LLVM-based Communication Optimizations for PGAS Programs

2nd Workshop on the LLVM Compiler Infrastructure in HPC @ SC15

Akihiro Hayashi (Rice University)
Jisheng Zhao (Rice University)
Michael Ferguson (Cray Inc.)
Vivek Sarkar (Rice University)
A Big Picture

PGAS Languages

- High-productivity features:
  - Global-View
  - Task parallelism
  - Data Distribution
  - Synchronization

Communication is implicit in some PGAS Programming Models

- Global Address Space
  - Compiler and Runtime is responsible for performing communications across nodes

Remote Data Access in Chapel

```
1: var x = 1;       // on Node 0
2: on Locales[1] { // on Node 1
    ... = x;       // DATA ACCESS
3: }
```
Communication is Implicit in some PGAS Programming Models (Cont’d)

Remote Data Access

1: var x = 1; // on Node 0
2: on Locales[1] { // on Node 1
3:  ... = x; // DATA ACCESS

Compiler Optimization

1: var x = 1;
2: on Locales[1] {
3:  ... = 1;

Runtime affinity handling

if (x.locale == MYLOCALE) {
    *(x.addr) = 1;
} else {
    gasnet_get(...);
}
Communication Optimization is Important

A synthetic Chapel program on Intel Xeon CPU X5660 Clusters with QDR Infiniband

Lower is better

Optimized (Bulk Transfer) vs Unoptimized

Latency (ms) vs Transferred Byte

59x

1,500x
PGAS Optimizations are language-specific

Chapel Compiler

Language Specific!

UPC Compiler

Habanero-C Compiler

X10, Habanero-UPC++,…

Our goal


© Argonne National Lab.

© Berkeley Lab.

© RIKEN AICS
Why LLVM?

- Widely used language-agnostic compiler

- C/C++ Frontend
  - Clang

- C/C++, Fortran, Ada, Objective-C Frontend
  - dragonegg

- Chapel Frontend
- UPC++ Frontend

LLVM Intermediate Representation (LLVM IR)

Analysis & Optimizations

- x86 backend
  - x86 Binary

- Power PC backend
  - PPC Binary

- ARM backend
  - ARM Binary

- PTX backend
  - GPU Binary
Summary & Contributions

- Our Observations:
  - Many PGAS languages share semantically similar constructs
  - PGAS Optimizations are language-specific
- Contributions:
  - Built a compilation framework that can uniformly optimize PGAS programs (Initial Focus: Communication)
    - Enabling existing LLVM passes for communication optimizations
    - PGAS-aware communication optimizations

Photo Credits: http://chapel.cray.com/logo.html, http://llvm.org/Logo.html,
Overview of our framework

- Chapel Programs
- UPC++ Programs
- X10 Programs
- CAF Programs

- Chapel-LLVM frontend
- UPC++-LLVM frontend
- X10-LLVM frontend
- CAF-LLVM frontend

LLVM IR

LLVM-based Communication Optimization passes

Lowering Pass

1. Vanilla LLVM IR
2. Use address space feature to express communications

Need to be implemented when supporting a new language/runtime
Generally language-agnostic
How optimizations work

Chapel

```
// x is possibly remote
x = 1;
```

UPC++

```
shared_var<int> x;
x = 1;
```

```
store i64 1, i64 addrspace(100)* %x, ...
```

treat remote access as if it were local access

1. Existing LLVM Optimizations
2. PGAS-aware Optimizations

Address space-aware Optimizations

Runtime-Specific Lowering

Communication API Calls

12
LLVM-based Communication Optimizations for Chapel

1. Enabling Existing LLVM passes
   - Loop invariant code motion (LICM)
   - Scalar replacement, ...

2. Aggregation
   - Combine sequences of loads/stores on adjacent memory location into a single memcpy

These are already implemented in the standard Chapel compiler
An optimization example:
LICM for Communication Optimizations

LICM by LLVM

```
for i in 1..100 {
    %x = load i64 addrspace(100)* %xptr A(i) = %x;
}
```

LICM = Loop Invariant Code Motion
An optimization example: Aggregation

// p is possibly remote
sum = p.x + p.y;

llvm.memcpy(...);

load i64 addrspace(100)* %pptr+0
load i64 addrspace(100)* %pptr+4

llvm.memcpy(...);
LLVM-based Communication Optimizations for Chapel

3. Locality Optimization
   - Infer the locality of data and convert possibly-remote access to definitely-local access at compile-time if possible

4. Coalescing
   - Remote array access vectorization

These are implemented, but not in the standard Chapel compiler
An Optimization example:
Locality Optimization

1: proc habanero(ref x, ref y, ref z) {
2:    var p: int = 0;
3:    var A:[1..N] int;
4:    local { p = z; }
5:    z = A(0) + z;
6:}

1. A is definitely-local
2. p and z are definitely local
3. Definitely-local access! (avoid runtime affinity checking)
An Optimization example: Coalescing

Before

1: for i in 1..N {
2:   ... = A(i);
3: }

After

1: localA = A;
2: for i in 1..N {
3:   ... = localA(i);
4: }

Perform bulk transfer

Converted to definitely-local access
## Performance Evaluations: Benchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith-Waterman</td>
<td>185,600 x 192,000</td>
</tr>
<tr>
<td>Cholesky Decomp</td>
<td>10,000 x 10,000</td>
</tr>
<tr>
<td>NPB EP</td>
<td>CLASS = D</td>
</tr>
<tr>
<td>Sobel</td>
<td>48,000 x 48,000</td>
</tr>
<tr>
<td>SSCA2 Kernel 4</td>
<td>SCALE = 16</td>
</tr>
<tr>
<td>Stream EP</td>
<td>$2^{30}$</td>
</tr>
</tbody>
</table>
Performance Evaluations: Platforms

- **Cray XC30™ Supercomputer @ NERSC**
  - Node
    - Intel Xeon E5-2695 @ 2.40GHz x 24 cores
    - 64GB of RAM
  - Interconnect
    - Cray Aries interconnect with Dragonfly topology

- **Westmere Cluster @ Rice**
  - Node
    - Intel Xeon CPU X5660 @ 2.80GHz x 12 cores
    - 48 GB of RAM
  - Interconnect
    - Quad-data rated infiniband
Performance Evaluations: Details of Compiler & Runtime

Compiler
- Chapel Compiler version 1.9.0
- LLVM 3.3

Runtime:
- GASNet-1.22.0
  - Cray XC: aries
  - Westmere Cluster: ibv-conduit
- Qthreads-1.10
  - Cray XC: 2 shepherds, 24 workers / shepherd
  - Westmere Cluster: 2 shepherds, 6 workers / shepherd
BRIEF SUMMARY OF PERFORMANCE EVALUATIONS
Results on the Cray XC (LLVM-unopt vs. LLVM-allopt)

Higher is better

Performance Improvement over LLVM-unopt

<table>
<thead>
<tr>
<th></th>
<th>SW</th>
<th>Cholesky</th>
<th>Sobel</th>
<th>StreamEP</th>
<th>EP</th>
<th>SSCA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improvement</td>
<td>2.1x</td>
<td>19.5x</td>
<td>1.1x</td>
<td>2.4x</td>
<td>1.4x</td>
<td>1.3x</td>
</tr>
</tbody>
</table>

4.6x performance improvement relative to LLVM-unopt on the same # of locales on average (1, 2, 4, 8, 16, 32, 64 locales)
Results on Westmere Cluster (LLVM-unopt vs. LLVM-allopt)

- Coalescing
- Locality Opt
- Aggregation
- Existing

Performance Improvement over LLVM-unopt

- SW: 2.3x
- Cholesky: 16.9x
- Sobel: 1.1x
- StreamEP: 2.5x
- EP: 1.3x
- SSCA2: 2.3x

4.4x performance improvement relative to LLVM-unopt on the same # of locales on average (1, 2, 4, 8, 16, 32, 64 locales)
Performance Evaluation

DETAILED RESULTS & ANALYSIS OF CHOLESKY DECOMPOSITION
Cholesky Decomposition

Node0
0
0 0
0 0 0

Node1
1 1 1 1
1 1 1 1 1

Node2
2 2 2 2 2 2 2
2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2

Node3
3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3

dependencies
Metrics

1. Performance & Scalability
   - Baseline (LLVM-unopt)
   - LLVM-based Optimizations (LLVM-allopt)

2. The dynamic number of communication API calls

3. Analysis of optimized code

4. Performance comparison
   - Conventional C-backend vs. LLVM-backend
Performance Improvement by LLVM (Cholesky on the Cray XC)

LLVM-based communication optimizations show scalability
Communication API calls elimination by LLVM (Cholesky on the Cray XC)

<table>
<thead>
<tr>
<th>API Call</th>
<th>LLVM-unopt</th>
<th>LLVM-allopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL__GET</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>REMOTE_GET</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>LOCAL_PUT</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>REMOTE_PUT</td>
<td>89.2%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

**Dynamic number of communication API calls** (normalized to LLVM-unopt)

- LOCAL__GET: 8.3x improvement
- REMOTE_GET: 500x improvement
- LOCAL_PUT: 1.1x improvement
Analysis of optimized code

LLVM-unopt

```
for jB in zero..tileSize-1 {
  for kB in zero..tileSize-1 {
    for iB in zero..tileSize-1 {
      4GETS
      for iB in zero..tileSize-1 {
        9GETS + 1PUT
      }
    }
  }
}
```
Performance comparison with C-backend

C-backend is faster!
In LLVM 3.3, many optimizations assume that the pointer size is the same across all address spaces.

1. Needs more instructions
2. Lose opportunities for Alias analysis
Conclusions

- LLVM-based Communication optimizations for PGAS Programs
  - Promising way to optimize PGAS programs in a language-agnostic manner
  - Preliminary Evaluation with 6 Chapel applications
    - Cray XC30 Supercomputer
      - 4.6x average performance improvement
    - Westmere Cluster
      - 4.4x average performance improvement
Future work

- Extend LLVM IR to support parallel programs with PGAS and explicit task parallelism
  - Higher-level IR

Parallel Programs (Chapel, X10, CAF, HC, …)

1. RI-PIR Gen
2. Analysis
3. Transformation

1. RS-PIR Gen
2. Analysis
3. Transformation

LLVM
Runtime-Independent Optimizations
e.g. Task Parallel Construct

LLVM
Runtime-Specific Optimizations
e.g. GASNet API

Binary
Acknowledgements

- Special thanks to
  - Brad Chamberlain (Cray)
  - Rafael Larrosa Jimenez (UMA)
  - Rafael Asenjo Plaza (UMA)
  - Habanero Group at Rice
Backup slides
Compilation Flow

Chapel Programs

AST Generation and Optimizations

C-code Generation

C Programs

Backend Compiler’s Optimizations (e.g., gcc–O3)

Binary

LLVM IR Generation

LLVM IR

LLVM Optimizations

Binary