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.

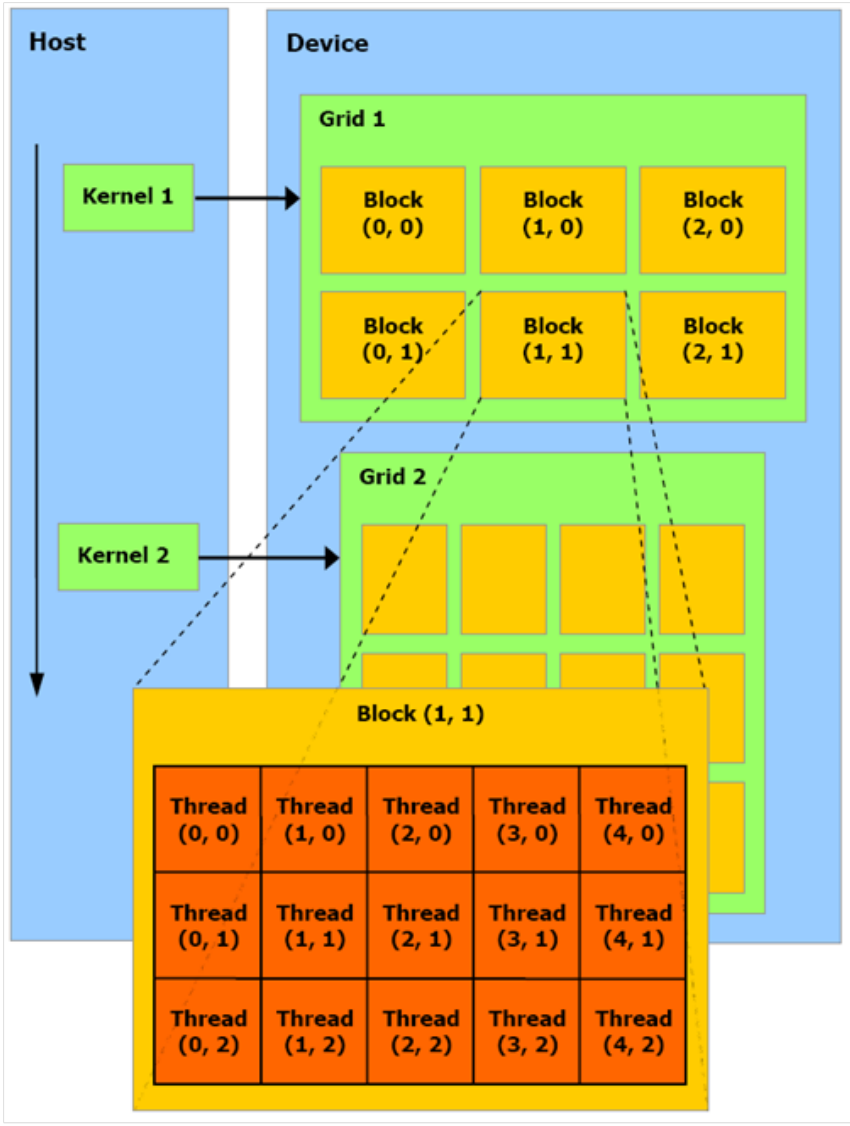
How Is This Relevant to the Course?

- 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

The OpenMP Accelerator Model

GPU Model



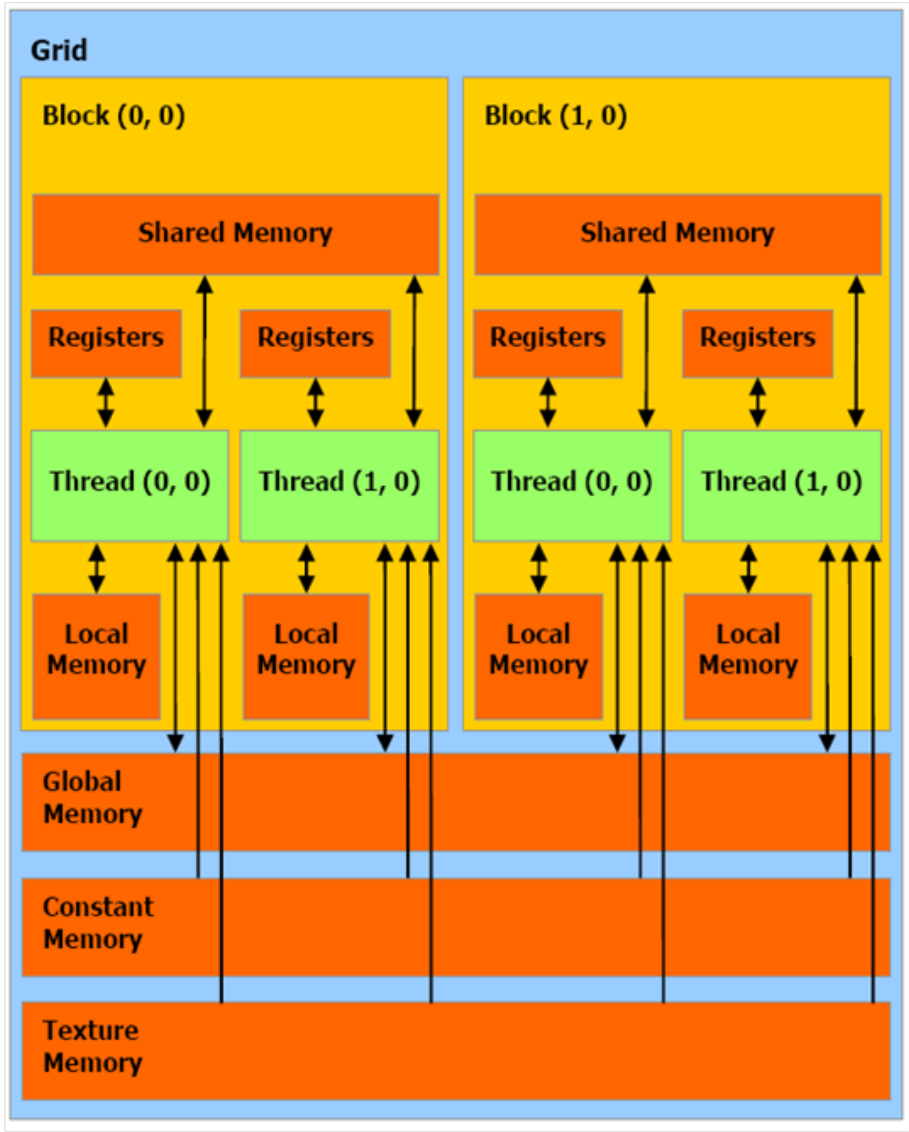


*Image taken from [1]

target Constructs

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

```
1         #pragma omp target
2         ...
```

- Designates data transfers between the host and the device

```
1         #pragma omp target map(from: C) map(to: B, A)
2         ...
```

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

```
1         #pragma omp target map(from: C) map(to: B, A)
2         #pragma omp teams num_teams(N/1024) thread_limit(1024)
3         ...
```

target Constructs

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

```
1     #pragma omp target map(from: C) map(to: B, A)
2     #pragma omp teams num_teams(N/1024) thread_limit(1024)
3     #pragma omp distribute parallel for \
4                 dist_schedule(static, distChunk) schedule(static, 1)
5     for (int i = 0; i < N; i++) {
6         C[i] = A[i] + B[i];
7     }
```

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

target Constructs

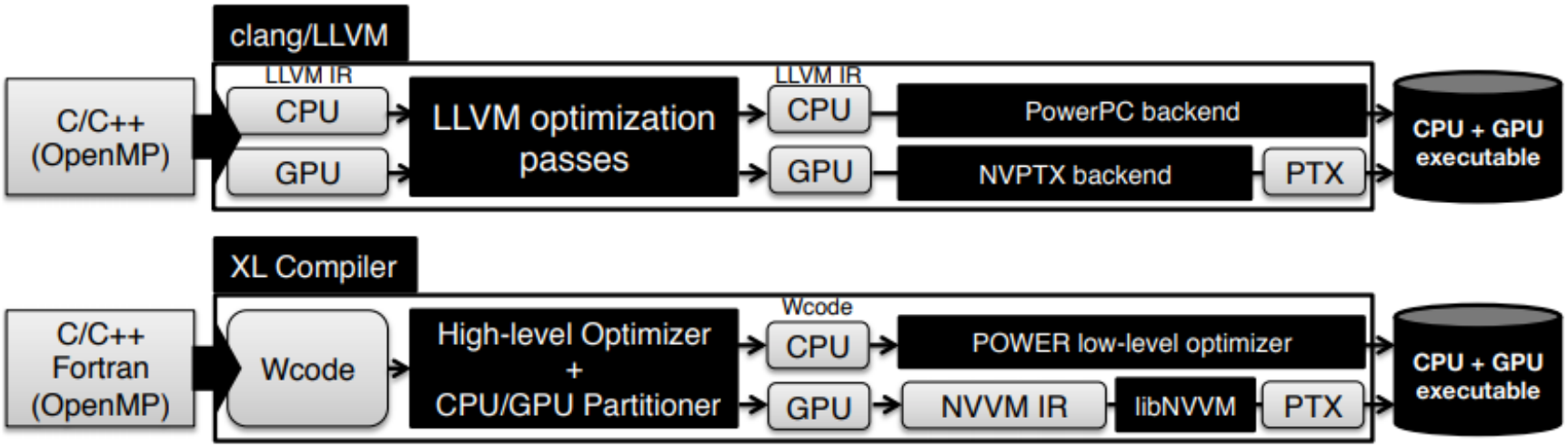
- These constructs can be split across pragmas or placed on the same line, the same as all other OpenMP constructs

```
#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

```
#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

State Machine Execution

```
1  bool finished = false;
2  while (! finished ) {
3      switch ( labelNext ) {
4      case SEQUENTIAL_REGION1 :
5          if ( threadIdx .x != MASTER)
       break;
6          // code for sequential region
       1
7          if (...) {
8              labelNext =
       PARALLEL_REGION1 ;
9          }
10         break;
11     case PARALLEL_REGION1 :
12         // code for parallel region 1
13         if ( threadIdx .x == MASTER) {
14             // update labelNext;
15         }
16         break;
17     // other cases
18     case END:
19         labelNext = -1;
20         finished = true;
21         break;
22     }
23 __syncthreads();
24 }
```

- Sequential and parallel regions are assigned different states
- State transitions occur dynamically
- Drawbacks:
  - Register pressure can increase
  - Large numbers of control-flow instructions can be generated

Master/Worker Execution

```
1  if ( masterWarp ) {
2      // code for sequential region 1
3      if (...) {
4          // code for parallel region 1
5          [activate workers]
6          bar.sync 0 // synchronization
7          bar.sync 0 // synchronication
8      }
9  } else {
10     // Worker Warps
11     bar.sync 0 // synchronization
12     // get a chunk of parallel loop
13     // and execute it in parallel
14     executeParallelLoop ();
15     bar.sync 0 // synchronization
16 }
17 // outlined work for worker warps
18 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

Alternative Code Generation Scheme

- 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 paralellism (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

Performance Evaluation

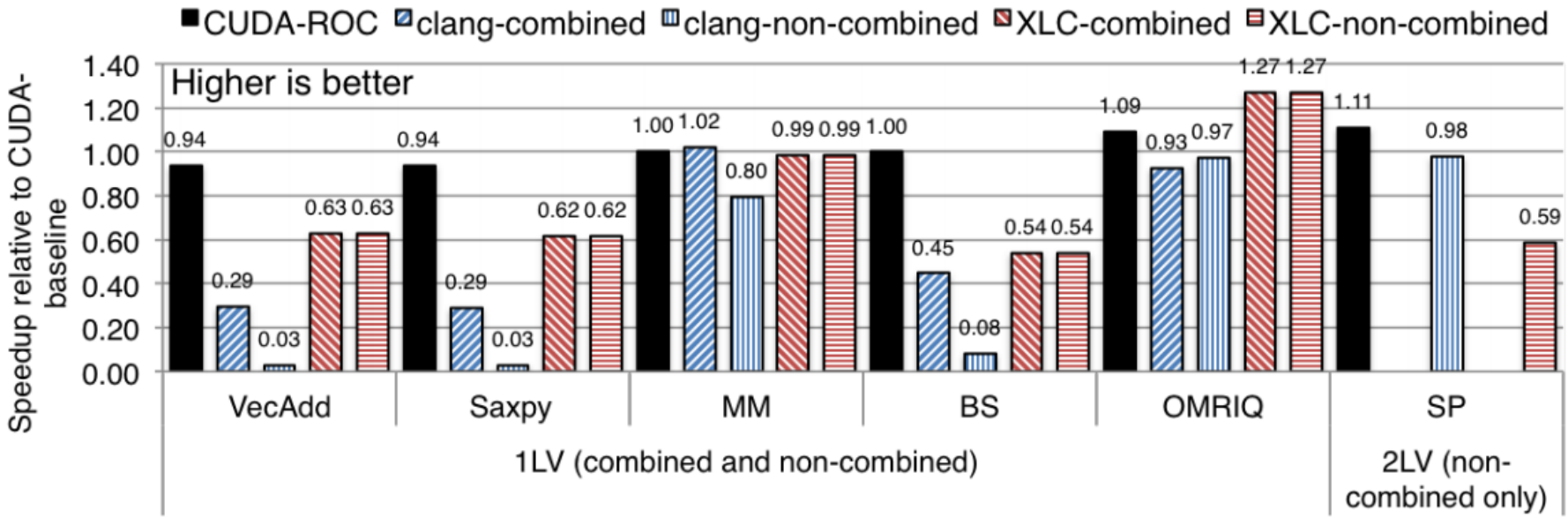
Experimental Setup

- 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

Benchmark	Description	Data Size	Target Directives
VecAdd	Vector Addition (C=A+B)	67,108,864	1-level
Saxpy	Single-Precision scalar multiplication and vector addition (Z=A×X+Y)	67,108,864	1-level
MM	Matrix Multiplication (C=A×B)	2,048 × 2,048	1-level
BlackScholes	Theoretical estimation of the European style options	4,194,304	1-level
OMRIQ	3-D MRI reconstruction from SPEC ACCEL™ (SPEC 2015)	32,768	1-level
SP-xsolve3	Scalar Penta-diagonal solver from SPEC ACCEL™ (SPEC 2015)	5 × 255 × 256 × 256	2-level

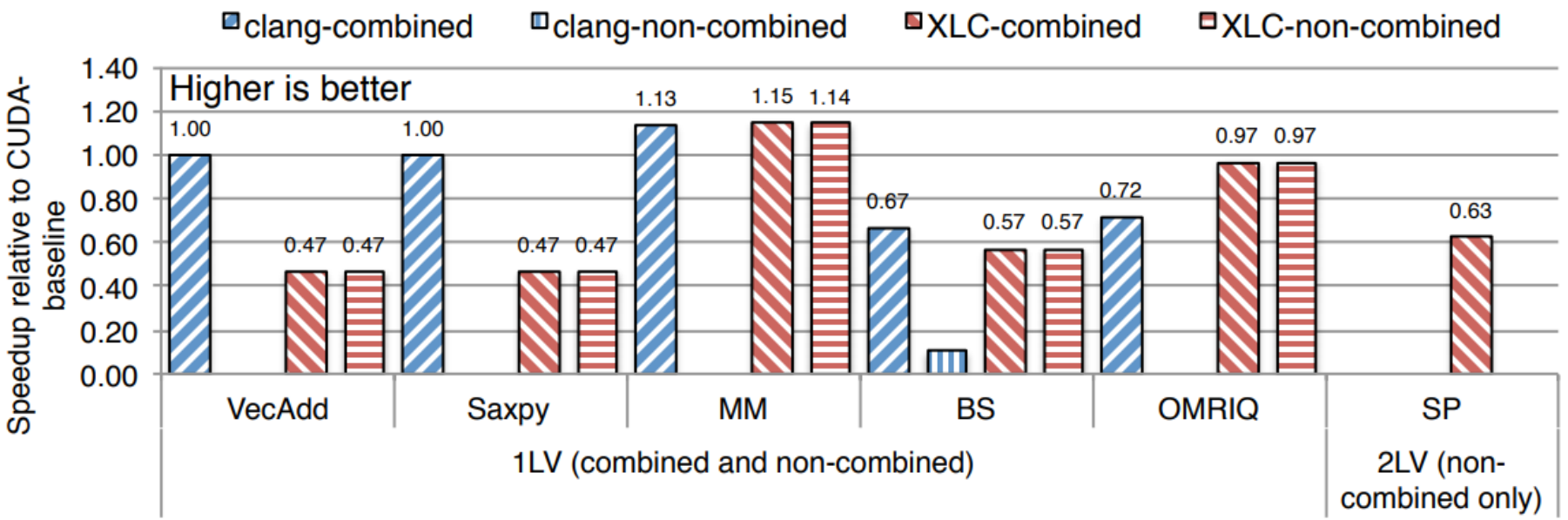
Performance Evaluation : Naive Code Comparisons

Results - IBM POWER8 + NVIDIA Tesla K80



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

Results - IBM POWER8 + NVIDIA Tesla P100



*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
}
```

	Grid Size (N/1024)	1	64	1024	4096	16384	65536
K80	clang	5.5 us	20.3 us	281.3 us	1.1 ms	4.4 ms	17.6 ms
	XL C	3.6 us	9.2 us	117.9 us	464.6 us	1.8 ms	7.3 ms
P100	clang	1.1 us	1.4 us	7.3 us	26.5 us	103.5 us	411.2 us
	XL C	3.3 us	6.2 us	43.7 us	163.5 us	643.5 us	2.5 ms

Results - Math Function Code Generation

- 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

```
1  // a[] and b[] are float arrays
2  #pragma omp target teams distribute parallel for ...
3  for (int i = 0; i < N; i++) {
4      float T = exp(a[i]);          // double exp(double)
5      b[i] = (float)log(a[i])/T;    // double log(double)
6  }
```

		CUDA	clang	XL C
K80	Original	472.5 us	734.0 us	725.4 us
	Hand Conversion	472.5 us	`ptxas` error	494.2 us
P100	Original	139.8 us	229.7 us	171.8 us
	Hand Conversion	139.8 us	`ptxas` error	153.3 us

- 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

Results - FMA and Memory Coalescing

- 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

	Chunk Size	1	2	4	8	16	32
K80	clang	20.8 ms	37.4 ms	40.1 ms	52.1 ms	89.8 ms	228.6 ms
	XL C	9.6 ms	13.4 ms	15.3 ms	22.8 ms	42.8 ms	106.2 ms
P100	clang	2.2 ms	2.3 ms	2.5 ms	5.0 ms	16.4 ms	26.1 ms
	XL C	4.7 ms	4.8 ms	5.0 ms	5.7 ms	6.1 ms	10.2 ms

Performance Evaluation : Highly Tuned Code Comparisons

Hand Tuning Kernel in SP Benchmark

```
1   #pragma omp target teams distribute ...
2   for (int k = 1; k <= nz2; k++) {
3       #pragma omp parallel for ...
4       for (int j = 1; j <= ny2; j++) {
5           // loop1
6           for (int i = 0; i <= gp01; i++) {
7               rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
8           }
9           // loop2
10          for (int i = 1; i <= nx2; i++) {
11              lhsX[0*LHSX1 + k*LHSX3 + j] = 0.0;
12              ...
13          }
14      }
15  }
```

- 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

	Variants	CUDA	clang	XL C
K80	Original	102.4 ms	104.5 ms	174.3 ms
	Transformed	27.1 ms	30.5 ms	39.3 ms
	Transformed+SharedMemory	9.1 ms	-	-
P100	Original	40.9 ms	40.9 ms	65.3 ms
	Transformed	12.6 ms	Error	11.3 ms
	Transformed+SharedMemory	3.5 ms	-	-

Transformed Kernel

```
1     #pragma omp target teams distribute ...
2     for (int k = 1; k <= nz2; k++) {
3         #pragma omp parallel for ...
4         for (int i = 0; i <= gp01; i++) {
5 /* loop1 */ for (int j = 1; j <= ny2; j++) {
6                 rhonX[k*RHONX1 + j*RHONX2 + i] = ...;
7             }
8         }
9         #pragma omp parallel for ...
10        for (int j = 1; j <= ny2; j++) {
11 /* loop2 */ for (int i = 1; i <= nx2; i++) {
12                lhsX[0*LHSX1 + k*LHSX3 + j] = 0.0;
13                ...
14            }
15        }
16    }
```

Hand Tuned MM Benchmark

	Variants	CUDA	clang	XL C
K80	Original	231.7 ms	223.1 ms	234.8 ms
	Transformed (Tiling)	192.3 ms	224.9 ms	157.9 ms
	Transformed+SharedMemory	70.6 ms	-	-
P100	Original	74.7 ms	65.9 ms	65.4 ms
	Transformed (Tiling)	49.6 ms	74.6 ms	62.4 ms
	Transformed+SharedMemory	8.6 ms	-	-

- 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

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

References I

Imen Chakroun, Nouredine Melab, Mohand-Said Mezmaz, and Daniel Tuyttens.
Combining multi-core and gpu computing for solving combinatorial optimization problems.
Journal of Parallel and Distributed Computing, 73:1563–1577, 12 2013.

Shirako J. Tiotto E. Ho R. Hayashi, A. and V. Sarkar.
Performance evaluation of openmp's target construct on gpus'.
International Journal of High Performance Computing and Networking, x(x):xxx–xxx.