Exploring Parallel Programming Models for Heterogeneous Computing Systems

Mayank Daga, Zachary S. Tschirhart, Chip Freitag
AMD Research
Advanced Micro Devices, Inc., USA
{*Mayank.Daga, Zachary.Tschirhart, Chip.Freitag*}*@amd.com*

***Abstract*—Parallel systems that employ CPUs and GPUs as two heterogeneous computational units have become immensely popular due to their ability to maximize performance under restrictive thermal budgets. However, programming heterogeneous systems via traditional programming models like OpenCL or CUDA involves rewriting large portions of application-code. They also lead to code that is *not* performance-portable across different architectures or even across different generations of the same architecture.**

In this paper, we evaluate the current state of two emerging parallel programming models: C++ AMP and OpenACC. These emerging programming paradigms require minimal code changes and rely on compilers to interact with the low-level hardware language, thereby producing performance-portable code from an application standpoint. We analyze the performance and productivity of the emerging programming models and compare them with OpenCL using a diverse set of applications on two different architectures, a CPU coupled with a discrete GPU and an Accelerated Programming Unit (APU). Our experiments demonstrate that while the emerging programming models improve programmer-productivity, they do not yet expose enough flexibility to extract maximum performance as compared to traditional programming models.

Keywords: Programming models, performance, productivity, evaluation, GPU, APU, OpenCL, OpenACC, C++ AMP

I. Introduction

The exigent demands of emerging applications to maximize performance while staying under power and thermal constraints have made heterogeneous computing ubiquitous [1, 2]. Heterogeneous systems have also been recognized to play an important role on the path to extreme scale computing as evident by the fact that half of the top ten supercomputers on the Top500 list possess heterogeneous capabilities [3]. Among various heterogeneous systems, those consisting of a graphics processing unit (GPU) coupled with a traditional CPU have emerged as the most popular due to their remarkable performance-price ratio [4, 5].

GPU programming has made great advancements, from requiring application developers to program geometric shaders in assembly to the evolution of general-purpose programming models like OpenCL™ [6]. However, these programming models suffer from two major issues. First, they require the use of explicit library calls and special compute kernels which leads to rewriting large portions of application-code, making the process sometimes infeasible. Second, they require architecture-specific optimizations which leads to code that is not performance-portable across different architectures like GPUs and multicore CPUs or even across different generations of the same architecture [7, 8]. The compound effect of these issues has inhibited the widespread adoption of GPUs.

Emerging programming models such as C++ AMP and OpenACC hold the promise of alleviating the bottlenecks found in traditional GPU programming models [9–11]. Both are high-level programming models which allow offloading the parallel loops to GPUs. They use advanced language features like pragmas and lambda functions in order to require minimal changes to the source-code and rely on compilers to generate the low-level GPU parallel code. Since the compilers are now responsible for managing architecture-specific data management and execution constructs, these emerging programming models provide performance-portability across a large sweep of architectures.

Heterogeneous computing systems are programmed using a combination of programming models referred to as MPI+X [12], where MPI (or Message Passing Interface) is the de-facto standard for managing inter-node execution and 'X' refers to the programming model for on-node parallel execution such as OpenCL, C++ AMP and OpenACC. In this paper, we evaluate and contrast the popular programming models used for on-node execution in a high performance computing (HPC) scenario. Previous research has compared OpenACC and OpenCL but to the best of our knowledge, we are the *first* to include C++ AMP in a comparative study of GPU programming models. For evaluation purposes, we use a diverse set of four *hand-tuned* scientific proxy applications belonging to different domains and one micro-benchmark: (i) LULESH, (ii) CoMD, (iii) XSBench (iv) miniFE, and (v) an indigenous read-memory benchmark. We use two different architectures to ascertain the portability aspect of the programming models: a GPU coupled with a CPU and an Accelerated Processing Unit (APU). We focus our studies on programming models for a single-node as MPI has been universally chosen in HPC to manage inter-node communication in a multi-node environment.

Our metrics for holistically comparing the programming models include performance, programmer-productivity and the current state of their compilers. We present a performance comparison of the programming models in both single and double precision. We then compare the source lines

of code (SLOC) to deduce the approximate programmer-productivity and also provide a qualitative description of the application development process. This provides an idea of the learning curve involved with each programming model as lower SLOC does not always translate to higher productivity. We also evaluate the programming models in terms of the flexibility offered by their compilers in enabling the application developer to optimize performance.

Our experiments demonstrate that OpenACC and C++ AMP substantially lag behind OpenCL on the discrete GPU because their compilers do not optimally manage the data-transfers on a discrete GPU. However, for OpenCL the programmer explicitly manages these data-transfers which results in better performance. The APU mitigates the data-transfer requirement and hence, provides a level-field for the comparison of programming models. C++ AMP is the most productive on APU on average and is as much as 3× more productive than OpenCL. Of the two emerging languages evaluated, C++ AMP is more promising and provides both improved performance and productivity. Both OpenACC and C++ AMP greatly improve programmer productivity but their current compilers are not robust enough to contest the performance of OpenCL. To provide a "best of both worlds" scenario, AMD is developing *Heterogeneous Compute*, a programming model which is highly productive while consisting of a rich-set of features for performant programming.

The rest of the paper is arranged as follows. Section II provides a background on discrete GPU and APU. Section III describes the three programming models that we evaluate. Section IV presents the description of the proxy applications followed by the experimental setup in Section V. Section VI presents the comparative study of the programming models. Section VII briefly introduces Heterogeneous Compute. Section VIII presents related work followed by conclusions in Section IX.

II. Heterogeneous Computing Systems

This section presents an overview of the two heterogeneous computing systems that are used to evaluate various programming models.

A. Discrete GPU paired with a CPU

A discrete GPU is a co-processor that resides on the PCIe interface in a traditional CPU+discrete GPU setup, as shown in Figure 1. The x86 cores and the host-memory interfaces are part of the CPU core. A discrete GPU is a separate physical entity with vector cores and its own memory space, thereby requiring data to be first transferred to its own memory before any computation can start. The data-transfer requirement leads to significant bottlenecks for certain classes of applications, like large-scale data analytics [13]. Therefore, only those applications which amortize the cost of data-transfers benefit from GPU execution.

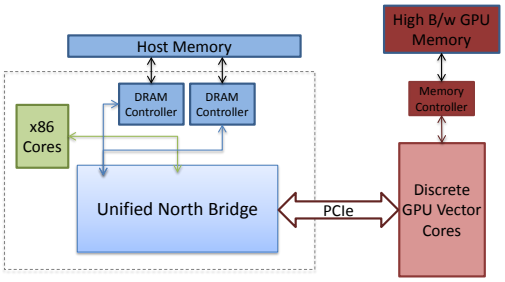


Figure 1: High-level block diagram illustrating a CPU and discrete GPU setup across PCI Express (PCIe).

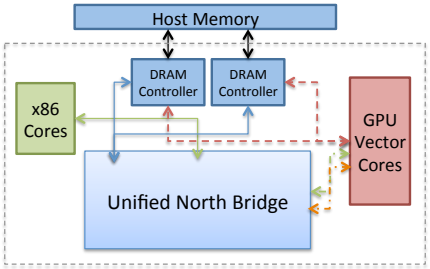


Figure 2: High-level block diagram of an AMD A10-7850K APU.

The current generation of AMD GPUs are made up of one or more compute units (CUs). Each CU consists of 4 lanes of 16 ALUs which results in 64 GPU threads being executed in a single-instruction-multiple-data (SIMD) fashion. CUs also consist of parallel resources like registers and a highly-banked local data store which are shared among the threads executing on that CU. Discrete GPUs have their own instruction set architecture (ISA) that is different from the x86 ISA of CPUs, which has led to different programming models for CPUs and GPUs.

B. Accelerated Processing Unit (APU)

An Accelerated Processing Unit (APU) is AMD's implementation of the emerging heterogeneous system architecture (HSA) [14]. Fundamentally, the APU combines scalar x86 CPU cores and the vector GPU cores on the same physical die. Figure 2 illustrates a high-level block diagram of an AMD A10-7850K APU. The fused CPU and GPU cores as well as an unified memory interface allow APUs to eliminate the data-transfer requirement and also mitigate the data-size limitation which the discrete GPUs suffer from.

Programming the APUs is similar to programming discrete GPUs but with several advantages. Since the APUs do not require data transfers and can directly work with pointers, the host-code, if OpenCL is used, is much simpler without the need for creating separate buffers and staging data. For C++ AMP and OpenACC, the programmers can omit any data-transfer code and rely on compilers to generate code which does not perform any implicit data transfers.

III. Programming Models

This section describes the three programming models that we compare - (i) OpenCL, (ii) OpenACC, and

```
function read-serial-CPU (in[], out[], size) {
   // stream through the 'in' buffer
   for i ∈ size, i += BLOCKSIZE do
      sum = 0.;
      // compute the sum of BLOCKSIZE continuous elements
      for j    ∈ BLOCKSIZE do
         sum += in[i + j];
      end for
      // write computed sum to 'out' buffer
      out[i / BLOCKSIZE] = sum;
   end for
}
```

(a) Read-memory Serial CPU

```
function read-openmp-CPU (in[], out[], size) {
   /* OpenMP directives */
   #pragma omp parallel for

   // stream through the 'in' buffer
   for i ∈ size, i += BLOCKSIZE do
      sum = 0.;
      // compute the sum of BLOCKSIZE continuous elements
      for j    ∈ BLOCKSIZE do
         sum += in[i + j];
      end for
      // write computed sum to 'out' buffer
      out[i / BLOCKSIZE] = sum;
   end for
}
```

(b) Read-memory OpenMP CPU

Figure 3: Pseudocode for serial and OpenMP CPU implementations of the read-memory benchmark.

(iii) C++ AMP. We use a simple `read-memory` benchmark to illustrate the differences in these programming models. The `read-memory` benchmark streams through a region of memory and computes the sum of a block of continuous elements. The block size of 64 is used for our experiments. The computed sum is then written to an output buffer to ensure that the compiler does not optimize out the code. The pseudocode for serial CPU implementation of the `read-memory` benchmark is shown in Figure 3a. The OpenMP implementation on the CPU is straightforward and just adds one line to include the parallel OpenMP `pragma`, as shown in Figure 3b.

A. OpenCL

The OpenCL programming model was a giant leap towards improving the productivity of GPU application developers. OpenCL programs consist of separate host and device components, as shown in Figure 4. The host code runs on the CPU and is used to initialize the OpenCL runtime which involves creating GPU contexts, command queues and compiling code. All GPU memory allocations and data transfers are also done from the host code. When a discrete GPU is used, explicit data transfers to the GPU memory are required. Once the data is staged for execution, a GPU kernel is launched with the required number of threads. Figure 4a shows the host code for the `read-memory` benchmark.

The device code consists of kernels, which are functions that run on the GPU. Each kernel consists of one- to three-dimensional matrix of threads known as workgroups. All the threads in a workgroup are executed on one compute

```
function read-opencl-host (in[], out[], size) {
   // OpenCL boilerplate code to initialize device, context,
   // command queues and compile kernels, etc.
   InitCl();
   // create OpenCL 'cl_mem' buffers
   CreateClBuffer(in_cl[], out_cl[], size);
   // copy data into GPU memory if on discrete GPU
   CopyClDataToGPU(in[], in_cl[], out[], out_cl[], size);
   // set OpenCL kernel arguments
   SetCLKernelArgs(in_cl[], out_cl[], size);
   // compute number of threads to launch on the GPU
   numGPUThreads = size / BLOCKSIZE;
   // launch OpenCL kernel
   LaunchKernel(kernelName, numGPUThreads);
   // copy data back to host memory if on discrete GPU
   CopyClDataToHost(out[], out_cl[], size);
}
```

(a) Read-memory OpenCL - Host CPU

```
function read-opencl-GPU (in[], out[], size) {
   // compute global thread id
   int tid = get_global_id(0);
   int st_idx = tid * BLOCKSIZE;

   sum = 0.;
   // compute the sum of BLOCKSIZE continuous elements
   for j    ∈ BLOCKSIZE do
      sum += in[st_idx + j];
   end for
   // write computed sum to 'out' buffer
   out[tid] = sum;
}
```

(b) Read-memory OpenCL - Device GPU

Figure 4: Pseudocode for host and device OpenCL implementation of the read-memory benchmark.

unit in entirety and can synchronize among each other. The pseudocode for the GPU kernel for `read-memory` benchmark is shown in Figure 4b. From the figure, we first compute the ID for every thread. The `thread_id` is used to find the start-index for that thread in the memory region. Each thread then accesses the `BLOCKSIZE` number of contiguous elements, sums them and writes the output to memory. The GPU kernel is similar to the serial CPU code without the top-level `for` loop.

OpenCL incurs a significant overhead of rewriting applications because of the requirement to create segregated host and device code as well as to explicitly allocate data and stage them in GPU memory. Moreover, the GPU kernels are required to be tuned for a particular architecture in terms of how they access data and take advantages of different micro-architectural features to achieve optimal performance. The combined effects of these inhibit the performance portability of OpenCL. However, the ability to write hand-tuned kernel code usually results in better performance than compiler-generated kernels, as will be shown in Section VI.

B. OpenACC

OpenACC is an accelerator programming interface that provides OpenMP-like compiler directives to annotate code. The programmer can define compute and data regions, which are then offloaded to GPUs for execution. OpenACC substantially boosts programmer productivity as implementing GPU applications only requires annotating the code

```
1  function read-openacc-GPU (in[], out[], size) {
2     // copy data into GPU memory if on discrete GPU
3     CopyAccDataToGPU(in[], out[], size);
4     /* OpenACC directives */
5     #pragma acc kernels loop                \
6     gang(size/BLOCKSIZE) vector(BLOCKSIZE) independent
7
8     // stream through the 'in' buffer
9     for i ∈ size, i += BLOCKSIZE do
10       sum = 0.;
11       // compute the sum of BLOCKSIZE continuous elements
12       for j    ∈ BLOCKSIZE do
13          sum += in[i + j];
14       end for
15       // write computed sum to 'out' buffer
16       out[i / BLOCKSIZE] = sum;
17    end for
18 }
```

Figure 5: Pseudocode for OpenACC implementation of the read-memory benchmark.

```
1  function read-cppamp-GPU (in[], out[], size) {
2     // copy data into GPU memory if on discrete GPU
3     CopyAmpDataToGPU(in[], out[], size);
4     // compute number of threads to launch on the GPU
5     extent<1> numGPUThreads(size / BLOCKSIZE);
6     /* C++ AMP lambda function */
7
8     // stream through the 'in' buffer
9     parallel_for_each(numGPUThreads.tile<tile-size>(),
10       [=] (tiled_index<tile-size> t_idx) restrict (amp) {
11       // compute global thread id
12       int tid = t_idx.global[0];
13       int st_idx = tid * BLOCKSIZE;
14
15       sum = 0.;
16       // compute the sum of BLOCKSIZE continuous elements
17       for j    ∈ BLOCKSIZE do
18          sum += in[st_idx + j];
19       end for
20       // write computed sum to 'out' buffer
21       out[tid] = sum;
22    } end parallel_for_each
23 }
```

Figure 6: Pseudocode for C++ AMP implementation of the read-memory benchmark.

without any major structural changes. OpenACC relies on the compiler to generate low-level GPU programs using the hints provided by the programmer.

The OpenACC implementation of the `read-memory` benchmark modifies the OpenMP CPU implementation, as shown in Figure 5. The `kernels` directive is used to identify the parallel loops which should be executed on the GPU. Once the loops are identified, the number of GPU threads are computed and passed to the `gang` and `vector` clauses. The number of gangs and vectors map to the number of workgroups and threads per workgroup, respectively, in the OpenCL parlance. The parallel loops around the `kernels` directive maps to the low-level kernel functions executed on the GPU. Several other clauses can be passed to the `kernels` directive to optimize the data management schemes chosen by the compiler including transferring data to the GPU memory only if required. OpenACC also provides a `data` directive which is used to decouple data movement to GPU memory from compute regions and is particularly useful on discrete GPUs.

From the pseudocode for OpenMP and OpenACC (Figures 3b and 5), we note that both of them lead to similar application code barring the code for data movement required for OpenACC. Therefore, OpenACC makes it significantly easier to accelerate existing applications on GPUs. OpenACC also generates performance-portable code because the compiler can identify the underlying GPU architecture and generate appropriate code for that particular architecture. However, OpenACC lacks the flexibility provided by OpenCL to develop hand-tuned kernels. Several features like fine-grained synchronization primitives and the ability to explicitly use local data store on the GPU are also missing in OpenACC, which adversely affect performance.

C. C++ AMP

C++ AMP is an open standard which was proposed as a productivity focussed alternative to low-level GPU programming models like OpenCL. C++ AMP is a combination of library and extensions to the C++-11 standard. C++ AMP requires similar programming efforts as OpenACC for developing GPU applications but supports the rich-set of object-oriented features like templates, abstract data types and the standard template library (STL).

The C++ AMP implementation of the `read-memory` benchmark is shown in Figure 6. Once the data is transferred to GPU memory, as required, an `extent` is created to define the dimensions and the number of GPU threads. The code is annotated using a `parallel_for_each` function template, which is a lambda function that is used to identify the region of code to be executed on the GPU. The code encompassed within the `parallel_for_each` construct is almost identical to the GPU kernel for OpenCL, shown in Figure 4b. An `extent` of threads is divided into tiles which map to the OpenCL workgroups. C++ AMP provides APIs to query the IDs for tiles and threads which are used to compute the working set for each thread.

C++ AMP bridges the gap between OpenCL and OpenACC by providing the flexibility found in OpenCL along with the ease of programming of OpenACC. C++ AMP provides sufficient control to the programmer to optimize data management and execution while requiring minuscule code changes. C++ AMP also supports fine-grained synchronization primitives and allows the use of local data store by introducing a `tiled_static` storage class, which were missing in OpenACC.

IV. Proxy Applications

This section presents an overview of the four proxy applications that are used to compare the programming models. Table I shows the application characteristics and Figure 7 demonstrates how their performance scales with memory and core frequencies on a GPU, thereby providing an insight into the application's compute and bandwidth requirements.

Our test-suite consists of applications belonging to different domains in HPC and possess inherently different memory and compute and bandwidth characteristics.

For an explanation and verification of our methodology, we show the performance of `read-benchmark` at various core and memory frequencies in Figure 7a. `Read-benchmark` is a memory intensive application, as described in Section III and hence, its performance scales with the increase in memory frequency; the best performance is achieved at 1250 MHz. The change in core frequency does not affect performance at lower memory frequencies. However, at higher memory frequencies, performance scales with the core frequency to an extent until the L2 cache starts to become optimally used. At lower core frequencies, not enough memory requests are generated to efficiently utilize the L2 cache shared by the GPU compute units.

A. LULESH

LULESH is a shock hydrodynamics proxy application that solves the spherical Sedov blast problem using Lagrange hydrodynamics. LULESH is an iterative application and proceeds as follows: initialize spatial domain, advance node quantities, advance element quantities, and update time constraints. Advancing the node quantities is the most computationally intensive part of the simulation.

LULESH contains a large number of parallel loops resulting in 28 different kernels which makes it amenable for parallelization on both APUs and discrete GPUs. LULESH also portrays good data locality as shown by the low miss rate in the last level cache in Table I. Figure 7b demonstrates the performance of LULESH at varying core and memory frequencies. From the figure, it can be seen that LULESH is a balanced application; its performance scales with both memory and core frequencies.

B. CoMD

CoMD is a molecular dynamics proxy application which performs atomic-scale simulation by solving the Newton's laws between particles. In CoMD, every particle interacts with all other particles within a set cutoff distance to simulate the deformation of materials in extreme environments. CoMD is an iterative application which proceeds by computing forces between the particles followed by an update to velocities and positions. Computation of forces accounts for more than 90% of total execution time and hence, is the critical part of the application.

Computing the forces between particles is a data-parallel operation and hence, suitable for GPU acceleration. This application does not demonstrate high data-reuse as shown by the relatively high last-level cache-miss rate in Table I. CoMD is a compute bound application as shown in Figure 7c. The performance of CoMD scales almost linearly with the increase in core frequency. Whereas, the change in memory frequency does not affect its performance.

Table I: Characteristics of Proxy Applications

Application	Last-Level Cache Miss Rate	Instructions Per Cycle (IPC)	Number of Kernels	Boundedness
LULESH	11%	0.65	28	Balanced
CoMD	26%	0.69	3 (LJ)	Compute
XSBench	53%	0.14	1	Compute
miniFE	39%	0.88	3	Memory
Command Line Parameters				
LULESH	`./LULESH -s 100 -i 100`			
CoMD	`./CoMD -x 60 -y 60 -z 60`			
XSBench	`./XSBench -s small`			
miniFE	`./miniFE -nx 100 -ny 100 -nz 100`			

C. XSBench

XSBench computes the intensive macroscopic neutron cross-section lookup and particle transport simulation. XSBench works with the Hoogenboom-Martin reactor material properties data-set and creates a random set of energy and material pairs representing particle or material interactions. The pairs are then used to lookup cross-section probability.

XSBench manifests poor data-locality as shown by the high last-level cache misses in Table I. The data-locality is so appalling that any increase in memory bandwidth does not affect application performance (shown in Figure 7d) which also results in poor instructions per cycle (IPC). Similar to CoMD, XSBench is a compute bound application; there is a steady increase in performance with the increase in core frequency, except at extremely low memory frequencies at which the memory requests are not optimally serviced.

D. miniFE

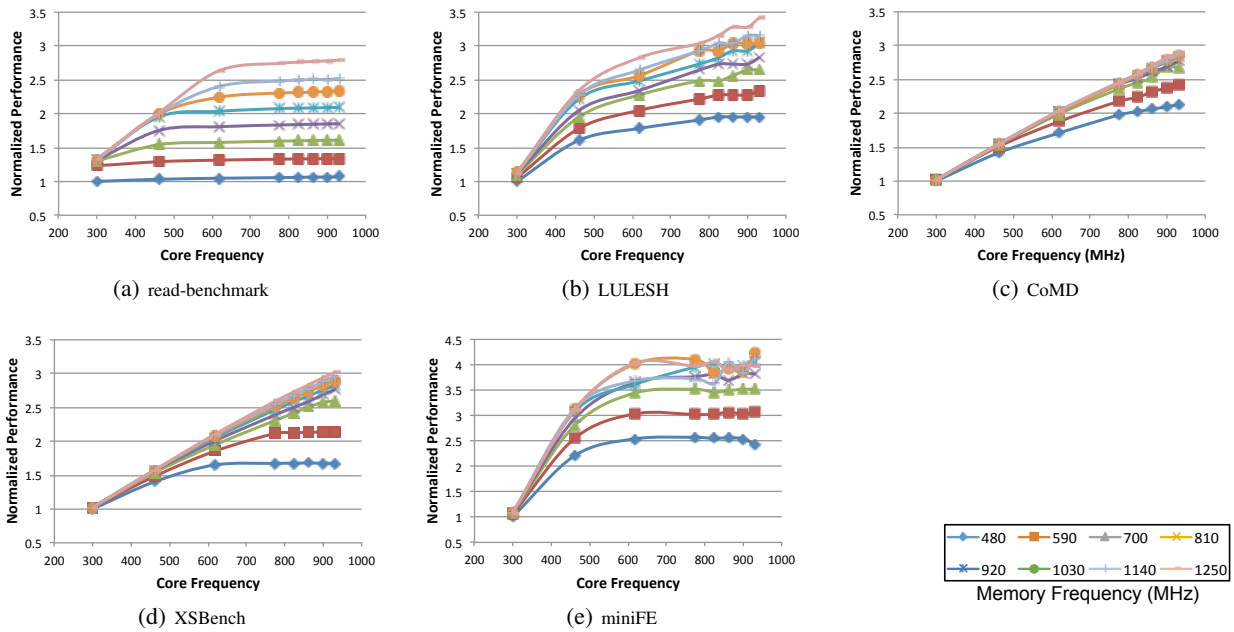
miniFE is a finite element proxy application that solves a sparse linear-system using a simple un-preconditioned conjugate-gradient (CG) algorithm. Once the element-operators are generated and assembled into a sparse matrix and vector, miniFE executes the following kernels until the solution converges: sparse matrix-vector multiplication (SpMV), axpy and dot product. Among the different kernels, SpMV is the most computationally intensive. We use the CSR-Adaptive algorithm to compute SpMV [15].

Figure 7e illustrates the performance of miniFE at varying core and memory frequencies. miniFE portrays initial signs of compute boundedness but quickly establishes itself as a memory-bandwidth bound application once enough compute resources are present to saturate the L2 cache on the GPU. miniFE also demonstrates high IPC, as shown in Table I, thereby validating that the higher core frequency correlates to higher performance.

V. Experimental Setup

We used the AMD A10-7850K APU and the AMD Radeon™R9 280X discrete GPU to compare the programming models. The discrete GPU was paired with an AMD A10-7850K APU as the host processor[1]. The hardware specifications of these platforms is shown in Table II.

[1]the integrated GPU on APU was not used in APU+discrete GPU setup



(a) read-benchmark (b) LULESH (c) CoMD

(d) XSBench (e) miniFE

Figure 7: Normalized performance of proxy applications with varying core and memory frequencies on a GPU using OpenCL.

The software environment consists of Ubuntu 14.04 with Linux kernel 2.17. Different programming models used different compilers and runtimes which are presented in Table III. C++ AMP required different runtimes on the APU and the discrete GPU. We used the HSA software stack v1.0 on the APU [16] and the AMD Catalyst™ software stack v14.6 on the discrete GPU.

Table II: Hardware Specification of Accelerators

Name	AMD Radeon™ R9 280X	AMD A10-7850K
Stream Processors	2048	768
Compute Units	32	12 (4 CPU + 8 GPU)
Core Clock Frequency	925 MHZ	720 MHZ
Memory Bus type	GDDR5	DDR3
Device Memory	3 GB	2 GB
Local Memory	64 KB	64 KB
Peak Bandwidth	258 GB/s	33 GB/s
Peak Single Precision Perf.	3800 GFLOPS	738 GFLOPS
Host Processor	AMD A10-7850K	AMD A10-7850K
CPU frequency	3.7 GHZ	3.7 GHZ
System memory	32 GB	32 GB

Table III: Compilers Used for Programming Models

Programming Model	Compiler
OpenCL	AMD Catalyst™ driver v14.6
C++ AMP	CLAMP v0.6.0 [9]
OpenACC	PGI v14.10 with AMD Catalyst driver v14.6

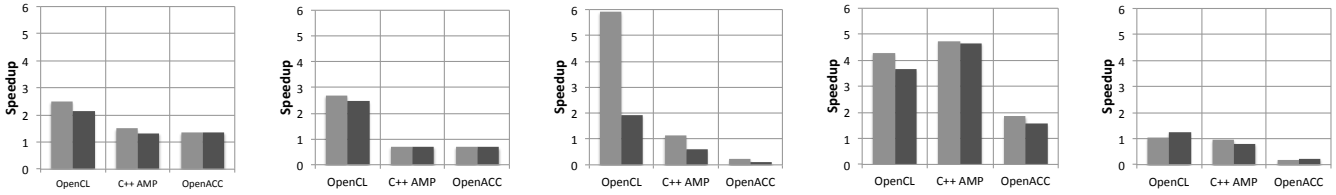
VI. Results and Discussion

This section presents the analysis on performance, productivity and the ease of optimizations offered by various programming models.

A. Performance

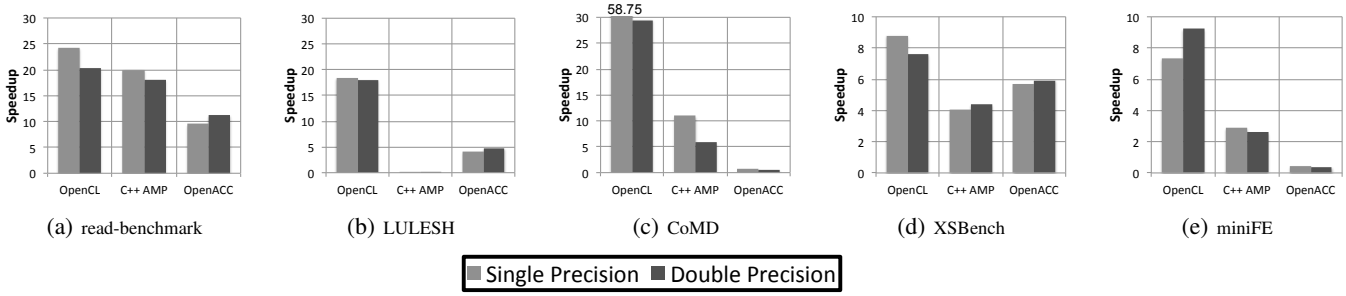
Figures 8 and 9 demonstrate the performance of proxy applications implemented and hand-tuned in three programming models on the APU and the discrete GPU, respectively, compared to 4-core OpenMP implementation. `Read-benchmark` being an extremely simple kernel is an apt choice to understand the quality of code generation by the compilers of emerging programming models compared to OpenCL. Figures 8a and 9a depict the comparison of kernel execution times in `read-benchmark` on the APU and discrete GPU, respectively. The data-transfer times, if any, were left out in order to only focus on the low-level code that executes on the GPU. From the figures, it can be noted that OpenCL performs the best and is better than C++ AMP and OpenACC by 1.3× and 2×, respectively. This shows that C++ AMP generates relatively better low-level code than OpenACC when only the GPU kernel performance is considered, without any host-side overhead. The difference in speedups between APU and discrete GPU compared to the OpenMP CPU implementation is due to an order of magnitude more bandwidth available on the discrete GPU which attests to the performance portability of these programming models.

Figures 8b and 9b shows the performance comparison of LULESH. From the figures, OpenCL performed the best on both the APU and the discrete GPU. Both C++ AMP and OpenACC achieved similar performance on the APU. However, on the discrete GPU, C++ AMP performed poorly because we were able to implement only 27 out of the 28 kernels on the GPU due to a compiler bug; one kernel was



(a) read-benchmark (b) LULESH (c) CoMD (d) XSBench (e) miniFE

Figure 8: Performance comparison of programming models on AMD A10-7850K. Baseline: 4-core OpenMP CPU implementation.



(a) read-benchmark (b) LULESH (c) CoMD (d) XSBench (e) miniFE

Figure 9: Performance comparison of programming models on AMD Radeon R9 280X. Baseline: 4-core OpenMP CPU implementation.

implemented on the CPU which led to data-transfer overhead. The performance of emerging programming models can be improved if the programmer is allowed to manually control the data-transfers rather than relying on the compiler.

Figures 8c and 9c shows the performance comparison of CoMD. Overall, OpenACC demonstrated the worst performance on both architectures because of the compiler's inability to expose vector-parallelism in the accelerator code. C++ AMP performed better than OpenACC but was outperformed by OpenCL. Since CoMD is a compute bound application, its performance scaled up when running on the discrete GPU for both OpenCL and C++ AMP, thereby demonstrating performance portability. The significant performance difference between single- and double-precision across all programming models is due to the lower double-precision compute throughput which is $1/16th$ on the APU and $1/4th$ on the discrete GPU.

XSBench uses a configurable lookup-table size which was set to 240 MB for our experiments because of the problem-size limitation imposed by the discrete GPUs; the next step in the lookup-table size was 5 GB. Moving this lookup-table to the GPU memory accounts for a significant amount of total execution time. Figures 8d and 9d shows the performance comparison of XSBench. C++ AMP resulted in the best performance on the APU. However, on the discrete GPU, the OpenCL implementation performed the best with an improvement of up to $2\times$ over the other programming models. C++ AMP resulted in poor performance on the discrete GPU than the APU which is atypical for a compute bound application. This demonstrates that on architectures which do not impose data-transfer requirements, the emerging programming models generate better low-level code and can provide promising performance improvements.

miniFE is a memory bound application which implies that high bandwidth memory will impact the performance the most. This is attested by Figures 8e and 9e which show the performance comparison of miniFE. On the APU, OpenMP and all other programming models are limited to the same host-memory bandwidth due to which OpenCL and C++ AMP just match OpenMP's performance. Whereas, the OpenACC implementation results in a slowdown. Both OpenCL and C++ AMP implementations scale with improved memory bandwidth on the discrete GPU, thereby demonstrating performance portability. OpenACC performs the slowest because specialized sparse matrix operations cannot be easily expressed at a high level, and the compiler is unable to recognize and take advantage of the complicated memory access patterns.

Observations: Our experiments illustrate that the emerging programming models match the performance and at times even outperform a low-level programming model like OpenCL. We summarize our observations as following.

- C++ AMP outperformed OpenACC in most cases.
- OpenCL was best for compute-bound applications due to suboptimal vectorization by other compilers.
- C++ AMP performed the best on the APU for applications which incurred large data-transfers cost.
- The emerging programming models are slower than OpenCL on discrete GPUs because compiler-generated code for data-transfers performs worse than explicit programmer-written code.

- OpenCL requires hand-tuned code for each architecture for performance portability. Whereas, the emerging programming models do *not* require any modification to the code, as shown by the performance improvement in all cases when moved from APU to discrete GPU.

B. Productivity

Table IV shows the number of lines of code that were added to implement the applications in various programming models starting from the serial CPU implementation. The number of lines were measured using the SLOCCount tool which does not consider the comments in the code [17]. The source lines of code provide an intuition regarding the productivity associated with each programming model.

The `read-benchmark` shows that OpenCL requires $4\times$ more lines of code than both C++ AMP and OpenACC. OpenCL implementations of the proxy applications also resulted in an order of magnitude more lines of code. The only exception is LULESH, which required almost similar number of lines of code across all the programming models. The OpenCL implementation of XSBench substantially deteriorated productivity by requiring more lines of code modified than were present in the original serial CPU implementation. Among all the programming models examined, OpenACC required minimal changes to the serial code. C++ AMP came a close second by requiring 15% more changes on an average than OpenACC.

Merely considering the source lines of code alone does not provide a legitimate estimate of the productivity associated with programming models. Therefore, we compute productivity as a function of speedup and the ratio of number of lines across all the three programming models to deduce which provides the "biggest bang for buck". The speedup is computed as the ratio of the execution times of OpenMP implementation on the CPU and each programming model.

$$productivity_{prog\_model} = \frac{(time_{OMP}/time_{prog\_model})}{(lines_{prog\_model}/lines_{OMP})} \quad (1)$$

Productivity, as defined above, is depicted in Figure 10 for double-precision implementation of the proxy applications. We chose double-precision because that is most relevant from a scientific application standpoint in HPC. The emerging programming models are more productive than OpenCL on multiple occasions on the APU, as shown in Figure 10a. This is because the emerging programming models provide comparable performance to OpenCL albeit requiring considerably less lines of code. C++ AMP results in best productivity on average and is $3\times$ more productive for XSBench on the APU.

Table IV: Source Lines of Code Changed Starting from the CPU Serial Implementation [17]

Application	OpenMP	OpenCL	C++ AMP	OpenACC
read-benchmark	3	181	42	40
LULESH	107	1357	1087	1276
CoMD	23	3716	188	183
XSBench	13	1468	83	113
miniFE	18	2869	260	43

OpenCL provided substantially better performance on the discrete GPU and hence, it is worthwhile to undergo the arduous programming effort and still achieve better productivity with OpenCL, as shown in Figure 10b.

If learning curve is to be considered, OpenACC was the most straight-forward given its resemblance to OpenMP.

C. Ease of Optimizations

Figure 11 illustrates the features available in the programming models to assist in tuning application-performance either by manual intervention or by providing hints to the compiler. OpenCL provides the maximum flexibility to tune GPU code by allowing the programmers to write hand-tuned kernels. OpenCL allows for the use of the local-data-store (LDS) as well as unrolling loops to improve kernel performance. The programmer can also reduce loop invariant code motion by manually moving code and transform data-structures to perform efficient memory accesses.

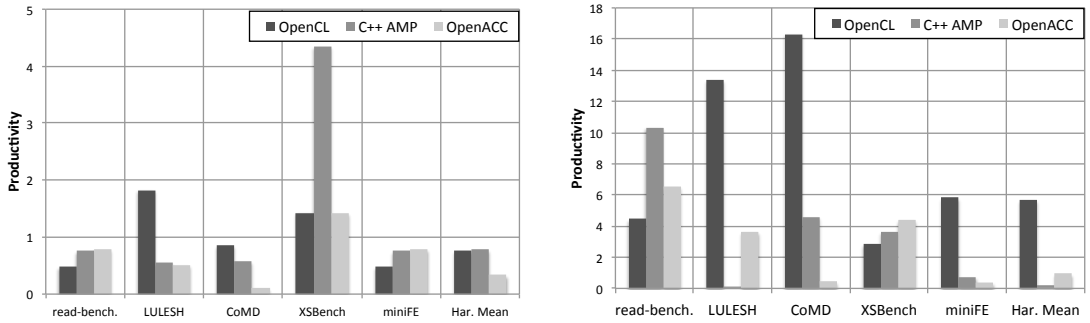
Amongst the emerging programming models, OpenACC is the least flexible and does not provide access to advanced tuning features like the use of LDS. OpenACC also proved challenging in terms of mapping the parallelism to appropriately use GPU vector cores. OpenACC also does not provide synchronization primitives which inhibits implementing complex applications.

C++ AMP provides the use of tiling which leads to improved vectorization and better performance. For example, exposing parallelism in the form of tiles improved the performance of CoMD by almost $3\times$. C++ AMP allows programmers to use LDS via its `tile_static` storage class and also provides primitives for synchronization. C++ AMP lacks performance features like the explicit unrolling of loops and reducing code motion.

VII. Heterogeneous Compute: The future of programming heterogeneous computing systems

The emerging programming models improve programmer productivity but lack a rich-set of powerful features such as those found in OpenCL for performant programming. OpenCL provides supreme performance but requires rewriting application-code which can be an engineering hurdle. To provide a "best of both worlds" scenario, AMD is developing *Heterogeneous Compute (HC)* - a simple and powerful programming model for heterogeneous systems.

HC builds on the advantages of OpenCL and C++ AMP. In particular, HC provides a single-source C++ development environment for both host and kernel code, thereby eliminating the productivity concerns of OpenCL. The requirement to rely on the compiler for data-transfers was the single biggest reason for poor performance with C++ AMP and



(a) AMD A10-7850K APU (b) AMD Radeon R9 280X discrete GPU

Figure 10: Productivity (Eq. 1) comparison of programming models on AMD A10-7850K and AMD Radeon R9 280X.

	Vectorization	Use of Local Data Store (LDS)	Fine-grained Synchronization	Explicit Loop Unrolling	Reducing Code Motion
OpenCL	✓	✓	✓	✓	✓
OpenACC	✓	✗	✗	✗	✗
C++ AMP	✓	✓	✓	✗	✗

Figure 11: Optimizations allowed by each programming model.

OpenACC. HC improves upon that and allows the programmer to explicitly manage data-transfers including asynchronous kernel launches which help in overlapping kernel execution with data-transfers, resulting in further speedup. The programmer is also able to directly use raw pointers to data in kernel-code without the need for wrapping them in a `CL::Buffer` or `array/array_view` like in OpenCL and C++ AMP, respectively. HC supports platform atomics for global synchronization, offline kernel compilation and advanced C++ language features like virtual functions and abstract datatypes.

VIII. Related Work

Several academic papers have been written to compare OpenACC and the traditional GPU programming models. Karlin et al. compare several traditional and emerging parallel programming models using the LULESH proxy application [11]. They note that the emerging programming models offer improved productivity and performance-portability. Hoshino et al. compare CUDA and OpenACC using several micro-benchmarks and one computational fluid dynamics (CFD) application [18]. Their experiments reveal several shortcomings of OpenACC which inhibit the optimal use of GPU resources. In particular, they note that the inability to use local data store (LDS) on the GPU can serverly limit performance. Herdman et al. compare OpenACC with both OpenCL and CUDA from the perspective of accelerating the CloverLeaf mini-application [19]. They illustrate that OpenACC vastly improves programmer-productivity and provides better performance than albeit unoptimized CUDA and OpenCL implementations.

Wang et al. studied the performance portability of OpenACC on discrete GPUs and many-core processors [20]. This work concluded that the portability of OpenACC is related to the arithmetic intensity in the application and there exists a performance gap by executing the same code on different architectures. Ghosh et al. compared different OpenACC implementations from different vendors [21].

Previous academic research has used OpenACC to accelerate a plethora of applications. Wienke et al. were among the first to evaluate the programmability and productivity of OpenACC using a set of real-world applications [22]. The authors noted that though OpenACC improves productivity, it achieves sub-optimal performance. Hart et al. present their experience in porting and scaling OpenACC applications on GPU-accelerated supercomputers [23]. Levesque et al. implement the S3D proxy application using OpenACC [24].

Not many works have explored the use of C++ AMP to accelerate GPU applications. One previous research sped up micromagnetic simulations using C++ AMP [25]. There have also been concerted efforts to develop compiler frameworks that can accelerate C++ applications on GPUs. Barik et al. present the Concord compiler which enables the efficient mapping of irregular C++ applications on GPUs [26]

As noted above, there does not exist previous research which compares and contrasts the different programming models using more than one real-world application. We believe that using just one application may result in a biased evaluation of a programming model. On the other hand, our research uses four different proxy applications and are among the first to comprehensively use and evaluate C++ AMP for accelerating GPU applications.

IX. Conclusions

Programming models like OpenCL and CUDA have considerably ameliorated the process of using GPUs for general-purpose computation. However, these programming models present a large engineering hurdle due to their prerequisite of rewriting existing CPU applications. Moreover, the resulting accelerated application does not manifest performance portability. Emerging programming models like C++ AMP

and OpenACC hold the promise of mitigating the challenges of GPU programming by making use of advanced language features like pragmas and lambda functions. Both C++ AMP and OpenACC rely on compilers to generate low-level code, thereby offering improved productivity and portability from an application standpoint.

In this paper, we present the comparative study of a traditional programming model like OpenCL and two emerging programming models, i.e., OpenACC and C++ AMP. To the best of our knowledge, we are the *first* to compare C++ AMP with other GPU programming models. Our evaluations encompass the studies on performance, productivity and ease of optimizations across various programming models using four diverse proxy applications on two different architectures, a discrete GPU and an Accelerated Processing Unit (APU). Our experiments illustrate that the emerging programming models greatly improve productivity and also match performance of OpenCL on an APU. However, on a discrete GPU, OpenCL performs substantially better than both OpenACC and C++ AMP because their compilers do not optimally manage the data-transfers required on the discrete GPU. Amongst the two emerging programming models, C++ AMP looks more promising than OpenACC in all three of our evaluation criteria.

ACKNOWLEDGMENTS

We would like to thank Jonathan Gallmeier for providing constructive feedback.

REFERENCES

[1] D. Bouvier and B. Sander, "Applying AMD's Kaveri APU for Heterogeneous Computing." in *Hot Chips: A Symposium on High Performance Chips (HC26)*, 2014.

[2] L. Codrescu, "Qualcomm Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications." in *Hot Chips: A Symposium on High Performance Chips*, 2013.

[3] "The Top500 Supercomputer Sites," http://www.top500.org.

[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[5] M. Daga, W. Feng, and T. Scogland, "Towards Accelerating Molecular Modeling via MultiScale Approximation on a GPU," in *Proceedings of the 1st IEEE International Conference on Computational Advances in Bio and medical Sciences*, 2011.

[6] A. Munshi, "The OpenCL Specification," 2012, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[7] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2008, pp. 195–204.

[8] M. Daga, T. Scogland, and W. Feng, "Architecture-aware mapping and optimization on a 1600-core gpu," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec. 2011.

[9] "Clamp : An open source C++ compiler for heterogeneous devices," https://bitbucket.org/multicoreware/cppamp-driver-ng/wiki/Home.

[10] "OpenACC: Directives for Accelerators," http://www.openacc.org/.

[11] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 919–932.

[12] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. T. Pedretti, "Toward an evolutionary task parallel integrated mpi + x programming model," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '15. New York, NY, USA: ACM, 2015, pp. 30–39.

[13] M. Daga, M. Nutter, and M. Meswani, "Efficient breadth-first search on a heterogeneous processor," in *Big Data (Big Data), 2014 IEEE International Conference on*, Oct 2014, pp. 373–382.

[14] "The HSA Foundation," 2012, http://hsafoundation.com/.

[15] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 769–780.

[16] "HSA Specification v1.0," http://www.hsafoundation.com/hsa-foundation-launches-new-era-of-pervasive-energy-efficient-computing-with-hsa-1-0-specification-release/.

[17] D. A. Wheeler, "SLOCCount," http://www.dwheeler.com/sloccount/.

[18] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 136–143.

[19] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, "Accelerating hydrocodes with openacc, opecl and cuda," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 465–471.

[20] Y. Wang, Q. Qin, S. C. W. SEE, and J. Lin, "Performance portability evaluation for openacc on intel knights corner and nvidia kepler," *HPC China*, 2013.

[21] S. Ghosh, T. Liao, H. Calandra, and B. Chapman, "Experiences with openmp, pgi, hmpp and openacc directives on iso/tti kernels," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 691–700.

[22] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870.

[23] A. Hart, R. Ansaloni, and A. Gray, "Porting and scaling openacc applications on massively-parallel, gpu-accelerated supercomputers," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 5–16, 2012.

[24] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing s3d into an exascale application using openacc: An approach for moving to multi-petaflops and beyond," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 15:1–15:11.

[25] R. Zhu, "Speedup of micromagnetic simulations with C++ AMP on graphics processing units," *CoRR*, vol. abs/1406.7459, 2014.

[26] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai, "Efficient mapping of irregular c++ applications to integrated gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 33:33–33:43.