Exploring Compiler Optimization Opportunities for the OpenMP 4.x Accelerator Model on a POWER8+GPU Platform

Akihiro Hayashi (Rice), Jun Shirako (Rice)
Ettore Tiotto (IBM), Robert Ho (IBM)
Vivek Sarkar (Rice)
GPUs Everywhere

This talk = **OpenMP** + **LLVM** + **GPGPU Compilers**

- **Supercomputers**: ✔100+ accelerated systems on Top500 list
- **Personal Computers**: ✔GPU acceleration for 2D/3D Rendering, Video encoding/decoding, and so on.
- **Smartphones**:

A gap between domain experts and hardware

Application Domain (Domain Experts)

Hardware: GPUs (Concurrency Experts)

Want to get significant performance improvement “easily”

Hard to exploit the full capability of hardware

Prog Lang. Compilers Runtime
A gap between domain experts and hardware

Application Domain (Domain Experts)

We believe Languages and Compilers are very important!

Want to get significant performance improvement easily

Prog Lang. Compilers Runtime

Hard to exploit the full capability of hardware

Hardware: GPUs (Concurrency Experts)
Directive-based Accelerator Programming Models

- High-level abstraction of GPU parallelism
  - OpenACC
  - OpenMP 4.0/4.5 target

```c
#pragma omp target map(tofrom: C[0:N]) map(to: B[0:N], A[0:N])
#pragma omp teams num_teams(N/1024) thread_limit(1024)
#pragma omp distribute parallel for
for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```
From the perspective of compilers...

- Directive-based programming models impose additional optimizations on compilers
  - Kernel optimizations
  - Data transfer optimizations

Additional dragon: GPUs

Credits: dragon by Cassie McKown from the Noun Project, crossed swords by anbileru adaleru from the Noun Project, https://en.wikipedia.org/
Where are we so far?

- Two OpenMP compilers
  - Clang+LLVM Compiler
  - IBM XL C/C++/Fortran Compiler

Matrix Multiplication (Kernel Only)

- Speedup over naïve CUDA
  - Naïve: 1.00
  - Hand-tuned: 3.16
  - clang: 1.41
  - XL C: 1.45

MRI Reconstruction (Kernel Only)

- Speedup over naïve CUDA
  - Naïve: 1.00
  - CUDA: 0.67
  - OpenMP: 0.82

GPU: Tesla K80
Open Question:
How to optimize OpenMP programs?

- Problem:
  - OpenMP versions are in general slower than well-tuned CUDA version

- This talk is
  - About studying potential performance improvements by OpenMP compilers
    - With detailed performance analyses
    - Current Focus: GPU Kernel Only
  - NOT about proposing a new GPU optimization
  - NOT about pushing a specific compiler
Compiling OpenMP to GPUs

- C/C++ (OpenMP)
  - Clang+LLVM
    - LLVM IR
    - CPU
    - LLVM optimization passes
    - GPU
    - LLVM IR
    - CPU
    - PowerPC backend
    - GPU
    - NVPTX backend
    - PTX
    - CPU + GPU executable

- XL Compiler
  - C/C++ Fortran (OpenMP)
    - Wcode
    - High-level Optimizer + CPU/GPU Partitioner
    - Wcode
    - CPU
    - POWER low-level optimizer
    - GPU
    - NVVM IR
    - libNVVM
    - PTX
    - CPU + GPU executable
OpenMP Threading Model on GPUs

```c
#pragma omp target teams {
    // sequential region 1
    if (…) {
        // parallel region 1
        #pragma omp parallel for
        for () {} {}
    }
    
    Team 0
    SMX0 (GPU)
    Master thread only

    Team 1
    SMX1 (GPU)
    Master thread only
```
State Machine Execution on GPUs

- Requires state machine execution in general:
  - Increasing register pressure
  - Increasing branch divergence

Note: the latest clang and XL no longer use this scheme
Optimization Example: Simplifying State Machine Execution on GPUs

- **What programmers can do**
  - Use “combined” directives if possible

```c
#pragma omp target teams distributed parallel for ...
for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
}
```

- **What compilers can do**
  - Remove state machine execution if possible
  - IBM XL compiler performs such an optimization
Performance Evaluation & Analysis

- **Platform**
  - CPUs: IBM POWER8 CPUs (S824)
    - 2 x 12-core IBM POWER8 running at up to 3.52GHz
    - 1 TB of main memory
  - GPUs: NVIDIA Tesla K80 GPU
    - 13 SMXs, each of which has 192 CUDA cores running at up to 875 MHz
    - 12 GB of global memory

- **Compilers**
  - clang+LLVM (as of October 2015)
    - clang 3.8: -O3
  - XL C Compiler (as of June 2016)
    - A beta version of IBM XL C/C++ 13.1.4: -O3
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>Memory Bounded</td>
<td>67,108,864</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Memory Bounded</td>
<td>67,108,864</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>Memory Bounded</td>
<td>2,048x2,048</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>Computation Bounded</td>
<td>4,194,304</td>
</tr>
<tr>
<td>SPEC ACCEL SP-XSOLVE3</td>
<td>Memory Bounded</td>
<td>5x255x256x256</td>
</tr>
<tr>
<td>SPEC ACCEL OMRIQ</td>
<td>Computation Bounded</td>
<td>262,144</td>
</tr>
</tbody>
</table>

**Variants:**
- CUDA: CUDA-baseline, CUDA+Read-only-cache (ROC)
- OMP4: clang+llvm, XL C (Both utilize ROC by default)
**Insight #1 Minimize OpenMP runtime overheads when possible**

- **Combined:** use combined constructs (No State Machine)
- **Non-combined:** put OMP directives on different lines (State Machine)

<table>
<thead>
<tr>
<th>Function</th>
<th>clang-combined</th>
<th>clang-non-combined</th>
<th>XLC-combined</th>
<th>XLC-non-combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saxpy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMRIQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Higher is better*

State Machine versions are slower than NON-State machine versions (1LV only)

**XL C optimizes it!**

Speedup over CUDA-baseline

1LV (combined and non-combined)

Geomean
Insight #2 Utilize read-only cache carefully

- The Read only cache does not always contribute to performance improvements.

<table>
<thead>
<tr>
<th>CUDA-baseline vs. CUDA-baseline + the read only cache</th>
<th>Higher is better</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup over CUDA-baseline</td>
<td></td>
</tr>
<tr>
<td>VecAdd</td>
<td>0.97</td>
</tr>
<tr>
<td>Saxpy</td>
<td>0.97</td>
</tr>
<tr>
<td>MM</td>
<td>0.98</td>
</tr>
<tr>
<td>BS</td>
<td>0.96</td>
</tr>
<tr>
<td>OMRIQ</td>
<td>1.01</td>
</tr>
<tr>
<td>SP</td>
<td>1.12</td>
</tr>
<tr>
<td>Geomean</td>
<td>1.00</td>
</tr>
<tr>
<td>1LV (combined and non-combined)</td>
<td></td>
</tr>
<tr>
<td>2LV (non-combined only)</td>
<td></td>
</tr>
</tbody>
</table>
Insight #3 Improve Math functions code generations

- BS and OMRIQ use Math functions heavily
- CUDA < XLC < clang

![Bar chart showing speedup over CUDA-baseline for BS, OMRIQ, and Geomean. Higher is better.]

- BS: clang-combined = 0.59, XLC-combined = 0.72
- OMRIQ: clang-combined = 0.67, XLC-combined = 0.82
- Geomean: clang-combined = 0.63, XLC-combined = 0.77
Insight #3 Improve Math functions code generations (Cont’d)

```c
#pragma omp target distributed parallel for
for (int i = 0; i < N; i++) {
    float T = exp(randArray[i]);
    call[i] = (float)log(randArray[i])/T;
}
```

<table>
<thead>
<tr>
<th>Compiler</th>
<th>exp (double)</th>
<th>expf (float)</th>
<th>exp (double)</th>
<th>expf (float)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA</td>
<td>515 us</td>
<td>933 us</td>
<td>922 us</td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xlc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Math API is better than libdevice.bc

Math API	API	is	beWer than libdevice

721 us when using expf and logf manually
Insight #4 Other important insights

- Generate Fused-Multiply-Add (FMA) if possible
  - \( fadd + fmul = fma \)
  - E.g. 1.5% Performance improvement in SAXPY (clang)

- Use schedule(static, 1)
  - For better memory coalescing

<table>
<thead>
<tr>
<th>Chunk Size</th>
<th>Speedup Over chunk=1(clang)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk=1</td>
<td>1.00</td>
</tr>
<tr>
<td>chunk=2</td>
<td>0.48</td>
</tr>
<tr>
<td>chunk=4</td>
<td>0.27</td>
</tr>
<tr>
<td>chunk=8</td>
<td>0.14</td>
</tr>
</tbody>
</table>
Insight #5  Loop transformation is much more important (High-level Restructuring)

Performing loop transformations by hand

Significant Performance improvement by loop distribution + permutation + tiling

Future work: High-level loop transformation framework
Insight #5  Loop transformation is much more important (Parameter tuning)

Matrix Multiplication

```java
for (int k = 0; k < N; k++) {
    sum += A[i*N+k] * B[k*N+threadIdx.x];
}
```

unrolling factor = 8
L2 Hit (14.46%).

unrolling factor = 2
L2 Hit (92.02%).

8 stride + 8 offset access simultaneously

2 stride + 2 offset access simultaneously

1.41x
Summary

- OpenMP versions are in general slower than well-tuned CUDA version
- For future high-level compiler optimizations
  - Insight #1: Minimize OpenMP overheads when possible
  - Insight #2: Utilize read-only cache carefully
  - Insight #3: Improve math functions generation
  - Insight #4: Other important insights
    - E.g. FMA contraction, scheduling clause
  - Insight #5: Loop transformation is much more important
- The paper includes more detailed information including hardware counter numbers
On Going & Future work

- Minimizing OpenMP runtime overhead
  - The latest versions of clang and XL C no longer generate state machine
  - Plan to performance evaluation & analysis again

- High-level loop transformations for OpenMP programs
  - For better memory coalescing and the read-only cache/shared memory exploitation
    - High-level Restructuring: Loop permutation, loop fusion, loop distribution, loop tiling,
    - Turning parameters: Tile size, Unrolling factor
  - Using the polyhedral model + DL Model [SC’14]
  - Taking additional information from users [PACT’15]

- Automatic Device Selection [PACT’15, PPPJ’15]
Acknowledgement

- IBM CAS
- Dr. Wayne Joubert
- WACCPD reviewers and committee members
Backup
Insight #3 Improve Math functions code generations (Cont’d)

<table>
<thead>
<tr>
<th>BlackScholes</th>
<th>CUDA</th>
<th>clang</th>
<th>xlc</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP insn (single)</td>
<td>704M</td>
<td>452M</td>
<td>340M</td>
</tr>
<tr>
<td>FP insn (double)</td>
<td>29M</td>
<td>583M</td>
<td>406M</td>
</tr>
</tbody>
</table>

- CUDA tries to use float versions of math functions as much as possible
- XLC does the similar things, but CUDA compiler is more powerful
Overall Results

higher is better

<table>
<thead>
<tr>
<th>1LV (combined and non-combined)</th>
<th>2LV (non-combined only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td>1.38 1.41 1.45 1.45</td>
</tr>
<tr>
<td>Saxpy</td>
<td>0.97 0.97 0.97 0.97</td>
</tr>
<tr>
<td>MM</td>
<td>0.95 0.75 0.75 0.75</td>
</tr>
<tr>
<td>BS</td>
<td>0.96 0.59 0.38 0.72</td>
</tr>
<tr>
<td>OMRIQ</td>
<td>1.01 0.67 0.56 0.82</td>
</tr>
<tr>
<td>SP</td>
<td>1.20 0.82 0.82 1.14</td>
</tr>
</tbody>
</table>