UNDERSTANDING THE CHALLENGES WITH USING HAWRDWARE PRE-FETCHERS FOR CPU-BASED MATRIX MULTIPLY UNITS

A Dissertation
Presented to
The Academic Faculty

By

Michael Goldstein

In Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science in the
College of Computing
School of Computer Science

Georgia Institute of Technology

August 2024

© Michael Goldstein 2024

UNDERSTANDING THE CHALLENGES WITH USING HAWRDWARE PRE-FETCHERS FOR CPU-BASED MATRIX MULTIPLY UNITS

Thesis committee:

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Callie Hao
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Computer Science
Georgia Institute of Technology

Date approved: August 23, 2024

What a wild life, and what a fresh kind of existence!

But, ah, the discomforts!

Henry Wadsworth Longfellow

For my mother, Sherri

ACKNOWLEDGMENTS

I would first like to thank my advisor Hyesoon Kim, without whom this work likely never would have begun. I would also like to express my gratitude to the other members of the committee for taking the time out of their busy lives to read through my work. Special thanks to Seonjin Na and Jaewon Lee, whose collective support have been invaluable throughout this project.

I would like to thank my family and the many friends and colleagues who have supported and encouraged me over the last few years, including Alexis Goldstein, Sherri Schwartz, Felix Spink, Michelle Spink, Louis Schwartz, Mikayla Schwartz, Rick Goldstein, Irene Berson, Sam Jijina, John Parrish, Pulkit Gupta, Camille Bossut, Brendon Im, Rohan Menon, Arshya Srinivas, Arina Moradinia, Connor Cummings, Carly Johnson, Divya Ventarapragada, Mitchell Stasko, Jonathan Nase, Kevin Adams, Jonathan Buchanan, Daniel Yang, Ethan Gamble, Ethan Fiber, Anurag Kar, and many others.

The author gratefully acknowledges the support for this work offered by Sandia National Laboratories under the Critical Skills Recruiting Program. Any views and conclusions contained herein are those of the author and do not necessarily represent the official positions, express or implied, of the funders.

TABLE OF CONTENTS

Acknowledgments v

List of Tables ix

List of Figures xi

List of Acronyms xii

Summary xiii

Chapter 1: Introduction and Background 1

1.1 CPUs in AI Workloads 1

1.1.1 SIMD Programming on CPUs 2

1.1.2 AMX Instructions 3

1.2 GPUs in AI Workloads 5

1.3 Prefetching 6

1.3.1 Stream Prefetcher 7

1.3.2 Stride Prefetcher 8

1.3.3 GHB Prefetcher 9

1.3.4 2DC Prefetcher 10

1.4 LLMs 11

1.5 Other Workloads . . . 11

Chapter 2: Methodology . . . 12

2.1 MacSim Upgrade . . . 12

2.1.1 Trace Generation . . . 13

2.1.2 AMX Instruction Support . . . 14

2.1.3 Validation . . . 14

2.2 Prefetching in MacSim . . . 15

Chapter 3: Results . . . 17

3.1 Baseline Performance . . . 17

3.2 Prefetcher Testing . . . 19

3.3 Stream Prefetcher Manipulation . . . 19

3.4 Data Trends . . . 21

3.5 Data Outliers . . . 22

3.5.1 L1 Caches in 512x512 Benchmark . . . 22

3.5.2 L3 Caches in 256x3072x768 Benchmark . . . 23

3.6 Prefetched Data Usage . . . 25

Chapter 4: Discussion . . . 26

4.1 How the Prefetchers are Failing . . . 26

4.1.1 AMX Instruction Memory Accesses . . . 26

4.1.2 Alternative Stream Buffer . . . 27

4.2 Possible Solution . . . 28

Chapter 5: Conclusion 30

Appendices 31

Appendix A: Experimental Equipment and Software 32

Appendix B: Data Processing 33

References 39

LIST OF TABLES

3.1 Baseline Memory System Configuration 17

3.2 Baseline Square Matrix Simulation Results 17

3.3 Baseline Small ML Matrix Simulation Results 18

3.4 Baseline Large ML Matrix Simulation Results 18

3.5 Prefetcher Basic Configurations 19

B.1 Experiment: 16-entry Prefetch Distance and 64 Stream Buffers 33

B.2 Experiment: 16-entry Prefetch Distance and 128 Stream Buffers 33

B.3 Experiment: 16-entry Prefetch Distance and 256 Stream Buffers 34

B.4 Experiment: 16-entry Prefetch Distance and 512 Stream Buffers 34

B.5 Experiment: 64-entry Prefetch Distance and 64 Stream Buffers 34

B.6 Experiment: 64-entry Prefetch Distance and 128 Stream Buffers 34

B.7 Experiment: 64-entry Prefetch Distance and 256 Stream Buffers 35

B.8 Experiment: 64-entry Prefetch Distance and 512 Stream Buffers 35

B.9 Experiment: 256-entry Prefetch Distance and 64 Stream Buffers 35

B.10 Experiment: 256-entry Prefetch Distance and 128 Stream Buffers 35

B.11 Experiment: 256-entry Prefetch Distance and 256 Stream Buffers 36

B.12 Experiment: 256-entry Prefetch Distance and 512 Stream Buffers 36

B.13 Experiment: 1024-entry Prefetch Distance and 64 Stream Buffers 36
B.14 Experiment: 1024-entry Prefetch Distance and 128 Stream Buffers 36
B.15 Experiment: 1024-entry Prefetch Distance and 256 Stream Buffers 37
B.16 Experiment: 1024-entry Prefetch Distance and 512 Stream Buffers 37
B.17 Experiment: 2048-entry Prefetch Distance and 64 Stream Buffers 37
B.18 Experiment: 2048-entry Prefetch Distance and 128 Stream Buffers 37
B.19 Experiment: 2048-entry Prefetch Distance and 256 Stream Buffers 38
B.20 Experiment: 2048-entry Prefetch Distance and 512 Stream Buffers 38

LIST OF FIGURES

1.1 Illustration of float16 and bfloat16 bit layout 4

1.2 Illustration of stream buffers 8

3.1 Average runtime decrease for each of the four prefetchers tested in their default configurations. 20

3.2 Average performance increase with increased stream length. 21

3.3 Average performance increase with increased number of stream buffers. . . 22

3.4 Average performance improvement per benchmark across all experiments with different stream prefetcher configurations. 23

3.5 Average percentage of matched address loads of prefetched data vs. prefetch distance. 24

LIST OF ACRONYMS

2DC 2-Delta Correlation

AI Artificial Intelligence

AMX Advanced Matrix Extensions

API Application Programming Interface

BF16 Brain Float 16

CPU Central Processing Unit

DCPT Data-Correlating Prediction Tables

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

FIFO First-In First-Out

GHB Global History Buffer

GPU Graphics Processing Unit

IT Index Table

LLC Last-Level Cache

LLM Large Language Model

PC Program Counter

PTX Parallel Thread Execution

RPT Reference Prediction Tables

SIMD Single Instruction, Multiple Data

TILECFG Tile Control Register

TMUL Tile Matrix Multiply Unit

XED X86 Encoder Decoder

SUMMARY

Many advancements in the machine learning field have led to Graphics Processing Unit (GPU) being the de facto standard of large scale matrix computation accelerator. This work aims to answer questions about how difficult it is to use the traditional Central Processing Unit (CPU) with matrix accelerators and prefetching hardware integrated into the core instruction set for Artificial Intelligence (AI)- and Large Language Model (LLM)-like workloads. Specifically, this work focuses on the usage of stream prefetchers to optimize the performance of tile multiply instructions on Intel's Sapphire Rapids processors, referred to as the Advanced Matrix Extensions (AMX) instruction set.

CHAPTER 1
INTRODUCTION AND BACKGROUND

The field of high performance computing has long been focused on maximizing computational performance. Performance can come from many areas, whether that is increased power, optimized code, architectural advancements, parallelism in all of its different variations, speculation, caching, or any of the other great advancements over the last few decades. The contemporary computing landscape is focused on AI workloads and all the ways that the computational resources we have at our disposal can be leveraged to accelerate them. Primarily among those is the usage of GPUs in massively parallel data center compute clusters that manipulate hundreds of gigabytes of data to draw inferences on every dataset from images to language to processor behavior. This work intends to analyze a small portion of relatively new hardware advancements in CPUs from Intel Corporation that may show a new way forward for the miniaturization and distribution of this technology to a greater number of people.

1.1 CPUs in AI Workloads

Typically, the role of the CPU in AI workloads is that of a host processor that coordinates the efforts of the accelerator hardware that executes the various kernels of a neural network. In this case, the CPU packages data, and sends it over various interfaces into the accelerators' memory and initiates the kernel that will be executed on that data before retrieving it and storing it somewhere else. However, CPUs in most modern machines utilize complex instructions that allow them to execute sophisticated operations using hardware that is integrated into the processor's chip itself, as shown by extensions like SSE, AVX, MMX, etc [1] [2]. Many of these extensions allow CPUs to exploit Single Instruction, Multiple Data (SIMD) behaviors [3]. A common example of this is vector units being used

to replace loops at compile-time by converting a set of identical instructions into a single vector instruction that executes all or part of a loop in parallel. This is useful when these instructions would normally require a looping construct to execute. In this work, we focus mostly on the AMX instruction set [2].

1.1.1 SIMD Programming on CPUs

SIMD programming can be done in multiple ways: inline assembly, intrinsic functions, and other higher-level programming approaches such as classes or dedicated multi-threaded approaches.

Inline Assembly

Inline assembly is the most direct way to access the exact instructions needed to perform a given operation other than directly writing assembly-level code, but it also tends to be unwieldy or slow to write.

Intrinsic Functions

Intrinsic Functions are functions that provide an abstraction of Inline Assembly that is friendlier to use for programmers. The trade-off is that the programmer must know of the existence of said functions and the limitations that they have. There is also no guarantee that the intrinsics are optimized as highly as possible.

Higher-Level Programming

Writing code that performs SIMD-style execution can often make the most sense to programmers but comes at a much greater performance cost than using intrinsics. This kind of implementation should generally only be used when a given operation is too sophisticated to use the available intrinsics, and can often be combined with intrinsics for further optimization.

1.1.2 AMX Instructions

Intel's AMX Instructions are an extension of the instruction set that enables the use of hardware matrix multipliers on the INT-8 and BF16 data types directly on the CPU.

INT-8 Data

INT-8 data is 8-bit integer data that can represent the values 0 to 255 or -128 to 127. It is popular for its use in quantization in Deep Neural Network (DNN) applications [4]. It is used in this context by setting a known scaling factor for your data, and then storing all of the data in a matrix of 8-bit integers. This trades the runtime execution of $Real_Number = stored_integer \times scaling_factor$ for the reduced memory footprint of 8-bit integer data. For example, a scaling factor of 2^{-10} might be chosen such that all of the weights stored in the INT-8 matrix are scaled by a factor of 2^{-10} when used at runtime. The reason this is done is that the weight data for matrices in many DNN workloads exceeds hundreds of MB, and in the cases of LLMs can even reach hundreds of TB [5] [6].

BF16 Data

Brain Float 16 (BF16) is a floating point data storage type that acts as a shortened 16-bit version of the classical 32-bit IEEE 754 floating point number representation [7]. It consists of 1 sign bit, 8 exponent bits, and 8 significand bits, 7 of which are explicitly stored and one being implicitly stored as 1, according to the implicit bit convention [8]. This is different from the typical float16 storage type in that the number of bits that are spread between the exponent and the significand are 5 and 11, respectively, in float16 as illustrated in Figure 1.1. This is typical in AI workloads because it allows the hardware to easily use the same memory space to expand to different data sizes, at the cost of precision in the significand.

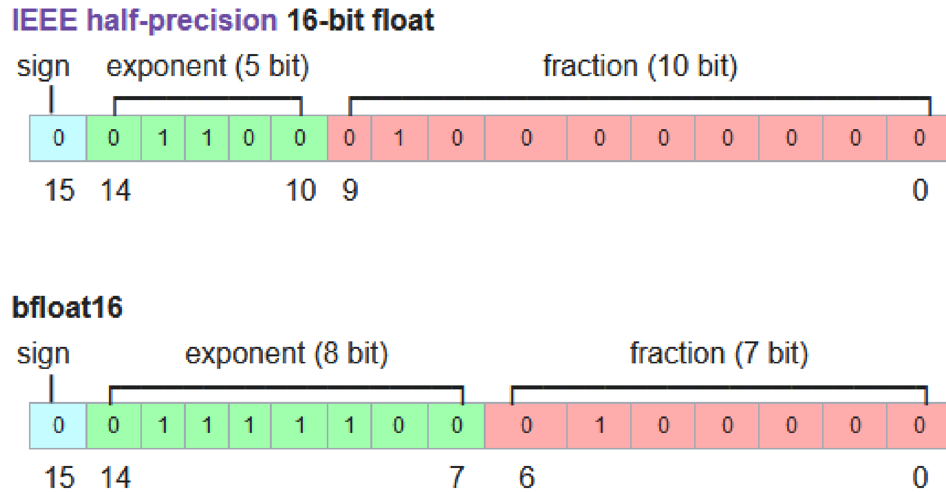

Figure 1.1: The different floating point representations use different bit layouts, which give different precisions and value ranges. This choice can also affect the ease with which a stored value can be converted to a higher precision value in hardware. Image credit: Wikimedia Foundation.

Hardware Components

Intel's AMX instructions make use of eight tile registers, a Tile Control Register (TILECFG) and a matrix accelerator called the Tile Matrix Multiply Unit (TMUL). The tiles are two-dimensional, and can be configured up to 16x32-entry BF16 or 16x64-entry INT-8. TMUL is a generic accelerator that can be extended or upgraded by later revisions to the hardware, as long as it complies with the instructions as defined in [2].

Typical Operation

The AMX instruction set utilizes twelve different instructions for its operations [2]. These instructions allow the programmer to configure the tile sizes, load data, store data, zero out tiles, release tiles, and run different matrix multiply operations using provided CPU intrinsics.

The workflow of using this hardware is shown in Algorithm 1. The matrix multiply function can be either the dedicated BF16 function, if the data is in BF16 format, or one of the four INT-8 functions, depending on the ordering and signedness of the input tile

data in INT-8 format. Depending on the size of the output matrix $data0$ and each input matrix $data1, data2$, it might be necessary to write a looping version of this code that loads data from the input tiles and performs the matrix multiply function multiple times before storing the data back to $data0$, as shown in Algorithm 2. In our testing, AMX instructions support this accumulation implicitly without needing to store after every invocation of the matrix multiply function because the data storage format of the output tile defaults to INT-32 when using INT-8 data, and the data storage format of BF16 is a floating-point number representation.

Algorithm 1 General AMX hardware workflow

Require: $data0, data1, data2, config$
  load_config($config$)
  load_data($tile0, data0$)
  load_data($tile1, data1$)
  load_data($tile2, data2$)
  matrix_multiply($tile0, tile1, tile2$)
  store_data($tile0, data0$)
  release_tile($tile0$)
  release_tile($tile1$)
  release_tile($tile2$)

1.2 GPUs in AI Workloads

The role of the GPU in AI workloads is that of the accelerator. All the data sent from the CPU is executed in kernels chosen by the programmer that run on the GPU. GPUs are designed to exploit massive-scale SIMD-style parallelism directly in the programming model. They contain thousands of processing units that batch data together such that hundreds of identical instructions can operate on different values simultaneously. Traditionally, these were developed as a means to accelerate the linear algebra workloads that are common to graphics rendering, but the hardware has since been employed in other workloads and is used more commonly in data-processing and neural networks, as evidenced by NVIDIA's earnings in data center GPUs totaling over $18 billion and its other earnings totaling to less

than $4 billion [9].

GPUs are the dominant force in this area. The goal of this work is not to disprove the value of GPU-dominance in this field, but to analyze whether the CPU can be leveraged as more than a host device. As most computing technology advances, there are pushes to miniaturize existing solutions, either to increase density or portability, or to reduce power consumption. An increase in CPU-based solutions could present a way to shift computation to mobile processors, like cell phones, or possibly for end users to run these demanding workloads on their own machines without needing expensive dedicated processors or a fast internet connection. At present, GPUs are the best solution for performance and cost, but we hope to show that there is viability in more CPU-based approaches.

1.3 Prefetching

Prefetching is a speculation technique that involves fetching data from farther-away memory, such as Dynamic Random Access Memory (DRAM) or the disk, and loading it into closer memory in the expectation that it will be used soon by the processor. It is employed in numerous ways, both automatically and manually, statically and dynamically, to increase performance in nearly all applications executed on modern processors. The most common form of prefetching is done in the caching hierarchy on CPU chips, where a full cache line is loaded in upon a cache miss, instead of a single value. This is done because of the simple logic that data close to whatever the processor just loaded from memory is likely to be used soon after the data that is needed during the current instruction. An entire cache line is then prefetched from higher up the hierarchy so that on the high chance that close data is needed, there is not another immediate cache miss. This logic can be extended to page faults loading full pages into DRAM from disk, and many other intermediate memory accesses and loads [10].

Another typical form of prefetching is manual prefetching expressed by a programmer. In this case, an entire chunk of data might be loaded directly into memory because the

programmer knows that it will be used in the immediate future. This kind of prefetching is very useful for regular data access patterns, like those in matrix arithmetic operations, where a large amount of data whose location is computable or known is worked on for a significant volume of instructions. The prefetching that we will be focused on this work is from this latter category.

When using prefetching to accelerate matrix workloads, there are a number of advantages. First, matrix sizes might be known at compile time, and they will definitely be known at runtime. This is hugely advantageous for prefetching, because if the programmer knows how much data needs to be loaded, they can manually insert prefetch instructions that the compiler can use to optimize memory accesses [2]. Alternatively, should the system anticipate a matrix workload, cache loads can be optimized to stream data in and out of the memory system so that the majority of execution time is not spent waiting on memory.

The prefetching models we have focused on are the following:

1.3.1 Stream Prefetcher

Stream prefetching, or the use of stream buffers in caches, is the most typical form of hardware prefetching. The original use of stream buffers is that, upon loading an address A, an additional n subsequent addresses are also loaded under the assumption that they will be used soon [11]. These additional subsequent addresses of data are stored in a stream buffer of depth n and will be loaded into the cache in a queue-like fashion if the processor requests that data, instead of going up to the next level of memory.

For example, if address A is requested by the processor, and the cache loads it, then addresses $A + 1$ through $A + n$ will be loaded into the stream buffer in sequential order. When the processor requests address $A + 1$, it is loaded from the head of the stream buffer queue into the cache, and the rest of the addresses in the stream buffer are moved forward, such that address $A + 2$ is now at the head, followed by $A + 3$ and so on, until $A + n$ [11]. This approach is called Sequential Prefetching and can be extended such that there

are multiple stream buffers for a given cache (shown in Figure 1.2) [12]. The exact depth of a stream buffer that is ideal for a given workload and microarchitecture must be determined experimentally [11] [13]. Typically, this type of hardware prefetching is used for instruction prefetching, as it is fairly likely that the next instruction in memory is the one that will be executed next.

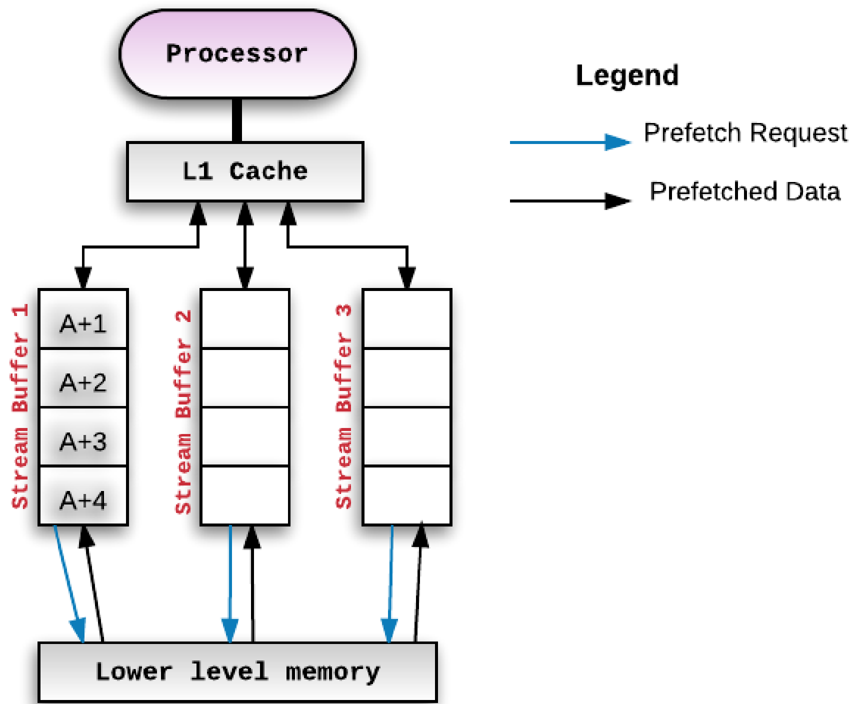

Figure 1.2: An illustration of stream buffers as they were originally proposed by Norman Jouppi in 1990 and expanded upon by Yasuo Ishii in 2009. Image credit: Wikimedia Foundation.

1.3.2 Stride Prefetcher

This hardware prefetching technique actively monitors memory accesses to determine patterns in the distance between subsequent accesses, referred to as strides. The types of strides can be broken down further into three subcategories: regular strides, irregular spatial strides, and irregular temporal strides. This was first implemented via Reference Prediction

Tables (RPT) in [14] in 1995.

Regular strides s are consistent accesses at consistent intervals. For example, a looping construct is consistently accessing every third address in a memory region, so the prefetcher computes $s = 3$, then prefetches every third entry in the region, anticipating that the CPU will continue this pattern.

Irregular spatial strides $s_{0..n}$ feature inconsistent access distances that still follow a pattern. For example, consider a memory access pattern of $A, A+1, A+3, A+7, A+15, A+16, A+17, A+19, A+23, A+31, ...$ This pattern is two iterations of five accesses, where the distance between each subsequent access is doubled, starting with $s = 1$. There are prefetcher designs that could predict this access pattern, as discussed in [15], and measurably increase performance through an intelligent prefetching scheme.

Irregular temporal strides $s_{0..n}$ feature consistent patterns that insert themselves in otherwise inconsistent streams of accesses. For example, an access stream featuring the same four addresses, or (more generally) the same pattern between four subsequent addresses, at seemingly random intervals, with no discernible pattern separating intervening accesses to those four addresses. There is a discussion at [16] that utilizes a Global History Buffer (GHB) to design a prefetcher that is able to predict these types of access patterns.

1.3.3 GHB Prefetcher

The GHB prefetcher was created with the goal of solving three major problems with prefetch tables. First, prefetch accuracy suffers when table data becomes stale. Second, reducing table conflicts results in ballooning memory requirements and increases the proportion of stale data in the table. Third, increasing the prefetch history stored per entry has the same drawbacks as increasing the table size [17]. To solve these issues, Nesbit and Smith designed the GHB prefetcher using two major components: the Index Table (IT) and GHB. The IT is a table keyed with a value like the Program Counter (PC) or cache miss address which accesses an entry in the GHB. The GHB is an n-entry First-In First-Out

(FIFO) table that holds the n most recent L2 miss addresses [17]. Each entry contains the global miss address and a link pointer, which is used to chain GHB entries into an address list. An address list is the time-ordered sequence of addresses that have the same IT key.

Because of how the tables are structured, GHB prefetching can be used to implement many history-based prefetching algorithms, including stride prefetching discussed in subsection 1.3.2 [17]. Nesbit and Smith explain in their paper how to use the GHB prefetching to implement local delta correlation, stride, and markov prefetchers using this system. The GHB prefetcher implemented in our testing framework is the C/DC prefetcher described in [18].

1.3.4 2-Delta Correlation (2DC) Prefetcher

The 2DC prefetcher is an implementation of the Data-Correlating Prediction Tables (DCPT) discussed in [19]. DCPT combines RPTs with a technique known as PC/DC prefetching, which calculates the deltas between successive cache misses and stores them in a delta-buffer. This delta-buffer can be used to correlate the deltas in an access stream and determine a pattern which can be used to prefetch data. When used, the correlated deltas are added to the currently missed address to prefetch the data that is assumed to be needed next.

The design of a DCPT uses a large table for delta correlation indexed by the PC of a given load instruction. Each entry in the table contains the PC, the last address, the last prefetch, the deltas for accesses 1 through n, and the delta pointer. The last address field is filled in with the missed address (as in the RPT prefetcher), and each delta is initialized to zero with the delta pointer pointing the to the head of the circular buffer containing the n delta fields. When the circular buffer is updated, the deltas are traversed in reverse order, looking for a match to the two most recently-inserted deltas. If a match is found, the first prefetch candidate is computed by adding the last address value to the matched delta, and the next is computed by adding the next delta to the first computed candidate. These values

are stored, and the process is repeated for each delta after the matched pair, including the two newest-inserted deltas. If the addresses in the prefetch candidate buffer do not already exist in cache, they are checked against the miss status registers in the memory system to see if the cache lines have already been requested. If not, they are checked against another prefetch request register to see if the thirty-two registers in their system hold the addresses, indicating that they have already been requested to be prefetched. If this buffer does not contain them, but is full, the prefetch is discarded. At the end of this process, the last prefetch field is set to the address of the issued prefetch, if it was not discarded [19].

The "2-delta" portion of this correlation refers to the fact that the newest two deltas inserted into this system are the deltas being correlated against the access history.

1.4 LLMs

LLMs are an application of a DNN to generate human-like language. At a high level, they do this using massive training data sets that allow the neural network to predict words or phrases based on the context of the previously-generated or user-prompted linguistic input. The primary hardware systems that are used for these models are massive data center-scale GPU systems, but there are ports of some LLMs to run entire on CPU [20].

1.5 Other Workloads

There are other workloads, such as graphics rendering and various forms of scientific computing, which utilize matrix operations extensively to model physical systems.

CHAPTER 2
METHODOLOGY

The data collection process began with a performance analysis of various LLMs running natively on Intel Sapphire Rapids machines. The purpose of this was to confirm that the primary compute kernel consisted of vector operations that could be modeled instead as matrix operations. Sapphire Rapids was chosen because it has instruction set-level support for matrix multiplication operations that are not present in other commercial processors. These operations are referred to as AMX instructions.

After collecting these native data, a simulation model of the AMX instructions was required in MacSim with a configurable simulated prefetching system. Before these could be built, MacSim required upgrading and extending to support AMX instructions. Once these pieces were in place, MacSim could be used to generate cycle-accurate simulation data on traces of matrix multiplication workloads.

2.1 MacSim Upgrade

MacSim is a cycle-driven heterogeneous computer architecture simulator developed in Georgia Tech's HPArch Lab that supports simultaneous simulation of CPU and GPU programs. Before the start of this project, it supported x86 architectures like Intel Skylake and NVIDIA instructions like Parallel Thread Execution (PTX), which is used in CUDA as an assembly layer of instructions. To collect simulated performance data on AMX code, we needed to support two additional things: trace generation of AMX code, and simulation of AMX instructions in MacSim.

2.1.1 Trace Generation

To generate traces of running code for MacSim, we use Intel's Pin program, which allows us to instrument programs at the instruction level to perform a large variety of tasks. This can be used to generate a large number of statistics for code running in Pin's black box, and for our purposes can be used to generate a text output that details various things about an instruction, like its PC, the registers it uses, the X86 Encoder Decoder (XED) category of an instruction, the exact instruction mnemonic, etc. All of this information is used to generate a runtime trace of the executing program, but it forces the execution to take significantly longer.

In order to support AMX instructions in the trace generation step, a number of small changes were required in the trace generation code that comprises our "pintool." A pintool is a small application that uses Pin's Application Programming Interface (API) and is passed into the Pin program to perform the exact instrumentation needed for a given purpose. HPArch lab has already created a pintool for this purpose, and our job was to extend it. First, we had to take the most up-to-date list of XED categories from Intel, and expand the support for these categories at every stage of the MacSim's pipeline. Second, we needed to add logic into the trace generator pintool to determine which AMX instructions were being executed so that we could grab the information that we needed from them. This required checking the XED category and then extracting the instruction mnemonic using Pin's API. Most instructions did not require any additional special instrumentation, but all memory operations did. In these cases, we determined that, while it would be possible to support arbitrary AMX tile configurations, it would be a better usage of our time to assume that the tiles always used their full size, and that any matrix operations performed were using matrix dimensions that were multiples of the base AMX tile matrix. This limits the applications that we can model and test in the current framework, but it does not reduce our ability to experiment on high-volume matrix workloads.

AMX Code Traces

Traces were generated of code that executed the algorithm presented in Algorithm 2. The code is structured such that the matrices are first generated with the given dimensions. Once generated, the AMX system is initialized by loading the chosen configuration, which in our case was always the maximum tile size configuration of 16x64 INT-8. Then, the tiling of the result matrix is computed to determine the looping dimensions for the multiplication. Finally, in the loop body each tile is loaded into the appropriate tile register to be multiplied and accumulated in the result tile. The tile loading instruction supports striding through a buffer, and allows us to avoid costly memory operations for computing the stride ourselves. After accumulation, the tile is stored back to the result matrix, and when all result tiles are computed, the tile registers are released.

2.1.2 AMX Instruction Support

As stated in subsection 2.1.1, we assumed that tile data was being loaded at its maximum size. This means that for INT-8 data and BF16 data, there are 16x64 entries and 16x32 entries, respectively, in a tile. What this meant for simulator support was essentially routing data to the correct locations. Initially, we were worried that the increased size of the memory loads would be a problem in MacSim because it was written with the assumption that data loads never exceeded 16, but this did not matter. Due to our earlier assumption that we load matrices only of size equal to the entire tile, we can model the tile load as 16 loads of 64 bytes each. This greatly simplified the implementation of this instruction handling and allowed us to proceed to generating and simulating workloads fairly quickly.

2.1.3 Validation

Concurrently with the work to modify the trace generator in MacSim, we also developed a manual trace generation scheme that produced simple traces of AMX code without a running application. This code was identical to the basic application test that we used,

which consisted of loading the tile configuration, creating and filling three matrices of size 16x64, loading those matrices into tiles, executing an INT-8 matrix multiply, storing the data back to memory, and then releasing the tiles. We knew what the expected results were for this simple program and executed both traces on different versions of MacSim, both of which supported the new AMX instructions. Having generated the same statistics for AMX instruction count, loads, and stores, we felt confident that the upgrade was successful and proceeded to creating a larger testing framework, which would also be used for performance analysis once the prefetching schemes had been implemented.

2.2 Prefetching in MacSim

As discussed in section 1.3, we integrated four different prefetching methods into MacSim. The descriptions of each algorithm are included in their respective subsection in chapter 1, and the exact implementation in MacSim differs very little from these descriptions. Namely, the stream prefetchers do not utilize separate buffers in our design, and instead prefetch directly into the cache for the sake of simulation simplicity. It is assumed that if data is in the stream buffer, then it can be accessed immediately by the cache, and the stream buffer's start and end points are stored to check whether a given address would be placed in the cache by that buffer.

Algorithm 2 AMX Traces

Require: $dimA, dimB, dimC, config$

  $matA \leftarrow$ gen_matrix($dimA, dimB$)
  $matB \leftarrow$ gen_matrix($dimB, dimC$)
  $matC \leftarrow$ zero_matrix($dimA, dimC$)
  load_config($config$)
  $[matCTilesPerColumn, matCTilesPerRow] \leftarrow$ compute_tiling_dimensions($matC$)
  $[matATilesPerResTile, matBTilesPerResTile] \leftarrow$ compute_tiling_length($matA, matB$)
  if $matATilesPerResTile \neq matBTilesPerResTile$ **then**
    abort()
  end if
  $V \leftarrow 0$
  while $V < matCTilesPerColumn$ **do**
    $H \leftarrow 0$
    while $H < matCTilesPerRow$ **do**
      load_tile($tile0, dataC$)
      $tile \leftarrow 0$
      while $tile < matATilesPerResTile$ **do**
        shift_src_windows($dataA, dataB, tile$)
        load_tile($tile1, dataA$)
        load_tile($tile2, dataB$)
        amx_mult($tile0, tile1, tile2$)
      end while
      $H \leftarrow H + 1$
    end while
    store_tile($tile0, dataC$)
    copy_tile($dataC, matC$)
    $V \leftarrow V + 1$
  end while
  release_tile($tile0$)
  release_tile($tile1$)
  release_tile($tile2$)

CHAPTER 3

RESULTS

3.1 Baseline Performance

A baseline configuration is needed to accurately assess the value of any changes made during the experiments. The memory system configuration for all of the base runs is shown in Table 3.1. This base configuration was chosen to match the largest per-core cache configuration of an Intel Sapphire Rapids processor. Additionally, a perfect instruction cache was assumed.

Table 3.1: Baseline Memory System Configuration

Memory Component	L1	L2	L3
Number of Sets	48	512	14440
Set Associativity	16	32	64
Cache Line Size (bytes)	64	128	128
Latency (cycles)	3	24	50
Total Size (KiB)	48	2048	115200

Table 3.2: Baseline Square Matrix Simulation Results

Matrix Dimensions	512x512	1024x1024	2048x2048	4096x4096
Cycle Count (Millions)	16.592	88.057	593.551	4476.096
L1 Hits (%)	4.907	6.845	11.305	15.069
L2 Hits (%)	64.343	70.546	21.990	91.847
L3 Hits (%)	0.235	3.394	73.466	97.596
AMX Instructions (%)	7.71918	20.8951	30.8621	14.8731
Binary Core Instructions (%)	31.42	39.7539	46.0535	20.2618
BR Instructions (%)	18.6285	15.0794	13.1963	5.18295
Data Xfer Instructions (%)	23.6168	14.9861	6.87301	0.422488

In Table 3.2, Table 3.3, and Table 3.4, the cycle count, cache hits, and the four instruction categories with the largest presence in runtime are given for each trace. The square matrix traces are each a single matrix multiply operation between two square matrices of

Table 3.3: Baseline Small ML Matrix Simulation Results

Matrix Dimensions	256x256x768	256x768x256	256x768x768
Cycle Count (Millions)	10.402	10.159	19.634
L1 Hits (%)	3.431	3.879	3.619
L2 Hits (%)	64.612	49.401	56.904
L3 Hits (%)	0.364	0.350	0.191
AMX Instructions (%)	3.86523	3.47137	8.10351
Binary Core Instructions (%)	28.9853	28.8648	31.5616
BR Instructions (%)	19.4049	19.7165	18.1872
Data Xfer Instructions (%)	26.63	26.4373	24.4127

Table 3.4: Baseline Large ML Matrix Simulation Results

Matrix Dimensions	256x768x2304	256x768x3072	256x3072x768
Cycle Count (Millions)	46.668	60.173	60.500
L1 Hits (%)	20.482	23.267	4.901
L2 Hits (%)	86.184	87.583	53.099
L3 Hits (%)	0.327	0.279	1.699
AMX Instructions (%)	3.05501	3.12271	7.55835
Binary Core Instructions (%)	6.64637	6.20462	17.1885
BR Instructions (%)	2.89325	2.50564	7.60857
Data Xfer Instructions (%)	2.80016	2.20829	8.98159

the given dimensions that are tiled using Intel's AMX instructions. In Table 3.3 and Table 3.4, the middle dimension is the shared dimension, and the traces are also a single matrix multiply. The dimensions of the matrices in the traces used in Table 3.3 and Table 3.4 were chosen based on the six layers of the Bert benchmark.

As expected, the baseline workloads presented here show a positive correlation between cycle count and the dimensions of the result matrix. This gives a trend of increasing Last-Level Cache (LLC) hits as cycle count and result matrix dimensions increase. The increased number of cache hits can be seen as well in the massively decreased volume of data transfer instructions as the matrices get larger and can fetch data from higher level caches instead of DRAM. Likewise, there appears to be no relationship between the size of the shared dimensions and execution time. The execution time scales the most with the dimensions of the result matrix. This is likely because larger dimensions in the result matrix results in a larger number of looping passes which execute the interior multiplication logic.

The number of multiplies per result tile scales only with the size of the shared dimensions and remains constant throughout execution.

3.2 Prefetcher Testing

Table 3.5: Prefetcher Basic Configurations

Prefetcher	2DC	GHB	Stream	Stride
Prefetch Degree	16	16	—	—
Max Prefetch Degree	32	32	—	4
Associativity	4	—	—	—
Line Size	1	—	—	—
Prefetch Distance	—	—	64	16
Buffers	—	256	256	—

The first thing to test was each of the four prefetchers on their base configurations. These configurations are given in Table 3.5, and the results of these tests are summarized in Figure 3.1. The stream prefetcher, with no special configuration, showed the best performance increase of the four tested, so further efforts were focused on optimizing the performance of the stream prefetcher with the benchmarks enumerated in section 3.1.

3.3 Stream Prefetcher Manipulation

Two major parts of the stream prefetcher were manipulated to find optimized performance: the prefetch distance and the number of stream buffers. The prefetch distance determines how many addresses are pulled into the buffer when a stream buffer is allocated, and the number of stream buffers determines the number of outstanding streams that can be allocated to prefetch data from. The prefetch distance was varied across the following values: 16, 64, 256, 512, 1024, and 2048. The number of stream buffers was varied across the following values: 64, 128, 256, and 512. The results are summarized in Figure 3.2 and Figure 3.3. The data that all of the figures in this chapter are derived from are presented as tables in Appendix B, as they are too numerous to display in this chapter.

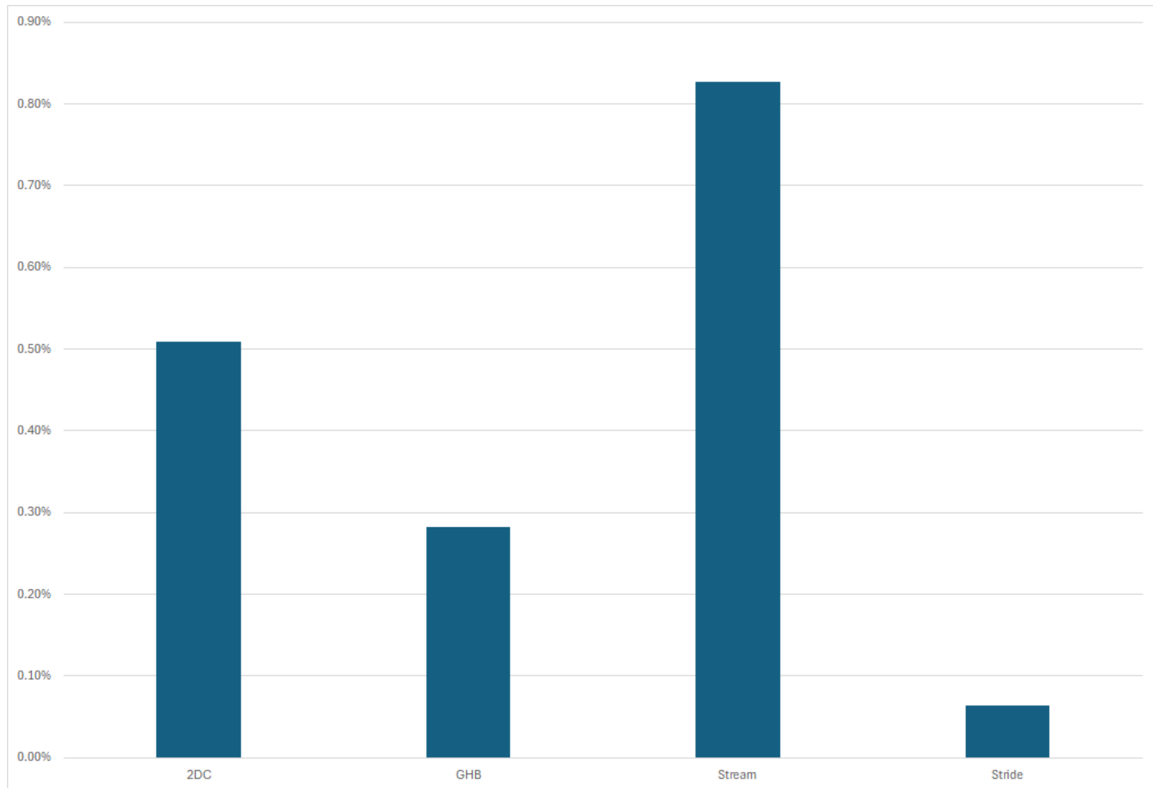

Figure 3.1: Average runtime decrease for each of the four prefetchers tested in their default configurations.

In each figure, the data points represent the average increase in performance for all experiments matching that configuration. For example, the first data point in Figure 3.2 is the average performance increase for every experiment where the stream buffer has a prefetch distance of 16 addresses, the second data point is the average performance increase of every experiment where the stream buffer has a prefetch distance of 64 addresses, and so on.

These two figures allow us to pinpoint an expected range in which we will see the best performance increase for a given stream prefetcher configuration: 256-address prefetch distance in 128-256 buffers. Additionally, the two figures show us that prefetch distance has a larger positive and negative affect on the performance outcome than modifying the number of stream buffers. After 128 stream buffers are allowed in the system, the performance not only does not improve, but it worsens.

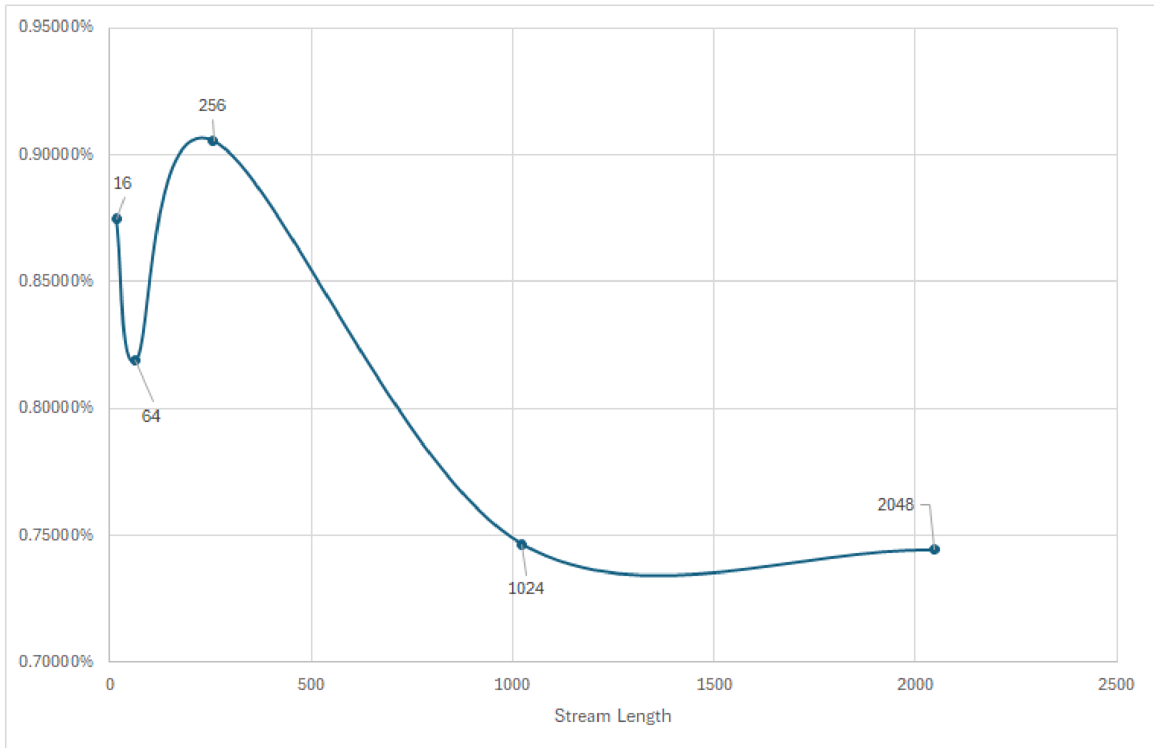

Figure 3.2: Average performance increase with increased stream length.

3.4 Data Trends

The tables in Appendix B show that smaller benchmarks benefit from stream prefetching more. Figure 3.4 shows that four out of the ten benchmarks (all of those which have dimensions less than 1000 in all matrices) are the only benchmarks which consistently reached and exceeded 1% improved performance with prefetching enabled. All four of these benchmarks are those which can fit all three working matrices into the L2 and L3 caches at once. Despite this being far lower than expected, it was surprising that this occurred because their ability to stay present in L2 and L3 should give them the largest headroom of all the benchmarks tested, and therefore less responsiveness to prefetching beyond the L1 cache. However, it is just as likely that simple cache fetching was enhanced by the prefetcher's ability to fill the cache with the necessary data quickly, and because the caches had the space for it, a sustained performance increase was more noticeable.

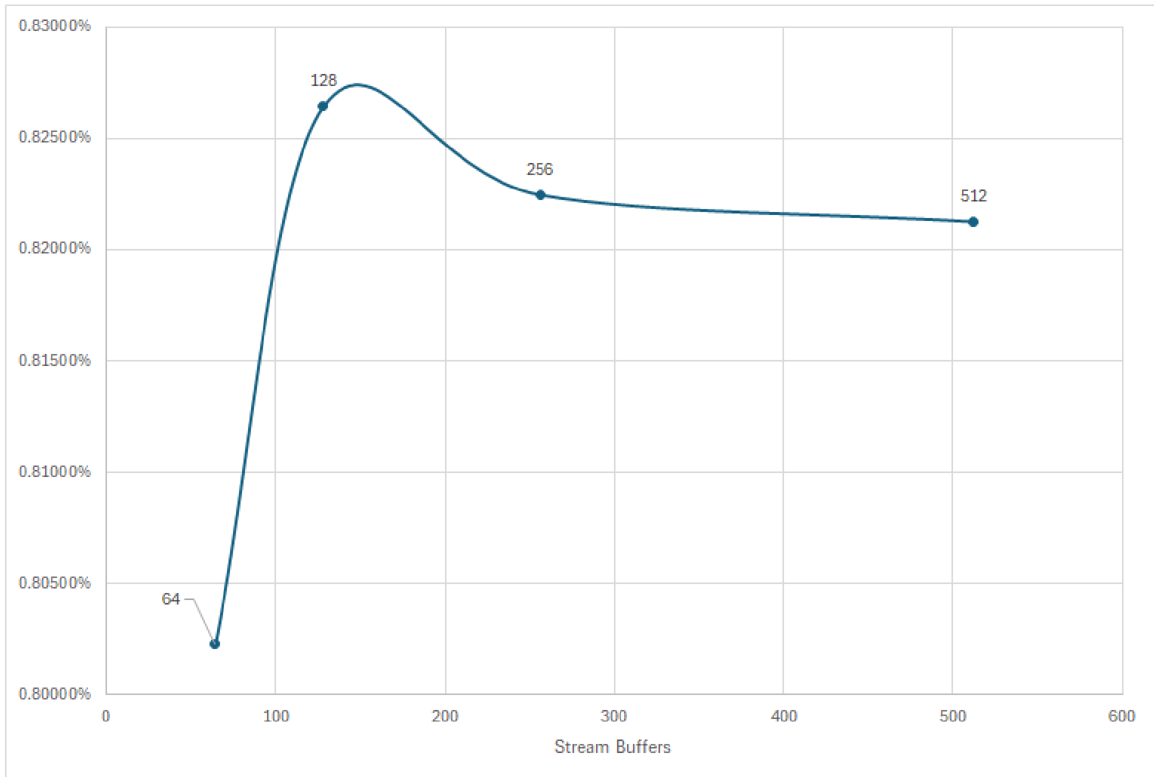

Figure 3.3: Average performance increase with increased number of stream buffers.

3.5 Data Outliers

The tables in Appendix B show some consistent outliers across multiple configurations. Namely, the L1 cache performance of the 512x512 square matrix benchmark is always significantly improved over the baseline (over 800% increase in cache hit percentage), and the L3 cache performance of the 256x3072x768 benchmark, especially once the stream prefetchers' sizes reach an excess of 128 buffers with 64-byte prefetching distance (160-700+% increase in cache hit percentage).

3.5.1 L1 Caches in 512x512 Benchmark

Considering the sizes of the caches in the system, shown in Table 3.1, the sizes of the matrices used in these benchmarks are all too large to fit entirely in L1. The L1 data cache is 48 KiB, and the smallest benchmark (256x256x768) contains 256 KiB in matrix source

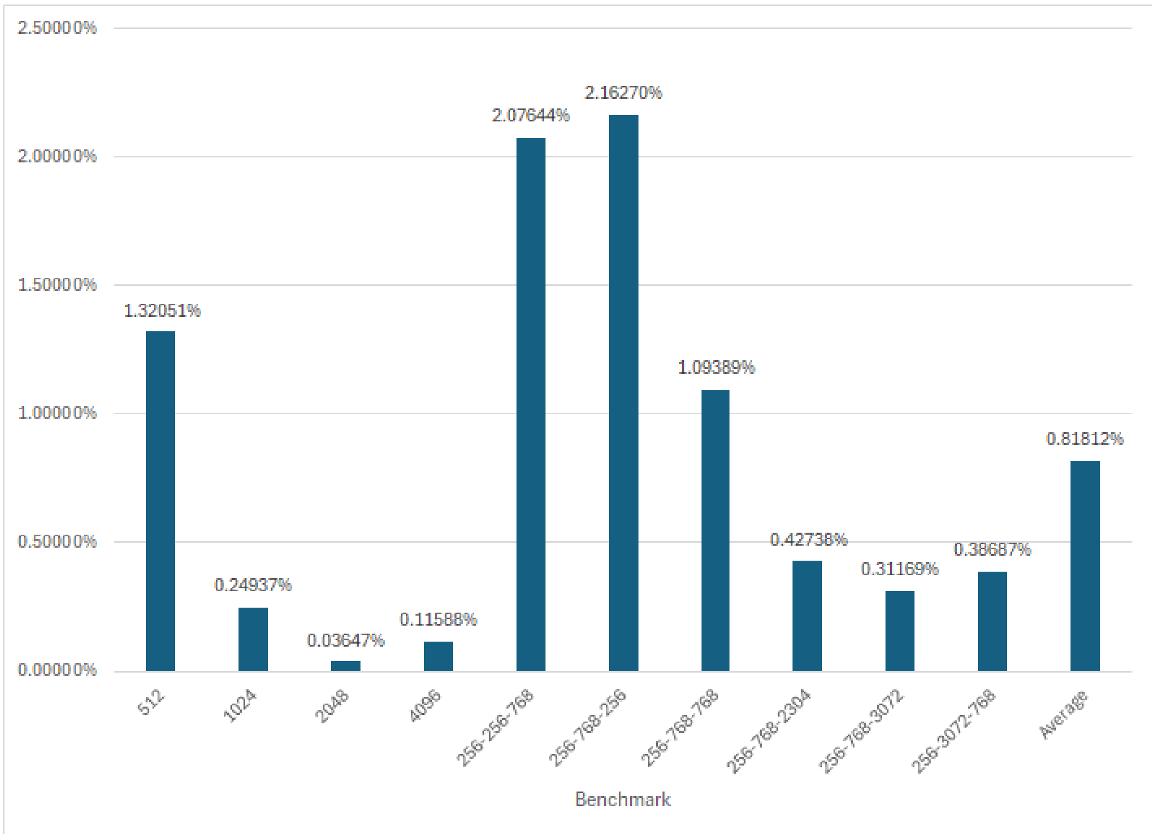

Figure 3.4: Average performance improvement per benchmark across all experiments with different stream prefetcher configurations.

data; the 512x512 matrix benchmark is double this size at 512 KiB of matrix source data. Knowing this, we can expect a decent amount of performance gains in L1 from prefetching, but only after each tile load has initially missed. Due to the nature of AMX instructions, matrix elements are not loaded completely sequentially. The 512x512 matrices benefit from this because the amount of data that they use can be prefetched in an almost perfect sequential ordering, with a few caveats, as discussed in section 4.1.

3.5.2 L3 Caches in 256x3072x768 Benchmark

The L3 cache of Intel's Sapphire Rapids processor line is capped at 112.5 MiB. This large size allows this cache to hold the entirety of any and all of the matrices used in these experiments. The largest storage requirement is in the 4096x4096 benchmark, which is 48 MiB between both sources and the result matrix. As expected, the number of cache hits

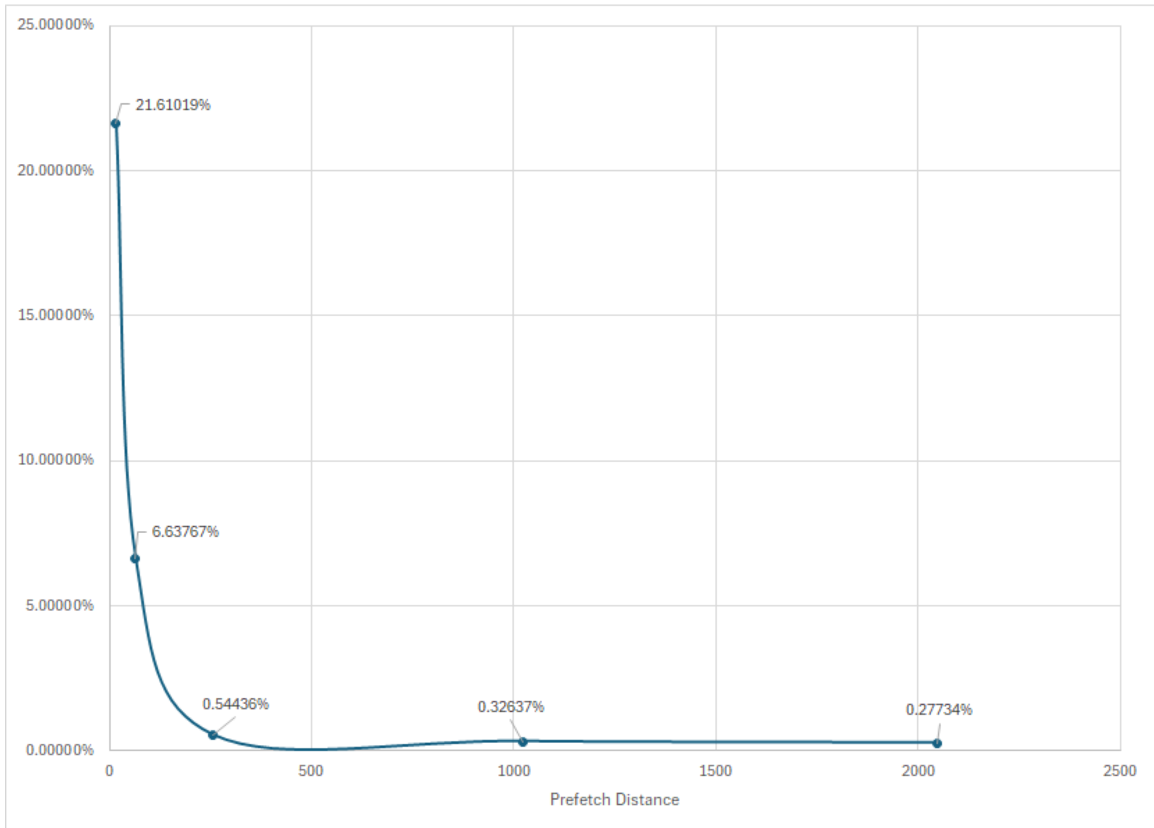

Figure 3.5: Average percentage of matched address loads of prefetched data vs. prefetch distance.

in L3 for this benchmark is very high with and without prefetching, never going below 93%. The 256x3072x768 benchmark, in contrast, has very different behavior. Its baseline L3 performance is the best of the Bert-derived benchmarks, but it only has a 1.699% hit rate. This again, is likely due to the imperfect sequential accesses of AMX instructions. In a square matrix like the 4096x4096 benchmark, the accesses can be predicted somewhat easily because every matrix has the same size and shape. This is not the case with the 256x3072x768 benchmark, as all three matrices involved have a different size and shape. What this leads to is an inconsistent number and sequence of prefetch requests being sent through the memory system at all levels of the cache hierarchy. Because all three tiles (both sources and the result) must be present for the instruction to execute, this can cause erratic stalling as one tile's data is available but another tile has not yet finished being fetched. This will be discussed further in subsection 4.1.1.

3.6 Prefetched Data Usage

As the prefetch distance increases, the amount of data loaded into the cache by a single prefetch request increases multiplicatively. Likewise, the usage of this data drops exponentially as the caches are overfilled with data that they do not use before it is replaced. Figure 3.5 shows how rapidly the usage of prefetched data falls when the prefetch distance is increased. Because L1 is the primary issuer of prefetch requests, when the prefetch distance is increased, it results in the cache being flooded with data on a miss, followed by immediate misses on any non-matrix loads. This issue is discussed extensively in subsection 4.1.1 and a potential solution to these problems is discussed in section 4.2.

CHAPTER 4
DISCUSSION

As shown in chapter 3, the hardware prefetchers used in conjunction with Intel's AMX hardware were less effective than expected. Our hypothesis is that this occurs because of way that instructions are structured and executed with the AMX instruction set alongside the behavior of a traditional caching system and classical hawrdware prefetchers.

4.1 How the Prefetchers are Failing

In a traditional $\mathbf{O}(n^3)$ matrix multiply operation, the matrices are accessed perfectly sequentially. A skilled programmer could possibly reorient the right operand matrix such that it is stored as a transposition, assuming row-major data preparation. This would allow the access patterns to each matrix to be perfectly sequential as each result entry in the computed matrix is calculated. This is a perfect case for the stream prefetcher and works in favor of a cache's spatial locality-focused design. AMX enforces this row-major and transposed-right-operand assumption for the programmer's convenience, but in a larger matrix it only benefits from cache's spatial locality-focused design.

4.1.1 AMX Instruction Memory Accesses

AMX's tile load and tile store instructions accesses memory in a strided pattern. The way this is implemented in hardware is by up to sixteen consecutive loads of up to sixty-four bytes of data, depending on the configuration of the tile, separated by however many bytes are indicated in the stride variable. In the case of our tile configuration, this results in sixteen sixty-four-byte loads per tile load instruction and the same amount of stores per tile store instruction.

In the smallest stream prefetcher configuration with the 512x512 benchmark discussed

in section 3.3, this will result in sixteen cache lines being prefetched into L1 on each miss. At sixty-four bytes each, this is enough to prefetch two whole rows of one of the source matrices. Two whole rows prefetched means that only eight of those load instructions will allocate stream buffers and issue prefetch requests. Additonally, because our prefetcher places data directly into the cache rather than queuing data to be placed in the cache when needed, we do not have to access the entire first row before the second row starts to be available at the head of the prefetch queue. The problem is that we are prefetching a massive amount of data into L1. The cache is 48 KiB, and the requests generated from one tile load equate to $64 \times 16 \times 8 \times 3 = 24$ KiB. This is why the L1 cache performance of this benchmark is so greatly increased, as discussed in section 3.5. We fill half the cache with relevant data on the first set of misses, assuming there are no cache collisions. But what happens when the matrix is bigger? The 1024x1024 benchmark would fetch one entire row from each matrix on the first missed tile load, but that uses $64 \times 16 \times 16 \times 3 = 48$ KiB of cache. The entire cache, again assuming there are no collisions, has now been completely filled with matrix data from a single tile load instruction. This means that every time the computation loop executes, there will be a new cache miss on any data that is not from the matrices. This intense cache pollution in L1 cascades through every step of the multiplication operation, and is shown by the extremely small usage of prefetched data, as shown in section 3.6. This same behavior explains the massively increased L3 cache performance for the larger matrix benchmark, discussed in subsection 3.5.2.

4.1.2 Alternative Stream Buffer

If we had instead used a traditional stream which stores a queue of memory blocks and moves them into the cache from the head of the queue when they are requested, it is possible that this pollution issue would be resolved. However, it would introduce a new issue in the smaller benchmarks when the stream buffer contains two overlapping rows of matrix data. In this case, the eight prefetch requests above would become sixteen, and half the

data in each prefetch request would be redundant. This might not have a noticeable effect on performance, and also might be preferable to cache pollution, but the inefficiency of the hardware prefetcher would be a detriment in other ways, such as power usage or the increased latency of servicing more requests.

4.2 Possible Solution

To stop the extreme cache pollution while maintaining a sequential access pattern, the following solution is proposed: a tile prefetcher. This prefetcher would achieve two things. First, the tile prefetcher will enable the prefetching of the large chunks of data that are consumed by the AMX instruction set. Second, the prefetcher will avoid cache pollution caused by overloading the cache with matrix data. The prefetcher will function as a classical queue-based stream prefetcher with a few major changes. Primarily, the memory blocks that it enqueues will be the exact shape and size of the configured AMX tiles. If it is possible to utilize the AMX hardware to issue tile loads through the prefetching hardware, this will enable the creation of a sizable prefetch queue that continuously loads a new tile from the head of the queue into the L1 cache without pollution.

Because the tile configuration is known, and an initial tile load instruction must cause a cache miss for the prefetcher to engage, the exact number of tiles to prefetch should be dynamically computable. For example, in the 512x512 benchmark, a single tile in the result matrix is computed with the accumulated result of eight 16x64 AMX matrix multiply operations. The prefetcher could compute this by taking the stride value from the tile load instruction and dividing it by the width of the tile. It could then allocate a buffer of size $16 \times 64 \times (512/64) = 8192$ bytes to prefetch those tiles. When the next tile load instruction is issued, it can then pass the data from the stream buffer into the tile register and shift the rest of the queue forward. This could be extended to track how many tiles would be necessary against how many have been prefetched so that the maximum size of the buffer can be optimized to relatively low number. In this state, the buffer could continue to stream

in tiles as they are consumed, until all required tiles have been accumulated.

A system like this would consume significant space on the processor die, as it would require a large amount of storage to function, but there are prefetchers in existence with history tables up to 18 KB in size [17]. More experimentation and analysis would be required to determine the optimal size of the buffers used in this proposed design.

CHAPTER 5

CONCLUSION

Prefetchers have evolved significantly since their inception, but the core principal is the same as it has always been: bring the data closer to the processor faster, and stay out of the way. Current prefetcher designs achieve the former, but fail to achieve the latter when used in conjunction with Intel's AMX instruction set, leading to marginal gains in performance. These gains could potentially be enhanced through the design of a highly specific hardware prefetcher that is designed with an understanding of the memory access patterns and cache limitations of the hardware which utilizes it, but without this advancement, it is unlikely that these prefetching schemes will result in significant improvements for these applications.

Appendices

APPENDIX A

EXPERIMENTAL EQUIPMENT AND SOFTWARE

The following software was used to build and test the pre-fetcher designs used in this work:

- MacSim
  - MacSim is a heterogeneous trace-driven cycle-accurate simulator used to model systems that contain multiple distinct processing units. In this case, it was used to model the pre-fetchers alongside a CPU model of Intel's Sapphire Rapids processor.
  - https://github.com/gthparch/macsim
- Intel Pin
  - Intel's Pin is a tool used to instrument software live as it runs on a computer system. In this case, it allows us to create a PinTool that we configured to instrument and print out traces that we use in MacSim.
  - https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
- Linux Perf
  - Linux's Perf tool is a program that allows users to generate performance metrics of programs when running them on a given computer system. In this case, it was used to generate performance metrics of software running natively to compare against our simulation results.
  - https://perf.wiki.kernel.org/index.php/Main_Page

APPENDIX B

DATA PROCESSING

Data was processed using Microsoft Excel before being added to this document. The following tables were the result of that data processing. In each row labeled "% Difference from Baseline," the baseline referred to is that shown in section 3.1. This percent difference is calculated as follows: $diff = (exp_val - base_val)/base_val$. Each row labeled "X %" is the calculation of Hits or Matches as a percentage of the total. For the caches, this is calculated as $hits/(hits + misses)$; for the prefetch queue requests, this is calculated as $matched_requests/sent_requests$. The benchmarks' naming scheme is described in section 3.1.

Table B.1: Experiment: 16-entry Prefetch Distance and 64 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16375821	87819461	593337991	4466102891	10175421	9918531	19387561	46455751	59985391	60212911	
% Difference from Baseline	-1.30143%	-0.26970%	-0.03586%	-0.22325%	-2.17635%	-2.36274%	-1.25331%	-0.45466%	-0.31260%	-0.47509%	-0.88650%
L1 Hit	333150	2173813	16246312	94465450	165160	159502	359197	953922	1252039	1229963	
L1 Miss	6432625	29567295	127091668	518606622	4621286	3924490	9536642	3706386	4138693	23841091	
L1 %	4.92405%	6.84857%	11.33427%	15.40854%	3.45058%	3.90554%	3.62978%	20.46908%	23.22577%	4.90591%	9.81021%
% Difference from Baseline	866.39773%	0.05935%	0.25850%	2.25062%	0.57615%	0.69451%	0.28496%	-0.06226%	-0.17717%	0.11006%	87.03924%
L2 Hit	38471	194520	315312	37083310	17369	13325	34991	86817	114100	69827	
L2 Miss	15422	57003	944653	3829914	9663	9399	17820	17277	22402	44749	
L2 %	71.38404%	77.33686%	25.02546%	90.63893%	64.25348%	58.63844%	66.25703%	83.40250%	83.58852%	60.94383%	68.14691%
% Difference from Baseline	10.94285%	9.62565%	13.80297%	-1.31581%	-0.55446%	18.69933%	16.43607%	-3.22745%	-4.56126%	14.77342%	7.46213%
L3 Hit	36	1972	674107	3503133	36	29	38	35	33	1001	
L3 Miss	14886	53218	206063	86288	9448	9213	17615	12045	14338	43225	
L3 %	0.24125%	3.57311%	76.58827%	97.59605%	0.37959%	0.31378%	0.21526%	0.28974%	0.22963%	2.26337%	18.16901%
% Difference from Baseline	2.71766%	5.28549%	4.24992%	-0.00018%	4.25981%	-10.24740%	12.58143%	-11.44066%	-17.76407%	33.18735%	2.28294%
Prefetch Queue Matched Requests	588	58054	352813	9108956	593	1778	5219	14738	19452	32975	
Prefetch Queue Sent Requests	74580	316173	961175	8234052	34864	28433	69044	138073	179355	112852	
Prefetch Queue %	0.78842%	18.36147%	36.70643%	110.62544%	1.70089%	6.25330%	7.55895%	10.67406%	10.84553%	29.21969%	23.27342%

Table B.2: Experiment: 16-entry Prefetch Distance and 128 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16374501	87803981	593342721	4467031731	10176521	9916001	19397021	46451021	59980001	60199271	
% Difference from Baseline	-1.30939%	-0.28728%	-0.03506%	-0.20250%	-2.16578%	-2.38764%	-1.20513%	-0.46479%	-0.32155%	-0.49763%	-0.88768%
L1 Hit	333176	2173812	16246299	94468772	165191	159501	359194	953932	1252064	1229982	
L1 Miss	6427703	29558860	127093403	519227857	4616289	3925617	9530409	3703990	4132950	23833346	
L1 %	4.92800%	6.85039%	11.33412%	15.39340%	3.45481%	3.90444%	3.63204%	20.47978%	23.25090%	4.90750%	9.81354%
% Difference from Baseline	867.17304%	0.08591%	0.25721%	2.15015%	0.69951%	0.66612%	0.34735%	-0.01002%	-0.06918%	0.14247%	87.14426%
L2 Hit	38326	194565	317226	37082980	17376	13420	35303	86031	114418	71697	
L2 Miss	15481	57030	943992	3830929	9737	9531	17903	18406	22088	45129	
L2 %	71.22865%	77.33262%	25.15235%	90.63661%	64.08734%	58.47240%	66.35154%	82.37598%	83.81903%	61.37076%	68.08273%
% Difference from Baseline	10.70135%	9.61963%	14.38003%	-1.31834%	-0.81159%	18.36321%	16.60215%	-4.41854%	-4.29807%	15.57744%	7.43973%
L3 Hit	37	1955	673638	3502304	38	32	39	31	33	1129	
L3 Miss	14911	53264	206112	88458	9509	9320	17653	12071	14377	43262	
L3 %	0.24752%	3.54045%	76.57153%	97.53651%	0.39803%	0.34217%	0.22044%	0.25616%	0.22901%	2.54331%	18.18851%
% Difference from Baseline	5.38730%	4.32304%	4.22712%	-0.06118%	9.32579%	-2.12754%	15.28940%	-21.70432%	-17.98664%	49.65995%	4.63329%
Prefetch Queue Matched Requests	711	58103	352280	8666589	588	1671	5231	14792	19577	32093	
Prefetch Queue Sent Requests	72540	316374	963842	8238831	34289	28856	69127	136574	180138	116182	
Prefetch Queue %	0.98015%	18.36529%	36.54956%	105.19197%	1.71484%	5.79082%	7.56723%	10.83076%	10.86778%	27.62304%	22.54814%

Table B.3: Experiment: 16-entry Prefetch Distance and 256 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16359761	87819131	593333701	4467207531	10187301	9934041	19398451	46455531	59989941	60204991	
% Difference from Baseline	-1.39822%	-0.27008%	-0.03658%	-0.19857%	-2.06214%	-2.21006%	-1.19784%	-0.45513%	-0.30503%	-0.48818%	-0.86218%
L1 Hit	333179	2173805	16246385	94466733	165197	159493	359206	953929	1252065	1229974	
L1 Miss	6423357	29561075	127090826	519317384	4619133	3924491	9531114	3704047	4136293	23836280	
L1 %	4.93121%	6.84989%	11.33438%	15.39087%	3.45288%	3.90533%	3.63189%	20.47947%	23.23649%	4.90689%	9.81193%
% Difference from Baseline	867.80344%	0.07862%	0.25949%	2.13338%	0.64318%	0.68902%	0.34343%	-0.01150%	-0.13111%	0.13013%	87.19381%
L2 Hit	33648	205474	319871	37085279	17371	13657	36048	86330	114167	74756	
L2 Miss	15539	58048	943756	3830043	9745	9502	17924	18132	22479	45433	
L2 %	68.40832%	77.97224%	25.31372%	90.63910%	64.06181%	58.97059%	66.79019%	82.64249%	83.54946%	62.19870%	68.05466%
% Difference from Baseline	6.31808%	10.52630%	15.11384%	-1.31563%	-0.85110%	19.37169%	17.37301%	-4.10930%	-4.60585%	17.13669%	7.49577%
L3 Hit	37	2427	673245	3502827	33	31	40	31	33	1365	
L3 Miss	14925	53265	206133	87245	9518	9289	17695	12075	14366	43289	
L3 %	0.24729%	4.35790%	76.55923%	97.56983%	0.34551%	0.33262%	0.22554%	0.25607%	0.22918%	3.05684%	18.31800%
% Difference from Baseline	5.28869%	28.41004%	4.21038%	-0.02705%	-5.09894%	-4.86052%	17.95884%	-21.73019%	-17.92399%	79.87833%	8.61056%
Prefetch Queue Matched Requests	646	41954	350440	8686041	647	1335	4205	14836	19533	30978	
Prefetch Queue Sent Requests	63704	330511	967914	8185821	33479	29068	69655	137133	179596	119221	
Prefetch Queue %	1.01407%	12.69368%	36.20570%	106.11081%	1.93255%	4.59268%	6.03690%	10.81869%	10.87608%	25.98368%	21.62648%

Table B.4: Experiment: 16-entry Prefetch Distance and 512 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	**16359761**	**87819131**	**593333701**	**4466569421**	**10187301**	**9934041**	**19398451**	**46455531**	**59990391**	**60206591**	
% Difference from Baseline	-1.39822%	-0.27008%	-0.03658%	-0.21283%	-2.06214%	-2.21006%	-1.19784%	-0.45513%	-0.30429%	-0.48553%	-0.86327%
L1 Hit	333179	2173805	16243827	94465725	165197	159493	359206	953903	1252047	1229968	
L1 Miss	6423357	29561045	127094958	519073908	4619133	3924491	9531114	3705148	4136752	23836750	
L1 %	4.93121%	6.84990%	11.33247%	15.39684%	3.45288%	3.90533%	3.63189%	20.47419%	23.23425%	4.90678%	9.81157%
% Difference from Baseline	867.80344%	0.07871%	0.24260%	2.17299%	0.64318%	0.68902%	0.34343%	-0.03729%	-0.14072%	0.12779%	87.19231%
L2 Hit	33183	208283	345168	37084079	17364	14030	36536	87685	116445	77967	
L2 Miss	15521	58110	1003079	3828577	9745	9497	17920	16143	20318	45507	
L2 %	68.13198%	78.18636%	25.60124%	90.64207%	64.05253%	59.63361%	67.09270%	84.45217%	85.14364%	63.14447%	68.60808%
% Difference from Baseline	5.88861%	10.82982%	16.42136%	-1.31239%	-0.86547%	20.71381%	17.90462%	-2.00952%	-2.78567%	18.91781%	8.37030%
L3 Hit	37	2508	724324	3502297	33	31	40	33	37	1383	
L3 Miss	14924	53265	206131	86712	9517	9281	17698	14174	15379	43292	
L3 %	0.24731%	4.49680%	77.84622%	97.58396%	0.34555%	0.33290%	0.22550%	0.23228%	0.24001%	3.09569%	18.46462%
% Difference from Baseline	5.29573%	32.50295%	5.96219%	-0.01257%	-5.08901%	-4.77878%	17.93889%	-29.00221%	-14.04628%	82.16469%	9.09356%
Prefetch Queue Matched Requests	617	32753	251475	8720546	582	891	2806	10991	18064	29684	
Prefetch Queue Sent Requests	61951	335431	1123040	8218044	33459	29346	69552	140507	184080	122588	
Prefetch Queue %	0.99595%	9.76445%	22.39235%	106.11462%	1.73944%	3.03619%	4.03439%	7.82239%	9.81312%	24.21444%	18.99273%

Table B.5: Experiment: 64-entry Prefetch Distance and 64 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16364051	87838491	593309831	4472709321	10190421	9948671	19433761	46482031	59965081	60272101	
% Difference from Baseline	-1.37237%	-0.24809%	-0.04060%	-0.07566%	-2.03215%	-2.06604%	-1.01800%	-0.39835%	-0.34635%	-0.37725%	-0.79749%
L1 Hit	333148	2173839	16231067	94111826	165208	159506	359224	953935	1252046	1229912	
L1 Miss	6422619	29557887	127094098	529125719	4609775	3921107	9533502	3689356	4120092	23836432	
L1 %	4.93131%	6.85068%	11.32465%	15.10047%	3.45987%	3.90887%	3.63119%	20.54437%	23.30629%	4.90663%	9.79643%
% Difference from Baseline	867.82354%	0.09013%	0.17337%	0.20631%	0.84690%	0.78041%	0.32405%	0.30536%	0.16890%	0.12472%	87.08437%
L2 Hit	37710	244743	295287	37139514	17398	12846	33958	87604	116056	76125	
L2 Miss	15405	55730	1087579	4179002	9835	9640	18007	14306	17263	45511	
L2 %	70.99689%	81.45258%	21.35326%	89.88589%	63.88573%	57.12888%	65.34783%	85.96212%	87.05136%	62.58427%	68.56488%
% Difference from Baseline	10.34116%	15.45971%	-2.89629%	-2.13570%	-1.12363%	15.64358%	14.83830%	-0.25751%	-0.60750%	17.86281%	6.71249%
L3 Hit	39	1926	794626	3795014	34	28	36	34	34	1206	
L3 Miss	14980	53364	206208	105130	9573	9389	17764	12315	14691	43368	
L3 %	0.25967%	3.48345%	79.39638%	97.30446%	0.35391%	0.29733%	0.20225%	0.27533%	0.23090%	2.70561%	18.45093%
% Difference from Baseline	10.55878%	2.64356%	8.07224%	-0.29895%	-2.79310%	-14.95271%	5.77528%	-15.84491%	-17.30900%	59.21071%	3.50619%
Prefetch Queue Matched Requests	473	632	68047	6929826	503	459	408	373	406	46762	
Prefetch Queue Sent Requests	83751	528206	1688905	17296279	40828	32839	83041	188535	247820	148015	
Prefetch Queue %	0.56477%	0.11965%	4.02906%	40.06542%	1.23200%	1.39773%	0.49132%	0.19784%	0.16383%	31.59274%	7.98544%

Table B.6: Experiment: 64-entry Prefetch Distance and 128 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16360641	87828591	593322921	4472690651	10188031	9936351	19419461	46474831	59951581	60267421	
% Difference from Baseline	-1.39292%	-0.25933%	-0.03840%	-0.07607%	-2.05512%	-2.18732%	-1.09083%	-0.41377%	-0.36878%	-0.38499%	-0.82675%
L1 Hit	333148	2173830	16230928	94111354	165215	159517	359215	953957	1252056	1229901	
L1 Miss	6417917	29557070	127098011	529323905	4612664	3920531	9536647	3686066	4120584	23831295	
L1 %	4.93475%	6.85083%	11.32425%	15.09561%	3.45792%	3.90968%	3.62995%	20.55932%	23.30430%	4.90759%	9.79742%
% Difference from Baseline	868.49761%	0.09232%	0.16988%	0.17403%	0.79004%	0.80132%	0.28975%	0.37832%	0.16034%	0.14439%	87.14980%
L2 Hit	34064	244627	294722	37137106	17247	12768	33555	87467	115904	84041	
L2 Miss	15718	55781	1088374	4172323	9894	9724	18090	14387	17335	46757	
L2 %	68.42634%	81.43159%	21.30886%	89.89983%	63.54593%	56.76685%	64.97241%	85.87488%	86.98955%	64.25251%	68.34687%
% Difference from Baseline	6.34609%	15.42995%	-3.09821%	-2.12052%	-1.64954%	14.91074%	14.17855%	-0.35874%	-0.67808%	21.00455%	6.39648%
L3 Hit	36	1930	795165	3791210	31	27	32	35	35	2043	
L3 Miss	15029	53423	206254	105095	9620	9440	17824	12387	14715	43413	
L3 %	0.23896%	3.48671%	79.40383%	97.30270%	0.32121%	0.28520%	0.17921%	0.28176%	0.23729%	4.49446%	18.62313%
% Difference from Baseline	1.74264%	2.73967%	8.08237%	-0.30075%	-11.77425%	-18.42325%	-6.27240%	-13.87885%	-15.02119%	164.47445%	11.13684%
Prefetch Queue Matched Requests	470	623	68293	6952250	425	447	392	368	369	33092	
Prefetch Queue Sent Requests	75736	528206	1688636	16960325	40808	32679	79027	188445	247629	171424	
Prefetch Queue %	0.62058%	0.11795%	4.04427%	40.99125%	1.04146%	1.36785%	0.49603%	0.19528%	0.14901%	19.30418%	6.83279%

Table B.7: Experiment: 64-entry Prefetch Distance and 256 Stream Buffers

Benchmark	512	1024	2048	4096	256-256-768	256-768-256	256-768-768	256-768-2304	256-768-3072	256-3072-768	Average
Cycles	16361191	87823641	593318411	4472697001	10191811	9936351	19413631	46473351	59957091	60265771	
% Difference from Baseline	-1.38961%	-0.26495%	-0.03916%	-0.07593%	-2.01879%	-2.18732%	-1.12053%	-0.41695%	-0.35963%	-0.38772%	-0.82606%
L1 Hit	333150	2173833	16226161	94114678	165217	159523	359220	953902	1252025	1229905	
L1 Miss	6418047	29557196	127101536	529073116	4614677	3919636	9535110	3687255	4120575	23830424	
L1 %	4.93468%	6.85081%	11.32102%	15.10214%	3.45650%	3.91068%	3.63056%	20.55311%	23.30389%	4.90778%	9.79712%
% Difference from Baseline	868.48449%	0.09206%	0.14133%	0.21735%	0.74877%	0.82708%	0.30667%	0.34801%	0.15861%	0.14818%	87.14725%
L2 Hit	33172	240726	358814	37134587	17162	12607	33943	77290	109976	93664	
L2 Miss	15708	57149	1049360	4174253	9937	9740	18099	19743	20538	47363	
L2 %	67.86416%	80.81444%	25.48080%	89.89501%	63.33075%	56.41473%	65.22232%	79.65331%	84.26376%	66.41565%	67.93549%
% Difference from Baseline	5.47236%	14.55514%	15.87364%	-2.12576%	-1.98257%	14.19796%	14.61773%	-7.57766%	-3.79029%	25.07832%	7.43189%
L3 Hit	37	2669	758722	3792882	31	30	31	37	42	2397	
L3 Miss	15032	53435	206292	105284	9657	9457	17836	16029	16916	43424	
L3 %	0.24554%	4.75724%	78.62290%	97.29914%	0.31998%	0.31622%	0.17350%	0.23030%	0.24767%	5.23123%	18.74437%
% Difference from Baseline	4.54107%	40.17700%	7.01940%	-0.30440%	-12.11120%	-9.55025%	-9.25729%	-29.60736%	-11.30292%	207.82936%	18.74334%
Prefetch Queue Matched Requests	423	3009	66785	6966817	445	413	425	1553	1018	23067	
Prefetch Queue Sent Requests	73525	489493	1744433	16953222	40550	32520	79104	161453	232002	191947	
Prefetch Queue %	0.57531%	0.61472%	3.82846%	41.09435%	1.09741%	1.26999%	0.53727%	0.96189%	0.43879%	12.01738%	6.24356%

Table B.8: Experiment: 64-entry Prefetch Distance and 512 Stream Buffers

Benchmark	512	1024	2048	4096	256-256-768	256-768-256	256-768-768	256-768-2304	256-768-3072	256-3072-768	Average
Cycles	16361191	87823641	593319291	4472705611	10191811	9936351	19413631	46473351	59957941	60266261	
% Difference from Baseline	-1.38961%	-0.26495%	-0.03901%	-0.07574%	-2.01879%	-2.18732%	-1.12053%	-0.41695%	-0.35821%	-0.38691%	-0.82580%
L1 Hit	333150	2173833	16221484	94118035	165217	159523	359220	953855	1251923	1229905	
L1 Miss	6418034	29557073	127106525	529141633	4614677	3919646	9535110	3687254	4121877	23830583	
L1 %	4.93469%	6.85084%	11.31773%	15.10093%	3.45650%	3.91067%	3.63056%	20.55231%	23.29679%	4.90775%	9.79588%
% Difference from Baseline	868.48636%	0.09244%	0.11224%	0.20936%	0.74877%	0.82683%	0.30667%	0.34410%	0.12808%	0.14755%	87.14024%
L2 Hit	32343	227826	392104	37134276	17176	12768	33872	72100	96307	100572	
L2 Miss	15783	57934	1045222	4182621	9937	9753	18100	23152	26924	48295	
L2 %	67.20484%	79.72634%	27.28010%	89.87673%	63.34968%	56.69375%	65.17355%	75.69395%	78.15160%	67.55829%	67.07088%
% Difference from Baseline	4.44767%	13.01276%	24.05594%	-2.14567%	-1.95326%	14.76277%	14.53203%	-12.17175%	-10.76896%	27.23021%	7.10018%
L3 Hit	37	2691	750159	3797008	31	30	31	37	71	3008	
L3 Miss	15027	53438	206285	106837	9654	9461	17834	18228	21544	43433	
L3 %	0.24562%	4.79431%	78.43209%	97.26329%	0.32008%	0.31609%	0.17352%	0.20257%	0.32848%	6.47704%	18.85531%
% Difference from Baseline	4.57577%	41.26950%	6.75967%	-0.34114%	-12.08398%	-9.58837%	-9.24713%	-38.08223%	17.63532%	281.13852%	28.20359%
Prefetch Queue Matched Requests	371	1663	32899	6936816	394	369	403	1025	3058	14303	
Prefetch Queue Sent Requests	71057	448311	1791940	17172530	40582	32617	78375	149120	195339	207407	
Prefetch Queue %	0.52212%	0.37095%	1.83594%	40.39484%	0.97087%	1.13131%	0.51419%	0.68737%	1.56548%	6.89610%	5.48892%

Table B.9: Experiment: 256-entry Prefetch Distance and 64 Stream Buffers

Benchmark	512	1024	2048	4096	256-256-768	256-768-256	256-768-768	256-768-2304	256-768-3072	256-3072-768	Average
Cycles	16345131	87813521	593321601	4471908371	10175131	9916991	19405271	46437961	59962911	60252181	
% Difference from Baseline	-1.48640%	-0.27645%	-0.03862%	-0.09355%	-2.17914%	-2.37790%	-1.16311%	-0.49278%	-0.34995%	-0.41018%	-0.88681%
L1 Hit	333138	2173886	16232194	94115473	165230	159549	359249	953951	1252094	1229943	
L1 Miss	6427469	29552204	127008765	527805790	4611163	3914667	9533924	3691129	4126426	23834236	
L1 %	4.92763%	6.85205%	11.33209%	15.13302%	3.45930%	3.91607%	3.63128%	20.53680%	23.27953%	4.90717%	9.79749%
% Difference from Baseline	867.10164%	0.11008%	0.23922%	0.42229%	0.83055%	0.96586%	0.32650%	0.26841%	0.05388%	0.13589%	87.04543%
L2 Hit	37946	244761	324429	37125565	17256	12608	33764	87635	115825	104770	
L2 Miss	15888	55926	1088098	4651861	10151	10098	18494	14630	17526	49411	
L2 %	70.48705%	81.40059%	22.96799%	88.86513%	62.96202%	55.52717%	64.61020%	85.69403%	86.85724%	67.95260%	68.73240%
% Difference from Baseline	9.54879%	15.38602%	4.44665%	-3.24706%	-2.55326%	12.40132%	13.54203%	-0.56858%	-0.82914%	27.97280%	7.60996%
L3 Hit	37	1925	788668	4139694	34	23	33	36	37	3781	
L3 Miss	15295	53611	206464	118939	9832	9711	18127	12492	14957	43675	
L3 %	0.24133%	3.46622%	79.25260%	97.20711%	0.34462%	0.23629%	0.18172%	0.28736%	0.24677%	7.96738%	18.94314%
% Difference from Baseline	2.74781%	2.13584%	7.87653%	-0.39870%	-5.34496%	-32.41481%	-4.96145%	-12.16774%	-11.62715%	368.83726%	31.46826%
Prefetch Queue Matched Requests	325	504	584	537776	649	264	291	206	237	483	
Prefetch Queue Sent Requests	90200	538743	1856293	28700118	44946	35683	85216	191197	250965	260179	
Prefetch Queue %	0.36031%	0.09355%	0.03146%	1.87378%	1.44395%	0.73985%	0.34149%	0.10774%	0.09444%	0.18564%	0.52722%

Table B.10: Experiment: 256-entry Prefetch Distance and 128 Stream Buffers

Benchmark	512	1024	2048	4096	256-256-768	256-768-256	256-768-768	256-768-2304	256-768-3072	256-3072-768	Average
Cycles	16343041	87814511	593309501	4471646241	10162111	9919521	19390751	46428131	59956681	60264951	
% Difference from Baseline	-1.49900%	-0.27532%	-0.04066%	-0.09941%	-2.30431%	-2.35299%	-1.23706%	-0.51384%	-0.36031%	-0.38907%	-0.90720%
L1 Hit	333141	2173862	16231936	94124491	165226	159550	359253	953950	1252091	1229937	
L1 Miss	6426254	29564633	127095210	527392462	4612044	3915311	9530884	3690912	4125535	23834947	
L1 %	4.92856%	6.84929%	11.32510%	15.14432%	3.45859%	3.91547%	3.63244%	20.53775%	23.28334%	4.90701%	9.79819%
% Difference from Baseline	867.28375%	0.06984%	0.17735%	0.49724%	0.80960%	0.95051%	0.35842%	0.27301%	0.07027%	0.13259%	87.06226%
L2 Hit	37961	236097	302469	37140352	17159	12542	33725	87403	115773	98398	
L2 Miss	15922	58156	1102922	4637123	10245	10106	18564	14795	17559	52874	
L2 %	70.45079%	80.23606%	21.52205%	88.90042%	62.61495%	55.37796%	64.49731%	85.52320%	86.83062%	65.04707%	68.10004%
% Difference from Baseline	9.49243%	13.73528%	-2.12871%	-3.20863%	-3.09042%	12.09927%	13.34365%	-0.76679%	-0.85953%	22.50091%	6.11174%
L3 Hit	37	2902	800793	4125088	36	24	33	37	39	5690	
L3 Miss	15376	53951	206511	119979	9885	9756	18177	12599	15011	43687	
L3 %	0.24006%	5.10439%	79.49864%	97.17368%	0.36287%	0.24540%	0.18122%	0.29281%	0.25914%	11.52358%	19.48818%
% Difference from Baseline	2.20784%	50.40630%	8.21143%	-0.43295%	-0.33263%	-29.80803%	-5.22241%	-10.49952%	-7.19684%	578.10062%	58.54338%
Prefetch Queue Matched Requests	321	2637	771	545812	510	211	204	171	181	706	
Prefetch Queue Sent Requests	90465	507466	1836973	28597504	45026	36027	85696	191080	251101	243969	
Prefetch Queue %	0.35483%	0.51964%	0.04197%	1.90860%	1.13268%	0.58567%	0.23805%	0.08949%	0.07208%	0.28938%	0.52324%

Table B.11: Experiment: 256-entry Prefetch Distance and 256 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16340331	87814181	593310821	4471840061	10156941	9917341	19392841	46426821	59965361	60257901	
% Difference from Baseline	-1.51533%	-0.27570%	-0.04044%	-0.09508%	-2.35402%	-2.37445%	-1.22642%	-0.51665%	-0.34588%	-0.40073%	-0.91447%
L1 Hit	333142	2173871	16207611	94112540	165227	159553	359251	953837	1251924	1229956	
L1 Miss	6425792	29560384	126812184	527590918	4610251	3915331	9531231	3693140	4131439	23835340	
L1 %	4.92891%	6.85023%	11.33242%	15.13785%	3.45990%	3.91552%	3.63229%	20.52597%	23.25543%	4.90701%	9.79455%
% Difference from Baseline	867.35263%	0.08363%	0.24219%	0.45434%	0.84804%	0.95184%	0.35436%	0.21550%	-0.04971%	0.13249%	87.05853%
L2 Hit	37998	235411	389513	37129104	17149	12627	33866	67546	89665	96096	
L2 Miss	15956	58103	1059994	4650573	10258	10068	18475	24425	30656	54422	
L2 %	70.42666%	80.20435%	26.87210%	88.86882%	62.57161%	55.63781%	64.70262%	73.44272%	74.52149%	63.84353%	66.10917%
% Difference from Baseline	9.45492%	13.69034%	22.20057%	-3.24305%	-3.15750%	12.62526%	13.70445%	-14.78387%	-14.91371%	20.23433%	5.58117%
L3 Hit	37	2942	749864	4139571	36	24	33	46	151	6581	
L3 Miss	15352	53938	206576	118400	9887	9713	18126	19790	24427	43732	
L3 %	0.24043%	5.17229%	78.40157%	97.21933%	0.36279%	0.24648%	0.18173%	0.23190%	0.61437%	13.08012%	19.57510%
% Difference from Baseline	2.36724%	52.40706%	6.71813%	-0.38617%	-0.35272%	-29.49806%	-4.95622%	-29.11783%	120.02146%	669.69427%	78.68972%
Prefetch Queue Matched Requests	298	3380	2932	546363	398	160	147	468	349	1391	
Prefetch Queue Sent Requests	90293	504946	2002892	28590939	45469	36178	85743	163733	216174	238833	
Prefetch Queue %	0.33004%	0.66938%	0.14639%	1.91097%	0.87532%	0.44226%	0.17144%	0.28583%	0.16144%	0.58242%	0.55755%

Table B.12: Experiment: 256-entry Prefetch Distance and 512 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16340331	87813961	593310841	4472224121	10156941	9917341	19392841	46426711	59963671	60258891	
% Difference from Baseline	-1.51533%	-0.27595%	-0.04043%	-0.08650%	-2.35402%	-2.37445%	-1.22642%	-0.51689%	-0.34869%	-0.39909%	-0.91378%
L1 Hit	333143	2173874	16205320	94040356	165227	159553	359251	953857	1251952	1229955	
L1 Miss	6426104	29560296	126530923	527730009	4610251	3915331	9531231	3696955	4136429	23835345	
L1 %	4.92870%	6.85026%	11.35333%	15.12461%	3.45990%	3.91552%	3.63229%	20.50947%	23.23429%	4.90700%	9.79154%
% Difference from Baseline	867.31074%	0.08404%	0.42713%	0.36649%	0.84804%	0.95184%	0.35436%	0.13497%	-0.14055%	0.13239%	87.04694%
L2 Hit	38080	236266	386960	37055918	17149	12627	33859	68341	90463	96533	
L2 Miss	16037	58218	1063253	4699694	10256	10068	18472	24182	30278	54343	
L2 %	70.36606%	80.23050%	26.68298%	88.74476%	62.57617%	55.63781%	64.70161%	73.86380%	74.92318%	63.98168%	66.17085%
% Difference from Baseline	9.36074%	13.72741%	21.34052%	-3.37811%	-3.15043%	12.62526%	13.70267%	-14.29529%	-14.45507%	20.49451%	5.59722%
L3 Hit	37	2945	750568	4129808	36	24	33	44	186	6600	
L3 Miss	15366	53903	206609	189922	9883	9713	18129	19417	23936	43738	
L3 %	0.24021%	5.18048%	78.41476%	95.60338%	0.36294%	0.24648%	0.18170%	0.22609%	0.77108%	13.11137%	19.43385%
% Difference from Baseline	2.27419%	52.64835%	6.73608%	-2.04192%	-0.31253%	-29.49806%	-4.97192%	-30.89320%	176.14315%	671.53308%	84.16172%
Prefetch Queue Matched Requests	262	2696	7352	569777	380	160	115	187	190	2109	
Prefetch Queue Sent Requests	90267	505333	2030048	28376604	45485	36178	85749	164037	215633	237912	
Prefetch Queue %	0.29025%	0.53351%	0.36216%	2.00791%	0.83544%	0.44226%	0.13411%	0.11400%	0.08811%	0.88646%	0.56942%

Table B.13: Experiment: 1024-entry Prefetch Distance and 64 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16399011	87855241	593344921	4471111061	10209701	9969241	19437831	46493641	60010181	60306631	
% Difference from Baseline	-1.16166%	-0.22907%	-0.03469%	-0.11136%	-1.84680%	-1.86355%	-0.99727%	-0.37347%	-0.27140%	-0.32018%	-0.72094%
L1 Hit	333098	2173847	16232131	94122459	165203	159522	359185	953939	1252076	1229893	
L1 Miss	6466689	29556862	127013166	527419684	4610380	3921407	9538147	3697276	4131635	23838305	
L1 %	4.89865%	6.85092%	11.33170%	15.14338%	3.45933%	3.90896%	3.62911%	20.50946%	23.25675%	4.90619%	9.78944%
% Difference from Baseline	861.41380%	0.09371%	0.23580%	0.49100%	0.83118%	0.78272%	0.26648%	0.13490%	-0.04403%	0.11577%	86.43213%
L2 Hit	35064	244656	325991	37117080	17096	11595	33110	87464	115878	96082	
L2 Miss	17975	56032	1087242	4656885	10134	10615	18842	14814	17744	54669	
L2 %	66.10984%	81.36540%	23.06704%	88.85218%	62.78369%	52.20621%	63.73191%	85.51595%	86.72075%	63.73556%	67.40885%
% Difference from Baseline	2.74586%	15.33614%	4.89709%	-3.26116%	-2.82925%	5.67884%	11.99857%	-0.77521%	-0.98498%	20.03100%	5.28369%
L3 Hit	34	1915	788145	4141285	32	27	31	33	37	6881	
L3 Miss	15952	53687	206506	118028	9837	10157	18406	12633	14987	44004	
L3 %	0.21269%	3.44412%	79.23835%	97.22894%	0.32425%	0.26512%	0.16814%	0.26054%	0.24627%	13.52265%	19.49111%
% Difference from Baseline	-9.44577%	1.48465%	7.85713%	-0.37633%	-10.93998%	-24.16662%	-12.06270%	-20.36431%	-11.80361%	695.73480%	61.59173%
Prefetch Queue Matched Requests	650	507	552	236997	586	459	224	169	225	541	
Prefetch Queue Sent Requests	89305	541324	1861269	29960705	47727	37609	88614	192364	252258	252423	
Prefetch Queue %	0.72784%	0.09366%	0.02966%	0.79103%	1.22782%	1.22045%	0.25278%	0.08785%	0.08919%	0.21432%	0.47346%

Table B.14: Experiment: 1024-entry Prefetch Distance and 128 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16393091	87868441	593352731	4471103411	10192911	9951641	19442971	46487971	60007321	60294751	
% Difference from Baseline	-1.19734%	-0.21408%	-0.03338%	-0.11153%	-2.00821%	-2.03681%	-0.97109%	-0.38562%	-0.27615%	-0.33982%	-0.75740%
L1 Hit	333111	2173780	16231901	94119628	165206	159521	359183	953945	1252081	1229898	
L1 Miss	6466634	29600970	127098971	527183879	4608050	3920193	9539130	3695680	4129145	23838604	
L1 %	4.89887%	6.84122%	11.32478%	15.14874%	3.46108%	3.91010%	3.62873%	20.51660%	23.26758%	4.90615%	9.79038%
% Difference from Baseline	861.45726%	-0.04811%	0.17453%	0.52657%	0.88216%	0.81210%	0.25598%	0.16977%	0.00253%	0.11496%	86.43478%
L2 Hit	35166	233365	303786	37139679	16948	11772	32980	87186	115632	94693	
L2 Miss	17652	61597	1101880	4637418	10287	10536	18891	14763	17626	55396	
L2 %	66.57958%	79.11697%	21.61154%	88.89962%	62.22875%	52.77031%	63.58081%	85.51923%	86.77303%	63.09123%	67.01710%
% Difference from Baseline	3.47591%	12.14897%	-1.72180%	-3.20951%	-3.68814%	6.82071%	11.73303%	-0.77140%	-0.92529%	18.81756%	4.26800%
L3 Hit	34	3498	799994	4127008	35	26	33	33	37	6995	
L3 Miss	15971	55477	206545	115454	9956	10082	18473	12749	15060	44311	
L3 %	0.21243%	5.93133%	79.47968%	97.27861%	0.35032%	0.25722%	0.17832%	0.25818%	0.24508%	13.63388%	19.78250%
% Difference from Baseline	-9.55326%	74.77281%	8.18563%	-0.32544%	-3.78007%	-26.42621%	-6.73836%	-21.08703%	-12.23008%	702.28032%	70.50983%
Prefetch Queue Matched Requests	550	1126	774	221411	241	172	144	203	212	330	
Prefetch Queue Sent Requests	89925	524099	1840689	28887499	48242	38274	88849	192084	252047	250923	
Prefetch Queue %	0.61162%	0.21484%	0.04205%	0.76646%	0.49956%	0.44939%	0.16207%	0.10568%	0.08411%	0.13151%	0.30673%

Table B.15: Experiment: 1024-entry Prefetch Distance and 256 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16395401	87868661	593354491	4471517961	10192911	9949461	19442861	46491111	60014051	60294421	
% Difference from Baseline	-1.18342%	-0.21383%	-0.03308%	-0.10227%	-2.00821%	-2.05827%	-0.97165%	-0.37889%	-0.26497%	-0.34036%	-0.75549%
L1 Hit	333099	2173793	16215724	94107764	165206	159531	359187	953844	1251947	1229893	
L1 Miss	6467260	29593195	125464573	527707388	4608050	3921101	9539234	3695873	4131515	23837345	
L1 %	4.89826%	6.84293%	11.44529%	15.13436%	3.46108%	3.90947%	3.62873%	20.51402%	23.25543%	4.90638%	9.79959%
% Difference from Baseline	861.33582%	-0.02308%	1.24057%	0.43119%	0.88216%	0.79574%	0.25601%	0.15718%	-0.04971%	0.11960%	86.51455%
L2 Hit	35201	236037	403135	37132881	16948	11818	33407	67258	89221	95407	
L2 Miss	17805	60248	1048572	4646785	10289	10484	18693	24941	31201	55012	
L2 %	66.40946%	79.66552%	27.76972%	88.87788%	62.22418%	52.99076%	64.12092%	72.94873%	74.09028%	63.42749%	65.25250%
% Difference from Baseline	3.21152%	12.92655%	26.28249%	-3.23318%	-3.69521%	7.26697%	12.68220%	-15.35704%	-15.40605%	19.45082%	4.41291%
L3 Hit	34	3080	737055	4141764	35	26	33	37	128	7046	
L3 Miss	15992	55110	207673	111656	9956	10088	18313	20159	24968	44025	
L3 %	0.21216%	5.29301%	78.01769%	97.37491%	0.35032%	0.25707%	0.17988%	0.18320%	0.51004%	13.79648%	19.61748%
% Difference from Baseline	-9.67178%	55.96400%	6.19561%	-0.22676%	-3.78007%	-26.46986%	-5.92500%	-44.00237%	82.65859%	711.84824%	76.65906%
Prefetch Queue Matched Requests	342	663	5903	190504	171	157	120	264	236	339	
Prefetch Queue Sent Requests	89769	531295	2438097	27564288	48415	38306	89344	169572	222825	252327	
Prefetch Queue %	0.38098%	0.12479%	0.24212%	0.69113%	0.35320%	0.40986%	0.13431%	0.15569%	0.10591%	0.13435%	0.27323%

Table B.16: Experiment: 1024-entry Prefetch Distance and 512 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16395631	87869101	593354821	4472716761	10192911	9949461	19442861	46491111	60014051	60294421	
% Difference from Baseline	-1.18203%	-0.21333%	-0.03302%	-0.07549%	-2.00821%	-2.05827%	-0.97165%	-0.37889%	-0.26497%	-0.34036%	-0.75262%
L1 Hit	333099	2173787	16218086	93992522	165206	159531	359187	953844	1251947	1229893	
L1 Miss	6468048	29598553	125275570	525021476	4608050	3921101	9539234	3695873	4131515	23837345	
L1 %	4.89769%	6.84176%	11.46206%	15.18423%	3.46108%	3.90947%	3.62873%	20.51402%	23.25543%	4.90638%	9.80608%
% Difference from Baseline	861.22443%	-0.04020%	1.38888%	0.76212%	0.88216%	0.79574%	0.25601%	0.15718%	-0.04971%	0.11960%	86.54962%
L2 Hit	35167	233114	407971	37015001	16948	11818	33407	67258	89221	95407	
L2 Miss	17792	60697	1048426	4770346	10289	10484	18693	24941	31201	55012	
L2 %	66.40420%	79.34148%	28.01235%	88.58369%	62.22418%	52.99076%	64.12092%	72.94873%	74.09028%	63.42749%	65.21441%
% Difference from Baseline	3.20334%	12.46721%	27.38583%	-3.55348%	-3.69521%	7.26697%	12.68220%	-15.35704%	-15.40605%	19.45082%	4.44446%
L3 Hit	34	2955	734891	4093006	35	26	33	37	128	7046	
L3 Miss	16001	55444	207533	277770	9956	10088	18313	20159	24968	44025	
L3 %	0.21204%	5.06002%	77.97881%	93.64484%	0.35032%	0.25707%	0.17988%	0.18320%	0.51004%	13.79648%	19.21727%
% Difference from Baseline	-9.72248%	49.09878%	6.14268%	-4.04871%	-3.78007%	-26.46986%	-5.92500%	-44.00237%	82.65859%	711.84824%	75.57998%
Prefetch Queue Matched Requests	330	563	3094	174280	171	157	120	264	236	339	
Prefetch Queue Sent Requests	89782	517382	2445204	27912721	48415	38306	89344	169572	222825	252327	
Prefetch Queue %	0.36756%	0.10882%	0.12653%	0.62437%	0.35320%	0.40986%	0.13431%	0.15569%	0.10591%	0.13435%	0.25206%

Table B.17: Experiment: 2048-entry Prefetch Distance and 64 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16400461	87855241	593344921	4471152411	10209701	9969241	19437831	46493641	60010181	60308501	
% Difference from Baseline	-1.15292%	-0.22907%	-0.03469%	-0.11044%	-1.84680%	-1.86355%	-0.99727%	-0.37347%	-0.27140%	-0.31709%	-0.71967%
L1 Hit	333084	2173847	16232131	94122690	165203	159523	359184	953939	1252076	1229887	
L1 Miss	6498708	29556862	127013166	527398776	4610380	3921406	9538148	3697276	4131635	23839073	
L1 %	4.87550%	6.85092%	11.33170%	15.14392%	3.45933%	3.90899%	3.62910%	20.50946%	23.25675%	4.90602%	9.78717%
% Difference from Baseline	856.86963%	0.09371%	0.23580%	0.49459%	0.83118%	0.78335%	0.26620%	0.13490%	-0.04403%	0.11224%	85.97776%
L2 Hit	32698	244656	325991	37115659	17082	11152	32227	87464	115878	94913	
L2 Miss	19608	56032	1087242	4659207	10136	10794	19453	14814	17744	55189	
L2 %	62.51290%	81.36540%	23.06704%	88.84687%	62.75994%	50.81564%	62.35875%	85.51595%	86.72075%	63.23234%	66.71956%
% Difference from Baseline	-2.84439%	15.33614%	4.89709%	-3.26695%	-2.86602%	2.86395%	9.58546%	-0.77521%	-0.98498%	19.08329%	4.10284%
L3 Hit	34	1915	788145	4141429	32	27	31	33	37	6652	
L3 Miss	16511	53687	206506	118915	9837	10314	18833	12633	14987	44527	
L3 %	0.20550%	3.44412%	79.23835%	97.20879%	0.32425%	0.26110%	0.16433%	0.26054%	0.24627%	12.99752%	19.43508%
% Difference from Baseline	-12.50529%	1.48465%	7.85713%	-0.39697%	-10.93998%	-25.31795%	-14.05322%	-20.36431%	-11.80361%	664.83371%	57.87941%
Prefetch Queue Matched Requests	659	507	552	201495	296	210	226	169	225	280	
Prefetch Queue Sent Requests	89438	544396	1864341	30421784	50171	39483	91032	193388	253282	253661	
Prefetch Queue %	0.73682%	0.09313%	0.02961%	0.66234%	0.58998%	0.53187%	0.24826%	0.08739%	0.08883%	0.11038%	0.31786%

Table B.18: Experiment: 2048-entry Prefetch Distance and 128 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16395221	87865581	593352731	4471100461	10192801	9953841	19442971	46487971	60006991	60302011	
% Difference from Baseline	-1.18450%	-0.21733%	-0.03338%	-0.11160%	-2.00927%	-2.01515%	-0.97109%	-0.38562%	-0.27670%	-0.32782%	-0.75324%
L1 Hit	333097	2173758	16231904	94124186	165207	159518	359179	953945	1252082	1229886	
L1 Miss	6502476	29649599	127098953	527160130	4607900	3920232	9539157	3695680	4128952	23837822	
L1 %	4.87299%	6.83070%	11.32478%	15.14994%	3.46120%	3.90999%	3.62868%	20.51660%	23.26843%	4.90626%	9.78696%
% Difference from Baseline	856.37768%	-0.20178%	0.17456%	0.53455%	0.88592%	0.80932%	0.25463%	0.16977%	0.00617%	0.11716%	85.91280%
L2 Hit	32600	229262	303134	37139189	16938	10997	32185	87186	115611	94336	
L2 Miss	19736	64668	1102401	4639277	10300	10912	19349	14763	17665	55495	
L2 %	62.28982%	77.99884%	21.56716%	88.89553%	62.18518%	50.19398%	62.45391%	85.51923%	86.74555%	62.96160%	66.08108%
% Difference from Baseline	-3.19110%	10.56401%	-1.92359%	-3.21396%	-3.75557%	1.60557%	9.75270%	-0.77140%	-0.95666%	18.57343%	2.66834%
L3 Hit	34	3782	800296	4126459	35	26	33	33	37	6892	
L3 Miss	16597	57036	206552	115772	9959	10396	18831	12749	15068	44555	
L3 %	0.20444%	6.21855%	79.48528%	97.27096%	0.35021%	0.24947%	0.17494%	0.25818%	0.24495%	13.39631%	19.78533%
% Difference from Baseline	-12.95773%	83.23625%	8.19325%	-0.33327%	-3.80895%	-28.64288%	-8.50827%	-21.08703%	-12.27656%	688.30048%	69.21153%
Prefetch Queue Matched Requests	508	771	742	194399	326	166	156	203	204	275	
Prefetch Queue Sent Requests	89608	524938	1843081	29241386	50337	39528	91237	193108	253055	253346	
Prefetch Queue %	0.56691%	0.14687%	0.04026%	0.66481%	0.64763%	0.41996%	0.17098%	0.10512%	0.08061%	0.10855%	0.29517%

Table B.19: Experiment: 2048-entry Prefetch Distance and 256 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16394091	87866351	593356691	4471414371	10192801	9951311	19443101	46491171	60015041	60297831	
% Difference from Baseline	-1.19131%	-0.21645%	-0.03271%	-0.10459%	-2.00927%	-2.04005%	-0.97043%	-0.37876%	-0.26332%	-0.33473%	-0.75416%
L1 Hit	333101	2173771	16215957	94116028	165207	159524	359185	953854	1251941	1229886	
L1 Miss	6483322	29629081	125341387	527486437	4607900	3920201	9539245	3695982	4133435	23838168	
L1 %	4.88674%	6.83514%	11.45540%	15.14087%	3.46120%	3.91017%	3.62871%	20.51371%	23.24705%	4.90619%	9.79852%
% Difference from Baseline	859.07603%	-0.13684%	1.32996%	0.47438%	0.88592%	0.81373%	0.25536%	0.15567%	-0.08571%	0.11577%	86.28843%
L2 Hit	34109	232190	401709	37121612	16936	11500	33091	67214	89098	95318	
L2 Miss	18373	63119	1051722	4647032	10306	10643	18850	24990	31312	54988	
L2 %	64.99181%	78.62612%	27.63867%	88.87435%	62.16871%	51.93515%	63.70882%	72.89705%	73.99552%	63.41596%	64.82522%
% Difference from Baseline	1.00824%	11.45318%	25.68653%	-3.23702%	-3.78106%	5.13013%	11.95800%	-15.41700%	-15.51425%	19.42911%	3.67159%
L3 Hit	34	3321	737023	4141168	35	26	33	37	141	6745	
L3 Miss	16291	56525	209145	112250	9961	10200	18450	20188	24845	44321	
L3 %	0.20827%	5.54924%	77.89557%	97.36095%	0.35014%	0.25425%	0.17854%	0.18294%	0.56432%	13.20840%	19.57526%
% Difference from Baseline	-11.32619%	63.51431%	6.02938%	-0.24107%	-3.82820%	-27.27519%	-6.62230%	-44.08266%	102.09567%	677.24277%	75.55065%
Prefetch Queue Matched Requests	350	472	3748	177366	166	158	116	260	359	396	
Prefetch Queue Sent Requests	92059	531553	2457308	27821746	50528	40088	92176	170982	223833	254140	
Prefetch Queue %	0.38019%	0.08880%	0.15252%	0.63751%	0.32853%	0.39413%	0.12585%	0.15206%	0.16039%	0.15582%	0.25758%

Table B.20: Experiment: 2048-entry Prefetch Distance and 512 Stream Buffers

Benchmark	**512**	**1024**	**2048**	**4096**	**256-256-768**	**256-768-256**	**256-768-768**	**256-768-2304**	**256-768-3072**	**256-3072-768**	**Average**
Cycles	16394451	87866461	593354601	4472726991	10192801	9951311	19443101	46491961	60014851	60297831	
% Difference from Baseline	-1.18915%	-0.21633%	-0.03306%	-0.07526%	-2.00927%	-2.04005%	-0.97043%	-0.37707%	-0.26364%	-0.33473%	-0.75090%
L1 Hit	333097	2173738	16216049	93998692	165207	159524	359185	953866	1251935	1229886	
L1 Miss	6490544	29652474	125301604	524327962	4607900	3920201	9539245	3695607	4132685	23838156	
L1 %	4.88151%	6.83002%	11.45868%	15.20211%	3.46120%	3.91017%	3.62871%	20.51557%	23.25020%	4.90619%	9.80444%
% Difference from Baseline	858.05002%	-0.21165%	1.35895%	0.88075%	0.88592%	0.81373%	0.25536%	0.16475%	-0.07216%	0.11582%	86.22415%
L2 Hit	33694	228292	406154	37016935	16936	11500	33091	67127	89011	95369	
L2 Miss	18796	64195	1048852	4779423	10306	10643	18850	25151	31489	54957	
L2 %	64.19127%	78.05202%	27.91425%	88.56498%	62.16871%	51.93515%	63.70882%	72.74432%	73.86805%	63.44145%	64.65890%
% Difference from Baseline	-0.23592%	10.63938%	26.93972%	-3.57386%	-3.78106%	5.13013%	11.95800%	-15.59423%	-15.65979%	19.47712%	3.52995%
L3 Hit	34	3214	735513	4115579	35	26	33	37	160	6690	
L3 Miss	16408	57207	208655	254737	9961	10200	18450	20292	24955	44311	
L3 %	0.20679%	5.31934%	77.90065%	94.17120%	0.35014%	0.25425%	0.17854%	0.18201%	0.63707%	13.11739%	19.23174%
% Difference from Baseline	-11.95718%	56.74005%	6.03629%	-3.50938%	-3.82820%	-27.27519%	-6.62230%	-44.36873%	128.15051%	671.88749%	76.52533%
Prefetch Queue Matched Requests	384	568	1758	173125	166	158	116	173	161	396	
Prefetch Queue Sent Requests	91278	515532	2464960	28489178	50528	40088	92176	171014	223708	254141	
Prefetch Queue %	0.42069%	0.11018%	0.07132%	0.60769%	0.32853%	0.39413%	0.12585%	0.10116%	0.07197%	0.15582%	0.23873%

REFERENCES

[1] A. Peleg and U. Weiser, "Mmx technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, 1996.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. USA: Intel Corporation, 2024.

[3] *Single instruction, multiple data*, Wikimedia Foundation, Jun. 2024.

[4] R. Cherukuri, *What is int8 quantization and why is it popular for deep neural networks?* MathWorks, 2019.

[5] D. Narayanan *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021, ISBN: 9781450384421.

[6] M. Isaev, N. McDonald, and R. Vuduc, "Scaling infrastructure to support multi-trillion parameter llm training," *Proceedings of the International Symposium on Computer Architecture*, 2023.

[7] *Bfloat16 floating-point format*, Wikimedia Foundation, Jun. 2024.

[8] *Floating-point arithmetic*, Wikimedia Foundation, Jun. 2024.

[9] *Nvidia announces financial results for fourth quarter and fiscal 2024*, NVIDIA, Feb. 2024.

[10] *Cache prefetching*, Wikimedia Foundation, Jun. 2024.

[11] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 364–373, May 1990.

[12] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, Yorktown Heights, NY, USA: Association for Computing Machinery, 2009, pp. 499–500, ISBN: 9781605584980.

[13] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 63–74.

[14] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[15] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 141–152, ISBN: 9781450340342.

[16] K. Nathella, *Making temporal prefetchers practical: The misb prefetcher*, Jun. 2019.

[17] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, 2004, pp. 96–96.

[18] K. Nesbit, A. Dhodapkar, and J. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004, pp. 135–145.

[19] M. Grannæs, M. Jahre, and L. Natvig, "Storage efficient hardware prefetching using delta-correlating prediction tables," *J. Instr. Level Parallelism*, vol. 13, 2011.

[20] G. Gerganov, *Ggerganov/llama.cpp: Llm inference in c/c++*.