Understanding Performance Portability of OpenACC for Supercomputers

Suttinee Sawadsitang*, James Lin*†, Simon See*‡, Francois Bodin§, Satoshi Matsuoka†
*Shanghai Jiao Tong University, China
†Tokyo Institute of Technology, Japan
‡NVIDIA, Singapore
§INRIA, France

{m.suttinee, james, simon.see}@sjtu.edu.cn, francois.bodin@irisa.fr, matsu@is.titech.ac.jp

Abstract—Scientific applications need to be moved among supercomputers, such as Tianhe-2 and TSUBAME 2.5. OpenACC provides a directive-based approach for a single source code base with function portability across different accelerators used in the supercomputers. However, the performance portability is not guaranteed by the OpenACC standard. Therefore, we propose a systematic optimization method, instead of auto-tuning by compilers, to achieve reasonable portable performance with minor code modifications. With this method, we evaluate the four kernels from Rodinia benchmark suite and one mini-application Hydro on our hybrid “CPU+GPU+MIC” supercomputer π with the CAPS and PGI compilers. We analyze Parallel Thread Execution (PTX) codes to further understand the performance portability, and find CAPS adopts a different strategy from PGI in thread distribution. The evaluation results show the optimized OpenACC versions can archive a better performance portability ratio than the OpenCL version in some cases. The understanding and the method are valuable for OpenACC application developers to efficiently and correctly use the available OpenACC compilers.

Keywords Performance Portability, OpenACC, OpenCL, GPU, MIC

I. INTRODUCTION

Accelerators (or co-processors) have been widely used in supercomputers. More than 10% of supercomputers in TOP 500 list use the accelerators, especially the TOP ones such as Tianhe-2 in China and TSUBAME 2.5 in Japan. As scientific applications live longer than supercomputers, we need to move the codes among the supercomputers. Therefore, application developers have to write different versions for different accelerators, such as OpenMP for Intel MIC or CUDA for NVIDIA GPU. Instead of developing and maintaining the different versions, a single source code base seems to be the only reasonable economic solution.

OpenCL provides the solution for a single source code to be executed across different accelerators, however, it is complex and less productive for application developers. By contrast, OpenACC not only has the same function portability as OpenCL but also offers more considerable simplification and productivity improvements than OpenCL. However, performance portability across different accelerators used in the supercomputers such as Intel MIC and NVIDIA GPU is not guaranteed by the OpenACC standard. Proposed by the CAPS and OpenARC compilers respectively, the auto-tuning technology aims to archive performance portability by compilers. The technology seems, however, not ready for production codes yet.

Therefore, inspired by the traditional programming approach [1], we propose a systematic hand-written optimization method for OpenACC codes to achieve reasonable portable performance. The method includes the four steps: 1) adding independent directives; 2) thread distribution; 3) unrolling loops; 4) tiling. We applied this method to the four kernels, including LU Decomposition (LUD), Gaussian Elimination (GE), Breadth First Search (BFS), and Back Propagation (BP), from Rodinia benchmark suite and one mini-application Hydro, and evaluated them on the “CPU+GPU+MIC” supercomputer π of Shanghai Jiao Tong University (SJTU) with the CAPS compiler 3.4.3 and the PGI compiler 14.9. We also analyzed the performance portability of OpenACC with the Parallel Thread Execution (PTX) codes, the pseudo-assembly language for CUDA [2].

Concisely, the two contributions of our work are the following:

- We propose the systematic optimization method with minor code modifications for OpenACC to archive reasonable portable performance across GPU and MIC.
- We analyze the PTX codes for the OpenACC codes compiled by CAPS and PGI on GPU. To our best knowledge, this is the first study to adopt PTX analysis in the performance evaluation of OpenACC. By analyzing the PTX codes, we can explain the similarity and difference between the two OpenACC compilers in terms of their translation strategies and optimization behaviors.

The rest of the paper is organized as follows: Section II illustrates the OpenACC programming model and the latest OpenACC standard. Section III describes the systematic optimization method. Section IV introduces the evaluation approach including the test bed and the test cases. Section V discusses the results with PTX codes analysis. Section VI presents related work about performance portability of OpenACC. Section VII concludes the paper with future work.

---

1Equal contribution by the first two authors.

2In this paper, NVIDIA GPU and Kepler K40 refer to the same device. Intel MIC and Xeon Phi 5110P refer to the same device.
II. OpenACC Programming Model

A. High level directive-based approach

Unlike low level approach as CUDA or OpenCL, the OpenACC programming model provides a high level directive-based approach for accelerators. It requires programmers to add OpenMP-like compiler directives to compute-intensive parallel regions that can be offloaded to accelerators. It also allows programmers to define the data movements between the host CPU and accelerators.

A high level approach such as OpenACC or OpenMP is believed to enable a higher productivity at the cost of performance compared to a low level approach. A low level approach offers, on the other hand, fine-grained synchronization primitives (thread barriers) and enable on-chip memory management (shared memory). The high level approach is, however, able to maintain a single source code base, thus, making it more readable and maintainable for the application developers.

B. New features of OpenACC 2.0

Released in July 2013, the latest version of OpenACC standard offers the several major improvements over OpenACC 1.0, including:

1) **Routine Directives** specifies that the compiler should generate a device copy of the function/subroutine in addition to the host copy.
2) **Unstructured Data Regions** provides a means for beginning and ending a data region in different program scopes.
3) **Atomics Directive** ensures a variable is accessed atomically, preventing race conditions and inconsistent results.
4) **Multiple Device Type** enables programmers to set different gang/worker/vector for NVIDIA GPU and AMD GPU with the device_type clause.
5) **Tiling loops clause** will be discussed in Section III-D.

C. OpenACC compilers

The three commercial compilers support OpenACC: CAPS, PGI, and CRAY. The CAPS compiler is a source-to-source compiler to generate CUDA and OpenCL codes based on OpenACC directives for NVIDIA GPU, AMD GPU, and Intel MIC. It planned to fully support OpenACC 2.0 standard in December 2013, but had been stopped developing due to bankruptcy in July 2014. The PGI compiler can only compile OpenACC codes for NVIDIA GPU and AMD GPU. It is likely plans to support Intel MIC in the future. The CRAY compiler is for CRAY computers only.

There are two open-source compilers available. OpenARC, developed in Oak Ridge National Lab, is a C-based source-to-source compiler framework built on the top of Cetus infrastructure. It is able to support NVIDIA GPU, AMD GPU, and Intel MIC but still in a closed beta. A branch of GCC Fortran for OpenACC 1.0 is also available.

III. Systematic Optimization Method

The systematic optimization method can be applied either by inserting directives or changing the compiler flags as listed in Table I. It includes the four steps as shown below:

**TABLE I: Compiler flags used in the method**

<table>
<thead>
<tr>
<th>Flags</th>
<th>Compilers</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O4</td>
<td>PGI</td>
<td>Specifying optimization level</td>
</tr>
<tr>
<td>-fast</td>
<td>PGI</td>
<td>Using fast math library</td>
</tr>
<tr>
<td>-Mvect</td>
<td>PGI</td>
<td>Using vectorization</td>
</tr>
<tr>
<td>-Munroll</td>
<td>PGI</td>
<td>Using ILP unrolling optimization</td>
</tr>
<tr>
<td>-Msafepr</td>
<td>PGI</td>
<td>Specifying no pointer aliasing</td>
</tr>
<tr>
<td>-fastmath</td>
<td>CUDA C</td>
<td>Using fast math library</td>
</tr>
<tr>
<td>-prec-div=false</td>
<td>CUDA C</td>
<td>Specifying architecture</td>
</tr>
<tr>
<td>-code=sm_35</td>
<td>CUDA C</td>
<td>Specifying architecture</td>
</tr>
<tr>
<td>-arch=compute_35</td>
<td>CUDA C</td>
<td>Specifying architecture</td>
</tr>
<tr>
<td>-Xhmppcg -grid-block-size,32x4</td>
<td>CAPS</td>
<td>Changing numbers of gridify mode</td>
</tr>
</tbody>
</table>

A. Step 1: Adding independent directives

We identify the independent loop as listed in Table II, then add #pragma acc loop independent directives to enable the loop to be executed in parallel. One consequence worth noting is the PGI compiler cannot change the thread distribution configuration once the independent directives are added. This step is more likely parallelization instead of optimization, however, as it has a big impact on the performance portability, we include it as an optimization step.

**TABLE II: The dependency in loops**

<table>
<thead>
<tr>
<th>dependent loop</th>
<th>independent loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=2; i&lt;5;++) for (i=2; i&lt;5;++)</td>
<td></td>
</tr>
</tbody>
</table>

B. Step 2: Thread distribution

OpenACC defines three-level parallelism as gang(n), worker(n), and vector(n), comparing two levels in CUDA and OpenCL, as listed in Table III. PGI cannot change thread distribution configurations once independent directives are added, because it will automatically choose the “best” thread and thread block numbers. By contrast, the CAPS compiler has two modes to set thread distribution:

1) **Gang mode**. The gang and worker directives can be used in the #pragma acc loop gang(n), worker(n) clause.
2) **Gridify mode**. Gridify is a special mode in CAPS and can be only applied when the independent directives are added. The Gridify size can be set by either the #pragma hmppcg blocksize 32x4 directive or the compiler flag (-Xhmppcg -grid-block-size,32x4). The CAPS compiler generates one-dimension grid for a single for loop and two-dimensions grid for nested loops.

C. Step 3: Unrolling loops

CAPS only provides the unroll-and-jam functions as the HMPP (Hybrid Multicore Parallel Programming) directives [3]: #pragma hmppcg(cuda) unroll(8), jam and
TABLE III: Parallelism defined in OpenACC and implemented by the compilers

<table>
<thead>
<tr>
<th>OpenACC Standard</th>
<th>CAPS</th>
<th>PGI</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gang Worker</td>
<td>Gang Worker</td>
<td>Gang Worker</td>
<td>Thread block</td>
<td>Global work</td>
</tr>
<tr>
<td>Vector</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#pragma hmppcg(opencl) unroll(8), jam. As the PGI compiler has not provided the directives yet, we use the compiler flag -Munroll instead, as shown in Table I.

D. Step 4: Tiling

The latest OpenACC 2.0 standard can group a part of a loop into a tile to reduce data transfers by reusing variables. The CAPS compiler provides the OpenACC directive #pragma acc tile(n) as well as the HMPP directive #pragma hmppcg tile i:n for tiling. The tiling optimization version generates more PTX instructions, as a single for loop may be transformed into a nested loop. The PGI compiler has not supported tiling yet, although experienced programers can manually tile loops by two dimensional thread blocks. Therefore, we do not apply tiling with PGI due to complexity of programming.

Tiling in OpenACC cannot be expected to improve the same performance as in CUDA, because as OpenACC codes still have to access the global memory on GPU. The CUDA codes can read data from GPU shared memory, shown in Figure 1, which is on-chip and has much lower latency than off-chip global memory [4].

```plaintext
1 for t < size/Tile_Width do
  2    Sa ← a /*load global to shared memory*/
  3    Barrier synchronize
  4    Check the boundary case
  5    for k < Tile_Width do
  6        Computing sum by using Sa instead of a
  7    end
  8    Barrier synchronize
  9    a ← sum
10 end

(a)
```

```plaintext
1 for t < size/Tile_Width do
  2    Check the boundary case
  3    for k < Tile_Width do
  4        Computing sum by using a
  5    end
  6    a ← sum
10 end

(b)
```

Fig. 1: Tiling in CUDA (a) and OpenACC (b)

IV. EVALUATION APPROACH

A. Test bed

1) Hardware: We conduct the experiment on our 260TF hybrid “CPU+GPU+MIC” supercomputer π, which ranks NO.1 in China’s universities. We use one GPU node which has two NVIDIA Kepler K40 with two Intel Sandy Bridge CPU E5-2570 2.6GHz, and one MIC node which has two Intel Xeon Phi 5110P with two same type of CPU as in the GPU node.

2) Software: We use the CAPS compiler 3.4.3 to generate CUDA and OpenCL codes and the PGI compiler 14.9 to generate CUDA codes only. Then we compile the CUDA and OpenCL codes with CUDA 5.5 on GPU, and the Intel C/C++ compiler 14.0.2 to compile the OpenCL codes on MIC, as shown in Figure 2.

```
Fig. 2: The code generation process in this study
```

B. Test cases

We evaluate the four kernels from Rodinia benchmark suite [5] and one mini-application Hydro [6][7]. We choose LU Decomposition (LUD), Gaussian Elimination (GE), Breadth First Search (BFS), and Back Propagation (BP) as they are available in OpenACC. We modify the source codes to support both CAPS and PGI compilers. As shown in Table IV, LUD is a compute-intensive kernel and can be seen as a matrix form of GE. The well-known graph algorithm BFS is a data-intensive kernel. BP for neural network model is both compute-intensive an data-intensive. The mini-application Hydro [6] is developed to study the galaxy formation. We have optimized these kernels with our method and put the source codes on github [8].

TABLE IV: The four kernel benchmarks

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Dwarves</th>
<th>Domains</th>
<th>Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU Decomposition</td>
<td>Dense Linear Algebra</td>
<td>Linear Algebra</td>
<td>4K matrix</td>
</tr>
<tr>
<td>Gaussian Elimination</td>
<td>Dense Linear Algebra</td>
<td>Linear Algebra</td>
<td>8K matrix</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>Graph Traversal</td>
<td>Graph Algorithms</td>
<td>32M nodes</td>
</tr>
<tr>
<td>Back Propagation</td>
<td>Unstructured Grid</td>
<td>Pattern Recognition</td>
<td>20M layers</td>
</tr>
</tbody>
</table>

C. Comparing the PTX codes

We count and compare the PTX instructions for the four kernels to further understand the performance of the codes generated by different compilers and optimized by each step of our method on NVIDIA GPU. We categorize PTX instructions as listed in Table V. In the plots detailing the PTX instruction composition data movement encompasses both data transfers to shared and global memory. It is obvious that the latter takes more clock cycles to complete.

The analysis only considers a static count of the PTX instructions and cannot actually count the number of actually executed instructions. This is due to two reasons: First, the platform independent PTX assembly is translated on the fly to the device specific Shader Assembly (SASS) and, second, there is no easily available profiling mechanism for counting the executed SASS instructions.
V. RESULTS AND ANALYSIS

A. LU Decomposition (LUD)

1) Optimization results: LU Decomposition (where ‘LU’ stands for ‘Lower Upper’) is an algorithm to calculate solutions of a set of linear equations. The compute-intensive LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix.

We have applied the three optimizations from our method: 1) thread distribution; 2) unrolling loops; 3) tiling. The baseline version is an unoptimized C code with OpenACC directives. As shown in Figure 3, the baseline version compiled by CAPS has almost the same performance on both GPU and MIC but is about 1000 times slower than the same version compiled by PGI on GPU. We use the Gang mode to distribute the threads because the Gridify mode in CAPS can be only applied when the independent directives are added. This optimization bridges the performance gap between the CAPS and PGI versions on GPU. It may be because the default thread distribution configuration of CAPS is less optimized than PGI’s. Neither the unrolling loops for both CAPS and PGI nor the tiling for CAPS improves the performance.

We do not present the results of adding the independent directives as it is the same as the baseline version, because the independent directives cannot be added due to the dependencies found in the loops. PGI does not support tiling, either with directives or compiler flags.

Due to the different algorithms used in the OpenCL version and the OpenACC version of Rodinia benchmark suite, we did not compare their results. Furthermore, the OpenCL version was applied several optimizations, which are unavailable in OpenACC, and it aims to be an ideal solution. If we were to present both versions, it would lead to an unfair comparison between two different implementations.

To further understand why and how the thread distribution can work well while the other two optimizations cannot, we analyze different thread distributions in detail and compare the PTX codes, as presented below.

2) Thread distribution: We analyze the thread distribution to answer these two questions: 1) Why the baseline version of CAPS is 1000 times slower than the same version of PGI? 2) How to find the best configuration for thread distribution to archive the best portable performance across GPU and MIC?

For the first question, the CAPS compiler should set the thread distribution as gangs(192) and workers(256) by default, shown in Table III, as the compilation logs print ‘Loop ‘i’ was shared among gangs(192) and workers(256)’. However, we find it actually sets to gang(1) and worker(1) when we examine the generated HMPP codelet files. We find the same situation in other test cases we evaluated, therefore, it may be a bug of the CAPS compiler.

For the second question, we build the three heat maps with various thread block sizes (gang) and thread sizes (worker or vector) for the elapsed time with CAPS on GPU/MIC and PGI on GPU to find out the best thread distribution configuration, as shown in Figure 4. The scale colors of the maps are from bright to dark. The brightest background with the smallest number has the best performance, while the darkest has the worst.

CAPS and PGI have the similar performance on K40. To fully utilize SIMD on K40, which the width is 32 bits, we need to set the worker for threads around 32 and the gang more than 256. However, the performance also depends on the memory bandwidth. In LUD, thread equaling to 16 has the best performance because LUD is bounded by the memory bandwidth. Therefore, the gang and worker for the best performance of LUD on GPU K40 are (>256, ~16).

For MIC 5110P which has 60 cores and each of them can execute four threads, the gang and worker to archive the best performance is (240, 1). Combining the results on GPU with the results on MIC, we believe the thread distribution for the best performance portability across GPU and MIC can be found in (>256, ~16).

3) Comparing the PTX codes: For the same OpenACC source code, PGI generates more PTX instructions than CAPS, as shown in Figure 6. The optimized thread distribution version does not change PTX for both CAPS and PGI, although it significantly improves the performance of the CAPS OpenACC version. Unrolling loops increases the PTX instructions in different categories for CAPS as expected. However, we suspect neither unrolling loops for PGI nor tiling for CAPS can work, as the PTX instructions remain the same.
B. Gaussian Elimination (GE)

1) Optimization results: Another compute-intensive kernel we evaluated is Gaussian Elimination (GE), which computes results row by row and solves for all of variables in a linear system. It must synchronize between iterations, but the values calculated in each iteration can be computed in parallel [5].

We have applied the six optimizations including four of them from our systematic optimization method. The other two are: 1) reorganizing loops, which can turn three kernel loops into two; 2) advanced thread distribution. We cannot apply loop-reorganization with PGI, because the PGI compiler detects a loop and execute it in parallel automatically when we apply two-dimension parallelism for nested loops. When the compiler fails to detect, few things can be done by programmers. By contrast, the CAPS compiler allows programmers to insert the directives manually to both outer and inner loops.

The baseline OpenACC version is an unoptimized C code with the three kernel loops and has the similar performance on GPU and MIC, as shown in Figure 7. Adding independent directives makes the CAPS and PGI compilers automatically apply the thread distribution optimization (with their default configurations), especially when each iteration has different global work sizes according to the i iteration. The CAPS compiler can set two-dimension parallelism for the global and local work sizes, for example [32,4]. The PGI compiler can only set one-dimension parallelism, for example [128,1], to execute the outer loop in parallel and the inner loop sequentially, once it detects any suspicious dependency in the inner loop.

Neither by inserting the directives to the outer loops in the version compiled by CAPS nor by compiling with the compiler flag with PGI does not improve the performance for the unroll and jam optimization. However, the reason is different, according to the PTX analysis in Section V-B3.

The tiling optimization in CAPS can use two-dimension Gridify on a single for loop by transforming the loop into a nested loop. However, as the optimization does not reuse shared variables, the performance remains the same.

We also evaluate the performance portability of the OpenCL version, which has only two kernel loops. To compare fairly, we reorganize the OpenACC version to have the same two kernel loops. As the OpenCL version usually only sets global work size to constant input numbers in two-dimensions of the local work size, the optimized CAPS OpenACC version with independent directives can run even faster than the OpenCL version with a constant number in the thread distribution setting.

2) Advanced thread distribution optimization for OpenCL

Inspired by the CAPS compiler: We find an efficient thread dis-
// i is the loop iteration of outer loop.
__hmppecg_call.setSizeX((Size - i - 1) / 32 + 1);  // set global work group size X dimension in OpenCL
__hmppecg_call.setSizeY((Size - i - 1) / 4 + 1);  // global work group size Y dimension
__hmppecg_call.setSizeX(32);  // set local work group size
__hmppecg_call.setSizeY(4);  // set local work group size
__hmppecg_call.setWorkDim(2);

Fig. 8: Advanced thread distribution configuration

distribution configuration automatically set by the CAPS compiler for OpenACC when examining the generated HMPP codelets, shown in Figure 8. It sets global work and local work group size for two-dimensions in the nested loops. We apply the same optimization, which may not be commonly used by application developers, to the OpenCL version and archive the best performance in this evaluation.

3) Comparing the PTX codes: The CAPS compiler and the OpenCL compiler generated the similar PTX codes for the baseline version, as shown in (Figure 9). Therefore, the thread distribution optimization becomes the key to improve the performance. The CAPS compiler generated five more global instructions than the OpenCL compiler. The CAPS OpenACC version optimized with the advanced thread distribution can achieve better performance than the baseline OpenCL version.

The underlying reason of marginal performance improvement with the unroll-and-jam optimization is different for CAPS and PGI. We find the PTX codes of the CAPS version with the optimization are almost the same as the version without the optimization. By contrast, the PGI version has more PTX instructions, especially the arithmetic and data movement instructions are almost double. Therefore, we believe the CAPS compiler just provided the fake successful message while the PGI compiler optimized but somehow failed to improve the performance.

We also find tiling in CAPS did not use shared memory in GPU because no ld.shared or st.shared instructions have been found.

C. Breadth First Search (BFS)

1) Optimization results: The data-intensive kernel Breadth First Search (BFS) is a widely used searching algorithm in graph theory. We have applied the three optimizations and the reorganization optimization to the OpenACC version to be a fair-comparison with the OpenCL version.

The baseline version compiled by CAPS runs faster on MIC, as shown in Figure 10, than GPU because the HMPP codelet implies the kernels were executed sequentially as gang(1) and worker(1) and the MIC has a higher single thread performance than the GPU. The baseline version compiled by PGI seems have the best performance, however, we find the kernels do not run on GPU after we set the environment variable PGI_ACC_TIME to 1 and profile the kernels with nvprof. By contrast, the OpenCL baseline version runs 9 times slower on MIC than GPU.

The versions with the independent directives compiled by CAPS run as the Gridify mode in parallel and archives 400 times and 30 times speedup on both GPU and MIC to the baseline version, respectively. However, The same version compiled by PGI still runs sequentially, even it detects the independent directives, because the PGI compiler adopts a more conservative strategy for the thread distribution with the hints input from programers. It may ignore the independent directives in complex loops to avoid the potential risk of getting wrong results. Although the PGI version with independent directives runs sequentially, it is still faster than the CAPS version, because the PGI version only needs to transfer the data four times in total between CPU and GPU while the CAPS version needs to transfer three times in each iteration, as shown in Table VII.

<table>
<thead>
<tr>
<th>TABLE VII: BFS execution modes and data transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default modes</td>
</tr>
<tr>
<td>CAPS</td>
</tr>
<tr>
<td>PGI</td>
</tr>
</tbody>
</table>
2) Comparing the PTX codes: Usually the baseline version compiled by PGI runs much faster than the same version compiled by CAPS. It means PGI has the “smarter” default thread distribution than CAPS. However, it may be too smart to be correct in some cases. We find few PTX instructions for the baseline version compiled by PGI, shown in Figure 11, therefore, we suspect the version does not execute the kernels on GPU at all.

We regroup the loops to make the OpenACC versions have the same structure as the OpenCL version as possible. After reorganizing, PTX instructions show no big difference between the PGI OpenACC version and the OpenCL version in every category. By contrast, the CAPS compiler generates fewer data movement instructions, especially the expensive global memory access instructions, such as cvta.to.global, ld.global, and st.global.

![Fig. 11: PTX instructions of BFS for CAPS and PGI](image)

D. Back Propagation (BP)

1) Optimization results: BP is a machine learning algorithm to train weights of connecting nodes on a layered neural network, which the processing of all nodes in each layer can be done in parallel. For this compute-intensive and data-intensive kernel, we have implemented the OpenACC version of `bpnn_adjust_weights` and `bpnn_layer_forward` functions based on the OpenMP version and applied the four optimizations and the reduction optimization, which significantly improves the performance for PGI.

As shown in Figure 12, The baseline OpenACC version compiled by CAPS is faster on MIC than GPU, because the kernels are executed sequentially and the MIC has a higher single thread performance than the GPU. The same reason we explained in BFS. Adding the independent directives can improve its performance 9 times on GPU and twice on MIC.

Usually the CUDA version generated by the CAPS compiler runs faster than, or at lest the same as, the OpenCL version. However, the OpenCL codes generated by the unroll-and-jam version runs faster than the generated CUDA codes on GPU, because the CAPS compiler may have applied the unroll-and-jam to the OpenCL codes while failed to apply the same optimization to the CUDA codes. We were informed by CAPS that we cannot profile the PTX codes of the generated OpenCL codes to further understand the reason.

The OpenCL version runs faster than the OpenACC version is mainly because it can use the shared memory effectively for the `bpnn_layer_forward` function while the OpenACC version cannot [4].

2) Inserting the reduction directive: After adding the independent directives to the outer loops, we insert the reduction directive `#pragma acc parallel reduction` to the inner loops. The example of reduction in CUDA is shown in Figure 13. Both the CAPS and PGI compilers generate `st.shared` and `ld.shared` instructions with this directive. The PGI version runs much faster than the CAPS version, because the PGI version executes the `bpnn_layer_forward` function in parallel, but the CAPS version fails to perform the reduction optimization on GPU and even cannot get the correct results on MIC.

```c
for(unsigned int s=1; s < blockDim.x; s += 2) {
    if (tid % (2+s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
// write the result for this block to global mem
if (tid == 0) g_odata[blockIdx.x] = sdata[0];
```

Fig. 13: The example of reduction in CUDA

3) Comparing the PTX codes: PGI generates more PTX instructions than CAPS on GPU, and it also ignores the independent directives as the two bars (Base and Indep) of the PTX instructions are identical as shown in Figure 14. Both the CAPS and PGI compilers generate the `st.shared` and `ld.shared` instructions with this directive. However, the reason why the CAPS version fails to improve the performance on GPU and cannot execute on MIC is still unknown. The unrolling optimization takes no effects for either CAPS or PGI as the generated PTX instructions remain the same as the previous step.

E. Hydro

The mini-application Hydro [6], developed by RAMSES [9] to study the galaxy formation, includes both data-intensive movements and compute-intensive kernels. It is open source and has about 5000 lines of code. The OpenCL version has 27 kernels while the OpenACC version encompasses 22 nested
loops distributed into 22 OpenCL or CUDA kernels. We change GCC to the Intel compiler, add independent directives, and distribute threads with the Gridify mode for the CAPS OpenACC version. We also change thread distribution sizes for the OpenCL version. However, we cannot compile Hydro with the PGI compiler because PGI is sensitive with pointer allocations and pointer conversions. We need more time to modify the head files of Hydro to fix this problem.

The performance results of Hydro are the same as we expected, as shown in Figure 15. The baseline OpenACC version runs faster on GPU than MIC, but slower than OpenCL. With the Intel compiler, the OpenACC versions can run faster than the versions compiled with GCC because the Intel compiler decreases the elapsed time on CPU. The adding independent directive and the thread distribution optimizations improves the performance 1.3 times on GPU and 200 times on MIC.

![Fig. 14: PTX instructions of BP for CAPS and PGI](image)

![Fig. 15: The elapsed time of the OpenCL version (left) and the CAPS OpenACC version (right) of Hydro on GPU and MIC](image)

**F. Performance Portability across GPU and MIC**

We define the Performance Portability Ratio (PPR) as in Equation 1 to qualitatively measure the performance difference of a single source code base application across GPU and MIC.

\[ PPR = \frac{MIC_{elapsedTime}}{GPU_{elapsedTime}} \] (1)

We only compare the results of the OpenACC versions of three kernel benchmarks and the mini-application compiled by CAPS with the hand-written OpenCL versions on GPU and MIC, because 1) the OpenACC version of LUD cannot be compared fairly with the OpenCL version as they use different algorithms; 2) the PGI compiler has not supported MIC yet.

![Fig. 16: The PPR of optimized the CAPS OpenACC versions and the OpenCL versions across GPU and MIC](image)

According to the results shown in Figure 16, we find:

1) The optimized OpenACC versions are able to have better a PPR than the OpenCL versions across GPU and MIC, according to the yellow bars and the green bars (lower is better).
2) Both optimized OpenACC and OpenCL versions run faster on Kepler K40 than MIC 5110P as all the PPR are larger than 1.

**G. Limitations**

- **Number of test cases:** We choose the four OpenACC kernels from Rodinia benchmark suite and one mini-application for this study and conclude that our proposed systematic optimization method can work well based on the evaluation results. We plan to evaluate more OpenACC benchmarks and applications to strengthen our conclusion.
- **PTX comparison:** Comparing PTX instructions of two versions to explain the performance differences needs two assumptions: 1) data transfers between CPU and GPU/MIC should be the same and 2) the thread distribution configuration also needs to be the same. This profiling approach may not be accurate in some cases because PTX is static information and the CUDA runtime do some optimizations when translating PTX codes to SASS.

**VI. RELATED WORK**

The performance portability of OpenACC has been studied recently on different accelerators with different compilers. Dolbeau et al [10] presented the results about Hydro OpenCL version, generated by the CAPS OpenACC compiler, to archive portable performance across different accelerators by using the auto-tuning technology. By contrast, our work shows the hand-written optimization method with minor programming efforts can also archive reasonable performance portability for OpenACC.
The authors’ previous work [11] evaluated the OpenACC kernels from SHOC, STREAM, and EPCC benchmark suites by using the CAPS compiler. This work extends the previous work to evaluate the kernels and the mini-applications with the CAPS and PGI compilers. Furthermore, we apply the optimizations to the test cases.

Sabne et al [12] studied the performance portability with the OpenARC compiler for the twelve kernels across GPU, MIC, and AMD GCN. By contrast, our work is compiler-neutral with the CAPS and PGI compilers which adopt different strategies for the thread distribution.

Herdman et al [13] evaluated two OpenACC constructs parallel and kernels with the CAPS and PGI compilers for CloverLeaf, a mini-application, to understand the performance difference of these two constructs, while this study focuses on the impact of the thread distribution on the performance portability.

VII. Conclusions and Future Work

To understand the performance portability of OpenACC, we evaluated the performance of the four kernels from Rodinia benchmark suite and the mini-application Hydro on GPU and MIC, and compared their PTX codes on GPU. The results showed the performance portability of the OpenACC codes can be improved by our systematic optimization method and achieve better performance portability than the OpenCL versions in some cases. This comparison study can be valuable for OpenACC application developers to efficiently and correctly use the available OpenACC compilers. Therefore, we believe:

- OpenACC is a promising approach to archive performance portability across GPU and MIC. The systematic optimization method with the minor code modifications is a practical solution.

- The thread distribution plays an important role in the performance portability of OpenACC. The OpenACC compilers provide sophisticated strategies for thread distribution configurations.

We will improve the systematic optimization method, such as inserting the data region directives for data-intensive kernels. We plan to explore the possibility of adopting the OpenARC compiler as our main research vehicle for performance portability of OpenACC since the CAPS compiler had been stopped developing.

VIII. Acknowledgment

We would like to thank Dr. Guido Juckeland of TU Dresden for kindness mentoring and anonymous reviewers for insightful comments. We also thank Prof. Akira Nukada and Mr. Tetsuya Hoshino of Tokyo Institute of Technology, Dr. Eric Germaneau of SJTU, and Mr. Jeremy Fouriaux of CAPS/Nodasys for their valuable suggestions. This research is supported by China National High-Tech R&D Plan (863 plan) 2014AA01A302. James also greatly acknowledges support from Japan JSPS RONPAKU (Thesis PhD) Program.

REFERENCES


[8] “Source codes of the test cases used in this study” Available: https://github.com/suttinee/OpenACC


