Tree-based methods on GPUs

Felipe Cruz[1] and Matthew Knepley[2,3]

[1]Department of Mathematics
University of Bristol

[2]Computation Institute
University of Chicago

[3]Department of Molecular Biology and Physiology
Rush University Medical Center

International Workshop on
Modern Computational Geoscience Frontiers
GUCAS, Beijing July 1, 2009

Outline

FMM Applications for Geoscience

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity

FMM Applications for Geoscience

FMM can accelerate both integral and boundary element methods for:

- Laplace
- Stokes
- Elasticity

Advantages

- Mesh-free
- $\mathcal{O}(N)$ time
- GPU and distributed parallelism
- Memory is greatly reduced in 3D for BEM

FMM Applications for Geoscience

Constant coefficient versions can precondition full equations:

- Work by Dave May at ETH
  - Solve Stokes
  - Scale identity by viscosity magnitude
- Advantages over MG
  - No grids have to be created
  - No iterative problems

Stokes Flow
Vorticity Formulation

In vorticity form, the Stokes equation conserves vorticity

$$\frac{\partial \omega}{\partial t} + u \cdot \nabla \omega = \frac{\mathrm{D}\omega}{\mathrm{D}t} = 0$$

and we can recover the velocity using the Biot-Savart law

$$\begin{aligned} u(x,t) &= \int (\nabla \times \mathbb{G})(x - x')\omega(x',t)dx' \\ &= \int \mathbb{K}(x - x')\omega(x',t)dx' = (\mathbb{K} * \omega)(x,t) \end{aligned}$$

where $\mathbb{G}$ is the Green function for the Poisson equation.

Stokes Flow
RBF Expansion

We expand the vorticity

$$\omega(x,t) \approx \omega_\sigma(x,t) = \sum_i^N \gamma_i \zeta_\sigma(x, x_i)$$

in a basis of radial functions

$$\zeta_\sigma(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{-|x-y|^2}{2\sigma^2}\right)$$

resulting in the following kernel

$$\mathbb{K}_\sigma(x) = \frac{1}{2\pi|x|^2}(-x_2, x_1)\left(1 - \exp\left(-\frac{|x|^2}{2\sigma^2}\right)\right).$$

Stokes Flow
N-body Formulation

Thus the velocity evaluation is an *N*-body summation:

$$u_\sigma(x,t) = \sum_{j=1}^{N} \gamma_j \, \mathbb{K}_\sigma(x - x_j).$$

Fast Multipole Method

FMM accelerates the calculation of the function:

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

- 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

Fast Multipole Method

FMM accelerates the calculation of the function:

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

- 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

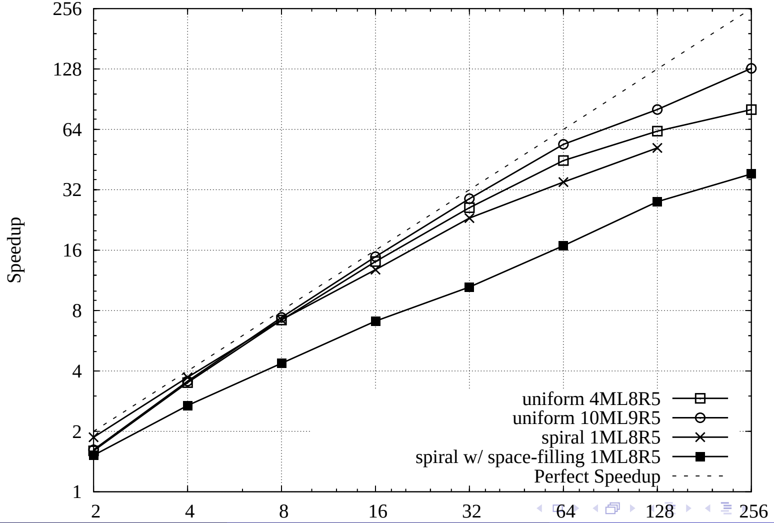
PetFMM

PetFMM is an freely available implementation of the
Fast Multipole Method
http://barbagroup.bu.edu/Barba_group/PetFMM.html

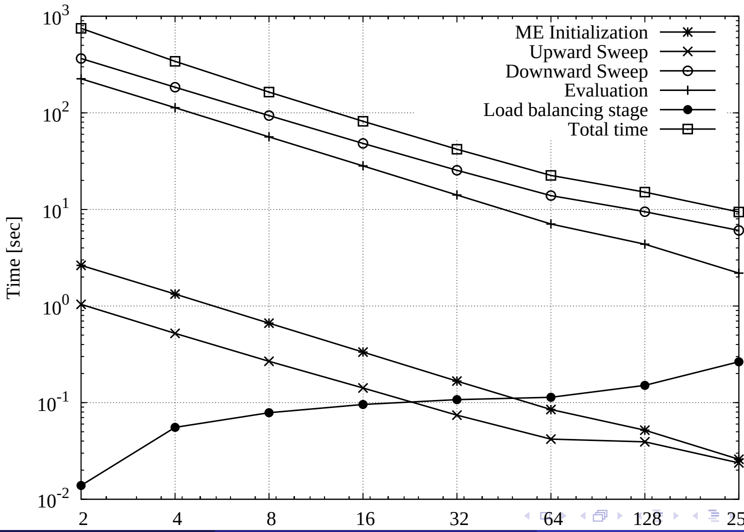
- Leverages PETSc
  - Same open source license
  - Uses Sieve for parallelism
- Extensible design in C++
  - Templated over the kernel
  - Templated over traversal for evaluation
- MPI implementation
  - Novel parallel strategy for anisotropic/sparse particle distributions
  - PetFMM–A dynamically load-balancing 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



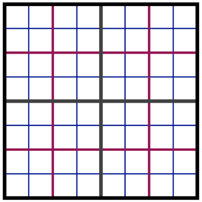
PetFMM CPU Performance
Strong Scaling

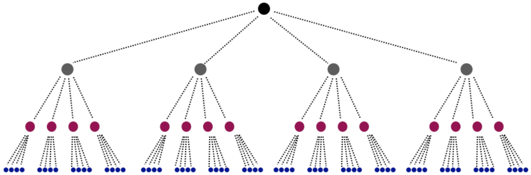


Outline

Spatial Decomposition

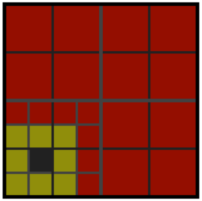
Pairs of boxes are divided into *near* and *far*:

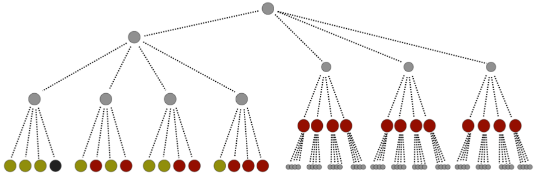




Spatial Decomposition

Pairs of boxes are divided into *near* and *far*:





Neighbors are treated as *very near*.

Outline

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
  - Box centers, Neighbors
- box + neighbors
  - Blobs
- box + interaction list
  - Interaction list cells and values
  - Multipole and local coefficients

FMM Sections

FMM requires data over the Quadtree distributed by:

- box
    - Box centers, Neighbors
- box + neighbors
    - Blobs
- box + interaction list
    - Interaction list cells and values
    - Multipole and local coefficients

Notice this is multiscale since data is divided at each level

Outline

Greengard & Gropp Analysis

For a shared memory machine,

$$T = a\frac{N}{P} + b\log_4 P + c\frac{N}{BP} + d\frac{NB}{P} + e(N,P) \qquad (2)$$

1. Initialize multipole expansions, finest local expansions, final sum
2. Reduction bottleneck
3. Translation and Multipole-to-Local
4. Direct interaction
5. Low order terms

A Parallel Version of the Fast Multipole Method,
L. Greengard and W.D. Gropp, *Comp. Math. Appl.*, **20**(7), 1990.

Outline

GPU vs. CPU

A GPU looks like a big CPU with no virtual memory:

- Many more hardware threads encourage concurrency
- Makes bandwidth limitations even more acute
- *Shared memory* is really a user-managed cache
- *Texture memory* is also a specialized cache
- User also manages a very small code segment

GPU vs. CPU

Power usage can be very different:

Platform	TF	KW	GB/s	Price ($)	GF/$	GF/W
IBM BG/P	14	40.00	57.0*	1,800,000	0.008	0.35
IBM BlueGene	280	5000	???	350,000,000	0.0008	0.55
NVIDIA C1060	1	0.19	102.0	1,475	0.680	5.35
ATI 9250	1	0.12	63.5	840	1.220	8.33

Table: Comparison of Supercomputing Hardware.

GPU programming in General

- What design ideas are useful?
- How do we customize them for GPUs?
- Can we show an example?

Break Operations Into Small Chunks

Usually called modularity

- Also called *orthogonality* or *separation of concerns*
- Allows reduction of complexity
    - eXtreme programming
- Just concerned with functionality

Break Operations Into Small Chunks
GPU Differences

We now have to worry about code size!

- 16K total for NVIDIA 1060C board
    - Instructions can be a significant portion of memory usage
- Have to split operations which logically belong together
- Also allows aggregation of memory access
    - Computation can be regrouped
- Needs tools to manage many small tasks

Break Operations Into Small Chunks

Example

Reduction over a dataset

- For instance, computation of finite element integrals
- Break into *computation* and *aggregation* stages
- Model this by:
  - Maximum flop rate stage
  - Bandwidth limited stage

Break Operations Into Small Chunks
Example

Reduction over a dataset

- For instance, computation of Multipole-to-Local transform
- Break into *computation* and *aggregation* stages
- Model this by:
  - Maximum flop rate stage
  - Bandwidth limited stage

Reorder for Locality

Exploits “nearby” operations to aggregate computation

- Can be *temporal* or *spatial*
- Usually exploits a cache
- Difficult to predict/model on a modern processor

Reorder for Locality
GPU Differences

We have to manage our “cache” explicitly

- The NVIDIA 1060C shared memory is only 16K for 32 threads
- We must also manange “main memory” explicitly
  - Need to move data to/from GPU
- Must be aware of limited precision when reordering
- Can be readily modeled
- Need tools for automatic data movement (marshalling)

Reorder for Locality
Example

Data-Aware Work Queue

- A work queue manages many small tasks
  - Dependencies are tracked with a DAG
  - Queue should manage a single computational phase (supertask)
- Nodes also manage an input and output data segment
  - Specific classes can have known sizes
  - Can hold main memory locations for segments
- Framework manages marshalling:
  - Allocates contiguous data segments
  - Calculates segment offsets for tasks
  - Marshalls (moves) data
  - Passes offsets to supertask execution

Outline

PetFMM-GPU

We break down sweep operations into `Tasks`

- Cell loops are now tiled
- Tasks are queued
- We can form a DAG since we know the dependence structure
- Scheduling is possible

This asynchronous interface can enable

- Overlapping direct and multipole calculations
- Reorganizing the downward sweep
- Adaptive expansions

GPU Classes

`Section`

- `size()` returns the number of values
- `getFiberDimension(cell)` returns the number of cell values
- `restrict/update()` retrieves and changes cell values
- `clone/extract()` converts between CPU and GPU objects

`Evaluator`

- `initializeExpansions()`
- `upwardSweep()`
- `downwardSweepTransform()`
- `downwardSweepTranslate()`
- `evaluateBlobs()`
- `evaluate()`

GPU Classes

`Section`

- `size()` returns the number of values
- `getFiberDimension(cell)` returns the number of cell values
- `restrict/update()` retrieves and changes cell values
- `clone/extract()` converts between CPU and GPU objects

`Task`

- Input data size
- Output data size
- Dependencies (future)

`TaskQueue`

- Manages storage and offsets
- `evaluate()`

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Transform Task`

- cell block

in cell and interaction list centers, interaction list multipole coeff

out cell temp local coeff

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Transform Task`

- cell block

in cell and interaction list centers, cell multipole coeff

out interaction list temp local coefficients

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Tasks

`Upward Sweep Task`

- cell block

in cell and child centers, child multipole coeff

out cell multipole coeff

`Downward Sweep Reduce Task`

- cell block

in interaction list temp local coefficients

out cell temp local coefficients

`Downward Sweep Expansion Task`

- cell block

in cell and parent centers, cell temp local coeff, parent local coeff

out cell local coeff

Transform Task

Shifts interaction cell multipole expansion to cell local expansion

- Add a task for each interaction cell
- All tasks with same origin are merged
- Local memory:
  - 2 (p+1) blockSize (Pascal) + 2 p blockSize (LE) + 2 p (ME)

8 terms	4416 bytes
17 terms	9096 bytes

- Execution
  - 1 block per ME
  - Each thread reads a section of ME and the MEcenter
  - Each thread computes an LE separately
  - Each thread writes LE to separate global location

Reduce Task

Add up local expansion contributions from each interaction cell

- Add a task for each cell
- Local memory:
  - 2*terms (LE)

8 terms	64 bytes
17 terms	136 bytes

- Execution
  - 1 block per output LE
  - Each thread reads a section of input LE
  - Each thread adds to shared output LE

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

GPU Performance

- In our C++ code on a CPU, M2L transforms take 85% of the time
  - This does vary depending on N
- New M2L design was implemented using PyCUDA
  - Port to C++ is underway
- We can now achieve 500 GF on the NVIDIA Tesla
  - Previous best performance we found was 100 GF
- We will release PetFMM-GPU in the new year

CPU vs GPU

Sample run for 250,000 vortex particles in an 8 level tree

Section	Time(s)	
	PyCUDA	Laptop C++
Setup	0.55	0.00
InitExpansions	10.74	0.93
UpSweep	0.36	5.02
DownSweepEnqueue	0.09	—
GPUOverhead	2.97	—
DownSweepM2LTrns	2.08	363.21
DownSweepM2LRed	0.45	—
DownSweepL2L	0.36	4.11

Notice that once direct evaluation is moved to the GPU, Python can easily outperform C++.

Outline

Outline

3 Multicore Implementation
- Complexity Analysis
- Redesign

Greengard & Gropp Analysis

For a shared memory machine,

$$T = a\frac{N}{P} + b\log_4 P + c\frac{N}{BP} + d\frac{NB}{P} + e(N,P) \tag{3}$$

1. Initialize multipole expansions, finest local expansions, final sum
2. Reduction bottleneck
3. Translation and Multipole-to-Local
4. Direct interaction
5. Low order terms

A Parallel Version of the Fast Multipole Method,
L. Greengard and W.D. Gropp, *Comp. Math. Appl.*, **20**(7), 1990.

Outline

Question

What is the optimal number of particles per cell?

- Greengard & Gropp
  - Minimize time and maximize parallel efficiency
  - $B_{opt} = \sqrt{\frac{c}{d}} \approx 30$
- Gumerov & Duraiswami
  - Follow GG, but also try to consider memory access
  - $B_{opt} \approx 91$, but instead, they choose 320
  - Heavily weights the N^2 part of the computation
- We propose to cover up the bottleneck with direct evaluations

Problem
Missing Concurrency

We can balance time in direct evaluation with idle time for small grids.

- The direct evaluation takes time $d\frac{NB}{p}$
- Assume a single thread group works on the first L tree levels

Thus, we need

$$B \geq \frac{b}{d}\frac{4^{L+1}p}{N} \tag{4}$$

in order to cover the bottleneck. In an upcoming publication, we show that this bound holds for all modern processors.

Problem
Missing Bandwidth

We can restructure the M2L to conserve bandwidth

- Matrix-free application of M2L
- Reorganize traversal to minimize bandwidth

Old	Pull in 27 interaction MEs, transform to LE, reduce
New	Pull in cell ME, transform to 27 interaction LEs, partially reduce

Matrix-Free M2L

The M2L transformation applies the operator

$$M_{ij} = -1^{i} t^{-(i+j+1)} \binom{i+j}{j} \tag{5}$$

Notice that the t exponent is constant along perdiagonals. Thus we

- divide by t at each perdiagonal
- calculate the C_{ij} by the recurrence along each perdiagonal
- carefully formulate complex division (STL fails here)

What's Important?

Interface improvements bring concrete benefits

- Facilitated code reuse
  - Serial code was largely reused
  - Test infrastructure completely reused
- Opportunites for performance improvement
  - Overlapping computations
  - Better task scheduling
- Expansion of capabilities
  - Could now combine distributed and multicore implementations
  - Could replace local expansions with cheaper alternatives