

XPU-Point: Simulator-Agnostic Sample Selection Methodology for Heterogeneous CPU-GPU Applications

Alen Sabu[§]

Arm Ltd

Harish Patil

Intel Corporation

Wim Heirman

Intel Corporation

Changxi Liu

National University of Singapore

Trevor E. Carlson

National University of Singapore

Abstract—Heterogeneous computing has become increasingly prevalent, driven by the huge computational demands of high-performance computing (HPC) and artificial intelligence (AI) workloads. Yet, evaluating these workloads on modern systems poses significant challenges. Existing tools for the instrumentation and analysis of CPU and GPU applications run separately, introducing timing differences and limiting the ability to capture their runtime interactions. To address this problem, we introduce XPU-Pin, a framework that enables simultaneous CPU and GPU binary instrumentation in a single execution. XPU-Pin integrates the CPU instrumentation framework Pin with GPU instrumentation frameworks such as NVBit (for NVIDIA GPUs) and GTPin (for Intel GPUs). This approach allows for the holistic analysis of heterogeneous workloads irrespective of the platform it executes.

Leveraging the co-analysis capabilities of XPU-Pin, we present a novel methodology called XPU-Point to select simulator-agnostic representative samples of heterogeneous CPU-GPU workloads. The XPU-Point methodology employs tools developed with the XPU-Pin framework to: (a) capture the execution signature of heterogeneous programs and (b) evaluate the accuracy of selected samples on silicon, which were not possible before. We evaluate XPU-Point on diverse hardware platforms (x86 CPU with Intel/NVIDIA GPUs) using workloads such as SPECaccel 2023, SPEChpc 2021, GROMACS, AutoDock, and PyTorch. We demonstrate that XPU-Point predicts overall application performance with sampling errors typically less than 5% as measured on native hardware.

Index Terms—Heterogeneous systems, performance evaluation, sampled simulation, binary instrumentation

I. INTRODUCTION

Computation exists everywhere in this era, spanning from large-scale systems to low-power devices and mobile CPUs. As an example, there has been a profound increase in the demand for high-performance computing (HPC) resources in recent years [1]. However, the limitations of multi-core architectures to scale due to the associated power and thermal constraints (power wall) restricts their ability to deliver significant performance improvements [2], [3]. This has resulted in a shift toward domain-specific architectures and accelerators like

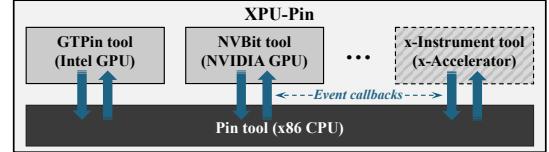


Fig. 1: A high-level schematic of XPU-Pin. The x86 CPU instrumentation tool Pin interacts with GPU instrumentation tools (like GTPin and NVBit) for event-based callbacks. Integration with similar tools for other hardware components ($x=$ TPUs, NPUs, accelerators, etc.) is feasible. The simulation phase (not discussed in this work), which is performed using a variety of tools, is handled separately.

GPUs [4], TPUs [5], and FPGAs [6]. Embracing heterogeneity in architectures is one way forward for continued performance improvements [7] to meet these growing computational demands. The use of a combination of architectures is needed to continue to scale the performance of future systems [8], to achieve accelerator-level parallelism [9].

The prevalence of CPU-GPU architectures in heterogeneous computing arises from their ability to address the evolving demands of modern workloads, coupled with their well-established programming models and their ability to exploit parallelism at a massive scale. GPUs have emerged as the most widely used general-purpose accelerators in modern data centers [10] and supercomputers [11] that accelerate massively parallel big data analysis [12], [13] and machine learning [14], [15] workloads. While previous works have investigated characterizing workloads that consist of CPU components [16]–[20] and GPU [21]–[24] components independently, as well as their comparative analyses [25], hybrid solutions that support analysis and workload reduction, like sampling, for multiple types of heterogeneous workloads from CPUs, GPUs, and even custom hardware accelerators have not yet been identified. Given the importance of these workloads, from HPC systems to data center use, simulation of heterogeneous workloads is key to understanding the interactions between compute

[§]This work was completed while the author was at the National University of Singapore.

components and how those affect overall runtime performance.

The growing significance of heterogeneous computing architectures necessitates a refined approach to performance analysis. While GPUs have become indispensable for accelerating workloads like AI training and inference, the CPU plays a critical role in scheduling tasks and managing memory. A performance bottleneck within the CPU can have a cascading effect, given Amdahl’s law [26], impacting overall system performance. Prior works [27] discuss the shortcomings of traditional GPU-centric analysis methods that overlook the role of the CPU in data movement and task management. Sampled simulation techniques, while prevalent for independent CPU and GPU performance analysis, suffer limitations when applied to tightly coupled CPU-GPU systems. Such simulations or performance analyses often neglect the effects of inter-core communication and cache coherency, that significantly impact the microarchitectural state. Additionally, they may not accurately capture synchronization behavior, leading to unrealistic execution order and resource usage.

Existing instrumentation and analysis tools are insufficient to capture the interactions between CPU and GPU in heterogeneous applications. While there are instrumentation and analysis frameworks for CPUs, such as Pin [28] or DynamoRIO [29], [30] for x86 programs, and for GPUs, such as GTPin [31] for Intel GPU programs and NVBit [32] for NVIDIA GPU programs, there is no framework for co-analysis of CPU and GPU code. In this paper, we introduce *XPU-Pin*, a novel framework designed to bridge this gap by enabling simultaneous analysis of both CPU (x86) and GPU (Intel, NVIDIA) code. XPU-Pin has a Pin-based driver that loads the GPU tool library (GTPin or NVBit) explicitly and triggers it, as shown in Figure 1. CPU and GPU analyses can thus be integrated within the same environment, simplifying development and allowing for a unified and more accurate analysis. Additionally, the GPU tool can trigger functions registered by the driver on certain GPU events, such as the start or end of a GPU kernel. The CPU and GPU tools can thus coordinate their analyses around GPU events.

Evaluating the performance of large workloads on heterogeneous systems presents significant challenges due to long simulation times, which can take several months or years, as illustrated in Figure 2. Training large language models (LLMs) with multi-billion parameters [35]–[37] can take several months, while the inference runs may take several seconds even on powerful hardware [38], [39]. Simulation serves as a powerful tool for architects to explore potential hardware improvements that suit certain workload types. However, simulating such workloads in their entirety can be prohibitively long. Workload sampling stands as a popular technique for CPUs [16], [17], [19], [40], [41] and GPUs [21]–[23], [42], presenting a compelling solution by selecting a representative subset of the workload for detailed simulation. This approach delivers substantial speedups while maintaining accurate performance estimates. However, there are currently no sample selection solutions that apply to heterogeneous workloads. We build on XPU-Pin to propose *XPU-Point*, a

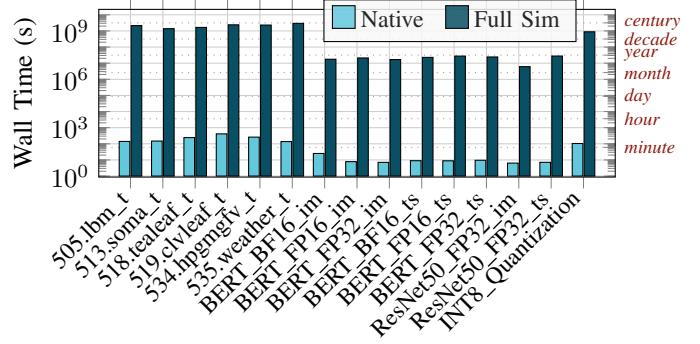


Fig. 2: The wall time (in seconds) comparison between native execution and full-detailed simulation for realistic heterogeneous CPU-GPU workloads, such as SPECChpc 2021 benchmarks (tiny set) using ref inputs and PyTorch Inference runs, on machines with Intel Sapphire Rapids CPU and Intel Ponte Vecchio GPU. The simulation wall times were estimated using the simulation rates of gem5 [33] and Accel-Sim [34]. im=Imperative, ts=TorchScript.

unified sample selection solution for heterogeneous workloads that can accurately build a representative sample for the fast and accurate performance analysis of such workloads. Through XPU-Point, we propose a comprehensive methodology across a broad spectrum of real-world workloads, from scientific simulations to artificial intelligence. This enables computer architects and performance researchers to quickly estimate the performance of long-running, heterogeneous workloads using sampled simulation on existing simulators [43], [44] which was not possible before.

The accuracy of the XPU-Point methodology is assessed (sample validation) based on sampling errors – the difference between the full workload performance and the performance extrapolated from the samples. Traditionally, sample validation is performed based on a detailed, and slow, timing simulation platform. We have identified two issues with simulation-based sample validation (i) it assumes that an accurate simulator for the target system is available which is not the case in the early stages of system design and (ii) it requires simulation of the entire test program to get the full workload performance which can be impractically slow as illustrated in Figure 2. We instead separate sample validation from simulation and perform the validation on real hardware with *XPU-Timer* (Figure 3). Using XPU-Timer, sample validation can be performed at near-native speed, whereas simulation-based validation can be significantly slower.

The high-level overview of the entire framework is shown in Figure 3. The focus of XPU-Point methodology is on selecting samples for simulation and validating those samples in a simulator-independent manner. The samples can be used to drive simulation using the platform of choice. Leveraging XPU-Point samples for simulation is left for future work.

In this work, we make the following contributions:

- We propose XPU-Point, a methodology that goes beyond

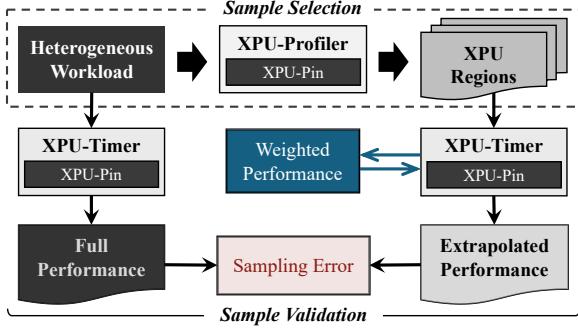


Fig. 3: The end-to-end workflow of the XPU-Point methodology to sample heterogeneous workloads. XPU-Point uses XPU-Profiler to capture execution profiles of a heterogeneous workload. Once the representative regions (samples) are identified for the workload, their performance, as estimated by XPU-Timer (or a heterogeneous simulator), is extrapolated and compared with that of the full workload to validate the sample.

prior sample selection techniques to be the first to allow for the support of heterogeneous applications. This enables researchers to conduct a unified and accurate performance evaluation of large-scale applications through sampled simulation.

- ii. We introduce XPU-Pin, an instrumentation framework that we developed in this work to evaluate heterogeneous CPU-GPU applications. XPU-Point is built upon XPU-Pin, and supports both Intel- and NVIDIA-based CPU-GPU workloads.
- iii. We experimentally assess the efficacy of XPU-Point in terms of sampling accuracy and potential simulation speedup of CPU-GPU workloads on various hardware platforms using our XPU-Timer tool instead of using a simulator. We have open-sourced the XPU-Pin framework and the XPU-Point project on GitHub [45].
- iv. We extensively evaluate XPU-Point across several heterogeneous workloads, including industry-standard benchmarks such as SPECaccel 2023, SPEChpc 2021, AutoDock, GROMACS, and PyTorch inference demonstrating high accuracy (absolute sampling errors less than 5%).

The rest of the paper is organized as follows. In Section II and Section III, we discuss the background and challenges involved in the performance evaluation of heterogeneous workloads on modern architectures. Section IV presents the XPU-Point methodology in detail. We then describe the experimental infrastructure in Section V, followed by an extensive evaluation of XPU-Point in Section VI to demonstrate the applicability of the proposed methodology. Finally, we present the related work in Section VII and conclude the paper in Section VIII.

II. XPU-PIN FRAMEWORK

In this section, we explore prominent solutions for binary instrumentation, along with insights into programming models

designed for heterogeneous workloads. We also delve into the development of XPU-Pin, a tool we specifically built to facilitate the co-analysis of CPU-GPU heterogeneous workloads.

A. Instrumentation and Analysis Tools

1) *Intel Pin*: Pin [28] is an x86 binary instrumentation framework that allows users to write Pin tools, which are C/C++ programs specifying analysis to be done at certain points (for example, every instruction, basic block, etc.) in program execution. Pin works in two modes, namely JIT mode and Probe mode. JIT mode works by loading an input x86 binary in memory and translating (just-in-time) its x86 code into x86 code to another region of memory called the code cache. During translation, Pin inserts extra code, specified by a PinTool, at specific points (such as an instruction, basic block, or routine) within the translated code. The translation overhead in the JIT mode can result in a 40% slowdown [46] in performance even without instrumentation taking place. The slowdown and performance perturbation can be exacerbated further by the additional analysis routines. In contrast to JIT mode, probe mode works by loading an input x86 binary in memory and patching the code at certain probe points as specified by the Pin tool. Probe mode does not translate the code, but instead redirects the code inline to analysis routines. In general, Probe mode demonstrates lower overheads at the cost of programmer effort.

2) *Intel GTPin*: Intel’s GPU instrumentation framework, GTPin [31], works by inserting analysis code into the GPU program via GTPintools as shared libraries. The Intel Graphics compiler generates code for the specific Intel GPUs at run time. GTPin then dynamically adds extra code specified by a GTPintool into the generated code. This modified code is then offloaded to the GPU and runs there. Any results created by the extra GTPintool code are stored in a memory buffer, and that buffer is processed on the CPU at various synchronization points.

3) *NVIDIA NVBit*: NVBit [32] is a dynamic binary instrumentation framework for NVIDIA GPUs that works on the Linux operating system. It provides a high-level Application Programmer’s Interface (API) for writing instrumentation tools as Linux-shared libraries. A tool library is injected in a GPU application using the LD_PRELOAD [47] feature in Linux. NVBit tools can inspect and modify the NVIDIA GPU assembly code (SASS) of GPU applications without requiring recompilation.

B. Implementation of XPU-Pin Framework

Existing tools mainly focus on either CPU or GPU components of an application due to the limitations of traditional instrumentation tools. Intel Pin [28] and DynamoRIO [29], [30] are used to analyze CPU applications, while GTPin [31] and NVBit [32] are used for GPU kernel analysis. In contrast, our newly designed framework, XPU-Pin, allows users to analyze and instrument heterogeneous CPU-GPU workloads with a single tool. XPU-Pin starts as an x86 analysis tool based on Pin (either JIT or Probe mode) and then invokes the GPU analysis

<code>xpu-pin-driver.so</code>	<code>GPUAnalysis.so</code>
<code>dl_open (GPUAnalysis.so)</code> <code>[optional]</code> <code>GPUAnalysisRegisterCallbacks (</code> <code>callback_on_kernel_build,</code> <code>callback_on_kernel_run,</code> <code>callback_on_kernel_complete)</code> <code>[GT-Pin only]</code> <code>GTPin_Entry (argc_gtpin, argv_gtpin)</code>	<code>[optional]</code> <code>GPUAnalysisRegisterCallbacks (..)</code> <code>OnKernelBuild</code> <code>• cpu_callback_on_kernel_build (kernel_name)</code> <code>OnKernelRun</code> <code>• cpu_callback_on_kernel_run (kernel_name)</code> <code>OnKernelComplete</code> <code>• cpu_callback_on_kernel_complete (kernel_name)</code> <code>[GT-Pin only]</code> <code>GTPin_Entry (..)</code> <code><start GPU analysis></code>

Fig. 4: The control flow of XPU-Pin co-analysis tool for an x86 CPU and Intel GPU or NVIDIA GPU.

shared library. The straightforward approach of linking in a GPU analysis library is not an option, as Pin requires all the libraries a Pin tool uses to be built with a special Pin runtime provided. Modifying GPU analysis tools to use special Pin runtime can be restrictive and not practical with a large number of legacy GPU analysis tools. For Intel GPUs, an alternative to linking in a GTPin tool library is to explicitly load it at runtime from the driver Pin tool. However, this requires using `dlopen()` from the application's runtime instead of Pin runtime. For NVIDIA GPUs, the NVBit analysis framework utilizes the LD_PRELOAD feature to inject itself into the application. This mechanism remains effective when combined with x86 Pin. CPU and GPU analyses can thus be integrated within the same environment, simplifying development and allowing for a unified and more accurate analysis.

Legacy GPU analysis tools can thus be executed unmodified with x86 CPU analysis using Pin (either using `dlopen` or LD_PRELOAD). For coordinated CPU and GPU analysis, GPU analysis tools can implement an optional callback handler registration. When the Pin driver explicitly loads the GPU analysis library, it calls this registered handler. The GPU tool then uses this handler to obtain and store pointers to Pin driver functions, which it can later invoke in response to specific GPU events (like kernel start and end). This mechanism enables CPU and GPU tools to synchronize their analysis based on these GPU events. Figure 4 shows the control flow for an XPU-Pin tool combining x86 Pin tool (`xpu-pin-driver.so`) and GTPin analysis tool (`GPUAnalysis.so`). Including a GPU analysis library can inadvertently cause the CPU analysis tool to treat it as part of the application. To prevent this, the CPU tool must explicitly exclude it from instrumentation. The XPU-Pin framework provides an API enabling the CPU analysis tool to retrieve the name of the GPU analysis library for this purpose.

III. THE IMPERATIVE FOR EFFICIENT HETEROGENEOUS SIMULATION

This section highlights the need for efficient methodologies to evaluate the performance of large workloads running on heterogeneous computing systems in a fast and accurate way.

A. Limitations of Traditional Analysis Methodologies

Traditional heterogeneous systems tend to underutilize the available computing power of CPU and GPU [48], [49]. Most traditional heterogeneous applications use the CPU to schedule

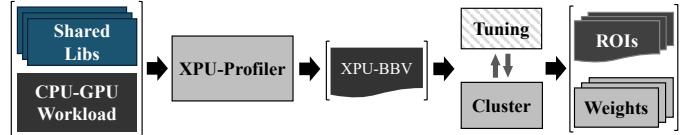


Fig. 5: The workflow of XPU-Point methodology to capture representative regions (or ROIs) along with their corresponding weights suitable for the sampled simulation of heterogeneous workloads.

computing tasks for accelerators like GPUs. While the highly parallel computation happens in the GPU, the CPU waits, causing the CPU cycles to be wasted. However, this may not be the case with emerging applications that may fully utilize the CPU resources by executing tasks concurrently on the CPU and GPU. Independent performance evaluations using simulation techniques can yield misleading microarchitectural state estimations due to the misrepresentation of synchronization between the processing units. These evaluations cannot accurately capture the shared memory and cache access patterns, which are influenced by the underlying cache coherency protocols. Consequently, resource usage and execution order might be misrepresented. Therefore, co-analysis and co-simulation techniques are essential for accurate microarchitectural state evaluation in CPU-GPU systems.

B. Effective Sample Selection of Heterogeneous Workloads

There are several methodologies that address the problem of sampling single-threaded [16], [17], [50] and multi-threaded [19], [20], [40], [41], [51] CPU workloads. There are several sampled simulation techniques that consider GPU workloads [21]–[24], [42], [52], [53] to speed up GPU-only simulation. Among the prior works for the sampled simulation of GPU workloads, Kambadur et al. [22] proposed a GTPin-based methodology for the sample selection of Intel GPU workloads, utilizing basic block information (along with other program features) to characterize program execution. Prior works like TBPoint [21] and PKA [23] utilize handpicked feature vectors, including kernel size and control flow divergence, to classify similar GPU regions. Photon [42] employs a cluster-based summarization technique that groups similar warps based on their behavior and constructs BBVs for each cluster by aggregating individual warp profiles and concatenating them. All these works consider GPUs as independent computing units, which rely on the assumption that the heterogeneous workload could be divided into CPU-only and GPU-only components. Under this assumption, the total execution time of the heterogeneous application can be calculated by summing up the CPU execution time, GPU execution time, and the data transfer time between CPU and GPU. However, this assumption may no longer be valid for emerging workloads. Independent analyses may result in inconsistent timings for workload-specific CPU and GPU events, such as kernel launches, memory allocations, and warp divergence. Therefore, a combination of CPU-only and GPU-only sample

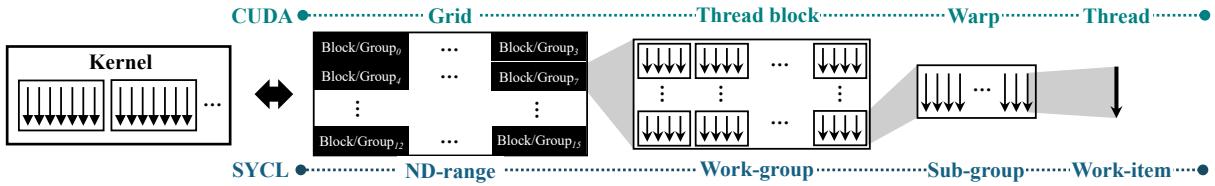


Fig. 6: A comparison of the hierarchical structures used in CUDA and SYCL programming models to distribute kernel execution tasks, showing the level of granularity at which work is assigned to the execution units. CUDA primarily utilizes the SIMT execution model, while in SYCL, underlying architecture and implementations determine the execution model.

selection methods for heterogeneous systems could lead to inaccurate performance measurements.

C. Effects of Microarchitectural Warmup

In sampled simulation, microarchitectural warmup is necessary to ensure the simulated microarchitecture reflects a realistic state prior to detailed performance measurements. Previously proposed microarchitecture warmup methodologies [54]–[57] enable the detailed simulation of the regions of interest starting at the right state. Warmup methodologies can be categorized into functional warming and statistical warming. Functional warming techniques [19], [40] rely on actual program execution, whereas statistical warming [54]–[57] leverages profiling tools to gather memory access information.

XPU-Point provides a framework for collecting memory access data necessary for developing integrated CPU-GPU warmup methodologies. Architects tend to integrate computing devices like CPUs and GPUs to share last-level cache, memory, etc. For instance, NVIDIA Grace-Hopper [58] utilizes a CPU-GPU coherent memory model. The trend towards tightly coupled CPU-GPU computing will continue, especially with the advancement in interconnects (like CXL [59] and NVLink [60]) and chiplet-based [61] IC packaging [62] technologies. XPU-Point can be extended to gather shared memory access patterns, enabling the generation of combined warmup data that addresses this crucial requirement in integrated GPU systems and multi-GPU systems.

IV. XPU-POINT SAMPLE SELECTION METHODOLOGY

In this section, we introduce XPU-Point, a novel methodology to sample heterogeneous CPU-GPU workloads. To the best of our knowledge, XPU-Point represents the first solution to efficiently co-sample heterogeneous workloads. The overall workflow for the XPU-Point methodology is outlined in Figure 5. The methodology relies on our XPU-Profiler to generate the combined execution signature vectors of CPUs and GPUs. These heterogeneous execution vectors are clustered to identify representative regions that can be used for simulation-based performance evaluations of future heterogeneous architectures.

A. Workload Distribution on GPUs

GPUs follow a hierarchical structure in both their hardware and programming models to efficiently manage the massive number of threads. NVIDIA GPUs typically comprise multiple

Streaming Multiprocessors (SMs), with each SM containing several CUDA cores. To leverage the parallel architecture of NVIDIA GPUs, several threads (usually 32 or 64 threads) are grouped into a warp (or waveform). NVIDIA GPUs primarily utilize a Single Instruction, Multiple Thread (SIMT) [63] model of CUDA, where threads within a warp share the same program counter (PC) and consequently execute the same instruction concurrently on the same CUDA core. Furthermore, multiple warps are grouped into thread blocks, which are then scheduled for execution on the same SM and utilize the shared cache. The GPU kernels (functions offloaded to the GPUs for parallel processing) are structured as Grids to orchestrate the execution across all thread blocks.

The concept of work groups in SYCL directly maps to how Intel’s Xe [64] cores distribute tasks. Work-groups, analogous to thread blocks, group a defined number (SIMD-width) of threads for cooperative execution and data sharing. Intel GPUs typically employ a more flexible SIMD (Single Instruction, Multiple Data) [65] model redesigned for high performance [66] on Intel GPUs in the SYCL programming model. This means that threads within a work-group can execute a single instruction on multiple data elements simultaneously. Work-groups are subdivided into sub-groups (similar to warps) that share resources like local memory. Execution occurs on Vector Engines (VE) within the Xe cores. SYCL employs queues to manage the submission of work-groups for execution on the VE. ND-range defines the high-level structure of the kernel for parallel execution across the processing elements, specifying a multidimensional grid of thread blocks to be launched on the GPU. Figure 6 shows the workflow of a GPU kernel execution on Intel and NVIDIA systems.

XPU-Point takes into account both CUDA and SYCL programming models to represent the amount of execution done by the device. The execution in traditional CPU workloads can be quantified by the number of instructions or basic blocks (code blocks that have single entry and exit points) executed by each thread. In this work, we adopt a similar approach to quantify the execution of GPUs. However, GPU execution differs due to the SIMT/SIMD paradigm, where multiple threads or work-items collaborate to execute an instruction. To account for this, XPU-Point uses warp (or subgroup) as the fundamental unit of execution for GPUs, whereas instruction is the fundamental unit of execution for CPUs.

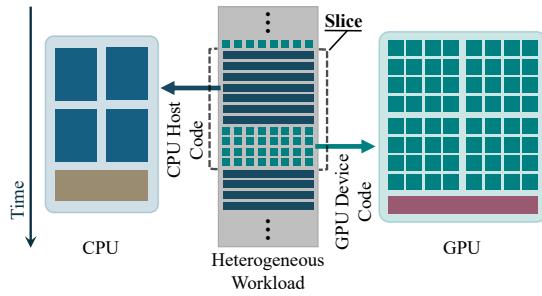


Fig. 7: The representation of a slice (or region) in XPU-Point. A slice is defined as the execution window between consecutive kernel calls within a heterogeneous application.

B. Slices of Heterogeneous Applications

A slice (or region) represents a chosen segment of the application’s execution flow generated by splitting the application at a well-defined point. For a slice to be effective for workload characterization, it must be *repeatable* across multiple runs of the application to ensure consistency in the behavior for accurate sampled simulation and performance evaluation. Traditional CPU workload sampling methodologies such as SimPoint [16], BarrierPoint [20], and LoopPoint [41] identify slices based on repeatable program constructs. Simpoint, for instance, focuses on identifying intervals based on fixed-size instructions. Meanwhile, BarrierPoint and LoopPoint target regions that are delineated by synchronization barriers and loops, respectively. Previously proposed sampled simulation techniques for GPU workloads [21], [23], [42] focus solely on the GPU kernels, completely ignoring any interactions with the CPUs. In this work, we propose a novel approach for slice identification, which is a contiguous code segment that spans from the end of one kernel call to the end of the subsequent kernel call, as shown in Figure 7. Therefore, the slice of a heterogeneous application includes both CPU and GPU execution. Similar to loops in LoopPoint or inter-barrier regions in BarrierPoint, the slices identified by XPU-Point are repeatable across multiple executions on platforms with similar compute capabilities.

C. Capturing CPU-GPU Execution Profiles

Understanding execution profiles within CPU-GPU systems demands a comprehensive representation that integrates execution profiles from both processing units. Traditionally, the classification of regions based on the similarity of the code executed [67], [68] works well for CPU workloads. The regions are represented as basic block vectors (BBVs), which comprise basic blocks and their frequency. A number of prior works on selecting representative regions of a workload have been built on this idea of representing regions using code signatures. In the case of multi-threaded workloads where threads split the work to execute on multiple cores, the amount of work done by the threads is represented by concatenating the BBVs of each thread. Concatenating BBVs across CPU

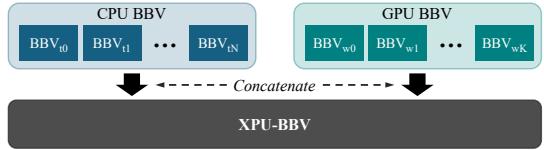


Fig. 8: The concatenation of CPU and GPU BBVs into a longer, combined XPU-BBV that represents a heterogeneous region in XPU-Point methodology.

and GPU threads is evident to be a promising technique, as it merges CPU thread profiles with detailed GPU data, including both global and individual thread profiles. Prior works show that concatenating per-thread profiles to form CPU BBV [20], [41], [69] or per-warp profiles to form GPU BBV [42] leads to the accurate representation of thread-level parallelism.

In this work, we devise a technique to represent the heterogeneous regions of the workload. We utilize XPU-Profiler, the profiling tool that we built upon XPU-Pin, to simultaneously generate BBVs for CPU and GPU execution. Within this framework, we refer to CPU BBVs as those derived from program execution on the CPU, while GPU BBVs refer to those obtained during program execution on the GPU. We demonstrate that concatenating all GPU warps (or sub-groups) is efficient in representing the GPU BBV. By concatenating the CPU BBV and GPU BBV forming XPU-BBV (shown in Figure 8), we construct a unified representation that captures the behavior of a heterogeneous region. Previous studies [20], [41] have demonstrated that concatenating feature vectors effectively captures individual thread behavior.

D. Selecting the Representative Slices

Representing the sheer number of GPU threads (or warps) in an XPU-BBV leads to a significant increase in vector dimensionality, resulting in the *curse of dimensionality* [70]. This phenomenon slows down clustering algorithms, which are critical for identifying representative regions from the profile data. To address this challenge, we employ traditional dimensionality reduction techniques such as random linear projections. The algorithm selected for a desired level of dimensionality reduction is pivotal in minimizing information loss within the profile data [71]. This ensures that the resultant lower-dimensional space retains the vital characteristics necessary for accurate workload characterization. Due to the differences in magnitude between CPU and GPU dimensions, these feature vectors need to be projected separately. Further, we employ k-means clustering algorithm [72] to cluster these heterogeneous regions as represented by the lower-dimensional BBVs. The region closest to the centroid of each cluster serves as the representative of the cluster [16]. Typically, the clustering phase involves fine-tuning parameters, such as *maxk*, to achieve optimal results. Advances have been made in program representation using embeddings [73] and in clustering using deep neural nets [74]. We believe that such improvements are orthogonal to the basic idea of XPU-Point and can be incorporated here.

E. Sample Validation and Tuning

The representativeness of the regions selected using the proposed methodology needs to be validated. In addition to simulations, sample validation employing real hardware measurements like those demonstrated in prior works [24] can be leveraged here. To validate the representativeness of the selected slices within heterogeneous workloads, we introduce XPU-Timer, a tool built upon XPU-Pin. XPU-Timer leverages the x86 `rdtsc` instruction to provide system timestamps (TSCs) at critical points during the native execution of the workload: program start, program end, and the boundaries of each predefined slice. These timestamps allow us to extract the execution time for each representative slice, eliminating the need for long-running simulations. By weighing these region execution times with their corresponding weights, we extrapolate the execution time of the entire program. The difference between the full execution time and the extrapolated time is known as the sampling error (or prediction error). A lower sampling error indicates a more accurate selection of representative slices.

F. Estimating Full Application Performance

Microarchitecture simulation and exploration greatly benefit from sampling large, heterogeneous workloads. Instead of simulating entire workloads for microarchitecture exploration, which is computationally expensive, representative slices of the workload can be simulated rapidly. These slices capture the complex characteristics of heterogeneous workloads, enabling researchers to explore how future microarchitectures can be optimized for such workloads.

XPU-Point identifies representative slices of a heterogeneous workload that can be used for detailed cycle-level or cycle-accurate microarchitecture simulations. The regions can also be simulated on execution-driven heterogeneous simulators like Multi2Sim [75] and gem5-gpu [43]. We would like to point out that the XPU-Pin framework can be used for heterogeneous trace generation to support trace-driven simulators like MacSim [76]. Accurate performance measurements in sampled simulations rely on a warmed-up microarchitectural state before detailed simulation. However, warmup reconstruction for heterogeneous systems remains an open research area. As this falls outside the scope of our current work, we will not explore it further in the paper.

V. EXPERIMENTAL SETUP

Establishing the fidelity of sample selection techniques – the ability to accurately represent full program behavior using the selected sample set – is essential. Simulation-based performance comparisons are typically employed to assess this characteristic. However, this approach is impractical for large, heterogeneous workloads. Thus, we follow a hardware-based performance measurement methodology using XPU-Timer for all applications. XPU-Timer utilizes system timestamps to accurately measure the execution time of large applications and their selected samples on native hardware, eliminating the need for long-running simulations.

TABLE I: The CPU-GPU combinations of Intel- and NVIDIA-based systems used to evaluate XPU-Point methodology.

CPU	GPU
Intel Alder Lake [84]	Intel Discrete Graphics 2 (DG2)
Intel Alder Lake	Intel Iris Xe (integrated)
Intel Ice Lake-SP [85]	Intel Ponte Vecchio (PVC) [86]
Intel Sapphire Rapids [87]	Intel Ponte Vecchio (PVC)
Intel Cascade Lake [88]	NVIDIA A100 [89]
Intel Skylake [90]	NVIDIA GeForce GTX 1080
Intel Skylake	NVIDIA TITAN Xp

We evaluate a combination of standard heterogeneous benchmarks as well as real-world HPC and AI workloads that use both CPUs and a GPU for computation. We evaluate SPECaccel 2023 [77] benchmarks and SPEChpc 2021 [78] benchmarks, along with real-world workloads like AutoDock [79]–[81], GROMACS [82], and PyTorch [83] inference runs.

We evaluate all of the workloads with XPU-Timer using native hardware runs on both Intel-GPU-based and NVIDIA-GPU-based heterogeneous systems, and, therefore, we separately compile the benchmarks suitable for these systems. For Intel-based systems, we use Intel’s oneAPI [91], [92] toolkit to build the benchmarks, whereas for NVIDIA-based systems, we use the NVIDIA CUDA toolkit [93]. The machines that we used for our evaluation are listed in Table I. Our focus is on demonstrating the methodology’s efficacy across heterogeneous workloads on both Intel-based and NVIDIA-based GPU systems. The results presented here are not a direct comparison of individual machine performance but rather highlight the broad applicability of the methodology.

VI. EVALUATION

This section evaluates the effectiveness of XPU-Point in selecting representative regions using realistic CPU-GPU heterogeneous workloads. The aim of this work is to allow for fast and accurate microarchitecture simulations of these workloads to explore future heterogeneous systems.

We extrapolate the performance of the full workload from the performance of N representative regions using the formula:

$$P_{proj} = \sum_{i=1}^N P_i \times multiplier_i,$$

where P_{proj} denotes the projected or extrapolated performance of the full workload. In addition, P_i and $multiplier_i$ denote the performance obtained and the multiplier associated with the representative region $region_i$, respectively. The multiplier of a representative region is dependent on the number of regions that belong to the cluster that $region_i$ represents [20]. This formula allows us to extrapolate performance metrics like runtime, cache behaviors, branch behaviors, and IPC for the entire workload.

Sampling Error and Speedup. We quantify the difference between the extrapolated performance metrics and the actual measured performance of the full workload to obtain sampling error or prediction error [16]. We estimate the performance

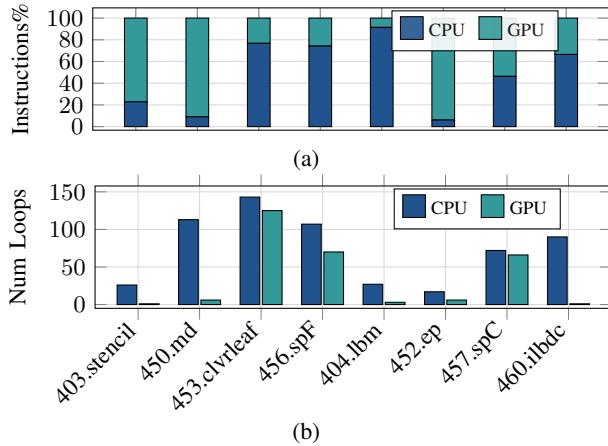


Fig. 9: Analysis of loop execution for SPECaccel 2023 benchmarks using train inputs. The top graph (a) shows the instruction split while the bottom graph (b) shows the number of loops executed on CPU and GPU.

of the workloads and the representative regions leveraging the system timestamp counter (TSC) on real hardware, which is equivalent to runtime obtained through microarchitecture simulation. The long simulation times required for full workloads make simulation-based validation impractical for large workloads.

The sampling error Δ_{sample} can be computed using the formula:

$$\Delta_{sample} = \left| 1 - \frac{P_{proj}}{P_{real}} \right|,$$

where P_{real} is the actual performance obtained through the measurement of the full workload. Usually, sampling error is expressed as a percentage (error rate). This is obtained by multiplying the absolute value of the sampling error (Δ_{sample}) by 100.

We define the speedup obtained using XPU-Point as the reduction in the number of instructions to be simulated in detail after sampling [20]. That means,

$$speedup = \frac{num_slices}{N},$$

where N is the number of representative regions, and num_slices is the total number of slices in the entire workload. This is equivalent to the serial speedup, which is achieved by simulating the representative regions sequentially. Note that this speedup represents the minimum achievable reduction in simulation time. The simulation of these representative regions can be parallelized, which could lead to significantly higher speedups than the values presented here.

Cross-microarchitecture Validation. XPU-Point methodology relies on the microarchitecture-independent selection of representative regions. This allows researchers to profile an application binary on one hardware and reuse the chosen regions for simulations on different hardware within the same architecture. This is possible because XPU-Point utilizes

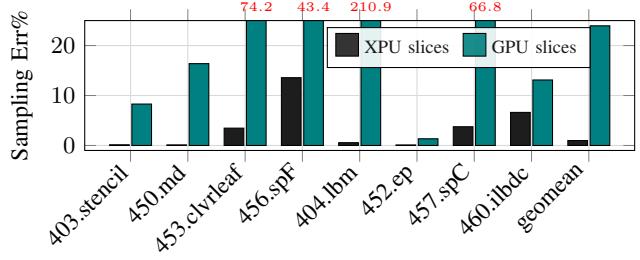


Fig. 10: The sampling errors for the SPECaccel 2023 benchmarks with GPU-only profiles (GPU-Point) vs. CPU-GPU profiles (XPU-Point).

BBVs to capture the control flow structure of the program, a characteristic independent of the underlying architecture. To verify the effectiveness of this approach across microarchitectures, XPU-Point employs cross-microarchitecture validation. This validation involves selecting regions on one machine using XPU-Profiler and then validating their representativeness on another machine with a different CPU-GPU combination within the same architecture using XPU-Timer.

Runtime Overhead. XPU-Point, like other dynamic binary instrumentation tools, introduces analysis-dependent runtime overhead. XPU-Profiler has a large overhead due to extensive library usage and process/thread creation of large workloads. By default, it instruments all libraries, processes, and threads. However, XPU-Point offers the flexibility to reduce the analysis overhead by instrumenting specific processes/threads and libraries. The XPU-Timer tool employs a Pin-probes mode driver, avoiding CPU instrumentation altogether. The GPU component of the tool utilizes low-overhead instrumentation to track key events like GPU initialization and kernel start/stop.

A. Comparison with GPU Sample Selection

We present a detailed analysis of SPECaccel 2023 [77], a benchmark suite with computationally intensive parallel heterogeneous applications that exercises the performance of the accelerator (GPU in our case), host CPU, memory transfer between host and accelerator, compilers, and the runtime system [94]. We used Intel x86 Sapphire Rapids server with Intel Data Center GPU Max 1100 for this evaluation.

Figure 9 shows the analysis of loops in the main image of the benchmarks identified using Intel Software Development Emulator (Intel SDE) [95]. The number of loops on the GPU was obtained using Intel GTPin [31]. For SPECaccel 2023 workloads using CPU and GPU, we wanted to test the effect of focusing just on the GPU computation. We tested two profilers: XPU-Profiler that collects combined CPU-GPU BBVs; and GPU-Profiler that collects the GPU BBVs. In both the cases, the region boundaries are kernel boundaries leading to the same number of BBVs. GPU-Profiler uses Intel GTPin to collect per-thread BBVs for the entire computation which are copied to the CPU at the end of each GPU kernel execution. The average slowdown of the GPU-Profiler for SPECaccel test cases was $4.7\times$. XPU-Profiler on the other hand uses Pin JIT

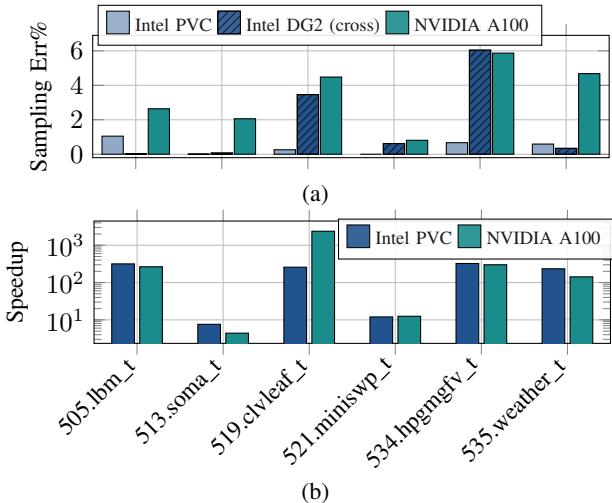


Fig. 11: The sampling errors (a) and speedup (b) plotted with shared x-axis obtained for the **SPECChpc 2021** benchmarks with **test** inputs from the tiny set. The benchmarks are sampled on an Intel PVC machine are cross-validated on an Intel DG2 machine. The benchmarks were also sampled and validated on an NVIDIA A100 machine.

mode instrumentation at the basic block level. Synchronization between multiple threads is necessary in this case. The average slow-down for the XPU-Profiler for the SPECaccel test cases was $102\times$.

Figure 10 plots the sampling errors for SPECaccel 2023 benchmarks using XPU-Point and GPU-Point evaluations. Overall, the sampling errors with GPU-only approach (geometric mean of 23.9%) are higher than those with the combined CPU-GPU approach (geometric mean of 0.99%). In the case of 452.ep, focusing on just the GPU computation in isolation predicts the overall performance with a low error (1.34%) although still higher than the combined CPU-GPU approach (0.10%). 404.ibm demonstrates another extreme, where the GPU-only approach only found one representative region leading to 210% sampling error. Using heterogeneous profile, 15 regions were identified by XPU-Point leading to a much lower sampling error (0.56%).

B. Sample Validation using Native Hardware

While simulation provides a controlled environment for workload analysis, validating samples on native hardware is often practical for large workloads. To enable this, we employ XPU-Timer to gather precise performance metrics from native hardware executions, as mentioned in Section IV-E. The results of sample validation using XPU-Timer, categorized by benchmark suite, are presented here.

1) **SPECChpc 2021:** The SPECChpc 2021 benchmark suites [78] provide application benchmarks from selected science and engineering applications that are portable across CPUs and accelerators. The suites include Tiny, Small, Medium, and Large workloads, supporting multiple programming models and requiring varying amounts of memory and

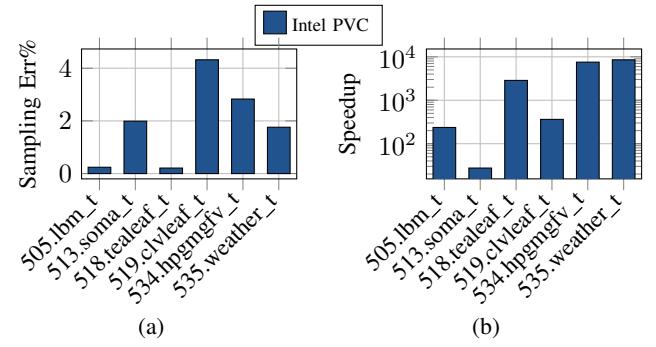


Fig. 12: The sampling errors (a) and speedup (b) obtained for the representative regions identified for **SPECChpc 2021** benchmarks that use **ref** inputs from the tiny set. The benchmarks are sampled and validated on an Intel PVC machine.

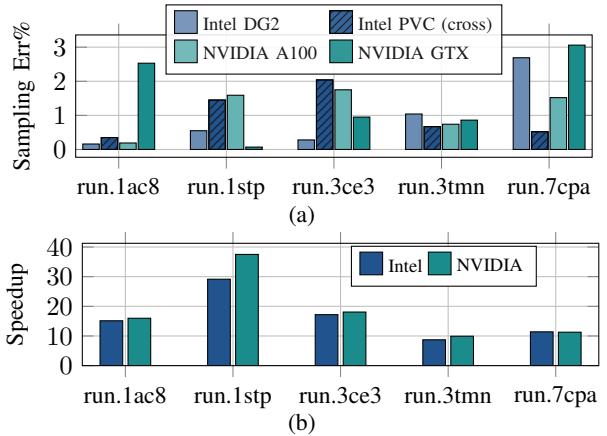


Fig. 13: Sampling errors (a) and speedup (b) for **AutoDock** (work-item=8) using different inputs on Intel and NVIDIA GPU platforms.

number of ranks to run. We chose the Tiny workload (60 GB memory requirement) and limited our testing to a single node/rank. We tested both the *test* and *ref* inputs for evaluation. The sampling errors and speedups for the benchmarks running test inputs are shown in Figure 11 for the Intel-based systems and NVIDIA-based systems. Due to the huge memory requirements, we evaluated the *ref* input set of SPECChpc benchmarks only on the Intel-based systems, and the results are shown in Figure 12.

2) **AutoDock-GPU:** AutoDock is a widely used software that performs molecular docking simulations commonly used for benchmarking and performance evaluation of heterogeneous systems. Figure 13 shows the sampling results obtained for AutoDock using XPU-Point on Intel-based and NVIDIA-based systems. We use four different platforms to validate the sample selected for the AutoDock [79], [81] application using various inputs. The SYCL implementation of AutoDock [80] is used for evaluating Intel GPU systems. The samples collected on the Intel DG2 platform are cross-validated on the Intel PVC platform. Due to binary incompatibility, cross-validation could

TABLE II: The classification of **GROMACS** based on the offloading device for the execution of each calculation. We also use `-nsteps 200` with `-notunepme` for all types. The last column shows the number of slices for each type.

Type	nb	pme	pmefft	bonded	update	#slices
A	GPU	CPU	CPU	CPU	CPU	305
B	GPU	CPU	CPU	GPU	CPU	506
C	GPU	CPU	CPU	CPU	CPU	707
D	GPU	CPU	CPU	GPU	CPU	908
E	GPU	CPU	CPU	CPU	CPU	3730
F	GPU	CPU	CPU	CPU	CPU	3931

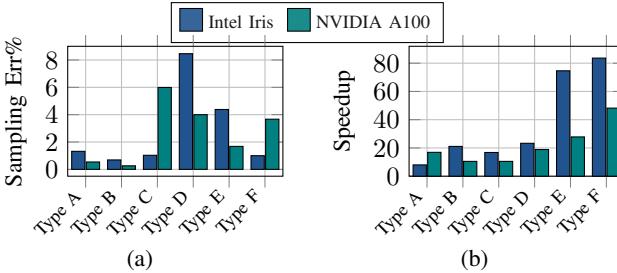


Fig. 14: The sampling errors (a) and speedup (b) for **GROMACS** in different settings on Intel Iris and NVIDIA A100 using XPU-Point.

not be performed on NVIDIA platforms.

3) *GROMACS*: GROMACS [82], [96] is a widely used open-source tool [97] for the simulation of molecular dynamics, which uses both CPU and GPU for computation. We manually configure the type of computation to be offloaded to the CPU and GPU, as shown in Table II, and identify the representative regions. The sampling errors and speedups are reported in Figure 14 for Intel-based systems and NVIDIA-based systems. We infer that the GROMACS workload Type F, with the most number of slices, benefits the most from the XPU-Point methodology achieving the maximum speedup. For Type A, the regions are expected to be larger due to the predominantly CPU-intensive nature of the workload.

C. Evaluation of PyTorch Inference Workloads

We evaluated PyTorch [98] inference workloads running text processing tasks using the BERT (Bidirectional Encoder Representations from Transformers) [99] model and image classification tasks using the ResNet50 (Residual Network with 50 layers) [100] model, across various configurations: data precision (BF16, FP16, FP32, and INT8 quantization) and execution mode (imperative Python vs. pre-compiled TorchScript). These workloads were optimized with the Intel Extension for PyTorch [83] to be evaluated on the machines with Intel PVC GPUs. We present the sampling results of PyTorch workloads using XPU-Point in Figure 15. Although profiling more libraries increases XPU-Profiler overhead (as observed with BERT_BF16_Ts, BERT_FP16_Ts, and BERT_FP32_Ts), the cost is amortized over multiple simulations. In general, the profiling and analysis of all shared libraries is necessary. To speed up the analysis, we chose to

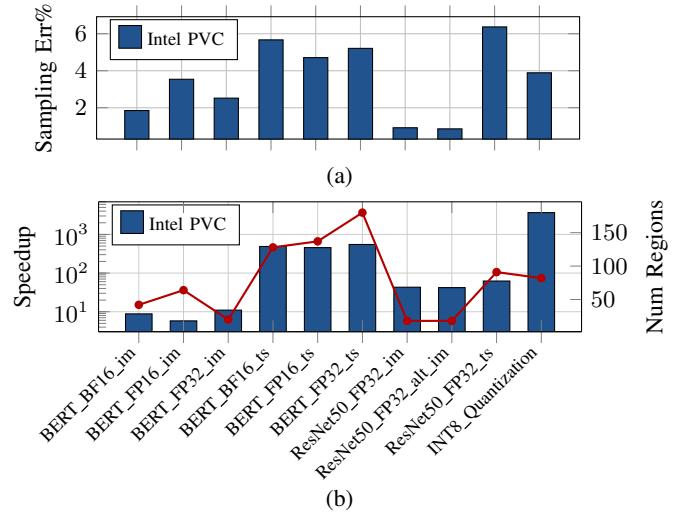


Fig. 15: The sampling errors (a) and expected simulation speedups (b) of **PyTorch Inference runs** plotted with shared x-axis. In (b), the line graph (plotted with the secondary y-axis) shows the number of representative regions selected using XPU-Point. im=Imperative, ts=TorchScript.

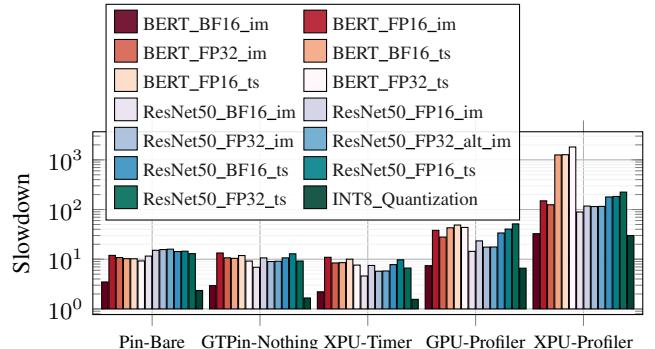


Fig. 16: The slowdowns (normalized with the native runtime of the application) for **PyTorch Inference runs** on Intel Ponte Vecchio GPU. The *Pin-Bare* mode measures the slowdown due to running the benchmarks under Pin with no instrumentation. The slowdown caused by the GTPin Tool is evaluated using a basic instrumentation tool, *Nothing*. *XPU-Timer* uses XPU-Pin to collect the timing information of the benchmarks. The *GPU-Profiler* profiles using GTPin to collect BBVs. *XPU-Profiler* uses XPU-Pin to collect BBVs of the CPU-GPU execution. im=Imperative, ts=TorchScript.

analyze the libraries that significantly impact workload runtime. The PyTorch workloads use a large number of libraries (>150), processes (≈ 60), and threads (>100), causing a large overhead in analyzing them. In this work, we focused on the main libraries and processes during instrumentation. Figure 16 shows the runtime overhead of evaluation tools used with these workloads.

VII. RELATED WORK

A. Sample Selection Methodologies

1) *CPU Workloads*: Prior research on the sample selection of single-threaded workloads explores profile-driven and statistical approaches. Profile-driven techniques, like SimPoint [16], target representative code sections for detailed simulation, aiming to capture the overall workload behavior. Conversely, statistical sampling methods, like SMARTS [17], randomly select instructions or program phases for simulation. LiveSim [50] enhances existing sampled simulation methodologies by incorporating in-memory checkpoints, thereby enabling interactive simulation capabilities. Traditional single-threaded sampled simulation techniques are not well-suited for multi-threaded applications [101]. While SimFlex [18] offered a solution to sample data center workloads, it is not suitable to evaluate synchronizing multi-threaded workloads. Time-based sampling techniques [19], [40] emerged as the first attempt to address this challenge. Subsequent advancements included profile-driven methodologies like BarrierPoint [20] for barrier-synchronized applications and TaskPoint [51] for task-based applications. LoopPoint [41] introduced loop-based sample selection and simulation techniques to accurately estimate the performance of generic multi-threaded applications, an approach that influenced later proposals [69], [102].

2) *GPU Workloads*: GTPin [31] is an ahead-of-time instrumentation tool for workloads that run on Intel GPUs. Leveraging GTPin, Kambadur et al. [22] proposed a solution to sample workloads running on Intel GPUs. They utilize kernel names, arguments, and basic block entries to select representative regions of the GPU programs at a kernel-level granularity. Yu et al. [52], [53] propose a SimPoint-like strategy to detect representative loops that can be used to extrapolate kernel performance. TBPoint [21] uses BBVs and other kernel-specific features to identify representative kernels, whereas Principal Kernel Analysis (PKA) [23] monitors the IPC difference between sampling units to determine the regions to fast-forward. Sieve [24] extends on prior works to show that using the kernel name and instruction count allows for better sample selection. Photon [42] utilizes GPU Basic Block Vectors (BBVs) for inter-kernel and intra-kernel workload sampling, resulting in significant improvement in sampling accuracy. STEM+ROOT [103] is a kernel-level sampling methodology that leverages the distribution of kernel execution times as the signature for representing the intervals.

B. Simulation Infrastructures

The most widely used CPU simulators are gem5 [33], Sniper [104], and ZSim [105]. GPU simulators are extremely slow as compared to the real execution [34], as they run on the CPU, which typically has fewer cores than the GPU under simulation. Trace-driven GPU simulators, such as MacSim [76], execute functionally generated traces with a timing model to generate the performance results. Execution-driven GPU simulators, such as Multi2Sim [75], gem5-gpu [43], MG-PUSim [106], gem5 APU [107], [108] and GPGPUSim [109],

directly execute the binary for performance simulation. Simulators like Accel-Sim [34] and NVArchSim [110] support both execution- and trace-driven simulation modes. Among these simulators, Multi2Sim, gem5-gpu, MacSim, and gem5 APU support heterogeneous CPU-GPU workloads.

C. Synthetic Benchmark Generation

Synthetic benchmark generation is a workload reduction technique that involves creating lightweight workload clones that mimic the performance characteristics of real-world applications. SynchroTrace [111] is a trace-based multi-threaded simulation methodology that accurately replays synchronization- and dependency-aware traces for chip multiprocessor systems. GPGPU-Minibench [53] captures the execution behavior of existing GPGPU workloads in a profile, which includes a divergence flow statistics graph to characterize the dynamic control flow behavior of a GPGPU kernel. G-MAP [112] statistically models the GPU memory access stream locality by considering the regularity in code-localized memory access patterns and the parallelism in the execution model to create miniaturized memory proxies. Mystique [113] is yet another technique that generates benchmarks from production AI models by leveraging PyTorch execution traces. Ditto [114] focuses on synthesizing workloads for data centers mimicking traditional CPU performance behaviors.

D. Heterogeneous Programming Models

Several programming models cater to heterogeneous computing, including OpenMPI [115], StarPU [116], OpenCL [117], OpenMP [118], OmpSs [119], CUDA [93], [120], and AMD HIP [121]. Prior works [122]–[125] compare the performance of CUDA and OpenCL programming models and show that the translation of one model to another works well for various applications. SYCL [126] is a modern heterogeneous programming model built on C++. There are several implementations of source-to-source translation tools from CUDA to SYCL [80], [127].

VIII. CONCLUSION

This paper proposes XPU-Point, a methodology for the sample selection of heterogeneous CPU-GPU workloads. XPU-Point leverages XPU-Pin, our instrumentation framework to combine CPU and GPU analysis. We demonstrate the accuracy and efficiency of the XPU-Point through the evaluation of real-world heterogeneous workloads, highlighting its ability to significantly reduce the simulation time. This work forms the basis for selecting representative regions applicable across a host of simulators, from cycle-accurate to high-level architectural models.

ACKNOWLEDGMENT

This work benefited immensely from the expertise of several individuals. In particular, we would like to acknowledge Roland Schulz, Edward Mascarenhas, Aleksandr Bobry, and the GTPin team at Intel. We also like to thank our colleagues at NUS for their support. This research was supported by a grant from Intel Corporation.

APPENDIX A ARTIFACT APPENDIX

A. Abstract

In this artifact, we provide the required tools and information needed to replicate the primary experiments demonstrated in this paper. The artifact provides the necessary tools and scripts to analyze heterogeneous applications through three main components:

- 1) **XPU-Profiler**: Profiles heterogeneous applications and collects basic block vectors for similarity analysis;
- 2) **XPU-Timer**: Evaluates the performance of selected regions on native hardware; and
- 3) **Performance Extrapolation**: Extrapolates performance results and generates visualization plots.

This appendix describes these parts and how to run them to replicate our experiments. We have also included an example benchmark, GROMACS, to demonstrate the end-to-end methodology.

B. Artifact check-list (meta-information)

- **Program**: C++ programs, Python/Shell scripts.
- **Compilation**: CUDA, oneAPI, Make, GCC.
- **Binary**: Pin 3.30, GTPin 4.5.0, NVBit 1.5.5.
- **Run-time environment**: NVIDIA and Intel GPU Drivers.
- **Hardware**: NVIDIA GPU systems and Intel GPU systems.
- **Metrics**: Cycles, TSC, Runtime.
- **Output**: XPU-Point sampling results of the benchmarks.
- **Experiments**: Use the scripts to run the experiments.
- **How much disk space required (approximately)?**: 500 GB.
- **How much time is needed to prepare workflow (approximately)?**: 1 day.
- **How much time is needed to complete experiments (approximately)?**: 1 week.
- **Publicly available?**: Yes
- **Code licenses (if publicly available)?**: Yes.
- **Archived (provide DOI)?**: 10.5281/zenodo.16801115

C. Description

- 1) **How to access**: GitHub [45] and Zenodo [128]
- 2) **Hardware dependencies**:
 - 1) Linux x86 system with NVIDIA GPU.
 - a) CUDA version: $\geq 8.0 \ \&\& \leq 11.x$
 - b) CUDA driver version: $\leq 495.xx$
 - 2) Linux x86 system with Intel GPU.
 - 3) At least 500 GB of free disk space.
- 3) **Software dependencies**: Docker, NVIDIA drivers, Intel drivers, CUDA, oneAPI.
- 4) **Data sets and Models**: None

D. Installation

We recommend the users to use the latest version of the XPU-Point tools released on GitHub [45].

- 1) Clone the XPU-Point repository and navigate to the base directory.

```
$ git clone https://github.com/nus-comparc/xpupoint && cd xpupoint
```
- 2) Setup the appropriate environment using docker. Note that this step requires an already installed GPU drivers.

- a) Build the docker image.

```
$ make docker.build
```
 - b) Run the docker image.

```
$ make docker.run
```
 - c) Compile XPU-Point tools and benchmarks.

```
$ make
```
 - d) All the necessary tools should be downloaded and built with this step.
- 3) The benchmark directory now contains the compiled benchmark(s) for evaluation on the respective platform using XPU-Point.

E. Experiment workflow

The installation scripts sets up all the directories and benchmarks for evaluation. The steps to run the end-to-end methodology are given below.

- 1) Navigate to the gromacs benchmark in the benchmarks directory.

```
$ cd benchmarks/gromacs
```
- 2) There are several test cases provided for gromacs. Use the run-xpupoint script to run all the tests or use the test directory name as argument to run individual tests.

```
$ ./run-xpupoint all
```

Or

```
$ ./run-xpupoint <test_dir>
```
- 3) The selected test cases run XPU-Profiler to identify the representative regions.
- 4) Further XPU-Timer determines the performance of the full application and each of the representative regions using native runs.
- 5) All of these are handled by run-xpupoint script. Once the tests are finished, the sampling results can be visualized as a table or a graph.

```
$ ./make-graphs
```

F. Evaluation and expected results

To replicate the results shown in this paper, it is necessary to run each of the benchmarks evaluated in this paper. We have set up GROMACS benchmark to run using our scripts to reproduce the results in Figure 14. Note that the profiling run for larger applications can take a few hours. To use this artifact, it is recommended to first verify that the Intel and/or NVIDIA GPU platforms are functioning correctly. Commands like nvidia-smi and sycl-ls are typically used to verify the functionality of NVIDIA and Intel GPUs respectively.

G. Notes

- 1) The results of the workload analyses are closely tied to the specific execution environment.
- 2) The results from XPU-Timer may be dependent on other jobs running on the same machine.

H. Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-and-badging-current>
- <https://ctuning.org/ae>

REFERENCES

- [1] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbahn, and P. Vilalobos, “Compute trends across three eras of machine learning,” in *International Joint Conference on Neural Networks (IJCNN)*, 2022, pp. 1–8.
- [2] H. Esmaeilzadeh, E. Blehm, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.
- [3] S. Borkar and A. A. Chien, “The future of microprocessors,” *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.
- [4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for gpus: stream computing on graphics hardware,” *Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 777–786, 2004.
- [5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.
- [6] W. S. Carter, I. Duong, R. R. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, N. T. Ngo, and S. L. Sac, “A user programmable reconfigurable logic array,” in *Custom Integrated Circuits Conference*, 1986, pp. 515–521.
- [7] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, “Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?” in *International Symposium on Microarchitecture (MICRO)*, 2010, pp. 225–236.
- [8] V. Garcia, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, “Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications,” in *International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.
- [9] M. D. Hill and V. J. Reddi, “Accelerator-level parallelism,” *Communications of the ACM*, vol. 64, no. 12, pp. 36–38, 2021.
- [10] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.
- [11] TOP500, “Top500 supercomputer sites,” <https://www.top500.org/>, 2022.
- [12] C. Chen, K. Li, A. Ouyang, Z. Tang, and K. Li, “Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data,” *Transactions on Systems, Man, and Cybernetics: Systems (T-SMC)*, vol. 47, no. 10, pp. 2740–2753, 2017.
- [13] H. Jiang, Y. Chen, Z. Qiao, T.-H. Weng, and K.-C. Li, “Scaling up mapreduce-based big data processing on multi-gpu systems,” *Cluster Computing*, vol. 18, pp. 369–383, 2015.
- [14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “TensorFlow: A system for large-scale machine learning,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.
- [15] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze *et al.*, “Tvm: An automated end-to-end optimizing compiler for deep learning,” *arXiv preprint arXiv:1802.04799*, 2018.
- [16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.
- [17] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, “SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling,” in *International Symposium on Computer Architecture (ISCA)*, Jun. 2003, pp. 84–97.
- [18] T. F. Wenisch, R. E. Wunderlich, M. Ferdinand, A. Ailamaki, B. Falsafi, and J. C. Hoe, “SimFlex: Statistical sampling of computer system simulation,” *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.
- [19] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sampled simulation of multi-threaded applications,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2013, pp. 2–12.
- [20] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, “BarrierPoint: Sampled simulation of multi-threaded applications,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 2–12.
- [21] J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee, “Tbpoint: Reducing simulation time for large-scale gpgpu kernels,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 437–446.
- [22] M. Kambadur, S. Hong, J. Cabral, H. Patil, C.-K. Luk, S. Sajid, and M. A. Kim, “Fast computational gpu design with gt-pin,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 76–86.
- [23] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, “Principal kernel analysis: A tractable methodology to simulate scaled gpu workloads,” in *International Symposium on Microarchitecture (MICRO)*, 2021, pp. 724–737.
- [24] M. Naderan-Tahan, H. SeyyedAghaei, and L. Eeckhout, “Sieve: Stratified gpu-compute workload sampling,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 224–234.
- [25] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 451–460.
- [26] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.
- [27] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, “Redefining the role of the cpu in the era of cpu-gpu integration,” *IEEE Micro*, vol. 32, no. 6, pp. 4–16, 2012.
- [28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.
- [29] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A transparent dynamic optimization system,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2000, p. 1–12.
- [30] D. Bruening, E. Duesterwald, and S. Amarasinghe, “Design and implementation of a dynamic optimization framework for windows,” in *Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, 2001, p. 20.
- [31] A. Skaletsky, K. Levit-Gurevich, M. Berezalsky, Y. Kuznetcova, and H. Yakov, “Flexible binary instrumentation framework to profile code running on intel gpus,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 109–120.
- [32] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, “Nvbit: A dynamic binary instrumentation framework for nvidia gpus,” in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.
- [33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
- [34] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-sim: An extensible simulation framework for validated gpu modeling,” in *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.
- [35] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.
- [37] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth *et al.*, “Gemini: a family of highly capable multimodal models,” *arXiv preprint arXiv:2312.11805*, 2023.
- [38] J. Choquette, “Nvidia hopper h100 gpu: Scaling performance,” *IEEE Micro*, 2023.
- [39] K. Yuan, C. Bauinger, X. Zhang, P. Baehr, M. Kirchhart, D. Dabert, A. Tousnakhoff, P. Boudier, and M. Paulitsch, “Fully-fused multi-layer perceptrons on intel data center gpus,” *arXiv preprint arXiv:2403.17607*, 2024.
- [40] E. K. Ardestani and J. Renau, “ESESC: A fast multicore simulator using time-based sampling,” in *International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2013, pp. 448–459.

- [41] A. Sabu, H. Patil, W. Heiman, and T. E. Carlson, “LoopPoint: Checkpoint-driven sampled simulation for multi-threaded applications,” in *International Symposium on High Performance Computer Architecture (HPCA)*, 2022, pp. 604–618.
- [42] C. Liu, Y. Sun, and T. E. Carlson, “Photon: A fine-grained sampled simulation methodology for gpu workloads,” in *International Symposium on Microarchitecture (MICRO)*, 2023, p. 1227–1241.
- [43] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, “gem5gpu: A heterogeneous cpu-gpu simulator,” *Computer Architecture Letters (CAL)*, vol. 14, no. 1, pp. 34–36, 2014.
- [44] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhitto, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, “The gem5 simulator: Version 20.0+,” *arXiv preprint arXiv:2007.03152*, 2020.
- [45] “XPU-Point source code,” <https://github.com/nus-comparch/xpupoint>, 2025.
- [46] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, “Analyzing parallel programs with pin,” *Computer*, vol. 43, no. 3, pp. 34–41, 2010.
- [47] Linux, “Linux programmer’s manual,” <https://man7.org/linux/man-pages/man8/ld.so.8.html>, 2024.
- [48] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, “Managing gpu concurrency in heterogeneous architectures,” in *International Symposium on Microarchitecture (MICRO)*, 2014, pp. 114–126.
- [49] J. Hestness, S. W. Keckler, and D. A. Wood, “Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 87–97.
- [50] S. Hassani, G. Southern, and J. Renau, “LiveSim: Going live with microarchitecture simulation,” in *International Symposium on High Performance Computer Architecture (HPCA)*, Mar 2016, pp. 606–617.
- [51] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé, “TaskPoint: Sampled simulation of task-based programs,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 296–306.
- [52] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. John, H. Jin, and C. Xu, “Accelerating gppu architecture simulation,” in *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013, pp. 331–332.
- [53] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C. Xu, and J. Wu, “Gppu-minibench: accelerating gppu micro-architecture simulation,” *Transactions on Computers (TC)*, vol. 64, no. 11, pp. 3153–3166, 2015.
- [54] J. W. Haskins and K. Skadron, “Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2003, pp. 195–203.
- [55] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic, “Accelerating multiprocessor simulation with a memory timestamp record,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2005, pp. 66–77.
- [56] M. Van Biesbrouck, B. Calder, and L. Eeckhout, “Efficient sampling startup for simpoint,” *IEEE Micro*, vol. 26, no. 4, pp. 32–42, 2006.
- [57] N. Nikoleris, A. Sandberg, E. Hagersten, and T. E. Carlson, “Coolsim: Statistical techniques to replace cache warming with efficient, virtualized profiling,” in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016, pp. 106–115.
- [58] “Nvidia gh200 grace hopper superchip architecture,” <https://resources.nvidia.com/en-us/grace-cpu/nvidia-grace-hopper/>, 2023.
- [59] D. D. Sharma, “Compute express link,” *CXL Consortium White Paper*, 2019.
- [60] D. Foley and J. Danskin, “Ultra-performance pascal gpu and nvlink interconnect,” *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.
- [61] DARPA, “Common heterogeneous integration and ip reuse strategies (chips),” <https://www.darpa.mil/program/common-heterogeneous-integration-and-ip-reuse-strategies>, 2024.
- [62] C. Wong and M. M. Wong, “Recent advances in plastic packaging of flip-chip and multichip modules (mcm) of microelectronics,” *Transactions on Components and Packaging Technologies (T-CPMT)*, vol. 22, no. 1, pp. 21–25, 1999.
- [63] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.
- [64] D. Blythe, “The xe gpu architecture,” in *Hot Chips Symposium (HCS)*, 2020, pp. 1–27.
- [65] M. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.
- [66] G.-Y. Lueh, K. Chen, G. Chen, J. Fuentes, W.-Y. Chen, F. Fu, H. Jiang, H. Li, and D. Rhee, “C-for-metal: High performance simd programming on intel gpus,” in *International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 289–300.
- [67] T. Sherwood, E. Perelman, and B. Calder, “Basic block distribution analysis to find periodic behavior and simulation points in applications,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001, pp. 3–14.
- [68] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder, “The strong correlation between code signatures and performance,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.
- [69] A. Sabu, C. Liu, and T. E. Carlson, “Viper: Utilizing hierarchical program structure to accelerate multi-core simulation,” *IEEE Access*, vol. 12, pp. 17 669–17 678, 2024.
- [70] K. L. Clarkson, “An algorithm for approximate closest-point queries,” in *Annual Symposium on Computational Geometry (SoCG)*, 1994, pp. 160–164.
- [71] A. Globerson and N. Tishby, “Sufficient dimensionality reduction,” *Journal of Machine Learning Research (JMLR)*, vol. 3, no. Mar, pp. 1307–1331, 2003.
- [72] E. W. Forgy, “Cluster analysis of multivariate data: efficiency versus interpretability of classifications,” *Biometrics*, vol. 21, pp. 768–769, 1965.
- [73] Y. Fang, Z. Liu, Y. Lu, J. Liu, J. Li, Y. Jin, J. Chen, Y. Chen, H. Zheng, and Y. Xie, “Nps: A framework for accurate program sampling using graph neural network,” *arXiv preprint arXiv:2304.08880*, 2023.
- [74] J. Xie, R. Girshick, and A. Farhadi, “Unsupervised deep embedding for clustering analysis,” in *International Conference on Machine Learning (ICML)*, 2016, pp. 478–487.
- [75] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2sim: A simulation framework for cpu-gpu computing,” in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.
- [76] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, “Performance characterisation and simulation of intel’s integrated gpu architecture,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 139–148.
- [77] Standard Performance Evaluation Corporation (SPEC), “Specaccel® 2023 benchmark,” <https://www.spec.org/accel2023/>, 2023.
- [78] J. Li, A. Bobry, S. Boehm, W. Brantley, H. Brunst, A. Cavelan, S. Chandrasekaran, J. Cheng, F. M. Ciorba, M. Colgrave, T. Curtis, C. Daley, M. Ferrato, M. G. de Souza, N. Hagerty, R. Henschel, G. Juckeland, J. Kelling, K. Li, R. Lieberman, K. McMahon, E. Melnichenko, M. A. Neggaz, H. Ono, C. Ponder, D. Radatz, S. Schueler, R. Searles, F. Vasilev, V. M. Vergara, B. Wang, B. Wesarg, S. Wienke, and M. Zavala, “SPECchpc 2021 benchmark suites for modern HPC systems,” in *International Conference on Performance Engineering (ICPE)*, 2022, p. 15–16.
- [79] G. M. Morris, D. S. Goodsell, R. Huey, W. E. Hart, S. Halliday, R. Belew, and A. J. Olson, “Autodock,” *Automated docking of flexible ligands to receptor-User Guide*, 2001.
- [80] L. Solis-Vasquez, E. Mascarenhas, and A. Koch, “Experiences migrating CUDA to SYCL: A molecular docking case study,” in *International Workshop on OpenCL (IWOCCL)*, 2023.

- [81] D. Santos-Martins, L. Solis-Vasquez, A. F. Tillack, M. F. Sanner, A. Koch, and S. Forli, "Accelerating autodock4 with gpus and gradient-based local search," *Journal of chemical theory and computation*, vol. 17, no. 2, pp. 1060–1073, 2021.
- [82] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.
- [83] "Intel extension for PyTorch sources," <https://github.com/intel/intel-extension-for-pytorch/>.
- [84] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger *et al.*, "Intel alder lake cpu architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.
- [85] I. E. Papazian, "New 3rd gen intel® xeon® scalable processor (code-name: Ice lake-sp)," in *Hot Chips Symposium (HCS)*, 2020, pp. 1–22.
- [86] H. Jiang, "Intel's ponte vecchio gpu: Architecture, systems & software," in *Hot Chips Symposium (HCS)*, 2022, pp. 1–29.
- [87] N. Nassif, A. O. Munch, C. L. Molnar, G. Pasdast, S. V. Lyer, Z. Yang, O. Mendoza, M. Huddart, S. Venkataraman, S. Kandula *et al.*, "Sapphire rapids: The next-generation intel xeon scalable processor," in *International Solid-State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 44–46.
- [88] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman *et al.*, "Cascade lake: Next generation intel xeon scalable processor," *IEEE Micro*, vol. 39, no. 2, pp. 29–36, 2019.
- [89] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.
- [90] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.
- [91] *oneAPI Specification*, <https://spec.oneapi.io/versions/1.1-rev-1/oneAPI-spec.pdf>.
- [92] Intel, "Intel oneapi," <https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html>, 2023.
- [93] NVIDIA, *NVIDIA CUDA Toolkit Documentation*, <https://docs.nvidia.com/cuda/>, 2024.
- [94] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu *et al.*, "SPEC ACCEL: A standard application suite for measuring hardware accelerator performance," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2014, pp. 46–67.
- [95] "Intel Software Development Emulator (Intel SDE)," <https://www.intel.com/software/sde>.
- [96] H. J. Berendsen, D. van der Spoel, and R. van Drunen, "Gromacs: A message-passing parallel molecular dynamics implementation," *Computer physics communications*, vol. 91, no. 1-3, pp. 43–56, 1995.
- [97] "The GROMACS molecular simulation toolkit." <https://github.com/gromacs/gromacs>.
- [98] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.
- [99] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2019, pp. 4171–4186.
- [100] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [101] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.
- [102] C. Liu, A. Sabu, A. Chaudhari, Q. Kang, and T. E. Carlson, "Pac-sim: Simulation of multi-threaded workloads using intelligent, live sampling," *Transactions on Architecture and Code Optimization (TACO)*, vol. 21, no. 4, Nov. 2024.
- [103] E. Chung, S. Na, S. H. Kang, and H. Kim, "Swift and trustworthy large-scale gpu simulation with fine-grained error modeling and hierarchical clustering," in *International Symposium on Microarchitecture (MICRO)*, 2025, pp. 1397–1411.
- [104] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 52:1–52:12.
- [105] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *International Symposium on Computer Architecture (ISCA)*, Jun. 2013, pp. 475–486.
- [106] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao *et al.*, "Mgpusim: enabling multi-gpu performance modeling and optimization," in *International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.
- [107] B. M. Beckmann and A. Gutierrez, "The amd gem5 apu simulator: Modeling heterogeneous systems in gem5," in *Tutorial at the International Symposium on Microarchitecture (MICRO)*, 2015.
- [108] A. Gutierrez, B. M. Beckmann, A. Dutta, J. Gross, M. LeBeane, J. Kalamaitianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor *et al.*, "Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.
- [109] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.
- [110] O. Villa, D. Lustig, Z. Yan, E. Bolotin, Y. Fu, N. Chatterjee, N. Jiang, and D. Nellans, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 868–880.
- [111] S. Nilakantan, K. Sangaiyah, A. More, G. Salvadory, B. Taskin, and M. Hempstead, "Synchrotrace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 278–287.
- [112] R. Panda, X. Zheng, J. Wang, A. Gerstlauer, and L. K. John, "Statistical pattern based modeling of gpu memory access streams," in *Design Automation Conference (DAC)*, 2017, pp. 1–6.
- [113] M. Liang, W. Fu, L. Feng, Z. Lin, P. Panakanti, S. Zheng, S. Sridharan, and C. Delimitrou, "Mystique: Enabling accurate and scalable generation of production ai benchmarks," in *International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–13.
- [114] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "Ditto: End-to-end application cloning for networked cloud services," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 222–236.
- [115] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open mpi: A high-performance, heterogeneous mpi," in *International Conference on Cluster Computing (CLUSTER)*, 2006, pp. 1–9.
- [116] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2009, pp. 863–874.
- [117] A. Munshi, "The OpenCL specification," in *Hot Chips Symposium (HCS)*, 2009, pp. 1–314.
- [118] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "Openmp for accelerators," in *OpenMP in the Petascale Era: International Workshop on OpenMP (IWOMP)*, 2011, pp. 108–121.
- [119] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.
- [120] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
- [121] A. ROCm, "Hip: C++ heterogeneous-compute interface for portability," <https://github.com/ROCM/HIP>, 2024.
- [122] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *International Conference on Parallel Processing (ICPP)*, 2011, pp. 216–225.
- [123] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

- [124] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee, “Bridging OpenCL and CUDA: a comparative analysis and translation,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015, pp. 1–12.
- [125] M. Daga, Z. S. Tschirhart, and C. Freitag, “Exploring parallel programming models for heterogeneous computing systems,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2015, pp. 98–107.
- [126] L. Howes and M. Rovatsou, *SYCL Specification – SYCL integrates OpenCL devices with modern C++*, <https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf>, 2015.
- [127] Z. Wang, Y. Plyakhin, C. Sun, Z. Zhang, Z. Jiang, A. Huang, and H. Wang, “A source-to-source CUDA to SYCL code migration tool: Intel® DPC++ compatibility tool,” in *International Workshop on OpenCL*, 2022, pp. 1–2.
- [128] A. Sabu, H. Patil, W. Heirman, C. Liu, and T. E. Carlson, “Xpu-point artifacts,” Aug. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.16801115>