HYDRA: Extending Shared Address Programming for Accelerator Clusters

Putt Sakdhnagool, Amit Sabne, Rudolf Eigenmann
Purdue University
Programming on Accelerator Clusters

- Accelerator clusters become a common standard for supercomputers
- State of the art: MPI + accelerator specific programming model

- Programming accelerator is hard
  - Unique programming model
  - Different memory hierarchy
  - Multiple level of parallelism
Shared-Address Programming Model for Accelerator Clusters

- Shared-address programming model
  - Simpler programming model
    - Hide program complexity
  - Higher Productivity

- Problems for extending to accelerator clusters
  - High level of abstraction
  - Different accelerator architectures
Our Approach

- Source-to-source translator that generates accelerated MPI programs from shared-address programs
- Two compile-time analyses to extract information than is necessary for program scalability
  - Memory allocation
  - Data transfer
- Compiler design to support multiple accelerator architectures
OMPD

- Hybrid compiler-runtime translation for distributed system

- OMPD Compiler
  - Work partitioning and distribution
    - Partition program into program blocks.
    - Each block represents either a serial or parallel loop
    - Each block is distributed based on its type

- OMPD Runtime
  - Host-to-host message generation
Array Data Flow Analysis

- Core technique for OMPD and our proposed analyses.

- Analyze the data producer and consumer relationships between program blocks.
  - A set of local uses (LUSE) and local definitions (LDEF) of each program block defined as
    \[
    LUSE = \{use_i | 1 \leq i \leq n\} \\
    LDEF = \{def_j | 1 \leq j \leq m\}
    \]
  - use represents a read access in the program block
    \[
    use_i = [lb_{p-1} : ub_{p-1}]...[lb_1 : ub_1][lb_0 : ub_0]
    \]
  - def represents a write access in the program block
    \[
    def_i = [lb_{p-1} : ub_{p-1}]...[lb_1 : ub_1][lb_0 : ub_0]
    \]
HYDRA: Programming Model

- Directive-based shared address programming model

- Have only one construct for parallel loops.
  
  #pragma hydra parallel for [clauses]

- 4 available clauses
  - Syntactically optional
  - Might be needed for program semantic

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>shared(varlist)</td>
</tr>
<tr>
<td>private</td>
<td>private(varlist)</td>
</tr>
<tr>
<td>firstprivate</td>
<td>firstprivate(varlist)</td>
</tr>
<tr>
<td>reduction</td>
<td>reduction(op:varlist)</td>
</tr>
</tbody>
</table>
for (k=0; k<ITER; k++)
{
    #pragma hydra parallel for private(i,j)
    for (i=1; i<SIZE+1; i++) {
        for (j=1; j<SIZE+1; j++) {
            a[i][j] = (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1]) / 4;
        }
    }

    #pragma hydra parallel for private(i,j)
    for (i=1; i<SIZE+1; i++) {
        for (j=1; j<SIZE+1; j++) {
            b[i][j] = a[i][j];
        }
    }
}
Data Transfer

- Precise data transfer between host and accelerator memory is critical
  - Excessive transfer overhead can limit scalability

- Simple approach
  - Transferring the entire shared data before/after parallel program block.
Data Transfer Analysis

- **Two-step algorithm**

**First step**: Identify necessary shared data for a program block
- Use LUSE information to determine a live-in and live-out data.

**Second step**: Determine the transfer range of the shared data
- Transfer range can be defined by the minimum lower bound and the maximum upper bound of local read accesses

\[
\text{transferredSection} = [\min(lb_{p-1}) : \max(ub_{p-1})][\min(lb_1) : \max(ub_1)][\min(lb_0) : \max(ub_0)]
\]
Memory Allocation

- Accelerator memory is limited
- Full data allocation could exceed memory capacity
  - Failure of single accelerator execution
  - Limit the problem size to accelerator memory capacity
Memory Allocation Optimization

- Perform global analysis to summarize all accesses of the shared data
  - Need only 1 allocation and 1 deallocation
  - Small sacrifice in the size of memory allocated

- All accesses of a shared data $A$ can be computed using the equation

\[
LUSE^A \quad LDEF^A
\]

- The allocation size can be found using the minimum lower bound and maximum upper bound of all accesses

- Compiler deals with the misalignment of the newly allocated and old shared data
HYDRA Translation System

- Consist of a compiler and a runtime system

Compiler
- Generate accelerated MPI from HYDRA programs
- Support multiple accelerator architectures
Supporting Multiple Accelerator Architectures

- Architecture-agnostic internal representation (IR)

- Four common accelerator operations
  - Memory allocation
  - Data transfer
  - Kernel execution
  - Memory deallocation

- The compiler design minimizes the number of architecture specific passes
  - Only 1 out of 8 passes are architecture specific
HYDRA Translation Process

- HYDRA Program
- MPI+ Accelerator Program
- Code Generation
  - GPU Code Generator
  - MIC Code Generator
- Work Distribution
- DEF/USE Analysis
- Kernel Extractor
- Memory Allocation Optimization
- Accelerator Optimizer
- Memory Transfer Analysis

Diagram showing the process of HYDRA translation with various steps and tools involved.
HYDRA Runtime System

- Responsible for remote accelerator communication

- Host-side runtime system
  - Generate communication message
  - Execute host-to-host communication

- Accelerator runtime extension
  - Map host and accelerator data
  - Exchange message between host and accelerator
## Evaluation Setup

<table>
<thead>
<tr>
<th></th>
<th>GPU Cluster</th>
<th>MIC Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of Compute Nodes</strong></td>
<td>264</td>
<td>580</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>2x 8-Core Xeon E5-2670 (2.6GHz)</td>
<td>2x 8-Core Xeon E5-2670 (2.6GHz)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>32GB</td>
<td>64GB</td>
</tr>
<tr>
<td><strong>Accelerators</strong></td>
<td>3x NVIDIA Tesla M2090 GPUs</td>
<td>2x Xeon Phi P5110</td>
</tr>
<tr>
<td><strong>Accelerator Memory</strong></td>
<td>6GB</td>
<td>8GB (Maximum size per allocation is 1.88GB)</td>
</tr>
<tr>
<td><strong>Interconnection</strong></td>
<td>Infiniband FDR</td>
<td>Infiniband FDR-10</td>
</tr>
</tbody>
</table>

- The evaluation uses up to 64 nodes with one MPI process and one accelerator per node.
Evaluation

- 5 common benchmarks
  - Bilateral Filter, Blackscholes, Filterbank, Jabobi, Heat3D

- Scalability
  - Strong scaling
    - 2 problem classes: Class-A and Class-B
  - Weak scaling

- Memory allocation
Strong Scaling

**GPU**

**JACOBI**

**HEAT3D**

**BLACKSCHOLES**

**MIC**

**JACOBI**

**HEAT3D**

**BLACKSCHOLES**
Strong Scaling

Average speedup: 24.54x on MIC, 27.56x on GPU for class-A problems
Memory Allocation: Strong Scaling

- The size of allocated accelerator memory reduces as the number of node increases for all benchmarks
Weak Scaling

**GPU**

**MIC**

- Jacobi
- Heat3D
- Blackscholes
- BilateralFilter
- Filterbank
Memory Allocation: Weak Scaling

- **JACOBI**
- **HEAT3D**
- **BLACKSCHOLES**
- **BILATERAL FILTER**

### Graphs

- **Per Node Allocation**
- **Problem Size**
- **Limit**
Memory Allocation: Weak Scaling

![Graph showing memory allocation for different numbers of nodes.](image)
Conclusion

- Two architecture-agnostic compile-time optimizations
  - Ensure scalability of the generated program

- HYDRA translation system
  - Generate accelerated MPI programs from simple programming model
  - Architecture-agnostic IR

- Evaluate on 64-node GPU and Xeon Phi clusters
  - 24.54x speed up on the 64-node Xeon Phi cluster
  - 27.56x speed up on the 64-node GPU cluster
Thank you