Conflict-Free Vectorization of Associative Irregular Applications with Recent SIMD Architectural Advances

Peng Jiang        Gagan Agrawal

Department of Computer Science and Engineering
The Ohio State University, USA
Motivation - Platforms

• CPU architectures are evolving
  • More parallelism is incorporated (MIMD+SIMD)
  • More advanced SIMD features
    • Wider SIMD lanes
    • Gather/scatter operations enable irregular memory accesses
    • Mask data type and operations enable computation on specified SIMD lanes
  • Supported in: Intel Xeon Phi, Intel Xeon E5-26xx v5 and future CPUs
Motivation - Applications

- SIMD were traditionally considered only suitable for regular applications
  - Dense matrix multiplication, pixel-wise image processing

- More irregular applications do not benefit from SIMD
  - Graph algorithms
  - Particle simulations
  - Data aggregation
Challenges with Irregular Applications

• **Irregular Computation Pattern**

```c
nodes[Nnodes];
edges[Nedges][2];
// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] = res; nodes[ny] = res; }
```

• **Irregular Memory Accesses**
  • Possible write conflicts among SIMD lanes
  • Only the non-conflicting lanes can safely write to memory
Conflicts among SIMD lanes

- Load the nodes’ indices from edges

```c
nodes[Nnodes];
edges[Nedges][2];
// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] += res; }
```

<table>
<thead>
<tr>
<th>nx</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ny</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Conflicts among SIMD lanes

- Read values from the nodes
  - Indirect memory accesses

```
nodes[Nnodes];
edges[Nedges][2];

// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] += res; }
```

<table>
<thead>
<tr>
<th>nx</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ny</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Conflicts among SIMD lanes

- Compute based on the nodes’ values

```c
nodes[Nnodes];
edges[Nedges][2];
// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] += res;
}
```

<table>
<thead>
<tr>
<th>nx</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ny</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>res</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Conflicts among SIMD lanes

- **Write the results back to nodes**
  - Indirect memory accesses
  - Multiple lanes can access the same memory location

```c
nodes[Nnodes];
edges[Nedges][2];
// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] += res;
}
```

```
<table>
<thead>
<tr>
<th>nx</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ny</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>res</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>nodes:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Example:
- `nx`: 0, 4, 0, 5
- `ny`: 1, 1, 2, 3
- `x`: nodes[0], nodes[4], nodes[0], nodes[5]
- `y`: nodes[1], nodes[1], nodes[2], nodes[3]
- `res`: *
- `nodes`: 0, 1, 2, 3, 4, 5, ....

Conflicts:
- Node 0 accesses memory location 0, 4, and 5.
- Node 4 accesses memory location 0 and 5.
- Node 0 and Node 4 access the same memory location 5.

```
conflict
```
Previous Approaches

- **Data Reorganization**
  - Reorganize the edges into non-conflicting groups
  - Pros: high SIMD utilization ratio
  - Cons: large overhead

- **Conflict Masking**
  - Only write the non-conflicting lanes into memory, defer conflicting lanes
  - New SIMD feature supports: vpconflict, mask operations
  - Pros: no data reorganization needed
  - Cons: low SIMD utilization ratio
Conflict Masking

- **Mask out the conflicting lanes**
  - The third lane need to be deferred to the next iteration (SIMD utilization rate: 75%)

```c
nodes[Nnodes];
edges[Nedges][2];
// iterates over all edges
for (i=0; i<Nedges; i++) {
    nx = edges[i][0]; ny = edges[i][1];
    // loads data from the two ends of an edge
    x = nodes[nx]; y = nodes[ny];
    // compute with the data from nodes
    res = compute(x, y);
    // update the result to the nodes
    nodes[nx] += res;
}
```

| nx | 0 | 4 | 0 | 5 |
|----------------|
| ny | 1 | 1 | 2 | 3 |

```
|----------------|

| res: | * | * | * | * |
|----------------|
| nodes: | 0 | 1 | 2 | 3 | 4 | 5 | .... |
```
Conflict Masking

• The SIMD utilization rate can be quite low for certain inputs
  • Extreme case: all SIMD lanes write to the same memory address
  • One edge processed per SIMD vector
  • Same as sequential execution; no benefit from SIMD

• Can we save the cost of data reorganization and still maintain a high SIMD utilization rate?
Observation

- The memory accesses in many irregular applications are associative reductions
  - PageRank: the rank values of multiple vertices are added to one vertex, the value is the sum of the ranks
    
    ```
    // iterate over all edges
    for(int j=0; j<nedges; j++) {
      // obtain vertex indices
      int nx = n1[j];
      int ny = n2[j];
      // add up rank values
      sum[ny] += rank[nx] / nneighbor[nx];
    }
    ```

  - SSSP: multiple vertices can update the distance of one vertex, the result is the minimum distance
  - Molecular Dynamics: the force imposed on a particle is the sum of the forces from all neighboring particles
  - Data aggregation: \{+, *, min, max\} are also associative reductions
Observation

- **The computation parts in these applications are conflict-free**
  - PageRank: dividing the rank value of a particular vertex even to its neighbors
    ```java
    // iterate over all edges
    for(int j=0; j<nedges; j++) {
      // obtain vertex indices
      int nx = n1[j];
      int ny = n2[j];
      // add up rank values
      sum[ny] += rank[nx] / nneighbor[nx];
    }
    ```
  - SSSP: comparing the current distance of a vertex with the distance from an incoming edge
  - Molecular Dynamics: computing the forces of all neighboring particles according to their distances
  - Hash-based aggregation: hashing of different keys
Our Approach

Main idea
- Let the computation parts run at full utilization of SIMD lanes
- Do not defer the update of conflicting lanes
- Resolve the data conflicts just before writing to memory

nx:
- 0 4 0 5

ny:
- 1 1 2 3

x:

y:

res:
- * * * *

nodes:
- 0 1 2 3 4 5 ....

Merge the results within the vector

nodes[0]+res[0]+res[2] = nodes[0] + (res[0]+res[2])
Our Approach

- **Main idea**
  - Let the computation parts run at full utilization of SIMD lanes
  - Do not invalidate the results in conflicting lanes
  - Resolve the data conflicts just before writing to memory

- Pros: No lanes are wasted (100% SIMD utilization rate)
- Cons: Overhead for merging the results in the conflicting lanes
Our Approach

• How to merge the values in conflicting lanes efficiently?
  • No direct hardware support
  • Software solution
    • Should avoid expensive memory accesses

• Our Solution (invector-reduction):
  • Identify the conflicting and non-conflicting lanes
  • For each of the conflicting lanes
    • Merge them into the first lane with the same writing address
  • Mask out the lanes that have been merged, lanes left hold the merged values
# Invector-reduction

Index vector: 0 1 1 1 2 2 2 2 5 0 1 1 1 5 5 5

Data vector:

- Non-conflicting lanes
- Lanes being merged in each iteration
- Reduced lanes
Invector-reduction

**Index vector:**

|   | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 5 | 0 | 1 | 1 | 1 | 5 | 5 | 5 |

**Data vector:**

- Non-conflicting lanes
- Lanes being merged in each iteration
- Reduced lanes

**1st iteration:**

- Non-conflicting lanes
- Lanes being merged in each iteration
- Reduced lanes
### Invector-reduction

<table>
<thead>
<tr>
<th>Index vector:</th>
<th>0 1 1 1 2 2 2 5 0 1 1 1 5 5 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data vector:</td>
<td>![Data vector diagram]</td>
</tr>
<tr>
<td>1st iteration:</td>
<td>![1st iteration diagram]</td>
</tr>
<tr>
<td>2nd iteration:</td>
<td>![2nd iteration diagram]</td>
</tr>
</tbody>
</table>

- **Non-conflicting lanes**
- **Lanes being merged in each iteration**
- **Reduced lanes**
Invector-reduction

**Index vector:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Data vector:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**1st iteration:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**2nd iteration:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**3rd iteration:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Non-conflicting lanes**
- **Lanes being merged in each iteration**
- **Reduced lanes**
Invector-reduction

**Index vector:**

```
0 1 1 1 2 2 2 5 0 1 1 1 5 5 5
```

**Data vector:**

1st iteration:

2nd iteration:

3rd iteration:

4th iteration:

Non-conflicting lanes

Lanes being merged in each iteration

Reduced lanes
Invector-reduction

Index vector: 0 1 1 1 2 2 2 2 5 0 1 1 1 5 5 5

Data vector:

1\textsuperscript{st} iteration:

2\textsuperscript{nd} iteration:

3\textsuperscript{rd} iteration:

4\textsuperscript{th} iteration:

Final result:

Non-conflicting lanes  Lanes being merged in each iteration  Reduced lanes

Store the merged results
Evaluation Results

• **Platform**
  - Intel Xeon Phi 7250 processor (KNL)
  - 512-bit SIMD units, AVX-512
  - 68 cores, 1.4 GHz

• **Applications**
  - PageRank, SSSP, SSWP, WCC
  - Molecular Dynamics
  - Hash-based aggregation

• **Comparison with different versions**
  - Sequential
  - Data reorganization (tiling-and-grouping)
  - Conflict masking
Evaluation Results

• **Tiling-and-grouping for SIMD processing of irregular applications** (Chen et al. CGO’16, Jiang et al. ICS’16)

  - Nonzeros represent edges
  - Tile the edges to improve data locality
  - Reorganize the edges in each tile into non-conflicting groups
  - For dynamic irregular applications, store the active edges in an indexing data structure
Evaluation Results

- **Evaluation Results**

- **Execution Time (sec)**
  - **nontiling_serial**: 2.3x
  - **tiling_and_grouping**: 4.2x
  - **nontiling_and_mask**: 2.5x
  - **nontiling_and_invec**:

- **SSSP**

- **Graph** showing the execution times for different scenarios.
Evaluation Results

Molecular Dynamics

- tiling_serial
- tiling_and_grouping
- tiling_and_mask
- tiling_and_invec

Execution Time (sec)
- Computing
- Tiling
- Grouping

- 2.6x
- 13x
- 6.8x

26
Conclusion

- **Resolving conflicts for associative irregular applications**
  - Irregular reductions are associative
  - Guarantee full SIMD utilization for computing part

- **Merging conflicting lanes within SIMD vectors**
  - No access to memory
  - Utilize conflict-detection and horizontal reduction

- **Performance**
  - Eliminate most of the data reorganization overhead
  - Maintain a high SIMD utilization
Thanks for your attention!

Q?

• Peng Jiang jiang.952@osu.edu
• Gagan Agrawal agrawal@cse.ohio-state.edu