Harnessing Raspberry Pi's Potential for Mathematical Workloads

Shamma Alblooshi

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

This study explores the optimization of two computational workloads, Fibonacci sequence calculation and matrix multiplication, specifically tailored for the Raspberry Pi platform. Through techniques like memoization, dynamic programming, and compiler-level adjustments, the research achieves significant efficiency improvements. The approach applied here serves as a blueprint for similar optimizations across various applications. Additionally, the study lays the groundwork for future exploration into the distributed computing paradigm, utilizing an array of Raspberry Pi devices to create a cost-effective system that rivals traditional servers. The research not only demonstrates optimized implementations for two critical workloads but also suggests a broader applicability of these methods, opening avenues for innovation in both academia and industry.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Dr. Yuvraj Patel, for his unwavering support and guidance throughout my dissertation. His weekly meetings and exceptional assistance have been instrumental in my academic success. Without his expertise and dedication, this achievement would not have been possible. His commitment to my growth and development as a student has truly made a significant impact. I am incredibly grateful for his continuous encouragement and the valuable knowledge he has imparted to me. It has been an honor to learn from him, and I am privileged to have had such an exceptional mentor.

I would also like to thank my husband and family for their mental support during my studies.

Table of Contents

Chapter 1

Introduction

The Raspberry Pi has emerged as a game-changer in the world of single-board computers, offering an affordable and versatile computing platform that has captured the imagination of hobbyists, educators, and technology enthusiasts alike. While desktop computers and servers have long been the go-to options for high-performance computing, the Raspberry Pi presents a compelling alternative with its compact form factor, cost-effectiveness, and low power consumption. Over the years, the Raspberry Pi ecosystem has evolved, culminating in the release of the Raspberry Pi 4 Model B, which boasts significant improvements in terms of CPU power, memory capacity, storage options, and networking capabilities.

In this project, we embark on a captivating exploration to determine if an array of Raspberry Pis can match or even surpass the performance of a traditional server. Individually, Raspberry Pis may be considered modest in their computing capabilities, but when combined into an array, the aggregate resources at our disposal rival those of a dedicated server. The fundamental question we seek to answer is whether the cumulative power of multiple Raspberry Pis can outshine that of a conventional computing setup.

Rather than focusing solely on hardware specifications, our project emphasizes the identification of workloads that are well-suited to run on an array of Raspberry Pis. While previous research endeavors have approached this from the perspective of selecting specific workloads and evaluating their performance on Raspberry Pis [2, 8, 11], our methodology takes a different path. Instead, we aim to understand the inherent characteristics of the Raspberry Pi architecture and discover workloads that seamlessly align with its strengths and limitations.

The absence of an L3 cache and the presence of Gigabit Ethernet are just a few examples of the distinguishing features of Raspberry Pis that shape their performance profiles. Leveraging this knowledge, we will explore workloads that can efficiently utilize the available resources, such as optimizing computations to fit within the L1 and L2 caches. By considering the unique traits of Raspberry Pis and their potential bottlenecks, we can identify workloads that are most likely to benefit from this array-based approach.

Through this project, we hope to unlock the untapped potential of Raspberry Pis as a collective computing resource. By pushing the boundaries of what these small yet mighty devices can achieve when united, we aim to redefine their role in various computational domains. Additionally, our findings will shed light on the optimization strategies and workload considerations necessary to harness the true power of Raspberry Pi arrays.

1.1 Contributions

Our contributions to this project focused on identifying and optimizing mathematical applications for execution on Raspberry Pi devices. Our key contributions include:

- Identification of math applications suitable for Raspberry Pi arrays.
- In-depth analysis and optimization of the math algorithms:
  - Implementation of multithreading techniques for parallel execution.
  - Resolution of segmentation faults encountered with larger matrix sizes.
  - Optimization of cache behavior and memory access patterns to minimize cache misses.
  - Utilization of transparent page tables and page management techniques to reduce TLB refills and enhance memory access efficiency.
- Comprehensive exploration of optimization techniques at various levels:
  - Robust error handling mechanisms for stability and reliability.
  - Cache optimization to improve data locality and reduce memory access latency.
  - Memory management enhancements to minimize memory swapping and improve computational throughput.

1.2 Outline of report

Chapter 2 provides the necessary background information and basic knowledge of the Raspberry Pi highlighting its characteristics as a low-cost single-board computer and its popularity in various applications

Chapter 3 describes the overall research design and approach

Chapter 4 provides a detailed implementation of the selected algorithms and optimizations and an overview of the process followed for each stage of optimization

Chapter 5 provides a discussion of the successes, challenges, and unexpected outcomes

Chapter 6 provides a summary of key findings and contributions and future work

Chapter 2

Background

In this chapter, we provide the background information that is necessary for the subsequent chapters. In Sec. 2.1 we provide a brief overview of the Raspberry Pi, highlighting its characteristics as a low-cost single-board computer and its popularity in various applications. This chapter concludes in Sec. 2.2 where we mention any relevant previous research that has explored the use of Raspberry Pi arrays or focused on specific workloads and highlight the unique contribution of our project in terms of identifying suitable mathematical applications.

2.1 Raspberry pi

2.1.1 Evolution of Raspberry Pi

The evolution of Raspberry Pi devices over time has witnessed remarkable advancements, empowering users with enhanced capabilities and improved performance. In its early stages, Raspberry Pi was characterized by limited CPU power, modest memory capacity, and basic storage options. However, as technology progressed, subsequent models introduced significant upgrades [7].

The latest iteration, Raspberry Pi 4 Model B, stands as a testament to these advancements. It offers notable improvements, such as increased CPU power, expanded memory options, and enhanced storage capabilities [6]. The Raspberry Pi 4 Model B is powered by a quad-core ARM Cortex-A72 processor, providing a substantial boost in computational capacity compared to its predecessors. This increase in processing power enables users to tackle more demanding tasks and run resource-intensive applications.

Furthermore, the Raspberry Pi 4 Model B offers a wider range of memory options, allowing users to choose from variants with 2GB, 4GB, or even 8GB of RAM. This expanded memory capacity facilitates smoother multitasking and enables the execution of memory-intensive applications with ease. Storage options have also seen improvements in the latest Raspberry Pi models. The Raspberry Pi 4 Model B incorporates USB 3.0 ports and supports booting from high-speed external storage devices, such as USB flash drives or solid-state drives (SSDs). This opens up new possibilities for faster data transfer and efficient storage management.

2.1.2 Raspberry Pi Components

The Raspberry Pi is a versatile single-board computer that integrates various components to enable its functionality. Each component plays a critical role in the overall operation and capabilities of the Raspberry Pi. The key components include [9]:

- System-on-a-Chip (SoC): At the heart of the Raspberry Pi is the System-on-a-Chip, which combines several essential components into a single integrated circuit. The SoC typically includes the CPU (Central Processing Unit), GPU (Graphics Processing Unit), memory controllers, and other necessary peripherals.
- CPU (Central Processing Unit): The CPU is responsible for executing instructions and performing calculations. Raspberry Pi models employ different CPU architectures, such as ARM-based processors. Over the years, Raspberry Pi devices have seen significant improvements in CPU power, with newer models featuring faster clock speeds and increased processing capabilities.
- Memory (RAM): The Raspberry Pi incorporates RAM (Random Access Memory) to provide temporary storage for data and instructions that the CPU actively uses. RAM capacity varies across Raspberry Pi models, ranging from 2GB to 8GB, allowing for efficient multitasking and handling memory-intensive applications.
- Storage Options: Raspberry Pi devices support various storage options, including microSD cards, USB flash drives, and external hard drives. The primary storage medium is usually the microSD card, which stores the operating system, applications, and user data.
- Graphics Processing Unit (GPU): The GPU in Raspberry Pi models enables accelerated graphics rendering, making it suitable for multimedia applications

and graphical interfaces. The GPU provides hardware acceleration for graphics-related tasks, offloading the workload from the CPU and enhancing overall performance.

2.1.3 Cost-effectiveness and Power Efficiency

Raspberry Pi devices offer significant advantages in terms of cost-effectiveness and power efficiency when compared to traditional desktops and servers. These advantages make them an attractive choice for various applications [7].

Raspberry Pi boards are highly cost-effective compared to traditional desktops and servers. The affordable price point of Raspberry Pi models, ranging from $35 to $70 [12], makes them accessible to a wide range of users, including hobbyists, students, and small-scale projects. This low cost opens up possibilities for cost-sensitive projects, where deploying multiple Raspberry Pi units can be more affordable than investing in expensive server hardware.

Furthermore, Raspberry Pi devices eliminate the need for additional components commonly found in desktops and servers, such as graphics cards, high-capacity storage drives, and complex cooling systems. The all-in-one nature of the Raspberry Pi reduces overall hardware costs and simplifies the setup process.

Raspberry Pi devices are designed with power efficiency in mind. Compared to traditional desktops and servers that consume considerable amounts of power, Raspberry Pi boards have significantly lower power requirements [14]. The low power consumption makes them ideal for applications where energy efficiency is critical, such as IoT deployments, remote monitoring systems, or solar-powered setups. The power-efficient nature of Raspberry Pi devices also translates to reduced operating costs. By utilizing low-power components and optimized power management techniques, Raspberry Pi boards can deliver efficient performance while keeping electricity bills at a minimum.

Moreover, Raspberry Pi's cost-effectiveness and low power requirements enable the possibility of scaling up deployments with multiple units. Instead of relying on a single high-end server, distributing computational tasks across a cluster of Raspberry Pi devices can provide comparable performance at a fraction of the cost. This scalability allows for tailored solutions that can adapt to specific requirements, whether it's a small-scale project or a large-scale deployment.

Additionally, the flexibility of Raspberry Pi devices enables diverse applications. Their small size and low power consumption make them suitable for embedding into various devices and environments, providing computing capabilities where traditional desktops and servers may not be feasible or practical.

2.2 Related Work

Previous research has explored the potential of Raspberry Pi as an alternative to traditional cloud computing for big data applications. Hajji and Tso [8] constructed a Raspberry Pi cluster and evaluated its feasibility for various workloads, analyzing factors such as energy consumption, memory usage, CPU usage, and network throughput. However, their experiment revealed that the Raspberry Pi's response time was 2809 requests per second for a 1 KB workload, with a noticeable decline as the workload increased.

Another study by Mados et al.[11] focused on using Raspberry Pi 3 as a downsized web server design, comparing its performance to a 1U Intel Rack server. The authors assessed energy consumption and response time for a range of HTTP requests. They found that the Raspberry Pi outperformed the traditional server platform, with a significantly lower average processor load of 23% when serving 10,000 concurrent HTTP requests, compared to the rack server's average load of 87%.

In a different paper, Dubey and Kagdi [4] investigated the performance of a Raspberry Pi cluster supercomputer specifically for matrix multiplication tasks. While not comparing it to a traditional server, their study aimed to optimize the performance of Raspberry Pi cluster supercomputers. The authors demonstrated that a cluster of Raspberry Pis is a cost-effective and energy-efficient solution for matrix multiplication, outperforming individual Raspberry Pi setups. They observed that as the number of Raspberry Pis in the cluster increased, the time required to complete matrix multiplication tasks decreased, resulting in improved performance. The paper emphasized the importance of workload distribution and minimizing communication latency to enhance the performance of Raspberry Pi cluster supercomputers for matrix multiplication tasks.

While past research has focused on utilizing Raspberry Pis for high-performance computing, their methodology has been restricted to choosing specific workloads and comparing their performance against traditional servers. In contrast, our study takes a different approach by aiming to identify workloads that can optimally operate on an array of Raspberry Pis, considering their unique architecture, limitations, and capabilities.

Chapter 3

Methodology

3.1 Workload Selection Decisions

The successful utilization of Raspberry Pi arrays relies on the careful selection of workloads that align with the architecture and capabilities of these devices. By identifying and choosing appropriate workloads, we can maximize the performance and efficiency of Raspberry Pi arrays, taking into account factors such as cache sizes, memory constraints, and networking capabilities.

One key consideration in workload selection is the cache sizes of Raspberry Pi devices. These devices typically have limited cache sizes, such as L1 and L2 caches. It is crucial to choose workloads that can effectively utilize these caches to minimize cache misses and improve overall performance. By understanding the characteristics of the workload and its memory access patterns, we can select workloads that are well-suited for the cache hierarchy of Raspberry Pi arrays.

Memory constraints are another important aspect to consider when selecting workloads. Raspberry Pi devices often have limited memory compared to traditional servers. Therefore, it is essential to choose workloads that can efficiently manage memory usage and avoid excessive swapping or page faults. Workloads that require excessive memory or have high memory bandwidth demands may not be suitable for Raspberry Pi arrays. Instead, selecting workloads that can optimize memory utilization and minimize data movement can greatly improve performance.

Additionally, networking capabilities should be taken into account when selecting workloads for Raspberry Pi arrays. While Raspberry Pi devices offer networking options such as Ethernet, the bandwidth and latency may be different compared to traditional server environments. Workloads that heavily rely on network communication or data transfer may need to be adapted or optimized to take advantage of the networking capabilities of Raspberry Pi arrays. By considering the network characteristics and selecting workloads accordingly, we can ensure the efficient utilization of these devices.

The selection of workloads plays a crucial role in effectively utilizing Raspberry Pi arrays. By considering the mentioned points above, we can identify workloads that align with the architecture of Raspberry Pi devices. This strategic approach enables us to optimize performance, minimize resource bottlenecks, and achieve efficient utilization of Raspberry Pi arrays in various computing scenarios .

3.1.1 Matrix Multiplication

Matrix multiplication is a good baseline for analyzing the performance of Raspberry Pi devices due to its computational intensity and relevance in various domains, such as scientific computing, machine learning, and image processing [4]. By selecting matrix multiplication as our workload, we can gain valuable insights into the performance characteristics and limitations of Raspberry Pi devices in executing computationally intensive tasks.

There are several reasons why matrix multiplication is a good starting point for performance analysis on Raspberry Pi:

Matrix multiplication involves a significant number of mathematical operations and data dependencies, making it a computationally intensive task. By analyzing the performance of matrix multiplication, we can assess the ability of Raspberry Pi to handle such intensive computations and identify any bottlenecks or areas for improvement. In addition, it involves accessing and manipulating large amounts of data stored in matrices. This places significant demands on the memory subsystem, including the RAM and cache hierarchy. By analyzing the memory usage and cache behavior during matrix multiplication, we can evaluate the efficiency of memory access patterns and identify opportunities for optimization.

Matrix multiplication also exhibits inherent parallelism, allowing for the utilization of multiple CPU cores or threads. Raspberry Pi devices typically feature multiple CPU cores, making them suitable for parallel execution of matrix multiplication. By analyzing the performance of matrix multiplication with varying degrees of parallelization, we can assess the scalability and efficiency of parallel execution on Raspberry Pi.

3.1.2 Fibonacci

The Fibonacci series is another captivating subject that finds itself woven into the tapestry of advanced mathematics, nature, statistics, and even agile development. It's not just an elementary sequence of numbers that appeals to mathematicians but has a deep-rooted presence in various fields, shaping the way we understand patterns and growth.

One of the primary reasons for its inclusion in our performance evaluation on Raspberry Pi is its computational complexity. The recursive algorithm used to calculate Fibonacci numbers, especially for larger terms, is known to be computationally intensive. By examining the efficiency and speed of Raspberry Pi devices in computing the Fibonacci sequences, we hope to gain a clearer understanding of their prowess in managing intricate recursive function calls.

Additionally, the Fibonacci series, when enhanced with optimization techniques like memoization or dynamic programming, offers an in-depth study into memory access patterns. Given the series' nature, where previously calculated terms are stored and subsequently retrieved, it presents a chance to challenge the memory subsystem of the Raspberry Pi. Insights into how these devices manage such memory-intensive tasks can potentially unveil avenues for further enhancement, making them more adept at handling similar operations.

Beyond its computational and memory challenges, the Fibonacci series also boasts of extensive real-world applications. It is not confined to textbooks but finds practical applications in predicting natural phenomena, aiding in statistical analyses, and even being a part of agile software development techniques. Thus, when we scrutinize the performance characteristics of Raspberry Pi devices in the context of the Fibonacci series, we indirectly gauge their competency to tackle other applications that draw from similar mathematical paradigms.

3.2 Experimental Setup

The table presents an overview of the specifications of the Raspberry Pi 4 Model B (8 GB) single-board computer that was used during this project. It provides key details about its CPU, CPU cores, CPU caches, RAM, storage, and the supported operating system [15]. These specifications highlight the capabilities and resources of the Raspberry Pi 4 Model B, making it a versatile and powerful option for various projects and applications.

Table 3.1: Specifications of Raspberry Pi 4 Model B (8 GB)

Component	**Specification**
CPU	Broadcom BCM2711, Quad-core Cortex-A72
CPU Clock Speed	1.5 GHz
CPU Cores	4
CPU Caches	L1: 48 KB instruction cache, 32 KB data cache
	L2: 1 MB per core
RAM	8 GB LPDDR4-3200 SDRAM
Storage	MicroSD card slot
Operating System	Ubuntu 22.04

3.3 Evaluation methods

3.3.1 Tools

To measure the performance improvements achieved through our optimization efforts on Raspberry Pi devices, we will utilize several evaluation metrics and measurement techniques.

Profiling tools, such as perf [10], can be employed to gather detailed performance data. These tools analyze the execution of the program at run time, providing valuable insights into the time spent in different functions, cache behavior, memory access patterns, and potential bottlenecks.

System monitoring tools like top and htop [13] can be used to monitor system-level metrics during workload execution. These tools provide information on CPU utilization, memory usage, disk I/O, and network activity. By monitoring these metrics, we can assess the impact of our optimizations on resource utilization and identify potential areas of improvement.

3.3.2 Metrics

In order to assess the effectiveness of our optimization techniques and strategies, we will employ various evaluation metrics to measure the impact of our efforts. These metrics will help us quantitatively analyze the performance improvements, identify potential bottlenecks, and gauge the overall efficiency of our optimizations on Raspberry Pi devices. In addition, it will allow us to identify successful optimizations and fine-tune our strategies to further enhance the overall performance and efficiency of computations. We will first measure the performance of our optimized workloads in terms of execution time, instructions per cycle (IPC), and overall throughput. By comparing the performance of the optimized workloads against the baseline, we can assess the extent to which our optimizations have improved computational efficiency.

Furthermore, we will monitor and record the occurrence of segmentation faults during the execution of our optimized workloads. By analyzing the frequency and reasons behind these faults, we can evaluate the success of our efforts in reducing segmentation faults and enhancing the stability of computations.

Cache behavior metrics will also be analyzed such as cache hit rate, cache miss rate, and cache utilization, to assess the effectiveness of our cache optimization techniques. By optimizing data locality and minimizing cache misses, we aim to improve cache performance and reduce memory access latency. In addition, we will analyze memory access patterns, including the frequency and efficiency of memory accesses, to evaluate the impact of our memory optimization strategies. For workloads utilizing multithreading, we will evaluate the scalability and efficiency of parallel execution. We will measure metrics such as speedup, thread utilization, and synchronization overhead to assess the effectiveness of multithreading in improving performance.

Chapter 4

Implementation and Results

4.1 Matrix multiplication

4.1.1 Algorithm

Matrix multiplication is a fundamental operation in linear algebra and finds applications in various scientific and engineering domains. The algorithm computes the product of two matrices by performing a series of multiplications and additions. Given two matrices, A and B, the resulting matrix C is obtained by multiplying the corresponding elements of each row in matrix A with the corresponding elements of each column in matrix B and summing the products.

4.1.2 Unoptimized Matrix Multiplication Results

This subsection aims to explain the results of matrix multiplication without any optimizations. The purpose is to establish a baseline performance for comparison with the optimized versions that will be discussed in the subsequent sections. The unoptimized implementation serves as a starting point, and the insights gained from this section will guide the optimization strategies discussed later.

In addition, this subsection will contain an in-depth explanation of each metric presented in the tables. This serves as a baseline to understand the various performance indicators. However, please note that in the subsequent sections, I will assume familiarity with these metrics and their implications. This approach allows us to delve directly into the optimizations and their impact without reiterating the explanations of each metric. Thus, this section serves as a comprehensive introduction to the metrics, enabling a better understanding of the subsequent discussions on optimization strategies.

Table 4.1: Branches Results for Non-optimized Code

Metrics	500x500	1000x1000	2000x2000	5000x5000
Br mispred	655,736	2,154,564	9,815,120	252,051,974
Br pred	151,042,577	1,065,316,696	8,291,453,122	135,316,732,271
Bus accesses	27,187,485	690,614,024	2,256,012,423	530,334,394,214
Bus cycles	1,676,642,118	21,293,227,885	162,702,449,645	6,698,329,807,220

Table 4.1 presents the performance metrics for matrix multiplication operations at various matrix sizes. The table features four metrics: Branch mispredictions (Br mispred), Branch predictions (Br pred), Bus accesses, and Bus cycles. Br mispred indicates the number of times the processor predicted the wrong outcome of a branch instruction. Br pred represents how many times the processor predicted the outcome of a branch instruction. Bus accesses indicate how many times the processor accessed the system bus. Finally, bus cycles measure the efficiency of bus utilization.

Notably, there is a sharp rise in each of these values as the matrix size increases. For instance, the number of Branch Mispredictions goes up from 655,736 for a 500x500 matrix to a staggering 252,051,974 for a 5000x5000 matrix. Similar trends are observed with the other metrics as well, highlighting the strain unoptimized matrix multiplication places on the system's resources. Furthermore, the number of mispredictions increasing can be a bad indicator of performance because each misprediction results in wasted cycles since the CPU has to flush the wrongly predicted instructions.

Table 4.2: TLB and Page Faults Results for Non-optimized Code

Metrics	500x500	1000x1000	2000x2000	5000x5000
l1d_tlb_refill	124,660,549	1,004,186,491	8,030,409,938	125,734,877,957
l1i_tlb_refill	6,332	89,537	686,664	23,556,368
dTLB-load-misses	124,410,946	1,003,635,311	8,028,255,825	125,701,943,182
dTLB-store-misses	7,187	89,291	736,701	30,795,421
page-faults	850	3,057	11,862	73,439

Table 4.2 shows the TLB (Translation Lookaside Buffer) performance metrics for non-optimized matrix multiplication. The metrics include Level 1 Data TLB refills (l1d_tlb_refill), Level 1 Instruction TLB refills (l1i_tlb_refill), Data TLB load misses (dTLB-load-misses), Data TLB store misses (dTL-store-misses), and page faults (page-

faults). The l1d_tlb_refill measures the number of times the Level 1 Data TLB had to be refreshed. The TLB is a cache that the memory management unit (MMU) uses to hold recent translations of virtual memory to physical memory. The l1i_tlb_refill measures the number of times the Level 1 Instruction TLB had to be refreshed. dTLB-load-misses metric represents the number of times a data reference needed for an operation missed the TLB and had to be retrieved from the main memory, which is a much slower process. Whereas the dTLB-store-misses metric represents the number of times a store operation could not find the requested data in the TLB. Finally, a page fault occurs when a program tries to access a portion of memory that is not currently available in RAM.

The TLB and cache are vital parts of a CPU's architecture, designed to minimize the time it takes to access memory by storing frequently or recently accessed data. A high number of refills and misses indicate that these functionalities are not being utilized efficiently, which can impact the performance of the computations.

Regarding the page faults, the total size of the matrices for different sizes will differ based on the size of the matrices. Let's calculate the total bytes used by each matrix set and compare it with the number of page faults. The three arrays together for the 500 by 500 matrices are 3 (matrices) * 500 (rows) * 500 (columns) * 4 (bytes) = 4,000,000 bytes. With a page size of 4096 bytes, this means 4,000,000/4096 = 732 pages would be required. After applying the same method to matrices of different sizes, we observed varying numbers of page faults. Specifically, the 1000x1000 matrix resulted in 2,929 page faults, the 2000x2000 matrix had 11,719 page faults, and the largest matrix, 5000x5000, experienced 73,242-page faults. As the matrix multiplication algorithm initializes three large arrays, the first-time access to these arrays triggers page faults. This occurrence is expected since the system needs to allocate and initialize the required memory pages. The number of page faults calculated closely corresponds to the approximate event count for page faults during the initialization phase shown in Table 4.2. This analysis suggests that the number of page faults observed is within the expected range for the given matrix sizes and the matrix multiplication algorithm being used. Consequently, it validates that the page faults are a natural part of the initialization process and not a cause for concern in terms of the overall efficiency and performance of the matrix multiplication algorithm.

Table 4.3 shows the cache performance metrics for Non optimized matrix multiplication algorithm. The l1d_cache measures the number of Level 1 Data Cache accesses. The L1D cache is a high-speed cache memory that is used to store data that is frequently accessed by the CPU. The l1d_cache_refill measures the number of refills, also known as

Table 4.3: Cache Results for Non-optimized Code

Metric	500x500	1000x1000	2000x2000	5000x5000
l1d_cache	2,996,015,325	24,186,801,057	192,770,047,708	3,021,442,315,402
l1d_cache_refill	70,711,134	1,101,239,200	9,261,796,612	146,292,926,862
l1i_cache	1,506,507,434	12,205,886,640	96,993,598,658	1,528,919,539,454
l1i_cache_refill	380,177	3,923,896	29,801,876	1,113,759,702
l2d_cache	153,423,135	4,810,208,180	32,537,200,404	604,514,516,548
l2d_cache_refill	5,973,799	170,999,354	554,799,158	131,957,855,504
mem_access	2,987,533,513	24,159,974,934	192,834,608,179	3,022,327,041,086
L1-icache-loads	1,535,229,533	12,171,880,419	96,920,689,988	1,526,973,582,276
L1-icache-load-misses	321,798	3,815,650	27,625,851	1,113,568,778
cache-references	3,050,763,292	24,148,209,276	192,698,792,825	3,022,007,452,111
cache-misses	72,381,748	1,109,626,391	9,267,511,000	146,270,182,622
L1-dcache-loads	3,033,675,561	24,128,651,075	192,837,494,489	3,023,414,162,006
L1-dcache-load-misses	72,291,084	1,110,306,720	9,266,316,368	146,340,007,928
L1-dcache-stores	255,402,235	2,036,420,616	16,248,425,881	259,386,611,432
L1-dcache-store-misses	44,904	508,394	4,363,062	159,055,499

cache misses, for the Level 1 Data Cache. This occurs when the CPU tries to read from the L1D cache and finds that the data it needs is not present. The l1i_cache: This metric measures the number of Level 1 Instruction Cache accesses. The L1I cache is a high-speed cache memory that stores frequently executed instructions. The l1i_cache_refill measures the number of refills, also known as cache misses, for the Level 1 Instruction Cache. This occurs when the CPU tries to read from the L1I cache and finds that the instructions it needs are not present. The l2d_cache measures the number of Level 2 Data Cache accesses. The L2D cache is a slower but larger cache memory that stores data that is less frequently accessed than the data in the L1D cache. The l2d_cache_refill measures the number of refills, also known as cache misses, for the Level 2 Data Cache. This occurs when the CPU tries to read from the L2D cache and finds that the data it needs is not present. The mem_access metric measures the total number of memory accesses, both read and write operations. This includes accessing both cache and main memory. The L1-icache-loads measure the total number of instruction fetches from the Level 1 Instruction Cache. The L1-icache-load-misses measures the total number of instruction fetch misses from the Level 1 Instruction Cache. This occurs when the CPU tries to fetch an instruction from the L1I cache and finds that the instruction it needs is not present. The cache-references measure the total number of cache references or attempts to access the cache. This includes both hits (successful finds) and misses. The cache-misses measures the total number of cache misses, or unsuccessful attempts to find the required data in the cache. The L1-dcache-loads measure the total number of data fetches from the Level 1 Data Cache. The L1-dcache-load-misses measures the total number of data fetch misses from the Level 1 Data Cache. This occurs when the CPU tries to fetch data from the L1D cache and finds that the data it needs is not present.

The cache miss percentages for different matrix sizes (2.373% for 500x500, 4.595% for 1000x1000, 4.809% for 2000x2000, and 4.809% for 5000x5000) indicate that a considerable proportion of data accesses result in cache misses. These percentages aren't close to zero, suggesting that a significant portion of data accesses results in cache misses. In an ideal scenario, where the entire array can fit within the L1 cache, we would expect the average cache miss penalty to be near zero. A high cache miss rate indicates the majority of the data accessed by the workload is not present in the Level 1 (L1) cache. This implies the workload isn't well-optimized for the L1 cache, leading to an increased reliance on slower tiers of memory, like the main memory, which in turn can result in lower system performance.

Table 4.4: Instructions and Time Results for Non-optimized Code

Metrics	500x500	1000x1000	2000x2000	5000x5000
instructions	5,712,103,252	45,244,310,410	361,435,936,760	5,674,513,062,959
insn per cycle	1.12	0.71	0.74	0.27
CPUs utilized	0.995	0.996	0.996	0.996
time elapsed (seconds)	2.88	36.10	274.76	11894.97

The last table presents the instructions and time performance metrics for matrix multiplication using a non-optimized code. Task-clock metric represents the total CPU time used to execute the matrix multiplication operation. The instructions metric shows the number of instructions executed during the matrix multiplication process. Insn per cycle metric represents the number of instructions executed per CPU cycle. CPUs utilized indicate the percentage of CPU utilization during the matrix multiplication.

A significant observation emerged during our implementation - the matrix of size 5000 required an extensive processing duration of 3 hours. This considerable time frame underscores the critical necessity of enhancing our matrix multiplication algorithm, facilitating it to manage larger inputs with improved efficiency. The low value of insn per cycle suggests that there might be potential bottlenecks or inefficiencies in the code or algorithm design, which can be targeted for optimization to improve the overall performance. Furthermore, the matrix multiplication process only utilized one of the available CPUs, leaving the other three CPUs largely unused. This emphasizes the importance of optimizing the code to take advantage of all available CPU resources. By distributing the workload across multiple CPUs, we can achieve better parallelism and significantly improve overall performance.

4.1.3 Optimization 1: Multithreading Results

Multithreading is a technique where multiple threads of a single process are executed concurrently. For the task of matrix multiplication, which involves repetitive and independent calculations, multithreading can be a significant optimization. It allows the workload to be distributed across multiple cores of a processor, thereby performing multiple computations simultaneously. The primary difference between the multi-threaded version and the non-multithreaded version is the way tasks are executed. In a multithreaded environment, several computations occur in parallel, while in a non-multithreaded environment, they happen sequentially.

Table 4.5: Branches Results For Multithreading Code

Metrics	500x500	1000x1000	2000x2000	5000x5000	10000x10000
Br. Mispred.	516,079	4,483,754	36,751,410	581,108,299	4,637,042,956
Br. Pred.	124,105,040	1,211,550,208	9,963,413,783	153,380,084,235	1,229,055,697,288
Bus Access	6,175,845	34,798,524	438,727,043	9,408,763,919	100,012,224,929
Bus Cycles	875,830,885	9,683,824,248	79,659,094,943	1,379,041,324,818	11,170,211,508,792

Table 4.5 depicts the branches performance metrics for the multithreaded version of matrix multiplication. We see that the number of branch mispredictions (Br. Mispred.) and correctly predicted branches (Br. Pred.) both increase with the size of the matrix. Moreover, when observing the branch misprediction rates for sizes 500, 1000, 2000, and 5000, they exhibit incredibly low figures: 0.4%, 0.2%, 0.19%, and 0.18%, respectively. These values are essentially nearing zero. There is a noticeable 29.25%, 45.37%, 67.80%, and 50.85% decrease from the non-optimized code. The implications of such low branch misprediction rates are profound. First and foremost, it indicates an efficient prediction algorithm within the processor, demonstrating that the system can predict the

direction of a branch instruction with remarkable accuracy. This significantly optimizes the overall performance of the computational tasks, as less time is wasted on rectifying incorrect predictions.

Table 4.6: TLB and Page Faults Results for Multithreading Code

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
l1d_tlb_refill	65,213,608	521,661,401	3,302,334,644	48,957,517,988	377,674,687,235
l1i_tlb_refill	9,367	35,759	686,664	6,464,982	48,376,064
dTLB-load-misses	65,213,608	521,661,401	8,028,255,825	48,936,918,643	377,535,775,617
dTLB-store-misses	13,182	55,254	736,701	9,141,882	65,158,812
page-faults	855	3,061	13,779	92,878	380,952

With reference to the page faults, the multithreaded version displayed a varying pattern across different matrix sizes. Particularly, the 500x500 matrix encountered 732 page faults, while the 1000x1000 and 2000x2000 matrices resulted in 2,929 and 11,719 page faults, respectively (similar to the non-optimized code). These figures fall within expected norms and can primarily be attributed to the standard initialization behavior of the matrices as discussed in the previous section. However, larger matrix sizes of 5000x5000 and 10000x10000 demonstrated a comparatively higher rate of page faults, with 73,242 and 292,968 page faults respectively. This escalation for larger matrix sizes can potentially be attributed to the fact that these matrices are likely to exceed the system's available memory, triggering a higher number of page faults as the operating system is forced to swap pages in and out of the physical memory. This discrepancy between the observed and expected number of page faults for larger matrices suggests an opportunity for further optimization. Optimizing memory usage, particularly for larger data structures, could help in reducing the number of page faults and TLB refills, thereby improving overall system performance.

Table 4.7 depicts the cache results for the multithreaded version. The cache miss rates for different matrix sizes show a dramatic improvement, each clocking in at barely above zero: 0.06% for 500x500, 0.06% for 1000x1000, 0.06% for 2000x2000, 0.069% for 5000x5000, and slightly higher 0.08% for 10000x10000. The cache miss rates in the optimized multithreaded code exhibit a substantial reduction compared to the non-optimized version, with percentage decreases of approximately 97.47% for 500x500, 98.69% for 1000x1000, 98.75% for 2000x2000, and 98.20% for 5000x5000 matrices. These rates, being strikingly close to zero, suggest that the vast majority of

Table 4.7: Caches Results for Multithreading Code

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
l1d_cache	2,487,058,381	24,241,050,100	202,619,151,935	3,082,262,865,963	24,648,325,161,786
l1d_cache_refill	1,317,576	17,045,345	124,942,062	2,124,828,090	19,858,261,052
l1i_cache	1,322,279,239	12,466,307,029	102,524,116,536	1,574,990,870,274	12,554,531,452,155
l1i_cache_refill	104,416	1,526,415	16,194,058	344,349,358	2,415,738,380
l2d_cache	4,072,120	91,956,642	486,864,759	8,485,776,712	182,444,910,727
mem_access	2,700,651,425	24,666,349,126	197,364,735,910	3,077,145,119,305	24,567,641,553,267
L1-icache-loads	1,578,909,022	12,571,994,711	100,420,613,942	1,564,547,582,031	12,544,768,828,261
L1-icache-load-misses	213,035	997,484	9,323,938	344,365,050	2,423,283,196
cache-references	3,073,592,271	24,624,780,201	196,802,199,828	3,061,899,246,264	24,594,219,183,946
cache-misses	1,707,814	15,603,809	113,886,429	2,107,550,303	19,812,328,987
L1-dcache-loads	3,087,631,580	24,622,007,048	196,840,543,125	3,062,443,561,383	24,607,936,772,447
L1-dcache-load-misses	1,729,691	15,614,858	113,904,026	2,108,878,705	19,823,786,667

data accesses hit the L1 cache successfully, thereby reducing the need to access slower memory tiers. A low cache miss rate, as in this instance, indicates that the majority of the data needed by the workload is available in the L1 cache. This leads to less dependency on slower memory layers like the main memory, resulting in a boost in overall system performance.

Table 4.8: Instructions and Time Results for Multithreading Code

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
CPUs utilized	3.870	3.947	3.958	3.973	3.977
instructions	4,233,361,417	45,271,902,959	373,976,156,024	5,735,441,124,907	45,949,356,479,680
insn per cycle	1.60	1.56	1.57	1.39	1.38
time elapsed (seconds)	0.52	4.38	34.04	816.98	8087.02

Upon transitioning to the multithreaded version, an improvement in computational performance was clearly observed. In particular, the execution time reduced drastically across all matrix sizes when compared to the non-optimized version. For instance, the time elapsed for the optimized code demonstrates substantial percentage decreases in execution time compared to the non-optimized code: around 82.64% for 500x500, 87.79% for 1000x1000, 87.58% for 2000x2000, and a remarkable 98.53% for 5000x5000. Notably, the multithreaded version was able to handle a 10000x10000 matrix within approximately 8087 seconds (2.25 hours), a task that was previously unfeasible and led to a nearly two-day computation before it was forcibly terminated.

In addition, the instructions per cycle for the optimized code exhibit significant percentage increases compared to the non-optimized code: approximately 42.86% for

500x500, 119.72% for 1000x1000, 111.35% for 2000x2000, and a remarkable 414.81% for 5000x5000.

In terms of CPU utilization, the multithreaded approach effectively harnessed almost all of the available processing resources, leveraging nearly four CPUs as indicated by the 'CPUs utilized' metric compared to the non-optimized version which only used one CPU. The high degree of CPU utilization is advantageous as it denotes effective parallelism. By distributing the workload across multiple CPUs, we were able to achieve enhanced parallelism, leading to a marked improvement in overall performance.

4.1.4 Optimization 2: Memory Optimization

The second phase of the optimization process focused primarily on memory management, with an additional emphasis on enhancing the already multithreaded version of the matrix multiplication algorithm. The objective here was to further boost the efficiency of the algorithm by improving its interaction with the computer's memory architecture, in particular with the Translation Lookaside Buffer (TLB) and page faults.

The memory optimization phase was subdivided into two primary strategies. The first strategy involved modifying the matrix multiplication method to a row-by-row approach within each block, as opposed to the initial row-by-column method. This modification retained the block-based matrix multiplication design, capitalizing on the principle of spatial locality - the tendency of processing units to access data that are close together within a short period of time. By accessing memory in a more contiguous manner, the algorithm optimized TLB utilization, thus potentially reducing the TLB miss rate. Accessing memory in a more contiguous manner ensures that the same pages are accessed repeatedly within a short time period, making it more likely that the necessary page-table mappings will be stored in the TLB, reducing TLB misses.

In the second part, the matrix initialization process was revamped. Traditionally, the matrix was initialized within the code itself, contributing to the overhead of the computation process. To address this, an alternative approach was adopted where the initialization was decoupled from the computation. Specifically, the matrices were saved in a file using a separate code, and then, during the matrix multiplication computation, these files were retrieved. This approach reduces the computational burden during the multiplication process, thereby reducing overall execution time.

It's also worth noting that the performance metrics such as branches, caches, instruction per cycle, and CPUs utilized were already performing satisfactorily in the multithreaded version. Therefore, the optimization focused on elements that needed improvement - TLB refills and page faults. However, a post-optimization assessment confirmed that these metrics maintained their performance levels, validating the effectiveness of the memory optimization changes.

Table 4.9: TLB and Page Faults Results for Memory Optimization

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
l1d_tlb_refill	42,479,207	499,129,221	2,103,540,755	47,114,534,615	365,470,923,777
l1i_tlb_refill	7,527	29,284	267,555	5,869,394	33,862,985
dTLB-load-misses	42,560,532	455,569,728	2,065,312,197	45,127,599,382	325,456,836,281
dTLB-store-misses	7,318	85,055	317,779	7,654,821	61,285,556
Page faults	865	3,071	13,791	92,890	380,964
time elapsed (seconds)	2.10	4.91	34.82	985.35	8072.05

Table 4.9 provides an overview of tlb and page faults metrics related to memory management and execution time for matrices of different sizes, under the memory-optimized algorithm.

As seen in the table, the number of both L1 data and instruction TLB refills has considerably decreased with memory optimization. For 500x500, the decrease is approximately 53.35%; for 1000x1000, it's about 45.43%; for 2000x2000, approximately 57.25%; for 5000x5000, it's about 43.12%; and for 10000x10000, it's around 35.22%. This indicates a more effective utilization of the TLB, which is consistent with the optimization strategy of reordering the matrix multiplication computation to be row by row within each block. The Data TLB load and store misses show how often the processor couldn't find the required data in the TLB, leading to slower memory access. Similar to the TLB refills, these metrics also decreased significantly, indicating a more efficient memory access pattern due to the optimization.

While the memory optimization was successful in reducing TLB refills and misses, it seems to have had no significant impact on the page faults, as their numbers remain similar to the previous version (variations ranging from a decrease of 1.38% for 500x500 to 1.33% for 10000x10000). This might be due to the initialization approach where matrices are stored and retrieved from a file, which could introduce additional page faults. In addition, the elapsed time shows the overall execution time for each operation. The values in the table suggest that the overall performance has remained stable, even with memory optimization. This can be attributed to the fact that while the TLB refills and misses have decreased, the number of page faults remained unchanged.

Hence, even though the memory access pattern improved, the page faults that occurred during the initialization process might have offset the gains from the TLB utilization improvement. The unchanged number of page faults and stable overall performance suggests further optimizations could be explored.

4.1.5 Optimization 3: Transparent huge tables

The third optimization that was explored in this work was the use of Transparent Huge Pages (THP). THP is a mechanism in modern Linux kernels that enables the operating system to use huge pages in a way that is transparent to applications [5]. A typical virtual memory system in Linux uses a page size of 4KB. However, with THP, it can use larger page sizes, such as 2MB, which can help reduce the number of page faults and TLB refills.

To understand how this works, it's essential to discuss the concepts of a page and a Translation Lookaside Buffer (TLB). The virtual memory of an application is divided into pages, which are chunks of memory that are loaded into physical memory as needed. When an application needs to access memory, the CPU must translate the virtual memory address into a physical memory address, a process known as a page table lookup. However, this operation is time-consuming, so the CPU keeps a cache of these translations in a structure called the TLB.

The TLB, however, has a limited size, and when it can't find a needed translation (a situation known as a TLB miss), it has to refill it from the page table, which is a slow operation. Similarly, a page fault occurs when an application tries to access a memory page that is not currently in physical memory, causing the operating system to load that page from disk, which is another slow operation.

Using THP can help reduce these costly operations. By increasing the page size from 4KB to 2MB, each entry in the TLB can cover a larger range of memory addresses, reducing the number of entries needed and thus possibly reducing the likelihood of a TLB miss. Similarly, fewer but larger pages reduce the number of page faults, as more data can be loaded into memory with each page.

Table 4.10 presents an in-depth view of the cache behavior when the Transparent Huge Pages (THP) optimization was applied to various matrix sizes. Generally, it indicates a considerable improvement in cache utilization compared to the typical cache performance without THP and surprisingly defies the expectation of potential cache degradation that can sometimes accompany the use of larger page sizes in THP.

Table 4.10: Cache Results for Transparent Huge Tables

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
l1d_cache	2,734,366,718	23,423,758,185	195,580,794,888	3,081,309,213,086	24,634,035,669,075
l1d_cache_refill	1,646,820	16,022,927	117,700,347	2,227,639,270	17,224,073,848
l2d_cache	4,829,470	89,354,148	473,540,380	7,535,225,861	96,808,379,843
l2d_cache_refill	261,067	7,724,434	109,720,053	2,138,804,855	20,251,058,714
mem_access	2,939,424,066	23,814,296,224	195,803,053,319	3,076,955,964,995	24,565,854,606,945
L1-icache-loads	1,180,652,449	11,852,701,681	99,771,152,689	1,575,837,515,058	12,574,894,706,974
L1-icache-load-misses	71,436	955,854	9,172,183	179,112,726	2,020,619,098
cache-references	2,352,511,757	23,286,023,221	195,656,949,099	3,092,084,415,399	24,627,921,586,114
cache-misses	1,313,559	13,850,626	113,740,199	2,237,374,584	17,201,498,145
L1-dcache-loads	2,388,205,387	23,315,713,469	195,668,747,006	3,091,830,915,153	24,636,798,709,228
L1-dcache-load-misses	1,289,008	13,814,002	113,934,311	2,236,861,552	17,212,219,340
L1-dcache-stores	212,610,047	2,038,381,897	17,090,192,906	270,212,205,758	2,159,175,359,329
L1-dcache-store-misses	27,165	270,712	2,394,739	48,496,199	473,958,947

It's particularly interesting to see that the miss rate is remarkably low for all the matrix sizes considered. The cache miss rates, calculated as the proportion of cache accesses that resulted in a miss, were only 0.05% for the 500x500 matrix, 0.06% for both the 1000x1000 and 2000x2000 matrices, and 0.07% for both the 5000x5000 and 10000x10000 matrices. Furthermore, it exhibits a remarkable reduction compared to the unoptimized code, with percentage decreases of approximately 97.89% for the 500x500 matrix, 98.75% for both the 1000x1000 and 2000x2000 matrices, and 98.54% for both the 5000x5000 and 10000x10000 matrices. This result indicates that the majority of data was successfully retrieved from the cache upon the first request, thereby enhancing overall system performance.

One reason for this positive outcome could be the increased spatial locality afforded by the larger page sizes. Spatial locality refers to the principle that when a data location is accessed, nearby data is likely to be accessed in the near future. With larger page sizes, more of this 'nearby' data is loaded into memory at once, making it more likely that future data accesses can be served directly from the cache.

Table 4.11 shows that the utilization of THP significantly reduces the number of TLB refills for all matrix sizes, as evidenced by the l1d_tlb_refill, l1i_tlb_refill, dTLB-load-misses, and dTLB-store-misses metrics. For instance, the TLB refill values for the transparent huge tables show decreases of about 25.05% for 500x500, 13.37% for 1000x1000, 15.16% for 2000x2000, 79.47% for 5000x5000, and 86.52% for 10000x10000 compared to the multithreading code. The reason behind this significant decrease in TLB refills lies in the essence of THP. As THP increases the page

Table 4.11: TLB and Page Faults Results for Transparent Huge Tables

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
l1d_tlb_refill	49,058,369	513,755,171	3,135,827,298	10,321,582,081	53,998,649,402
l1i_tlb_refill	2,066	20,270	201,049	3,828,288	71,548,179
dTLB-load-misses	51,477,439	514,306,713	3,135,701,751	10,328,287,038	54,058,678,710
dTLB-store-misses	2,914	30,426	269,387	5,228,591	56,903,688
page-faults	855	3,062	13,782	61,164	231,092

size from the standard 4KB to 2MB, the same amount of data can be referenced with fewer TLB entries. Thus, the probability of experiencing a TLB miss decreases, which directly leads to fewer TLB refills.

In addition, the table indicates a substantial reduction in page faults for each matrix size, bringing the values back down to those calculated in the initial analysis (the first section). In addition, the page-fault values for the transparent huge tables indicate a decrease of 34.14% for 5000x5000, and 39.50% for 10000x10000 in comparison to the multithreading code. A page fault occurs when a program tries to access a part of memory that is not currently available in the RAM, causing the system to read it from the disk. The significant reduction in page faults implies fewer expensive disk accesses, leading to faster memory accesses and overall improved system performance.

Table 4.12: Instructions and Time Results for Transparent Huge Pages

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
Instructions	4,264,540,338	43,350,643,785	365,693,811,039	5,771,246,275,362	46,043,137,099,191
Insn per Cycle	1.61	1.55	1.56	1.59	1.61
CPUs Utilized	3.847	3.949	3.937	3.977	3.968
Time Elapsed (seconds)	0.52	2.24	25.96	730.49	7248.18

Table 4.12 shows that the critical metric of Instructions Per Cycle (IPC) has increased with the implementation of THP for the sizes 5000 and 10000 (as it was good for all the other values before).

The percentage increase between this version and the multithreaded version is approximately 12.26% for the 5000x5000 matrix and 30.62% for the 10000x10000 matrix. IPC, which measures the number of instructions executed for each clock cycle, is a useful performance indicator as a higher IPC often corresponds to better system performance. This increase suggests that with THP, the system is able to execute more instructions within the same amount of time, thereby improving its efficiency. Indeed, the increase in IPC could be attributed to the fewer TLB misses and page faults observed with THP. As a result, less time is wasted on memory access latencies, allowing the system to spend more time executing instructions, which consequently leads to a higher IPC. Furthermore, the table also indicates a decrease in the ”Time Elapsed” for all the matrix sizes when THP is implemented. The time shows a decrease of approximately 49.11% for the 1000x1000 matrix, 24.73% for the 2000x2000 matrix, 25.23% for the 5000x5000 matrix, and 10.65% for the 10000x10000 matrix than the multithreaded version. This metric signifies the total runtime of the computation. A reduction in this metric is favorable as it suggests that the computations are completed faster, further supporting the increased system performance evidenced by the higher IPC.

4.1.6 Optimization 4: Compiler Optimization

While it’s widely acknowledged that compiler optimizations can lead to significant performance improvements, I believe it’s essential to underscore the need for developers to embrace this as a default practice. Instead of considering it an afterthought or a secondary step in the software development process, compiler optimization should be integrated into the standard workflow. This proactive approach ensures that the code is not only functional but also optimized for efficiency from the outset. It introduces more aggressive optimizations that may significantly improve the runtime performance. These can include inline function expansion, prediction of branching, loop unrolling, and vectorization, among others [3].

Table 4.13: TLB and Page faults Results for Compiler Optimization

Metric	1000x1000	2000x2000	5000x5000	10000x10000
l1d_tlb_refill	9,926,270	101,267,975	1,725,164,139	5,853,731,382
l1i_tlb_refill	2,885	60,255	941,799	6,465,547
dTLB-load-misses	10,130,421	101,800,674	1,725,598,584	5,855,970,124
dTLB-store-misses	17,473	169,265	1,519,557	10,591,297
Page-faults	3,061	13,781	92,917	278,323

Table 4.13 presented shows the effects of compiler optimization, particularly the -O3 optimization flag, on metrics related to Translation Lookaside Buffer (TLB) refills and page faults. It’s noteworthy that the figures for the 500x500 size matrix are not

included in the table because the execution speed was so swift that the perf tool couldn't record the metrics.

Let's first focus on the remarkable decrease in Translation Lookaside Buffer (TLB) refills, marked by l1d_tlb_refill and l1i_tlb_refill for both data and instruction caches. For the matrix sizes of 1000x1000, 2000x2000, 5000x5000, and 10000x10000, the tlb refill metric exhibited approximate percentage decreases of 98.10%, 96.93%, 96.47%, and 98.45% respectively, when comparing it to the multithreaded version. In addition, when comparing it to the transparent huge table version, it shows a show a percentage decrease of approximately 98.07%, 96.77%, 83.28%, and 89.17% for the matrix sizes of 1000x1000, 2000x2000, 5000x5000, and 10000x10000.

This decline is highly beneficial as TLB refills represent a time-consuming operation since they involve accessing the page tables located in the main memory or even higher up in the memory hierarchy. The -O3 optimization flag implements several optimization techniques, which improve the memory access pattern and, as a result, reduce TLB refills. Furthermore, these optimization strategies are capable of enhancing the locality of reference, resulting in more frequent access to the same set of pages and hence fewer TLB misses.

However, the table also reveals a slight downside to the optimization. Specifically, there is a minor increase in page faults (Page-faults) for the 5000x5000 configuration, which could imply an increased need for fetching pages from the disk. This scenario might have resulted from aggressive optimizations, leading to an altered memory access pattern that might not align well with the operating system's page replacement algorithm. Despite this, the page faults for the other sizes remain consistent with the results from the previous optimization, indicating that this anomaly might be confined to specific cases and does not diminish the overall effectiveness of the -O3 optimization.

Table 4.14: compiler

Metric	500x500	1000x1000	2000x2000	5000x5000	10000x10000
CPUs utilized	3.177	3.767	3.927	3.959	3.974
Instructions	-	5,649,300,250	66,123,395,653	1,086,842,924,230	8,710,694,278,456
Insn per cycle	-	2.11	1.90	1.81	1.70
Time elapsed (seconds)	0.094	0.61	6.17 sec	110.38	1068.60

Table 4.14 presented demonstrates the remarkable effect of the -O3 compiler optimization on the execution time and the efficiency of instructions executed per cycle. As

with previous tables, the metrics for the 500x500 size matrix are not included, given that the operation was completed too swiftly for perf to record accurate metrics.

One of the most striking results in this table is the substantial decrease in the total run time of the program (Time elapsed). This decrease is most notable for the largest matrix size, 10000x10000, which was completed in only 1068.6 seconds, or approximately 17.8 minutes. For the shared matrix sizes, it showed a percentage decrease of approximately 81.92%, 86.07%, 81.88%, 86.49%, and 86.79% compared to multithreading code. Furthermore, it showed a percentage decrease of approximately 81.92%, 72.77%, 76.24%, 84.88%, and 85.27% respectively for matrix sizes 500x500, 1000x1000, 2000x2000, 5000x5000, and 10000x10000, when comparing it to Transparent Huge Pages. Comparing it to the unoptimized version that was done in the first section, it showed a percentage decrease of approximately 96.736%, 98.311%, 97.754%, and 99.073%. This outcome is remarkable considering the computational complexity and resource demand associated with handling such large matrix sizes. This significant speedup is a testament to the effectiveness of the -O3 optimization.

4.2 Fibonacci

The Fibonacci sequence is an intriguing mathematical concept with numerous real-world applications, making it a compelling case study for optimization on the Raspberry Pi platform. Named after Italian mathematician Leonardo of Pisa, known as Fibonacci, this sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence commences as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on.

4.2.1 Unoptimized Recursive Fibonacci Results

To commence this exploration, we started with an unoptimized Fibonacci sequence calculation algorithm. This initial version implements the Fibonacci sequence in a recursive manner. The recursive algorithm, while being a straightforward representation of the Fibonacci sequence's definition, is known for its inefficiency due to a large amount of repeated computation. Each call to calculate Fibonacci(n) results in two more recursive calls, leading to an exponential growth in computation as 'n' increases.

Starting our optimization journey with this unoptimized version allows us to fully appreciate the impact of different optimization techniques. It provides a clear baseline against which we can measure the improvements and gives us a broad scope for potential performance enhancements. In the subsequent sections, we will delve into the different optimization techniques applied to this Fibonacci sequence calculation algorithm, evaluate their effectiveness, and compare their results to our initial unoptimized version.

Table 4.15: Branches Results for Non-optimized Code

Metric	n=40	n=50	n=60
br_mis_pred	20,131,916	2,475,156,868	304,368,129,667
br_pred	1,903,974,952	237,582,877,425	29,223,379,550,135
bus_access	2,055,552	3,944,178	818,238,605
bus_cycles	1,625,618,438	203,404,136,957	25,302,054,753,254

Table 4.15 presents performance metrics for different values of ’n’ in the Fibonacci sequence calculation using the unoptimized recursive algorithm.

The data clearly indicate that the branch mispredictions rate sharply escalates with increasing ’n’, specifically from 0.01% for n=40 and n=50, and then increases to 10.4% for n=60. This substantial jump in the branch misprediction rate when calculating Fibonacci of 60 can be attributed to the inherent nature of the recursive algorithm. To explain this further, consider the branching nature of recursion. The algorithm makes a decision every time it encounters a recursive call: it needs to ”predict” whether the recursion will branch off (i.e., make more recursive calls) or hit a base case and start returning. In the Fibonacci sequence’s case, due to the substantial amount of repeated computation in the recursive algorithm, it becomes increasingly challenging for the branch predictor to accurately predict the branches, especially for large values of ’n’.

This increase in the branch misprediction rate is highly detrimental to performance. Every mispredicted branch causes a pipeline stall or delays as the processor needs to discard or ”flush” the incorrectly predicted instructions and then fetch and decode the correct ones. This not only wastes time and computational resources but also disrupts the smooth flow of instructions through the processor’s pipeline, degrading overall performance. Therefore, the escalation of branch mispredictions from n=40 to n=60, as seen in the table, indicates a steep deterioration in the efficiency and performance of the Fibonacci calculation.

Table 4.16 summarizes the cache behavior for the unoptimized Fibonacci code for three different sizes of ’n’. One of the most striking aspects of the table is the extremely low cache miss rates, ranging from 0.01% for n=40 and n=50 to slightly increased

Table 4.16: Caches results for Non-optimized Code

Metric	n=40	n=50	n=60
l1d_cache	4,719,542,002	577,399,676,740	71,011,552,280,387
l1d_cache_refill	200,259	22,465,808	2,720,683,077
l2d_cache	1,006,732	86,582,111	10,354,352,438
l2d_cache_refill	79,383	305,583	39,319,913
mem_access	4,703,347,568	577,362,044,609	71,013,840,137,689
L1-icache-loads	3,975,204,277	486,548,925,738	59,836,129,339,467
L1-icache-load-misses	317,498	35,621,189	4,316,918,791
cache-references	4,710,321,622	577,350,323,915	71,008,518,178,585
cache-misses	186,402	22,014,099	2,728,781,183
L1-dcache-loads	4,708,880,525	577,458,134,196	71,011,437,477,305
L1-dcache-load-misses	189,084	22,531,595	2,724,560,128
L1-dcache-stores	2,228,382,856	272,977,509,591	33,572,976,505,147
L1-dcache-store-misses	23,664	2,618,564	312,922,104

0.02% for n=60.

These results indicate very efficient cache utilization. The low cache miss rate, particularly for the L1 data cache (L1-dcache), can be attributed to the inherent simplicity of the Fibonacci function. The Fibonacci function, especially in its unoptimized recursive version, is quite simple. It includes a small number of instructions and does not demand extensive data manipulation or complex computations that would require large data structures. As a result, the program's footprint is relatively small and can comfortably fit within the L1 cache.

In addition, the Fibonacci function's recursive nature may further contribute to efficient cache utilization. Recursive calls entail a repeated set of instructions, which benefits from the temporal locality - an important principle of cache memory where if a location is referenced, it is likely to be referenced again shortly. Hence, once the instructions for the Fibonacci function are fetched into the L1 instruction cache (L1-icache), subsequent recursive calls can effectively retrieve these instructions from the cache rather than fetching them from slower memory regions.

Table 4.17 provides essential information about the execution time and instruction counts for the non-optimized Fibonacci code. It clearly illustrates the impact of increasing Fibonacci sequence size 'n' on the program's execution time.

Table 4.17: Instructions and Time Results for Non-optimized Code

Metric	n=40	n=50	n=60
CPUs utilized	0.996	0.998	0.998
instructions	8,072,132,995	989,867,092,695	121,733,764,587,628
insn per cycle	1.59	1.60	1.60
time elapsed (seconds)	2.83	347.23	42792.48

As the size 'n' increases from 40 to 60, the task-clock metric, representing the CPU time in milliseconds, grows from a manageable 2,826.90 msec for n=40 to a staggering 42,696,757.04 msec (11.6 hours) for n=60. This dramatic increase in computation time underlines the exponential complexity of the recursive Fibonacci algorithm, as each increase by 10 in the sequence size leads to an approximately 100-fold increase in execution time. This substantial rise in computation time naturally makes exploring sequence sizes greater than 60 impractical and, indeed, unproductive. The experiment clearly shows that without optimization, computing the Fibonacci sequence recursively for larger values of 'n' would result in unacceptably long computation times.

On the other hand, the metrics relating to CPU utilization and instructions per cycle (IPC) show consistent results across different sequence sizes. The CPU utilization is consistently close to 1, which indicates that the Fibonacci function is able to make effective use of the CPU during its execution. However, the constant CPU utilization across different sequence sizes also shows that the recursive Fibonacci algorithm does not support multithreading, and therefore, cannot take advantage of multiple CPU cores to speed up the computation. This is a characteristic inherent to the Fibonacci algorithm, as it relies on the results of previous computations, thus rendering concurrent computation of separate sequence values impossible.

The instructions per cycle (IPC) values are also consistent, hovering around 1.60 for all sequence sizes. This IPC value is decent, indicating that on average, the CPU is able to execute 1.60 instructions per clock cycle. However, the consistent IPC across different sequence sizes once again highlights the non-parallel nature of the Fibonacci algorithm, as it doesn't allow for an increase in instructions per cycle with larger sequence sizes.

Overall, the table underlines the necessity for optimizing the Fibonacci function for larger sequence sizes. Despite efficient CPU utilization and a good IPC, the exponential increase in computation time for larger 'n' values presents a significant obstacle that needs to be overcome.

4.2.2 Optimization 1: Memoization Results

In a bid to improve the performance of the Fibonacci function, especially for larger sequence sizes, a second version of the Fibonacci algorithm was introduced leveraging a technique known as ”memoization”. Memoization is a specific type of cache optimization used to speed up programs by storing the results of expensive function calls and reusing them when the same inputs are passed again [1]. This concept is based on the premise that it is often faster to retrieve a known result from a lookup table or a similar data structure than to perform complex, time-consuming calculations every time a function is called. Essentially, memoization transforms a function into a lookup table by caching previously computed results.

In the context of the Fibonacci function, memoization is particularly beneficial. The recursive Fibonacci algorithm without memoization is known to exhibit overlapping subproblems, i.e., it repeatedly computes the same sub-sequences within the Fibonacci sequence. By storing the result of each Fibonacci calculation within a cache or memory, we avoid these redundant calculations. Once the function calculates the Fibonacci number for a certain value of ’n’, it stores this number in memory. If the function is later called again with the same ’n’, it will not re-calculate the result but instead fetch it directly from the memory. By using memoization, the time complexity of the Fibonacci function can be reduced from exponential to linear, as each Fibonacci number is now calculated only once.

Table 4.18: Branches Results for Memoization Code

Metric	n=100,000	n=500,000	n=1,000,000	n=5,000,000	n=100,000,000
br_mis_pred	1,350,229	7,458,693	15,137,232	88,856,257	1,858,051,890
br_pred	65,667,576	591,229,717	1,280,684,424	6,455,204,189	130,798,838,298
bus_access	3,591,127	5,966,656	10,559,531	86,427,914	1,966,257,017
bus_cycles	143,214,755	783,680,461	1,633,945,850	8,175,187,001	167,361,267,008

It is noticeable from Table 4.18 that the Fibonacci function’s performance has considerably improved, especially with regard to the branch misprediction rate, which has decreased to 0.02% for n=100,000, and 0.01% for all larger Fibonacci sequences signifying a 99.9% reduction when compared to the non-optimized code. This can be explained by the inherent characteristics of memoization. In essence, memoization reduces the number of recursive function calls. For a function like Fibonacci, which involves a considerable amount of repetitive and overlapping computations, memoization avoids redundancy, thereby reducing the total number of function calls. Since each function call involves a decision (or branch) on whether to calculate or retrieve the value from the memory, fewer function calls will consequently lead to fewer branches and thereby fewer opportunities for branch mispredictions.

What makes this particularly interesting is the consistency of the low branch misprediction rate across different sequence sizes. The rates are as low as 0.01 even for considerably large Fibonacci sequence sizes like n=1,000,000, n=5,000,000, and n=100,000,000. This indicates that memoization has not just reduced the misprediction rate, but also made it more consistent irrespective of the size of the sequence, a marked improvement over the non-optimized version.

Table 4.19: Caches Results for Memoization Code

Metric	n=100,000	n=500,000	n=1,000,000	n=5,000,000	n=100,000,000
l1d_cache	358,720,562	1,503,682,645	3,033,051,220	15,145,674,160	305,130,443,232
l1d_cache_refill	484,471	5,595,396	6,243,769	29,751,539	915,329,995
l2d_cache	5,382,676	30,743,248	54,756,226	288,116,599	6,392,628,480
l2d_cache_refill	354,122	1,999,532	6,737,231	16,050,901	274,843,119
mem_access	352,814,258	1,514,797,978	3,445,590,030	15,429,511,879	301,069,432,705
L1-icache-loads	-	1,395,013,780	2,675,364,509	13,539,178,826	272,560,639,926
L1-icache-load-misses	-	13,915,565	27,754,965	159,027,781	3,299,520,730
cache-references	-	1,539,589,542	2,965,399,019	14,954,081,256	298,836,883,005
cache-misses	-	5,956,499	5,677,323	30,374,775	897,672,279
L1-dcache-loads	-	1,530,570,560	2,978,991,199	14,695,359,625	298,837,040,905
L1-dcache-load-misses	-	5,968,208	5,776,350	31,762,271	899,888,779
L1-dcache-stores	-	718,575,292	1,383,706,720	6,854,694,631	139,514,972,891
L1-dcache-store-misses	-	1,345,340	1,716,412	9,625,204	210,118,124

As we can see from Table 4.19, the L1 data cache miss rate is relatively low: 0.39% for n=500,000, 0.22% for n=1,000,000, 0.30% for n=5,000,000, and 0.19% for n=100,000,000. This is a good indication as it suggests that most of the required data were present in the L1 data cache during the execution of the program, which generally leads to improved performance.

The reason for the low cache miss rates in this scenario is the memoization optimization technique. By storing and reusing previously calculated results, memoization reduces the total number of computations, which subsequently reduces the amount of data that needs to be frequently accessed. Therefore, the required data tends to fit in the relatively small L1 cache, leading to lower cache miss rates. This not only speeds up the Fibonacci function execution but also makes it more efficient in terms of memory usage.

It's worth mentioning that for n=100,000, certain metrics could not be recorded due to the quick execution of the program. Perf, the tool used for performance analysis, might not have been able to capture the cache metrics due to the rapid execution. This in itself is a good indicator of how well memoization has improved the performance of the Fibonacci function, making it significantly faster.

Table 4.20: Instructions and Time Results for Memoization Code

Metric	**n=100,000**	**n=500,000**	**n=1,000,000**	**n=5,000,000**	**n=100,000,000**
CPUs utilized	0.433	0.897	0.909	0.813	0.947
instructions	821,910,895	2,882,387,450	5,512,085,746	28,553,979,939	562,498,114,256
insn per cycle	1.44	1.13	1.14	1.15	1.13
time elapsed (seconds)	0.74	6.39	20.14	84.14	840.98

The most notable outcome from this optimization is a significant reduction in the execution time as seen in Table 4.20. The optimized version calculates Fibonacci for n=100,000,000 in approximately 840 seconds or less than 14 minutes. This stark improvement demonstrates the efficiency of memoization, particularly in recursive algorithms where computations often repeat.

However, one metric that seems to have regressed is the instructions per cycle (IPC). The program is executing about 1.13 to 1.44 instructions per cycle, depending on the input size. In general, a higher IPC indicates that the processor is effectively utilizing its instruction-level parallelism, which typically leads to better performance. The lower IPC in this case could be attributed to a few factors. One possible explanation is the overhead introduced by the memoization process. While the reuse of precomputed results significantly reduces the computational workload, it involves additional operations, such as checking the cache for stored results and updating the cache with new results. These operations can increase the total instruction count, which might explain the lower IPC. While it's clear that memoization has significantly improved the execution time of the Fibonacci function, the decrease in IPC suggests there might still be room for further optimization.

4.2.3 Optimization 2: Dynamic programming and Compiler Optimization Results

The final stage of optimization was executed in two main parts, bringing together both memoization and dynamic programming. While recursion has its merits, it can lead to substantial overhead in certain algorithms like Fibonacci, where computations are often repeated. Dynamic programming mitigates this by avoiding recursion altogether. Instead of calling the function recursively for previous terms, the dynamic programming-based approach calculates the Fibonacci sequence iteratively. By storing intermediate results in an array or vector, this method ensures that each Fibonacci term is calculated only once, leveraging previously computed results.

Along with the mentioned changes to the code, the function was compiled using the -O3 optimization flag. As detailed in a previous section, this compilation flag enables the compiler to apply a series of performance-enhancing transformations. In this stage of optimization, the primary focus was directed toward addressing the instructions per cycle and the execution time, both of which presented challenges in the previous memoization stage. Since these metrics were identified as areas requiring improvement, the subsequent optimization efforts were specifically tailored to enhance them. By transitioning to an iterative approach and utilizing advanced compilation techniques, we concentrated on refining these critical performance indicators, while other metrics, already displaying satisfactory results, were monitored for consistency but were not the central concern.

It is also worth noting that the cache results from the final optimization stage have not been included in this section. The reason behind this omission is the remarkable consistency between these results and those observed in the previous memoization-based optimization. The cache behavior remained largely unchanged, exhibiting very good performance characteristics similar to before. This consistency in the cache results further underscores the effectiveness of the memoization and dynamic programming combination in maintaining optimal cache utilization. Therefore, the detailed cache table was deemed unnecessary for this section, as it would only reiterate previously established findings. Similarly, the branch misprediction rates from this final optimization stage were not explicitly detailed, as they remained consistent with the previously observed rates in the memoization code.

Table 4.21 provides an insightful overview of the final optimization stage, illustrating significant improvements in critical areas. One of the most noteworthy enhance-

Table 4.21: Instructions and Time Results for Dynamic and Compiler Optimization

Metric	n=100,000	n=500,000	n=1,000,000	n=5,000,000	n=100,000,000
instructions	-	1,427,428,097	2,572,502,805	13,069,387,640	249,130,170,601
insn per cycle	-	1.51	1.68	1.71	1.79
CPUs utilized	0.094	0.091	0.078	0.085	0.166
time elapsed (seconds)	0.76	5.98	14.75	64.14	586.58

ments is observed in the instructions per cycle (IPC). In this optimization phase, the IPC values display an upward trend, reaching 1.79 for n=100,000,000. The dynamic and compiler optimization method exhibited IPC increases of approximately 33.63%, 47.37%, 48.70%, and 58.41% for Fibonacci sequence sizes n=500,000, n=1,000,000, n=5,000,000, and n=100,000,000, respectively, compared to the memoization method. This is an encouraging indicator of the optimization's efficacy, demonstrating a more efficient utilization of CPU cycles. The increase in IPC is a clear indication of better parallelization of instruction execution, thus leading to more optimal use of the processor's resources.

Equally significant is the marked reduction in the overall execution time. So, for the Fibonacci sequences of sizes n=500,000, n=1,000,000, n=5,000,000, n=100,000,000, the dynamic and compiler optimization method exhibited decreases in elapsed time by approximately 6.42%, 26.79%, 23.78%, and 30.23%, respectively, compared to memoization. By moving away from recursion and employing a combination of memoization and dynamic programming, along with compiler optimization techniques, the execution time for the given Fibonacci sequence calculations has been substantially reduced. For instance, the execution time for n=100,000,000 has been cut down to approximately 586.59 seconds, a tremendous improvement that underscores the effectiveness of the adopted strategies. The final optimization stage, characterized by dynamic programming and compiler adjustments, has successfully addressed previously identified challenges.

Chapter 5

Discussion

This study explored various facets of performance optimization for matrix multiplication, spanning from threading, memory optimization, and compiler optimization, to employing Transparent Huge Tables (THT). Each phase of optimization was methodically scrutinized, and relevant metrics were collected to quantify their impact.

The initial optimization technique deployed was multithreading, which capitalized on the intrinsic parallelism in matrix multiplication. Results confirmed that threading significantly curtailed execution times, especially for larger matrix dimensions. The profound acceleration attained by the multithreaded version underlined the vitality of harnessing hardware parallelism to optimize computationally intensive tasks, a concept aligned with Amdahl's Law.

Subsequently, the focus shifted to memory optimization, aiming to improve the algorithm's engagement with the computer's memory architecture, specifically the Translation Lookaside Buffer (TLB) and page faults. The strategic alterations in the matrix multiplication method and the decoupling of matrix initialization from computation formed the crux of this strategy. Despite the successful reduction in TLB refills and misses, the number of page faults remained unaffected, indicating potential areas for further optimization.

Transparent Huge Tables (THT) was another technique employed, which leveraged larger page sizes to optimize memory access patterns. The enabling of THT led to a decrease in TLB misses, substantiating the effectiveness of this approach in improving the algorithm's interaction with memory systems. The use of THT optimized memory utilization, bolstered processing speed, and revealed how this technique can become an integral part of optimizing similar high-performance computing tasks.

Lastly, compiler optimization was explored, specifically the -O3 optimization flag.

This measure proved significantly beneficial, leading to a substantial drop in cache misses and TLB refills. However, a slight uptick in page faults was observed for specific matrix sizes. Despite this, the overall execution time saw a significant drop, and the number of instructions executed per cycle displayed a remarkable increase.

The optimization of the Fibonacci function unfolded through a carefully planned, multi-stage process, demonstrating the intrinsic relationship between various optimization techniques and their cumulative effect on improving performance. Initially, the recursive version of the Fibonacci function was analyzed, revealing inherent inefficiencies such as overlapping subproblems. The introduction of memoization transformed the function by improving the time complexity from exponential to linear. It reduced not only the branch misprediction rate but also the L1 data cache miss rate. This stage provided a foundational understanding of the power of caching and avoiding redundant calculations in recursive algorithms.

Moving from recursion to memoization was a crucial step, but further optimization was achieved by employing dynamic programming. This eliminated recursion altogether and iteratively calculated the sequence, leveraging previously computed results. Combined with the -O3 optimization flag that allowed the compiler to perform more detailed optimizations, the function exhibited significant improvements in execution time and instructions per cycle (IPC). This stage emphasized the importance of a comprehensive approach that includes both algorithmic changes and compiler-level enhancements.

One of the key achievements in this process was the reduction in time complexity from exponential to linear. This success, coupled with consistent cache behavior and the significant increase in IPC, underscored the importance of targeted optimization. The marked reduction in execution time for large Fibonacci calculations, such as cutting down the time for n=100,000,000 to 586.59 seconds, demonstrated the effectiveness of combining memoization, dynamic programming, and compiler adjustments.

While the optimizations discussed in the report have demonstrated significant performance enhancements, it's vital to recognize their context. These strategies were meticulously tailored to harness the unique characteristics of the Raspberry Pi (RPi) architecture. Even though one could think that similar optimizations could be applied to server-based code, potentially yielding more robust results than an array or cluster of RPis, the intention of this study was specific. It aimed to elucidate methods to maximize the performance of code on RPi devices. The Raspberry Pi has its distinctive set of challenges and advantages due to its hardware constraints and design philosophy. Hence, the specifics in this report are particularly suited for RPi environments.

Chapter 6

Conclusions

The comprehensive and multi-layered exploration of two distinct workloads, Fibonacci sequence calculation, and matrix multiplication, has revealed the profound impact of methodical optimization. By engaging with these workloads at a granular level and employing a combination of techniques ranging from memoization, dynamic programming, and compiler-level enhancements, remarkable improvements in efficiency, performance, and resource utilization were achieved.

Through the Fibonacci sequence, we uncovered how recursive inefficiencies could be overcome with memoization and further refined with dynamic programming. Meanwhile, the deep analysis of matrix multiplication served as a vivid illustration of the role of algorithm design in maximizing computational efficiency. Both these workloads were meticulously tailored to run efficiently on the Raspberry Pi, a platform that symbolizes accessible and low-cost computing.

The methodologies employed in these two applications are not isolated to these specific problems but represent a robust and versatile framework. The same approach, which involves identifying inefficiencies, iteratively applying optimization techniques, and carefully analyzing the results, can indeed be generalized to any application. This adaptability signifies a path towards greater optimization in a wide array of computational problems, enhancing the efficiency and potential of the devices and systems we rely on.

The future of this research and its application is tantalizingly promising. With two fully working and optimized workloads tailored for the Raspberry Pi, the next step involves broadening the horizon to more complex architectures. The objective now shifts to orchestrating these workloads across an array of Raspberry Pi devices, endeavoring to create a system that can compete with traditional servers. This paradigm could herald a new era in distributed computing, leveraging the cost-effective and energy-efficient nature of Raspberry Pi to deliver robust computational capabilities.

Ultimately, this research has illuminated not just the paths to optimization for specific problems but also the art and science of computational efficiency itself. By doing so, it has laid down a roadmap that can be followed for diverse applications, unlocking new potentials and guiding future explorations in both academia and industry. The convergence of detailed analysis, innovative techniques, and visionary goals offers an inspiring testament to what can be achieved through relentless inquiry, creativity, and commitment to excellence.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 14–25, New York, NY, USA, 2003. Association for Computing Machinery.

[2] Israel Tekahun Basha and Meron Istifanos. Performance evaluation of raspberry pi 3b as a web server : Evaluating the performance of raspberry pi 3b as a web server using nginx and apache2, 2020.

[3] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, 1993.

[4] Parikshit Dubey and Arva Kagdi. Run time analysis of matrix multiplication using raspberry pi cluster supercomputer. 05 2020.

[5] Johannes K Fichte, Norbert Manthey, Julian Stecklina, and André Schidler. Towards faster reasoners by using transparent huge pages. In *International Conference on Principles and Practice of Constraint Programming*, pages 304–322. Springer, 2020.

[6] Hirak Ghael. A review paper on raspberry pi and its applications. 01 2020.

[7] Scott Gilbertson. The raspberry pi 4 can't quite replace a pc, but it tries, Aug 2019.

[8] Wajdi Hajji and Fung Po Tso. Understanding the performance of low power raspberry pi cloud for big data. *Electronics*, 5(2), 2016.

[9] Gareth Halfacree and Eben Upton. *Raspberry Pi user guide*. John Wiley & Sons, 2012.

[10] Michael Kerrisk. perf(1). Linux man-pages, July 2021.

[11] Branislav Madoš, Ján Hurtuk, Eva Chovancová, Peter Fecil'ák, and Dávid Bajkó. Downsizing of web server design using raspberry pi 3 single board computer platform. In *2017 IEEE 14th International Scientific Conference on Informatics*, pages 238–242, 2017.

[12] Anand Nayyar and Vikram Puri. Raspberry pi-a small, powerful, cost effective and efficient form factor computer: A review. *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE)*, 5:720–737, 12 2015.

[13] Attila Orosz. The power user's guide to htop, October 2015.

[14] K.G. Orphanides. Raspberry pi 4 review: Finally ready to replace your desktop pc, Jun 2019.

[15] Raspberry Pi Foundation. Raspberry pi 4 product brief. https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-4-Product-Brief.pdf, 2020. Accessed on [Insert Date].