Scalable Deep Learning-Based Microarchitecture Simulation on GPUs

Santosh Pandey
Stevens Institute of Technology
Hoboken, USA
spande1@stevens.edu

Lingda Li
Brookhaven National Laboratory
Upton, USA
lli@bnl.gov

Thomas Flynn
Brookhaven National Laboratory
Upton, USA
tflynn@bnl.gov

Adolfy Hoisie
Brookhaven National Laboratory
Upton, USA
ahoisie@bnl.gov

Hang Liu
Stevens Institute of Technology
Hoboken, USA
hliu77@stevens.edu

***Abstract*—Cycle-accurate microarchitecture simulators are essential tools for designers to architect, estimate, optimize, and manufacture new processors that meet specific design expectations. However, conventional simulators based on discrete-event methods often require an exceedingly long time-to-solution for the simulation of applications and architectures at full complexity and scale. Given the excitement around wielding the machine learning (ML) hammer to tackle various architecture problems, there have been attempts to employ ML to perform architecture simulations, such as Ithemal and SimNet. However, the direct application of existing ML approaches to architecture simulation may be even slower due to overwhelming memory traffic and stringent sequential computation logic.**

This work proposes the first graphics processing unit (GPU)-based microarchitecture simulator that fully unleashes the potential of GPUs to accelerate state-of-the-art ML-based simulators. First, considering the application traces are loaded from central processing unit (CPU) to GPU for simulation, we introduce various designs to reduce the data movement cost between CPUs and GPUs. Second, we propose a parallel simulation paradigm that partitions the application trace into sub-traces to simulate them in parallel with rigorous error analysis and effective error correction mechanisms. Combined, this scalable GPU-based simulator outperforms by orders of magnitude the traditional CPU-based simulators and the state-of-the-art ML-based simulators, i.e., SimNet and Ithemal.

***Index Terms*—Computer microarchitecture simulation, Machine learning, High performance computing, GPU acceleration**

I. Introduction

Processor design is predicated on design space exploration capabilities, which, in turn, rely on methods and tools that can estimate architecture behavior and performance with confidence ahead of manufacturing. Consequently, cycle-accurate architecture simulators are of particular interest. Unfortunately, existing cycle-accurate simulators, such as traditional discrete event simulators (e.g., gem5 [1]) and their optimized statistical versions [2], [3], cannot *accurately* simulate applications of *real-world scale* within *an acceptable execution time* for two primary reasons.

First, traditional discrete event simulators often experience extremely long simulation time. For example, using gem5 to simulate all SPEC CPU2017 benchmarks [4] requires months to complete on a system with 128 CPU cores and 1 TB memory. Although many research efforts have been conducted to improve simulation speed, including software engineering optimizations [5], [6], [7] and parallelization [8], [9], such endeavors still fall short at reducing the simulation time to support practical use. Second, statistical simulations, [10], [3] while benefiting from shorter turnaround time, experience lower prediction accuracy as they rely heavily on instruction sampling schemes that cannot capture all details of the application workload being considered.

Given the excitement around wielding the ML hammer to tackle various architecture problems, such as branch prediction [11], [12] and memory controller [13], [14], there have been additional attempts to use ML for architecture simulations, e.g., Ithemal [15] and SimNet [16]. Instead of modeling each hardware architecture component, ML-based simulation works at instruction level prediction. It is trained to capture the impacts of a particular architecture on the currently executing instructions. Notably, instruction latency can be expressed as complex functions of the currently executing instructions and the processor hardware configurations. As ML excels at deriving the sophisticated rules that govern various complex functions, recent work shows that it can capture the architecture simulations in a similar way (see Section VII-A).

Existing ML-based architecture simulators face performance and efficiency problems, and our analysis reveals two significant challenges when trying to accelerate them on GPU machines. First, current ML-based simulators only use GPUs for inference because the other steps are not GPU friendly. However, this design would result in redundant data movement as the inputs of various instructions share similar contexts. Further exacerbated by the stringent host/device throughput, existing ML-based simulators usually fail to offer sufficient workloads to fully saturate the GPU resources. Second, the simulation process is still sequential, i.e., the latency prediction

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

of instructions must happen in order, which limits the degree of parallelism. Unfortunately, sequential simulation again restricts and diminishes the potential benefits from using single-GPU to distributed-GPU machines.

Although our proposed optimizations could be applied to various ML-based simulators (see Section VII-B), such as SimNet and Ithemal, this paper bases our designs and implementations on SimNet, because: 1) SimNet can handle lengthy program simulation and complex out-of-order superscalar architectures, while Ithemal is limited to static basic block prediction and targets simplified processor architectures without caches and branch prediction, and 2) SimNet is about 3× faster than Ithemal due to the efficient use of convolutional neural networks (CNNs) rather than sequential models.

The contributions detailed in this paper include:

- We rigorously analyze the SimNet simulator and propose impactful optimizations to avoid various redundant data movements. In particular, we design GPU-based input construction, a sliding window-based instruction queue, a custom convolution layer, and pipelined simulation to amortize the cost of data movements.
- We introduce a parallel simulation paradigm that partitions the program trace into disjoint sub-traces and simulates them in parallel to achieve scalable performance. Further, we rigorously analyze the simulation errors and develop effective solution techniques to mitigate errors introduced by the parallelization approach. Our evaluation demonstrates comparable accuracy against the gem5 simulator with parallelization.
- We implement the first fully GPU-based computer architecture simulator that can scale up to 282 GPUs and achieve an unprecedented 553.68 million instructions per second (MIPS) simulation rate. Overall, the proposed simulator leads to an average speedup of 2,796×, 425,907×, and 971,368× versus state-of-the-art simulators gem5, accurate sequential SimNet, and Ithemal, respectively.

The reminder of the paper goes as follows: Section II unveils the background and motivation for ML-based simulator. Section III discusses the challenges faced by existing ML simulators. Sections IV and V present the proposed optimizations. Section VI describes the evaluation results. Section VIII discusses the related work and Section IX concludes.

II. Background and Motivations

A. Limitations of Traditional Simulation

Traditional simulators [1], [17] are made up of a collection of software modules that mimic the behavior of individual hardware components. Each component in the simulator takes/sends "transactions" from/to other components as inputs/outputs. The simulation of a computer system is achieved by simulating every component and its interactions. In recent years, parallel accelerators, such as GPUs, have been successfully applied to many domains, from ML to scientific computing [18], [19], [20], [21], [22]. They provide enormous performance benefits. However, accelerating architecture simulation on GPUs is rarely explored because of two critical limitations. First, the simulation of different components is heterogeneous, while GPUs prefer homogeneous behavior. For example, a GPU's single instruction multiple threads (SIMT) paradigm requires adjacent threads to have similar execution paths and memory access addresses. Several parallel simulators have been proposed to simulate individual cores in parallel in a multi/many-core system [23], [24], [25], [26]. Again, the simulation of different cores is heterogeneous, and the number of cores cannot saturate the massive parallelism offered by GPUs. Second, the fact that different components frequently interact with each other mismatches with the nature of GPUs, which fall short in handling frequent communications efficiently. Due to these impediments, traditional simulators simply cannot leverage the performance advantages of GPUs.

B. Motivation: ML-based Simulation

ML-based solutions could be the key approach to overcoming the limitations of traditional simulators. In ML-based approaches [15], [16], the simulated processor is abstracted as a whole, eliminating the need to simulate individual components within the processor. Heterogeneous simulation with communication is replaced with accelerator-friendly parallelizable matrix multiplication. Notable ML-based simulation methods Ithemal and SimNet both perform instruction-wise simulation.

Ithemal uses a trainable model to predict the throughput of instructions based on opcode and operands in a basic block. It maps a basic block into a vector space by recursively feeding instructions in the block to a long short-term memory (LSTM) model. This approach has a limitation: it assumes perfect memory accesses and can only predict basic block performance. Meanwhile, SimNet predicts the latency of each instruction based on the static instruction properties and dynamic processor behavior and can be applied to real-world programs. It simulates out-of-order superscalar processors with caches with high accuracy using CNNs. Both Ithemal and SimNet focus on performance prediction or determining cycle per instruction (CPI) and can achieve high prediction accuracy. Ithemal reports an average error below 9% for a range of basic blocks, and SimNet reports average simulation errors of 1-2% for SPEC CPU2017 [4] benchmarks.

Because Ithemal is limited to basic block prediction and simple architectures, this paper employs SimNet to demonstrate how ML-based simulation can be accelerated, although the optimizations can still be applied to Ithemal (see Section VII-B). An ML-based instruction latency predictor is at the center of SimNet's simulation approach. The predictor uses the properties of the to-be-predicted instruction and a collection of concurrently executing instructions (i.e., context instructions) as its input features. Context instruction features are used to determine the architecture states. How fast an instruction can be executed depends on how busy the required processor resources are. For example, an instruction that depends on the result of a previous instruction will have a larger latency, and providing the destination registers of previous instructions can help ML models determine if such a dependency exists (refer to SimNet [16] for more details).

Benchmark	Abbr.	Benchmark	Abbr.	Benchmark	Abbr.
500.perlbench	perl	523.xalancbmk	xala	557.xz	xz
502.gcc	gcc	525.x264	x264	997.specrand_f	spef
503.bwaves	bwav	526.blender	blen	505.mcf	mcf
508.namd	namd	527.cam4	cam4	538.imagick	imag
507.cactuBSSN	bssn	544.nab	nab	554.roms	roms
519.lbm	lbm	548.exchange2	exch	531.deepsjeng	deep
521.wrf	wrf	549.fotonik3d	foto	999.specrand_i	spei

TABLE I: Benchmarks.

C. Benchmarks and Model Accuracy

We use the SPEC CPU2017 benchmark (Table I) suite [4] to evaluate the simulation performance. Out of 21 benchmarks that are used for experiments, four benchmarks (perl, gcc, bwav, and namd) are used for training, and the remaining are employed for evaluation. For most of our evaluation, we use the trained 3C+2F model from SimNet, which has three convolution layers and two fully connected layers. The ML models are trained against a classic out-of-order superscalar CPU, and Table II shows its configurations. For this model, the average simulation error of SimNet across 17 test benchmarks is 2% over gem5.

Parameter	Configuration
Core	3-wide fetch, 8-wide out-of-order issue/commit, Bi-mode branch predictor, 32-entry instruction queue, 40-entry reorder buffer, 16-entry LQ, 16-entry store queue
L1 ICache	48KB, 3-way, Least recently used (LRU), 4 miss status holding registers (MSHRs)
L1 DCache	32KB, 2-way, LRU, 16 MSHRs, 5-cycle latency
I/DMMU	2-stage translation lookaside buffer (TLB), 1KB 8-way TLB caches with 6 MSHRs
L2 Cache	1MB, 16-way, LRU, 32 MSHRs, 29-cycle latency

TABLE II: Target processor configuration.

III. Challenges for ML-based Simulation

Challenge No. 1: Frequent CPU-GPU communication.

To construct an ML-based simulator, two fundamental components are required. One is an ML-based instruction latency predictor, and the other is an input constructor that provides the required instruction features to the predictor. While instruction latency prediction (i.e., ML inference) is computation intensive and well suited for GPUs, input construction is memory intensive and done by CPUs in previous work [15], [16]. These two components require frequent communication, which generates abundant inefficient data movement between CPU and GPU. These expensive communications in SimNet are detailed as follows.

Figure 1 uses a program of six instructions to exemplify the SimNet simulation workflow. SimNet first performs functional simulation to generate a trace of features required by the ML model (step ⓿), which can be done through fast instruction simulators/emulators, such as QEMU [27]. Subsequently, the trace is encoded into numerical representations where each instruction in the program is represented by an array of 50 entries of instruction attributes. In Figure 1, $\{0,5,3,..,0,1,0,3,0,..\}$ represents the instruction attributes of the first instruction. The instruction attributes include the primary static properties of instruction (e.g., opcode and registers used) and dynamic processor states (e.g., memory dependency; branch prediction; cache miss). All features are represented as integers. A detailed list of features can be found in [16].

Steps ❶ - ❹ in Figure 1 depict how the simulator goes through each instruction in the trace to predict the latency. Across these steps, *each instruction experiences four copies* in order to construct the input for latency inference as follows: the simulator adds a single instruction from the trace to the instruction queue every iteration (❶) (first copy). For iteration 0, a trace array of <add eax, 12> is added to the instruction queue. Of note, the instruction queue holds all the instructions currently executing (to-be-predicted and context) on the processor. A variable $Clock$ (highlighted in orange color on the top left of the instruction queue) is used to track the current clock cycle of the processor. The $Clock$ is set to 0 before starting the simulation. During simulation, a variable retire clock is added to the tail of each instruction in the instruction queue. It stores the clock cycle when that specific instruction would retire (highlighted in orange color).

Subsequently, we prepare input for the ML model by concatenating all instructions in the instruction queue (2.1) (second copy). <add eax, 12> is copied to the input, which is the second copy. To ensure the input of each instruction is the same length, SimNet pads the remaining fields of input with zero. Here, we assume the input length ($Context\_length$+1) is three instructions. For the processor configuration shown in Table II, the input length is 111 instructions. The input then is copied to the GPU for inference (2.2) (third copy). Before inference, SimNet transposes the input to spatially align the input features of context instructions for the convolution operator (2.3) (fourth copy). Notably, the third and fourth copies are comparatively expensive as we need to copy and transpose $Context\_length$+1 number of instructions in each iteration. At the end, the model predicts three different latencies for the current instruction, i.e., fetch, execute, and store (❸), which respectively represent the latencies to fetch the instruction, execute it, and store any result from the instruction to the memory. Overall, there are four redundant data movements in the simulator design.

Challenge No. 2: ML-based simulation is sequential in nature.

Similar to traditional simulators, ML-based simulators are sequential in nature, i.e., they need to go through each instruction in the execution order. Sequential ML-based simulators fail to outperform their traditional counterparts as they cannot efficiently leverage hardware accelerators, such as GPUs, designed for massive parallel ML inference/training.

Again, using SimNet as an example, the latency prediction of an instruction requires the context instructions and their latencies collected from earlier simulations. Specifically, the predicted latencies from an iteration are used to perform the update and retire of instructions in the instruction queue as they could be part of the input for the next iteration. In Figure 1, the predicted latencies for the first instruction are

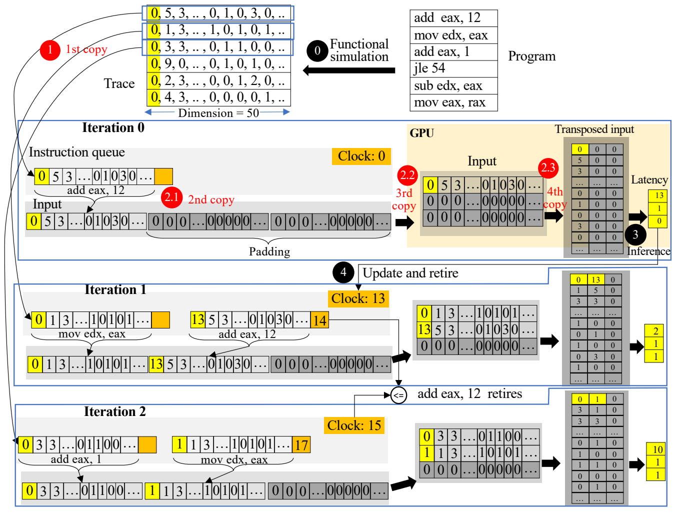


Fig. 1: Instruction streams executing in SimNet. Red circle represents the data copy steps. (better viewed in color)

13, 1, and 0. These latencies are updated for the respective instructions in the instruction queue (4). For brevity, we only show one latency per instruction. The latency of <add eax, 12> is updated to 13. The *Clock* is advanced by the sum of the current *Clock* and fetch latency, i.e., {0}+{13}=13. Retire clock for each instruction is the sum of all predicted latencies and the current *Clock*, i.e., {13+1+0}+{0}=14.

Whenever the *Clock* value is greater or equal to the retire clock, we remove the instruction from the instruction queue. Iteration 2 shows an example of how an instruction retires. The processor *Clock* in Iteration 2 is 15, and the retire clock of <add eax, 12> is 14. As *Clock* is higher than retire clock, <add eax, 12> retires from the instruction queue and will not be considered as a part of the input for future simulations of other instructions. The simulation continues likewise until all instructions are simulated. After simulating the final instruction, the processor *Clock* provides the total execution cycle of all instructions.

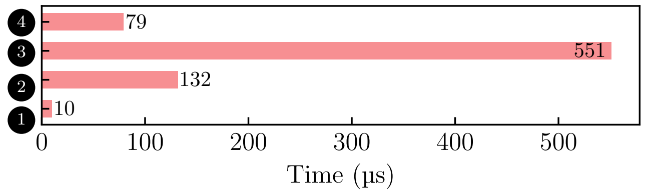


Fig. 2: Profile of a single iteration of sequential simulation.

The aforementioned stringent dependencies of an instruction with its context instructions prevent ML-based simulators from conducting parallel simulations. Figure 2 profiles the execution time for each simulation step of SimNet on a DGX A100 system [28] with AMD EPYC 7742 64-core CPUs and A100 GPU. The simulation of a single instruction takes 772 μs. The simulation throughput for this design is 0.0013 MIPS, which is 153× slower than gem5, which achieves a throughput of 0.198 MIPS when evaluated on the same system.

It is evident that the inference takes 71% of the total simulation time. The reason is that single inference has a workload of only 3.19 million floating point operations, while an A100 GPU has a maximum throughput of 19.5 trillion floating point operations per second for FP32 computation [28]. This workload gap indicates that the sequential inference is far from saturating the GPU computation capabilities. A possible way to increase the inference workload is to have multiple inferences performed in parallel, i.e., batched inference. Regrettably, the simulation's sequential nature makes parallel inference difficult. We also confirm that the simulation workflow experiences high overhead due to redundant data movements. For example, 70% of execution time is spent on redundant data movement, excluding ML inference time.

IV. Efficient External-GPU Simulation

This section details how we optimize the single instruction simulation. We avoid the redundant data movement via GPU-based input construction, sliding window-based instruction queue, and transpose-free optimizations. Subsequently, we optimize inference via a custom convolution layer and overlap the simulation and data movement in a pipelined fashion.

A. Bandwidth-efficient Input Construction

GPU-based input construction. For an instruction, the inference input used to predict its latency encompasses the features of itself and those of the context instructions, i.e., instructions that enter the processor previously and remain under execution at the moment. Clearly, context instructions are already copied from CPU to GPU when we infer their respective latencies. Copying the context instructions again to the GPU (2.2) is redundant. To avoid this redundancy, we directly offload the inference input construction onto GPUs. While moving the instruction queue on to GPUs completely removes the transfer of context instructions—that is, avoiding copying $Context\_length$ out of the $Context\_length$+1 instructions—we also need to perform updates to context instructions. Because we pack various context instructions together, updating the latency entry means updating the first column of the input in Figure 1, which will experience expensive strided global memory access. Correspondingly, we allocate a dedicated latency vector in shared memory to resolve this problem.

Sliding window-based instruction queue. It targets the issue of copying instructions from the instruction queue to the input (2.1 in Figure 1). The reason for maintaining an instruction queue and input separately is to link the context instructions together along with padding for inference.

We introduce a novel sliding window-based instruction queue that can be directly used as input, avoiding this redundant data copy. We allocate a contiguous memory block for $Context\_length$+1 and additional N instructions (i.e., ($Context\_length$+1) + N) for the instruction queue, where N represents the batch size. The queue can accommodate N future instructions and a window of $Context\_length$+1 instructions slides in this queue to ensure that the current instruction is always in the first position. In every iteration, we slide the window by one instruction. The sliding window also automatically adjusts the padding at the tail. Further, instead of having a data copy for every instruction, this design also amortizes the cost of data copy (❶) by copying a batch of N+1 instructions together to the instruction queue. During the batched copy, to ensure the current instruction is always in the first position in the queue, the instructions within the batch are copied in a reversed order, i.e., the last instruction is in the first index of the queue.

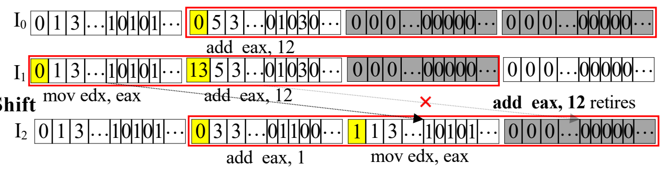
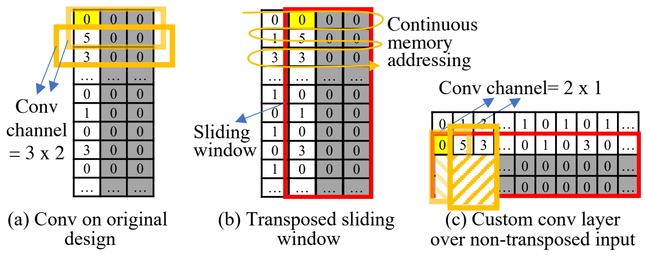


Fig. 3: Sliding window-based instruction queue.

Figure 3 shows an example of how a sliding window-based instruction queue works. Assuming $N = 1$ and the length of the input is 3, each red box marks the inference input for the current iteration. First, the instructions {0,5,3,..,0,1,0,3,0,..} and {0,1,3,..,1,0,1,0,1,..} are copied in batch to the instruction queue in a reversed order, i.e., second instruction {0,1,3,..,1,0,1,0,1,..} is stored in the first index of the instruction queue's memory. In the first iteration, the start pointer is the rightmost three instructions, with the first one as the {0,5,3,..,0,1,0,3,0,..} instruction and the remaining two as zero padding. For the next iteration, we slide the red box left by one instruction. Finally, we have reached the memory limit, and all non-retired instructions are shifted to the initial position. Of note, as {0,3,3,..,0,1,1,0,0,..} retires at the end of the second iteration, we do not need to copy that instruction. In this example, the cost of copying is amortized over three iterations.

Avoiding transpose. Here, we address the final redundant copy caused by the transpose of the input in Step (2.3). The input is transposed for the convolution operator to align the same features of different instructions. In fact, for each instruction inference, a transpose of $Context\_length$+1 instructions is performed, which is the most expensive redundant copy.

To avoid transposing too many instructions at each inference, a naive idea would be to prepare the inputs directly in the transposed format. However, this design cannot work with our sliding window-based instruction queue, and Figure 4 illustrates the reason. In Figure 4(a), the convolution operation multiplies a channel of dimension W×H, i.e., 3×2, with the inputs by sliding a kernel from top to bottom. As this input does not have a sliding window, the memory addresses of consecutive rows will be continuous. When the extra spaces are added in the sliding window as shown in Figure 4(b), the red box marks the current input. Clearly, in such a design, the consecutive rows of the red box become discontinuous due to the future inputs. Markedly, the leftmost instruction {0, 1, 3, ..., 1, 0, 1, 0, 1, ...} has separated the continuous rows of the three instructions in the red box. This fact implies that changing the data structures alone will not help avoid this redundant data movement.



Conv channel = 3 x 2
Sliding window
Continuous memory addressing
Conv channel= 2 x 1
(a) Conv on original design
(b) Transposed sliding window
(c) Custom conv layer over non-transposed input

Fig. 4: Convolution (Conv) on (a) original transposed input, (b) transposed sliding window input, and (c) custom layer on non-transposed input with sliding window.

Therefore, we propose to customize a convolution layer that can perform convolution computation on non-transposed inputs. Pointedly, we do not want to customize the entire inference logic because NVIDIA TensorRT [29] offers state-of-the-art inference performance. By adding a single custom convolution layer for merely the first convolution, the follow-up layers can still enjoy the benefits of TensorRT.

As our custom layer offers more benefits than avoiding transpose in the subsequent designs, the details about custom layer implementation and the integration with TensorRT are featured in Section IV-B.

B. Inference Optimization and Pipeline

Once the redundant data movement is eliminated, inference becomes the next bottleneck. Using the popular PyTorch libtorch inference library as an example, inference takes 71% of the simulation time in SimNet. We first discuss how we improve the inference with model and system optimizations. Then, we present our custom layer optimization followed by pipelining the inference with data movements.

Inference optimization. For system optimization, we use NVIDIA's TensorRT [29] high-optimization inference library. It performs various system optimizations, such as kernel fusion, auto-kernel tuning, and dynamic memory allocation, as well as providing for efficient use of Tensor Core. TensorRT offers 2-3× speedup for inference over LibTorch. We adopt NVIDIA half precision and 2:4 model pruning [30] support to improve the inference speed further. For lower precision, we use half precision as it is supported by recent GPU pipelines, including Tensor Core. It reduces the number of bytes accessed during inference and enables fast half-precision matrix multiplication. We also perform 2:4 sparsity-based pruning to more thoroughly benefit from Tensor Core hardware. This technique prunes two least values from four consecutive values. Tensor Core can provide nearly 2× speedup for matrix multiplication with 2:4 sparsity [30].

Custom convolution layer. We make an important observation that most computations on the first layer are performed in the paddings, which can be avoided. In Figure 4(a), the convolution channel of dimension 3×2 is used only when the first column would result in non-zero values. *Therefore, our custom layer is designed to replace the first layer of the original model to perform convolution on non-transposed and strided* *memory address matrix, as well as avoiding the padding computations.* Figure 4(c) shows how the custom convolution layer avoids transposing, computes the padding, and supports the sliding window optimization. Instead of transposing the input, we transpose the convolution channel itself, i.e., W×H → H×W (3×2 → 2×3). Because the convolution channel is small and transposing it is a one-time cost, the overhead is negligible. Then, to avoid computing the paddings, the custom convolution layer dynamically adjusts the dimensions of the convolution channel to perform computation only over the non-paddings. With TensorRT's convolution layer implementation, the width of the convolution channel is always static, i.e., 2×3 (after transpose), and cannot be dynamically adjusted. The custom layer provides the flexibility to pass an extra parameter to the convolution layer, which defines the count of context instruction or non-padded columns (here, 1). In this example, the convolution channel is dynamically adjusted to 2×1 from 2×3. Of note, the 3C+2F model in SimNet has 64 convolution channels in the first layer.

Pipelined simulation. Now, the only data movement is copying a trace of N instructions from CPU to GPU. Afterward, the CPU remains idle for N iterations of simulation. To further reduce the simulation time, we overlap the batch data copy of N instructions with the simulation of N instructions on the GPU. We use two instruction queues, or *double buffering*, to enable the pipeline. To properly hide the data copy, the data movement time should be similar to or less than the simulation time on the GPU. To balance the execution time, we propose to tune the value of N, so the time to copy a batch of N instruction is similar to the simulation time of N instructions. The good news is that copying a batch of N instruction takes more time than simulating it, while the data copy time increases sublinearly with respect to the increase of the instructions due to the throughput-oriented bandwidth of CPU-GPU. Hence, we can find the sweet spot: the value of N where the time to copy N instructions is equal or similar to the simulation time of N instructions. A proper value of N is studied in Section VI-B.

V. Parallel Simulation with Accuracy Recovery

This section details efforts to achieve efficient distributed ML-based simulation via a carefully designed parallel scheme that balances simulation throughput and accuracy. In particular, we discuss the parallel simulation of a single program (Section V-A), analysis of parallel simulation errors due to dependency interruption, and recovery from the parallel simulation errors (Section V-B). The proposed parallel simulation scheme efficiently exploits GPU computation resources, enabling significant simulation throughput improvements.

A. Parallel and Multi-GPU Simulation

Simulation can be simply performed in parallel to simulate multiple benchmarks. However, parallel simulation of a single benchmark is not straightforward. To extract parallelism, SimNet proposes to partition the entire instruction trace into multiple, disjoint sub-traces then simulate all sub-traces in

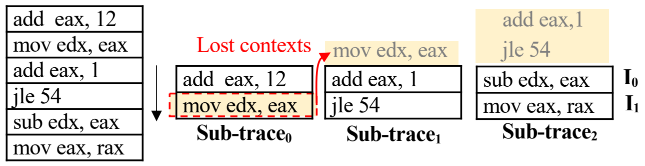


Fig. 5: Parallel simulation with two sub-traces.

parallel and independently. Notably, the instructions within each sub-trace are simulated in a sequential order to preserve the instruction dependency within the specific sub-trace. As a result, the context instructions for the first few instructions in each sub-trace will be lost and can result in inaccurate predictions. The impact on accuracy is studied in Section V-B.

After partitioning the trace, we can perform inference for #sub-traces number of instructions simultaneously. Figure 5 shows an example of how the six-instruction trace is partitioned into three sub-traces with two instructions in each partition. After partitioning, we can infer the i-th instruction of all sub-traces together. The first instruction of $sub\text{-}trace_1$ and $sub\text{-}trace_2$ will not have any required context instructions as it is the first instruction to be simulated from those sub-traces. If the instructions are simulated sequentially, the first instructions from $sub\text{-}trace_1$ and $sub\text{-}trace_2$ would have {<mov edx, eax>} and {<add eax, 1>, <jle 54>} as their context instructions. Now, one and two context instructions are lost for $sub\text{-}trace_1$ and $sub\text{-}trace_2$, respectively.

It is important to mention that this design enables embarrassingly parallel simulations in distributed settings. During simulation, no inter-GPU communication is required. Communication is done once after completing the simulation to gather the $Clock$ from each sub-trace. Based on how we handle the simulation error resulting from partitioning the trace, we will experience only very light communication or workload imbalance challenges. The impacts of different numbers of sub-traces and GPUs are examined in Section VI-B.

B. Parallel Simulation Error and Recovery

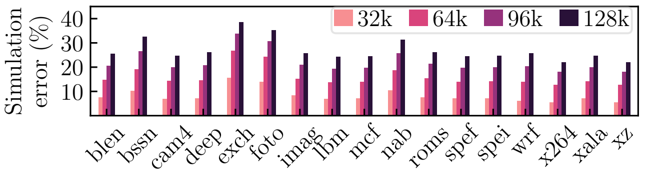


Fig. 6: Parallel simulation errors for 10M instructions of all benchmarks with 32k, 64k, 96k, and 128k sub-traces incurred by SimNet.

Error study. Figure 6 shows the parallel simulation error incurred by SimNet for all 17 benchmarks used in our evaluation. The simulation error is calculated as $(\frac{CPI_{sequential} - CPI_{parallel}}{CPI_{sequential}} \times 100\%)$ where CPI corresponds to $\frac{Clock}{Total\ instructions}$. The general trend across all benchmarks is that simulation error increases with the number of sub-traces. Specifically, the simulation error can go as high as 40% with 128k sub-traces for benchmark exch. Even the lowest error is 22% for benchmark x264. This result projects that the rapidly increased simulation errors will limit the scalability of our parallel simulation.

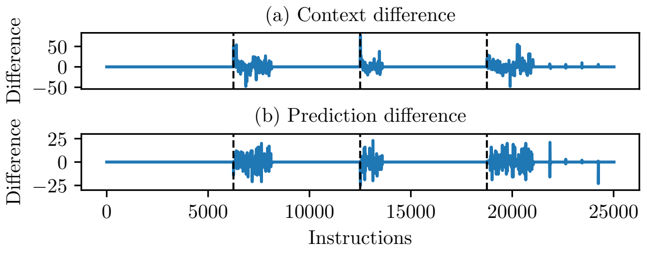


Fig. 7: Instruction-wise context difference (a) and prediction difference (b) for 25k instructions with four sub-traces.

The root cause of the simulation error is either the loss of context instructions or inaccurate latency of the required context instructions or both. Because the first few instructions of the sub-trace experience these issues more severely, the majority of the simulation errors appear then. Figure 7 examines the root cause via partitioning the 25k instructions in the xz benchmark into four sub-traces. The black dotted vertical lines represent the partition boundary, and the blue lines plot the differences. Notably, whenever there is a difference in the context instructions in Figure 7(a), the predicted cycles in Figure 7(b) differ for some number of consecutive instructions in each sub-trace. Moreover, even after we simulate the $Context\_length$ number of instructions that should gather sufficient context instructions, the error still exists. This is caused by the fact that the latency of the context instruction is not as expected. Overall, once the simulation proceeds through an instruction with minimal context, the prediction error starts trending down.

Warmup. To reduce the parallel simulation error, our warmup strategy aims to compensate for the missing context instructions for the first few instructions in each partition. As such, our warmup step simulates W instructions before each partition to offer the required context instructions. As the maximum number of context instructions for a to-be-predicted instruction is $Context\_length$, we keep $W = Context\_length$. Our warmup approach is inspired by the warmup simulation in traditional cycle-accurate simulations [31], [3], [5]. Particularly in traditional simulators, warmup initializes the hardware component states. For components such as cache memory and branch predictor, a lengthy warmup of millions to billions of instructions is required to get their accurate states.

The goal and design of our warmup instructions differ from those of traditional simulators. *Goal*: we need prior instructions to fill up the context instruction space for the inference input. Although we still call those context instructions as warmup instructions, they are not used to warm up the hardware components from the simulators. *Design*: as the number of context instructions is determined, before simulation, we can add a set number of instructions at the beginning of each partition to fill the context instruction space. This design avoids inter-partition communications. In contrast, traditional simulators would require many warmup instructions, which, if used, will involve inter-partition communications.

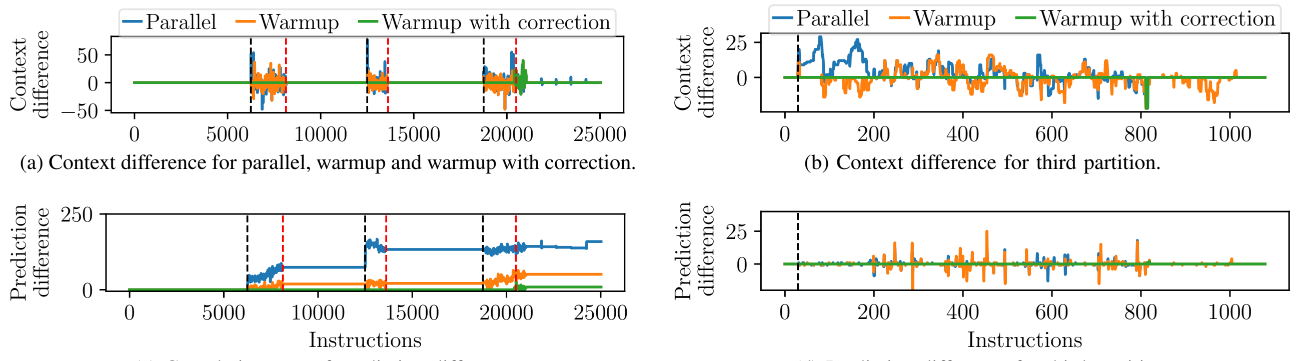
Despite the warmup design's simplicity, this approach turns out to be quite effective at reducing the overall prediction error. Figures 8(a) and (c) show the difference in the number of context instructions and predictions. We use the cumulative sum of the prediction differences in Figure 8(c), which corresponds to the overall simulation error. As the figure shows, warmup reduces the differences in context, as well as the prediction difference. With the warmup of $Context\_length$ instructions, the simulation error is reduced from 10% to 3%. Figure 8(b) also shows the context difference for the third partition with and without warmup. As the figure illustrates, although the context difference is zero for a few earlier instructions, it differs for the instructions that follow. Hence, warmup may still fail to recover the errors even if the context is correct.

Post-error correction. Warmup alone fails to fully recover the error for the first few instructions as it may not get the correct latency for context instructions. Performing a longer warmup is not a viable option because the required warmup length varies with partition to get the correct context, and there is no way to discern it before starting the simulation. Along with incorrect latency, we would construct the wrong inputs for the inference. Additionally, as the latency of instruction affects when a context instruction retires, such a difference can also impact the number of context instructions.

We propose to re-simulate the first few instructions after the preceding sub-trace completes its simulation as they can offer more accurate heuristic context instructions for the earlier instructions of the sub-trace. This post-error correction approach differs from the warmup design as the former re-simulates the first few instructions after completing the entire simulation. The number of required corrections varies for each partition. As the prediction difference vanishes after a minimum context, the correction is performed until the number of context instructions is zero or within a maximum limit during the simulation. This is because there is a high probability of fully recovering the context for following instructions when the number of context instructions is minimal. For instance, with only one context instruction, it is easier to match the latency of that context instruction, and there will not be follow-up side effects. Of note, the correction may not always completely recover the error if there is no instruction with the minimum context in the preceding partition. Yet, post-error correction can still reduce the error when compared to warmup as it may provide more accurate contexts. Warmup is still used with correction to reduce the error for instructions beyond the correction limit.

Figure 9 shows how post-error correction works when 20 instructions are partitioned into four sub-traces and simulated on two GPUs. Initially, all instructions in each partition are sequentially simulated. Then, if this partition is not the last one on the GPU, it will re-simulate the following instructions, which were previously simulated by the next partition. As shown in Figure 9, partitions 0 and 2 re-simulate $\{I_4, I_5\}$ and $\{I_{15}, I_{16}, I_{17}\}$ from partitions 1 and 3, respectively. The re-simulated cycles will replace the previously simulated ones.

Our post-error correction includes two key design choices to maintain efficiency, accuracy, and scalability. First, for the re-simulation termination criterion, our re-simulation continues until either the number of context instructions of the re-



(a) Context difference for parallel, warmup and warmup with correction.

(b) Context difference for third partition.

(c) Cumulative sum of prediction difference.

(d) Prediction difference for third partition.

Fig. 8: Warmup versus warmup with a correction to recover the parallel simulation error. Of note, (a) and (c), respectively, show the context and prediction differences for 25k instructions across four partitions. (b) and (d) zoom into the third partition for detailed analysis. Respective simulation errors are 10%, 3%, and 0.1% for baseline, warmup, and warmup with correction.

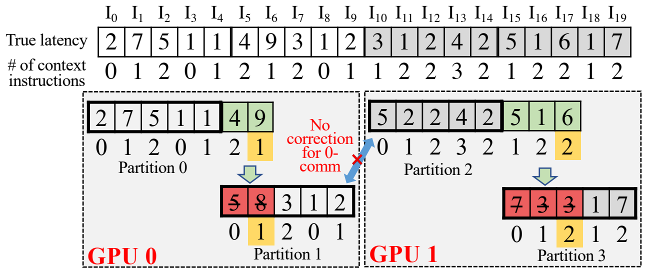


Fig. 9: Post-error correction for multi-GPU simulation.

simulation matches the prior outcomes or a user-defined re-simulation limit. The termination instructions are instructions I_6 and I_{17}, whose number of contexts in the first and re-simulation match. Second, we do not perform post-error correction for the first partition of each GPU because the re-simulation termination criterion requires sending the correct number of context instructions of the first partition to the preceding GPUs. Because each GPU can simulate 32k partitions in a batch, not correcting the first partition among 32k partitions results in a negligible non-corrected rate of $\frac{1}{32{,}768}$. Considering this design helps maintains zero inter-GPU communications during the simulation, we opt for this choice.

Figure 8(c) shows the reduction in the prediction difference when correction is applied after warmup. The correction required for each partition is different because it is performed until we encounter the minimum context. The red dotted lines represent the instructions until a correction is made. For the second and third partitions, the prediction difference is completely eliminated. For the fourth partition, we do not encounter minimum context until the limit. Hence, the following instructions still have some errors, but less than the warmup-only solution. Figures 8(b) and (d) show that the context instruction and prediction differences are completely eliminated for the third partition. Warmup with correction reduced the error from 3% using only warmup to 0.1%.

VI. Evaluation

We evaluate our work on a NVIDIA DGX A100 system with eight A100 GPUs (40 GB) and AMD EPYC 7742 64-core CPU. Both gem5 and SimNet are evaluated in the same system. This simulator is implemented in C++/CUDA and compiled with gcc 9.3.0 and the CUDA Toolkit 11.4. For inference, we use TensorRT 8.0. For the scalability test, we use the Summit supercomputer at Oak Ridge National Laboratory [32]. Each Summit node has six NVIDIA V100 GPUs (16 GB), dual-socket 22-core POWER9 CPUs, and 512 GB of memory. To compare with state-of-the-art parallel simulators, we also evaluate ZSim. ZSim can only simulate selective x86 microarchitectures, while gem5, SimNet, and this simulator simulate ARMv8. Because ZSim requires some tools that support up to gcc 4.8.4 and Ubuntu 14.04, we cannot directly run it on the DGX A100 system. Instead, we set up the ZSim environment on a dual-socket 16-core Intel Xeon E5-2637 server with 128 GB memory. For fair comparison, we also run gem5 on this server, and project ZSim's performance on the DGX A100 server based on the performance of gem5 on both systems.

We use MIPS to measure the simulation throughput. For all experiments except the scalability test, we use the first 100 million instructions from each benchmark. To saturate all GPUs, the scalability test runs up to 100 billion instructions from each benchmark. The reported simulation error is benchmarked against the gem5 simulator.

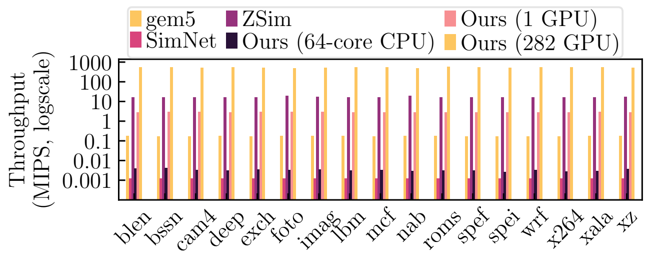


Fig. 10: Our approach versus the state-of-the-art simulators.

A. Comparison with State-of-the-Art

Figure 10 compares our work with state-of-the-art, including gem5, ZSim, Ithemal, and sequential SimNet. Notably, the original SimNet manuscript [16] presents the performance of parallel SimNet on 1 GPU or 8 GPUs (Figure 9 in [16]). However, as shown in Figure 6, SimNet on 1 GPU experiences up to 16% accuracy loss (1 GPU requires 32K trace partitions)—let alone the 8 GPU version. Therefore, neither 1 nor 8 GPU SimNet is regarded as accurate. Consequently, we report the performance of sequential SimNet, which is accurate.

Our work achieves 2.86 and 2.45 MIPS throughput on one A100 and V100 GPU, respectively. Of note, parallel CPU implementation on DGX A100 with a 64-core AMD EPYC 7742 CPU achieves a throughput of 0.0033 MIPS. When scaled to 282 V100 GPUs of 47 Summit nodes, we achieve an average throughput of 553.68 MIPS. We use 10 billion instructions for this scalability test. A detailed scalability study is included in Section VI-C. gem5, the traditional state-of-the-art simulator, achieves a throughput of 0.198 MIPS. ML-based architectural simulators, Ithemal and SimNet, achieve an average throughput of 0.00057 and 0.0013 MIPS, respectively. Although both can resort to hardware acceleration, they are significantly slower than gem5 as they cannot scale to many GPUs while maintaining the desired accuracy. We also point out that ZSim achieves an average throughput as high as 16.45 MIPS. It is important to note that the ZSim's parallelism is limited to the number of simulated cores. Hence, unlike this simulator, ZSim cannot further scale its performance.

For *functional simulation overhead* and *hardware validation*, the instruction set emulators, e.g., QEMU-KVM [33], [27], are used to generate the functional trace, which achieves a throughput of 1290 MIPS. This is much faster than the throughput of the ML-based simulator. Therefore, we negate the overhead to generate the functional traces. The gem5 simulation of the A64FX architecture configuration is verified to have an average absolute error of 6.6% [34]. When evaluated against the gem5 A64FX model, the ML-based version has an average absolute error of 6.0% [16]. This result shows that our work can provide similar accuracy as gem5.

B. Data Movement Optimizations Study

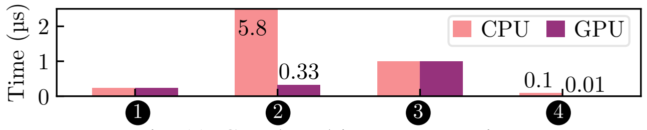


Fig. 11: GPU-based input construction.

GPU-based input construction. Figure 11 compares the performance of CPU- and GPU-based input construction. Input construction and data transfer take 4 μs and 1.8 μs, respectively, and 5.8 μs in total. Data transfer is reduced from 4 to 0.04 μs as we only need to transfer one instruction instead of the whole input from CPU to GPU. Thanks to the massive parallelism of GPUs, the input construction time (❷) further reduces from 1.84 μs to 0.33 μs per instruction. Also, update and retire (❹) time reduces from 0.1 μs to 0.01 μs per instruction. Overall, GPU-based input construction provides 4.5× speedup in simulation.

Sliding window-based instruction queue. The sliding window-based instruction queue reduces the input construction time by avoiding the concatenated copy from the instruction queue to the input. Figure 12 shows that the input construction time decreases along with the increase in N. For our

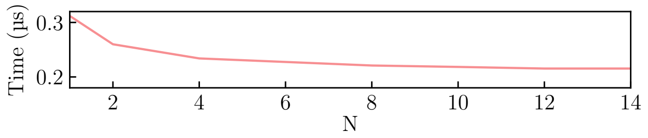


Fig. 12: Average input construction time with increasing N.

experiments, we use N=10 because increasing N will increase the memory consumption, resulting in negligible performance improvement. This optimization reduces the input construction time from 0.33 μs to 0.21 μs per instruction—a 36% reduction in input construction time.

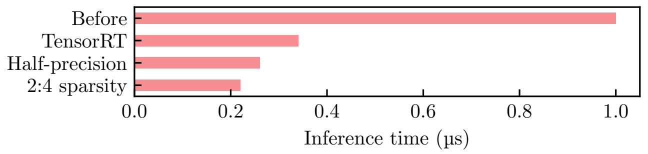


Fig. 13: Inference improvement.

Inference optimization. Figure 13 plots the inference time improvements with respect to each optimization. Using TensorRT, the inference time reduces from 1 μs to 0.34 μs per instruction. Furthermore, with half-precision and 2:4 sparsity pruning, the inference time is reduced to 0.26 μs and 0.22 μs per instruction. Combined, the inference time is improved by 4.6× with negligible accuracy loss.

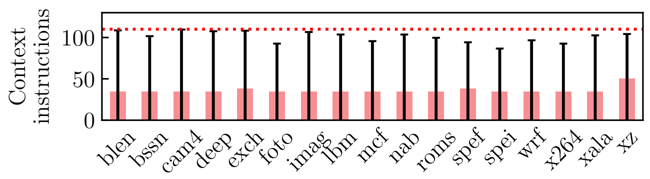


Fig. 14: Average, maximum, and minimum # of context instructions.

Custom convolution layer. Figure 14 shows the average and maximum numbers of context instructions in each benchmark. On average, more than 68% of the instructions are paddings. By avoiding transposing through a custom convolution layer, the input construction time further reduces from 0.23 μs and 0.1 μs per instruction. During simulation, avoiding computation for the paddings reduces the inference time from 0.22 μs to 0.18 μs per instruction.

Pipelined simulation. Figure 15 presents the time consumption of copying a batch of instructions versus the simulation of these instructions with respect to increased batch size. In particular, for a single instruction, copy takes 0.45 μs, while simulation takes 0.3 μs. Because copy uses NVLink, which is a throughput-oriented interface, the time consumption increases slower than computing the same number of instructions. We find the sweet spot (i.e., when copying and simulation take similar time) is $N = 3$. Note that the sliding window size and batch size are related parameters. Because the optimal sliding window size is 10 instructions, as shown in Figure 12, we keep $N = 10$ as the optimal batch size.

Figure 16 gleans the overall effectiveness of data movement optimizations. Collectively, GPU-based input construction (GIC), sliding window-based instruction queue (SWIQ),

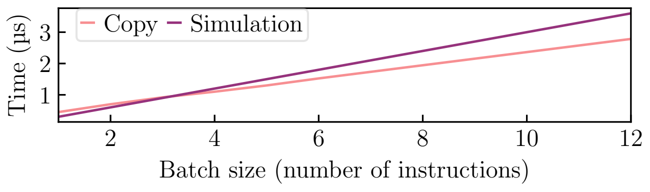


Fig. 15: Execution time for instructions.

custom convolution layer (CC), optimized inference (OI), and pipelined simulation (PS). With all optimizations, the simulation throughput increases from 0.133 MIPS to 2.86 MIPS, which is 21.5× speedup on average.

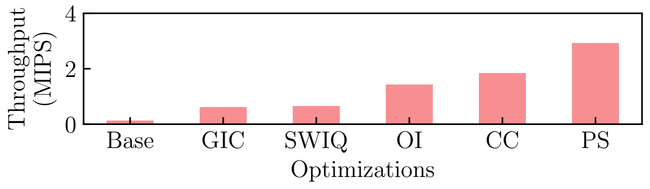


Fig. 16: Simulation with proposed optimizations.

C. Scalability Study

Figure 17 shows the strong and weak scaling of our approach on the Summit supercomputer. In strong scaling (Figure 17(a)), simulation speed climbs by 5.43×, 10.28×, 19.96×, 40.59×, 79.45×, 160.09×, and 225.89× when scaling from 1 to 6, 12, 24, 48, 96, 192, and 282 GPUs. This shows that the proposed simulator can achieve near linear scalability with an increasing number of GPUs. We cannot achieve exact linear scalability because different partitions could perform post-error correction for a dissimilar number of instructions, leading to slightly different workloads across GPUs.

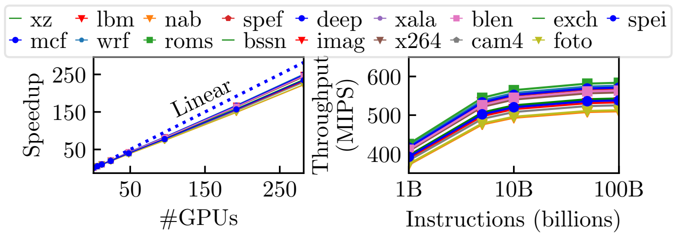


Fig. 17: Scalability study for (a) strong scaling of 10 billion instructions per benchmark and (b) weak scaling of 282 V100 GPUs.

In weak scaling (Figure 17(b)), we use 282 GPUs. When the number of instructions grows from 1 to 100 billion, simulation throughput increases accordingly. The reason is that when more instructions are simulated, the ratio of re-simulated instructions in post-error correction drops, which decreases the ratio of redundant workloads. Note, we cannot use 100 million instructions for the scalability test because it leads to very low workloads per partition, i.e., ∼10, when scaling to 282 GPUs.

Parallel simulation error. Figure 18 shows the reduction in parallel simulation error for 100 million instructions partitioned into 8 GPUs with 32k sub-traces in each GPU. The length of the warmup and the maximum threshold for error correction are kept $Context\_length$ and 100 instructions, respectively. To make sure the simulator has comparable accuracy as a cycle-accurate simulator, the simulation error is evaluated against gem5. As Figure 18 illustrates, both warmup and warmup with post-error correction effectively reduce the simulation error for all benchmarks. With warmup, the average error is reduced from 16% to 3.4%. When additional correction is performed, the simulation error is further reduced to 2.3%. As the figure demonstrates, the ML-based simulator can attain high accuracy comparable to gem5.

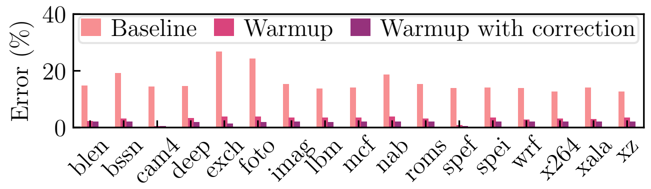


Fig. 18: Improvement in parallel simulation error. Simulation error is benchmarked against gem5.

D. Prediction Error for Different Instructions

Table III studies the prediction error of different types of instructions. Particularly, memory instructions include memory read and write instructions, and the arithmetic and logic unit (ALU) operations include all arithmetic and logical operations, such as add, subtract, multiply, shift, and divide instructions. The data is collected across all the benchmarks. As Table III shows, memory instructions have a slightly higher prediction error compared with ALU instructions, i.e., 1.175% versus 2.96%. Of note, memory instructions are potentially subject to more prediction errors because they are influenced by more complex hardware components, such as caches, queues, and other memory components.

Operation type	**Prediction error (%)**
ALU instructions	1.175
Memory instructions	2.96

TABLE III: Prediction error for different types of instructions.

E. Predicting Other Simulation Metrics

The proposed simulator can tackle various architectural metrics, e.g., CPI, cache miss rates, branch misprediction rates, and memory bandwidth. Here, we discuss how the simulator collects the memory bandwidth and CPI metrics.

For the memory bandwidth, we derive it based on the predicted latency and the total amount of data that is accessed from memory. The latter item is derived in two steps. First, we record the load and store instructions from the instruction queue. Second, for each load/store instruction, the input trace further offers the information about where a particular data is accessed (i.e., '0' for L0 cache, '1' for L1 cache and '2' for memory). Only the accesses from memory are accounted. Then the the memory bandwidth is the ratio of the amount of data accessed from memory and the latency. For computing the CPI, the simulator tracks the instruction retire events in the instruction queue at a certain interval. Note that computing CPI for an interval is important to capture the variability and phase behavior during the execution. While other instructions retire in order, store instructions do not. We tracked store instructions separately to get the correct count. The CPI is then computed as the ratio of the collective predicted instruction latency over the number of retired instructions for an interval.

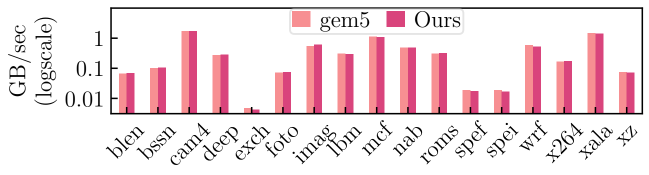


Fig. 19: Memory bandwidth for different benchmarks.

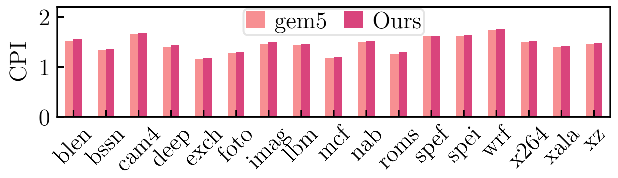


Fig. 20: CPI for different benchmarks.

Figures 19 and 20 show the memory bandwidth and CPI for all benchmarks collected from the ML-based simulator for the default architecture. Quantitatively, for most benchmarks, the predicted CPI and memory bandwidth are close to the simulated results of gem5 and show similar trends.

F. Use for Design Space Exploration

Table IV lists various microarchitecture parameters that do not require any retraining. Particularly, any variation in the listed parameters will directly impact the branch predictions (correct/incorrect) and/or memory access locations (L1 cache, L2 cache or memory). Thus, such changes will lead us to generate a new input trace. Note that generating a trace with functional simulation is significantly faster than retraining or parallel simulation i.e., 1290 MIPS. Subsequently, we can use the same trained model for simulations.

Hardware Components	Designs
Branch Predictor	algorithm, branch history table size, branch target buffer size
Memory Management Unit	Size, associativity
L1 ICache	Size, replacement policy, associativity
L1 DCache	Size, replacement policy, associativity
L2 Cache	Size, replacement policy, associativity

TABLE IV: List of parameters that do not require retraining.

Figure 21 shows one scenario for how simulation parameters can be changed without retraining. Specifically, we seek to determine the best L2 cache size for an architecture configuration shown in Table II using wrf as a test benchmark and CPI as a metric. Compared to gem5, the simulator shows similar CPI trends. In this point-wise analysis, when the L2 cache size is in a range from 256KB to 4MB, a significant performance improvement is observed for a cache size up to 1 MB, which is the optimal size in this case. Just like for the case of using traditional simulators, this conclusion is obtained in the context of all the relevant architectural features that contribute to this quantitative conclusion.

For changes to other microarchitecture parameters, retraining is required. Training data can be generated using a cycle-accurate simulator for the configurations of interest. If the number of instructions to be simulated is large enough, as with

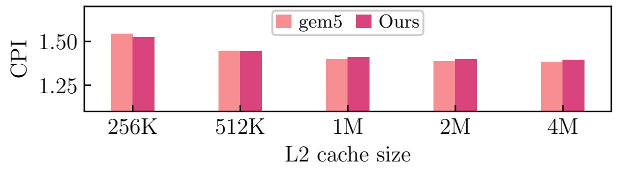


Fig. 21: L2 cache size design exploration.

design space exploration using realistic application workloads, the retraining overhead would be negligible. More details on training overhead can be found in [16].

VII. Discussion

A. Why ML-based Microarchitecture Simulation Works

Computer architecture performance prediction is a nonlinear, multi-parameter, highly complex optimization problem. An added difficulty is that the system performance is dynamic in nature, meaning that important aspects of the application workload and utilization of system resources cannot be described *a priori* in a static fashion. As an alternative methodology, analytical models use a collection of equations that serve as parameterized functions to model hardware performance [35] (also refer to equations 10-24 from [35]). Apparently, the discriminative nature of each aforementioned equations can be well captured by properly designed neural networks, which excel at approximating linear or nonlinear relationships [36].

While the analytical model cannot capture the dynamic behavior of programs, the ML-based approach can learn to capture the nonlinear and dynamic relationships between instructions based on the instruction property, context instructions, and actual latency of that instruction. In addition to this intuition, ML-based modeling is much easier than analytical modeling as designing an analytical model is "crafty", time-consuming, requires tremendous domain knowledge and experiments to gather the parameters, and is applicable to a narrow spectrum of application-architecture pairings.

B. Generalizing the Optimizations to Ithemal

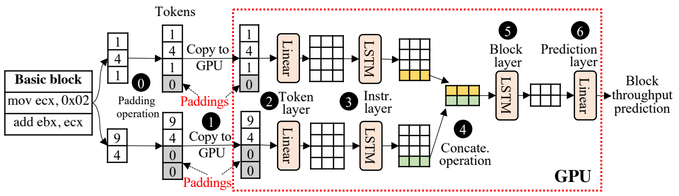


Fig. 22: Generalizing optimizations to offload Ithemal on GPUs.

Figure 22 shows how our proposed technical designs can be generalized to Ithemal [15]. When offloading Ithemal on GPUs, we observe similar *redundant communication* and *parallel simulation* challenges. Specifically, both Ithemal and SimNet use the information of instructions as input. The difference is while SimNet uses CNNs and tracks context instructions explicitly, Ithemal uses hierarchical and sequential LSTMs as a model to derive the dependency between instructions implicitly.

For latency prediction of a basic block, Ithemal hierarchically generates embeddings for each token, instruction, and the entire basic block sequentially. First, the tokens for each instruction are padded (⓿) and copied to GPU (❶). The token layer then generates instruction embeddings (❷). The instruction layer then uses an LSTM model to generate an instruction embedding sequentially from the token embeddings (❸). Then, the instruction embeddings are concatenated (❹) to eventually generate the block embedding (❺). Finally, Ithemal uses a linear layer to predict the block throughput (❻).

Although Ithemal does not require an instruction queue, it still faces redundant data movement due to the hierarchical embedding construction nature (❹). Furthermore, because different instructions are of various lengths, padding is required to use the TensorRT inference engine (⓿). Finally, the current Ithemal design provides limited parallelism. Therefore, the Ithemal workflow can benefit from the optimizations for SimNet (i.e., data movement reduction) and parallel simulation.

To avoid redundant data movement due to concatenation (❹), a custom LSTM layer can be implemented (❸) that writes the required output of the instruction layer in different memory locations so it may be used directly as input for the following hierarchy. Similarly, a sliding window can be used to batch more than one instruction at a time and transfer them to GPUs for better memory bandwidth utilization (❶). To avoid padding computation (⓿), a custom token layer can be implemented (❷). Finally, other optimizations such as TensorRT inference, half-precision, or pruning also may be used to accelerate inference, as well as pipelining the inference with host/device data transfers.

For parallel simulation, the original Ithemal uses basic-block-based thread parallelism, where each CPU thread works on a basic block independently. Yet, because the model is hierarchical and composed of different layers with the instruction and prediction layers being sequential, it imposes stringent data dependencies and cannot be efficiently parallelized in GPUs. However, analogous to SimNet, instructions from different basic blocks can be batched together for parallel inference. In this direction, we first parallelize the instruction embeddings at the token level across all instructions in the same basic block. Subsequently, the inference computation can be performed for various instructions across different basic blocks in a batched manner. This parallelism can be applied to both training and inference.

VIII. Related Work

Traditional simulator. Traditional computer architecture simulators simulate every component of the hardware. gem5 is one of the most popular simulators and supports a range of instruction set architectures (ISAs) and microarchitectures. ZSim is an x86 simulator that parallelizes many-core system simulation [9]. It decouples the simulation of individual cores with shared resources and adopts a simplified core model but has limited parallelism. FireSim [37] is a field programmable gate array (FPGA) emulator that runs significantly faster than software simulators but requires considerable effort to develop and validate register-transfer level models. In comparison, this work accelerates ML-based simulation on the most widely available ML accelerators, i.e., GPUs.

Simulation workload reduction. Simulation can also be accelerated by sampling representative instructions from the whole program. The primary goal is to process or simulate a smaller instruction trace by reducing the size of the input sets. MinneSpec [38] achieves this goal by generating a representative smaller dataset from a large one based on statistical characteristics (instruction mix, memory behavior, etc.). Similarly, Conte et al. [39] use a state-reduction method to sample the simulation traces statically. SimPoint [2] is a popular statistical method that uses a clustering technique to find representative instructions that can be simulated and statistically represent the performance of a whole program. SMARTS [3] is another simulation reduction method that samples fixed instructions at regular intervals to represent a program's performance. Instead of using clustering technique, SMARTS applies sampling theory to reduce the simulation traces. The simulation detailed herein can be orthogonally integrated with these techniques.

ML-based simulation and modeling. Ithemal and SimNet propose using deep neural networks for instruction-wise performance prediction [40]. Meanwhile, other works utilize ML for performance and power prediction without performing instruction-wise prediction. Specifically, [41], [42] formulates nonlinear regression models as a statistical inference tool to predict performance for aid in design space exploration. [43] uses performance counters as the input of an ML model and measurements from real hardware to predict GPU performance and power. Some approaches predict both performance and power based on those obtained on different types of processors [44], [45], [46] or ISAs [47], [48].

IX. Conclusion

This work addresses ML-based microarchitecture simulation from the perspective of its optimization, scalability, and practicality for actual use. We propose optimizations and parallelization techniques and offer a thorough performance evaluation of their impacts on the design and GPU implementation of efficient ML-based simulators. The optimizations proposed significantly reduce the time-to-solution of computer architecture simulation compared to discrete-event simulation with the same level of accuracy. These approaches are applicable and general to all ML-based simulators.

X. Acknowledgment

We thank the anonymous reviewers for their helpful suggestions and feedback. This work was supported in part by the Brookhaven National Laboratory through the SMaSH (Smart Modeling and Simulation for High Performance Computing) Project. Brookhaven National Laboratory is managed by the U.S. Department of Energy's Office of Science under Contract No. DE-SC0012704.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and et al. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[2] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, June 2003.

[3] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, page 84–97, 2003.

[4] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation Compute Benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 41–42, 2018.

[5] Andreas Sandberg, Nikos Nikoleris, Trevor E Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. Full Speed Ahead: Detailed Architectural Simulation at Near-native Speed. In *IEEE International Symposium on Workload Characterization (ISWC)*, pages 183–192, 2015.

[6] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. CMP$im: A Pin-based On-the-fly Multi-core Cache Simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[7] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[8] Curtis L Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P Kenny, Ali Pinar, David A Evensky, and Jackson Mayo. A Simulator for Large-scale Parallel Computer Architectures. *International Journal of Distributed Systems and Technologies (IJDST)*, 1(2):57–73, 2010.

[9] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, June 2013.

[10] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *ACM Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 45–57, 2002.

[11] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based Static Branch Prediction using Machine Learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[12] Daniel A Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *IEEE Seventh International Symposium on High-Performance Computer Architecture (HPCA)*, pages 197–206, 2001.

[13] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A Machine-Learning Supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[14] Wongyu Shin, Jeongmin Yang, Jungwhan Choi, and Lee-Sup Kim. NUAT: A Non-uniform Access Time Memory Controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 464–475, 2014.

[15] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation Using Deep Neural Networks. In *International Conference on Machine Learning (ICML)*, pages 4505–4515, 2019.

[16] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. SimNet: Computer Architecture Simulation using Machine Learning. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[17] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems (NIPS)*, 25:1097–1105, 2012.

[19] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[20] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. C-SAW: A Framework for Graph Sampling and Random Walk on GPUs. In *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15, 2020.

[21] Anil Gaihre, Xiaoye Sherry Li, and Hang Liu. Gsofa: Scalable sparse symbolic lu factorization on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):1015–1026, 2021.

[22] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, et al. Trust: Triangle Counting Reloaded on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, pages 2646–2660, 2021.

[23] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation Of Thousand-core Systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[24] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: Scalable and Accurate Parallel Multi-core Simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, pages 91–94, 2012.

[25] Thomas F Wenisch, Roland E Wunderlich, Babak Falsafi, and James C Hoe. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):408–409, 2005.

[26] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[27] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46, 2005.

[28] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2021.

[29] NVIDIA. TensorRT. https://developer.nvidia.com/tensorrt, 2021.

[30] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating Sparse Deep Neural Networks. *arXiv preprint arXiv:2104.08378*, 2021.

[31] Gary Lauterbach. Accelerating Architectural Simulation by Parallel Execution of Trace Samples. In *IEEE Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 1, pages 205–210, 1994.

[32] Oak Ridge National Lab. SUMMIT Oak Ridge National Laboratory's 200 Petaflop Supercomputer. Accessed: 2020, March 6.

[33] Fuentes Morales and Jose Luis Bismarck. Evaluating gem5 and qemu Virtual Platforms for ARM Multi-core Architectures, 2016.

[34] Yuetsu Kodama, Tetsuya Odajima, Akira Asato, and Mitsuhisa Sato. Evaluation of the Riken Post-k Processor Simulator. *arXiv preprint arXiv:1904.06451*, 2019.

[35] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.

[36] Christopher M Bishop and Nasser M Nasrabadi. *Pattern Recognition and Machine Learning*, volume 4. Springer, 2006.

[37] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *IEEE Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, page 29–42, 2018.

[38] AJ KleinOsowski and David J Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1(1):7–7, 2002.

[39] Thomas M. Conte, Mary Ann Hirsch, and W-MW Hwu. Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation. *IEEE Transactions on Computers*, 47(6):714–720, 1998.

[40] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently Exploring Architectural Design Spaces Via Predictive Modeling. In *ACM 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 195–206, 2006.

[41] B. C. Lee and D. M. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2007.

[42] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 249–258, 2007.

[43] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU Performance and Power Estimation Using Machine Learning. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, 2015.

[44] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *ACM Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, page 725–737, 2015.

[45] I. Baldini, S. J. Fink, and E. Altman. Predicting GPU Performance from CPU Runs Using Machine Learning. In *IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 254–261, 2014.

[46] Kenneth O'neal, Philip Brisk, Ahmed Abousamra, Zack Waters, and Emily Shriver. GPU Performance Estimation Using Software Rasterization and Machine Learning. *ACM Transaction on. Embedded Computer System (TECS)*, 16(5s), 2017.

[47] Xinnian Zheng, Pradeep Ravikumar, Lizy K John, and Andreas Gerstlauer. Learning-based Analytical Cross-platform Performance Prediction. In *IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 52–59, 2015.

[48] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. Accurate Phase-level Cross-platform Power and Performance Estimation. In *ACM Proceedings of the 53rd Annual Design Automation Conference (DAC)*, New York, NY, USA, 2016.