Performance Evaluation of OpenMP’s Target Construct on GPUs
- Exploring Compiler Optimizations -[2] ¹

Shelby Lockhart
CS 598

¹*All graphs and tables taken from paper unless otherwise noted.*
This paper addresses the same types of questions we have discussed in class, but in reference to OpenMP.

The **target** construct is a high level abstraction for offloading to GPUs.

Because the offloading is now hidden, there is the question of optimality.

- Is the compiler actually generating the code I want?

- Can I expect similar performance to hand-tuned CUDA code using these constructs?
Outline

1. The OpenMP Accelerator Model
   1. GPU Model
   2. OpenMP target Constructs

2. Compiling OpenMP to GPUs
   1. Compilation Flow for the Tested Compilers
   2. OpenMP Threading Model on GPUs

3. Performance Evaluation
   1. Benchmark Results
   2. Comparisons with Naive CUDA Code
   3. Comparisons with Highly-Tuned CUDA Code

4. Conclusions

5. References
The OpenMP Accelerator Model
GPU Model

*Image taken from [1]
target Constructs

- Introduced in OpenMP 4.0
- Designates portion of code to be offloaded to a device

```
#pragma omp target
...
```

- Designates data transfers between the host and the device

```
#pragma omp target map(from: C) map(to: B, A)
...
```

- Sets grid size - number of blocks and threads per block

```
#pragma omp target map(from: C) map(to: B, A)
#pragma omp teams num_teams(N/1024) thread_limit(1024)
...
```
target Constructs

- Specifies the number of iterations per team and the iterations per thread

```cpp
#pragma omp target map(from: C) map(to: B, A)
#pragma omp teams num_teams(N/1024) thread_limit(1024)
#pragma omp distribute parallel for \
        dist_schedule(static, distChunk) schedule(static, 1)
for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

- Scheduling 1 thread per iteration is necessary to take advantage of memory coalescing
target Constructs

- These constructs can be split across pragmas or placed on the same line, the same as all other OpenMP constructs

```c
#pragma omp target map(from: C) map(to: B, A) \ 
  teams num_teams(N/1024) thread_limit(1024) \ 
  distribute parallel for \ 
  dist_schedule(static, distChunk) schedule(static, 1) 
for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
}
```

- clang+LLVM and IBM XL C compilers treat both variants the same ... so they claim
Compiling OpenMP to GPUs
Tested Compilers

- Tested compilers: clang+LLVM and IBM XL C
- The main difference between these compilers is how they generate the PTX GPU assembly code. clang+LLVM generates the PTX directly, while the IBM XL C uses the NVVM IR.
OpenMP Threading Model on GPUs

- OpenMP code can generally include sequential and parallel sections interleaved

```c
#pragma omp target teams { // GPU region
  // sequential region 1 executed by the master thread of each team
  if (...) {
    // parallel region 1
    #pragma omp parallel for
    for () {}
  } else {
    ... }  
}
```

- How does this work on GPUs?
  - State Machine Execution
  - Master/Worker Execution
bool finished = false;
while (!finished) {
  switch (labelNext) {
    case SEQUENTIAL_REGION1:
      if (threadIdx.x != MASTER)
        break;
      // code for sequential region
      if (...) {
        labelNext = PARALLEL_REGION1;
      }
      break;
    case PARALLEL_REGION1:
      // code for parallel region 1
      if (threadIdx.x == MASTER) {
        // update labelNext;
      }
      break;
    // other cases
    case END:
      labelNext = -1;
      finished = true;
      break;
  }
  __syncthreads();
}
Master/Worker Execution

```c
if ( masterWarp ) {
  // code for sequential region 1
  if (...) {
    // code for parallel region 1
    [activate workers]
    bar.sync 0 // synchronization
    bar.sync 0 // synchronization
  }
} else {
  // Worker Warps
  bar.sync 0 // synchronization
  // get a chunk of parallel loop
  // and execute it in parallel
  executeParallelLoop ();
  bar.sync 0 // synchronization
}
// outlined work for worker warps
executeParallelLoop ();
```

- Similar to OpenMP standard fork/join model
- Runtime distinguishes a master warp within each block and all other warps are worker warps
- Advantages
  - Simplifies code generation
  - Less register pressure than State Execution Model
  - Can support orphaned parallel directives in external functions
Generated when there is no sequential region within a **target** region

There is no performance penalty from control-flow instructions

clang+LLVM supports this only for combined constructs

IBM XL C can generate the appropriate execution scheme for combined and non-combined constructs
Potential Optimizations

- Using shared memory and the read-only data cache on GPUs can improve kernel performance
  - Neither compiler optimizes target constructs to use shared memory
  - NVPTX backend and libNVVM use read-only cache for all data when the target architecture is sm_35 or later
  - Placing all possible data in the read-only data cache can cause performance slowdown

- Leveraging the instruction level parallelism (ILP) on GPUs
  - The thread level parallelism of the OpenMP model cannot always be interchanged with the ILP of GPUs
  - clang+LLVM, NVPTX backend, and libNVVM take advantage of ILP by unrolling sequential loops to increase ILP when possible
Experimental Setup

- Testing - look at potential compiler optimizations for OpenMP in terms of kernel performance
- Comparing against naive CUDA implementation, then hand-tuned CUDA against 'optimized' OpenMP
- NVIDIA GPUs
  - Tesla K80 - 13 SMs w/ 192 cores each, speed of 875 MHz, 12 GB of memory
  - Tesla P100 - 56 SMs w/ 64 cores each, speed of 1.36 GHz, 4 GB of memory
- Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Data Size</th>
<th>Target Directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAdd</td>
<td>Vector Addition (C=A+B)</td>
<td>67,108,864</td>
<td>1-level</td>
</tr>
<tr>
<td>Saxpy</td>
<td>Single-Precision scalar multiplication and vector addition (Z=A×X+Y)</td>
<td>67,108,864</td>
<td>1-level</td>
</tr>
<tr>
<td>MM</td>
<td>Matrix Multiplication (C=A×B)</td>
<td>2,048 × 2,048</td>
<td>1-level</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>Theoretical estimation of the European style options</td>
<td>4,194,304</td>
<td>1-level</td>
</tr>
<tr>
<td>OMRIQ</td>
<td>3-D MRI reconstruction from SPEC ACCEL™ (SPEC 2015)</td>
<td>32,768</td>
<td>1-level</td>
</tr>
<tr>
<td>SP-xsolve3</td>
<td>Scalar Penta-diagonal solver from SPEC ACCEL™ (SPEC 2015)</td>
<td>5 × 255 × 256 × 256</td>
<td>2-level</td>
</tr>
</tbody>
</table>
Performance Evaluation: Naive Code Comparisons
CUDA (baseline): A CUDA version with the read-only data cache disabled
CUDA-ROC (K80 only): All read-only arrays within a kernel are accessed through the read-only data cache
*CUDA-ROC gone because read-only data cache no longer available in P100 GPUs
Results - Overhead of OpenMP’s Execution Model

- Non-Combined vs Combined Pragmas
  - IBM XL C shows the same speedup for both pragma types
  - clang+LLVM sees worse performance for non-combined pragmas, and the assembly code showed more integer, control flow, and load-store instructions

- OpenMP Runtime Library Overhead
  - clang+LLVM eliminated unnecessary OpenMP runtime calls on the P100 GPU, but not the K80
  - IBM XL C failed to eliminate unnecessary OpenMP runtime calls on either GPU
  - For the vector addition benchmark, this overhead accounted for 85% of the execution time for clang+LLVM combined and 75.3% of execution time for IBM XL C combined

```
#pragma omp target teams num_teams(N/1024) thread_limit(1024) \\
    distribute parallel for schedule(static, 1)
for (int i = 0; i < N; i++) {
    // do nothing
}
```

<table>
<thead>
<tr>
<th>Grid Size (N/1024)</th>
<th>1</th>
<th>64</th>
<th>1024</th>
<th>4096</th>
<th>16384</th>
<th>65536</th>
</tr>
</thead>
<tbody>
<tr>
<td>K80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>5.5 us</td>
<td>20.3 us</td>
<td>281.3 us</td>
<td>1.1 ms</td>
<td>4.4 ms</td>
<td>17.6 ms</td>
</tr>
<tr>
<td>XL C</td>
<td>3.6 us</td>
<td>9.2 us</td>
<td>117.9 us</td>
<td>464.6 us</td>
<td>1.8 ms</td>
<td>7.3 ms</td>
</tr>
<tr>
<td>P100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>1.1 us</td>
<td>1.4 us</td>
<td>7.3 us</td>
<td>26.5 us</td>
<td>103.5 us</td>
<td>411.2 us</td>
</tr>
<tr>
<td>XL C</td>
<td>3.3 us</td>
<td>6.2 us</td>
<td>43.7 us</td>
<td>163.5 us</td>
<td>643.5 us</td>
<td>2.5 ms</td>
</tr>
</tbody>
</table>
Subtracting the overhead of OpenMP from the BlackScholes benchmark which has a large number of floating point operations, the CUDA version is fastest with IBM XL C coming in second.

Benchmarking the math function code generation

```c
// a[] and b[] are float arrays
#pragma omp target teams distribute parallel for ...
for (int i = 0; i < N; i++) {
    float T = exp(a[i]); // double exp(double)
    b[i] = (float)log(a[i])/T; // double log(double)
}
```

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>clang</th>
<th>XL C</th>
</tr>
</thead>
<tbody>
<tr>
<td>K80</td>
<td>Original</td>
<td>472.5 us</td>
<td>734.0 us</td>
</tr>
<tr>
<td></td>
<td>Hand Conversion</td>
<td>472.5 us</td>
<td>ptxas error</td>
</tr>
<tr>
<td>P100</td>
<td>Original</td>
<td>139.8 us</td>
<td>229.7 us</td>
</tr>
<tr>
<td></td>
<td>Hand Conversion</td>
<td>139.8 us</td>
<td>ptxas error</td>
</tr>
</tbody>
</table>

- clang+LLVM generates double-precision versions of `exp()` and `log()`
- Both nvcc for CUDA version and IBM XL C generate single-precision versions of `exp()` and `log()`, and inline the functions in PTX assembly
- Both clang+LLVM and IBM XL C use libdevice
- nvcc compiled CUDA uses the CUDA Math API
Fused-Multiply-Add (FMA) instructions
- clang+LLVM does not generate FMA instructions by default
- Users can force clang+LLVM to generate FMA instructions when converting to PTX by providing the flags: `-mllvm -nvptx-fma-level=1 or 2`

Schedule(static, 1) for memory access coalescing
- Scheduling a chunk size of 1 for each thread allows consecutive threads to access consecutive global memory locations
- Default scheduling is implementation defined, so it’s best to specify a chunk size of 1 because performance degrades as chunk size increases, as seen below for the VecAdd benchmark

<table>
<thead>
<tr>
<th>Chunk Size</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>K80</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>20.8 ms</td>
<td>37.4 ms</td>
<td>40.1 ms</td>
<td>52.1 ms</td>
<td>89.8 ms</td>
<td>228.6 ms</td>
</tr>
<tr>
<td>XL C</td>
<td>9.6 ms</td>
<td>13.4 ms</td>
<td>15.3 ms</td>
<td>22.8 ms</td>
<td>42.8 ms</td>
<td>106.2 ms</td>
</tr>
<tr>
<td><strong>P100</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>2.2 ms</td>
<td>2.3 ms</td>
<td>2.5 ms</td>
<td>5.0 ms</td>
<td>16.4 ms</td>
<td>26.1 ms</td>
</tr>
<tr>
<td>XL C</td>
<td>4.7 ms</td>
<td>4.8 ms</td>
<td>5.0 ms</td>
<td>5.7 ms</td>
<td>6.1 ms</td>
<td>10.2 ms</td>
</tr>
</tbody>
</table>
Performance Evaluation: Highly Tuned Code Comparisons
#pragma omp target teams distribute ...
for (int k = 1; k <= nz2; k++) {
    #pragma omp parallel for ...
    for (int j = 1; j <= ny2; j++) {
        // loop 1
        for (int i = 0; i <= gp01; i++) {
            rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
        }
        // loop 2
        for (int i = 1; i <= nx2; i++) {
            lhsX[0*LHSX1 + k*LHSX3 + j] = 0.0;
            ...
        }
    }
}

- Memory accesses for loop2 are coalesced, but not loop1
- Transform the kernel by splitting the j-loop into 2 different loops and making both have coalesced memory accesses
- Additionally, rhonX and lhsX could be loaded into shared memory, but only for the CUDA implementation
Hand Tuning Kernel in SP Benchmark

Performance Results

<table>
<thead>
<tr>
<th></th>
<th>Variants</th>
<th>CUDA</th>
<th>clang</th>
<th>XL C</th>
</tr>
</thead>
<tbody>
<tr>
<td>K80</td>
<td>Original</td>
<td>102.4 ms</td>
<td>104.5 ms</td>
<td>174.3 ms</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>27.1 ms</td>
<td>30.5 ms</td>
<td>39.3 ms</td>
</tr>
<tr>
<td></td>
<td>Transformed+SharedMemory</td>
<td>9.1 ms</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P100</td>
<td>Original</td>
<td>40.9 ms</td>
<td>40.9 ms</td>
<td>65.3 ms</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>12.6 ms</td>
<td>Error</td>
<td>11.3 ms</td>
</tr>
<tr>
<td></td>
<td>Transformed+SharedMemory</td>
<td>3.5 ms</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Transformed Kernel

```c
#pragma omp target teams distribute ... 
for (int k = 1; k <= nz2; k++) {
    #pragma omp parallel for ...
    for (int i = 0; i <= gp01; i++) {
        /* loop1 */ for (int j = 1; j <= ny2; j++) {
            rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
        }
    }
    #pragma omp parallel for ...
    for (int j = 1; j <= ny2; j++) {
        /* loop2 */ for (int i = 1; i <= nx2; i++) {
            lhsX[0*LHSX1 + k*LHSX3 + j] = 0.0;
            ...
        }
    }
}
```
Hand Tuned MM Benchmark

- Transformed Matrix Multiply Kernel by employing loop tiling for more coalesced memory accesses for the OpenMP variants
- The CUDA kernel employed loop tiling and utilized shared memory

<table>
<thead>
<tr>
<th>Variants</th>
<th>CUDA</th>
<th>clang</th>
<th>XL C</th>
</tr>
</thead>
<tbody>
<tr>
<td>K80 Original</td>
<td>231.7 ms</td>
<td>223.1 ms</td>
<td>234.8 ms</td>
</tr>
<tr>
<td>Transformed (Tiling)</td>
<td>192.3 ms</td>
<td>224.9 ms</td>
<td>157.9 ms</td>
</tr>
<tr>
<td>Transformed+SharedMemory</td>
<td>70.6 ms</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P100 Original</td>
<td>74.7 ms</td>
<td>65.9 ms</td>
<td>65.4 ms</td>
</tr>
<tr>
<td>Transformed (Tiling)</td>
<td>49.6 ms</td>
<td>74.6 ms</td>
<td>62.4 ms</td>
</tr>
<tr>
<td>Transformed+SharedMemory</td>
<td>8.6 ms</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Conclusions

- The OpenMP `target` construct is not consistently slower or faster than CUDA implementations.

Areas for improvement:

- Minimizing OpenMP runtime overheads.
- Better data placement policy for the read-only cache and shared memory.
- Improving code generation for threads (math functions and coalesced memory accesses).
- Employing the use of high-level loop transformations.
Imen Chakroun, Nouredine Melab, Mohand-Said Mezmaz, and Daniel Tuyttens.
Combining multi-core and gpu computing for solving combinatorial optimization problems.

Performance evaluation of openmp’s target construct on gpus’.