CMSC 858M/AMSC 698R

Fast Multipole Methods

Nail A. Gumerov & Ramani Duraiswami

Lecture 20

Outline

- Two parts of the FMM
- Data Structures
- FMM Cost/Optimization on CPU
- Fine Grain Parallelization for Multicore Nodes
- Graphics Processors (GPUs)
- Fast Matrix-Vector Multiplication on GPU with On-Core Computable Kernel
- Change of the Cost/Optimization scheme on GPU compared to CPU
- Other steps
- Test Results
  - Profile
  - Accuracy
  - Overall Performance
- Heterogeneous algorithm

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.
- Q. Hu, N.A. Gumerov, and R. Duraiswami
  - Scalable Fast Multipole Methods on Distributed Heterogeneous Architectures, Proceedings of the Supercomputing'11 Conference, Seattle, November 12-18, 2011. (The paper is nominated for the best student paper award).

Two parts of the FMM

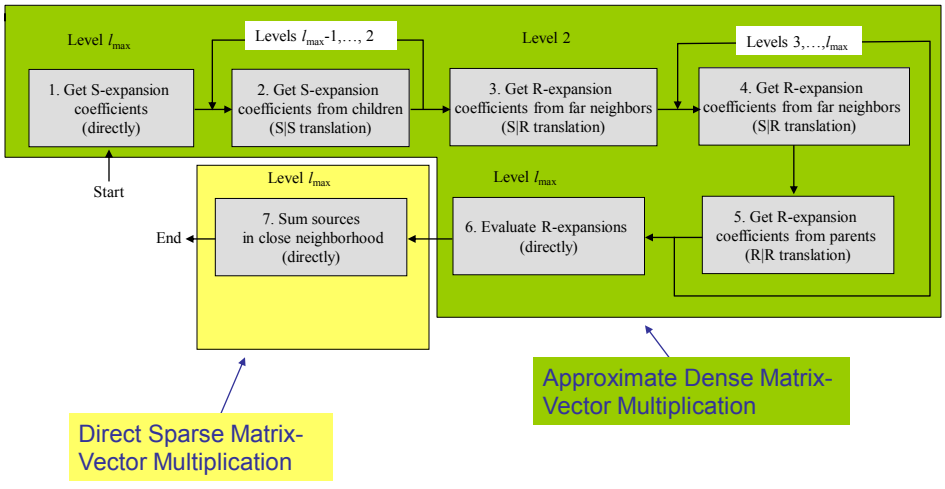
- Generate data structure (define the matrix);
- Perform Matrix-vector product (run);
- Two different types of problems:
  - Static matrix (e.g. for iterative solution of large linear system):
    - Data structure should be generated once, while run should be performed many times with different input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

Data Structures

- 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.
- The most recent work on data structures on GPU is performed by Hu Qi (up to 100 times speed up compared to the standard CPU).
- In the present lecture we will focus on the run algorithm, assuming that the necessary lists are generated and stored in the global memory.

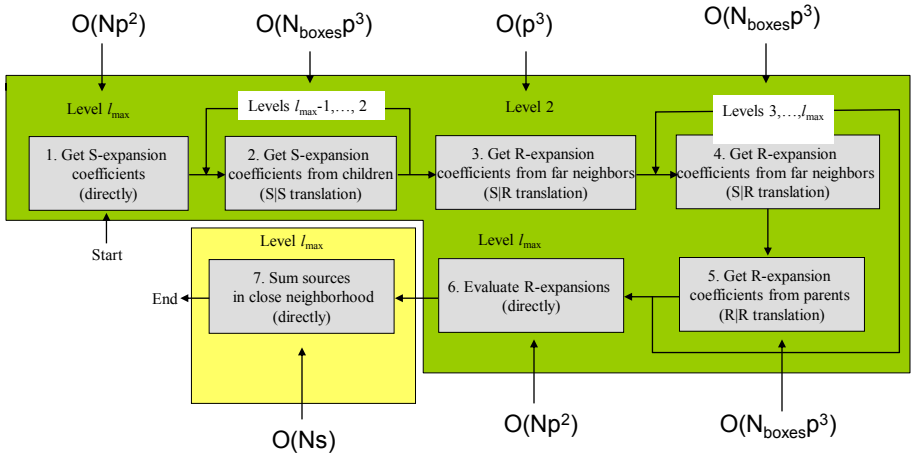
FMM on CPU

Flow Chart of FMM Run



FMM Cost (Operations)

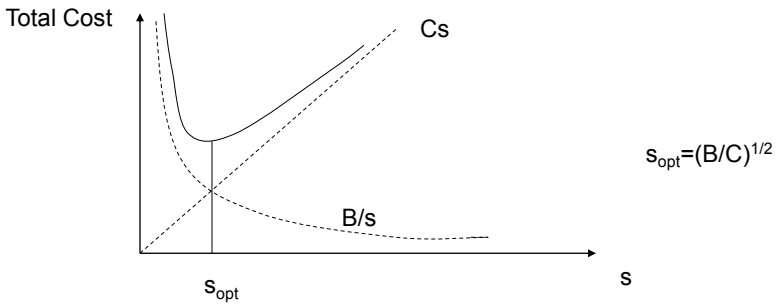
Points are clustered with clustering parameter *s*, means not more that *s* points in the smallest octree box. Expansions have length p^2 (number of coefficients). There are O(N) points and $O(N_{boxes})=O(N/s)$ in the octree.



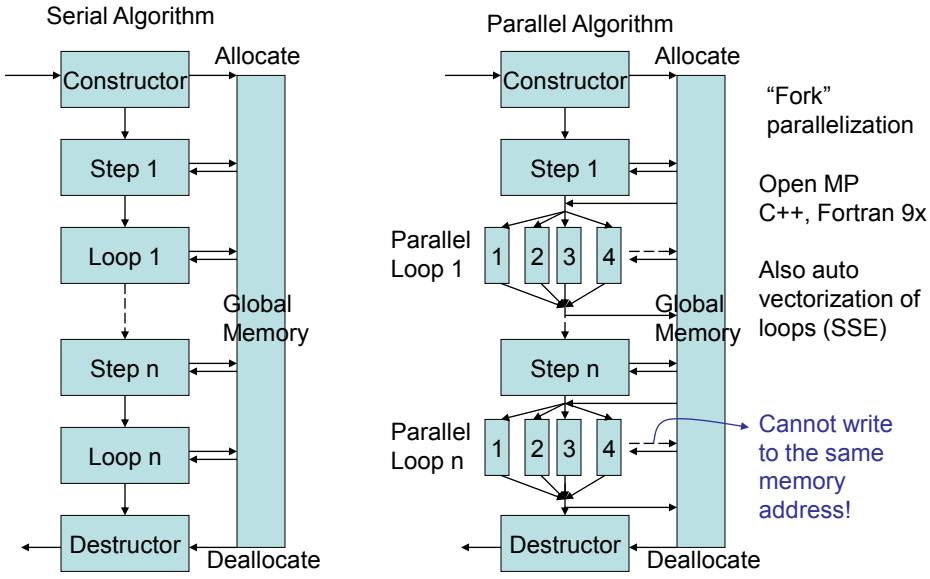
FMM Cost/Optimization (operations)

Since p=O(1), we have

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



Fine Grain Parallelization for Multicore PC's



Some test results for OMP parallelization (4 cores)

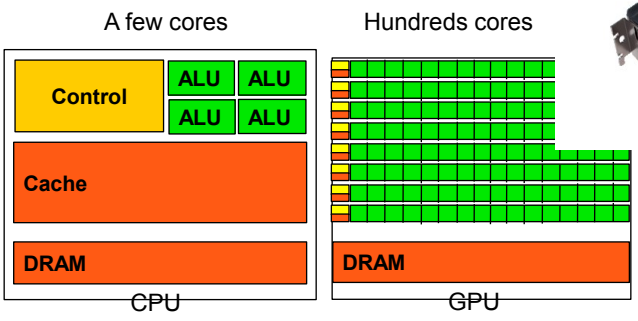
Table 1. Comparison of profiles of serial and parallel (OMP) FMM implementations at $N = 1,048,576$.

□	S-expansions	Up–tree translations	Down–tree translations	R-evaluations	Direct Summations	Total	Efficiency
$p = 4, l_{max} = 6, \epsilon_2 = 2.3 \cdot 10^{-4}$							1.0
Serial (s)	0.34	0.19	15.06	0.34	6.31	22.25	□
Parallel (s)	0.09	0.05	3.77	0.09	1.59	5.59	□
Speedup (times)	3.8	3.8	4.0	3.8	4.0	3.98	□
$p = 8, l_{max} = 5, \epsilon_2 = 8.8 \cdot 10^{-6}$							0.98
Serial (s)	0.83	0.12	8.42	0.85	40.95	51.17	□
Parallel (s)	0.23	0.03	2.17	0.22	10.47	13.12	□
Speedup (times)	3.6	4.0	3.9	3.9	3.9	3.90	□
$p = 12, l_{max} = 5, \epsilon_2 = 1.3 \cdot 10^{-6}$							0.98
Serial (s)	1.91	0.33	21.30	1.93	40.95	66.42	□
Parallel (s)	0.53	0.09	5.43	0.50	10.42	16.97	□
Speedup (times)	3.6	3.7	3.9	3.9	3.9	3.91	□

GPUs

A Quick Introduction to the GPU

- Graphics processing unit (GPU) is a highly parallel, multithreaded, many-core processor with high computation power and memory bandwidth
- GPU is designed for single instruction multi computation; more transistors for processin data caching and flow control

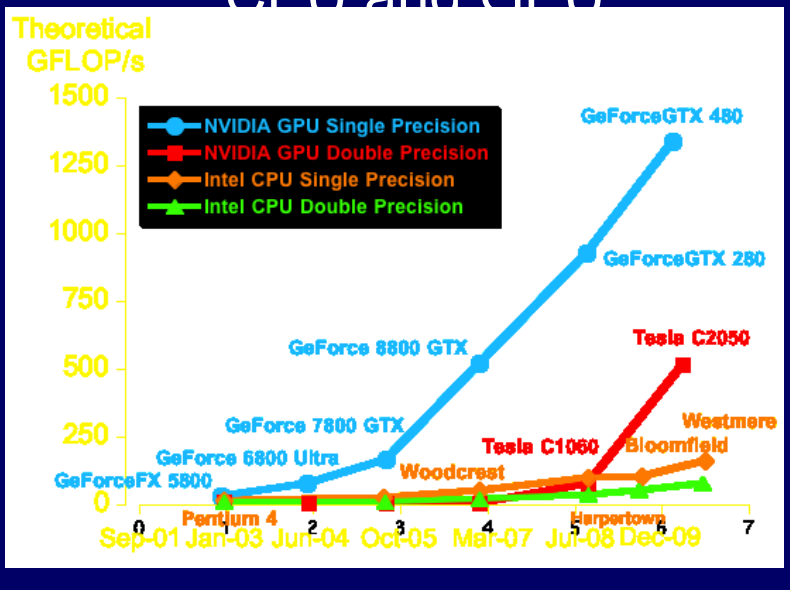


NVIDIA Tesla C2050:
1.25 Tflops single
0.52 Tflops double
448 cores

Is It Expensive?

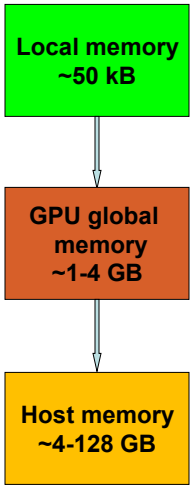
- Any PC has GPU which probably performs faster than the CPU
- GPUs with Teraflops performance are used in game stations
- Tens of millions of GPUs are produced each year
- Price for 1 good GPU in range $200-500
- Prices for the most advanced NVIDIA GPUs for general purpose computing (e.g. Tesla C2050) are in the range $1K-$2K
- Modern research supercomputer with several GPUs can be purchased for a few thousand dollars
- GPUs provide the best Gflops/$ ratio
- They also provide the best Gflops/watt

Floating-Point Operations for CPU and GPU



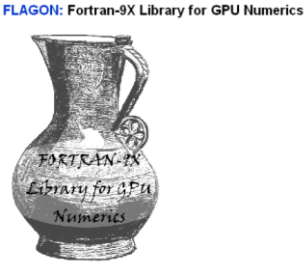
Is It Easy to Program A GPU ?

- For inexperienced GPU programmers
  - –Matlab Parallel Toolbox
- Middleware to program GPU from Fortran
  - –Relatively easy to incorporate to existing codes
  - –Developed by the authors at UMD
  - –Free (available online)
- For advanced users
  - –CUDA: a C-like programming language
  - –Math libraries are available
  - –Custom functions can be implemented
  - –Requires careful memory management
  - –Free (available online)



Availability

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



FMM on GPU

Challenges

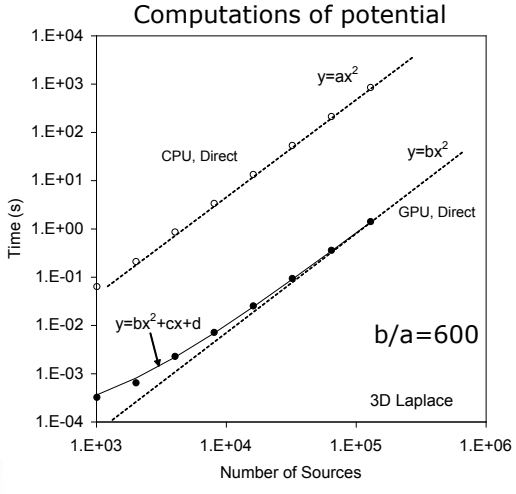
- Complex FMM data structure;
- Problem is not native for SIMD semantics;
  - non-uniformity of data causes problems with efficient work load (taking into account large number of threads);
  - serial algorithms use recursive computations;
  - existing libraries (CUBLAS) and middleware approach are not 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.

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

Fast Kernel Matrix-vector multiplication on GPU

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).

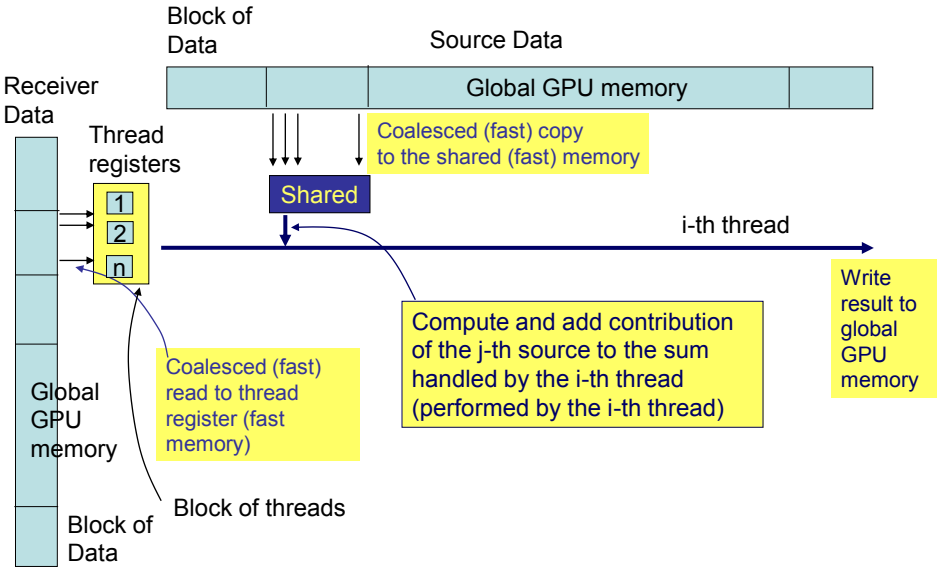
GPU: NVIDIA GeForce 8800 GTX (peak 330 GFLOPS).

Estimated achieved rate: 190 GFLOPS.

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

(simple execution of nested loop)

Algorithm for direct summation on GPU (Kernel matrix multiplication)



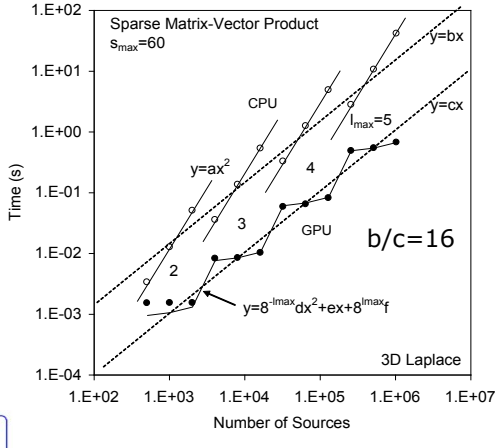
Change of Cost/Optimization on GPU

Algorithm modification for direct summation in the FMM

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).

Direct summation on GPU (final step in the FMM)

Computations of potential, optimal settings for CPU



CPU:

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

GPU:

Time=A_1 N+B_1 N/s+C_1 Ns

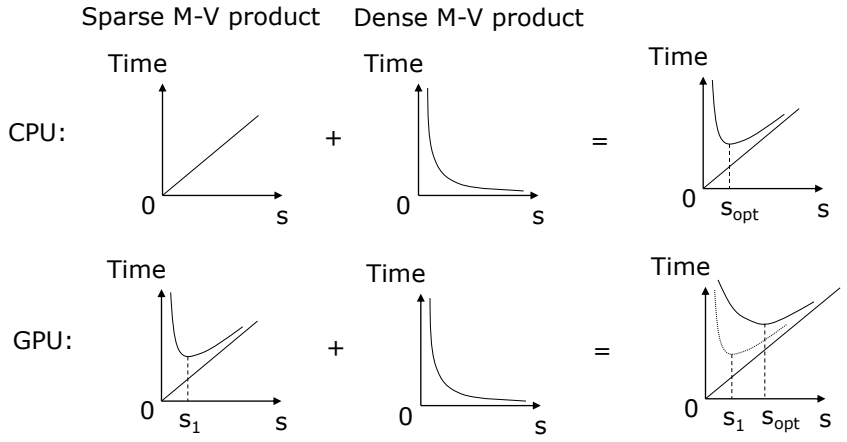
read/write

float computations

access to box data

These parameters depend on the hardware

Optimization



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

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

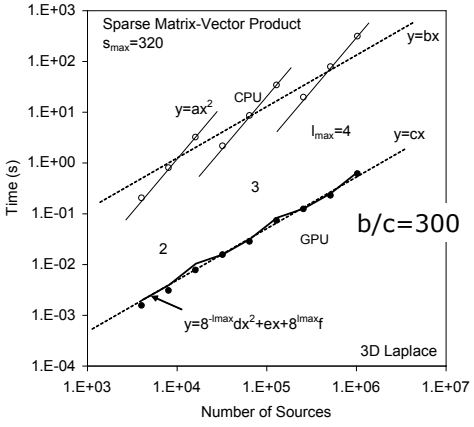
and this can be only increased for the full algorithm, since its complexity

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

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

Other steps of the FMM

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.

Test results

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:

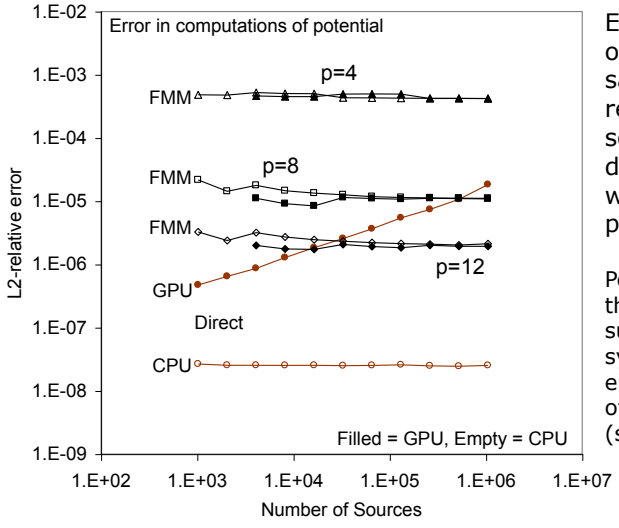
$$\epsilon_2 = \frac{\epsilon_2^{(abs)}}{\| \phi_{exact}(\mathbf{y}) \|_2},$$

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

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

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).

Performance

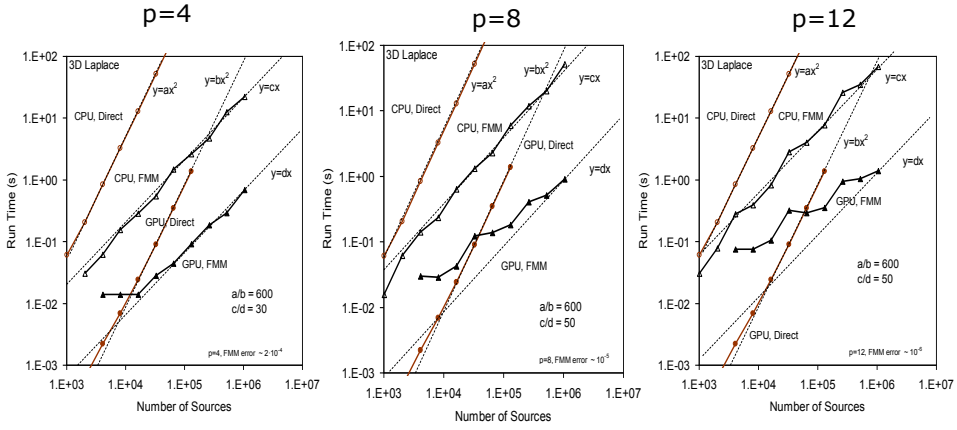
N=1,048,576 (potential only)

	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

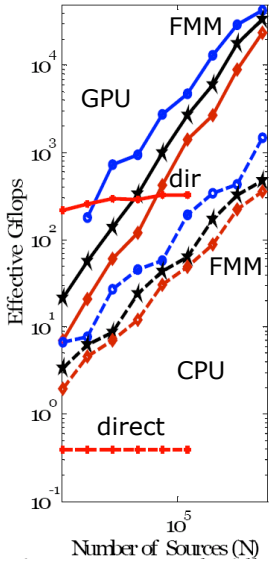


Performance

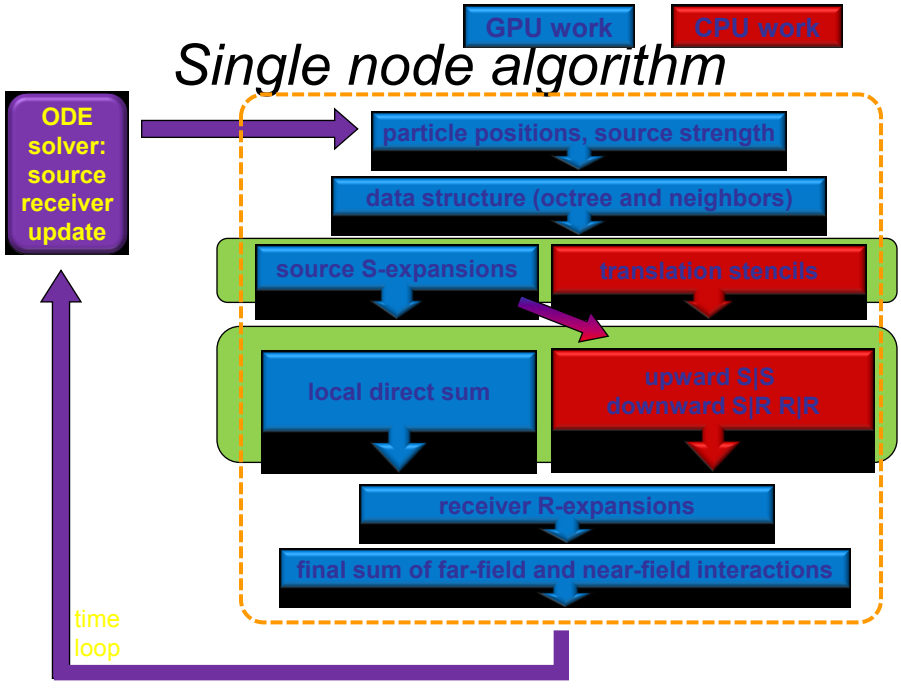
Computations of the potential and forces:

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.



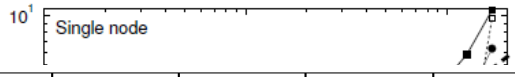
Heterogeneous Algorithm



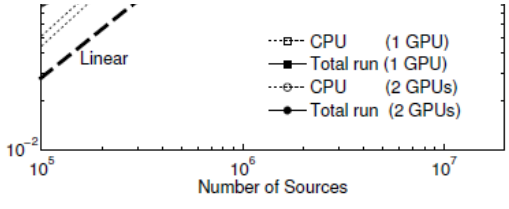
Advantages

- CPU and GPU are tasked with the most efficient jobs
  - Faster translations: CPU code can be better optimized which requires complex translation stencil data structures
  - Faster local direct sum: many cores on GPU; same kernel evaluation but on multiple data (SIMD)
- The CPU is not idle during the GPU-based computations
- High accuracy translation without much cost penalty on CPU
- Easy visualization: all data reside in GPU RAM
- Smaller data transfer between the CPU and GPU

Single node tests



Time (s) \ N	1,048,576		2,097,152		4,194,304		8,388,608		16,777,216	
Num of GPUs	1	2	1	2	1	2	1	2	1	2
CPU wall clock	0.13	0.13	1.06	1.08	1.07	1.11	1.02	1.10	8.53	8.98
C/G parallel region	0.58	0.30	1.06	1.08	1.58	1.11	4.38	2.21	8.55	8.98
Force+Potential total run	0.71	0.39	1.23	1.22	1.96	1.34	5.11	2.63	10.3	10.1
Potential total run	0.40	0.24	1.16	0.89	1.27	1.25	2.94	1.52	9.76	6.30
Partitioning	–	0.14	–	0.32	–	0.58	–	1.14	–	3.09



The billion size test case

- Using all 32 Chimera nodes
- Timing excludes the partition
- 1 billion runs potential kernel in 21.6 s

Actual performance: 10-30 TFs

To compute without FMM for the same time:

1 EFs performance