MINIME-GPU: Multicore Benchmark Synthesizer for GPUs

Etem Deniz and Alper Sen, Bogazici University

34

We introduce MINIME-GPU, a novel automated benchmark synthesis framework for graphics processing units (GPUs) that serves to speed up architectural simulation of modern GPU architectures. Our framework captures important characteristics of original GPU applications and generates synthetic GPU benchmarks using the Open Computing Language (OpenCL) library from those applications. To the best of our knowledge, this is the first time synthetic OpenCL benchmarks for GPUs are generated from existing applications. We use several characteristics, including instruction throughput, compute unit occupancy, and memory efficiency, to compare the similarity of original applications and their corresponding synthetic benchmarks. The experimental results show that our synthetic benchmark generation framework is capable of generating synthetic benchmarks that have similar characteristics with the original applications from which they are generated. On average, the similarity (accuracy) is 96% and the speedup is $541\times$. In addition, our synthetic benchmarks use the OpenCL library, which allows us to obtain portable human readable benchmarks as opposed to using assembly-level code, and they are faster and smaller than the original applications from which they are generated. We experimentally validated that our synthetic benchmarks preserve the characteristics of the original applications across different architectures.

Categories and Subject Descriptors: B.1.2 [Computer Systems Organization]: Multiple Instruction, Multiple Data; E.2.1 [Language Types]: Computing Methodologies—*Parallel programming languages*

General Terms: Performance, Experimentation, Parallel Programming Languages, Heterogeneous Architectures

Additional Key Words and Phrases: GPU, OpenCL, synthetic benchmarks, benchmark characterization, benchmark generation

ACM Reference Format:

Etem Deniz and Alper Sen. 2015. MINIME-GPU: Multicore benchmark synthesizer for GPUs. ACM Trans. Archit. Code Optim. 12, 4, Article 34 (November 2015), 25 pages.
DOI: <http://dx.doi.org/10.1145/2818693>

1. INTRODUCTION

Graphics processing units (GPUs) have become a crucial platform for general-purpose computing because of their high performance on data-parallel applications. GPUs present high performance, but they require optimizations to achieve this high performance. Hence, in early design exploration of GPUs, it is important to have benchmarks that have similar performance characteristics with the applications that will run on the GPU. However, existing benchmarks and customer applications are large,

This is a new article, not an extension of a conference paper.

The research was supported in part by Semiconductor Research Corporation under task 2082.001, Bogazici University Research Fund with grant 7223, and the Turkish Academy of Sciences.

Authors' address: E. Deniz and A. Sen, Department of Computer Engineering, Bogazici University, Istanbul, 34342, Turkey; emails: {etem.deniz, alper.sen}@boun.edu.tr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3566/2015/11-ART34 \$15.00

DOI: <http://dx.doi.org/10.1145/2818693>

and it is difficult to run these benchmarks and applications on early performance models and simulators. Furthermore, customers can hesitate to share their proprietary applications.

To meet the preceding needs, we present a novel synthetic benchmark generation approach for GPUs. Synthetic benchmarks are simplified artificial applications that can represent real-life applications. These benchmarks can be derived from existing applications, or they can be developed by varying application characteristics. Our approach is capable of generating synthetic benchmarks that are small and fast, and they accurately mimic the characteristics of the original applications from which they are generated. They can be used for early performance studies of GPUs in both actual hardware and simulation. As described in Yu et al. [2015], CPU simulation acceleration techniques such as sampling and statistical simulation cannot be easily applied to GPUs, as each thread in a GPU application executes a small number of instructions compared to CPU applications. Our approach helps developers and researchers focus on analyzing the synthetic benchmark results by hiding the difficulties of the benchmark development from them.

Our fully automated synthetic benchmark generation approach is composed of two steps: (1) characterizing a GPU application to capture its inherent characteristics and modeling the captured application characteristics into an abstract benchmark model and (2) generating a synthetic GPU benchmark using the abstract benchmark model. We generate our synthetic benchmarks using Open Computing Language (OpenCL) [Khronos OpenCL Working Group 2012], which is a framework for developing applications that run across heterogeneous platforms consisting of CPUs, GPUs, and DSPs.

During benchmark characterization, unique behaviors of an application of interest are captured as a set of quantifiable abstract characteristics. Accurate synthetic benchmark generation requires a sufficient number of characteristics to capture and model the behaviors of an existing application. The characteristics that we use to capture the behavior of an original application are instruction throughput, compute unit occupancy, computation-to-memory access ratio (CMAR), memory instruction mix, and memory efficiency. In other words, we speed up GPU architectural simulation by generating synthetic benchmarks from existing benchmarks that mimic these characteristics. These characteristics are widely used in the literature [Che and Skadron 2014; Seo et al. 2011; Bakhoda et al. 2009; Kerr et al. 2009; Goswami et al. 2010]. We also apply principal component analysis (PCA) to these characteristics to find the most important characteristics. Once we characterize and model the original application as an abstract benchmark model, we generate a synthetic benchmark from this model. The synthetic benchmark consists of host (CPU) and compute device (GPU) code, and it is generated in C++ using the OpenCL library. Our synthetic benchmarks preserve all of the characteristics captured from the original application, and they can be executed either on a simulator or a target platform.

MINIME-GPU is our fully automated benchmark synthesis framework for GPUs, and we experimentally validated the efficiency of our approach using MINIME-GPU. This is an extension of our earlier tool MINIME, which targets synthetic CPU benchmark generation [Deniz et al. 2015]. During the experiments, we used the Multi2Sim [2014] simulator to collect the characteristics of GPU applications. We generated synthetic benchmarks from the AMD benchmark suite [AMD 2014a] for AMD Southern Islands (AMD SI) GPUs [AMD 2014b], which are available with Multi2Sim distribution [Multi2Sim 2014]. The experimental results showed that our synthetic benchmarks mimic the characteristics of the original applications from which they are derived, where the average similarity is 96% and average speedup is $541\times$. We also experimentally validate that our synthetic benchmarks mimic the behaviors of the originals across different architectures on both the Multi2Sim simulator and on real GPU hardware.

Furthermore, they are human readable and obfuscated with respect to the original applications.

This article makes the following contributions:

- A synthetic benchmark generation framework is proposed and implemented to generate synthetic OpenCL benchmarks for GPUs from a given GPU application.
- We use principal component data analysis methodology to identify critical GPU application characteristics.
- Our synthetic GPU benchmarks are portable, human readable, smaller, and faster than the original applications from which they are generated.
- Our synthetic benchmarks do not compromise the proprietary nature of the original applications, as one cannot obtain the original application from our synthetic benchmarks by reverse engineering.

2. BACKGROUND

In this section, we provide a summary of the background information required to understand benchmark characterization and synthesis for GPUs carried out in Section 4 and 5.

2.1. OpenCL Programming Model

OpenCL is an open standard based upon C for portable parallel applications across heterogeneous platforms including CPUs, GPUs, and DSPs [Khronos OpenCL Working Group 2012]. OpenCL provides an API to develop parallel applications using task- and data-based parallelism. An OpenCL platform has a host connected to a number of compute devices. Note that the host is the CPU that submits works to the compute devices. A compute device consists of one or more compute units (cores), where a compute unit is composed of a set of processing elements.

OpenCL (programming model) allows the development of OpenCL programs, which requires developing codes for the host side (host program) and the device side (kernel program). The kernel program has parallel functions called *kernels*, which are the basic units of executable code. The host is developed in C/C++ using OpenCL API (library), and it manages the device to execute the kernel program. When a host program invokes a kernel, an index space called *N-Dimensional Range* (NDRange), which can be arranged into one, two, or three dimensions, is defined. An NDRange consists of work-items, and several work-items are organized into a work-group. Note that each work-item executes the same kernel (usually on different data). OpenCL provides the notion of dimension to define the number of work items, where global dimensions define the range of computation (whole computation space) and local dimensions define the size of the work-groups. OpenCL memory model defines a three-level memory hierarchy for a compute device. All work-items can access a global memory, work-items in a single work-group share a local memory, and every work-item has a private memory (registers) that is not accessible from other work-items. Note that synchronization is allowed between work-items within a work-group, and there is no synchronization between work-groups.

2.2. Target GPU Architecture

In this work, we target the most recent AMD SI GPUs. This is because the current version of Multi2Sim simulator does not fully support other GPU models (AMD Evergreen GPU and NVIDIA Fermi GPU, as we will further address in Section 6. However, in principle, our approach can be applied to other GPUs as well. In the OpenCL terminology, we denote AMD SI GPU as the compute device, SIMD units as compute units, and SIMD lanes as processing elements. AMD SI has scalar and vector arithmetic-logic

Fig. 1. MINIME-GPU, a multicore benchmark synthesizer for GPUs.

units, and it has multiple levels of memory. An ultrathreaded dispatcher in AMD SI schedules work-groups that are pending on the running ND-Range. In addition, wave-fronts that consist of 64 work-items are created to efficiently run the same code in an SIMD fashion.

3. HIGH-LEVEL FRAMEWORK

Figure 1 shows MINIME-GPU, our fully automated high-level benchmark synthesis framework for GPUs. Our framework contains two main modules: *benchmark characterizer* and *benchmark synthesizer*. Benchmark characterizer captures the characteristics of a GPU application and generates an abstract GPU benchmark model from these characteristics. Benchmark synthesizer, first, generates a candidate synthetic GPU benchmark from the generated abstract benchmark model. Then, benchmark synthesizer iteratively calculates the similarity between the original application and synthetic benchmark and improves the similarity by generating new synthetic benchmarks. This approach is similar to the MINIME framework proposed by us [Deniz et al. 2015] for multicore CPU benchmark synthesis. However, this work differs from the earlier work by targeting GPU architectures, using GPU specific characteristics and generating synthetic benchmarks using the OpenCL library. We further discuss this in Section 8. Note that in this work, we generate synthetic benchmarks that only preserve the characteristics of kernel programs—that is, we do not mimic the characteristics of host programs as in Yu et al. [2015]. This is because we target the GPU applications in which the whole solution of the problem is implemented on the compute device side. Additionally, we are not aware of a benchmark suite in which some parts of a problem are solved on the host side and the other parts are solved on the compute device side. In any case, for such CPU/GPU benchmarks, we can use MINIME (CPU) to generate synthetic benchmarks for host programs and MINIME-GPU to generate synthetic GPU benchmarks for kernel programs.

4. BENCHMARK CHARACTERIZATION

When generating a synthetic benchmark, the efficacy of benchmark characterization to capture the behaviors of an original application is crucial. This is because only the behaviors captured from the original application can be preserved in the corresponding

Table I. GPU Benchmark Characteristics

Group	Characteristics
Instruction throughput	Instructions per cycle
Computation-to-memory access	Computation-to-memory access ratio
Dynamic memory instruction mix	Private memory ratio
	Local memory ratio
	Global memory ratio
Memory efficiency	Memory coalescing
	Hit ratio
Compute unit occupancy	Work-items per work-group
	Registers per work-item
	Local memory per work-group
	Number of (in-flight) wavefronts

synthetic benchmark. Hence, we designed a benchmark model to cover the most crucial characteristics that capture the major behaviors of a GPU application. In our model, we use instruction throughput, CMAR, memory instruction mix, memory efficiency, and compute unit occupancy characteristics. Note that we also use these characteristics to determine program similarity, as we show in Section 5. These characteristics are widely used in the literature, and they effectively capture the behaviors of GPU applications [Che and Skadron 2014; Seo et al. 2011; Bakhoda et al. 2009; Kerr et al. 2009; Goswami et al. 2010]. Additionally, we validate the effectiveness of these characteristics using PCA, as shown in Section 6.

Now, we provide a detailed description of each characteristic shown in Table I:

- Instruction throughput* represents the total throughput of an application, and we use instructions per cycle (IPC) to measure it.
- Computation-to-memory access ratio* (CMAR) is the ratio of computations (the number of scalar, vector, and branch instructions) to memory accesses (the number of private, local, and global memory operations). We use this characteristic to determine if an application is compute insensitive or memory insensitive, where a higher CMAR indicates a compute-insensitive application and a lower CMAR indicates a memory-insensitive application.
- Dynamic memory instruction mix* is the distribution of memory instruction types that are executed. We use private, local, and global memory ratios to determine the memory instruction mix. We measure the private memory ratio as the number of private memory instructions executed divided by the total number of memory instructions executed. Similarly, we calculate local and global memory ratios. This characteristic is crucial for performance because different memory instructions have different throughputs. For example, a higher global memory ratio can result in poor performance and scalability.
- Memory efficiency* is measured by using memory coalescing and hit ratio in this work. Memory coalescing refers to combining multiple memory accesses into a single combined access. Since fewer requests result in less contention to global memory, a high ratio of coalesced memory accesses improves application performance. Hence, memory coalescing can be the first optimization to consider in which memory bandwidth usage is reduced. Note that the maximum memory coalescing can be 1 when all accesses are coalesced, and the minimum memory coalescing can be 0 when there is no coalesced access. In addition, we measure the hit ratio for caches, TLBs, and main memory, which is the number of hits divided by the number of accesses. Similarly, a higher hit ratio results in a high performance, and the hit ratio ranges from 0 to 1.


```

1  Benchmark_syn() {
2      /* Constructor: Initialize class member variables. */
3  }
4
5  int setup() {
6      /* Perform all benchmark setup */
7      setupBenchmark_syn(); /* Allocate and initialize host memory. */
8      setupCL(); /* Perform OpenCL related initializations:
9      * create kernel0 from syntheticKernel0, create context, command queue, etc. */
10 }
11
12 int runCLKernels0() {
13     /* Set values for kernel's arguments, enqueue calls to the kernel,
14     * and wait until the kernel execution is completed. */
15     setWorkGroupSize0(); /* Set the work-group size */
16 }
17
18 int run() {
19     /* Run OpenCL kernel program(s). */
20     runCLKernels0(); /* run kernel0 created in setupCL function. */
21 }
22
23 int cleanup() {
24     /* Cleanup OpenCL API (context, memory buffer, etc.) resources and
25     * program (input/output memory, etc.) resources. */
26 }
27
28 int main(int argc, char * argv[]) {
29     Benchmark_syn clBenchmark_syn; // Create a synthetic benchmark object
30     clBenchmark_syn.setup();        // Setup
31     clBenchmark_syn.run();          // Run
32     clBenchmark_syn.cleanup();      // Cleanup
33 }

```

Fig. 2. Host program of a synthetic benchmark.

—*Compute unit occupancy* refers to the utilization of the computation resources (wavefronts) of a compute unit on a GPU. Work-items per work-group (work-group size), registers per work-item, and local memory per work-group limit the number of (in-flight) wavefronts per compute unit. Note that a higher compute unit occupancy indicates a higher utilization of computation resources. To mimic the compute unit occupancy behavior of an original application, we need to capture these characteristics.

During the characterization of a GPU application, we analyze the final executable binary files instead of analyzing the source code. Hence, we do not need the source code of an original application, and therefore our approach can work on proprietary applications.

5. BENCHMARK GENERATION

In this section, we elaborate on how we generate a synthetic benchmark from the captured characteristics of an original application—the abstract benchmark model. The generated synthetic benchmark consists of a host program and a kernel program. We show the host program with its basic functions in Figure 2 and the kernel program in Figure 3. The host program has a main function and other functions to set up and run OpenCL kernel(s). In the *main* function, first, we create a (C++) synthetic benchmark object by using the class constructor and then perform setup, run, and cleanup operations on this object, respectively. In the *setup* function, we adjust the width and the height of inputs/outputs and then allocate and initialize host memory. We then create OpenCL constructs including context, device list, command queue, and memory buffers. Note that we create the OpenCL program construct by using an offline compilation mechanism in which we build the kernel program executable offline and

```

1  /* matrixA and matrixB are inputs and matrixC is output. */
2  __kernel void syntheticKernel0(__global float4 *matrixA, __global float4 *matrixB,
3    __global float4 *matrixC, uint widthA, uint widthB, __local float4 *blockA)
4  {
5      /* Code blocks to improve the similarities of characteristics.
6       * For example, code block to decrement IPC or to increment memory coalescing */
7  }

```

Fig. 3. Kernel program of a synthetic benchmark.

then load the binary at runtime. In the *run* function, we execute all kernel program(s), each corresponding to a kernel in the original application. In the *runCLKernels0* function, we set values for the kernel's arguments including inputs, outputs, and sizes such as height and width; enqueue calls to the kernel by using the command queue; and wait until the kernel execution is completed. In the *setWorkGroupSize0* function, we set the work-group size (work-items per work-group) based on the characteristic of the original application. Last, in the *cleanup* function, we remove the allocated/created resources, including memory, context, and memory buffer. The kernel program in Figure 3 has two inputs (*matrixA* and *matrixB*) and an output (*matrixC*) matrix where matrices can be one- or two dimensional depending on the input/output dimensions of the original application. In addition, if a local memory is used by the original application, we define and use a local memory (*blockA*) in the synthetic application. An important feature of our work is the code blocks inserted into the kernel to mimic the characteristics of the original application. Next we describe how we calculate the similarity between an original application and the synthetic benchmark before we elaborate on code blocks.

5.1. Similarity Measurement

From the list of characteristics shown in Table I, we use IPC, CMAR, private, local, and global memory ratio, memory coalescing, and local memory per work-group characteristics to calculate the similarity (accuracy) between a synthetic and the original benchmark. We do not use the number of in-flight wavefronts and work-items per work-group, because we make sure that a synthetic benchmark has the same values for the number of kernels; the number of work dimensions, global sizes, local sizes, work-items per work-group; and the number of in-flight wavefronts. We also do not use registers per work-item characteristic, since due to a bug in Multi2Sim, we can collect this characteristic for all original benchmarks that are available with Multi2Sim distribution but not for synthetic benchmarks that we create. Finally, although we do not use the hit ratio characteristic in similarity calculation, the synthetic benchmarks preserve this characteristic, as we will show in the experiments. This is because characteristics such as global memory ratio and memory coalescing implicitly capture (mimic) this characteristic.

We define the *individual similarity rate* for a characteristic *ch* as $[1 - \text{errorrate}_{ch}] \times 100$ and $\text{errorrate}_{ch} = |(\text{chsyn} - \text{chorg})| / \text{chorg}$, where *chsyn* and *chorg* are the values for a given characteristic in the synthetic and the original benchmark, respectively. Note that an individual similarity rate ranges from 0 to 100. Then we calculate an *overall similarity rate (osr)* as an equal weighted average of all individual similarity rates. Thresholds for individual and overall similarity rates are given by the user.

5.2. Code (Block) Generation

After we measure the similarity between the original application and the corresponding synthetic benchmark, if the individual and overall similarity rates meet the individual and overall thresholds defined by the user, the synthesis process is completed. Note that our code blocks do not contain instruction set architecture (ISA)-specific assembly

instructions, as was done in earlier synthetic benchmark generation works, as they break portability. Additionally, we compile our synthetic benchmarks with the “-O0” option so that the compiler does not remove our code blocks. Otherwise, we start iterations, where at each iteration we add, remove, or change a code block for the characteristic that has the least similarity with the original application in the kernel code. Figure 4 shows sample code blocks, which we experimentally obtained, to increment/decrement the values of characteristics. Now we explain each of these code blocks.

Code blocks to mimic instruction throughput. For example, when the IPC of an original application is higher than the IPC of the synthetic benchmark, we add CB1, which has instructions with high IPC. We have two different code blocks (CB2.1 and CB2.2) to decrement IPC where we use CB2.1, if the IPC of the synthetic benchmark minus the IPC of the original application is greater than 9. Otherwise, we use CB2.2. Note that both CB2.1 and C2.2 have instructions with low IPC.

Code blocks to mimic computation-to-memory access. We use CB3 to increment CMAR where there exist many computation instructions and a few memory access instructions. In the code block, the number of *cmar1* used in sum operation ($matrixC[0] = cmar1 + \dots + cmar1$) can change depending on the CMAR of the original application. For example, if CMAR of an original application is around 0.80, the number of *cmar1* used in sum operation is 1 (i.e., $matrixC[0] = cmar1$), and if CMAR of an original application is around 0.44, the number of *cmar1* used in sum operation is 2 (i.e., $matrixC[0] = cmar1 + cmar1$). Note that we experimentally characterized code blocks and found the specific values, such as 0.80 and 0.44. Although we experimentally validate these values on AMD GPUs, these characteristics are platform independent, as will be shown in the experiments. Similarly, we use CB4 to decrement CMAR where there exist many memory access instructions and a few computation instructions. In the code block, the number of iterations (*LOOP_COUNT*) is a variable and depends on the CMAR of the original application. To find *LOOP_COUNT*, we initialize it as 1 and then increment it until we meet the CMAR similarity between the original and synthetic.

Code blocks to mimic dynamic memory instruction mix. We use code blocks CB5.1 and CB5.2 in which we perform operations on private memories to increment private memory ratio. If the number of instructions of an original application is less than or equal to 500 instructions, we use CB5.1; otherwise, we use CB5.2, which has more instructions than CB5.1. Hence, using CB5.1 provides higher speedups for small applications, as it has fewer instructions, and using CB5.2 provides a higher private memory ratio, as it has more private memory instructions. Note that the value of 500 for the number of instructions is experimentally obtained—that is, when we use CB5.2 in the synthetics for original applications with fewer than 500 instructions instead of CB5.1, we observe that the synthetics can be slower than the corresponding originals, which is something that we do not want. We use CB6, in which there exist memory operations on global memories such as *matrixA*, *matrixB*, and *matrixC* to increment global memory ratio. In the code block, the number of iterations (*LOOP_COUNT*) can vary depending on the global memory ratio of the original application. In CB7, we perform memory operations on private memories to decrement global memory ratio of the synthetic benchmark. In this code block, we define a number of private memories (*gmr1*, *gmr2*, ..., *gmrN*), where the number is chosen high if the difference between the global memory ratios of the original and synthetic is high; otherwise, the number is chosen low.

Code blocks to mimic memory efficiency. We have two different code blocks, CB8.1 and CB8.2, to increment memory coalescing of a synthetic benchmark. If the memory coalescing of an original benchmark is greater than 0.7, we use CB8.1; otherwise, we use CB8.2. In CB8.2, *STRIDE* is variable, and if the memory coalescing of an original benchmark is greater than 0.4, *STRIDE* is 2; else if it is greater than 0.1, *STRIDE* is 3;


```

/* CB1: Code block to increment IPC */
int ipc1;
for (ipc1 = 0; ipc1 < 1; ipc1++)
    matrixC[0] = 1.2;

/* CB2.1: Code block to decrement IPC used if (candIPC - origIPC) > 9 */
int xValue = get_global_id(0);
int yValue = get_global_id(1);
float4 ipc1 = matrixA[yValue * widthA + xValue];
matrixA[0] = ipc1;

/* CB2.2: Code block to decrement IPC used otherwise */
float ipc1 = 1, ipc2 = 2;
for (i = 0; i < LOOP.COUNT; i++) { // LOOP.COUNT is variable
    matrixA[i * 4 + 1] = ipc1 + ipc2;
    mem_fence(CLK_GLOBAL_MEMFENCE);
}

/* CB3: Code block to increment CMAR */
float4 cmar1 = matrixA[0];
matrixC[0] = cmar1 + cmar1; // the number of cmar1 used in sum operation can change

/* CB4: Code block to decrement CMAR */
float4 cmar1, cmar2;
int cmari;
for (cmari = 0; cmari < LOOP.COUNT; cmari++) { // LOOP.COUNT is variable
    cmar1 = cmar1 + cmar2;
}
matrixC[0] = cmar1;

/* CB5.1: Code block to increment private memory ratio
 * used if #instructions of original application <= 500 */
float4 pmr1, pmr2;
matrixC[0] = pmr1 + pmr2;

/* CB5.2: Code block to increment private memory ratio used otherwise */
float4 pmr1 = matrixA[0];
float4 pmr2 = matrixB[0];
matrixC[0] = pmr1 + pmr2;

/* CB6: Code block to increment global memory ratio */
for (i = 0; i < LOOP.COUNT; i++) { // LOOP.COUNT is variable
    matrixC[i] = matrixA[i] + matrixB[i];
}

/* CB7: Code block to decrement global memory ratio */
float4 gmr1 = matrixA[0];
float4 gmr2 = gmr1 + 2;
float4 gmr3 = gmr2 + 2;
float4 gmr4 = gmr3 + 2;
matrixC[1] = gmr4;

/* CB8.1: Code block to increment memory coalescing used if origMcol > 0.7 */
matrixC[0] = blockA[0]; // blockA is allocated from local memory

/* CB8.2: Code block to increment memory coalescing used otherwise */
size_t tid = STRIDE * get_global_id(0); // STRIDE is variable.
// if origMcol > 0.4, it is 2; else if origMcol > 0.1, it is 3; otherwise, it is 4.
if (tid < 32) {
    blockA[tid] = 1.2;
    matrixC[0] = blockA[tid];
}

/* CB9: Code block to increment or decrement local memory per work-group */
..local float localds[LOCALMEM.SIZE]; // LOCALMEM.SIZE is variable

```

Fig. 4. Sample code blocks to increment/decrement the values of kernel program characteristics.

```

1  __kernel void syntheticKernel_local0(__global float4 *matrixA, __global float4 *matrixB,
2  __global float4 *matrixC, uint widthA, uint widthB, __local float4 *blockA)
3  {
4  /* CB3: Code block to decrement CMAR */
5  float4 cmar1, cmar2;
6  int cmari;
7  for (cmari = 0; cmari < 8; cmari++) {
8      cmar1 = cmar1 + cmar2;
9  }
10 matrixC[0] = cmar1;
11
12 /* CB5.2: Code block to increment private memory ratio */
13 float4 pmr1 = matrixA[0];
14 float4 pmr2 = matrixB[0];
15 matrixC[0] = pmr1 + pmr2;
16
17 /* CB8.1: Code block to increment memory coalescing */
18 matrixC[0] = blockA[0];
19 }

```

Fig. 5. Synthetic benchmark (kernel program) for QuasiRandomSequence.

otherwise, *STRIDE* is 4. In CB8.1, each work-item accesses the same memory location (*blockA*), and hence we have high memory coalescing. On the other hand, in CB8.2, each work-item accesses strided memory locations where stride lengths are 2, 3, and 4, and hence they have lower memory coalescing.

Code blocks to mimic compute unit occupancy. We use CB9 to increment or decrement local memory per work-group characteristic of a synthetic benchmark. In this code block, we define a local memory (*localds*) that has the same size (*LOCAL_MEM_SIZE*) as the local memory used in the original application.

We continue until the iteration upper bound set by the user is reached or the individual and overall similarity rates are satisfied.

5.3. A Detailed Example

We demonstrate our approach on the GPU benchmark QuasiRandomSequence from AMD APP SDK, where we set the overall similarity rate threshold as 90% and the individual similarity rate threshold as 60%. Figure 5 shows the final synthetic benchmark, where it takes 11 iterations to obtain 98% overall similarity rate. The individual similarity rates other than CMAR are 100%, and the CMAR similarity rate is 88%. The speedup defined as the ratio of the GPU simulation time of the original application to the GPU simulation time of the corresponding synthetic benchmark is $6.6\times$. For the initial synthetic benchmark, the overall similarity rate is 57%, and each one of CMAR, private memory ratio, and memory coalescing similarity rates are the lowest (0%). Hence, during iterations, we add (CB3, CB5.2, and CB8.1) code blocks shown in the figure to obtain the final synthetic benchmark. Specifically, for CB3, we set *LOOP_COUNT* as 1 initially and then update it until we mimic the CMAR of the original application, where it becomes 8. We use CB5.2 to increment private memory ratio because the number of instructions of QuasiRandomSequence, which is 32,288, is greater than 500. Since the memory coalescing of QuasiRandomSequence, which is 0.98, is larger than 0.7, we use CB8.1 to mimic the memory coalescing characteristic of the original application. We observe that adding more code blocks decreases speedup as expected—that is, the initial synthetic benchmark is faster than the final one. This indicates that decreasing similarity thresholds can result in faster synthetic benchmarks.

6. EXPERIMENTS

We performed several experiments to validate our GPU synthetic benchmark generation framework MINIME-GPU. MINIME-GPU and all of our benchmarks can be

Table II. GPU Architectural Configurations

Platform	Compute Unit		Memory Hierarchy		
Device	Compute Units (CUs)	Processing Elements per CU	Register File Size per CU	Local Memory Size per CU	Global Memory Size
HD 7970	32	64	256KB	64KB	3GB
HD 7870	20	64	256KB	64KB	2GB
HD 7850	16	64	256KB	64KB	2GB
HD 7770	10	64	256KB	64KB	1GB
HD 7870d	20	64	64KB	16KB	2GB
HD 7850d	16	64	128KB	32KB	2GB
HD 7770d	10	64	512KB	128KB	1GB

downloaded from our Web site.¹ We set the overall similarity rate threshold as 90%, the individual similarity rate threshold as 60%, and iteration upper bound to 20. These similarity rates are the maximum achievable scores with our framework. Note that we performed all experiments using Multi2Sim, except the real hardware experiments in Section 6.7.

Table II shows the platforms on which we run our experiments. Unless otherwise specified, we perform all experiments except the ones in the assessing architecture changes section on the AMD HD 7970 GPU platform. In the next section, we provide details for benchmarks and tools we used.

6.1. Simulation and Benchmarks

We performed experiments on a system running Ubuntu 12.04 x64. We use Multi2Sim 4.2, which is a cycle-based detailed simulation framework for CPU-GPU heterogeneous computing, to run original applications and synthetic benchmarks. Multi2Sim is a fully configurable open source simulator that supports several CPU and GPU architectures, such as x86 CPU and AMD SI GPU. Since we cannot gather all of the characteristics that we described earlier with a default Multi2Sim tool, we have added new extensions to the tool to gather our missing characteristics. For example, we capture and dump work dimension and global/local size characteristics to mimic work-items per work-group characteristic. Target architectures can be configured via configuration files, and a user can create a new configure file or can select an existing one depending on the target platform. We also use the integrated Multi2C, the kernel compiler, to produce kernel binaries for synthetic benchmarks. Note that we only use the AMD SI architecture in the experiments, as Multi2C fails to generate AMD Evergreen architecture binaries for synthetic benchmarks. We produce host binaries for synthetic benchmarks using gcc 4.6.3. We use the AMD Catalyst 13.20 driver and AMD APP SDK v2.9 with OpenCL 1.2 during generation of synthetic benchmarks. We compiled synthetic benchmarks with the “-O0” option so that the compiler would not remove our code blocks. Hence, our synthetic benchmarks cannot be used in compiler optimization studies.

We ran experiments on all 23 benchmarks from AMD APP SDK 2.5 provided with Multi2Sim. These are the only GPU benchmarks that run on Multi2Sim. We used default (medium) inputs to generate synthetic benchmarks. We also used small and large inputs to assess input dependence of synthetic benchmarks. Note that we cannot use benchmarks from other benchmark suites, including Rodinia [Che et al. 2009] and Parboil [Parboil Benchmark Suite 2015], because they are not supported by the current distribution of Multi2Sim.

¹<http://depend.cmpe.boun.edu.tr/tools/minimegpu>.

Fig. 6. PCA loadings with respect to each characteristic.

6.2. Applying PCA to Validate the Importance of Characteristics

After the characterization of an original GPU application, we gather a set of characteristics and generate a dataset. It is important that each characteristic in this dataset contributes to the behavior of the application. Hence, we perform PCA statistical analysis [Duntelman 1989] on all original benchmarks for all (11) characteristics described earlier to select the most suitable characteristics for benchmark generation. We use MATLAB [2014] and appropriate libraries to implement PCA.

Typically, most of the variance is contained in the first two or three principal components (PCs). However, in our case, we need to use the first seven PCs that capture more than 90% of the total variance of our dataset. We show factor loadings for the first three (PCs) in Figure 6. We use only the first three PCs because they capture nearly 70% of the total variance of our dataset and the other PCs capture a small amount of the total variance. Note that a factor loading closer to -1 or $+1$ indicates a higher influence on the PC. We observe that instruction throughput, memory instruction mix, and memory efficiency have the highest correlation with PC1, compute unit occupancy and CMAR have the highest correlation with PC2, and compute unit occupancy has the highest correlation with PC3. These correlations validate that each characteristic we gather is important to describe the behaviors of an application.

6.3. Synthetic Benchmark Generation Results

Table III shows our synthetic benchmark generation results on the AMD HD 7970 GPU platform. In the table, we show *dwarf* type [Asanovic et al. 2006] and the number of instructions (*#OrgInst*) for each original application (benchmark). Note that a dwarf defines an algorithmic method that captures computations and communication patterns of an application. Hence, benchmarks with different dwarfs have different behaviors, and we validate that our approach can work on a diverse set of applications. We denote the number of iterations that it takes to generate a synthetic benchmark by *#Iter*. In many cases, we generate synthetic benchmarks in fewer than 10 iterations where the maximum number of iterations is 15 for MatrixTranspose. We use *Speedup(x)* to denote the ratio of the GPU simulation time of the original application to the GPU simulation time of the corresponding synthetic benchmark. On average, our approach speeds up GPU simulation by a factor of $513\times$, $539\times$, $550\times$, and $567\times$ for HD 7970, HD 7870, HD 7850, and HD 7770 platforms, respectively. The harmonic mean speedup is $541\times$. The minimum speedup is $1.1\times$ for URNG, and the maximum speedup is $7,284.8\times$ for EigenValue. We observe that obtaining a higher speedup for small applications is difficult, because they are already fast with a small number of instructions. In addition, when an original application has a characteristic with a very low or high

Table III. Synthesis Results on the AMD HD 7970 Platform

Benchmark	Dwarf	OrgIns (#)	Iter (#)	Speedup(x)	OSR(%)
BinarySearch	Graph Traversal	128	2	2.8	95
BinomialOption	Dense Linear Algebra	1,984,512	4	516.2	95
BitonicSort	Graph Traversal	1,267,200	4	44.5	95
BlackScholes	Sparse Linear Algebra	1,139,712	8	3.4	94
DCT	Spectral Methods	20,672	9	3.6	96
DwtHaar1D	Spectral Methods	2,113	10	1.4	97
EigenValue	Spectral Methods	19,826,681	6	7,284.8	100
FastWalshTransform	Dense Linear Algebra	4,000	4	8.0	99
FFT	Spectral Methods	1,316	8	14.1	98
FloydWarshall	Dynamic Programming	11,046,882	2	280.9	90
Histogram	Structured Grids	334,912	3	190.1	96
MatrixMultiplication	Sparse Linear Algebra	8,000	6	11.1	100
MatrixTranspose	Dense Linear Algebra	3,392	15	1.2	96
MersenneTwister	Combinational Logic	1,347,584	8	4.0	94
PrefixSum	Dense Linear Algebra	1,341	10	2.6	97
QuasiRandomSequence	Combinational Logic	32,288	11	6.6	98
RadixSort	Graph Traversal	39,488	3	2,244.0	99
RecursiveGaussian	Dense Linear Algebra	1,606,368	10	1,159.0	99
Reduction	Dense Linear Algebra	714	7	4.3	95
ScanLargeArrays	Backtrack & Branch+Bound	3,688	5	8.3	93
SimpleConvolution	Structured Grids	17,900	6	2.0	94
SobelFilter	Dense Linear Algebra	1,297,944	7	1.3	93
URNG	Structured Grids	2,146,304	11	1.1	92

Fig. 7. Comparison of IPC between the synthetic and original benchmarks.

value, the size of our code blocks increase, and hence we have a low speedup value. For example, CMAR of the original MersenneTwister is very low (0.2), and we add CB4 to decrement CMAR of the synthetic benchmark where *LOOP_COUNT* is 27. Since we have a high *LOOP_COUNT* value, the dynamic instruction count of the synthetic is also high, which results in a low speedup. Furthermore, when the work-group count of a synthetic benchmark, which is the total number of work-groups executed, is high, the number of instructions is also high, and hence the speedup is low. For example, SobelFilter and URNG have 1,024 and 4,096 work-groups and their speedups are $1.3\times$ and $1.1\times$, respectively. In the table, we show the overall similarity rate (*OSR (%)*) for each benchmark where the average overall similarity rate is 96%, the minimum overall similarity rate is 92% for URNG, and the maximum overall similarity rate is 100% for EigenValue and MatrixMultiplication.

6.4. Assessing Similarity

Now we demonstrate individual similarity rates for each characteristic. Figure 7 shows IPCs for the synthetic and original benchmarks where the average IPC similarity rate

Fig. 8. Comparison of memory coalescing between the synthetic and original benchmarks.

is 91%, the minimum IPC similarity rate is 60% for only 3 benchmarks, and the maximum IPC similarity rate is 100% for 16 benchmarks. Note that Multi2Sim captures IPC values as integer values and rounds them down to 0 when they are smaller than 1. In the figure, the IPCs of BlackScholes and its corresponding synthetic benchmark are 15 and 21, respectively, and hence the IPC similarity rate is 60%. We observe that once we add code blocks that increment private memory ratio and decrement CMAR of the synthetic, we meet the overall (90%) and individual (60%) similarity rate thresholds. We performed a new experiment in which we set the individual similarity rate threshold to 80% to check whether we could improve the IPC similarity for BlackScholes. In this case, we observe that adding a code block to improve IPC similarity has a side effect on CMAR similarity and decreases it. We observe that the similarity rate thresholds (90% and 60%) are the maximum achievable scores with our framework due to side effects, and we are planning to handle these side effects in future work. Due to side effects, the original and the synthetic benchmark can diverge, and there is no systematic bias for similarity rates of the synthetic benchmarks. Note that handling side effects in C is harder than handling them in assembly.

Figure 8 shows memory efficiencies for the synthetic and original benchmarks where the average memory coalescing similarity rate is 94%, the minimum memory coalescing similarity rate is 69% for only 1 benchmark and the maximum memory coalescing similarity rate is 100% for 12 benchmarks. Due to lack of space, we display the results for IPC and memory coalescing, but we had similar results for other characteristics as well.

6.5. Assessing Architecture Changes

When developing synthetic benchmarks, it is important that synthetic benchmarks are portable—that is, they preserve the behaviors of original applications from which they are generated across different architectures. This is because we want benchmarks to allow architectural exploration. To assess the usage of our synthetic benchmarks on different architectures, we perform a set of experiments on four different existing platforms: AMD HD 7970, HD 7870, HD 7850, and HD 7770. We show these platforms, which have different configurations, in Table II. In the experiments, we generated synthetic benchmarks on the AMD HD 7970 GPU and then ran these synthetic benchmarks on other platforms (without regeneration). We observe that our synthetic benchmarks mimic the behaviors of original applications across different platforms. For example, we show comparison of sensitivity to architecture changes for BitonicSort and QuasiRandomSequence in Figure 9. The IPC of the BitonicSort benchmark decreases going from the HD 7970 (on which the synthetic benchmark is generated) to other platforms, and the IPC of the synthetic benchmark follows this change. The IPC of QuasiRandomSequence does not change going from the HD 7970 to other platforms, and similarly the IPC of the synthetic benchmark does not change. The correlation coefficients for BitonicSort and QuasiRandomSequence are 0.99 and 1, respectively, and the average of the correlation coefficient for all benchmarks is 0.93.

Fig. 9. Comparison of sensitivity to architecture changes for BitonicSort and QuasiRandomSequence.

Fig. 10. The IPC error rate for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

Fig. 11. The hit ratio error rate for different GPUs (HD 7970, HD 7870, HD 7850, and HD 7770).

Figure 10 shows the IPC error rate for all platforms. From the figure, it is clear that our synthetic benchmarks mimic the behavior changes of the original benchmarks across different platforms. We observe that the BinomialOption and Histogram benchmarks have high IPC error rates moving from the HD 7970 to other platforms. This is because the IPC of original benchmarks are very low (near to zero), and a small difference between the IPC of the original benchmark and the synthetic benchmark results in a high error rate. For instance, the IPCs of the original and synthetic Histogram benchmarks is 1 on the HD 7970 platform, where the error rate is 0%. When we move the original benchmark to other platforms, the IPC becomes 0. However, the IPC of the corresponding synthetic benchmark remains 1. Hence, this small difference (1, in terms of IPC) results in a high (100%) error rate. Figure 11 shows the hit ratio error rate for all platforms. Note that although we do not use the hit ratio characteristic, which is the average of the L1 cache, L2 cache, and global memory hit ratio, in similarity measurement during benchmark generation, the average hit ratio similarity between

Fig. 12. The number of (in-flight) wavefronts error rate for the GPUs with derived architectural configurations (HD 7970, HD 7870d, HD 7850d, and HD 7770d).

the original and synthetic benchmarks on the HD 7970 is 82%, and our synthetic benchmarks mimic the hit ratio changes of the original benchmarks across different platforms. We also observe that our synthetic benchmarks mimic the L1, L2, and global memory hit ratio changes of the original benchmarks across different platforms, where the average L1, L2, and global memory hit ratio similarity between the original and synthetic benchmarks on the HD 7970 is 84%, 70%, and 98%, respectively.

6.5.1. Synthesis for GPUs with Derived Architectural Configurations. We observe that the only difference between the existing four platform configurations is the number of compute units. Hence, we performed a new set of experiments on platforms that we derived from the existing platforms by changing register file size and local memory size per CU. Table II shows these platforms (HD 7870d, HD 7850d, and HD 7770d), which are not part of Multi2Sim distribution. Global memory sizes were not changed, as the memory demands of the benchmarks that we use are small. In the experiments, we ran the synthetic benchmarks generated on the HD 7970 on the other platforms (without regeneration). We observe that CMAR, private memory ratio, local memory ratio, global memory ratio, memory coalescing, work-items per work-group, and local memory per work-group characteristics are platform independent and do not change across platforms as expected. IPC and hit ratio characteristics change only when the number of compute units changes. For example, these characteristics do not change moving from the HD 7870 to the HD 7870d, but they change moving from the HD 7970 to the HD 7870, and our synthetic mimics these changes as shown in Figures 10 and 11.

The number of (in-flight) wavefronts characteristic depends on work-items per work-group, registers per work-item, and local memory per work-group characteristics. Our experiments demonstrate that number of (in-flight) wavefronts characteristic changes moving from the HD 7970 to a derived platform due to changes in registers per work-item and local memory per work-group characteristics. Figure 12 shows the numbers of (in-flight) wavefronts error rate for all derived platforms. For example, the number of (in-flight) wavefronts of the FFT benchmark on the HD 7970, HD 7870d, HD 7850d, and HD 7770d is 7, 10, 3, and 1, respectively, and the corresponding synthetic benchmark mimics these values on each platform. Hence, the error rates are 0% on all platforms. We observe that the local memory per work-group limits the number of (in-flight) wavefronts for FFT. Hence, increasing the local memory size per compute unit increases the number of (in-flight) wavefronts, and we mimic the number of (in-flight) wavefronts for FFT since we mimic the local memory size per compute unit. However, the number of (in-flight) wavefronts of the MatrixMultiplication benchmark on the HD 7970, HD 7870d, HD 7850d, and HD 7770d is 10, 10, 7, and 3, respectively, and it is 10 for the

Table IV. GPU Architectural Parameters

Parameter	HD 7970	HD 7970d1	HD 7970d2	HD 7970d3
Front-end issue latency	1	2	4	8
Front-end issue width	5	8	12	16
SIMD unit width	1	2	4	8
Scalar unit width	1	2	4	8
Scalar unit ALU latency	4	8	12	16
Vector memory unit width	1	2	4	8
Branch unit width	1	2	4	8

Fig. 13. The IPC error rate for GPUs with derived architectural parameters (HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3).

corresponding synthetic benchmark on all platforms. Hence, the error rates are 0%, 0%, 43%, and 100%, respectively. Note that we cannot mimic the changes across different platforms, as registers per work-item limits the number of (in-flight) wavefronts for the MatrixMultiplication, which results in a high error rate, and we cannot mimic registers per work-item characteristic. Once the bug in Multi2Sim is fixed, we can also mimic the registers per work-item characteristic as well as the number of (in-flight) wavefronts characteristic.

We also perform a new set of experiments on platforms that we derived from the existing HD 7970 platform, which are denoted as HD 7970d1, HD 7970d2, and HD 7970d3. We show these platforms, which have different architectural parameters including front-end issue latency, SIMD unit width, and scalar unit ALU latency, in Table IV. In the table, *front-end issue latency* is the number of cycles that it takes to issue a wavefront to its execution unit, and *front-end issue width* is the maximum number of instructions that can be executed in a single cycle. *SIMD unit width* is the number of instructions processed by each stage of the pipeline per cycle. *Scalar unit width*, *vector memory unit width*, and *branch unit width* are similar to SIMD unit width. *Scalar unit ALU latency* is the number of cycles it takes to execute a scalar arithmetic logic instruction. In the experiments, we run the synthetic benchmarks generated on the HD 7970 on other platforms derived from the HD 7970 (without regeneration). We observe that IPC changes moving from the HD 7970 to other derived platforms and our synthetic mimics these changes as shown in Figure 13, where the IPC similarity rate is 91%, 91%, 91%, and 89% on the HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3, respectively. For example, the IPC of the original Histogram benchmark is 1 on all platforms, and similarly the IPC of the synthetic benchmark is 1 on all platforms. On the other hand, the IPC of the original BitonicSort benchmark decreases moving from the HD 7970 to other platforms, and the corresponding synthetic benchmark mimics these changes. In other words, the IPC of the original BitonicSort benchmark

Table V. GPU Cache Configurations

Configuration	Config-0	Config-1	Config-2	Config-3
L1: Sets	64	128	32	64
L1: Associativity	4	4	4	2
L2: Sets	128	256	64	128
L2: Associativity	16	16	16	8

Fig. 14. The cache hit ratio error rate for different cache configurations (*Config-0*, *Config-1*, *Config-2*, and *Config-3*).

is 18, 18, 14, and 9 and the IPC of the corresponding synthetic benchmark is 16, 15, 13, and 9 on the HD 7970, HD 7970d1, HD 7970d2, and HD 7970d3, respectively.

Next we considered other characteristics, including memory instruction mix and hit ratio, to evaluate the portability of our synthetic benchmarks. We find the synthetic benchmarks to accurately mimic these characteristics compared to the original benchmarks across GPUs having different architectural parameters.

In addition, we perform a new set of experiments on HD 7970 platforms with different cache configurations. We show these cache configurations in Table V. In the experiments, we run the synthetic benchmarks generated on the HD 7970 with *Config-0* on other platforms with different cache configurations (without regeneration). We observe that the cache hit ratio changes moving from *Config-0* to other configurations and our synthetic mimics these changes as shown in Figure 14, where the average hit ratio similarity rate is 82%, 84%, 85%, and 82% on *Config-0*, *Config-1*, *Config-2*, and *Config-3*, respectively. For example, the hit ratio of the original BinomialOption benchmark is 36% on all configurations, and similarly the hit ratio of the synthetic benchmark is 34% on all configurations. On the other hand, the hit ratio of the original SobelFilter benchmark changes moving from *Config-0* to other configurations, and the corresponding synthetic benchmark mimics these changes. In other words, the hit ratio of the original SobelFilter benchmark is 52%, 56%, 50%, and 48% and the cache hit ratio of the corresponding synthetic benchmark is 45%, 51%, 40%, and 39% on *Config-0*, *Config-1*, *Config-2* and *Config-3*, respectively.

6.6. Assessing Input Changes

In these experiments, we analyze whether we can use the synthetic benchmark for an original application using a different input than the one for which the synthetic benchmark was generated. In this analysis, we generate a synthetic benchmark for an original benchmark using a medium input. Then, we measure the similarity between this synthetic benchmark and the original benchmark using a different (small or large) input. If the characteristics of the original application do not vary much from using medium input to a different input—that is, the similarity meets the user thresholds—we can use the same synthetic benchmark for the original benchmark using a different

Fig. 15. The IPC values of original benchmarks for small, medium, and large inputs on the HD 7970.

input. Otherwise, we need to generate a new synthetic benchmark. Note that this analysis helps to reduce the effort of generating new synthetic benchmarks.

Figure 15 shows the IPC values of original benchmarks for small, medium, and large inputs on the HD 7970. We observe that some benchmarks have similar IPC values for different inputs, and some benchmarks have different IPC values. For example, we need to generate a new synthetic benchmark for DwtHaar1D using small (and also for large) input, as the individual IPC similarity rate is less than the individual similarity threshold. However, the IPC of URNG does not vary much from using medium input to small (and also large) input, and hence we can use the same synthetic benchmark for URNG using small, medium, and large input. Similarly, we measure the individual similarity rates for other characteristics and overall similarity rate, and we decide whether we can use the existing synthetic benchmark or need to generate a new one. Note that due to lack of space, we do not show the values for other characteristics.

6.7. Validation of Synthetic Benchmarks on Real Hardware

In this section, we validate the robustness of the synthetic benchmarks generated on the simulator platform by running them and the original applications on an actual hardware and checking their similarity. In these experiments, we use the source code of the synthetic benchmarks that are generated on the simulator for the AMD HD 7970 GPU and run these synthetic benchmarks on a real AMD HD 7950 GPU hardware (without regeneration). The AMD HD 7950 GPU is similar to the 7970 GPU, except it has 28 compute units instead of 32.

We performed experiments on an HP Z800 Desktop Workstation system running Windows 7 SP1 x64. We compiled host binaries for synthetic benchmarks using Microsoft Visual Studio 2010. We use the AMD Catalyst 14.12 driver and AMD APP SDK v2.9 with OpenCL 1.2 during the generation of kernel code of the synthetic benchmarks. We use the “-cl-opt-disable” option, which disables all optimizations, so that the compiler would not remove our code blocks. We used the AMD CodeXL 1.7 tool [AMD CodeXL 2015] to collect performance characteristics of original and synthetic benchmarks on the real hardware. CodeXL is the most commonly used tool in the literature for collecting OpenCL program characteristics on AMD GPUs.

We observe that our synthetic benchmarks mimic all but IPC and memory coalescing characteristics of original applications on the real hardware. We were not able to check IPC and memory coalescing, because CodeXL does not provide these characteristics. However, CodeXL provided other characteristics that were not available in Multi2Sim. These characteristics include vector arithmetic logic unit (VALU) utilization and scalar general-purpose registers (SGPRs). Figure 16 shows the VALU utilization characteristic, which is the percentage of active vector ALU threads in a wave, for the synthetic and original benchmarks where the average VALU utilization similarity

Fig. 16. Comparison of VALU utilization between the synthetic and original benchmarks on real hardware (HD 7950).

Fig. 17. Comparison of SGPRs utilization between the synthetic and original benchmarks on real hardware (HD 7950).

rate is 83%. The maximum VALU similarity rate is 100% for 12 benchmarks. The minimum similarity rate is 0% for DwtHaar1D, where the VALU utilization of the original benchmark is 39% and the VALU utilization of the corresponding synthetic benchmark is 100%. This is because we cannot collect the registers per work-item characteristic due to a bug in Multi2Sim, and hence we cannot preserve the compute unit occupancy characteristic of the original benchmark that influences VALU.

Figure 17 shows the SGPR characteristic, which is the number of SGPRs used by the kernel, for the synthetic and original benchmarks where the average SGPRs similarity rate is 73%. The maximum similarity rate is 100% for Histogram, and the minimum similarity rate is 50% for BlackScholes and DwtHaar1D.

We observe that the average cache hit ratio similarity on the HD 7950 is 68%, whereas it is 82% on the HD 7970. Note that the cache hit ratio (provided by CodeXL) is the percentage of fetch, write, atomic, and other instructions that hit the data cache, and this characteristic is different from the one we used in Multi2Sim. Several reasons exist behind this observation. The AMD Catalyst driver version that Multi2Sim supports is version 13.20, whereas CodeXL supports version 14.12. Hence, different drivers can generate different OpenCL binaries. Another reason is the estimation error of the Multi2Sim simulator, which can be more than 30% due to the lack of fidelity in the way the memory subsystem is simulated.

We also observe that our synthetic benchmarks mimic the MemUnitStalled, WriteUnitStalled, and LDSBankConflict characteristics of the original benchmarks, where MemUnitStalled is the percentage of GPU time the memory unit is stalled, WriteUnitStalled is the percentage of GPU time the write unit is stalled, and LDSBankConflict is the percentage of GPU time the local data storage (LDS) is stalled by bank conflicts. The average MemUnitStalled, WriteUnitStalled, and LDSBankConflict similarity between the original and synthetic benchmarks is 65%, 73%, and 85%, respectively. Note that we do not use these characteristics during benchmark generation, and hence the similarity score can be low (e.g., 65%). One can use these characteristics of an original application in the synthetic benchmark generation process to improve the similarity, although this will result in runtime penalty.

Although our synthetics do not include code blocks for the characteristics including VALU utilization and SGPRs, they are still preserved. This further confirms the robustness of our synthetic benchmarks. To improve the similarity between synthetic and original benchmarks in a real hardware environment, we can generate synthetics in a real hardware environment instead of a simulator environment similar to our previous work for CPUs [Deniz et al. 2015].

7. DISCUSSION

We observe that the main cost of generating synthetic GPU benchmarks is collecting the characteristics of original applications. This is because we use a simulator (Multi2Sim) to collect these characteristics, and simulating large (original) applications takes long time. On the other hand, our synthetic benchmarks are small, and simulating these synthetic benchmarks is fast. For example, collecting the characteristics of the FloydWarshall benchmark takes 767.04 seconds; however, the corresponding synthetic benchmark is generated in two iterations, and collecting the characteristics of the synthetic benchmark takes only 1.47 seconds in the first iteration and 9.40 seconds in the second iteration. Note that characteristics collection in the initial iterations during benchmark generation is faster because the number of code blocks can increase with iterations.

It is clear that the speedup we obtain by using a synthetic benchmark instead of an original application is more important than the time required to generate the synthetic benchmark. This is because we generate a synthetic benchmark only once, and we run this synthetic benchmark many times. For example, generating the synthetic benchmark for FloydWarshall takes nearly 780 seconds, but we obtained a synthetic benchmark that is $431.29\times$ faster than the original benchmark.

When generating synthetic benchmarks, selecting the right characteristics to mimic is crucial for the time required to generate the synthetic benchmark, for speedup, and the similarity score. For example, we performed a set of experiments in which we only mimic the IPC of original applications. We observed that the average IPC similarity goes from 91% to 94%, the average speedup goes from $513\times$ to $56,438\times$, and the minimum IPC similarity goes from 60% to 66% on the HD 7970 platform compared to using 11 characteristics. These results validate that using fewer characteristics provides higher similarity and higher speedup. This is because using too many characteristics can result in a long characterization process as well as less similar synthetic benchmarks, because adding a code block to improve one characteristic similarity can decrease another characteristic similarity due to side effects. On the other hand, using too few characteristics can fail to mimic the behavior of an original application. In this work, we use 11 characteristics that capture the diverse behaviors of an original application. We also experimentally validate that many of these characteristics are platform independent, and using platform-independent characteristics makes our benchmarks portable across a wide range of platforms, as shown in the experiments.

Another important point when generating synthetic benchmarks is that the synthetic benchmarks can be used for a study where they were not originally intended to be used. To validate this case, we performed a set of experiments. In these experiments, we compared IPC similarity when it was used during synthesis and when it was not used during synthesis. We observed that the average IPC similarity goes from 91% to 84%, the average speedup goes from $513\times$ to $665\times$, and the minimum IPC similarity goes from 60% to 50% on the HD 7970 platform when IPC similarity is used and not used during synthesis, respectively. These results show that average IPC similarity is still high and acceptable.

Our synthetic benchmarks do not have any useful functionality, and one cannot obtain the original application from our synthetic benchmarks by reverse engineering.

Hence, customers can share a synthetic benchmark that is generated for their proprietary application without compromising the proprietary nature of the proprietary application. Once the hardware developers, architects, or designers have the synthetic benchmark, they can optimize the (GPU) platform to provide improved performance for the proprietary application.

Our synthetic benchmarks are meant to be used in early design exploration. In later stages of development where more accurate performance is required, original applications are still going to be used. Additionally, when using our synthetics, one should note the characteristics that are kept similar in synthetics with respect to the originals and use synthetics for early design exploration of such characteristics.

8. RELATED WORK

In Goswami et al. [2010], the authors present a set of microarchitecture-independent GPU characteristics that capture important behaviors of GPU applications: kernel stress, kernel characteristics, divergence characteristics, instruction mix, and coalescing characteristics. They use these characteristics with PCA and hierarchical clustering analysis to analyze diversity of GPU benchmark suites. We also use many of these characteristics and apply PCA to validate the importance of our characteristics. Che and Skadron [2014] identify a set of important GPU characteristics that are similar to our characteristics, including instruction throughput, CMAR, and memory instruction mix to predict the performance of GPU applications by correlating their characteristics to existing applications' characteristics. The authors also conduct a PCA to illustrate similarity among benchmarks. Bakhoda et al. [2009] present performance characteristics and performance bottlenecks of CUDA applications by analyzing characteristics including IPC, instruction mix, memory coalescing, and warp occupancy on different hardware configurations. Similarly, Kerr et al. [2010] use different characteristics that are similar to ones we use to identify relationships between application behavior and performance on different heterogeneous systems. Since we preserve the characteristics of an original application across different platforms, one can use our synthetic benchmarks, which are fast and small, in performance characterization and bottleneck identification studies instead of using large original applications. In Ukidave et al. [2015], the authors introduce the NUPAR benchmark suite, including OpenCL and CUDA applications, and they characterize these applications in terms of several characteristics, including occupancy, register utilization, and local/shared memory utilization, which are similar to the ones we use.

So far, synthetic benchmarks have been mainly developed for CPU applications (e.g., as in Deniz et al. [2015], Ganesan and John [2013], and Joshi et al. [2008]), which target performance and power characteristics. These synthetics do not target GPUs, which we can target in this work. There exist only a few recent works on synthetic benchmark generation for GPUs. In Yu et al. [2015], the authors generate miniature (synthetic) proxies of CUDA GPGPU kernels where they mimic performance characteristics similar to us. In addition, they only focus on kernel programs and do not mimic host programs and data passing between CPU and GPU similar to us. This is because our goal is to accelerate GPU architecture simulation. They obtain $49\times$ average and $589\times$ maximum speedup with an average IPC error of 4.7%, whereas we obtain $541\times$ average and $7284\times$ maximum speedup with an average IPC error of 9% and overall error of 4%. Note that they measure the similarity (accuracy) between the original and synthetic only in terms of a single characteristic (IPC). However, we use 11 characteristics in similarity measurement and validate that our synthetics preserve all of them. Their approach cannot generate faster synthetics for benchmarks that do not execute any loops, whereas our approach generated faster synthetics for all benchmarks in the experiments. They generate synthetics in CUDA and target NVIDIA GPUs, and they

embed assembly code in CUDA. However, our portable synthetics are in OpenCL and do not include assembly code, and we target different GPU platforms, including AMD, NVIDIA, and Intel. We also validated our approach on a simulator and real hardware, whereas they only validated on a simulator. Huang et al. [2014] use sampling technique to speed up GPU architecture simulation for CUDA applications where they achieve up to $10\times$ speedup, whereas we achieve up to $7,284\times$ speedup. Similarly, Lee and Ro [2013] parallelize the GPU architecture simulation, where they gain up to $4.15\times$ speedup.

In Matsumoto et al. [2012], the authors present a code generator that produces matrix multiply kernels written in OpenCL from a set of user-given parameters. However, they do not mimic the characteristics of existing applications, and they also only target matrix multiplication applications, whereas we can generate synthetic benchmarks for different types of applications. PARAPHRASE FastFlow [FastFlow: Programming Multicore 2014] provides OpenCL-based heterogeneous skeletons that make it easier to understand and develop GPU applications for hybrid CPU/GPU architectures. Similarly, our small and accurate synthetic benchmarks and collected application characteristics can be used to understand large existing applications. In Che et al. [2014], the authors present a directive-based API, Dymaxion++, to enable programmers to optimize memory access patterns. This approach is orthogonal to our approach. They obtain an average of $3.2\times$ performance improvement, whereas we obtain on average $541\times$ speedup. They also need the source code of original applications, as they use source-to-source code translation. However, we only need the binaries of original applications.

Rodinia and Parboil benchmark suites [Che et al. 2009; Parboil Benchmark Suite 2015] target heterogeneous multicore systems. In Seo et al. [2011], the authors characterize the performance of OpenCL benchmarks from NAS Parallel Benchmark suite. Once Multi2Sim supports collecting characteristics of benchmarks from these suites, we plan to extend our experiments on these suites.

Our approach is similar to the approach proposed by us [Deniz et al. 2015] in which we generated synthetic benchmarks in C using Pthreads and MCA libraries for multicore CPU systems. However, this work differs from the previous work in many ways. Specifically, we generate synthetic benchmarks in C++ using the OpenCL library for GPU systems. We collect GPU specific characteristics including computation-to-memory access, memory instruction mix, and compute unit occupancy to capture the behaviors of a GPU application. Our GPU benchmarks consist of a host program and a kernel program where CPU benchmarks have only a CPU program. We develop new code blocks to increment/decrement the values of GPU-specific characteristics, and we add these code blocks to kernel programs. We do not detect or use parallel software architectural patterns such as recursive data and pipeline in this work, but we plan to use dwarfs in synthetic GPU benchmark generation in future work.

9. CONCLUSIONS AND FUTURE WORK

We developed MINIME-GPU, a new benchmark synthesis framework for GPUs to speed up architectural simulation of modern GPU architectures. Our framework captures important characteristics of original GPU applications and generates synthetic GPU benchmarks from those applications. We compared the similarity (accuracy) of original existing applications and the corresponding synthetic benchmarks in terms of several characteristics including instruction throughput, compute unit occupancy, and memory efficiency. The experimental results showed that our synthetic benchmarks mimic the characteristics of the original applications from which they are generated where the average similarity is 96% and average speedup is $541\times$. Additionally, we experimentally validated that our synthetic benchmarks preserve these characteristics

across different architectures on a simulator as well as on a real GPU. Our synthetic benchmarks are generated in OpenCL, which is portable and human readable, and they are also faster and smaller than the original applications. Hence, our framework helps developers and researchers in performance analysis and early architectural exploration of software and hardware.

We plan to extend our experiments on other GPU architectures such as Intel and NVIDIA. In addition, power consumption and communication characteristics between host and compute devices are characteristics that we want to preserve in the future.

REFERENCES

- AMD. 2014a. AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK). Retrieved October 4, 2015, from <http://developer.amd.com/sdks/amdappsdk/>.
- AMD. 2014b. AMD Southern Islands Series Instruction Set Architecture. Retrieved October 4, 2015, from http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.
- AMD CodeXL. 2015. AMD CodeXL—Powerful Debugging, Profiling, and Analysis. Retrieved October 4, 2015, from <http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/>.
- Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley.
- Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Los Alamitos, CA, 163–174.
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Los Alamitos, CA, 44–54.
- Shuai Che, Jiayuan Meng, and Kevin Skadron. 2014. Dymaxion++: A directive-based API to optimize data layout and memory mapping for heterogeneous systems. In *Proceedings of the 2014 IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Los Alamitos, CA, 916–924.
- Shuai Che and Kevin Skadron. 2014. BenchFriend correlating the performance of GPU benchmarks. *International Journal of High Performance Computing Applications* 28, 2, 238–250.
- Etem Deniz, Alper Sen, Brian Kahne, and Jim Holt. 2015. MINIME: Pattern-aware multicore benchmark synthesizer. *IEEE Transactions on Computers* 64, 8, 2239–2252.
- George Henry Dunteman. 1989. *Principal Component Analysis*. Sage Publications, London, United Kingdom.
- FastFlow: Programming Multicore. 2014. FastFlow: Pattern-based multi/many-core parallel programming framework. Retrieved October 4, 2015, from <http://sourceforge.net/projects/mc-fastflow/>.
- Karthik Ganesan and Lizy Kurian John. 2013. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Transactions on Computers* 99, 1. DOI: <http://dx.doi.org/10.1109/TC.2013.36>
- Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li. 2010. Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Los Alamitos, CA, 1–10.
- Jen-Cheng Huang, Lifeng Nai, Hyesoon Kim, and Hsien-Hsin S. Lee. 2014. TBPoint: Reducing simulation time for large-scale GPGPU kernels. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Los Alamitos, CA, 437–446.
- Ajay M. Joshi, Lieven Eeckhout, Robert H. Bell, Jr., and Lizy K. John. 2008. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization* 5, 2, 10:1–10:33.
- Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Los Alamitos, CA, 3–12.
- Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*. ACM, New York, NY, 31–42.

- Khronos OpenCL Working Group. 2012. The OpenCL Specification Version 1.2. Retrieved October 4, 2015, from <https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>.
- Sangpil Lee and Won Woo Ro. 2013. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Los Alamitos, CA, 107–117.
- MATLAB. 2014. MATLAB: The Language of Technical Computing—MathWorks. Retrieved October 4, 2015, from <http://www.mathworks.com/products/matlab/>.
- Kazuya Matsumoto, Naohito Nakasato, and Stanislav G Sedukhin. 2012. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. In *Proceedings of the IEEE International Symposium on Embedded Multicore Socs (MCSoc)*. IEEE, Los Alamitos, CA, 198–204.
- Multi2Sim. 2014. Multi2Sim: A Heterogeneous System Simulator. Retrieved October 4, 2015, from <https://www.multi2sim.org/>.
- Parboil Benchmark Suite. 2015. Parboil Benchmarks. Retrieved October 4, 2015, from <http://impact.crhc.illinois.edu/Parboil/parboil.aspx>.
- Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS parallel benchmarks in OpenCL. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Los Alamitos, CA, 137–148.
- Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. 2015. NUPAR: A benchmark suite for modern GPU architectures. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, New York, NY, 253–264.
- Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy K. John, Hai Jin, Chengzhong Xu, and Junmin Wu. 2015. GPGPU-MiniBench: Accelerating GPGPU micro-architecture simulation. *IEEE Transactions on Computers* 64, 11, 3153–3166.

Received March 2015; revised July 2015; accepted August 2015