Assembly of Finite Element Methods on Graphics Processors

Cris Cecka Toru Takahashi Adrian Lew Eric Darve

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

January 12th, 2011
Institute for Mathematics and its Applications
1 Motivation

2 FEM Assembly

3 GPU Assembly
   - LocalElem
   - GlobalNZ
   - SharedNZ
   - Scatter and Reduction Arrays

4 Results

5 Application

6 Conclusion

7 FMM GPU

Cecka, Takahashi, Lew, Darve

FEM/FMM on GPU
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - **Analyzable DSL**: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Productivity
  - Separate computational science from computer science
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Productivity
  - Separate computational science from computer science

- Portability
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Productivity
  - Separate computational science from computer science

- Portability

- Performance
  - Use domain knowledge and platform knowledge
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Productivity
  - Separate computational science from computer science

- Portability

- Performance
  - Use domain knowledge and platform knowledge

- Innovation
  - Change architectures and programming models under-the-hood
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Understands and uses topology
  - Domain decomposition
  - Sparsity pattern

for (f <- faces(cell))
for (c <- cell(mesh)) {
  ...
}
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - **Analyzable DSL**: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Understands and uses topology
  - Domain decomposition
  - Sparsity pattern

- Understands, transforms, and parallelizes loops
  
  ```
  for( f <- faces(cell) )
  for( c <- cell(mesh) ) {
    ...  
  }
  ```
Domain Specific Languages

- Liszt @ Stanford
  - Mesh-based PDEs on Heterogeneous Platforms
  - Analyzable DSL: Language with domain-specific features and restrictions that provide context for domain specific transformations

- Understands and uses topology
  - Domain decomposition
  - Sparsity pattern

- Understands, transforms, and parallelizes loops
  - `for( f <- faces(cell) )
    for( c <- cell(mesh) ) { ... }`

- Stencils statically analyzed from access patterns
Why FEM Assembly on the GPU?

- Sparse Linear Algebra coming of age on GPU.
  - Extensive research on Sparse Solvers on GPU.
  - Extensive research on SpMV.
Why FEM Assembly on the GPU?

- Sparse Linear Algebra coming of age on GPU.
  - Extensive research on Sparse Solvers on GPU.
  - Extensive research on SpMV.
- Non-linear and time-dependent problems require many assembly procedures.
Why FEM Assembly on the GPU?

- Sparse Linear Algebra coming of age on GPU.
  - Extensive research on Sparse Solvers on GPU.
  - Extensive research on SpMV.

- Non-linear and time-dependent problems require many assembly procedures.

- Want to use topology efficiently with a black-box element kernel.
Why FEM Assembly on the GPU?

- Sparse Linear Algebra coming of age on GPU.
  - Extensive research on Sparse Solvers on GPU.
  - Extensive research on SpMV.

- Non-linear and time-dependent problems require many assembly procedures.

- Want to use topology efficiently with a black-box element kernel.

- Can assemble, solve, update, and visualize on the GPU
  - Completely avoid transfers with CPU.
  - Fast (real-time) simulations with visualization.
FEM Direct Assembly

Most common FEM assembly procedure:
- Compute element data.
- One by one.

\[ K^e \mathbf{u} = f^e \]
FEM Direct Assembly

Most common FEM assembly procedure:

- Compute element data.
  - One by one.
- Accumulate into global system.
  - Using a local index to global index mapping.

\[
\mathbf{K}_e \mathbf{u} = \mathbf{f}_e
\]

\[
\mathbf{Ku} = \mathbf{f}
\]
Data Flow

Nodal Data

---
Data Flow

Nodal Data  Element Data

Diagram showing the flow from nodal data to element data.
Data Flow

Nodal Data  Element Data
Data Flow

Nodal Data  Element Data  FEM System
Two Key Choices:

Store Element Data In

Threads Assemble By

- Global Memory
  - Min computation
  - Fast read/write
  - No shared element data

- Local Memory
  - Fast read/write
  - Small size

- Shared Memory
  - Fast read/write
  - Small size

- Threads Assemble By Non-zero (NZ)
  - Simple - Indexing
  - Imbalanced

- Row
  - More balanced
  - Lookup tables

- Element
  - Race conditions
GPU FEM Assembly Strategies

Two Key Choices:

**Store Element Data In**
- Global Memory
- Local Memory
- Shared Memory

**Threads Assemble By**
- Non-zero (NZ) Simple - Indexing
- Row
- Lookup tables
- Imbalanced
- Element Race conditions
GPU FEM Assembly Strategies

Two Key Choices:

Store Element Data In
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
- Shared Memory

Threads Assemble By
- Non-zero (NZ) Simple - Indexing.
  - Imbalanced.
- Row More balanced.
  - Lookup tables.

Cecka, Takahashi, Lew, Darve
FEM/FMM on GPU
GPU FEM Assembly Strategies

Two Key Choices:

**Store Element Data In**
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory

**Threads Assemble By**
Two Key Choices:

Store Element Data In
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory
  - Fast read/write.
  - Small size.

Threads Assemble By

Cecka, Takahashi, Lew, Darve
Two Key Choices:

**Store Element Data In**
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory
  - Fast read/write.
  - Small size.

**Threads Assemble By**
- Non-zero (NZ)
- Row
- Element
Two Key Choices:

**Store Element Data In**
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory
  - Fast read/write.
  - Small size.

**Threads Assemble By**
- Non-zero (NZ)
  - Simple - Indexing.
  - Imbalanced.
- Row
- Element
GPU FEM Assembly Strategies

Two Key Choices:

**Store Element Data In**
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory
  - Fast read/write.
  - Small size.

**Threads Assemble By**
- Non-zero (NZ)
  - Simple - Indexing.
  - Imbalanced.
- Row
  - More balanced.
  - Lookup tables.
- Element
Two Key Choices:

**Store Element Data In**
- Global Memory
  - Computation / Assembly.
  - Min computation.
- Local Memory
  - Fast read/write.
  - No shared element data.
- Shared Memory
  - Fast read/write.
  - Small size.

**Threads Assemble By**
- Non-zero (NZ)
  - Simple - Indexing.
  - Imbalanced.
- Row
  - More balanced.
  - Lookup tables.
- Element
  - Race conditions.
Local-Element

- Assign one thread to one element.
  - Compute the element data.
  - Assemble directly into system.

Race conditions still possible!
Partition elements to resolve race conditions.
Transform into a coloring problem.

Problems.
No sharing of nodal or element data.
Little utilization of GPU resources.
Assign one thread to one element.
  - Compute the element data.
  - Assemble directly into system.

Race conditions still possible!
Assign one thread to one element.
- Compute the element data.
- Assemble directly into system.

Partition elements to resolve race conditions.
- Transform into a coloring problem.
Assign one thread to one element.
- Compute the element data.
- Assemble directly into system.

Partition elements to resolve race conditions.
- Transform into a coloring problem.

Problems.
- No sharing of nodal or element data.
- Little utilization of GPU resources.
Global-NZ

- Kernel1 - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.

- Kernel2 - Assign one thread to one NZ.
  - Assemble from global memory.

Optimizing:
- Cluster the elements so they share nodes.
- Prefetch nodal data into shared memory.

Problems:
- Extra passes through global memory.
Global-NZ

- Kernel1 - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.
- Kernel2 - Assign one thread to one NZ.
  - Assemble from global memory.

Optimizing:
Cluster the elements so they share nodes.
Prefetch nodal data into shared memory.

Problems.
Extra passes through global memory.
Global-NZ

- **Kernel1** - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.
- **Kernel2** - Assign one thread to one NZ.
  - Assemble from global memory.

**Optimizing:**
- Cluster the elements so they share nodes.
- Prefetch nodal data into shared memory.
Global-NZ

- Kernel1 - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.
- Kernel2 - Assign one thread to one NZ.
  - Assemble from global memory.

Optimizing:
- Cluster the elements so they share nodes.
- Prefetch nodal data into shared memory.

Problems.
- Extra passes through global memory.
The optimized algorithm looks like:

Nodal Data:
Gather:
Nodal Data:
Thread Sync
Block Element Matrix $E_k$:
Element Subroutine:
Coalesced Write:
Element Data:
Kernel Break
Reduction:
System of Equations:
Assign one thread to one element.
- Compute the element data.
- Store element data in shared memory.

Reassign threads to NZs.

Assemble from shared memory.

A set of NZs requires a set of elements.

Must compute all "halo" element data.

A set of elements requires a set of nodes.

Must gather all "halo" nodal data.

Problems.

Shared memory size is very limiting.
Assign one thread to one element.
  - Compute the element data.
  - Store element data in shared memory.

Reassign threads to NZs.
  - Assemble from shared memory.
Assign one thread to one element.
- Compute the element data.
- Store element data in shared memory.

Reassign threads to NZs.
- Assemble from shared memory.

A set of NZs requires a set of elements.
- Must compute all “halo” element data.
Shared-NZ

- Assign one thread to one element.
  - Compute the element data.
  - Store element data in shared memory.
- Reassign threads to NZs.
  - Assemble from shared memory.

A set of NZs requires a set of elements.
- Must compute all “halo” element data.

A set of elements requires a set of nodes.
- Must gather all “halo” nodal data.
Shared-NZ

- Assign one thread to one element.
  - Compute the element data.
  - Store element data in shared memory.
- Reassign threads to NZs.
  - Assemble from shared memory.

A set of NZs requires a set of elements.
- Must compute all “halo” element data.

A set of elements requires a set of nodes.
- Must gather all “halo” nodal data.
Assign one thread to one element.
- Compute the element data.
- Store element data in shared memory.

Reassign threads to NZs.
- Assemble from shared memory.

A set of NZs requires a set of elements.
- Must compute all “halo” element data.

A set of elements requires a set of nodes.
- Must gather all “halo” nodal data.

Problems.
- Shared memory size is very limiting.
The optimized algorithm looks like:

Nodal Data:

Scatter:

Nodal Data:

Element Subroutine:

Element Data:

Reduction:

System of Equations:
Scatter and Reduction Arrays

General procedure:

- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.
Scatter and Reduction Arrays

General procedure:
- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.

Scatter Array:
Scatter and Reduction Arrays

General procedure:
- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.

Scatter Array:  

Reduction Array:
Scatter and Reduction Arrays

General procedure:
- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.

Scatter Array:
- Very fast.
- Highly adaptable.

Reduction Array:
- Significant setup cost.
- Significant memory cost.
Scaling with Element Number

![Graph showing scaling with element number across different mesh sizes and methods: Serial, ElemColor, SharedNZ, LocalNZ, GlobalNZ Assem, and GlobalNZ Elem. The x-axis represents mesh sizes in terms of nodes and elements, while the y-axis shows the log10 of time per element.](image-url)
Scaling with Element Order
Application

GPU non-linear neoHookean model.
Application

GPU non-linear neoHookean model.

- Newton-Raphson update at each step.
- Assemble, solve, update, and render at each step.
Application

GPU non-linear neoHookean model.

- Newton-Raphson update at each step.
- Assemble, solve, update, and render at each step.
- 28,796 Nodes. 125,127 Elements.
Application

**GPU non-linear neoHookean model.**
- Newton-Raphson update at each step.
- Assemble, solve, update, and render at each step.
- 28,796 Nodes. 125,127 Elements.

Implementation

- Newton-Raphson
  - NR_CPU
  - NR_GPU

- Assembly
  - AssemblyCPU
  - AssemblyCPU_Opt
  - AssemblyGlobalNZ
  - AssemblySharedNZ

- Conjugate Gradient
  - CG_CPU
  - CG_GPU
  - DCG_CPU
  - DCG_GPU

- Sparse Matrix Format
  - COO_Matrix
  - CSR_Matrix
  - HYB_Matrix
  - DCOO_Matrix
  - DCSR_Matrix
  - DHYB_Matrix

- All SpMVs on CPU and GPU.
Improvement

Total: 21.3s

- SpMV: 70%
- Assembly: 19%
- CG: 11%

Total: 5.5s

- SpMV: 16%
- Assembly: 74%
- CG: 5%
- NR: 3%
- Transfer: 2%

Total: 1.3s

- SpMV: 66%
- Assembly: 15%
- CG: 19%
Improvement

SpMV

Total: 21.3s

70%

19%

11%

CG

Assembly

SpMV

Total: 5.5s

74%

16%

19%

11%

5%

3%

2%

Transfer

NR

CG
Improvement

- **Total:** 21.3s
  - SpMV: 70%
  - Assembly: 19%
  - CG: 11%

- **Total:** 5.5s
  - SpMV: 16%
  - Assembly: 74%
  - CG: 19%

- **Total:** 1.3s
  - SpMV: 66%
  - Assembly: 15%
  - CG: 19%

Cecka, Takahashi, Lew, Darve
Assembly Speed

![Bar chart showing Assembly Speed for different matrices and GPUs.](chart.png)


- Create and classify several GPU FEM assembly algorithms.

FEM Conclusion


- Create and classify several GPU FEM assembly algorithms.
- Identification of optimizations and limitations of each algorithm.

FEM Conclusion


- Create and classify several GPU FEM assembly algorithms.
- Identification of optimizations and limitations of each algorithm.
- Optimal method depends on the element:
  - Memory requirements of element kernels.
  - Computational requirements of element kernels.

FEM Conclusion


- Create and classify several GPU FEM assembly algorithms.
- Identification of optimizations and limitations of each algorithm.
- Optimal method depends on the element:
  - Memory requirements of element kernels.
  - Computational requirements of element kernels.
- Precomputation algorithms and support data structures.

FEM Conclusion


- Create and classify several GPU FEM assembly algorithms.
- Identification of optimizations and limitations of each algorithm.
- Optimal method depends on the element:
  - Memory requirements of element kernels.
  - Computational requirements of element kernels.
- Precomputation algorithms and support data structures.


- Applying the methods to a high-performance FEM application.
Implement the M2L operation of a generalized FMM
FMM GPU Algorithms

- Implement the M2L operation of a generalized FMM
  - Know the structure and topology
  - Leave the transfer matrix as arbitrary

\[ \text{M2L stage: } L(O) := \sum_{S \in I} (O) D(O, S) M(S) \]
Implement the M2L operation of a generalized FMM

- Know the structure and topology
- Leave the transfer matrix as arbitrary

**M2L stage:**

\[ L(O) := \sum_{S \in I(O)} D^{(O,S)} M(S) \]
FMM GPU Algorithms

- Implement the M2L operation of a generalized FMM
  - Know the structure and topology
  - Leave the transfer matrix as arbitrary

- M2L stage:

  \[
  L(O) := \sum_{S \in I(O)} D^{(O,S)} M(S)
  \]

  - 316 Transfer Matrices
  - \( |I(O)| \leq 189 \)
Basic Algorithm

Block = Observation Cell \( O \)
Thread = \( L_i \)
Basic Algorithm

Block = Observation Cell \( O \)
Thread = \( L_i \)

For all \( S \in I(O) \)

Read \( D(S, O) \)
Read \( M(S) \)
\( L_i + = D(O, S) \)
\( M_j(S) \)
Write \( L_i \) per observation cell

Operation counts \( \text{flop} \)
(2r - 1)
Flop-to-word ratio \( \text{flop/word} \)
1.9 for \( r = 32 \) \((n = 4)\)
2.0 for \( r = 256 \) \((n = 8)\)
Basic Algorithm

Block = Observation Cell \( O \)
Thread = \( L_i \)

For all \( S \in \mathcal{I}(O) \)
   Read \( D(S, O) \)
   Read \( M(S) \)
Basic Algorithm

Block = Observation Cell \( O \)
Thread = \( L_i \)

For all \( S \in \mathcal{I}(O) \)
  Read \( D(S, O) \)
  Read \( M(S) \)
  \( L_i + = D_{ij}^{(O,S)} M_j(S) \)
Write \( L_i \)
Basic Algorithm

Block = Observation Cell \( O \)
Thread = \( L_i \)

For all \( S \in \mathcal{I}(O) \)

Read \( D(S, O) \)
Read \( M(S) \)

\[ L_{i+} = D_{ij}^{(O,S)} M_j(S) \]

Write \( L_i \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>per observation cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read M-data</td>
<td>[word]</td>
</tr>
<tr>
<td>Read D-data</td>
<td>[word]</td>
</tr>
<tr>
<td>Write L-data</td>
<td>[word]</td>
</tr>
<tr>
<td>Operation counts</td>
<td>[flop]</td>
</tr>
<tr>
<td>Flop-to-word ratio</td>
<td>[flop/word]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Blocking Siblings

Iterate over clusters instead of cells
Iterate over transfer class
Reuse the $D$ matrix

Per observation cluster
Per observation cell
Read $M$-data $8 \times 26 \times r$

Read $D$-data $26 \times 27 \times r$

Read/Write $L$-data $8 \times 26 \times r$

Operation counts [flop] $8 \times 189 \times r(2r-1)$

Flop-to-word ratio [flop/word] 4.2 for $r = 32$
4.3 for $r = 256$
Blocking Siblings

- Iterate over clusters instead of cells

<table>
<thead>
<tr>
<th></th>
<th>Per observation cluster</th>
<th>Per observation cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read M-data</td>
<td>[word] 8 \cdot 26 \cdot r</td>
<td>26r</td>
</tr>
<tr>
<td>Read D-data</td>
<td>[word] 26 \cdot 27 \cdot r^2</td>
<td>\frac{26 \cdot 27}{8} r^2</td>
</tr>
<tr>
<td>Read/Write L-data</td>
<td>[word] 8 \cdot 26 \cdot r</td>
<td>26r</td>
</tr>
<tr>
<td>Operation counts</td>
<td>[flop] 8 \cdot 189 \cdot r(2r - 1)</td>
<td>189r(2r - 1)</td>
</tr>
<tr>
<td>Flop-to-word ratio</td>
<td>[flop/word] 4.2 for r = 32</td>
<td>4.3 for r = 256</td>
</tr>
</tbody>
</table>
Blocking Siblings

- Iterate over clusters instead of cells
- Iterate over transfer class
- Reuse the $D$ matrix

<table>
<thead>
<tr>
<th>Operation</th>
<th>Per observation cluster</th>
<th>Per observation cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read M-data</td>
<td>[word]</td>
<td>$8 \cdot 26 \cdot r$</td>
</tr>
<tr>
<td>Read D-data</td>
<td>[word]</td>
<td>$26 \cdot 27 \cdot r^2$</td>
</tr>
<tr>
<td>Read/Write L-data</td>
<td>[word]</td>
<td>$8 \cdot 26 \cdot r$</td>
</tr>
<tr>
<td>Operation counts</td>
<td>[flop]</td>
<td>$8 \cdot 189 \cdot r(2r - 1)$</td>
</tr>
<tr>
<td>Flop-to-word ratio</td>
<td>[flop/word]</td>
<td>4.2 for $r = 32$</td>
</tr>
</tbody>
</table>
Chunking Clusters

Parallel in the observation cell
Iterate over $i$ and $j$ of the matrix
Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
Read all $316 \times ij$ into shared memory
Perform the 189 interactions for each observation cell
Can coalesce $M_j$ reads
Can prevent shared memory bank conflicts
Chunking Clusters

- Parallel in the observation cell
- Iterate over $i$ and $j$ of the matrix
Chunking Clusters

- Parallel in the observation cell
- Iterate over $i$ and $j$ of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all 316 $D_{ij}$ into shared memory
Parallel in the observation cell
Iterate over $i$ and $j$ of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all 316 $D_{ij}$ into shared memory
Chunking Clusters

- Parallel in the observation cell
- Iterate over $i$ and $j$ of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all 316 $D_{ij}$ into shared memory
  - Perform the 189 interactions for each observation cell
Chunking Clusters

- Parallel in the observation cell
- Iterate over $i$ and $j$ of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all 316 $D_{ij}$ into shared memory
  - Perform the 189 interactions for each observation cell
    - Can coalesce $M_j$ reads
    - Can prevent shared memory bank conflicts
**Chunking Clusters**

M-data for a sibling-index in 6x6x6 shared-memory array

<table>
<thead>
<tr>
<th>ζ=0</th>
<th>ζ=1</th>
<th>ζ=2</th>
<th>ζ=3</th>
<th>ζ=4</th>
<th>ζ=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>e f 0 1 2 3</td>
<td>2 3 4 5 6 7</td>
<td>6 7 8 9 a b</td>
<td>a b c d</td>
<td>e f 0 1 2 3</td>
<td>2 3 4 5 6 7</td>
</tr>
<tr>
<td>8 9 a b c d</td>
<td>c d e f 0 1</td>
<td>0 1 2 3 4 5</td>
<td>4 5 6 7 8 9</td>
<td>8 9 a b c d</td>
<td>c d e f 0 1</td>
</tr>
<tr>
<td>2 3 4 5 6 7</td>
<td>6 7 8 9 a b</td>
<td>a b c d e f</td>
<td>e f 0 1 2 3</td>
<td>2 3 4 5 6 7</td>
<td>6 7 8 9 a b</td>
</tr>
<tr>
<td>0 1 2 3 4 5</td>
<td>4 5 6 7 8 9</td>
<td>8 9 a b c d</td>
<td>c d e f 0 1</td>
<td>0 1 2 3 4 5</td>
<td>4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Bank numbers for 6x6 data in each ζ are given as follows:

- η=0: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1
- η=1: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1
- η=2: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1
- η=3: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1
- η=4: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1
- η=5: 0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1

Assign half-warps 0 to 3

**Parallel in the observation cell**

- Iterate over i and j of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all 316 $D_{ij}$ into shared memory
  - Perform the 189 interactions for each observation cell
    - Can coalesce $M_j$ reads
    - Can prevent shared memory bank conflicts
### Chunking Clusters

<table>
<thead>
<tr>
<th></th>
<th>per chunk of size $B$</th>
<th>per observation cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read M-data</td>
<td>[word] $(2B + 4)^3 \cdot r \cdot P$</td>
<td>$\frac{(B+2)^3 P}{B^3} r$</td>
</tr>
<tr>
<td>Read D-data</td>
<td>[word] $316 \cdot r \cdot \frac{r}{P} \cdot P$</td>
<td>$\frac{316 P^2}{8B^3} r^2$</td>
</tr>
<tr>
<td>Read/Write L-data</td>
<td>[word] $2 \cdot 8B^3 \cdot r \cdot \frac{r}{P} \cdot P$</td>
<td>$2r^2$</td>
</tr>
<tr>
<td>Operation counts</td>
<td>[flop] $8B^3 \cdot 189 \cdot \frac{r}{P} (2r - 1) \cdot P$</td>
<td>$189r(2r - 1)$</td>
</tr>
<tr>
<td>Flop-to-word ratio</td>
<td>[flop/word] $108$ for $r = 32, B = 4, P = 8$</td>
<td>$133$ for $r = 256, B = 4, P = 16$</td>
</tr>
</tbody>
</table>

- Parallel in the observation cell
- Iterate over $i$ and $j$ of the matrix
  - Read sibling-equivalent $M_j$ into shared memory (plus ghosts)
  - Read all $316 D_{ij}$ into shared memory
  - Perform the 189 interactions for each observation cell
    - Can coalesce $M_j$ reads
    - Can prevent shared memory bank conflicts
FMM Performance

![Graphs showing FMM performance on GPU, comparing GPU1, GPU2, GPU3, GPU4, and CPU (8 cores). The graphs illustrate the performance of M2L kernels over varying N values, with the y-axis representing performance in GFLOPS.]
Summary

- Domain specific language which is connectivity and platform aware.
Summary

- Domain specific language which is connectivity and platform aware.
- By providing a feature set and restrictions, it is analyzable.
Summary

- Domain specific language which is connectivity and platform aware.
- By providing a feature set and restrictions, it is analyzable.
- FEM assemblies on GPU which are connectivity aware and kernel independent.
Summary

- Domain specific language which is connectivity and platform aware.
- By providing a feature set and restrictions, it is analyzable.

- FEM assemblies on GPU which are connectivity aware and kernel independent.
- FMM M2L computations on GPU designed to address connectivity and remain kernel independent.