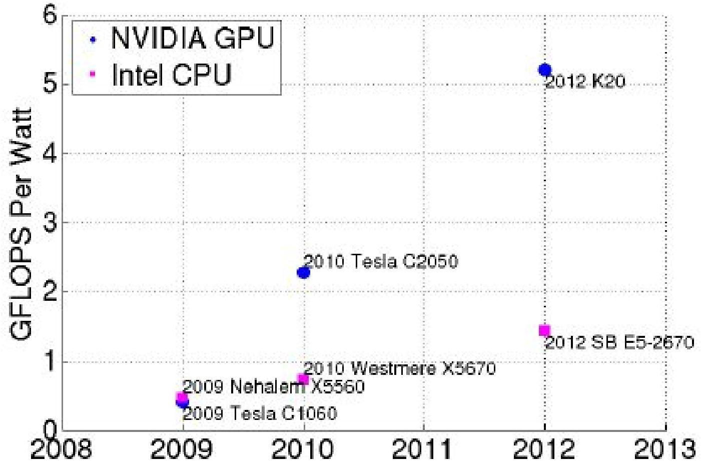
GPGPU Offloading with OpenMP 4.5 In the IBM XL Compiler

Taylor Lloyd *University of Alberta*
Jose Nelson Amaral *University of Alberta*
Ettore Tiotto *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



"A Step towards Energy Efficient Computing: Redesigning a Hydrodynamic Application on CPU-GPU", Dong et al,2014

Agenda

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

GPU Programming Models

GPU Parallel Programming Models

NVIDIA CUDA

(2006)

- NVidia GPUs

OpenCL

(2009)

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

OpenACC
Directives for Accelerators

(2012)

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

OpenMP

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

OpenCL

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

Language Philosophies

OpenACC
Directives for Accelerators

- 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)

OpenACC
Directives for Accelerators

OpenMP®

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;
}
```

- 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 = gridDim.x*blockDim.x;
  while(index<len) {
    c[index] = a[index] + b[index];
    index += grid;
  }
}
```

- Index derived from thread indices
- Memory must be copied back and forth
- Explicit parallelism

Example: Vector Addition

OpenACC
Directives for Accelerators

```
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

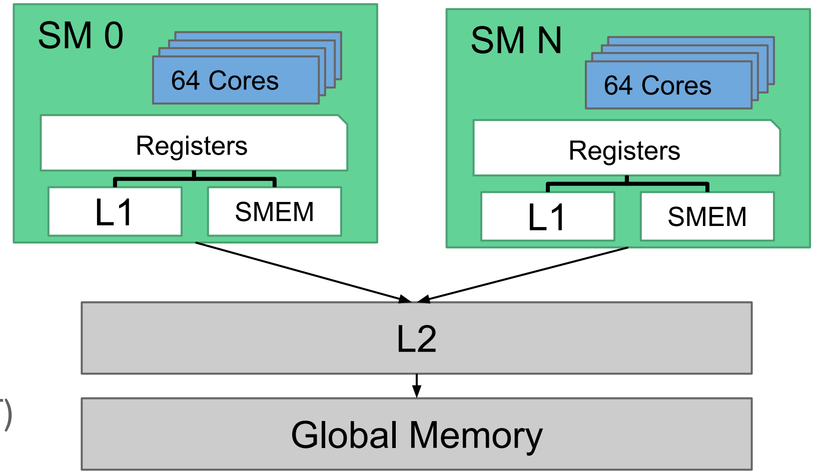
```
int* vecAdd(int* a, int* b, size_t len) {
  size_t len_bytes = len * sizeof(int);
  int* c=malloc(len_bytes);




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

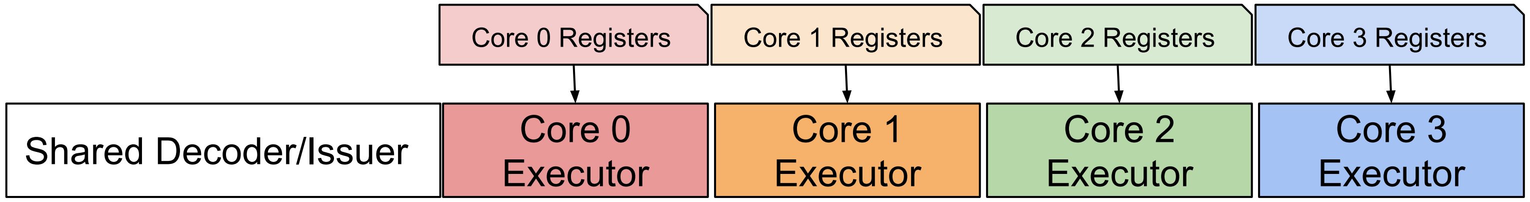
- Memory must be copied back and forth
- Programmer specifies parallelism

GPU Architecture

The Hardware Perspective

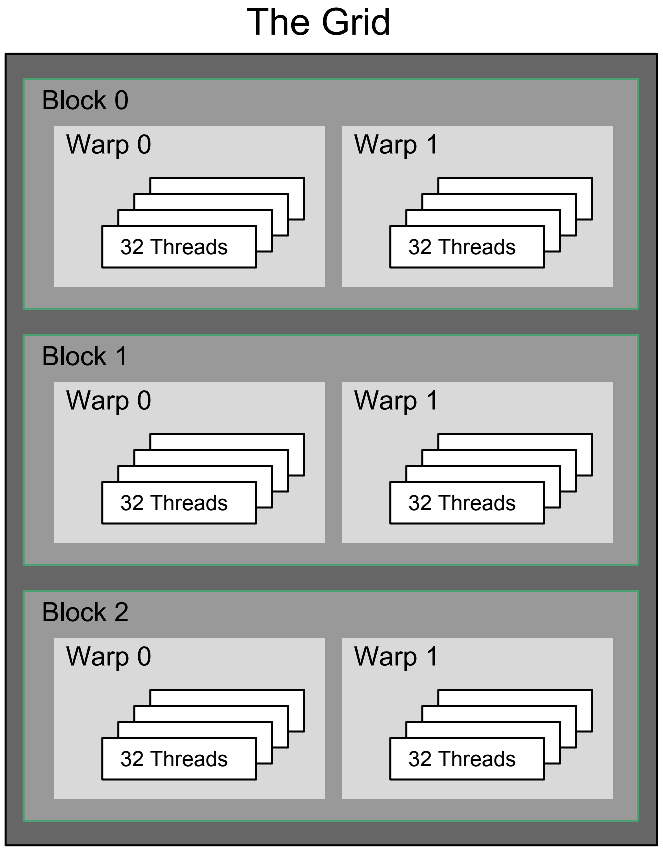
- Each GPU has 10 Streaming Multiprocessors (SMs)
- Each SM has 64 processing cores (CUDA Cores)
- Each SM has own registers, L1 cache, shared memory
- All SMs share L2 cache and Global Memory
- SMs execute in Single Instruction Multiple Thread (SIMT)
- Context switches are free





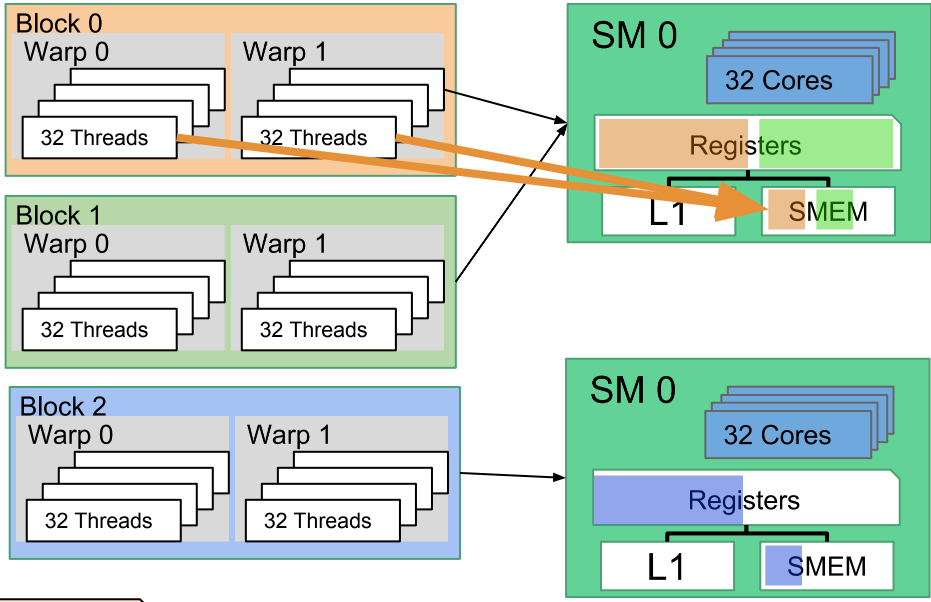
The Software Perspective

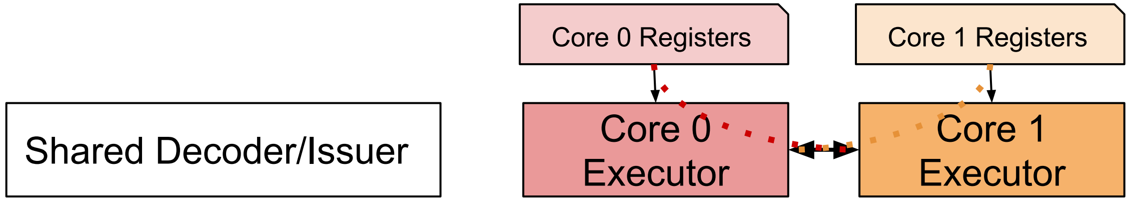
- Threads are grouped into warps of up to 32 threads.
- Warps are grouped into blocks
- Blocks are grouped into a single grid for execution
- Warps execute logically in lock-step



Bringing it Together

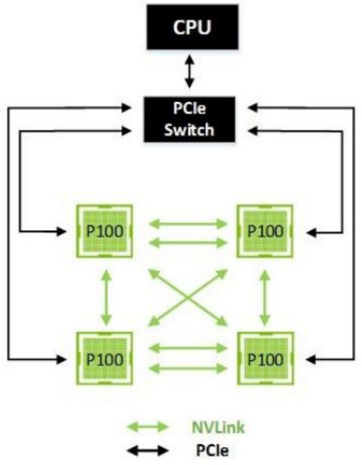
- Each block is assigned to an SM
- SM immediately allocates registers and shared memory for the whole block
- Threads within a block can communicate through their Shared Memory
- Threads within a warp can communicate directly through special warp instructions

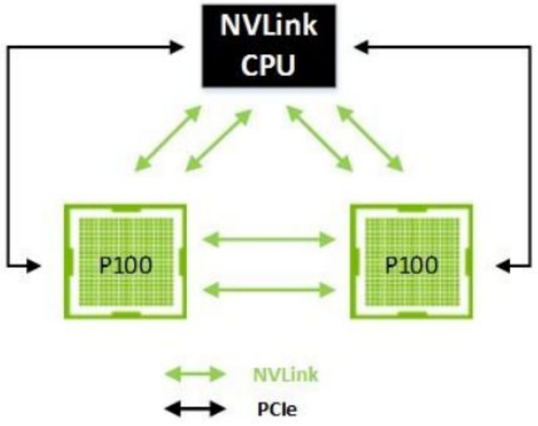




Data Transfer

- 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





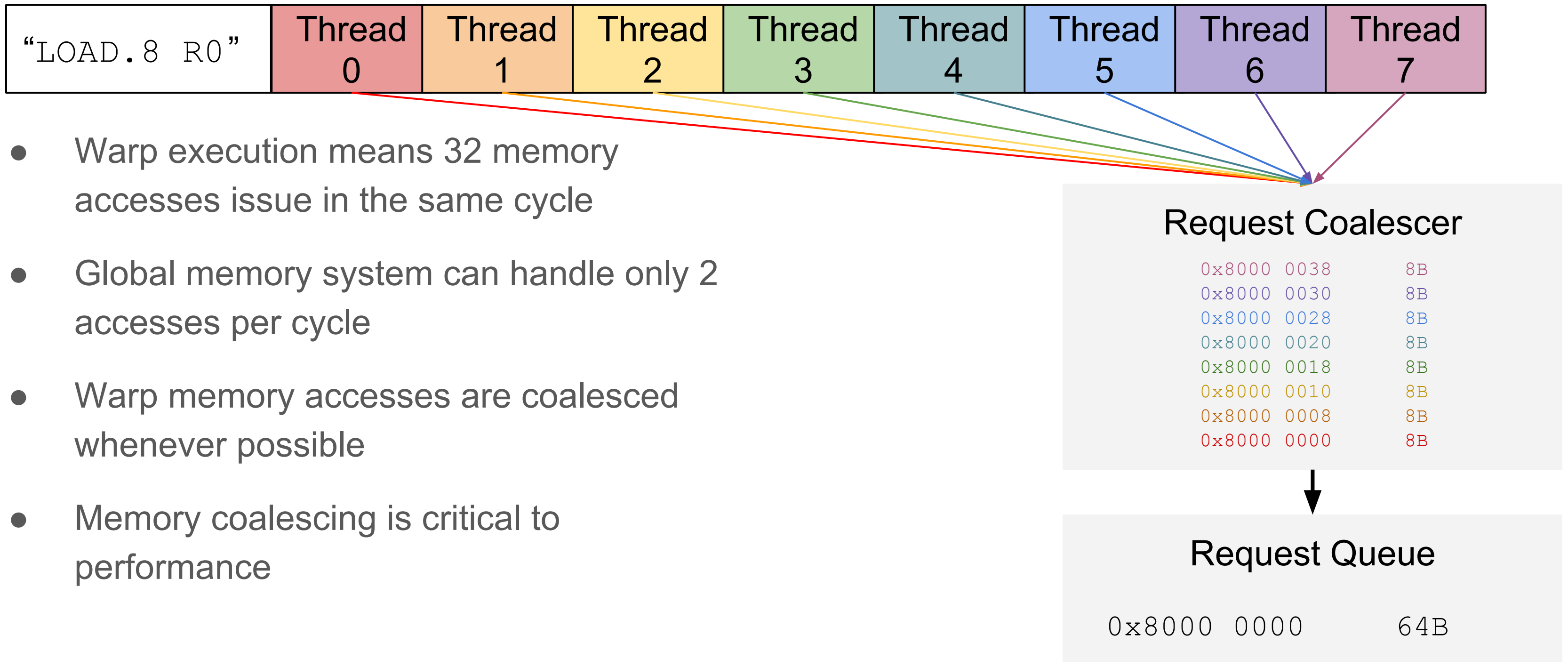
GPU Memory Accesses

Access Type	Visibility	Cache Hit	Cache Miss
Local Memory Access	Thread-Local	30 Cycles	300 Cycles
Shared Memory Access	Block-Local	33 Cycles	N/A
Global Memory Access	Globally Visible	175 Cycles	300 Cycles
Constant Memory Access	Globally Visible	42 Cycles	215 Cycles

*Numbers for NVidia Tesla K20

Wong *et al.*, Demystifying GPU Microarchitecture through Microbenchmarking, ISPASS 2010

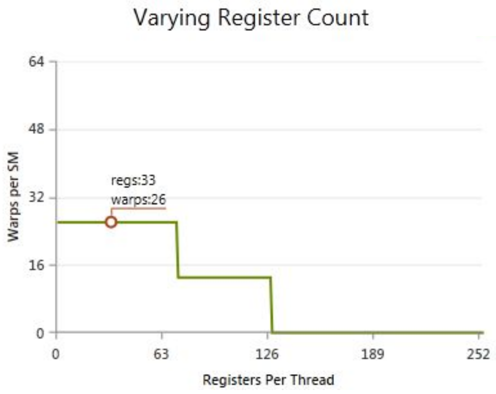
GPU Global Memory Coalescing

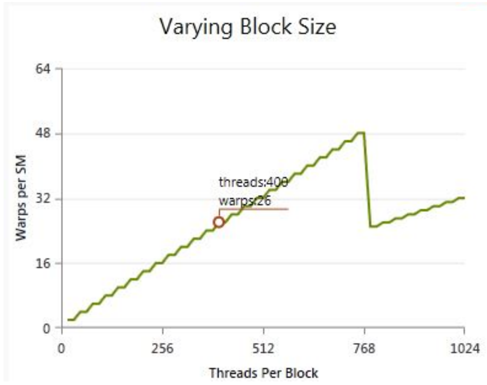


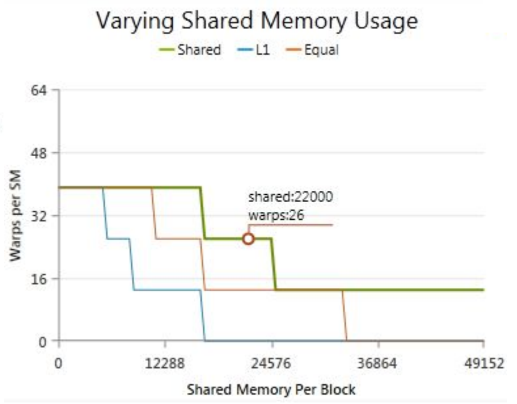
- Warp execution means 32 memory accesses issue in the same cycle
- Global memory system can handle only 2 accesses per cycle
- Warp memory accesses are coalesced whenever possible
- Memory coalescing is critical to performance

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

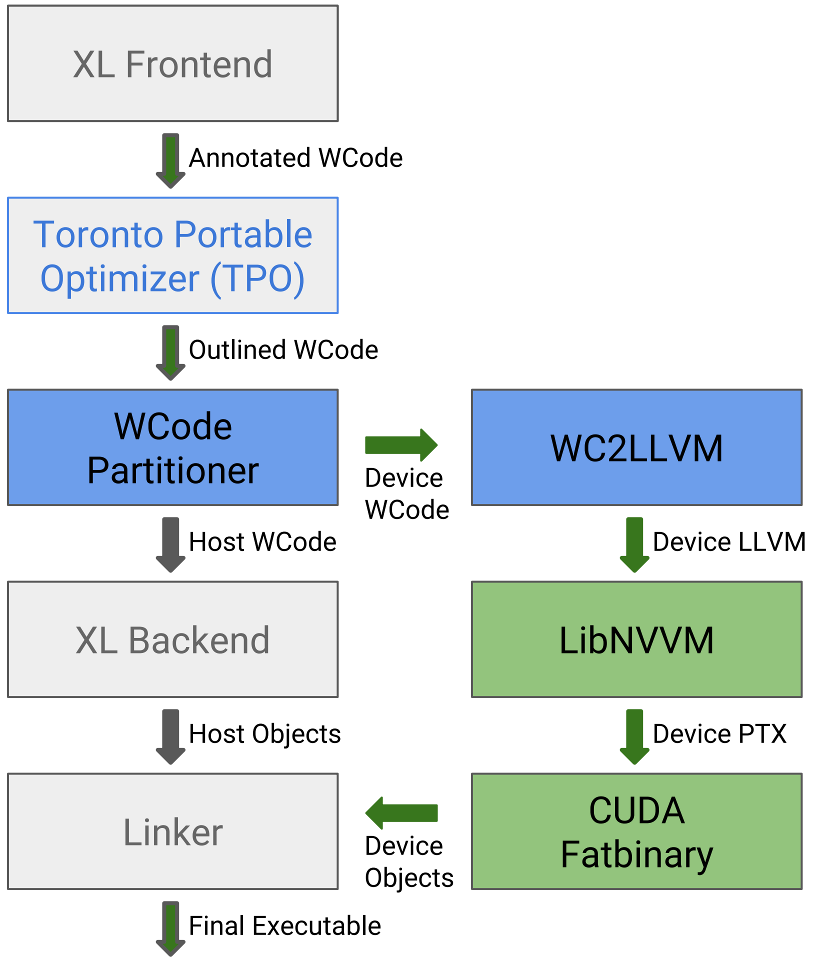






Mapping OpenMP to GPUs

XL Compiler GPU Architecture



Legend:
- ⬜ Pre-existing XL Components
- 🟦 XL GPU Components
- 🟩 NVidia CUDA Components
- ➡ CPU Code
- ➡ GPU Code
- ➡ CPU/GPU Combined Code

Code Generation

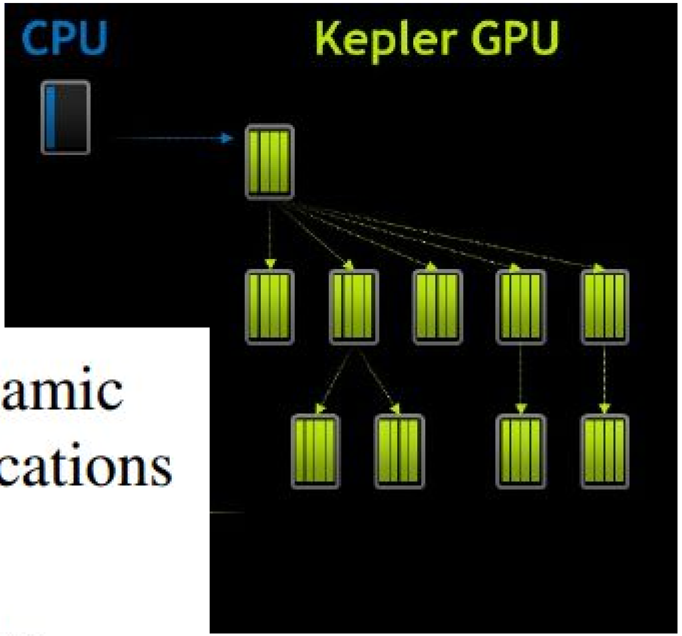
OpenMP		NVIDIA CUDA
Target Regions	→	GPU Kernel Functions
Teams	→	Thread-Blocks
Threads	→	Threads

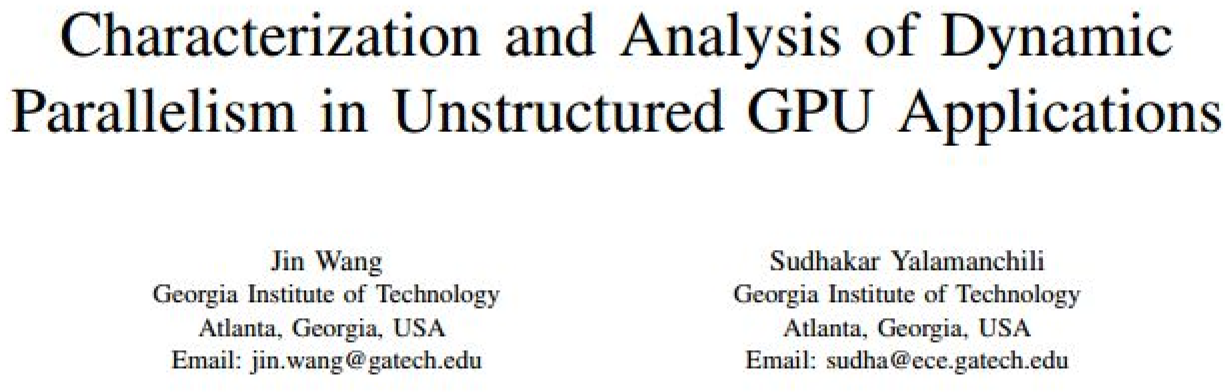
Code Generation: A Motivating Example

```
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

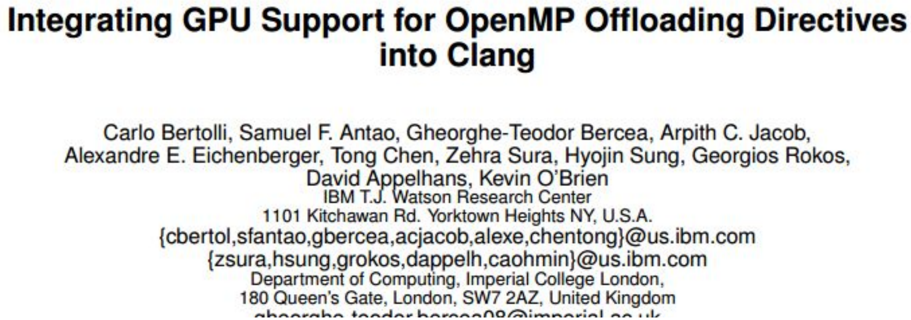
Jin Wang
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: jin.wang@gatech.edu

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



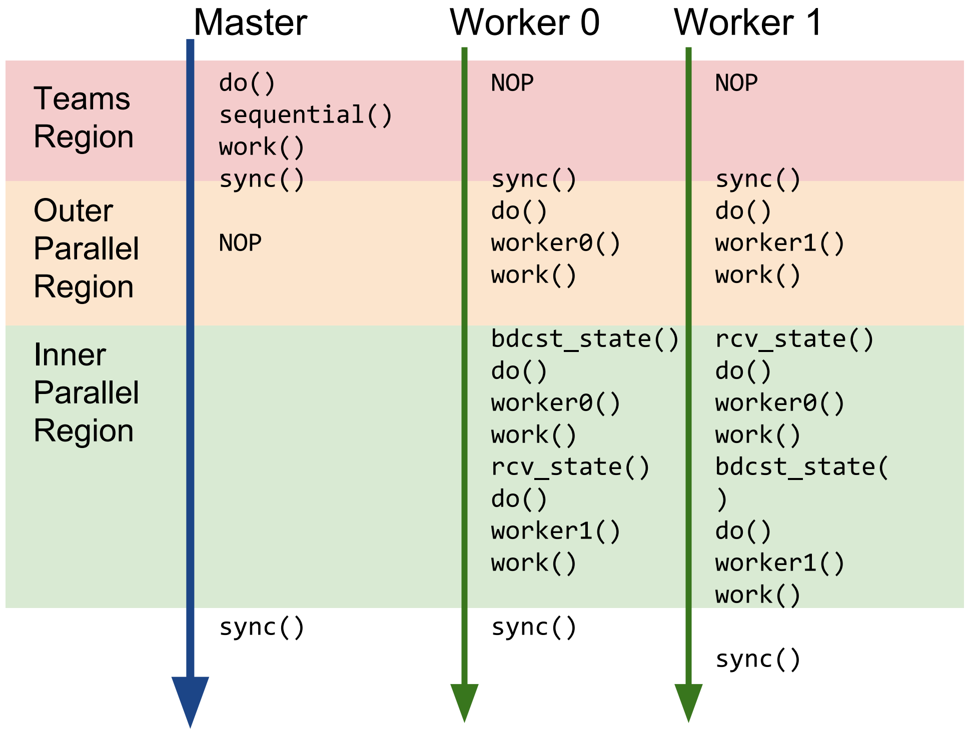
Integrating GPU Support for OpenMP Offloading Directives into Clang

Carlo Bertolli, Samuel F. Antao, Gheorghe-Teodor Bercea, Arpith C. Jacob,
Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos,
David Appelhans, Kevin O'Brien
IBM T.J. Watson Research Center
1101 Kitchawan Rd. Yorktown Heights NY, U.S.A.
{cbertol,sfantao,gbercea,acjacob,alexe,chentong}@us.ibm.com
{zsura,hsung,grokos,dappelh,caohmin}@us.ibm.com
Department of Computing, Imperial College London,
180 Queen's Gate, London, SW7 2AZ, United Kingdom
[illegible]

"The paper characterizes occupancy as the limiting factor. There is a large difference in t
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



	Master	Worker 0	Worker 1

Teams Region	do()
	sequential()
	work()
	sync()
		NOP
		sync()
			NOP
			sync()

Outer Parallel Region	NOP	do()
		worker0()
		work()
			do()
			worker1()
			work()

Inner Parallel Region		bdcst_state()
		do()
		worker0()
		work()
		rcv_state()
		do()
		worker1()
		work()
			rcv_state()
			do()
			worker0()
			work()
			bdcst_state(
			)
			do()
			worker1()
			work()

	sync()	sync()	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

$$maxThreads =$$
$$min(\frac{registerFileSize}{registersPerThread}, \frac{sharedMemorySize}{threadsPerBlock * sharedMemoryPerThread})$$

Locals	**Usage**
idx0	14
tmp62	8
arrbase2	3

.

.

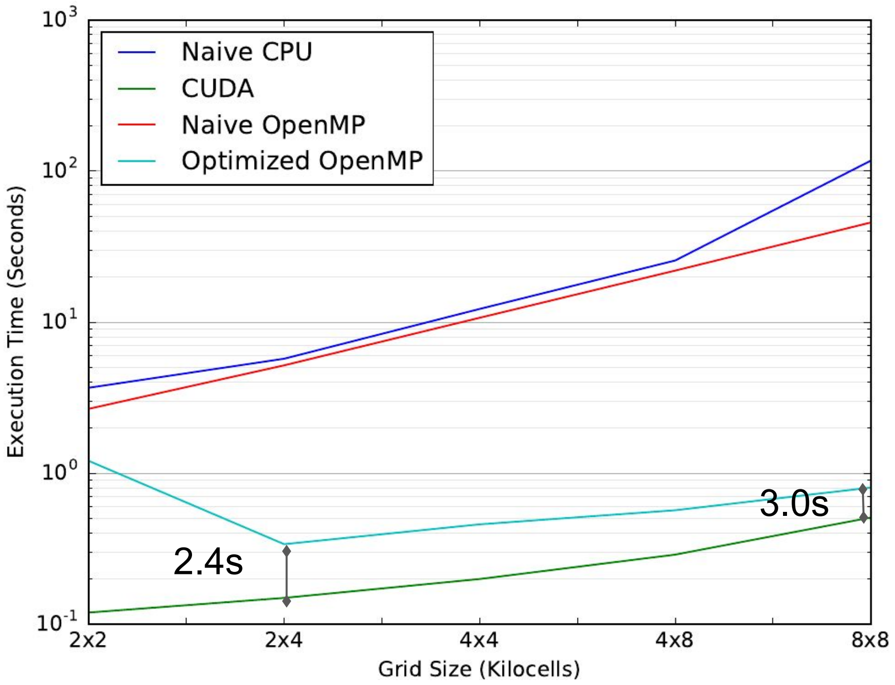
.

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

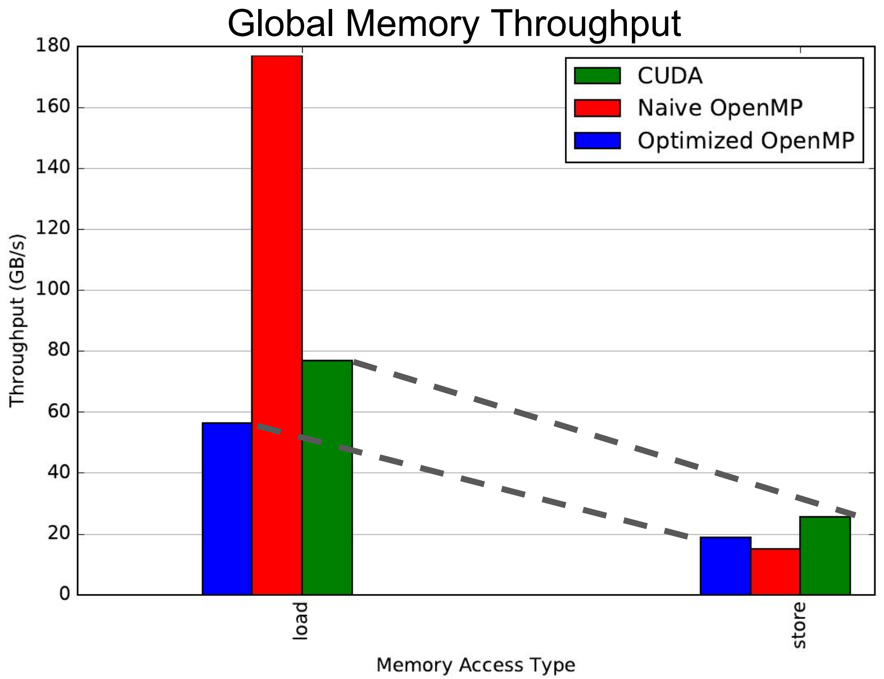
Performance: Conway's Game of Life

- 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



Performance: Conway's Game of Life

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



Taylor Lloyd
MSc at University of Alberta

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