IBM XL Compiler: OpenMP offloading support for GPU

Ettore Tiotto, Kelvin Li
IBM Canada Laboratory

Some slides courtesy of Alex Eichenberg (IBM)
Modern high-performance scientific applications must exploit **heterogeneous resources** in a performance portable manner.

What Programming Models should we use? What’s the right level of abstraction?
Programming Heterogeneous Systems

- Extracting maximum performance:
  - to program a GPU: you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC
  - to program a host SIMD unit: you have to use Intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints)
  - to program the CPU threads, you might use C11, C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, …

- With OpenMP 4.0/4.5:
  - you can use the same standard to program the GPU, the SIMD units, and the CPU threads
  - Better yet: you can do so in a portable way
OpenMP 4.5

- OpenMP is an industry standard for directive based parallel programming
  - OpenMP has been (and is) widely used to program CPUs
  - In OpenMP 4.0/4.5, new features have been added to provide support for offloading computation to accelerators
  - Industry-wide acceptance: IBM, Intel, PathScale, Cray, PGI, Oracle, MS ➔ application portability
How do we exploit an accelerator in OpenMP?

Simply add a **target** construct around the computation to be offloaded to the accelerator

**map** clauses are used to copy data

```c
#pragma omp target map(to: A, B) map(from: C)
#pragma omp parallel for
for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = A[i][k] * B[k][j];
}
```
A Quick Introduction to OpenMP 4.5

- **target** transfer control of execution to a SINGLE device thread
- the compiler packages the target region into a function
- the OpenMP runtime transfer execution of the function to the device

```
#pragma omp target map(to: A, B) map(from: C)
{ ... }
```
The "distribute" directive can be used to assign loop iterations to teams

```c
#pragma omp target teams
  map(to: a, b) map(from: c)
{
  #pragma omp distribute
  for (int i=0; i<n; i++) {
    #pragma omp parallel for
    for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
        c[i][j] = a[i][k] * b[k][j];
  }
}
```

- the target region is executed by several teams, each team gets a subset of iteration space for the i-loop
- the j-loop iterations are distributed amongst the threads in a team
- distribute schedule controls size of iterations per team, **there is no synchronization between teams**
Optimization: omp distribute parallel for

- Programming model: OpenMP vs CUDA
  - OpenMP uses a fork-join abstraction
  - team regions start with one thread, and parallel threads are created as needed when a parallel region is found
  - CUDA kernels are launched using a grid of blocks/threads (SPMD model)

- Orchestrating CUDA threads to fit the OpenMP programming model can have significant overhead (runtime manages state transitions)

- However OpenMP provides “SPMD-like” directives
  - `distribute parallel for` directive can be used to distribute loop iterations amongst teams and then execute those iteration in parallel using the threads in each team
  - Compiler can generate efficient GPU code for this construct (state transitions not required \(\Rightarrow\) bypass OpenMP runtime system)
  - Default schedule recommended to maximize performance portability
    - HW coalescing on GPU, good cache locality on CPU

```c
#pragma omp target map(from: z) map(to:x,y)
#pragma omp teams
#pragma omp distribute parallel for
for (i=0; i<N; i++)
  z[i] = a*x[i] + y[i];
```
XL C/C++ and XL Fortran Compilers

- XL C/C++ and XL Fortran are full-featured compilers that has been targeting the POWER platform since 1990
  - Aggressively tuned to take maximum advantage of IBM processor technology as it becomes available
  - Industry leading customer support & service
- The XL compiler products use common optimizer and backend technology
  - Leverage mature compiler optimization infrastructure for both CPU and GPU exploitation, across source languages
OpenMP 4.5 support in XL C/C++ and XL Fortran

- CPU/GPU Code Partitioner
- Fortran source: XL Fortran Frontend
- C/C++ source: XL C/C++ Frontend
- High-Level Optimizer: Data flow, loop, other optimizations
- POWER Low-level Optimizer: Low-level Optimizations, Register Allocation + Scheduling, POWER Code Generation
- CPU/GPU W-Code Partitioner
- W-Code (XL IR)
- CPU W-Code
- GPU W-Code
- W-Code to LLVM IR translator
- LLVM Optimizer
- PTX CodeGen
- PTX Assembler
- nvlink
- XL Device Libraries
- CUDA Device Libraries
- Executable for POWER/GPU system

- CPU code is aggressively optimized for POWER
- CUDA Toolkit optimizes device code
- XL’s optimizer sees both host and device code

- XL C/C++ Frontend
- XL Fortran Frontend
- W-Code (XL IR)
- POWER Low-level Optimizer

Libraries
- System Linker
- XL Device Libraries
- CUDA Device Libraries
- CUDA Runtime
- CUDA Driver
- Executable for POWER/GPU system

Libraries
- CUDA Runtime
- CUDA Driver
- Executable for POWER/GPU system

- XL C/C++ Frontend
- XL Fortran Frontend
- W-Code (XL IR)
- POWER Low-level Optimizer

Libraries
- System Linker
- XL Device Libraries
- CUDA Device Libraries
- CUDA Runtime
- CUDA Driver
- Executable for POWER/GPU system

Libraries
- CUDA Runtime
- CUDA Driver
- Executable for POWER/GPU system

IBM

11/30/16
OpenMP 4.0 & 4.5 offloading features

<table>
<thead>
<tr>
<th>Features</th>
<th>OpenMP 3.1 (in target region)</th>
<th>OpenMP 4.0</th>
<th>OpenMP 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OpenMP Directive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Parallel Construct</strong></td>
<td>*omp parallel</td>
<td>*omp target data</td>
<td>*firstprivate, private, default map</td>
</tr>
<tr>
<td></td>
<td>*omp sections</td>
<td>*omp target</td>
<td>*map changes (4.5 semantics)</td>
</tr>
<tr>
<td></td>
<td>*parallel workshare</td>
<td>*omp target update</td>
<td>*if clause for combined directives</td>
</tr>
<tr>
<td><strong>Worksharing</strong></td>
<td>*parallel do/for</td>
<td>*omp declare target</td>
<td>*implicit firstprivate (4.5)</td>
</tr>
<tr>
<td></td>
<td>*omp ordered</td>
<td>*omp teams</td>
<td>*omp target enter data</td>
</tr>
<tr>
<td></td>
<td>*omp single</td>
<td>*omp distribute</td>
<td>*omp target exit data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*omp distribute parallel for</td>
<td>*omp target parallel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*omp declare target</td>
<td>*target nowait &amp; depend</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*combined constructs</td>
<td>*omp target simd</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>*omp master</td>
<td>*omp loop simd</td>
<td>[italic means in progress]</td>
</tr>
<tr>
<td></td>
<td>*omp critical</td>
<td>*Omp distribute parallel do simd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*omp barrier</td>
<td>*omp simd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*omp atomic</td>
<td>*omp declare simd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>*omp flush</td>
<td>*omp distribute simd</td>
<td></td>
</tr>
</tbody>
</table>

[italic means in progress]
Initial support for OpenMP V4.5 features for GPU offloading
Support S822LC systems (POWER8 + P100 via NVLink)
Support for NVIDIA K40, K80, and P100 GPUs
Support for CUDA Toolkit 8.0
Supported Operating Systems: Ubuntu 16.04, RHEL 7.3 ...
compiling the OpenMP programs

-qsmp=omp option: enables the OpenMP compile in the compiler
-qoffload option: enables the target constructs being offloaded to GPU (if available)
  - without the –qoffload option, the target regions are executed on the CPU host

for example

$ xlf90 -qsmp=omp -qoffload test1.f

$ xlc -qsmp=omp -qoffload -qhot -O3 test2.c
use profiling tool

- nvprof provides information about execution
- output:

```
[357x444] use profiling tool

11/30/16

nvprof provides information about execution

output:

```
kli@yc01sros:~/wrk$ nvprof ./test1
==123002== NVPROF is profiling process 123002, command: ./test1
3
==123002== Profiling application: ./test1
==123002== Profiling result:
<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>89.18%</td>
<td>21.632us</td>
<td>1</td>
<td>21.632us</td>
<td>21.632us</td>
<td>21.632us</td>
<td>main$$_SOL$$$_1</td>
</tr>
<tr>
<td>5.94%</td>
<td>1.4400us</td>
<td>2</td>
<td>720ns</td>
<td>512ns</td>
<td>928ns</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>4.88%</td>
<td>1.1840us</td>
<td>1</td>
<td>1.1840us</td>
<td>1.1840us</td>
<td>1.1840us</td>
<td>[CUDA memcpyHtoD]</td>
</tr>
</tbody>
</table>

==123002== API calls:
<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>83.50%</td>
<td>89.855ms</td>
<td>1</td>
<td>89.855ms</td>
<td>89.855ms</td>
<td>89.855ms</td>
<td>cuCtxCreate</td>
</tr>
<tr>
<td>12.24%</td>
<td>13.170ms</td>
<td>1</td>
<td>13.170ms</td>
<td>13.170ms</td>
<td>13.170ms</td>
<td>cuModuleLoadDataEx</td>
</tr>
<tr>
<td>2.05%</td>
<td>2.2069ms</td>
<td>364</td>
<td>6.0620us</td>
<td>208ns</td>
<td>229.30us</td>
<td>cuDeviceGetAttribute</td>
</tr>
<tr>
<td>0.75%</td>
<td>803.09us</td>
<td>4</td>
<td>200.77us</td>
<td>195.09us</td>
<td>203.61us</td>
<td>cuDeviceTotalMem</td>
</tr>
<tr>
<td>0.59%</td>
<td>638.39us</td>
<td>1</td>
<td>638.39us</td>
<td>638.39us</td>
<td>638.39us</td>
<td>cuMemAlloc</td>
</tr>
<tr>
<td>0.46%</td>
<td>521.08us</td>
<td>4</td>
<td>521.08us</td>
<td>521.08us</td>
<td>521.08us</td>
<td>cudaCudaMmu</td>
</tr>
</tbody>
</table>
Questions?