GPGPU Offloading with OpenMP 4.5
In the IBM XL Compiler

Taylor Lloyd
Jose Nelson Amaral
Ettore Tiotto

University of Alberta
University of Alberta
IBM Canada
Why?
Supercomputer Power/Performance

- GPUs exhibit much better power scaling characteristics
- Upcoming CORAL Sierra Supercomputer will deliver 150 Petaflops
- Cooling is becoming a serious issue at scale

Agenda

1. GPU Programming Models
2. GPU Architecture
3. Mapping OpenMP to GPUs
4. OpenMP Performance
GPU Programming Models
GPU Parallel Programming Models

- NVidia GPUs
- AMD(ati) GPUs
- Xeon PHI (2014)
- CPU Parallelism

(2006) NVidia GPUs
(2009) NVidia GPUs, AMD(ati) GPUs, Xeon PHI (2014), CPU Parallelism
(2012) NVidia GPUs, AMD(ati) GPUs, Xeon PHI (2014), CPU Parallelism

Version 4.0 (2013) NVidia GPUs, AMD(ati) GPUs, Xeon PHI (2014), CPU Parallelism
Language Philosophies

- Exploit and NVidia CUDA GPUs
- Maximize performance by exposing hardware details (warps, shared memory)
- Ease debugging and performance through tooling and profiling
Language Philosophies

- Run Anywhere: CPU, GPU, Specialized Accelerators
- Runtime library provides access to parallel primitives
- OpenCL Compiler can build kernel functions at runtime
Language Philosophies

- Annotate existing programs with pragmas (acc kernels, acc parallel, acc data)
- Default to hierarchical parallelism
- The compiler knows best (Loose Spec)
Language Philosophies

- Annotate existing programs with pragmas (omp target, omp parallel, omp target data)
- Hierarchical parallelism available, simple parallelism by default
- The programmer knows best (Tight Spec)
Which one is “best”?

“The stream of code needed to parallelize for CUDA C was about 50 higher than for OpenACC and OpenCL was about three times”

_NextPlatform - Is OpenACC the Best Thing to Happen to OpenMP? (2015)_

“OpenMP is richer and has more features than OpenACC, we make no apologies about that. OpenACC is targeting scalable parallelism, OpenMP is targeting more general parallelism”

_Michael Wolfe, OpenACC Technical Chair (2015)_
Example: Vector Addition

```c
int* vecAdd(int* a, int* b, size_t len) {
    size_t len_bytes = len * sizeof(int);
    int* c = malloc(len_bytes);
    for(int i=0; i<len; i++) {
        c[i] = a[i] + b[i];
    }
    return c;
}
```

- Highly parallel problem
- Want to parallelize on GPU as much as possible
Example: Vector Addition

void vecAddKernel(int* a, int* b, int* c, size_t len) {
    int index = blockIdx.x*blockDim.x+threadIdx.x;
    int grid = blockDim.x*gridDim.x;
    while(index<len) {
        c[index] = a[index] + b[index];
        index += grid;
    }
}
Example: Vector Addition

```c
int* vecAdd(int* a, int* b, size_t len) {
    size_t len_bytes = len * sizeof(int);
    int* c = malloc(len_bytes);

    for(int i=0; i<len; i++) {
        c[i] = a[i] + b[i];
    }
    return c;
}
```

- Memory must be copied back and forth
- Compiler determines parallelism
Example: Vector Addition

```
int* vecAdd(int* a, int* b, size_t len) {
    size_t len_bytes = len * sizeof(int);
    int* c = malloc(len_bytes);

    for(int i=0; i<len; i++) {
        c[i] = a[i] + b[i];
    }
    return c;
}
```

- Memory must be copied back and forth
- Programmer specifies parallelism
GPU Architecture
The Hardware Perspective

- Each GPU has 10 Streaming Multiprocessors (SMs)
- Each SM has 64 processing cores (CUDA Cores)
- Each SM has own registers, L1 cache, shared memory
- All SMs share L2 cache and Global Memory
- SMs execute in Single Instruction Multiple Thread (SIMT)
- Context switches are free

"GP100 Pascal Whitepaper", NVidia, 2016
The Software Perspective

- Threads are grouped into warps of up to 32 threads.
- Warps are grouped into blocks.
- Blocks are grouped into a single grid for execution.
- Warps execute logically in lock-step.
Bringing it Together

- Each block is assigned to an SM
- SM immediately allocates registers and shared memory for the whole block
- Threads within a block can communicate through their Shared Memory
- Threads within a warp can communicate directly through special warp instructions
Data Transfer

- CPU ↔ GPU data transfer can occur in parallel with GPU computation (OpenMP nowait data clause)
- Pascal brings NVLink, with bonded unidirectional 20GB/s links
- GPUs can exchange data with each other independently over NVLink
- Supported CPUs (Power8, Power9) can also communicate over NVLink

"GP100 Pascal Whitepaper", NVidia, 2016
## GPU Memory Accesses

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Visibility</th>
<th>Cache Hit</th>
<th>Cache Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Memory Access</td>
<td>Thread-Local</td>
<td>30 Cycles</td>
<td>300 Cycles</td>
</tr>
<tr>
<td>Shared Memory Access</td>
<td>Block-Local</td>
<td>33 Cycles</td>
<td>N/A</td>
</tr>
<tr>
<td>Global Memory Access</td>
<td>Globally Visible</td>
<td>175 Cycles</td>
<td>300 Cycles</td>
</tr>
<tr>
<td>Constant Memory Access</td>
<td>Globally Visible</td>
<td>42 Cycles</td>
<td>215 Cycles</td>
</tr>
</tbody>
</table>

*Numbers for NVidia Tesla K20*
GPU Global Memory Coalescing

- Warp execution means 32 memory accesses issue in the same cycle
- Global memory system can handle only 2 accesses per cycle
- Warp memory accesses are coalesced whenever possible
- Memory coalescing is critical to performance
GPU Occupancy

- GPUs use context switches to hide instruction/memory latency
- Only so many resources available on each SM
- Occupancy limited by: Registers, Shared Memory, Thread Count, Block Count
- Shared memory doesn’t suffer coalescing, but is very limited
- Compiler must balance these resources to maintain performance

“Achieved Occupancy”, NVidia, docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm
Mapping OpenMP to GPUs
XL Compiler
GPU Architecture

Legend:
- Pre-existing XL Components
- XL GPU Components
- NVidia CUDA Components
- CPU Code
- GPU Code
- CPU/GPU Combined Code

XL Frontend

Annotated WCode

Toronto Portable Optimizer (TPO)

Outlined WCode

WCode Partitioner

Device WCode

WC2LLVM

Device LLVM

LibNVVM

Device PTX

CUDA Fatbinary

Final Executable

Host WCode

XL Backend

Host Objects

Linker

Host Objects

Device Objects
Code Generation

![OpenMP and CUDA logos]

- Target Regions → GPU Kernel Functions
- Teams → Thread-Blocks
- Threads → Threads
void f() {
    #pragma omp target teams
    #pragma omp distribute parallel for
    for(int i=0; i<1024; i++)
        g(i);
}

void g(int i) {
    #pragma omp parallel for
    for(int j=0; j<1024; j++)
        doWork(i,j);
}
Code Generation: Dynamic Parallelism

- NVidia GPUs support parent/child kernel relationships
- Extract each parallel region into a subkernel?

Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications

Jin Wang  
Georgia Institute of Technology  
Atlanta, Georgia, USA  
Email: jin.wang@gatech.edu

Sudhakar Yalamanchili  
Georgia Institute of Technology  
Atlanta, Georgia, USA  
Email: sudha@ece.gatech.edu

“CDP implementation can achieve 1.13x-2.73x potential speedup but the huge kernel launching overhead could negate the performance benefit”
Code Generation: State Machine

- Start all threads immediately
- *master* threads manipulate state in shared memory, release worker threads when parallel
- Synchronize before each state change
- Implemented in original XL OpenMP 4 Beta

“The paper characterizes occupancy as the limiting factor. There is a large difference in the amount of registers per thread required to execute the OpenMP versions and the CUDA C/C++ one”
Code Generation: Cooperative Multithreading

- Kernel launched with all threads
- Worker threads released for outer parallel regions
- At inner parallel regions, all threads in warp perform each thread’s work in sequence

<table>
<thead>
<tr>
<th>Teams Region</th>
<th>Master</th>
<th>Worker 0</th>
<th>Worker 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outer Parallel Region</td>
<td>do()</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td></td>
<td>sequential()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>work()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sync()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inner Parallel Region</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bdcst_state()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>do()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worker0()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>work()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rcv_state()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>do()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worker0()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>work()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bdcst_state()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>do()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>worker1()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>work()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sync()
Optimizations: Shared Memory Promotion

- Local accesses are sorted by usage
- GPU model determines available shared memory through cost model
- Locals promoted until available shared memory is full
- *Future Work:* Better ordering criteria for local accesses
Optimizations: Multi-Hierarchy Parallelism

- Runtime Initialization is required to emulate the OpenMP fork-join parallelism model
- When certain patterns are detected, more performant CUDA equivalents can be used
- For Example: 
  `omp distribute parallel for ⇔ CUDA multiblock loop`
Performance: Conway’s Game of Life

- Iterative 2D 9-point stencil
- Tested CUDA, OpenMP, CPU
- Runtime overhead is most of Optimized OpenMP overhead
- 64-bit pointer arithmetic accounts for the remainder

![Graph showing execution time vs. grid size for different methods: Naive CPU, CUDA, Naive OpenMP, Optimized OpenMP.](image)
Performance: Conway’s Game of Life

- Optimized OpenMP and CUDA have the same ratio of loads/stores.
- Naive OpenMP uses global memory for runtime state.
- Lower optimized OpenMP throughput is an artifact of unnecessary 64-bit operations.

![Global Memory Throughput Chart]

- The chart shows the throughput (in GB/s) for different memory access types (load and store) across CUDA, Naive OpenMP, and Optimized OpenMP.
Taylor Lloyd
MSc at University of Alberta

Want to discuss GPU computing further?
Come talk to me in the Expo, poster A19.