IPMACC: Translating OpenACC API to OpenCL

Ahmad Lashgar
ECE Department
University of Victoria
lashgar@uvic.ca

Alireza Majidi
Department of Computer Science and Engineering
Texas A&M University
a.majidi@tamu.edu

Amirali Baniasadi
ECE Department
University of Victoria
amiralib@ece.uvic.ca

ABSTRACT

In this paper, we introduce IPMACC a framework for executing OpenACC for C applications over OpenCL runtime. We use over framework to compare performance of OpenACC and OpenCL. OpenACC API abstractions remove the low-level control from programmers' hand. To understand the low-level OpenCL optimizations that are not applicable in OpenACC, we compare highly-optimized OpenCL and OpenACC versions of a wide set of benchmarks. We show that under the investigated benchmarks, exploiting scratch-pad memory as a fast-communication link is the most important optimization that is not applicable in OpenACC. We also introduce a micro-benchmarking suit to investigate the overhead of various OpenACC operations. We compare our framework to a previous open source OpenACC compiler in various aspects.

1. INTRODUCTION

During past few years the OpenCL programming model has evolved as a mature accelerator programming language [24]. There are different OpenCL implementations from different vendors available [28, 29, 30, 31], allowing the programmer to harness the processing capability of multi-core CPUs, many-core GPGPUs and co-processors. In spite of the prevalence of OpenCL support on various platforms, OpenCL development effort is still non-trivial.

In this paper, we introduce IPMACC the framework for executing OpenACC for C applications over OpenCL runtime. IPMACC comes with a set of translators to generate an OpenCL code which is equivalent to the OpenACC code. This offloads the programmer from tedious OpenCL programming efforts. Instead, the OpenCL programmer can spend time on optimizing the code generated by IPMACC. After translation to OpenCL, IPMACC uses the system compiler to generate the accelerator binary from the OpenCL code. Besides the translators, IPMACC also includes a runtime library to support dynamic memory management in OpenACC API.

We verify the correctness and evaluate the performance of our framework under various standard benchmarks. We include benchmarks from Rodinia Benchmark Suit [1] and NVIDIA GPU Computing SDK [6]. We compare the performance of three different implementations of these benchmarks; OpenACC, OpenCL, and CUDA. We also introduce a benchmarking suit to measure the timing overhead of various OpenACC operations. We report the timing overhead of copyin, copyout, and reduction operations under OpenCL and CUDA backends.

We investigate the reason behind the gap between the OpenACC implementation and the highly optimized OpenCL implementation of the benchmarks. Our goal is to identify key bottlenecks in the OpenACC programming model. We show that OpenCL optimizations in using software-managed cache is the main reason causing a huge gap between CUDA and OpenCL. We also show that reduction operations in OpenACC contribute to this gap too.

In summary, we make following contributions:

- We introduce an open-source framework that translates OpenACC applications to OpenCL and CUDA backends and executes the OpenACC applications over OpenCL and CUDA runtimes.
- We use our framework and study OpenACC and OpenCL implementations of various applications. We compare the performance of the implementations and investigate the algorithmic differences between them.
- We introduce an OpenACC benchmarking suit measuring the timing overhead of various OpenACC operations.
- We compare the performance of the code generated by our compiler to a previous work. We provide insight on the decisions made by the compiler which can significantly impact performance in OpenACC applications.

The remaining of the paper is organized as follows. In Section 2 we review OpenCL and OpenACC programming languages. In Section 3 we overview the structure of IPMACC. In Section 4 we explain system and software configurations for the evaluations. Experimental results are presented in 5. In Section 6 we overview related works. Finally, in Section 7, we present concluding remarks.

2. BACKGROUND

2.1 OpenCL Model

In OpenCL [24], an application is composed of host and device codes. The host code executes on CPU and the device code executes on system's accelerator. The host controls the operations of the device through procedure calls to OpenCL API. OpenCL allows programmers to explicitly allocate memory on device and transfer data between the host and the device. The device obtains the device code from kernel and executes it by thousands of light-weight work-items or threads. All threads share common off-chip DRAM memory or global memory. In software, threads are grouped into coarser scheduling elements, referred to as work-groups. Threads within the same work-group execute concurrently and communicate through a fast, per- work-group, on-chip software-managed cache, referred to as local memory. Local memory is much faster than global memory; e.g. under NVIDIA GTX 280, the latency of global memory and local memory are 440 and 38 core cycles, respectively [19].

2.2 OpenACC Model

OpenACC API introduces a set of compiler directives, library routines, and environment variables to offload a region of code from the CPU to the system's accelerator [13]. We refer to this region as the accelerator region. OpenACC has two classes of directives: i) data management and ii) parallelism control. Each directive has clauses providing finer-grain control. Data management directives perform data allocation on the accelerator, data transfer between the host and the accelerator, and passing pointers to the accelerator. Parallelism control directives allow the programmer to mark regions of code, usually work-sharing loops, intended to run in parallel on the accelerator. They also control parallelism granularity, variable sharing/privatization, and variable reduction.

Listing 1: OpenACC matrix multiplication.

```
#pragma acc data \
  copy(a[0:LEN*LEN],b[0:LEN*LEN],c[0:LEN*LEN])
#pragma acc kernels
#pragma acc loop independent
for(i=0; i<LEN; ++i){
#pragma acc loop independent
 for(j=0; j<LEN; ++j){
  float sum=0;
  for(l=0; l<LEN; ++l){
   sum += a[i*LEN+l]*b[l*LEN+j];
  }
  c[i*LEN+j]=sum;
 }
}
```

Listing 2: OpenCL matrix multiplication.

```
int main(){
 ...
 bytes=(LEN*LEN)*sizeof(float);
 d_a=(void*)clCreateBuffer(clctx, CL_MEM_READ_WRITE,
        bytes, NULL, &clerr);
 d_b=(void*)clCreateBuffer(clctx, CL_MEM_READ_WRITE,
        bytes, NULL, &clerr);
 d_c=(void*)clCreateBuffer(clctx, CL_MEM_READ_WRITE,
        bytes, NULL, &clerr);
 clEnqueueWriteBuffer(clcmdqueue, (cl_mem)d_a, CL_TRUE,
        0, bytes, a, 0, NULL, NULL);
 clEnqueueWriteBuffer(clcmdqueue, (cl_mem)d_b, CL_TRUE,
        0, bytes, b, 0, NULL, NULL);
 const char* kernelSource0 ="__kernel void\\
  matrixMul(__global float * a,__global float * c,__global
        float * b, int LEN){\\
  int i=get_global_id(0);\\
  int j=get_global_id(1);\\
  float sum = 0;\\
  for(l = 0; l < LEN; l++) sum += a [i * LEN + l] * b [l *
        LEN + j];\\
  c [i * LEN + j] = sum;\\
 }";
 clpgm0=clCreateProgramWithSource(clctx, 1,
        &kernelSource0, NULL, &clerr);
 clerr=clBuildProgram(clpgm0, 0, NULL, " ", NULL, NULL);
 clkern0=clCreateKernel(clpgm0, "matrixMul", &clerr);
 clerr=clSetKernelArg(clkern0, 0, sizeof(cl_mem),
        (void*)&d_a);
 clerr=clSetKernelArg(clkern0, 1, sizeof(cl_mem),
        (void*)&d_c);
 clerr=clSetKernelArg(clkern0, 2, sizeof(cl_mem),
        (void*)&d_b);
 clerr=clSetKernelArg(clkern0, 3, sizeof(int), (void*)&LEN);
 size_t clgridDim[2]={LEN,LEN};
 size_t clblockDim[2]={16,16};
 size_t cloffsets [2]={0,0};
 uint clndims=2;
 clerr=clEnqueueNDRangeKernel(clcmdqueue, clkern0,
        clndims, cloffsets,
  clgridDim, clblockDim, 0, NULL, NULL);
 clEnqueueReadBuffer(clcmdqueue, (cl_mem)d_c, CL_TRUE, 0,
        bytes, c, 0, NULL, NULL);
 ...
}
```

2.3 Matrix-Matrix Multiplication Example

Listing 1 and 2 illustrate a simple matrix-matrix multiplication in OpenACC and OpenCL, respectively. Ignoring the directive lines, Listing 1 shows the baseline serial multiplication of a and b, storing the result in c. Each matrix is LEN*LEN in size. The outer loops iterated by i and j variables can be executed in parallel. Listing 1 shows how these loops can be parallelized using OpenACC. In this code, *kernels* directive marks a region intended to be executed on the accelerator. *loop* directive guides the compiler to consider the loop as a parallel work-sharing loop. Programmers can control the parallelism using kernels and loop directives. As an example of parallelism control, the independent clause is used to force the compiler to parallelize the loop. This clause overwrites the compiler's auto-vectorization and loop dependency checking. In Listing 1, data clauses hint the compiler to copy a, b, and c arrays from the host to the accelerator, and copy them out from the accelerator to the host. For each array, the [start:n] pair indicates that n elements should be copied from the start element of array. Listing 2 shows how the parallelization can be exploited in OpenCL. To simplify the illustration, we assume the OpenCL context is created on the device and the pointer is stored in clctx. We also assume a command queue is create in this context and the pointer is stored in clcmdqueue. kernelSource0 character pointer points to the string containing the declaration of kernel code. Parallel threads execute the kernel and operate on different matrix elements, based on their unique indexes (i and j). Inside the host code, device memory is allocated for a, b, and c, keeping the pointer in d_a, d_b, and d_c, respectively. Then, input matrices are copied into device memory. Then, a total of LEN*LEN light-weight accelerator threads are launched on the device to execute matrixMul kernel. After kernel completion, the resulting matrix d_c is copied back to the host memory. As presented in Listing 1, OpenACC significantly reduces the accelerator programming effort in comparison to OpenCL. OpenACC hides low-level accelerator-related code from the programmer, leaving the task to the compiler to generate low-level code. Additionally, OpenACC provides a unified view over both host and accelerator code, simplifying debugging and development effort.

3. IPMACC FRAMEWORK

IPMACC is a research framework composed of a set of translators translating OpenACC applications to various accelerator languages (e.g. OpenCL or CUDA). In addition to the translators, IPMACC comes with a runtime library to support dynamic memory management operations in OpenACC API. Our goal in designing IPMACC was to generate the target source code (e.g. OpenCL) which is equivalent to the OpenACC application. This has two advantages. Firstly, this allows taking advantage of the latest innovations in the target compilers (e.g. OpenCL runtime compilers) for executing OpenACC applications. Secondly, the programmer can have an equivalent version of her serial code on accelerators by simply augmenting her code with OpenACC notation. Later, experienced OpenCL or CUDA programmer can perform further optimization on top of that, avoiding development from scratch and saving huge amount of development effort. To this end, we did our best to minimize abstraction and generate direct target source code. Currently, IPMACC can translate OpenACC application two different backends: OpenCL or CUDA. Both translators and runtime library of IPMACC are developed flexible enough to allow easy inclusion of more backends.

Structure. IPMACC framework has a command-line interface for compiling an OpenACC application and generating the destination binary. Compilation starts with validating the OpenACC syntax used in the application. Then, the OpenACC kernels and data regions are extracted from the code to be translated to proper target (OpenCL or CUDA). Then, several static passes parse the code to find dimensions of the parallel loops, type and size of the data, user-defined types, and user-defined procedure calls. After finding all this information, the target source code will be generated. Finally, the target source code is passed to the system compiler (g++ if the target is OpenCL or nvcc if the target is CUDA) to generate the final object code. The command-line tool accepts all compilation flags that the system compiler understands. Hence, the command-line tool can be used for generating intermediate object codes or final binaries.

Features. IPMACC supports most of OpenACC procedure calls and directives. Currently, all procedure calls except synchronizations are supported. IPMACC supports kernels, loop, data, enter, exit, and cache directives. parallel, device selection, and synchronization clauses are yet to be implemented. IPMACC supports the use of user-defined types and user-defined procedure calls in the kernels region. Nested loops are supported and parallel iterations of each loop nest is mapped to a unique dimension of the OpenCL work-group (or CUDA thread-block). IPMACC is an open-source framework and the code is available on github [25].

4. METHODOLOGY

Benchmarks. We use benchmarks from NVIDIDA GPU Computing SDK [6] and Rodinia Benchmark Suit [1]. NVIDIA GPU Computing SDK includes a large set of CUDA and OpenCL test cases, each implementing a massively-parallel body of an application in CUDA and OpenCL very efficiently. Most test cases also include a serial C/C++ implementation. We developed an OpenACC version of these benchmarks over the serial C/C++ code. Rodinia is a GPGPU benchmark suite composed of a wide set of workloads implemented in C/C++. Originally, each of these benchmarks was implemented in CUDA and OpenCL parallel models. Recently, OpenACC implementation of the benchmarks has been added by a third-party [10]. We include N-Body simulation from the SDK and the remaining benchmarks from Rodinia.

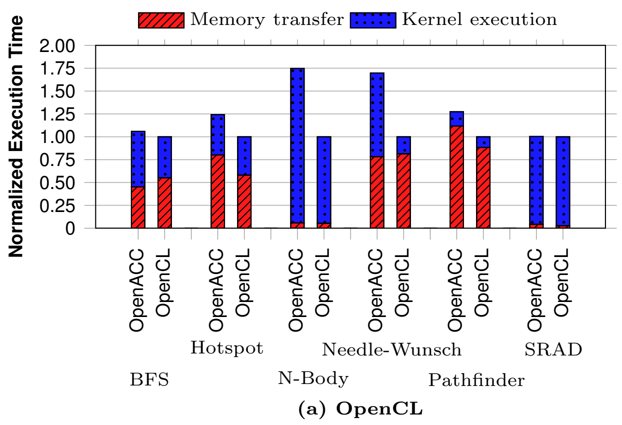
OpenACC Compilers. We use our in-house framework, IPMACC, for compiling OpenACC applications. The framework and benchmarking suit can be obtained from github [25]. IPMACC translates OpenACC to either CUDA or OpenCL and executes OpenACC application over CUDA or OpenCL runtime (e.g. NVIDIA GPUs or AMD GPUs). We validated the correctness of our framework by comparing the results of OpenACC benchmarks against the serial version. For the last part of evaluations, we compare performance of IPMACC to Omni OpenACC compiler [27].

Performance evaluations. We compile the OpenACC version of benchmarks by our framework and run it over CUDA and OpenCL runtime. We compare these to CUDA and OpenCL implementations available in NVIDIA GPU Computing SDK and Rodinia. In order to evaluate performance, we report the total time of kernel execution and memory transfer between host and accelerator. We use *nvprof* for measuring these times in CUDA [7]. We also use *COMPUTE_PROFILE* environment variable of CUDA runtime to measure these times in OpenCL [8]. Every reported number is the harmonic mean of 30 independent runs.

Platforms. We perform the evaluations under an OpenCL and CUDA-capable accelerator. We use NVIDIA Tesla K20c as the accelerator. This system uses NVIDIA CUDA 6.0 [6] as the CUDA implementation backend and NVIDIA OpenCL 1.1 as the OpenCL implementation backend. The other specifications of this system are as follows: CPU: Intel® Xeon® CPU E5-2620, RAM: 16 GB, and operating system: Scientific Linux release 6.5 (Carbon) x86_64. We use GNU GCC 4.4.7 for compiling C/C++ files.

5. EXPERIMENTAL RESULTS

In this section, we evaluate performance of IPMACC under various aspects. First, we compare a set of OpenACC applications to their highly optimized OpenCL/CUDA version. Our goal is to identify OpenACC's programming limitations resulting in the performance gap between OpenACC and OpenCL performance. Second, we compare the execution time of various OpenACC operations under OpenCL and CUDA backends of IPMACC. Finally, we compare performance of IPMACC to a previous open-source compiler.



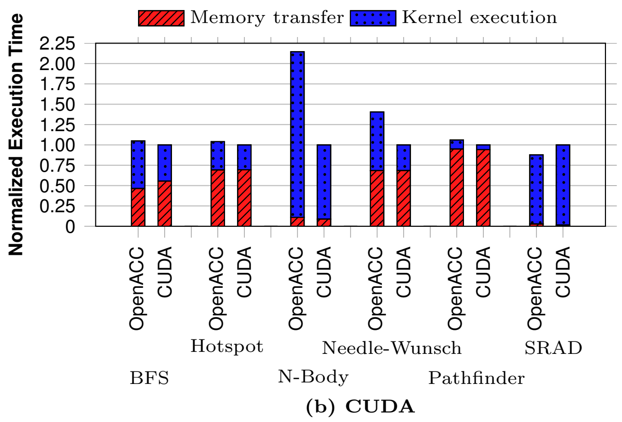


Figure 1: Comparing the execution time of OpenACC with OpenCL or CUDA backend to highly-optimized (a) OpenCL or (b) CUDA implementations. Each bar shows the duration of time that the application spends on memory transfer and kernel execution.

5.1 Standard Benchmarks

Figure 1 reports the execution time for OpenACC applications, compared to their OpenCL (1a) and CUDA (1b) versions. The figure reports the breakdown of time spent on the accelerator; kernel execution (kernel) or memory transfer between host and accelerator (memory). Same trend can be seen under both CUDA and OpenCL. In interest of space, below we limit the discussion to OpenCL implementations.

In most cases, OpenCL's kernel execution portion is shorter than OpenACC. Also, memory transfer times are comparable on both OpenCL and OpenACC. There are exceptions where OpenACC memory transfers are faster (e.g. BFS). Below we investigate the differences between OpenCL and OpenACC. We discuss applications separately providing insight into why OpenCL and OpenACC implementations presented in Figure 1 have different kernel execution and memory transfer times.

BFS. BFS visits all the nodes in the graph and computes the visiting cost of each node. Each node is visited only once. Parallel threads of a kernel visit the nodes belonging to the same graph depth concurrently and the algorithm traverses through the depth iteratively. The operation stops once there is no child to visit. Compared to the OpenCL version, the OpenACC version of BFS spends less time on memory transfers. This can be explained by the fact that the OpenACC version performs data initializations on the GPU. However, the OpenCL version initializes the inputs on the host and transfers the inputs to GPU. Compared to the OpenCL version, OpenACC spends more time on kernel execution, since it forces a debilitating reduction on a global variable. The global variable is a boolean indicating whether there remains more nodes to visit or not. OpenCL avoids global reduction by initializing the variable to FALSE on the host and imposing a control-flow divergent in the kernel to guard the global variable from FALSE writes (allowing TRUE writes).

Hotspot. Hotspot simulates chip characteristics to model the temperature of individual units. At every iteration, the algorithm reads the temperature and power consumption of each unit and calculates new temperatures. Compared to OpenACC, the OpenCL implementation spends the less amount of time on memory transfers and kernel execution.

We explain the faster memory transfers by the memory flags used in clCreateBuffer in the OpenCL version. The OpenCL version creates device buffers using CL_MEM_USE_HOST_PTR flag, avoiding explicit memory transfer API calls. We found that NVIDIA OpenCL implementation performs much faster memory transfers when CL_MEM_USE_HOST_PTR is used[1]. Comparing Hotspot in Figure 1a and 1b also reveals that the memory transfers of the Hotspot OpenCL version is even faster than the CUDA version[2].

We explain faster kernel execution by the ghost zone optimization [4] used in the OpenCL version. In Hotspot, the temperature of each unit depends on its power consumption and neighbors' temperatures. OpenCL kernel exploits this behavior to localize the communication and reduce global memory accesses as follows. In OpenCL, threads of the same work-group calculate the temperature of neighbor units. The OpenCL version updates the new temperature of neighbor units using the threads of the same work-group locally. This local communication reduces the number of kernel launches used to synchronize the temperature across all work-groups, explaining why the OpenCL version comes with shorter execution time. In OpenACC, unlike OpenCL, the software-managed cache cannot be exploited for local communication. Hence, In OpenACC there are higher number of global synchronizations and kernel launches, which in turn harms performance.

N-Body simulation. N-Body models a system of particles under the influence of gravity force. In each timestep, operations of O(N^2) complexity are performed (for a system of N particles) to calculate forces between all pairs of particles. Inherently, there are many redundant memory reads, since the mass and position information of each particle is fetched by other particles N-1 times to calculate its interaction with

[1]We derived a dedicated test case to make sure this flag is the only cause to faster memory transfers in OpenCL.
[2]Notice that using aforementioned flag in the OpenCL backend of OpenACC my harm application semantic. Because automatic update of host or accelerator memory can override the logic of copyin/copyout clauses used by the OpenACC programmer.

other particles.

While both OpenCL and OpenACC memory transfers take about the same time, OpenCL kernels are much faster. The OpenCL version tiles the computations to reduce redundant memory reads [9]. OpenCL exploits local memory to share the particles among all threads of a work-group. In OpenACC, however, the redundant memory accesses are not filtered out by the software-managed cache. As reported, redundant memory accesses can degrade performance significantly.

Needleman-Wunsch. Needleman-Wunsch is a sequence alignment algorithm used in bioinformatics. In either OpenCL or OpenACC, this application traverses a 2D matrix and updates the costs. Upon updating a new cost, four memory locations are read and one location is written.

Although both OpenCL and OpenACC versions spend the same amount of time on memory transfers, OpenCL kernel launch/executions are much faster than OpenACC kernels. The OpenCL version fetches a data chunk of *costs* matrix into shared memory and traverses the matrix at the shared memory bandwidth. This mechanism comes with three advantages: i) filtering redundant global memory accesses by local memory, ii) minimizing global communication by sharing intermediate results stored in the local memory, iii) reducing the number of kernel launches and global communications. The fewer number of kernel launches explains why the kernel time of OpenCL is much lower than OpenACC.

Pathfinder. In Pathfinder kernel, every working element iteratively finds the minimum of three consequent elements in an array. The OpenCL version of Pathfinder performs two optimizations: i) finding the minimum by accessing the data from local memory, and ii) sharing the updated minimum locally among neighbor threads for certain iterations and then reflecting the changes globally to other threads. Such local communications reduce the number of global synchronizations and kernel launches.

However, OpenACC's API is not flexible enough to allow the programmer exploit the local memory in a similar way. Therefore neighbor threads in the OpenACC version do not communicate via local memory. Therefore, each thread fetches the same data multiple times and threads communicate only through global memory. Communication through global memory is implemented through consequent kernel launches.

We explain the faster memory transfers in the OpenCL version in the same way that we explained Hotspot. The OpenCL version uses CL_MEM_USE_HOST_PTR during creating memory buffers.

Speckle reducing anisotropic diffusion. Speckle reducing anisotropic diffusion (SRAD) is an image processing benchmark performing noise reduction through partial differential equations iteratively. Compared to OpenCL, the kernel time of OpenACC version is slightly lower (this difference is more obvious in Figure 1b, comparing OpenACC and CUDA). Three code blocks construct the computation iterative body of this benchmark: one reduction region and two data parallel computations. Our evaluation shows OpenACC version performs 22% slower than OpenCL, upon executing two data parallel computations. However, OpenACC outperforms OpenCL in executing the reduction portion. This is explained by the difference in reduction implementations. First, the OpenCL implementation launches two kernels to reduce two variables, while the OpenACC version performs these two reductions in one kernel region. Second, our OpenACC framework performs the reduction in two levels: reducing along work-items of work-group on GPU and reducing along work-groups on CPU. In the OpenCL version, however, reduction is performed by multiple serial kernel launches, all on the GPU. The OpenACC version spends less time on executing the kernel as part of the computation is carried on host. Meanwhile, performing two levels of reduction imposes the overhead of copying intermediate data from GPU to CPU. This explains why the OpenACC version spends slightly more time on memory transfers and less time on kernel launch/execution.

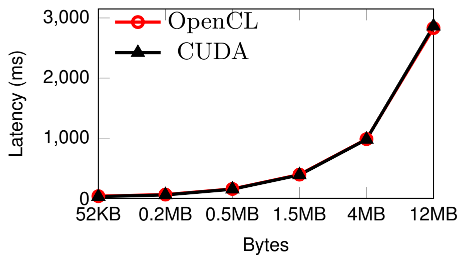
5.2 OpenACC Benchmarking

Figure 2 compares performance of IPMACC backends (OpenCL and CUDA) under various OpenACC operations. These operations include copying data from host to accelerator (*copyin*), copying data from accelerator to host (*copyout*), reducing writes from parallel threads by maximum (*reduction (max)*) and sum (*reduction (+)*) operators, and kernel launch overhead. To perform this experiment, we measure the time for completing one of these operations (e.g. data directive with copyin clause or kernels loop directive with reduction clause). The directive is called within a sequential loop which iterates for 30 times. We report the harmonic mean of these 30 iterations. The OpenACC benchmarking suit that we use here is included in the IPMACC package [25].

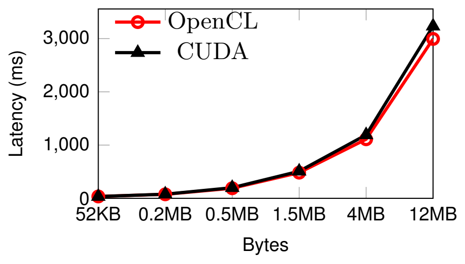
copyin and **copyout**. OpenCL and CUDA backend perform similar under copyin clause. For copyout clause, OpenCL performs slightly faster under larger data sizes.

reduction (max) and **reduction (+)**. IPMACC implements a two-level reduction algorithm; first, reducing the values within the work-group on the accelerator, then, reducing the results of all work-groups on the host. It is introduced by Mark Hariss in [2]. Under both OpenCL and CUDA, reduction time starts to grow remarkably after 59K values. Reduction (max) performs slower than reduction (+) under both OpenCL and CUDA, since finding the maximum causes branch divergence. Comparing OpenCL and CUDA backends, CUDA backend performs slightly faster under both types of reduction.

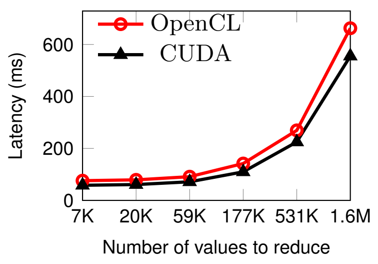
kernel launch. We measure the time taking to launch and execute a kernel with different number of arguments; ranging from one to 16 arguments. To make sure that the compiler is not optimizing the code by removing the kernel arguments, within the kernels region we sum the value of all arguments and write the result back to global memory. Under both OpenCL and CUDA, increasing the number of arguments increases the measured latency. Part of this increase comes from the launch and portion of the increase comes from the increase in the number of sum operations in the kernel. It is not possible to separate these two by OpenACC control. Comparing OpenCL and CUDA, OpenCL kernels perform



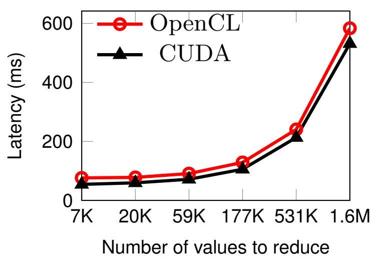
(a) copyin



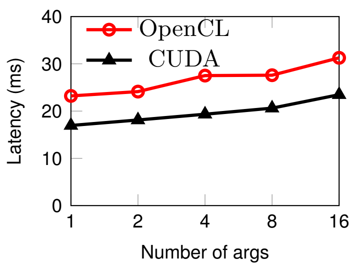
(b) copyout



(c) reduction (max)



(d) reduction (+)



(e) Kernel launch

Figure 2: Comparing the latency of CUDA and OpenCL backends for IPMACC under various OpenACC operations: (a) copyin, (b) copyout, (c) reduction (max), (d) reduction (+), (e) kernel launch.

4% to 8% slower than CUDA.

5.3 Compiler Performance

In this Section, we compare our framework work to Omni OpenACC compiler [27]. We compare the performance of the code generated by Omni and IPMACC under three basic workloads; matrix-matrix multiplication, vector-vector add, and reduction. Our goal is to compare loop-to-thread mapping, overhead of generated code, and efficiency of reduction. Since Omni is only able to run the application over CUDA, here we limit the comparison to CUDA back end of IPMACC.

Matrix-matrix Multiplication. This benchmark consists of three nested loops. The most inner loop is a reduction. Two outer loops can be executed independently in parallel. In this comparison, we parallelize the two outer loops. Figure 3 compares the performance of this implementation compiled by IPMACC or Omni. As reported, under small problem size, IPMACC performs faster and under larger problem sizes, Omni performs faster. We explain this by the mapping of the two nested loops to CUDA threads and thread blocks. IPMACC maps each loop nest along one dimension of the grid. This means IPMACC launches as parallel threads as the number of parallel loop iterations. Omni, however, maps all loops to one single-dimensional grid. Omni maps the first loop across the thread blocks and the second loop across the threads of the thread blocks. Since the number of threads per thread block is limited by an small number (e.g. 1024), Omni also limits the number of parallel threads which are launched for executing the second loop. Accordingly, more than one loop iteration (or task) is mapped to each thread. Below we discuss when it is advantageous and when it is not.

Omni's mapping reduces the number of threads and thread blocks. This is advantageous over IPMACC's mapping when there are large number of parallel iterations, as we see under 256 x 256 and 1024 x 1024 problem sizes in Figure 3. This is because Omni's mapping reduces the overhead of scheduling large number of threads and thread blocks. On the other hand, assigning more than one task to threads and thread blocks comes with an overhead. This overhead includes control statements for the two loops, assigning more than one job (or one loop iteration) to threads and thread blocks. As we see under 16 x 16 problem size in Figure 3, this overhead causes Omni to stay behind IPMACC.

We also found that for different problem sizes, Omni always generates a loop to assign more than one job to each thread block. In our experiments, however, this loop only assigns one job to each thread. Hence, the generation of this loop essentially imposes a control flow overhead and reduces the performance. To clarify and show this overhead, we change the matrix-matrix multiplication implementation and merge and flatten the two outer loops into single unified loop. Figure 4 reports the numbers under the latter implementation. In this case, both compilers generate a code with the same mapping of loops to threads. The difference is an extra loop that Omni generates to assign more than one task to each thread (while always assigning one task to each thread). As reported, in this implementation, IPMACC performs faster than Omni.

Vector-vector Addition. This benchmark adds two input arrays, element by element, and stores the output in another array. Each individual sum can be performed in parallel. Figure 5 compares the performance of vector add under IPMACC and Omni. As reported, IPMACC performs faster than Omni. We explain this by the code generated by the two compilers. IPMACC generates a control statement to control number of active threads. Omni, however, generates a loop to control the number of loop iterations assigned to each thread (assigning more than one task to each thread). The generation of this loop is an extra computation on GPU

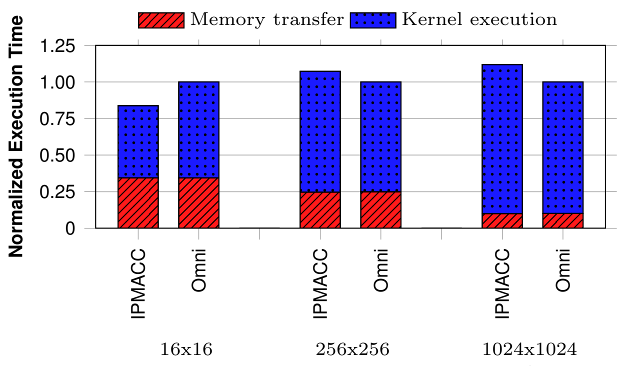


Figure 3: Comparing the performance of IPMACC and Omni under matrix-matrix multiplication workload. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

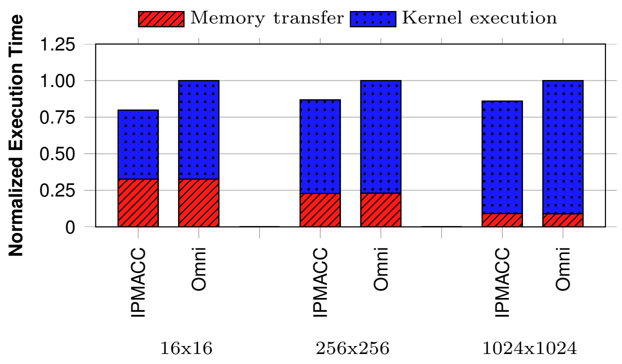


Figure 4: Comparing the performance of IPMACC and Omni under a matrix-matrix multiplication which two outer loops are merged and flattened. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

since the runtime profiling indicates Omni launches enough number of threads and assigns one task to each thread.

Reduction. Figure 6 compares the performance OpenACC *reduction* clause under IPMACC and Omni. Since our implementation reduces the results of thread blocks on the host, we measure the time taking to complete the whole kernels region (before kernels region starts and after it completes). As reported, Omni implementation of reduction is much faster than IPMACC.

6. RELATED WORK

Reyes et al. [11] introduce an open-source tool, named accULL, to execute OpenACC applications on accelerators. The tool consists of a source to source compiler and a runtime library. The compiler translates OpenACC notations to the runtime library routines. The runtime library routines are implemented in both CUDA and OpenCL. Tian et al. [12] introduce an OpenACC implementation integrated in OpenUH [3]. They evaluate the impact of mapping loop iterations over GPU parallel work-items.

Hoshino et al. [23] investigate the impact of memory layout on the performance of NVIDIA Kepler, Intel XeonPhi, and Intel Xeon processors, under directive-based programming languages. They found that having structure-of-arrays is much more efficient than array-of-structures under Kepler and XeonPhi, while it has minor impact on the performance of Xeon. They explain this by the relatively smaller cache used by Kepler (110 Bytes per hardware thread) and XeonPhi (128 KBytes per hardware thread), compared to Xeon (1048 KBytes per hardware thread).

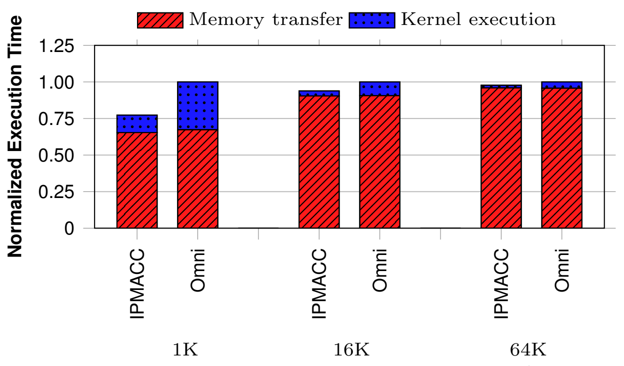


Figure 5: Comparing the performance of IPMACC and Omni under vector-vector addition. Each bar shows the duration of time that the application spends on memory transfer and kernel execution. Each bar group reports for particular problem size.

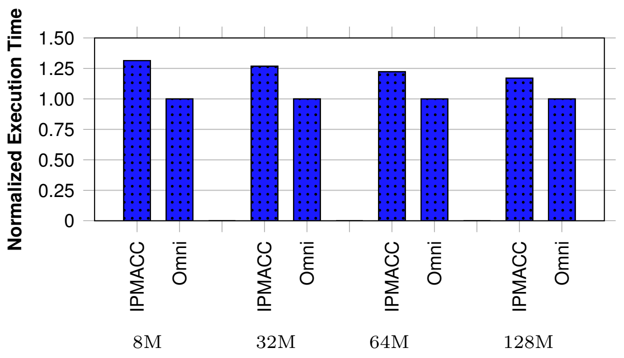


Figure 6: Comparing the performance of IPMACC and Omni under reduction clause. Each bar shows the duration of time that the application spends to complete whole reduction. Each bar group reports for particular problem size.

Herdman et al. [21] compare performance of 'parallel' and 'kernels' constructs under various vendor implementations of OpenACC. They found that most vendors focus on one of these constructs. Comparing quickest construct of the vendors, their performance variations found to be below 13%. They also found 15% to 20% gap between OpenACC and CUDA on NVIDIA GPUs.

Nakao et al. [5] introduce XACC as an alternative to MPI + OpenACC programming model to harness the processing power of cluster of accelerators. XACC offers higher productivity since XACC abstractions reduce the programming efforts. Under small and medium problem sizes, XACC performs up to 2.7 times faster than MPI + OpenACC. This higher performance comes from the PEACH2 interface that XACC communicates through. PEACH2 performs faster than GPUDirect RDMA over InfiniBand under data transfer size of below 256KB. Increasing the problem size, XACC and MPI + OpenACC perform comparable, since the latency of PEACH2 and GPUDirect RDMA over InfiniBand would be equal.

Murai et al. [14] propose an extension to OpenACC to facilitate multi-accelerator processing. Language extension includes on_device clause for data, kernels, and loop directives.

Xu et al. [15] evaluate the performance of Kirchhof Migration under OpenACC. They found OpenACC code executed on the GPU is 20.54X faster than the code executed on single 10-core CPU. They also found that gang serialization may degrade performance significantly. Instead of serialization and broadcast, they suggest to perform the computations redundantly.

Kraus et al. [16] investigated the opportunity to improve the performance of CFD workloads through OpenACC. They applied several CUDA-like optimizations at the OpenACC level, including texture cache and occupancy optimizations. They apply texture memory optimization by declaring variables as constant. They alter the streaming multiprocessor's occupancy by specifying vector_length (or thread-block size). They found that the optimal occupancy is the point with higher cache hit rate, since the CFD workloads tend to work on large working sets. They also transform array-of-structures to structure-of-arrays to optimize memory layout (returned nearly 52% performance improvement).

Govett et al. [17] compare the performance of three different OpenACC implementations under NIM workload. They perform three optimizations in their own implementation, called F2C-ACC. Among these optimizations, they found that variable demotion can improve performance significantly. Variable demotion avoids transferring of the entire dimension of array when only certain indecies are accessed. This can decrease the memory transfer time and also allow generation of more efficient kernel code. For instance, variable demotion on a 1D array, where possible, can replace global memory array accesses with scalar or register accesses.

Wienke et al. [18] compare the performance and development cost of two OpenACC applications to their OpenCL equivalent. In terms of modified code lines, they found OpenACC demands 6.5X lower development effort compared to OpenCL. Meanwhile, they found the best-effort performance gap is of 2.5X. They observed that the OpenACC inflexibility in exploiting software-managed cache causes this large performance gap. Williams et al. [20] study different applications to evaluate the performance gap between OpenACC and CUDA implementations. Niemeyer and Sung [22] investigate the OpenACC and CUDA performance in CFD applications. They show that the performance gap between OpenACC and CUDA narrows down to a negligible value for large data sets. Herdman et al. [21] compare the performance and programming productivity of OpenACC, OpenCL, and CUDA. Under hydorodynamics applications, they found that OpenACC outperforms CUDA and OpenCL on average by 1.35X and 1.24X, respectively. In term of words of code, they conclude OpenACC applications are 11.9X and 8.67X more productive for programming than OpenCL and CUDA, respectively. Additionally, they found OpenACC applications easier to debug and verify, compared to CUDA and OpenCL.

Tabuchi et al. [27] introduce a source-to-source compiler for translating OpenACC applications to CUDA. They have included their translator in Omni compiler [26]. They compare the performance of their compiler to Cray GPU compiler under three different benchmarks. They report speedups of 0.74 to 2.1 times over the Cray compiler. They found that relying on the NVIDIA compiler for generating the accelerator code is the reason why their compiler outperforms Cray compiler. They also compare the performance of OpenACC versions of the applications to hand-optimized CUDA versions.

7. CONCLUSION

In this paper, we introduced IPMACC, a framework for source-to-source translation of OpenACC for C applications to OpenCL and CUDA. We verified the correctness of our framework under various benchmarks. We also investigated the performance gap between OpenACC and highly-optimized OpenCL versions of the benchmarks. We also introduced a benchmarking suit to measure the time overhead of various OpenACC operations, including *copyin* and *reduction*. We also compared the performance of our framework to Omni OpenACC compiler.

We found that there would be a huge gap between the performance of OpenACC and OpenCL implementations once the OpenCL version is optimized to exploit software-managed cache. By comparing IPMACC to Omni compiler, we also found that the performance of OpenACC applications can be highly affected by the decision of compiler in mapping loop iterations to work-items or threads.

8. REFERENCES

[1] S. Che et al., Rodinia: A benchmark suite for heterogeneous computing, IISWC 2009.

[2] Mark Hariss. Optimizing Parallel Reduction in CUDA. Available: http://developer.download.nvidia.com/assets/cuda/files/reduction.

[3] C. Liao et al., OpenUH: an optimizing, portable openmp compiler, Concurrency and Computation: Practice and Experience, vol. 19, no. 18, pp. 2317.2332, 2007.

[4] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). 2009. ACM, New York, NY, USA, 256-265. DOI=10.1145/1542275.1542313 http://doi.acm.org/10.1145/1542275.1542313

[5] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Bokut, and Mitsuhisa Sato. 2014. XcalableACC: extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In Proceedings of the First Workshop on Accelerator Programming using Directives (WACCPD '14). IEEE Press, Piscataway, NJ,

USA, 27-36. DOI=10.1109/WACCPD.2014.6 http://dx.doi.org/10.1109/WACCPD.2014.6

[6] NVIDIA Corporation, CUDA Toolkit 6.0. 2014. Available: https://developer.nvidia.com/cuda-downloads

[7] NVIDIA Corporation, Profiler's User Guide: nvprof. 2015. Available: http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview

[8] NVIDIA Corporation, Profiler's User Guide: command-line profiler. 2015. Available: http://docs.nvidia.com/cuda/profiler-users-guide/#compute-command-line-profiler-overview

[9] J. P. L. Nyland and M. Harris. GPU Gems 3. 2007.

[10] PathScale. Modified rodinia benchmark suite, 2013. [Online]. Available: https://github.com/pathscale/rodinia

[11] R. Reyes et al., accULL: An openacc implementation with cuda and opencl support, Euro-Par 2012.

[12] X. Tian et al. Compiling a High-level Directive-Based Programming Model for GPGPUs. The 26th International Workshop on Languages and Compilers for Parallel Computing. 2013.

[13] The OpenACC application programming interface, 2013. [Online]. Available: http://www.openacc-standard.org

[14] Hitoshi Murai, Masahiro Nakao, Takenori Shimosaka, Akihiro Tabuchi, Taisuke Bokut, and Mitsuhisa Sato. XcalableACC - a Directive-based Language Extension for Accelerated Parallel Computing. Proceedings of the Supercomputing Conference poster. SC '14 poster session. 2014. New Orleans, Louisiana, USA.

[15] Rengan Xu, Maxime Hugues, Henri Calandra, Sunita Chandrasekaran, and Barbara Chapman. 2014. Accelerating Kirchhoff migration on GPU using directives. In Proceedings of the First Workshop on Accelerator Programming using Directives (WACCPD '14). IEEE Press, Piscataway, NJ, USA, 37-46. DOI=10.1109/WACCPD.2014.8 http://dx.doi.org/10.1109/WACCPD.2014.8

[16] Jiri Kraus, Michael Schlottke, Andrew Adinetz, and Dirk Pleiter. Accelerating a C++ CFD Code with OpenACC. Proceedings of the First Workshop on Accelerator Programming Using Directives. WACCPD '14. New Orleans, Louisiana, USA.

[17] Mark Govett, Jacques Middlecoff, and Tom Henderson. Directive-based Parallelization of the NIM Weather Model for GPUs. Proceedings of the First Workshop on Accelerator Programming Using Directives. WACCPD '14. New Orleans, Louisiana, USA.

[18] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC: first experiences with real-world applications. In Proceedings of the 18th international conference on Parallel Processing (Euro-Par'12), Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer-Verlag, Berlin, Heidelberg, 859-870. DOI=10.1007/978-3-642-32820-6_85 http://dx.doi.org/10.1007/978-3-642-32820-6_85

[19] H. Wong et al., Demystifying gpu microarchitecture through microbenchmarking, ISPASS 2010.

[20] David Williams, Valeriu Codreanu, Po Yang, Baoquan Liu, Feng Dong, Burhan Yasar, Babak Mahdian, Alessandro Chiarini, Xia Zhao, and Jos Roerdink. Evaluation of autoparallelization toolkits for commodity GPUs. Parallel Processing and Applied Mathematics. Springer. 2014.

[21] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. High Performance Computing, Networking, Storage and Analysis (SCC). 2012. SC Companion.

[22] K. Niemeyer and Chih-Jen Sung. Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. Journal of Supercomputing. February 2014.

[23] Tetsuya Hoshino, Naoya Maruyama, and Satoshi Matsuoka. An OpenACC Extension for Data Layout Transformation. Proceedings of the First Workshop on Accelerator Programming Using Directives. WACCPD '14. 2014. New Orleans, Louisiana, USA.

[24] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. 2015. Available: https://www.khronos.org/opencl/

[25] Ahmad Lashgar. IPMACC. Available: http://github.com/lashgar/ipmacc/

[26] Omni Compiler Project. Available: http://omni-compiler.org/index.html

[27] Akihiro Tabuchi, Masahiro Nakao, Mitsuhisa Sato. A Source-to-Source OpenACC compiler for CUDA. HeteroPar'2013, Aachen, Germany, August 2013.

[28] Mali OpenCL SDK. Available: http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/

[29] Intel SDK for OpenCL Applications. Available: https://software.intel.com/en-us/intel-opencl

[30] NVIDIA OpenCL SDK. Available: https://developer.nvidia.com/opencl

[31] AMD OpenCL APP SDK. Available: http:// developer.amd.com/ appsdk