precision





Speed-up of GPU vs CPU-12x



Gflops performance: M2L

M2L computation

	P	n	d	10^4	10^5	10^6	10^7	10^8
CPU-12x	s	4	0	95(3)	120(4)	123(5)	126(6)	128(7)
GPU	s	4	0	50(3)	173(4)	302(5)	359(6)	379(7)
CPU-12x	s	8	-1	35(2)	39(3)	41(4)	43(5)	46(6)
GPU	s	8	-1	63(2)	65(3)	192(4)	319(5)	375(6)
CPU-12x	d	4	0	58(3)	73(4)	75(5)	75(6)	NA
GPU	d	4	0	26(3)	83(4)	140(5)	165(6)	NA
CPU-12x	d	8	-1	17(2)	18(3)	19(4)	20(5)	NA
GPU	d	8	-1	41(2)	30(3)	88(4)	146(5)	NA

Gflops performance: P2P

P2P computation

	P	n	d	10^4	10^5	10^6	10^7	10^8
CPU-12x	s	4	0	144(3)	169(4)	182(5)	194(6)	210(7)
GPU	s	4	0	192(3)	282(4)	376(5)	491(6)	629(7)
CPU-12x	s	8	-1	221(2)	261(3)	275(4)	282(5)	286(6)
GPU	s	8	-1	373(2)	705(3)	826(4)	852(5)	861(6)
CPU-12x	d	4	0	35(3)	38(4)	39(5)	40(6)	NA
GPU	d	4	0	69(3)	96(4)	124(5)	157(6)	NA
CPU-12x	d	8	-1	33(2)	38(3)	40(4)	41(5)	NA
GPU	d	8	-1	98(2)	207(3)	243(4)	249(5)	NA

Conclusion

- Efficient and flexible fast multipole method: can be used for any analytic kernel, smooth or oscillatory.
- Good fraction of peak performance can be achieved. There is still room for improvement (e.g., instruction throughput optimization).
- We have not considered the case of an adaptive tree with a varying number of levels.
- Current work: use starPU to parallelize the method on multiple GPUs and CPUs. Overcome the tree-root bottleneck by dynamically scheduling the M2M, M2L, L2L and P2P operators. Increase arithmetic intensity using symmetries.
- The code, `bbFMM`, used for this work is available from `http://sourceforge.net/projects/bbfmmgpu`