FMM implementation on CPU and GPU

Nail A. Gumerov

(Lecture for CMSC 828E)

Outline

- Two parts of the FMM
- Data Structure
- Flow Chart of the Run Algorithm
- FMM Cost/Optimization on CPU
- Programming on GPU
- Fast Matrix-Vector Multiplication on GPU with On-Core Computable Kernel
- FMM Sparse Matrix-Vector Multiplication on GPU
- Change of the Cost/Optimization scheme on GPU compared to CPU
- Other steps
- Test Results
  - Profile
  - Accuracy
  - Overall Performance

Publication

- The major part of this work has been published:
  - N.A. Gumerov and R. Duraiswami
  - **Fast multipole methods on graphics processors.**
  - *Journal of Computational Physics*, 227, 8290-8313, 2008.

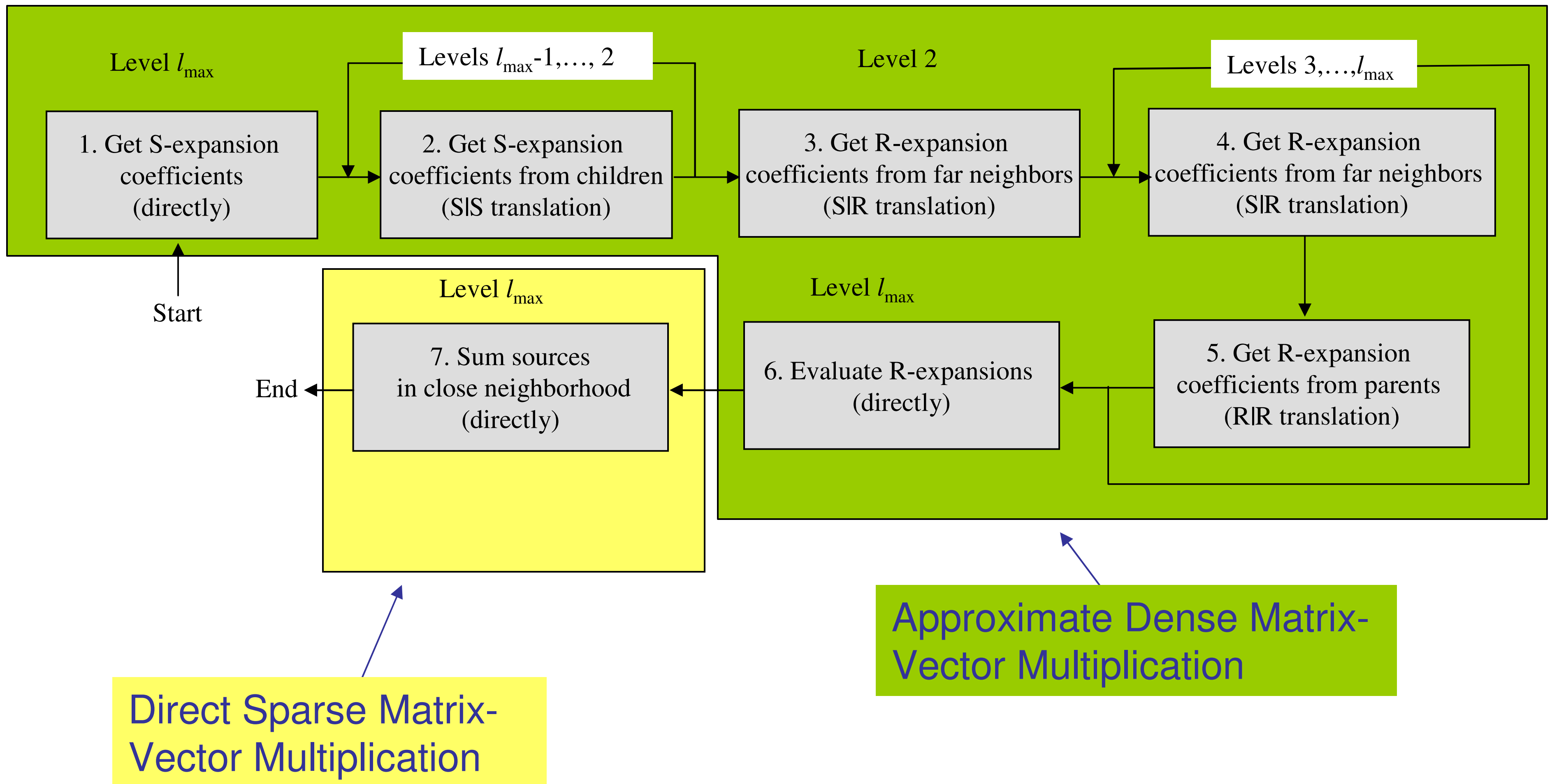
Two parts of the FMM

- Generate data structure (define the matrix);
- Perform Matrix-vector product (run);
- Two different types of problems:
  - Static matrix (e.g. for iterative solution of large linear system):
    - Data structure should be generated once, while run should be performed many times with different input vectors;
  - Dynamic matrix (e.g. N-body problem)
    - Data structure should be generated each time, when matrix-vector multiplication is needed.

Data Structure

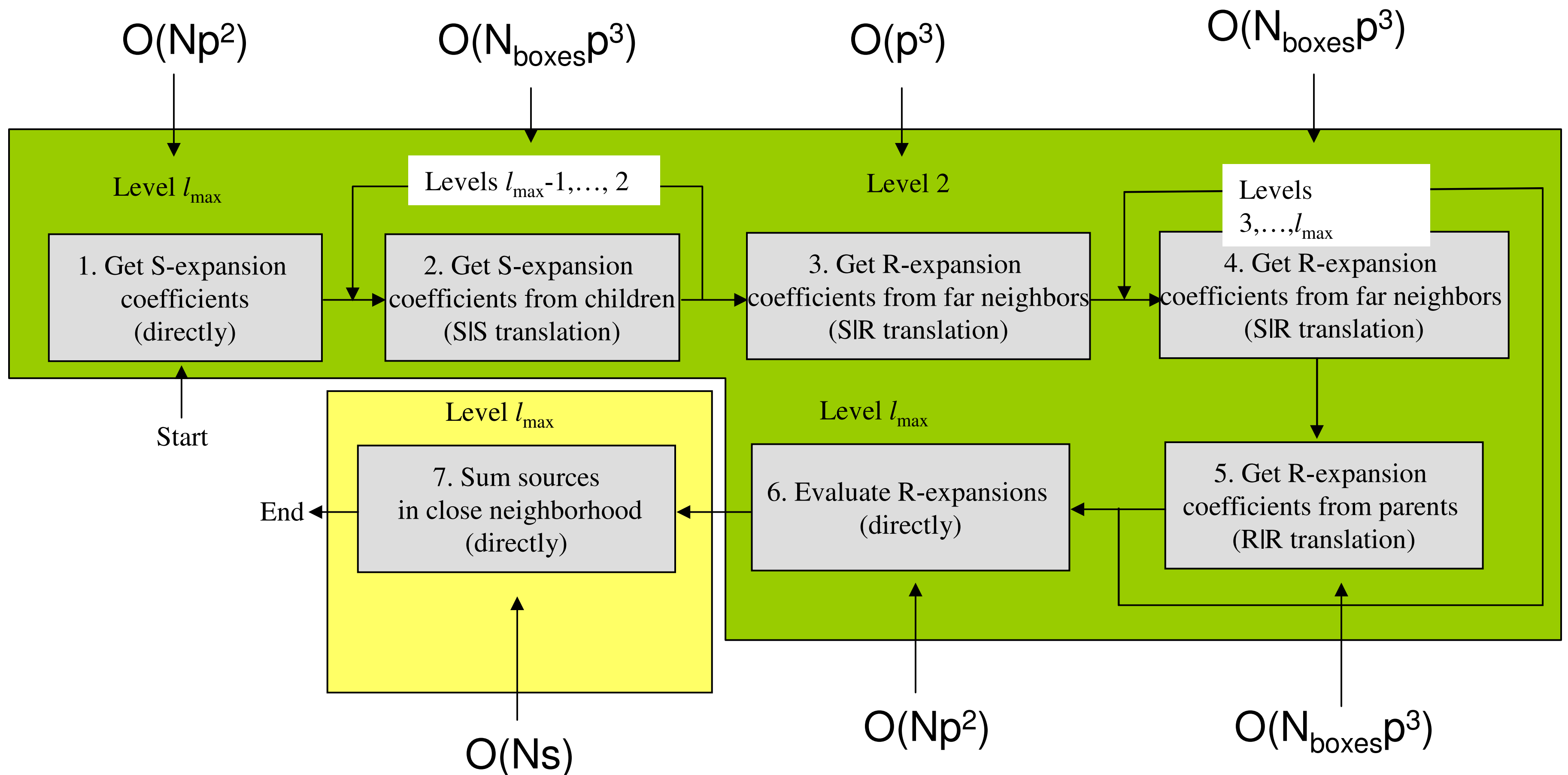
- We need O(NlogN) procedure to index all the boxes, find children/parent and neighbors;
- Implemented on CPU using bit-interleaving (Morton-Peano indexing in octree) and sorting;
- We generate lists of Children/Parent/Neighbors/E4-neighbors and store in global memory;
- Recently, graduate students Michael Lieberman and Adam O'Donovan implemented basic data structures on GPU (prefix sort, etc.) and found of order 10 speedups compared to the CPU for large data sets.
- In the present lecture we will focus on the run algorithm, assuming that the necessary lists are generated and stored in the global memory.

Flow Chart of FMM Run



FMM Cost (Operations)

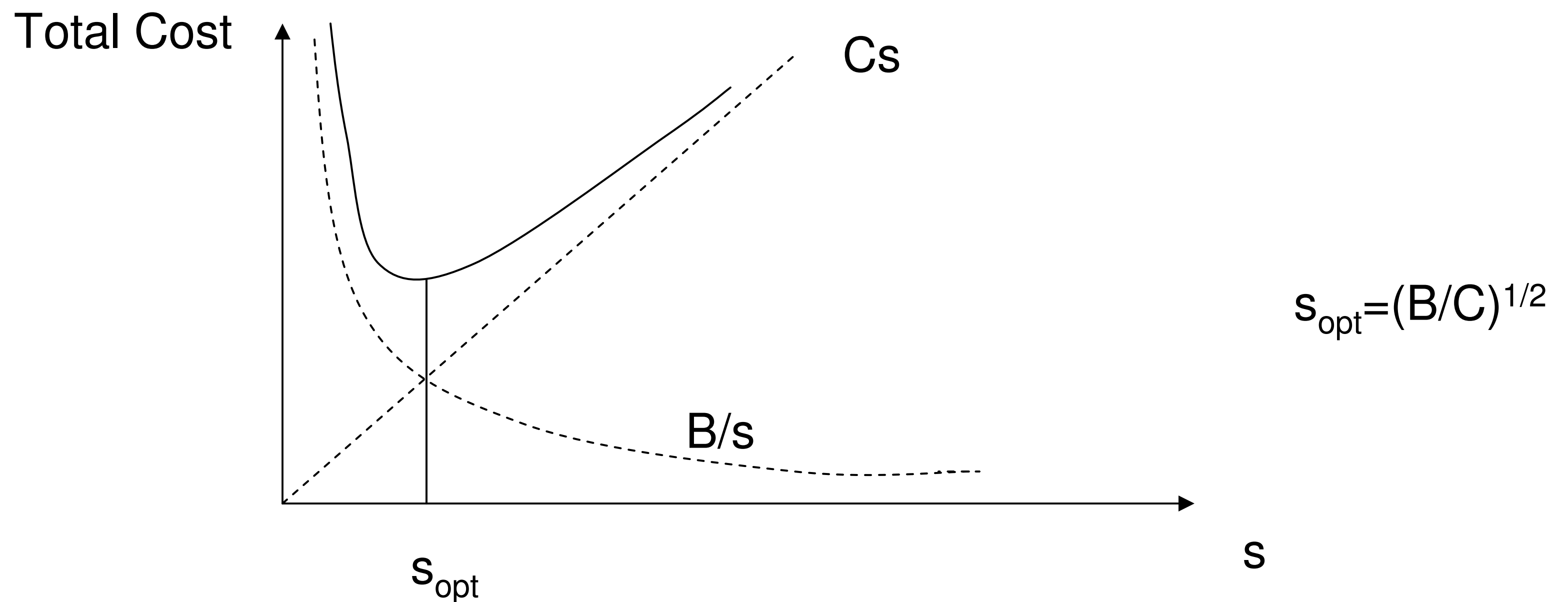
Points are clustered with clustering parameter *s*, means not more that *s* points in the smallest octree box. Expansions have length p^2 (number of coefficients). There are O(N) points and $O(N_{boxes})=O(N/s)$ in the octree.



FMM Cost/Optimization (operations)

Since p=O(1), we have

$TotalCost = AN+BN_{noxes}+CNs$
$=N(A+B/s+Cs)$



Challenges

- Complex FMM data structure;
- Problem is not native for SIMD semantics;
  - non-uniformity of data causes problems with efficient work load (taking into account large number of threads);
  - serial algorithms use recursive computations;
  - existing libraries (CUBLAS) and middleware approach are not sufficient;
  - high performing FMM functions should be redesigned and written in CU;
- Low fast (shared/constant) memory for efficient implementation of translation operators;
- Absence of good debugging tools for GPU.

Accelerated by Fantalgo

FMM on GPU

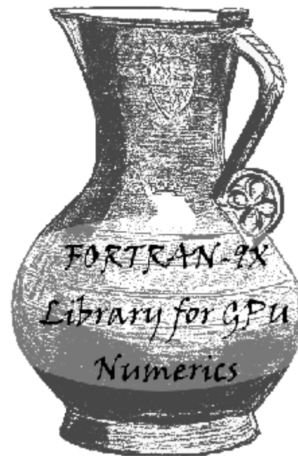
Implementation

- Main algorithm is in Fortran 95
- Data structures are computed on CPU
- Interface between Fortran and GPU is provided by our middleware (Flagon), which is a layer between Fortran and C++/CUDA
- High performance custom subroutines written using Cu

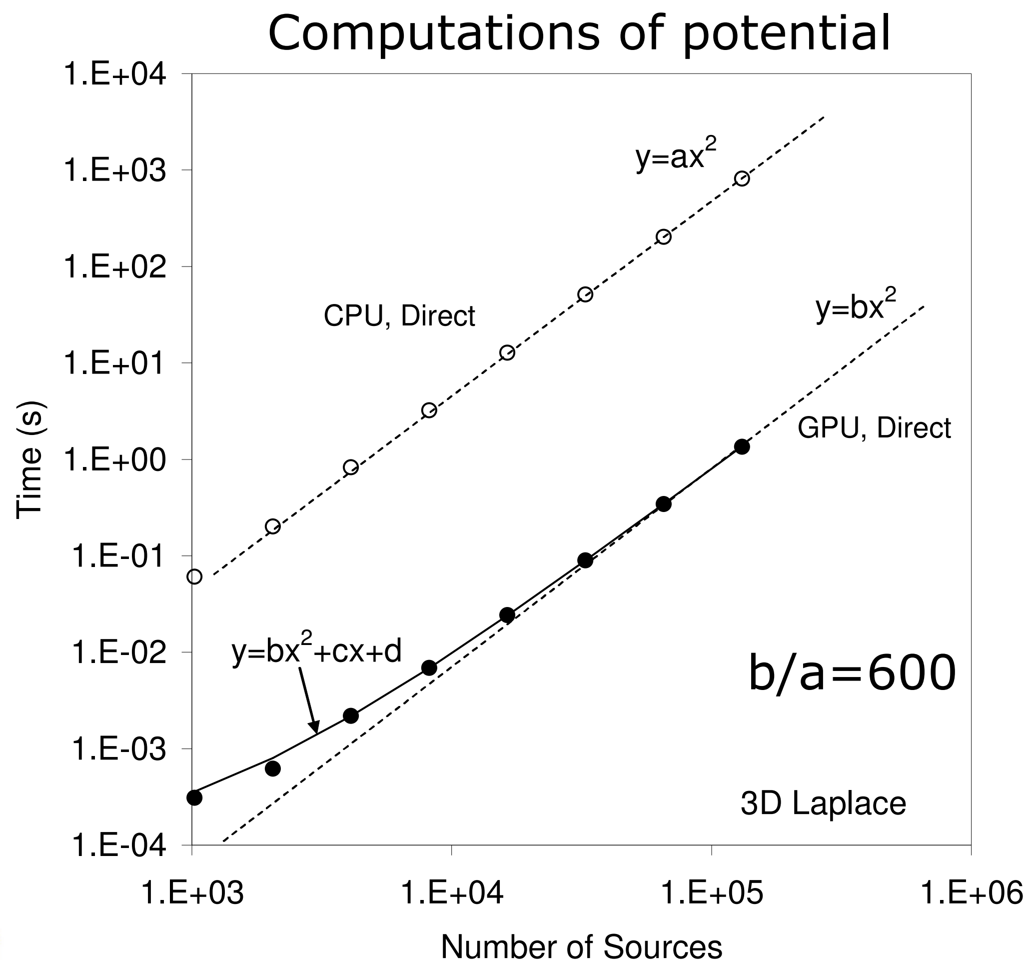
Availability

- Fortran-9X version is released for free public use. It is called FLAGON.
- http://flagon.wiki.sourceforge.net/

FLAGON: Fortran-9X Library for GPU Numerics



High performance direct summation on GPU (total)



CPU: 2.67 GHz Intel Core 2 extreme QX 7400 (2GB RAM and one of four CPUs employed).

GPU: NVIDIA GeForce 8800 GTX (peak 330 GFLOPS).

Estimated achieved rate: 190 GFLOPS.

CPU direct:
- serial code;
- no use of partial caching;
- no loop unrolling;

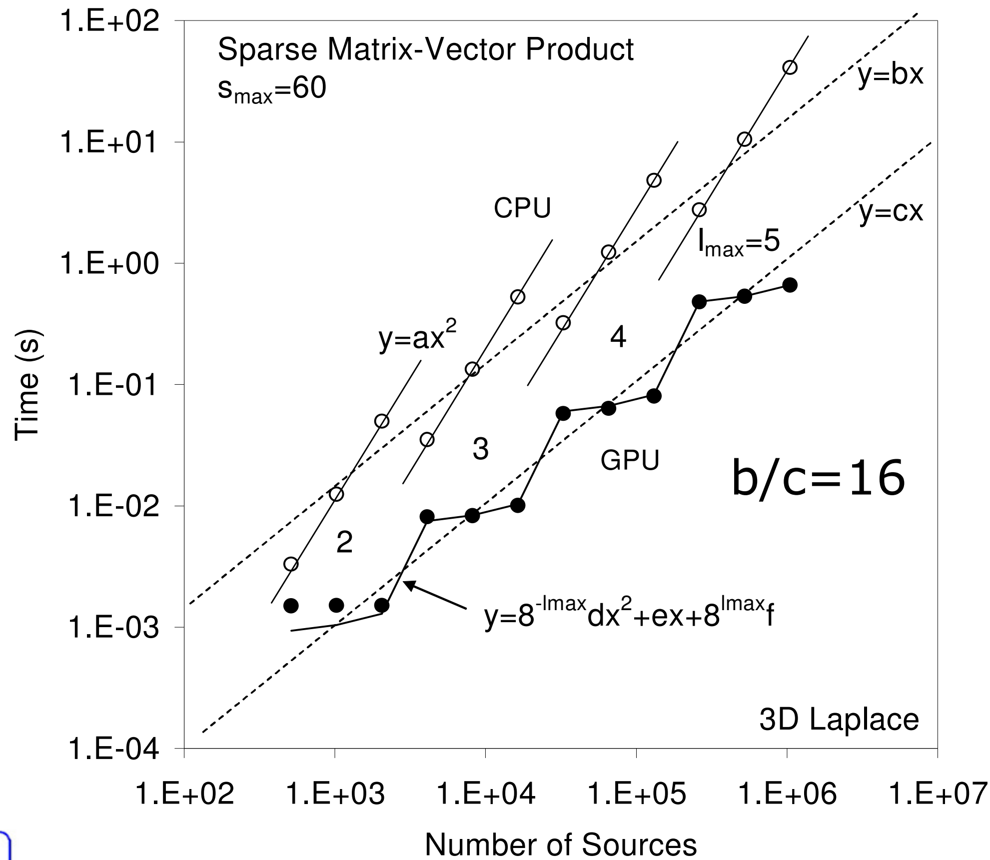
(simple execution of nested loop)

Algorithm (in words)

1. All source data, including coordinates and source intensities are stored in a single array in the global GPU memory (the size of the array is 4N for potential evaluation and 7N for potential/force evaluation). Two more large arrays reside in the global memory, one with the receiver coordinates (size 3M), and the other is allocated for the result of the matrix vector product (M for potential computations and 4M for potential and force computations). In case the sources and receivers are the same, as in particle dynamics computations, these can be reduced.
2. The routine is executed in CUDA on a one dimensional grid of one dimensional blocks of threads. By default each block contains 256 threads, which was determined via an empirical study of optimal thread-block size. This study also showed that for good performance the thread block grid should contain not less than 32 blocks for a 16 multiprocessor configuration. If the number of receivers is not a divisor of the block size, only the remaining number of threads is employed for computations of the last block.
3. Each thread in the block handles computation of one element of the output vector (or one receiver). For this purpose a register for potential calculation (and three registers for calculation of forces) are allocated and initialized to zero. The thread can take care of another receiver only after the entire block of threads is executed, threads are synchronized, and the shared memory can be overwritten.
4. Each thread reads the respective receiver coordinates directly from the global memory and puts them into the shared memory.
5. For a given block the source data are executed in a loop by batches of size B floats, where B depends on the type of the kernel (4 or 7 floats per source for monopole or monopole/dipole summations, respectively) and the size of the shared memory available. Currently we use 8 kB, or 2048 floats of the shared memory (so e.g. for monopole summation this determines B j 512). If the number of sources is not a divisor of B the last batch takes care of the remaining number of sources.
6. All threads of the block are employed to read source data for the given batch from global memory and put them into the shared memory (consequent reading: one thread per one float, until the entire batch is loaded followed by thread synchronization).
7. Each thread computes the product of one matrix element (kernel) with the source intensity (or in the case of dipoles the dipole moment with the kernel) and sums it up with the content of the respective summation register.
8. After summation of contributions of all sources (all batches) each thread writes the sum from its register into the global memory.

Direct summation on GPU (final step in the FMM)

Computations of potential, optimal settings for CPU



CPU:

$$\text{Time}=CNs,\ s=8^{-lmax}N$$

GPU:

$$\text{Time}=A_1\ N+B_1\ N/s+C_1\ Ns$$

read/write

access to box data

float computations

These parameters depend on the hardware

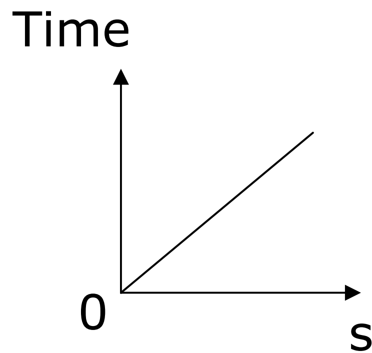
Algorithm modification

1. Each block of threads executes computations for one receiver box (Block-per-box parallelization).
2. Each block of threads accesses the data structure.
3. Loop over all sources contributing to the result requires partially non-coalesced read of the source data (box by box).

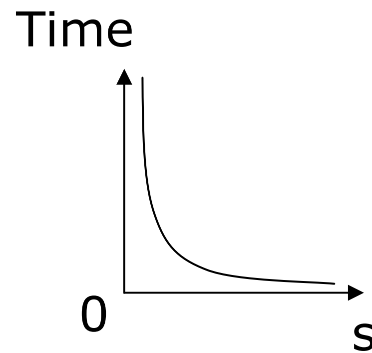
Optimization

Sparse M-V product   Dense M-V product

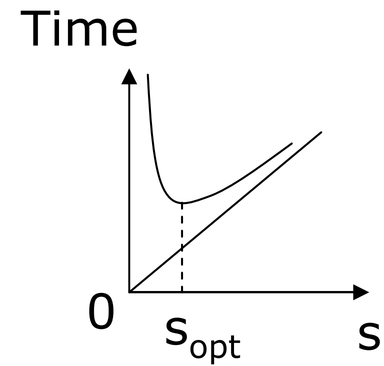
CPU:



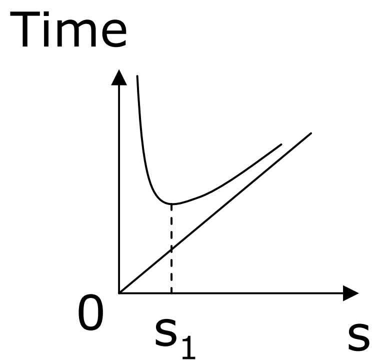
+



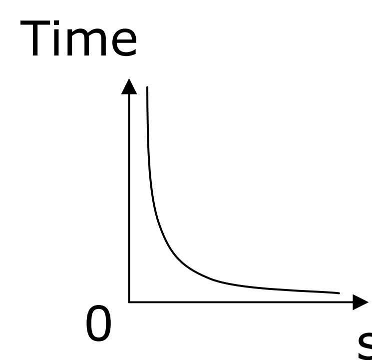
=



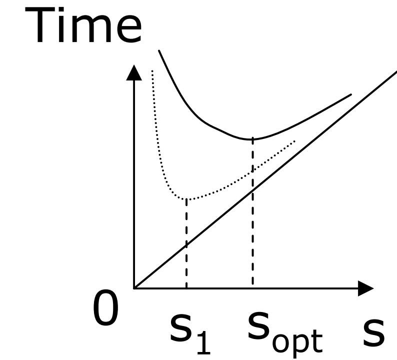
GPU:



+



=



Direct summation on GPU (final step in the FMM)

Compare GPU final summation complexity:

$$\text{Cost} = A_1 N + B_1 N/s + C_1 Ns.$$

and total FMM complexity:

$$\text{Cost} = AN + BN/s + CNs.$$

Optimal cluster size for direct summation step of the FMM

$$s_{opt} = (B_1 / C_1)^{1/2},$$

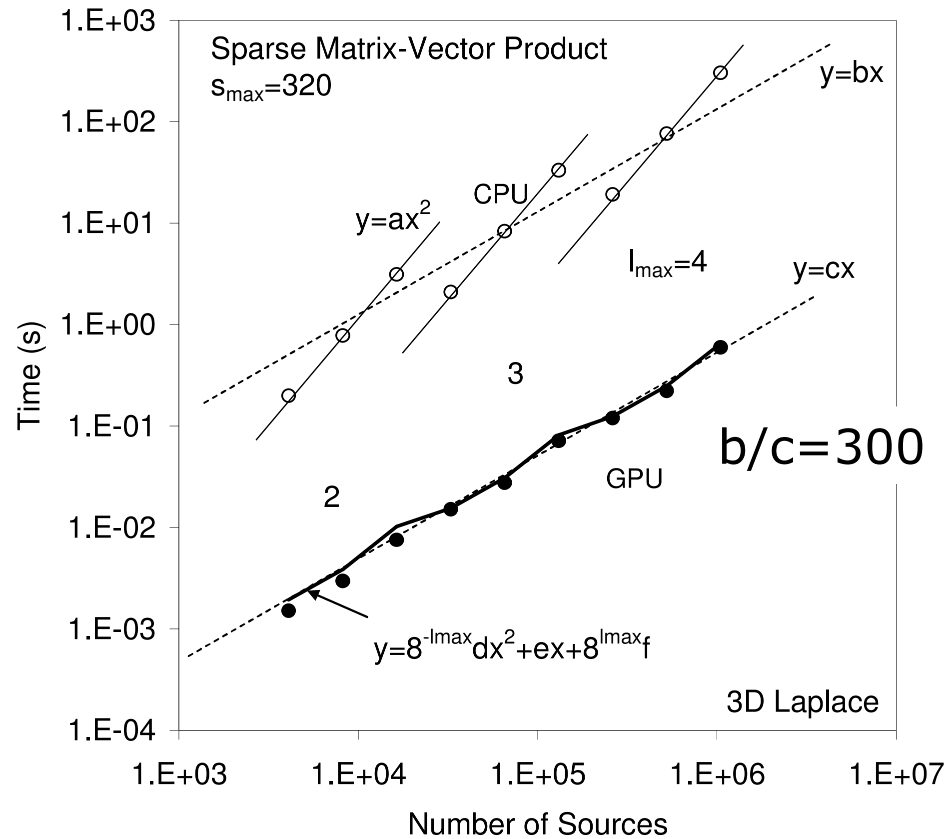
and this can be only increased for the full algorithm, since its complexity

$$\text{Cost} = (A + A_1)N + (B + B_1)N/s + C_1 Ns,$$

$$\text{and } s_{opt} = ((B + B_1)/C_1)^{1/2}.$$

Direct summation on GPU (final step in the FMM)

Computations of potential, optimal settings for GPU



Direct summation on GPU (final step in the FMM)

Computations of potential, optimal settings for CPU and GPU

N	Serial CPU (s)	GPU(s)	Time Ratio
4096	3.51E-02	1.50E-03	23
8192	1.34E-01	2.96E-03	45
16384	5.26E-01	7.50E-03	70
32768	3.22E-01	1.51E-02	21
65536	1.23E+00	2.75E-02	45
131072	4.81E+00	7.13E-02	68
262144	2.75E+00	1.20E-01	23
524288	1.05E+01	2.21E-01	47
1048576	4.10E+01	5.96E-01	69

Important conclusion:

Since the optimal max level of the octree when using GPU is lesser than that for the CPU, the importance of optimization of translation subroutines diminishes.

Other steps of the FMM on GPU

- Accelerations in range 5-60;
- Effective accelerations for N=1,048,576 (taking into account max level reduction): 30-60.

Profile

Table 8
Comparison of optimal CPU and GPU FMM: $N = 1,048,576$

	S exp	Up trans	Down trans	R eval	Sparse	Total
$p = 4$, CPU: $l_{max} = 6, \epsilon_2 = 2.3 \times 10^{-4}$, GPU: $l_{max} = 4, \epsilon_2 = 2.3 \times 10^{-4}$						
Serial CPU (s)	0.34	0.19	15.06	0.34	6.31	22.25
GPU (s)	0.011	0.00046	0.061	0.011	0.60	0.6835
Speedup (times)	31	413	247	31	11	32.6
$p = 8$, CPU: $l_{max} = 5, \epsilon_2 = 8.8 \times 10^{-6}$, GPU: $l_{max} = 4, \epsilon_2 = 8.3 \times 10^{-6}$						
Serial CPU (s)	0.83	0.12	8.42	0.85	40.95	51.17
GPU (s)	0.020	0.00218	0.265	0.021	0.60	0.9082
Speedup (times)	42	55	32	40	68	56.3
$p = 12$, CPU: $l_{max} = 5, \epsilon_2 = 1.3 \times 10^{-6}$, GPU: $l_{max} = 4, \epsilon_2 = 9.5 \times 10^{-7}$						
Serial CPU (s)	1.91	0.33	21.30	1.93	40.95	66.56
GPU (s)	0.040	0.00562	0.72	0.030	0.59	1.395
Speedup (times)	48	59	30	67	69	47.7

Accuracy

Relative L_2 –norm error measure:

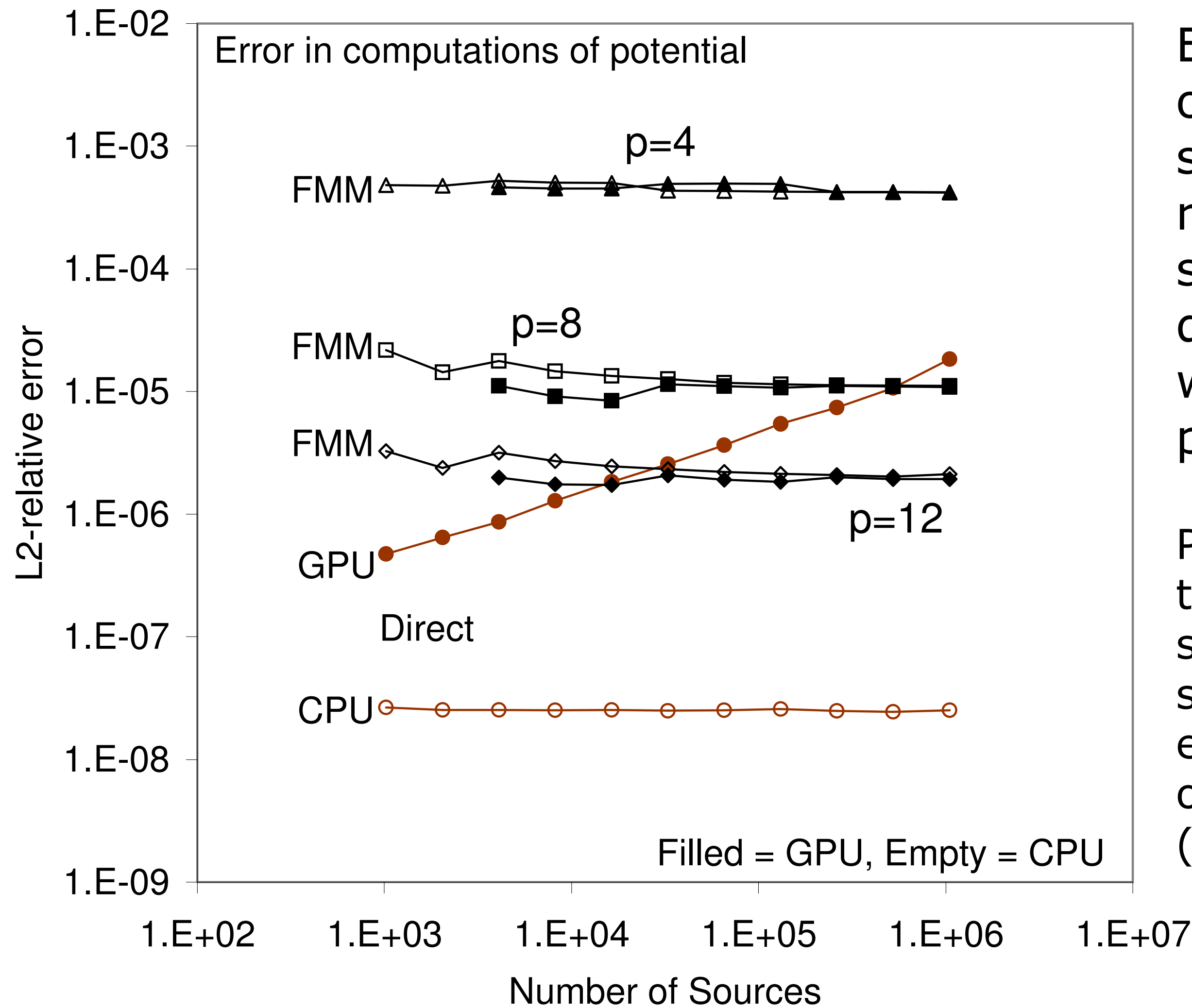
$$\epsilon_2 = \frac{\epsilon_2^{(abs)}}{\|\phi_{exact}(\mathbf{y})\|_2},$$

$$\epsilon_2^{(abs)} = \left[\frac{1}{M} \sum_{j=1}^{M} \left| \phi_{exact}\left(\mathbf{y}_j\right) - \phi_{approx}\left(\mathbf{y}_j\right) \right|^2 \right]^{1/2},$$

$$\|\phi_{exact}(\mathbf{y})\|_2 = \left[\frac{1}{M} \sum_{j=1}^{M} \left| \phi_{exact}\left(\mathbf{y}_j\right) \right|^2 \right]^{1/2}.$$

CPU single precision direct summation was taken as “exact”;
100 sampling points were used.

What is more accurate for solution of large problems on GPU: direct summation or FMM?



Error computed over a grid of 729 sampling points, relative to "exact" solution, which is direct summation with double precision.

Possible reason why the GPU error in direct summation grows: systematic roundoff error in computation of function 1/sqrt(x). (still a question).

Accelerated by Fantalgo

Performance

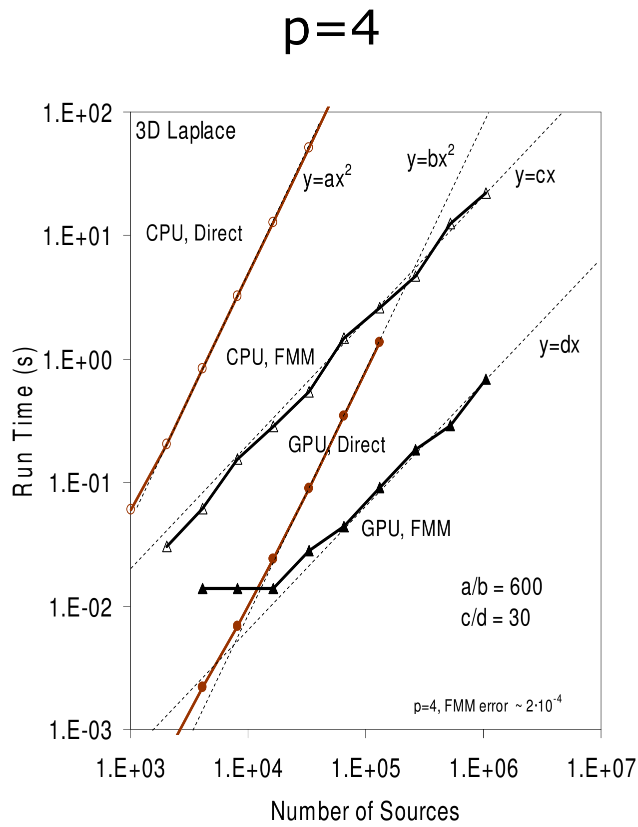
N=1,048,576 (potential only)

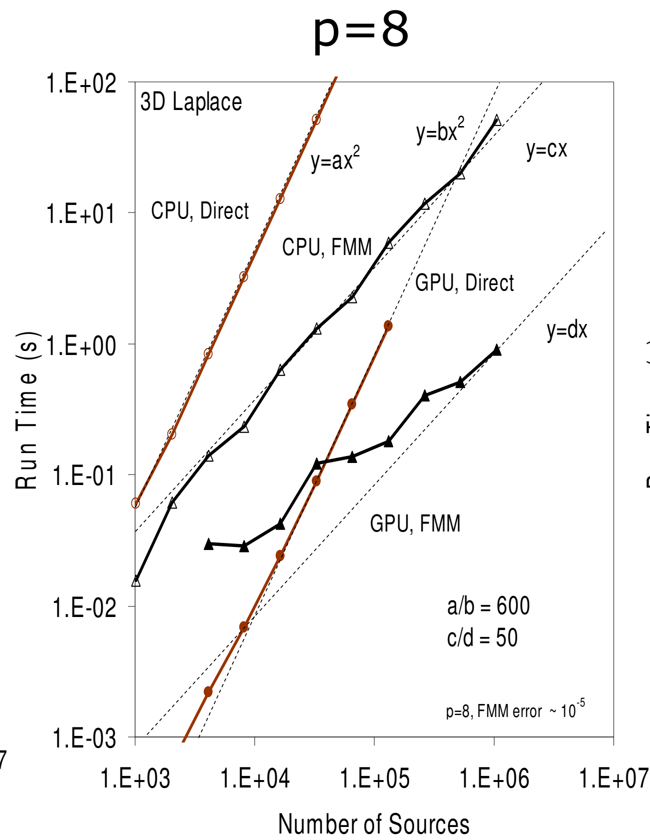
	serial CPU	GPU	Ratio
p=4	22.25 s	0.683 s	33
p=8	51.17 s	0.908 s	56
p=12	66.56 s	1.395 s	48

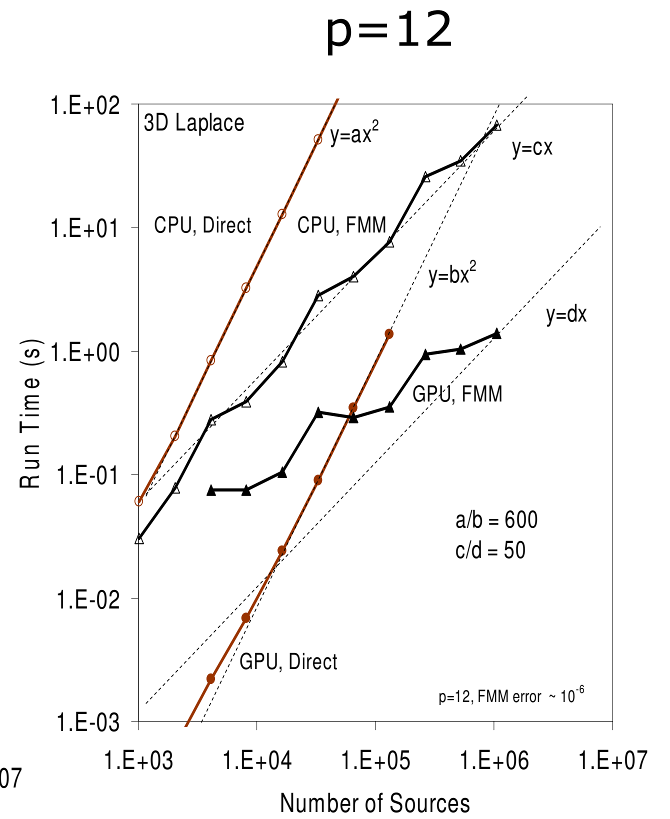
N=1,048,576 (potential+forces (gradient))

	serial CPU	GPU	Ratio
p=4	28.37 s	0.979 s	29
p=8	88.09 s	1.227 s	72
p=12	116.1 s	1.761 s	66

Performance







Performance

Computations of the potential and forces:

Peak performance of GPU for direct summation 290 Gigaflops, while for the FMM on GPU effective rates in range 25-50 Teraflops are observed (following the citation below).

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.