Supporting Multiple Accelerators in High-Level Programming Models

Yonghong Yan†, Pei-Hung Lin‡, Chunhua Liao‡, Bronis R. de Supinski‡, and Daniel J. Quinlan‡

†Department of Computer Science and Engineering, Oakland University, Rochester MI
‡Center of Applied Scientific Computing, Lawrence Livermore National Laboratory

The 6th International Workshop on Programming Models and Applications for Multicores and Manycores, February 2015

Acknowledgement: NSF under project CNS-1205708 and SHF-1422961, DoE LLNL under contact DE-AC52-07NA27344 (LLNL-PRES-666922), and OpenMP Language Committee/Subcommittees
Contents

• Motivation
• Language Extensions
• Compiler and Runtime Implementation
• Evaluation Results
• Related Work
• Conclusions
Motivations

• Accelerator architectures become popular
  – GPUs and Xeon Phi
• Multiple accelerators are common
  – 2, 4, or 8
  – Mostly are used one by one

Programming
1. CUDA, OpenCL, etc
   – Low-level
2. Library, e.g. cublas, cufft, etc
3. OpenMP, OpenACC
   – High level models

Cirrascale GB5400 8-GPU blade server

Focus on using one accelerator a time
AXPY Example with OpenMP: single device

- \( y = \alpha \cdot x + y \)
- \( x \) and \( y \) are vectors of size \( N \)
- \( \alpha \) is scalar

```c
void axpy_ompacc(REAL* x, REAL* y, int n, REAL a) {
    #pragma omp target device (0) map(tofrom: y[0:n]) \n    map(to: x[0:n], a, n)
    #pragma omp parallel for shared(x, y, n, a)
    for (int i = 0; i < n; ++i)
        y[i] += a * x[i];
}
```

- **target** directive: annotate an offloading code region
- **map** clause: map data between host and device \( \rightarrow \) moving data
  - to|tofrom|from: mapping directions
  - Use array region
AXPY Example with OpenMP: Multiple device

```c
void axpy_mdev_v1(REAL* x, REAL* y, int n, REAL a) {
    int ndev = omp_get_num_devices();
    #pragma omp parallel num_threads(ndev)
    {
        /* chunk it for each device */
        int devid = omp_get_thread_num();
        omp_set_active_device(devid);
        int remaint = n % ndev;
        int esize = n / ndev;
        int psize, starti;
        if (devid < remaint) {
            psize = esize + 1;
            starti = psize * devid;
        } else {
            psize = esize;
            starti = esize * devid + remaint;
        }
        #pragma omp target device (devid) \
        map(tofrom: y[starti:psize]) \
        map(to: x[starti:psize], a, psize) \
        #pragma omp parallel for shared(x, y, psize, \
        for (int i = 0; i < psize; ++i)
            y[i] += a * x[i];
    }
}
```

- Parallel region
  - One thread per device
- Manually partition array x and y
- Each thread offload subregion of x and y
- Be careful of the loop iteration
AXPY: Make it Simpler

```c
void axpy_mdev_v2(REAL* x, REAL* y, int n, REAL a) {
    #pragma omp target device (*) \ 
    map(tofrom: y[0:n] dist_data(BLOCK)) \ 
    map(to: x[0:n] dist_data(BLOCK),a,n) \ 
    #pragma omp parallel for shared(x, y, n, a) \ 
    dist_iteration(BLOCK) \ 
    for (int i = 0; i < n; ++i) \ 
        y[i] += a * x[i];
}
```

- **device(**): multiple target devices
- **dist_data(BLOCK):** BLOCK distribution of array
- **dist_iteration(BLOCK):** BLOCK distribution of loop iteration
- **Recall HPF**
Language Extensions

- Currently only one target as id: `device (0)`
- Device types, multiple targets, and device virtual topology

<table>
<thead>
<tr>
<th>Device Selection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device(0:10), device(:10)</code></td>
<td>Select 10 devices starting with device id 0</td>
</tr>
<tr>
<td><code>device(5:3), device(5,6,7)</code></td>
<td>Select 3 devices starting with device id 5</td>
</tr>
<tr>
<td><code>device(*)</code></td>
<td>Select all available devices</td>
</tr>
<tr>
<td><code>device(0:10 &amp; OMP DEVICE_NVGPU)</code></td>
<td>Select the OMP DEVICE_NVGPU type devices from the first 10 devices</td>
</tr>
<tr>
<td><code>device(0:4) topology(top1[2][2])</code></td>
<td>Select the first 4 devices and create a 2-d Cartesian virtual topology named top1.</td>
</tr>
<tr>
<td><code>device(4:8) topology([*][*])</code></td>
<td>Select 8 devices to create a system-chosen 2-d Cartesian.</td>
</tr>
</tbody>
</table>

- Virtual topology has the same principle as MPI Cartesian topology
Array Distribution

- **dist_data** appendix of **map** clause

\[
\text{dist_data} (\text{dist_policy}[,. . .]) \quad [\text{topology} \ (\text{dist_target})]
\]

- Similar ideas to HPF

<table>
<thead>
<tr>
<th>DUPLICATE</th>
<th>The full range of this dimension is duplicated into multiple copies, one for each device. This is the default policy if no policy is specified.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK(n)</td>
<td>Divides the indices in a dimension into contiguous, equal-sized blocks of size ( N/P ) (( P ) is the number of devices in the target dimension of the topology) and each device takes one block (( n ) is the number of element in the block; default: ( n=N/P ))</td>
</tr>
<tr>
<td>CYCLIC(n)</td>
<td>Maps every ( i )th block to number ( i ) device of the target dimension of the device topology. (default: ( n=1 ))</td>
</tr>
</tbody>
</table>
Array Distribution Examples

• Matrix multiplication
  – A: (BLOCK, DUPLICATE)
  – B: (DUPLICATE, DUPLICATE)
  – C: (BLOCK, DUPLICATE)
## Array Distribution Examples

<table>
<thead>
<tr>
<th><strong>dist_data</strong>(<em>dist_policy[,...]</em>)</th>
<th><strong>topology (dist_target)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device (0:4) map(to:x[0:n],a,n)</strong></td>
<td>Implicitly map and distribute the full array <em>x</em> and scalars <em>a</em> and <em>n</em> to each of the four devices.</td>
</tr>
<tr>
<td><strong>device (0:4) map(to:x[0:n] dist_data(DUPLICATE))</strong></td>
<td>Explicitly map and distribute the full array <em>x</em> to each of the four devices.</td>
</tr>
<tr>
<td><strong>device (0,1) map(to: x[0:n/2] dist_data(DUPLICATE) topology (0,1), x[n/2:n] dist_data(DUPLICATE) topology (1),a,n))</strong></td>
<td>Distribute the first half of array <em>x</em> to both device 0 and 1, and the second half to device 1. Map scalars <em>a</em> and <em>n</em> to each of the two devices.</td>
</tr>
<tr>
<td><strong>device (0:2) map(to: x[0:n] dist_data(BLOCK))</strong></td>
<td>BLOCK distribution (evenly partition) of array <em>x</em> to the two devices.</td>
</tr>
<tr>
<td><strong>device(0:4) map(to:A[100][0:n][64:1024] dist_data (DUPLICATE, BLOCK, DUPLICATE))</strong></td>
<td>Partition array <em>A</em> from its second dimension using BLOCK policy among the 4 devices. Map the full range of the first dimension and the specified range (64:1024) of the first dimension in each partition to each device.</td>
</tr>
<tr>
<td><strong>device(0:4) topology([2][2])</strong></td>
<td>Distribute the first and second dimension of the partition to the four devices.</td>
</tr>
</tbody>
</table>
# Halo Regions and Halo Update

- **Boundary exchange in domain decomposition**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(to:A[100][100][100] dist_data(BLOCK, BLOCK, DUPLICATE) topology([2][4]) halo(2:periodic,4,))</code></td>
<td>Distribute array A onto 2x4 device topology. BLOCK distribution policy is applied for the first and second dimension. For the first dimension, 2 elements are defined as a halo region with neighbors from both descending index (left) and ascending index (right) directions. Periodic edging policies is selected. For the second dimension, 4 elements for both left and right neighbors and non-edging policy define the halo regions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`map(to:A[100][100][100] dist_data(BLOCK, BLOCK,DUPLICATE) topology([2][4]) halo(2:reflecting</td>
<td>3:periodic,4,))`</td>
</tr>
<tr>
<td><code>halo_update left(A[*][])</code></td>
<td>Update the left halo region in the first dimension from neighbors.</td>
</tr>
<tr>
<td><code>halo_update right(A[][*])</code></td>
<td>Update the right halo region in the second dimension.</td>
</tr>
<tr>
<td><code>halo_update left-right(A[*][*])</code></td>
<td>Update left halo regions first and then right ones in the first and second dimensions.</td>
</tr>
<tr>
<td><code>halo_update leftright(A, A)</code></td>
<td>Update the left and then right halo regions.</td>
</tr>
</tbody>
</table>
Loop Iteration Distribution

- **dist_iteration** clause of **parallel for**
  - Three policy: **BLOCK, CYCLIC and DUPLICATE**
  - **loop chunking**

```c
#pragma omp target data device(*) \
  map(to:n, m, omega, ax, ay, b,\ 
  f[0:n][0:m] dist_data(BLOCK,Duplicate)) \ 
map(tofrom:u[0:n][0:m] dist_data(BLOCK,Duplicate))\ 
map(alloc:uold[0:n][0:m] \ 
  dist_data(BLOCK,Duplicate) halo(1,))
while (((k<=mits)&&(error>tol))
{
  #pragma omp target data device(*)
  #pragma omp parallel for collapse(2) \ 
  dist_iteration(BLOCK)
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      uold[i][j] = u[i][j];
  #pragma omp halo_exchange (uold)
  #pragma omp target data device(*)
```
Compiler and Runtime Implementation

• Using ROSE compiler
• Transformation approach: two-stages
  – 1: multi-device directives → OpenMP standard: e.g. omp parallel +
    single-device, similar to axpy_mdev_v1
  – 2: OpenMP + single device → CPU and CUDA threading code
• Advantages:
  – Reuse existing compiler support
  – Easy debugging
  – One OpenMP thread per device
  – Leverage other OpenMP compiler support
• CUDA Kernel code generation
  – Round-robin schedule to dispatch loop iterations batch by batch
  • Details in the paper
Runtime Implementation

• Two options of device management:
  – OPT1: One user thread + asynchronous calls
  – OPT2: One user thread (or OpenMP thread) per device + sync calls
    • Results shown later

• Inter-device data movement
  – Leveraging GPU P2P data movement if available
  – A relay buffer is needed if P2P is not available
    • Runtime check to auto-switch

• Reduction: multiple level reduction
  – Per device block, per device, multiple device
Runtime Implementation

• Moving data to/from an array subregion
  – Non-contiguous memory space
• Runtime data marshalling
  – Gather data into one location
• Runtime data unmarshalling
  – Scatter data into its original location
Evaluation

• Evaluation platform:
  – NERSC Dirac GPU cluster
  – 2 Intel 5530 2.4 GHz, 8MB cache, 5.86GT/sec QPI Quad core Nehalem processors (8 cores per node)
  – 24GB DDR3-1066 Reg ECC memory
  – 4 NVIDIA Tesla C2050 (Fermi) GPUs, 3GB GPU memory
  – CUDA SDK version 5.5

• Goals
  – To achieve speedup
OPT1: One user thread + asynchronous calls

OPT2: One user thread (or OpenMP thread) per device + sync calls
AXPY Computation vs Data Movement

• Scalability issue for AXPY:
  – Execution time dominated by data movement overhead

<table>
<thead>
<tr>
<th>map_to (X &amp; Y)</th>
<th>computation</th>
<th>map_from (Y)</th>
<th>total GPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.31</td>
<td>1.06</td>
<td>15.77</td>
<td>44.14</td>
</tr>
</tbody>
</table>
Matrix Multiplication

Matrix Multiplication speedup (OPT2/OPT1)

Matrix multiplication execution time (ms)

- 1024x1024
- 2048x2048
- 4096x4096
- 8912x8912
- 12276x12276

1 GPU
2 GPUs
3 GPUs
4 GPUs

1024x1024
2048x2048
4096x4096
8912x8912
12276x12276
16384x16384
20480x20480
Observation

• Hardware limitation prohibits linear performance increment
  – 4 GPUs share a single PCIe switch in our testing platform.
• Better reported performance using multiple threads to manage multiple devices
• Hiding data movement overhead is a necessity
  – AXPY has little computation to fully cover data movement
• Halo region exchange increases overhead
  – Runtime support can detect P2P support and enable GPUDirect for direct communication between GPUs
Related Work: Directly Related

• Generic runtime libraries:
  – StarPU: cost model for scheduling tasks
  – Xkaapi: work-stealing for task scheduling

• Domain-specific abstractions
  – SkelCL: high-level abstractions to express pre-implemented parallel skeletons (patterns), does not support reduction, halo regions
  – SkePU: C++ template library with generic skeletons and containers

• Directive-based approaches
  – OmpSs: manual coding for code/data regions to be offloaded
  – Komada’13: extending OpenACC, implicitly manage inter-GPU communication
Related Work: Broader Scopes

• Single GPU support with high-level directives
  – OpenACC compilers: Tian’13 and Reyes’12
  – OpenMP 4.0 accelerator compiler: Liao’13

• Hybrid programming models for multiple accelerators
  – OpenMP + OpenACC: Xu’13 for a single node
  – Co-Array Fortran + OpenACC: Hart’12 for GPU clusters
  – MPI + OpenACC: Levesque’12 for S3D

• Our approach:
  – Directive-based, compiler-based automatic code generation, explicit control over data and loop distribution
Conclusion

• OpenMP extensions to support multiple accelerators
  – Device types and device topology
  – Array and loop iteration distribution, halo region and update
  – Compiler and runtime support
• Large code base: distributing a single parallel loop to more than 4 (or even 2) is not always what you what:
  – If you do, this work helps
• Multiple GPUs
  – Hardware bus may become bottleneck
• Future work:
  – More applications and Xeon Phi support
Acknowledgement

- Funding agency:
  - NSF and DoE
- NERSC resources
- OpenMP language committee and subcommittees for comments