Parallel FMM

Matthew Knepley

Computation Institute
University of Chicago

Department of Molecular Biology and Physiology
Rush University Medical Center

Conference on High Performance Scientific Computing
In Honor of Ahmed Sameh’s 70th Birthday
Purdue University, October 11, 2010
Using estimates and proofs, a simple software architecture, gets good scaling, efficiency, and adaptive load balance.
Using estimates and proofs, a simple software architecture, gets good scaling, efficiency, and adaptive load balance.
Using estimates and proofs, a simple software architecture, gets good scaling, efficiency, and adaptive load balance.
The **PetFMM team:**

- **Prof. Lorena Barba**
  - Dept. of Mechanical Engineering, Boston University

- **Dr. Felipe Cruz**, developer of GPU extension
  - Nagasaki Advanced Computing Center, Nagasaki University

- **Dr. Rio Yokota**, developer of 3D extension
  - Dept. of Mechanical Engineering, Boston University
Collaborators

Chicago Automated Scientific Computing Group:

- **Prof. Ridgway Scott**
  - Dept. of Computer Science, University of Chicago
  - Dept. of Mathematics, University of Chicago

- **Peter Brune**, (biological DFT)
  - Dept. of Computer Science, University of Chicago

- **Dr. Andy Terrel**, (Rheagen)
  - Dept. of Computer Science and TACC, University of Texas at Austin
FMM Work

- Queue-based hybrid execution
  - OpenMP for multicore processors
  - CUDA for GPUs

- Adaptive hybrid Treecode-FMM
  - Treecode competitive only for very low accuracy
  - Very high flop rates for treecode M2P operation

- Computation/Communication Overlap FMM
  - Provably scalable formulation
  - Overlap P2P with M2L
Outline

1. Complementary Work
2. Short Introduction to FMM
3. Parallelism
4. What Changes on a GPU?
5. PetFMM
FMM Applications

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity
FMM Applications

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity

Advantages

- Mesh-free
- $O(N)$ time
- Distributed and multicore (GPU) parallelism
- Small memory bandwidth requirement
FMM accelerates the calculation of the function:

$$\Phi(x_i) = \sum_j K(x_i, x_j)q(x_j)$$  \hspace{1cm} (1)

- Accelerates $O(N^2)$ to $O(N)$ time

- The kernel $K(x_i, x_j)$ must decay quickly from $(x_i, x_i)$
  - Can be singular on the diagonal (Calderón-Zygmund operator)

- Discovered by Leslie Greengard and Vladimir Rohklin in 1987

- Very similar to recent wavelet techniques
FMM accelerates the calculation of the function:

$$\Phi(x_i) = \sum_j \frac{q_j}{|x_i - x_j|}$$  \hspace{1cm} (1)

- Accelerates $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ time
- The kernel $K(x_i, x_j)$ must decay quickly from $(x_i, x_i)$
  - Can be singular on the diagonal (Calderón-Zygmund operator)
- Discovered by Leslie Greengard and Vladimir Rohklin in 1987
- Very similar to recent wavelet techniques
Pairs of boxes are divided into near and far:
Pairs of boxes are divided into *near* and *far*:

Neighbors are treated as *very near*. 
Create Multipole Expansions.
Evaluate Local Expansions.

Upward Sweep
Downward Sweep

P2M  M2M  M2L
L2L  L2P
Outline

1. Complementary Work
2. Short Introduction to FMM
3. Parallelism
4. What Changes on a GPU?
5. PetFMM
The Quadtree is a Sieve with optimized operations.

- Multipoles are stored in Sections.
- Two Overlaps are defined:
  - Neighbors
  - Interaction List
- Completion moves data for:
  - Neighbors
  - Interaction List
The Quadtree is a Sieve with optimized operations.

- Multipoles are stored in Sections.
- Two Overlaps are defined:
  - Neighbors
  - Interaction List
- Completion moves data for:
  - Neighbors
  - Interaction List
FMM in Sieve

- The Quadtree is a Sieve with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List
The Quadtree is a Sieve with optimized operations.

Multipoles are stored in Sections.

Two Overlaps are defined:
  - Neighbors
  - Interaction List

Completion moves data for
  - Neighbors
  - Interaction List
Parallelism

FMM in Sieve

- The Quadtree is a Sieve with optimized operations
- Multipoles are stored in Sections
- Two Overlaps are defined
  - Neighbors
  - Interaction List
- Completion moves data for
  - Neighbors
  - Interaction List
The Quadtree is a **Sieve** with optimized operations

- Multipoles are stored in **Sections**
- Two **Overlaps** are defined
  - Neighbors
  - Interaction List

Completion moves data for
- Neighbors
- Interaction List
Parallelism

FMM in Sieve

- The Quadtrees are a Sieve with optimized operations.
- Multipoles are stored in Sections.
- Two Overlaps are defined:
  - Neighbors
  - Interaction List
- Completion moves data for:
  - Neighbors
  - Interaction List
FMM Control Flow

Kernel operations will map to GPU tasks.
Kernel operations will map to GPU tasks.
Parallel Tree Implementation

- Divide tree into a root and local trees
- Distribute local trees among processes
- Provide communication pattern for local sections (overlap)
  - Both neighbor and interaction list overlaps
  - Sieve generates MPI from high level description
Parallel Tree Implementation

How should we distribute trees?

- Multiple local trees per process allows good load balance
- Partition weighted graph
  - Minimize load imbalance and communication
  - Computation estimate:
    - Leaf \( N_i p \) (P2M) + \( n_i p^2 \) (M2L) + \( N_i p \) (L2P) + \( 3^d N_i^2 \) (P2P)
    - Interior \( n_c p^2 \) (M2M) + \( n_i p^2 \) (M2L) + \( n_c p^2 \) (L2L)
- Communication estimate:
  - Diagonal \( n_c (L - k - 1) \)
  - Lateral \( 2^d \frac{2^m (L - k - 1) - 1}{2^m - 1} \) for incidence dimension \( m \)
- Leverage existing work on graph partitioning
  - ParMetis

- Good partitions exist for non-uniform distributions
  - 2D: $\mathcal{O}(\sqrt{n}(\log n)^{3/2})$ edgecut
  - 3D: $\mathcal{O}(n^{2/3}(\log n)^{4/3})$ edgecut

- As scalable as regular grids

- As efficient as uniform distributions

- ParMetis will find a nearly optimal partition

- Good partitions exist for non-uniform distributions
  - 2D $C_i = 1.24^i C_0$ for random matching
  - 3D $C_i = 1.21^i C_0$ for random matching

- 3D proof needs assurance that average degree does not increase

- Efficient in practice
Parallelism

Parallel Tree Implementation

Advantages

- Simplicity
  - Complete serial code reuse
  - Provably good performance and scalability
Parallel Tree Implementation

Advantages

- Simplicity
- Complete serial code reuse
- Provably good performance and scalability
Parallelism

Parallel Tree Implementation

Advantages

- Simplicity
- Complete serial code reuse
- Provably good performance and scalability
Distributing Local Trees

The interaction of locals trees is represented by a weighted graph.

This graph is partitioned, and trees assigned to processes.
Here local trees are assigned to processes:
Parallel Data Movement

1. **Complete neighbor section**

2. **Upward sweep**
   - 1. Upward sweep on local trees
   - 2. Gather to root tree
   - 3. Upward sweep on root tree

3. **Complete interaction list section**

4. **Downward sweep**
   - 1. Downward sweep on root tree
   - 2. Scatter to local trees
   - 3. Downward sweep on local trees
Local Tree Distribution

Here local trees are assigned to processes for a spiral distribution:

(a) 2 cores
(b) 4 cores
Here local trees are assigned to processes for a spiral distribution:

(c) 8 cores

(d) 16 cores
Here local trees are assigned to processes for a spiral distribution:

(e) 32 cores

(f) 64 cores
What Changes on a GPU?
What Changes on a GPU?

Multipole-to-Local Transformation

Re-expands a multipole series as a Taylor series

- Up to 85% of time in FMM
  - Tradeoff with direct interaction
- Dense matrix multiplication
  - $2p^2$ rows
- Each interaction list box
  - $(6^d - 3^d) \cdot 2^{dL}$
  - $d = 2, L = 8$
    - 1,769,472 matvecs
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- \( p = 12 \)
- Matrix size is 2304 bytes
- Plenty of work per thread (81 Kflops or 36 flops/byte)
- **BUT**, 16K shared memory only holds 7 matrices
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- $p = 12$
- Matrix size is 2304 bytes
- Plenty of work per thread (81 Kflops or 36 flops/byte)
- **BUT**, 16K shared memory only holds 7 matrices
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- \( p = 12 \)
- Matrix size is 2304 bytes
  - Plenty of work per thread (81 Kflops or 36 flops/byte)
  - **BUT**, 16K shared memory only holds 7 matrices
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- $p = 12$
- Matrix size is 2304 bytes
- Plenty of work per thread (81 Kflops or 36 flops/byte)

**BUT**, 16K shared memory only holds 7 matrices
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- $p = 12$
- Matrix size is 2304 bytes
- Plenty of work per thread (81 Kflops or 36 flops/byte)
- **BUT**, 16K shared memory only holds 7 matrices
One thread per M2L transform

- Thread block (TB) transforms one Multipole Expansion (ME) for each Interaction List (IL) box — 27 times
- $p = 12$
- Matrix size is 2304 bytes
- Plenty of work per thread (81 Kflops or 36 flops/byte)
- **BUT**, 16K shared memory only holds 7 matrices

Memory limits concurrency!
Apply M2L transform matrix-free

\[ m_{2l}^{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  \hspace{1cm} (2)

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \( 27 \times 8 = 216 \) threads, **BUT** max is 512
Apply M2L transform matrix-free

\[ m_{2l_{ij}} = -1^i \binom{i+j}{j} t^{i-j-1} \]  

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \(27 \times 8 = 216\) threads, **BUT** max is 512
Apply M2L transform matrix-free

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (2)

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \(27 \times 8 = 216\) threads, **BUT** max is 512

Algorithm limits concurrency!
Apply M2L transform matrix-free

\[ m_{2L_{ij}} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \( 27 \times 8 = 216 \) threads, **BUT** max is 512
Apply M2L transform matrix-free

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \(27 \times 8 = 216\) threads, **BUT** max is 512

20 GFlops

5x Speedup of Downward Sweep
GPU M2L
Version 1

Apply M2L transform matrix-free

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (2)

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)
- \( 27 \times 8 = 216 \) threads, \textbf{BUT} max is 512

20 GFlops
5x Speedup of Downward Sweep

Algorithm limits concurrency!
Apply M2L transform matrix-free

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  

(2)

Additional problems: Not enough parallelism for data movement

- Move 27 LE to global memory per TB
- \( 27 \times 2p = 648 \) floats
- With 32 threads, takes 21 memory transactions
One thread per *element* of the LE

\[ m2l_{ij} = -1^i \binom{i + j}{j} t^{-i-j-1} \] (3)

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t^{-i-1} \)
  - All threads loop to \( p+1 \), only store \( t^{-i-1} \)
- Loop unrolling
- No thread synchronization
One thread per *element* of the LE

\[ m_{2l_{ij}} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t^{-i-1} \)
  - All threads loop to \( p + 1 \), only store \( t^{-i-1} \)
- Loop unrolling
- No thread synchronization

\[ \text{M2L} \quad \text{ME} = \text{LE} \]
One thread per *element* of the LE

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (3)

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t^{-i-1} \)
  - **All** threads loop to \( p + 1 \), only *store* \( t^{-i-1} \)
- Loop unrolling
- No thread synchronization
One thread per *element* of the LE

\[ m_{2lij} = -1^i(i + j) t_{i-j-1} \]  

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t_{-i-1} \)
  - **All** threads loop to \( p + 1 \), only **store** \( t_{-i-1} \)
- Loop unrolling
- No thread synchronization
One thread per \textit{element} of the LE

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \]  

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t^{-i-1} \)
  - \textbf{All} threads loop to \( p + 1 \), only \textbf{store} \( t^{-i-1} \)
- Loop unrolling
- No thread synchronization

300 GFlops

15x Speedup of Downward Sweep
One thread per *element* of the LE

\[
m2l_{ij} = -1^i \binom{i + j}{j} t^{-i-j-1}
\]  

- Each thread does a dot product
- Cannot use diagonal traversal, more work
- Avoid branching
  - Each row precomputes \( t^{-i-1} \)
  - **All** threads loop to \( p + 1 \), only *store* \( t^{-i-1} \)
- Loop unrolling
- No thread synchronization

300 GFlops

15x Speedup of Downward Sweep

Examine memory access
Superior GPU memory bandwidth is due to both

**bus width** and **clock speed**.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Width (bits)</td>
<td>64</td>
<td>512</td>
</tr>
<tr>
<td>Bus Clock Speed (MHz)</td>
<td>400</td>
<td>1600</td>
</tr>
<tr>
<td>Memory Bandwidth (GB/s)</td>
<td>3</td>
<td>102</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>240</td>
<td>600</td>
</tr>
</tbody>
</table>

Tesla always accesses blocks of 64 or 128 bytes.
Coalesce and overlap memory accesses

Coalescing is

- a group of 16 threads
- accessing consecutive addresses
  - 4, 8, or 16 bytes
- in the same block of memory
  - 32, 64, or 128 bytes
Coalesce and overlap memory accesses
Memory accesses can be overlapped with computation when

- a TB is waiting for data from main memory
- another TB can be scheduled on the SM
- 512 TB can be active at once on Tesla
Coalesce and overlap memory accesses

Note that the theoretical peak (1 TF)
- MULT and FMA must execute simultaneously
- 346 GOps
- Without this, peak can be closer to 600 GF

480 GFlops
25x Speedup of Downward Sweep
M2L required all of these optimization steps:

- Many threads per kernel
- Avoid branching
- Unroll loops
- Coalesce memory accesses
- Overlap main memory access with computation
Outline

1. Complementary Work
2. Short Introduction to FMM
3. Parallelism
4. What Changes on a GPU?
5. PetFMM
PetFMM is an freely available implementation of the Fast Multipole Method
http://barbagroup.bu.edu/Barba_group/PetFMM.html

- Leverages PETSc
  - Same open source license
  - Uses Sieve for parallelism
- Extensible design in C++
  - Templated over the kernel
  - Templated over traversal for evaluation
- MPI implementation
  - Novel parallel strategy for anisotropic/sparse particle distributions
  - PetFMM–A dynamically load-balancing parallel fast multipole library
  - 86% efficient strong scaling on 64 procs
- Example application using the Vortex Method for fluids
- (coming soon) GPU implementation
PetFMM CPU Performance

Strong Scaling

Graph showing the speedup of different data layouts and load histograms on the PetFMM benchmark. The x-axis represents the number of processors, and the y-axis represents the speedup. Different line styles and markers represent various data layouts and load histograms:

- uniform 4ML8R5
- uniform 10ML9R5
- spiral 1ML8R5
- spiral w/ space-filling 1ML8R5
- Perfect Speedup
PetFMM CPU Performance

Strong Scaling

- ME Initialization
- Upward Sweep
- Downward Sweep
- Evaluation
- Load balancing stage
- Total time

Time [sec]
Number of processors

M. Knepley (UC)
Largest Calculation With Development Code

- 10,648 randomly oriented lysozyme molecules
- 102,486 boundary elements/molecule
- More than 1 billion unknowns
- 1 minute on 512 GPUs
10,648 randomly oriented lysozyme molecules
102,486 boundary elements/molecule
More than 1 billion unknowns
1 minute on 512 GPUs
What do we need for Parallel FMM?

- Urgent need for reduction in complexity
  - Complete serial code reuse
  - Modeling integral to optimization

- Unstructured communication
  - Uses optimization to automatically generate
  - Provided by ParMetis and PETSc

- Massive concurrency is necessary
  - Mix of vector and thread paradigms
  - Demands new analysis