Memory/Computation Patterns Unknown at Compile-time

- **Irregular and dynamic applications**
  - Irregular data structures are unknown until run time
  - Data and their uses may change during the computation
    
    ```
    for ( i=0; i<n; i++)
    a[i] = f( a[g(i)], a[h(i)], ... );
    ```

- **Example Applications**
  - Molecular dynamics
  - Sparse matrix

- **Problems**
  - How to optimize at run time?
  - How to automate?
Dynamic Irregular Patterns

memory

... = A[P[tid]];

a mem seg.

control flow (thread divergence)

if (A[tid]) {...}

for (i=0; i<A[tid]; i++) {...}
Performance Potential

- Applications: CFD, DNA Sequence Analysis, Data Mining, ...

Potential Speedup

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMMER</td>
<td>5.27</td>
</tr>
<tr>
<td>3D-LBM</td>
<td>1.46</td>
</tr>
<tr>
<td>CUDA-EC</td>
<td>1.5</td>
</tr>
<tr>
<td>NN</td>
<td>2.51</td>
</tr>
<tr>
<td>CFD</td>
<td>2.75</td>
</tr>
<tr>
<td>CG</td>
<td>1.8</td>
</tr>
<tr>
<td>Unwrap</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Host: Xeon 5540.
Device: Tesla C1060. (240 cores)
Three Basic Transformations

- Data reordering (packing): $A[i] \rightarrow A'[Q[i]]$
- Computation regrouping: $\text{task}[i] \rightarrow \text{task}[P[i]]$
- Hybrid: combination of the above

“Every problem can be solved by adding one more level of indirection.”
Irregular Memory Access Pattern in GPUs

- Data reordering for irregular mem. accesses

![Diagram showing irregular memory access pattern]

- Total Memory Loads = 4
  - Warp 1 Loads: 2
  - Warp 2 Loads: 2
Irregular Memory Access Pattern in GPUs

- Data reordering for irregular mem. accesses

![Diagram showing irregular memory access pattern and data layout in GPUs]
Irregular Memory Access Pattern in GPUs

- Data reordering for irregular mem. accesses

Data Layout: $A[ ]$

Threads: tid: 0 1 2 3 4 5 6 7

Warp 1: $\text{Load: 1}$
Warp 2: $\text{Load: 2}$

Total Memory Loads = 3
Warp 1 Loads: 1
Warp 2 Loads: 2
Irregular Memory Access Pattern in GPUs

- Computation regrouping for irregular mem.

![Diagram showing irregular memory access pattern in GPUs](image-url)

Warp 1 Loads: 2
Warp 2 Loads: 2
Total Memory Loads = 4
Irregular Memory Access Pattern in GPUs

- Computation regrouping for irregular mem.

```
<table>
<thead>
<tr>
<th>Threads</th>
<th>tid:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td>A[ ]:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Warp 1 Loads: 2
Warp 2 Loads: 2

Total Memory Loads = 4
Irregular Memory Access Pattern in GPUs

- Computation regrouping for irregular mem.

Transformation on Thread ID
\[ \text{tid} = \text{Indirect}[\text{tid}] \]
Irregular Memory Access Pattern in GPUs

- Hybrid method for irregular mem. accesses

![Diagram showing thread allocation and memory loads in two warps. Each warp has 4 threads, and the total memory loads are calculated based on the data layout. Total Memory Loads = 4, Warp 1 Loads: 2, Warp 2 Loads: 2.]
Irregular Memory Access Pattern in GPUs

- Hybrid method for irregular mem. accesses
Irregular Memory Access Pattern in GPUs

- Hybrid method for irregular mem. accesses
Irregular Memory Access Pattern in GPUs

- Hybrid method for irregular mem. accesses
Irregular Memory Access Pattern in GPUs

- Hybrid method for irregular mem. accesses
Control Divergence in GPUs

- Hybrid

control flow (thread divergence)

```
if (A[tid]) {...}
```

<table>
<thead>
<tr>
<th>tid: 0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[]: 0 0 6 0 0 2 4 1</td>
</tr>
</tbody>
</table>
Control Divergence in GPUs

- Hybrid

control flow (thread divergence)

tid: 0 1 2 3 4 5 6 7

A'[ ]: 0 0 0 6 2 4 1

if (A'[tid]) {...}

Can you tell how computation regrouping and data reordering are combined?
Control Divergence in GPUs

- Computation regrouping
Control Divergence in GPUs

- Computation regrouping

```plaintext
if (A[tid']) { ... }
```
Finding Optimal Reordering or Regrouping

- NP-Complete
- Heuristics Algorithms (in Retrospect)
  - Representing spatial locality using a graph
  - Consecutive Packing [Ding+:PLDI99]
  - Breadth-first Search [Al-Furaih+:IPP…2000]
  - Gpart [Han+:LCR2000]

<table>
<thead>
<tr>
<th>Threads</th>
<th>tid:</th>
<th>Warp 1</th>
<th>Warp 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Layout

\[A[\_]:\]

\[A\{0, 2, 3, 5\} \quad A\{2, 3, 6, 7\}\]

conflicting pattern
Inspector Executor Model

- Compiler generates 2 pieces of customized code for such loops

\[
\text{for ( } i=0; i<n; i++) \\
\quad a[i] = f(a[g(i)], a[h(i)], ...);
\]

- Inspector
  - Calculate values of index expression by simulating the whole loop
  - Computes implicitly the locality graph (or dependence graph)
  - Computes a parallel schedule as wavefront traversal of the graph

- Executor
  - Follows the schedule to execute the loop or parallel tasks

- Overhead Amortization
  - Typically the pattern is invariant for many loop iterations
  - Or the irregular pattern can be determined statically
do iter=1, num

S1   doall pe=1, num_processors
S2       do i=1, nlocal(pe)
S3           next = schedule(i, pe)
S4             do j = low(next), high(next)
S5             x(next) = x(next) + a(j) * xold(column(j))
S6            xold(next) = x(next)
S7       end do
S8   end doall

Sparse Mesh Jacobi

Transformed Sparse Mesh Jacobi
Inspector Executor in Message Passing Paradigm

- **Message Passing Paradigm**
  - Compiler or programmer must insert necessary `Send/Recv` operations to move data from owning to reading processor
  - Necessary for both regular and irregular parallel loops
  - A challenge for compilers in irregular application, but still doable

- **Inspector**
  - Determines communication (schedule): who has to send which owned elements to whom
  - Allocate buffer for received elements; adapt access functions

- **Executor**
  - Communicates according to schedule
  - Executes loop
Inspector Executor in Message Passing Paradigm

forall (i, 0, 12, #)
y[i] = y[i] + a[ip[i]] * x[i]

//send data according to communication map
for each Pj in dest
    send requested a[:] elements to Pj

//received data according to communication map
for each Pi in source
    recv a[:] elements, write to respective buffer entries

//execute loop with modified access function
forall (i, 0, 12, #)
y[i] = y[i] + buffer[map[i]] * x[i]

Shared Memory Code

Executor Code (statically generated)