Performance Analysis of the Apple AMX Matrix Accelerator

by

Jonathan Zhou

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by:	Jonathan Zhou Department of Electrical Engineering and Computer Science August 15, 2025
Certified by:	Saman Amarasinghe Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Performance Analysis of the Apple AMX Matrix Accelerator

by

Jonathan Zhou

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

ABSTRACT

Apple Silicon integrates a dedicated Apple Matrix Coprocessor (AMX) that executes outer-product style computations with high throughput, but its public programming model remains largely hidden behind the Accelerate framework. This thesis turns AMX into a more predictable and practical target by combining (i) empirical throughput characterization, (ii) a case study on AMX specific matrix multiplication (GEMM) design, and (iii) an interpretable rule-based latency model that predicts cycle counts for short AMX instruction sequences.

First, microbenchmarks quantify AMX load/store and compute limits across matrix and vector modes and data types. We analyze throughput in both GFLOPS and AMX instructions per cycle, and also observe output register based throughput limitations. Second, we develop an in-place GEMM that uses masked outer products and strategically overlapping tiles to avoid scratch buffers used by Accelerate, outperforming Accelerate while preserving simplicity.

Third, we introduce a compact latency model that decomposes cycles into per-instruction *BaseTime*, symmetric *SwitchLatency* for instruction changes, and instruction *FullLatency* (data dependency) terms. Fitted with non-negative coordinate descent on length-2 loops and validated on length-3 sequences via a lightweight loop simulation, the model obtains reasonably high accuracy while remaining helpful for those trying to understand the architecture.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to acknowledge Yishen Chen for helping guide me on my work in this project. I would also like to thank Ajay Brahmakshatriya for and other members of the MIT COMMIT lab for any advice given during our weekly lunches. Finally, I would like to thank all of the amazing friends that I have made here at MIT. I would also like to acknowledge that ChatGPT was used for drafting and proofreading parts of this thesis in addition to many steps in the research and coding process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Apple's transition to custom Apple Silicon beginning with the M1 brought a tightly integrated system-on-chip (SoC) design to the Mac platform. In addition to high-performance CPU and GPU cores, these SoCs contain domain-specific accelerators such as the Neural Engine and the Apple Matrix Coprocessor (AMX). While AMX is officially exposed only through high-level frameworks like Accelerate, independent reverse engineering has revealed a rich instruction set with dedicated X/Y input register pools and a large Z output register file, designed to execute outer-product style computations at very high throughput.

Matrix multiplication is a foundational kernel for scientific computing, graphics, cryptography, and machine learning. The combination of AMX's outer-product datapath and wide register file makes it an attractive target for high-performance implementations. This thesis tackles three practical questions regarding AMX: (1) What are the *achievable* data-movement and compute throughputs on modern Apple Silicon? (2) How should one structure a matrix multiplication to maximally exploit AMX's tile-oriented execution model? (3) Can we build an interpretable latency model that predicts cycles for small instruction sequences and generalizes beyond the training length?

This thesis answers these questions empirically and algorithmically. First, we measure AMX load/store and compute throughput under controlled microbenchmarks, separating matrix and vector modes and quantifying how instruction-level parallelism (ILP) and instruction variants affect performance. Second, we design and evaluate a GEMM implementation that uses masked outer products and carefully chosen overlapping tiles to handle non-multiple-of-8 sizes entirely in place—avoiding scratch buffers and, in several regimes, outperforming Apple's Accelerate baseline. Third, we develop an interpretable rule-based latency model. Trained on length-2 instruction loops and validated on length-3 sequences, the model decomposes cycle counts into *BaseTime* per instruction, symmetric *SwitchLatency* for instruction changes, and instruction-specific *FullLatency* (data dependency) terms, and uses a simple loop-simulation to predict longer sequences.

Contributions.

- **AMX throughput characterization.** We provide a measurement framework and report data-movement and compute throughputs for the major AMX instruction classes and data types, clarifying when AMX reaches one instruction per cycle, and analyzing how ILP and vector width influence sustained rates. In addition, we document existence of output register throughput limitations for vector computation instructions.

- **An in-place GEMM for non-multiple-of-8 sizes.** We propose a tiled, masked outer-product strategy that writes results directly into the destination matrix. The method eliminates extra scratch-space copies seen in Accelerate's approach and can exceed Accelerate's performance on common matrix sizes and achieve close to theoretical maximum throughput.

- **An interpretable latency model.** We introduce a data-driven, rule-based model that fits non-negative parameters (*BaseTime*, symmetric *SwitchLatency*, and instruction *FullLatency*) using coordinate descent with regularization and early stopping. Trained on length-2 loops, the model attains high accuracy on held-out length-3 sequences using a lightweight loop-simulation for inference.

Scope and assumptions. Our experiments target Apple Silicon on specifically the M2 Pro and consider both matrix and vector instruction modes across float and integer precisions. While the methodology is general, absolute numbers will vary by SoC generation; we focus on mechanisms (tiling, masking, dependency-aware modeling) in addition to sample data from our hardware.

Chapter 2

Related Work

Despite Apple's decision to keep the AMX accelerator undocumented and hidden from programmers behind high level libraries such as Accelerate, efforts from developers have successfully reverse-engineered the AMX instruction set. These reverse engineering efforts uncover a hardware unit with three separate register pools involving 64 byte registers and instructions for matrix and vector computations in addition to data movement between AMX registers as well as CPU memory. This direct access to AMX has enabled other research to optimize certain workloads such as fast polynomial multiplication on these M1/3 SoCs.

2.1 AMX Core Layout

Figure 2.1 is a good one image summary of the AMX processor[1]. The core contains a 32x32 grid of compute units. Each unit can perform a 16 bit FMA. In addition, each 2x2 subgrid can perform a 32 bit FMA and each 4x4 subgrid can perfrom a 64 bit FMA. One significant distinction with a typical CPU is that there exist X and Y register pools or 8 registers, which act as inputs to the compute units. The Z register pool stores the outputs to AMX instructions, and can be imagined as an 8 by 8 grid of registers.

As the figure suggests, the multiply-accumulates essentially perform a matrix outer-product between the lanes of desired X and Y register. The output is stored in columns of the Z register pool.

Notably, in comparison to ARM NEON vector instruction, which operate on 128 bit vectors for both input and output registers, AMX offers a grid of computational units capable of outer product calculations. Therefore, rather than a per-element $O(n)$ calculation typical of typical SIMD, $O(n^2)$ outer product calculations are supported. This explains the hardware need for such a large number of dedicated output registers. From a performance standpoint, we see that the $32 \cdot 32 = 1024$ 16 bit multiplication units generate a level of parallelism on the order of smaller GPUs.

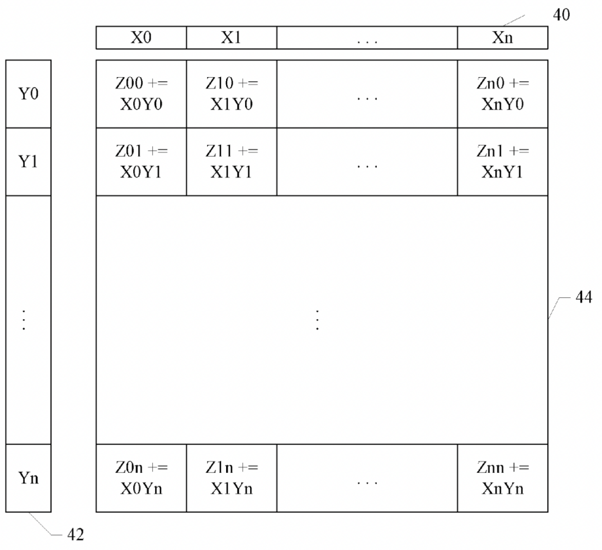


Fig. 2

Figure 2.1: AMX Register Layout

2.2 Reverse Engineered Instructions

Peter Cawley was able to reverse engineer the AMX instructions issued from CPU to AMX accelerator [1]. As described above, AMX mainly supports matrix operations via vector outer product, however it also supports vector inner products. There are also data movement instructions, and each instruction has variants for different data types. The instructions are callable via a header file via usage of inline assembly with a mix of immediate type instructions and more complex instructions having data stored in a general purpose register. The actual matrix computation data is stored in AMX registers, and thus these general purpose registers hold instruction variant choices.

2.2.1 Data Load and Store

Instructions include AMX_LDX, AMX_LDY, AMX_LDZ for loading data from memory into X, Y, Z registers. In addition, AMX_STX, AMX_STY, AMX_STZ instructions store data from these registers back into memory. Each instruction will typically operate on 64 bytes though later apple silicon iterations(M2/M3) support 128 byte and maybe more. Interleaved loads and stores are possible to Z with alternating lanes corresponding to 2 different Z registers.

2.2.2 Computation

Instructions include AMX_FMA{64/32/16} and AMX_FMS{64/32/16} for fused multiply add and fused multiply subtract on floating point values of 64, 32, or 16 bits. These instructions write only to the Z register pool, but may input data from any combination of the X, Y, and Z registers.

Flags allow the instruction to perform either outer product style or inner product style FMA/FMS. Similarly, AMX_MAX16 does the same with integer types. AMX_MATINT and AMX_MATFP perform similar FMA type computations on integers and floating point values but may perform additional ALU choices via instruction input flags. For example for integers it can perform shifting, popcnt, integer saturation.

2.2.3 Data movement

AMX_EXTRX and AMX_EXTRY instructions transfer data from either {X,Y,Z} registers into either the X or Y register pool in preparation for computation. For copying between X and Y register pools, this is quite straightforward. However, when copying from Z register pool, 2 basic options involve interpreting a set of consecutive Z registers in a column as a matrix of values. Then, elements can be extracted in a row or a column. As an example, for 64 bit doubles, a $8 \cdot 8$ matrix would take up 8 consecutive registers. A row-wise extraction would simply copy a complete register, but a column-wise extraction would take a specific lane from each of the 8 consecutive registers. This data extraction becomes more complex for smaller data types.

One major implication of these extract functions is that data movement to the Z register pool can only be done by the existing computation instructions. Rather than taking a X and Y register as input and performing an outer product, it is possible to simply copy data from either X or Y into Z via those instructions.

2.2.4 Other

Two basic additional instructions include control instructions AMX_SET and AMX_CLR. These instructions initialize the AMX coprocessor and clears the AMX register state respectively. AMX_GENLUT is another powerful instruction that can both generate look up tables(LUT) and perform indexed loads using this table in some sort of VPSHUFB type operation.

2.3 Miscellaneous AMX applications

This reverse engineering of AMX allows for other use cases of AMX outside of Apple's limited high-level abstractions. One major usecase includes cryptography. In one paper, Filho implements a fast polynomial multiplication routine on Apple Silicon and achieves several times speedup over previous state of the art [2]. In another example, certain lattice based cryptography methods are demonstrated to also experience speedup from better usage of the AMX coprocessor[3].

Chapter 3

Apple Accelerate

3.1 Reverse Engineering Accelerate

3.1.1 Accessing Accelerate library code

AMX code can generated by anyone with the help of existing reverse engineered AMX instructions bitfields[1]. However, understanding the usage of these instructions from a performance viewpoint requires looking at their actual usage inside AMX accelerate library code.

The relevant Accelerate vecLib library can be found at the following codepath:

```
/System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/

vecLib.framework/Versions/A/
```

Then, inside this folder we have .dylib files or dynamic library files of the included Accelerate library code.

- `libvMisc.dylib`
- `libvDSP.dylib`
- `libBLAS.dylib`
- `libLAPACK.dylib`

- `libLinearAlgebra.dylib`
- `libSparseBLAS.dylib`
- `libSparse.dylib`
- `libQuadrature.dylib`
- `libBNNS.dylib`

When analyzing the implementation of Apple's dynamic libraries (frameworks and private APIs), simply attempting to open these .dylib files results in empty files. This is because macOS bundles multiple dynamically linked shared objects into a single "shared cache" file (dyld_shared_cache) for faster startup and reduced memory usage. When these libraries are included in the system shared cache, they are merged into one large cache file and stripped of their individual Mach-O container formats(.dylib).Therefore, it is not possible to open or disassemble individual libraries in the cache simply by pointing standard Mach-O file reading tools, such as otool, at the .dylib filenames—those files no longer exist in their original form on the filesystem.

Instead, all code resides within dyld_shared_cache blobs, which must be parsed, decompressed, and re-assembled into standalone Mach-O images.

To automate this extraction, we employ Keith Harrison's dyld-shared-cache-extractor tool [4]. This tool identifies the active cache file and extracts the compressed data into valid .dylib files with complete symbol tables and code sections. It then identifies the active cache file (e.g., /var/db/dyld/dyld_shared_cache_x86_64) and reads its header to enumerate the embedded Mach-O images. After this step, we can now dump the assembly code inside these .dylib files and locate AMX instructions appropriately.

As a sample of some of the functions that may involve AMX instructions, we can perform a quick analysis of the basic blocks located inside these .dylib files and work back from named functions that directly contain AMX instructions to functions that indirectly call these previous functions, and so on. In table 3.1, a sample of the AMX instruction containing functions from the libvDSP.dylib accelerate library file is listed.

3.1.2 Accessing AMX instruction bitfields

Unlike standard hardware instructions, AMX instructions use a level of indirection in setting instruction bit fields. For example, in AMX_FMA64, there exists a single instruction 0x201140. In order to specify things such as the input X and Y register and output Z register number, or any other important instruction variants, the actual bit encoding of this information is stored inside a general purpose register. This means that code obtained from the above .dylib files can only tell us the fact that AMX_FMA64 is being performed, but no other information on say what the input operands are or whether special masking modes or compute modes are being used.

To solve this, we employ a simple debugger script that steps through instructions during the running of an Accelerate library function. When an AMX instruction is called, we have to access the GPR specified and extract the useful instruction encoding from that register.

Category	Functions
Direct AMX functions (18)	_vDSP_convD, _vDSP_dotpr, _vDSP_fft2d_zropD, _vDSP_fft_zript, _vDSP_fft_zriptD, _vDSP_fft_zroptD, _vDSP_imgfirD, _vDSP_meanv, _vDSP_svdiv, _vDSP_vabsi, _vDSP_viclipD, _vDSP_vintbD, _vDSP_vmmaD, _vDSP_vsimpsD, _vDSP_vswmax, _vDSP_vtrapz, _vDSP_zrvmul, _vDSP_zvmagsD
Functions 1 call away (21)	_vDSP_conv, _vDSP_DCT_CreateSetup, _vDSP_desamp, _vDSP_DFT_CreateSetup, _vDSP_DFT_Interleaved_Execute, _vDSP_DFT_Interleaved_ExecuteD, _vDSP_DFT_zop_CreateSetupD, _vDSP_DFT_zrop_CreateSetup, _vDSP_distancesqD, _vDSP_dotpr2, _vDSP_fft2d_zipt, _vDSP_fft2d_zripD, _vDSP_fft2d_zroptD, _vDSP_fftm_zopD, _vDSP_mmul, _vDSP_mvessq, _vDSP_vminD, _vDSP_vsorti, _vDSP_vswap, _vDSP_zconvD, _vDSP_zvmmaa
Functions 2 calls away (18)	_vDSP_create_fftsetup, _vDSP_create_fftsetupD, _vDSP_destroy_fftsetup, _vDSP_DFT_Interleaved_CreateSetup, _vDSP_DFT_Interleaved_CreateSetupD, _vDSP_DFT_zop_CreateSetup, _vDSP_DFT_zrop_CreateSetupD, _vDSP_fft2d_zop, _vDSP_fft2d_zopt, _vDSP_fft2d_zriptD, _vDSP_fft_zop, _vDSP_fft_zropD, _vDSP_fftm_zipD, _vDSP_fftm_zoptD, _vDSP_maxvi, _vDSP_vdist, _vDSP_vflt8D, _vDSP_zvmmaaD
Functions 3 calls away (17)	_vDSP_DCT_Execute, _vDSP_destroy_fftsetupD, _vDSP_DFT_zop, _vDSP_fft2d_zip, _vDSP_fft2d_zrop, _vDSP_fft3_zop, _vDSP_fft5_zop, _vDSP_fft_zip, _vDSP_fft_zopD, _vDSP_fft_zopt, _vDSP_fft_zripD, _vDSP_fft_zrop, _vDSP_fftm_ziptD, _vDSP_fftm_zop, _vDSP_fftm_zropD, _vDSP_zvabs, _vDSP_zvneg
Functions 4 calls away (17)	_vDSP_fft2d_zopD, _vDSP_fft2d_zrip, _vDSP_fft2d_zript, _vDSP_fft2d_zropt, _vDSP_fft3_zopD, _vDSP_fft5_zopD, _vDSP_fft_zipD, _vDSP_fft_zipt, _vDSP_fft_zoptD, _vDSP_fft_zrip, _vDSP_fft_zropt, _vDSP_fftm_zip, _vDSP_fftm_zipt, _vDSP_fftm_zopt, _vDSP_fftm_zripD, _vDSP_fftm_zrop, _vDSP_fftm_zroptD
Functions 5 calls away (9)	_vDSP_fft2d_zipD, _vDSP_fft2d_zoptD, _vDSP_fft_ziptD, _vDSP_fftm_zrip, _vDSP_fftm_zript, _vDSP_fftm_zriptD, _vDSP_fftm_zropt, _vDSP_vmin, _vDSP_zvaddD
Functions 6 calls away (1)	_vDSP_fft2d_ziptD

Table 3.1: vDSP routines grouped by call distance from AMX use.

Chapter 4

Basic Throughput Analysis

4.1 Hardware Setup

All experiments were run on a Mac mini with an Apple M2 Pro SoC. The CPU complex comprises eight performance (P) cores and four efficiency (E) cores. The memory subsystem is LPDDR5–6400 with a peak documented unified-memory bandwidth of 204.8 GB/s. The AMX matrix accelerator appears as one coprocessor per P-core cluster (four P-cores each), for a total of two AMX instances on this system.[1] For reference, the capabilities of the M2 Pro SoC should be about double the compute and bandwidth of the baseline M2 and about half the compute and bandwidth of the Apple M2 Ultra.

4.2 Data Movement Throughput

AMX loads/stores interface through the CPU cluster's L2 and, indirectly, unified memory. Given one AMX per P-cluster composed of 4 Performance cores, direct L1 access would be atypical for a cluster-level unit.

Load throughput. We begin with basic throughput testing of memory load and store speeds between the matrix accelerator and unified SoC memory. In Table 4.1 we experiment with Instruction Level Parallelism(ILP) factors of load instructions into different registers. In

[1]This observation is consistent with measured scaling across threads; see §4.3.1.

addition, AMX_LDX and AMX_LDY allows for loading either 1, 2, or 4 registers of data in a single instruction, and this is encoded as the instruction vector width.

Table 4.1: Measured load throughput (GB/s) on the AMX accelerator for varying ILP and instruction vector-widths (1MB data size).

Register Pool	**ILP Factor**	**Instruction Vector Width**	**Throughput (GB/s)**
X, Y	1	1	195
X, Y	1	2	364
X, Y	1	4	599
X, Y	2	1	254
X, Y	2	2	419
X, Y	2	4	699
X, Y	4	1	335
X, Y	4	2	645
Z	any	any	195

The first immediate observation is that loads targeting Z saturate near ∼195 GB/s. Specifically, this means that even though AMX has multi-register loads in Z, this does not have a material impact—likely a microarchitectural limit on the Z ingress path

However, for the X and Y register pools, both increasing the instruction vector width as well as the ILP factor will increase throughput. Expanding the vector width from one to two and four elements increases throughput from 195GB/s to 364GB/s (1.87×) and 599GB/s (3.07×), respectively. Introducing additional ILP further elevates performance: at a fixed width of one element, increasing the ILP factor from one to two and then to four increases throughput from 195GB/s to 254GB/s (1.3×) and 335GB/s (1.7×). The maximum load throughput into either X or Y is 699 GB/S.

On memory-bandwidth perspective. The documented unified-memory peak is 200 GB/s [5]. Cache-resident measurements exceeding this (e.g., ∼700 GB/s) therefore reflect L2→AMX bandwidth, not DRAM streaming. The working set of 1MB from Table 4.1 is well within L2 limits, however sizes larger than L2 cache size would likely result in reduced bandwidth. This 200 GB/s helps explain the 195GB/s Z memory store bandwidths as well.

Store throughput. We can also measure the store throughput of AMX_STX and AMX_STY and AMX_STZ instructions. We see in table 4.2 that increasing parallelism beyond the size of a single register or 64 bytes does not significantly improve performance. From the perspective of a compiler engineer, near maximal performance can be achieved via simply calling the single register variant of these AMX store instructions.

Table 4.2: Measured store throughput (GB/s) on the AMX accelerator for varying ILP and vector-register widths (1 MB data).

Register Pool	**ILP Factor (n_{block})**	**Vector Width**	**Throughput (GB/s)**
X, Y, Z	1	1	158
X, Y, Z	1	2	167
X, Y, Z	2	1	174
X, Y, Z	2	2	174
X, Y, Z	4	1	174
X, Y, Z	4	2	174

4.3 Instruction Throughput

Now, we can consider the instruction throughput of the main compute instructions. Since the M2 Pro processor used contains 2 AMX cores, Apple's Grand Central Dispatch framework was used to assign work to both threads. The kernels being run must also have logically parallel instructions to ensure maximum throughput within the AMX core.

4.3.1 Matrix Mode

As a point of reference, the boost clock of the M2 Pro processor is 3.47 GHz, so we see immediately from $3.47 \cdot 10^9$ and $6.95 \cdot 10^9$ instructions per second for the $FMA16$, $FMA32$, and $FMA64$ instructions that each AMX core has a throughput of 1 AMX instruction per cycle. For $FMA16$, the throughput drops to 1 instruction every 2 cycles. The code being run for this workload is simply a single thread with a large loop of AMX instructions for the (single) core workload and an increased number of threads(up to 6x) for the multicore results. Therefore, the purely singlethreaded workload indeed targets a single AMX core but may include overhead from instruction decoding that the multithreaded results are able to

Table 4.3: Matrix Mode compute instruction throughput for both single and double AMX core workloads

	GFLOPS	Giga Insn./sec	GFLOPS (single)	Giga Insn./sec (single)	Ops/insn.
FMA16	7108	3.47	3320	1.62	2048
FMA32	3560	6.95	1669	3.26	512
FMA64	890	6.95	417	3.26	128
i8,i16	7108	3.47	3319	1.62	2048
(i8,i32)(i16,i32)	3313	1.62	1660	0.81	2048

overcome. In general, this indicates that it may be useful to fully make use of the AMX cores via dispatching computation of many smaller kernels rather than having only 1 or 2 threads running AMX instruction programs.

Because outer products scale quadratically with the number of lanes per register, halving the data width from 32 bits to 16 bits doubles the number of values stored in a single register from 16 to 32, and quadruples the number of operations to be performed in the outer product. Going from $FMA64$ to $FMA32$, we see exactly a $4x$ speedup from 417 GFLOPS to 1669 GFLOPS

Going from $FMA32$ to $FMA16$ we see only a 2x improvement in GFLOPS, caused by a halving to a maximum throughput of 1 instruction every 2 cycles. This may suggest some sort of data movement based limitation due to taking 2 registers as input and outputting to a total of 32 registers, or half of the Z register file in a single instruction.

A full analysis of mixed precision modes and other data type variants can be seen in table 4.4.

Several observations can be made here. First, we see that GFLOPS are halved when going from Fused multiply-add to multiply or add only. In addition, bf16 performance is essentially the same as normal 16 bit float performance.

For integer type fused multiply-add calculations, the GFLOPS are of i8i16 are similar to that of f16f16, however the GFLOPS of i16i16, i8i32, i16i32 are all similar to that of f32f32. Inherently, integer type multiplication should be more difficult than floating point multiplication. This is since FP16, FP32, FP64 use 10, 23, and 52 mantissa bits with remaining bits left for sign and exponent. Multiplication of these floating point types should really just be multiplication of the mantissa combined with addition of the exponent bits, so from a hardware point of view, the AMX architecture appears to be optimized for floating point multiplication. In fact, 64 bit integer multiplication is simply not supported here.

For floating point mixed precision, the only supported mode is f16f32, which matches the 1660 GFLOPS of f32f32 outer products, indicating a simple extension of the 16 bit floating point values to 32 bits.

Table 4.4: Matrix Mode Instruction Throughput Benchmark Results using a single AMX core

test_name	**GFLOPS (single)**	**Giga Insn./sec (single)**	**Ops/insn.**
matfp_f16f16_x*y+z	3315.555797	1.618924	2048.0
mac16_mat_i8i16_x*y+z	3319.576216	1.620887	2048.0
matfp_bf16bf16_x*y+z	3316.118980	1.619199	2048.0
fma16_mat_f16f16_x*y+z	3329.348850	1.625659	2048.0
matint_i8i16_x*y+z	3321.187686	0.810837	4096.0
fma32_mat_f32f32_x*y+z	1669.245686	3.260245	512.0
matfp_f32f32_x*y+z	1661.100501	3.244337	512.0
fma32_mat_f16f32_x*y+z	1658.159777	3.238593	512.0
fma16_mat_f16f16_y+z	1659.863657	1.620961	1024.0
fma16_mat_f16f16_x*y	1659.383397	1.620492	1024.0
fma16_mat_f16f16_x+z	1660.190878	1.621280	1024.0
mac16_mat_i8i16_x+z	1660.552468	1.621633	1024.0
mac16_mat_i8i16_x*y	1661.236853	1.622302	1024.0
mac16_mat_i8i16_y+z	1660.202287	1.621291	1024.0
fma16_mat_f16f32_x*y+z	1657.516304	0.809334	2048.0
matfp_f16f32_x*y+z	1660.849274	0.810962	2048.0
mac16_mat_i8i32_x*y+z	1659.537990	0.810321	2048.0
matfp_bf16f32_x*y+z	1661.406073	0.811233	2048.0
mac16_mat_i16i32_x*y+z	1661.094791	0.811081	2048.0
matint_i16i16_x*y+z	1660.266463	0.810677	2048.0
mac16_mat_i16i16_x*y+z	1661.587470	0.811322	2048.0

Table 4.4: (continued)

test_name	**GFLOPS (single)**	**Giga Insn./sec (single)**	**Ops/insn.**
matint_i8i32_x*y+z	1659.191086	0.810152	2048.0
matint_i16i32_x*y+z	1659.659833	0.810381	2048.0
fma32_mat_f32f32_x*y	835.071446	3.261998	256.0
fma32_mat_f16f32_x+z	831.354256	3.247478	256.0
fma32_mat_f16f32_x*y	830.988411	3.246048	256.0
fma32_mat_f32f32_y+z	834.231071	3.258715	256.0
fma32_mat_f32f32_x+z	834.405747	3.259397	256.0
fma32_mat_f16f32_y+z	830.728209	3.245032	256.0
mac16_mat_i16i16_y+z	831.754068	0.812260	1024.0
mac16_mat_i8i32_y+z	831.572271	0.812082	1024.0
mac16_mat_i16i32_y+z	831.462981	0.811976	1024.0
mac16_mat_i16i16_x*y	832.067747	0.812566	1024.0
fma16_mat_f16f32_x+z	831.081674	0.811603	1024.0
fma16_mat_f16f32_y+z	830.174237	0.810717	1024.0
fma16_mat_f16f32_x*y	831.302943	0.811819	1024.0
mac16_mat_i16i16_x+z	831.470671	0.811983	1024.0
mac16_mat_i8i32_x+z	831.982679	0.812483	1024.0
mac16_mat_i16i32_x*y	831.363732	0.811879	1024.0
mac16_mat_i8i32_x*y	831.809016	0.812313	1024.0
mac16_mat_i16i32_x+z	831.797740	0.812302	1024.0
fma64_mat_f64f64_x*y+z	417.076200	3.258408	128.0
matfp_f64f64_x*y+z	416.080850	3.250632	128.0
fma64_mat_f64f64_x*y	208.127483	3.251992	64.0
fma64_mat_f64f64_y+z	207.920442	3.248757	64.0
fma64_mat_f64f64_x+z	205.821804	3.215966	64.0

4.3.2 Vector Mode

Table 4.5: Vector Mode compute instruction throughput for both single and double AMX core workloads

Kernel	GLOPs	Giga Insn./sec	GFLOP (single)	G Insns (single)	Ops/insn.
vecfp_f16(x2,x4)	1113	8.69, 4.35	414	3.24, 1.62	256, 128
FMA16_vec	1042	16.28	415	6.48	64
vecfp_f32(x2,x4)	557	8.7, 4.35	208	3.26, 1.62	128, 64
FMA32_vec	522	16.3	207	6.46	32
vecfp_f64(x2,x4)	278	8.7, 4.35	104	3.24, 1.62	64, 32
FMA64_vec	261	16.3	104	6.48	16

Now, we look at Vector mode throughput. As a whole, the compute capabilities of the AMX core are unable to be fully maximized while in vector mode. This is since $FMA16$ outer product is able to compute 32 registers of data from 2 input registers. In vector mode, there exist 2x and 4x variants that allow input from 2 and 4 pairs of input registers respectively, but this still only outputs a maximum of 2 or 4 Z registers of data. In table 4.5, the 2x and 4x vector mode instructions seem to only decrease instruction decoding. We see in the first versus 2nd row that the introduction of x2 and x4 variants in the second generation AMX_VECFP instruction compared to the AMX_FMA16 instruction leads to identical GFLOPS for a single core, but approximately 6.7% greater max throughput when using both AMX cores. In comparison to matrix mode, there is no longer an $O(n^2)$ type input to output behavior, and therefore throughput from FMA16 to FMA32 to FMA64 is perfectly linear with the GFLOPS halving for each doubling in datatype width.

When looking at the number of instructions run per second, values of $6.48 \cdot 10^9$ suggest that this vector mode calculation is able to run up to 2 instructions per cycle compared to a 3.47 Ghz clock. However, we also see up to 16.3 Giga Instructions per second across 2 AMX cores in FMA16 in vector mode. This is similar to the matrix mode results from above, where increasing the number of threads containing AMX instructions from 1 to 2 to more than 2 allows for a throughput increase of 1x to 2x to up to 2.67x compared to a single threaded program. Maximizing AMX core usage should therefore use at a thread count of at a minimum the number of AMX cores present in the system. However, additional threads should likely be used to achieve full usage of the AMX cores, perhaps with these AMX instruction kernels being used as small subroutines with a large number of parallel kernel calls.

Table 4.6: Vector Mode Instruction Throughput Benchmark Results using a single AMX core

test_name	**GFLOPS (single)**	**Giga Insn./sec (single)**	**Ops/insn.**
vecint_i8i32_x*y+z_x2	828.184952	3.235097	256.0
vecint_i8i16_x*y+z_x4	823.951709	1.609281	512.0
vecint_i8i16_x*y+z_x2	823.639204	3.217341	256.0
vecint_i8i32_x*y+z_x4	822.304079	1.606063	512.0
vecint_i8i16_x*y+z	829.710220	6.482111	128.0
vecint_i8i32_x*y+z	822.675096	6.427149	128.0
vecfp_f16f16_x*y+z_x4	413.554657	1.615448	256.0
vecfp_f16f16_x*y+z_x2	413.947926	3.233968	128.0
vecfp_bf16bf16_x*y+z_x2	413.231932	3.228374	128.0
vecfp_bf16bf16_x*y+z_x4	413.056176	1.613501	256.0
mac16_vec_i8i16_x*y+z	411.716084	6.433064	64.0
fma16_vec_f16f16_x*y+z	413.240748	6.456887	64.0
vecfp_f16f16_x*y+z	414.602361	6.478162	64.0
vecfp_bf16bf16_x*y+z	411.071588	6.422994	64.0
vecfp_bf16f32_x*y+z_x4	412.724509	1.612205	256.0
vecint_i16i32_x*y+z_x4	413.689192	1.615973	256.0
vecfp_bf16f32_x*y+z_x2	413.737027	3.232321	128.0
vecint_i16i16_x*y+z_x4	413.315872	1.614515	256.0
vecfp_f16f32_x*y+z_x4	413.232805	1.614191	256.0
vecfp_f16f32_x*y+z_x2	413.208068	3.228188	128.0
vecint_i16i16_x*y+z_x2	413.737027	3.232321	128.0
vecint_i16i32_x*y+z_x2	413.616609	3.231380	128.0
vecint_i16i32_x*y+z	412.587949	6.446687	64.0
mac16_vec_i16i16_x*y+z	411.914393	6.436162	64.0
vecfp_bf16f32_x*y+z	411.442614	6.428791	64.0
vecint_i16i16_x*y+z	411.877516	6.435586	64.0
vecfp_f16f32_x*y+z	410.926504	6.420727	64.0

Table 4.6: (continued)

test_name	**GFLOPS (single)**	**Giga Insn./sec (single)**	**Ops/insn.**
vecfp_f32f32_x*y+z_x2	208.747568	3.261681	64.0
vecfp_f32f32_x*y+z_x4	207.236058	1.619032	128.0
mac16_vec_i8i16_x+z	206.780426	6.461888	32.0
mac16_vec_i8i16_y+z	206.858771	6.464337	32.0
mac16_vec_i8i16_x*y	206.821140	6.463161	32.0
fma32_vec_f32f32_x*y+z	206.748573	6.460893	32.0
fma16_vec_f16f16_x+z	206.760074	6.461252	32.0
fma16_vec_f16f16_x*y	206.925654	6.466427	32.0
fma16_vec_f16f16_y+z	207.859441	6.495608	32.0
fma32_vec_f16f32_x*y+z	208.588516	6.518391	32.0
vecfp_f32f32_x*y+z	206.854787	6.464212	32.0
mac16_vec_i16i16_x*y	207.295850	6.477995	32.0
mac16_vec_i16i16_x+z	206.736189	6.460506	32.0
mac16_vec_i16i16_y+z	206.877813	6.464932	32.0
vecfp_f64f64_x*y+z_x2	103.662712	3.239460	32.0
vecfp_f64f64_x*y+z_x4	104.035562	1.625556	64.0
fma64_vec_f64f64_x*y+z	103.642365	6.477648	16.0
fma32_vec_f32f32_y+z	103.929498	6.495594	16.0
fma32_vec_f16f32_x*y	103.834019	6.489626	16.0
vecfp_f64f64_x*y+z	103.646479	6.477905	16.0
fma32_vec_f16f32_x+z	103.627252	6.476703	16.0
fma32_vec_f16f32_y+z	103.702759	6.481422	16.0
fma32_vec_f32f32_x+z	103.743171	6.483948	16.0
fma32_vec_f32f32_x*y	103.664491	6.479031	16.0
fma64_vec_f64f64_y+z	51.892811	6.486601	8.0
fma64_vec_f64f64_x*y	51.923565	6.490446	8.0
fma64_vec_f64f64_x+z	51.977913	6.497239	8.0

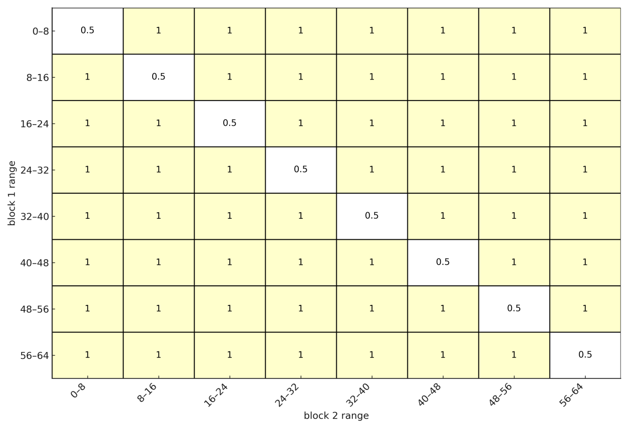
Now, looking at mixed precision and integer datatype vector computations in table 4.6, we similarly see that GFLOPS are halved when going from Fused multiply-add to multiply or add only. In addition, bf16 performance is essentially the same as normal 16 bit float performance.

For integer type fused multiply-add calculations, there exists slightly different behavior compared to matrix mode. i8i16 and i8i32 computations achieve a maximum throughput of around 830 GFLOPS, whereas f16f16, i16i16, i16i32 all achieve up to 415 GFLOPS. At a maximum of around 208 GFLOPS we see f32f32 and f16f32. Finally, only f64f64 achieves a maximum of 104 GFLOPS. Variants such as f32f64 or f16f64 or i32i32 do not seem to be implemented. At the same time, we also see that integer FMAs can actually be done faster than floating point, with both i8i16 and i8i32 achieving double the maximum throughput of f16f16 FMAs. This contrasts with matrix mode where i8i16 was the same speed as f16f16 and i16i16 was the same speed as f32f32. From an architectural viewpoint, it may seem reasonable to assume that the outer products are designed primarily for usage for floats for say Machine learning type applications. However, vector mode calculations offer reasonable performance for both integer and floating point types.

4.4 Output Register Dependencies

When looking at maximum possible throughput of vector computation instructions, some peculiarities can appear regarding throughput limitations depending on the output Z register location. As a test, when we run a loop of 16 of the same AMX vector instruction, changing only the output Z register locations with a granularity of size 8 blocks, vector mode instructions follow behavior that can be seen in figure 4.1. In this test, within each block of 8 registers, each instruction writes to a separate register in that block.

Analysis. We can see first off that since these vector mode instructions have significantly less computation being performed in compared to the outer product modes, it is actually possible to achieve greater than 1 AMX instruction per cycle. This is likely since each vector instruction only makes use of some subpart of the actual compute units underlying the AMX


(a) Diagonal only

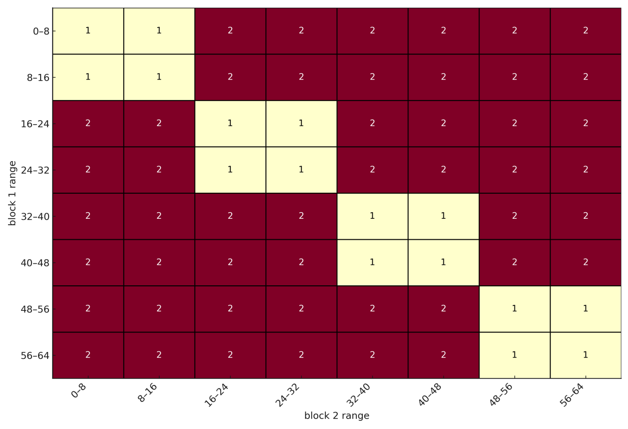

(b) 2×2 along diagonal

Figure 4.1: AMX instruction throughput patterns for size-8 output-register blocks. Axes show *block 1 range* (rows) and *block 2 range* (columns).

accelerator. We further see from the heatmaps that writing to either the same output register or say the same block of 16 registers in the Z register can result in being limited in throughput. This likely indicates some sort of spatial based connection between the compute units and specific registers of the Z register pool. There appears to be distinctions at both the block of 8 register and block of 16 register levels, as seen in figure 4.1, where some instructions conflict within the same block of 8, but others conflict in the same block of 16.

Specifically, we list the instructions and the categories they fall into in table 4.7. The first 2 classes are equivalent to the ones seen in figure 4.1 while the other 2 classes are simply instructions where the throughput of the instruction is fixed at most 0.5 or 1 amx instructions per cycle. As noticed in the earlier analysis, x2 and x4 variants tend to achieve lower throughput as measured in cycles per instruction. For example, vecfp_f32f32 exists in the Same-16 block class, while the x2 and x4 variants have constant throughputs of 1.0 and 0.5 instructions per cycle. From this, we also see that the maximum throughput in units of GFLOPS is identical between these variants. However, this more in depth analysis demonstrates that choosing the single vector x1 variant requires ensuring that the output registers are chosen correctly in order to achieve this maximal throughput value. These additional x2 and x4 modes can therefore be seen as a simpler way to achieve maximal performance.

Class (8-block, op-variant condensed)	Instructions
Diagonal (0.5 on diag, 1 otherwise)	`mac16_vec_i16i16` `vecint_i16i16_x2` `vecfp_bf16f32` `vecint_i16i32` `vecfp_bf16f32_x2` `vecint_i16i32_x2` `vecfp_f16f32` `vecint_i8i16` `vecfp_f16f32_x2` `vecint_i8i16_x2` `vecint_i16i16` `vecint_i8i32`
Same-16 block (1 within 16-reg quarter, 2 otherwise)	`fma16_vec_f16f16` `vecfp_bf16bf16` `fma32_vec_f16f32` `vecfp_f16f16` `fma32_vec_f32f32` `vecfp_f32f32` `fma64_vec_f64f64` `vecfp_f64f64` `mac16_vec_i8i16`
Constant 1.0	`vecfp_bf16bf16_x2` `vecfp_f32f32_x2` `vecfp_f16f16_x2` `vecfp_f64f64_x2`
Constant 0.5	`vecfp_bf16bf16_x4` `vecfp_f64f64_x4` `vecfp_bf16f32_x4` `vecint_i16i16_x4` `vecfp_f16f16_x4` `vecint_i16i32_x4` `vecfp_f16f32_x4` `vecint_i8i16_x4` `vecfp_f32f32_x4` `vecint_i8i32_x2`

Table 4.7: AMX instruction throughput classes for 8-wide output-register blocks

AMX generational differences Finally, we note that there appears to be distinct behavior in each of the apple silicon generations. The earlier results are generated from an M2 Pro processor. However, on an M3 processor, it is possible to achieve vector mode throughput that follow a completely different pattern as seen in figure 4.2. In it, we see behavior involving blocks of 32 registers. However, rather than having register 0-32 and 32-64 be disinct units, we see 16-48 as registers competing for the same resource while the other 32 registers are also in the same block in a wrapped around sort of fashion.

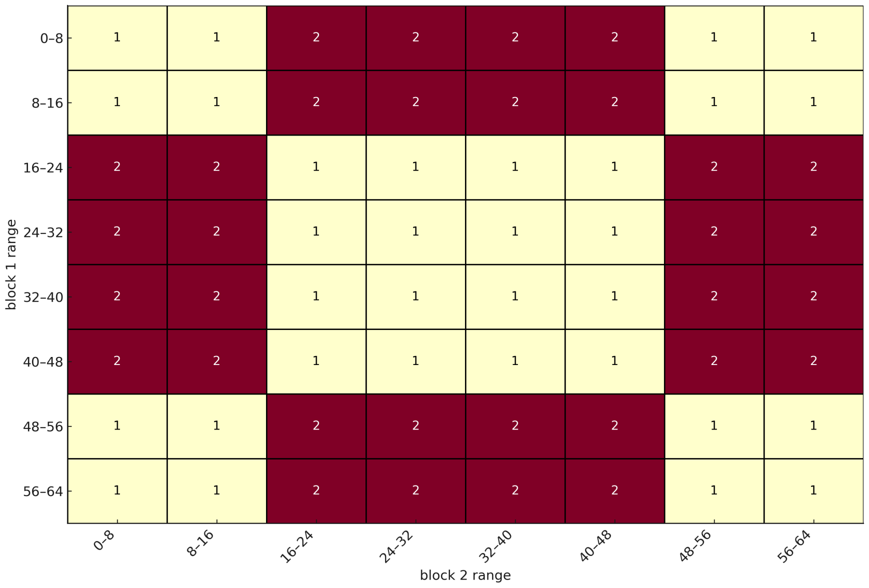


Figure 4.2: Output register heatmap behavior of a vector mode instruction on M3 architecture.

Chapter 5

Matrix Multiplication Case Study

5.1 Matrix Multiply

The target problem here is multiplying matrices A and B of size $M \cdot K$ and $K \cdot N$ respectively. The resulting matrix C should then have resultant dimensions of $M \cdot N$. For simplicity, we initially assume in this section that M and N are divisible by a sufficiently large power of 2.

5.1.1 Methodology

The main overarching algorithm using the AMX core involves performing a reduction across the K dimension. We work a properly sized tile of $8 \cdot 8$ doubles or $16 \cdot 16$ floats within the output matrix C. Within this block, the AMX_FMA instruction is used to accumulate the K different outer products for the corresponding range of {16/8} values of matrices A and B.

In this basic strategy we already see that tiling is used to load data from matrices A and B in the appropriate sized chunks. Reducing along the K dimension using the accumulation portion of the fused multiply-add also reduces the number of data stores. From this we can calculate that the number of FMAs when operating on 64 bit values is

$$K \cdot \frac{M}{8} \cdot \frac{N}{8}$$

In addition, the reduction ensures that the number of data stores when operating on 64 bit values is

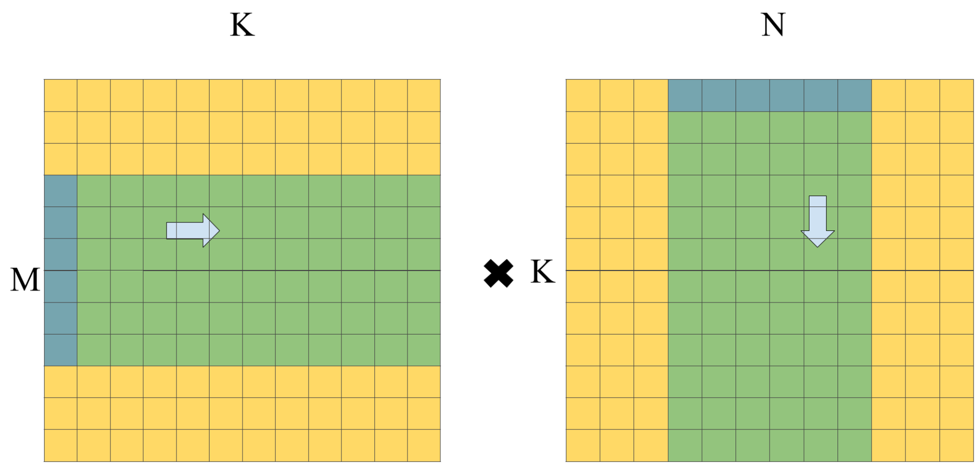


Figure 5.1: Outer Product data access pattern example. This example involves a 6x6 tiling strategy, but this would be 8x8 or 16x16 or 32x32 depending on data type. The green cells represent the elements used across the tiled reduction on the K dimension. Each outer product takes an individual column or row shaded blue.

$$\frac{M}{8} \cdot \frac{N}{8}$$

The number of 64 byte data loads from matrices A and B is simply

$$K \cdot \frac{M}{8} \cdot \frac{N}{8}$$

or the same as the number of FMAs performed.

5.1.2 Basic Optimizations

Basic optimizations include using parallelism to make full use of the AMX computational resources. From a single threaded viewpoint, we try to maximize instruction level parallelism by performing several $8 \cdot 8$ (M,N) tiles in parallel. This of course also introduces more data level parallelism from loading more than 64 bytes from memory into AMX registers in a single instruction. A basic implementation of this can be seen in listing 5.1, where we can achieve 4x instruction level parallelism for the AMX_FMA64 instruction by tiling to a 16 by 16 size rather than 8 by 8.

Listing 5.1: Simple optimized implementation for $C+=A^T\times B$.

```
for(uint64_t m = 0; m <= M-16; m += 16){
        for(uint64_t n = 0; n <= N-16; n+= 16){
            for (uint64_t i = 0; i < 8; ++i) {
                AMX_LDZ(load_store_2 | ((i * 8ull + 0) << 56) | (uint64_t)(C + (m + i) * N +
    n));
                AMX_LDZ(load_store_2 | ((i * 8ull + 2) << 56) | (uint64_t)(C + (m + 8 + i)* N
     + n));
            }
            for (uint32_t k = 0; k < K; k ++) {
                AMX_LDY(load_store_2 | (uint64_t)(A + k * M + m));
                AMX_LDX(load_store_2 | (uint64_t)(B + k * N + n));

                AMX_FMA64((0ull << 20) | (( 0ull) << 10) | (( 0ull) << 0));
                AMX_FMA64((1ull << 20) | ((64ull) << 10) | (( 0ull) << 0));
                AMX_FMA64((2ull << 20) | (( 0ull) << 10) | ((64ull) << 0));
                AMX_FMA64((3ull << 20) | ((64ull) << 10) | ((64ull) << 0));
            }
            for (uint64_t i = 0; i < 8; ++i) {
                AMX_STZ(load_store_2 | ((i * 8ull + 0) << 56) | (uint64_t)(C + (m + i) * N +
    n));
                AMX_STZ(load_store_2 | ((i * 8ull + 2) << 56) | (uint64_t)(C + (m + 8 + i)* N
     + n));
            }
        }
    }
```

Additionally, as seen in listing 5.1, the memory access pattern will require transposing the data being accessed in at least one of the matrices. By loading a square grid of data of the same size as a single outer product, the AMX_EXTRX instruction is able to correctly read columns of the submatrix into a single register, avoiding this issue. If we compare this tiled strategy compared to the basic implementation suggested earlier, we see that that the number of FMAs computes is

$$4 \cdot (\frac{M}{16} \cdot \frac{N}{16}) = \frac{M \cdot N}{64}$$

which is identical to a naive implementation. The same is true regarding number of data stores. However, we now see that for data loads, there are now only

$$2 \cdot K \cdot (\frac{M}{16} \cdot \frac{N}{16}) = \frac{K \cdot M \cdot N}{128}$$

This is actually half as many loads as a naive non-tiled implementation.

Regarding transposes, we also see that it is better to perform this transpose step all at the start before any actual outer product steps.

5.1.3 Performance Observations

Memory usage considerations Note that the Fused Multiply-add instruction performs an outer product between a single X register and a single Y register. However, the number of Z registers depends on the datatype size. For 64 bit data types, which can be either doubles or longs, each of the X and Y registers will hold 8 lanes, resulting in 64 resultant outer product values. This will take 8 Z registers. The same calculation in 32 bits will however now have 16 input lanes and thus 256 resultant outer products taking up 16 Z registers. When we reduce again to 16 bits, it now goes to 32 Z registers.

When we consider the total of 64 Z registers, this indicates a maximum of 8x, 4x, or 2x instruction level parallelism(ILP) based off of the data type we are using for these FMA outer products. In practice, when performing large batches of fp16, fp32, or fp64 matrix computations, taking full use of this 2x, 4x, or 8x ILP is essential for maximizing performance.

5.1.4 Generalizing to non multiples of 8

The previous simple tiling methods work well when the matrix dimensions are multiples of 8. However, when this condition is relaxed, there will remain a tail strip of incomplete tiles that must be dealt with.

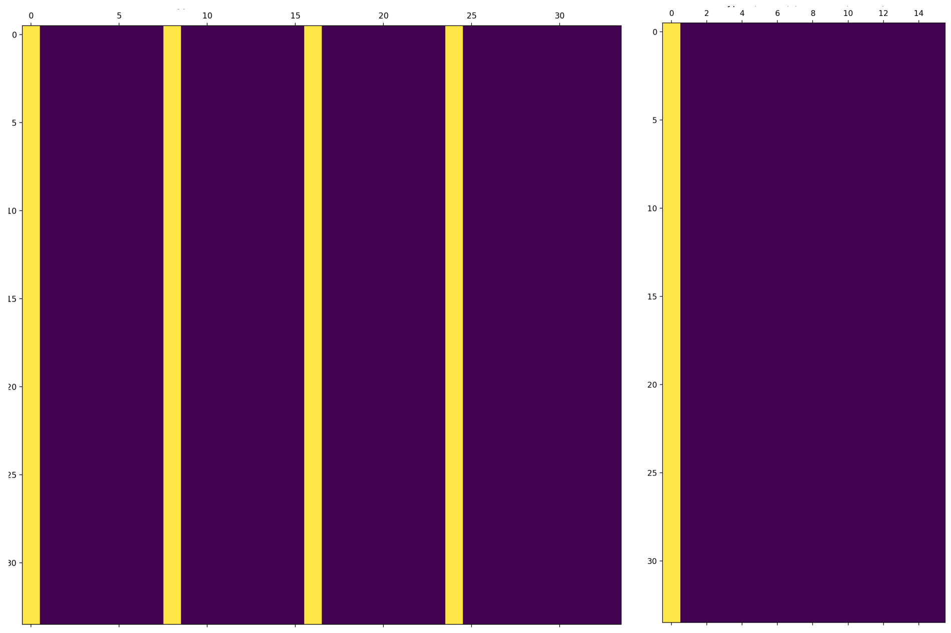
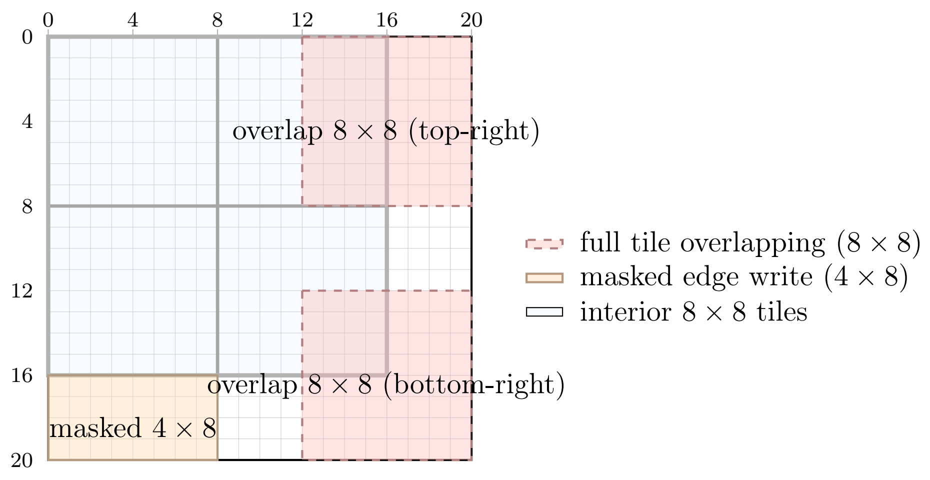


Figure 5.2: Locations of Stores relative to the C matrix for a 34×34 size matrix multiplication when calling CBLAS_SGEMM. Yellow indicates existence of a data store to that location.

The accelerate library deals with this by writing the full tiles directly back to memory inside the desired C matrix. They then use masked variants of the FMA instructions in order to generate the incomplete tiles into a separate buffer in memory. This can be easily detected, with the gap between consecutive addresses in memory being 16 for the extra buffer rather than 34. Finally, these buffers are written back to the C matrix. In figure 5.2, we see the locations of AMX store instructions relative to the C matrix, where we see writes to not only the C matrix but also a completely separate scratch space of size 34×16 bytes.

5.1.5 Handling Non–Multiples of Eight via Masked Outer Products

In comparison to Accelerate, we propose an in place GEMM algorithm that does not require these extra steps involving copying of data between the scratch space and the actual C matrix. The general strategy can be seen in Figure 5.3. In it, we describe how to tile a 20x20 output C matrix using 8x8 outer product result tiles. Notably, for GEMM type operations, we need to preserve the existing data inside the Z registers at the beginning of the function call. For tiles at the bottom of the matrix, we already copy over data in essentially a row by row



0
4
8
12
16
20
0
4
8
12
16
20
overlap 8 × 8 (top-right)
overlap 8 × 8 (bottom-right)
masked 4 × 8
full tile overlapping (8 × 8)
masked edge write (4 × 8)
interior 8 × 8 tiles

Figure 5.3: Tiled view of a 20×20 output matrix using 8x8 tiles. Additional partial tiles must be written to with masking in order to avoid scalar operations

fashion, so we can simply stop early. However, for the right side of the matrix, we can align the right side of the tile with the right side of the matrix when loading data. Then, masked compute instructions(allowing toggling lanes for both X and Y register pools), allow parts of the data to remain untouched. This means that we can copy back the entire register contents row by row without destroying any data.

This strategy increases the number of tiled outer product accumulations we need to perform in the 20×20 example from 4 to 9, but this still increases the throughput drastically over scalar methods.

5.1.6 Empirical Results

We compare our proposed optimizations against the baseline of the Accelerate library. The main test we make is of a simple transposed matrix multiplication or $C+ = A^T \cdot B$. In Figure 5.4, we see performance of our implementation compared to Accelerate CBLAS on square matrices. Every 8 data points we see a spike in throughput up to around 350-400 GFLOPS of FMA64 throughput. For reference, the maximum GFLOPS of FMA64 compute generated from simple throughput testing in Table 4.3 is 417, indicating efficiency close to that suggested by a simple compute bandwidth bound. This is in spite of redundant loads of data inside the M and N axes of the A and B matrices by a factor of approximately $\frac{N}{\text{tile width}}$ and $\frac{M}{\text{tile width}}$ respectively.

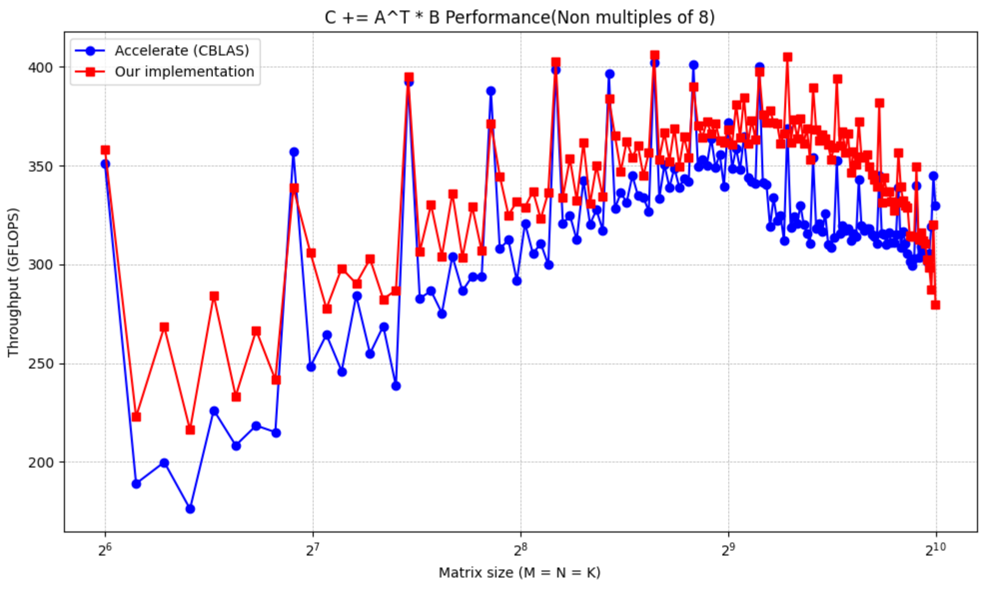


Figure 5.4: Performance of $C+=A^T\cdot B$ on 64 bit floating point square matrices of size $64+7\cdot n$ from 64 up to 1024.

Chapter 6

Latency Analysis

Deriving throughput numbers for each of the AMX instructions can be simply done by running the single instruction over and over, while making sure to prevent data dependencies between the output and input of consecutive instructions. However, if we desire to derive the latency of these instructions, then we must follow a different approach. A necessary requirement to being able to observe instruction latencies is the ability to count the number of cycles a certain program takes. We begin this section by explaining the methodology for benchmarking the cycle counts of programs before proceeding on towards our instruction latency calculation and proposed model for predicting cycle counts of small arbitrary sequences of AMX instructions.

6.1 Benchmarking AMX program cycle counts

Due to the fact that the AMX accelerator has not been publicly announced by Apple, there is little to no support for performance profiling tools that are capable of operating on these AMX instructions. However, thanks to reverse engineering work from both Daniel Lemire and other individuals in the open source community [6], there exist performance counters that are able to target Apple silicon cpus. With some testing, these performance profiling tools were determined to be able to detect cycle counts with an accuracy of around ±2000 cycles on the M2 Pro processor with which we worked.

For our purposes, we would need much higher accuracy, so we decided to run all of our latency experiments as sequences of instructions within a loop. The loop would run until a total of 200000 AMX instructions were run. From this, the up to 2000 cycle count errors could be averaged over the entire program to theoretically provide approximately .1% error on the average cycle count per iteration of the loop. This should deal with random noise; however, to further improve accuracy, we ran non-AMX instruction programs with a known number of cycles through the cycle counter and applied a linear transformation on the provided cycle counts to remove any bias in the outputs from the performance profiler.

6.2 Generating AMX instruction sequence datasets

The limitation of having to run each test case for a total of 200000 AMX instructions meant that there would be difficulty in producing datasets larger than a few gigabytes. As such, the latency modeling was simplified to operate on a smaller subset of instructions. Notable ignored instructions include genlut, matint, and vecint.

Listing 6.1: Performance-case table initialization.

```
static const perfcase_t perfcases[] = {
    {"fma16_mat", 0, 32, 0x3f, 0, 15, fma16_width_modes_mat, fma_alu_modes, 1, 1},
    {"fma32_mat", 0, 16, 0x03, 0, 12, fma32_width_modes,     fma_alu_modes, 1, 1},
    {"fma64_mat", 0,  8, 0x07, 0, 10, fma64_width_modes,     fma_alu_modes, 1, 1},

    {"fma16_vec", 1ull << 63, 1, 0x3f, FLAG_VEC, 15, fma16_width_modes_vec,  fma_alu_modes, 1,
     1},
    {"fma32_vec", 1ull << 63, 1, 0x3f, FLAG_VEC, 12, fma32_width_modes,      fma_alu_modes, 1,
     1},
    {"fma64_vec", 1ull << 63, 1, 0x3f, FLAG_VEC, 10, fma64_width_modes,      fma_alu_modes, 1,
     1},

    {"mac16_mat", 0, 32, 0x01, FLAG_INT, 14, mac16_width_modes_mat, fma_alu_modes, 1, 1},
    {"mac16_vec", 1ull << 63, 1, 0x3f, FLAG_INT | FLAG_VEC, 14, mac16_width_modes_vec,
    fma_alu_modes, 1, 1},

    {"matfp", 0, 1, 0x07, 0, 21, matfp_width_modes, matfp_alu_modes, 4, 4},
    {"vecfp", 0, 1, 0x3f, FLAG_VEC, 19, vecfp_width_modes, vecfp_alu_modes, 1, 1},

    {"extr_h", 1ull << 26, 8, 0x3f, 0, 8, extr_h_width_modes, extr_h_alu_modes, 1, 1},
    {"extr_v", 1ull << 26, 8, 0x3f, 0, 9, extr_v_width_modes, extr_v_alu_modes, 1, 1},
};
```

For each instruction we store a shortened list of the allowed mixed width data modes in addition to certain ALU modes as well as associated information such as masks of the relevant instruction operand bitfields and the necessary flags or bits that must be set for each variant. We see in Listing 6.2 and Listing 6.3 a sample of the associated width and ALU modes chosen. These shortened subsets of the available variants were chosen as a subset of the variants that had the greatest impact on loop latency and performance. For example, changing the width from f16f16 to f64f64 changes the performance for many of the different matrix type instructions due to the compute difference between f16 and f64 outer products. ALU modes were also often used in ways that would change data dependencies in the program. We can see examples of skipping either X,Y, or Z register file within a standard fused multiply-add as well as changing output register pool from X to Y in Listing 6.3.

Listing 6.2: Chosen Width modes for Compute and extract instructions (excerpt).

```
static const perfmode_t matfp_width_modes[] = {
    {"bf16bf16", 0ull << 42, 32 * 32, 0x01, FLAG_M2},
    {"bf16f32" , 1ull << 42, 32 * 32, 0x00, FLAG_M2},
    {"f16f16"  , 2ull << 42, 32 * 32, 0x01, 0},
    {"f16f32"  , 3ull << 42, 32 * 32, 0x00, 0},
    {"f32f32"  , 4ull << 42, 16 * 16, 0x03, 0},
    {"f64f64"  , 7ull << 42,  8 *  8, 0x07, 0},
    {},
};
```

Listing 6.3: Chosen ALU modes for Compute and extract instructions (excerpt).

```
static const perfmode_t fma_alu_modes[] = {
    {"x*y+z",          0, 2, 0x3f, 0},
    {"x*y"  , 1ull << 27, 1, 0x3f, 0},
    {"x+z"  , 1ull << 28, 1, 0x3f, 0},
    {"y+z"  , 1ull << 29, 1, 0x3f, 0},
};
```

```c
static const perfmode_t extr_h_alu_modes[] = {
    {"x1(x)", 0, 1, 0x3f, 0},
    {"x2(x)",  1ull << 31                 , 2 , 0x1f, FLAG_M2},
    {"x4(x)", (1ull << 31) | (1ull << 25), 4 , 0x0f, FLAG_M2},
    {"x1(y)", (1ull << 10), 1, 0x3f, 0},
    {"x2(y)", (1ull << 10) | (1ull << 31), 2 , 0x1f, FLAG_M2},
    {"x4(y)", (1ull << 10) | (1ull << 31) | (1ull << 25), 4 , 0x0f, FLAG_M2},
};
```

After these choices, we iterated over the cartesian product of all of these possible instruction variants over loop lengths of 1,2,3, and 4 respectively. Since our benchmark runs these instruction sequences in a loop, we can shift all of them so that the "smallest" instruction by some predetermined ordering is first. This results in a total of approximately 38000 tests for length 2 loops and $6.6 \cdot 10^6$ tests for length 3 loops. On an M2 Pro processor, this took around 3 minutes for all of the length 2 loops and half a day for the length 3 programs. Iterating over and benchmarking all of the length 4 programs using this strategy would probably taking on the order of several months.

Notably, from small tests we confirmed that the AMX architecture is able to determine data dependencies with respect to different registers, and chose to input 0 as the register number for all choices of register in X,Y, and Z register pools for these generated test cases. This drastically reduces the size of the dataset while still allowing us to observe certain data dependency derived behavior.

6.3 Rule-Based Latency Model and Parameter Estimation

Keys and read/write sets. Each instruction instance I is mapped to a *key* $K(I) \in \mathcal{K}$. Keys can be chosen at varying granularity (e.g., as a single `inst`, a `inst` and `width mode` pair, or a `inst`, `width mode`, `alu mode` triplet). This concept of keys is introduced so that `inst`, `width mode`, `alu mode` triplets that share the exact same behavior within the benchmarking dataset can be reduced into a reduced number of equivalence classes.

Parameters. The model attaches three nonnegative quantities to keys/pairs of keys:

$$\text{Base}(K) \geq 0, \qquad \text{Full}(K) \geq 0, \qquad \text{Switch}\big(\{K_1, K_2\}\big) \geq 0,$$

where:

- BaseLatency(K) is a per-instruction *issue* (or staging) cost,
- FullLatency(K) is a *producer latency* charged when the result produced by K is consumed (one directional hop),
- SwitchLatency($\{K_1, K_2\}$) is a symmetric *context/switch* cost between two consecutive keys (unordered pair; Switch($\{A, B\}$) = Switch($\{B, A\}$)).

The SwitchLatency() term is inspired from the length 2 dataset, where we see behavior such as that in Table 6.1. In it, the same instruction when paired with itself leads to rather low loop cycle counts. This behavior extends beyond just the same instruction. By eye, it appears that there exist 3 main groups of extract instructions, floating point instructions, and integer instructions. Instructions that stay within a certain group will have often have single digit latencies, while those that cross these group boundaries appear to have much higher. Crossing smaller boundaries from say FMA_16 to FMA_64 appears to bring about smaller penalties of only a handful of cycles. In order to generalize these understandings, the model assigns some sort of switching cost between pairs of keys.

Table 6.1: Cycle latencies of length 2 instruction sequences inside a loop.

Op A			Op B			Latency (cycles)
Kernel	Mode	Expr	Kernel	Mode	Expr	
extr$_h$	f16f16	x1(x)	extr$_h$	16->16	x1(x)	2.00
fma16$_{mat}$	f16f16	x*y+z	fma16$_{mat}$	f16f16	x*y+z	8.00
mac16$_{mat}$	i16i16	x*y+z	mac16$_{mat}$	i16i16	x*y+z	7.99
fma16$_{mat}$	f16f16	x*y+z	mac16$_{mat}$	i16i16	x*y+z	28.13
mac16$_{mat}$	i16i16	x*y+z	extr$_h$	f16f16	x1(x)	22.95
fma16$_{mat}$	f16f16	x*y+z	extr$_h$	16->16	x1(x)	23.06

6.4 Fitting Latency model to Two-instruction loops

Many existing instruction latency calculation techniques techniques make use of the ability of having fine grained cycle counters [7]. Other difficulties exist in our case due to the existence of separate X,Y, and Z register pools that prevent many instructions from having self-dependencies. Because of our instruction sequence within a loop technique, even the smallest loop containing 2 instructions may have somewhat complicated data dependencies. We describe in this section the possible patterns that exist within our length 2 sequence dataset. In addition, we describe how we fit the latency model described in section 6.3 to the different dependency chain patterns.

Register roles. We consider the three architectural register files: input pools X, Y and an output pool Z. Compute instructions read from subsets of $\{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$ and *write only* to Z. Extract instructions move data from Z into either X or Y.

Instruction classes. We group opcodes by their read/write sets:

Class	Example mnemonic	Reads	Writes
c_1	$x * y$	$\{\mathrm{X}, \mathrm{Y}\}$	$\{\mathrm{Z}\}$
c_2	$x + z$	$\{\mathrm{X}, \mathrm{Z}\}$	$\{\mathrm{Z}\}$
c_3	$y + z$	$\{\mathrm{Y}, \mathrm{Z}\}$	$\{\mathrm{Z}\}$
c_4	$x * y + z$	$\{\mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$	$\{\mathrm{Z}\}$
e_1	$\mathrm{extr}(\cdot \to \mathrm{X})$	$\{\mathrm{Z}\}$	$\{\mathrm{X}\}$
e_2	$\mathrm{extr}(\cdot \to \mathrm{Y})$	$\{\mathrm{Z}\}$	$\{\mathrm{Y}\}$

Here, we choose to model only read after write data dependencies for simplicity. One can easily work out that the data dependency graphs of these loops of 2 sequences of instructions fall into one of only four different types from our simplified dataset.

Length-2 loop dependency patterns

We provide here a list of the 4 main dependency patterns and this cases that fall under this category.

(a) No dependencies (pure throughput).

$$\begin{tikzcd} A_1 \arrow[r, dashed] & B_1 \arrow[r, dashed] & A_2 \arrow[r, dashed] & B_2 \end{tikzcd}$$

Either 2 Extract functions in a row or 2 type c_1 compute functions that take in X and Y and write to Z will fulfill this.

(b) Alternating hops ($A \to B$ only).

$$\begin{tikzcd} A_1 \arrow[r, dashed] \arrow[r, bend left] & B_1 \arrow[r, dashed] & A_2 \arrow[r, dashed] \arrow[r, bend left] & B_2 \end{tikzcd}$$

Examples here include pairing a c_1 instruction with either of the c_2, c_3, c_4 functions.

(c) Size-1 chain ($A \leftrightarrow B$).

$$\begin{tikzcd} A_1 \arrow[r, dashed] \arrow[r, bend left] & B_1 \arrow[r, dashed] \arrow[r, bend right] & A_2 \arrow[r, dashed] \arrow[r, bend left] & B_2 \end{tikzcd}$$

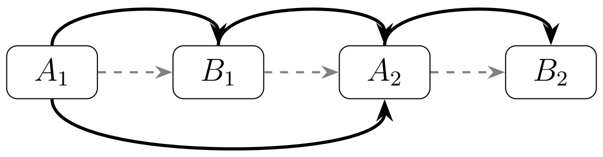
Examples here include pairing 2 instructions of type c_2, c_3, c_4. In addition, pairing a c_1 instruction with an e_2 instruction or a c_2 instruction with an e_1 instruction also suffices.

(d) Alternating hop + size-2 chain .

$$\begin{tikzcd} A_1 \arrow[r, dashed] \arrow[r, bend left] \arrow[rr, bend right] & B_1 \arrow[r, dashed] & A_2 \arrow[r, dashed] \arrow[r, bend left] & B_2 \end{tikzcd}$$

The main example here is either pairing a c_3 instruction with an e_1 instruction or c_2 instruction with an e_2 instruction.

(e) Size-1 chain + size-2 chain .



The main examples here are either c_2 e_1, or c_3 e_2, or c_4 e_1, or c_4 e_2.

Mapping to the latency model. Consider a steady two-op loop $\cdots, A, B, A, B, \cdots$ with keys K_A, K_B. Define

$$X(K_A, K_B) \;=\; \text{BaseLatency}(K_A) + \text{BaseLatency}(K_B) \;+\; 2\,\text{SwitchLatency}\big(\{K_A, K_B\}\big).$$

Let $\mathbf{1}[A \to B]$ be the indicator of a true data dependence from instruction A to B. The predicted period (cycles per loop) is

$$T(A,B) \;=\; X(A,B) \;+\; \mathbf{1}[A\!\to\!B]\cdot \text{Full}(K_A) \;+\; \mathbf{1}[B\!\to\!A]\cdot \text{Full}(K_B). \tag{6.1}$$

This specializes in the 5 cases above:

$$\begin{aligned}
\text{No dependencies:}\quad & T = X(A,B),\\
\text{Alternating hop } (A\!\to\!B \text{ only}):\quad & T = X(A,B) + \text{Full}(K_A),\\
\text{Size-1 chain } (A\!\leftrightarrow\!B):\quad & T = X(A,B) + \text{Full}(K_A) + \text{Full}(K_B).\\
\text{Alternating hop + size-2 chain }:\quad & T = X(A,B) + \text{Full}(K_A),\\
\text{Size-1 chain + size-2 chain}:\quad & T = X(A,B) + \text{Full}(K_A) + \text{Full}(K_B).
\end{aligned}$$

The final 2 cases offer a bit of a simplifying assumption that the latency is not increased by the introduction of the size-2 chain. This may not universally be correct, but in this situation the size 2 chain is due to a dependency from a compute instruction to itself from the previous loop. Taking into account the switching behavior , we decided to make this simplification as to make the model fitting process easier.

6.4.1 Fitting by Nonnegative Least Squares (Coordinate Descent)

Design matrix. For each measured two instruction sample (A, B) with observed cycle latency of $T^{\star}$, we form a sparse row in a nonnegative linear model. Let K_A, K_B be the keys and $P = \mathbf{1}[A \to B]$, $Q = \mathbf{1}[B \to A]$. We place coefficients as follows:

- $+1$ in the column of $\mathrm{BaseLatency}(K_A)$ and $+1$ in that of $\mathrm{BaseLatency}(K_B)$,
- $+2$ in the column of $\mathrm{SwitchLatency}\big(\{K_A, K_B\}\big)$ (unordered pair),
- $+P$ in the column of $\mathrm{FullLatency}(K_A)$ and $+Q$ in that of $\mathrm{FullLatency}(K_B)$.

Stacking rows yields $A\theta \approx y$ with $\theta \geq 0$, where y contains the observed cycle latencies.

Objective. We would like to solve $A\theta = y$ for θ, but a better choice is to fit θ by ridge-regularized Non-negative Least Squares (NNLS):

$$\min_{\theta \geq 0} \ \|W(A\theta - y)\|_2^2 \ + \ \lambda\|\theta\|_2^2, \qquad W = \mathrm{diag}(w_i),$$

with either *absolute* loss ($w_i = 1$) or *relative* loss

$$w_i \ = \ \frac{1}{\max(|y_i|, \varepsilon)} \quad \Rightarrow \quad \text{each row contributes a squared \% error.}$$

Coordinate Descent (CD). While the specific method of solving this NNLS problem is simply a question of what library to call, in this instance we used a fast cyclic Coordinate Descent with non-negativity projections. Let A_j be column j, $r = y - A\theta$ the residual, and $d_j = \|A_j\|_2^2 + \lambda$. A single coordinate update is

$$\theta_j \leftarrow \max\left\{0, \ \frac{A_j^\top r + (d_j - \lambda)\theta_j}{d_j}\right\}, \qquad r \leftarrow r - (\theta_j^{\mathrm{new}} - \theta_j^{\mathrm{old}})\, A_j.$$

We iterate until an iteration/time budget is reached, printing training progress (objective, median % error, within-1% rate) on a held-out split for quick monitoring.

Granularity and adaptive splitting. As described in section 6.3, keys can start at `inst` and be refined (*inst/width*, then *+alu*) only when residual analysis finds statistically significant, sustained structure across settings. In addition to keeping the model compact for training time purposes, this also guards against overfitting.

6.5 Inference for Sequences Longer Than Two (Simple Simulation)

In order to predict loop cycle counts for sequences longer than 2, we rely on simulation. Given a loop of length $L > 2$ such as $A \to B \to C \to A$, we estimate its steady-state period *ignoring instruction reordering.* This is a major simplifying assumption that is likely false. The details of instruction reordering as used in AMX are difficult to reverse engineer, but may be able to implemented here in future work. Here, we schedule strictly in program order, allowing only the delays implied by (i) per-instruction issue costs $\text{BaseLatency}(\cdot)$, (ii) symmetric switching costs $\text{SwitchLatency}\{\cdot, \cdot\}$, and (iii) full instruction latencies $\text{FullLatency}(\cdot)$ when a true data dependency exists.

What the model charges. For two neighboring instructions X then Y:

- A **base path** cost: $\text{BaseLatency}(X) + \text{SwitchLatency}\{X, Y\}$.
- If X produces a value that Y reads ($X \to Y$ by register overlap), add a dependency latency. $\text{FullLatency}(X)$.

Simulation algorithm Let the loop body be $I_0, I_1, \ldots, I_{L-1}$ and repeat it once more to $I_L, \ldots, I_{2L-1}$ with $I_{t+L} \equiv I_t$. We compute a start time $\text{exec}[t]$ for each position:

1. Initialize: $\text{exec}[0] = 0$.
2. For $t = 1, \ldots, 2L - 1$:
    (a) **Base (non-dep) path.** The next instruction cannot start earlier than

$$b_t \;=\; \text{exec}[t-1] \;+\; \text{BaseLatency}(I_{t-1}) \;+\; \text{SwitchLatency}\{I_{t-1}, I_t\}.$$

(b) **Dependency paths.** For any earlier instruction $k < t$ that *produces* a register consumed by I_t (i.e., a true data dependency $I_k \to I_t$):

$$d_{k\to t} \;=\; \text{exec}[k] + \text{BaseLatency}(I_k) + \sum_{i=k}^{t-1} \text{SwitchLatency}\{I_i, I_{i+1}\} + \text{FullLatency}(I_k).$$

This says: the consumer must wait until the producer issues, we bridge all intervening neighbor switches in order, and we pay the producer's latency once. This is where we assume that no instruction reordering is being performed, as we must pay the cost of all switch latencies between I_k and I_t. In addition, we pessimistically model the switch latencies as being unable to pass in parallel with the full instruction execution latency.

(c) **Choose the tightest start time.** We take the maximum of the time restrictions imposed by each of the dependency paths above.

$$\text{exec}[t] \;=\; \max\Big(b_t,\; \max_{k<t \text{ s.t. } I_k \to I_t} d_{k\to t}\Big).$$

3. **Period from two iterations.** The steady loop period is the largest phase-to-phase distance across one body:

$$T \;=\; \max_{0\le i<L} \big(\text{exec}[i+L] - \text{exec}[i]\big).$$

Why this works on any instruction length. The base path enforces a minimal cycle count (issue + neighbor switch), while each dependency path prevents a consumer from starting before its producer's value is ready. Taking the maximum over all applicable paths at each step yields a safe schedule in program order. Because we only ever reference BaseLatency$(\cdot)$, SwitchLatency$\{\cdot,\cdot\}$, and an instruction's FullLatency$(\cdot)$, the very same procedure scales to any loop length L. Two-iteration unrolling and phase-to-phase differences then give a stable estimate of the loop period for validation against our loop-based benchmarking dataset. However, this can also be done for an arbitrary non-branching basic block of AMX instructions.

Future Refinement. If work continues in this rule-based methodology. It seems likely that different variations of this existing simulation process should be used and compared against the observed behavior. Iterating over this step can lead to improved understanding of the architecture and performance characteristics of the AMX accelerator. Of course, for optimal performance, understandability can be sacrificed by introducing Transformer based models with much larger expressive power. This would however also require solving an issue of being able to generate a useful dataset of sufficiently large size to train the ML model.

6.6 Empirical Results

We evaluate the rule-based latency model on two datasets: (i) length-2 loops, on which the model is fit, and (ii) length-3 loops, which are *unseen* during training and exercise the inference procedure from Section 6.5. We report both relative (% error) and absolute (cycles) metrics.

6.6.1 Aggregate Accuracy

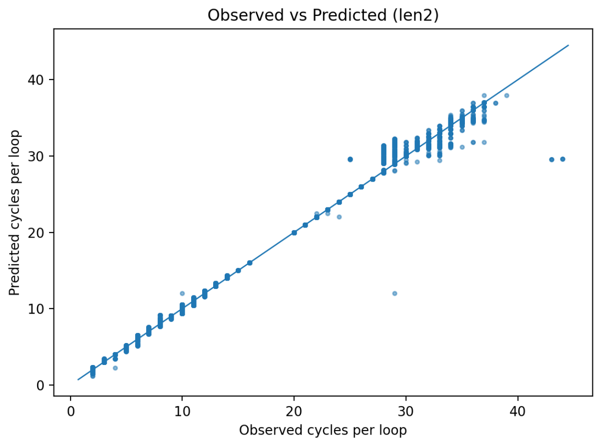
Table 6.2 summarizes performance on both datasets. We measure Mean average error and RMSE for both the difference from observed baseline as well as a percentage difference. The model is highly calibrated on length-2 (train) and retains some accuracy on length-3 (validation). In particular, we see both the model is able to predict within 5% accuracy on the cycle count of the length-3 dataset more than 70% of the time.

6.6.2 Scatter plots

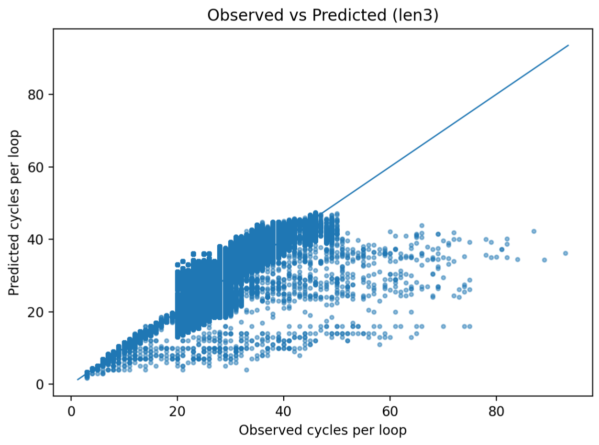
Figure 6.1 overlays predicted vs. observed loop periods with a $y = x$ reference line. For length 2 instruction sequences, we see near perfect prediction of the cycle count for the loop. Some accuracy is lost for larger cycle counts. This may not be surprising as our dataset generation method begins to lose accuracy due to noise as the cycle counts go up to 40 cycles and more. For the length 3 dataset, we see that a cloud of points located along the $y = x$ reference line, however there seem to be a number of outliers with higher observed cycle counts than we

Table 6.2: Empirical accuracy on length-2 (train) and length-3 (validation) loops.

Metric	Length-2	Length-3
MAE (%)	0.432	4.826
RMSE (%)	1.753	9.103
Within 1%	0.909	0.488
Within 2%	0.933	0.588
Within 5%	0.969	0.707
MAE (cycles)	0.070	1.355
RMSE (cycles)	0.354	2.525
Exact int match	0.963	0.582
Off-by-1 (int)	0.987	0.715



(a) Length-2



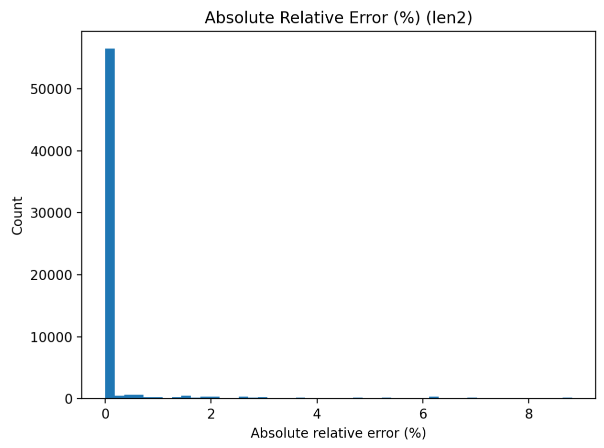
(b) Length-3

Figure 6.1: Observed vs. predicted cycles per loop .

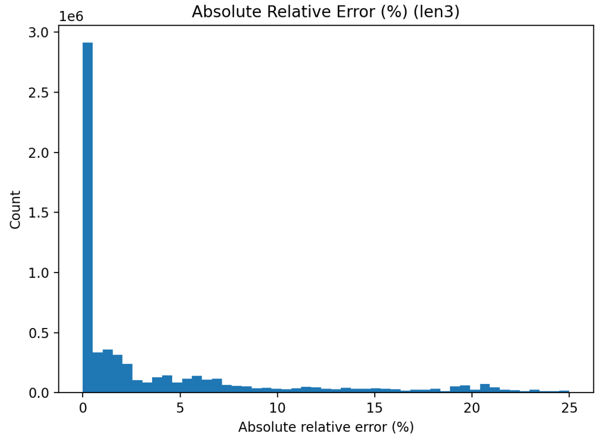
have predicted. This indicates existence of certain higher order complexities within cycle count prediction that exist beyond our simple extended latency based model.

6.6.3 Error Distributions

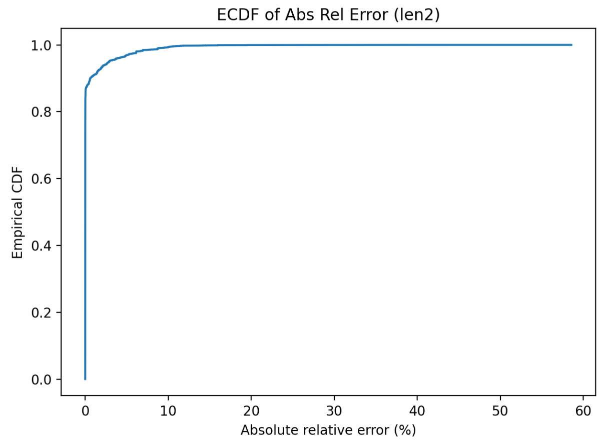
To characterize uncertainty, Figure 6.2 shows the histogram and ECDF of absolute relative error. Length-2 errors are tightly concentrated below 1–2%, while length-3 exhibits a heavier tail as expected. The integer error histograms visualize how often rounded predictions match rounded observations exactly or within 1 cycle.



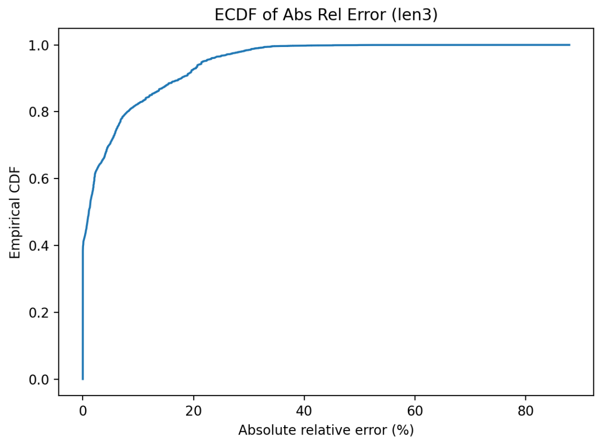
(a) Rel. error hist (Len-2)



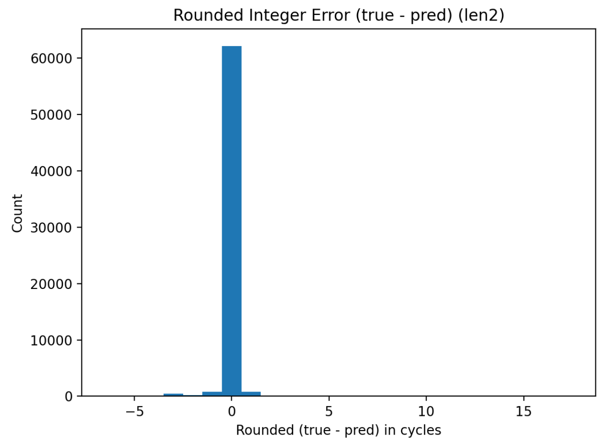
(b) Rel. error hist (Len-3)



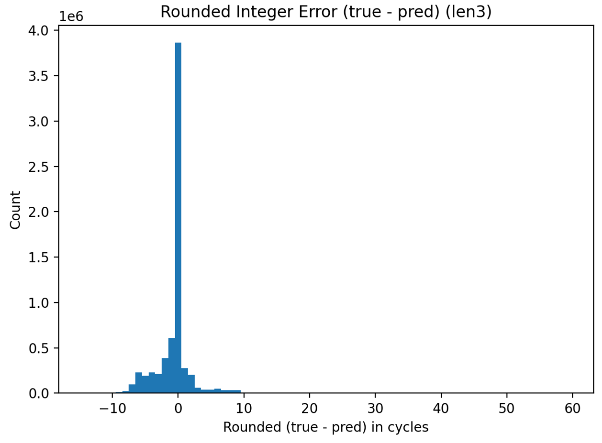
(c) ECDF (Len-2)



(d) ECDF (Len-3)



(e) Integer error (Len-2)



(f) Integer error (Len-3)

Figure 6.2: Error distributions of the cycle count predictor model on length 2 and 3 datasets. Integer error uses rounded cycles.

6.7 Future work

This latency model begins to explain some of the latency behaviors involving data dependencies between different AMX instructions. However, a more in depth analysis is needed of the inconsistencies between the predicted cycle counts of the length 3 dataset compared to the observed data.

In addition, a more comprehensive model could be created that combines the observations from this existing data dependency viewpoint in addition to other known AMX behaviors. For example, output register throughput limitations from section 4.4 as well as instruction reordering are known behaviors that are not modeled in this simplified model.

Chapter 7

Conclusion

We have examined how to unlock the Apple Matrix Coprocessor's performance beyond the use of Accelerate through three complementary threads: measurement, design, and modeling. Our throughput microbenchmark study clarified basic throughput values for both data movement and different compute instructions. Building on these findings, we looked at a case study of optimizing general matrix multiplication(GEMM)and designed an in-place GEMM that handles non-multiple-of-8 sizes with masked outer products and strategically overlapped tiles, avoiding scratch buffers and outperforming Accelerate in certain cases.

Finally, we introduced a simple interpretable rules-based latency model that decomposes cycle counts into base dispatch time, symmetric switching cost, and instruction-specific dependency latency. Trained on length-2 loops and validated on length-3 sequences through a simple loop-simulation, the model achieved high accuracy with modest complexity, while offering improved understanding of both the AMX architecture and how to write performant AMX code.

Limitations. Our measurements and models target a specific Apple Silicon generation; absolute constants may differ on newer SoCs. The latency cycle prediction inference procedure currently ignores potential instruction reordering by a small hardware buffer and does not model multi-core contention or output register throughput limits. Some rare instruction combination interactions remain under-sampled and may still lie undiscovered.

Future work. Promising directions include: (i) extending the simulator to incorporate a bounded reorder buffer and contention on shared compute units; (ii) cross-generational retuning and transfer learning for M3/M4; (iii) integrating the latency model into a compiler pass for AMX-targeted scheduling and tiling; and (iv) broadening case studies (e.g., batched GEMM, attention primitives) and exploring other important applications of the AMX coprocessor.

References

[1] P. Cawley. *Corsix-AMX*. https://github.com/corsix/amx. 2024.

[2] D. L. G. Filho, G. Brandão, and J. López. "Fast polynomial multiplication using matrix multiplication accelerators with applications to NTRU on Apple M1/M3 SoCs". In: *IACR Communications in Cryptology* 1.1 (Apr. 9, 2024). ISSN: 3006-5496. DOI: 10.62056/a3txommol.

[3] D. L. G. Filho, G. Brandão, G. Adj, A. Alblooshi, I. A. Canales-Martínez, J. Chávez-Saab, and J. López. *PQC-AMX: Accelerating Saber and FrodoKEM on the Apple M1 and M3 SoCs*. Cryptology ePrint Archive, Paper 2024/195. 2024. URL: https://eprint.iacr.org/2024/195.

[4] K. Harrison. Dec. 2023. URL: https://github.com/keith/dyld-shared-cache-extractor.

[5] In: *apple* (Jan. 2023). URL: https://www.apple.com/newsroom/2023/01/apple-unveils-m2-pro-and-m2-max-next-generation-chips-for-next-level-workflows/.

[6] D. Lemire. *Counting cycles and instructions on ARM-based Apple systems*. Mar. 2023. URL: Daniel%20Lemire%E2%80%99s%20blog.

[7] A. Abel and J. Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. ACM, Apr. 2019, pp. 673–686. DOI: 10.1145/3297858.3304062. URL: http://dx.doi.org/10.1145/3297858.3304062.