GATSim: Abstract Timing Simulation of GPUs

Kishore Punniyamurthy, Behzad Boroujerdian and Andreas Gerstlauer
The University of Texas at Austin
{kishore.punniyamurthy, behzadboro, gerstl}@utexas.edu

***Abstract*—General-Purpose Graphic Processing Units (GPUs) have become an integral part of heterogeneous system architectures. Ever increasing complexities have made rapid, early performance evaluation of GPU-based architectures and applications a primary design concern. Traditional cycle-accurate GPU simulators are too slow, while existing analytical or source-level estimation approaches are often inaccurate. This paper proposes a novel abstract GPU performance simulation approach that is based on flexible separation of functional and timing models, combining a fast functional execution either on existing simulators or native GPU hardware with a light, fast and accurate abstract timing model. Micro-architecture timing of individual GPU cores is abstracted through static, one-time pre-characterization of code, and only the dynamic scheduling effects are simulated. Using a native GPU for functional execution and excluding pre-characterization, our GPU simulation achieves a throughput of more than 80 MIPS. This is on average 400x faster with 4% error compared to a cycle-accurate GPU simulator for standard GPU benchmarks. Moreover, our simple timing model provides flexibility to target different GPU configurations with little or no extra effort.**

I. INTRODUCTION

Over the years, Graphics Processing Units (GPUs) have been established as an indispensable component in modern Multi-Processor Systems-on-Chip (MPSoCs), where they are used as accelerators to perform general-purpose computations. GPU designs are continuously evolving, and the complexity of GPU-based systems and applications has made system architecture exploration and application mapping an important challenge. This drives the need for fast and accurate performance evaluation of GPU kernels on different designs.

Traditionally, system designers and application developers have used cycle-accurate simulators of various system components for accurate performance feedback of their design or code. Cycle-accurate GPU simulators [1] model the entire micro-architecture precisely to ensure accuracy, but are slow. At the other extreme, complete analytical models can be used to statically evaluate GPUs [5]. While such models usually provide significant speedup, they are often inaccurate as they fail to faithfully capture all the dynamic effects in modern architectures and applications. Recently, hybrid approaches have been proposed that couple a functional simulation with analytical timing models either in a trace-based fashion [8] or by annotating natively executed source-level simulation code [11]. Such approaches can provide fast performance evaluation, but their accuracy is limited by that of the timing information. Furthermore, in annotation-based approaches, exploring different GPU configurations requires repeatedly annotating and re-executing the source code with timing information corresponding to every new configuration.

In this paper, we propose GATSim, a fast and accurate abstract timing simulation approach for GPU performance evaluation. GATSim is based on a flexible separation of functional and timing models, where we combine a fast functional execution either on existing GPU simulators or a native host GPU with a novel, fast and accurate timing model of a target GPU. In contrast to existing hybrid approaches, GATSim uses an abstracted functional-timing interface with a timing model that combines a static analysis of larger code blocks on single GPU cores with a lightweight simulation of dynamic GPU scheduling and macro-architecture effects. This allows accurately capturing complex dynamic interactions while statically abstracting all predictable low-level micro-architecture behavior. Moreover, our timing model allows exploration across different macro-architecture configurations simply by changing parameters.

The rest of the paper is organized as follows: after an overview of related work and GATSim, Section II provides a detailed description of our abstract timing simulation approach. Our experimental setup and results are presented in Section III. Finally, the paper concludes with a summary and outlook in Section V.

A. Related Work

Researches have proposed many approaches for simulation-based, analytical or hybrid performance evaluation of GPUs in the past. The most widely-used cycle-accurate but slow GPU simulator is GPGPUSim [1]. By contrast, there is a wide range of faster analytical methods. For example, the works in [2], [3], [4] and [5] propose analytical models but their accuracy generally differs significantly across different configurations indicating that they cannot capture all dynamic effects faithfully. Instead, analytical approaches often focus on relative fidelity for micro-architecture exploration, e.g. using machine learning-based scaling models [6], [7], but this is not sufficient when absolute accuracy is desired, e.g. when evaluating the impact of GPUs in a heterogeneous system.

Many hybrid approaches have been proposed. The authors in [11] model GPUs by annotating source-level code with static analytical timing and resource usage information. This approach provides reasonable accuracy (greater than 90%) for GPU architectures under certain constraints (1 streaming multiprocessor and perfect memory). Nevertheless, the obtained accuracy is limited because of the static timing analysis. These models cannot capture all the dynamic effects in more complicated architectures faithfully. Furthermore, static models have to be re-calibrated every time any aspect of the target architecture changes. Existing trace-based approaches collect detailed instruction traces during functional simulation to be combined with analytical timing estimation.

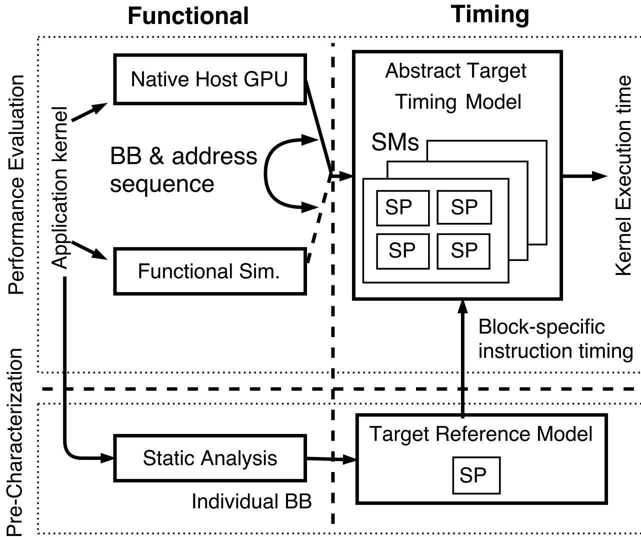

Fig. 1: GATSim overview.

The work in [10] uses instruction traces to extend worst case execution time analysis for sequential code to GPU applications. This comes with large estimation errors, but they do rightly point out that modeling concurrency is key to accuracy. TEG [8] and similarly [9] rely on an analytical timing model to estimate performance. Their approach shows good accuracy on the small number of benchmarks evaluated. However, their models do not consider synchronization effects, limiting accuracy for generic GPU kernels. Furthermore, neither [10] nor [8], [9] provide speedup numbers.

B. Overview

An overview of GATSim is shown in Fig. 1. GATSim uses an abstract timing model that can be coupled with any form of functional execution to achieve fast and accurate performance evaluation of an application kernel on a target GPU. Functional execution can be on existing simulators or natively on any available host GPU, where host and target can be of different architectures. Functional execution provides the timing model with the execution sequence of dynamically executed basic blocks (BBs) per warp and memory addresses accessed by each thread. In addition, we perform a static, one-time pre-characterization of the kernel code to obtain block-specific timing information. Individual basic blocks are evaluated on a cycle-accurate reference model of a single streaming processor (SP) of the target GPU to get their basic, statically abstracted and contention-free timing behavior.

The abstract timing model then simulates the scheduling, mapping, interleaving and contention of parallel work items across and within streaming multiprocessor (SMs) of the target GPU. This along with information collected during pre-characterization and functional execution allow us to obtain accurate kernel runtime. Furthermore, architectural changes that only affect concurrency within the GPU, such as changes in the warp scheduling algorithm or SM count can be modeled without having to pre-characterize again for the incremental architecture. This facilitates fast and convenient GPU macro-architecture exploration.

II. ABSTRACT TIMING SIMULATION

In GPUs, threads are grouped into fixed-size warps, which are further grouped into threadblocks and assigned to specific SMs. All threads within a warp execute the same stream of instructions and BBs in a SIMT lockstep fashion. GPUs execute multiple threads at various levels of parallelism. Within SMs, they rely on interleaving of warps to hide individual instruction latencies and achieve high throughput. Our timing model captures this behavior to evaluate performance accurately.

Fig. 2 shows our abstract timing simulation flow in more detail. The application source code is first compiled into its intermediate representation (IR) for pre-characterization and functional simulation. We use CUDA, *nvcc* (the NVIDIA CUDA compiler) and its PTX IR, respectively, in our setup. Block-specific timing information and block and address sequences collected during pre-characterization and functional execution, respectively, are then passed into the timing model. For every SM, the timing model simulates the issuing of instructions among all the warps assigned to the SM according to the warp scheduling policy, and it tracks the occupancy of SM resources with instructions over time. The simulation terminates once all warps are simulated till completion. The final execution time is the maximum of estimated times over all SMs. Details of each step in our timing simulation flow are described in the following sections.

A. Pre-characterization

During pre-characterization, the latency and type information of each IR instruction in each block of code is collected and passed to the timing model. Pre-characterization is performed only once per kernel and the data collected can be re-used for evaluating different architecture configurations as long as the pipeline latency is same. We use a modified GPGPUSim to statically simulate the PTX IR code and collect block- and instruction-specific latency and other information. GPGPUSim is modified to execute every basic block exactly once. This is achieved by modifying the implementation of branch instructions such that branches are never taken. All exception or assertion checks within the simulator, which could be triggered due to this modification, are bypassed. We thereby assume that exceptions do not occur during regular execution. Since functionality is of no concern during pre-characterization, and since there can be no other data-dependent execution within each block, this accurately captures true instruction sequences.

The kernel to be pre-characterized is launched with exactly 1 thread to execute on the simulator, which is configured for perfect memory and has SP execution lanes to support issuing only 1 warp per cycle. This ensures that pre-characterization accurately captures micro-architecture interactions of a single thread executing on a single SP. Every basic block is executed and a single latency value is associated with each instruction. The latency value is the number of cycles between issuing the current and subsequent instruction. These latency values capture the effect due to pipeline dependencies within a

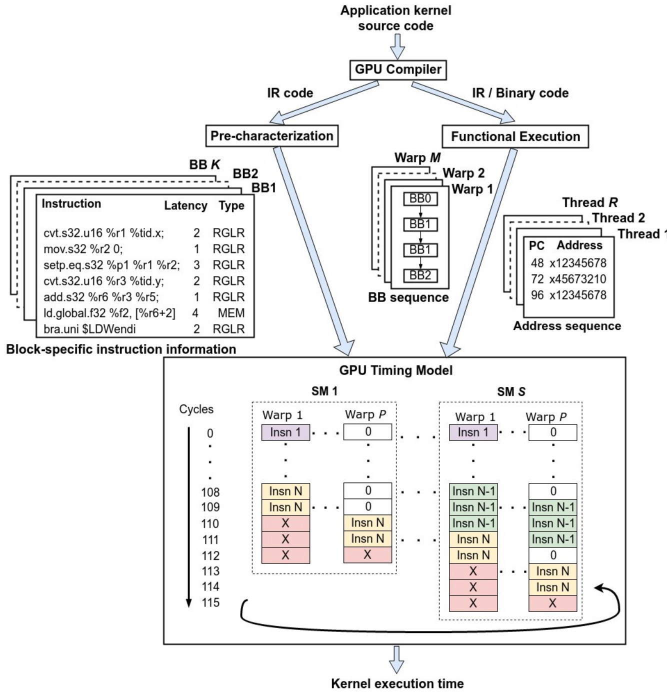

Fig. 2: Abstract timing simulation flow.

thread. We assume that there is no data-dependent timing impact except for memory instructions, which are handled by a separate memory model as explained later. Since we pre-characterize BBs in a particular order, we do not account for timing variations due to pipeline effects that span across BBs along different BB paths. However, with branch divergence of threads in a warp, the actual order in which BBs will be executed, will in many cases be the same complete fall-through sequence in which they were pre-characterized.

Finally, additional information about instruction types, shared memory and register usage is gathered. This information will be used by the timing model to compute the number of concurrent threadblocks supported within each SM and to accurately evaluate certain instructions like memory accesses and barriers, which require special consideration as explained in Section II-C.

B. Functional Execution

The kernel to be analyzed is functionally executed to capture the dynamic basic block sequence for every warp and the addresses accessed by each thread. Any functional model that can provide this information can be used. We specifically validate our scheme using the following functional models:

1) Native host GPU execution: The required information is obtained by executing an annotated application binary in a host GPU. The kernel is annotated at the IR level and then compiled down to a final binary, which is executed natively. Function calls are annotated in each basic block and corresponding to every memory instruction in the IR to collect the required data. Collected data is stored in and thus limited by GPU memory.

2) Functional simulator: The target application is executed in GPGPUSim operated in functional mode. GPGPUSim is modified to collect the dynamic BB sequence for every warp. Unlike in native GPUs, the addresses accessed by each thread within a warp are captured as base address and relative offsets in order to reduce the data transfer overhead.

C. Timing Model

Using the block-specific instruction, block sequences and address information collected during pre-characterization and functional simulation, the abstract timing model simulates the scheduling, mapping and interleaving of parallel work items across SPs and SMs to estimate total kernel execution time. The timing model maintains an *occupancy vector* data structure that tracks the occupancy for each active warp execution lane and SP within an SM. A lane occupancy vector L_s is an array $(l_{s,0}, ..., l_{s,P-1})$ of size equal to the number of active warps P in an SM s. Note that threadblocks are assigned to SMs, where P warps can be executed concurrently in each SM. The values $l_{s,p}$ in the occupancy table indicate the number of cycles after which each active warp will be ready to be scheduled and executed again. A value of 0 indicates a ready warp. The timing model then selects a ready warp based on the realized warp scheduling policy. Note that different scheduling policies can be modeled with little effort.

Operation of our timing model is summarized in Algorithm 1. For simplicity, we show warp scheduling under the assumption that only 1 threadblock can be executed concurrently, i.e. multiple threadblocks are scheduled sequentially. For each SM $0 \le s \le S-1$, the timing model processes assigned thread groups $tb \in TB_s$ in sequence (line 1). The occupancy vectors is initially set to 0 (line 4), the program counter PC_p consisting of tuple (pbc_p, pic_p) of block and instruction counters is initialized (line 5), and the algorithm then processes the sequences of BBs and instructions in all active warps (line 6). The warp scheduler selects a ready warp based on the realized scheduling policy, implemented within a $ScheduleWarp()$ function (lines 7-9). For every regular instruction issued, its pre-characterized latency value $(d_{b,k})$ is added to the entry corresponding to the warp in the occupancy vector (line 14). Special instructions are handled separately (lines 11-12, see Section II-D). Then, the warp's program counter is incremented to follow along the sequence of instructions in each block and blocks in the warp's BB sequence (lines 16-20). The warp is marked as completed once the end of its last block has been reached (line 19). Finally, time is advanced by one cycle (line 21,25). If no ready warp was available, the cycle count is advanced to the point when the wrap with the smallest remaining latency becomes ready (line 23,25). The count T_s of total SM cycles elapsed is incremented accordingly and all non-zero entries in the occupancy vector are decremented by that amount (lines 25-26). The process is repeated until all warps complete their execution (line 6). When a warp finishes, it is no longer considered for scheduling. When all the warps complete, the specific threadblock is finished and execution of new threadblock is started until all threadblocks in all SMs have been processed. The algorithm returns the total kernel execution time as the maximum cycle count across all SMs.

An example illustrating timing computation and warp scheduling is shown in Fig. 3 for a single SM with 4 active warps. At cycle 0, warp 0 is scheduled and the latency corresponding to the first instruction is updated in the occupancy vector. Time is advanced and values in the occupancy vector are decremented by 1. In the next cycle, the scheduler then selects warp 1. Its first instruction is issued and the occupancy vector is updated accordingly. The execution continues until cycle 72, when there are no more ready warps. The time is now advanced by 3 cycles, allowing for warp 0 to become ready, which is then scheduled in cycle 75. Once all instructions of a warp are issued, it is no longer considered for scheduling (cycle 108), and the process completes once all warps have finished (cycle 111).

D. Special Instruction Handling

Several instruction types, such as memory access or barrier instructions are handled specially to accurately evaluate their effect on the performance. This is done as part of $FUFree()$, $IsSpecial()$ and $HandleSpecialInstr()$ functions in Algorithm 1.

1) Memory operations: By default, we assume caches to be perfect (100% hit rate), and we currently do not support atomic and texture memory instructions. We will discuss integration of a cache model in Section II-E. Even with perfect caches, memory latencies can vary depending on the number of cache lines accessed by each load/store instruction across a warp. When a warp executes a memory instruction, requests by different threads to the same cache lines are coalesced. Thus, the actual number of memory accesses can vary from 1 to 32. We use the addresses accessed by each thread, gathered during functional execution, to compute the number of unique cache lines accessed for each memory instruction. The latency of each memory operation is then computed based on the number of accessed cache lines, cache access delays, and available load/store units in the micro-architecture. When a memory instruction is executed, the load/store unit handling the request is occupied for the entire latency of the operation. During this period, the scheduler may only issue another warp executing a memory operation if another load/store unit is available. In Algorithm 1, this is accounted for by the ready list passed to the scheduler, which only includes warps for which a functional unit is available (line 7).

2) Barrier instructions: GPUs use barrier instructions for synchronization, which can have a significant effect on performance. We currently model barriers that can synchronize threads across a thread block. Such barrier instructions are marked during the pre-characterization phase. In the timing model, every time a warp executes a barrier instruction, the warp is prevented from being scheduled by the warp scheduler until all the warps of the thread block have reached the barrier.

3) Special instructions: GPUs support floating point instructions and transcendental functions like $cos()$, $sqrt()$, etc. These instructions are executed by Special Functional Units (SFUs) present within each SM. Unlike SPs, which can process 1 regular instruction every cycle, there are fewer SFUs

Algorithm 1 Kernel timing computation.

Input: TB_s : Set of threadblocks tb assigned to SM s
W_{tb} : Warps $(w_{tb,1},\ldots,w_{tb,P})$ in threadbblock tb
BB_w: BB sequence $(bb_{w,1},\ldots,bb_{w,K_w})$ of warp w
$d_{b,i}$: Latency of instruction i in block b
$v_{b,i}$: Type of instruction i in block b
N_b : Number of instructions in block b
Local: PC_p: Program block/instruction counter (pbc_p, pic_p)
T_s : Execution time of SM s

```
1: for s = 1,...,S do
2:   T_s ⟵ 0
3:   for tb ∈ TB_s do
4:     L_s ← (0,...,0)
5:     PC_p ← (1,1), ∀p ∈ {1,...,P}
6:     while ∃p : pbc_p ≠ □ ∨ l_{s,p} ≠ 0 do
7:       rdy ← {p | l_{s,p} = 0, pbc_p ≠ □, FUFree()}
8:       if rdy ≠ ∅ then
9:         p ← ScheduleWarp(rdy, W_tb)
10:        w ← w_{tb,p}, b ← bb_{w,pbc_p}, i ← pic_p
11:        if IsSpecial(v_{b,i}) then
12:          l_{s,p} ← HandleSpecial(v_b, i)
13:        else
14:          l_{s,p} ← d_{b,i}
15:        end if
16:        pic_p ← pic_p + 1
17:        if pic_p > N_b then
18:          pic_p ← 1
19:          pbc_p ← { pbc_p + 1   pbc_p < K_w
                     { □          pbc_p = K_w
20:        end if
21:        t ← 1
22:      else
23:        t ← min{l_{s,p} | l_{s,p} ≠ 0}
24:      end if
25:      T_s ← T_s + t
26:      l_{s,p} ← max(0, l_{s,p} − t), ∀p ∈ {1,...,P}
27:    end while
28:  end for
29: end for
30: return max_s T_s
```

in each SM, so only a limited number of special instructions can be processed every cycle. The exact latency depends on the instruction type, number of SFUs and number of active threads in the warp. When a special instruction is executed, each SFU remains busy for the duration equal to its latency. During this period, no other special instruction can be issued to the same SFU. We model SFUs and their occupancy similar to load/store units and instructions. Their type information is obtained during pre-characterization, and their latency is obtained from the GPU specification.

E. Architecture Extensions

Algorithm 1 showed a GPU model with 1 warp scheduler that can execute 1 threadblock per SM. Modern GPU architectures support execution of multiple concurrent thread blocks and multiple warp schedulers per SM. Our model can capture the performance effects of such architecture variations.

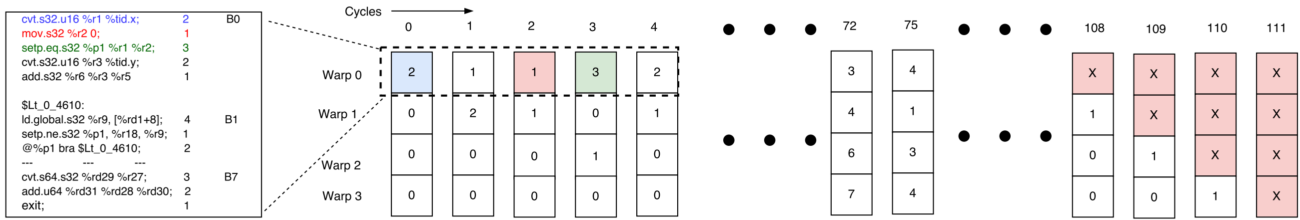

Fig. 3: Timing model and kernel timing estimation example.

1) Concurrent thread blocks: The number of thread blocks that can run concurrently in each SM depends on the available SM resources and the amount of shared memory and registers used by the program. This information can be obtained from the GPU specification and pre-characterization, respectively. In order to model multiple thread blocks running concurrently, the size of the occupancy vector is simply set to the total number of warps across all concurrently running thread blocks. The warp scheduler can now select a warp from any of the thread blocks according to the scheduling policy.

2) Multiple warp schedulers: In order to model multiple warp schedulers and the associated increase in SP resources per SM, the timing model is extended to select multiple ready warps on each scheduler call ($ScheduleWarp()$ in Algorithm 1, line 9). The occupancy vector and PC is then updated for both the warps. Any associated increase in resource contentions of shared load/store and special function units is handled in the same way as explained earlier.

3) Memory system: Our timing model allows easy integration of arbitrary memory models to account for memory access latencies. Implementing a fast, accurate GPU memory model is beyond the scope of this paper. To demonstrate capabilities, we have implemented and integrated a simple behavioral model for constant and L1 data caches. Our behavioral cache model consists of a tag store only to determine hits and misses on each request. The constant cache is read-only cache. The L1 data cache acts as a write-back cache with no write-allocate for local memory addresses, while write hits on global memory addresses result in eviction of the cache block. This behavior is in accordance with the cache behavior in GPGPUSim. Information regarding a memory operation being local or global is obtained from pre-characterization. For each memory instruction, the latency is then determined as the sum of hit and miss latencies over all accesses for the request.

III. Experiments and Results

We have implemented and tested GATSim with CUDA and *nvcc* v4.0. We use GPGPUSim v3.2.2 as functional simulator, cycle-accurate reference for comparisons and, in modified form, for pre-characterization. We model different GTX 480 configurations provided with GPGPUSim in our experiments. All experiments are carried out on an Intel Core i7-4771 machine with a NVIDIA GeForce GTX TITAN Black as host GPU. We validated our approach using the Rodinia 3.1 benchmark suite [12]. Due to compatibility issues with the CUDA version or reference simulator used, some

TABLE I: Benchmarks and kernels evaluated [12].

Kernel	Kernel name	Invoc.	Instr./Invoc.	Cycles/Invoc.
<1>	Backprop (layerforward)	1	120M	195k
<2>	Backprop (adjust_weight)	1	72M	109k
<3>	Bfs (Z6kernel)	10	1.7M	12.4k
<4>	Bfs (Z7kernel)	10	1.06M	1.7k
<5>	B+tree (findRangeK)	1	118M	155k
<6>	B+tree (findK)	1	77M	103k
<7>	Hotspot (calcuate_temp)	1	119M	189k
<8>	Pathfinder (dynproc)	5	131M	163k
<9>	LavaMD	1	986k	86.9k
<10>	Kmeans (invert_mapping)	1	10.3M	97.5k
<11>	Kmeans (kmeanspoint)	80	152M	162k
<12>	Nw (needle_1)	64	1.88M	19.9k
<13>	Nw (needle_2)	63	1.86M	19.8k
<14>	Particlefilter (find_index)	9	3.6M	88.7k
<15>	Particlefilter (likelihood)	9	7M	206k
<16>	Particlefilter (normalize_weights)	9	49k	51k
<17>	Particlefilter (sum)	9	10k	225
<18>	Streamcluster (compute_cost)	30	42.9M	76.7k
<19>	Hotspot3D (hotspotopt)	1	27M	33k
<20>	Nn (euclid)	1	1.28M	1.8k

benchmarks have been omitted. Furthermore, the inputs for some benchmarks have been changed to ensure that the native host GPU does not run out of memory while collecting functional simulation information. Benchmarks for which this could not be done have been excluded. The *huffman* benchmark could not be pre-characterized using our approach of executing only 1 thread. The list of benchmarks evaluated is summarized in Table I. For benchmarks with multiple kernels, we report results for each kernel separately, where we aggregate multiple kernel invocation in a benchmark.

Fig. 4 shows accuracy and speed of GATSim using either the native host GPU (GATSim-N) or GPGPUSim (GATSim-S) for functional execution. We report results for GTX480 configurations with different number of warp schedulers (WS) and perfect memory. We measure accuracy as the difference in total execution times from GATSim and GPGPUSim across all kernel invocations. We measure simulator throughput as the ratio of the total number of simulated PTX instructions reported by GPGPUSim over the sum of GPGPUSim or GATSim runtimes not including one-time pre-characterization. Using a simple GTX480 configuration with 1 WS and native functional execution on the host GPU, our approach provides on average a 460x speedup over GPGPUSim with 3% error. This corresponds to an average throughput of 90 million simulated instructions per second (MIPS). When evaluating a more representative GTX480 configuration with 2 warp schedulers, speed and accuracy are slightly worse at 82 MIPS with 400x speedup and 4% error on average. The maximum error for 2 WS is less than 10%. Note that *Kmeanspoint*

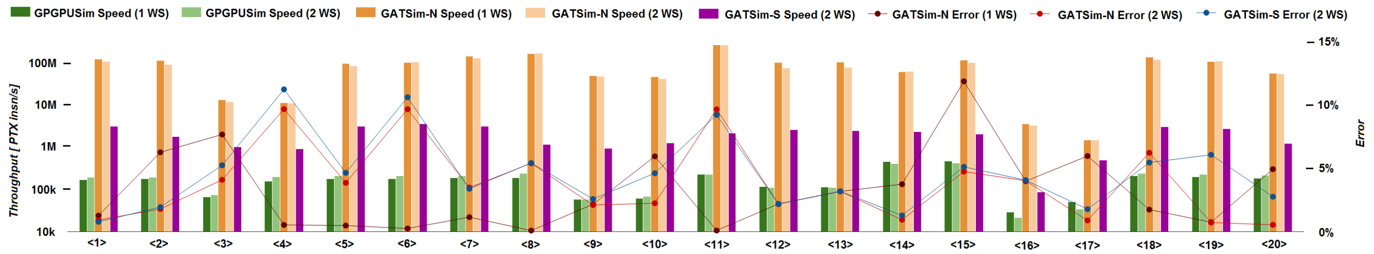

Fig. 4: Speed and accuracy of GATSim vs. GPGPUSim.

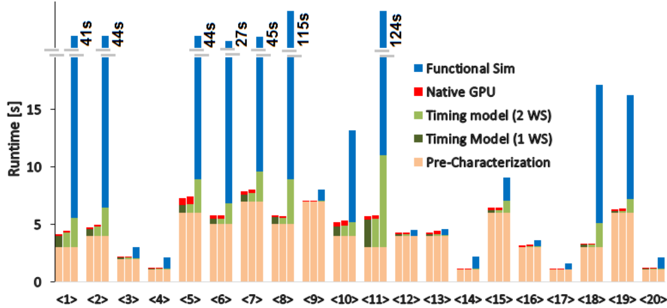

Fig. 5: GATSim runtime.

(<11>) uses texture instructions, which are currently not modeled, leading to high errors with contention effects in multiple WS. Higher errors in other benchmarks are due to the fact that those kernels have shorter execution times, where base inaccuracies in pre-characterized timing have a larger impact. Kernels with smaller instruction counts or more branches have lower throughput because of the constant simulation launch or dynamic branch simulation overhead, respectively.

Using GPGPUSim as functional model for the same 2 WS configuration provides similar accuracies at lower speed (on average 2 MIPS for a 12x speedup). Errors are slightly higher (up to 11% and 4.6% on average) due to the reduced address traces collected during functional simulation.

Fig. 5 shows a further breakdown of GATSim runtimes. Pre-characterization time takes a small but significant portion of total runtime. It is a function of static code complexity, and thus contributes relatively more in large but short-running kernels. However, pre-characterized timing values can be reused across macro-architecture variations and repeated simulations, as demonstrated for different WS configurations above. Functional simulation is otherwise the bottleneck. Note that using functional simulation requires file I/O for data exchange, which also increases the runtime of the timing model. This can be relegated by using native GPU execution, achieving an overall speed including pre-characterization of 8.2 MIPS (for a 40x speedup) on average.

Finally, Fig. 6 shows results for GATSim and GPGPUSim configurations that include memory system models with 1 WS. For simplicity, we assume that caches are flushed after every kernel completion. Some benchmarks could not be executed in GPGPUSim with this limitation. Results show that our simple memory model can capture memory effects with reasonable accuracy (<6% average error) and high speed (93 MIPS and 620x speedup on average) for the selected benchmarks.

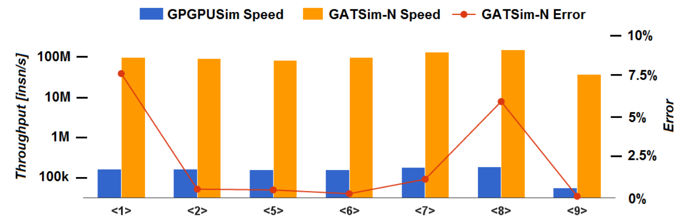

Fig. 6: Speed and accuracy with memory model.

IV. Summary and Conclusions

In this paper, we presented GATSim, a fast, accurate and modular abstract timing simulator for GPUs. Our approach is based on a flexible separation of functional and timing models, combining native host GPU execution or functional simulation with a novel timing modeling approach that uses static, one-time micro-architecture pre-characterization together with lightweight simulation of dynamic macro-architecture effects. Our approach supports rapid exploration of GPU architectures without repeated re-characterization, including easy integration of memory models. Using a native GPU and excluding pre-characterization, GATSim can on average run at more than 80 MIPS with 96% accuracy. We plan to investigate fast and accurate advanced memory models for inclusion into GATSim in future work.

Acknowledgments

The authors would like to thank Jay Patel for his initial contributions and Youssef Tobah for his help with scripting.

References

[1] A. Bakhoda et al., "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in ISPASS, 2009.

[2] J. Sim et al., "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in PPoPP, 2012.

[3] S.Hong and H.Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in ISCA, 2009.

[4] M. Boyer et al., "Improving GPU Performance Prediction with Data Transfer Modeling," in IPDPSW, 2013.

[5] S. S. Baghsorkhi et al., "An adaptive performance modeling tool for GPU architectures," in PPoPP, 2010.

[6] W. Jia et al., "Stargazer: Automated regression-based GPU design space exploration," in ISPASS, 2012.

[7] G. Wu et al., "GPGPU performance and power estimation using machine learning," in HPCA, 2015.

[8] J. Lai and A. Seznec, "Break down GPU execution time with an analytical method," in RAPIDO, 2012.

[9] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in HPCA, 2011. ‘

[10] A.Betts and A.Donaldson, "Estimating the WCET of GPU-Accelerated Applications using Hybrid Analysis," in ECRTS, 2013.

[11] C. Gerum et al., "Source Level Performance Simulation of GPU Cores," in DATE, 2015.

[12] S. Che et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in IISWC, 2009.