Evaluating Performance Portability of OpenACC

Amit Sabne[1], Putt Sakdhnagool[1], Seyong Lee[2], and Jeffrey S. Vetter[2,3]

[1] Purdue University, West Lafayette IN 47907, USA
[2] Oak Ridge National Laboratory
[3] Georgia Institute of Technology
{asabne, psakdhna}@purdue.edu, {lees2, vetter}@ornl.gov

Abstract. Accelerator-based heterogeneous computing is gaining momentum in High Performance Computing arena. However, the increased complexity of the accelerator architectures demands more generic, high-level programming models. OpenACC is one such attempt to tackle the problem. While the abstraction endowed by OpenACC offers productivity, it raises questions on its portability. This paper evaluates the performance portability obtained by OpenACC on twelve OpenACC programs on NVIDIA CUDA, AMD GCN, and Intel MIC architectures. We study the effects of various compiler optimizations and OpenACC program settings on these architectures to provide insights into the achieved performance portability.

Keywords : OpenACC, Performance Portability, High Performance Computing

1 Introduction

Recent years have seen a growing trend in the adoption of heterogeneous computing across many areas including mobile and high performance computing (HPC) [15]. Presently, NVIDIA CUDA GPUs, AMD GPUs, and Intel Xeon Phi are the prominent heterogeneous architectures. While all of these devices possess tremendous computational power, they have significant differences in their architectures as compared to traditional, latency-optimized CPUs. Furthermore, even within architectural families (e.g., CUDA GPUs), new generations of each architecture often differ significantly from their predecessor. Indeed, these devices are being more tightly-integrated into node architectures, eliminating limitations like the high-latency PCIe bus, and frequently changing the device memory model [14], adding on-chip programmer-managed caches, and having a lower number of core registers.

So far, many device-specific programming models and optimization strategies have been developed to exploit these architectures: CUDA [10] for NVIDIA GPUs, and Intel Language Extensions for Offload (LEO) [1] for Xeon Phi are two such architecture-specific programming models. However, despite the availability of these relatively high-level programming models, programming heterogeneous systems requires considerable expertise and architectural knowledge to obtain high performance.

Ultimately, the goal of any programming system is to maintain a reasonable level of *performance portability*, where, ideally, programmers can write their application once, and execute it *efficiently* on any architecture without manual intervention. Unfortunately, current solutions like CUDA and LEO are not even *functionally portable* across devices. So, programmers are forced to have multiple versions of the code for each device that they must maintain and validate, which is tedious, error prone, and generally unproductive. The first programming model to allow such functional portability was OpenCL [12]. However, due to the low-level nature of this model, programming in OpenCL was tedious. OpenACC [11] was therefore proposed to address two major issues: i) allow functional portability across various heterogeneous architectures, and ii) ease the process of porting a serial or OpenMP application to individual heterogeneous devices.

OpenACC provides a set of directives, or *pragmas* that allow programmers to port an existing serial/OpenMP (C or FORTRAN) application to a heterogeneous system. The programmer must determine the compute intensive, parallel regions in the application, and insert the OpenACC directives on these regions. The underlying compiler framework is then responsible for generating executable instructions for the target devices, while the runtime system is responsible for coordinating code execution and data movement among the multiple devices in the node. OpenACC offers the benefit that programmers can incrementally offload and control computation with these directives. Still, even with this level of abstraction in OpenACC, the compiler and runtime system must make many accurate decisions to ensure that the generated code executes with performance comparable to an expert's manually written version of the application.

Beyond the functional portability, *performance portability* is vital to the success of any programming system because code optimizations for specific architectural features can be evasive. While many researchers have provided systems to offload an application written in a high-level language with programmer annotations onto a specific heterogeneous architecture [13, 5, 7], we are unaware of any studies on understanding the performance portability aspects of a high-level programming model across heterogeneous architectures, like GPUs and Xeon Phis. We believe that this question is of high importance, since an optimization that yields significant benefits on one architecture may be inconsequential on another architecture. The understanding of performance impacts of various optimizations is crucial to an OpenACC programmer.

In this paper, we study the performance portability of several OpenACC applications across different heterogeneous devices. Due to the architectural differences across current heterogeneous architectures, a tuned OpenACC application that runs efficiently on one architecture may not run efficiently on other architectures. Since performance is an outcome of the OpenACC program settings and compiler optimizations, it becomes necessary to understand how each program setting and optimization performs on each target architecture, in order to reason about performance portability. To this end, we use various optimizations instrumented in our OpenARC [9] compiler to measure these trade-offs empirically. Then, using OpenARC's auto-tuning system, we obtain the best performing application configuration on each architecture. Next, to evaluate the OpenACC performance portability, we perform a cross-comparison experiment using the best performing application configuration for application/architecture pair against the other architectures. In the process, we highlight the effects of important program settings and compiler optimizations on each architecture, which, in turn, provide insights on realized performance.

To summarize, we make the following contributions:

- We evaluate the performance portability of the OpenACC programming model across NVIDIA GPUs, AMD GPUs, and Intel Xeon Phi coprocessor architectures on 12 OpenACC applications in an effort to understand the effects of various OpenACC program settings and compiler optimizations.
- From this evidence, we highlight the role of specific optimizations for individual architectural features, and provide a performance portability matrix showing the potential benefits (or costs) of highly-optimized applications.

The remainder of this paper is organized as follows: Section 2 gives a brief introduction to the target architectures and OpenACC programming model. Section 3 describes the OpenARC compiler framework. Section 4 evaluates performance portability of OpenACC. We conclude our work and present our future plan in section 5.

2 Background

2.1 Target Architectures

We now briefly describe the architectural details of NVIDIA's Kepler GPUs, AMD's Graphics Core Next (GCN) GPUs and Intel Xeon Phi (MIC) Coprocessors. The high-performing cards of all these architectures come with their own device memories. Since their primary focus is on achieving massive parallelism, they resort to simpler in-order cores. On Kepler, the individual cores are distributed across Streaming Multiprocessors. GCN distributes them across Compute Units. Being evolved as GPUs, Kepler and GCN provide texture memories that can be used for caching read-only data. Kepler and GCN rely upon the underlying runtime to perform SIMD operations. On the other hand, being evolved from CPU cores, the wider SIMD units on MIC enable it to obtain its peak performance. The compiler has to perform vectorization to make use of the SIMD units on MIC, unless SIMD intrinsics are explicitly inserted by programmers. SIMD width on GCN and MIC is 16, while on CUDA, it is 32 (warp size). MIC distributes parallelism across cores; each core runs four hardware threads. Despite being a coprocessor, MIC runs an Operating System that handles the thread scheduling. Table 1 provides further comparative details of these architectures.

2.2 OpenACC Programming Model

The OpenACC [11] programming model provides a high-level, functionally portable programming approach for accelerators. It requires the programmer to insert

Property	CUDA	GCN	MIC
Programming models	CUDA, OpenCL	OpenCL, C++ AMP	OpenCL, Cilk, TBB, LEO, OpenMP
Thread Scheduling	Hardware	Hardware	Software
Programmer Managed Cache	Yes	Yes	No
Global Synchronization	No	No	Yes
L2 Cache Type	Shared	Private per core	Private per core
L2 Total Size	upto 1.5MB	upto 0.5MB	25MB
L2 Line-size	128	64	64
L1 Data Cache	Read-only + Read-write	Read-only	Read-write
Native Mode	No	No	Yes

Table 1: Comparison of Heterogeneous Architectures

directives, or, *pragmas*, on the compute-intensive parallel regions that can be offloaded to an accelerator. Optionally, the programmer can also prescribe the data movements between the CPU and the accelerator.

In general, many OpenACC constructs are similar to those of OpenMP. A major distinguishing factor is the parallelism deployment. While OpenMP supports just a single level, OpenACC parallelism manifests in three levels. (The offloading model in the latest version of OpenMP (V4.0) supports multi-level parallelism similar to OpenACC, but existing compilers do not support the offloading model yet.) A parallel section can be split into gangs, which can further be split into workers, which can in turn control vectors. The presence of multiple levels helps in making use of the massive parallelism in the heterogeneous architectures. Secondly, these levels make it easier to map the parallelism to different architectures, e.g., in CUDA, the gangs can be mapped to threadblocks and workers can be mapped to threads of a threadblock. OpenACC allows the programmer to write a code without explicit memory transfer clauses, but for enhancing the performance, programmer can provide additional clauses, such as data copy-in and copy-out.

3 Overview of OpenARC Framework

Optimization	Architecture
Data Transfer Optimization	Common
Parallel Loop Swap	Common
Tree-based Reduction Generation	Common
Aligned Memory Access Generation	Common
Loop Unrolling	Common
Texture Memory Loading	CUDA
Automatic Shared Memory Loading	CUDA, GCN
Pitched Memory Allocation	CUDA

Table 2: Optimizations Performed by OpenARC (Partial list)

OpenARC [9] is an open-source compiler framework for C-based OpenACC programs. It is built on top of Cetus [4] infrastructure, a C-based source-to-source translation framework. The OpenARC Intermediate Representation (IR) [2] is derived from the very high-level, human-readable IR of Cetus. OpenARC currently supports OpenACC version 1.0, and some features of version 2.0a [11]. Current OpenARC version supports three architectures, namely, NVIDIA CUDA GPUs, AMD GPUs and Intel Xeon Phi coprocessors.

Setting	**Description**
num_gangs	Number of gangs that operate on a parallel region
num_workers	Number of worker threads that belong to a gang
num_vectors	Number of vector threads that belong to a worker [4]
1D Blocking	Using a single parallel loop
2D Blocking	Using nested parallel loops
Loop Collapse	Collapse nested loops so as to increase the iteration space
Tiling	Tile the parallel loops to gain cache benefits
Data Operations	Creation, destruction and copies

Table 3: OpenACC Program Settings that Affect Performance

3.1 Compiler and Runtime

OpenARC compiler is a source-to-source translation system. It translates the input OpenACC program into a CUDA program for NVIDIA CUDA GPUs, and into OpenCL programs for AMD GCN GPUs and Intel MICs. OpenARC contains a runtime system that handles offloading of code sections on the accelerators and maintains data mappings between the CPU and accelerators.

The compiler, along with the runtime system support brings out certain optimizations as shown in table 2. While some optimizations are architecture-dependent, others are common.

3.2 Automatic Tuning

Apart from compiler instrumented optimizations, programmer-specified program settings can affect the performance of an OpenACC program. Table 3 shows a description of such program settings. The programmer must choose the suitable program settings and compiler optimizations to obtain high performance. Due to a large search space, the task of tuning an OpenACC program is a non-trivial. OpenARC compiler therefore assists the programmer by providing an automatic tuning system, as shown in Fig. 1. The overall tuning process is as follows:

1) *Search Space Pruner* automatically prunes the exponential search space by choosing certain compiler optimizations and program settings that can affect the program performance. 2) With the resultant compiler optimizations and program settings, *Tuning Configuration Generator* generates all possible program configurations, each containing different compiler optimizations and program settings. These configurations guide the rest of the OpenARC compilation passes to generate configuration-specific output accelerator programs. 3) Then, the *Tuning*

[4]OpenARC currently does not support *vectors*, as most CUDA/OpenCL devices support only two-level parallelism.

Engine compiles and executes these code variants to find the best performing configuration. Note that the best performing configuration is specific to a given program on a given architecture.

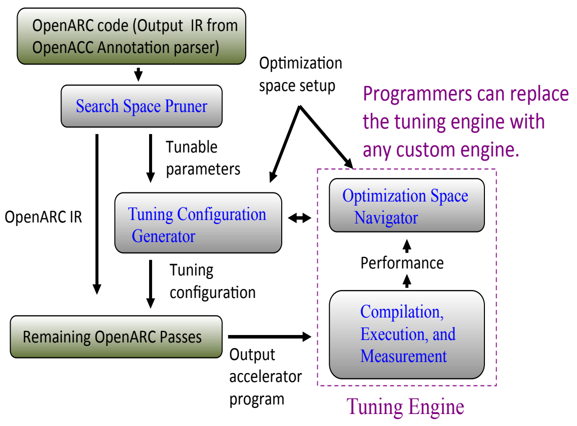

OpenARC code (Output IR from OpenACC Annotation parser)
Search Space Pruner
Tunable parameters
OpenARC IR
Tuning Configuration Generator
Tuning configuration
Remaining OpenARC Passes
Output accelerator program
Optimization space setup
Programmers can replace the tuning engine with any custom engine.
Optimization Space Navigator
Performance
Compilation, Execution, and Measurement
Tuning Engine

Fig. 1: A Built-in OpenARC Tuning Framework. The figure shows that the tuning-related passes are invoked after the OpenACC annotation parser in the overall compilation flows.

4 Performance Portability Evaluation

In this section, we evaluate the performance portability of OpenACC. To do so, we analyse the effects of different program settings and compiler optimizations on different architectures. With an automatic tuning system provided by OpenARC, we find the best performing configurations on each architecture, and analyse how these configurations perform on other architectures.

Table 4: System setup used in the evaluation

Accelerator	**Card**	**Host CPU**	**Driver**	**OS**	**Memory**
NVIDIA CUDA	GTX 680 (Kepler)	Intel Xeon E5520, 4 cores, 8 threads	CUDA 5.0, OpenCL 1.1	Scientific Linux 6.4	12GB
AMD GPU	Radeon HD 7970 (GCN)	2×Intel Xeon E5520, 4 cores, 8 threads	OpenCL 1.2	Scientific Linux 6.4	12GB
Intel Xeon Phi	Knights Corner (MIC)	2×Intel Xeon E5-2670, 8 cores, 16 threads	OpenCL 1.2	CentOS 6.2	256 GB

4.1 Experimental Setup

Table 4 describes the system setup used for this evaluation. We use twelve OpenACC programs in the evaluation, which were manually ported from OpenMP.

JACOBI , *LAPLACE*, *MATMUL* and *SPMUL* are four kernels, while *SRAD*, *HOTSPOT*, *NW*, *LUD*, *BFS*, *BACKPROP*, *KMEANS* and *CFD* are applications from the Rodinia benchmark suite [3]. These OpenACC programs were automatically ported to three different architectures (CUDA, GCN, and MIC) by the OpenARC compiler. Table 5 describes the input sizes used for the evaluation. The focus of our evaluation is to provide insights into the architectural aspects of OpenACC portability, but not to compare the performance of individual architectures. We first describe the effects of various program settings and compiler optimizations on different architectures, which evaluates their performance portability. We next present results of the achieved overall OpenACC performance portability.

Table 5: Benchmarks - Input Specification

Benchmark	Input	Benchmark	Input
JACOBI	grid: 8192×8192, 10 iterations	LAPLACE	grid: 8192×8192, 1000 iterations
MATMUL	matrix size 4096×4096	SPMUL	kkt_power
SRAD	grid : 4096×4096	HOTSPOT	gird 4096×4096
NW	Matrix size : 4096	LUD	Matrix size : 2048
BACKPROP	Input weights : 655360	KMEANS	819200 data points
CFD	232K elements	BFS	No. of nodes : 16 million

4.2 Arranging OpenACC Parallelism

In this section, we describe how the arrangement of parallelism impacts performance in OpenACC programs with nested parallel loops. To do so, we use three different versions. 1D and 2D versions are similar to the ones shown in listings 1.1 and 1.2. The third version uses the loop collapsing clause, which essentially provides a larger iteration space to exploit more parallelism.

Listing 1.1: Single-level (1D) Parallel Version of Jacobi

```
#pragma acc parallel num_gangs(1024) num_workers(64)
{
  #pragma acc loop gang worker
  for (j = 1; j <= SIZE; j++) {
    for (i = 1; i <= SIZE; i++)
          a[i][j] = (b[i - 1][j] + b[i + 1][j]
          + b[i][j - 1] + b[i][j + 1]) / 4.0f;
  }

}
```

Listing 1.2: Two-level (2D) Parallel Version of Jacobi

```
#pragma acc parallel num_gangs(1024) num_workers(64)
{
  #pragma acc loop gang
  for (i = 1; i <= SIZE; i++) {
    #pragma acc loop worker
    for (j = 1; j <= SIZE; j++)
          a[i][j] = (b[i - 1][j] + b[i + 1][j]
           + b[i][j - 1] + b[i][j + 1]) / 4.0f;
  }
}
```

Figures 2, 3 and 4 depict the performance of 1D, 2D and loop collapsed program versions when the number of workers is varied on three benchmarks that contain nested parallel loops. The execution times for a benchmark are normalized with respect to the best execution time of the corresponding benchmark on a target architecture. In these experiments, the number of gangs was fixed to a large constant number (1024), and the number of workers was varied. Fig. 2 shows the effects of such variation on CUDA, while Fig. 3 shows the effects on GCN. For both these architectures, if the number of workers used is low (i.e.

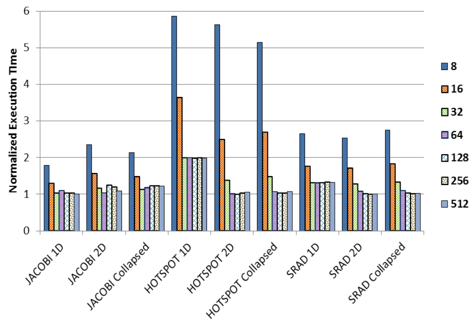


Fig. 2: NVIDIA CUDA

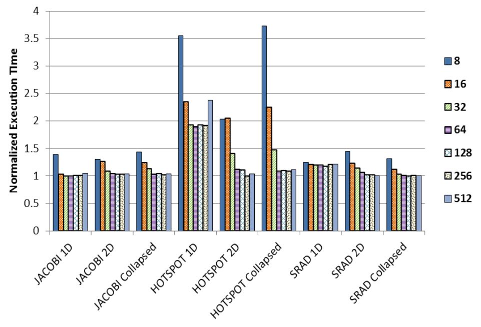


Fig. 3: AMD GCN

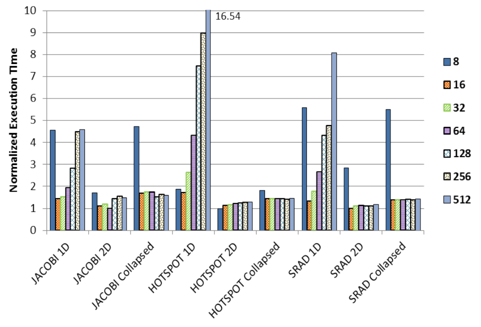


Fig. 4: Intel MIC

8 or 16), the performance is poor owing to the resource underutilization, since the SIMD width on CUDA is 32, while on GCN, it is 16. Since the outer parallel loop in these benchmarks has a high iteration count, parallelizing only the outer loop can suffice, especially if the loop body is small, e.g., in the *JACOBI* benchmark. On *JACOBI*, 1D and 2D versions perform almost equivalently on both GCN and CUDA. However, if the loop body is bigger, which is the case in *SRAD* and *HOTSPOT* benchmarks, the 1D versions perform poorer, compared to the 2D and collapsed versions. Resource underutilization is the reason for this behavior. As an example, consider the case with 1024 gangs and 16 workers, leading to 16384 total threads. In the 1D version, since only the outer loop is parallelized, 4096 iterations of *SRAD* would be scheduled on 16384 threads, with only the first 4096 threads performing useful work. This leads to resource underutilization. Note that on all these benchmarks, as the no. of workers grows large, the execution times saturate on CUDA and GCN. A deviating behavior to this observation is the performance of 1D *HOTSPOT* with 512 workers, on the GCN architecture - the higher number of workers leads to resource contention, resulting in poorer performance.

While the trends on the GPU architectures are almost similar, behavior on Intel MIC is quite different (Fig. 4). Notice that as the number of workers is increased, 1D versions perform much worse, while the 2D versions maintain the same performance. This unintuitive behavior is explained due to the software-thread management that MIC employs. At the beginning of execution, 240 threads are launched by the MIC driver, owing to the presence of 240 hardware threads. OpenCL workgroups are scheduled on these threads at runtime. When 16 workers are used, with 1024 gangs, to distribute 8192 iterations of the *JACOBI* kernel's outer loop, the total threads, as seen by the OpenACC programmer, are $1024 \times 16 = 16384$. Since the loop iteration count is less than this number, only the first 8192 threads would perform useful work. The first 8192 threads are placed in $8192/16 = 512$ workgroups(gangs), which in turn run on 240 software threads, leading to a considerably good performance. However, if the number of workers is 512, then only the first $8192/512 = 16$ workgroups would be placed on the 240 software threads, leading to a huge resource underutilization. So, despite of the number of workgroups being high, as suggested by the Intel OpenCL guide [6], the performance can be low, depending upon the workgroup size. In the 2D case, however, the outer loop of 8192 iterations is always split across 1024 gangs, leading to every workgroup performing useful work. Therefore, on MIC, for 1D versions, the lower the number of workers, better is the performance. Note however that when the number of workers falls below 16 (SIMD width on MIC), the performance reduces due to the lack of vectorization. Intel compiler performs scalarization of the kernel in such cases.

4.3 Effects of Memory Access Coalescing

Most heterogeneous architectures display enhanced performance if consecutive threads access consecutive memory locations in a parallel program. Loop ordering has a high impact on determining the nature of memory accesses. Loop interchange is therefore a primary optimization on heterogeneous architectures [8]. Fig. 5 shows effects of such contiguous memory accesses on three benchmarks, *JACOBI*, *HOTSPOT* and *SRAD*. These benchmarks work on a grid, and accesses are made primarily to the neighboring elements. To measure the memory effects, we manually interchanged the loop order to obtain two versions. First version consisted of consecutive threads accessing contiguous elements in a matrix row (coalesced accesses). The second one did so for contiguous elements in a matrix column (non-coalesced accesses). It can be clearly observed from Fig. 5 that coalesced accesses are crucial to high-performance on all accelerators. The performance impact is relatively less on MIC, owing to its large L2 cache (25MB).

4.4 Effects of Aligned Memory Accesses

Because all tested accelerators are based on SIMD architectures, their memory interfaces are most efficient when data are accessed in an aligned manner. Misaligned accesses will incur unrequested data being transferred, but the amount of the unnecessarily transferred data differs in each architecture. Moreover, the additional data may be used by later threads through L2 cache, having a prefetching effect. Therefore, the overall penalty of the misaligned accesses may not be statically predictable. Fig. 6 shows the effects of the aligned memory accesses on both regular (*MATADD* - a synthetic kernel demonstrating regular, continuous accesses) and irregular (*SPMUL*) benchmarks, where input data for misaligned versions are manually padded and shifted, and corresponding array index expressions are also shifted accordingly. The results indicate that memory access alignment have noticeable performance impact on programs with regular, continuous access patterns (*MATADD*), but less important than memory coalescing, as shown in Fig. 5. To verify the prefetching effect through L2 cache, we created no-caching versions of the translated CUDA programs by manually modifying PTX codes to bypass L2 cache (*CUDA (no-HW caching)*). Comparing caching versions (*CUDA*) with no-caching versions (*CUDA(no-HW caching)*) suggests that aligned accesses are less important in architectures with hardware caching, due to prefetching effects. The figure also shows that MIC has different behavior than CUDA and GCN; on MIC, irregular programs (*SPMUL*) may also have some benefits from aligned accesses due to large L2 caches. The abnormal performance drop *MATADD2D* is caused by too much prefetching; profiling results show that MIC performs more aggressive hardware prefetching (a built-in hardware feature in MIC) on aligned version of *MATADD2D*, but that causes more L2 cache conflicts, incurring more L2 cache misses.

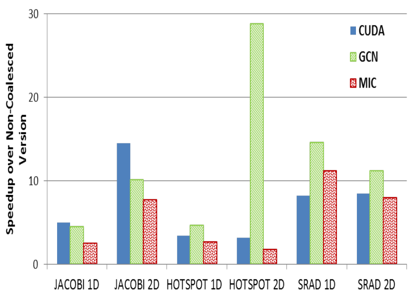


Fig. 5: Memory Coalescing Benefits on Different Architectures : MIC is impacted the least by the non-coalesced accesses

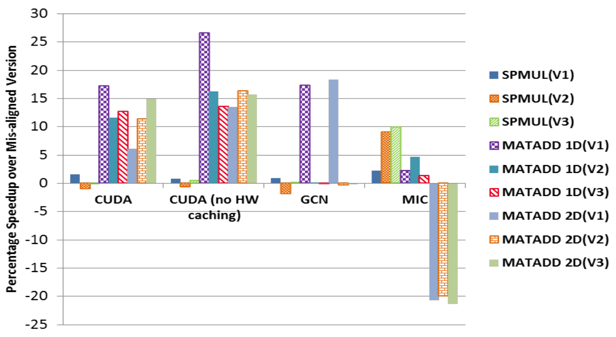


Fig. 6: Benefits of Aligned Memory Accesses on Different Architectures, where each benchmark is configured with different thread mappings. (The number of gangs decreases in v1, v2, and v3 order, while the number of workers is fixed.)

4.5 Tiling Transformation

The traditional tiling transformation is intended at increasing the temporal locality of data. It does so by blocking the computation and working on one block at a time, so that this block can fit in the cache. The transformation comprises loop stripmining, followed by loop interchange. OpenACC standard version 2.0 supports tiling with an additional clause that can be placed on loops.

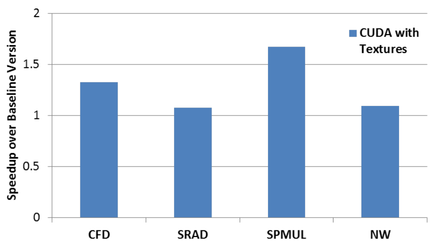


Fig. 7: Effects of caching read-only data on Texture Memory in CUDA

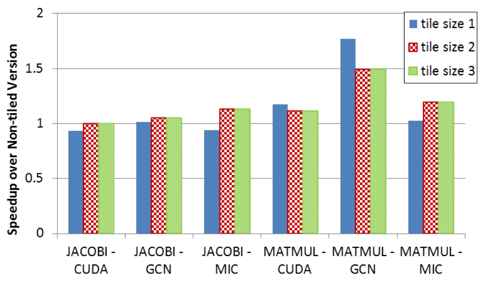


Fig. 8: Impact of Tiling Transformation : *MATMUL* shows higher benefits than *JACOBI* owing to more contiguous accesses

Fig. 8 shows the performance impact of tiling on *JACOBI* and *MATMUL* kernels for three different tile sizes. Tile sizes used are 32×32, 64×64, 128×128, respectively, except for *MATMUL* on MIC where they were 64×2, 64×4 and 64×8. Since all these architectures have a relatively small L1 cache, which is shared among many gangs, tiling can fail to provide performance benefits owing to the unpredictable cache accesses made by different gangs. However, on L2 cache, due to its larger size, we could observe a reduction in misses, resulting in performance improvements. The performance benefits are larger in *MATMUL* than in *JACOBI* owing to a more regular access pattern. The stellar performance improvement (1.7x) was seen on GCN on *MATMUL*, with about 30% reduction in cache misses. The overheads of tiling transformation include the extra loops and their corresponding conditions, which can outweigh caching benefits in certain cases. Using tiling transformation along with the programmer-managed cache can improve the tiling performance further.

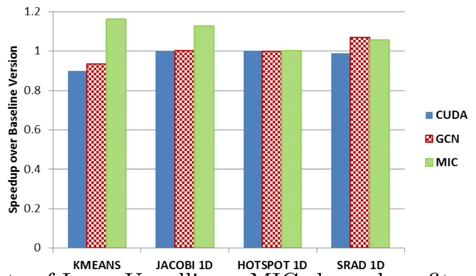


Fig. 9: Effects of Loop Unrolling - MIC shows benefits on unrolling

4.6 Exploiting Device-specific Memories

In order to maintain generality, the OpenACC standard fails to provide constructs to make use of device-specific memories, like textures, on CUDA and

AMD GPUs. To atone for this limitation, the OpenARC compiler can automatically detect the presence of read-only data structures and can place these data elements on device-specific memories. Fig. 7 shows the performance effects of placing read-only data structures on CUDA texture memory on four benchmarks, namely, *CFD*, *SRAD*, *SPMUL* and *NW*. The performance gain can be as high as 67% due to the utilization of texture memory.

4.7 Loop Unrolling

Loop unrolling is a common compiler optimization. By unwinding the loop body, the overhead of index calculation and branching can be reduced. With the help of OpenARC, we unrolled the loops inside kernel regions by various factors and analysed the effects. Fig. 9 shows the unrolling effects on four different benchmarks, which contained loops inside kernel bodies. The largest benefits are seen on MIC, with the speed-up being as high as 17%. We observed that on CUDA, the underlying nvcc compiler performs unrolling itself, leading to no performance benefits due to further unrolling.

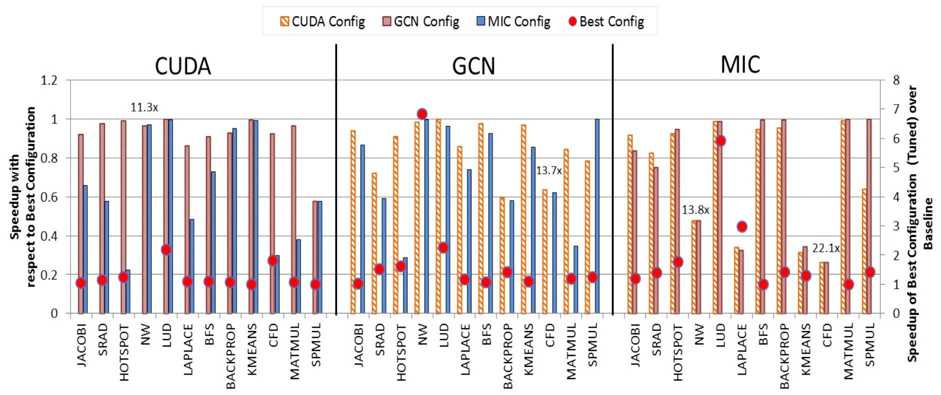


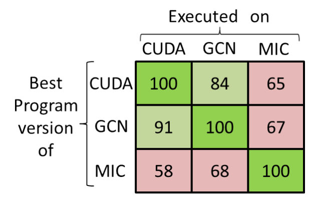
Fig. 10: Performance Portability Evaluation: Best performing OpenACC program on one architecture may not produce best performance on another architecture. Tuning can be highly beneficial over the baseline translated codes.

4.8 Performance Portability Evaluation using OpenARC Auto-Tuning

We now evaluate the achieved OpenACC performance portability by our compiler and runtime system. To do so, we execute the best performing program configuration of one architecture on other architectures and compare the obtained performance with the best possible performance that could be obtained on those architectures. We therefore must generate the best performing configuration of every benchmark on each architecture. We generate these program configurations, which consist of program settings and compiler optimizations, using the OpenARC tuning system.

Fig. 10 displays the cross-architecture performance comparison of the individual benchmarks. The results on the left are run on CUDA, the ones in the middle are run on GCN, while the rightmost ones are run on MIC. On CUDA, for a given benchmark, we found out the best configuration through the tuning system, and then compared its performance against the best configuration of the same benchmark on GCN and MIC architectures. The process was repeated for GCN and MIC. Fig. 11 shows the performance portability achieved across all benchmarks. The numbers in this figure are calculated by taking a geometric mean of the speedups achieved by every benchmark in Fig. 10. Each entry in the box represents the percentage of the best performance achieved on the corresponding architecture. The primary reasons for the performance gap were seen to be: (i) parallelism arrangement, (ii) usage of device-specific memories, and (iii) other architecture-specific optimizations, e.g., using pitched malloc on CUDA. It is worthwhile to note that the performance portability is higher among the GPU architectures. Note that the data transfer optimizations are completely architecture-independent for the evaluated devices; these optimizations, which optimize data allocations and transfers, are equally beneficial on all architectures.

The dots in Fig. 10 correspond to the secondary vertical axis (on the right). They display the performance improvement achieved by OpenARC automatic tuning over the baseline translated versions, for each benchmark on every architecture. The baseline versions are the default versions obtained from the compiler which uses heuristics to set the compiler options and program settings. The benefits of tuning are evident, with the maximum speedup being as high as 22x.



Best Program version of	Executed on CUDA	Executed on GCN	Executed on MIC
CUDA	100	84	65
GCN	91	100	67
MIC	58	68	100

Fig. 11: Performance Portability Achieved across benchmarks : Number in each box represents percentage of the best performance obtained

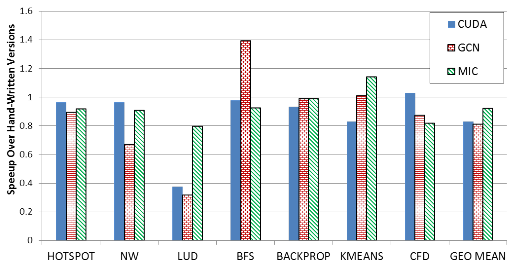


Fig. 12: Comparison of hand-written CUDA/OpenCL programs against auto-tuned OpenARC code versions. Tuned OpenACC programs perform reasonably well against hand-written codes

4.9 Performance Comparison Against Hand-Written Codes

We now present the performance comparison of hand-written CUDA/OpenCL programs against OpenARC generated versions on different architectures (Fig. 12).

For this comparison, we used those benchmarks from Rodinia suite that provided both CUDA and OpenCL versions. On benchmarks that benefit less from the programmer-managed caches, the performance of hand-written versions matches that of the OpenARC versions. Such benchmarks include *BFS*, *BACKPROP* and *CFD*. OpenARC version of *BFS* outperforms the hand-written OpenCL program due to the sub-optimal parallelism arrangement. In the case of *LUD* benchmark, which benefits highly due to the programmer-managed caches, the hand-written versions outperform since the automatic shared memory loading is not possible. However, since MIC lacks programmer-managed caches, OpenARC performs decently with respect to the hand-written OpenCL program on *LUD*. On *KMEANS*, OpenARC performs better than the hand-written code on MIC due to the compiler instrumented unrolling. The out-performance of *CFD* on CUDA is due to the automatic texture memory usage instrumented by OpenARC. The geometric mean bars in Fig. 12 indicate that OpenARC can reach 82%, 81% and 92% of the hand-written codes' performance on CUDA, GCN and MIC architectures respectively.

5 Conclusion and Future Work

Programming models have remained a key obstacle in the widespread adoption of accelerators for general purpose computing. While different high-level programming models have been proposed, there has been a little understanding on the portability aspects of such models. This paper has tried to analyse the portability of one such programming model, OpenACC. The paper gauged the performance portability of the OpenACC programming model across three major accelerators : NVIDIA CUDA GPUs, AMD GCN GPUs, and Intel Xeon Phi Coprocessors (MIC). The evaluation was performed by running program configurations achieving the top performance on one architecture on another architecture to see if the top performance could be achieved. Our analysis shows that while code portability is achievable, performance portability eludes a common programming model, owing to the diversity in the architectures that necessitates architecture-specific optimizations. We are tackling this issue in our ongoing work. Our plan is to use automatic runtime-instrumentation to achieve performance portability across different architectures.

Acknowledgements

The paper has been authored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy. This research is sponsored by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy. The final publication is available at Springer (http://www.springer.com/gp/).

References

1. The heterogeneous offload model for intel many integrated core architectures. [Online]. Available: `http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf`, (Accessed June 25, 2014)
2. OpenARC: Open Accelerator Research Compiler. [Online]. Available: `http://ft.ornl.gov/research/openarc`, (Accessed June 25, 2014)
3. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., ha Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC) (2009)
4. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A source-to-source compiler infrastructure for multicores. IEEE Computer 42(12), 36–42 (2009)
5. Han, T.D., Abdelrahman, T.S.: hiCUDA: a high-level directive-based language for GPU programming. In: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. pp. 52–61. ACM (2009)
6. Intel: OpenCL Design and Programming Guide for the Intel Xeon Phi Coprocessor. [Online]. Available: `http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor`, (Accessed June 25, 2014)
7. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing. IEEE press (2010)
8. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 101–110. ACM (Feb 2009)
9. Lee, S., Vetter, J.S.: Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. pp. 115–120. HPDC '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2600212.2600704`
10. NVIDIA: CUDA. [Online]. Available: `https://developer.nvidia.com/cuda-zone` (2013), (Accessed June 25, 2014)
11. OpenACC: OpenACC: Directives for Accelerators. [Online]. Available: `http://www.openacc-standard.org` (2011), (Accessed June 25, 2014)
12. OpenCL: OpenCL. [Online]. Available: `http://www.khronos.org/opencl/` (2013), (Accessed June 25, 2014)
13. Ravi, N., Yang, Y., Bao, T., Chakradhar, S.: Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 47–58. ICS '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2304576.2304585`
14. Spafford, K., Meredith, J.S., Lee, S., Li, D., Roth, P.C., Vetter, J.S.: The tradeoffs of fused memory hierarchies in heterogeneous architectures. In: ACM Computing Frontiers (CF). ACM, Cagliari, Italy (2012)
15. Vetter, J.S. (ed.): Contemporary High Performance Computing: From Petascale Toward Exascale, CRC Computational Science Series, vol. 1. Taylor and Francis, Boca Raton, 1 edn. (2013)