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**

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

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?

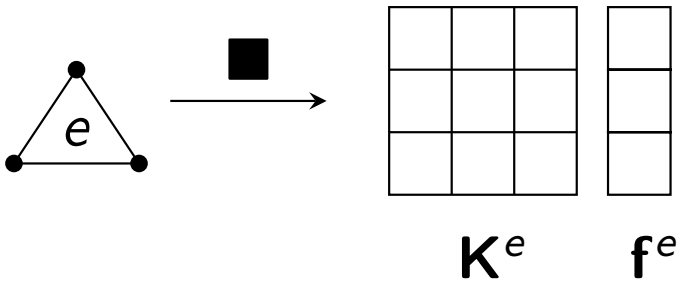
- Sparse Linear Algebra coming of age on GPU.
  - Extensive research on Sparse Solvers on GPU.
  - Extensive research on SpMV.
- Non-linear and time-dependent problems require many assembly procedures.
- Want to use topology efficiently with a black-box element kernel.

- 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:

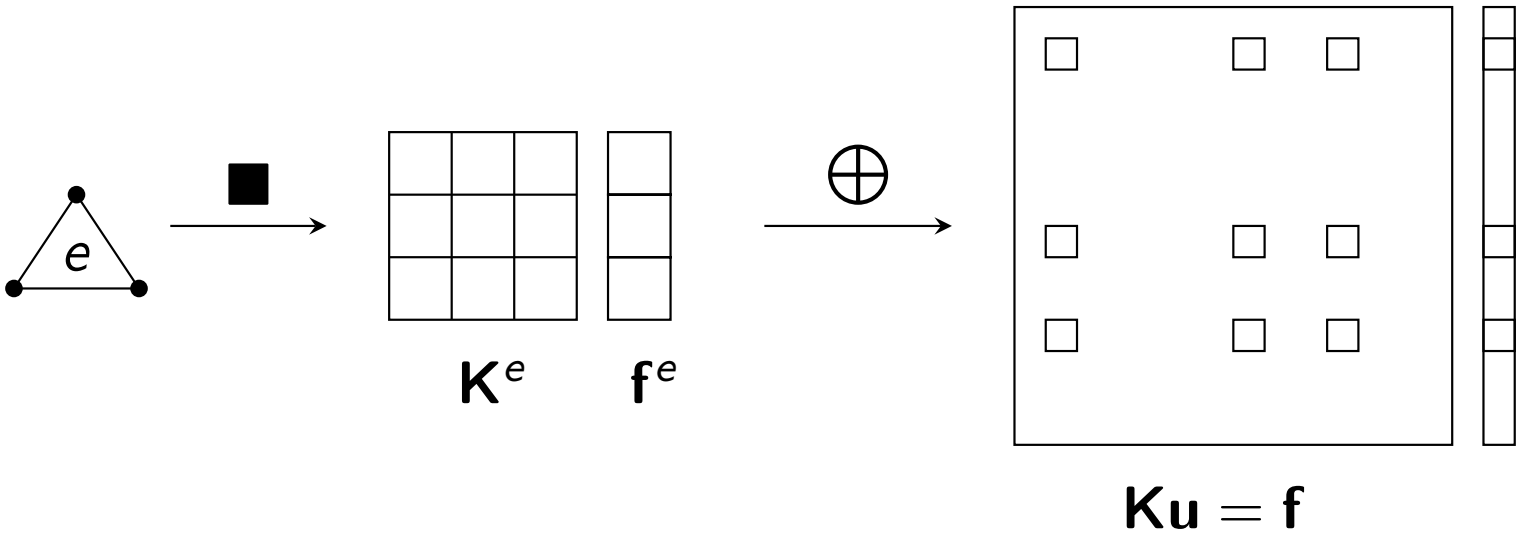
- Compute element data.
  - One by one.



FEM Direct Assembly

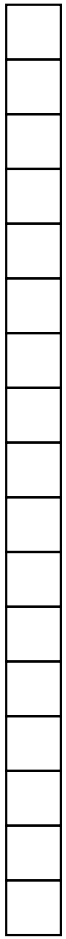
Most common FEM assembly procedure:

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

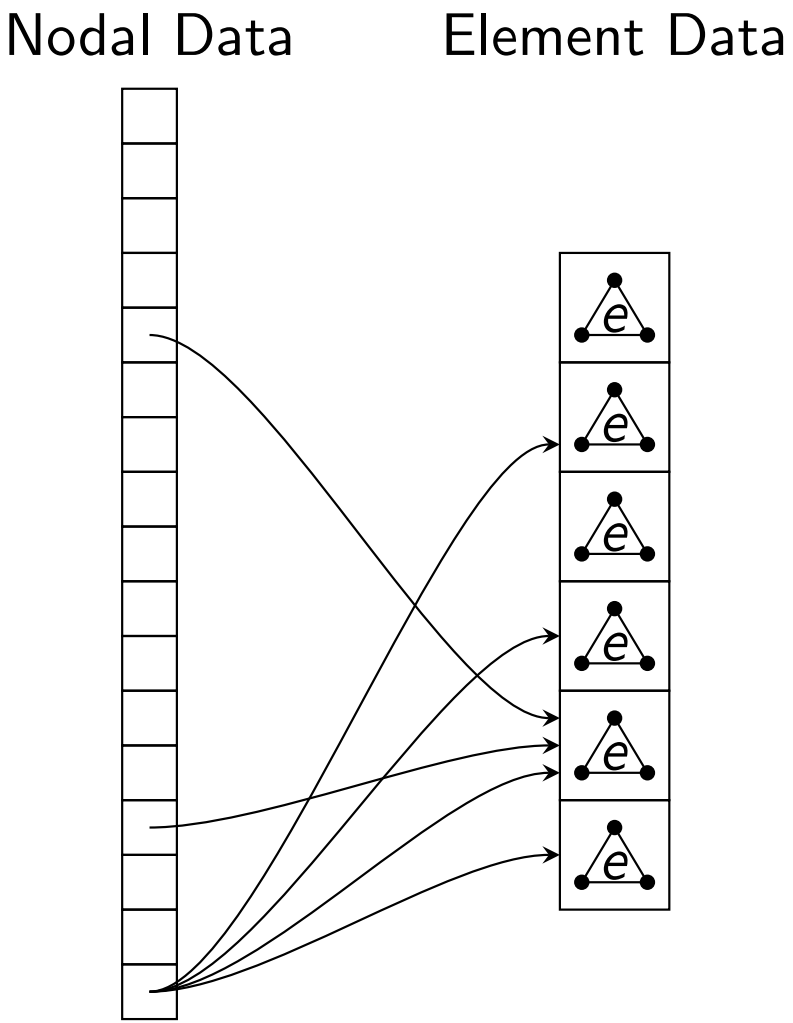


Data Flow

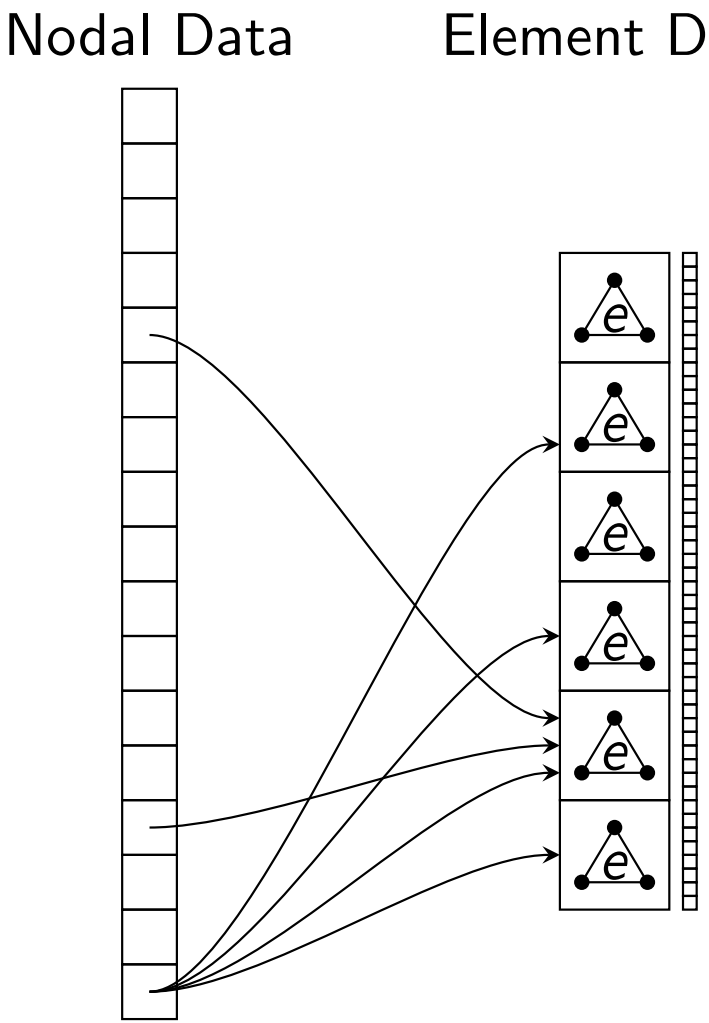
Nodal Data



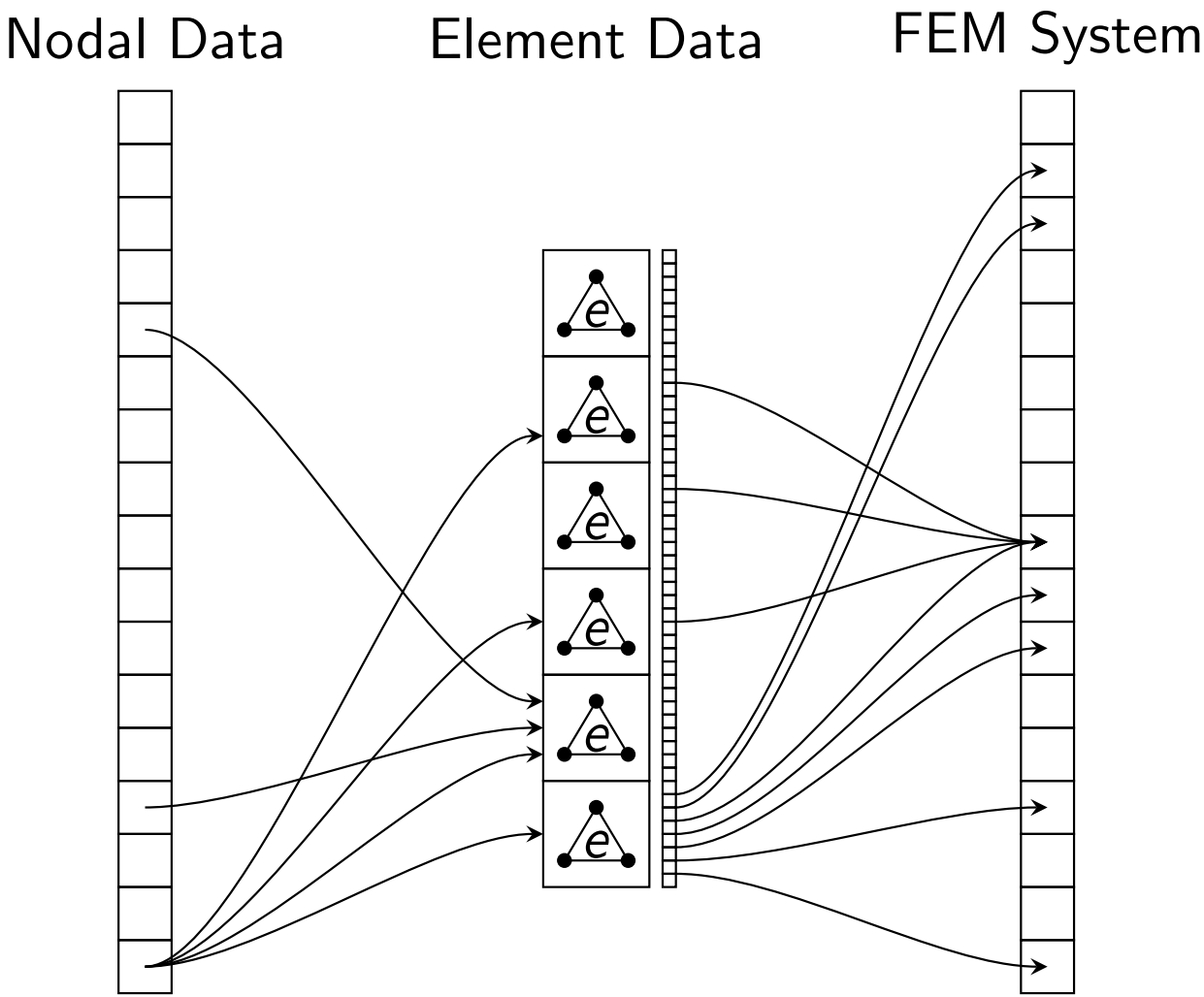
Data Flow



Data Flow



Data Flow



GPU FEM Assembly Strategies

Two Key Choices:

Store Element Data In

Threads Assemble By

GPU FEM Assembly Strategies

Two Key Choices:

Store Element Data In

- Global Memory
- Local Memory
- Shared Memory

Threads Assemble By

GPU FEM Assembly Strategies

Two Key Choices:

Store Element Data In

- Global Memory

  Computation / Assembly.
  Min computation.
- Local Memory
- Shared Memory

Threads Assemble By

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

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

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)
- 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
- 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

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
  - Race conditions.

Local-Element

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

Local-Element

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

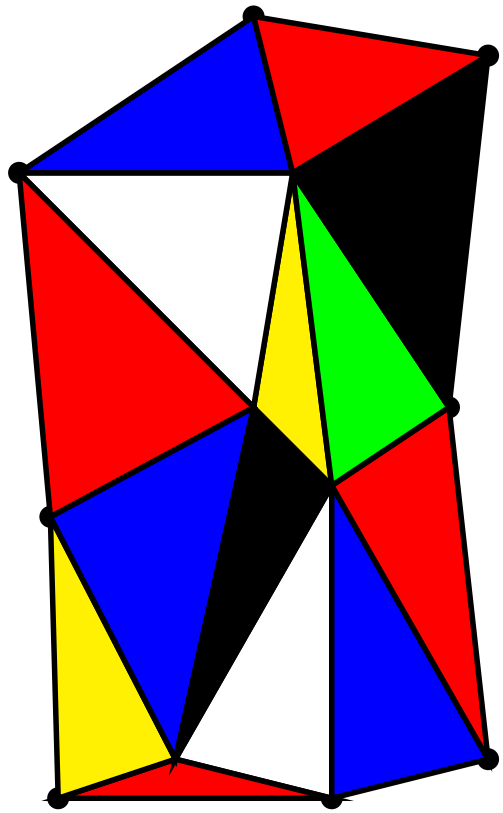
Race conditions still possible!

Local-Element - Coloring the Mesh

- 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.



Local-Element - Coloring the Mesh

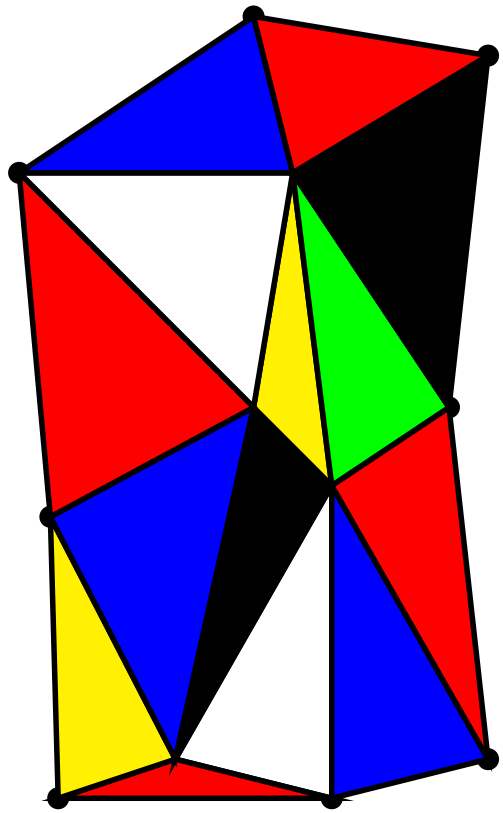
- Assign one thread to one element.
  - Compute the element data.
  - Assemble directly into system.

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.

Global-NZ

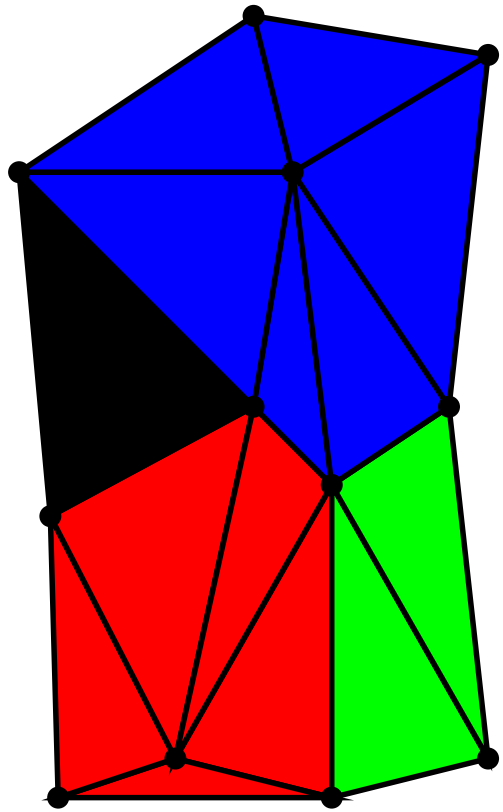
- Kernel1 - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.
- Kernel2 - Assign one thread to one NZ.
  - Assemble from global memory.

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

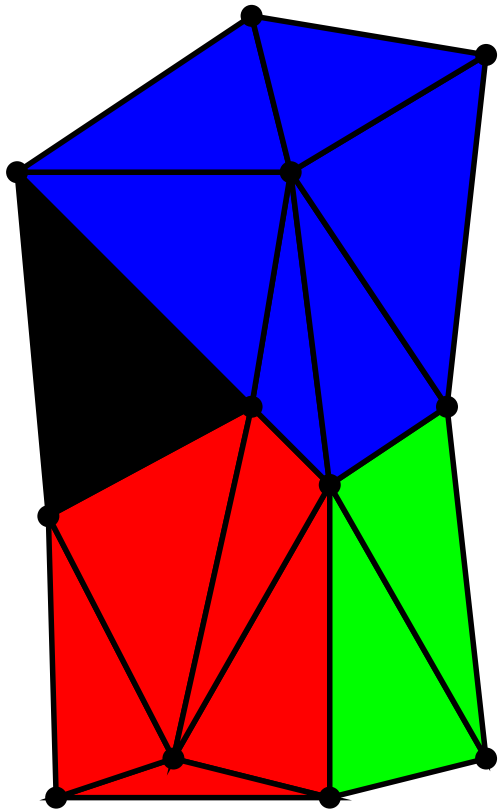
- Kernel1 - Assign one thread to one element.
  - Compute the element data.
  - Store element data in global memory.
- Kernel2 - Assign one thread to one NZ.
  - Assemble from global memory.

Optimizing:

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

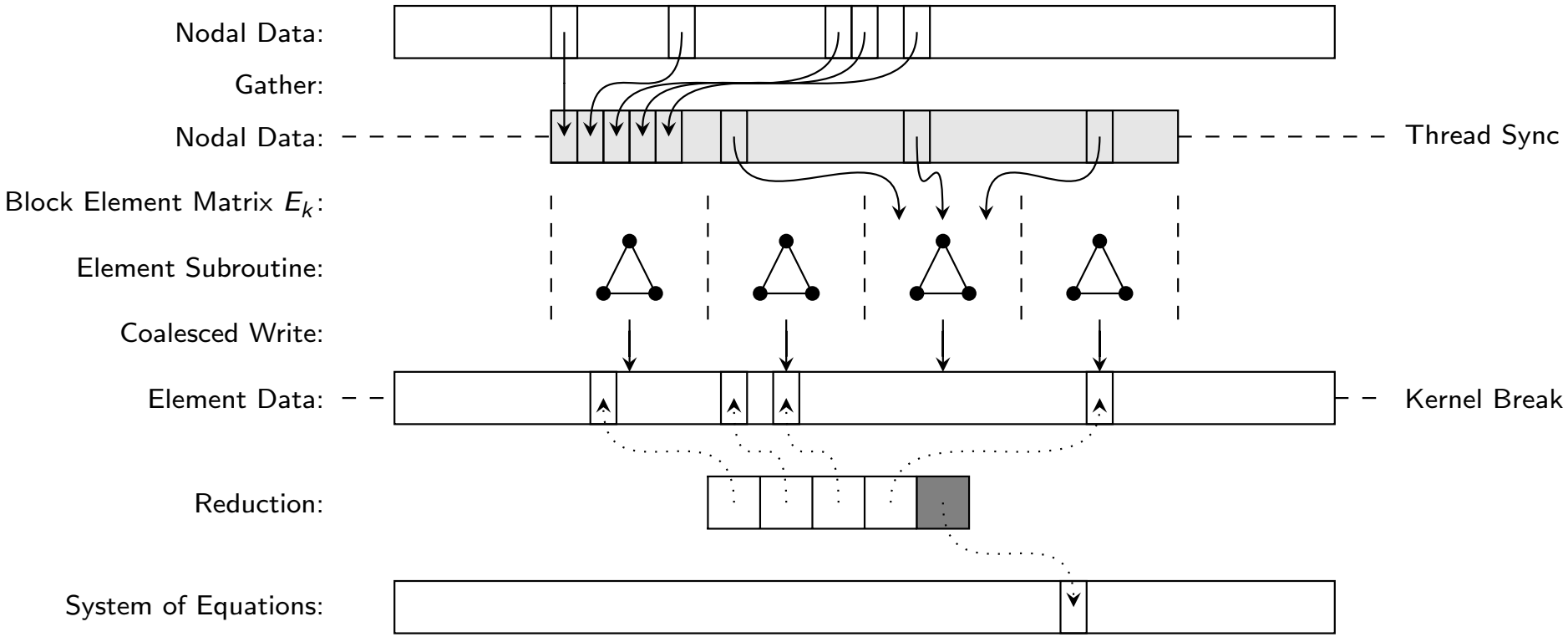
Problems.

- Extra passes through global memory.



Global-NZ Data Flow

The optimized algorithm looks like:



Shared-NZ

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

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.

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.

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

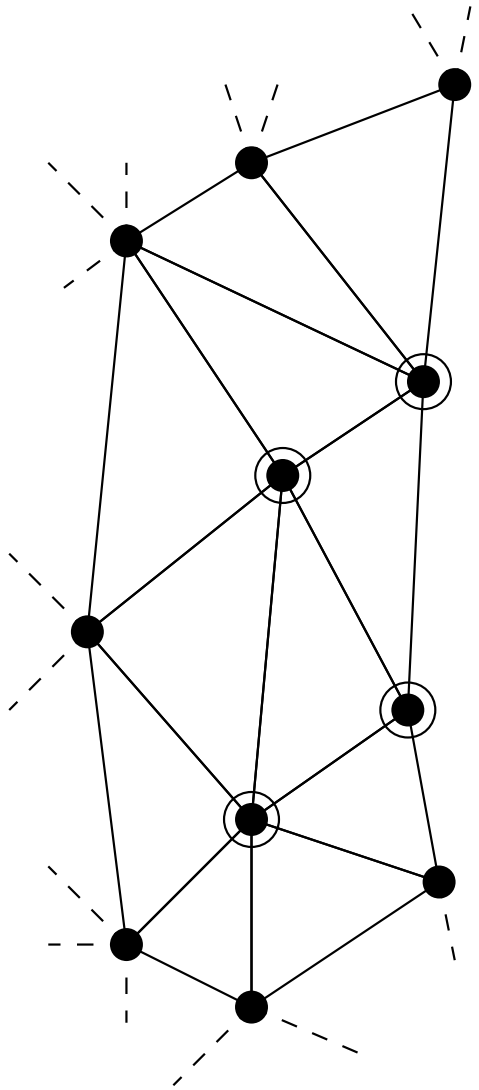
- Assign one thread to one element.
  - Compute the element data.
  - Store element data in shared memory.
- Reassign threads to NZs.
  - Assemble from shared memory.

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.



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.

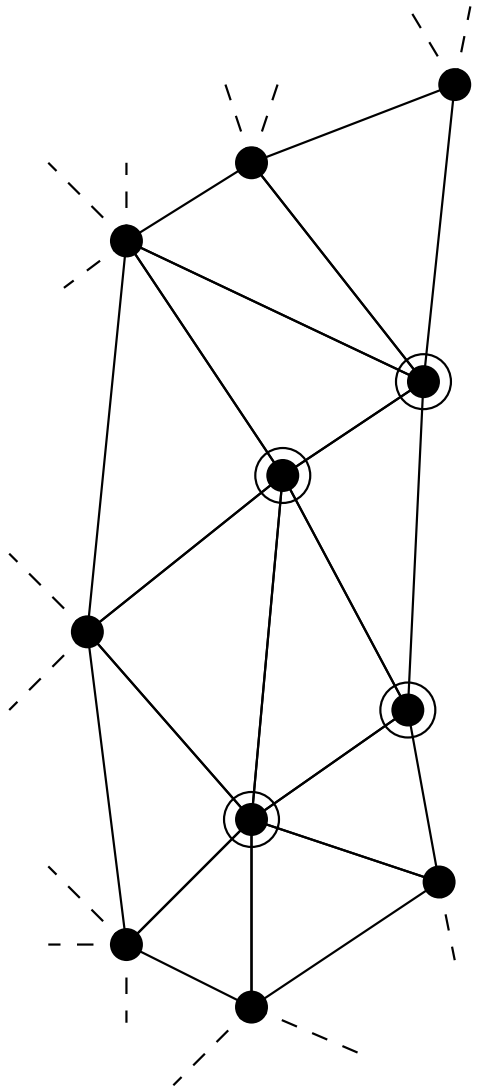
- Must compute all "halo" element data.

A set of elements requires a set of nodes.

- Must gather all "halo" nodal data.

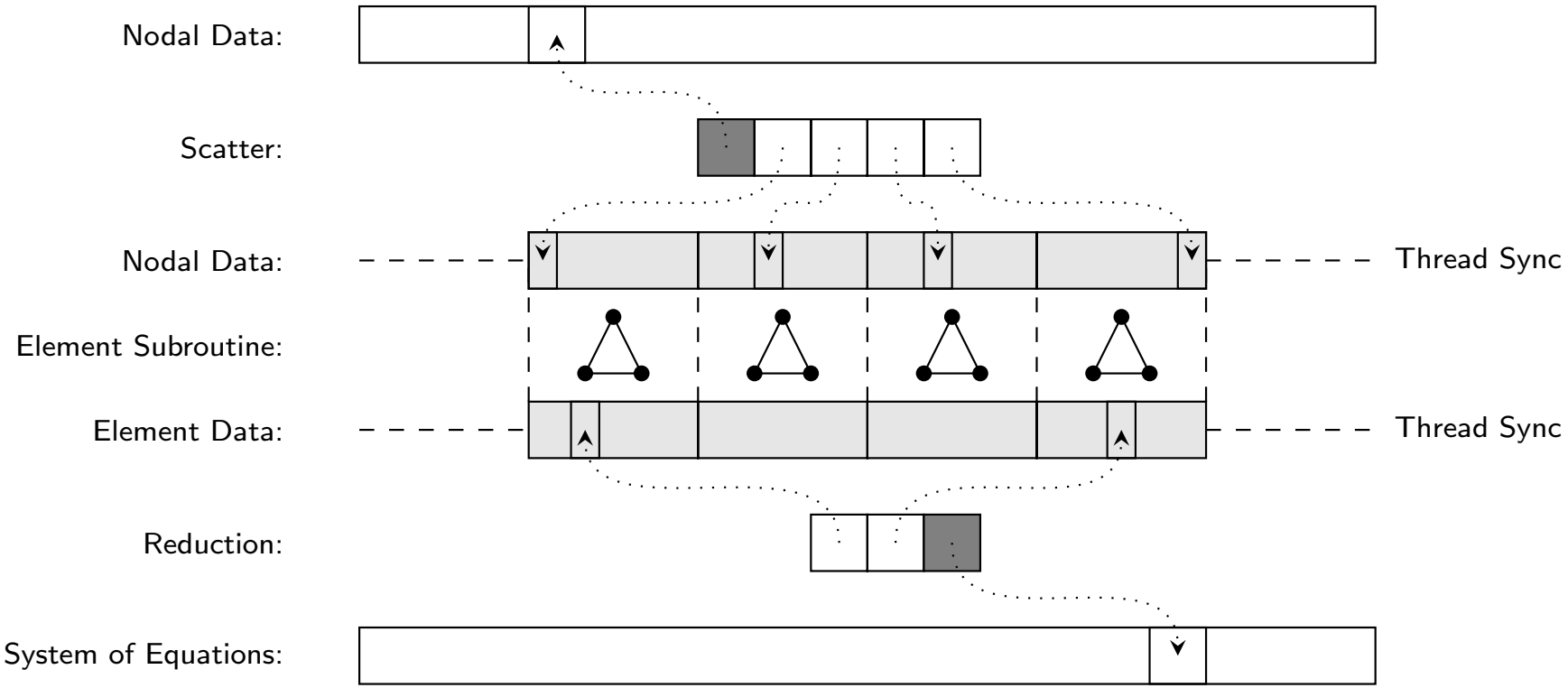
Problems.

- Shared memory size is very limiting.



Shared-NZ Data Flow

The optimized algorithm looks like:



Scatter and Reduction Arrays

General procedure:

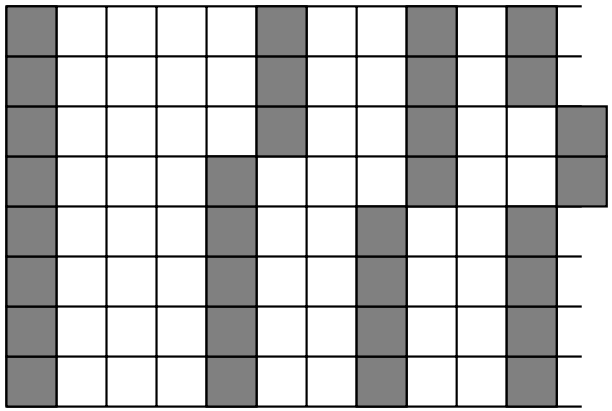
- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.

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:

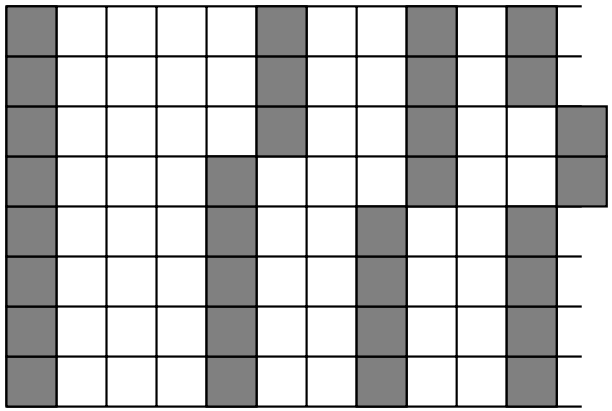


Scatter and Reduction Arrays

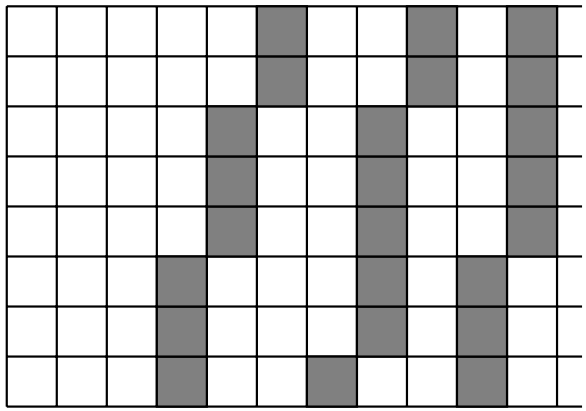
General procedure:

- 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:

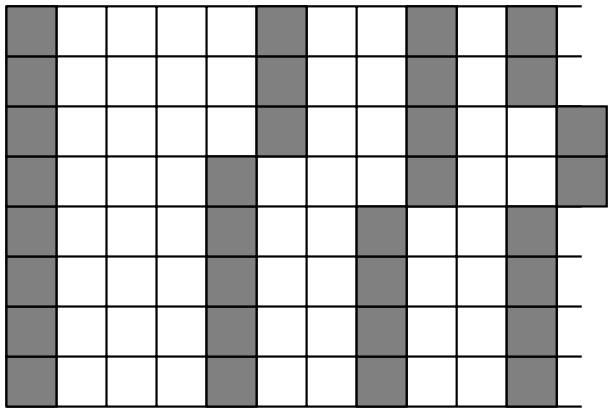


Scatter and Reduction Arrays

General procedure:

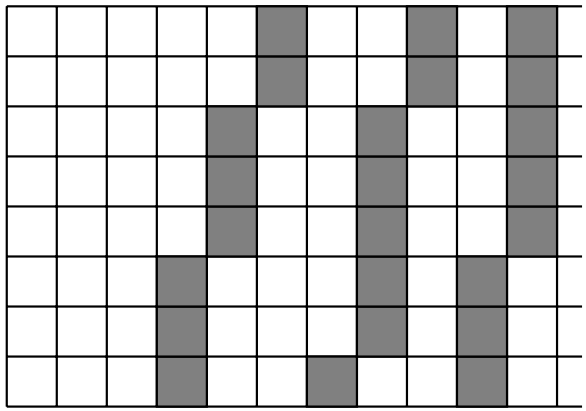
- Make a set of operations to be done for each partition.
- Pack these into an array such that reading is coalesced.

Scatter Array:



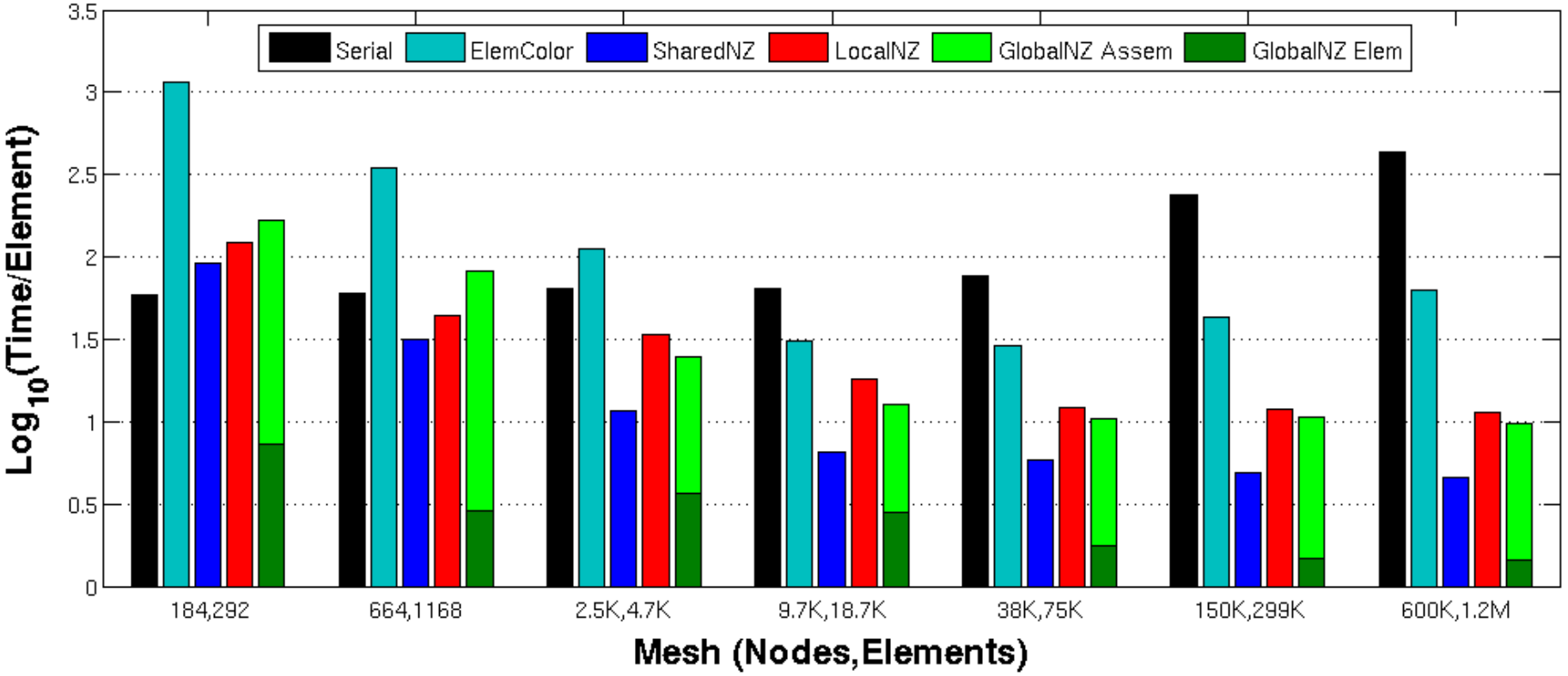
- Very fast.
- Highly adaptable.

Reduction Array:

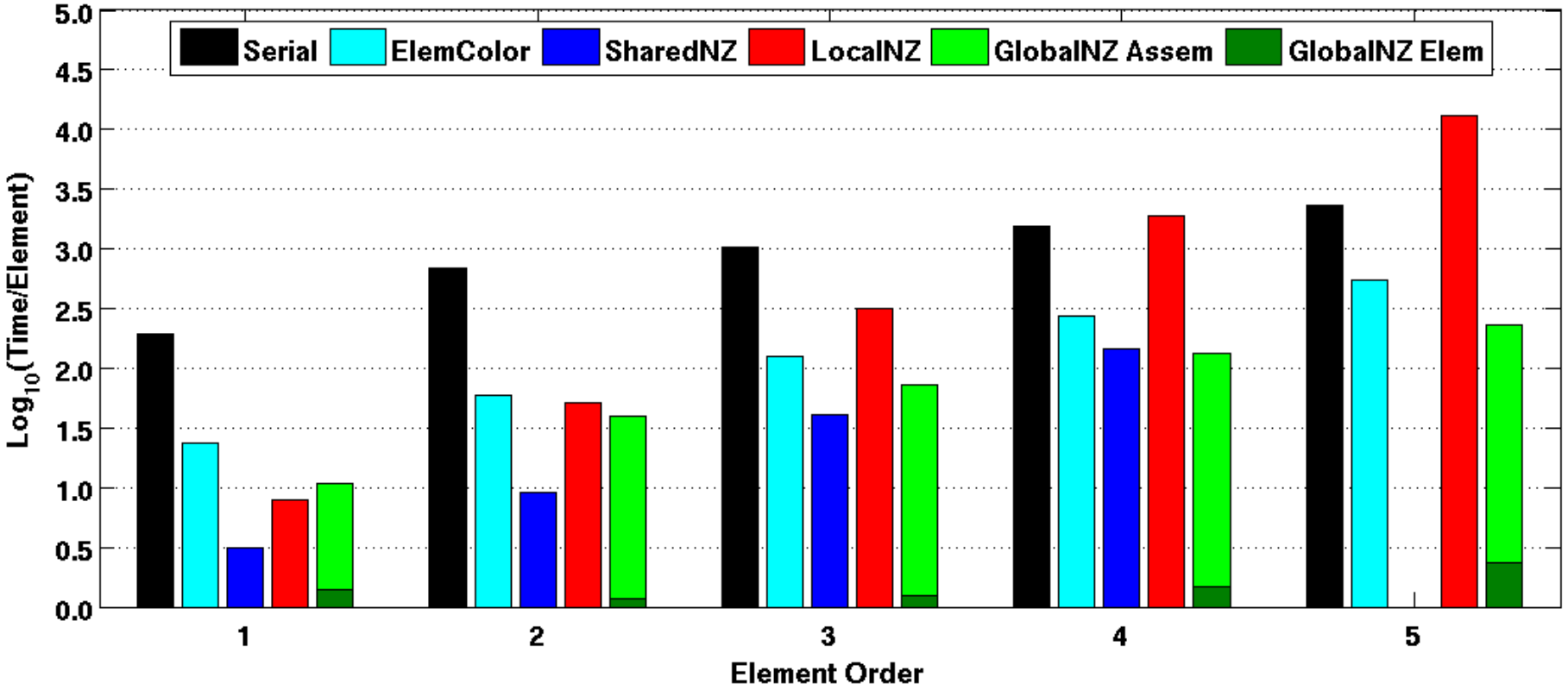


- Significant setup cost.
- Significant memory cost.

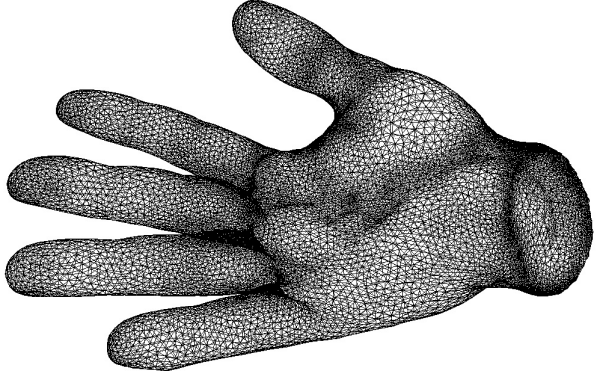
Scaling with Element Number



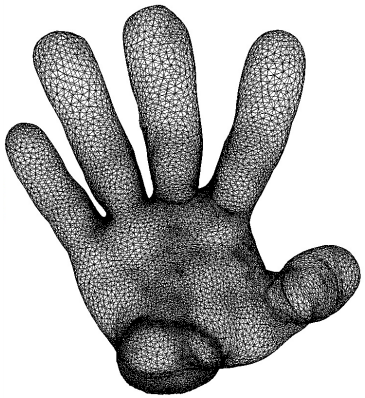
Scaling with Element Order



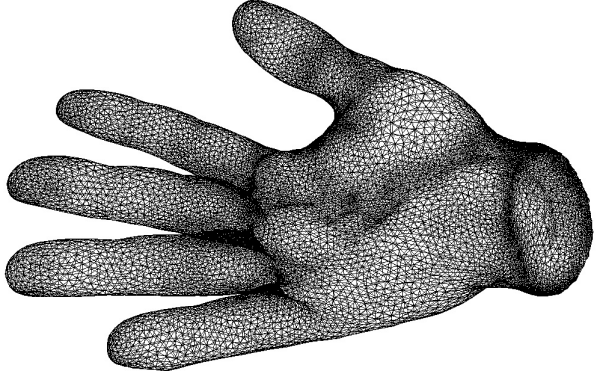
Application



GPU non-linear neoHookean model.

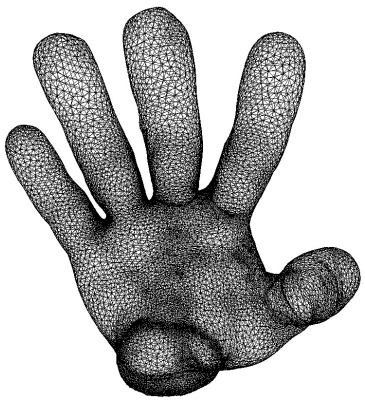


Application

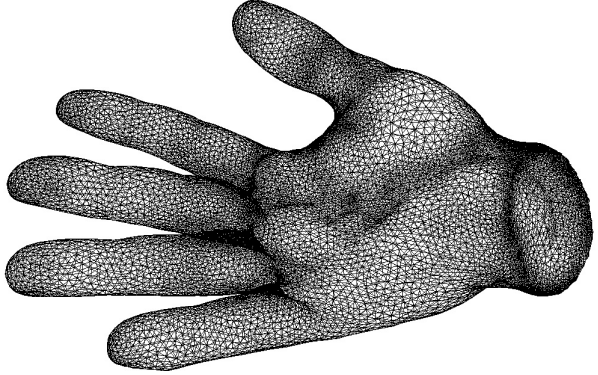


GPU non-linear neoHookean model.

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

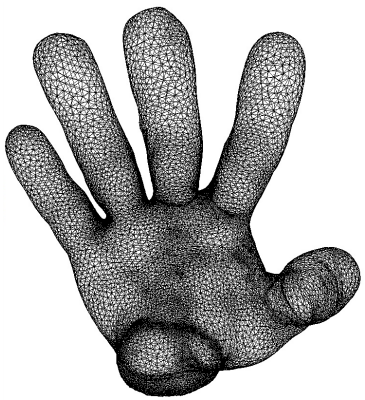


Application

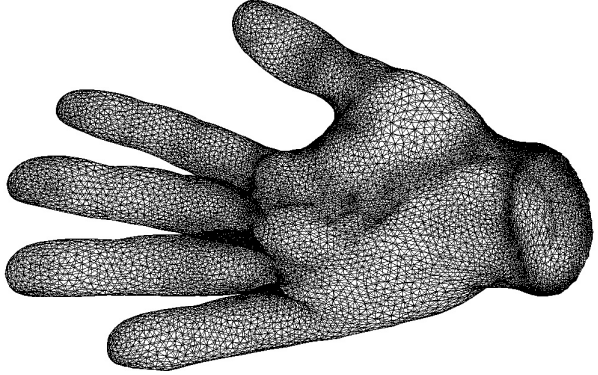


GPU non-linear neoHookean model.

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

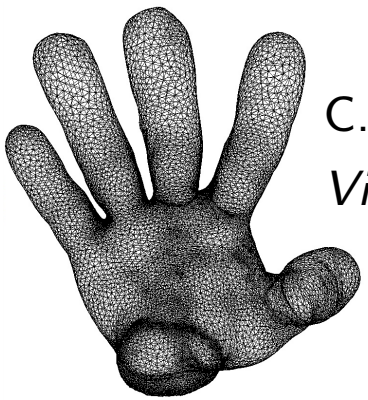


Application



GPU non-linear neoHookean model.

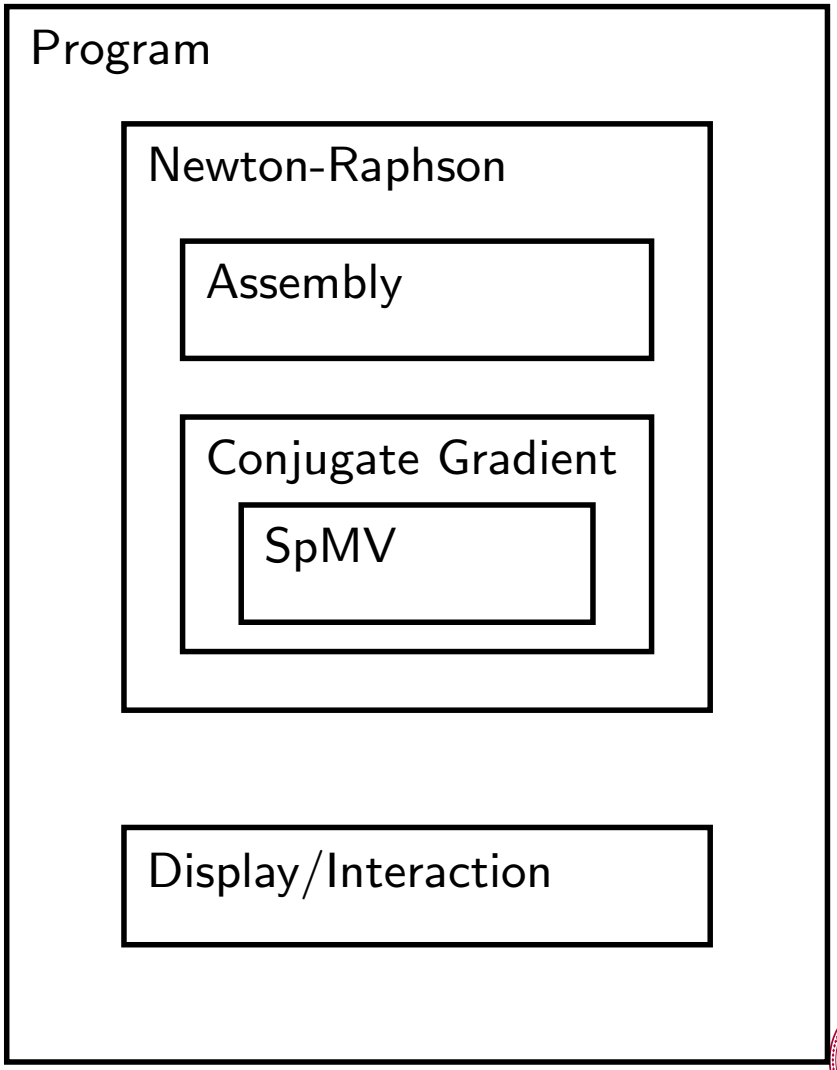
- Newton-Raphson update at each step.
- Assemble, solve, update, and render at each step.
- 28,796 Nodes. 125,127 Elements.



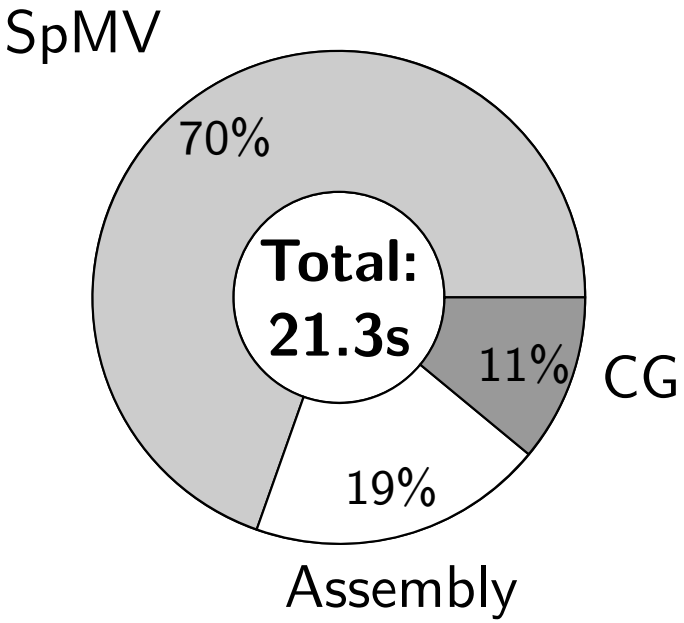
C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of FEM on GPUs.* GPU Gems. Preprint.

Implementation

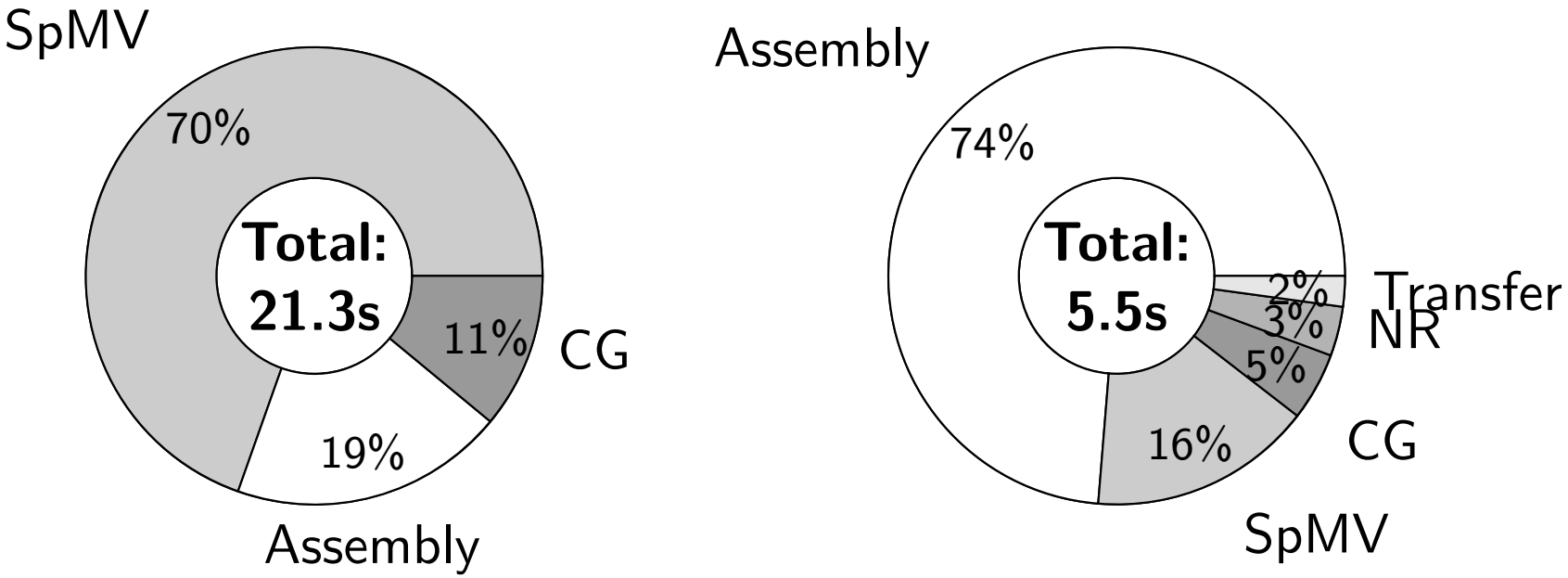
- Newton-Raphson
  - `NR_CPU`
  - `NR_GPU`
- Assembly
  - `AssemblyCPU`
  - `AssemblyCPU_Opt`
  - `AssemblyGlobalNZ`
  - `AssemblySharedNZ`
- Conjugate Gradient
  - `CG_CPU`
  - `CG_GPU`
  - `DCG_CPU`
  - `DCG_GPU`
- Sparse Matrix Format
  - `COO_Matrix`
  - `CSR_Matrix`
  - `HYB_Matrix`
  - `DCOO_Matrix`
  - `DCSR_Matrix`
  - `DHYB_Matrix`
- All SpMVs on CPU and GPU.



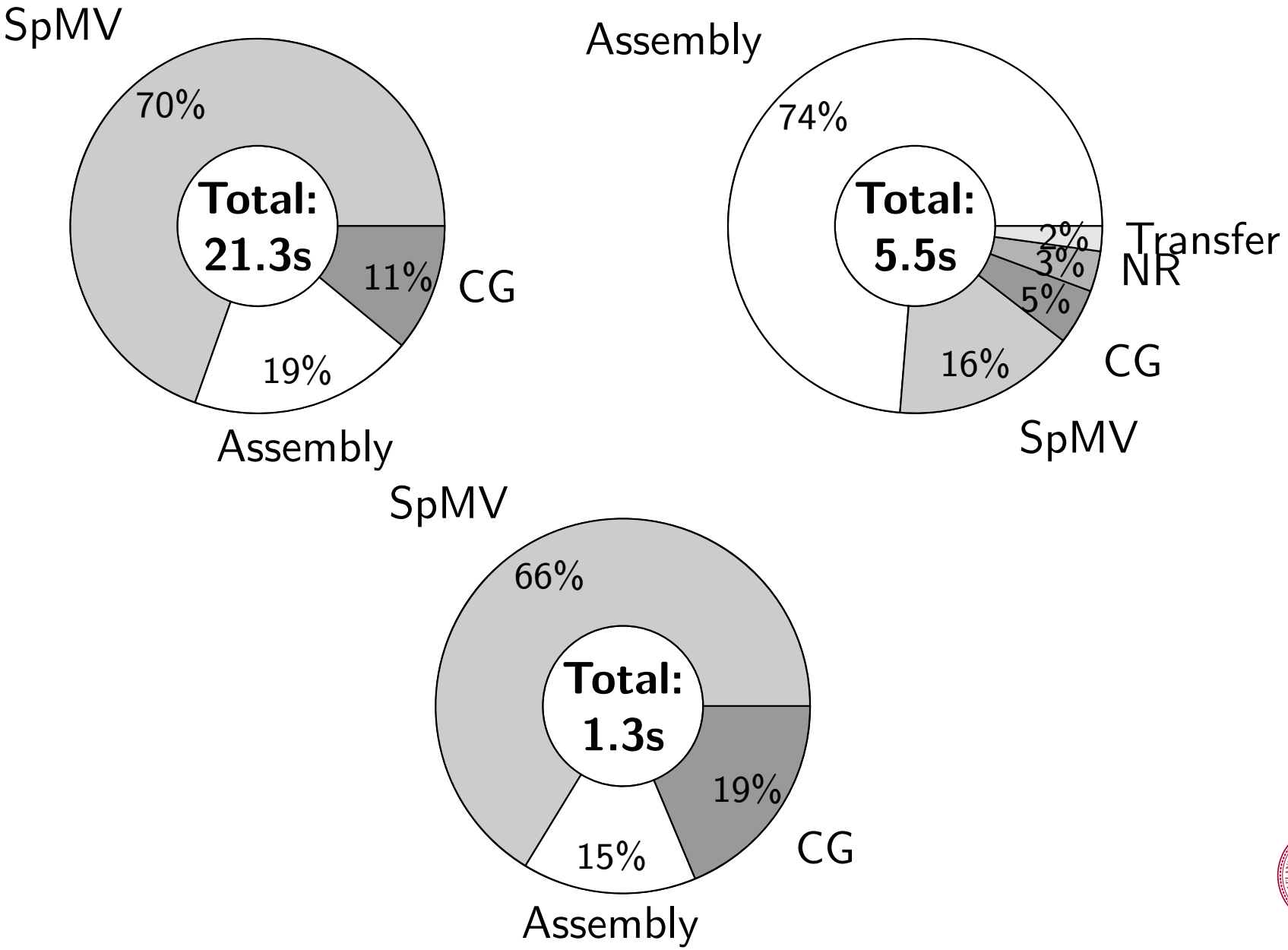
Improvement



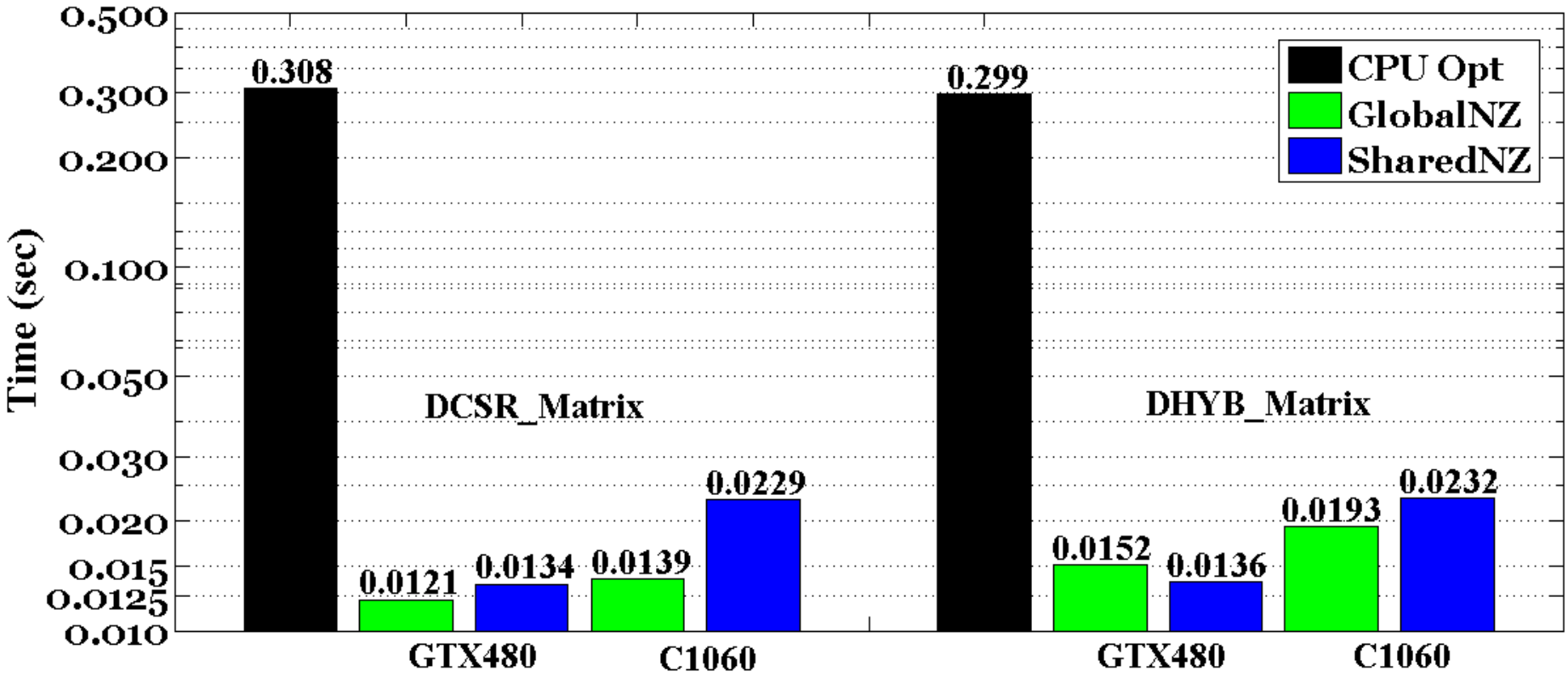
Improvement



Improvement



Assembly Speed



FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

- Create and classify several GPU FEM assembly algorithms.

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

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

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

- 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.

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

- 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.

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

FEM Conclusion

C. Cecka, A. Lew, E. Darve, *Assembly of Finite Element Methods on Graphics Processors*. IJNME, 2009.

- 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.

C. Cecka, A. Lew, E. Darve. *Application of Assembly, Solution, and Visualization of Finite Elements on Graphics Processors.* GPU Gems. Preprint.

- Applying the methods to a high-performance FEM application.

FMM GPU Algorithms

- 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

FMM GPU Algorithms

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

- M2L stage:

$$\mathbf{L}(O) := \sum_{S \in \mathcal{I}(O)} \mathbf{D}^{(O,S)}\, \mathbf{M}(S)$$

FMM GPU Algorithms

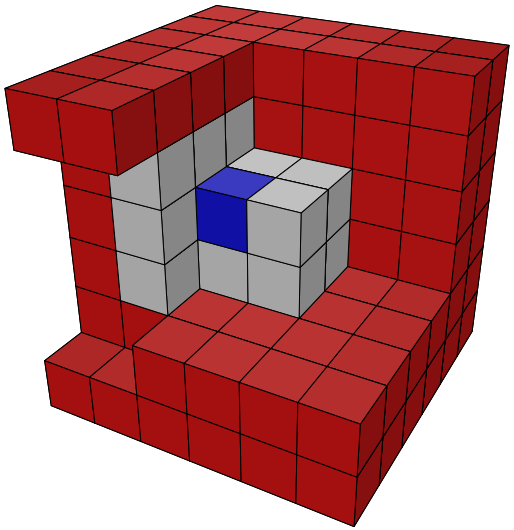
- Implement the M2L operation of a generalized FMM
  - Know the structure and topology
  - Leave the transfer matrix as arbitrary

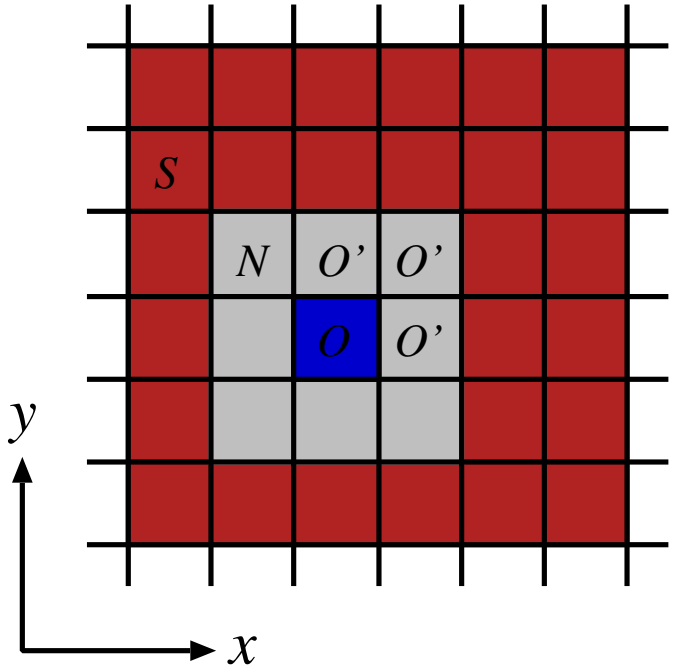
- M2L stage:

$$\mathbf{L}(O) := \sum_{S \in \mathcal{I}(O)} \mathbf{D}^{(O,S)}\, \mathbf{M}(S)$$

  - 316 Transfer Matrices
  - $|\mathcal{I}(O)| \leq 189$

FMM GPU Algorithms





Basic Algorithm

Block = Observation Cell O

Thread = $\mathbf{L}_i$

Basic Algorithm

Block = Observation Cell O

Thread = $\mathbf{L}_i$

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

Basic Algorithm

Block = Observation Cell O

Thread = $\mathbf{L}_i$

For all $S \in \mathcal{I}(O)$
    Read $\mathbf{D}^(S, O)$
    Read $\mathbf{M}(S)$

Basic Algorithm

Block = Observation Cell O

Thread = $\mathbf{L}_i$

For all $S \in \mathcal{I}(O)$
    Read $\mathbf{D}^(S, O)$
    Read $\mathbf{M}(S)$
    $\mathbf{L}_i += \mathbf{D}_{ij}^{(O,S)} \mathbf{M}_j(S)$
Write $\mathbf{L}_i$

Basic Algorithm

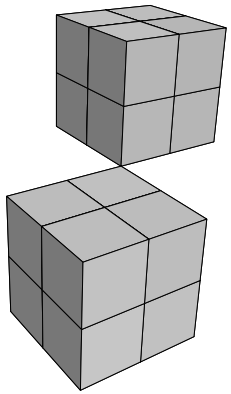
Block = Observation Cell O

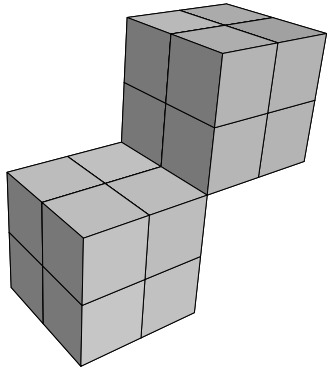
Thread = $\mathbf{L}_i$

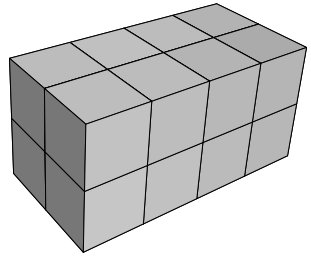
For all $S \in \mathcal{I}(O)$
  Read $\mathbf{D}^(S, O)$
  Read $\mathbf{M}(S)$
  $\mathbf{L}_i += \mathbf{D}_{ij}^{(O,S)} \mathbf{M}_j(S)$
Write $\mathbf{L}_i$

		per observation cell
Read M-data	[word]	$189r$
Read D-data	[word]	$189r^2$
Write L-data	[word]	r
Operation counts	[flop]	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	1.9 for $r = 32$ $(n = 4)$
		2.0 for $r = 256$ $(n = 8)$

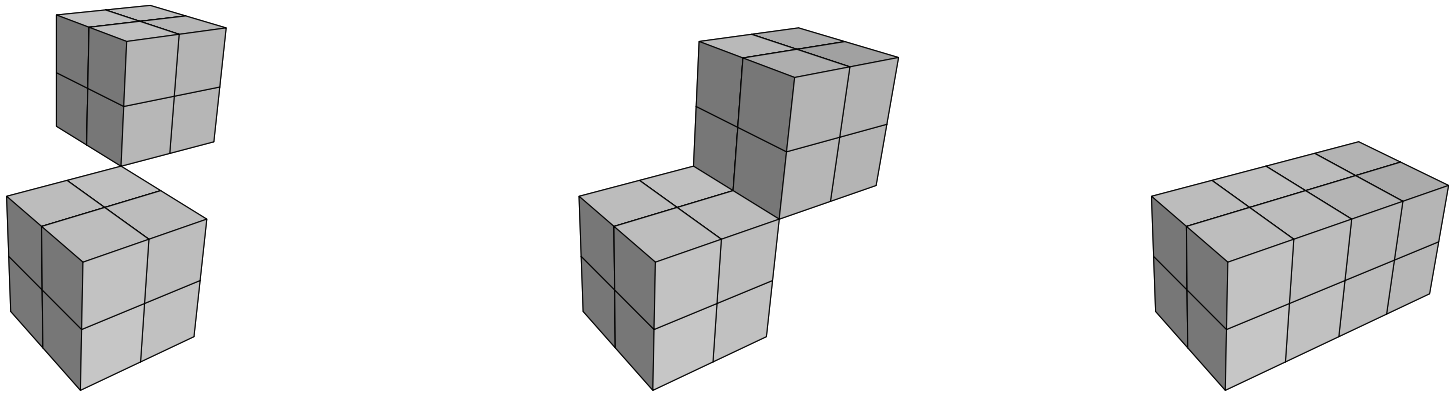
Blocking Siblings







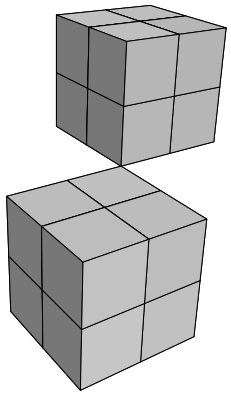
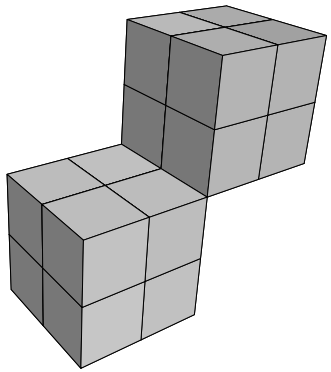
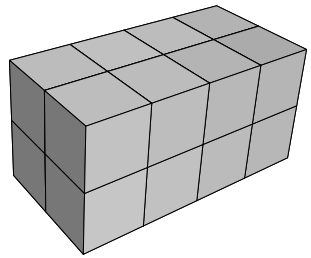
Blocking Siblings



- Iterate over clusters instead of cells

		Per observation cluster	Per observation cell
Read M-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Read D-data	[word]	$26 \cdot 27 \cdot r^2$	$\frac{26 \cdot 27}{8} r^2$
Read/Write L-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Operation counts	[flop]	$8 \cdot 189 \cdot r(2r-1)$	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	4.2 for $r = 32$ 4.3 for $r = 256$	

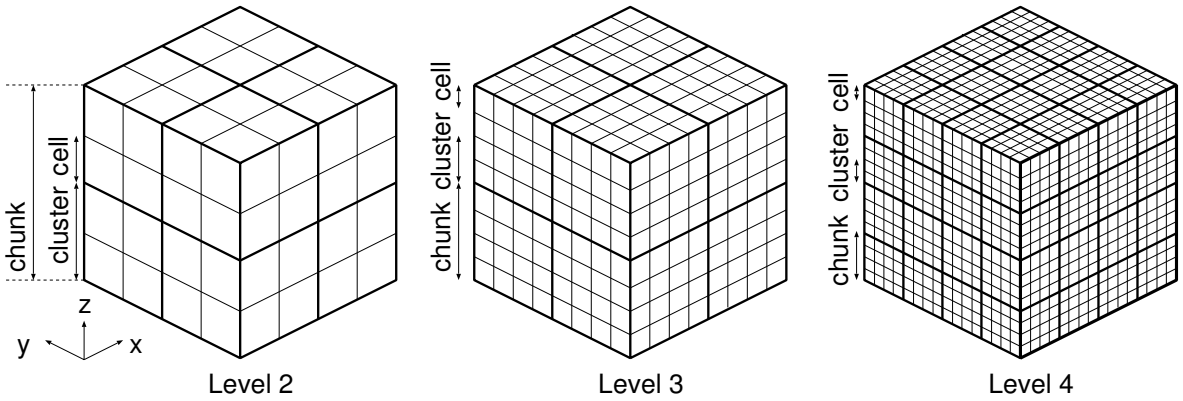
Blocking Siblings

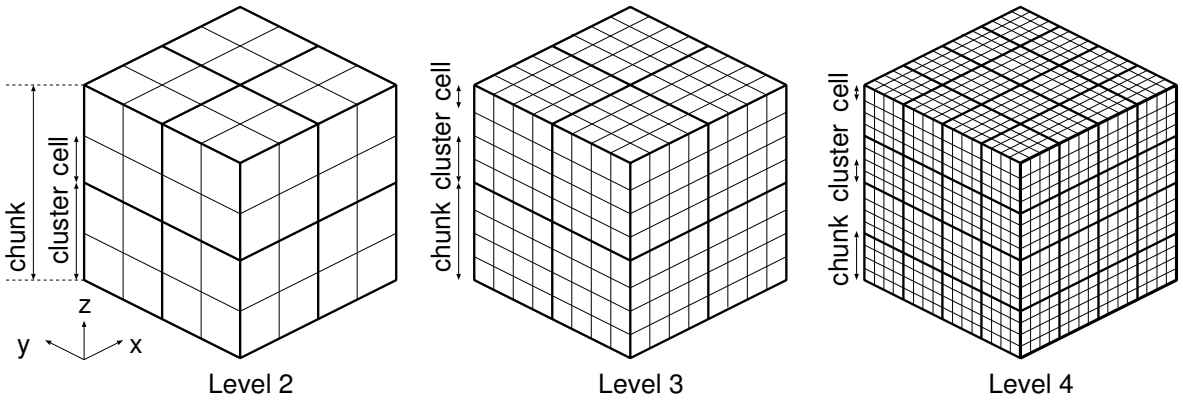
- Iterate over clusters instead of cells
  - Iterate over transfer class
  - Reuse the **D** matrix

		Per observation cluster	Per observation cell
Read M-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Read D-data	[word]	$26 \cdot 27 \cdot r^2$	$\frac{26 \cdot 27}{8} r^2$
Read/Write L-data	[word]	$8 \cdot 26 \cdot r$	$26r$
Operation counts	[flop]	$8 \cdot 189 \cdot r(2r-1)$	$189r(2r-1)$
Flop-to-word ratio	[flop/word]	4.2 for $r = 32$ 4.3 for $r = 256$	

Chunking Clusters

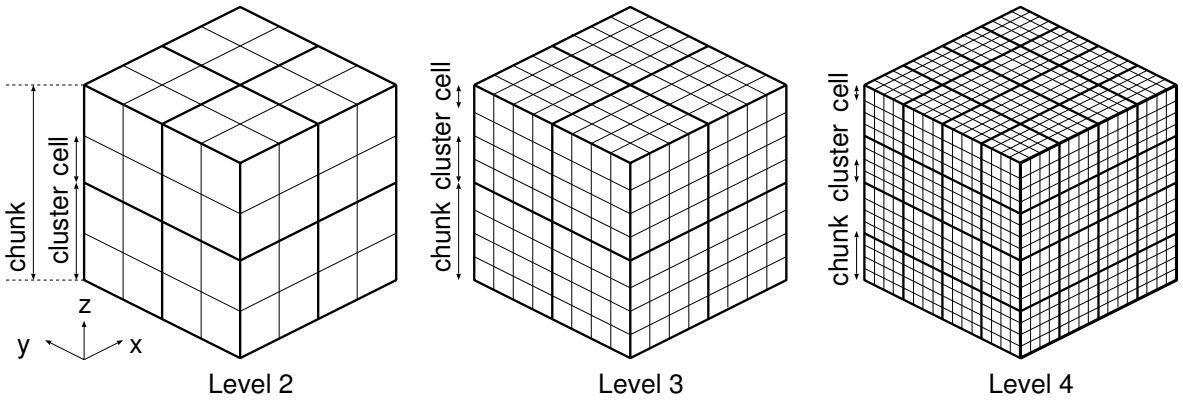


Chunking Clusters



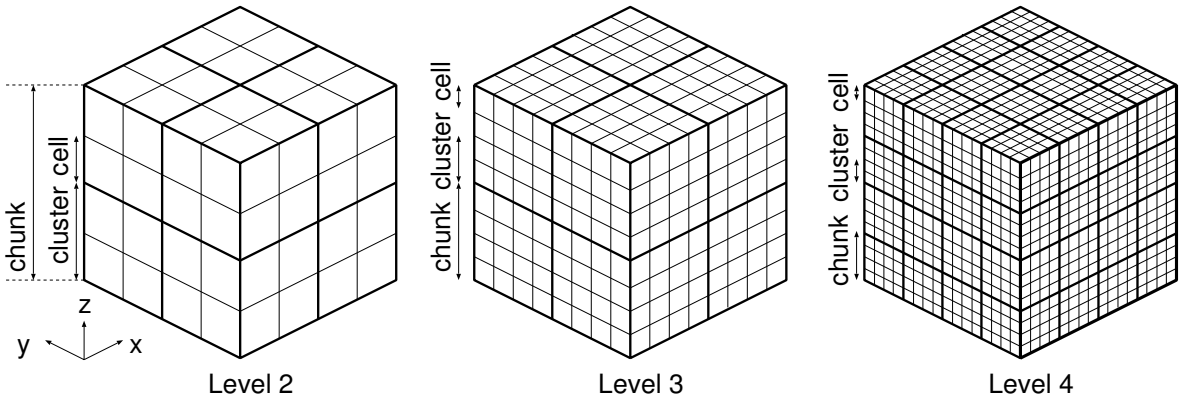
- Parallel in the observation cell
- Iterate over i and j of the matrix

Chunking Clusters



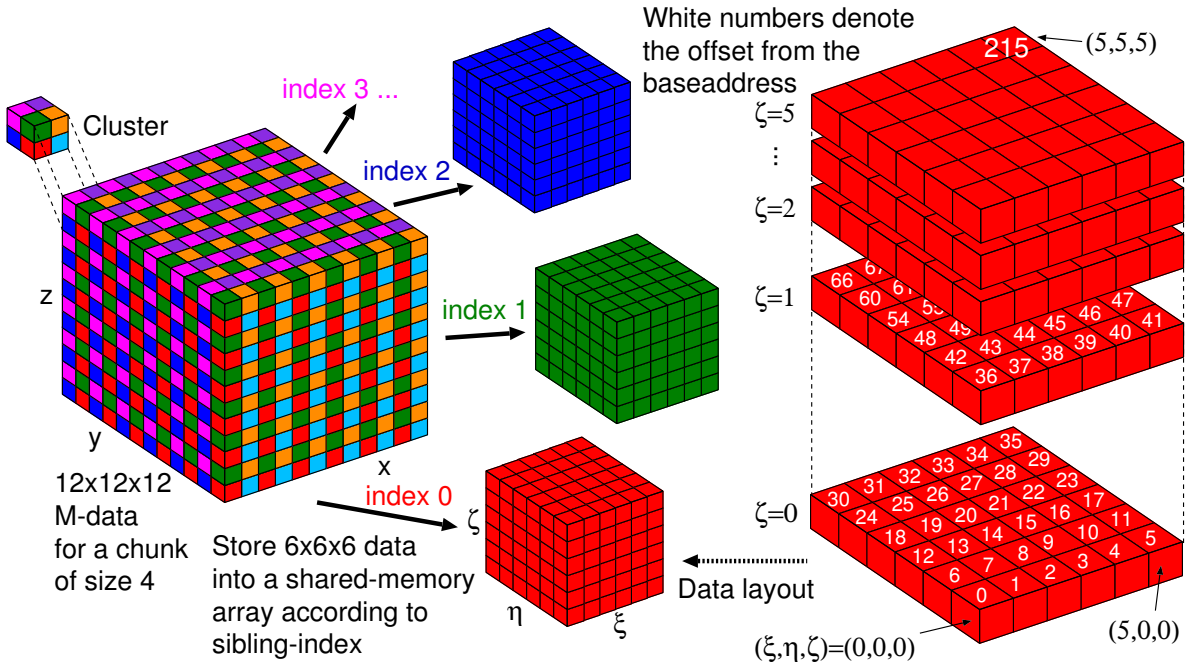
- Parallel in the observation cell
- Iterate over i and j of the matrix
  - Read sibling-equivalent M_j into shared memory (plus ghosts)
  - Read all 316 D_{ij} into shared memory

Chunking Clusters



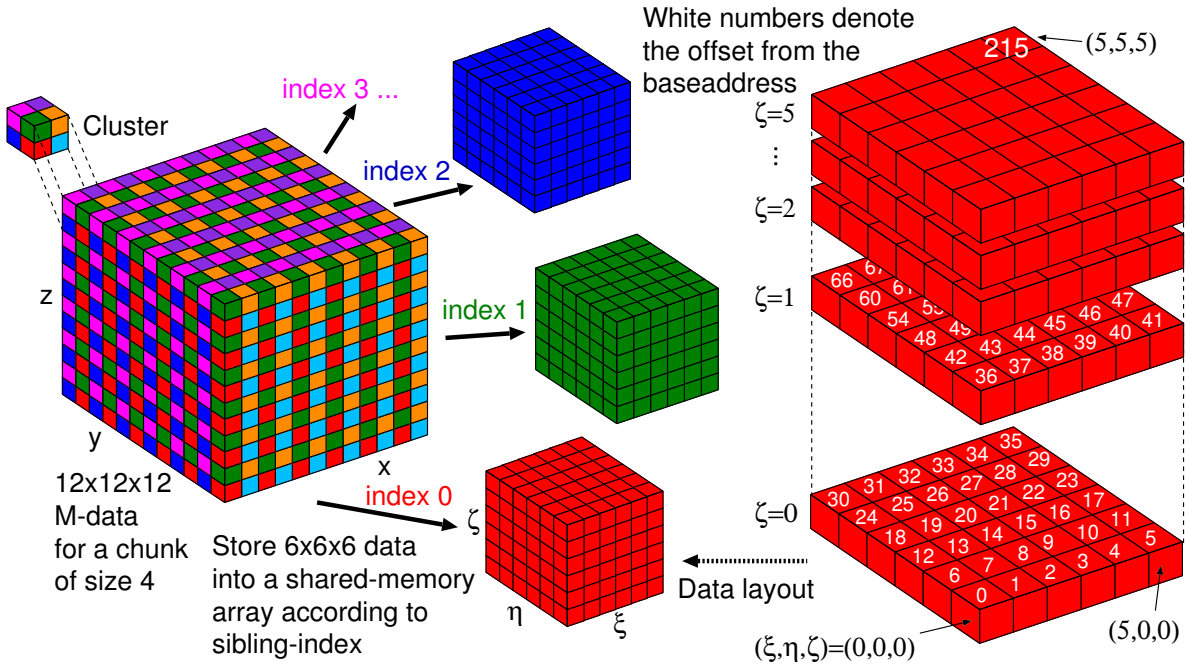
- Parallel in the observation cell
- Iterate over i and j of the matrix
  - Read sibling-equivalent M_j into shared memory (plus ghosts)
  - Read all 316 D_{ij} into shared memory

Chunking Clusters



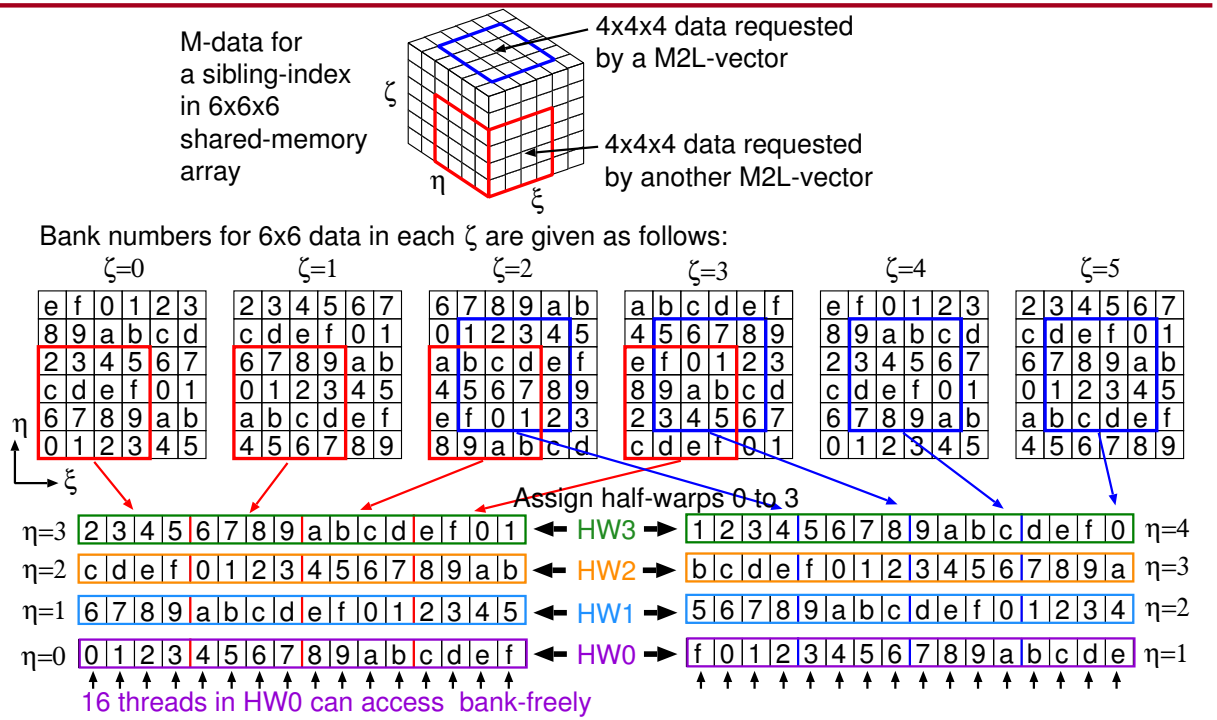
- Parallel in the observation cell
- Iterate over i and j of the matrix
  - Read sibling-equivalent M_j into shared memory (plus ghosts)
  - Read all 316 D_{ij} into shared memory
  - Perform the 189 interactions for each observation cell

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



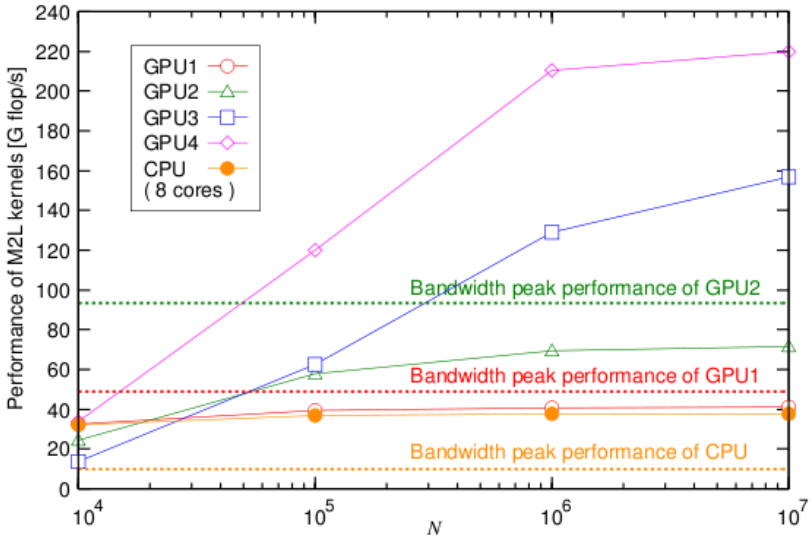
- Parallel in the observation cell
- Iterate over i and j of the matrix
  - Read sibling-equivalent M_j into shared memory (plus ghosts)
  - Read all 316 D_{ij} into shared memory
  - Perform the 189 interactions for each observation cell
    - Can coalesce M_j reads
    - Can prevent shared memory bank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

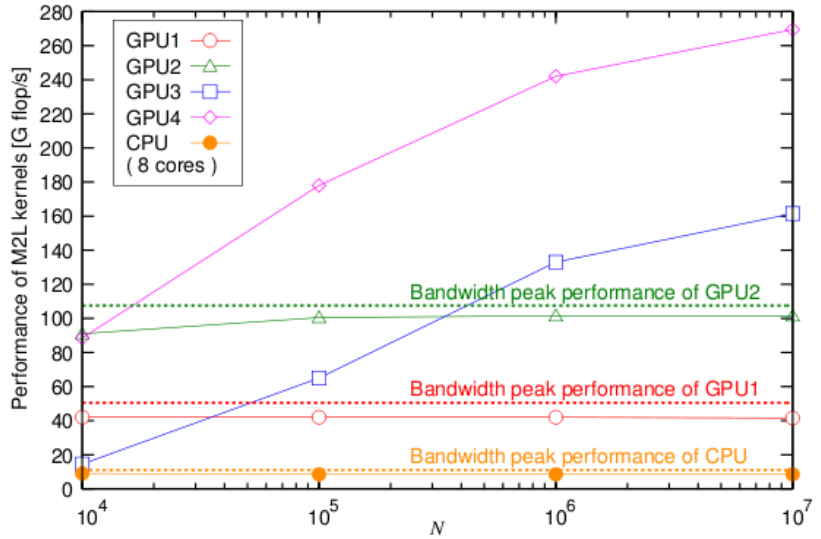
Chunking Clusters

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

- 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





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.