OpenACC2 vs. OpenMP4

The Strong, the Weak, and the Missing to Develop Performance Portable Applications on GPU and Xeon Phi

James Lin$^{1,2}$ and Satoshi Matsuoka$^2$

$^1$Center for High Performance Computing Shanghai Jiao Tong University

$^2$Satoshi MATSUOKA Laboratory Tokyo Institute of Technology
Outline

• OpenACC2 and OpenMP4

• Systematic Optimizations to Portable Performance on GPU and Xeon Phi

• Summary
Why Directive-based Programming?

- We may have many reasons to use directive-based programming, but for me, it can keep the code readable/maintainable from the application developers' point of view.
Directive-based Programming for Accelerators[1]

• **Standard**
  – OpenACC
  – OpenMP >=4.0

• **Product**
  – PGI Accelerators
  – HMPP

• **Research Projects**
  – R-Stream
  – HiCUDA
  – OpenMPC/OpenMP for accelerators

OpenACC2

- Standard version evolution is much faster than OpenMP and MPI: ~1.5year

<table>
<thead>
<tr>
<th>Standard/Version</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenACC</td>
<td>Nov 2011</td>
<td>July 2013</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MPI</td>
<td>June 1994</td>
<td>July 1997</td>
<td>Sept 2012</td>
<td>--</td>
</tr>
</tbody>
</table>

- Supported by PGI/CAPS/CRAY compiler
OpenMP 4.0 for Accelerators

• Released in July 2013, it supports on directive-based programming on accelerators, such as GPU and Xeon Phi

• Directives for
  – Parallelism: target/parallel
  – Data: target data
  – Two levels of parallelism: teams/distribute
<table>
<thead>
<tr>
<th>OpenACC2</th>
<th>OpenMP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Target</td>
</tr>
<tr>
<td>Kernel</td>
<td>N/A</td>
</tr>
<tr>
<td>Data</td>
<td>Target Data</td>
</tr>
<tr>
<td>Loop</td>
<td>Distribute/Do/for/SIMD</td>
</tr>
<tr>
<td>Host data</td>
<td>N/A</td>
</tr>
<tr>
<td>Cache</td>
<td>N/A</td>
</tr>
<tr>
<td>Update</td>
<td>Target Update</td>
</tr>
<tr>
<td>Wait</td>
<td>N/A</td>
</tr>
<tr>
<td>Declare</td>
<td>Declare Target</td>
</tr>
<tr>
<td>{enter, exit} data</td>
<td>N/A</td>
</tr>
<tr>
<td>routine</td>
<td>Declare target</td>
</tr>
<tr>
<td>Async wait</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Tile</strong></td>
<td>N/A</td>
</tr>
<tr>
<td>Device_type</td>
<td>N/A</td>
</tr>
<tr>
<td>Atomic</td>
<td>Atomic</td>
</tr>
<tr>
<td><strong>N/A</strong></td>
<td><strong>Critical sections... Barrier</strong></td>
</tr>
</tbody>
</table>
Experimental Setup

• 2 Target Devices
  – Accelerators used in Supercomputers, so NO AMD GPU
  – GPU Kepler: K40
  – Xeon Phi KNC: 5110p

• 3 Compilers
  – OpenACC: CAPS Compiler
    • V3.4.1, support OpenACC 2.0
    • OpenARC is not public available yet
  – OpenMP4: HOMP-ROSE for GPU, Intel Compiler for Xeon Phi

• 2 Test Cases
  – HydroBench is a miniapp [https://github.com/HydroBench/Hydro](https://github.com/HydroBench/Hydro)
  – Copy benchmark
HOMP, OpenMP compiler for CUDA

• Developed by LLNL, it is build on ROSE\textsuperscript{[1]}, a source-to-source compiler

• It is an early research implementation\textsuperscript{[2]} of OpenMP4.0 on GPU for CUDA5.0

• Support C/C++ only now

\textsuperscript{[1]} \url{http://rosecompiler.org}

\textsuperscript{[2]} C. Liao, Y. Yan, B. R. Supinski, D. J. Quinlan, and B. Chapman, “Early Experiences with the OpenMP Accelerator Model,” IWOMP13
Tools used

CUDA5.5
PTX is open
nvcc* based on LLVM
ptxas/oelelet
GPU Assemble Code
asfemi+
Kepler K20/20X/40

OpenACC
CAPS3.4

OpenMP4.0
HOMP

OpenCL1.2
ICC
Obj
Spir
X86 Assemble Code

KNC 3/5/7110p
Offload
ICC13.0
OpenACC Experiment with Gangs and Workers Configuration

- Gang & Worker with target OpenCL
- Gridify with target OpenCL
- Naive OpenCL

5110P Hydro Benchmark Experiment with Gang&Work mode and Gridify mode
OpenACC Hydro Benchmark Experiment with Grid Thread Configuration

<table>
<thead>
<tr>
<th>Config</th>
<th>K40</th>
<th>5110P</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>16x4</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>32x2</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>64x1</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>16x8</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>32x4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>64x2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>128x1</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>16x16</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>32x8</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>64x4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>128x2</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>256x1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>16x32</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>32x16</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>64x8</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>256x2</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
Performance impact of K40 and 5110P measure with data transfer
Outline

• OpenACC2 and OpenMP4

• Systematic Optimizations to Portable Performance on GPU and Xeon Phi

• Summary
Performance Portability

- Includes source code **portability** of a kernel implementation and **performance** that is commensurate with a device-specific implementation of that kernel [Kokks]

- **Portability** for Accelerators used in HPC
  - Different accelerators: GPU V.S. Xeon Phi
  - Different generations: Kepler V.S. Fermi
  - Different versions: K20 V.S. K40
Portable Performance

- Naïve version, simple directives, 0 optimization
- Ninja version, Best optimization, Ideal performance
- Frog version, achieving reasonable performance on both GPU and Xeon Phi

Algorithm changes

Compiler technology
Optimization for Portable Performance

Accelerators

- SM
- REG
- SHMEM
- GM
- HM

Systematic Optimizations

- SIMD Friendly Algo
- Register Level Data Locality
- SHMEM Level Data Locality
- GM Level Data Locality
- Algorithm Changes

Applications

- Arithmetic Intensity
  - SGEMM
  - FFT
  - LBM
  - Stencil
  - SPMV

Directives

- Data Directives
- Loop Directives

Compiler Technology
Global Memory Level

- Memory Coalescing
- Thread-Grid Mapping

A Single Memory Transaction

<table>
<thead>
<tr>
<th>Addr 1</th>
<th>Addr 2</th>
<th>Addr 3</th>
<th>Addr 4</th>
<th>Addr 5</th>
<th>Addr 6</th>
<th>⋮</th>
<th>Addr 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑ Thread 1</td>
<td>↑ Thread 2</td>
<td>↑ Thread 3</td>
<td>↑ Thread 4</td>
<td>↑ Thread 5</td>
<td>↑ Thread 6</td>
<td>⋮</td>
<td>↑ Thread 32</td>
</tr>
</tbody>
</table>

(b) [ISCA09]

Multiple Memory Transactions

<table>
<thead>
<tr>
<th>Addr 1</th>
<th>Addr 2</th>
<th>Addr 3</th>
<th>Addr 31</th>
<th>Addr 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑ Thread 1</td>
<td>↑ Thread 2</td>
<td>↑ Thread 3</td>
<td>↑ Thread 31</td>
<td>↑ Thread 32</td>
</tr>
</tbody>
</table>

Fig. 1. Performance correlation between K20C and SE10P. [MuCoCoS13]
Shared Memory/L2 (SHMEM) Level

- SHMEM Blocking (data tiling)
- Data Layout: AOS->SOA
- Avoid SHMEM bank conflict
Register Level

- Prefetching
- Loop Unrolling and Jam
- Undocumented Register Bank Conflicts in Kepler [CGO13]

### Difficulties on Kepler – Undocumented Register Bank Conflict

<table>
<thead>
<tr>
<th>Bank</th>
<th>Register Index</th>
</tr>
</thead>
</table>
| Bank 0 (Even 0)| 0 2 8 10 16 18 ...
| Bank 1 (Odd 0)| 1 3 9 11 17 19 ...
| Bank 2 (Even 1)| 4 6 12 14 20 22 ...
| Bank 3 (Odd 1)| 5 7 13 15 21 23 ...

FFMA RA, RB, RC, RD \( \rightarrow \) source registers RB, RC, RD
2-way bank conflict, the throughput drops by 50%
3-way bank conflict, the throughput is only 33.3% of the best case

To get the best instruction throughput
Choose 3 different source registers from different banks.
SIMD Friendly Algorithm

(a) Code from Complex 1D Conv. Example of inner-loop vectorization

```
#pragma omp parallel for
for (int p=0; p<IMAGE_SIZE; p++) {
  float reg_out_r = 0.0, reg_out_i = 0.0;
#pragma simd
  for (int f=0; f<FILTER_SIZE; f++) {
    reg_out_r += in_r[p+f] * coef[f] - in_i[p+f] * coef[f];
    ...
  }
}
```

(b) Code from Libor. Example of outer-loop vectorization

```
for (path=0; path < npath; path+=S) {
  float L[n][S], lam[S], con[S], vscal[S];
  for (j=0; j<nmat; j++) {
    for (i=j+1; i<n; i++) {
      lam = lambda[i-j-1];
      con = delta * lam;
      vscal += con * L[i] / (1+delta* L[i]);
      ...
    }
  }
}
```

[ISCA12]
Compiler Technology

• Fast math, `-fastmath`
  – For `__sin()`, replace several **FMAD** (Floating Point Multiply-Add) with SFU Intrinsic functions
  – Faster, but SP only and loss some accuracy
• Replace FMAD with FMA, `-fmad=false`
  – FMAD -> RND(RND(a*b)+c)
  – FMA (Fused Multiply-Add) -> RND(a*b+c),
  – Improve both accuracy and performance
• Asynchronous
  – Using `async` clauses (`OpenACC only`)
Opportunities for auto-tuning with single source code base

- **Code Level**: self-embed code
- **Compiler Level**: support vendor intrinsic
- **Tool Level**: CAPS Auto-tuning tool
- **Library Level**:
  - Lib for portable performance: Kokkos
  - Drop-in support for CUDA math Lib and MKL
Outline

• OpenACC2 and OpenMP4

• Systematic Optimizations to Portable Performance on GPU and Xeon Phi

• Summary
Summary

• Portable performance would be an important feature for directive programming approach

• OpenACC2 is in early stage
  – gridfication is recommended
  – gang/work/vector directive is weak
  – tile directive is weak
  – cache directive support is still missing

• OpenMP4 is not ready yet
  – teams/distribute support on GPU is still missing

• Systematic optimization would be needed