ThreadFuser: A SIMT Analysis Framework for MIMD Programs

Ahmad Alawneh, Ni Kang, Mahmoud Khairy*, Timothy G. Rogers
Elmore Family School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{aalawneh, kang222, abdallm, timrogers}@purdue.edu

***Abstract*—The broad usage of accelerators, such as GPUs, faces two important challenges. Developing code for a new accelerator is expensive and unpredictable. Porting large parallel programs from Multiple Instruction Multiple Data (MIMD) CPUs to Single Instruction Multiple Thread (SIMT) GPUs involves significant effort that may or may not result in improved performance versus the CPU. This high activation energy to create new workloads introduces the second challenge: architects and systems researchers lack a diverse SIMT codebase to study new designs.**

To tackle these challenges, we introduce ThreadFuser, an analysis framework that efficiently and accurately predicts the performance of any pre-written MIMD program on SIMT hardware. ThreadFuser conducts thorough control and data flow analysis on dynamic CPU program traces, determining the impact of lock-step execution on CPU binaries. ThreadFuser efficiently delivers accurate reports on a MIMD program's divergence and synchronization characteristics. Moreover, ThreadFuser seamlessly integrates with state-of-the-art GPU simulators to conduct detailed analyses and produce fine-grained performance measurements.

We evaluate ThreadFuser on a diverse set of 36 CPU workloads, demonstrating the potential and challenges of executing MIMD code on a SIMT machine. We demonstrate ThreadFuser's potential to inform software development decisions and open new areas to explore in data-parallel hardware design.

***Index Terms*—SIMT, GPU, Programmability, Simulation, Performance.**

I. Introduction

Most parallel programs are written in a Multiple Instructions Multiple Data (MIMD) fashion, primarily targeting CPUs [5], [20], [21], [33], [39]. However, the slowing of Moore's Law and the energy-efficiency limitations of CPUs have prompted a paradigm shift in architectural approaches, ushering in the era of accelerators, with the GPU's Single Instruction Multiple Thread (SIMT) model gaining the most commercial traction. Beyond their conventional role in graphics processing, SIMT-based architectures are used pervasively in High-Performance Computing (HPC) and Machine Learning (ML). However, they have yet to gain significant traction in other parallel domains. Additional classes of software can benefit from execution on efficient SIMT hardware, creating a more diverse set of applications for architects to explore SIMT designs outside of traditional GPUs. From a software developer's perspective, porting code to a SIMT accelerator demands significant development time, and there is a risk of wasting effort if the code is unsuitable for the hardware. As a result, most GPU codes

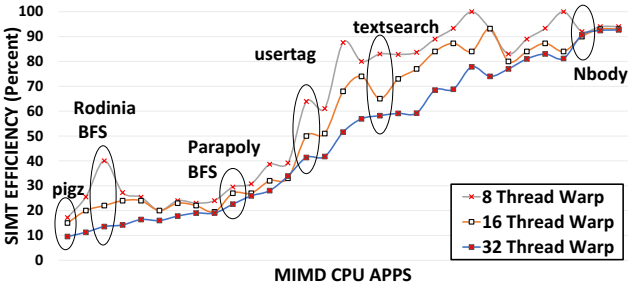


Fig. 1: Estimated SIMT Efficiency for 36 MIMD applications (Section V-A) using *ThreadFuser*.

are ported from CPU codes with easy-to-identify regularity in their control flow and access patterns. However, many parallel applications miss acceleration opportunities because no effort has been invested in moving them to a GPU. If developers had access to a zero-effort performance estimation framework, the risk of porting would be reduced, and more software could take advantage of SIMT hardware. Similarly, architects face limitations in analyzing and assessing the efficiency of new SIMT hardware due to the limited variety of available software for SIMT machines. Granting hardware designers the ability to examine the impact of SIMT hardware on any CPU binary can lead to more sophisticated SIMT accelerators that target workloads beyond graphics, HPC, and ML.

To address both the developer's lack of predictability and the architect's lack of software, we introduce *ThreadFuser*. *ThreadFuser* is an analysis and trace-generation framework designed to rapidly and precisely predict the performance of any MIMD CPU program on SIMT hardware. By gathering dynamic traces from unaltered parallel CPU binaries, ThreadFuser quickly analyses control flow and memory access patterns. This process results in a comprehensive, per-function breakdown of a parallel application's SIMT efficiency, memory divergence level, and synchronization characteristics if it were to be executed on SIMT hardware. This initial estimate is cheap to produce (generated with no programmer effort in only 2-6× native CPU execution time) and can be applied to any CPU binary, even closed source. For a more comprehensive analysis, ThreadFuser also generates a trace file that integrates

*Work conducted at Purdue University, currently at AMD Corporation

with trace-based SIMT hardware simulators [27], [45]. Using the open-source Accel-Sim [27] framework, we demonstrate *ThreadFuser*'s capability to generate detailed performance analysis that accounts for all the cycle-level factors that impact performance, allowing architects to evaluate SIMT designs that target diverse workloads.

To demonstrate *ThreadFuser*'s effectiveness, Figure 1 plots the estimated SIMT efficiency of 36 MIMD CPU workloads (described in Section V-A) if they were run on SIMT hardware with warp sizes of 8, 16, and 32. Some of these workloads have complimentary GPU implementations (which we use for correlation in Section IV); however, most do not. This information is instructive to developers, who can quickly see if the workload will perform poorly on a GPU if no additional optimizations are made. For instance, Pigz [1], a Linux binary that implements GZip, exhibits notably low efficiency compared to N-body simulation. A quick as-is port of Pigz is unlikely to perform well on the GPU, where the N-body code, as-written, will map very easily to SIMT hardware. Note that SIMT efficiency is a necessary but insufficient condition for performance improvement on a GPU. If the application's control flow, memory divergence, and synchronization patterns look promising, the software designer can use *ThreadFuser*'s simulator integration to analyze speedup predictions and bottlenecks deeply. However, for software developers, the quickly generated high-level information allows them to identify which applications or services might be good GPU candidates with little SIMT-specific code optimizations.

ThreadFuser leverages Intel's x86 PIN tool [30] to generate dynamic instruction traces for each CPU thread, forwarding them to a backend analyzer that employs a configurable batching algorithm to group threads into warps. Subsequently, ThreadFuser conducts a stack-based Immediate Post-Dominator (IPDOM) reconvergence analysis for the grouped threads [15]. This analysis calculates the SIMT control-flow efficiency and the application's anticipated memory divergence (average accesses per memory instruction). ThreadFuser also monitors the invocation of synchronization primitives, ensuring that serialization is enforced when multiple threads fused into the same warp access the same lock. To validate the accuracy of ThreadFuser, we correlate efficiency and performance metrics against real GPU hardware in 11 parallel workloads with existing CPU and CUDA implementations.

Utilizing ThreadFuser, we conduct case studies to showcase its potential application and shed light on untapped opportunities within contemporary parallel CPU workloads. For instance, we explore the viability of porting multi-threaded Linux utilities to SIMT hardware. Many of these workloads, with unaltered code, demonstrate promisingly convergent control flow, achieving up to 99% SIMT efficiency and projected speedups of 15-20x. We also explore the effect SIMT execution has on data center microservices [16], [41]. Modern data centers exhibit extensive degrees of similar request-level parallelism, receiving numerous independent requests from millions of users running the same service code. ThreadFuser's analysis reveals that some microservices exhibit high control efficiency, averaging 78%, indicating potential efficiency gains from executing them in a SIMT fashion. Using these workloads, we also examine their synchronization patterns and demonstrate the impact fine-grained locking can have on SIMT workloads.

While ThreadFuser helps analyze closed-source applications, it is also designed to help developers perform pre-port source code analysis. ThreadFuser generates a function-specific report on efficiency metrics that helps identify likely bottlenecks in GPU execution. Using this information, developers can make more informed decisions about which functions are easy-wins and which need more refactoring to make them amenable to GPU acceleration. We perform a case study using production microservice to identify code—buried deep within a library call—that single-handedly destroys the application's SIMT efficiency. Once identified, we demonstrate how a SIMT-aware modification to the code improves SIMT efficiency from 6% to 90%.

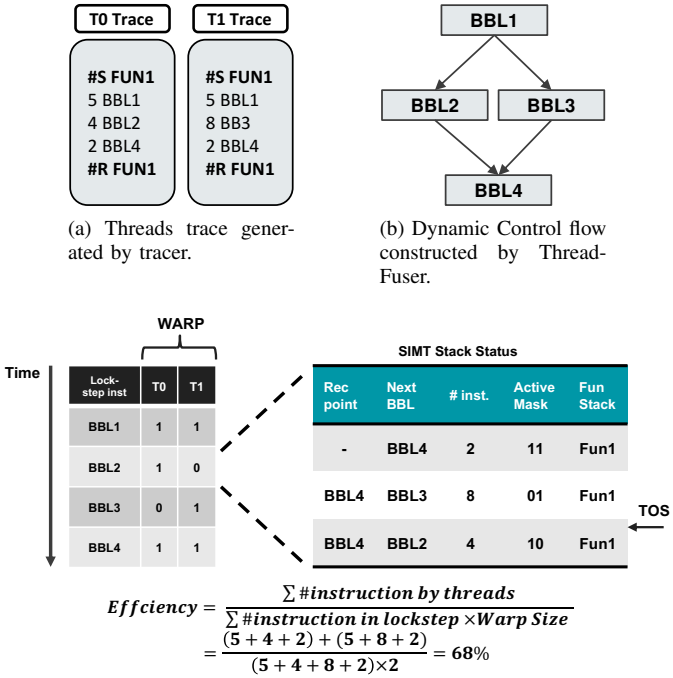
ThreadFuser is not limited to studying how developers might better use GPUs. Integrating with a cycle-level simulator, ThreadFuser can be used to design efficient SIMT machines with thread counts between a multicore CPU and a GPU. Prior work has demonstrated that SIMT architectures with a thread count in the hundreds to low thousands can be an energy-efficient way to execute general-purpose instruction sets [14], [26], [44]. The ThreadFuser infrastructure makes it possible to study any MIMD workload with any thread count using the Accel-Sim SIMT simulator.

This paper makes the following contributions:

- We introduce *ThreadFuser*, an innovative analysis framework designed to rapidly and accurately estimate the performance characteristics of parallel CPU workloads when executed on SIMT hardware. ThreadFuser is validated using GPU hardware across 11 OpenMP workloads with CUDA implementations (Section IV).
- We demonstrate ThreadFuser's ability to aid software developers in creating zero-cost estimates of GPU performance, showcasing how ThreadFuser can pinpoint code regions that are challenging for GPUs. Our analysis demonstrates that many CPU workloads have untapped potential on contemporary GPUs (Section V-A).
- We highlight ThreadFuser's ability to provide architects with more diverse SIMT software and drive innovation in data-parallel accelerators that target workloads outside of graphics, ML, and HPC. We integrate ThreadFuser with a state-of-the-art GPU simulator, demonstrating that accelerator innovations can be driven by evaluating MIMD software (Section V-B).

II. Background

Switching from conventional CPUs to SIMT architecture, a blend of SIMD and multithreading, responds to the demand for parallelism and energy efficiency. Through warps and control-flow divergence management, SIMT optimizes efficiency. This section details the SIMT execution model



(a) Threads trace generated by tracer.

(b) Dynamic Control flow constructed by ThreadFuser.

(c) SIMT Stack operations.

Fig. 2: Example of SIMT Stack operations performed by ThreadFuser.

and the effect of batching MIMD threads together for SIMT execution.

SIMT Execution Model: Modern SIMT architectures comprise multiple processing cores, each hosting numerous parallel lanes (SIMD units), a vector register file, and a shared memory segment. The primary unit of execution is referred to as a warp (or wavefront), representing a collection of threads executed in lockstep on a SIMD unit. Shared memory is distributed among the warps concurrently executing on a core.

SIMT execution on GPUs involves running multiple threads within a warp simultaneously, each potentially taking a different path in the program's control flow graph, as exemplified in Figure 2. When threads in the same warp diverge, as shown after BBL1 in Figure 2b, current SIMT architectures execute all control paths, such as BBL2 and BBL3, sequentially. This serial execution of divergent paths can reduce parallelism and SIMD unit utilization. To counter this, modern GPUs ensure the reconvergence of threads that follow different control paths.

Control-flow divergence in SIMT hardware is managed through a SIMT stack and immediate post-dominator (IPDOM)-based convergence mechanism, as depicted in Figure 2. In this context, all threads reaching a specific diverged branch reconverge at the immediate post-dominator basic block of that branch. The post-dominator (IPDOM) basic block, is the first basic block in the static control flow and is guaranteed to be on both diverged paths (as indicated in BBL4 in Figure 2b), plays a crucial role in orchestrating reconvergence and maintaining high SIMD unit utilization for optimal parallel execution. This approach is essential for maximizing GPU performance and utilization in parallel computing tasks [15], [31], [32].

Currently, the method for implementing IPDOM reconvergence in SIMT architectures is to treat control flow execution as a serial stack, as shown in Figure 2c . Initially, a new entry is pushed onto the SIMT stack with the active mask of all threads set to all 1s. The top of the stack (TOS) keeps track of the next basic block to execute. When encountering a divergence in threads' traces, such as BBL2 and BBL3, the SIMT hardware adds new entries to the SIMT stack and uses IPDOM analysis to identify the immediate post-dominator (reconvergence point), exemplified by BBL4 in Figure 2c. Each time control diverges, both the taken and not taken paths are pushed onto the stack (in arbitrary order), and the path at the new top of the stack is executed. As the SIMT hardware processes divergent entries (BBL2, BBL3), it executes them with distinct lock-step instructions and active masks (e.g., 10 and 01), resulting in limited parallelism. However, this approach allows for efficient handling of divergent paths within a warp. Upon reaching the reconvergence point (BBL4), the hardware removes the entry of the respective thread from the SIMT stack, enabling the reconvergence of threads. This mechanism allows threads to execute the instruction in lockstep with active lanes, effectively managing control-flow divergence in SIMT architectures.

From CPU to SIMT: The escalating demand for parallelism in computational tasks and the critical need for energy-efficient solutions have propelled the shift from conventional CPU architectures to SIMT (Single Instruction, Multiple Threads) architectures. SIMT combines the principles of SIMD (Single Instruction, Multiple Data) with multithreading, allowing a single instruction to be executed on multiple threads simultaneously while enabling divergence in data paths. Unlike SIMD, SIMT's implicit vectorization simplifies programming, eliminating the need for explicit predicates when lanes diverge. Its execution model, based on Single Program Multiple Data (SPMD) on SIMD hardware, allows each thread to be treated separately by the programmer while hardware groups threads together into warps (usually 64 or 32 threads wide) to aggregate control logic and exploit spatial data reuse among parallel threads. The inherent efficiency in SIMT hardware comes from (i) amortizing the pipeline front-end overhead by fetching, decoding, and scheduling each instruction only once for all the threads in the same warp and (ii) generating less traffic to the memory system by coalescing accesses from threads in the same warp. GPUs have traditionally been recognized as the predominant platform for SIMT hardware. However, recent studies by Kalathingal et al. [25], and Tino et al. [44] challenge this notion, demonstrating that SIMT efficiency can extend to Simultaneous Multi-Threading (SMT) CPU hardware as well. This is achieved by dynamically grouping multiple scalar SMT threads to execute in lockstep when they share a common

instruction stream.

Programming abstractions for SIMT, such as CUDA [36] or HIP [6], create the illusion of data parallelism with independent threads. However, in actual execution, a group of program instances (threads) is mapped to a warp and executed synchronously. Consequently, control-flow divergence in SPMD programs can significantly impact performance due to the inherent limitations of SIMT execution.

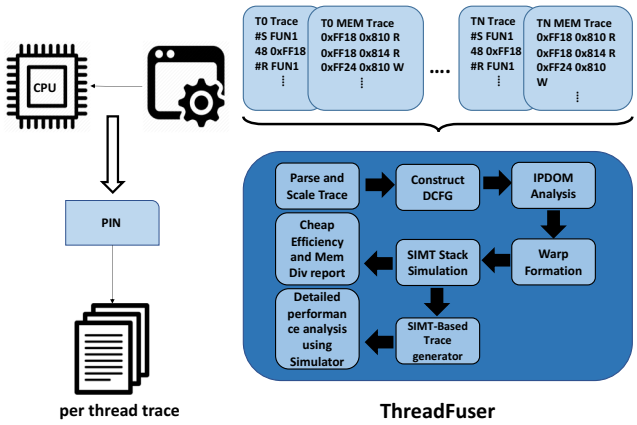
In SIMT hardware, memory coalescing is illustrated in figure 4, showcasing its optimization impact. This process consolidates multiple memory requests from threads within a warp into a single, efficient transaction. This streamlined approach minimizes the number of 32-byte transactions needed for load/store instructions, enhancing overall memory access performance. The coalescing process showcased in figure 4 mirrors algorithms commonly utilized in GPU hardware [35], [36], emphasizing its alignment with established optimization principles and its contribution to improved efficiency and throughput in SIMT architectures.

Whether utilizing GPUs or the previously suggested SIMT-based CPU hardware, a fundamental inquiry arises: How much can a specific workload capitalize on SIMT-based hardware? In simpler terms, what is the extent of SIMT efficiency in parallel CPU workloads?

III. System Overview

Figure 3 visually represents the design of the ThreadFuser system, providing insight into its structure and functionalities. The overarching goal of ThreadFuser is to establish a framework capable of accurately estimating and predicting the performance of diverse parallel MIMD programs without requiring to port them to SIMT hardware architecture platforms. At its core, ThreadFuser has two primary components: 1) The ThreadFuser tracer component (depicted in Figure 3a) responsible for capturing detailed traces of the MIMD application's instructions and memory accesses. This critical functionality forms the foundation for precise performance estimation and analysis within the ThreadFuser framework. (2) The ThreadFuser analyzer (depicted in Figure 3b) that analyzes the generated traces to project the SIMT efficiency and the memory divergence.
Additionally, ThreadFuser includes a separate component that can be utilized to generate warp-based instruction traces. These traces can be fed into a state-of-the-art traced-based GPU simulator like Accel-Sim [27], providing a detailed performance analysis.

ThreadFuser Tracer: The ThreadFuser Tracer leverages a tracing tool built on Intel's PIN platform [30] to systematically generate traces for each thread initiated by the CPU program. This tool inserts callbacks before the execution of each basic block (BBL) to capture essential information such as addresses and the number of instructions within executed basic blocks. Moreover, the traces include detailed per-instruction (x86 CISC instruction) memory access information. Additionally, the tracing tool instruments the recording of function calls,



(a) Step 1: Per-thread PIN trace generation

(b) Step 2: Control flow and memory divergence analysis

Fig. 3: *ThreadFuser* architecture.

capturing both call and return points and collecting the called function names. This enables the framework to construct a detailed function-call stack, providing valuable insights into the program's execution flow and aiding in comprehensive analysis. Furthermore, the tool captures the addresses of accessed synchronization locks, which are utilized by the ThreadFuser Analyzer component to study lock contention and synchronization overhead in multi-threaded programs.
The tool is configurable, allowing programmers to selectively choose specific functions for tracing or exclusion. This capability enables a more focused analysis, allowing for a detailed study of particular regions within the application. This flexibility showcases the tool's adaptability to various analyses and requirements. In essence, with its low overhead, typically ranging between only 2 to 6 X the native CPU execution time, the tracing tool serves as the foundational component of ThreadFuser, actively facilitating the generation of warp-based SIMT traces and enabling detailed analyses of SIMT efficiency across diverse MIMD workloads running on SIMT hardware.

ThreadFuser Analyzer: The initial goal of the analyzer is to provide a quick, cost-effective first-order estimate and prediction, covering SIMT efficiency and memory divergence. This initial estimation prepares for a more detailed performance study, enabled by the subsequent generation of SIMT-based traces. The ThreadFuser Analyzer pipeline, illustrated in Figure 3b, includes several key stages. Initially, the analyzer parses the basic block and memory traces, generated by the ThreadFuser Tracer.

The subsequent phases focus on preparing and analyzing the parsed traces to create intermediate data structures and metadata utilized in performance predictions. As mentioned in section II, SIMT hardware relies on IPDOM and SIMT divergence stacks to manage threads' divergence. To find the IPDOMs of basic blocks, the tool must construct the program's Control Flow Graph. The analyzer leverages the collected

traces of dynamically executed basic blocks to build the Dynamic Control Flow Graph (DCFG). This graph represents the program's evolving control flow during execution by identifying successors and predecessors for each Basic Block (BBL) to form the DCFG.To simplify matters, the DCFG is generated independently for each thread and subsequently merged into a unified graph. In case of threads divergance, the analyzer ensures to reconverge the threads at the reconvergence point(IMPDOM).

ThreadFuser implements the Immediate Post-Dominator (IPDOM) algorithm, similar to GPGPU-sim [2], [9]. This algorithm is pivotal in identifying the first basic block that is guaranteed to execute after all paths in the program converge. It iteratively refines the IPDOM information for each basic block (BB) in the program's control flow graph

Following the IPDOM analysis, ThreadFuser organizes the threads into warps, emulating the warp structure in SIMT hardware. The analyzer allows user configuration of the warp width, facilitating the exploration of different SIMT hardware different SIMT hardware widths, as we observed how that influences the applications in Section I. Moreover, different batching algorithm can be explored in the process of warp formation.

After these stages are completed, ThreadFuser proceeds to emulate warp execution in a lock-step manner.If the analyzer detects thread divergence based on the traces, ThreadFuser incorporates SIMT stack operations. It utilizes data obtained from the IPDOM analysis to ensure the convergence of threads at the reconvergence points, emulating the architecture of contemporary SIMT hardware (e.g., GPUs) as discussed in Section II.

In Figure 2a, the operation of ThreadFuser is illustrated, demonstrating how the tracer collects traces from two CPU threads executing the same function. These traces are then processed to construct the Dynamic Control Flow Graph (DCFG), showcased in Figure 2b, based on the observed basic block trace. Subsequently, ThreadFuser forms these threads into a warp, assuming a warp size of 2 in this example. The utilization of the SIMT stack algorithm, as detailed in Section II, guides this process. A visual representation of the SIMT stack operation in ThreadFuser is provided in Figure 2.

ThreadFuser captures a crucial performance metric for SIMT hardware—SIMT efficiency. This metric involves counting the instructions executed in lock-step mode and those executed by each thread within the warp. SIMT efficiency is key in assessing how effectively the program utilizes the SIMD processing capabilities offered by SIMT hardware. The overall SIMT efficiency for the program is then computed by averaging these efficiencies across all warps, with the specific formula outlined in Equation 1. For instance, a program with a 50% SIMT efficiency indicates that half of the computational resources in a SIMD hardware unit are effectively utilized during processing.

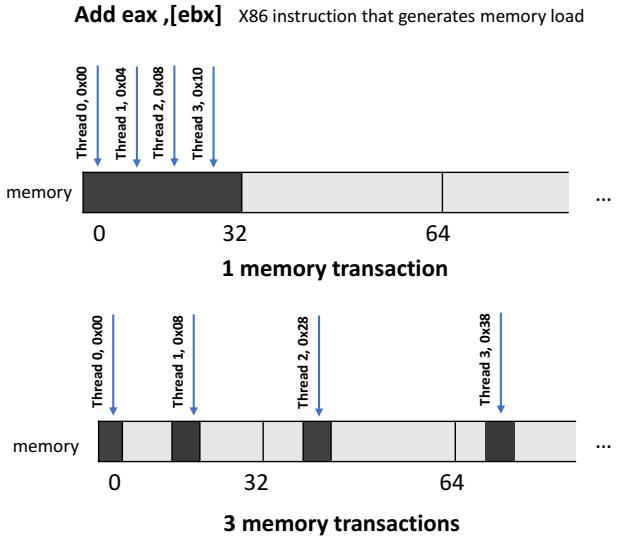


Fig. 4: Memory coalescing technique used in ThreadFuser.

$$\text{SIMT efficiency} = \frac{\#\text{Instructions by all threads}}{\#\text{Instructions in lock-step} \times \text{WarpSize}} \quad (1)$$

Selecting the optimal granularity for constructing the Dynamic Control Flow Graph (DCFG) presents a challenge. Building the DCFG based on the entire trace might lead to a situation where a single function is called from various parts of the application. Consequently, the function return instruction could point to multiple basic blocks in the DCFG, making the Immediate Post Dominator (IPDOM) algorithm more conservative and selecting distant reconvergence points.

To address this challenge, ThreadFuser employs a solution by introducing a virtual basic block at the end of each function and creating a per-function DCFG. This strategy compels divergent threads to converge at the conclusion of each function, mirroring the behavior seen in contemporary SIMT hardware like GPUs. This refined approach results in a more accurate DCFG.

Additionally, ThreadFuser maintains a function-call stack in its SIMT stack to accurately track the currently emulated function. Leveraging the per-function DCFG, ThreadFuser generates a per-function efficiency report that specifically excludes the influence of nested function calls. This feature allows programmers to identify bottleneck functions that could hinder SIMT performance, as exemplified in Section V-A. Through the analysis of per-function efficiency, programmers can optimize application performance by precisely identifying and enhancing less efficient functions. This proves particularly valuable in complex applications with numerous functions, providing a detailed assessment of performance at the function level.

ThreadFuser, in addition to evaluating SIMT efficiency, offers valuable insights into the memory performance of the application. The tool generates a comprehensive report on the total number of memory transactions per x86 instruction

Correlation Workloads	Workload	#SIMT Threads	Workloads with no GPU Implementation	Workload	#SIMT Threads	Workloads with no GPU Implementation	Workload	#SIMT Threads
	Rodinia 3.1 [12]			**μsuite [41]**			**ParSec 3.0 [10]**	
	BFS	4K		McRouter (Memcached ,Mid ,Leaf)	2K		blackscholes	1K
	Nearest Neighbors(NN)	42K		TextSearch(Mid, Leaf)	2K		streamcluster	8K
	Stream Cluster(SC)	16K		HDImageSearch(Mid, Leaf)	2K		bodytrack	1K
	b+tree	4K		**DeathStarBench**			facesim	1K
	Particle Filter(PF)	4K		Post	2K		fluidanimate	4K
	Parapoly [48]			Text	2K		freqmine	2K
	BFS	4K		URLShort	2K		swaptions	512
	Connected Components(CC)	4K		UniqueID	2K		vips	512
	Page Rank	4K		UserTag	2K		x264	4K
	Nbody	4K		User	2K		**Others**	
	Micro Benchmark			**Others**			Pigz [1]	128
	VectorAdd	1K		Rotate [7]	1K			
	Uncoalesced Vector operation	1K		MD5 [7]	512			

TABLE I: Studied Workloads. #SIMT threads is the number of threads simulated by ThreadFuser

that initiates memory accesses. This involves checking the accessed addresses by each thread within the warp, subsequently coalescing them to determine the number of 32-byte transactions necessary for the load or store instruction. Figure 4 provides an illustrative example of how ThreadFuser adeptly coalesces memory accesses. For instance, when an ADD instruction generates a load memory access, and all threads in the warp touch memory addresses that are 4 bytes apart, ThreadFuser coalesces these accesses into a single memory transaction (i.e., coalesced accesses). Conversely, in scenarios of divergent memory access, multiple transactions might be required to load the data necessary for the ADD instruction. Moreover, ThreadFuser breaks down the transactions based on the accessed segment, distinguishing between stack and heap transactions. As the tool coalesces transactions into 32-byte cache line transactions, the ideal number of memory transactions per coalesced memory instruction for a 32-thread warp is ideally 4x 32B transactions for 4-byte access or 8x 32B transactions for 8-byte access. This insightful breakdown aids in identifying memory access patterns and unveils optimization opportunities for enhancing application performance.

Synchronization handling in ThreadFuser: Dealing with synchronization and locking presents a significant obstacle when attempting to port MIMD applications to SIMT hardware. Locks contention can affect the control flow efficiency of the running application. As a result, to ensure accurate prediction of the performance of MIMD applications ThreadFuser monitors all the locks acquiring/releasing operations performed by each thread. The ThreadFuser Tracer records the calls to all synchronization primitives along with the addresses of the accessed locks. In the analysis phase, we use the traced information to quantify the effects of the synchronization. To identify the critical section being executed by the threads, we match the lock and unlock operations based on the lock address they refer to. As we emulate the execution of the warps, if a synchronization primitive is called, we check the addresses of locks accessed by all threads and allow threads acquiring different locks to execute in parallel. For threads that compete on the same lock, the analyzer will simulate their execution in a serial fashion. Therefore, ThreadFuser pushes multiple entries to the SIMT stack, reflecting the serial execution of the threads. The identification and location of the reconvergence point of serialized threads can impact the efficiency of control flow. We select one of the unlock pairs of one of the threads as the anticipated reconvergence point. We acknowledge that different choices of reconvergence points may have varying effects on the control flow efficiency, but we defer this investigation to future research. In section V-B, we explore the impact of the critical section on control flow efficiency.

Generating warp-based instruction traces using ThreadFuser: To provide a comprehensive performance evaluation of parallel MIMD applications on SIMT hardware like GPUs, ThreadFuser generates warp-based instruction traces.

These traces can be fed to trace-based SIMT hardware simulators, like Accel-Sim [27]. Utilizing these traces enables detailed microarchitectural performance predictions, including cycle-level timing predictions using the SIMT simulator. This capability significantly enhances the precision of performance estimation, providing a deeper understanding of the application's behavior on SIMT hardware. Section V-A discuss how ThreadFuser used to predict the actual speedups of various MIMD applications with minimum developer effort.

ThreadFuser generates x86 instruction SIMT traces, which Accel-Sim utilizes by mapping those instructions to Accel-Sim virtual instructions. Furthermore, ThreadFuser converts x86 CISC instructions to a set of multiple RISC instructions. For example a CISC *add* instruction with memory operand breaks down into a *load* and *add* when converted to RISC instructions. ThreadFuser directs stack accesses to local memory accesses and other accesses to global memory accesses.
This feature is particularly helpful for architects seeking to explore various application types, including those with complex control flow or memory access patterns. Such exploration is important in optimizing performance on SIMT hardware. The availability of this capability proves beneficial in the design of new domain-specific SIMT accelerators for applications like microservices, as exemplified by the

hardware architecture proposed in previous work [14], [26], [44]. In Section V-A, we discuss the accuracy and validity of these traces on Accel-Sim.

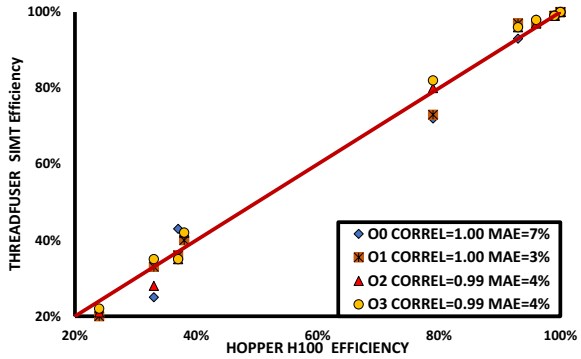
IV. Correlation Results

In this section, we examine the accuracy of our proposed framework, ThreadFuser (specifically the analyzer), by conducting a validation against real SIMT hardware, specifically an NVIDIA Hopper H100. The validation employs 11 multithreaded CPU workloads, outlined in Table I, implemented using Linux POSIX threads (Pthread) and OpenMP frameworks. Our selection includes applications derived from Rodinia 3.1 [12], featuring OpenMP implementations identical to their CUDA implementations. Furthermore, we re-implemented three applications with complex control flow graph from the Parapoly [48] suite using Pthreads. Also we implemented two microbenchmarks which are simple vector multiply-add kernels with different memory accessing patterns.

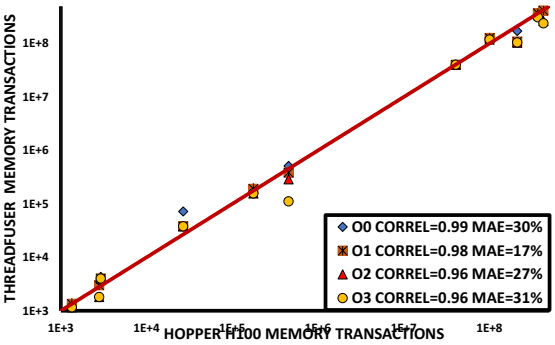
For validation, two metrics are employed: (1) Mean Absolute Error (MAE) and (2) Karl Pearson Coefficient of Dispersion (Correl). MAE quantifies the error between the analyzer data and hardware data, providing insight into the accuracy of our framework. Correl, on the other hand, assesses the strength of trends between the analyzer and hardware data.For our experiments, we employ the CUDA 12.3 toolkit, including the NVCC compiler toolchain, runtime library, and SDK utilities. Furthermore, we collect SIMT efficiency and memory transaction data for Volta GPU hardware using the Nvidia Profiling tool (*Nsight Compute*) [34]. These metrics and tools collectively form the basis for a robust validation process, ensuring the reliability and accuracy of ThreadFuser against real SIMT hardware.

We compile the CUDA workloads using *nvcc* at the *-O3* optimization level. We did not observe significant variations in the results when the CUDA workloads were compiled using different optimization options. In contrast, we adopt the *gcc* compiler for compiling CPU workloads, employing various optimization options such as *O0*, *O1*, *O2*, and *O3*. Subsequently, traces are collected for each optimized binary version. The tracing tool, developed using the Intel PIN 3.15 platform, captures x86 traces from our multithreaded applications. This tracing procedure is conducted on a machine with an Intel Xeon CPU E5-2630 with 20 cores.

In scenarios where the CPU application establishes a thread pool (e.g., OpenMP), the workload distribution among threads depends on the *OS* or the library's scheduling policy. In contrast, in GPUs, all threads within a kernel execute identical code and handle an equivalent amount of fine-grain work. To ensure an equitable comparison between CPU and GPU threads, ThreadFuser addresses this challenge by generating a trace for each loop iteration in OpenMP workloads and a trace for each call to the traced thread's worker function in Pthread workloads. This approach guarantees consistency in data partitioning and interleaving between CPU and



(a) ThreadFuser SIMT efficiency correlation.



(b) ThreadFuser memory divergence correlation.

Fig. 5: ThreadFuser SIMT efficiency and memory divergence correlation versus Nvidia Hopper H100.

GPU implementations throughout the study. In Table I, the "#SIMT Threads" column denotes both the number of threads executed on the actual GPU and the number of OpenMP iterations/Pthreads worker functions simulated by ThreadFuser.

SIMT Efficiency Correlation: Figure 5a illustrates a comparative analysis of SIMT control efficiency as projected by ThreadFuser in contrast to SIMT-based NVIDIA H100 GPU hardware. This experiment includes all correlation workloads detailed in Table I, compiled with varying optimization options to ensure a comprehensive evaluation. ThreadFuser demonstrates a perfect 1.0 correlation with hardware when employing *O0* and *O1* optimizations. Notably, *O1* exhibits a minimal 3% mean absolute error, making it the closest approximation to the real GPU.

In general, the analyzer tends to overestimate SIMT efficiency when *O3* optimizations are applied, given that the compiler employs more aggressive optimization strategies to enhance performance. Techniques such as loop unrolling and jump table implementation for switch statements play a role in minimizing code divergence.

An interesting observation we found when comparing the PTX generated by *nvcc* and the x86 assembly code generated

by *gcc*. Surprisingly, *gcc* applies more aggressive optimizations than *nvcc*, even when both compilers are invoked with the highest optimization option.*gcc* targets x86 CPUs, which have a rich instruction set and various optimization opportunities. For example, *gcc* can utilize SSE and AVX instructions for vectorization and parallel processing, optimizing performance for x86 architectures.

We calculated the standard deviation (std) of errors across the different optimization levels for all 11 applications to assess the variability in performance outcomes. The calculated std value is approximately 6%, reflecting the range of errors observed among the tested optimization levels. Additionally, the average error across these samples is approximately 4%. Notably, 30 out of these 44 samples, or approximately 83%, exhibit errors that fall within one standard deviation from the mean. This statistical measure provides insights into the consistency and predictability of error outcomes relative to the mean performance across the optimization spectrum.

Memory Accesses Correlation: In Figure 5b, we present a comparison of the total number of 32-byte memory transactions for each workload detailed in Table I. This figure illustrates the relationship between the load/store accesses estimated by ThreadFuser and the actual data collected during the execution of the workloads on the Nvidia H100 GPU. The analysis focuses on the overall number of **global** transactions issued per load/store on the GPU and the number of accesses per each x86 instruction generating memory accesses to the heap segment (as disccused in section III). Both the X and Y axes of the graph utilize a base-10 logarithmic scale.

Remarkably, ThreadFuser exhibits a robust correlation with hardware data for memory transactions, akin to the efficiency study. Specifically, the correlation factors are 0.99, 0.98, 0.98, and 0.96 for *O0*, *O1*, *O2*, and *O3*, respectively. Among these optimization options, *O1* boasts the lowest mean absolute error (MAE) of 17%. Similar to SIMT efficiency analysis, the study of memory transactions reveals that 85% of the applications exhibit transaction rates within one standard deviation of the mean rate, approximately 34%.

Upon analyzing the assembly code generated by the *gcc* compiler across different optimization levels, significant variations in memory accesses were observed. Specifically, *O0* exhibited a tendency to include a load or store instruction for each global variable access, whereas *O2* and *O3* applied more aggressive optimizations to minimize memory transactions. These optimizations leverage the differing number of architectural registers provided by SIMT and x86 ISAs as well as the significant differences in compiler toolchains, influencing the accuracy of memory transaction estimates in ThreadFuser.

Taking the example of VectorAdd, the use of *O2* and *O3* resulted in storing the cumulative sum value in the register file, reducing the number of issued transactions. *O1*, with a moderate level of optimization, found a middle ground, positioning itself between the extremes and achieving the lowest Mean Absolute Error (MAE) of 17% among the four levels of optimization.

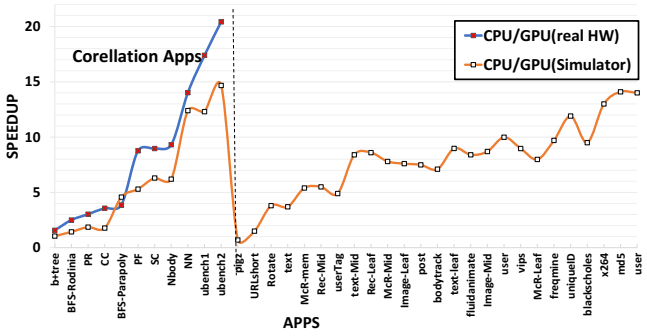


Fig. 6: Projected Speedup of the studied MIMD CPU workload using GPU simulator(Speedup Normalized to Multi-threaded CPU execution on actual CPU).

V. *ThreadFuser* Use Cases

This section unveils use cases of ThreadFuser for both developers and architects, illustrating the practical value and versatility of the ThreadFuser framework. In the first use case, covered in Section V-A, we explore a practical application of the ThreadFuser framework for developers. This section demonstrates the analyzer's ability to swiftly estimate SIMT efficiency with minimal time overhead and to project speedup when utilized with a warp-based simulator. We present a case study in which we identify the sources of SIMT inefficiency bottlenecking a particular workload. Importantly, this is achieved without the necessity of undergoing the porting effort. In Section V-B, we demonstrate the framework's utility for architects in guiding the design of future SIMT hardware. We emphasize the importance of designing future SIMT hardware to handle a broader range of general-purpose workloads. The section offers a detailed study of the impact of synchronization primitives on SIMT efficiency, explores the relationship between SIMT width and efficiency, and discusses the design considerations for compilers and underlying memory systems to exploit the efficiency of SIMT hardware.

A. Developers Use Case

Table I provides an exhaustive catalog of the workloads incorporated in this case study. These workloads primarily rely on CPU-based processing and currently lack a corresponding CUDA implementation. Consequently, we utilize the original CPU C++ implementation without any modifications to assess the Single Instruction, Multiple Thread (SIMT) efficiency of these multi-threaded workloads. We limit traced threads to capture essential patterns efficiently. Additional threads would repeat the same patterns without adding significant insights.

The focal point of this case study revolves around exploring the feasibility of executing these multi-threaded workloads on SIMT hardware, particularly GPUs [35], [36]. To achieve this, compute-intensive applications, alongside data center microservice workloads, are implemented using C++ OpenMP and Pthread libraries. The microservices, crafted in C++,

leverage a spectrum of libraries, including C++ stdlib, Intel MKL, gRPC, and FLANN. Compilation is carried out using *gcc* with the *-O3* optimization and SSE vectorization enabled.

To optimize performance, the applications with compute-intensive tasks are partitioned into multiple chunks, and each thread executes the same computation on the assigned chunk in parallel. This design aligns with the Single Program Multiple Data (SPMD) pattern, making the applications well-suited for efficient parallelization.

As illustrated in Figure 1, the data compression benchmark, *pigz*, displays constrained efficiency due to its control flow being intrinsically data-dependent [28], [29], [37]. Notably, microservices applications such as TextSearch exhibit remarkable SIMT efficiency. This is a noteworthy observation, considering that the SIMT hardware domain has, until now, remained relatively unexplored and unported for microservices applications.

By leveraging ThreadFuser, developers can rapidly and cost-effectively estimate the porting effort for these workloads, minimizing overhead. This provides developers with a valuable tool to gauge the potential SIMT efficiency of their code without undergoing the actual porting process. SIMT efficiency is a pivotal factor that dictates the extent of porting and code reengineering efforts required to make a code SIMT-friendly. Therefore, this estimation approach aids developers in making informed decisions about the feasibility and resource implications of adapting their code for SIMT architectures.

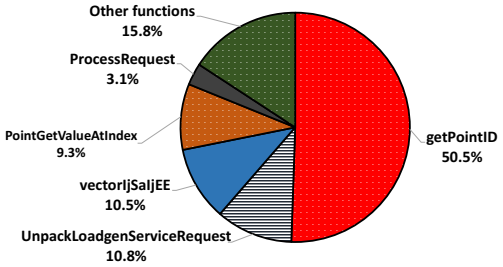
Moreover, through the ThreadFuser trace generation feature, developers can utilize state-of-the-art simulators such as AccelSim [27] to forecast the performance speedup gains when migrating CPU MIMD applications to SIMT hardware, like GPUs. Figure 6 delineates the projected speedups for all the applications listed in Table I.

For applications where a SIMT implementation exists in CUDA, traces for the CUDA implementation were generated using nvbit [46] and the tracing tool integrated with AccelSim [27] on an Nvidia Volta 100 GPU. For CPU implementations, traces were generated using ThreadFuser with a warp size of 32 threads. Subsequently, both sets of collected traces were executed in the AccelSim simulator, configured with Nvidia RTX 3070 settings.

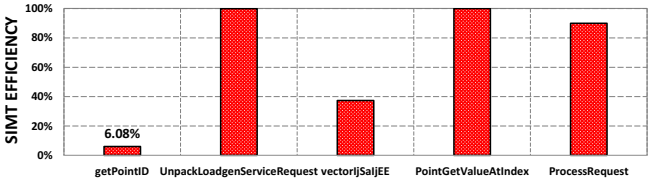
As depicted in the figure, ThreadFuser enables accurate measurement and prediction of performance, as evidenced by the close alignment of both series on the left side of the graph, following the same trend line. This capability shows the effectiveness of ThreadFuser in providing developers with valuable insights into the potential speedup of their applications on SIMT hardware, aiding in informed decision-making during the porting process.

In cases where access to the source code is available, ThreadFuser proves to be an invaluable tool for identifying the underlying causes of low SIMT efficiency when porting current MIMD implementations to SIMT hardware.

We conducted an in-depth analysis on one of the microservices, specifically *HDSearch-Midtier*. Initially, the SIMT efficiency for this workload was measured at only 7%. However,



(a) Distribution of warp instructions executed per function



(b) Warp control efficiency per function

Fig. 7: HDSearch-Midtier microservice analysis

leveraging ThreadFuser's per-function analysis, we were able to break down the SIMT efficiency on a per-function basis, enabling us to pinpoint specific code sections exhibiting high control divergence. The distribution of executed instructions per function call in *HDSearch-Midtier*, depicted in Figure 7a, revealed that half of the instructions were generated from the *getpoint* method, showing a substantial control divergence of 6%, as illustrated in Figure 7b.

Further examination of the source code of the *getpoint* function in the FLANN library, as shown in Listing 1, unveiled data-dependent control flow for kd-tree traversal. To address this issue, we ensured that the number of computations performed by *getpoint* matched the originally reported results by the *HDSearch-Midtier* microservice to the client. By fixing the number of computations to return the first top 10 results for all queries, we guaranteed uniform execution across all threads in the *for* loop at line #5. This adjustment significantly increased SIMT efficiency to 90%, while still maintaining an impressive 93% image search accuracy. Notably, the other two methods, *ProcessRequest* and *vector*, faced limitations associated with the serialization from dynamic memory allocation in the C++ glibc library.

```
1  for (; table != table_end; ++table) {
2    for (; xor_mask != xor_mask_end; ++xor_mask) {
3      sub_key = key ^ (*xor_mask);
4      ....
5      for (int j = 0; j < num_point; j++) {
6        point_id_vec->push_back(point);
7      }
8    }
9  }
```

Listing 1: Code snapshot from FLANN library used in HDSearch-midtier workload

B. Architects Use Case

ThreadFuser aimes at assisting architects in evaluating the efficiency and effectiveness of emerging SIMT hardware. It

provides architects with the opportunity to delve into a new realm of workloads that were previously unconsidered for SIMT hardware. This capability empowers architects to explore a diverse set of applications beyond the traditional focus on graphics and machine learning. By doing so, architects gain valuable insights into the hardware features necessary for future SIMT architectures, enabling them to design novel accelerators tailored to a broader spectrum of applications like the ones introduced in previous work [13], [14], [26], [44].

For example, utilizing ThreadFuser, architects can efficiently investigate the impact of SIMT warp width on SIMT efficiency. While a warp size of 32 is commonly employed in modern SIMT hardware, such as NVIDIA GPUs [36], we extend the configurability of the warp size in ThreadFuser, ranging it from 8 to 32. Comprehensive experiments are then conducted on all the workloads discussed in the preceding sections.

As depicted in Figure 1, the SIMT efficiency of the workloads is illustrated across various warp sizes. The results prominently showcase a consistent trend: as the warp size increases, the efficiency of all workloads experiences a decline. This observed decline can be attributed to the augmented number of threads accommodated within the warp, consequently escalating the likelihood of thread divergence.

The decreasing control efficiency with an increasing warp width stems from the expanded number of threads in the warp. This, in turn, amplifies the probability of thread divergence. Furthermore, the increase in control-flow divergence becomes more harmful as all divergent paths are executed serially, leading to a reduction in parallelism. This observed behavior aligns seamlessly with Equation 1, as previously discussed.

Figure 1 shows that high SIMT efficiency workloads are less affected by warp size. For example, *Nbody* and *MD5* have minimal variations below 5%.

On the other hand, lower SIMT efficiency workloads are more sensitive to warp size. *Pigz* achieves 18% efficiency with a warp size of 8, compared to 10% with a warp size of 32. Similarly, Rodinia BFS jumps to 40% efficiency with an 8-thread warp size.

These findings imply that workloads with lower SIMT efficiency could benefit from hardware optimizations or warp size adjustments. Understanding sensitivity variations across workloads is crucial for architects optimizing SIMT efficiency based on each application's unique characteristics.

Impact of synchronization primitives on SIMT efficiency: When looking at workloads that involve synchronization, like microservices, it's important to note that the SIMT efficiency we report assumes fine-grain locking and doesn't consider lock spinning in the traces we generate. For example, the C++ glibc allocator uses a single shared mutex for dynamic memory allocation, causing conflicts between threads, especially during the *new* operation.

We assume the use of fine-grain locking and a high-throughput concurrent memory manager, a common practice in optimized data center workloads for good performance scaling [17].

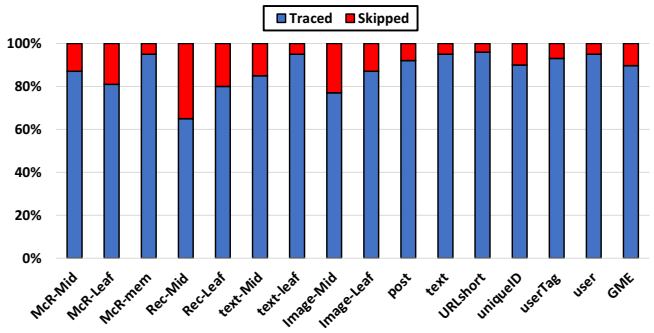


Fig. 8: Percentage Distribution of Traced and Skipped Instructions: Tracing (Non-I/O) vs. Skipping (I/O Operations and Lock Spinning).

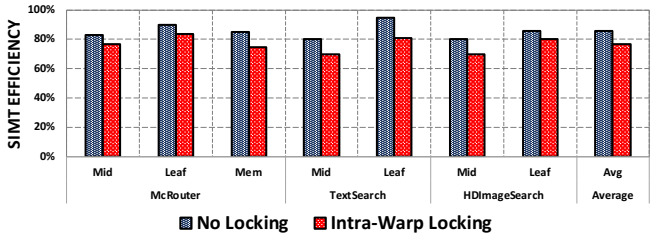


Fig. 9: Warp efficiency of microservices workloads (Warp size=32) when Intra warp locking is emulated .

In Figure 8, we show the percentage of instructions not traced, including spinning lock instructions and I/O operations. The GEOMEAN indicates that we are tracing 90% of the instructions, which means that we can safely skip the remaining 10%. This approach ensures a practical evaluation of SIMT efficiency in workloads with synchronization, giving a realistic view of performance scaling potential.

Furthermore, architects can leverage ThreadFuser to explore more complex workloads where synchronization primitives play a crucial role, an area often overlooked and underexplored in SIMT hardware like GPUs.

Using ThreadFuser, we investigate the impact of implementing locks and other synchronization primitives on the overall control flow efficiency. The study used a measure of SIMT efficiency built for data center workloads, as depicted in Figure 9.

Dealing with synchronization primitives, following the intra-warp approach discussed in Section III, did result in a decrease in SIMT efficiency. However, this decline was not found to be as substantial, aligning with the findings discussed in 8. This observation can be attributed to the nature of the workloads studied, where microservices handle independent requests from multiple clients, resulting in minimal data sharing. Additionally, upon closer examination of the source code, we identified the use of fine-grained locking, effectively mitigating lock contentions and reducing synchronization overheads.

This study shows the importance of carefully optimizing synchronization methods to ensure optimal control flow

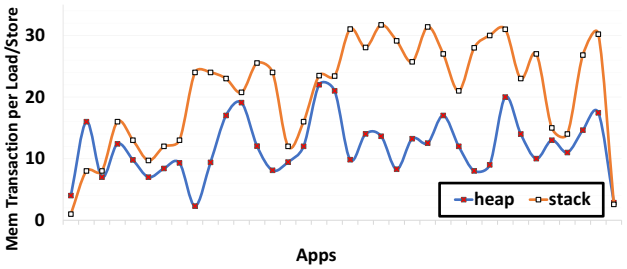


Fig. 10: Memroy transactions per load/store instruction (Warp size=32).

Metric	XAPP	ThreadFuser
Input	CPU code	CPU MIMD traces
Output	GPU speedup projection	Estimation SIMT efficiency Estimation memory divergence Cycle level detailed performance estimation Source code bottlenecks
Analysis	Profiling ML-based	Dynamic CFG
Accuracy (error %)	26.9% Execution time	3% SIMT efficiency , 17% memory 0.97 Speedup projection correlation 33% Execution time
Hardware support	Only GPUs	Any SIMT hardware including GPU

TABLE II: XAPP [8] vs ThreadFuser comparison

efficiency in parallel computing environments.

Impact of Memory divergence: In Figure 10, we present the memory divergence degree, quantified as the number of memory transactions per load/store instruction, for both heap and stack memory access. The depicted workloads exhibit significant memory divergence, a result of each thread having its private stack, and the memory manager allocating scattered data chunks in the heap segment. This allocation strategy diminishes the opportunity for data coalescing during runtime. To address this challenge, data restructuring, such as transitioning from a array-of-structure (AoS) to a structure-of-array (SoA) representation, can enhance the memory efficiency of these workloads [36].

Architects and compiler developers play a pivotal role in proposing hardware or software solutions, such as optimized memory allocators [17], [24], [38], [42], to minimize memory divergence. This proactive approach ensures that the memory access patterns align with the underlying hardware architecture, optimizing data layout, and consequently improving overall memory efficiency for these workloads.

VI. Related Work

The closest existing work to ThreadFuser is XAPP [8]. Table II outlines the main differences between XAPP and ThreadFuser. XAPP employs a machine-learning-based approach to predict GPU performance based on single-threaded CPU implementations, identifying code segments likely to benefit from GPU porting using 16 profile-based program properties. In contrast, ThreadFuser utilizes dynamic control-flow graph properties for more accurate SIMT control efficiency and memory divergence predictions. In contrast to XAPP, ThreadFuser is designed to analyze and simulate MIMD workloads, allowing for fine-grained identification of code bottlenecks and detailed architectural explorations. XAPP's opaque machine learning model is well suited to providing a quick estimate of code performance but lacks the comprehensive performance analysis ThreadFuser's architectural simulation integration provides.

From a prediction accuracy perspective, XAPP achieves a 26.9% error rate in predicting execution time, while ThreadFuser averages 33% for similar workloads. Moreover, while XAPP is limited to existing GPU architectures, ThreadFuser supports the analysis of multithreaded CPU workloads on various SIMT hardware platforms, including GPUs. This capability is facilitated by ThreadFuser's SIMT trace generator, which can feed any SIMT trace-based simulator, enabling the exploration of diverse SIMT hardware microarchitectures and offering insights for previously unstudied MIMD workloads. ThreadFuser stands out as the first framework enabling programmers and architects to comprehensively study and predict MIMD performance on SIMT machines without code porting effort. Furthermore, ThreadFuser excels in accurate predictions of dynamic workload performance that vary with input, distinguishing it from XAPP's reliance solely on static code analysis.

In GPU First [43], the authors present a compiler technique for executing CPU code on GPUs, but it is limited to OpenMP programs. In contrast, our approach supports a broader range of multithreading models, offering greater flexibility for porting CPU workloads to SIMT architectures.

Various GPU analytical performance models with distinct focuses have been proposed [22], [23], [47], [49]. Wu et al. [47] employ machine learning for GPU performance and power estimation, while Zhou et al. [23] introduce GPUMech, using interval analysis to model multithreading and resource contentions. Zhang et al. [49] develop a microbenchmark-based model to identify GPU program bottlenecks, and Hong et al. [22] propose an integrated power and performance prediction model for GPUs. Previous works [11], [40], [46], [50] have proposed performance profiling methodologies for CUDA workloads, aiming to pinpoint performance bottlenecks. Zhou et al. [50] introduce GPA, a performance advisor for NVIDIA GPUs that suggests potential code optimizations across multiple levels. Shen et al. [40] present CUDAAdvisor, a profiling framework designed to guide code optimization on modern GPUs, conducting fine-grained analyses based on profiling results from GPU kernels.

In contrast, ThreadFuser offers distinctive insights, specifically targeting CPU MIMD programs such as OpenMP or microservices. ThreadFuser determines whether these programs would benefit from porting to SIMT hardware and provides performance optimization suggestions to enhance SIMT efficiency.

Previous research [3], [4], [18] have found that server workloads have SIMT efficiency that can be exploited on GPUs. To take advantage of request similarity, Sandeep et al. [3] proposed running data center server workloads, SPEC-

Web benchmarks, in lock-step execution on GPUs. While achieving significant energy efficiency, the authors had to rewrite the workloads from PHP to CUDA. Similarly, Hetherington et al. [18], [19] run the Memcahced workload on a GPU. Agrawal et al. [4] investigate the SIMT efficiency of SPEC-web workloads, demonstrating that they have promising control and memory efficiency that can be executed on SIMT hardware. In all these previous studies, they focused on monolithic services and had to rewrite the workloads in CUDA, whereas ThreadFuser analyzes SIMT efficiency of microservices as-is on the CPU in its original programming language. Furthermore, they do not explore the bottlenecks of SIMT deficiency.

Tino et al. [44] describe how an out-of-order pipeline can be integrated with SIMT hardware to execute OpenMP workloads efficiently. ThreadFuser can be used to improve the reach of these previous works by providing control flow and memory efficiency estimations for a broad class of CPU workloads.

VII. Limitations

With zero coding effort, the framework enables programmers to project the first-order performance gain of multi-threaded CPU workloads on SIMT hardware, including arbitrary multithreaded code that does not follow the SPMD programming model. Although ThreadFuser gives the programmer a sense of how well the program will perform with zero restructuring, it does not directly project the potential performance possible if the programmer is willing to invest the effort required to leverage GPU-specific features such as shared memory or tensor cores. However, by leveraging ThreadFuser's detailed simulation results, the programmer can assess the potential impact of these optimizations. Programmers must explicitly use these features for maximum performance benefits; for example, existing SIMT compilers do not automatically target tensor cores. Similarly, optimizing data movement for shared memory requires careful manual consideration.

From an ISA perspective, since ThreadFuser relies on the output of a CPU compiler optimized for x86 machines, some of the assumptions made by the compiler could be sub-optimal for a SIMT design. For example, the register allocation mechanism is unaware of the large multi-threaded registers available for throughput hardware, which may lead to more register spills/fills than necessary. These kinds of hardware/software co-design considerations are possible for architects to assess by modifying ThreadFuser's simulation infrastructure but hamper direct correlation between equivalent CPU and GPU programs. For example, the CPU compiler's optimization level can have a non-trivial effect on memory system correlation, as shown in Figure 5.

From a programmer's perspective, ThreadFuser is most useful in projecting the performance of large codebases with less structured and less easily predictable patterns. Estimating the performance of a matrix multiplication CPU program, for example, is better left to higher-level roofline analysis since optimized library kernels already exist for these kinds of workloads. ThreadFuser focuses more on finding the next optimization frontier for SIMT-based accelerators than perfectly predicting their current use cases in workloads where a rich suite of software already exists. We believe that SIMT simulation of these workloads will enable architects to explore more exotic SIMT-based designs not limited to today's GPUs and drive the exploration of more heterogeneous hardware.

VIII. Conclusion

We introduce ThreadFuser, a correlated analysis framework designed to evaluate the effects of SIMT execution on arbitrary CPU binaries. Without changing a line of code, ThreadFuser enables application developers to evaluate the compatibility of multithreaded CPU applications with SIMT GPUs. We demonstrate that there is significant potential to accelerate MIMD CPU programs, previously considered too irregular for SIMT, with contemporary GPUs. ThreadFuser's per-function analysis helps to rapidly identify problematic code segments, providing a clear understanding of code modifications that are necessary for efficient GPU execution.

By integrating with state-of-the-art simulation tools [27], ThreadFuser enables architects and system designers to study the implications of data-parallel acceleration on diverse software. The GPU revolution in machine learning was enabled by giving programmers the ability write general-purpose code for a data-parallel SIMT machine. ThreadFuser seeks to build on this momentum, unhindered by the often painstakingly task of porting code to a GPU programming language. ThreadFuser allows architects to explore the wide space of CPU software and evaluate alternative SIMT accelerator designs in new domains that have yet to be accelerated.

Acknowledgment

We thank our anonymous reviewers for their feedback, which helped improve this paper. This work was supported, in part, by NSF CCF #1943379 (CAREER) and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA.

References

[1] "A Parallel Implementation of gzip for Modern Multi-processor, Multi-core Machines," https://zlib.net/pigz/.

[2] T. M. Aamodt, W. W. Fung, I. Singh, A. El-Shafiey, J. Kwa, T. Hetherington, A. Gubran, A. Boktor, T. Rogers, and A. Bakhoda, "GPGPU-Sim 3. x manual," http://gpgpu-sim.org/manual/index.php/Main_Page, 2016, accessed June 29, 2016.

[3] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing data parallel hardware for server workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, vol. 42, no. 1. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2654822.2541956

[4] V. Agrawal, M. A. Dinani, Y. Shui, M. Ferdman, and N. Honarmand, "Massively Parallel Server Processors," *IEEE Computer Architecture Letters*, 2019.

[5] B. Akil, Y. Zhou, and U. Röhm, "On the usability of Hadoop MapReduce, Apache Spark & Apache flink for data science," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 303–310.

[6] "AMD HIP Programming Guide ," https://rocmdocs.amd.com/en/latest/, AMD, 2021, accessed 17-Dec-2021.

[7] M. Andersch, B. Juurlink, and C. C. Chi, "A Benchmark Suite for Evaluating Parallel Programming Models," in *Proceedings 24th Workshop on Parallel Systems and Algorithms*, 2011. [Online]. Available: http://www.aes.tu-berlin.de/fileadmin/fg196/publication/andersch01.pdf

[8] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 725–737.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Usinga Detailed GPU Simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, 2009, pp. 163–174.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and Architectural Implications," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

[11] L. Braun and H. Fröning, "CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications ," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 73–81.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[13] C. Collange, "Simty: a Synthesizable General-Purpose SIMT Processor," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:11107857

[14] ——, "Simty : generalized SIMT execution on RISC-V Sylvain Collange Inria sylvain," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:27721335

[15] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 407–420.

[16] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[17] I. Gelado and M. Garland, "Throughput-oriented GPU memory allocation," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 27–37. [Online]. Available: https://doi.org/10.1145/3293883.3295727

[18] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "MemcachedGPU: Scaling-up Scale-out Key-value Stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2806777.2806836

[19] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems," in *IEEE International Symposium on Performance Analysis of Systems & Software(ISPASS)*, 2012, pp. 88–98.

[20] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, "A Pilot Study to Compare Programming Effort for Two Parallel Programming Models," *Journal of Systems and Software*, vol. 81, no. 11, pp. 1920–1930, 2008.

[21] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," in *Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05*, 2005.

[22] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture(ISCA)*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 280–289. [Online]. Available: https://doi.org/10.1145/1815961.1815998

[23] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU Performance Modeling Technique Based on Interval Analysis," in *47th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*. IEEE, 2014, pp. 268–279.

[24] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W.-m. Hwu, "Scalable SIMD-parallel memory allocation for many-core machines," *The Journal of Supercomputing*, vol. 64, no. 3, pp. 1008–1020, Jun 2013.

[25] S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec, "DITVA: Dynamic Inter-Thread Vectorization Architecture," *Journal of Parallel and Distributed Computing*, 2018.

[26] M. Khairy, A. Alawneh, A. Barnes, and T. G. Rogers, "SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices," in *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 441–463.

[27] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, 2020.

[28] F. Knorr, P. Thoman, and T. Fahringer, "Ndzip-Gpu: Efficient Lossless Compression of Scientific Floating-Point Data on GPUs," in *International Conference for High Performance Computing, Networking, Storage and Analysis(SC21)*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476224

[29] L. Lu and B. Hua, "G-Match: A Fast GPU-Friendly Data Compression Algorithm," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 788–795.

[30] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 40, no. 6. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[31] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 235–246.

[32] S. S. Muchnick, *Advanced Compiler Design and Implementation*, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:32801154

[33] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking Usability and Performance of Multicore Languages," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 183–192.

[34] NVIDIA, "NVIDIA profiling tools," https://docs.nvidia.com/nsight-compute/NsightComputeCli/, NVIDIA, 2018, accessed Aug 20, 2018.

[35] "NVIDIA CUDA C Programming Guide v7.5," https://www.spec.org/jbb2015/, NVIDIA Corp., 2016, accessed August 6, 2016.

[36] "NVIDIA CUDA C Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, NVIDIA Corp., 2020, accessed August 6, 2020.

[37] M. A. O'Neil and M. Burtscher, "Floating-Point Data Compression at 75 Gb/s on a GPU," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1964179.1964189

[38] M. Pham, H. Li, Y. Yuan, C. Mou, K. Ramachandran, Z. Xu, and Y. Tu, "Dynamic Memory Management in Massively Parallel Systems: A Case on GPUs," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3524059.3532387

[39] C. Sadowski and A. Shewmaker, "The Last Mile: Parallel Programming and Usability," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 309–314.

[40] D. Shen, S. L. Song, A. Li, and X. Liu, "CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization(CGO)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 214–227. [Online]. Available: https://doi.org/10.1145/3168831

[41] A. Sriraman and T. F. Wenisch, "μ Suite: A Benchmark Suite for Microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 1–12.

[42] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "Scatteralloc: Massively parallel dynamic memory allocation for the gpu," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[43] S. Tian, T. Scogland, B. Chapman, and J. Doerfert, "GPU First – Execution of Legacy CPU Codes on GPUs," 2023. [Online]. Available: https://arxiv.org/abs/2306.11686

[44] A. Tino, C. Collange, and A. Seznec, "SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3392032

[45] 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 868–880.

[46] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[47] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU Performance and Power Estimation Using Machine Learning," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 564–576.

[48] M. Zhang, A. Alawneh, and T. G. Rogers, "Characterizing Massively Parallel Polymorphism," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 205–216.

[49] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture(HPCA)*. IEEE, 2011, pp. 382–393.

[50] K. Zhou, X. Meng, R. Sai, and J. Mellor-Crummey, "GPA: A GPU Performance Advisor Based on Instruction Sampling," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 115–125.