Compiler Optimization for Irregular Memory Accesses in Chapel

Thomas B. Rolinger (UMD/LPS), Alan Sussman (UMD)

Contact: tbrolin@cs.umd.edu

CHIUW 2022
Motivation: Why target irregular memory accesses?

```
forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
        accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
}
```

Distributed Sparse Matrix-Vector Multiply (SpMV)
Motivation: Why target irregular memory accesses?

Distributed Sparse Matrix-Vector Multiply (SpMV)

```plaintext
forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
        accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
}
```

Rows is a distributed array
Motivation: Why target irregular memory accesses?

```
for all row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
        accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
}
```

Rows is a distributed array

Irregular memory access

x is a distributed array
Motivation: Why target irregular memory accesses?

```python
forall row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
        accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
}
```

Distributed Sparse Matrix-Vector Multiply (SpMV)

**High Productivity does not always lead to High Performance**
Motivation: Why target irregular memory accesses?

Distributed Sparse Matrix-Vector Multiply (SpMV)

How can we get better performance for these types of codes in Chapel?

High Productivity does not always lead to High Performance
Motivation: Why target irregular memory accesses?

```c
for all row in Rows {
    const id = row.id;
    var accum : real = 0;
    for k in row.columnOffset {
        accum += values[k] * x[col_idx[k]];
    }
    b[id] = accum;
}
```

Just a simple transformation...
Motivation: Why target irregular memory accesses?

NAS-CG (Conjugate Gradient)
Problem Size D (73 million non-zeros)

Manual optimizations can drastically improve performance
Motivation: Why target irregular memory accesses?

In this talk:

We can achieve this performance from the original “nice” code without modifying the code at all.

Manual optimizations can drastically improve performance.
Outline

• Optimization: selective data replication

• Implementation within compiler:
  • Code transformations
  • Static analysis

• Performance evaluation:
  • NAS-CG
  • PageRank
Selective Data Replication

• We focus on accesses of the form $A[B[i]]$ in \texttt{forall} loops
  • $A$ is a distributed array
  • the values in $B$ are not known until runtime
Selective Data Replication

- We focus on accesses of the form \( A[B[i]] \) in \texttt{forall} loops
  - \( A \) is a distributed array
  - the values in \( B \) are not known until runtime
- **Goal:** replicate remotely accessed elements of \( A \) so they can be used locally in the \texttt{forall}
  - \texttt{inspector}: runtime analysis that determines remote accesses
  - \texttt{executor}: optimized version of the \texttt{forall} that redirects remote accesses to the replicated copies
  - both generated by the \texttt{compiler} without user intervention
Selective Data Replication

- We focus on accesses of the form $A[B[i]]$ in `forall` loops
  - $A$ is a distributed array
  - the values in $B$ are not known until runtime
- **Goal**: replicate remotely accessed elements of $A$ so they can be used locally in the `forall`
  - **inspector**: runtime analysis that determines remote accesses
  - **executor**: optimized version of the `forall` that redirects remote accesses to the replicated copies
    - both generated by the **compiler** without user intervention
- **Requirements**:
  - the `forall` executes many times with the same access pattern
  - $A[B[i]]$ is on the RHS of an operation (i.e., read-only)
Selective Data Replication

• We focus on accesses of the form $A[B[i]]$ in forall loops
  • $A$ is a distributed array
  • the values in $B$ are not known until runtime
• Goal: replicate remotely accessed elements of $A$ so they can be used locally in the forall
• inspector: runtime analysis that determines remote accesses to the replicated copies
• executor: optimized version of the forall that redirects remote accesses to the replicated copies
• both generated by the compiler without user intervention
• Requirements:
  • the forall executes many times
  • $A[B[i]]$ is on the RHS of an operation (i.e., read-only)

We manually implemented this optimization in prior work:
• https://chapel-lang.org/CHIUW/2021/Rolinger.pdf

The optimization is influenced by other prior works:
• “Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures”, Das et al.
• “Automatic Support for Irregular Computations in a High-Level Language”, Su and Yelick
• “Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC”, Alvanos et al.
Code Transformations

```plaintext
forall i in B.domain {
    C[i] += A[B[i]];
}
```

```plaintext
if doOptimization {
    if doInspector(A, B) {
        inspectorPreamble(A);
        forall i in inspectorIter(B.domain) {
            inspectAccess(A, B[i]);
        }
        inspectorOff(A, B);
    }
    executorPreamble(A);
    forall i in B.domain {
        C[i] += executeAccess(A, B[i]);
    }
}
else {
    forall i in B.domain {
        C[i] += A[B[i]];
    }
}
```
Compiler Optimization: Code Transformations

**Inspector**: performs memory access analysis

**Key points:**
- inspector should only be performed (1) the **first time** we encounter the loop and (2) anytime the access pattern $A[B[i]]$ could have **changed**
- `inspectAccess()` does not issue the remote access to $A$; it just “queries” whether index $B[i]$ will be a remote access to $A$

```c
if doOptimization {
    if doInspector(A, B) {
        inspectorPreamble(A);
        forall i in inspectorIter(B.domain) {
            inspectAccess(A, B[i]);
        }
        inspectorOff(A, B);
    }
    executorPreamble(A);
    forall i in B.domain {
        C[i] += executeAccess(A, B[i]);
    }
} else {
    forall i in B.domain {
        C[i] += A[B[i]];
    }
}
```
Compiler Optimization: Code Transformations

**Executor**: executes the loop but redirects remote accesses to the replicated copies

**Key points:**
- `executorPreamble()` initializes replicated elements of A with values from original array
  - we only replicate an element once, regardless of how many times it is accessed → amortizes the cost of the remote read over multiple local accesses
- `executeAccess()` checks if index `B[i]` will be a remote access to A, and if so, returns the local copy.

```python
if doOptimization {
    if doInspector(A, B) {
        inspectorPreamble(A);
        forall i in inspectorIter(B.domain) {
            inspectAccess(A, B[i]);
        }
        inspectorOff(A, B);
    }
    executorPreamble(A);
    forall i in B.domain {
        C[i] += executeAccess(A, B[i]);
    }
} else {
    forall i in B.domain {
        C[i] += A[B[i]];
    }
}
```
May find later in Chapel’s compilation process that the optimization **cannot be applied**

In this case, **doOptimization** is set to false and dead-code elimination will “**undo**” our transformations and fall back to the original **forall** loop.
Static Analysis

• Everything just discussed for code transformations only holds if the optimization **CAN** and **SHOULD** be applied

• **We need to maintain correct program results**
  • detect when \( A[B[i]] \) access pattern changes so we can re-run the inspector

• **We should improve program performance**
  • ensure that the **forall** is nested in an outer loop, so it is likely to be executed multiple times \( \rightarrow \) amortizes the cost of the inspector over multiple iterations
  • also need to ensure that the inspector will not have to run EVERY time we execute the **forall**
Static Analysis

- Everything just mentioned for code transformations only holds if the optimization **CAN** and **SHOULD** be applied.

Static analysis and code transformations are performed **automatically**

The user **does not** add pragmas, code annotations, hints or anything to their code. They turn on a **compiler flag**

- multiple times → amortizes the inspector over multiple iterations
- also need to ensure that the inspector will not have to run EVERY time we execute the **forall**
Performance Evaluation

• Applications:
  • **NAS-CG** (conjugate gradient)
  • **PageRank** (iterative SpMV-like operations)

• Systems:
  • **FDR Infiniband**, 20 cores per node, 512 GB of memory per node
  • **Cray XC**, Aries interconnect, 44 cores per node, 128 GB of memory per node

• Experiments:
  • measured **runtime speed-ups** achieved by optimization relative to the original code
  • includes any overhead incurred by the inspector
NAS-CG Data sets

<table>
<thead>
<tr>
<th>Name</th>
<th>Rows</th>
<th>Non-zeros</th>
<th>Density (%)</th>
<th># of SpMVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>150k</td>
<td>39M</td>
<td>0.17</td>
<td>1950</td>
</tr>
<tr>
<td>D</td>
<td>150k</td>
<td>73M</td>
<td>0.32</td>
<td>2600</td>
</tr>
<tr>
<td>E</td>
<td>9M</td>
<td>6.6B</td>
<td>0.008</td>
<td>2600</td>
</tr>
<tr>
<td>F</td>
<td>54M</td>
<td>55B</td>
<td>0.002</td>
<td>2600</td>
</tr>
</tbody>
</table>

NAS-CG Optimization Speed-ups

<table>
<thead>
<tr>
<th>Locales</th>
<th>Cray XC</th>
<th>Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>3.2</td>
<td>2.8</td>
</tr>
<tr>
<td>4</td>
<td>3.6</td>
<td>3.4</td>
</tr>
<tr>
<td>8</td>
<td>5.7</td>
<td>6.2</td>
</tr>
<tr>
<td>16</td>
<td>8.6</td>
<td>11</td>
</tr>
<tr>
<td>32</td>
<td>6.4</td>
<td>8.4</td>
</tr>
<tr>
<td>64</td>
<td>4.1</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Take-aways:
- “—” means not enough memory, “NA” means not enough nodes
- lots of data reuse in the kernel, so the optimization performs very well
- inspector overhead is small due to many iterations w/o the access pattern changing
- optimization provides larger gains on Infiniband
  - higher latency for small messages than Aries
PageRank Data sets

| Name    | |V| | |E| | Density (%) | Iterations |
|---------|---|---|---|---|---|---|---|
| webbase-2001 | 118M | 992M | 7.1e-6 | 33 |
| sk-2005    | 51M  | 1.9B | 7.5e-5 | 40 |

PageRank Optimization Speed-ups

<table>
<thead>
<tr>
<th>Locales</th>
<th>Cray XC webbase-2001</th>
<th>sk-2005</th>
<th>Infiniband webbase-2001</th>
<th>sk-2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.88</td>
<td>1.2</td>
<td>5.2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0.98</td>
<td>1.6</td>
<td>8.6</td>
<td>7.1</td>
</tr>
<tr>
<td>8</td>
<td>0.97</td>
<td>1.3</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>0.94</td>
<td>1.7</td>
<td>9.6</td>
<td>5.4</td>
</tr>
<tr>
<td>32</td>
<td>1.3</td>
<td>1.4</td>
<td>4.5</td>
<td>4.2</td>
</tr>
<tr>
<td>64</td>
<td>1.2</td>
<td>2.1</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>geomean</td>
<td>1.04</td>
<td>1.5</td>
<td>7.3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Take-aways:
- **smaller speed-ups** overall due to **fewer iterations** than NAS-CG and **less data reuse**
  - because of both the algorithm and the graphs
- speed-ups on the Cray can be negative when the data reuse is low (webbase-2001)
- nevertheless, still significant speed-ups overall
Conclusions

• Optimization is producing promising results: runtimes improved from hours/days to minutes

• Limitations exists for this type of data replication
  • forall must execute multiple times without the memory access pattern changing
  • could use a lot of memory for the replication
  • currently limited to read-only data

• Not covered in this talk:
  • handling foralls in procedures with multiple call sites
  • special handling for arrays/domains that are fields in a record
  • inter-procedural analysis to detect modifications to arrays/domains across calls
  • alias analysis to detect modifications to arrays/domains
Future Work

• Use this framework to implement other optimizations
  • prefetching for Chapel’s remote cache
  • more generalized aggregation for remote writes than CopyAggregation
  • end goal is a single framework that can apply all these optimizations automatically, deciding which one to apply considering the specific scenario

• Acknowledgements:
  • Chapel team: extremely helpful and responsive to questions, and facilitated access to the Cray system
  • Specific shout outs to Engin K., Vass L., Elliot R., Brad C., Michelle S.
Bonus: BFS Results

• Implemented Breadth First Search (BFS) as a series of SpMV-like operations
• Not necessarily the best approach for raw performance but provides an interesting experiment
  • relatively few iterations are performed, so the cost of the inspector becomes more prominent
### BFS Data sets

| Name | $|V|$ | $|E|$ | Density (%) | Iterations |
|------|------|------|-------------|------------|
| s_25 | 33M  | 1B   | 9.5e-5      | 6          |
| s_26 | 67M  | 2.1B | 4.8e-5      | 7          |
| s_27 | 134M | 4.2B | 2.3e-5      | 7          |

### BFS Optimization Speed-ups

#### BFS: s_27

Optimization Runtime Speed-ups

<table>
<thead>
<tr>
<th>Locales</th>
<th>Cray XC</th>
<th>Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s_25</td>
<td>s_26</td>
</tr>
<tr>
<td></td>
<td>s_25</td>
<td>s_26</td>
</tr>
<tr>
<td>2</td>
<td>0.23</td>
<td>0.26</td>
</tr>
<tr>
<td>4</td>
<td>0.22</td>
<td>0.27</td>
</tr>
<tr>
<td>8</td>
<td>0.3</td>
<td>0.29</td>
</tr>
<tr>
<td>16</td>
<td>0.28</td>
<td>0.32</td>
</tr>
<tr>
<td>32</td>
<td>0.39</td>
<td>0.42</td>
</tr>
<tr>
<td>64</td>
<td>0.32</td>
<td>0.31</td>
</tr>
<tr>
<td>geomean</td>
<td>0.28</td>
<td>0.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locales</th>
<th>Cray XC</th>
<th>Infiniband</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s_25</td>
<td>s_26</td>
</tr>
<tr>
<td></td>
<td>s_25</td>
<td>s_26</td>
</tr>
<tr>
<td>2</td>
<td>30.3</td>
<td>27.6</td>
</tr>
<tr>
<td>4</td>
<td>11.7</td>
<td>13.6</td>
</tr>
<tr>
<td>8</td>
<td>6.5</td>
<td>7.1</td>
</tr>
<tr>
<td>16</td>
<td>3.9</td>
<td>2.8</td>
</tr>
<tr>
<td>32</td>
<td>2.6</td>
<td>1.2</td>
</tr>
<tr>
<td>64</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>geomean</td>
<td>7.5</td>
<td>6.2</td>
</tr>
</tbody>
</table>

#### Take-aways:

- **poor performance on the Cray**
  - not enough iterations to amortize the cost of the inspector
- **positive gains on the Infiniband** system despite the few number of iterations
  - performance increases dwindle as we increase the number of locales (inspector runtime does not scale as well as the runtime of each iteration)