A Systemic Approach to Maximize Heterogeneous System Performance

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Thomas Lenzy Randall
May 2025

Accepted by:
Dr. Rong Ge, Committee Chair
Dr. Kai Liu
Dr. Feng Luo
Dr. Prasanna Balaprakash
Dr. Xingfu Wu

Plain Language Abstract

My research focuses on maximizing the performance of some of the world's biggest applications running on the world's biggest computers. These machines are so large and complicated that it takes dedicated efforts to get the biggest bang for your buck (and these machines aren't cheap, so we really want it!). The research in this dissertation is composed of three connected parts: How do we measure the opportunities provided to us by new hardware? How do we design our applications to reduce the need to re-design them as hardware improves? How do we configure our hardware and software to work together as effectively as possible?

I provide deep insights into the opportunities that we can leverage right now to prepare us for future developments. This means that we can extend the lifetime of our efforts and solve new and bigger problems instead of just struggling to keep up with the problems that we're already working on. When applied in the United States Department of Energy, this means that my research helps other scientists do their work faster and with less taxpayer dollars footing the bill for energy and computer maintenance. My research also supports smaller organizations who many not have the expertise, manpower or resources to optimize every tiny detail. My research in performance tuning delivers results quickly with a known lower bound even when there isn't a lot of data to work with, making it easier for everyone to get the most of new computing technologies.

Abstract

Continuous increases in high performance computing (HPC) throughput have served as catalysts for industry and scientific advancement in countless manners that have fundamentally shaped our modern world. Our demands on compute resources continue to scale, but the limitations of Ahmdal's law and Dennard scaling have proven increasingly difficult to overcome when approached solely through hardware or software design. Furthermore, even on the most advanced supercomputers, many HPC applications fail to utilize the collective system's performance.

However, the resurgence of AI in industry has promoted an explosion of hardware and software codesign that have fueled massive improvements in GPU design and novel ASICs. These performance improvements are maximized on a broad variety of heterogeneous systems by specially tuning applications. Mimicking these developments across the whole of computing will require similarly holistic approaches combining specialty hardware, software that caters its design to the greatest strength of hardware, and fine-tuning on individual systems to truly maximize performance.

We use three distinct perspectives to holistically address scalable system performance. We analyze the impacts of liquid immersion cooling technologies on sustained application performance and energy efficiency. Next, we present a case study where intentional algorithmic redesign for GPU acceleration permits robust performance improvements that endure through multiple generations of hardware. We find that memory latency forms a primary bottleneck for GPU-accelerated performance and demonstrate how algorithm-specific optimizations can greatly improve performance over multiple architecture generations. Finally, we tie these concepts together in the form of performance optimization techniques that respect both software- and hardware-based performance constraints. We improve the re-usability of performance insights with novel transfer learning techniques that make the cost of performance optimization more predictable and more successful in the short-term. Our insights demonstrate the necessity of systemic approaches for performance tuning in HPC.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Building and operating supercomputers are massive undertakings that ultimately fall short of expectations wherever the hardware and software are not well-aligned. The ever-growing complexity of novel hardware and advanced software necessitate sophisticated solutions for both new and old problems that limit performance.

Effective optimizations must be designed against the actual constraints that limit performance; maximizing heterogeneous performance requires optimization of hardware, software, and their integration.

While decades of research have already dedicated significant efforts to improve system performance, the so-called "golden age of computer architectures" [1] has dramatically accelerated both the raw throughput of data computation and the complications of efforts to efficiently utilize hardware. CPU hierarchies are well-understood and GPU hardware have matured over time, but the rapid evolution of system design has left many heterogeneous systems regularly underperforming their theoretical peaks. Continual large-scale government and industry investments in specialized hardware for artificial intelligence, machine learning, cloud infrastructure and datacenter technologies have vastly outpaced our ability to maximize the utilization of these systems.

Many advancements in specialized hardware have been accompanied by increasing power consumption and thermal waste, raising legitimate concerns about the energy efficiency and environmental impacts of current and future large-scale computing. We also observe many applications failing to improve key metric performance at the rate of new hardware acquisition, suggesting un-

fortunate mismatches between software design and hardware implementation that are inefficiently addressed by newer technologies. Finally, we observe the largest gap in system performance is created by the immense efforts needed versus finite resources allocated for performance tuning. Even when efforts are made to facilitate the usage of new hardware with existing software, performance improvements remain lackluster compared to the cost of acquisition.

In this work, we tackle the larger problem of overall heterogeneous system performance by considering its three components: opportunities presented by hardware, methods by which software leverages these opportunities and automatic performance optimization to ensure that hardware is best-used by software.

Chapter 3 explores the performance opportunities provided to systems based on novel hardware in the form of a case study. We analyze the impacts of a Single-Phase Liquid Immersion Cooling (SPLIC) technology on cooling to determine how current and future applications may adapt to different physical constraints provided by this environment.

Chapter 4 demonstrates the complexity of designing software to properly exploit available performance across many generations of accelerator hardware through the lens of GPU implementations of the Word2Vec algorithm. We optimize Word2Vec performance on GPU accelerators to permit automatic performance improvements from successive hardware generations that have been neglected in prior work, remedying a history of poor GPU utilization that was previously outmatched by CPU implementations.

Chapter 5 unites these perspectives of hardware- and software- based optimization through automatic empirical performance tuning, also known as performance autotuning. Once performance opportunities are known, the process of tuning available interfaces for maximal impact is nontrivial. Existing performance autotuning has limited capability to exploit prior knowledge and is forced to waste some empirical samples when tuning related problems. We identify a novel transfer tuning technique that immediately leverages prior knowledge to access near-optimal areas of tuning search spaces, allowing practitioners to expand the scope and utilization of autotuning to applications at low and predictable cost.

The greatest opportunities for performance improvement are driven by understandings of all technologies in a system and their joint interactions. This work presents both the current-day challenges and future opportunities of heterogeneous system optimization at all levels: hardware, software and integration. By properly combining these techniques, a holistic view of heterogeneous

system performance permits the greatest impact to exceed the current boundaries of excellence in computing performance.

Chapter 2

Background

In this chapter, we provide supplementary knowledge that spans each of the components of our work. This includes an overview of SPLIC technology, the Word2Vec algorithm, and general concepts in performance autotuning.

2.1 SPLIC Technologies

One of the major ways to iteratively improve performance through hardware is to increase the density of computing components. However, denser circuitry increases the power density of the components and reduces cooling effectiveness for components such as those deeper within 3D dies, resulting in greater thermal accumulation within the hardware [2]. At the extreme scale, high heat can damage computing hardware and render it inoperable [3], creating a design limitation known as Dennard scaling [4]. To prevent such damage, thermal sensors in critical hotspots automatically trigger a decrease in the clock rate of devices to limit additional energy before the device becomes temporarily or permanently inoperable. To ensure performance and throughput, HPC and datacenter systems that push thermal limits require sophisticated cooling solutions. These solutions permit maximal usage for the longest possible period without detrimentally lowering clock rates or causing device failures.

Despite continued innovation in air-cooling technology, air is simply not dense enough to maintain pace dissipating heat from modern HPC and datacenter systems, let alone future designs that are predicted to only increase in Thermal Design Power (TDP). With nearly half of all

datacenter energy devoted to cooling hardware and datacenters accounting for 2% of U.S. energy consumption [5], liquid-based cooling technologies are promising and necessary for continued growth and innovation in computing. There are several different approaches to liquid-based cooling, including Direct Liquid Cooling (DLC) and Single- and Dual- Phase Liquid Immersion Cooling (SPLIC and DPLIC, respectively). As immersion cooling methods, the use of DPLIC or SPLIC require far more modifications and planning than DLC. At the time of writing, DPLIC has less favor and adoption within industry start-ups. This is partially due to concerns about the environmental impacts of currently available choices for DPLIC coolants and the relative degree of risks involved in containing gases as opposed to liquids used in SPLIC. We limit the focus of our discussion to SPLIC as the most prominently growing immersion cooling technology for the nearer future.

SPLIC has different cooling mechanisms, controls, and operational capabilities compared to air cooling. Circulation is induced to reach a temperature equilibrium over the entire fluid volume faster and to facilitate more effective heat exchange. The heat exchange is performed with chilled water from the facility; the fluids exchange heat while pumped through thermally conductive material (such as copper pipes). In comparison to Direct Liquid Cooling (DLC) [6, 7], immersion cooling affects all computing components rather than just the areas covered by cold plates where liquid is always flowing. SPLIC systems have high energy-efficiency because the pumps only activate periodically to circulate fluid or while the coolant temperature exceeds a set point. Unlike air cooling, which directly monitors computer hardware components and responds to their temperatures, SPLIC monitors the temperatures of the coolant and water entering and leaving the heat exchange within the tank. This approach to thermal monitoring is physically removed from the computing components, which inherently delays the system's response to heat accumulation within computing hardware. Therefore, understanding the actual impacts of this cooling technology on different hardware and software workloads is important for evaluating its utility and design implications beyond mere hardware configuration.

2.2 The Word2Vec Network and Training Algorithm

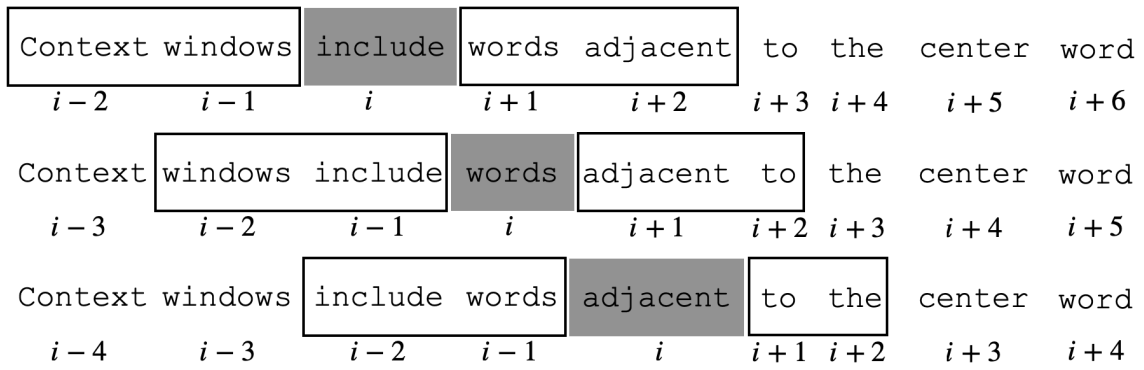
Word2Vec is a three-layer artificial neural network that learns to represent all words in a vocabulary V as d-dimensional vectors $v \in \mathbb{R}^d$ based on their usage in a set of sentences. These vectors are known as *word embeddings* and were of critical importance in developing the Natural

Language Processing (NLP) community's machine-learning approach, laying the groundwork for its successor in the transformer neural network architecture. Well-constructed word embeddings can reveal meaningful expressions of syntactic and semantic relationships between words. For instance, $distance(v_{cat}, v_{dog}) < distance(v_{cat}, v_{hammer})$ indicates that the word "cat" is more similar in meaning to "dog" than "hammer," and various verb tenses of the same word appear clustered in $\mathbb{R}^d$ to indicate similar syntactic uses. While transformers were designed to leverage GPU acceleration, Word2Vec and many other applications include implementations that are designed to benefit from GPU acceleration but do not receive the broad community support needed to optimize model performance. As such, time has proven that Word2Vec's GPU performance was strongly bottlenecked by limitations of the algorithm that proved adversarial to the hardware design. In Chapter 4, we demonstrate how such algorithms can be effectively designed to benefit from multiple generations of hardware. In this section, we provide the necessary background information about the Word2Vec network and training algorithm to understand the core foundation of this optimization problem.

The Word2Vec neural network is trained via an unsupervised objective. This objective is based on the fundamental hypothesis that words which appear nearby one another in human-written text must contain some grammatical, syntactical, or semantic connection. Word2Vec provides two different models of determining word proximity during training wherein the objective is to predict the most likely individual word to "fill in the blank" or the most likely context to "surround the blank". The Continuous Bag-of-Words (CBOW) model architecture attempts to "fill in the blank," while the Continuous Skip-Gram with Negative Sampling (SGNS) model architecture trains in an inverted fashion, attempting to "surround" a target word with a reconstruction of its context. Rogers et al. [8] found that the SGNS model architecture generally produces higher quality embeddings for downstream applications of interest, so we focus our attention on improving the performance of the Continuous SGNS model architecture.

The abstract overview of the Word2Vec algorithm is as follows: A vocabulary of trainable words V is formed from all words in a corpus of trainable text. The input to Word2Vec consists of this vocabulary of words and a corpus of those words organized into "sentences". Within each sentence, it is assumed that some meaningful relationship exists between nearby words. During training, the contents of individual sentences are consumed from beginning to end using a sliding *context window* of $2W$ "context" words where W is the window size, as shown in Figure 2.1. The center word is the current "target" word that the model is being trained against. The SGNS training algorithm

Figure 2.1: An example context window (bordered blocks) of size $W = 2$ centered on target words in gray. Most words are reused between successive context windows, providing predictable reuse opportunities.



focuses on reproducing the context seen in training at inference time by increasing the similarity of all context words to the target word. Such similarity is reflected in the vector representations by making each context word's vector more similar to the target word vector. The algorithm also assumes that words not present in the context window are *less* related to the target word, and their similarity to the target word is penalized. Rather than decreasing the similarity of all words not included in the context window, SGNS randomly samples a small number N words using a weighted distribution and makes them more dissimilar to the target word, hence the name "Continuous Skip-Gram with *Negative Sampling*". These "negatives" greatly reduce the data intensiveness of training as $N << |V|$. Like other neural network models, SGNS yields a converging solution for word embedding values with a sufficient number of iterations over the data set.

The context window size W and number N of negatives per word are typically defined as hyperparameters in Word2Vec models. Mikolov et al [9, 10] established that $W \in [2, 10]$ and $N \in [2, 20]$ are often sufficient for most datasets, accelerating training speed without reducing embedding quality, and smaller values of N are more appropriate for larger datasets.

2.3 Performance Autotuning

Autotuning [11, 12, 13] is a process that efficiently evaluates a number of parameter configurations from a user-defined parameterized kernel or application to optimize a given objective such as performance (e.g., runtime, FLOPS). Here we provide a walkthrough with the Polybench

kernel [14] "3mm" as a concrete example of basic autotuning concepts. The kernel performs dense matrix multiplication with four matrices A, B, C, D such that the output is $(A \times B) \times (C \times D)$.

Autotuning utilizes a finite budget (typically time or number of evaluations) to optimize a relationship $f(c;t) \in \mathbb{R}^d$ between a given parameter configuration c, out of all possible configurations C, a tuning task t, and d objective outputs such that $\arg\max_c f(c;t)\ \forall c \in \mathcal{C}$. Each task t is a specific instance from a set of related tasks $\mathcal{T}$, which may have different configurations for optimum performance. Each objective d is a real-valued metric that functionally depends on both the task and parameters according to $f(c;t)$. The exact closed form of $f(c;t)$ is unknown but is assumed to be a complex, nonlinear relationship.

An example task of tuning the 3mm kernel's runtime performance involves $n = 10$ parameters in the form of source code annotations that affect loop tile sizes (i.e., 4, 8, 32), loop interchanges (the order loop iterators appear in nested loops), and memory management (the packing used for tile memory structures). Each evaluation of the objective requires annotating the source with parameter values, then compiling and executing it on the benchmark system to collect timing data, which incurs considerable cost even for small input matrices. There are 376,320 unique combinations of the ten parameters that define our tuning space for 3mm, which is prohibitively costly to brute-force with empirical searches. Autotuning uses more intelligent approaches to identify the configurations that achieve optimal performance.

Autotuning must differentiate input scales as different tasks because changing the input scale frequently induces drastic changes in the optimum configuration. As shown in Table 2.1, small sizes require the packed-array technique for matrices A and E, but medium-sized inputs do not. The degree of improvement can also vary between input scales, where small 3mm inputs can gain 1.13× speedup from autotuning. However, medium-sized 3mm inputs gain 14.94× speedup over the respective baselines.

2.3.1 Transfer Learning in Autotuning

Several search methods have been developed to reduce the number of evaluations required to find the best configuration for autotuning tasks. They can be classified into model-based and model-free methods. The former methods learn the relationship between the parameter configurations and the objective function through an incrementally updated surrogate model and leverage it to cheaply evaluate multiple points and minimize the number of actual evaluations. Examples include Bayesian

Table 2.1: Matrix input scales affect speedup and the best configurations for the 3MM kernel.

	Input Scale		
	Small	Medium	Large
Input Scale Characteristics			
Array Dimensions	≤ 80	≤ 220	≤ 1200
Naive Tera-Ops	0.037	4.75	2924.24
Worst Runtime (s)	0.00017	0.1096	9.8631
Best Configuration Values			
Packed Arrays	A,E,F	F	A,B,E
Loop Interchanges	N/A	N/A	Outer Exchange
Tile Sizes	16, 2048, 4	96, 16, 4	4, 2048, 4
Speedup Over Default	1.13×	14.94×	50.50×

optimization that employs Gaussian process regression and random forest and their variants. The latter methods optimize the objective function without such models. Examples include random search, grid search, genetic algorithms, and Nelder-–Mead. The key advantage of the model-based methods is that they require significantly fewer evaluations than the model-free methods, especially for large search spaces [12, 13, 15, 16].

TL in autotuning is an emerging approach that leverages data from one autotuning task in related autotuning tasks to improve sample efficiency significantly. Related autotuning tasks are common in HPC applications, which include tuning different input sizes of the same kernel or application, tuning the same kernel across architectures, and tuning related kernels with the same computational signature. While the best configurations are often different for different autotuning tasks, TL is particularly effective when the related tasks share similar high-performing characteristics in the search space. Model-based search methods are promising for TL because the model can be pretrained or bootstrapped with the existing data from related tasks.

Chapter 3

Impacts of Immersion Cooling on HPC Applications

3.1 Introduction

Hardware innovations have driven computing performance for decades, but the decline of Moore's Law and limitations of Dennard Scaling have lead to many new and exciting hardware developments beyond merely scaling transistor counts. The densest computing components brush against the limits of Dennard Scaling and are vulnerable to thermal damage that can limit chip performance or terminate it entirely. As such, the importance of cooling hardware as a means of supporting high-energy and high-throughput workloads has received much greater attention recently.

Many large-scale computer systems are transitioning or considering transitioning from air-cooling solutions to liquid-cooling solutions. Thermodynamic physics dictate that newer, more dense computing components will produce more thermal waste than air as a medium can physically circulate – our computers will melt down or have to operate in degraded performance simply because they're too hot. As a denser physical medium, liquids can potentially improve over air in terms of cooling efficiency and costs while possibly also improving computing hardware reliability. Single Phase Liquid Immersion Cooling (SPLIC) is such a technology in which computing hardware is

Portions of this chapter are based on work that was originally published in the Proceedings of the International Green and Sustainable Computing Conference 2024 under the title "Thermal Behaviors in Liquid Immersion Cooling under Various Workloads: a Case Study".

completely submerged in a container filled with a dielectric coolant fluid. Compared to air, the coolant medium is denser and has higher thermal conductivity and heat capacity. These properties improve heat dissipation and permit more efficient heat recycling. Many benefits of SPLIC are well-understood in theory, but the practical effects of the technology on real software workloads have not been well-studied academically.

Previous studies generally examine the raw heat circulation properties of the coolant without considerations for particular applications or hardware configurations. However, different hardware components possess different power requirements and ranges and patterns of heat dissipation based on the software they are executing. For example, processing units consume more power and have a larger power range compared to memory devices and disk drives. Component power is also determined by the underlying hardware technologies; for instance, CPUs are generally less power efficient than GPUs, and DDR memory devices are less power efficient than high-bandwidth memory (HBM) devices for the same capacity. Finally, applications are the ultimate drivers of component utilization, meaning that different kinds of applications produce different cooling requirements from the same system. This means that the actual cooling demand of computing systems is highly variable across different hardware and software configurations.

Cooling's role in compute systems is to prevent damage to computing hardware and to reduce or eliminate measures individual components utilize to prohibit damage, such as downclocking, which ultimately cause performance losses. While most software applications operate blissfully unaware of the thermal conditions of computing hardware, intense and long-running processes are well-known to lead to performance degradation and faults that reduce or deny access to peak computing potential. The real cooling demands required by a system for sustained performance are neither represented by idle conditions nor the maximum draw from running all components at full capacity at all times. Therefore a static understanding of peak heat dissipation capability does not paint a reasonable expectation of how cooling may affect system performance.

In this chapter, we conduct a case study on thermal behaviors in a SPLIC tank with a immersed compute cluster. We utilize several CPU-based and GPU-based High Performance Computing (HPC) and datacenter applications with different compute and memory intensities and analyze hardware thermal behavior and their variations with the SPLIC system configurations and operations. We also compare the performance on highly similar hardware configurations in a traditional air-cooled datacenter rack. Unlike prior works that rely on simulations, our case study provides em-

pirical insights on thermal demands induced by applications and their effects within the immersed environment.

These experiments demonstrate the tradeoffs between thermal capacity and thermal responsiveness in SPLIC systems, the efficiencies of thermal dissipation, as well as the challenges introduced by modern applications on these systems.

3.2 Related Work

Many design decisions impact the performance of SPLIC, including the number of pumps, the pump arrangements and settings within the tank, and the choice of dielectric fluid, which have all been studied by many works [3, 17, 18], primarily via simulation or isolated experiments on individual components. Existing work has addressed several initial concerns about the technology, including the extent of dangers to hardware components from absorbing dielectric fluids [19] and the capital burden of procuring, installing and maintaining SPLIC rather than other more traditional cooling systems [20]. However, the existing literature largely lacks evaluations of SPLIC hardware in meeting actual application cooling demands, which we aim to address in this work.

3.3 Research Questions

We investigate specific questions with real application performance in an SPLIC environment to provide a more comprehensive perspective on the larger subject.

3.3.1 RQ1: What are the thermal behaviors of individual hardware components in SPLIC?

Heat accumulates in the pod as power is drawn and used by individual computing components. The heat accumulated at a particular component generally increases as that component utilizes more of its peak capability and when high utilization is sustained for extended durations. Most commodity hardware is technically compatible with SPLIC, but have thermal designs that do not explicitly consider properties of ambient liquid coolant. For instance, processors and accelerators typically comprises physical parts such as memory caches and processing cores accommodating both memory access and compute. Such integration improves performance but has a high thermal

conductivity and heat exchange within components. SPLIC coolant, if not actively circulated all the time, is unable to reach thermal equilibrium or transfer heat as fast as fan cooling between components in close physical proximity.

It is important to understand if high core temperatures dissipate more heat to proximal components such as memory, particularly for GPUs. Furthermore, the environment's tendency for equilibrium may result in certain hardware components operating at higher temperatures than would be typical given their power design, necessitating cooling interventions for components that may not have DVFS or thermal throttling capabilities.

We observe the actual thermal operating range and rate of temperature change for key hardware components before, during, and after prolonged execution of various applications. These characteristics can be compared relative to the range of temperatures and rate of temperature change in the tank's coolant to determine an appropriate response and tolerance within the environment. We also observe the rate of change in hardware hotspot temperatures and coolant temperature to determine if the indirect thermal measurement performed by the cooling system can appropriately respond to high demand from the computing components. We present per-component analyses and coolant-relative per-component analyses in Section 3.5.1.

3.3.2 RQ2: How do initial conditions affect SPLIC performance and efficiency?

Some liquid cooling techniques, such as DLC, yield higher heat dissipation rate as the coolant temperature rises, which is a very convenient upside. We aim to determine whether similar benefits can be observed in SPLIC by correlating heat dissipation with the initial temperature during cooling periods, ensuring that workloads do not interfere with the measurements. We also correlate various component activity measures with temperature dissipation during cooling cycles when applications are running to determine if different workloads exhibit distinct thermal patterns beyond the rate of heat introduction into the environment. We analyze these trends in Section 3.5.2.

3.3.3 RQ3: How do different workloads affect heat accumulation?

Exhaustive studies of application behavior are impractical. In this work, we use a selection of workloads to represent HPC and datacenter applications with similar primary resource demands

and study the resulting thermal challenges for the cooling system.

In particular, we consider applications where the primary device used for computation is CPU or GPU and if the computation is subject to a bottleneck in compute or memory throughput. Due to the small number of servers, we exclude network throughput as a bottleneck in our study, though we recognize its importance for large-scale distributed system and workloads. We present a correlative analysis between workload classification and induced cooling demand in Section 3.5.3.

3.3.4 RQ4: What differences are observed compared to air-cooled environments?

Finally, it is important to understand the experienced differences between air- and liquid-cooled environments. We focus on the differences in response times and the rate of heat accumulation in Section 3.5.4.

3.4 Experiment Design

We answer the research questions by conducting a set of experiments and analyses as a representative case study. Our experimental platform consists of an SPLIC environment with immersed servers, a suite of applications with various thermal footprints, and tools to monitor sensors and performance.

3.4.1 SPLIC and Computing Hardware

SPLIC Environment

Our SPLIC environment is visualized in Figure 3.1. We use a Submer SmartPod v3, which utilizes a proprietary synthetic dielectric fluid to cool all computing hardware within the tank's volume. The tank (also referred to as a cooling pod or "pod") uses two redundant pumps to circulate coolant. The pod exchanges heat with facility-chilled water, which flows through tubes at the bottom. The temperature of the in-flow chilled water is measured to be inclusively between 10 to 12 degrees Celsius at all times.

We are limited to only a few servers available for immersion, which cannot generate enough heat in a short period for our studies if the pod is always connected to the chilled water. To address

this limitation, we conduct thermal analyses while the chilled water is disconnected and continue after reconnecting chilled water to observe the system's capability to respond to strong cooling demand. This approach allows us to study the SPLIC system's response to high-heat environments without excessively stressing the system for prolonged periods. It also allows us to better understand the capacity of the coolant medium itself in our experiments, which could be overly-protected by the manufacturer's minimum settings that require circulation every half hour for at least five minutes. Each of our experiments maintain high levels of compute activity on a single application without access to chilled water for an extended duration before halting the application and re-engaging the chilled water, permitting the system to return to its normal state.

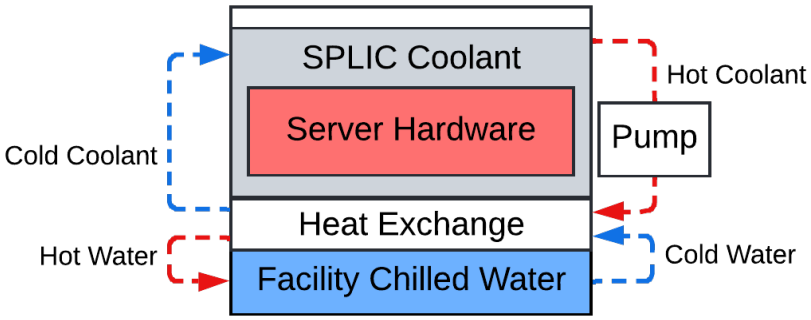


Figure 3.1: SPLIC tank arrangement used in our experiments.

Computing Hardware

We immerse three server nodes into the SPLIC pod. The nodes are composed of commodity hardware and colloquially referred to as "deepgreen," "n01," and "n02", as detailed in Table 3.1. The deepgreen node is the head node directing all application flows while n01 and n02 are compute nodes that provide additional compute capability to scale application performance.

In this work, we are limited to commodity servers, which generalize across various possible server configurations. We also leverage this to be as reasonably comparable to current air-cooled hardware configurations later in our work; our air-cooled hardware is detailed in Table 3.2.

Table 3.1: Immersed server hardware used in evaluations.

Nodes	Kind	Hardware	Specs
Deepgreen	Mother-board	S8021GM2NR-2T	Five (5) PCIe 3.0 x16 slots
	CPU	AMD EPYC 7551P	2.00 GHz 32 cores
	GPU	Two (2) NVIDIA Titan V	5120 cores 12 GB HBM2
n01, n02	Mother-board	X9DRG-QF	Four (4) PCIe 3.0 x16 slots
	CPU	Intel(R) Xeon(R) CPU E5-2670 v3	2.30 GHz 64 cores

Table 3.2: Air-cooled server hardware used in evaluations.

Nodes	Kind	Hardware	Specs
Palmetto S12	Mother-board	DELL 0D9WDC	Four (4) PCIe 3.0 x16 slots
	CPU	Intel(R) Xeon(R) E5-2680 v4	2.40 GHz 28 cores
	GPU	Two (2) NVIDIA P100	3584 cores 12 GB HBM2

Hardware Preparation for Immersion

The SPLIC coolant fluid includes plasticizer, so we replace thermal paste between hardware components and heat sinks with indium foil. The plasticizer also makes ethernet and power cables rigid and somewhat brittle after a few months. However, these cables remain at negligible risk of breaking while undisturbed.

We remove or disable all moving hardware on power supplies, motherboards, GPUs and CPUs. Server components designed to physically move in an air-cooling environment are unsuitable for immersion cooling. Take fans, for example; they would generate enormous heat from friction with the dense synthetic fluid, triggering alarms and risking burning [1]. Specialized hardware "immersion ready" or designed to improve SPLIC performance circumvents these issues.

3.4.2 Applications

We utilize a selection of HPC applications with different compute intensities, as shown in Table 3.3.

Table 3.3: Selected applications and classifications.

Application Name	GPU Accelerated	Throughput Bottleneck
CUDA Stream	✓	Memory
EMOGI	✓	Memory
CUDA DGEMM	✓	Compute
MD5 Bruteforcer	✓	Compute
NPB DT Class=C	X	Memory
NPB IS Class=D	X	Memory
NPB EP Class=E	X	Compute
HPCC HPL	X	Compute

GPU Applications

We select CUDA Stream as a GPU-enabled memory bandwidth test with minimal computation. For a more realistic representation of memory-intensive GPU-accelerated HPC applications, we utilize a manually constructed graph using the EMOGI graph tool [21]. This graph is designed to

[1] We indeed experienced this once.

induce a memory-antagonistic BFS traversal when executed via EMOGI, permitting minimal data reuse while extending data lifetimes as long as possible per the algorithm's implementation.

We also include compute-intensive GPU applications, including CUDA DGEMM, a common compute-intensive kernel utilized within many scientific and machine learning workloads. While machine-learning implementations may prefer lower precision datatypes than doubles (the "D" in DGEMM), we note that many scientific applications have not embraced lower precisions and that most of the computational instructions for lower precisions practically resolve into multiple words being loaded into the same double-precision pipeline for execution. We also provide the MD5 Brute-forcing algorithm as an example of datacenter workloads. This application's repeated hash computations permit long arithmetic instruction sequences without dependence upon large-scale memory accesses or highly variable control flows.

CPU Applications

We utilize several key kernels from the NAS Parallel Benchmark Suite [22], including **D**ata **T**raffic, **I**nteger **S**ort and **E**mbarrassingly **P**arallel. Each of these benchmarks are executed at the largest class size that can be executed while utilizing the entire server system. As these classes are designed based on the number of available MPI ranks, we note that the air-cooled replication occasionally utilizes smaller classes than those executed via the immersed server.

For memory-bottlenecked CPU applications, we focus on Data Traffic and Integer Sort. The Data Traffic benchmark measures the communication performance between cores and over interconnects in the parallel environment, demonstrating the memory latency and bandwidth constraints on the system. The Integer Sort benchmark produces a large, in-memory list of integers that are sorted in parallel with coordination between processors.

For compute-bottlenecked CPU applications, we utilize the NPB Embarrassingly Parallel benchmark and HPCC HPL benchmark. The Embarrassingly Parallel benchmark application represents highly parallel compute saturation with minimal coordination between parallel components and interconnections. Similarly, the High Performance Computing Challenge (HPCC) High Performance Linpack (HPL) represents a solver for a random dense linear system in double precision arithmetic [23].

3.4.3 Metrics and Monitoring Tools

We utilize multiple publicly available software tools to collect metrics on CPU, GPU and NVMe device thermal sensors and activity measures, as well as a vendor-provided API to monitor the SPLIC system. We monitor the power of the SPLIC system over SNMP interfaces to the PDUs that all components draw power from, and similarly monitor power usage of the air-cooled server via vendor-provided reporting tools. To minimize the impact of monitoring on application performance, our tool is designed to sample each metric at a target rate of 1Hz and caches some information to reduce the overhead of sample collection. Our software harness is publicly available via GitHub at `https://github.com/tlranda/LibSensorsTools`.

CPU Monitoring

We record per-core frequencies from the `cpufreq/scaling_cur_freq` files in the Linux `/sys/devices/system/cpu/cpu*/` directories. These frequencies are recorded by the CPU governor in these files as integer-valued KHz.

We also use the libsensors library provided by lm-sensors (version 3.6.0) to monitor CPU core temperatures. The temperature values are reported in tenths of degrees Celsius.

In the air-cooled server, we also record CPU power draw via RAPL utilities to better represent the uncore power devoted to cooling the system, which are reported in micro-watt precision.

GPU Monitoring

Within the SPLIC environment, we record GPU metrics listed in Table 3.4 using the NVML library based on NVIDIA driver version 535.54.03 and NVML version 12020, corresponding to CUDA version 12.2. Each metric is represented using integer values, so temperature data are reported as whole degrees Celsius. We record the same metrics in the air-cooled environment, but note that the P100 GPU architecture does not report memory temperatures, so we do not analyze this component between the air- and SPLIC- cooled environments. The air-cooled environment also runs slightly different software versions, notably NVIDIA driver version 550.54.15 with NVML version 12040 corresponding to CUDA version 12.0. These software versioning differences do not affect our metric recording outside of the previously noted exception for memory temperature reporting.

Table 3.4: NVIDIA GPU metrics recorded using NVML.

Metric	Meaning (Units)
GPU Temperature	Average SM temperature (°C)
Memory Temperature	Memory junction temperature (°C)
Power Usage	Power draw (mW)
Power Limit	Maximum power draw (mW)
GPU Utilization	NVIDIA metric of SM activity (%)
Memory Utilization	NVIDIA metric of memory activity (%)
Memory Used	Allocated global memory (bytes)
Performance State	NVIDIA metric of device state (integer 0-8)

NVMe Monitoring

We record NVMe temperatures using libnvme version 1.6, which are also reported as integer degrees Celsius.

Power Monitoring

We record single-precision PDU phase amperage to determine the complete system's power draw over an SNMP interface. The Rack PDUs for the immersed system are Schneider Electric model AP7811B, with a 30 Amp limit and 208 Volt output of single-phase AC current, so the conversion to Watts is straightforward.

We also collect power consumption within the rack-level of the air-cooled server by the Dell OMReport [24] tool, which also provides coarse power measurements of the instantaneous wattage.

SPLIC Tank Monitoring

We use a vendor-provided API endpoint to collect metrics listed in Table 3.5 using LibCurl version 7.68.0. The API reports temperatures as tenths of a degree Celsius and most other values as integers.

3.4.4 Experimental Procedure

We conduct independent tests utilizing a single application as the workload for the server for the entire observed duration. Each test begins with thirty minutes of idling activity while disconnected from chilled water to establish a baseline for experimental conditions. After this initial period, we repeatedly execute the workload application for 7.5 hours, after which we reconnect the

Table 3.5: Submer SmartPod metrics recorded from vendor-provided API.

Metric	Meaning (Units)
Temperature	Average coolant temperature (°C)
Consumption	Pod power consumption (W)
Dissipation	Thermal dissipation (both as °C and KW)
mPUE	Power Usage Effectiveness (scalar)
Pump RPM	Pump rotations per minute (scalar)
Coolant Temperatures	Input/output coolant temperature (°C)
Water Temperatures	Input/output chilled water temperature (°C)
Flow Rates	Flow rate of coolant and water (L/minute)

chilled water supply to the pod. We continue to monitor temperatures for an additional 24 hours as the pod dissipates accumulated heat and returns to its equilibrium state.

Due to potential harms to the air-cooled hardware, we do not disable any cooling components for our replicated experiments on these servers. We perform the same pre- and post- application idling periods to maintain as similar of conditions as possible between environments, but the cooling components are unaltered throughout our entire duration of monitoring. We also shorten the post-application idling period relative to the immersed server simply due to less time being required for air-cooling to return the system to baseline conditions, which are shown in our results.

3.5 Experimental Results

We report our results based upon the research questions posed in Section 3.3. The first three research questions are exclusive to the immersed environment, with the fourth question also including data from the air-cooled environment.

3.5.1 Thermal Behaviors of Individual Hardware Components (RQ1)

Importance. The range of measured temperatures of each hardware component determines the risk of violating thermal tolerances and how quickly temperatures may be expected to change. We also seek to identify variations in temperature based upon the application demand to determine if thermal behavior is purely driven by energy expenditure or if more complicated behaviors emerge.

Metrics. We begin by analyzing the minimum temperature as a small range of values collected during the idle baseline prior to each application test. As initial conditions are similar

across all experiments, we aggregate all data during these periods to establish the minimum, mean and maximum observed temperatures of each component.

We then consider the recorded temperatures across each experiment's active application periods to determine the greatest attained temperature over the baseline and the greatest linear rate of temperature increase throughout all experiments. We compare the dissipated coolant heat relative to hardware activity measures to determine if more complex modeling is needed than pure power demand. The hardware activity measures and collection techniques are defined in Section 3.4.3.

Results. Each of the above metrics are computed for all monitored hardware and presented in Table 3.6. All hardware components except the NVMe are capable of changing temperatures faster than the pod coolant, however only the GPU hardware was observed to reach levels where thermal throttling could become a reasonable concern when executing the DGEMM application, where it reached a maximum temperature of 96 °C. The Titan V GPU models are designed for a maximum operating temperature of 91 °C, with thresholds for slowdown and shutdown at 97 and 100 °C, respectively. Had the experiment continued beyond the eight allotted hours, the disconnected chilled water would not permit adequate heat exchange for the SPLIC system to protect immersed hardware and could have permanently damaged the GPUs. This represents a somewhat reasonable upper bound for our experimental conditions, however we note that our pod has additional coolant volume as not all compute racks are filled. A pod with all racks completely filled will both have more power draw within the volume and a lower ratio of coolant to heat-producing mass. This may require a faster response if chilled water becomes inaccessible to preserve the integrity of the system.

Notably, GPU temperatures fell to 46 °C almost immediately upon terminating the DGEMM application for this experiment, and a repeated execution of the benchmark without interrupted access to chilled water failed to exceed a GPU temperature of 80 °C. With the re-introduction of chilled water for heat exchange, the SPLIC system was able to return all temperatures to normal idling levels over the next 9 hours. When comparing the drop in temperatures at the moment the applications end, we note that the increase in coolant temperature is nearly one-to-one with the excess heat retained by hardware components, delaying a return to normal idling conditions until the entire fluid volume is cooled. Our results cannot generalize across all vendor products and configurations, but we believe this forms one of the most reliable stress-tests to calibrate emergency response thresholds. Figures 3.2 and 3.3 show detailed temperatures of this experiment in without chilled water and with uninterrupted access to chilled water.

Table 3.6: Thermal behaviors within SPLIC system by hardware component.

Hardware Component	Idle Temperature Minimum/Mean /Maximum (°C)	Interval (s) to Increase Temp by 1°C	Maximum Observed Operating Temperature (°C)
SPLIC Coolant	19.80 / 21.41 / 22.50	970.99	47.10
deepgreen CPU	14.75 / 17.51 / 19.66	702.20	55.62
deepgreen GPU	24.25 / 26.54 / 28.00	266.83	96.00
deepgreen NVMe	14.00 / 17.91 / 20.00	1068.48	44.00
n01 CPU	17.44 / 21.30 / 24.04	607.33	66.00
n02 CPU	17.67 / 20.81 / 24.26	560.35	69.00

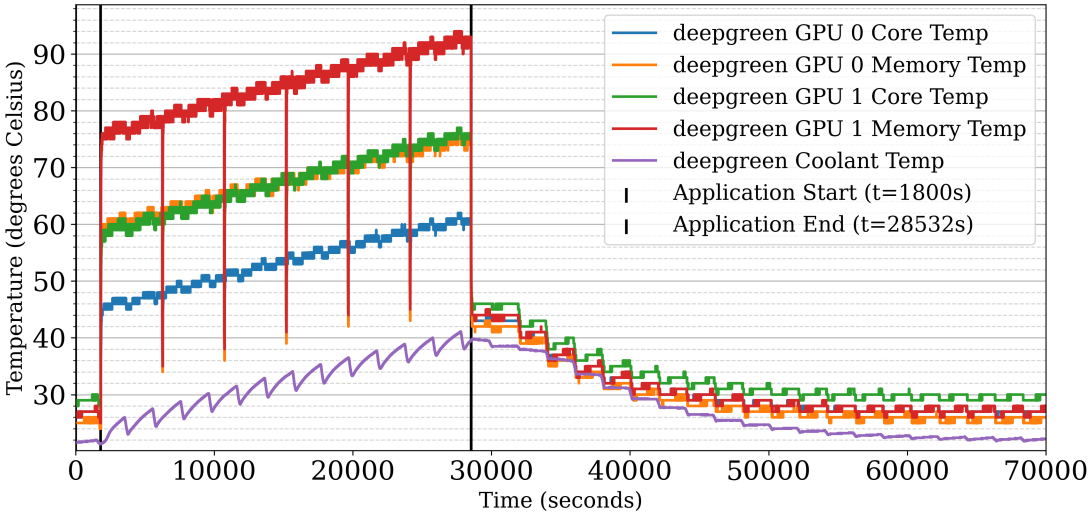


Figure 3.2: Thermal Behavior of DGEMM Without Chilled Water. DGEMM runs on GPU 1.

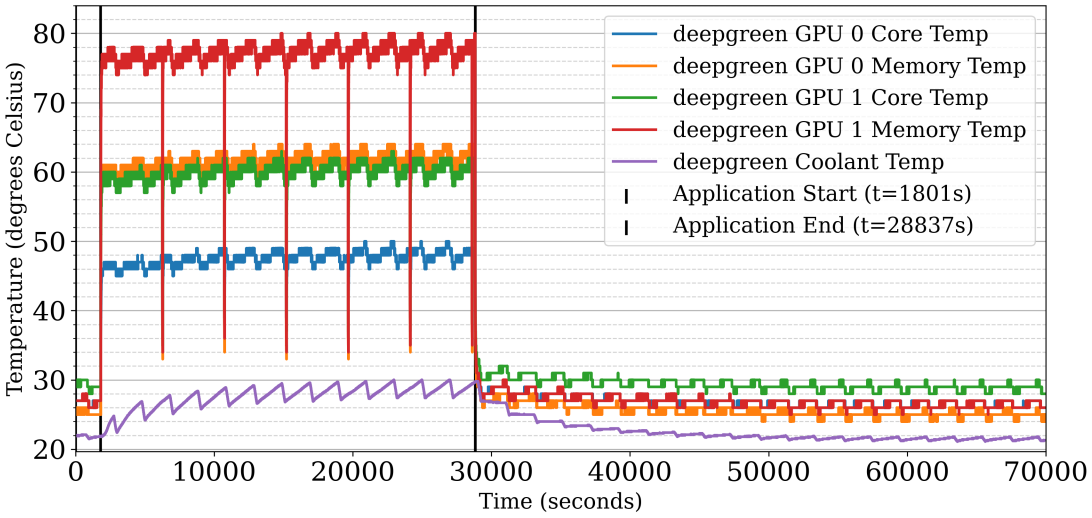


Figure 3.3: Thermal Behavior of DGEMM With Chilled Water. DGEMM runs on GPU 1.

3.5.2 Effects of Initial Conditions (RQ2)

Importance. SPLIC's energy efficiency may correlate with fluid temperature, making a trade-off between maximum thermal tolerance and long-term energy expenditure.

Metrics. We monitor the dissipated heat during periods where pumps are active and chilled water is available for the SPLIC heat exchange.

Results. We present the aggregated data across all observations in Figure 3.4. It's clear that the maximum heat removed positively correlates with a higher initial temperature, however the minimum heat removed during a cycle remains roughly constant regardless of initial temperature. Because the pump activity duration, chilled water temperature and pump power draw are held invariant in our experiments, this implies that SPLIC can yield increased energy efficiency at higher temperatures, however the behavior is not guaranteed to be observed.

3.5.3 Effects of Workloads on Heat Accumulation (RQ3)

Importance. Given that almost all hardware components can heat up faster than the average coolant temperature, it's important to know what workloads will likely produce the greatest heat accumulation within the pod. This simplifies benchmarking a system's cooling demands and ensures system responses can be properly calibrated for extended application executions.

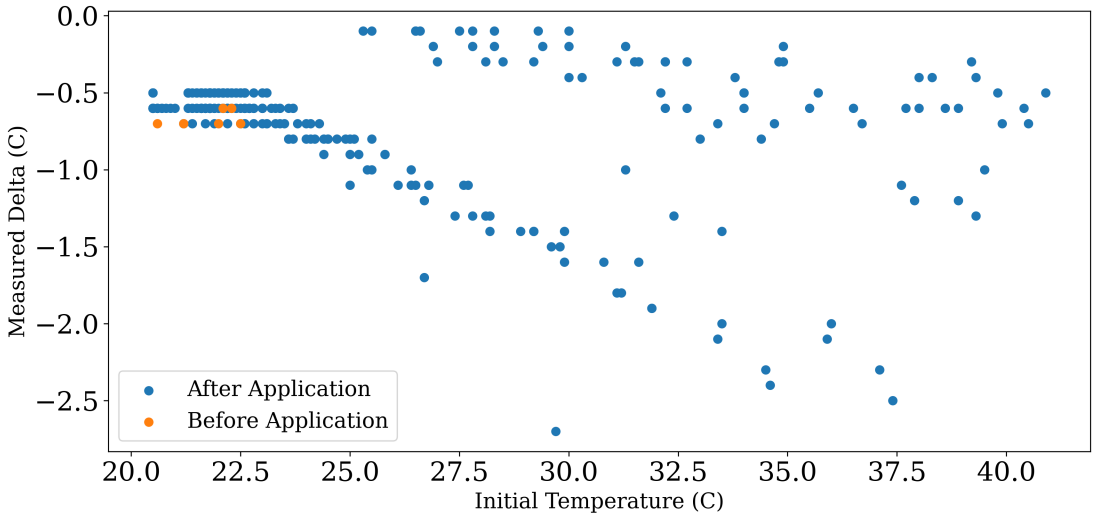


Figure 3.4: Temperature at start of idle pump cycle vs amount of heat removed.

Metrics. We use the average interval to increase the temperature by one degree Celsius when the temperature delta reaches the maximum during application execution. To simplify comparisons, we group applications based upon CPU- or GPU-centric classifications and application throughput being Compute- or Memory-bound as previously denoted in Table 3.3.

Results. We display the interval of each experiment application in Figure 3.5. The lower values indicate faster heat accumulation. These applications present a good range of heat accumulation rates. In contrast to common impression that GPU applications accumulate heat faster, only compute-intensive GPU applications do, while memory-bound GPU applications yield relatively slow temperature changes. In general, compute-bound applications accumulate heat faster than memory bound applications, no matter whether they are programmed to run on CPU or GPU.

3.5.4 Comparison to Air-Cooled Environment (RQ4)

Importance. To fully understand the current state of this SPLIC technology, a comparison to current air-cooling is warranted. Air-cooling includes fans within hardware components, the chassis, and the rear door heat exchange on the server closet. In particular, we want to detail the differences between the responsiveness of the cooling systems and the rate of heat accumulation within the system to determine differences in thermal management strategies. Additionally, we attempt to monitor power usage conditions to estimate the energy usage of the cooling systems

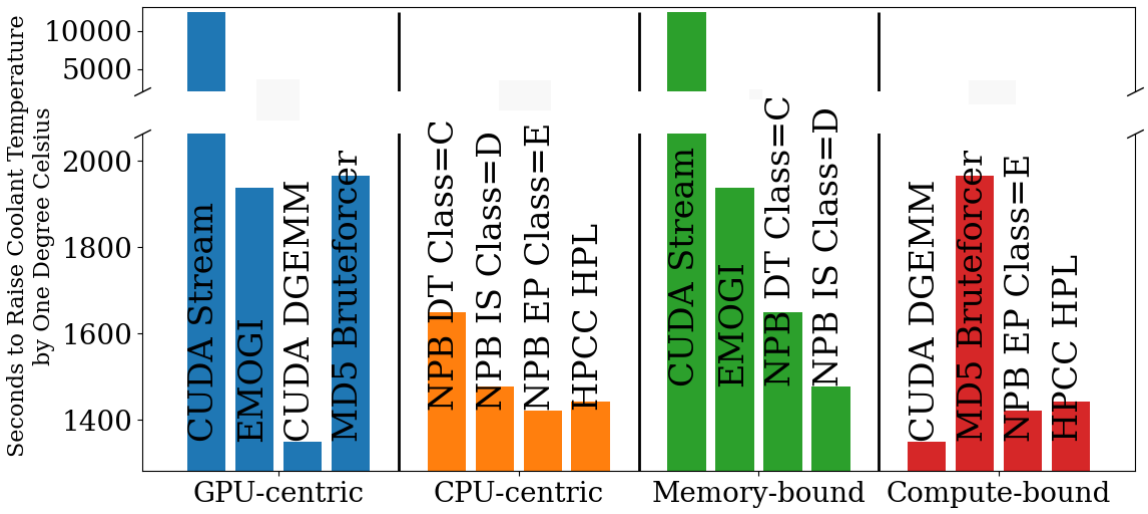


Figure 3.5: Time intervals required by applications to accumulate heat. The smaller the interval, the faster the application execution increases coolant temperature by 1° Celsius.

themselves.

Metrics. We measure heat accumulation and responsiveness in the air-cooled environment following the same methodology from Section 3.5.1. Unlike the immersion-cooled experiments, we cannot disable the air cooled fans due to safety protocols, however all other conditions remain the same.

For power measurements in the air-cooled server, we rely upon Dell's OMReport utility for node-level power, which differs from the SNMP PDU power readings we utilize for the immersed servers as the latter reports overall power including possibly idle components (i.e., unused servers in GPU experiments). To account for this and slight hardware variations between the two setups, we report power differentials over the pre-experiment idle baseline to ensure fairer comparisons between the systems. To ensure the energy measurements are more comparable between the two environments, we repeat all SPLIC experiments without interrupting access to cooling, ensuring that the system operates under its expected conditions. we also subtract the power draws of components that report their own power usage, primarily the GPUs themselves, to further isolate the overall power draw due to cooling demand. Notably, we find no statistically significant change in application runtimes throughout our experiments in either environment, so less total power draw directly correlates with less cooling energy demand from the system to perform a given task.

Results. Unlike our experiments with the immersion cooling tank, we cannot measure the

coolant fluid (air) for the dry server environment in isolation. This makes Figure 3.3 the nearest comparison to Figure 3.6 where we present the same DGEMM experimental benchmark studied on the air-cooled system. Despite highly similar idling temperatures, the air-cooled server sees slightly increased GPU core temperatures relative to the immersion-cooled server while under load. Even with access to chilled water, a gradual increase in the component temperatures was observed in the SPLIC environment. In the air-cooled environment, this is not the case, with temperatures holding steady over the entire experiment duration due to constant cooling conditions. However, the component temperatures reveal the limited thermal capacity of air. For instance, the CPUs in the air cooled environment accumulate heat 1.3X faster than the immersed CPUs despite the active cooling conditions. The air-cooled GPUs have fans deliberately positioned to prevent them from overheating, and therefore their internal temperatures rise slower than the immersed GPUs by a factor of 1.2X. The SPLIC coolant fluid removes heat more efficiently from components while it has a temperature differential between components, and excess coolant volume grants higher heat capacity in the immersed environment.

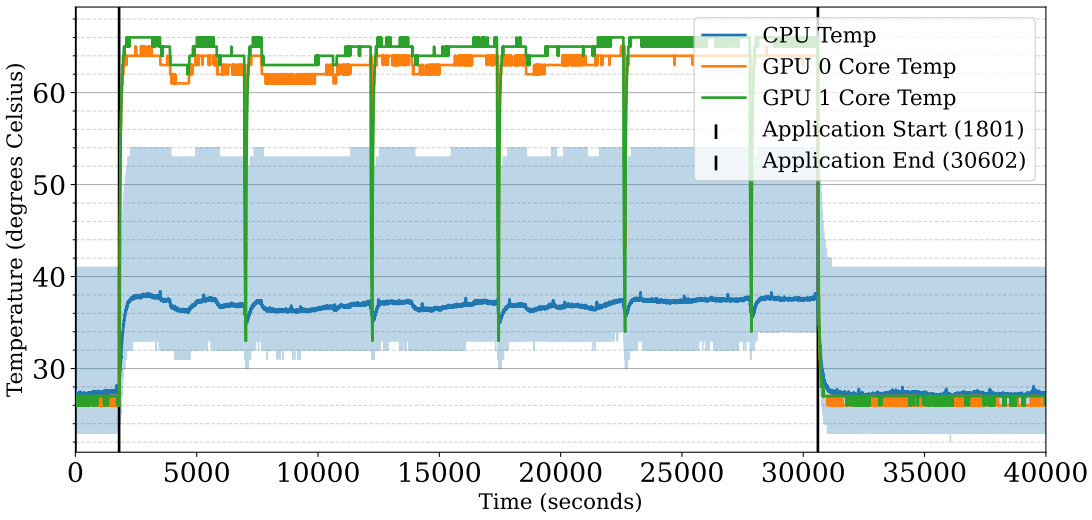


Figure 3.6: The air-cooled server has similar idle and active temperatures as the immersion-cooled server, however there is no gradual heat accumulation and temperatures return to nominal idle levels much faster. The lighter area for CPU temperature denotes the highest and lowest observed core temperatures while the line represents the average across all CPU cores.

We also observe that the air-cooled components return to nominal idle temperatures faster once the application period terminates, needing only 4 minutes compared to over 3 hours for the

immersed hardware. Despite the lower capacity and thermal differential between coolant mediums (the circulated air is roughly 11 C warmer than the chilled water), there is a faster rate of coolant flow and normalization in air-cooling. Our results without access to chilled water during the application period represent an exaggerated version of this effect, where the GPUs require 6.25 hours to return to nominal idle temperatures and the overall coolant volume requires an additional 9 hours (a total of 15 hours after ceasing compute activity) to fully dissipate the heat. The SPLIC system is only designed to proactively remove heat beyond a set point, but these results demonstrate how thermal activity within the system can have a long-lasting lifetime due to the relaxed cooling responses and the recycled coolant and small proportion of coolant volume that passes through the heat exchange. By contrast, the air-cooled server returns to nominal temperatures faster due to its constant replacement of air as coolant fluid.

Hardware Component	**Idle Temperature Minimum/Mean /Maximum (°C)**	**Interval (s) to Increase Temp by 1°C**	**Maximum Observed Operating Temperature (°C)**
node0091 CPU	27.06 / 27.32 / 28.10	475.86	54.00
node0091 GPU	26.00 / 26.49 / 27.00	309.89	66.00
node0048 CPU	25.32 / 25.66 / 26.26	1325.07	45.13
node0048 GPU (idle)	25.00 / 25.50 / 26.00	315.08	27.00

Table 3.7: Thermal behaviors within air-cooled system by hardware component. The node0048 GPUs are not used to mirror the conditions of the SPLIC environment.

Compared to Table 3.6, the air-cooled results in Table 3.7 have higher CPU idle temperatures but similar idle GPU idle temperatures. The maximum observed temperatures of individual components are lower than those measured in the immersed environment. This is largely due to the heated air being pushed out of the system rather than lingering around the hardware as it increases in temperature, which limits opportunities for accumulation and heat-sharing between components.

Finally, we estimate the energy devoted to cooling between the two environments under normal operational conditions. For the SPLIC environment, the power draw for the tank itself is directly provided, allowing us to directly measure energy utilized for circulating coolant via the pumps. In the air-cooled environment, we attempt to isolate the uncore power of the system minus a reasonable baseline, then estimate the fan power as a percentage of the remaining uncore power.

Hardware	Type	Application	SPLIC Pump Energy Estimate (Kilo-Joules)	Air-Cooled Fan Energy Estimate (Kilo-Joules)
GPU	Memory	CUDA Stream	21.1929	5.8531
		EMOGI	20.5241 *	–
	Compute	CUDA DGEMM	18.4060	22.5619
		MD5 Bruteforcer	29.9982	19.4239
CPU	Memory	NPB DT	29.8733 *	13.6337
		NPB IS	27.9306 *	–
	Compute	NPB EP	28.6444 *	35.1672
		HPCC HPL	19.2698	0.0537

Table 3.8: Estimated energy utilization for all cooling required for each benchmark's workload until temperature normalization. Fan energy estimation is based on 2% of the above-idle unaccounted uncore power of the air-cooled system. (*) denotes values that lacked chilled water during application period; these will be updated for the final draft.

Table 3.8 shows the energy estimations for cooling across our experiments. We repeat the experiments with uninterrupted access to chilled water for the SPLIC environment, as the periodic pump cycles dissipate heat ineffectively while the application is on, but still consume energy. Notably, when comparing DGEMM results with and without access to chilled water as the application runs, the total cooling demand is increased from 20 kJ to 31 kJ. The amount of heat introduced in these scenarios are identical, so a decrease in the energy required to normalize temperatures is representative of more efficient cooling conditions. For lower-heat applications such as CUDA Stream, the SPLIC pumps expend half of their cooling energy while the application is active, which is still double the estimated energy used by fans, resulting in large excesses of power demand from the cooling system. For higher-heat applications such as DGEMM, our estimates indicate less energy used by the SPLIC system, suggesting greater dissipation efficiency than air-cooling despite a similarly prolonged duration for temperatures to normalize. This suggests a nuanced need for scheduling and controlling pump activity not only to prevent hotspots from adversely affecting hardware, but also to optimize the energy efficiency of the system with respect to the thermal loads within the system.

Under perfectly replicable conditions the energy to be removed from the system would be identical between SPLIC and air-cooled environments, which should largely require the same energy expenditure to remove heat from the system. However, the energy required to dissipate heat is not guaranteed to be the same. For instance, in Shah et al [19], a white mineral oil is measured to have 6.5× more thermal conductivity and 1.6× more thermal capacity compared to air, but at the cost of 100× more viscosity and 693× the fluid density, meaning that the energy efficiency per unit of

heat dissipation is not strictly one-to-one between these mediums. The primary importance of our results is to further validate that similar energy is used by both systems for actual heat dissipation, and the practical difference between the systems lies in the duration over which cooling activity occurs. Estimating fan power using a different percentage of uncore power can increase or decrease the energy estimation of air-cooling estimates proportionally, however since both environments also permit untracked ambient heat dissipation we do not belabor the point. A more rigorous experiment would be required to track affects such as heat naturally radiating from the air-cooling fins and motherboard surface without fan circulation or heat that dissipates through the SPLIC tank walls.

3.6 Conclusions

Our experiments reveal a few initial insights about the current state of SPLIC immersion technology.

Strong resilience to thermal changes. Current tank designs use temperature readings at coolant pumps, instead of relying on embedded sensors in hotspots and computer components, to respond to thermal changes. This is a fixed strategy. As evidenced by our experiments, the technology can handle fluctuations in computing demand for extended durations. Despite the thermodynamic advantage of liquid fluid over air as a coolant medium, our experiments provide evidence that current SPLIC technology does not dissipate heat as quickly as traditional air-based cooling systems. The limited volume of SPLIC fluid that can perform heat dissipation via the exchange with chilled water also reduces the rate of change in overall component temperatures, which may require special considerations.

Delayed responses to temperature differentials. Our study shows that hot spots within hardware components have significant lag time before pod sensors can detect changes in temperature, meaning that responses must be calibrated to be more aggressive in inducing a cooling response in case hardware conditions outpace the system's detection. Especially compared to air-based cooling responses, SPLIC systems' focus on energy-efficient cooling via indirect means can result in certain components overheating before the system is able to initiate a response. Future iterations of the hardware that can directly read computer components' sensors would be capable of producing more appropriate demand cooling responses in these circumstances.

Second, the amount of fluid that circulates through the heat exchange is minimal compared

to air-based cooling system. While the SPLIC fluid volume is capable of holding more heat than air, only a small portion of the volume can pass through the heat exchange at any time, which limits the rate of heat dissipation. This also redistributes heat retained by the coolant volume throughout all computing components, including ones that typically do not warm up during operation. After long durations, the reduced thermal differential between hardware and coolant reduces the rate of heat dissipation and extends the time required for hardware temperatures to return to nominal levels. This limits the proportion of thermal capacity within the coolant that can be leveraged within safety tolerances.

Ease of operation and maintenance. Modifying commodity hardware for initial server setup is intrusive but can be circumvented by utilizing specialty hardware designed for SPLIC. Non-specialized hardware can be limited by server layout, such as connectors that cannot bend sharply to reach ports and certain components that are inaccessible for maintenance without completely removing server blades due to the pod rack's vertical orientation. After removing hardware from the pod for maintenance, liquid residue can create slipping hazards and must be carefully monitored and cleaned. Finally, adding or removing components in the pod changes the level of coolant, which may require adding or removing coolant to maintain an appropriate volume for the container.

Reliability. Throughout our extended immersion of hardware, we have not observed degradation in peak performance. Plastic components affected by the plasticizer are eventually damaged and may require replacement during or after maintenance.

We make note that a power event at our datacenter inadvertently placed the pod's management software into a "demonstration" state. This state altered behaviors of the pod's operation and visible feedback mechanisms, including displaying inaccurate information on fluid temperatures and pump activity. The behavioral changes included increased facility water draw (we measured a constant $9\frac{m^3}{h}$ compared to our normal peak draw of $6\frac{m^3}{h}$) while disabling coolant circulation. These changes did not immediately damage any equipment, but could have prevented active cooling and created a legitimate risk of damaging the immersed hardware if these changes were not observed by our facility's staff. The "demonstration" settings persisted through power cycling the entire unit, requiring us to shut off the equipment until the issue was properly identified and rectified with vendor assistance. We have advised the vendor against shipping this capability in future versions of the management firmware as it is intended for use in showrooms and not in production systems. Due to the closed-source nature of these systems, unexpected behaviors such as these may arise periodically

as the technology matures and adapts to different environments and use cases.

Chapter 4

Lasting GPU Acceleration: A Case Study in Word2Vec Optimization

4.1 Introduction

Improvements to computing hardware can provide some performance improvements to software applications transparently. However, most applications are only designed to leverage the current scale of hardware capabilities at the time of development. This can limit performance improvements of software relative to the increased peak capability provided by newer hardware. This frequently occurs in GPU software, where new hardware often provides additional capacity and throughput for both memory and compute but software fails to realize proportional performance improvements.

We observed such performance scaling limitations in GPU implementations of Word2Vec [10], a distributed artificial neural network. Word2Vec trains dense vector representations of words, known as word embeddings, from natural human language. The trained embeddings enable programs to meaningfully interact with human language through vector geometry that efficiently captures syntactic and semantic similarities between words for further learning and inference [25]. Vector operations permit combinations of words to expose more complex and conceptual relationships. For example, the words 'Rome' and 'London' cluster relatively near to one another in the embedding

Portions of this chapter are based on work that was originally published in the Proceedings of the International Conference on Supercomputing 2021 under the title "FULL-W2V: Fully Exploiting Data Reuse for W2V on GPU-Accelerated Systems".

space, and the distance between them is similar in direction and magnitude to the distance between the words 'Italy' and 'UK'.

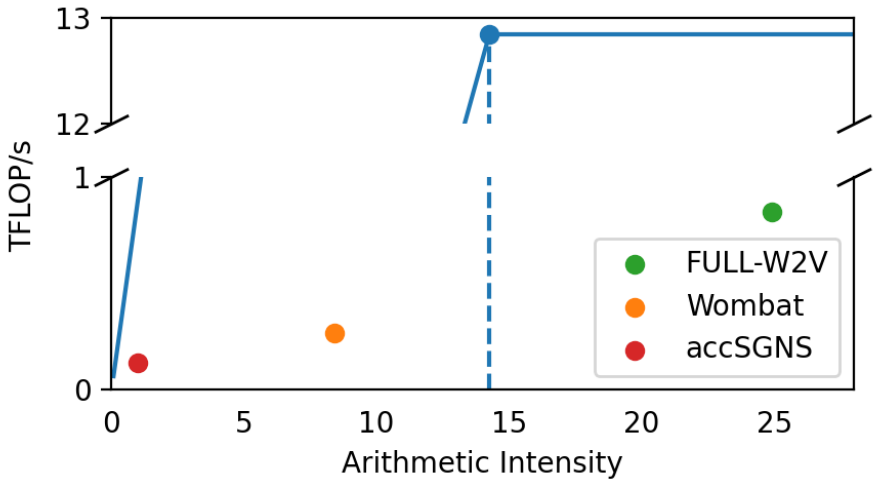


Figure 4.1: Roofline benchmarks for state-of-the-art Word2Vec kernels on a V100 GPU. The solid blue line is the roofline boundary, the dotted blue line marks the inflection point between memory-bound (left) and compute-bound (right). Previous work is memory-bound and exhibits poor overall throughput despite being data-intensive; our work, FULL-W2V, presents a significant improvement.

The state-of-the-art [26, 27, 28, 29] GPU implementations of Word2Vec — Wombat [28] and accSGNS [29] — struggle to effectively utilize the architecture as shown in Figure 4.1. While it is well known that data-intensive workloads struggle to achieve high arithmetic throughput, GPU implementations of Word2Vec have thus far not approached the peak of its potential performance on this architecture. Our implementation — known as FULL-W2V — represents a significant improvement in overall performance and a significant climb in effective arithmetic throughput. These results confirm the challenges in managing latency for Word2Vec on GPUs for data-intensive workloads like Word2Vec as well as the necessity of highly-targeted optimizations.

It is worth noting that many tasks within the natural language procesing (NLP) landscape that used to rely on word embeddings have replaced the use of word embeddings with transformer networks [30] and large language models, including language translation, image captioning, and automatic summarization. However, many "2Vec" variations based on Word2Vec remain relevant to this day, especially within field-specific text analyses, recommendation systems and graph processing [8, 31, 32, 33, 34, 35, 36]. Applications that continue to utilize word embeddings continually need to be updated to capture the latest domain knowledge that can be extracted from ever-growing and ever-evolving corpora and graphs. However useful these word embeddings may be, it is com-

puatationally expensive to train new Word2Vec embeddings for three primary reasons. First, the Word2Vec algorithm [10] sequentially trains small moving context windows from the corpus with minimal data parallelism, repeating the process until convergence. Second, the algorithm's computational complexity scales with both the embedding size and number of unique words to be embedded, the vocabulary, which are ever-increasing for many applications. Third, the state-of-the-art algorithms involve intensive memory accesses and have low arithmetic intensity, limiting hardware scalability. Using Word2Vec and similar techniques with larger and larger datasets requires throughput scaling for training in order to remain effective. While multiple techniques have been proposed to explore parallelism on GPUs [26, 27, 29, 28] and to reduce memory accesses by improving data reuse [27, 37, 38], these works fell behind the performance capabilities of recent generations of GPU hardware and are unlikely to scale to future hardware architectures due to the aforementioned issues.

We devise novel technologies to significantly improve Word2Vec performance on GPU architectures. Our key idea is to fully exploit reuse opportunities for different types of words during training, and explicitly cache them in registers or shared memory based on their request size and duration. While we explicitly study the Word2Vec algorithm in its original text-based form, we note that all algorithmic concepts are transferrable to other domains (i.e.: graph nodes operate as "words" and edges act as "sentences" in Node2Vec [33], the graph-based variant of Word2Vec) and therefore our work generalizes to other applications within this class of algorithms. Word2Vec has a high degree of reuse opportunities that are particularly suited to the storage technologies on GPU if the algorithm is expressed correctly to take advantage of them. For example, high numbers of available and flexible registers, and explicitly allocable shared memory with the same latency as low-level caches. We take advantage of these technologies to cache reused data fully for the extent of its lifetime, use techniques such as ring buffers to limit the overhead and management cost of such techniques, and balance heavy storage use to ensure scheduling units are still saturated and latency is fully hidden.

In this chapter, we present FULL-W2V, a fine-grain parallelism, highly scalable Word2Vec GPU algorithm with optimized data reuse. First and foremost, this algorithm maintains the required semantic ordering of context windows, maintaining prior guarantees of convergence. Second, it exploits three levels of work partition including batches, sentences, and embedding to create high degrees of parallelism on GPUs. Third, compared to previous work, our algorithm fully exploits data reuse opportunities, resulting in increased throughput and greater performance scalability,

nearly eliminating per-thread memory stalls. Lastly, our algorithm coordinates CPUs and GPUs to seamlessly provision data and launch concurrent kernels to saturate GPUs with work.

We have implemented the algorithm and evaluated the prototype on multiple generations of GPUs. Experimental results show that our algorithm produces embeddings with the similar quality as existing works. It automatically achieves 2.966X speedup when moving from Nvidia Pascal 100 and Volta 100 cards. In comparison to the state-of-the-art CPU and GPU algorithms, it outperforms state-of-the-art multithreaded CPU implementations by 5.44X, and modern GPU implementations accSGNS and Wombat by 5.724X and 8.647X respectively. Deep analysis shows that our algorithm increases the arithmetic intensity by 23.90 and 16.46 over accSGNS and Wombat respectively by improving register locality and utilizing advanced caching techniques to control data reuse.

We make the following contributions in this chapter:

- We present FULL-W2V, a fine-grain parallelized, highly scalable Word2Vec GPU algorithm, which overcomes the challenges of latency hiding inherent in data intensive Word2Vec training. It achieves 8.647X speedup over the state-of-the-art on Nvidia V100 GPUs.
- FULL-W2V is the first Word2Vec implementation to exploit *independence of negative samples* to enable opportunities to cache and reuse negative samples in registers for Word2Vec training. It improves arithmetic intensity and instruction level parallelism by interleaving memory demand and computation.
- Realizing that memory access is still a performance bottleneck, FULL-W2V exploits *lifetime reuse of context words* to eliminate 91% of overall memory demand, significantly reducing average memory access latency while optimizing data sharing, reuse, locality, and coalescing.

4.2 Related Work

All Word2Vec implementations historically stem from foundational work by Mikolov et al [10, 9], which expresses high level data parallelism between sentences of the corpus for improved performance. According to Hogwild! SGD [39], as long as large models are trained with batches that have sufficiently varied contents, parallel gradient descent training can be performed in a lock-free environment without synchronization. This condition is generally true for Word2Vec with distinct sentences from a given corpus, so data parallelism amongst sentences is commonly exploited using

CPU threads or GPU thread blocks.

There have been many implementations of Word2Vec since the seminal works, including implementations for the Tensorflow [40] and Gensim [41] machine learning frameworks. The algorithm has been ported to many architectures, including the cloud-based BlazingText [26], cluster implementation BIDMach [42], and FPGA architectures [43]. We focus the rest of our discussion on published Word2Vec implementations that push the boundaries of the algorithm's throughput on single-node CPU and GPU architectures.

4.2.1 State-of-the-Art CPU-based Implementations

pWord2Vec. Ji et al [38] reduce memory intensity of Word2Vec by "sharing" the first N negative samples with all other context words in each window. For data-intense networks such as Word2Vec, reusing many vectors in each context window's update greatly improves arithmetic intensity, which is further exaggerated by allowing high-performance BLAS libraries to perform the matrix arithmetic. While the authors were able to show that the semantic changes to the Word2Vec algorithm did not affect embedding quality, the matrix sizes are relatively small and the implementation's performance still fails to approach peak CPU throughput.

pSGNScc Rengasamy et al [37] utilize advanced batching techniques to combine multiple context windows into larger matrix batches. The technique allows CPU architectures to achieve much greater throughputs, but computation still takes place entirely on the CPU architecture and is otherwise equivalent in performance to pWord2Vec.

4.2.2 State-of-the-Art GPU-based Implementations

accSGNS Bae and Yi [29] utilize a fine-grain parallel implementation of Mikolov et al's original Word2Vec to bring the algorithm to GPU architectures. Their parallel hierarchy maps GPU threads directly to embedding layers while thread blocks and grids exploit data parallelism between sentences. The vector parallelism utilized in their implementation allows for some scalability on newer architectures, but is largely memory-latency-bound as little is done to affect the data-intensive nature of the Word2Vec algorithm, leading to workload imbalance and poor performance scaling on

newer architectures.

Wombat Simonton [28] focuses on Shared Memory optimizations for Word2Vec, leveraging the architecture's caches to exploit reuse within context windows. The implementation's parallel formulation uses relatively small thread blocks to operate on fixed word pairings from a context window while grids scale this parallelism across sentences. The techniques provide state-of-the-art performance on older architectures, but scheduling limitations imposed by the parallel decomposition hold back performance on newer architectures, leaving large room for improvement.

PARW2V Moon et al [27] more recently provided CPU and GPU implementations of Word2Vec that induce locality by reordering operations in Word2Vec's training updates and allow for reuse of negative samples beyond a single context window. The exact degree of negative sample reuse that can be exploited prior to reducing the quality of embeddings was not well understood, and the implementation mandates strict hyperparameter values that limit generalizability. Furthermore, we were unable to replicate the paper's reported results on our own systems, so this work is not discussed further in this chapter.

Our FULL-W2V is most related to accSGNS as both implementations utilize the same parallel hierarchy. However, we improve upon prior techniques by developing a cache for context words that fully reuses them throughout their lifetime in the sliding window with minimal management. We also utilize a local register cache and modified the workload decomposition to gain lifetime reuse of negative samples. All of these methods take advantage of data reuse and problem decomposition in manners previously unseen in Word2Vec.

4.3 Challenges on GPU

Addressing Memory Intensity and Latency. Like most machine learning algorithms, Word2Vec trains on a large amount of data, has inherently low computational intensity, and is generally latency-bound. For CPU implementations, Word2Vec is parallelized coarsely along independent sentences. The relatively low computational intensity and throughput of existing GPU implementations have demonstrated that a proper decomposition is difficult. To appropriately take advantage of the massive number of cooperative threads and memory hierarchies on the GPU, fine-grain parallelism within sentences must also be exploited, so as to effectively minimize high-cost memory accesses, hide latency, and maximize the effectiveness of cooperative threads.

Managing GPU Resource Tradeoffs. GPUs have fundamental tradeoffs when using different resources to improve performance. Shared memory and register caches are both high-speed options for caching data locally for high-locality operations. These two resources are more reliable than implicit caching, especially when (1) the required data for many threads may exceed the available footprint in L1 or lower level caches and (2) data with different levels of locality are required for the same computation. However, these resources also have limited capacity, and their overallocation can restrict the total number of resident threads available for execution, leading to reduced ability to hide memory latency. Eliminating expensive memory operations by caching data and hiding latency by using cache to support thread execution both serve to improve performance, but it is difficult to predict which is more effective for a given problem. We must tune the usage of these resources to the application's data locality and balance their usage to maximize performance.

Preserving Embedding Quality. When exploring new avenues for parallelization and data caching, it is critical that we avoid data dependency violations and minimize race conditions. All parallel implementations of Word2Vec thus far have had implicit race conditions between sentences containing the same words, but the impact of this is minimal and does greatly impact the rate at which the algorithm converges under the principles described by Hogwild [39]. However, when introducing further parallelization at the sentence level we risk introducing data dependency violations and compromising the quality of the embeddings produced. Additionally, changes to allocable memory and explicit caching potentially introduce additional coherency issues that must be managed. Careful algorithmic analysis and study of data lifetime is required to preserve overall embedding quality.

4.4 Methodology

In this section we introduce FULL-W2V, a highly optimized Word2Vec algorithm that is scalable on GPU accelerators. It overcomes the limited data locality in the state-of-the-art implementations and effectively exploits GPU architectures in two key ways:

- It exploits independence of key arithmetic sequences and decouples computations in fine granularity for improved parallelism and reduced data dependency.
- It fully exploits the temporal locality and data reuse to reduce access to lower levels of memory and average memory access time.

4.4.1 The Independence of Negative Samples

We first introduce the *negative sample independence* property of Word2Vec that allows us to make fine-grain parallelism and highly-effective memory access optimizations. When processing each context window in a sentence each context word is paired against each negative, and the sum result all pairings is applied as the model update. Because the sum is commutative, each pairing may be computed independently in any order. Acknowledging this independence offers us two opportunities. First, each negative sample can be independently paired with the context words without synchronization, allowing *fine-grain parallel* processing among the negative samples. Second, we can change the order of processing such that all context words are processed for a fixed negative, enabling *temporal locality* for each negative sample. Recognizing the property of negative sample independence, FULL-W2V flexibly manages the order that negatives are processed within a single context window and cache them to maximally reduce accesses to low memory levels.

Fine-Grain Parallelism and Temporally Distributed Data Dependencies. Each individual negative is independently iterated over the context words in a context window, and the $N + 1$ negatives can be fully decoupled from one another. The decoupling enables two types of opportunities: (1) fine-grain parallelism and (2) reduced simultaneous data dependencies. Fine-grain parallelism is crucial to latency hiding and scalable performance on GPUs, and provides flexibility for the scheduler to utilize available hardware resources. The decoupling reduces the simultaneous data dependency to a single negative sample instead of the whole collection, distributing the total number of accesses over the lifetime of the computation. Thus it eliminates the need for a thread block to simultaneously access and store *all* $N + 1$ negatives locally for the duration of the entire context window. Instead, each thread block only accesses the corresponding negative sample and stores its embedding vector directly for its lifetime.

With only one dependent negative, FULL-W2V stores the vector representation in a per-thread register cache. Using registers instead of shared memory has two advantages. First, register access incurs a much lower latency than shared memory access and alleviates the demand for latency hiding. Second, a negative sample does not have a large number of reuses, which shared memory requires for best practice. Indiscriminately and aggressively using shared memory reduces the space for thread warps and leads to degraded parallelism, performance, and limits the quantity that we can use for other optimizations better-suited for additional data reuse.

Temporal Locality and Reuse. FULL-W2V stores each negative sample in a register and accumulates all the required embedding updates in-register before writing it back to memory. Each negative sample is reused by $2W$ times spanning a single context window. In this way, we ensure our negative reuse has minimal impact on the quality of the resultant embeddings. While prior works [27, 37, 38, 42] indicate that the particular negative samples do not need to be independent across context windows, Moon et al [27] show that excessive reuse for negative samples has harmful impacts on the final embedding quality. Nevertheless, the limitations are not deeply understood by established literature. The reuse in a single window has notable improvement for minimal embedding quality cost [38], and greatly improves the access and storage patterns of negatives for GPU architectures.

One complexity of progressing to the next context window is the consistency requirements imposed by incremental model updates. As the context window slides across a sentence, context words are reused several times and therefore the corresponding model parameters have data dependencies on prior updates, requiring *strict sequential context window ordering*. Violating this constraint significantly harms algorithmic convergence and the resultant quality of trained embeddings. In order to adhere to *strict context window ordering* but take advantage of *negative sample independence*, FULL-W2V uses each thread block to process an entire sentence, with individual windows processing all negative samples independently before synchronously sliding the window. This approach optimizes the targeted negative reuse without violating any data dependencies and without over-reusing possibly stale data.

4.4.2 Lifetime Reuse of Context Words

The second optimization enables maximum data reuse for context words in the algorithmic characteristics of Word2Vec. As shown in Figure 2.1, we can determine the exact lifetime of context words based on the algorithmic structure of Word2Vec.

In traditional Word2Vec, each context window has a width that is randomly sampled between 1 and W and each window shifts the boundary and target word over by one position [44]. We can equivalently say that a context word's lifetime occupies at least 2 and no more than $2W$ windows plus one use as the target word of a context window, but the exact number of contextual uses varies based on sampling. Despite these known bounds for reuse, existing GPU algorithms fail to leverage this or compete with other memory demands of the algorithm by relying on the hardware's implicit memory management. This means that if a word is not sampled as context but

still has remaining lifetime, it could be evicted from the lower-level cache as other thread blocks induce memory demand, only for the word to be re-included in subsequent windows and need to be reloaded from higher-level caches or global memory. Separated from existing work, FULL-W2V fully exploits data reuse between context windows for the first time by explicitly caching and reusing context words with minimal resources.

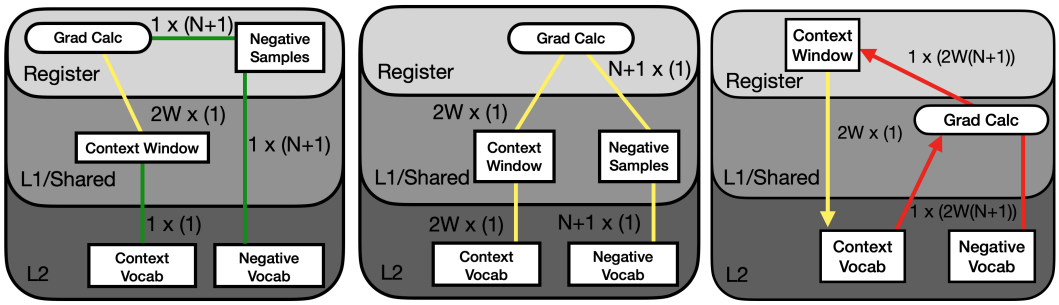
To reduce expensive high-latency memory accesses, FULL-W2V carefully utilizes GPU shared memory to cache context words for their entire lifetime and no further duration beyond that lifetime. A naive approach to enable reuse across multiple context windows is to allocate sufficient shared memory for all context words across the maximum lifespan. Naively re-implementing the original usage of context windows in a GPU shared memory cache would require $2W \times d$ space for each context window, with a full d-length vector per word in shared memory, or complex dynamic memory management to handle the vectors. In addition to this memory, relevant metadata to reproduce the context windows at training time would also need to be stored and fetched from memory. This approach would require a prohibitive amount of shared memory and management control flow, so a more sophisticated and scalable solution is required.

In order to minimize complexity and maximize scalability, FULL-W2V slightly alters the implementation of the context width hyperparameter. FULL-W2V uses a fixed context width $W_f = \lceil \frac{W}{2} \rceil$, or the average of the original random distribution. On average, the fixed context width produces the same quality embeddings while reducing (1) the need to sample or store per-context window metadata, (2) the shared memory allocation requirement by half, and (3) the overall implementation complexity of memory management. To simplify memory management and eliminate the need for costly GPU-based control flows, FULL-W2V builds a circular ring buffer in shared memory that matches the conceptual sliding window in Word2Vec. Each context word vector can be stored in shared memory upon its first introduction as a context word up until the final context window of its lifetime is trained. At this point, the new context word introduced by the next window can overwrite the oldest word whose lifetime just expired. Using this explicit memory management, FULL-W2V avoids contention among thread blocks over implicit caches to maximally reuse hot data. The circular ring buffer also minimizes the amount of shared memory required to store all necessary data for its full lifetime with relatively trivial overhead.

With this implementation, FULL-W2V can cache all values in context windows as soon as they appear and accumulate updates in the shared memory until the word is no longer eligible to

be a context word. The overall benefit is a further reduction of global memory accesses by $\frac{2W_f}{2W_f+1}$, approximately 86% for $W_f = 3$, or equivalently a 91% reduction over $W_f = 5$. In terms of the GPU architecture, this reduces the overall latency and therefore the demand for latency hiding, significantly improving on a key bottleneck.

4.4.3 Performance Implications



(a) FULL-W2V **(b)** Wombat **(c)** accSGNS

Figure 4.2: Parallelism and effective data traffic in the memory hierarchy involved in a single context window in the average case. Accesses are shown as *size* x (*iterations*), where $2W$ represents the number of context words in a context window and $N + 1$ represents N negative samples and the 1 target word. The colors correspond to how the traffic is related to Wombat (same = yellow, reduced = green, increased = red).

By combining fine-grain parallelism and data reuse enabled by the aforementioned technologies, FULL-W2V has significantly improved ability to hide memory access latency, and is scalable with GPU architectures. Figure 4.2 summarizes the resulting parallelism and effective data traffic involved in one single context window in the average case, where this current window is in the middle of sentence and share context words with multiple precedent and subsequent windows. FULL-W2V is distinct from the state-of-the-art Wombat in two key measures.

- Full-lifetime explicit context and negative caching at the shared memory and register level, respectively.
- Reduced traffic to each low memory level. FULL-W2V reduces access to L1/shared memory cache by 50% and access to L2 cache and GPU device memory by 42% compared to Wombat.

These differences have several performance implications with GPU architectures and resources. Our method of fine-grain work decomposition supports a high degree of parallelism, which

is critical for memory intensive workloads such as Word2Vec to hide latency and improve instruction-level parallelism. Meanwhile, the thread blocks and warps in FULL-W2V have far fewer accesses to low level memory and thus experience fewer memory stalls, improving overall computational efficiency. Consequently, FULL-W2V is able to better utilize the computing resource and achieve higher performance. Our experimental results provide detailed data to demonstrate these performance gains.

FULL-W2V has scalable performance across multiple generations of GPU architectures. New and more powerful GPUs constantly become available. They are typically equipped with more and faster processing units or SMs, more scheduler units, larger caches, and higher bandwidth. Given a newer architecture, FULL-W2V can automatically scale up the degree of parallelism to utilize these additional SMs and provide eligible ready warps to be handled by more scheduler units. Equally importantly, improved latency hiding, instruction-level parallelism, and reduced overall memory cost cooperatively improve overall performance and execution efficiency.

4.5 FULL-W2V Design and Implementation

We implement a prototype of FULL-W2V for test and evaluation of our methodology. The prototype materializes the methodology introduced in Section 4.4 as well as optimize the CPU-GPU coordination.

4.5.1 CPU-GPU Coordination

There are two primary goals that require coordination between CPU and GPU devices. The first goal is to ensure that the GPU remains occupied and utilizes its hardware to the greatest possible extent as an accelerator. The second goal is to allocate the workload between devices in such a way that the CPU handles all batch-related precomputation and indirected accesses that would hamper GPU performance if it were instead performed within the kernel.

GPU Utilization. As with other GPU implementations, FULL-W2V partitions the Word2Vec workload into a *batching* component on the CPU, and offloads the batches for *training* on the GPU. In our implementation, the CPU-side workload includes precomputation (including vocabulary setup and text-to-vocabulary translation), random sampling (particularly for negative sample identification) and assembling sentence data and training metascheduling as batches in a

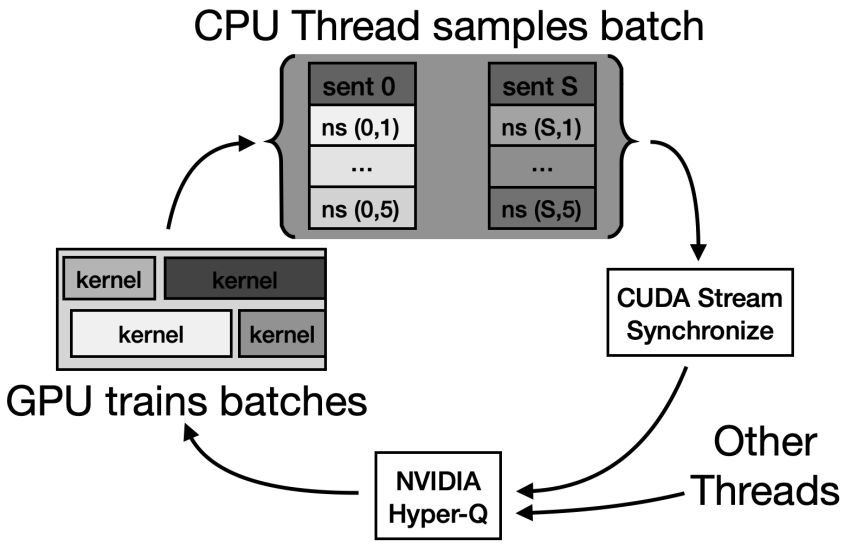


Figure 4.3: The per-stream coordination in FULL-W2V. On each stream, $S = 10,000$ sentences (sent) are sampled from the corpus and $N = 5$ negative samples (ns) are selected for each context window in each sentence.

format that is amenable for GPU usage. The GPU-side workload includes training via execution of the Word2Vec algorithm. The heterogeneous coordination is represented in Figure 4.3. Because (1) this is a synchronous process, and (2) batches are relatively small relative to the total computational capability of a GPU, we take advantage of NVIDIA Hyper-Q with CUDA Streams to allow many cooperative CPU threads to batch simultaneously, launching GPU kernels executing in parallel to saturate the GPU.

Workload Preprocessing on CPU. Similar to Wombat [28], the FULL-W2V batching process includes sentence preparation and negative sample selection. Performing this work on the CPU reduces the number of indirect memory accesses in Word2Vec that need to be performed on the GPU and entirely eliminates GPU copies of several Word2Vec data structures, ultimately improving memory access efficiency on the device. However, unlike Wombat, FULL-W2V does not expand batches into context windows via the CPU. FULL-W2V instead uses the GPU to create windows within the kernel as described in Section 4.4.2. Sentence and negative samples are provided as constant memory to the kernel, allowing GPU hardware to utilize the constant memory buffer and alleviate pressure on the cache hierarchy to better serve the model weights during training.

Additionally, FULL-W2V adjusts the traditional Word2Vec workload to facilitate more consistent and efficient GPU utilization without impacting model quality. Typical Word2Vec pre-

Table 4.1: CPU batching speed in millions of words/sec without memory transfers or kernels. Batching speed can become a bottleneck for faster implementations of W2V.

Implementation	Text8 Batching Speed	1bw Batching Speed
FULL-W2V	210.340633	265.212834
Wombat	16.957496	16.653851
accSGNS	16.527374	15.263448

processing treats all characters until a newline character as part of the same "sentence". This creates very large imbalances in sentence length that can lead to poor workload balancing on GPUs. To address this, we maintain the sentence delimiter token but only terminate sentences upon batching up to a fixed maximum sentence length or upon reaching the end of training data. This increases the average length of sentences in our tested datasets and therefore the per-batch workload size, incurring $< 0.5\%$ additional word pairings relative to other Word2Vec implementations. Due to the trivial fraction of word pairings this creates and small context window size that is typically used in Word2Vec training, we do not observe adverse impacts on the quality of learned embeddings but do enjoy greater performance due to improved balance between parallel workloads.

Finally, we note that achieved batching speed is now important for the effective execution of Word2Vec. Table 4.1 represents the rate at which GPU workloads are batched in millions of words-per-second. Previously, GPU training speeds did not approach the CPU's batching speed, so the CPU performance in heterogeneous implementations was of negligible importance. However, with FULL-W2V, we have accelerated the GPU training speed beyond that of previous CPU-side batching and require improved CPU batching to maintain performance across the heterogeneous system. Our improvements to batching speed are partially due to the algorithmic changes we previously discussed. To further increase the batching speed, we memory map the training corpus into the CPU and preprocess the text using the training vocabulary instead of performing string conversion during the batching phase. Because the GPU holds all of the model weights during training, the CPU can utilize its memory to store the converted corpus and leverage its own memory hierarchy to accelerate batching speed. The greater batching speed from CPU allows the GPU's improved performance to remain occupied by useful work, and represents an upper bound on the training speed that can be achieved by a particular Word2Vec implementation.

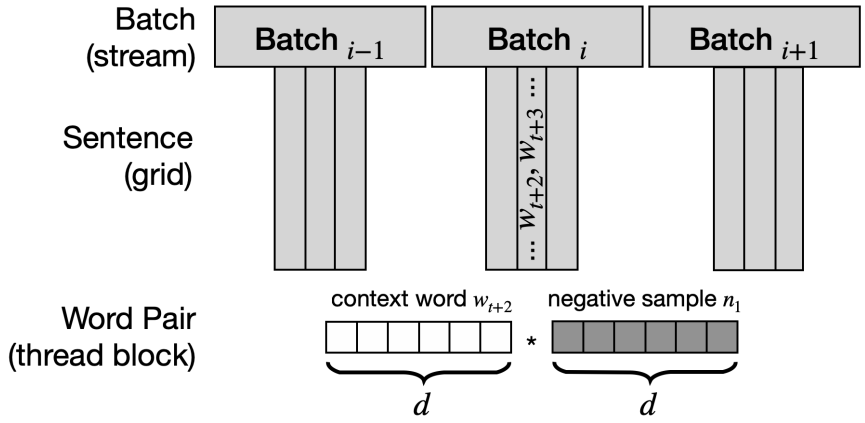


Figure 4.4: The multi-level workload decomposition and parallelism of FULL-W2V. Multiple sentences are batched for each CUDA stream, which launch grids with one thread block per sentence. Each thread block parallelizes embedding layers to operate on pairs of words with many threads.

4.5.2 Parallelism Hierarchy

Maximizing the utilization of GPU architectures demands massive concurrency, particularly for memory intensive applications like Word2Vec, to hide data access latency. FULL-W2V uses a fine-grain, hierarchical parallel approach to meet this demand, as shown in Figure 4.4. The hierarchy consists of three levels of parallel Word2Vec training: multi-sentence batch, a sentence and its current context, and a pair of words or embedding vectors.

Batch: The training corpus is divided into batches and multiple batches are simultaneously trained. This highest level is realized through CPU multithreading and Nvidia Hyper-Q CUDA streams as discussed in Section 4.5.1. We recommend creating one CPU thread per logical core for FULL-W2V. Each CPU thread iteratively manages batches and offloads the corresponding training to the GPU via its own independent CUDA stream.

Sentence/context window: There are a number S of sentences in each batch concurrently trained by a 1-d S GPU block grid. We parameterize the number of sentences S per kernel and use $S = 10,000$ as an empirical baseline for performance on our systems. Due to the strict sequential context window ordering, a sentence can only have one current context window, which slides over one word at a step. This context window requires the embedding updates of its context words and the negative samples. In our implementation, we pair one sample with all the context words and calculate the updates, and then iterate over the samples.

Word pairing: As each word is represented as a vector, a word pairing involves vector

operation, e.g., multiplication. The vector computations are parallelized among d threads in the same thread block. This level of parallelism enables coalescing and broadcasting of memory accesses, as well as cache availability. Because all threads within a block require adjacent vector items, independent warps coalesce their accesses, while potentially making the same data available in L1/L2 caches for other warps in collaborating thread blocks.

This hierarchical design can flexibly scale along multiple dimensions to provide strong throughput guarantees under a variety of problem settings and port to new architectures without source code modification. Our prototype implementation is capable of utilizing *word pairing* level scaling to accommodate larger embeddings without modification and automatically gains speedup on architectures with more SMs and warp schedulers.

4.6 Experimental Results

In this section we provide our experimental results to quantify FULL-W2V's general performance characteristics and success of our methodology at overcoming the challenges detailed in Section 4.3.

4.6.1 Experimental Platform and Evaluation Method

We evaluate FULL-W2V on three generations of Nvidia GPUs: V100, Titan XP, and P100 with different processing and memory technologies in Table 4.2.

Our analyses compare the following Word2Vec implementations:

- FULL-Register is a GPU algorithm that implements the techniques described in Sections 4.4.1 and 4.5.
- FULL-W2V is an extension of FULL-Register that additionally implements techniques described in Section 4.4.2, and represents our full contribution.
- pWord2Vec [38] CPU algorithm is closely related to FULL-W2V and is highly influential on the design of many other Word2Vec works, providing a baseline of expected embedding quality for Word2Vec under Shared Negative Sampling.
- pSGNScc [37] CPU algorithm has the greatest multicore CPU throughput on our systems

Table 4.2: Word2Vec evaluation platforms.

Hardware	GPU Specs	CPU Specs
GPU: V100 Gen-6 Volta CPU: Xeon Gold 6148 Gen-6 Skylake	80 SMs 14 TFLOP/s 16 GB HBM2 900 GB/s 4 Warp Schedulers	2 20-core CPUs 2.40 GHz 27.5 MB L3
GPU: Titan XP Gen-5 Pascal CPU: Xeon E5-2670 v3 Gen-4 Haswell	60 SMs 12.15 TFLOP/s 12 GB GDDR5x 548 GB/s 2 Warp Schedulers	2 12-core CPUs 2.30 GHz 30 MB L3
GPU: P100 Gen-5 Pascal CPU: Xeon E5-2680 v4 Gen-5 Broadwell	56 SMs 9.3 TFLOP/s 12 GB HBM2 549 GB/s 2 Warp Schedulers	2 14-core CPUs 2.40 GHz 35 MB L3

with a unique batching mechanism to demonstrate state-of-the-art throughput for Word2Vec on CPU architectures.

- accSGNS [29] GPU algorithm represents a somewhat naive benchmark for CPU-style Word2Vec implemented on GPU hardware.

- Wombat [28] GPU algorithm provides a state-of-the-art GPU performance for SGNS utilizing shared memory matrix multiplication and in-warp shuffle operations.

Corpora. Following existing literature, we evaluate the quality of generated embeddings using the Text8 corpus [45] as well as the One Billion Words corpus [46]. Table 4.3 presents summary details regarding each corpus under our experimental conditions. The Text8 corpus is commonly used for benchmarking evaluations, while the latter includes much more text, allowing it to more reliably predict downstream task performance on a much larger vocabulary [8]. Therefore we focus on Text8 for throughput analyses and One Billion Words for quality analyses.

Evaluation Metrics. We evaluate the algorithms with two types of metrics.

Training speed and performance. We report multiple measures of performance including the training throughput in words per second, and various fine-grain GPU performance data obtained from the *nsight* profiling tool.

Training quality. We utilize Spearman's rank correlation coefficient to compare the cosine similarity of word vectors to human similarity judgements established in WS-353 [35] and SimLex-

Table 4.3: Word2vec corpus information. Both corpi only train on words that are used at least five times and are limited to up to 1,000 words per sentence.

Corpus	Vocabulary Size	Words/Epoch	Sentences
Text8	71,291	16,718,845	17,006
One Billion Words	555,514	804,269,957	30,607,795

999 [36]. We also use Hyperwords [44] to perform analogy reconstruction with cosine addition and multiplication as in the famous Kings-Queens example, and utilize Mikolov's original analogy set [10] as analogy prompts.

Evaluation Procedure. For overall throughput and all embedding quality measures we report the mean and standard deviation of five identical executions to reduce the impact of variance inherent to the Word2Vec algorithm. All evaluations follow conventional Word2Vec hyperparameters established in Mikolov et al. [10] with the following noted exceptions. All experiments use the embedding size of 128, which equalizes GPU performance between all implementations by ensuring each algorithm allocates complete warps and benefit equally from aligned global memory offsets regardless of the thread block size used by any given kernel — the fundamental performance of FULL-W2V and other GPU algorithms are largely unaffected by this choice. We allow each implementation to utilize one CPU thread per logical core on the platform. We allow 20 epochs of training on the Text8 corpus, which was empirically determined to be sufficient for convergence across all implementations; for similar reasons we train the One Billion Words corpus for 5 epochs.

4.6.2 Overall Performance

We first evaluate the overall performance benefits created by our algorithm. Figure 4.5 shows the training throughput for each algorithm using the Text8 on each experimental architecture. We make several important observations.

- FULL-Register on the XP architecture outperforms all prior works on any architecture and has greater performance scaling cross-architecture than prior works. FULL-W2V on the older P100 hardware nearly doubles FULL-Register's XP performance (15.13 million words/sec vs 8.64 million words/sec), but FULL-W2V scales its own performance between architectures to a similar degree as prior works.

- The margin of performance gain for FULL-W2V and FULL-Register over prior works increases

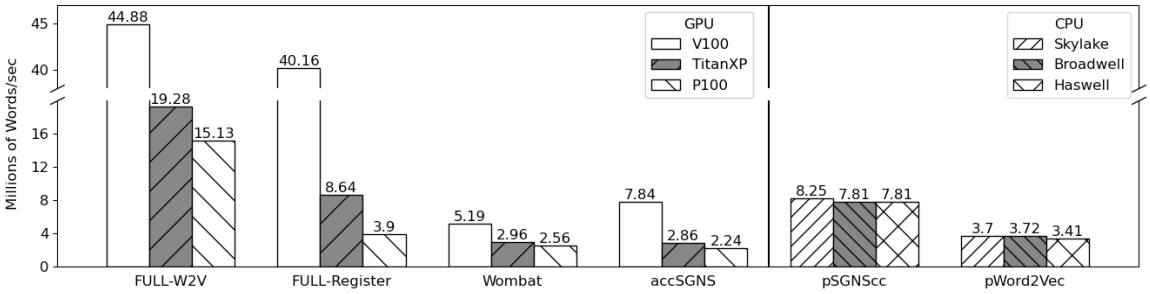


Figure 4.5: Throughput in words/second on Text8 corpus on various architectures. $d = 128$, $N = 5$, $W_f = 5$.

with successive hardware generations. FULL-W2V is 6.754X and 5.910X faster than accSGNS and Wombat respectively on P100 and 5.724X and 8.647X faster than the counterparts on V100. FULL-Register is 1.741X and 1.523X faster than accSGNS and Wombat respectively on P100 and 5.122X and 7.738X on the V100.

- Only the FULL-W2V and FULL-Register GPU algorithms are capable of outperforming the peak performance from state-of-the-art CPU algorithms. Only FULL-W2V can outperform CPU implementations using the P100 GPU architecture, with greater performance gains on newer and more powerful GPUs. AccSGNS on V100 cards achieves comparable performance to the CPU-based algorithms, while Wombat has a lower performance than the pSGNScc algorithm on all three CPUs for the Text8 benchmark and only reaches CPU performance on V100 with the 1bw benchmark.

We also present the throughput performance for the One Billion Words corpus in Figure 4.6. While this corpus is not traditionally used for throughput analyses, we note that once again how FULL-W2V and FULL-Register attain higher throughputs when utilizing older GPU architectures than prior works can achieve using newer GPUs. Once again, only FULL-W2V is able to outperform all CPU implementations using the P100 GPU, with additional performance gains on newer architectures.

4.6.3 Method Evaluation

4.6.3.1 Addressing Memory Intensity and Latency

We observe that the register-exploited *independence of negative samples* in FULL-Register results in significant reductions in DRAM demand compared to accSGNS, which has a similar access

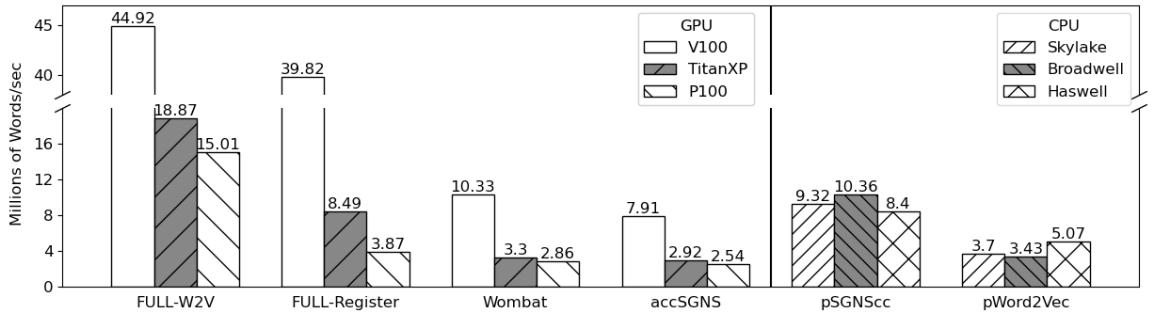


Figure 4.6: Throughput in words/second on One Billion Words corpus on various architectures. $d = 128$, $N = 5$, $W_f = 5$.

Table 4.4: Memory demand in gigabytes-per-epoch collected via *Nsight* with the Text8 corpus for a fixed number of epochs.

Implementation	L1/TEX	L2	DRAM	Sum
FULL-W2V	**94.760**	**88.723**	**41.851**	**225.334**
FULL-Register	885.065	781.576	66.555	1,733.196
accSGNS	1,134.448	493.614	226.578	1,854.640
Wombat	2,303.525	1,432.774	45.799	3,782.098

pattern to FULL-Register, eliminating 70.6% of the longest latencies in the memory hierarchy. The lack of register caching and shared negative samples in accSGNS leads to much more data demand than the lower level caches can satisfy. The memory hierarchies on GPUs are not as robust as CPU memory hierarchies and have to handle much more parallel contention. Our work demonstrates that a large portion of the memory demands of Word2Vec are not fundamentally important to the resultant quality of embeddings and can be eliminated for performance gains. As a data-intensive algorithm, reducing memory demand helps to circumvent the memory latency bottleneck of Word2Vec on GPU architectures, contributing to the massive performance increases seen in Section 4.6.2 between the FULL-Register and FULL-W2V implementations. The latency bottleneck is more pronounced on older architectures, where fewer SMs with smaller caches and higher latency memory technologies expose threads to longer access delays that are otherwise difficult for the architecture to hide with other data-intensive work.

We further leverage latency elimination in FULL-W2V, which explicitly manages memory with *lifetime reuse of context words* in addition to the negative sample optimizations. Table 4.4 shows that FULL-W2V reduces overall memory demand by 94.0%, 87.9%, and 87.0% over Wombat, accSGNS, and FULL-Register respectively. The extreme reduction in memory demand on other parts of the hierarchy is replaced by reuse in Shared Memory, guaranteeing L1 hit latency on each

access that is not serviced by the rest of the memory hierarchy. This is important because the data-intensive access pattern of Word2Vec is both sparse and highly stochastic. Under these conditions, the GPU's hardware-managed caches cannot be expected to provide proper eviction policies to maximize reuse for the algorithm, but *lifetime reuse of context words* guarantees cache hits for the Word2Vec algorithm for as long as can be statically known.

Memory latency is a complex facet of GPU performance, so we study its performance impact as a composition of memory demand per cache level and the L1 cache read hit rates based on Table 4.4. We note that the reduction in memory intensity is not wholly sufficient to explain FULL-W2V's performance, as FULL-Register on the V100 architecture is capable of high performance relative to prior state-of-the-art with similar memory demand, however it is a significant source of improvement provided by our work.

The GPU memory hierarchy on all studied architectures includes L1 and L2 caches backed by DRAM for global memory accesses. The demand per cache level represents where the memory access is ultimately resolved from while the hit rates determine how often a higher memory unit is needed to fulfill accesses. The combination of shared negative sampling and *independence of negative samples* in FULL-Register relocates about 22% of the memory demand in accSGNS from L1 to L2 but simultaneously eliminates 70.6% of all DRAM accesses. With fewer L1 accesses and register-allocated negatives instead of repeated global operations, the L1 hit rate in FULL-Register is merely 11.70%. However, these misses go to L2 and rarely require DRAM, resulting in lower average latency for L1 misses than accSGNS's misses. The *lifetime reuse of context words* in FULL-W2V then further reduces FULL-Register's memory demand for L1, L2 and DRAM by 89.3%, 88.6% and 37.1% respectively, requiring the fewest memory accesses across all levels of the GPU memory hierarchy. The extreme amount of memory demand removed by *lifetime reuse of context words* significantly reduces the impact of memory latency on the implementation's performance by entirely eliminating the accesses in the first place or forcing them to be guaranteed hits in the program-managed L1 cache. The L1 hit rate for FULL-W2V moderately improves to 20.44%. The dramatic reduction in L1 and L2 demand relative to FULL-Register mean that most accesses are made with minimal latency, allowing the hardware greater flexibility in scheduling computation to hide memory access times and sustain high throughput.

Table 4.5: Average Issue Eligibility per Warp Scheduler per Cycle. Maximum active warps on both architectures is 16. FULL-W2V is always near-peak occupancy and has near-ideal eligible warps, indicating good latency hiding as well as scheduler saturation.

	XP Architecture				**V100 Architecture**			
	Wombat	accSGNS	FULL-Register	FULL-W2V	Wombat	accSGNS	FULL-Register	FULL-W2V
Max Warps	11.03	12	16	13	11.03	12	16	9
Active Warps	4.59	11.08	15.86	9.59	4.66	9.41	14.92	8.99
Eligible Warps	0.16	1.33	0.42	0.99	0.18	1.09	1.86	1.90

4.6.3.2 Managing GPU Resource Tradeoffs

We analyze our effectiveness in managing resource tradeoffs by examining scheduler and thread-level statistics, starting with device processor and scheduler saturation. Table 4.5 shows that FULL-W2V is within 99% of its theoretical occupancy, indicating both inter-SM and intra-SM saturation of threads. Additionally, we can see that active warps are near-peak levels with appropriate (near 1) eligible warps-per-scheduler on the both architectures. This indicates that many warps are progressing on some operation, while a sufficient number of warps are available for scheduling. High activity and balanced eligible warps is a good indication that our latency reduction operations eliminated a sufficient amount of latency to justify a lowered overall occupancy without harming our ability to hide the remaining latency and still improve overall performance. This is also an indication that we can continue to scale to future GPU architectures, as we are not approaching any current hardware limitations and FULL-W2V can scale to new SMs by simply batching additional sentences per kernel.

We validate that our method reduces overall time spent on latency, we look at lower-level per-thread metrics, including overall IPC and its constituent breakdown. Table 4.6 shows that, despite improved performance, FULL-Register still spends a great number of cycles stalled on latency costs, particularly long scoreboard memory operations. On both GPU architectures, the introduction of *lifetime reuse of context words* nearly eliminates the cost of long-access memory, indicating that we significantly reduce the memory latency performance bottleneck. In turn, IPC is drastically increased, shifting much of the remaining time to compulsory overheads needed for algorithmic integrity such as thread synchronization. This single-thread improvement is highly validating of our overall throughput gains.

Table 4.6: Instructions per Cycle and Thread Stall Breakdown. Arithmetic stalls include math pipe throttle and MIO. Overhead stalls include wait, selection, barriers, dispatch, branch, no instruction, drain, sleep and miscellaneous stalls. FULL-W2V shows significant improvements between hardware architectures, and also nearly eliminates memory stalls through effective manual caching and data reuse.

	XP Architecture		**V100 Architecture**	
	FULL-Register	FULL-W2V	FULL-Register	FULL-W2V
IPC	1.19	2.78	2.38	3.22
Long Scoreboard	38.66	1.25	11.00	0.97
Short Scoreboard	4.49	3.43	4.19	2.95
Arithmetic	0.16	0.18	1.14	0.66
Overhead	13.05	10.48	7.93	6.35

Table 4.7: Mean embedding quality of five repeated trials using One Billion Words. Higher values are better.

	WS-353	**SimLex-999**	**COS-ADD**	**COS-MUL**
pWord2Vec	0.6070	0.3499	29.895%	29.166%
Wombat	0.5952	0.3596	29.661%	28.988%
FULL-W2V	0.5923	0.3582	29.775%	29.386%

4.6.3.3 Preserving Embedding Quality.

We evaluate the embedding quality of FULL-W2V and compare it against Wombat and pWord2Vec as presented in Table 4.7. These counterparts use the nearly identical batching semantics and negative sample reuse policies as FULL-W2V and thus create a fair comparison. FULL-W2V is statistically equivalent to the results generated by both Wombat and pWord2Vec by every measure of the training quality.

This positive result confirms that our algorithmic adjustments described in Section 4.4.2, including fixed context window sizes, are valid. As demonstrated in [47], larger window sizes are connected to divergence in learning quality between high and low-frequency words, but variance in window sizes does not appear to be critical to generating quality embeddings.

4.7 Conclusion and Future Work

FULL-W2V advances the state-of-the-art single-GPU performance across multiple hardware generations. We find that each negative sample in a collection can be independently updated over context words without affecting embedding quality, however the sequential accumulation of context word updates throughout sliding windows remains necessary for convergence. Based on these findings, we improve the efficiency of fine-grain parallelism with highly effective memory access op-

timizations — cache negatives in registers and context words in shared memory — to fully exploit their reuse. We show that the combination of fine-grain parallelism, novel memory demand reductions, and data reuse optimizations can generate synergistic performance gains and benefits on GPU hardware.

There are several directions for future work. There is a lack of understanding of the exact limitations of negative sample reuse without adversely affecting embedding quality. FULL-W2V and future algorithms can benefit from reuse of negatives over more than one context window. Related work shows that altering sentence batching and negative sample selection increases limits of guaranteed locality for additional performance benefits. FULL-W2V is positioned to explore such benefits. Finally, FULL-W2V can be extended to support multiple GPUs on the same node to further accelerate training and support large networks and corpus.

Chapter 5

Efficient and Transferable Multi-Scale Performance Autotuning

5.1 Introduction

The arrival of diverse architectures in high-performance computing (HPC) systems has unlocked many new opportunities and also permits existing applications to push beyond their former limitations. In Chapter 3 we demonstrated opportunities that can be provided by new hardware and in Chapter 4 we demonstrated opportunities that can be provided by improved software. Maximizing system performance for a given application and system requires more than independent optimization and is usually attained via performance tuning. Because hardware systems and software applications frequently allow many potential optimizations, searching for the best configurations is typically too expensive to be performed by hand or via exhaustive trial and error. Empirical performance tuning, widely known as autotuning, is a promising approach that evaluates a small subset of parameter configurations of a given kernel or application from a large user-defined search space by running them on the target platform to identify the best-performing configurations. A sophisticated search algorithm is often employed to intelligently navigate the large search space. Such autotuning approaches have achieved success in several prior works [13, 12, 16, 48, 15, 49, 50].

Despite prior successes, however, autotuning has faced adoption challenges for real appli-

Portions of this chapter are based on work that was originally published in the Proceedings of the International Conference on Supercomputing 2023 under the title "Transfer-learning-based Autotuning using Gaussian Copula".

cations because it is still resource expensive. Each empirical evaluation involves generating the executable with the parameter configuration and actual execution. Even simple kernels may require several hours to tune, while more advanced and complex applications with larger search spaces may require days. To reduce the computational expense of autotuning, researchers have developed transfer learning (TL) methods to leverage data from related autotuning tasks (e.g., similar input sizes or kernels). Although the optimum for a kernel changes with input size, high-performing regions in the search space are related across input sizes. This allows TL in autotuning to tune new input sizes of that kernel efficiently.

Existing TL autotuning methods are ineffective for few-shot, *i.e., a minimal number of empirical evaluations*, as they require many samples for new tasks to model the transfer relationship. To overcome this issue, we develop a new generative autotuning approach that uses Gaussian copula (GC), a data-efficient statistical model, to enable rapid TL autotuning. We use GCs to model each configuration parameter's distribution and codependencies. GCs permit generative tuning via conditional sampling, which restricts sample generation to configurations to satisfy constraints such as high performance for the input size or kernel of interest. Conditional sampling enhances the explainability of generated configurations and improves the likelihood of success on transferred problems. We enhance the GC's ability to model the marginal and joint distributions of parameters while mitigating its limitations for autotuning settings.

Our main contributions are as follows:

- A new generative modeling approach based on a data-efficient GC model, which enables few-shot TL based autotuning with a small number of empirical evaluations for new tasks; a generative modeling approach has never been developed or applied for TL autotuning before.
- Estimation of success probability for generative modeling to determine the necessary budget to expect quality autotuning results; this is the first work that provides probability estimation for TL autotuning.
- We demonstrate new performance insights for Polybench and Exascale Computing Project mini-applications by utilizing few-shot autotuning.
- We demonstrate limitations of our techniques and prior art in multiscale tuning, providing insights into current limitations and opportunities for future research.

Our code is open source and available at `https://github.com/tlranda/GC_TLA`.

5.2 Model Selection and Background

Our generative modeling-based TL approach is based on the Gaussian Copula, a well-known multivariate probability distribution in statistics literature. Copulas are a class of statistical modeling techniques that utilize the probability integral transform to decompose a multivariate probability distribution into its marginal distributions and use a separate function to couple those distributions. This means that the model distinctly represents each variable's observed behaviors observed within the data and separately models the between-variable interactions.

The Gaussian Copula model is a multivariate probability distribution that is trained to reproduce the training data during unbiased sampling. Training begins by creating a normal distribution for each independent variable in the training data. Then, a covariance matrix models the correlation between variables to represent all cross-variable interactions. The multivariate distribution is defined using the probability integral transform over the marginal distributions and covariance matrix, which can be sampled with a zero mean vector to reproduce the training data in expectation.

The Gaussian Copula also supports a method of inference known as conditional sampling, wherein arbitrary constraints can be imposed to bias the process. Because the Gaussian Copula distinctly models each variable's component representation and the joint interaction between variables, conditional sampling permits very powerful and flexible adaptation at inference time. One or more *conditions*, or limitations on the expression of a particular marginal variable, define the imposed behavior for inference. The probability integral transform allows this condition to be expressed through the covariance matrix. This adjusts the covariance and all unconstrained marginal variables, effectively tightening the scope of random generation to the remaining variance after the condition has been satisfied. This biasing process permits intentional sampling of particular subsets of the joint probability distribution without relying on rejection sampling.

We refer the reader to the work of Masarotto and Varin [51] for a more detailed mathematical exposition of the statistical model and mechanics. As a simple example within autotuning, suppose we have two variables: the input scale and a tunable parameter. A GC trained on this data learns to represent the distribution of each variable's values independently, then how those values tend to

correlate with one another. When sampling from the GC with a zero-mean vector, we can assume that many instances from the training data will be directly reproduced. During training for the transfer learning problem, we will have multiple input scales present within the data. However when we transfer to a particular input scale, any samples that are generated for other input scales are irrelevant and would normally be discarded. This is called "rejection sampling" and is a necessary component of many deep-learning sampling regimes. However, conditional sampling allows us to constrain one or more variables, including the input scale. When sampling with an input that is conditioned to be an exact value, the resulting outputs capture the expected variation in the tunable parameter given the fixed input scale. This means that every sample we produce at inference time will describe the input scale we seek to tune and rejection is not required, greatly increasing the efficiency of sample generation. Furthermore, the overhead of conditional sampling is constant with respect to the number of samples we generate because the probability integral transform only needs to be performed once.

5.2.1 Advantages over Alternative Generative Models

CopulaGAN [52], CTGAN [53] and TVAE [53] are two alternative generative models that have previously been proposed for synthetic data generation, including similar capabilities to support constrained data generation like the GC. These models are designed to scale their fitting capabilities to available data, but in autotuning we often have limited data that does not permit this advantage to materialized. Furthermore, these techniques are often computationally inefficient when driving searches. Table 5.1 demonstrates the inference latency of these models; GC has the lowest latency of all, which is comparable to purely random sampling.

Table 5.1: Time required and observed rejection rates when generating 1,000 unique samples using various techniques. Conditional sampling with the GC has latency similar to random sampling but represents learned relationships without ill-conditioned data.

Method	Time (s)	Sample Acceptance Rate	Reject Reason (%)	
			Repeated	Ill-Conditioned
Random	0.24	90.8%	9.2%	0%
GC	0.52	37.87%	62.13%	0%
CTGAN	1.28	4.8%	7.33%	87.87%
TVAE	80.77	0.05%	2.25%	97.70%

The GC's advantage in latency is partially due to the acceptance rate of generated samples. The separation of joint and marginal models permits the GC to satisfy constraints before generating

other values, so only repeated parameter configurations are removed from its generated configurations. While many more duplicates are generated compared to random sampling, the overall acceptance rate is nearly 10 times larger than CTGAN. Some other models, such as the Copula-GAN [52], can also utilize conditional sampling; however, it can fail to generate any configurations when prompted to produce out-of-distribution data, which is important for transferring tuning to new tasks.

CTGAN and TVAE generate excess samples and then employ filters to discard ill-conditioned data. These methods are computationally inefficient. While relaxing constraints can help reduce their generation latency, it comes at the cost of compromising the quality of the generated data, which no longer best fits the desired task. This supports our choice of the GC model for few-shot transfer learning autotuning scenarios, where both latency and utility are crucial factors to consider.

In summary, we use the Gaussian Copula because it provides us with the following key advantages for HPC transfer learning autotuning:

- **Minimal Training Data.** Technically, a joint probability distribution can be meaningfully formed from as few as two data points. Data is limited within HPC and deep-learning approaches that scale to data availability are prohibitively costly to use in many scenarios.
- **Inference Control.** Conditional Sampling permits highly specialized and compute-efficient inference. This increases the utility of biased inferences from the model while reducing overhead relative to other model choices, especially when generating values that must represent a particular portion of the learned distribution.
- **Known Statistical Properties.** The model is highly explainable, permitting deeper analyses of its strengths and weaknesses. We will later demonstrate how the probability model permits effective forecasting of a trained model's capabilities prior to using it in a new domain.

5.3 Few-Shot Autotuning Framework

The key idea of our TL approach is to leverage the GC to predict high-performing configurations on related tasks in few-shot autotuning.

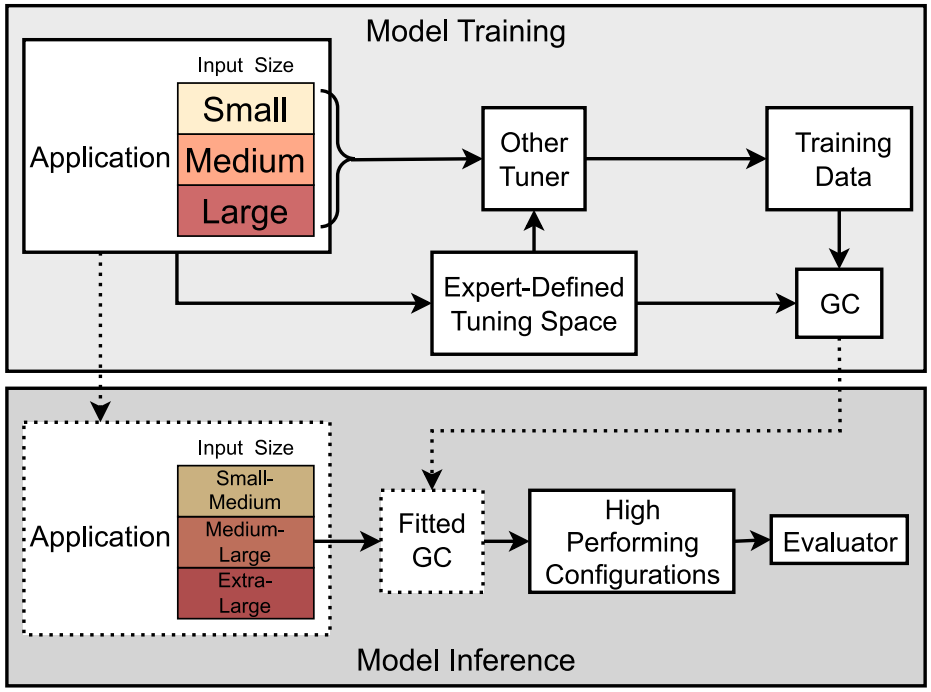


Figure 5.1: TL-based Autotuning Framework Using GC. TOP: Model Training, which uses GC to train fitted models with data collected from source tasks (multiple input sizes of an application) in a human-designed tuning space. BOTTOM: Model Inference, which uses the fitted GC models to propose high-performing configurations for new tasks and evaluates them.

Our proposed method consists of two phases: model training and model inference, as shown in Figure 5.1. Model training uses GC to fit data collected from source tasks in an expert-defined tuning space. In our work, the source tasks correspond to different input sizes for the same application, and the tuning space is specified via application source code annotation and predefined parameter values. The source tasks, tuning space, and source input sizes are presented to an existing autotuner [16, 50, 54, 55, 56, 57, 58] with a fixed evaluation budget to collect a small, quality training dataset of empirical performance. Model inference uses the fitted GC model to propose high-performing configurations for new tasks, which are then empirically evaluated. We discuss the modules in greater detail in the remainder of this section.

5.3.1 Model Training

Autotuning problems require experts to delineate key high-level features. The GC also requires several interventions to permit its general usage in autotuning and to improve its utility as a few-shot TL autotuner.

5.3.1.1 GC for Autotuning

We make several adaptations for GC to generalize it for the autotuning problem.

Variable Preprocessing Standard GCs model real variables but do not model mixed-integer (discrete, integer, and categorical) variables. To address this issue, we adopt a new GC approach proposed for synthetic data generation [59]. In this GC approach, numeric variables (real or integer) are modeled by truncated Gaussian distributions, and categories are reordered by their frequency in the fitting data. The GC also reduces the bias from distribution shape by converting all variable distributions to standard normal distribution before computing covariance.

GC as an Transfer-Learning Autotuner GC can be used as an transfer learning autotuner given a dataset of observed configurations in a defined tuning space. The trained model will capture the underlying distribution of input scales and tunable parameters independently and jointly. Conditional sampling will allow us to generate new samples through the probability integral transform for a given input scale. This includes input scales that were not included in the training data, which follow the expected covariances to predict marginal behaviors on new tasks. The generated configurations can be empirically evaluated to determine their fitness. We note that unlike other autotuning techniques, the GC does not utilize iterative feedback. This is because the samples are generated from the distribution, so re-training as additional samples are collected only serves to reinforce the models' biases and reduces variations.

5.3.1.2 GC Model Fitting for Few-Shot Tuning

Unlike existing TL autotuning methods, the GC does not require extensive or exhaustive datasets. Because the model does not attempt to regress against the performance relationship, it can scale down to whatever data is available. However, the GC also lacks a mechanism to disfavor parameter configurations with subpar performance, meaning that it is not inherently selective. We attempted to see if the performance relationship could be conditionally sampled to any affect from the model, however as may be expected the performance relationship is too complex for the GC to effectively model. Nevertheless, we can intentionally filter the data used for training based on known performance and fit the GC only to the highest-performing configurations. In the TL setting, GC-based autotuners generate high-performing configurations immediately and minimize the exploration

of low-performing regions. Because the GC separately models each variable and between-variable correlation, it is capable of modeling changes with respect to input scale, provided these changes are somewhat continuous in nature. As such the performance data from source tasks are only needed to *select* the training data for the GC; modeling the expected performance is empirically observed to be non-destructive, but we do not recommend including the performance objective in the GC's trained variables as it is of negligible utility.

Quantile Filtering We investigate dataset filtering based on performance quantiles to include only top-performing configurations in the training data for GC modeling. The best quantiles should result in a training data subset with similar distribution for high-performing configurations while maintaining ample tuning space coverage. By tuning space coverage, we mean the proportion of the original search space that can be generated through arbitrary combinations of variable values observed during sampling. We collect a number of samples from the GC equal to the original search space size — samples are not guaranteed to be unique — and consider any generated value to be "reachable" by the GC under some conditions. If the GC never generates a particular variable value in this manner, it is vanishingly likely that any configuration including that value can ever be generated by the GC, reducing the proportion of the tuning space that can be covered by the model.

To motivate the need for the proper threshold quantile, we present a brief analysis from an exhaustively tuned Syr2k task using Kullback–Leibler (KL) divergence [60], a statistical measure of the difference between two probability distributions. Zero KL divergence indicates that the compared distributions are identical; increasing differences between compared distributions increases divergence. We also analyze the tuning space coverage based on the filtered dataset since filtering can prevent some configurations from being generated.

Quantile filtering cannot be too aggressive. As shown in Table 5.2, the tuning space coverage decreases gradually at first but decreases dramatically (i.e., 0.82 to 0.07) when the filtering quantile decreases from 30% to 20%. The significant decrease suggests that a majority of parameter options, especially categorical parameters, have been eliminated. This proportion of the search space still includes 745 configurations which is much more than the few-shot evaluations we intend to perform. However, the KL-divergence is also shown to dramatically increase, indicating that the extreme tightening of coverage has eliminated many of the best candidates from consideration. Divergence decreases with fewer quantiles as the model becomes more precise, however this particular quantile

Table 5.2: The tuning space coverage and average marginal KL divergence of quantile-based filtering for the Syr2k benchmark. The KL divergence is calculated using the top 10% of all configurations as a reference, obtained through brute-force.

Filtering Quantile (%)	Tuning Space Coverage	KL Divergence
100	1.00	0.1878
90	1.00	0.1713
80	1.00	0.1609
70	1.00	0.1525
60	0.91	0.1409
50	0.91	0.1212
40	0.91	0.1333
30	0.82	0.1713
20	0.07	0.2766
10	0.06	0.3079

requires more precision than the GC and its training data are capable of providing, leading to poor outcomes.

Reducing KL divergence as the coverage decreases will increase the likelihood of sampling optimal configurations by redistributing the sampling probability from suboptimal areas of the space to regions that closely resemble near-optimal configurations. Based on this trend and our experiments, we recommend using less than 50% of the original tuning data to exclude low-performing characteristics from prior data. This excludes many evaluations that prior autotuners made to inform the surrogate model rather than to improve the best-known optimum. We empirically determine across our benchmarks and training data that at least a 15% filtering quantile is needed to avoid over-specification. We utilize the top 30% of prior data in all of our experiments to ensure adequate information is available for complex tuning tasks without overly harming the tuning space coverage.

5.3.2 Model Inference

The fitted GC model represents learned distributions that can be used for inference, but additional steps must be taken to utilize it as an effective TL autotuner.

5.3.2.1 Conditional Sampling

Quantile filtering increases the likelihood that a sampled configuration from the GC will reproduce optimal traits in new tasks, but this fails to respect the specific tuning needs for different tasks. Meaningful transfer between tasks requires us to label fitting data with a representation

of the task, t. As previously discussed in Section 5.2, conditional sampling allows us to imposes arbitrary constraints on the model during generation. We conditionally sample on the input scale to sample the remaining variance via tuning variables, prompting the GC to reconstruct the best-fit distribution it learned for the indicated task. If that task was not observed in prior tuning, the same model mechanisms "recover" a transferred relationship for the new task by projecting the learned relationship between tasks onto the marginal variables. Because the GC is trained on filtered high-performing source data, conditional sampling generates configurations that are expected to perform well for the transferred task.

5.3.2.2 Managing Probability of Success

The success rate for generative autotuning is subject to randomness, even though the transferred distribution is biased toward values that are expected to be near-optimal. Therefore, it is crucial to understand the probabilities involved in GC generation to determine whether the technique is appropriate and what evaluation budget is necessary to expect a certain threshold of success.

The GC's autotuning process samples k unique configurations from a distribution that spans $|C|$ potential candidates. Within the population C are $|I|$ ideal candidates, which are optimal or near-optimal. Frequently, the top 1% of evaluations in real-world benchmarks have nearly equivalent performance, so we consider I to be the set of configurations that have global fitness in the top 1% of all configurations C. Identifying one or more of the candidates from I within the budgeted k trials is an acceptable goal for few-shot TL autotuning. The probability that one or more such ideal candidates are selected within k trials is hypergeometric sampling, described by Equation 5.1:

$$P(\#Optimal \geq 1) = \sum_{i=1}^{k} \frac{\binom{|I|}{i}\binom{|C|-|I|}{k-i}}{\binom{|C|}{k}}. \tag{5.1}$$

If we fit all source task data and $|C|$ is the size of the entire configuration space, then sampling the top 1% of performance within a few shots is unlikely. For example: random sampling would require 70 trials for a 50% chance of sampling just one such configuration in expectation; 95% confidence of observing one ideal candidate would require 300 unique random samples. Using quantile filtering on the source data for the GC can make some configurations statistically improbable or impossible to generate, eliminating them from the search. These excluded configurations are expected to be suboptimal because they fail to exhibit characteristics common with known optimal-

like data from source task tuning.

Eliminating suboptimal configurations with quantile filtering reduces the size $|C_{\mathrm{GC}}|$ of configurations the GC may generate. Recall from Table 5.2 that the tuning space coverage decreases dramatically after a certain quantile. The best filtering quantile will minimize KL divergence from the optimal distribution and limit the size of $|C_{\mathrm{GC}}|$ without overspecifying the search space since the latter also contributes to the probability of the few-shot success. We can determine the reduced $|C_{\mathrm{GC}}|$ from the GC by estimating the number of unique samples generated by the fitted GC. It is possible that this reduction excludes some ideal candidates, meaning that $|I_{\mathrm{GC}}| \leq |I|$.

The exact reduction in $|I|$ represented by $|I_{\mathrm{GC}}|$ is unknown but we may estimate the opportunity cost of some optimal configurations being removed by reducing $|I_{\mathrm{GC}}|$ by a factor of the ratio between $|C|$ and $|C_{\mathrm{GC}}|$. For our experiments, we cautiously assume 5% of the eliminated configurations from C are also members of I — this is five times the naively expected rate. With adjusted $|C_{\mathrm{GC}}|$ and $|I_{\mathrm{GC}}|$, the value of k in Equation 5.1 can be increased until the probability meets a desired confidence level. This provides an adequate budget of evaluations k that generates one or more ideal candidates with probability equal to the specified confidence (e.g., 95%). This budget-engineering calculation operates similarly to a convergence guarantee because it permits evaluations of the GC's viability via the size of its budget constraint without performing any empirical evaluations.

5.3.3 Addressing Limitations for Autotuning

Even with our modifications, a few of the known limitations of GC models have limited significance in our intended use case of TL autotuning for source code annotations.

5.3.3.1 Underfitting Cross-Variable Dependencies

The GC expresses codependence between variables using linear correlation, which will underfit complex variable codependencies. The GC's correlation is expressed between variable pairs, so the number of simultaneously interacting variables is less important than the complexity of dependence between variable pairs. In most cases for source-code autotuning, annotations are functionally independent of one another or adhere to the linear correlation that the model can express.

5.3.3.2 False Ordering and Transitivity for Categories

The GC's linearized representation of categorical values implies and attempts to leverage a total ordering that may not exist between categories. This creates transitive relationships that may prove counterproductive for the marginal optimization of categorical data. One way to counteract this behavior is to utilize binary expansion or one-hot encodings for each category, but this can create many variables when applied to large categories. Many source code annotations consist of only two values, such as the presence or absence of a `#pragma` annotation, which limits the variable to two categories. Other categorical variables in annotation autotuning are limited to fewer than ten values, which bound the error that marginal kernels must overcome to acceptable degrees.

5.3.3.3 Model-Fitting Complexity

Fitting a GC has cubic time complexity based on the number of variables due to the joint covariance model. Other TL methods gain a competitive edge when the GC models fifty or more variables, which can make some modifications, such as one-hot encoding, less desirable in practice. Source code annotations pose some inherent limits on the number of tunable variables due to the decreasing performance significance of additional, non-bottleneck optimization points in an application. Larger applications require explicit measures, such as importance sampling, to identify the most critical variables to tune. Our current techniques continue to rely on experts for annotation and can also rely on them to curate an appropriately sized set of variables.

5.4 Experiment Design

We evaluate our method and several existing techniques in few-shot TL autotuning with a variety of benchmarks empirically evaluated on a real system.

TL Autotuning Benchmarks We use source code modifications to the Polybench 4.2 [14] benchmark suite and several Exascale Computing Project (ECP) proxy mini-applications to evaluate our GC autotuning methodology. The Polybench and ECP benchmarks are multithreaded, and one (SWLite) is a GPU-accelerated program. The selected applications are based on our ability to define valuable optimizations in our tuning spaces, ergo we do not consider every application within either suite.

Polybench consists of numerical computation kernels extracted from various application domains. We utilize six of the most complex benchmarks spanning the application domains of linear algebra, image processing, stencils, and data mining: Syr2k, 3MM, Heat-3d, LU, Covariance, and Floyd–Warshall.

The ECP proxy applications represent essential computational kernels from high-performance computing programs, allowing for highly effective performance analyses and tuning without requiring the time-intensive execution of the entire application. We include four mini-applications — AMG, RSBench, XSBench, and SW4Lite — which feature different compute-memory access ratios and memory accesses patterns.

We parameterize each kernel with source code modifications in performance-critical sections of the benchmark that may improve performance. These modifications include tile sizes, loop optimization techniques, parallelization and scheduling strategies, data allocation formats, and multiprocess synchronization frequencies. Table 5.3 shows the number of unique parameters in each experimental benchmark as well as the combinatoric search space size of all possible parameter configurations. The largest search space has over 5 million potential configurations. The tuning spaces for Polybench and ECP applications are described in greater detail in Tables 5.4 and 5.5, respectively.

Source Tasks and Training Dataset To form the prior knowledge for TL autotuning, we use offline autotuning through YTOPT [16] to collect 200 evaluations in each of three non-target tasks: small, medium, and large. We use YTOPT with the Random Forest backend for source task tuning because it can be explicitly tuned to balance the degree of exploration (variability within the tuning space) and exploitation (optimization refinement) via a simple hyperparameter, ensuring the limited training data both informs techniques about the near-optimal performance while also representing some broader information about the performance relationship. The YTOPT autotuning is configured to target 90% confidence in establishing the 50^{th} quantile of performance. This forms a highly valuable training dataset for all techniques, but we limit these searches to 200 evaluations per task, no more than 5% of the search space coverage for any benchmark.

Table 5.3 summarizes the tuning spaces of source tasks and includes the GC's predicted evaluation budget based on filtered source data. The prediction is based on the model's capability to identify one or more evaluations in the top 1% with 95% confidence, assuming that as much as 5%

Table 5.3: Tuning spaces for each benchmark alongside the GC's coverage and budget based on the top-30% of source evaluations. Specific parameters are described in Tables 5.4 and 5.5.

Benchmark	#Params	# Configurations	GC Coverage	GC Budget
3mm	10	376,320	≈ 2,500	–
Covariance	5	5,324	≈ 110	–
Floyd–Warshall	5	5,324	≈ 1,800	15
Heat3d	6	10,648	≈ 1,600	8
LU	5	5,324	≈ 210	–
Syr2k	6	10,648	≈ 800	3
AMG	9	1,180,980	≈ 108,500	5
RSBench	9	5,196,312	≈ 316,800	3
XSBench	8	577,368	≈ 77,500	7
SW4Lite	8	4,752	≈ 1,800	15

Table 5.4: Parameters used to tune Polybench Kernels. Values within brackets indicate the options available for an independent parameter, and a list of brackets represents multiple independent parameters.

Parameter	3MM	Covariance	Floyd-Warshall	Heat3d	LU	Syr2k
Tile Sizes	[4-2048], [4-2048], [4-2048]	[4-128], [4-2048], [4-256]	[4-128], [4-2048], [4-256]	[4-128], [4-2048], [4-256]	[4-128], [4-2048], [4-256]	[4-128], [4-2048], [4-256]
Loop Interchange	[Yes, N/A]	[Yes, N/A]	[Yes, N/A]	[Yes, N/A]	[Yes, N/A]	[Yes, N/A]
Array Packing	[Yes, N/A] × 6	[Yes, N/A]	[Yes, N/A]	[Yes, N/A] × 2	[Yes, N/A]	[Yes, N/A] × 2

of pruned configurations were potentially optimal. A dash represents an unknown budget, where the overall problem size is reduced to such a degree that it is impossible to predict a budget requirement using Equation 5.1. In this case, the GC's tuning space coverage could fail to include the optimal region if the transfer relationship is poorly informed. Hence, we cautiously treat indeterminate budgets the same as few-shot TL for techniques that cannot determine their own budget, and we determine how well the GC can perform using the same budget as prior techniques.

[1] Specifically unroll loops by a factor of 6 iterations

Table 5.5: Parameters used to tune ECP mini-applications.

Parameter	AMG	RSBench	XSBench	SW4Lite
Tile Sizes	[10-200], [2-256], [2-256], [10-200]	[2-256], [2-256]	[2-256], [2-256]	—
Optional Parameters	Parallel For	Parallel For	Parallel For	Parallel For, Nowait, MPI_Barrier
Parallel For Schedule	—	[100-2000], [10-200]	[10-160]	[dynamic, static]
Unrolling Options	[unroll, N/A]	[unroll, N/A]	[unroll, N/A]	[unroll (6) [1], unroll, no-unroll]
# Threads	[4-8]	[2-256]	[2-256]	[2-256]
KMP Affinity	[compact, scatter, balanced]	[compact, scatter, balanced, none, disabled, explicit]	[compact, scatter, balanced, none, disabled, explicit]	—
OMP Proc Bind	—	—	—	[close, spread, master]
OMP Places	[core, threads, sockets]	[core, threads, sockets]	[core, threads, sockets]	[core, threads, sockets]

Compared Approaches We evaluate the following autotuning approaches to demonstrate our advantage in comparison to prior art:

- **Baseline.** Parameter values are taken directly from their respective sources; no parameter tuning is performed — this configuration represents one particular option within the tuning space and is selectable by all other techniques.
- **Bayesian Optimization.** Bayesian optimization (BO) without TL using YTOPT [16, 12, 13]. The autotuner utilizes a random forest surrogate model and a hedged Gaussian process to evaluate the expected improvement of proposed configurations. We utilize the same tuning hyperparameters as those used during source task collection, representing the capability of "starting from scratch" without transfer learning.
- **GPTune DTLA.** GPTune [15] with DTLA is a state-of-the-art autotuner that is capable of utilizing TL using a neural joint model to combine Gaussian processes representing individual parameters. Following the original paper authors' recommendations, we allocate the first half of the evaluations to be randomly sampled, then use GPTune's MLA to tune for the remaining half of the tuning budget.
- **Gaussian Copula (ours).** A GC is fit to the top 30% performing data from source tasks, then conditionally sampled on the target task to generate configurations. We determine the predictive budget but go beyond it to the maximum number of evaluations produced by other autotuners to demonstrate the reliability of the predictive properties.

Autotuning Procedure Each benchmark has three source task sizes (small, medium, and large) based on given magnitudes of performance-scaling input features. For the Polybench benchmarks, these sizes are pre-defined scales given for each benchmark; for the ECP mini-apps, we determine various inputs that are separated by similar scales of performance. We utilize transfer learning for each compared technique to optimize three novel target task sizes: small-medium (SM), medium-large (ML), and extra-large (XL). The first two target task sizes represent interpolations between source tasks, and the final target task is an extrapolation beyond the scope of source tasks. The Polybench/C applications define the extra-large task size but not the small-medium or medium-large sizes; for these, we linearly interpolate between the indicated sizes to produce new tasks. We also define the input sizes for the ECP target tasks, following similar scaling patterns for all three sizes.

While GPTune can optimize multiple target tasks simultaneously, we ignore this capability for fairer comparison to other works that only tune for one target size at a time. None of the compared techniques perform cross-benchmark training, so each target size is tuned independently with only training data from source tasks of the same benchmark. In order to permit the fairest possible comparison among different techniques, the same source dataset is presented to each transfer-capable technique in the appropriate format for each technique, but the GC filters source datasets for its benefit as described in Section 5.3.

We permit each autotuning technique a fixed budget of at most 30 evaluations per target task. To mitigate the variance of empirical measurements on the system, during both source and target task tuning, each code configuration is compiled once and evaluated three times as a "single" evaluation for purposes of tuning budgets. The autotuning objective is reported as the mean of the last two evaluations, with timing data collected internally within each benchmark to ensure that overheads such as process startup and data initialization are excluded. In order to mitigate the variance of randomness employed by each compared tuning technique, the entire few-shot tuning process is repeated with three random seeds and results are reported using the average across all seeds. Even when the GC can predict a viable budget of fewer than 30 evaluations shown in Table 5.3, we collect all 30 and specifically note the intermediate results when the predicted budget is exhausted. Since we expect TL techniques to extract some understanding of the problem from prior data, we evaluate success primarily based on the best-observed performance among the limited target task evaluations.

Experimental Platform All experiments are conducted on a Linux machine with 320 GB 2x AMD EPYC 7742 64-core processor (128 total cores) 1 TB DDR4 with Ubuntu 20.04.2 LTS. The machine also includes a 40 GB NVIDIA A100, which we use for evaluating the GPU-based SW4Lite ECP application. Measurements of elapsed time during the tuning process include time for sample generation, source code compilation using the Clang compiler, and program execution. Each benchmark internally measures empirical performance.

Because the tuning spaces we defined express optimizations through Polly [61], a loop optimizer for LLVM, we use a Clang compiler (version 13.0.0) for compilation. However, any compiler that supports Polly is suitable for replicating our experiments. Some Polly optimizations can be applied heuristically based on analysis of the LLVM intermediate representation, while others can

be induced by programmer-supplied `#pragma` directives in the source code. Currently, not all code transformations can be specified by directives, such as unroll-and-jam, loop fusion, and code motion. For this reason, two of our applications (3mm and LU) adopt heuristic optimizations.

5.5 Input-Scaling Experiments

We separate the presentation of our results between the Polybench and Exascale benchmark suites and identify key successes and limitations of our technique compared with the state-of-the-art approaches.

5.5.1 Polybench Autotuning

The Polybench benchmarks demonstrate several different behaviors for generative autotuning with the GC, including aggressive space pruning, uncertain optimization signals, and high-confidence benchmarks that represent a best-case scenario for the technique.

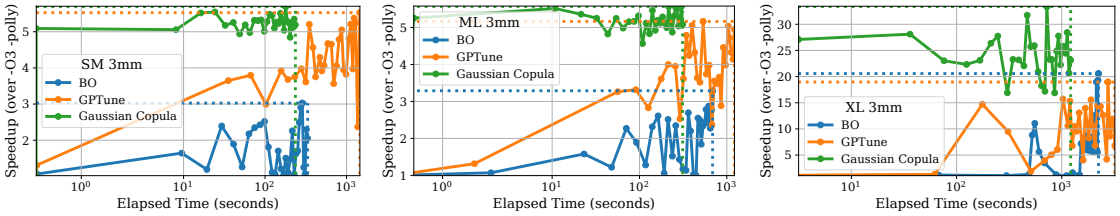


Figure 5.2: Observed speedup vs. log-scale elapsed time for few-shot TL autotuning. The dotted lines indicate results trimmed to the GC's predicted budget.

General results for the Polybench benchmarks are presented in the upper portion of Table 5.6. On the 3MM XL task, the GC yields an additional 12.81× speedup (i.e., 33.39× vs. 20.58×) compared with prior autotuning techniques. In half of the Polybench tasks, the GC's first evaluation outperforms the best tuning result discovered by BO or GPTune. When we utilize the GC's expected budget or the maximum number of evaluations whenever the budget is undefined, the GC outperforms GPTune and BO in over 80% of all tuning tasks. Even on tasks where the GC does not outperform prior work, the peak speedup sampled by the GC is within 5.5% of the peak performance sampled by prior work.

The GC is highly successful both on its first evaluation and within its allotted evaluation budget because of the effectiveness of its search space reductions and distribution transfer through

Table 5.6: Autotuning results after a maximum of 30 evaluations; results are averaged across three repeated tuning attempts with unique seeds.

	App.	Scale	Peak Speedup (# Evaluation Discovered)				
			GC			BO	GPTune
			1^{st}	Budget	Best	Best	Best
Polybench Kernels	3MM	SM	5.09	5.70 (23)	5.70 (23)	3.03 (26)	5.53 (30)
		ML	5.25	5.57 (29)	5.57 (29)	3.29 (30)	5.16 (16)
		XL	27.10	33.39 (18)	33.39 (18)	20.58 (30)	18.96 (25)
	Cov.	SM	21.10	21.98 (21)	21.98 (21)	21.83 (28)	13.30 (30)
		ML	4.13	4.27 (26)	4.27 (26)	3.87 (25)	4.07 (30)
		XL	23.04	23.96 (2)	23.96 (2)	8.43 (12)	17.88 (9)
	Floyd-W.	SM	1.01	1.02 (17)	1.02 (17)	1.02 (20)	1.01 (26)
		ML	1.02	1.02 (1)	1.02 (1)	1.01 (25)	1.01 (3)
		XL	0.99	1.00 (29)	1.00 (29)	1.01 (16)	1.01 (20)
	Heat3d	SM	1.83	2.03 (5)	2.06 (18)	2.21 (15)	2.30 (28)
		ML	1.89	1.89 (1)	2.06 (10)	2.12 (25)	1.80 (6)
		XL	1.50	2.92 (2)	3.09 (18)	2.16 (13)	2.75 (29)
	LU	SM	1.16	1.18 (25)	1.18 (25)	1.12 (30)	1.11 (19)
		ML	1.15	1.20 (24)	1.20 (24)	1.17 (26)	1.19 (5)
		XL	1.00	1.00 (3)	1.00 (3)	0.98 (13)	1.00 (29)
	Syr2k	SM	2.06	2.90 (2)	3.32 (18)	2.34 (12)	2.41 (11)
		ML	0.80	1.17 (2)	1.22 (16)	0.93 (29)	0.85 (30)
		XL	0.95	1.09 (2)	1.09 (2)	0.42 (23)	0.85 (26)
Exascale Computing Proxies	AMG	SM	0.87	0.91 (3)	0.91 (3)	0.92 (19)	0.90 (19)
		ML	0.93	0.93 (1)	0.93 (1)	0.93 (20)	0.87 (3)
		XL	0.95	0.95 (5)	0.98 (23)	0.97 (27)	0.93 (25)
	RSBench	SM	1.40	1.40 (3)	1.40 (8)	1.25 (29)	1.13 (22)
		ML	1.02	1.04 (2)	1.04 (15)	0.97 (22)	1.04 (27)
		XL	1.00	1.00 (1)	1.01 (10)	0.97 (14)	1.02 (18)
	XSBench	SM	1.20	1.20 (7)	1.21 (28)	1.17 (24)	1.21 (24)
		ML	1.05	1.06 (4)	1.06 (4)	1.04 (6)	1.07 (5)
		XL	1.01	1.02 (5)	1.03 (24)	0.99 (6)	1.05 (5)
	SW4Lite	SM	0.99	1.00 (6)	1.00 (6)	0.98 (26)	0.99 (17)
		ML	0.99	0.99 (10)	0.99 (16)	0.99 (3)	0.99 (30)
		XL	0.99	0.99 (12)	0.99 (12)	0.99 (1)	0.99 (14)

conditional sampling. Both GPTune and BO must allocate portions of their evaluation budget to explore the space and refine the model's transfer or general surrogate knowledge. The GC does not need to perform these subpar evaluations, and immediately focuses on leveraging source data to locate and improve the optimum of the transfer task. This focus allows the GC to be extremely aggressive in the few-shot tuning, as shown in Figure 5.2 where nearly every proposed evaluation of the GC outperforms all evaluations proposed by other tuning methods.

The GC prunes spaces for 3MM, Covariance, and LU too aggressively for us to predict an evaluation budget. Our results demonstrate that the GC still identifies the best speedup across

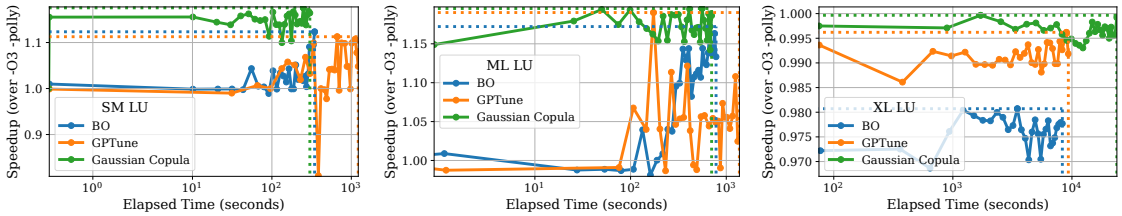


Figure 5.3: Ambiguous responses to tuning yield minimal speedup, but the GC remains competitive with prior work.

all techniques in all tasks for these benchmarks when given the same tuning budget allocated to other techniques. The search space reduction performed by the GC outperforms prior autotuning by properly identifying characteristics of optimal configurations across tasks and correctly modifying these relationships for each target task.

The Floyd–Warshall and LU benchmarks are challenging for any autotuning technique to optimize. Without exhaustive data for these benchmarks, it is unclear whether this is due to the original source code parameters being near-optimal or the tuning space exposing mostly unhelpful alterations to the benchmark source. Critically, the GC still produces highly consistent and comparatively valuable results on each evaluation, as shown for the LU benchmark in Figure 5.3.

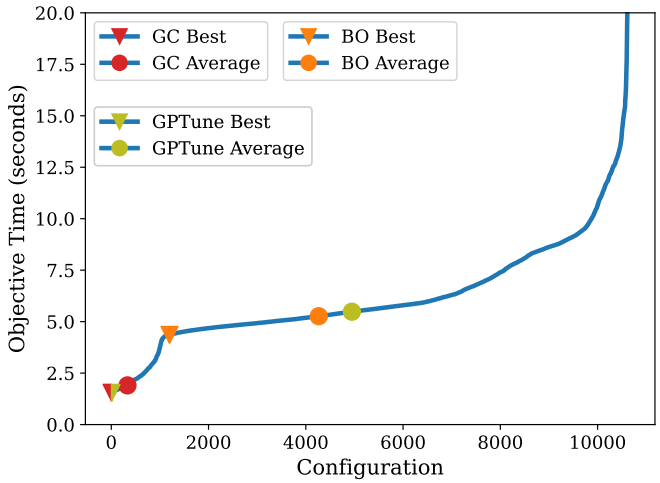


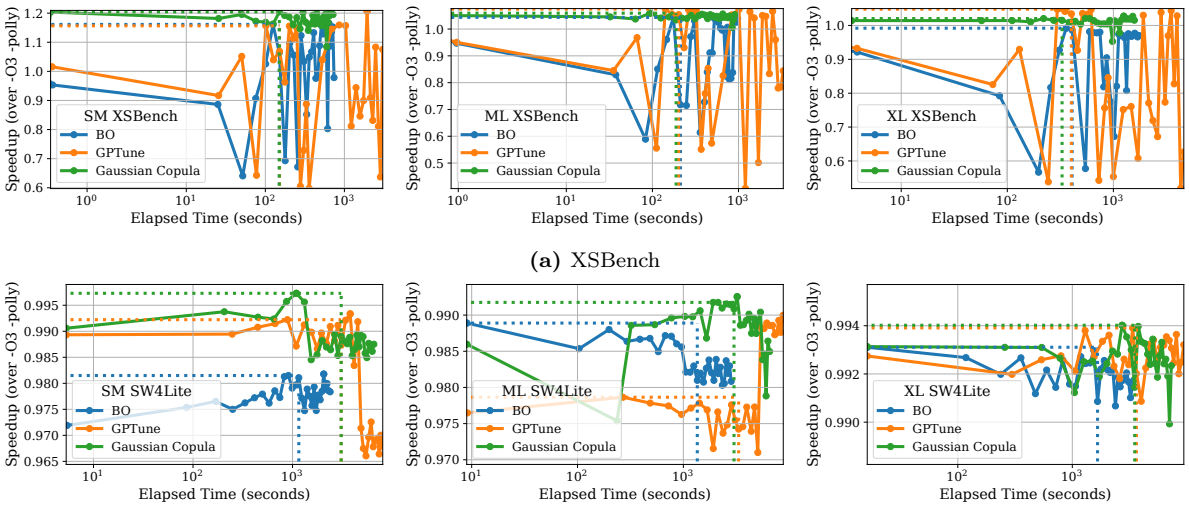
Figure 5.4: Brute-forcing the Syr2k XL task proves that the GC and GPTune can identify the global optimum in 30 evaluations, but the GC avoids poor evaluations, giving it better average performance.

To ensure that few-shot TL autotuning is effective, we brute-force all configurations of the Syr2k XL task in Figure 5.4. Both the GC and GPTune closely approximate the global optimum within a few shots. However, all evaluations proposed by the GC are near-optimal, while other

methods require repeated exploration of poor-performing regions to identify their transfer relationships.

5.5.2 Exascale Miniapplications Autotuning

The selected exascale benchmarks represent the most significant challenge for few-shot TL autotuning, with search spaces that are orders of magnitudes larger than those present in the Polybench kernels and complex interplay between many variables. We expect less speedup from autotuning spaces for these applications for several reasons. First, the tuning spaces are orders of magnitude larger than Polybench tuning spaces; we use the same number of source task evaluations for all experiments, which means that TL operates on less complete information about each ECP tuning problem. Second, for more advanced applications, it is more challenging to represent highly effective tunable optimizations than the more straightforward Polybench kernels. Third, some speedup from system-related tuning parameters can be hidden by other tuning parameters. The choice of core affinity, for example, has a greater impact on performance if the configuration also includes many threads. Finally, some parameter defaults, such as loop tiling values, are already highly effective, which limits the improvement that can be extracted from the tuning space. Although we temper our expected improvement from autotuning, these experiments represent more realistic tuning scenarios where autotuning refines more complex and partially optimized code.



(a) XSBench

(b) SW4Lite

Figure 5.5: The GC remains competitive with state-of-the-art techniques on complex ECP benchmarks.

Even though our GC technique cannot leverage information gained through iterative evaluations, the technique meets or exceeds the original expert-optimized performance on over half of the exascale tuning tasks. The AMG task is the most difficult for any technique to optimize, but the GC outperforms GPTune either from its first evaluation or within its predicted budget for all transfer tasks. Even as the relationship between parameters and performance becomes more complex and search spaces grow orders of magnitude larger, the GC can identify high-performing traits in prior data and produce high-quality candidates in the few-shot tuning scenario. Across all exascale benchmarks, the GC produces configurations within a performance margin of 2% of those discovered by GPTune at worst. Notably, GPTune's best evaluations for two XSBench tuning tasks are better than ours, but the superior evaluations are collected during its random sampling for the new task, as shown in Figure 5.5a. This may indicate that the prior tuning data does not adequately inform autotuning techniques of characteristics of the optimum for this benchmark.

We also note that the GC retains the black-box characteristics enjoyed by prior methods such as BO. Unlike other benchmarks in this work, SW4Lite is a GPU-enabled benchmark, and the tuned kernel is executed on GPU hardware. As shown in Figure 5.5b, the GC evaluates higher-performing configurations than exploratory techniques such as GPTune do. The proposed tuning budget is also reliable across multiple seeds, such that the GC reliably makes its best evaluation within the budgeted number of evaluations. If much larger budgets are allowed, the GC has less chance of improving than other TL autotuning techniques have. In such cases, or if any possible performance gain is desired, our technique may be best utilized to perform initial exploration of new spaces within a limited few-shot budget to bootstrap iterative techniques.

5.6 Extension to Multi-Scale Tuning

Our success in tuning for fixed hardware configurations at different scales shows the promise of the GC as an autotuner. However, most HPC programs scale hardware and software together, which vastly complicates the performance relationship. To this end, we design an additional experiment to evaluate the GC's suitability for this mode of multiscale tuning.

Given the complexity and ambiguity of the single-node scale ECP mini-application results, we seek a different application that can be expected to provide clearer insights to the capabilities of compared autotuners. For this purpose, we select the Highly Efficient Fast Fourier Transform for

Table 5.7: Hardware details for the ALCF Polaris system.

	Type	Details
CPU	AMD EPYC Milan 7543P	32-cores at 2.8 GHz
GPU	NVIDIA A100	40 GB HMB2 Memory 9.7 TFLOP/s peak FP64 and 19.5 TFLOP/s peak FP32

Exascale (heFFTe) [62].

To support multi-scale evaluations more effectively, we integrate GCTLA into YTOPT's LibEnsemble library [63], which allows for parallel evaluations across resources. As a result, portions of the source code for these experiments are contained in a branch of the LibEnsemble library located at `https://github.com/tlranda/ytopt-libensemble/tree/multiscale`.

5.6.1 Evaluation System

We utilize a different system to support larger-scale tuning for this benchmark: Polaris. Polaris is a cluster at Argonne National Laboratory's Leadership Computing Facility (ALCF). The hardware details are provided in Table 5.7. The supercomputer is organized with two nodes per chassis, seven chassis per rack and a total of 40 racks. GPUs within the same chassis are connected by 600 GB/s NVLink, other connections are made over 64 GB/s PCIe with a pair of Slingshot 11 network adapaters.

5.6.2 Benchmark Application and Tasks

The heFFTe software package includes a benchmark application, speed3d_r2c, that benchmarks a GPU-aware MPI implementation of the Fast Fourier Transform algorithm commonly utilized in scientific applications. The speed3d_r2c application organizes a 3D FFT computation between nodes and devices, but utilizes a backend library with 1D FFT support to execute the majority of the work. For the GPU architectures, the 1D FFTs are handled by the cuFFT backend [64].

We represent each task in the space by the number of compute nodes and each dimension of the FFT input volume and perform weak data scaling between the problem size and number of compute nodes. The full set of tasks are detailed in Table 5.8. Because the input problem is defined as a 3D FFT, we double the size of one dimension of the FFT for each doubling of the number of nodes involved in the computation. We alternate the axes that are doubled in round-robin order (ie: Z-axis, then Y-axis, then X-axis) to ensure tasks remain relatively consistent in terms of the data

Table 5.8: Tuning tasks are weakly scaled across FFT sizes and compute devices. The observed variability in performance is presented in GFLOP/s based on empirically observed values. The GCTLA budget estimate per-task is also provided.

# Nodes	Total GPUs	# MPI Ranks	FFT Dimensions			Worst Known Performance	Best Known Performance
			X	Y	Z		
2	8	8	256	256	256	53.7	1,308
4	16	16	256	256	512	86.0	2,233
8	32	32	256	512	512	160.5	4,170
16	64	64	512	512	512	174.6	8,664
32	128	128	512	512	1024	146.9	12,018

management requirement and constraints. We assign one MPI rank per GPU (four per compute node) and double the number of nodes to weakly scale the computational resources. As the single-node performance can fully rely on the NVLINK interconnect between devices with much higher bandwidth, its performance would be a sharp outlier compared to multi-node decompositions that must communicate some data over PCIe. We use no fewer than two nodes so that this complex change in behavior does not need to be modeled by the transfer learning techniques. We are not aware if jobs are physically scheduled to the same rack or chassis, especially at lower ranks, but make efforts to have our jobs scheduled within the same chassis or rack wherever possible. Because of the system's hardware configuration, the 16- and 32- node problems are guaranteed to scale beyond a single rack regardless of job placement on nodes.

We utilize the benchmark's reported throughput (in GFLOP/s) as the metric of merit to be optimized. As shown in Table 5.8, the range between the best and worst performing configurations grows faster than the floor performance. This underscores the importance of transfer tuning for HPC, as tuning on smaller subsets of the cluster can be done at larger scale to rapidly identify performance behaviors at much cheaper cost than the largest-scale evaluations of interest. If effective, transfer learning can provide massive leverage while reducing the overall resource cost associated with the tuning process.

5.6.3 Tuning Space Definition

The tuning space for heFFTe is expressed by a selection of runtime flags.

Precision. While precision is not often a tunable component, we permit autotuners to freely select between single- and double- precision.

Reordering. This flag can be enabled or disabled; enabling it will reorder all data to be

contiguous before calling the backend library. The heFFTe authors default to *not reordering* for the cuFFT backend.

MPI Communication Strategy. This option determines the kind of MPI collective communications to utilize amongst All-to-All, All-to-All-v (vector), Peer-to-Peer, and Peer-to-Peer-Pipelined.

Reshaping. This option determines the shape of intermediate steps of the computation to be either pencils (1D) or slabs (2D). Slabs are supposed to offer higher performance when fewer MPI ranks participate in the problem, while pencils should dominate performance at high numbers of MPI ranks.

Conversion Direction. The FFT can be performed as a complex-to-complex (general FFT), real-to-complex (forward FFT), or complex-to-real (inverse FFT). These represent different flavors of problems FFTs solve, but since we are focused on maximizing overall throughput the autotuner must determine if the average performance is improved or not by any particular FFT.

Virtual MPI Topologies. The virtual topology for MPI communications does not necessarily layer one-to-one over physical hardware topology, but organizes computations between computing elements. There are two virtual MPI topologies to separately define the in-grid and out-grid specifications. Each topology is defined in three dimensions which map to a decomposition of the 3D FFT's volume, changing the message sizes and volumes between various ranks. The virtual topology has the greatest impact on performance as it affects network latency and bandwidth, but has the least guidance from the heFFTe authors for effective tuning. The number of unique virtual MPI topologies are task-dependent, ranging from 10 to 36 within our experiments. We exclude topology selections that do not utilize all available compute units (under-specified) as well as topology arrangements that are transpositions of previously-defined topologies (functionally identical) as the virtual-to-physical mapping cannot guarantee consistent differences in behavior for re-arranged axes.

Overall Tuning Space. Excluding the two parameters defining virtual MPI topologies, this tuning space has a size of 96 elements. For the smallest task with two nodes (10 virtual topologies), the total search space includes 9,600 configurations. For the largest task with 32 compute nodes (36 virtual topologies), the total search space includes 124,416 configurations.

5.6.4 Multi-Scale Results

We find that GCTLA and GPTune are largely incapable of performing transfer-tuning for weak-scaling across hardware and software. However, there is reason to believe that GPTune will improve with additional target task data while GCTLA is less likely to significantly change with additional target task evaluations.

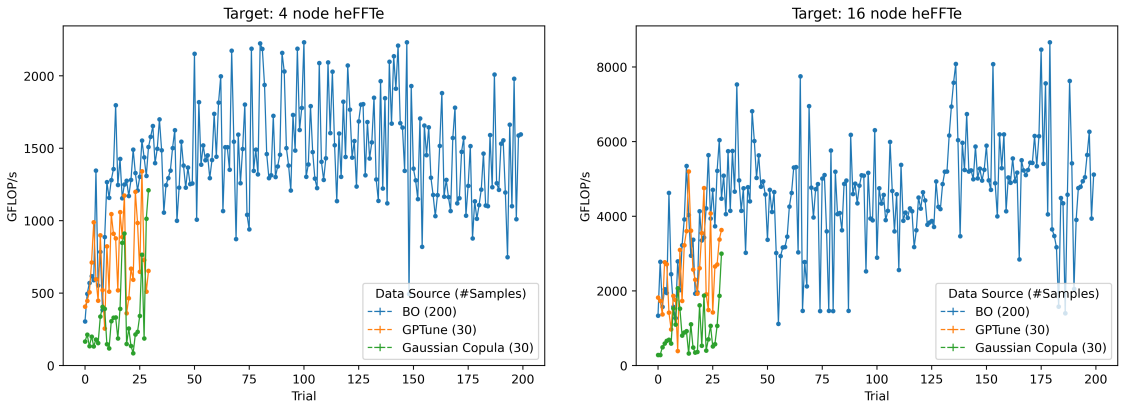


Figure 5.6: Left: GCTLA's best performance is quite underwhelming in the transferred task domain. Right: GPTune is also out-performed by Bayesian Optimization from scratch on most datasets.

Figure 5.6 shows how our GCTLA technique fails to generalize performance from source to target tasks. The average sampled configuration is worse than GPTune's initial random exploration and the efforts of Bayesian Optimization from scratch. While GPTune performs better than GCTLA on this problem, it is also out-performed by restarting Bayesian Optimization from scratch. This suggests that the transfer relationship is too complex to be learned from smaller-scale source data. Even for sophisticated techniques such as GPTune the optimization efforts are most productive when exploring the new task.

Target Task	GCTLA Peak Relative to BO Peak	BO Trials to Exceed Best GCTLA	GPTune Peak Relative to BO Peak	BO Trials to Exceed Best GPTune
2 Nodes	39.7%	1	94.4%	32
4 Nodes	54.2%	5	60.1%	5
8 Nodes	38.6%	1	66.0%	14
16 Nodes	34.6%	5	60.0%	13
32 Nodes	86.6%	59	145.3%	N/A

Table 5.9: Optimal performance of transfer learning searches as a percentage of the highest performance identified by Bayesian Optimization from scratch, as well as the number of Bayesian Optimization evaluations needed to outperform the transfer learing search's best result.

Table 5.9 shows the relative advantage of Bayesian Optimization over each transfer learning technique across all target tasks. GPTune is able to outperform GCTLA on all tasks as the transfer learning relationship is not well-formulated, but Bayesian Optimization further outperforms GPTune on all but one task. Typically, Bayesian Optimization is able to exceed the performance of the transfer learning search within the number of alloted evaluations for each transfer learing search. This strongly demonstrates that the knowledge between tasks is harmful to the searches and validates that GPTune's continuous learning capability allows it to gradually recover as its search progresses. GCTLA has no such mechanism and is unsuitable for autotuning when the transfer learning relationship must be learned within the target task.

5.6.4.1 Cross-Task Replication

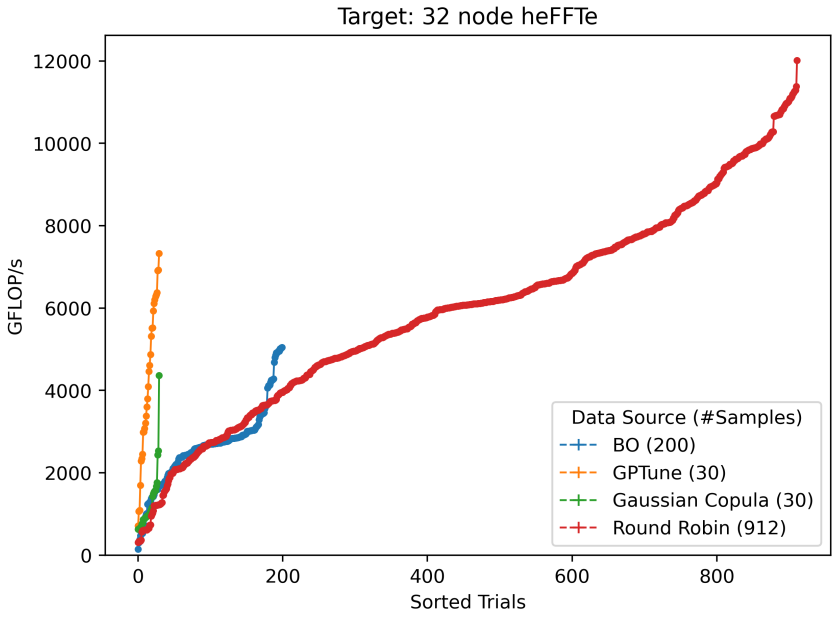


Figure 5.7: When comparing against the extended dataset, it is obvious that the transfer learning techniques have failed to identify high-quality mappings between source and target tasks.

To further explain the discrepancies between these results and our prior experiments, we replicate observed configurations on a closest-match effort between all tasks. We cannot directly map the virtual MPI topology of each observed configuration to all other tasks because the topologies are dependent upon the total number of MPI ranks. Instead, we preserve all other configuration values and convert the the MPI topology to the new task's topology space based on its rank-normalized

$\log_2$ scale in the original task. This conversion process allows us to closely match topology features between scales by preserving the relative difference between axes while still utilizing all ranks as required for use in the new task.

The resulting dataset allows us to inspect changes in the performance relationship between tasks without relying on an evaluation of the entire search space. As shown in Figure 5.7, even in tasks where it appears that transfer learning techniques have provided high levels of utility to the search, a broader perspective reveals these results to be at best half of the optimum value. While GPTune strongly outperformed Bayesian Optimization, the extended evaluation set shows us that other configurations in the search space deliver 1.64X more throughput than the GPTune optimum.

5.6.4.2 Transferability Analyses

Our expanded dataset allows us to try and understand what limitations prevented transfer learning from performing as desired. We observe that the relative importance of each parameter value for high throughput computations are stable across different scales. This means that the transfer learning techniques remain capable of identifying the most important parameters and these assumptions are not misleading. Rather, the behavior within these highly important parameters must change between tasks for the transfer relationship to be under-constrained.

Lower impact parameters include precision, reordering, conversion direction, and reshaping (in least-to-greatest order). These parameters have nonzero impact on the final throughput, but are frequently dominated by other parameter selections and only need to tuned once a local optimum is identified. Both GCTLA and GPTune consistently make reasonable selections for these parameters, supporting these techniques' capability to identify transferrable components from source task data.

The high-impact parameters in least-to-greatest order are the MPI communication strategy, the in-grid topology and the out-grid topology. Unlike our previous transfer learning benchmarks, the relationship between these three parameters are highly complex — a near-optimal value for one parameter in absentia of near-optimal values for other parameters will result in poor performance. In other words, these separate parameters require joint-tuning of highly complex nature.

This requirement for joint tuning of parameters with a highly complex relationship far outpaces the capabilities of GCTLA's covariance modeling, which can only represent pairwise correlations between variables. If the highest-performing configurations happened to be near-identical, the GC may be able to model such behaviors, but this is not the case for heFFTe's optimization and

the model is too underfit to properly represent the behaviors observed in training data. GPTune also fails to model this degree of complexity with the minimal dataset, but shows some signs of breaking through to partially model the in- and out-grid parameters to some effect. With additional training data and a longer tuning budget for the target tasks, it is likely that GPTune will successfully identify the high-performing regions of each new task, but we do not experimentally validate this expectation.

5.6.5 Lessons Learned

The primary challenge for effective multi-scale tuning of this benchmark lies in effectively determining the in-grid and out-grid topology shapes. The complexity of these inter-dependent variables proves to be too difficult for the Gaussian Copula to model. While some autotuning budgets were indeterminate in our input-scaling experiments, the Gaussian Copula proved capable of providing a surprising amount of utility. However, this represents a case where an indeterminate autotuning budget is a good indication that the GCTLA is not well-suited for use in the problem. We also manually check the covariance matrix and determine that the fitted Gaussian Copula finds near-zero correlation between the topology parameters. Several other variable pairs have similar near-zero correlations as they are truly decoupled variables, but observing this behavior for variables with known interdependency may serve as an additional warning that the Gaussian Copula model is incapable of representing the complexity of behaviors in the training data.

5.7 Related Work

Prior TL autotuning has enabled data reuse on related tasks for increased sampling efficiency and reduced modeling overhead. BLISS [50] attributes significant cost to tuning multiple models for large-scale applications but also demonstrates that it is difficult to generalize between small datasets and the full range of potential performance. Other work [65] employs cost models to substitute cheaper sources of information and utilizes TL to generalize information as needed. However, in situations with a limited budget, the cost model is less relevant to the target problem and requires model reconstruction. Projecting an optimum via machine learning techniques such as GPTune [66, 15] enables more budget optimization for few-shot transfer, but these models require blind evaluations in each new task to form the basis for the transfer relationship. Other works such

as Active Harmony, ANGEL, and ParEGO [54, 55, 56] focus on multiobjective efficiency by refining a surrogate Pareto frontier. These algorithms provide stronger long-term convergence guarantees rather than few-shot performance. Our work permits immediate access to the most efficient samples through conditional sampling, allowing for aggressive few-shot tuning.

Prior works have also used biased sample distribution and importance sampling to increase autotuning capabilities. Marathe et al. [67] found that the correlation between different input scales and available parallelism improves performance predictions. Their work, however, intends to optimize for common-case average outputs and cannot drive the search aggressively. GEIST [68] transforms the problem of bias and variance in parameter spaces into undirected graphs and reframes the optimization problem into predicting labels for high or low performance. The autotuning framework Tuneful [57] utilizes incremental sensitivity analysis in BO and explicitly utilizes importance to identify performance trends. Chen et al. [58] use random forest importance measures in massive search spaces by limiting the number of simultaneously tuned parameters to permit full-space exploration. Our biased generative GC reinforces *and* benefits from increased likelihood to sample the most important parameters of a search space.

Copulas have been reported in the literature as part of an autotuning process. Salinas et al. [69] used the GC process to bootstrap expected improvement from a small number of initial samples in a BO framework based on ranked quantiles. More recently, Zhang et al. [70] utilized the correlation identified by a GC to explore the multiobjective Pareto frontier. Both studies used the GC to aid the BO process in TL. Salinas et al. [69] used GCs to build an expected improvement autotuning model with minimal initial random samples or prior data and iteratively refit the model as information became available. The effectiveness of copulas in these techniques is limited to variable correlations in relatively low degrees.

Our work uses the traditional GC with some modifications from the SDV [59] implementation. While our experiments do not yield evidence that special care in dependence modeling is necessary, we note that different copulas or GCs are available [71]. Users can select between variations better suited for tail distributions for which Pearson correlation is insufficient to describe covariant behaviors jointly.

5.8 Conclusions

In this chapter, we propose the GC as the first generative TL-based autotuning technique. Our technique aggressively searches for best-performing configurations in few-shot settings using quantile filtering and conditional sampling to bias distributions learned by the GC model. We provide the first TL-based autotuning technique that includes a budget-identifying measure to predict the expectation of few-shot performance. We then evaluate our technique on various real-world benchmark applications, demonstrating remarkable effectiveness in few-shot TL settings where continued explorations of benchmark characteristics performed by other methods are wasteful resource expenditures.

We also report the limitations of our technique and others for simultaneous tuning across hardware and software scales. Our experiments and post-hoc analyses indicate that these problems underscore yet-to-be-solved challenges for transfer learning autotuning. In particular, we identify tight parameter dependencies with highly complex interactions that not only exceed the simple modeling capabilites of the Gaussian Copula, but also challenge iterative learning techniques such as GPTune and Bayesian Optimization.

Many additional avenues remain open to future research with this generative TL-based autotuning framework. We believe the GCTLA may be usable in multiobjective tuning problems. It may also be possible to replace the covariance matrix with higher-order modeling that still permits usage of the probability integral transform for conditional sampling, increasing the utility of GCTLA on highly-complex tuning problems.

Chapter 6

Conclusion

This work emphasizes the importance of specific and holistic perspectives on computing performance. We demonstrate these perspectives in the forms of hardware-granted opportunities, software-driven leverage of hardware and integration-based optimization across both fronts, especially within the field of High Performance Computing.

Specifically, we explore hardware-granted opportunities in the form of SPLIC hardware developments. This technology promises greater energy efficiency in cooling systems while theoretically supporting higher computing density and software performance relative to familiar air-cooled environments. For software-driven leverage of hardware, we optimize the performance of Word2Vec based on core principles of GPU hardware implementation and strengths, permitting both retroactive and proactive performance boosts between architectures without need for specialized code modifications for individual hardware iterations. Finally, we present a novel integration technique in the Gaussian Copula for Transfer-Learning based Autotuning. This technique opens the door for Generative Autotuning, an entirely new approach to optimization that leverages knowledge of both hardware and software at their integration points on systems. Below, we provide additional discussion of these findings and contributions in addition to opportunities for future work.

6.1 Discussion of Key Findings and Contributions

Impacts of Immersion Cooling on HPC Applications Our case study of the Submer Smartpod v3 SPLIC immersion tank provides empirical insights into actual hardware and application

behaviors in relation to cooling demands and activities in small-scale clusters.

We demonstrate that the liquid coolant shares more heat between hardware components than air-cooled systems, possibly leading to hotter operating conditions for certain components than are typical. This is primarily driven by the thermal capacity of the SPLIC fluid volume and limited heat exchange. We do not observe adverse performance impacts due to insufficient cooling for any applications, spanning memory and compute bottlenecks for CPUs and GPUs alike. This suggests some minimal risk under normal operating conditions despite the general heat accumulation within the system. However, we also observe that the high-energy hardware components can reach critical operating temperatures without heating enough of the fluid volume to trigger a response from the tank controller, which could damage equipment such as GPUs over prolonged periods of intense usage. Mitigating this possibility requires more adaptable controls and direct integration between hardware temperature sensors and coolant control systems, but can be somewhat controlled by under-estimating the coolant temperature's set point at the cost of some energy efficiency gains that could otherwise be exploited. Finally, we observe small defects in current technology's operation and amenability to use with commodity hardware. The software and firmware defects will likely be addressed over time, but other aspects such as the plasticizer included in coolant may continually necessitate specialized care and alternative hardware purchases catered for usage in SPLIC systems.

Lasting GPU Acceleration: A Case Study in Word2Vec Optimization Novel hardware can provide many new opportunities, but if software is not positioned to leverage these opportunities then these developments will under-serve actual use cases. We demonstrate this effect through previous iterations of the Word2Vec algorithm on GPUs, which were outperformed by CPUs despite massive performance improvements for GPU accelerators relative to CPU architectures.

Our FULL-W2V implementation of the Word2Vec algorithm for GPUs relies on keen insights into the continued developments and opportunities provided by GPUs coupled with deeper knowledge of the algorithm itself to permit continuously scaling performance. We are the first to explicitly recognize and leverage the *independence of negative samples* to permit register-level caching for GPUs and increase the algorithm's arithmetic intensity on GPUs from 11.2 to 24.9. Additionally, we identify memory latency on GPUs as the primary bottleneck of performance and introduce *lifetime reuse of context words* to reduce overall memory demand of the algorithm by 91%. Together, these optimizations lead to 5.44X speedup over CPU implementations and 8.65X speedup over prior GPU

implementations. We further demonstrate how our approach scales across generations of hardware, with automatic 2.97X speedup between NVIDIA P100 and V100 GPUs. All of these improvements are made with trivial cost to the quality of results despite several semantically significant alterations to the algorithm.

Efficient and Transferable Multi-Scale Performance Autotuning When software is well-designed to leverage opportunities provided by hardware, there are a myriad number of configurations that must be tuned to achieve maximum performance. This tuning can be extremely costly, especially when data is not reused between highly related tuning efforts. To meet this need, we proposed a novel automatic tuning technique called Gaussian Copula for Transfer-Learning Autotuning, which is specifically designed to meet common performance tuning requirements found in HPC optimization.

Our technique relies on correlating similarities between high-performing configurations discovered by prior tuning efforts, leveraging this prior knowledge to rapidly identify high-performing configurations for new, related tuning problems. Furthermore, as a generative autotuning technique, we demonstrate the novel and largely reliable capability to predict the success of tuning under given conditions before making a single evaluation in the new domain. The input-scaling experiments demonstrate how this technique exceeds the capability of prior autotuners in 80% of all tasks, yielding as much as 12.81X additional speedup on Polbench benchmarks. For the remaining 20% of tasks, our technique remains competitive with prior art by suffering no more than 5.5% performance degradation compared to the best-known performance. We then demonstrate the limitations of our technique in multi-scale autotuning, where hardware and software are jointly tuned. Indeed, this problem remains challengeing at a fundamental level and requires additional research to propose novel solutions. Fortunately, our model is able to indicate its difficulty optimizing these problems prior to beginning tuning efforts, an affordance that other autotuners cannot provide.

6.2 Recommendations for Further Research

Each chapter of this manuscript presents opportunities for additional novel research.

Impacts of Immersion Cooling on HPC Applications Our experiments within a single SPLIC environment can be studied in a variety of other environments where additional questions can be answered. In particular, several companies provide forced induction compartments for SPLIC

with localized pumps to increase the coolant flow rate around critical hot spots. This may address the system's ability to respond to high-demand components more appropriately while also providing the means to more directly monitor local thermal conditions, especially for the high-heat components like GPUs as we identified in Chapter 3.

Our experiments are also limited in scope to the local demands and effects within a single chassis or cooling tank. For instance, we were not able to fully account for all water and air flow in either system. Expanding the scope to account for more external measurements of air, water, and power usage involved in components such as evaporators and Computer Room Air Conditioning would present a far more complete understanding of current energy efficiency than the estimations provided in this work. Performing these analyses during experiments that scale to all available resources (such as in benchmarks like the Top500 and Green500) present additional opportunities to gain insight over the entire system's demands when servicing full-scale workloads.

Based on our observations of possible divergence between hardware temperatures and coolant temperatures, there appears to be ample opportunity for improvements to the firmware monitoring and control of this particular SPLIC product. Efficiently predicting and managing responses will require additional research in real-time thermal modeling and integration between sensors. These models could exist within a similar framework as the monitoring tools built in our work, but are most useful when fully integrated into the coolant firmware for automatic control. Better predictions of thermal activity will also permit better responses from the coolant system, which will necessarily rely upon improvements in predictive energy modeling and job scheduling.

Finally, we believe our work forms a foundation for benchmarking the cooling capabilities and application demands of a fixed hardware-software benchmark. This could be further developed into benchmarks for cooling power efficiency, thermal dissipation capability and responsiveness, all of which will remain key distinguishing capabilities of SPLIC technology for the foreseeable future.

Lasting GPU Acceleration: A Case Study in Word2Vec Optimization While Word2Vec has received less attention in the wake of LLMs and transformer architectures, further improvements to our GPU implementation such as multi-GPU and multi-node scaling are possible. Additionally, we expect many latency-bound GPU applications can benefit from algorithm-specific modifications similar to our register and shared memory utilization that pivot away from common matrix-operation style computation towards vector-operation computation that the hardware may more efficiently

process.

Our work highlights the fundamental importance of maintaining locality between operations, which has widely been embraced by the machine learning community via various frameworks with advanced kernel fusion techniques. Other scientific applications that primarily rely on BLAS libraries continue to suffer from a lack of kernel fusion techniques, presenting opportunities for compilers and other frameworks in scientific computing to improve GPU locality between such operations.

Finally, we believe additional improvements to our implementation of Word2Vec remain feasible, given interest in the subject. Other research has already indicated a number of relaxations similar to those used in our work, such as shared negative sampling and our fixed context window width. We believe that properly determining the algorithmic extent of permissible reuse will reveal greater opportunities to improve the GPU-side control flow, permitting GPU performance to scale closer to the roofline potential of the architecture. Improved understandings of the algorithm's convergence properties may also pave the way for additional parallel scaling, especially in multi-node and multi-GPU implementations.

Efficient and Transferable Multi-Scale Performance Autotuning We believe that there is ample reason to look forward to other generative or non-regressive transfer learning approaches for performance autotuning to further explore the regime of data-constrained few-shot transfer learning. Non-regressive autotuning techniques are highly suited for the data constraints imposed by many applications of interest. Other probability models may be better suited to modeling complex behaviors such as those we observed while weak-scaling heFFTe and can provide even more precision when predicting outcomes and greater explainability of generated results.

We also note the obvious failure of transfer-learning techniques on the weakly scaled FFT problem as a clear indicator of an under-served challenge within the autotuning space. Multi-scale tuning appears to present complex interdependencies between variables that frustrates current modeling and representative techniques. Improving the mathematical understanding of these behaviors to suggest the necessary amount of information for proper modeling and better ways to measure and predict the most-informative configurations that contribute to that information will be invaluable to the continued growth and practice of autotuning within HPC.

There are also possible improvements to the Gaussian Copula in the form of supporting model paradigms, where the Gaussian Copula's low-cost generative technique can be supported

by a less capable regressive model that can sort and refine the set of generated candidates prior to evaluation. This permits the supporting model an opportunity to simplify its approach to only discriminate within a higher-quality region of the search space. It also provides the Gaussian Copula means to continue generating for longer-term searches without degrading towards the expectation of pure random sampling.

Bibliography

[1] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, Jan 2019.

[2] Venkat Natarajan, Anand Deshpande, Sudarshan Solanki, and Arun Chandrasekhar. Thermal and power challenges in high performance computing systems. *Japanese Journal of Applied Physics*, 48(5S2):05EA01, May 2009.

[3] Electronic and Photonic Packaging Division. *Computational Analysis for Thermal Optimization of Server for Single Phase Immersion Cooling*, volume ASME 2019 International Technical Conference and Exhibition on Packaging and Integration of Electronic and Photonic Microsystems of *International Electronic Packaging Technical Conference and Exhibition*, 10 2019.

[4] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[5] Richard Brown, Alliance to Save Energy, ICF Incorporated, ERG Incorporated, and U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency: Public law 109-431. Technical report, Lawrence Berkeley National Laboratory, 2008. Retrieved from `https://escholarship.org/uc/item/74g2r0vg`.

[6] Bharath Ramakrishnan, Yaser Hadad, Sami Alkharabsheh, Paul R. Chiarot, and Bahgat Sammakia. Thermal Analysis of Cold Plate for Direct Liquid Cooling of High Performance Servers. *Journal of Electronic Packaging*, 141(4):041005, 07 2019.

[7] A. Bar-Cohen, M. Arik, and M. Ohadi. Direct liquid cooling of high flux micro and nano electronic components. *Proceedings of the IEEE*, 94(8):1549–1570, 2006.

[8] Anna Rogers, Shashwath Hosur Ananthakrishna, and Anna Rumshisky. What's in your embedding, and how it predicts task performance. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2690–2703. Association for Computational Linguistics, 2018.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

[11] Balaprakash, P. et al. Autotuning in high-performance computing applications. *Proc. of the IEEE*, 106(11):2068–2083, 2018.

[12] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall. Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurrency and Computation: Practice and Experience*, Volume 34, Issue 20, e6683, https://doi.org/10.1002/cpe.6683, 2021.

[13] X. Wu, P. Balaprakash, M. Kruse, J. Koo, B. Videau, P. Hovland, V. Taylor, B. Gelts, S. Jana, and M. Hall. ytopt: Autotuning scientific applications for energy efficiency at large scales. In *Proceedings of Cray User Group Conference 2023*, CUG'23, Helsinki, Finland, May 7-11, 2023, 2023.

[14] Tomofumi Yuki and Louis-Noel Pouchet. PolyBench 4.2, 2016.

[15] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. GPTune: multitask learning for autotuning exascale applications. In *Proceedings of PPoPP '21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'21, pages 234–246, New York, NY, USA, February 2021. Association for Computing Machinery.

[16] ytopt: a machine learning-based autotuning software package. Argonne National Laboratory. https://github.com/ytopt-team/ytopt, Last accessed, March 11, 2021.

[17] Shengchun Liu, Zhiming Xu, Zhiming Wang, Xueqiang Li, Haiwang Sun, Xinyu Zhang, and Haoran Zhang. Optimization and comprehensive evaluation of liquid cooling tank for single-phase immersion cooling data center. *Applied Thermal Engineering*, 245:122864, 2024.

[18] Harshad Shrigondekar, Yueh-Cheng Lin, and Chi-Chuan Wang. Investigations on performance of single-phase immersion cooling system. *International Journal of Heat and Mass Transfer*, 206:123961, 2023.

[19] Jimil M. Shah, Keerthivasan Padmanaban, Hrishabh Singh, Surya Duraisamy Asokan, Satyam Saini, and Dereje Agonafer. Evaluating the Reliability of Passive Server Components for Single-Phase Immersion Cooling. *Journal of Electronic Packaging*, 144(2):021109, 10 2021.

[20] Nugroho Agung Pambudi, Alfan Sarifudin, Ridho Alfan Firdaus, Desita Kamila Ulfa, Indra Mamad Gandidi, and Rahmat Romadhon. The immersion cooling technology: Current and future development in energy saving. *Alexandria Engineering Journal*, 61(12):9509–9527, 2022.

[21] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei W. Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. *CoRR*, abs/2006.06890, 2020.

[22] David H. Bailey. *NAS Parallel Benchmarks*, pages 1254–1259. Springer US, Boston, MA, 2011.

[23] Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Cleary, and Piotr Luszczek. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. `https://www.netlib.org/benchmark/hpl/index.html`, 2018. Accessed: 2023-10-01.

[24] Dell. Command List of OpenManage Server Administrator (OMSA), accessed 2025.

[25] J. R. Firth. *A Synopsis of Linguistic Theory, 1930-1955.* n.p., 1957.

[26] Saurabh Gupta and Vineet Khare. Blazingtext: Scaling and accelerating word2vec using multiple gpus. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, New York, NY, USA, 2017. Association for Computing Machinery.

[27] G. E. Moon, D. Newman-Griffis, J. Kim, A. Sukumaran-Rajam, E. Fosler-Lussier, and P. Sadayappan. Parallel data-local training for optimizing word2vec embeddings for word and graph embeddings. In *2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 44–55, 2019.

[28] T. M. Simonton and G. Alaghband. Efficient and accurate word2vec implementations in gpu and shared-memory multicore architectures. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2017.

[29] Seulki Bae and Youngmin Yi. Acceleration of word2vec using gpus. In *Proceedings of the 23rd International Conference on Neural Information Processing*, volume 9948, pages 269–279, 10 2016.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. Attention is all you need. *ArXiv*, abs/1706.03762, 2017.

[31] Denis R, Peter Jose P, and Sushma Margaret A. Performance analysis of machine learning - semantic relational approach based job recommendation system. In *2023 10th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1478–1486, 2023.

[32] Akhilesh Kumar Srivastava, Ishanki Verma, and Puneet Garg. Improvements in recommendation systems using graph neural networks. In *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*, pages 668–672, 2024.

[33] Aditya Grover and J. Leskovec. node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[34] Bryan Perozzi, Rami Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *KDD '14*, 2014.

[35] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited, 2001.

[36] Felix Hill, Roi Reichart, and Anna Korhonen. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, 2015.

[37] Vasudevan Rengasamy, Tao-Yang Fu, Wang-Chien Lee, and Kamesh Madduri. Optimizing word2vec performance on multicore systems. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, New York, NY, USA, 2017. Association for Computing Machinery.

[38] S. Ji, N. Satish, S. Li, and P. K. Dubey. Parallelizing word2vec in shared and distributed memory. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2090–2100, Sep. 2019.

[39] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.

[40] Tensorflow. Tensorflow. `https://www.tensorflow.org/tutorials/text/word2vec`.

[41] Radim Řehůřek. Gensim. `https://radimrehurek.com/gensim/models/word2vec.html`.

[42] J. Canny, H. Zhao, B. Jaros, Y. Chen, and J. Mao. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 233–242, 2015.

[43] Taisuke Ono, Tomoki Shoji, H. M. Waidyasooriya, M. Hariyama, Yuichiro Aoki, Yuki Kondoh, and Yaoko Nakagawa. Fpga-based acceleration of word2vec using opencl. *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2019.

[44] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[45] Matt Mahoney. Large text compression benchmark. Unpublished paper, 2011.

[46] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013.

[47] David Mimno and Laure Thompson. The strange geometry of skip-gram with negative sampling. In *EMNLP*, 2017.

[48] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: Online autotuning through local competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 91—100, New York, NY, USA, 2012. Association for Computing Machinery.

[49] David Buchaca Prats, Felipe Albuquerque Portella, Carlos H. A. Costa, and Josep Lluis Berral. You only run once: Spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management*, 17(4):2039–2051, 2020.

[50] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. *Bliss: Auto-Tuning Complex Applications Using a Pool of Diverse Lightweight Learning Models*, pages 1280—1295. Association for Computing Machinery, New York, NY, USA, 2021.

[51] Guido Masarotto and Cristiano Varin. Gaussian copula marginal regression. *Electronic Journal of Statistics*, 6:1517–1549, 2012.

[52] Tertsegha J. Anande, Sami Al-Saadi, and Mark S. Leeson. Generative adversarial networks for network traffic feature generation. *International Journal of Computers and Applications*, 0(0):1–9, 2023.

[53] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional GAN. *CoRR*, abs/1907.00503, 2019.

[54] Ananta Tiwari and Jeffrey K. Hollingsworth. Online adaptive code generation and tuning. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 879–892, 2011.

[55] Ray S. Chen and Jeffrey K. Hollingsworth. ANGEL: A hierarchical approach to multi-objective online auto-tuning. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, New York, NY, USA, 2015. Association for Computing Machinery.

[56] J. Knowles. ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, 2006.

[57] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. Tuneful: An online significance-aware configuration tuner for big data analytics. *CoRR*, abs/2001.08002, 2020.

[58] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via Bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209, 2021.

[59] N. Patki, R. Wedge, and K. Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, Oct 2016.

[60] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[61] Tobias Grosser, Armin Grösslinger, and Christian Lengauer. Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012.

[62] Stanimire Tomov, Azzam Haidar, Alan Ayala, Daniel Schultz, and Jack Dongarra. Fft-ecp fast fourier transform. *n.d.*, 2019-01 2019.

[63] Stephen Hudson, Jeffrey Larson, John-Luke Navarro, and Stefan M. Wild. libEnsemble: A library to coordinate the concurrent evaluation of dynamic ensembles of calculations. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):977–988, 2022.

[64] CUFFT library user's guide. NVIDIA Corporation, 2022.

[65] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. Transfer learning for improving model predictions in highly configurable software. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 31–41. IEEE, 2017.

[66] Wissam M Sid-Lakhdar, Mohsen Mahmoudi Aznaveh, Xiaoye S Li, and James W Demmel. Multitask and transfer learning for autotuning exascale applications. *arXiv preprint arXiv:1908.05792*, 2019.

[67] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. Performance modeling under resource constraints using deep transfer learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[68] Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, Alfredo Gimenez, Rahul Sridhar, Aniruddha Marathe, Tao Wang, Murali Emani, Abhinav Bhatele, and Todd Gamblin. Bootstrapping parameter space exploration for fast tuning. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 385—-395, New York, NY, USA, 2018. Association for Computing Machinery.

[69] David Salinas, Huibin Shen, and Valerio Perrone. A quantile-based approach for hyperparameter transfer learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8438–8448. PMLR, 13–18 Jul 2020.

[70] Zheng Zhang, Tinghuan Chen, Jiaxin Huang, and Meng Zhang. A fast parameter tuning framework via transfer learning and multi-objective bayesian optimization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 133—-138, New York, NY, USA, 2022. Association for Computing Machinery.

[71] Ralph Silva and Hedibert Lopes. Copula, marginal distributions and model selection: A Bayesian note. *Statistics and Computing*, 18:313–320, 09 2008.