GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation

Zhibin Yu, *Member, IEEE,* Lieven Eeckhout, *Member, IEEE,* Nilanjan Goswami, Tao Li, Lizy K John, *Fellow, IEEE,* Hai Jin, *Senior Member, IEEE,*Chengzhong Xu, *Senior Member, IEEE,* Junmin Wu

Abstract—Graphics processing units (GPU), due to their massive computational power with up to thousands of concurrent threads and general-purpose GPU (GPGPU) programming models such as CUDA and OpenCL, have opened up new opportunities for speeding up general-purpose parallel applications. Unfortunately, pre-silicon architectural simulation of modern-day GPGPU architectures and workloads is extremely time-consuming. This paper addresses the GPGPU simulation challenge by proposing a framework, called GPGPU-MiniBench, for generating miniature, yet representative GPGPU workloads. GPGPU-MiniBench first summarizes the inherent execution behavior of existing GPGPU workloads in a profile. The central component in the profile is the Divergence Flow Statistics Graph (DFSG), which characterizes the dynamic control flow behavior including loops and branches of a GPGPU kernel. GPGPU-MiniBench generates a synthetic miniature GPGPU kernel that exhibits similar execution characteristics as the original workload, yet its execution time is much shorter thereby dramatically speeding up architectural simulation. Our experimental results show that GPGPU-MiniBench can speed up GPGPU architectural simulation by a factor of 49$\times$ on average and up to 589$\times$, with an average IPC error of 4.7% across a broad set of GPGPU benchmarks from the CUDA SDK, Rodinia and Parboil benchmark suites. We also demonstrate the usefulness of GPGPU-MiniBench for driving GPU architecture exploration.

Index Terms—computer architecture, GPGPU, simulation acceleration, workload synthesis

1 Introduction

In recent years, interest has grown rapidly towards harnessing the power of graphics hardware to perform general-purpose parallel computing, so-called GPGPU computing. Thanks to the affordable, powerful and programmable GPU hardware [1], developers are increasingly using commodity GPUs to perform computation-heavy tasks that would otherwise require a large compute cluster. GPGPU programming models such as CUDA [2], ATI Stream Technology [3], and OpenCL [4] allow programmers to use hundreds of thousands of threads to leverage the plenty of computation resources of today's GPUs to achieve massive computational power.

The computational power of modern-day GPUs is in sharp contrast to the slow speed of GPGPU architectural simulation. GPGPU architectural simulation is extremely time-consuming, for two reasons. First, architects need to simulate realistic applications with large input data sets to evaluate GPU micro-architecture designs with high fidelity. This implies that many (up to hundreds of thousands) threads need to be simulated. Second, if not single-threaded, parallel GPU simulators are limited by the available number of cores in the simulation host machine, which is typically a multi-core CPU [5] [6] [7]. (Although the latest version (3.x) of the publicly available GPGPU-Sim enables asynchronous kernel launches from the CPU using pthread parallelism, the GPU simulation engine itself is single-threaded [8].) Furthermore, given the increased complexity of GPUs, it is to be expected that the GPGPU simulation challenge is only going to increase in importance in the years to come.

Table 1 shows the execution time of several CUDA benchmarks on a real GPU device (NVidia GeForce 295) and a GPGPU performance simulator (GPGPU-Sim) [5]. These measurements show that GPGPU performance simulation is approximately 9 orders of magnitude slower compared to real hardware. Given how computer architects heavily rely on simulators for exploration purposes at various stages of the design, accelerating GPGPU architectural simulation is imperative.

Existing solutions to accelerating architectural simulation of CPUs, such as sampling or statistical simulation, cannot be readily applied to GPGPU simulation because of the relatively small number of instructions executed per thread (tens to hundreds of thousands of instructions) in a typical GPGPU kernel — CPU benchmarks typically execute billions to trillions of instructions per thread. Furthermore, the large number of branch instructions in GPGPU workloads prohibits the use of spreadsheet-based modeling techniques used for pure-graphics GPU performance evaluation. We

- *Zhibin Yu and Chengzhong Xu are with the Cloud Computing Center, Shenzhen Institute of Advanced Technology, Chinese Academy of Science, 518055, Shenzhen, China. E-mail:{zb.yu,cz.xu}@siat.ac.cn. Chengzhong Xu is also with the Department of ECE in Wayne State University, USA.*
- *Lieven Eeckhout is with the Department of Electronics and Information Systems, Ghent University, Belgium. E-mail: Lieven.Eeckhout@elis.UGent.be.*
- *Nilanjan Goswami and Tao Li are with Department of Electrical and Computer Engineering at the University of Florida, USA, E-mail: taoli@ece.ufl.edu, nil@ieee.org.*
- *Lizy K. John is with Department of Electrical and Computer Engineering at the University of Texas at Austin, TX, USA, E-mail: ljohn@ece.utexas.edu.*
- *Hai Jin is with College of Computer Science and Technology at the Huazhong University of Science and Technology, 430074, Wuhan, China, E-mail:jinhust@gmail.com.*
- *Junmin Wu is with Department of Computer Science and Technology at the University of Science and Technology of China, 230026, Hefei, China, E-mail: jmwu@ustc.edu.cn.*

Manuscript received November 19, 2013; revised August 17, 2014.

therefore propose an alternative approach in this paper.

This paper proposes GPGPU-MiniBench, a framework that generates miniature proxies of GPGPU workloads that are both accurate and fast to simulate. GPGPU-MiniBench first collects a profile to capture a GPGPU workload's execution behavior. The central component in the profile is the Divergence Flow Statistics Graph (DFSG) to characterize the control flow behavior of threads in a GPGPU kernel in a concise and comprehensive manner. Furthermore, we model shared memory bank conflicts, memory coalescing behavior, and thread hierarchy, next to a thread's control flow behavior and instruction mix. GPGPU-MiniBench generates a synthetic GPGPU kernel from this statistical profile that exhibits similar execution characteristics, yet is much shorter to simulate compared to the original GPGPU workload.

More specifically, we make the following contributions:

- We analyze why existing micro-architecture simulation acceleration techniques for CPUs and GPUs do not readily apply to GPGPU performance evaluation.
- We propose the new concept of the Divergence Flow Statistics Graph (DFSG) to characterize the dynamic control flow behavior of GPGPU kernel threads.
- We derive and characterize several typical loop patterns in existing GPGPU kernels, which we exploit to accelerate GPGPU architecture simulation.
- We develop a synthesis framework, called GPGPU-MiniBench, to generate miniature proxies of CUDA GPGPU kernels, which retain similar performance but require much shorter simulation times.
- We validate the framework on four GPGPU micro-architectures. Our experimental results show that GPGPU-MiniBench can speed up GPGPU micro-architecture simulation by a factor of 49× on average and up to 589× with an average IPC error of 4.7%.

The rest of the paper is organized as follows. Section 2 provides general background on GPGPU architectures and CUDA. Section 3 describes our characterization for 34 CUDA benchmarks and analyzes why existing simulation acceleration techniques for CPUs and GPUs are not suitable for GPGPU. Section 4 elaborates on the design of our GPGPU-MiniBench framework. Section 5 depicts our experimental methodology, while Section 6 presents evaluation results and analysis. Section 7 discusses related work and Section 8 concludes the paper.

2 Background

GPGPU refers to running General-Purpose computation on Graphics Processing Units. Enhancements in the hardware along with novel programming models such as CUDA [2], OpenCL [4], and Brook+ [9] have spurred this trend. In this paper, we consider NVidia's GPGPU architecture and the CUDA programming model without loss of generality.

TABLE 1: Execution time comparison of CUDA programs on a real NVIDIA GeForce 295 GPU device vs. the GPGPU-Sim architectural simulator.

benchmark	grid	CTA	GPU (ms)	Simulation
RPES	(65535,1,1)	(64,1,1)	0.5	> 3.8 days
TPACF	(201,1,1)	(256,1,1)	0.05	> 17 days
BLK	(480,1,1)	(128,1,1)	0.92	> 12 days
PNS	(256,1,1)	(256,1,1)	0.7	> 11 days

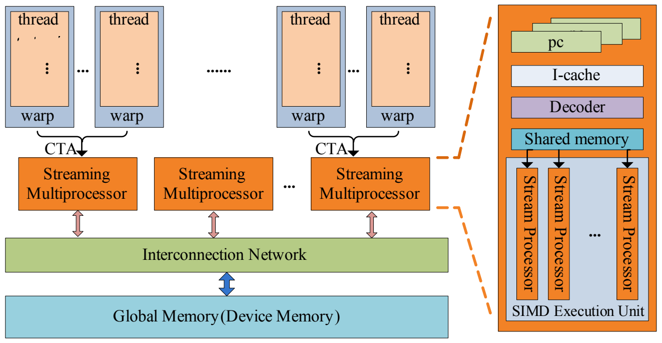


Fig. 1: GPGPU architecture overview.

Recently, NVidia introduced the Tesla, Fermi, and Kepler GPU architectures, which significantly extend GPU capabilities beyond graphics [2] [10]. Its massively multithreaded processor array becomes a highly efficient unified platform for both graphics and general-purpose parallel computing applications [10]. As shown in Figure 1, a typical GPU consists of a number of streaming multi-processors (SM, also called shader core), each containing multiple streaming processor (SP) cores. Each SM also has several special function units (SFUs), an instruction fetch and issue unit, a constant cache, and a shared memory. The number of SMs and SPs may vary across GPU generations. The SMs and the on-chip global memory interface are connected to an interconnection network.

CUDA (Compute Unified Device Architecture) is a programming model developed for NVidia GPUs, which allows programmers to write programs using C functions called kernels [2] [11]. The CUDA threads are organized in a three-level hierarchy. The lowest level is the thread itself. The next level is a group of threads called the Cooperative Thread Array (CTA) or thread block; threads in a CTA are allowed to execute concurrently on an SM, communicate via shared memory, and synchronize through barriers. At the top level, a number of CTAs are grouped together to form a grid. Threads may execute down different paths because of branches. When threads in a single hardware thread execution batch (an NVidia warp or an AMD wavefront) follow different execution paths, they must be executed sequentially — a notion called branch or warp divergence — which is harmful to performance.

A CUDA program typically consists of sequential and parallel parts. The sequential parts run on the CPU and the parallel parts run on the GPU. The parallel parts themselves consist of one or more kernels, which can run concurrently from the Fermi architecture [12] onwards. Since our goal is to accelerate GPGPU micro-architecture simulation, we focus on the parallel kernels. The sequential parts and the data passing between CPU and GPU are out of the scope of this paper.

3 Why CPU Simulation Acceleration Techniques Do not Apply to GPGPU

Before proposing our solution to the GPGPU architecture simulation challenge, we first revisit existing CPU simula-

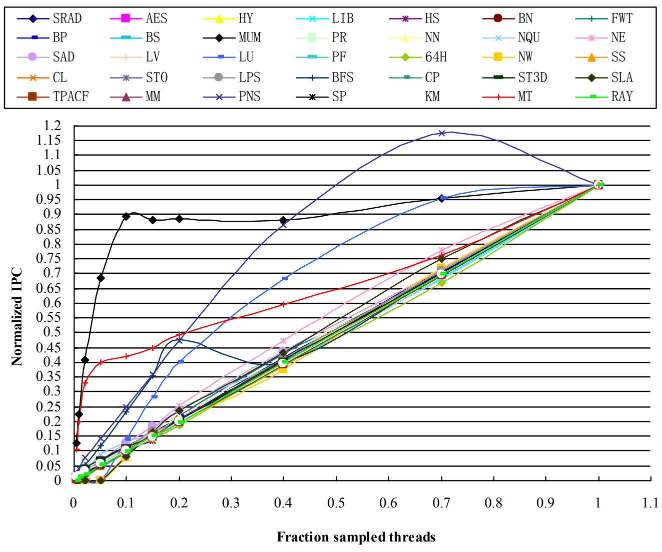


Fig. 2: Normalized IPC (vertical axis) as a function of the fraction of sampled threads (horizontal axis).

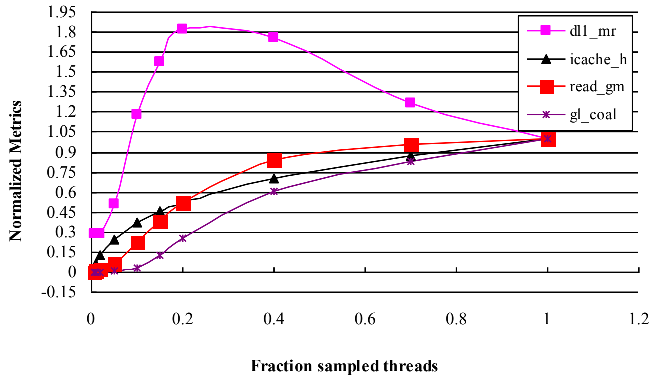


Fig. 3: Normalized performance metrics as a function of the fraction sampled threads for MUM. dl1_mr:L1 data cache miss rate; icache_h: instruction cache hit rate; read_gm: global memory read count; gl_coal: global memory coalescing stalls.

tion acceleration techniques.

3.1 CUDA Kernel Characterization

We do this by first characterizing existing CUDA kernels to provide supportive quantitative evidence during the analysis and discussion of this study. We collect the following characteristics using the GPGPU-Sim simulator [5]: instruction mix, basic block size, and the dynamic instruction count per thread. The 35 CUDA kernels that we consider in this analysis were drawn from the CUDA SDK, Rodinia and Parboil benchmark suites. (See later for a detailed description of the experimental setup.)

The number of static basic blocks in a CUDA kernel ranges from 10 to 25 for most benchmarks, with an average of 22.8 basic blocks, see Figure 4. Only two of the benchmarks, namely CL and PNS, show more than 100 basic blocks. The code footprint is substantially smaller for CUDA kernels compared to typical CPU benchmarks such as SPEC

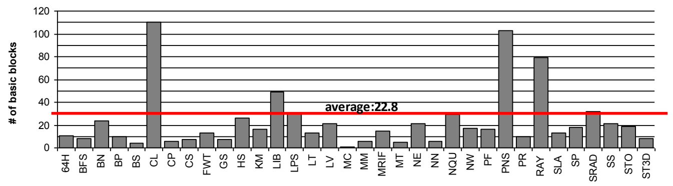


Fig. 4: Number of static basic blocks (executed at least once).

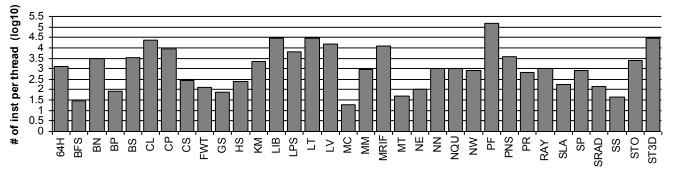


Fig. 5: Number of instructions per thread.

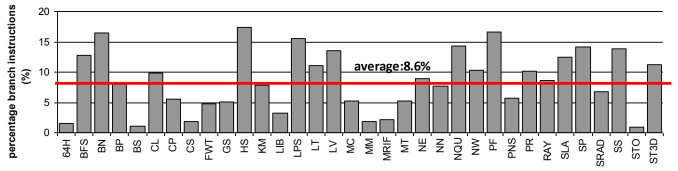


Fig. 6: Percentage dynamically executed branch instructions.

CPU and MediaBench with an average basic block count of 265.5 and 584.2, respectively [13].

Figure 5 illustrates the number of dynamically executed instructions per thread. This number typically varies from dozens to tens of thousands and is extremely small compared to SPEC CPU [14] and PARSEC [15] benchmarks, which have dynamic instruction counts in the tens to hundreds of billions range.

Furthermore, CUDA kernels feature large numbers of threads, up to hundreds of thousands of threads per kernel (i.e., unlimited for practical purposes). This is yet another key difference with typical CPU workloads, which are single-threaded or feature a limited number of threads.

3.2 Revisiting CPU Simulation Acceleration Techniques for GPGPU

There exist a number of CPU simulation acceleration techniques, which we revisit now in the context of GPGPU: sampled simulation in time and space, statistical simulation, and reduced input sets.

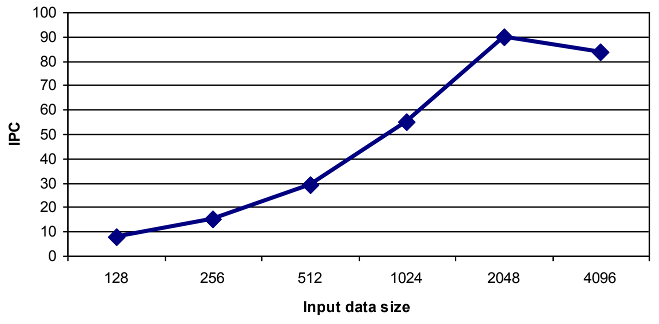
Sampling in time. Sampled simulation is a popular simulation acceleration technique for CPUs. Its basic idea is to sample the dynamic instruction stream and simulate a limited number of snapshots from the total program execution and then extrapolate performance from these snapshots to the entire program execution. Existing solutions in the CPU space sample randomly [16], periodically [17] [18], or based on application phase behavior [19]. TBPoint [20] very recently proposes sampling-in-time for GPGPU workloads. Although TBPoint achieves high accuracy while simulating 10% to 20% of the total kernel execution time, sampling workloads with high control/memory divergence behavior remains challenging. GPGPU-Minibench also targets these challenging workloads while achieving even higher speedups at small errors.

Sampling in space. An alternative to sampling in time might be to sample in space, or to simulate a limited number of threads out of the many (hundreds of thousands of) threads that constitute a GPGPU kernel. Note that sampling in space does not sample instructions from a single thread but instead samples threads from a GPGPU kernel. As the threads of a kernel may share hardware resources including global memory or interconnection network, sampling in space is likely to change several of the important GPGPU performance characteristics such as branch divergence [21] and memory coalescing behavior. In addition, the access distribution to the interconnection network, the partition camp [22], bandwidth utility and DRAM efficiency of the memory channels are also likely to be altered when reducing the number of threads. We experimentally confirm this as illustrated in Figure 2 which shows IPC as we (randomly) sample threads, for our 35 benchmarks. For sampling in space to work, we would need to observe a linear relationship between the sampled IPC and the fraction of sampled threads. This seems to be the case for many benchmarks. Unfortunately, not all benchmarks exhibit this behavior, see for example MUM. Figure 3 explains why: the L1 data cache miss rate, instruction cache hit rate, the global memory read count, and number of stalls caused by global memory coalescing does not change linearly with sampled thread count. (We also tried sampling warps instead of threads, and found it not to work either.) In summary, sampling in space works for many benchmarks, but not all, hence it is not a generally applicable technique. (Sampling in space by sampling a few cores and scaling down global resources such as memory bandwidth to preserve global resource contention may be a possible solution but falls out of the scope of this paper.)

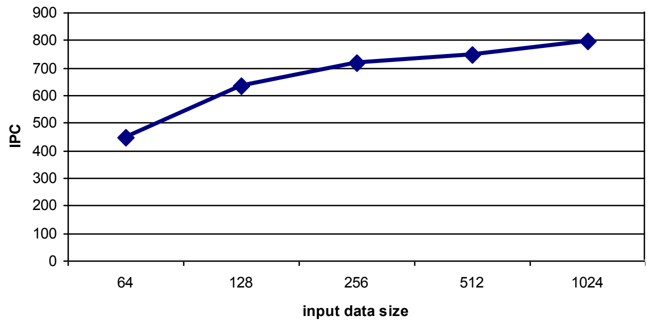
Statistical simulation. Statistical simulation reduces simulation time by generating a small synthetic program with the same behavioral characteristics as the original workload [23] [24]. The basic idea during the synthesis phase is to reduce the number of times each basic block is executed proportional to the execution of the basic block in the original workload. Given the small dynamic instruction count per thread and the small number of basic blocks in a GPGPU kernel, this approach is unlikely to work for GPGPU kernels, as infrequently executed basic blocks get eliminated during the synthesis process.

Reduced inputs. Reducing the input data set is yet another popular and easy-to-apply approach for reducing simulation time. A significant concern with reduced input sets for GPGPU workloads is that performance (IPC) heavily correlates with problem size, as illustrated in Figure 7 for the NW and HS benchmarks. (We made similar observations for the other benchmarks.) Hence, reducing input size may affect thread count, its interactions through shared resources and the workload's branch and memory behavior. KleinOsowski et al. [25] reported how challenging input set reduction is for CPUs; we expect input set reduction to be even more challenging in the GPGPU context.

GPU spreadsheet modeling. Current practice in GPU pre-silicon performance evaluation using graphics workloads — typically spreadsheet-based analysis — does not apply to GPGPU either. GPGPU workloads exhibit more irregular code than typical graphics workloads, which is apparent from its branch behavior, as illustrated in Figure 6. Although the percentage of branches (8.6% on average) is low compared to SPEC CPU programs [26], it is high compared to graphics applications: an average number of 4.5% of all dynamically executed instructions are branches in graphics benchmarks (OpenCL programs from NVidia GPU Computing SDK [27]), see Figure 8.



(a) NW (Needleman-Wunsch)



(b) HS (HotSpot)

Fig. 7: Performance (IPC) of CUDA benchmarks as a function of input data set size.

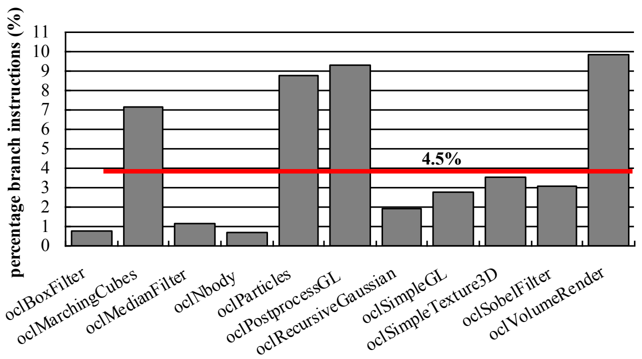


Fig. 8: The percentage of branch instructions in typical graphics GPU kernels.

4 GPGPU Benchmark Synthesis

We now propose GPGPU workload synthesis to address the GPGPU simulation challenge. Our framework, called

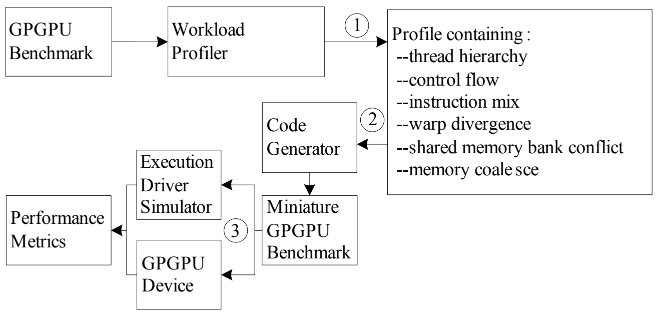


Fig. 9: The GPGPU-MiniBench synthesis framework.

GPGPU-MiniBench, generates miniature proxies of real GPGPU kernels that are both representative and fast to simulate. GPGPU-MiniBench consists of three steps as shown in Figure 9. In the first step, a profile is collected by capturing the threads' inherent execution characteristics by executing the GPGPU workload with a given input within the profiler. Subsequently, the profile is used as input to a code generator to generate a synthetic miniature GPGPU benchmark; the original benchmark's input is contained in the synthetic kernel clone. In the final step, the synthetic benchmark is simulated on an execution-driven architectural simulator, such as GPGPU-Sim. The final goal of GPGPU-MiniBench is to generate miniature GPGPU kernels that are representative of the original kernels, yet run for a shorter amount of time, yielding significant simulation speedups.

Note that we focus on synthesizing a miniature proxy for a single kernel with a large input data set. If a GPGPU benchmark consists of multiple kernels or the kernels are executed multiple times, GPGPU-MiniBench can be easily applied to each kernel (execution).

4.1 GPGPU Kernel Profiling

During kernel profiling, we collect a number of program characteristics that collectively capture the inherent execution characteristics of GPGPU workloads. These characteristics are such that they enable reusing synthetic miniature benchmarks across GPGPU micro-architectures. Profiling is a one-time cost and is done using a heavily modified version of CUDA-Sim, which is 5 to 15 times faster compared to timing simulation. We now describe the collected characteristics in more detail.

4.1.1 Thread Hierarchy

As mentioned in Section 2, the batch of threads that execute a kernel is organized as a grid of CTAs. The thread hierarchy characteristic controls the amount of thread-level parallelism of a kernel and how threads are assigned to streaming multiprocessors (SM). Since reducing the number of threads is likely to change the overall performance of a kernel as discussed in Section 3, we maintain the same grid and CTA dimensions in the synthetic as in the original kernel.

4.1.2 Instruction Mix

For GPGPU, different instructions may have dramatically different throughput [28] [29]. Note that the instruction throughput is defined as the number operations executed per clock cycle per multiprocessor. For example, for NVidia's compute capability 2.0, the throughput of the native 32-bit floating-point *add* instruction is 32, while it is only 4 for the 32-bit floating-point instructions *reciprocal* and *reciprocal square root* [28]. Preserving the instruction mix is thus important to accurately mimic the performance of a GPGPU kernel. We therefore profile instruction opcodes and data types, which are used to fill in instructions in the basic blocks of the synthetic clone. Additional information must be collected for branch instructions such as *setp* and *bra*, and memory instructions such as *ld* and *st*. More details will be presented in the following paragraphs regarding branch memory behavior.

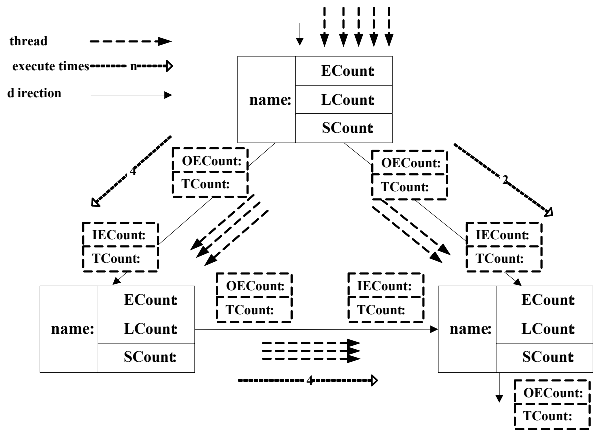


Fig. 10: The Divergence Flow Statistics Graph.

4.1.3 Control Flow

Capturing the control flow behavior for each thread by collecting a trace for each thread would be prohibitively costly because of the massive number of threads in a GPGPU kernel. CUDA-Sim for example, simulates each thread independently, rather than grouping them into warps and running warp-instructions in lockstep as the actual hardware does; this makes it difficult to collect warp divergence information without generating traces. We therefore propose the Divergence Flow Statistics Graph (DFSG) to characterize the control flow behavior of a kernel in a concise and comprehensive way.

Figure 10 illustrates the DFSG. The nodes (solid line boxes) represent basic blocks. Each node contains four fields: name, ECount, LCount, and SCount. The name field is a basic block's ID. ECount is the execution count denoting how many times a basic block is executed by all threads. LCount is the loop count and quantifies how many times the given basic block was iterated over in a loop by all threads. We use SCount to count how often synchronizations happen at the basic block.

The edges (solid arrows) represent the jumps between basic blocks. A dash box near the start of a solid arrow denotes the statistics of a basic block's out-edge. A box near the end of an arrow represents the basic block's in-edge. OECount and IECount represent how often control flow leaves or jumps to the basic block by all threads, respectively. TCount denotes the number of threads executing the edge. Note that because of loops, the number of times an edge is executed (OECount or IECount) does not need to be equal to the number of threads executing the edge (TCount); TCount counts the number of threads executing the edge at least once, and OECount and IECount count how often the edge is executed by any thread.

4.1.4 Shared Memory Bank Conflicts

Shared memory is as fast as a register on a typical GPU. However, the performance of shared memory decreases dramatically if there are bank conflicts between threads [2] [21] [30]. In order to preserve similar shared memory bank conflict behavior in the synthetic clone, we need to clone the shared arrays and their access behavior from the original to the synthesized version. We therefore profile the shared arrays which includes collecting (1) the number of shared arrays, (2) the data type and size of each array, and (3) the way the arrays are accessed. The latter is done by maintaining the array's base address and its index, which is a function of the thread's ID. By doing so, we reproduce the same memory access patterns as long as the address is an affine expression of the thread ID.

4.1.5 Memory Coalescing

GPUs provide a memory coalescing mechanism to improve memory performance by merging global memory accesses from different threads within a warp into a single memory transaction if contiguous memory addresses are accessed [2]. To clone the memory behavior, we glean the global array information in a similar way as we do for shared arrays. However, global arrays, in contrast to shared arrays, come in different forms: (i) the global arrays can be explicitly defined in the CUDA source code; or (ii) pointers to global arrays get passed as parameters to CUDA kernel functions, so-called parameter pointers, to copy data back and forth between the CPU and GPU. We account for both cases, and capture an array's base address and its index. As for shared array accesses, we reproduce the same memory access patterns as long as the address is an affine expression of the thread ID.

4.2 Code Generation

Having described how we collect a profile of a GPGPU kernel in the previous section, we now detail on how we generate a synthetic clone.

4.2.1 Loop Patterns

Our synthetic benchmark generation framework aims at faithfully mimicking control flow, branch divergence and memory access behavior. Moreover, as we will describe in detail, we leverage control flow behavior, and loops in particular, for controlling and reducing the simulation time of synthetically generated kernels. We therefore describe loop and control flow behavior in more detail now in the current and next section.

We identified three typical loop patterns from a detailed inspection of the DFSGs of the CUDA benchmarks considered in this study. Figure 11 illustrates these three typical loop patterns. (a) A self-loop consists of a single basic block that jumps to itself. (b) A normal-loop consists of multiple basic blocks in which a basic block jumps to

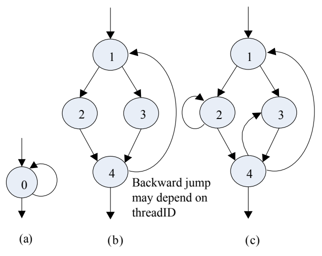


Fig. 11: The three typical loops in GPGPU kernels.

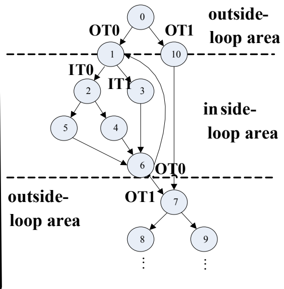


Fig. 12: Overview of divergence flow modeling.

a dominator block through a backward jump. Figure 11(c) illustrates the combination of (a) and (b): self-loops and normal-loops may be nested to form more complex loop structures. By analyzing all the experimented benchmarks, we found all loops for all benchmarks to fall under at least one of these three loop patterns (either directly or indirectly/recursively). Note that different threads executing a CUDA loop may iterate the loop for a different number of times, which leads to branch divergence behavior. In this case, the loop condition directly or indirectly depends on the thread's ID. This situation is quite common in CUDA benchmarks based on our observation.

4.2.2 Divergence Behavior Modeling

To achieve similar performance between the synthetic clone and the original GPGPU kernel, it is crucial to preserve its divergence and control flow behavior. In addition, it is also important that the synthesized kernel exhibits similar basic block execution behavior to that of the original kernel. GPGPU-MiniBench models these two aspects through the DFSG as previously introduced. Figure 13 shows the DFSG of the Kmeans (KM) benchmark as an example. We can easily derive how many threads execute a particular basic block from the DFSG. For example, basic block #0 is executed 57,600 times by 57,600 threads of which 51,200 threads jump to basic block #1 and the remaining 6,400 threads jump to basic block #12. For the ease of reasoning, we partition the control flow graph of a kernel into two parts, namely the outside- and the inside-loop area as shown in Figure 12. We define the basic blocks located outside any loop to be part of the outside-loop area; all other basic blocks are located in the inside-loop area. The modeling of the outside-loop area is relatively easy whereas the modeling of the inside-loop area is slightly more complicated.

In the discussion to follow, we define the global thread ID as:

$$gTID = bNInG \times tPB + tNInB \quad (1)$$

with $gTID$ the global thread ID, $bNInG$ the index of the given thread block within the grid, tPB the number of threads per block, and $tNInB$ the index of the given thread in its block. We consider 3 dimensions (3D). The 2D and 1D case can be derived from the 3D case by setting the corresponding value(s) to 0. The thread index within its block equals:

$$tNInB = tid.x + tid.y \times ntid.x + tid.z \times ntid.x \times ntid.y \quad (2)$$

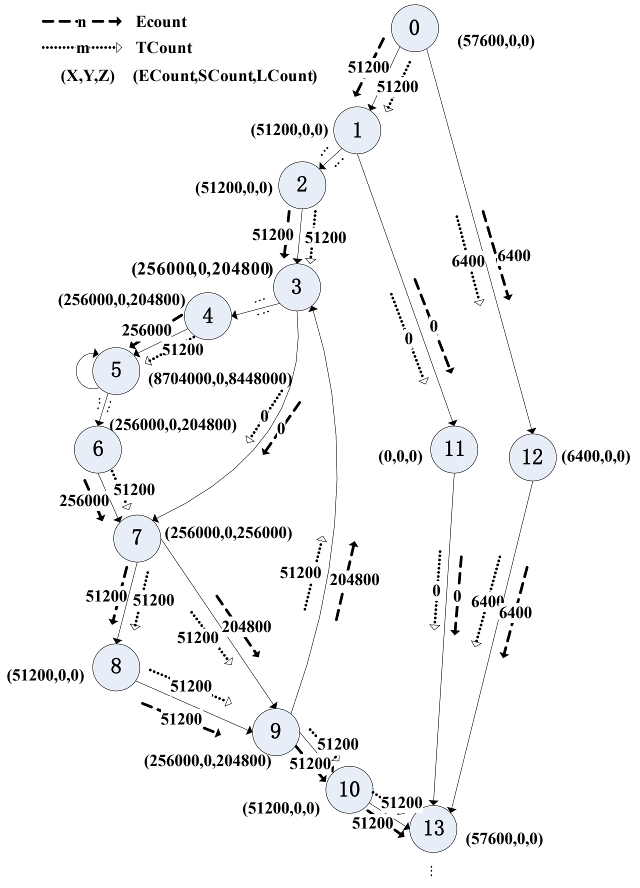


Fig. 13: The DFSG of the Kmeans benchmark.

4.2.2.1 Outside-loop area

For branches in the outside-loop area, we simply assume that the first N threads go in one direction, while the remaining threads go the other way. (We find this simple heuristic to work well for our workloads.) Knowing how many threads jump in one or the other direction is easily derived from the DFSG.

4.2.2.2 Inside-loop area

Modeling branches in the inside-loop area is a bit more complicated. We describe the modeling of the inside-loop in three cases: (1) self-loops, (2) jump point of a normal-loop, and (3) target point of a normal-loop.

1) Self-loop

Based on our observation, most of the self-loops are thread-independent, i.e., the number of iterations of a loop is typically constant across threads. Hence we simply set the number of iterations of a self-loop to the LCount as specified in the DFSG.

2) Jump point of a normal-loop

We define the jump point of a normal-loop as the basic block that jumps backwards. For example, in Figure 12, basic block #6 is the jump point of a normal-loop, and basic block #1 (OT0 direction) is the backward target while basic block #7 (OT1 direction) is the forward target. We make a distinction between the following cases based on the jump point's statistics.

(i) The number of threads jumping backwards equals the total number of incoming threads to the current basic block. In other words, all threads jump backwards.

(ii) The number of threads jumping backwards is less than the total number of incoming threads to the jump point. We set the first N threads to jump back-wards. (Again, we find this simple heuristic to work well for our set of workloads.)

3) Target point of a normal-loop

We define the target point of a normal-loop as the direct target of a backward jump (e.g., basic block #1 in Figure 12 is a target point). We consider four sub-cases.

(i) All threads take the same direction.

(ii) One group of threads goes to one direction and the other group of threads goes to the other direction, but the number of iterations of the two groups of threads is the same.

(iii) A number of threads go to one direction, the rest goes to the other direction, and the sum is greater than the number of incoming threads to the jump point. This indicates that the number of iterations varies across threads, which we model as such.

(iv) The branch condition does not depend directly on the thread ID but depends on a result calculated by the thread itself (indirect thread ID dependence). This is the same as the branches in traditional single-threaded programs. In this case, we use a uniform random number generator to generate the branch condition at each iteration (while making sure we obey the branch statistics, i.e., the number of threads going in either direction is identical to the original kernel).

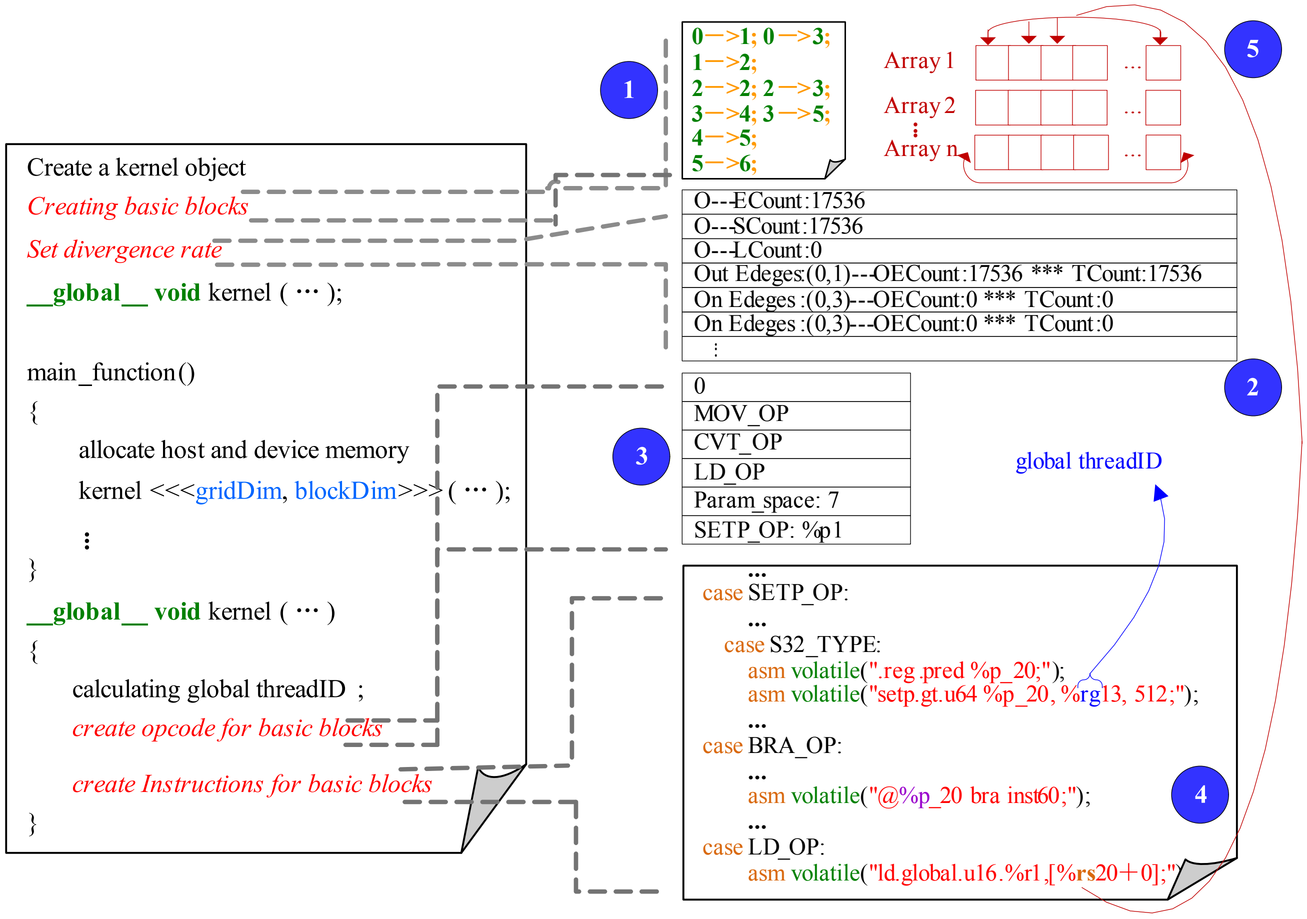
4.2.3 *Reducing Simulation Time*

The existence of loops in CUDA threads provides us with an opportunity to reduce simulation time while preserving performance and behavior. The central idea to our approach is to reduce the number of iteration counts by a given factor, called the *reduction factor*. The basic intuition is that reducing loop iteration count reduces simulation time of a kernel while preserving execution behavior in terms of the number of instructions executed per cycle (IPC). One key question now is how to determine the *reduction factor*. This is non-trivial given there are multiple loops in a kernel that are often nested; hence the question is, should we consider a single *reduction factor* across all loops, or should we determine *reduction factors* on a per-loop basis? Also, how should we determine the *reduction factor*?

The maximum possible *reduction factor* can be easily derived from the DFSG as the maximum LCount in the graph. The maximum simulation speedup is thus obtained by setting the iteration count to one for all loops. While we found this to be fairly accurate for most benchmarks while yielding high simulation speedups, accuracy can be improved significantly for some benchmarks by choosing a lower reduction factor (at the cost of slower simulation speed). We will evaluate the impact of the *reduction factor* on accuracy and simulation speed in the evaluation section.

4.2.4 *Code Generation Algorithm*

We now describe the various steps to generate a synthetic miniature kernel. Figure 14 shows the overview of our code generation framework. We follow a top-down hierarchy: kernel, basic block, and instruction. There are also auxiliary



```
Create a kernel object
Creating basic blocks
Set divergence rate
__global__ void kernel ( ··· );

main_function()
{
      allocate host and device memory
      kernel <<<gridDim, blockDim>>> ( ··· );
      ⋮
}
__global__ void kernel ( ··· )
{
      calculating global threadID ;
      create opcode for basic blocks

      create Instructions for basic blocks
}
```

```
...
case SETP_OP:
   ...
   case S32_TYPE:
      asm volatile(".reg .pred %p_20;");
      asm volatile("setp.gt.u64 %p_20, %rg13, 512;");
   ...
case BRA_OP:
   ...
      asm volatile("@%p_20 bra inst60;");
   ...
case LD_OP:
      asm volatile("ld.global.u16.%r1,[%rs20+0];");
```

Fig. 14: Overview of GPGPU synthetic workload generation.

codes generated between the different levels in the hierarchy to set information such as kernel parameters and divergence rate to control the execution of the synthesized kernel.

1) Create a kernel object to represent a CUDA kernel.
2) Set the parameter information for the kernel, especially the parameter pointers, the data types and the order of the parameter pointers. This is done in order to clone the memory coalescing behavior of the original kernel to the synthesized one, as described in Section 4.1.
3) Create basic blocks based on a .dot file as shown in box ① in Figure 14. The .dot file is obtained through profiling, and captures the control flow graph of the original kernel.
4) Set the divergence rate for each branch in the created control flow graph. The settings are based on the models described in Section 4.2.2 (box ②). If the branch is a loop branch, we set its loop count to LCount/reduction factor.
5) Generate kernel prototype code. This uses the parameter information set by step 2.
6) Generate the main function code. In the main function body, generate code to allocate host and device memory, copy data from host to device, call the kernel, and copy the results back to the host.
7) Generate code to define global arrays, if any.
8) Generate kernel definition code. Use the following steps to generate code inside the kernel body.
9) Generate code to define shared arrays, if any.
10) Generate code to calculate the global thread ID using Formula (1).
11) Generate code to define loop control variables and initialize them. This is based on step 4.
12) Fill opcode for basic blocks based on the opcode profiling described in Section 4.1. The output of the profiling is a file that contains the opcode chain. This file is one of the inputs of our code generator. Box ③ in Figure 14 shows an example.
13) Emit instructions. The *setp* and *bra* instructions determine the kernel's control flow behavior. The *setp* instruction sets branch conditions and *bra* executes the jump. The conditions set by *setp* are based on the models described in Section 4.2.2 and the information set by step 4. When emitting *ld* or *st* instructions, we need to check if these instructions are used to access shared or global arrays. For this step, we employ the distance information described in Section 4.1 ('shared memory bank conflicts' and 'memory coalescing'). The distance information is stored in the output file, see box ③ of Figure 14.

As shown in the box ④ of Figure 14, each instruction is emitted with assembly code using *asm* statements embedded in CUDA code [31]. The use of the *volatile* directive for each *asm* statement prevents the compiler from modifying these machine instructions, i.e., the code emitted by the compiler is exactly what the framework generates. The instructions are targeted towards a specific Intermediate Language (IL), PTX in our case. However, the code generator can be easily modified to emit instructions for any IL of interest. The generated code is compiled by the *nvcc* compiler [32] from NVidia and the binary can run on execution-driven GPGPU simulators and real GPGPU devices.

As an example, box ④ of Figure 14 also shows a part of the synthesized code. The *setp* and *bra* instructions control the divergence flow of the synthesized kernel. In this example, when the global thread ID is greater than 512, the execution jumps to *instr60* which is the first instruction of the target basic block. Otherwise, the execution goes to the next basic block. This dictates that the first 512 threads go to the next basic block while the other threads jump to the target basic block. The *ld* instruction in the box ④ illustrates how we control the memory coalescing ratio. %r20 is a register value related to the thread ID. If the element size of the global arrays shown in box ⑤ equals 4, register %r20 then equals the product of thread ID and 4. This makes the global memory accesses by threads within a warp to be coalesced.

Note that the proposed framework does not handle cache effects explicitly. This may need to change in the future as GPGPU kernels get optimized better for cache performance. For our workloads, we did not observe kernel performance to be highly sensitive to cache performance.

5 Experimental Setup

As previously mentioned, we analyzed instruction, basic block, and thread characteristics for 35 benchmarks from the CUDA SDK [27], Parboil [33], and Rodinia [34] benchmark suites. However, to reduce overall simulation time, we limit ourselves to 23 benchmarks in total. 15 benchmarks were randomly selected out of the top-25 long-running benchmarks, see Figure 15. The other 8 benchmarks (MUM, PNS, LU, MT, BFS, NW, SLA, and NE) are those that could not be accelerated well through sampling in space, according to Figure 2. Table 2 lists these 23 benchmarks along with their thread count and number of instructions per thread.

We use the GPGPU-sim_v2.1.1b [5] to evaluate our approach. The simulator is configured to evaluate the four GPU micro-architecture configurations with different numbers of streaming multiprocessors (Table 3).

6 Evaluation

In this section, we first evaluate the efficacy of GPGPU-MiniBench. Subsequently, we compare it with an approach that sets the number of loop iterations to one in the source codes of the CUDA workloads.

6.1 Evaluation of GPGPU-MiniBench

We consider the following factors in the evaluation of GPGPU-MiniBench: (i) the impact of the reduction factor on simulation speed and accuracy, (ii) achieved simulation speedup, (iii) accuracy (or IPC error), and (iv) other metrics such as shared memory bank conflicts, memory coalescing and branch divergence behavior.

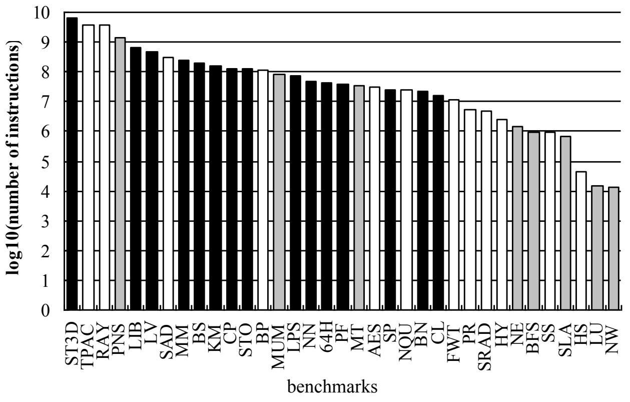


Fig. 15: Total dynamic instruction count for our 35 benchmarks, from which we randomly select 15 out of the top-25 long-running benchmarks(black bars); 8 more benchmarks, for which sampling in space fails, are included as well(gray bars).

benchmark	Abr.	# of threads	Insts/thread
MumerGPU (DNA)	MUM	50,176	739
Petri-Net Simulation	PNS	51,200	3,884
LU Decomposition	LU	20,480	976
Matrix Transpose	MT	4,194,304	48
Breadth First Search	BFS	32,768	30
Needleman-Wunsch	NW	40,000	831
Scan(Para-Prefix sum)	SLA	1,310,720	181
Nearest Neighbor	NE	60,032	101
Histogram 64	64H	17,536	1,310
Black-Scholes	BS	61,440	3,189
Cellular Automation	CL	512	24,083
3D Laplace Solver	LPS	12,800	6,385
K means	KM	57,600	2,119
LIBOR Monte Carlo	LIB	4,096	122,068
3D stencil computation	ST3D	230,400	10,471
Magnetic Resonance Imaging FHD	MRIF	32,768	11,883
Scalar Product	SP	32,768	795
Particle Filter	PF	256	153,622
Matrix multiplication	MM	256,000	950
Neural Network	NN	113,568	1,026
Leukocyte Tracking	LT	307,420	27,816
Levenshtein Distance	LV	32,768	16,065
Store GPU	STO	49,152	2,517

TABLE 2: The CUDA benchmarks used in this paper, the # of instructions per thread and the total # of threads.

We define simulation speedup as follows:

$$speedup = \frac{simulation\_time_{original}}{simulation\_time_{synthetic}} \quad (3)$$

with $simulation\_time_{original}$ the time to simulate the original benchmark on the GPGPU performance simulator, and $simulation\_time_{synthetic}$ the time to simulate the synthesized benchmark clone on the same simulator.

We define the relative error for a metric M as

$$RE_M = \frac{M_{syn} - M_{ori}}{M_{ori}} \quad (4)$$

with M_{syn} and M_{ori} obtained by simulating the synthetic and original benchmark, respectively.

Configurations	Conf1	Conf2	Conf3	Conf4
# of SMs	8	28	56	110
Warp size	32			
SIMD Pipeline Width	8			
# of Threads/SM	1024			
# of CTAs/SM	8			
# of Registers/SM	16384	16384	32768	32768
Shared Memory/SM (KB)	16 (16 banks, 1 access/cycle/bank)			
Constant Cache Size/SM	8KB (2-way set assoc, 64 lines)			
Texture Cache Size/SM	64KB (2-way set assoc, 64 lines)			
# of Memory Channels	8	8	8	8
L1 Data Cache	128KB	128KB	256KB	256KB
L2 Cache	None			
Bandwidth Per Memory Module	8 (Bytes/Cycle)			
DRAM Request Queue Capacity	32			
Memory Controller	FR-FCFS			
Branch Divergence Method	Immediate Post Dominator			
Warp Schedule Policy	Round Robin among read warps			

TABLE 3: Four GPGPU architecture configurations. SM — Streaming Multiprocessor.

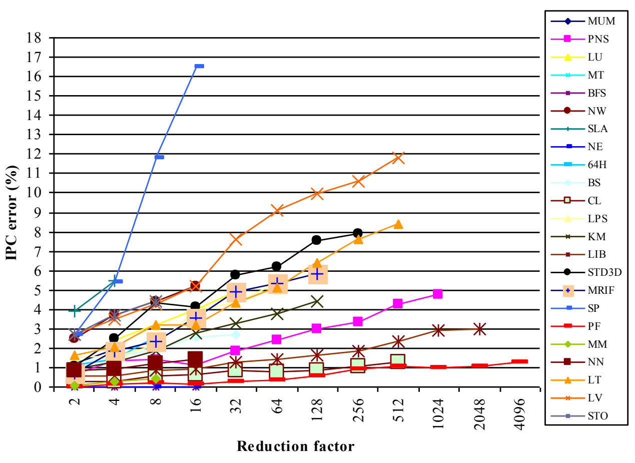


Fig. 17: The impact of reduction factor on IPC error.

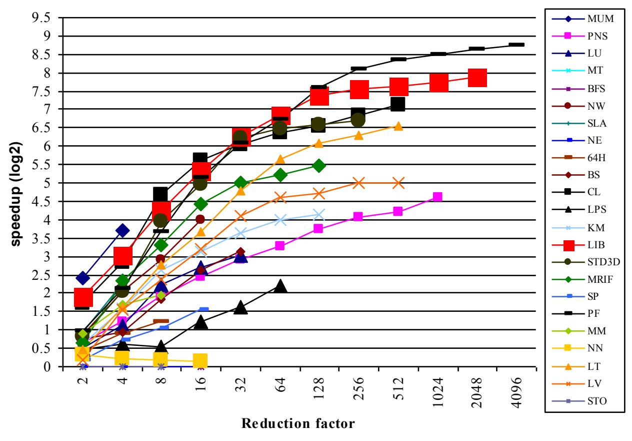


Fig. 16: The impact of reduction factor on speedup.

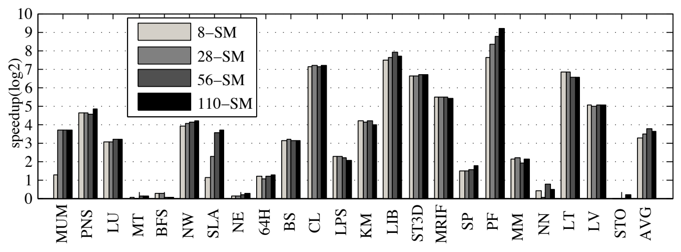


Fig. 18: Achieved simulation speedup (log scale) through GPGPU-MiniBench for our four architectures with varying numbers of of Streaming Multiprocessors (SM).

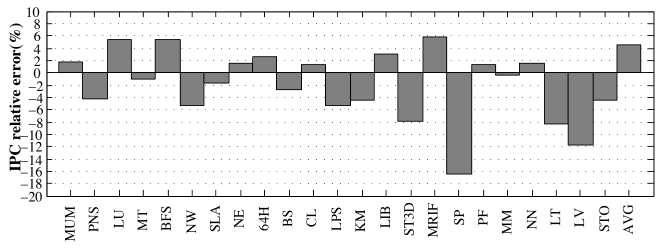


Fig. 19: The IPC error for the 56-SM configuration.

(i) Impact of reduction factor on simulation speed and accuracy. As previously mentioned, GPGPU-MiniBench reduces simulation time by decreasing loop iteration counts using a reduction factor. In this section, we study how this reduction factor affects GPGPU architecture simulation speed and accuracy. The values of reduction factors are 2, 4, 8, etc., in powers of 2. The maximum reduction factor of a benchmark depends on its maximum number of loop iterations. In our approach, the maximum number of loop iterations is divided by the reduction factor and the rounded down value of this result is set as the loop iteration count in the synthesized benchmark. If the rounded down value of the result is 1, then the maximum reduction factor is obtained. Because loop iteration counts are workload-specific, different benchmarks may have different maximum reduction factors.

Figures 16 and 17 quantify speed and accuracy, respectively, as a function of the reduction factor. Figure 16 shows that a larger reduction factor results in more speedup for all the benchmarks, while Figure 17 reveals that a larger reduction factor incurs higher IPC error. Apart from two benchmarks (SP and LV), the maximum IPC error is less than 8.5%. This indicates that we can choose the maximum reduction factor to achieve the largest possible simulation speedup while preserving good accuracy.

(ii) Simulation speed. Figure 18 shows simulation speedup for our benchmarks when their maximum reduction factors are employed. The maximum speedup is 589$\times$ for the PF benchmark. Looking at Table 2, the number of dynamic instructions per thread for this benchmark is about 150K. This result demonstrates that GPGPU-MiniBench is most effective at reducing simulation time for benchmarks with large per-thread instruction counts. On average, our approach can speed up GPGPU architecture simulation by a factor 40$\times$, 46$\times$, 52$\times$ and 58$\times$ for the 8-SM, 28-SM, 56-SM, and 110-SM configurations, respectively. The harmonic mean speedup equals 49$\times$.

Interestingly, we also obtain significant speedups for 5 of the 8 benchmarks for which sampling in space does not work. Our approach does not accelerate the simulation for MT, BFS, and NE, as these benchmarks do not execute any loops. Hence, both sampling in space and our approach

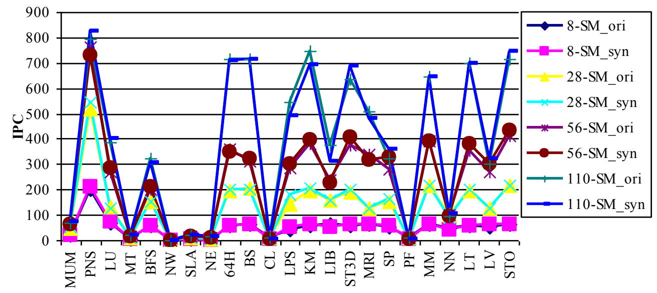


Fig. 20: IPC for four architectures for the synthetic and original workloads.

do not work well for these three benchmarks. How to speed up the GPGPU acceleration for these benchmarks is subject to future work.

(iii) IPC error. Figure 19 shows the relative IPC error of the synthesized versus original benchmarks for the 56-SM configuration. The absolute average error equals 4.5%. For the 8 benchmarks for sampling in space failed to work, we obtain an average error of 3.3%, and no error higher than 6.2%, which indicates that our approach outperforms sampling in space.

Note that we observe both positive and negative errors — for some benchmarks, GPGPU-MiniBench yields a performance overestimation, and for others it yields an underestimation — this indicates there is no systematic bias in the modeling framework. We analyzed the reasons for the errors, and we found these errors are due to various simplifying assumptions made in the framework. In particular, the number of loop iterations of a GPGPU kernel may vary across threads, which our model does not capture as it assumes that all threads iterate loops for the same number of times. Furthermore, we make the assumption that we preserve execution characteristics as we reduce the number of loop iterations; note we do this on purpose to reduce simulation time, yet it incurs some inaccuracy.

Figure 20 quantifies accuracy across four different GPU architectures — this is to illustrate how accurate GPGPU-MiniBench is to analyze relative performance differences across the GPU design space. There are a couple of observations to be made here. First, this graph reconfirms the accuracy reported in Figure 19 across different GPU architectures, i.e., the synthetic performance results closely match the ones for the original workload. Second, GPGPU-MiniBench is able to accurately track relative performance differences across architectures and workloads. For example, the performance difference seems to be small between the four GPU architectures for the CL, PF and NN benchmarks. For some benchmarks, performance improves with increasing number of SMs but stagnates beyond 56 SMs, see for example LIB, SP and LV; for the other benchmarks, performance continues to improve with an increasing number of SMs. GPGPU-MiniBench captures all of these trends accurately.

(iv) Other metrics. We considered other metrics next to IPC to evaluate the synthesis framework, namely shared memory bank conflicts, memory coalescing behavior and branch divergence rates. We find the synthetic clones to accurately mimic these metrics compared to the original workloads, in spite of the simplifying assumptions we make in the framework.

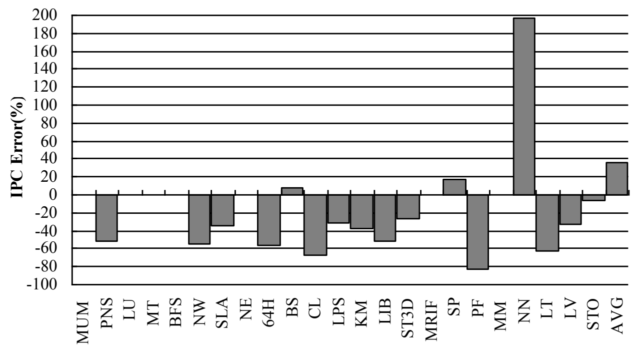


Fig. 21: IPC error obtained by reducing the number of loop iterations to one in the source code.

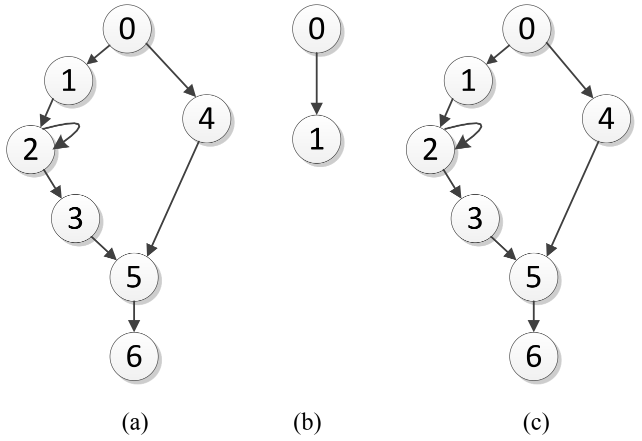


Fig. 22: The static control flow graphs of the benchmark MM with 3 versions. (a) The original version;(b) the loop reduction version;(c) the synthesized version.

6.2 Comparison against loop reduction

Since GPGPU-MiniBench reduces simulation time by reducing the number of loop iterations in the synthesized code to one, one may think that setting the number of loop iterations to one in CUDA source code directly might be equally accurate while being much simpler to implement. Unfortunately, loop reduction does not work well for most CUDA benchmarks, as shown in Figure 21. The IPC error for most benchmarks is very high, with an average IPC error as high as 34.4%.

The reason is that the CUDA compiler changes the control flow of a workload when we set the number of loop iterations to one in the source code which leads to non-representative code. GPGPU-MiniBench on the other hand preserves control flow behavior of the synthesized code compared to the original code. Figure 22 shows the control flow graph of the benchmark MM as an example. As can be seen, loop reduction significantly changes the static control flow graph of MM, while GPGPU-MiniBench does not.

7 Related Work

Micro-architecture simulation is a key tool for computer architecture research and development. A lot of research has been done towards accelerating CPU simulation, see for example [17] [18] [19] [35]. Only recently has interest grown regarding GPGPU architecture simulation, as exemplified by GPGPU-Sim [5] [8], Ocelot [6] [21] and Barra [7]. These simulators are critical to GPGPU architecture research, but they are slow. To the best of our knowledge, we are the first to attempt to accelerate GPGPU architecture simulation.

Recently, Huang et al. accelerate GPGPU architecture simulation by sampling thread blocks [20] using TBPoint. Sampling thread blocks is a good idea since CUDA encourages programmers to write programs with little communication between thread blocks. Although TBPoint achieves high accuracy while simulating 10% to 20% of the total execution time of the kernel (simulation speedup of 5 to $10\times$), sampling workloads with high control/memory divergence behavior remains challenging. GPGPU-Minibench also targets these challenging workloads with a very different approach, and achieves high accuracy and higher simulation speedups (by $49\times$ on average). Lee et al. tried to parallelize the GPGPU architecture simulation [36]. However, the speedup of their approach only achieves up to $4.15\times$.

Synthesizing workloads/traces for performance evaluation has been an active area of research. Several recent studies report good accuracy and significant simulation speedups using synthetically generated benchmark clones over running full workloads on cycle-accurate simulators [13] [23] [37] [38] [24]. However, all of this prior work focused on long-running CPU programs. For parallel programs, especially GPGPU kernels, it is very difficult to apply the CPU synthetic techniques, as argued in Section 2. We carefully analyze the unique characteristics of GPGPU kernels and propose GPGPU-MiniBench as an alternate workload synthesis approach for accelerating GPGPU architecture simulation.

8 Conclusion

Slow micro-architecture simulation speed has been a major and constant concern for several decades in the CPU domain, and now with the emergence for GPGPU computing, there is a strong need for simulation acceleration techniques for GPGPU. In this paper, we have argued that existing CPU simulation acceleration techniques do not readily apply to GPGPU. We therefore proposed a very different approach in this paper. GPGPU-MiniBench generates synthetic clones of real GPGPU workloads that exhibit similar execution characteristics (within 4.7% on average) as the original workloads while being much shorter to simulate, yielding a simulation speedup of a factor $49\times$ on average. GPGPU-MiniBench is accurate enough for making high-level design decisions and trend analyses in GPGPU architectures and systems.

Acknowledgments

This work is supported by the National Basic Research Program of China (973 program) under grant No. 2015CB352402 and 2015CB352403, the National Science Foundation of China under grant No. 61272132, the CAS special program for the next generation of information technology for sensing China under grant No. XDA06010500, and in part by NSF under grant CCF-1016966. This work was also supported in part by the European Research Council under the European Communitys Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement No. 259295.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Proceedings of the Eurographics 2005*. The Eurographics Association, 2005, pp. 21–51.

[2] "Cuda programming guide, version3.0," *NVIDIA CORPORATION*, 2010.

[3] "Ati stream technology," *Advanced Micro Devices,Inc. http://www.amd.com/stream.*, 2011.

[4] "Opencl," *Khronos Group. http://www.khronos.org/opencl.*, 2012.

[5] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*. IEEE Computer Society, 2009, pp. 163–174.

[6] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques(PACT)*. ACM Press, 2010, pp. 353–364.

[7] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *Proceedings of 18th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS)*. IEEE Computer Society, 2010, pp. 351–360.

[8] "The latest gpgpu-sim user manual," *http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual.*, 2013.

[9] "Advanced micro devices, inc. amd brook+." *http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf.*, 2011.

[10] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue GPU Computing*, vol. 6, no. 2, pp. 40–53, 2008.

[12] "Nvidia corporation. nvidias next generation cuda computer architecture: Fermi," *NVIDIA White Paper, v1.1*, 2009.

[13] A. Joshi, L. Eeckhout, R. H. B. JR., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 2, pp. 10:1–10:33, 2008.

[14] "http://www.spec.org/cpu/."

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architecture implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM Press, 2008, pp. 72–81.

[16] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss of effective trace sampling of superscalar processors," in *Proceedings of International Conference on Com-puter Design (ICCD)*. IEEE Computer Society, 1996, pp. 468–477.

[17] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, 2003, pp. 84–95.

[18] Z. Yu, H. Jin, J. Chen, and L. K. John, "Tss: Applying two-stage sampling in micro-architecture simulation," in *Proceedings of 17th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS)*. IEEE Computer Society, 2009, pp. 463–471.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 2002, pp. 45–57.

[20] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, "Tbpoint: Reducing simulation time for large-scale gpgpu kernels," in *Proceedings of the IEEE Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2014, pp. 437–446.
[21] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2009, pp. 3–12.
[22] G. Ruetsch and P. Micikevicius, "Optimizing matrix transpose in cuda," *NVIDIA*, 2009.
[23] L. Eeckhout, R. H. B. Jr, B. Stougie, K. D. Bosschere, and L. K. John, "Control flow modeling in statistical simulation for accurate and efficient processor design studies," in *Proceedings of the 31st Annual International Symposium on Computer Architecture(ISCA)*. IEEE Computer Society, 2004, pp. 350–361.
[24] V. S. Iyengar, L. H. Trevillyan, and P. Bose, "Representative traces for processor models with infinite cache," in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 1996, pp. 62–72.
[25] A. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based architecture research," *Computer Architecture Letters*, vol. 1, no. 1, pp. 7–11, 2002.
[26] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru, "Performance characterization of spec cpu benchmarks on intels core microarchitecture based processor," in *Proceedings of Austin Center for Advanced Studies Conference*. IEEE Computer Society, 2007, pp. 1–7.
[27] "Nvidia gpu computing sdk.https://developer.nvidia.com/gpu-computing-sdk."
[28] "Nvidia corporation. ptx: Parallel thread execution, isa version 2.2," 2010.
[29] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu micro-architecture through microbenchmarking," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*. IEEE Computer Society, 2010, pp. 235–246.
[30] Y. Kim and A. Shrivastava, "Cumapz: A tool to analyze memory access patterns in cuda," in *Proceedings of design automatic conference(DAC)*. ACM Press, 2011, pp. 128–133.
[31] "Nvidia corporation. using inline ptx assembly in cuda. february, 2011," 2011.
[32] "Nvidia corporation. the cuda compiler driver nvcc," 2008.
[33] "Impact. the parboil benchmark suite, 2007. [online]. available: http://impact.crhc.illions.edu/parbiol.php," 2007.
[34] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of IEEE International Symposium on Workload Characterization(IISWC)*. IEEE Computer Society, 2009, pp. 44–54.
[35] J. Chen, M. Annavaram, and M. Dubois, "Slacksim:a platform for parallel simulations of cmps on cmps," in *Proceedings of SIGARCH Computer Architecture News*. ACM Press, 2009, pp. 20–29.
[36] S. Lee and W. W. Ro, "Parallel gpu architecture simulation framework exploiting wrok allocation unit parallelism," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*. IEEE Computer Society, 2013, pp. 107–117.
[37] R. H. B. Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceeding of International Conference on Supercomputing(ICS)*. ACM Press, 2005, pp. 111–120.
[38] A. Joshi, L. Eeckhout, R. H. B. Jr., and L. K. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2006, pp. 105–115.

Zhibin Yu received his PhD degree in computer science from Huazhong University of Science and Technology (HUST) in 2008. He visited the Laboratory of Computer Architecture (LCA) of ECE of the University of Texas at Austin for one year and he worked in Ghent University as a postdoctoral researcher for half of a year. Now he is a professor in the SIAT. His research interests are micro-architecture simulation, computer architecture, workload characterization and generation, performance evaluation, multi-core architecture, GPGPU architecture, virtualization technologies and so forth. He won the outstanding technical talent program of Chinese Academy of Science (CAS) in 2014 and the 'peacock talent' program of Shenzhen City in 2013. He also won the first award in teaching contest of HUST young lectures in 2005 and the second award in teaching quality assessment of HUST in 2003. He is a member of IEEE and ACM. He serves for ISCA 2013, MICRO 2014 and HPCA 2015.

Lieven Eeckhout is a Professor at Ghent University, Belgium. His research interests include computer architecture with a specific emphasis on performance evaluation methodologies and dynamic resource management. He obtained his PhD from Ghent University in 2002. His work has been awarded with two IEEE Micro Top Pick awards and a Best Paper Award at ISPASS 2013. He published a Morgan & Claypool synthesis lecture monograph in 2010 on performance evaluation methods. He is the Program Chair for HPCA 2015, CGO 2013 and ISPASS 2009; and the Editor-in-Chief of IEEE Micro, Associate Editor-in-Chief of IEEE Computer Architecture Letters, and Associate Editor of ACM Transactions on Architecture and Code Optimization.

Tao Li is an associate professor in the Department of Electrical and Computer Engineering at the University of Florida. He received a Ph.D. in Computer Engineering from the University of Texas at Austin. His research interests include computer architecture, microprocessor/memory/storage system design, virtualization technologies, energy-efficient/sustainable/dependable data center, cloud/big data computing platforms, the impacts of emerging technologies/applications on computing, and evaluation of computer systems. Dr. Tao Li received 2009 National Science Foundation Faculty Early CAREER Award, 2008, 2007, 2006 IBM Faculty Awards, 2008 Microsoft Research Safe and Scalable Multi-core Computing Award and 2006 Microsoft Research Trustworthy Computing Curriculum Award. Dr. Tao Li co-authored a paper that won the Best Paper Award in HPCA 2011 and three papers that were nominated for the Best Paper Awards in DSN 2011, MICRO 2008 and MASCOTS 2006. Dr. Tao Li is one of the College of Engineering winners, University of Florida Doctor Dissertation Advisor/Mentoring Award for 2013-2014 and 2011-2012.

Nilanjan Goswami is architecture and modeling engineer at a leading Product Development Company. His research interest includes emerging technology based throughput processor design, power-performance co-optimization of throughput core architecture, interconnect, renewable energy based throughput architectures. He has a PhD in electrical and computer engineering from the University of Florida, Gainesville, FL.

Lizy Kurian John holds the B. N. Gafford Professorship in Electrical Engineering in the Department of Electrical & Computer Engineering at The University of Texas at Austin. She received her Ph.D. in computer engineering from The Pennsylvania State University in 1993. Her research is in the areas of computer architecture, multi-core processors, memory systems, performance evaluation and benchmarking, workload characterization, and reconfigurable computing. She is recipient of NSF CAREER award (1996), UT Austin Engineering Foundation Faculty Award (2001), Halliburton, Brown and Root Engineering Foundation Young Faculty Award (1999), University of Texas Alumni Association Teaching Award (2004), The Pennsylvania State University Outstanding Engineering Alumnus (2011) etc. Lizy John holds 8 U. S. patents and has published 16 book chapters, and approximately 200 papers. She has coauthored books on Digital Systems Design using VHDL (Cengage Publishers), Digital Systems Design using Verilog (Cengage Publishers) and has edited a book on Computer Performance Evaluation and Benchmarking (CRC Press). Prof. John is in the editorial board of IEEE-Micro and has served in the past as an associate editor of IEEE Transactions on Computers and IEEE Transactions on VLSI and. She is a member of IEEE, IEEE Computer Society, ACM, and ACM SIGARCH. She is an IEEE Fellow.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security.Jin is a senior member of the IEEE and a member of the ACM. He has co-authored 15 books and published over 500 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. Jin is the steering committee chair of International Conference on Grid and Pervasive Computing (GPC), Asia-Pacific Services Computing Conference (APSCC), International Conference on Frontier of Computer Science and Technology (FCST), and Annual ChinaGrid Conference. Jin is a member of the steering committee of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), the IFIP International Conference on Network and Parallel Computing (NPC), and the International Conference on Grid and Cooperative Computing (GCC), International Conference on Autonomic and Trusted Computing (ATC), International Conference on Ubiquitous Intelligence and Computing (UIC).

Junmin Wu received the PhD degree in computer science and engineering from University of Science and Technology of China (USTC) in 2005. He is an associate professor at the Department of Computer Science and technology of USTC, Hefei, and assistant dean of the Suzhou Institute for Advanced Study of USTC, Suzhou. His research interests include computer architecture, virtualization technology, cluster computing, multi-core computing and simulating. He has published more than 30 papers in international conferences and journals.

Cheng-Zhong Xu received his Ph.D. degree from the University of Hong Kong in 1993. He is currently a tenured professor of Wayne State University and the Director of the Institute of Advanced Computing and Data Engineering of Shenzhen Institute of Advanced Technology of Chinese Academy of Sciences. His research interest is in parallel and distributed systems and cloud computing. He has published more than 200 papers in journals and conferences. He was the Best Paper Nominee of 2013 IEEE High Performance Computer Architecture (HPCA), and the Best Paper Nominee of 2013 ACM High Performance Distributed Computing (HPDC). He serves on a number of journal editorial boards, including IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing, Journal of Parallel and Distributed Computing and China Science Information Sciences. He was a recipient of the Faculty Research Award, Career Development Chair Award, and the President's Award for Excellence in Teaching of WSU. He was also a recipient of the Outstanding Oversea Scholar award of NSFC. For more information, visit http://www.ece.eng.wayne.edu/ czxu.