PPIR: Parallel Pattern Intermediate Representation

Adrian Schmitz, Julian Miller, Lukas Trümper, Matthias S. Müller

November 14, 2021
Optimizing code for heterogeneous systems is hard
Portability between different heterogeneous systems is even harder

Approach:
  – Improve the dataflow
  – Handle scheduling to hardware
  – Perform static global optimizations

Goal:
  – Automate global optimization/scheduling
  – Generate optimized code for a target architecture
Agenda

• Approach
  – Abstract Pattern Tree
  – Global Optimization
  – Abstract Mapping Tree

• Evaluation

• Conclusion/Outlook
Hierarchically structured code
- Statically analysable
- Parallel Pattern
  - Matson et al.\textsuperscript{1}
  - McCool et al.\textsuperscript{2}
  - Add additional structure
  - Compress the representation

\textsuperscript{1}Mattson, T.G.; Sanders, B.; Massingill, B. Patterns for Parallel Programming.
\textsuperscript{2}McCool, M.D.; Robison, A.D.; Reinders, J. Structured Parallel Programming—Patterns for Efficient Computation.
Approach

Parallel Pattern Language
---
Abstract Pattern Tree
---
Global Optimizations
---
Abstract Mapping Tree
---
Code Generation

Hardware Language
---
Cluster Model
---
Global Optimizations

5 PPIR: Parallel Pattern Intermediate Representation | Adrian Schmitz, Julian Miller, Lukas Trümper, Matthias S. Müller | RWTH Aachen University | November 14, 2021
Approach

- Parallel Pattern Language
- Hardware Language
- Abstract Pattern Tree
- Cluster Model
- Global Optimizations
- Abstract Mapping Tree
- Code Generation

Intermediate Representation
**Approach: Hardware Language**

- JSON representation of the entire cluster
- Hierarchical structure
  1. Cluster
  2. Node
  3. Device (e.g. CPU/GPU)
  4. Cache group
  5. Cache
- Example with 2 nodes, 1 device, 2 cache groups

```json
{
  "topology": "fully",
  "connectivity-bandwidth": [
    ["0", "4148"], ["4148", "0"]
  ],
  "connectivity-latency": [
    ["0", "1840"], ["1840", "0"]
  ],
  "nodes": [
    {
      "identifier": "Node1",
      "address": "192.165.0.1.125",
      "template": "Nodes/node_c18.json"
    },
    {
      "identifier": "Node2",
      "address": "192.165.0.1.126",
      "template": "Nodes/node_c18.json"
    }
  ]
}
```
Approach: Parallel Pattern Language

- Make use of parallel patterns
- Static global optimization ⇒ static array sizes
- Batch classification algorithm
- $2^{19}$ elements with $2^{12}$ features

```plaintext
seq main() : Int {
  var [[Double]] data = init_List([524288, 4096])
  var [[Double]] normalized = init_List([524288, 4096])
  var [Double] features = init_List([524288])
  var [Int] classes = init_List([524288])
  data = read("data.txt")
  normalized = normalize <<<>>>(data)
  features = extract <<<>>>(normalized)
  classes = classify <<<>>>(features)
  return 0
}
```

```plaintext
map normalize([[Double]] data) : [[Double]] normalized {
  // Normalization of each input element
}
```
Approach: Parallel Pattern Language

- Make use of parallel patterns
- Static global optimization $\Rightarrow$ static array sizes
- Batch classification algorithm
- $2^{19}$ elements with $2^{12}$ features

```plaintext
seq main() : Int {
  var [[Double]] data = init_List([524288, 4096])
  var [[Double]] normalized = init_List([524288, 4096])
  var [Double] features = init_List([524288])
  var [Int] classes = init_List([524288])
  data = read("data.txt")
  normalized = normalize<<<>>>(data)
  features = extract<<<>>>(normalized)
  classes = classify<<<>>>(features)
  return 0
}
map normalize([[Double]] data) : [[Double]] normalized {
  // Normalization of each input element
}
```
Abstract Pattern Tree

- Functions/Patterns are generated as a forest
- Connect patterns via calls

- Meta information is stored within the patterns
- Size of the pattern is stored within the calls
- Data dependencies are extracted between nodes, e.g., extract depends on normalize
• Utilize data flow for pipelines as fused nodes
• Target architecture: 2 Nodes CLAIX2018
  – Not large enough for data transfers
  – Split the large task on 2 cache groups
• Ordered into steps, not a tree
  – Only nodes directly below the root are optimized
  – Similar to bulk synchronous programming
• Defined and implemented in previous work\(^3\)

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
<th>Device</th>
<th>Cache group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fused Node 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fused Node 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^3\) J. Miller, L. Trümper, C. Terboven, and M. S. Müller, A theoretical model for global optimization of parallel algorithms.
Abstract Mapping Tree

- Regenerate the tree structure
- Additional pattern calls depending on the optimization
- The prototype differentiates between AMT/APT in the implementation

- Generate synchronization and data transfers in the AMT
- Should be part of the optimization
Evaluation

- Rodinia OpenMP benchmarks version 3.1\(^4\)
- Optimization should not be measured, thus AMT is excluded
- Evaluation of the expressiveness of the APT
- Evaluation of the compile time of our prototype
- Evaluation of the APT memory consumption against LLVM IR

17 of 19 Rodinia benchmarks can be translated to PPL

- Multiple output/data structures need to be split or transformed into arrays
- Limited support for stencil halo data
- Dynamic programming is limited to 1D arrays and the last timestep
- Arrays with dynamic size can be overestimated to allow static analysis
- Bit shift is not yet supported
- Dynamic parallelism is not supported by design

<table>
<thead>
<tr>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Problems</td>
<td>6</td>
</tr>
<tr>
<td>Data Structures</td>
<td>4</td>
</tr>
<tr>
<td>Stencil Corner Cases</td>
<td>2</td>
</tr>
<tr>
<td>Dynamic Programming Limitation</td>
<td>2</td>
</tr>
<tr>
<td>Overestimation of Arrays</td>
<td>3</td>
</tr>
<tr>
<td>Unsupported Operations</td>
<td>1</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>1</td>
</tr>
</tbody>
</table>
Memory Usage of the APT

- Memory usage of the internal data structure
- APT contains additional information ⇒ more memory is used
- AMT is dependant on hyper parameters
- Size difference of about 1 order of magnitude
- Largest difference Myocyte
  - APT is 36 times larger than LLVM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RAM Usage [KB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LLVM</td>
</tr>
<tr>
<td>B+tree</td>
<td>757.76</td>
</tr>
<tr>
<td>Back Propogation</td>
<td>266.24</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>81.92</td>
</tr>
<tr>
<td>CFD Solver</td>
<td>1097.73</td>
</tr>
<tr>
<td>Heart Wall</td>
<td>1695.74</td>
</tr>
<tr>
<td>HotSpot</td>
<td>266.24</td>
</tr>
<tr>
<td>HotSpot3D</td>
<td>184.32</td>
</tr>
<tr>
<td>k-Nearest Neighbors</td>
<td>53.25</td>
</tr>
<tr>
<td>Kmeans</td>
<td>167.94</td>
</tr>
<tr>
<td>LavaMD</td>
<td>126.98</td>
</tr>
<tr>
<td>Leukocyte</td>
<td>2416.64</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>565.25</td>
</tr>
<tr>
<td>MUMmerGPU</td>
<td>NA</td>
</tr>
<tr>
<td>Myocyte</td>
<td>1097.73</td>
</tr>
<tr>
<td>Needleman-Wunsch</td>
<td>184.32</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>335.87</td>
</tr>
<tr>
<td>PathFinder</td>
<td>77.82</td>
</tr>
<tr>
<td>SRAD</td>
<td>139.26</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>389.12</td>
</tr>
</tbody>
</table>
### Compilation Time

- Time in seconds
- Java implementation
  - Applicable compilation times
  - Target order of magnitude
- Parsing scales with size and **expression length** ⇒ apply left recursive grammar.
- Printing as a sample for APT traversal
- IO bottleneck for large trees

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parse Time</th>
<th>APT Gen.</th>
<th>APT Print</th>
<th>Full Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+tree</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Back Propogation</td>
<td>1.47e+00</td>
<td>6.00e-02</td>
<td>7.00e-03</td>
<td>1.00e-02</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>5.80e-02</td>
<td>1.60e-02</td>
<td>5.00e-03</td>
<td>6.00e-03</td>
</tr>
<tr>
<td>CFD Solver</td>
<td>3.36e-01</td>
<td>2.40e-02</td>
<td>5.00e-03</td>
<td>7.00e-03</td>
</tr>
<tr>
<td>Heart Wall</td>
<td>3.24e-01</td>
<td>3.00e-02</td>
<td>6.00e-03</td>
<td>7.00e-03</td>
</tr>
<tr>
<td>HotSpot</td>
<td>2.68e-01</td>
<td>2.40e-02</td>
<td>5.00e-03</td>
<td>5.00e-03</td>
</tr>
<tr>
<td>HotSpot3D</td>
<td>8.40e+00</td>
<td>2.80e-02</td>
<td>5.00e-03</td>
<td>5.00e-03</td>
</tr>
<tr>
<td>k-Nearest Neighbors</td>
<td>4.20e-02</td>
<td>9.80e-02</td>
<td>1.20e-02</td>
<td>1.09e-01</td>
</tr>
<tr>
<td>Kmeans</td>
<td>5.70e-02</td>
<td>2.40e-02</td>
<td>4.00e-03</td>
<td>5.00e-03</td>
</tr>
<tr>
<td>LavaMD</td>
<td>4.33e-01</td>
<td>2.40e-02</td>
<td>4.00e-03</td>
<td>5.00e-03</td>
</tr>
<tr>
<td>Leukocyte</td>
<td>2.59e+00</td>
<td>1.10e+00</td>
<td>1.07e+00</td>
<td>1.34e+00</td>
</tr>
<tr>
<td>LU Decomposition</td>
<td>1.90e-02</td>
<td>2.30e-02</td>
<td>4.00e-03</td>
<td>4.00e-03</td>
</tr>
<tr>
<td>MUMmerGPU</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Myocyte</td>
<td>4.45e+00</td>
<td>2.42e-01</td>
<td>8.30e-02</td>
<td>3.89e+02</td>
</tr>
<tr>
<td>Needelman-Wunsch</td>
<td>1.01e-01</td>
<td>9.60e-02</td>
<td>1.70e-02</td>
<td>2.78e-01</td>
</tr>
<tr>
<td>Particle Filter</td>
<td>1.27e-01</td>
<td>1.95e-01</td>
<td>9.50e-02</td>
<td>8.02e+00</td>
</tr>
<tr>
<td>PathFinder</td>
<td>4.80e-02</td>
<td>1.17e-01</td>
<td>1.80e-02</td>
<td>4.66e+00</td>
</tr>
<tr>
<td>SRAD</td>
<td>1.30e-01</td>
<td>1.17e-01</td>
<td>5.40e-02</td>
<td>2.66e+00</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>2.91e-01</td>
<td>2.41e-01</td>
<td>1.34e-01</td>
<td>1.13e+01</td>
</tr>
</tbody>
</table>
Conclusion

- Introduced a parallel pattern based code representation (APT)
- Extended the APT to cover global optimizations (AMT)
- Evaluated the prototype with Rodinia
- APT can represent most of the benchmarks
- Applicable compilation times and memory consumption, but still room for improvement
Outlook

- Providing a C front-end (WIP)
- Extending the optimization
- Finalizing the code generator (WIP)
- Combining AMT and APT
- Improving memory consumption of the APT
- Addressing real world examples, e.g. Lulesh (WIP)
- https://github.com/RWTH-HPC/PPL
Outlook

- Providing a C front-end (WIP)
- Extending the optimization
- Finalizing the code generator (WIP)
- Combining AMT and APT
- Improving memory consumption of the APT
- Addressing real world examples, e.g. Lulesh (WIP)
- https://github.com/RWTH-HPC/PPL

Thank you for your attention