Software Design for PDEs on GPUs

Matthew Knepley

Computation Institute
University of Chicago
Department of Molecular Biology and Physiology
Rush University Medical Center

High Performance Computing and Emerging Architectures
Institute for Mathematics and Its Applications
Minneapolis, January 10, 2011
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
Collaborators

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
The **PyLith Team:**

- **Dr. Brad Aagaard** (PyLith)
  - United States Geological Survey, Menlo Park, CA

- **Dr. Charles Williams** (PyLith)
  - GNS Science, Wellington, NZ
To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
To be widely accepted, GPU computing must be transparent to the user, and reuse existing infrastructure.
Lessons from Clusters and MPPs

Failure
- Parallelizing Compilers
- Automatic program decomposition

Success
- MPI (Library Approach)
- PETSc (Parallel Linear Algebra)
- User provides only the mathematical description
Lessons from Clusters and MPPs

Failure
- Parallelizing Compilers
- Automatic program decomposition

Success
- MPI (Library Approach)
- PETSc (Parallel Linear Algebra)
- User provides only the mathematical description
Outline

1. PETSc-GPU
2. FEM-GPU
3. FMM-GPU
Thrust is a CUDA library of parallel algorithms

- Interface similar to C++ Standard Template Library
- Containers (vector) on both host and device
- Algorithms: sort, reduce, scan
- Freely available, part of PETSc configure (-with-thrust-dir)
Cusp is a CUDA library for sparse linear algebra and graph computations

- Builds on data structures in Thrust
- Provides sparse matrices in several formats (CSR, Hybrid)
- Includes some preliminary preconditioners (Jacobi, SA-AMG)
- Freely available, part of PETSc configure (\texttt{--with-cusp-dir})
Strategy: Define a new Vec implementation

- Uses Thrust for data storage and operations on GPU
- Supports full PETSc Vec interface
- Inherits PETSc scalar type
- Can be activated at runtime, -vec_type cuda
- PETSc provides memory coherence mechanism
PETSc-GPU

Memory Coherence

PETSc Objects now hold a coherence flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETSC_CUDA_UNALLOCATED</td>
<td>No allocation on the GPU</td>
</tr>
<tr>
<td>PETSC_CUDA_GPU</td>
<td>Values on GPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_CPU</td>
<td>Values on CPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_BOTH</td>
<td>Values on both are current</td>
</tr>
</tbody>
</table>

Table: Flags used to indicate the memory state of a PETSc CUDA Vec object.
Also define new Mat implementations

- Uses Cusp for data storage and operations on GPU
- Supports full PETSc Mat interface, some ops on CPU
- Can be activated at runtime, -mat_type aijcuda
- Notice that parallel matvec necessitates off-GPU data transfer
Solvers come for Free

- All linear algebra types work with solvers
- Entire solve can take place on the GPU
  - Only communicate scalars back to CPU
- GPU communication cost could be amortized over several solves
- Preconditioners are a problem
  - Cusp has a promising AMG
PETSc only needs

```bash
# Turn on CUDA
--with-cuda
# Specify the CUDA compiler
--with-cudac='nvcc -m64'
# Indicate the location of packages
# --download-* will also work soon
--with-thrust-dir=/PETSc3/multicore/thrust
--with-cusp-dir=/PETSc3/multicore/cusp
# Can also use double precision
--with-precision=single
```
PETSc-GPU

Example
Driven Cavity Velocity-Vorticity with Multigrid

```plaintext
ex19 -da_vec_type seqcuda
   -da_mat_type aijcuda -mat_no_inode
   -da_grid_x 100 -da_grid_y 100
   -pc_type none -dmmg_nlevels 1
   -preload off -cuda_synchronize
   -log_summary
```

# Setup types
# Set grid size
# Setup solver
# Setup run
Outline

1. PETSc-GPU

2. FEM-GPU
   - Analytic Flexibility
   - Computational Flexibility
   - Efficiency

3. FMM-GPU
What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
What are the Benefits for current PDE Code?

Low Order FEM on GPUs

- Analytic Flexibility
- Computational Flexibility
- Efficiency

http://www.bitbucket.org/aterrel/flamefem
Outline

2 FEM-GPU
- Analytic Flexibility
- Computational Flexibility
- Efficiency
Analytic Flexibility

Laplacian

\[ \int_{T} \nabla \phi_i(x) \cdot \nabla \phi_j(x) \, dx \]  \hspace{1cm} (1)

```
element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u)) * dx
```
\[ \int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) d\mathbf{x} \quad (1) \]

element = FiniteElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(grad(v), grad(u)) * dx
\[
\frac{1}{4} \int_T \left( \nabla \tilde{\phi}_i(x) + \nabla^T \tilde{\phi}_i(x) \right) : \left( \nabla \tilde{\phi}_j(x) + \nabla^T \tilde{\phi}_j(x) \right) \, dx
\] (2)

```python
element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u))) * dx
```
\[
\frac{1}{4} \int_T \left( \nabla \phi_i(x) + \nabla^T \phi_i(x) \right) : \left( \nabla \phi_j(x) + \nabla^T \phi_j(x) \right) \, dx
\]  
(2)

```
element = VectorElement('Lagrange', tetrahedron, 1)
v = TestFunction(element)
u = TrialFunction(element)
a = inner(sym(grad(v)), sym(grad(u))) * dx
```
\[ \frac{1}{4} \int_{\mathcal{T}} \left( \nabla \tilde{\phi}_i(x) + \nabla^T \tilde{\phi}_i(x) \right) : C : \left( \nabla \tilde{\phi}_j(x) + \nabla \tilde{\phi}_j(x) \right) \, dx \]  

\text{element} = \text{VectorElement('Lagrange', tetrahedron, 1)} \\
\text{cElement} = \text{TensorElement('Lagrange', tetrahedron, 1, (dim, dim, dim, dim, dim))} \\
v = \text{TestFunction(element)} \\
u = \text{TrialFunction(element)} \\
C = \text{Coefficient(cElement)} \\
i, j, k, l = \text{indices(4)} \\
a = \text{sym} (\text{grad}(v)) [i,j] \times C[i,j,k,l] \times \text{sym} (\text{grad}(u)) [k,l] \times dx \\

Currently broken in FEniCS release
\[ \frac{1}{4} \int_{\mathcal{T}} \left( \nabla \vec{\phi}_i(\mathbf{x}) + \nabla^T \vec{\phi}_i(\mathbf{x}) \right) : \mathbf{C} : \left( \nabla \vec{\phi}_j(\mathbf{x}) + \nabla^T \vec{\phi}_j(\mathbf{x}) \right) \, d\mathbf{x} \]  

\[ (3) \]

```python
element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                         (dim, dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j]*C[i,j,k,l]*sym(grad(u))[k,l]*dx
```

Currently broken in FEniCS release
\[ \frac{1}{4} \int_T \left( \nabla \vec{\phi}_i(x) + \nabla^T \vec{\phi}_i(x) \right) : C : \left( \nabla \vec{\phi}_j(x) + \nabla \vec{\phi}_j(x) \right) \, dx \] (3)

element = VectorElement('Lagrange', tetrahedron, 1)
cElement = TensorElement('Lagrange', tetrahedron, 1,
                             (dim, dim, dim, dim, dim))
v = TestFunction(element)
u = TrialFunction(element)
C = Coefficient(cElement)
i, j, k, l = indices(4)
a = sym(grad(v))[i,j] * C[i,j,k,l] * sym(grad(u))[k,l] * dx

Currently broken in FEniCS release
Outline

FEM-GPU

- Analytic Flexibility
- Computational Flexibility
- Efficiency
Element integrals are decomposed into **analytic** and **geometric** parts:

\[
\begin{align*}
\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) d\mathbf{x} &= \\
&= \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} d\mathbf{x} \\
&= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| d\mathbf{x} \\
&= \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} d\mathbf{x} \\
&= G^{\beta\gamma}(\mathcal{T}) K_{\beta\gamma}^{ij}
\end{align*}
\]

Coefficients are also put into the geometric part.
Form Decomposition

Additional fields give rise to multilinear forms.

\[
\int_T \phi_i(x) \cdot (\phi_k(x) \nabla \phi_j(x)) \, dA
= \int_T \phi_i^\beta(x) \left( \phi_k^\alpha(x) \frac{\partial \phi_j^\beta(x)}{\partial x_\alpha} \right) \, dA
= \int_{T_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} |J| \, dA
= \frac{\partial \xi_\gamma}{\partial x_\alpha} |J| \int_{T_{\text{ref}}} \phi_i^\beta(\xi) \phi_k^\alpha(\xi) \frac{\partial \phi_j^\beta(\xi)}{\partial \xi_\gamma} \, dA
= G^{\alpha\gamma}(T) K_{ij}^{\alpha\gamma}
\]

The index calculus is fully developed by Kirby and Logg in

A Compiler for Variational Forms.
Isoparametric Jacobians also give rise to \textbf{multilinear forms}

\[
\int_{\mathcal{T}} \nabla \phi_i(\mathbf{x}) \cdot \nabla \phi_j(\mathbf{x}) dA
\]

\[
= \int_{\mathcal{T}} \frac{\partial \phi_i(\mathbf{x})}{\partial x_\alpha} \frac{\partial \phi_j(\mathbf{x})}{\partial x_\alpha} dA
\]

\[
= \int_{\mathcal{T}_{\text{ref}}} \frac{\partial \xi_\beta}{\partial x_\alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \frac{\partial \xi_\gamma}{\partial x_\alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} |J| dA
\]

\[
= |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k J_{k}^{\beta \alpha} \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l J_{l}^{\gamma \alpha} \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} dA
\]

\[
= J_{k}^{\beta \alpha} J_{l}^{\gamma \alpha} |J| \int_{\mathcal{T}_{\text{ref}}} \phi_k \frac{\partial \phi_i(\xi)}{\partial \xi_\beta} \phi_l \frac{\partial \phi_j(\xi)}{\partial \xi_\gamma} dA
\]

\[
= G_{k l}^{\beta \gamma}(\mathcal{T}) K_{ijkl}^{\beta \gamma}
\]

A different space could also be used for Jacobians
from ffc.analysis import analyze_forms
from ffc.compiler import compute_ir

parameters = ffc.default_parameters()
parameters["representation"] = "tensor"
analysis = analyze_forms([a,L], {}, parameters)
ir = compute_ir(analysis, parameters)

a_K = ir[2][0]["AK"][0][0]
a_G = ir[2][0]["AK"][0][1]

K = a_K.A0.astype(numpy.float32)
G = a_G
We generate different computations on the fly, and can change

- Element Batch Size
- Number of Concurrent Elements
- Loop unrolling
- Interleaving stores with computation
Figure: Tensor Contraction $G^\beta\gamma(T_\tau)K^{ij}_{\beta\gamma}$
Figure: Tensor Contraction $G^{\beta\gamma}(T)K_{\beta\gamma}^{ij}$
Computational Flexibility

Basic Contraction

Figure: Tensor Contraction $G^\beta\gamma(T)K^i_j$
Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T}) K^{ij}_{\beta\gamma}$
Figure: Tensor Contraction $G^\beta_\gamma(T)K^ij_\beta_\gamma$
Computational Flexibility
Element Batch Size

Figure: Tensor Contraction $G^{\beta\gamma}(\mathcal{T})K_{\beta\gamma}^{ij}$
Figure: Tensor Contraction $G^\beta\gamma(T)K^{ij}_{\beta\gamma}$
Computational Flexibility
Element Batch Size

Figure: Tensor Contraction $G_\beta\gamma(T)K_\beta^i_j$
Concurrent Elements

\[ G_0^0 \quad G_0^1 \quad G_1^0 \quad G_1^1 \]

\[ G_0^2 \quad G_0^3 \quad G_1^2 \quad G_1^3 \]

\[ K \]

Figure: Tensor Contraction

\[ \beta \gamma \]

\[ (T) \]

\[ K_{ij} \]

M. Knepley (UC)

PDE on GPU

IMA '10 30 / 57
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction $G^0_0 \rightarrow K \rightarrow G^1_0$

M. Knepley (UC)
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K \beta\gamma(T) \]

M. Knepley (UC)
Computational Flexibility
Concurrent Elements

Figure: Tensor Contraction

\[ K \]

\[ T_{\beta\gamma} \]

M. Knepley (UC)
PDE on GPU
IMA '10 30 / 57
Loop Unrolling

/* G K contraction: unroll = full */
E[0] += G[0] * K[0];
E[0] += G[1] * K[1];
E[0] += G[7] * K[7];
E[0] += G[8] * K[8];
/* G K contraction: unroll = none */

```c
for (int b = 0; b < 1; ++b) {
    const int n = b*1;
    for (int alpha = 0; alpha < 3; ++alpha) {
        for (int beta = 0; beta < 3; ++beta) {
            E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
        }
    }
}
```
/* G K contraction: unroll = none */
for(int b = 0; b < 4; ++b) {
    const int n = b*1;
    for(int alpha = 0; alpha < 3; ++alpha) {
        for(int beta = 0; beta < 3; ++beta) {
            E[b] += G[n*9+alpha*3+beta] * K[alpha*3+beta];
        }
    }
}

/* Store contraction results */
elemMat[Eoffset+idx+0] = E[0];
elemMat[Eoffset+idx+16] = E[1];
elemMat[Eoffset+idx+32] = E[2];
elemMat[Eoffset+idx+48] = E[3];
n = 0;
for (int alpha = 0; alpha < 3; ++alpha) {
    for (int beta = 0; beta < 3; ++beta) {
        E += G[n*9+alpha*3+beta] * K[alpha*3+beta];
    }
}
/* Store contraction result */
elemMat[Eoffset+idx+0] = E;
n = 1; E = 0.0; /* contract */
elemMat[Eoffset+idx+16] = E;
n = 2; E = 0.0; /* contract */
elemMat[Eoffset+idx+32] = E;
n = 3; E = 0.0; /* contract */
elemMat[Eoffset+idx+48] = E;
FEM-GPU

- Analytic Flexibility
- Computational Flexibility
- Efficiency
## Price-Performance Comparison of CPU and GPU

3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>2</td>
<td>6.6</td>
</tr>
</tbody>
</table>
## Price-Performance Comparison of CPU and GPU

### 3D $P_1$ Laplacian Integration

<table>
<thead>
<tr>
<th>Model</th>
<th>Price ($)</th>
<th>GF/s</th>
<th>MF/s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX285</td>
<td>390</td>
<td>90</td>
<td>231</td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>300</td>
<td>12*</td>
<td>40</td>
</tr>
</tbody>
</table>

* Jed Brown Optimization Engine
Efficiency

Performance

Influence of Element Batch Sizes

CPU vs. GPU Flop Rate for 2D $P_1$ Lagrange ['Elasticity']

Interleave Stores = 1
Loop Unrolling = full
Explaining performance

- Increase shared memory and work/thread until you top out
  - Occupancies go down or level out as performance goes up

- Does not work without interleaved stores
  - Scheduler can switch to kernels who are computing
  - Larger number of smaller computations makes better fit

- Should I worry about detailed explanations for performance?
  - Sensible decompositions, coupled with exploration
  - FLAME methodology
Automated Tuning System

Components of our performance evaluation system:

- Generate set of kernels using:
  - Loop slicing, store reordering, etc.
  - Loop invariants ala FLAME
  - High level constructs ala Rheagen and FEniCS

- Store results and metadata in HDF5 using PyTables
  - Thousands of tests for this talk

- Interrogate and plot with Matplotlib

- Eventually couple to build system
  - FFTW, Spiral, FLAME
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.
Structured code generation, can allow easy integration of novel hardware and reconcile user physics with system traversals.
### Outline

1. PETSc-GPU
2. FEM-GPU
3. FMM-GPU
   - Quick FMM Intro
   - Differences on the GPU
Outline

3  FMM-GPU
   • Quick FMM Intro
   • Differences on the GPU
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:

\[
\Phi(x_i) = \sum_j K(x_i, x_j)q(x_j)
\]  

(20)

- 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} (20)

- 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
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
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*.
Functional Decomposition

- **Upward Sweep**
  - Create Multipole Expansions.
  - P2M → M2M → M2L
- **Downward Sweep**
  - Evaluate Local Expansions.
  - L2L → L2P
Outline

3 FMM-GPU
- Quick FMM Intro
- Differences on the GPU
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
  - $(6^d - 3^d) \times 2^{dL}$
- $d = 2, L = 8$
  - 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
- $\rho = 12$
- 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

\[ m_{2l_{ij}} = -1^i \binom{i + j}{j} r^{-i-j-1} \] (21)

- 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

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

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

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (21)

- Traverse matrix by perdiagonals
- Same work
- No memory limit on concurrency
- 8 concurrent TBs per MultiProcessor (MP)

\( 27 \times 8 = 216 \text{ threads, BUT max is 512} \)
Apply M2L transform matrix-free

\[ m_{ij} = -1^i \binom{i + j}{j} t^{-i-j-1} \] (21)

- 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
GPU M2L
Version 1

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

Algorithm limits concurrency!

20 GFlops

5x Speedup of Downward Sweep
Apply M2L transform matrix-free

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

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

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{i-j-1} \]  \hspace{1cm} (22)

- 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

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \quad (22) \]

- 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

\[ m2l_{ij} = -1^i \binom{i + j}{j} t^{-i-j-1} \]  \hspace{1cm} (22)

- 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

Examine memory access
One thread per *element* of the LE

\[ m2l_{ij} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (22)

- 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
GPU M2L
Version 2

One thread per *element* of the LE

\[ m_{2l_{ij}} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (22)

- 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
One thread per *element* of the LE

\[ m_{2l_{ij}} = -1^i \binom{i+j}{j} t^{-i-j-1} \] (22)

- 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 \textit{bus width} and \textit{clock speed}.

\begin{tabular}{|l|c|c|}
\hline
 & CPU & GPU \\
\hline
Bus Width (bits) & 64 & 512 \\
Bus Clock Speed (MHz) & 400 & 1600 \\
Memory Bandwidth (GB/s) & 3 & 102 \\
Latency (cycles) & 240 & 600 \\
\hline
\end{tabular}

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
How Will Algorithms Change?

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

- More attention to **memory management**
  - Blocks will only get larger
  - Determinant of performance