High Level Programming for OpenPOWER® and CORAL systems

Carlo Bertolli
cbertol@us.ibm.com
Advanced Compiler Technology Team
IBM T. J. Watson Research Center
Outline

OpenPower Hardware

OpenMP Language and Implementation

Applications and Performance

Status of Software Toolchain
OPENPOWER HARDWARE
Available OpenPower Hardware

POWER® S822LC
- 2 POWER8 Processors
  - 190 - 247W Turismo module
- 2 x16 Gen 3 FHFL PCIe slots
  - Supports 2 NVidia K80 GPU’s (300W)
  - Supports PCIe form factor adapters
- 1 x8 Gen 3 HHHL PCIe, CAPI enabled
- 1 x16 Gen 3 HHHL PCIe, CAPI enabled
- 1 x8 Gen 3 PCIe
- 32 DDR3 IS DIMM’s
  - 4, 8, 16, 32GB DIMMs
  - 32 – 1024GB Memory Capacity
- 2 SATA SFF HDD / SSD
- 2 1300W Power Supplies
  - 200VAC Input
- BMC support structure
  - IPMI, USB, EN, VGA
- Air cooled

POWER® S822LC “Minsky”
- 2 POWER8 w/ NVLink Processors
  - 190 - 247W module
- 1, 2, 4 NVidia “Pascal®” GPU’s
  - 300W, SXM2 Form Factor, NVLink 1.0
- 2 x16 Gen 3 HHHL PCIe, CAPI enabled
- 1 x8 Gen3 HHHL PCIe, CAPI enabled
- 32 DDR4 IS DIMM’s
  - 4, 8, 16, 32GB DIMM’s
- 2 SATA SFF HDD / SSD
- Pluggable NVMe storage adapter
  - 1.6, 3.2TB Capacity
- 2 1300W power supplies
  - 200VAC Input
- BMC Support Structure
  - IPMI, USB, EN, VGA
- Air and water cooled options
Compiler View of OpenPower Architecture

Processing
- SMT 8
- Up to 20 cores
- 2 sockets
- CORES

Storage
- L1: 64 KB/core
- L2: 512 KB/core
- L3: 8 MB/core 96 MB
- L4: 128 MB/proc. chip
- DRAM: 2 TB
- off chip

Global Memory bandwidth: 720 GB/s

Pascal
- Up to 20 cores
- 2 sockets
- SMX: 8
- Up to 56 SMX
- SP CUDA cores, Streaming Multiprocessors

Compiler View of OpenPower Architecture

IBM Systems
Compiler View of OpenPower Architecture

Fat cores
SMT 8 (8 threads/core)
160 threads/processor
Complex branching prediction
Out-of-order pipeline

Highly parallel cores
2048 threads/SMX
$O(10^5)$ threads/device
Thread divergence is serialized
Limited synchronization (within CUDA block only)
Compiler View of OpenPower Architecture

“Fat” cores
SMT 8 (8 threads/core)
160 threads/processor
Complex branching prediction
Out-of-order pipeline

Large caches
Low memory latency
Optimize for cache locality

Highly parallel “thin” cores
2048 threads/SMX
O(10^5) threads/device
Thread divergence is serialized
Limited synchronization (within CUDA block only)

Small on-chip caches compared to #threads
L2 is off-chip, equivalent to L4 in P8
Optimize for memory latency hiding using lots of in-flight threads
PROGRAMMING OPTIONS FOR OPENPOWER
OpenPower - Programming Languages and Compilers

**CUDA**

**Key Features:**
- Gives direct access to the GPU instruction set
- Supports C, C++ and Fortran
- Generally achieves best leverage of GPUs for best application performance
- Compilers: nvcc, pgi CUDA fortran
- Host compilers: gcc, XL

**OpenACC**

**Key Features:**
- Designed to simplify Programming of heterogeneous CPU/GPU systems
- Directive based parallelization for accelerator device
- Compilers: PGI, Cray, gcc

**OpenMP**

**Key Features:**
- OpenMP 4.5 offloading and support for heterogeneous CPU/GPU
- Leverage existing OpenMP high level directives support
- Compilers: Open Source LLVM OpenMP Compiler, IBM XL
Programming Strategy for OpenPower

OpenPower
- Ideal combination of Power “fat” cores, and NVIDIA “thin” GPU cores
- Different components of HPC workloads can be executed optimally
- Power cores low memory latency, GPU high memory bandwidth

### Programming Model Wish List

<table>
<thead>
<tr>
<th>High Level Parallel Abstractions</th>
<th>Parallel loops, simd, tasks, tasks+loops, BFS-like, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Architecture/accelerator independent</td>
<td></td>
</tr>
<tr>
<td>• High application pattern coverage (dense, sparse, graphs)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Portability</th>
<th>Early porting shows performance within CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Single version of kernels runs on Power and GPU</td>
<td></td>
</tr>
<tr>
<td>• With the best performance possible</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Easy incremental development</th>
<th>Code Annotations (#parallel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No need to rewrite entire application in new language</td>
<td></td>
</tr>
<tr>
<td>• Interoperability with assembly kernels and libraries (CUDA, CUDNN)</td>
<td></td>
</tr>
<tr>
<td>• Early Availability</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Continuity</th>
<th>Supported by: IBM, AMD, Intel, Cray, PGI, Pathscale, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Industrial standard</td>
<td></td>
</tr>
<tr>
<td>• Supported everywhere</td>
<td></td>
</tr>
<tr>
<td>• Survives project/architecture/fashion/etc.</td>
<td></td>
</tr>
</tbody>
</table>
OPENMP PROGRAMMING MODEL
Flexible Parallelism

Parallel Loops

```c
#pragma omp parallel for
double A[M][N];
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

Parallel Loops with SIMD

```c
#pragma omp parallel for
#pragma omp simd
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

- **#parallel** recruits threads
- **#for** schedules M iterations to parallel threads
- At the end of #parallel there is a barrier
- Significant performance optimizations for successive small parallel loops
Digging into Optimizations: Thread Affinity

```c
#pragma omp parallel for proc_bind(spread)
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

**Close affinity:** pack threads for cache locality

**Spread affinity:** spread threads to maximize bandwidth

- **Socket 0:**
  - Core 0: Red
  - Core 1: Blue
  - Core 2: White
  - Core 3: White

- **Socket 1:**
  - Core 4: White
  - Core 5: White
  - Core 6: White
  - Core 7: White

- **Socket 0:**
  - Core 0: Red
  - Core 1: Blue
  - Core 2: White
  - Core 3: White

- **Socket 1:**
  - Core 4: White
  - Core 5: White
  - Core 6: White
  - Core 7: White
Flexible Parallelism on Devices

Parallel Loops on GPU

```c
#pragma omp target teams distribute parallel for
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

On the GPU

- Target offloads region to GPU
- Each team corresponds to a CUDA block
- OpenMP threads are CUDA threads
- `distribute` schedules blocks of iterations to teams
Flexible Parallelism on Devices

Parallel Loops with SIMD on GPU

```c
#pragma omp target teams distribute parallel for
for (i = 0; i < M; i++)
#pragma omp simd
for (j = 0; j < N; j++)
```

- **simd** inside **parallel** is widely used on host
  - Leverage vector units per thread
- The GPU has no vector units
  - Map **simd** lanes into CUDA threads
  - **Perfect coalescing** at first two parallelism levels

- **CUDA grid**
  - CUDA block: 1 per team
  - CUDA thread: 1 per OpenMP thread & simd lane
- **GPU**
  - #target teams
Data mapping

double * a = malloc (sizeof (double) * n);

// ...

#pragma omp target teams distribute parallel for map(a[:n])
for (i = 0; i < M; i++)
a[i] += beta*accu[i%10];

‘a’ is a pointer to dynamically allocated memory area

When compiler can determine size
  • Scalars, static arrays, etc
  • Automatic mapping

‘a’ is now visible on device
  • Content is updated to host memory value

Programmers control data updates
  • Maps: to, tofrom, from, alloc
  • Target update

Host Memory

<table>
<thead>
<tr>
<th>0</th>
<th>n-1</th>
</tr>
</thead>
</table>

Device Memory

<table>
<thead>
<tr>
<th>0</th>
<th>n-1</th>
</tr>
</thead>
</table>
Abstract Memory Model

‘A’ is a shared variable

- Visible by a host thread when using it
- ‘map’ to device data environment
  - ‘map’ does not necessarily mean copy
  - e.g. unified memory does not need explicit copying
- GPU threads can access ‘A’ from global memory
  - Copies may be present in local caches
Abstract Memory Model

‘A’ is a shared variable
• Visible by a host thread when using it
• ‘map’ to device memory
• GPU threads can access ‘A’ from global memory
  ▪ Copies may be present in local caches
• A GPU thread modifies its local copy and this is flushed to the GPU global memory
Abstract Memory Model

‘A’ is a shared variable

- Visible by a host thread when using it
- ‘map’ to device memory
- GPU threads can access ‘A’ from global memory
  - Copies may be present in local caches
- A GPU thread modifies its local copy and this is flushed to the GPU global memory
- A host thread that spawned the GPU computation requests an update from device to host memory
- All host thread copies are now updated
Flexible Parallelism

Tasking Constructs

```c
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: a)
    TraverseForward(A);
    #pragma omp task depend(in: a)
    TraverseReverse(B);
}
```

- Tasks are well suited for parallelism that is dynamically uncovered: e.g. searches, graph processing
- Tasks are load balanced between threads in the parallel region
- A task is fired once all its dependent tasks have completed

Using Tasking Support for Loops

```c
#pragma omp taskloop simd grainsize(500)
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```
Digging into Optimizations: Target Task Dependencies

Task dependencies on device (in red):
- Resolved on device
- Based on CUDA streams

OpenMP runtime book-keeps host-device dependencies
Digging into Optimizations: Overlap Computation and Data Transfers

The OpenMP runtime may **overlap** computation and data transfer:

1. Transfer data to GPU memory and
2. Launch a kernel using that data
   - Dependency is satisfied on GPU by driver
3. Recruit threads for parallel work
   - The threads work independently of runtime and other devices
4. Device notifies runtime that kernel has finished
5. Runtime issues data transfer to host
6. Host threads return to runtime when finished
Architecture-based Flexible Parallelism

Real-world scientific applications feature **multi-physics**, often executed in parallel.

**Graph-like workload**
- Highly irregular
- e.g. particle simulation
- Benefits from large caches
- Master thread creates several tasks
  - Dynamically scheduled to free worker threads by runtime

**Data Parallel workload**
- Relatively Regular
- e.g. dense linear algebra
- Benefits from large numbers of FMA units

**Real-world scientific applications** feature multi-physics, often executed in parallel.
Interoperability with CUDA

- OpenMP 4.5 and CUDA can be mixed in the same program
- Runtime functions for memory:
  - allocate, free, check, copy
  - associate CUDA pointer to host variable
- Start from existing CUDA kernels
- Continue developing in OpenMP 4.5
- Mix & Match
CLANG/LLVM OPENPOWER COMPILER
LLVM Compiler Schematic

- **CLANG**
  - Front-end to parse source code and generate LLVM IR code
  - Modified to generate code for OpenMP device constructs
  - Produces two copies of code for target regions
  - Inserts calls to standardized OMP runtime interface functions
  - Compiler driver modified to process code copies through different backends

- **Fortran front-end**
  - Proprietary, closed source based on xlf
  - Interim solution
    - PGI announce of open source Fortran

- **NVPTX backend**
  - Produces ptx code which is then processed through ptxas to generate CUDA binary

- **POWER backend**
  - Preliminary performance analysis

Collaborating with wider community and industry partners (LLVM open-source, OMP standards)
OpenMP Contributions by Community

In this talk:
- OpenMP Code generation
- Driver extensions
- libomptarget

Full implementation of offloading

Language extension: parse+sema+codegen

Driver

Some device functions required

Code generation for accelerators

 projects

LLVM

Clang

OpenMP

libomptarget

projects

tools
History of

CODE GENERATION SCHEME
**What’s The Problem**

OpenMP allows fast offloading of application regions to accelerators

- Highly general programming model
- Any OpenMP construct can be used in a target

```c
#pragma omp target teams
{
    int k = omp_get_team_num()*2;
    #pragma omp distribute
    for (i0 = 0 ; i0 < N ; i0 += blockSize) {
        #pragma omp parallel for
        for (i = i0 ; i < min (N, i0+blockSize) ; i++)
    }
    Z[omp_get_team_num()] = k;
}
```

A target region may include arbitrary sequential, team-master only, and parallel regions
What’s The Problem

OpenMP allows fast offloading of application regions to accelerators

- Highly general programming model
- Practically, any OpenMP construct can be used in a offloading target region

```c
#pragma omp target teams
{
    int k = omp_get_team_num()*2;
    #pragma omp distribute
    for (i0 = 0 ; i0 < N ; i0 += blockSize) {
        #pragma omp parallel for
        for (i = i0 ; i < min (N, i0+blockSize) ; i++)
    }
    Z[omp_get_team_num()] = k;
}
```
Why is this a problem on a GPU

A GPU is optimized for executing highly parallel workloads.

Dynamic thread launch has a non-negligible overhead.

Coordinating threads may introduce lots of synchronizations.
Assumptions

1 OpenMP team is 1 CUDA block

- There is no language-defined synchronization between OpenMP teams and threads in different CUDA blocks
- Synchronization is defined for threads inside the same team and in same CUDA block

Research Strategy

- For all versions we can think of to solve this problem, implement benchmarks using each version
- Compare obtained performances
- Analyze how easy/hard it is to implement in Clang
Solution 1: stay on host!

Map all sequential and team-master regions to the CPU

- Data movement if distributed memory system
- Overhead of launching multiple kernels
- The application programmer wants it on the device!
- Nested sequential regions (single)?
- What about data being mapped to device data environments and accessed in sequential regions?
Solution 2: Dynamic Parallelism

Dynamically start threads when hit parallel region

- Dynamic parallelism overhead
- Amortized for large kernels, bad for small kernels
- If overhead reduces, still a trade-off
Solution 3: Thread Coordination Schemes

An alternative solution is to launch all required blocks and threads at the same time when we hit a target teams region

- Start all teams and threads when hit target
- In team-master regions only the masters are executing
- In parallel regions, all threads execute
  - What about num_threads clause?
- How do we program this behavior?
Thread Coordination: If-master

```c
__global__ void kernel(..) {
  // only team master
  if (threadIdx.x==MASTER)
    a[omp get team num()] = b*2;
  // all threads in block wait for master
  __syncthreads();

  <codegen for parallel for>
  __syncthreads(); //required by #for

  // only team master
  if (threadIdx.x==MASTER)
    a[omp get team num()]*= 2;
  // all threads in block wait for master
  __syncthreads();
}
```
Performance of Dynamic Parallelism vs If-Master

Are we aiming at the right direction?
Vector-add with #teams distribute and #parallel for

Using dynamic parallelism for target, teams, and parallel

Only one kernel nest level, use if-master for coordination
Limitations of If-master

Potentially, a synchronization for every C/C++ statement!

• It is much worse than that:
  – In Clang, for every C/C++ statement check if we are in a sequential or parallel region and generate code accordingly
  – Violates compiler’s modularity design: what happens in OpenMP stays in OpenMP!
First Step Towards a Solution: Inspect/Execute

We can characterize an inspection phase

- Executed only by the master
- Take decision on next parallel region to execute

```c
int myTeam = omp_get_num_team();
if (threadIdx.x == MASTER) {
    if (++a[myTeam] > 0 && ++b[myTeam] > 0)
        label = par_reg1;
    else
        label = par_reg2;
}
__syncthreads ();
switch (label) {
    case par_reg1: {
        < codegen parallel region 1>
        break;
    }
    case par_reg2: {
        < codegen parallel region 2>
        break;
    }
}
```

**Inspection:**
- take decision

**Execution:**
- parallel region

- It does not work well for nested parallel and sequential regions
- We may end up with many __syncthreads (bad for CUDA if not well controlled)
- Easy to integrate in Clang
- Start generating code in inspection block
- Each parallel region is a new case in the execution switch
- Set up labels to point to right parallel region in inspection
Control Loop with Inspector/Executor – Basic Idea

Use a switch with one case for every parallel and sequential region
Encapsulate it in a while-loop that does a team synchronization as first action

```c
while (!finished) {
    __syncthreads();
    int nextLabel = (threadIdx.x == MASTER)?
        masterLabel : nonMasterLabel;
    switch (nextLabel) {
    case idle:
        break;
    case seq_region: {
        <codegen seq region>
        masterLabel = par_reg1;
        nonMasterLabel = par_reg1;
        break;
    }
    case par_region: {
        <codegen par region>
        if (tId == MASTER) {
            masterLabel = finished;
            nonMasterLabel = finished;
        }
        break;
    }
}
```

Team master decides what to do next for itself and all other threads
Control Loop with Inspector/Executor

What happens at run-time

```c
if (++a[omp_get_team_num()] && ++b[omp_get_team_num()]) {
    con_gbl = 0;
}

#pragma parallel for
for (k = ...) { // par reg 1.1
    cond[k] = ... ;
for (k = 0 ; k < N ; k++)
    cond_gbl = cond_gbl && cond[k];
if (cond_gbl)
    #pragma parallel for
    for (l = ...) { // par reg 1.2
}
```

![Diagram](image-url)
CLANG DRIVER AND LIBOMPTARGET
OpenMP Offloading

**Main application file**

```c
#pragma omp declare target
t void linSolver(double * A, double * b);
#pragma omp end declare target

int main() {
    // set up A and b and map

    #pragma omp target teams
    {
        linSolver(A, b);
    }
}
```

**Library implementation**

```c
#pragma omp declare target
t void linSolver(double * A, double * b)
{
    // .. linear solver implementation
}
#pragma omp end declare target
```

- Same target body can run on host or device
- Different compilation units
Job creation is performed by visiting graph from root to leaves.

Job

```
clang --cc1 ...
```

- Collect multiple actions that can be performed by a tool
- Define input and output files
- Define parameters

Collect multiple actions that can be performed by a tool
- Define input and output files
- Define parameters
Driver Extensions for CUDA

Support for CUDA extends existing support with

- Host and device actions
- Selection of right toolchain per function based on host, device, global markups
- Linker tool (fatbinary) is used to combine objects for different compute capability
  - Not nvlink
- Host action is used to embed device code into host-produced binary

What is missing for OpenMP

- No relocation of symbols across host/device
- No dependencies between host and device toolchains possible
Generic Offloading Action

Replaces CUDA’s host and device actions
• The offloading kind (e.g. OpenMP, CUDA)
• The toolchain used by the dependencies (e.g. nvptx, amd)
• Device architecture (e.g. sm_60)

Host to device dependency
• The host builds a list of target regions to be compiled for device

Device to host dependency
• Bundling of object code in single binary
Offloading for PPC64 and NVPTX64

foo.cpp

Clang

LLVM IR (*.bc)
PPC64 target

.ppc64 asm

Id

LLVM

ptxas

.sass

nvlink

Libomptarget-nvptx.bc

Libomptarget-nvptx

Fat Binary

PPC64

NVPTX SM_20

NVPTX SM_60
Offloading for PPC64 and NVPTX64

Host bitcode file contains metadata that indicates what regions and functions should be compiled for device.

foo.cpp

Clang

LLVM IR (*.bc)
PPC64 target

LLVM IR (*.bc)
NVPTX64 target

Clang

Libomptarget-nvptx.bc

Libomptarget-nvptx

.ppc64 asm

.ptx

.sass

.ptxas

nvlink

Fat Binary

PPC64

NVPTX SM_20

NVPTX SM_60
Offloading for PPC64 and NVPTX64

** foo.cpp

** Clang

** LLVM

** ppc64 asm

** Id

** LLVM IR (*.bc) PPC64 target

** LLVM IR (*.bc) NVPTX64 target

** ptxas

** sass

** nvlink

** Linking of libomptarget-nvptx

** Libomptarget-nvptx

** Fat Binary

** PPC64

** NVPTX SM_20

** NVPTX SM_60
Offloading for PPC64 and NVPTX64

foo.cpp

Clang

LLVM IR (*.bc)
PPC64 target

LLVM

Clang

Libomptarget-nvptx.bc

ptxas

.sass

ppc64 asm

Id

Fat Binary

PPC64

NVPTX SM_20

NVPTX SM_60

Inlining of libomptarget-nvptx
(precompiled with clang-cuda)
Offloading for PPC64 and NVPTX64

foo.cpp

Clang

LLVM IR (*.bc)
PPC64 target

LLVM

Id

.ppc64 asm

.sass .ppc64

ptxas

واء

nvlink

Fat Binary

Linking of various sections done by linker script (generated by Clang)

PPC64

NVPTX SM_20

NVPTX SM_60
Fat Binaries and Libomptarget

A single binary for multiple devices
• FPGA, DSP accelerator, GPUs, etc.

No need to recompile source if device section is already present
OpenMP Runtimes

Clang/LLVM Binary

- PPC64
  - Non offloading constructs
  - Target constructs

- NVPTX SM_60

Host OpenMP RT
libomp

Libomptarget
- device agnostic
- device specific (plugin)

Device interface
(CUDA driver API)

Device OpenMP RT
libomptarget-nvptx

All OpenMP runtime calls generated by compiler on non-target regions

OpenMP Runtime calls on GPU
Libomptarget Offloading Support

- Implement calls from generated code for offloading
  - map data, start target region, etc.
- Not target a device type
  - only device IDs

Libomptarget

- device agnostic
  - Implement calls from libomptarget above it
    - map data, start target region, etc.
- device specific (plugin)
  - For a specific device type
    - Implementation is done by using device specific interface

Device interface
(CUDA driver API)
Libomptarget Offloading Support – Multiple Device Types

- Each device type requires to implement a plugin

Libomptarget (device agnostic)

- Libomptarget (cuda plugin)
- Libomptarget (elf plugin)
- Libomptarget (fpga plugin)

Device interface (CUDA driver API)

Elf-based OS

FPGA Driver
Libomptarget Offloading Support – Multiple Device Types

- Each device type requires to implement a plugin
- This design does not prevent optimized implementation of libomptarget where we have a single library

- Optimized libomp for GPU device
- Full inlining
Libomptarget Binary Registration

Required to allow host to offload to device
- Device code may be produced by a device-specific toolchain
- Can be loaded dynamically: based on device availability

Some devices require special binary registration support
- NVIDIA GPUs

The compiler generates a CUBIN object section with
- All code (kernels and device functions)
- All static data

Libomptarget asks CUDA driver to load CUBIN
- cuModuleLoadDataEx
- Requires start memory address of CUBIN
- Libomptarget retrieves the symbol address from a host/device table
- Host/device table is populated by Clang, linker, and loader
Patch Status

Full driver implementation is available in trunk
  • Thanks to everybody that made this happen!

Libomptarget is provided as three patches
  • Host agnostic, plugins for ELF and CUDA, device library for NVPTX
  • New patch versions available after third round of reviews

Code generation patches
  • In preparation
    • Working with Clang community to prepare code

Optimizations
  • Several optimizations are already standard implementation in our internal repo
  • Will have to come after full-fledged implementation
What is still missing – a lot!

Code generation for GPUs not yet available
• We are preparing patches
• Challenging to extract patches from full-fledged implementation of OpenMP 4.5. on GPUs
• Several design choices need to be discussed with community

Code generation for GPU is significantly different from CPU
• Due to SIMT GPU model
• No thread “spawning” or “recruiting”
• Cannot hide everything in runtime

Data sharing
• A single thread (team master) may need to share data with all other threads within its team
• Compiler needs to identify variables to be shared
• Requires multiple passes: current scheme in github not upstreaming friendly
How to play with full-fledged OpenMP 4.5. Implementation for GPU

Full-fledged implementation is available on Github

- https://github.com/clang-ykt
- Give it a try and report bugs

This will be discontinued as soon as upstreamed

- Only meant as a collaboration hub before upstreamed
- Only expect bug fixes on this version
- New features will be done directly in trunk
Porting to OpenMP and CUDA started at the same time

- OpenMP version with collapse
  - Complex code synthesis
  - Hard to reproduce in CUDA
- CUDA version uses multiple block dimensions
- Eventually CUDA catches up, after some debugging

OpenPower P8 and K40m NVIDIA GPU
Performance of Mantevo

Clang % Performance of CUDA

- Vec Add
- Vec Add Sqrt
- Vec Add and Mul
- Reverse Indirection
- Column Indirection
- 2pt Stencil
- 2pt Stencil 10 Gap
- 5pt Stencil (2d)
- 5pt Wavefront (2d)
- 9pt Stencil (2d)
- 7pt Stencil (3d)
- 27pt Stencil (3d)
- TealLeaf Cheby iter
- TealLeaf Collapsed
- CloverLeaf Energy Flux
- Snap Sweep
- Dense Mat Vec
- Compute Bound
- Compute Bound (1d)

Clang N/128 Teams, 128 Threads, best choice of NC/FMA

OpenPower P8 and K40m NVIDIA GPU
Software Status

Open source: compiler download and installation instructions at:
• https://www.ibm.com/developerworks/community/groups/community/openmp

Regular refreshes as improvements are made
• Internal repository is mirrored on github
  • https://github.com/clang-ykt/
  • Binaries are also available on request

Full support for OpenMP 4.5 (GPU offloading included)
• Kepler and Pascal architectures supported
• Host library from IBM (shared with XL compiler)
• Open source host library based on Intel contributed KMPC lib
• Open source GPU library using KMPC APIs, developed and contributed by IBM Research

Several optimizations available for GPU

Patches to Clang/LLVM trunk repositories in progress
Thank you!

ibm.com/systems/hpc
BACKUP SLIDES
OpenMP and OpenACC

Both based on the same principles
- Quick porting of existing applications
- High-level abstractions, no unnecessary detail

Differences
- OpenMP does not target a specific accelerator or processor: design is architecture independent
  - Processors, GPUs, custom accelerators, FPGAs, etc.
- OpenACC is designed for GPU acceleration
  - Support for Power not currently available
  - Requires OpenMP+OpenACC to target whole system
- OpenMP contains flexible parallelism constructs
  - Aside of common parallel loops and simd, also tasking, constructs for graph analysis, etc.
- OpenACC is limited to parallel loops and simd
- OpenMP is an industrial standard. Actively supported by IBM, Intel, AMD, TI, Pathscale, Cray, etc.
- OpenAcc is supported by Cray, PGI, and gcc
KRIPKE – Optimal Performance on CPU and GPU

Original CPU version

```c
#pragma omp parallel {
  // Loop over the hyperplanes (slices)
  for (int slice = ...) {
    #pragma omp for
    for (int element = ...) {
      for (int directions = ...) {
        // calculate data depending on d
        for (int group = ...) {
          // calculate new zonal flux
          // apply diamond-difference relationships
        }
      }
    }
  }
}
```

Basic performance portable version

- No loop-interchange necessary
- Multiple GPU with multiple host threads on different slices

```c
#pragma omp parallel {
  // Loop over the hyperplanes (slices).
  for (int slice = ...) {
    #pragma omp target teams distribute parallel for collapse(3)
    for (int element = ...) {
      for (int directions = ...) {
        // calculate data depending on d and new zonal flux
        // Apply diamond-difference relationships
      }
    }
  }
  //end element (distribute)
} //end of "for (slice"
```