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

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 has 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.

In this work, we look at three distinct points that holistically address continued scalable system performance. We will analyze the impacts of new liquid cooling technologies on application performance, providing a first look at the opportunities beyond energy efficiency that are enabled by this technology. We also analyze better means for software to utilize hardware through a case study of the GPU implementation of Word2Vec. 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 frameworks 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. Collectively these insights demonstrate the necessity of flexible codesign for generalized performance across specialized hardware and systemic means to optimize application performance.

Acknowledgments

Portions of this material are based on work supported by the National Science Foundation under Grant Nos. MRI# 2024205, MRI# 1725573, and CRI# 2010270. Clemson University is acknowledged for their generous allotment of compute time on the Palmetto Cluster.

Portions of this material are based on work supported by the National Science Foundation under Grants CNS-CCF-1942182: CCF: EAGER: DeepGreen: Modeling and Boosting Accelerated Computing on Liquid Immersion Cooled HPC Systems 10/1/2019 - 9/30/2024, CNS-1551262 Marcher - A Heterogeneous High Performance Computing Infrastructure for Research and Education in Green Computing 5/18/2015 - 8/31/2018, and CCF-1551511: CAREER: Cross-Layer Power-Bounded High Performance Computing on Emerging Heterogeneous Systems 2/1/2015 - 1/31/2022.

Portions of this material are based on work supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

2017 Turing Award winners John Hennessy and David Patterson declared the recent explosion of domain-specific hardware to be a core component of a so-called “golden age of computer architectures” [1]. In the years before and since, continual large-scale government and industry investments in specialized hardware for artificial intelligence, machine learning, cloud infrastructure and datacenter technologies have driven scientific knowledge to ever-greater heights. Machine learning has embraced the development of large language models, fueling GPU accelerator development as well as an entire ecosystem of hardware for massive-scale training and inference for machine learning models. Several world governments are actively building exascale supercomputers, with a few already fully in production, pushing the limits of natural sciences. The Covid-19 pandemic saw massive cooperative efforts between various cloud providers to deliver massive computing power to scientists and researchers, playing a key role in the accelerated vaccine development and response to such a highly infectious disease.

The golden age has not come without its own problems and blind spots, however. 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 large-scale computing. Furthermore, not all domains and applications have experienced equal benefits from specialized hardware advancements, even when novel hardware designs have great opportunities to cater to broader computing needs. Even when efforts are made to facilitate the usage of new hardware with existing software, performance improvements may be lackluster.

There is no singular solution to these problems, however they all indicate a greater need for joint consideration between the opportunities provided by hardware and the myriad needs of software applications. The tighter coupling between hardware and software requires greater cohesion and development on either side must hold greater respect for the goals and limitations of the other. To truly maximize collaborative performance, flexible tools that quickly provide insight and access to the best manner to utilize available components are needed.

In this work, we consider the impacts of hardware upon software and vice-versa from three distinct perspectives.

First, we consider the impacts of novel Single-Phase Liquid Immersion Cooling (SPLIC) technologies as a hardware approach to improve the sustained performance of software applications across many domains. Many studies have explored the properties of this technology and simulated many possible hardware configurations to improve system performance, however existing works do not analyze actual impacts experienced by software. Our experiments will produce novel contributions to science by analyzing the affects of this new cooling technology on different software workloads commonly executed in HPC environments. We will also attempt to identify if these differences are perceptible by software applications that do not explicitly monitor hardware thermal readings to indicate if special care must be exercised by software to permit greater cooperation between the tolerances of SPLIC responsiveness and software throughput.

Second, we consider the complexity of designing software to properly exploit available performance across many generations of accelerator technology through the lens of GPU implementations of Word2Vec. We optimize Word2Vec performance on GPUs to permit automatic performance improvements from successive hardware generations that have been neglected in prior work, resulting in poor GPU performance scaling that has historically been superceded by CPUs.

Third, we unite these perspectives of hardware- and software- based optimization through automatic empirical performance tuning, also known as performance autotuning, across both hardware and software interfaces. Existing performance autotuning has limited capability to exploit existing knowledge and is forced to waste some empirical samples when tuning related problems. We identify means to immediately leverage this knowledge to access near-optimal areas of tuning search spaces with a novel transfer tuning technique.

Collectively, these advancements represent the immense opportunities for performance improvement that are yielded from deeper systemic understanding of hardware and application performance. By properly combining these perspectives in a systemic manner, heterogeneous system performance can be greatly improved across many critical applications that form a foundation for the future of science and computing.

Chapter 2

Background

In this chapter, we offer an overview of pre-existing technologies related to our proposed work.

2.1 Thermal Management in Computing

Different computer architectures and accelerators cater to different computing tasks, but one of the major ways to iteratively improve performance via hardware is to increase the compute density. Denser circuitry decreases operational latency but also requires increased power delivery to power the components and reduced coolant access to components deeper within 3D dies, resulting in greater thermal accumulation within the hardware [2]. At the extreme scale, this heat can damage computing hardware and render it inoperable [3], so thermal sensors in critical hotspots automatically reduce the clock rate of devices to limit additional energy before the device becomes temporarily or permanently inoperable. Lower clock rates and device failure are obviously detrimental for performance, so sophisticated HPC systems that push thermal limits require equally sophisticated cooling to permit maximal usage for the longest possible period.

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

2.1.1 Liquid Cooling Technologies

We briefly introduce several cooling technologies that utilize liquids for cooling computing systems.

Direct Liquid Cooling DLC, sometimes also referred to as cold plates, are cooling systems where metal plates (typically copper) are installed directly on top of computer components. The cold plates include microchannels for a liquid to flow through, conducting heat away from a computing component to a centralized heat exchange with air or facility water [4]. DLC is a mature technology with serious attention in research and industry dating back to the 1980s [5]. Plate microchannels in nearly direct contact with the computing components allows for heat exchange to be much more efficient relative to air-cooling fans and often requires less physical space within the rack, permitting denser rack designs without overwhelming the cooling system. HPC systems typically recirculate liquid with a chiller that aggregates heated liquid from many components, but some technologies rely on less efficient fan-cooled radiators for heat exhaust [6]. Hotter coolant requires less pumping power to circulate due to pressure decrease in the fluid as it accumulates heat, leading to more energy-efficient circulation as more energy is required to remove heat from the system [4].

Single Phase Liquid Immersion Cooling SPLIC systems include container with pumps for forced circulation and a coolant heat exchange. The physical properties of the coolant diffuse heat throughout the entire fluid volume, balancing heat throughout the entire system somewhat passively, though circulation is induced to reach heat equilibrium at a faster rate and facilitate the heat exchange. In comparison to DLC, immersion cooling affects all submergd components rather than just areas covered by cold plates, however the greater volume of liquid is not managed at the same granularity. This permits SPLIC systems to have extreme energy efficiency as the system operates passively while thermal levels lie below the set point, only running periodically to maintain fluid circulation.

There are many design decisions that impact the performance of SPLIC, primarily the number of pumps, the pump arrangements and settings within the tank, and the choice of dielectric fluid, which have all been rigorously studied and simulated by many works [7, 8, 3]. Furthermore, research has already dispelled several doubts about the technology, including the extent of dangers to hardware components from absorbing dielectric fluids [9] and the capital burden of procuring, installing and maintaining SPLIC rather than other more traditional cooling systems [10].

Dual Phase Liquid Immersion Cooling The extreme energy efficiency of SPLIC is pushed to greater heights by DPLIC. The dielectric fluid in DPLIC systems is designed to gradually evaporate at high operating temperatures. The physics of density and thermodynamics move captured heat in the form of gasses through cooler liquids to the top of the tank where a condenser can remove heat and precipitate fluid back into the tank. The advantage of DPLIC over SPLIC is that there are no moving parts and no energy spent circulating fluid, however the phase change necessitates a completely sealed enclosure that inconveniences direct hardware maintenance.

2.1.2 Proposed Contribution

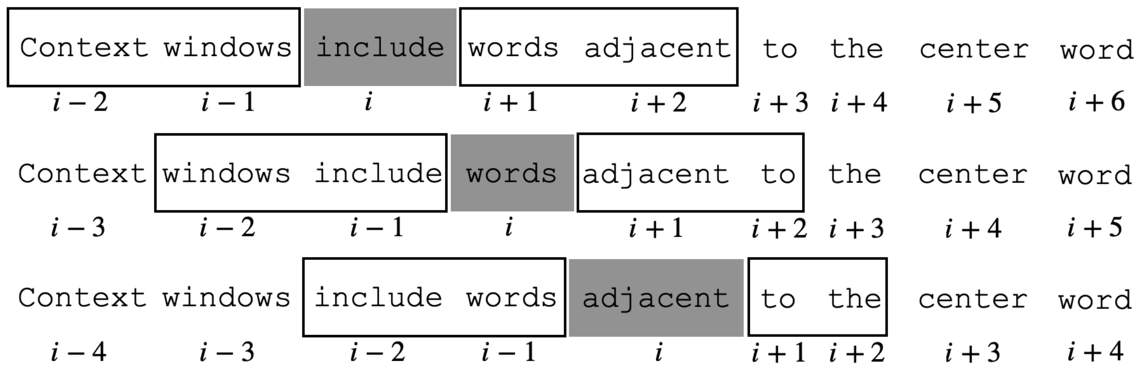
Existing research primarily utilizes simulation with specific empirical validation to model performance under different hardware configurations, such as the thermal properties of the system when utilizing specific coolants. While important, these analyses fail to demonstrate opportunities for applications to push performance boundaries beyond what is possible in air-cooled systems, if any exist. We intend to analyze the actual impact of SPLIC technology on energy efficiency and application performance to better understand the opportunities and limitaitons of current implementations.

2.2 Word2Vec

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*. 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.

There are a few model architectures that determine how words are related to one another during training. The Word2Vec introduced by Mikolov et al. [11] provides two model architectures: Continuous Bag-of-Words (CBOW) and Continuous Skip-Gram with Negative Sampling (SGNS).

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.


Context windows include words adjacent to the center word
$i-2$ $i-1$ i $i+1$ $i+2$ $i+3$ $i+4$ $i+5$ $i+6$
Context windows include words adjacent to the center word
$i-3$ $i-2$ $i-1$ i $i+1$ $i+2$ $i+3$ $i+4$ $i+5$
Context windows include words adjacent to the center word
$i-4$ $i-3$ $i-2$ $i-1$ i $i+1$ $i+2$ $i+3$ $i+4$

Rogers et al. [12] found that the SGNS model architecture generally produces higher quality embeddings for downstream applications of interest, so we focus our attention on the Continuous SGNS model architecture.

The abstract overview of the Word2Vec algorithm is as follows: the input to Word2Vec is typically a corpus of words that are organized into "sentences," and in each sentence meaningful relationships exist between nearby words; the vocabulary V is formed from all words in the corpus. In training, the contents of individual sentences are consumed from beginning to end using a sliding *context window* of $2W$ 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 training algorithm assumes that each word in the context window has increased similarity to the target word as it is positively related. Such similarity is reflected in the vector representations by increasing their vector similarity. The algorithm also assumes that words not present in the context window are *not* related, and their similarity to the target word decreases. 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 [13, 11] 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.

2.2.1 Prior Word2Vec Implementations

All Word2Vec implementations historically stem from foundational work by Mikolov et al [13, 11], which expresses high level data parallelism between sentences of the corpus for improved performance. According to Hogwild! SGD [14], as long as large models are trained with batches with sufficiently varying 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 [15] and Gensim [16] machine learning frameworks. The algorithm has been ported to many architectures, including the cloud-based BlazingText [17], cluster implementation BIDMach [18], and FPGA architectures [19]. 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.

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

pWord2Vec Ji et al [20] 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 [21] utilize advanced batching techniques to combine multiple context windows into larger matrix batches. The technique allows CPU architectures to achieve much greater throughput, but computation still takes place entirely on the CPU architecture and is otherwise equivalent in performance to pWord2Vec.

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

accSGNS Bae and Yi [22] 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 work's vector parallelism 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 [23] 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 [24] 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. We were unable to replicate the paper's reported results on our own systems, so we do not compare against it.

2.2.2 Proposed Contribution

The Word2Vec algorithm includes both fine-grained and coarse-grained parallelism which can both be exploited on GPU architectures. Despite this, GPU implementations have mostly failed to outperform CPUs in existing work, primarily due to a memory-latency bottleneck that exaggerates performance issues inherent to GPU parallelism. We intend to demonstrate a simple approach that relaxes key bottlenecks and permits much greater performance scaling across several generations of GPU architectures.

2.3 Performance Autotuning

Performance autotuning [25, 26, 27], also referred to as “autotuning,” 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 [28] “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. Due to the overhead of compiling configured source code, even the smallest input matrix scales requires at least 25 seconds to empirically evaluate. Even at such small scales, this represents over one hundred days of compute demand to brute-force evaluate all possible configurations for a single matrix scale–in other words it is prohibitively costly to empirically evaluate entire search spaces. Autotuning uses more intelligent approaches to identify the configurations that achieve optimal performance from selective samples.

To ensure high-fidelity results across many input scales, Autotuning must separately tune different inputs scales due to the input scale’s impact upon performance characteristics. As shown in Table 2.1, small-scale inputs require the packed-array technique for matrices A and E in order to achieve optimal performance, but medium-scale inputs do not. The degree of improvement can also

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×

vary between input scales, where small-scale 3mm inputs can gain 1.13× speedup from autotuning but medium-scale 3mm inputs gain 14.94× speedup over their respective baselines.

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 optimization that employs Gaussian process regression and random forest and their variants. Model-free methods optimize the objective function without the use of surrogate 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 [29, 30, 27, 26].

2.4 Transfer Learning in Autotuning

Transfer Learning (TL) in autotuning is an emerging approach that leverages data from one autotuning task in related autotuning tasks to significantly improve sample efficiency. Related autotuning tasks are common in HPC applications, which include tuning different input scales 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.

2.4.1 Existing Transfer Learning Autotuning Techniques

Prior TL autotuning has enabled data reuse on related tasks for increased sampling efficiency and reduced modeling overhead. BLISS [31] 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 [32] 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 [33, 29] 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 [34, 35, 36] focus on multiobjective efficiency by refining a surrogate Pareto frontier. These algorithms provide stronger long-term convergence guarantees rather than few-shot performance. Our proposed 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. [37] 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 [38] 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 [39] utilizes incremental sensitivity analysis in BO and explicitly utilizes importance to identify performance trends. Chen et al. [40] 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. [41] 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. [42] 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. [41] 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 [43] 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 [44]. Users can select between variations better suited for tail distributions for which Pearson correlation is insufficient to describe covariant behaviors jointly.

2.4.2 Proposed Contribution

The long-term perfectionism of many autotuning techniques often come at the cost underwhelming initial performance, limiting adoption due to the great cost required to properly tune applications. These costs can often be indeterminate as the optimum is only guaranteed at convergence, which may require massive resource commitments in large search spaces. The nuanced nature of performance relationships often fails to hold across all application and system scales, making the cost of tuning even higher for true generalized performance. We seek to greatly reduce the cost of transferring performance results to new settings by immediately leveraging existing knowledge within a fixed number of evaluations, providing known cost bounds to the tuning effort and maximizing the short-term effectiveness of performance tuning.

Chapter 3

Impacts of Immersion Cooling on Application Performance

In efforts to improve energy efficiency and reduce hardware replacement costs, many large scale computer systems are transitioning from air-cooling solutions to liquid-cooling solutions. Single Phase Liquid Immersion Cooling (SPLIC) is one such promising technology, wherein computing hardware is completely submerged within a container filled by a dielectric fluid with thermal properties similar to water. The denser medium has higher heat capacitance than air, can more effectively diffuse heat away from hotspots, and can efficiently transfer heat to chilled water systems or even be employed as input for heat-recycling. The environmental benefits of SPLIC are well-understood and continually improved, but the affects on actual software workloads within these containers have largely gone without study.

While it's true that 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. Unlike traditional air-cooling, which while ineffective, is always applied as directly to hardware hotspots as possible, SPLIC represents a more passive cooling solution, and typical configurations are only capable of increasing fluid circulation as an active cooling measure. In this sense, these systems do not directly monitor critical junctions for heat buildup and cannot directly address heat accumulation in hardware, so understanding the actual impact of this cooling technology on different software is important for evaluating its utility and design implications beyond hardware configuration.

In the following sections, we outline our experimental plans to answer some of these questions using a Submer SmartPod v3 and a small cluster of heterogeneous compute nodes.

3.1 Research Questions and Experiment Design

Hardware Preparation for SPLIC The Submer SmartPod v3 utilizes a proprietary synthetic fluid. The fluid includes plasticizer, so typical thermal paste between hardware components and heat sinks are replaced with indium foil. The plasticizer also leaches material from connection cord casings, such as ethernet cables and power supplies, but we do not completely replace these components. As time progresses, the leaching makes the cables rigid and somewhat brittle, however they remain operable and carry negligible risk of breaking while undisturbed. Due to the density change between synthetic fluid and air, hardware fans on power supplies, GPUs and CPUs are removed and disabled to prevent alarms calibrated for air-cooled environments from disabling components. This carries some risks to the hardware which will be studied as one of the research questions of this work: can SPLIC effectively protect hotspots in dense computing components?

Does Improved Heat Dissipation Permit Sustained Performance As the synthetic fluid is designed for thermal dissipation and is far denser than air, we expect both passive and active heat dissipation of the SPLIC environment to far outperform traditional air cooling. To investigate this hypothesis, we prepare a hardware duplicate of the immersed server without modifications specifically made for the SPLIC environment and benchmark a variety of applications, monitoring the system power, thermal sensors and application performance over extended durations. Due to the under-provisioning of compute hardware relative to the calculated potential heat dissipation of the Submer Pod, we temporarily disconnect the water for heat exchange to build heat in the system, then reconnect to observe the effects of passive and active heat dissipation within the system. We expect SPLIC to permit hardware to run for longer durations without reaching critical thermal levels that require throttling, leading to greater sustained application throughput. Then, we will replicate these experiments using similar hardware configurations in an air-cooled environment for comparison, paying special attention to hotspot management. We hypothesize that the air-cooled environment will require more energy to less effectively remove heat from the system, however as a more directed, active cooling system we expect it will return the entire system to its idle condition much faster than the SPLIC system.

Specific Impacts on Different Hardware Components Our experimental setup monitors memory, CPU and GPU components as well as the general environment. We expect that greater thermal insulation provided by air will allow under-utilized components to remain cool, while the liquid coolant will freely share heat between hardware components. It is possible that an abundance of dissipated heat could induce new thermal bottlenecks in other components.

3.2 Expected Outcomes

We anticipate that our experimental results will demonstrate the actual differences experienced by various applications in environments cooled by traditional air systems and SPLIC, especially in terms of long-horizon heat management and responsiveness to thermal changes in critical hotspots. While studied to some degree in prior work, we will also corroborate data regarding the effectiveness and required efforts for transitioning our air-cooled server to be operable in an SPLIC environment and the affects of long-term exposure to the SPLIC environment on our hardware.

3.3 Proposal Status

Our experimental environment for SPLIC is fully set up and ongoing experiments are covering a variety of software applications to demonstrate different workloads on the server. We have categorized the completed experiments as CPU- or GPU- intensive depending on the primary hardware utilized by the program, and Memory- or Compute- intensive depending on the primary performance bottleneck experienced by the software on our compute architecture. The applications and their status are represented in Table 3.1.

Application Name	Experiment Performed?	Primary Hardware	Compute / Memory Intensive
Baseline (do nothing)	Yes	–	–
CUDA Stream	Yes	GPU	Memory
EMOGI	Yes	GPU	Memory
CUDA DGEMM	Yes	GPU	Compute
CUDA MD5 Cracker	Yes	GPU	Compute
CUDA MD5 Bruteforcer	Yes	GPU	Compute
CUDA SHA256	No	GPU	Compute
HPLinpack	No	CPU	Compute
NPB.DT	No	CPU	Compute
MLPerf	No	CPU	Memory

Table 3.1: Proposed Experiments in SPLIC environment. Each experiment will be replicated in an air-cooled environment at a later date.

Chapter 4

Properly Leveraging GPU Acceleration: A Case Study in Word2Vec Optimization

Word2Vec remains one of the highly impactful innovations in the field of Natural Language Processing (NLP) that represents latent grammatical and syntactical information in human text with dense vectors in a low dimension. Word2Vec has high computational cost due to the algorithm's inherent sequentiality, intensive memory accesses, and the large vocabularies that it represents. While prior studies have investigated technologies to explore parallelism and improve memory system performance, they struggle to effectively gain throughput on powerful GPUs and have mostly failed to exceed CPU performance despite massive parallelism present in the algorithm.

Existing GPU implementations of Word2Vec experience subpar performance due to bottlenecks in the architecture's memory latency, which prevents these implementations from exceeding CPU performance despite multiple levels of parallelism in the algorithm. Without redesigning the Word2Vec implementation to specifically address GPU memory latency, the algorithm's performance will not reach adequate levels even in the presence of more sophisticated hardware technologies. To address this, we propose a that novel Word2Vec implementation is needed to better match GPU architectures over existing and future technology generations.

In the following sections, we outline our plan to implement a novel GPU implementation of

Word2Vec for this portion of the proposal.

4.1 Bottlenecks in GPU Performance

GPU implementations of the Word2Vec algorithm typically utilize parallel series of tiled matrix multiplications. Each parallel series operates on data-parallel sentences from the corpus without synchronization due to Hogwild! [14] stochastic gradient descent guarantees, while the series of context windows within an individual sentence can be serially computed using tiled matrix multiplication to benefit from shared memory and scale out computation along GPU threads. This form of performance scaling for the GPU is logical, but ultimately misses out on several key constraints that ultimately cause GPUs to exhibit subpar performance in practice.

The decision to train sentences in parallel on GPUs is similar to the parallelism exploited in CPU implementations. This parallel scaling adheres to Word2Vec's algorithmic requirements for convergence and semantic utility, but also induces large memory bandwidth demand on the hardware architecture. While the relatively large and sophisticated cache memory hierarchies on CPUs can often service the thread-parallel workloads with low contention due to the lower number of CPU threads compared to GPUs, greater threadblock parallelism on GPUs and more limited cache hierarchy result in much greater contention that cause performance degradation in GPU memory.

To counteract this, GPUs have very flexible thread scheduling and attempt to "hide" memory access latency by scheduling other parallel work while memory accesses are fulfilled. However, the individual context windows represent minimal arithmetic demand. In order for vector embeddings to be useful, their size must be much less than the size of the learned vocabulary. In practice, most vector embeddings consist of 10 to 100 elements and sizes greater than 1000 elements are rare. For the Word2Vec algorithm's learned semantic similarity to be useful, context windows often group ten or fewer words for a single tiled matrix multiply, severely limiting the amount of computation that can be performed before more memory demand is needed. The simplicity of the Word2Vec algorithm interjects only a handful of additional operations between matrix multiplies. This leads to low arithmetic intensity that fails to hide memory latency on GPUs and therefore poor performance scaling between hardware generations that have largely focused on improving compute throughput and scheduling parallelism.

4.2 Proposed Implementation

While GPUs have been utilized to facilitate matrix multiplications, we believe there is much greater opportunity for Word2Vec performance on GPU hardware when expressing the algorithm at the vector level. The limited vector sizes and number of vectors permit a vector-based implementation greater fine-grained parallel scaling than tiled matrix multiplication. At the same time, this limited data volume permits improved usage of GPU memory to precisely manage caches to coordinate with the well-known reuse patterns in the Word2Vec algorithm. Combining these techniques will reduce global memory demand to reduce memory bandwidth requirements while providing more opportunities for computation to hide memory latency and fill out idle cycles with useful work, increasing parallel throughput to better meet hardware capabilities even with generational improvements in the architecture.

4.2.1 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 a context window in a given sentence, each context word is paired against each negative sample 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. By leveraging the property of negative sample independence, we can flexibly manage 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, we can store the vector representation of the current negative sample in per-thread registers for the entire duration of its usage. 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 cases, as values in shared memory must be loaded to and stored from registers during computation, just at lower cost than higher levels of the memory hierarchy. 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 better-suited optimizations.

Temporal Locality and Reuse Locality is preserved for each negative sample by storing it in per-thread registers and allowing all the required embedding updates to accumulate in registers before writing the cumulative update back to memory once. Per the Word2Vec algorithm, each negative sample is reused by $2W$ times spanning a single context window, yielding $2W$ guaranteed reuses in register per load and store with minimal cache pressure.

In this way, we ensure our negative reuse has minimal impact on the quality of the resultant embeddings. While prior works [20, 24, 21, 18] indicate that the particular negative samples do not need to be independent across context windows, Moon et al [24] show that excessive reuse for negative samples has harmful impacts on the final embedding quality. Nevertheless, the limitations are not well understood by established literature. The reuse in a single window has notable improvement for minimal embedding quality cost [20], and greatly improves the access and storage patterns of negatives for GPU architectures.

One complexity of progressing to the next context window is incremental model updates. As the context window slides, context words are reused several times and therefore the corresponding model parameters have data dependencies on prior updates, requiring *strict sequential context* *window ordering*. In order to adhere to *strict context window ordering* but take advantage of *negative sample independence*, we must use separate thread blocks to process separate sentences, 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 or risking over-reusing data.

4.2.2 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. Almost every context word in a given window is also a context word in the subsequent window. Since successive context windows always shift the boundary and target word over by one word, every word in the sentence will be a target word once and can appear in up to $2W$ sequential windows as a context word. In other words, a context word's lifetime can be up to $2W + 1$ times of reuses. Despite this, existing GPU algorithms fail to realize this degree of reuse or adversely use excessive cache resource by relying on implicit hardware management. We intend to exploit this data reuse opportunity for the first time by explicitly caching and reusing context words at runtime using GPU shared memory.

To reduce expensive and high-latency global memory accesses, we utilize GPU shared memory to cache context words for their lifetime. A naive approach to reusing across multiple windows is to match the size of all words in a context window and allocate space for multiple windows. This approach would require a prohibitive amount of shared memory, so a more sophisticated and scalable solution is required. We allocate a fixed buffer of shared memory to store all words in the current context window, including the target word. When the context window shifts, we overwrite the ejected word with the new word, and merely use a few registers to track this process in a manner similar to a circular buffer. Using this explicit memory management, we avoid repeated global memory accesses for context words through the maximal period guaranteed by the Word2Vec algorithm and we also reduce contention amongst thread blocks over these accesses. The shared memory buffer size is limited by hardware constraints, but even older generations of GPU hardware have sufficient shared memory per Streaming Multiprocessor to cache all vectors fetched by common values for context widths.

While these innovations are useful, typical Word2Vec implementations randomly permute the context width of each window to select between 1 and W words to either side of the target word [45], which would reduce the actual effectiveness of our shared memory utilization. Complying with this treatment of context width would require at most $2W \times d$ space for a context window and rely on the GPU to properly permute the random width or for these random variations to be generated on CPU and transferred alongside the kernel call as additional overhead. To simplify implementation, we instead use 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 provides the same quality of result while reducing (1) per-context width metadata, (2) the shared memory allocation requirement by half, and (3) the overall implementation complexity of the shared memory buffer.

With this implementation, we 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. Furthermore, the shared memory cache is uniformly effective at providing reuse opportunities rather than fluctuating the degree of reuse, while the long-term average reuse is maintained exactly the same as other Word2Vec implementations. 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 $W_f = 5$, both of which are commonplace values for W. In terms of the GPU architecture, this reduces the overall latency and therefore requirement for latency hiding, significantly improving upon a key hardware bottleneck.

4.3 Proposal Status

The proposed ideas have been implemented in a prototype called “FULL-W2V,” which was published in the Proceedings of the International Conference on Supercomputing 2021.

This work supports the hypothesis that GPU performance scaling was limited by memory latency and affirms that the proposed modifications can improve performance across multiple hardware generations.

Chapter 5

Efficient and Transferable Multi-Scale Performance Autotuning

As diverse HPC systems are built, many opportunities arise for applications to solve larger problems than ever before. Given the significantly increased complexity of these HPC systems and application tuning, empirical performance tuning, such as autotuning, has emerged as a promising approach in recent years. Despite its effectiveness, autotuning is often a computationally expensive approach. Transfer learning (TL)-based autotuning seeks to address this issue by leveraging the data from prior tuning. Current TL methods for autotuning spend significant time modeling the relationship between parameter configurations and performance, which is ineffective for few-shot (that is, few empirical evaluations) tuning on new tasks.

We propose the first generative TL-based autotuning approach based on the Gaussian copula (GC) to model the high-performing regions of the search space from prior data and then generate high-performing configurations for new tasks. This allows a sampling-based approach that maximizes few-shot performance and provides the first probabilistic estimation of the few-shot budget for effective TL-based autotuning.

For the benefit of broader adoption and greater utility, the technique must also support transfer that simultaneously scales input data and compute resources. The practice of scaling data with compute together is typically referred to as weak scaling, as opposed to strong scaling where the input data scale is static and more compute is applied to this fixed problem, which is referred to as strong scaling. In HPC environments, weak scaling is frequently used to scale computations to available resources, solving problems that are infeasible to approach with less compute resources, however tuning these large-scale systems carries extreme cost. Under-utilizing large-scale resources carries its own costs, especially in terms of energy expenditure, but tuning must be as efficient as possible so as to not cost more energy than the optimized configuration will save. We believe that our proposed transfer technique can fill in this gap by immediately transferring from smaller-scale tuning data to the larger-scale operation of interest.

In the following sections, we outline our current methodology and the necessary work to complete this portion of the proposal.

5.1 The Gaussian Copula Model

The generative modeling-based TL approach that we propose is based on the GC, a well-known multivariate probability distribution in statistics literature [46]. Let us consider a simple autotuning example with three variables: the input scale, one tunable parameter, and a single performance metric. After collecting performance data across multiple parameter values at several input scales, we can model the distribution of the values of the three variables independently. These are referred to as marginal distributions. The three variables are correlated, however, so we also model their interactions with one another using a joint probability distribution.

Copulas are a class of statistical modeling techniques that decompose a multivariate probability distribution into its marginal distributions and use a separate function to couple those distributions. This approach allows us to specify the correlation separately via a correlation matrix. GCs adopt probability integral transform, a technique that can transform any probability distribution into a uniform distribution and vice versa. GCs use the uniform and normal distribution as the intermediate distribution to model complex joint probability distribution. This is achieved as follows. Given the values of the input variables, a covariance matrix is computed to model the correlation between variable pairs. A multivariate normal distribution is then defined using the computed covariance matrix with a zero mean vector. The probability integral transform is applied to convert the marginals of the Gaussian distribution to uniform distributions. The uniform marginal distributions are then converted into the original distribution using the probability integral transform. We refer the reader to the work of Masarotto and Varin [47] for a more detailed mathematical exposition of the statistical model and mechanics.

5.2 Proposed Methodology

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

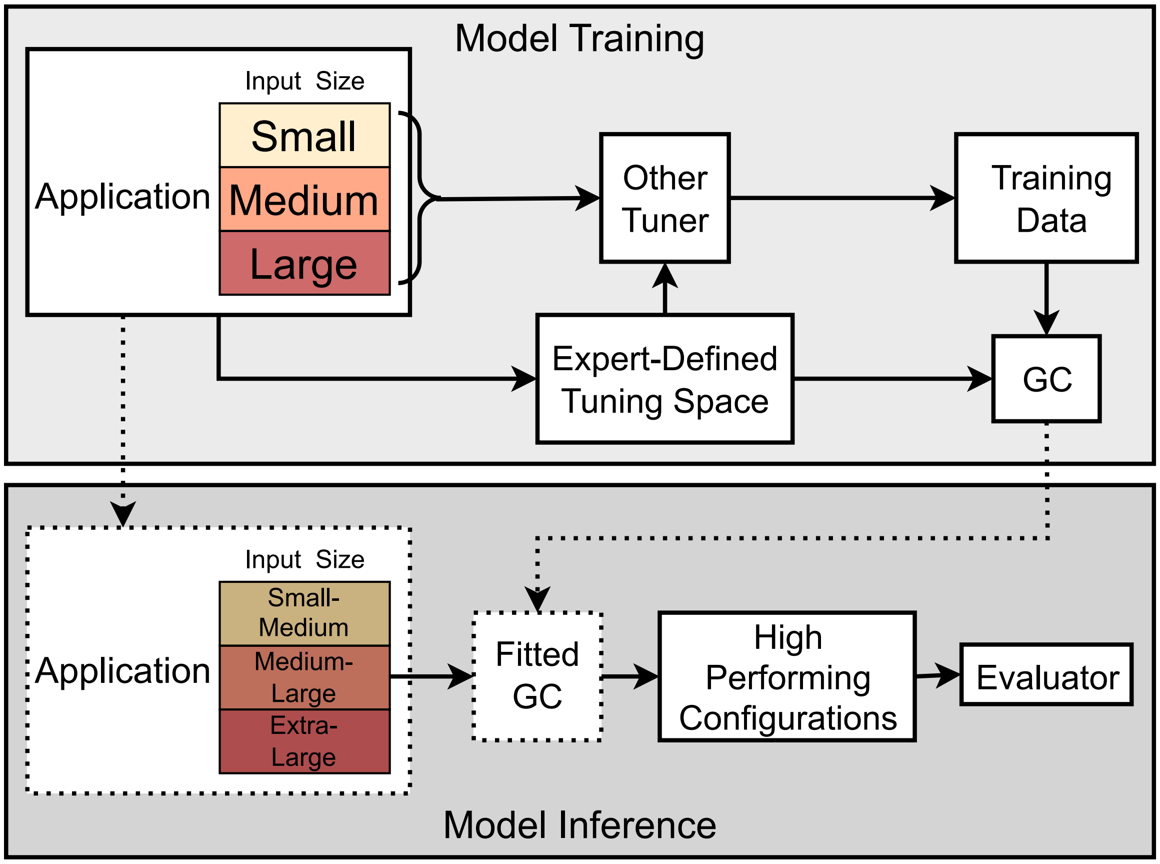


Figure 5.1: TL-based Autotuning Framework Using GC. TOP: Model Training, which uses GC to train fitted models with data collected from source tasks (multiple input sizes of an application) in a human-designed tuning space. **BOTTOM: Model Inference,** which uses the fitted GC models to propose high-performing 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. We will begin by only altering the scale of input data, leaving the hardware scaling as fixed. As such, each source task will correspond to a different input scale for the same application and the same hardware configuration, and the tuning space is specified via application source code annotation and predefined parameter values. Later, we will expand upon this basis for transfer to tackle the more complex problem of simultaneously tuning for input and system scaling. As such, the tasks will define both the scale of input data and the available compute resources. Our model's tuning space will be able to represent any configurable component that may affect performance, including source code, compilation flags, environment settings and runtime arguments.

Some initial tuning must be performed on source tasks to form the dataset basis for transfer learning. The source tasks, tuning space, and source task scales are presented to an existing autotuner, of which many prior works have already defined [30, 31, 34, 35, 36, 39, 40]. The autotuner is provided a fixed evaluation budget for each source task to collect a small training dataset that attempts to capture configurations that yield varied 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.

5.2.1 Model Training

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

Variable Preprocessing The conventional GC only models real variables, but compelling autotuning spaces often include mixed-integer (discrete, integer, and categorical) variables. To address this issue, we will adopt a new GC approach proposed for synthetic data generation [43]. In this GC approach, numeric variables with limited real or integer values 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 Autotuner GC can be used as an autotuner given a dataset of observed configurations in a defined tuning space. Each tunable parameter in an autotuning space can be represented by a marginal variable in the GC model; the combination of parameter interactions can be described through the joint model of the GC, permitting representation of distributions throughout a tuning space. The resulting fitted model identifies appropriate marginal and joint distributions. The fitted model can generate new configurations through the GC's probability integral transform, which statistically resembles the training data's observed marginal and joint behaviors. Any configurations a GCs generates can be empirically evaluated to determine its fitness. As a distribution-based model, attempting to re-incorporate the observed data will only increase the model's existing biases, so the observed fitness of transferred configurations during inference are not re-incorporated into the model itself. This prevents the GC from guaranteeing long-term convergence in its search, and instead it must maximize transfer performance within a very limited number of evaluations. Because the GC does not require as much training data as other alternatives, the best use-case for this model will be permitting data-constrained transfer for rapid but meaningful space exploration, which can kickstart other models for longer-term convergence in the transfer search.

5.2.1.1 GC Model Fitting for Few-Shot Tuning

Unlike existing TL autotuning methods, the GC does not strictly benefit from access to ever-increasing volumes of training data. Because the model is not regressive, the GC only delineates the likelihood of combinations of parameters and cannot effectively rank configurations relative to one another. This means that the GC has no mechanism to disfavor configurations that may yield subpar performance aside from completely omitting the configuration from consideration. Because the GC generates new outputs based on the distributions in its training data, omitting information biases the GC against representing it in the future. As such, we can intentionally filter training data based on quantiles of observed high-performance and fit the GC only to high performing configurations. We refer to this practice as "quantile filtering". Quantile filtering biases the GC to represent commonly observed traits that yield high performance and implicitly disfavors combinations of parameters that were observed to yield lower performance. Since we limit the extent of initial tuning, some optimal combinations are not presented to the GC, however if initial tuning is extensive enough and the optimal configurations have some self-similarity, it will not be unlikely for the missing optimums to be randomly sampled from the model.

5.2.2 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.

5.2.2.1 Conditional Sampling

Quantile filtering on the training data 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. This permits conditional sampling; we specify the condition that every sample indicates a particular task. Conditional sampling imposes arbitrary constraints on the model during generation, which affect the distributions of unconstrained variables before generating their values. Conditional sampling prompts 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 mechanics "recover" a transferred relationship for the new task.

Conditional sampling is particularly effective for the GC because it identifies and isolates critical information from the model. Since the GC operates on filtered high-performing source data, conditional sampling generates configurations that are expected to perform well for the transferred task.

5.2.2.2 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 configurations without replacement from a distribution that spans $|C|$ potential candidates. Within the distribution are $|I|$ ideal candidates, which are optimal or near-optimal. Frequently, the top 1% of evaluations in real-world benchmarks have nearly equivalent performance. Identifying one or more of these top 1% candidates 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:

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

The probability of observing the top-1% or greater performing configurations by utilizing uniform random sampling is explicitly 1% per evaluation attempt. Sampling in this manner without replacement is obviously a poor search strategy that rapidly diminishes in effectiveness as the desired probability band shrinks or as the search space grows. However, by using a GC with quantile filtering on its training data, some configurations become statistically improbable or impossible to generate, effectively 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 of the configuration space, $|C|$. The best filtering quantile will maintain a faithful representation of the optimal distribution and limit $|C|$ without over-specifying the search space since the latter also contributes to the probability of the few-shot success. We can determine the reduced $|C|$ from the GC by estimating the number of unique samples generated by the fitted GC. Nevertheless, we can only measure the resulting change in $|I|$ with exhaustive evaluation, which is needed to quantify the probability of success in Equation 5.1.

The exact reduction in $|I|$ is unknown but can be modeled as a proportion of the eliminated configurations, which represents the opportunity cost of some removed configurations being optimal. With adjusted $|C|$ and $|I|$, 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 Addressing Limitations for Autotuning

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

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. However, in other cases such as simultaneously tuning input and system scale, more complicated relationships are expected and alternatives to the typical correlation mechanism may be required for the technique to work.

False Ordering and Transitivity for Categories The SDV [43] 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.

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, environment settings and runtime arguments 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, but compiler flags are typically represent a large tunable space that has a broad variety of compelling options for performance improvement.

More complex tuning spaces require explicit measures, such as importance sampling, to identify the most critical variables to tune. Our current techniques continue to rely on experts to provide compelling options for source code annotation and to narrow down the scope of potential environment settings, compiler flags, and runtime arguments to tune. Because we rely upon human intervention to define the tunable space, we also rely on the space designers to curate an appropriately sized set of variables that are reasonably compelling from a performance perspective.

5.4 Proposal Status

We have successfully implemented the first stage of the proposal, which was published in the Proceedings of the International Conference on Supercomputing 2023.

This work supports the hypotheses that utilizing quantile filtering for the GC can permit highly aggressive transfer to new tasks within the evaluation budget described by hypergeometric sampling. We successfully transfer near-optimal areas between tasks based on input data scale for a variety of applications and tuning spaces, however the compute scale of these tasks remained constant. These transfer problems require some sophistication to properly identify changes in performance characteristics based on input scale, but weak-scaling requires even more sophistication. For instance, an application may transition from a compute-bound bottleneck to a memory-bound bottleneck, which can be effectively tuned by identifying the transition and appropriately scaling the response to a particular input data scale, but additional compute resources may alter these breakpoints in more advanced ways.

Initial work on weak scaling problems indicates that the linear correlation used in the GC model may be insufficient to properly capture the complexity of weak scaling relationships. The implementation of our prototype is prepared to experiment with different correlation models, allowing us to explore alternatives that may improve model capabilities for these problems.

Chapter 6

Summary Discussion and Future Work

This proposal seeks to address the necessity of collaboration between computer hardware and software for effective performance, especially in HPC environments. We demonstrate the opportunities present for hardware to better benefit software by analyzing novel SPLIC technologies and how they affect software performance relative to familiar air-cooled environments. We provide an example of modifying software to better adhere to hardware accelerator design by optimizing the GPU performance of Word2Vec based on core principles of hardware implementation and strengths, permitting both retroactive and proactive performance boosts without need for code modification specific to newer hardware generations. Finally, we demonstrate how the presence of tunable hardware and software requires special care to properly configure both components for maximum performance. To increase the efficiency of repeated optimization on similar tuning spaces, we introduce a novel performance autotuning technique that requires minimal data to produce insightful suggestions for attaining high performance within a small number of empirical trials.

6.1 Research Questions to be Answered

Impact of Immersion Cooling on Application Performance Our comparative studies of application performance between air-cooled and SPLIC environments will demonstrate the impact of cooling not just on energy efficiency, but on application performance. We anticipate our research on SPLIC systems will demonstrate that SPLIC is more energy efficient and has greater passive and active heat dissipation than air-cooling, but is less responsive to hotspots in the system and can actually be slower to address thermal issues in the system when heat accumulates. We expect that applications which thoroughly saturate available resources, especially power-hungry resources such as GPU compute, may continue to experience thermal throttling and may even induce thermal throttling in other components due to heat diffusion properties of the coolant fluid.

Properly Leveraging GPU Acceleration: A Case Study in Word2Vec Optimization Prior implementations of the GPU algorithm for Word2Vec experienced insufficient performance scaling due to hardware bottlenecks that were poorly handled by the software. Easing these bottlenecks can permit proper performance improvements from successive generations of GPU hardware even without code modification specific to particular architectures, improving the general benefit from acceleration with limited cost of development. While not all applications suffer from the same issues that Word2Vec experienced, nor do all application developers have the resources to rethink implementation for different accelerator architectures, our generation-agnostic optimizations demonstrate the effectiveness of general performance enhancements that can be tuned to successive hardware generations.

Efficient and Transferable Multi-Scale Performance Autotuning Maximizing performance is not simple even when both hardware and software perspectives are properly accounted for. Different scales of data and hardware availability can dramatically alter performance constraints even on a fixed system, much less on a similar architecture that experiences a partial upgrade. Vast search spaces that are infeasible to tune by hand and the nontrivial variety of performance needs across different scales necessitate automatic tools that can efficiently leverage existing knowledge in new contexts for rapid performance tuning.

We have demonstrated the GC's capability to rapidly prototype promising configurations based on input data scaling with minimal training data, but are still improving the technique to handle weak-scaling performance tuning where both input data scale and hardware resources are tuned simultaneously. We anticipate our contributions will greatly reduce the overhead of continuous performance tuning and permit additional tuning at additional data and system scales. By reducing the cost of these tuning efforts, users of all kinds of applications can benefit from improved performance over the lifetime of their work and the system.

6.2 Recommendations for Further Research

We anticipate multiple future directions for research beyond this proposal.

Our work on SPLIC environments can be studied in a variety of other environments, such as those equipped with forced induction components that have localized pumps to increase coolant flow rate around critical hot spots. There are also many varieties of server designs for SPLIC including different pump configurations, coolants and container form factors that may be compared with our own results to indicate better ways to utilize SPLIC technologies. Other accelerators have different hotspots and thermal constraints, which can also be studied in SPLIC environments to determine their suitability and innovations in SPLIC technology that may better support the massive varieties of accelerator hardware.

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.

Finally, we look forward to other non-regressive transfer learning approaches for performance autotuning to further explore the regime of data-constrained few-shot transfer learning. We also expect that modified systems and applications can benefit from similar transfer tuning approaches, permitting the knowledge gained from tuning previous iterations to be reused for even broader and longer-lived benefit to system users.

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] 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.

[5] 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.

[6] Henry C. Coles and Steve E. Greenberg. Direct liquid cooling for electronic equipment. Technical report, Lawrence Livermore National Laboratory, 03/2014 2014.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] Martin J. Sklar. Fonctions de repartition a n dimensions et leurs marges. In *Fonctions de repartition a n dimensions et leurs marges*, 1959.

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