TR: Debunking the Myths of Approximate Matrix Multiplication on Contemporary Processors

Anonymous Authors

Abstract

Approximate Matrix Multiplication (*AMM*) is an attractive technique for enhancing the speed of data-intensive applications. While substantial research has gone into devising *AMM* algorithms with lower theoretical runtime complexity, there is a conspicuous absence of empirical studies examining their practical runtime behavior on contemporary processors. To bridge this gap, we undertake a comprehensive evaluation of *AMM* algorithms, utilizing 8 real-world datasets across 4 different application domains. We categorize these algorithms into *pruning-based*, *extraction-based*, and *hybrid* types, and rigorously assess 12 representative algorithms implemented within the LibTorch framework to ensure fair comparisons of *AMM* algorithms. Through hardware profiling, we dissect the computational and memory access patterns inherent to each. Our findings reveal that: a) *AMM* can significantly accelerate some specific machine learning and scientific computing tasks, but a thoughtful algorithm selection and optimization is still required for its wider adoption; b) pruning-based and hybrid *AMM* generally outperform extraction-based algorithms, mainly because the latter incurs significant overhead in computation, memory, and flow control; c) *AMM* is predominantly memory-bound, emphasizing the need for memory optimization in future research.

1 Introduction

Matrix Multiplication (*MM*) serves as a cornerstone in a myriad of data-intensive applications, spanning domains from statistics and machine learning to scientific computing [38, 43, 53, 61]. Its inherent computational complexity, typically described by three tiers of nested loops and cube complexity, often makes it a significant bottleneck in these applications. For example, our evaluation demonstrates that *MM* can account for more than 85% of the total processing time in certain real-world applications (see Section 6.1 for further details).

To mitigate this bottleneck, Approximate Matrix Multiplication (*AMM*) has emerged as a crucial strategy [11, 12, 18, 21, 25, 28, 29, 36, 40, 41, 44, 47, 56, 58]. Unlike optimizations focused solely on the exact computation of *MM* loops [31, 32] or sparse matrix data exploitation [24, 57], *AMM* introduces approximate computing as an orthogonal avenue for improvement. The central aim of *AMM* is to find a middle ground between computational accuracy and efficiency. It accomplishes this by intentionally reducing the mathematical accuracy of *MM* operations in favor of significantly lower running overhead. In today's big data era, the development and stringent evaluation of *AMM* methods, which provide high-quality approximations with minimal overhead, have become critically important.

While there is a considerable volume of research on *AMM* algorithms, existing studies face certain limitations that question their comprehensive utility. First, there is a notable narrowness in the evaluated scenarios of *AMM* algorithms in many existing works. These are often validated only in highly specialized application contexts [11, 44, 56] or, in some instances, solely on synthetic datasets [21, 28, 42, 58]. Such constraints raise concerns about the broader real-world applicability of *AMM* algorithms. Second, the literature shows a marked propensity to prioritize theoretical complexity analysis, usually conveyed through *Big O Notation* [28, 29, 42, 44, 47, 56, 58]. Although theoretical analysis is valuable, focusing exclusively on it overlooks the subtleties of real-world hardware interactions [45, 49]. In essence, there is a critical need for more empirically grounded evaluations of *AMM* algorithms to bridge the gap between theoretical underpinnings and practical application scenarios.

Given these gaps in current research, our empirical study grapples with a unique set of challenges (***C1*** through ***C3***) aimed at providing more grounded, practical evaluations of *AMM* algorithms. ***C1***: The first challenge lies in the classification and selection of *AMM* algorithms for a rigorous evaluation. There is no universally accepted classification of *AMM* algorithms, and we propose to categorize *AMM* algorithms into three foundational approaches: *pruning-based*, *extraction-based*, and *hybrid*. Pruning-based *AMM* aims to reduce computational overhead by judiciously removing unnecessary or redundant matrix information before conducting matrix multiplication. Extraction-based *AMM*, on the other hand, focuses on capturing higher-level matrix features to leverage for accelerated computation. Hybrid *AMM* seeks to optimize both accuracy and computational overhead by balancing between pruning and extraction. Each category comes with its own unique set of design challenges and trade-offs, complicating the evaluation process. Within these approaches, different implementation strategies exist, adding another layer of complexity to our evaluation. For example, features can be extracted into matrices [44] or codebooks [18, 40], *and hybrid AMM can either use pruning to accelerate extraction or using extraction to refine pruning.*

C2: A second critical challenge arises from the scant empirical analysis in existing *AMM* research. While numerous studies propose theoretical improvements [11, 12, 19, 21, 25, 28, 29, 36, 42, 56, 58], there is a lack of empirical assessments coexists with inconsistencies across the literature, such as varying compilation of *AMM* algorithms. For example, the mixed just-in-time (JIT) and static compilation [11, 18, 44] employed by existing works complicates the task of direct comparisons. Moreover, some studies utilize brute-force *MM* as their baseline [42, 58], which may produce misleading conclusions. These inconsistencies and the lack of comprehensive empirical validation obstruct the development of reliable guidelines for applying *AMM* in various matrix configurations and application contexts.

C3: The role of contemporary hardware in the performance of *AMM* remains inadequately understood. While most existing studies on *AMM* emphasize theoretical complexity analysis to make a case for lower computational overhead [28, 29, 42, 44, 47, 56, 58], they often neglect two vital aspects critical for performance on modern hardware. The first is an evaluation of *memory access behaviors* during both matrix manipulation and the manipulation of intermediary data structures. The second is the quantification of computational efforts, which include not just *mathematical operations* but also *program flow control*. Moreover, hardware-specific attributes like cache utilization and out-of-order execution, can further significantly influence real-world performance.

To the best of our knowledge, this study represents the first empirical evaluation of the utility of *AMM* algorithms on contemporary processors. Our benchmark is both source-code and binary compatible with LibTorch and its Python frontend, PyTorch [8]. In response to ***C1***, we implement 12 *AMM* algorithms spanning *pruning-based*, *extraction-based*, and *hybrid* categories, and integrate them into a unified codebase. To address ***C2***, we standardize function calls and data structures by leveraging the static compilation features of the LibTorch C++ API [8]. This minimizes implementation discrepancies and ensures methodological consistency. Our evaluation is based on 8 representative real-world datasets and encompasses 4 distinct downstream tasks, providing a thorough assessment of *AMM* algorithms. For ***C3***, we consider in-depth hardware architectural profiling in our evaluation. Owing to our use of static compilation, the hardware counter metrics we obtain are strictly attributable to the *AMM* algorithms. This level of isolation is crucial for in-depth insights into the computational and memory-access characteristics intrinsic to *AMM* algorithms.

Our investigation acknowledges several orthogonal facets such as parallelization, resource utilization, out-of-memory processing, and error correction that are well-covered in the literature [15, 33, 37, 48, 51, 55, 59]. These elements, while crucial to the broader matrix multiplication ecosystem, are beyond this study's scope, which is narrowed to dissecting computational and memory access patterns unique to *AMM* on contemporary processors to better understand their strength and limitations on real-world application scenarios. This delimitation provides focused empirical performance benchmarks, circumventing the confounding variables presented by these orthogonal aspects. Insights derived may inform future research into *AMM* implementations across specialized hardware and parallel computing paradigms, though our study does not extend to hardware-level parallelism or bespoke digital logic designs.

To sum up, the key contributions of the paper are:

- In Section 3, we delve into our analysis of various studied algorithms. This includes an exploration of three fundamental approximation approaches (pruning-based, extraction-based, and hybrid) spanning twelve *AMM* algorithms. Additionally, we introduce two *MM* baseline methods, i.e., the brute force approach [28, 42, 58] and the optimized implementation used in PyTorch [8].
- Section 4 is dedicated to presenting the implementation details of our benchmark suite, alongside an overview of the performance metrics we've examined. Our benchmark has been thoughtfully designed to facilitate a comprehensive and fair comparison.
- In Section 5, we conduct a detailed examination of the *AMM* algorithms from three distinct perspectives: their practical applicability, interaction with modern hardware, and sensitivity to varying workloads or algorithm configurations. Notably, our findings 1) highlight the practical advantages of pruning-based and hybrid *AMM* over extraction-based *AMM*, while also 2) revealing the common nature of existing *AMM* algorithms being memory-bounded.
- In Section 6, we extend our investigation to explore the utilization of *AMM* in four different statistical and machine learning downstream tasks. We observe that *AMM* is especially attractive in machine learning inference and the unitary transformation problem in scientific computing, i.e., reducing up to 80% and 94% processing latency of the original exact computations, respectively, despite a set of factors hindering its border application.

2 Background

This section delves into the fundamentals of Approximate Matrix Multiplication (*AMM*), and the underlying principles of modern processor models that play a critical role in the execution of *AMM*.

Table 1. Notations used in this paper

Notation	Descriptions
A, B	Two matrixes to be multiplied
MM	The computation of accurate matrix multiplication
AMM	The computation of approximate matrix multiplication
l	The processing latency of $MM(A, B)$ or $AMM(A, B)$
ϵ	The relative frobenius error calculated as $\|\|AMM(A, B) - MM(A, B)\|\|_F / \|\|MM(A, B)\|\|_F$
e	The error of downstream tasks introduced by involving AMM
ω	The algorithm-specific tuning knob of AMM algorithms to control the preserved information

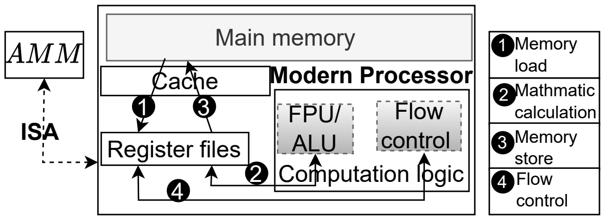


Figure 1. Interaction of *AMM* and modern processor.

2.1 Key Terms and Definitions

We summarize notations of this study in Table 1. Matrix multiplication (*MM*) is the operation of multiplying two matrices A and B, a computationally intensive task with substantial relevance across various domains. *AMM* is introduced as an efficient alternative, aiming to reduce computational demands by **approximating the product of** A **and** B **with acceptable accuracy trade-offs**, particularly beneficial in data-intensive applications.

The critical performance metrics for *AMM* are *Processing Latency* (l), *AMM Error* (ϵ), and *Application Error in Downstream Tasks* (e). l indicates the time efficiency of *AMM*. ϵ, calculated by $\frac{\|AMM(A,B)-MM(A,B)\|_F}{\|MM(A,B)\|_F}$, measures the accuracy of *AMM* relative to standard *MM*. The e reflects the impact of *AMM* on downstream applications. The tuning parameter (ω)[1] in *AMM* algorithms allows for adjusting the balance between l and ϵ. In addition, we also compare the *Energy Consumption* among *AMM* and *MM* algorithms.

AMM algorithms, traditionally analyzed and optimized in big O notation, encounter limitations when applied to real-world processor architectures [14]. While O provides a theoretical framework for understanding mathematical calculation aspects of *AMM* [28, 29, 42, 44, 47, 56, 58], it often overlooks the complexities of memory access and flow control, crucial in the context of modern processors.

[1]The details of ω, unique to each algorithm, will be expounded upon in Section 4.1.

2.2 Interacting with the Contemporary Processor

This study generalizes the interaction between *AMM* and modern processors, as illustrated in Figure 1. Processors typically employ a hierarchical storage model that includes registers, cache, and main memory, with the fastest operations occurring at the register level. These features are integral across a variety of processor units, such as CPUs [2, 6, 10, 34], GPUs [7], DSPs [3], and high-performance MCUs [4, 5], thereby broadening the scope of our analysis.

The interaction of *AMM* with processors involves manipulating their logic through an Instruction Set Architecture (ISA). Four primary types of ISA utilization for *AMM* are identified. **Memory Load (❶):** Fetching input matrices and intermediate data from main memory, which may be too large to fit into processor registers, with cache serving as an intermediary. **Mathematical Calculations (❷):** Performing operations like addition and multiplication within the register files once data is loaded. Processors such as X64 [34] and ARM-V8 [2] have specialized units like the Floating-Point Unit (FPU) and Arithmetic Logic Unit (ALU) for these tasks. **Memory Store (❸):** Writing modified data back to main memory, for either output production or updating intermediate structures. **Flow Control (❹):** Extensive use of programming constructs such as loops, branches, and pointers for algorithmic execution, distinct from mathematical computations.

In the following, the term *Memory Access* [52] will be used to collectively describe both memory load and store operations. Processes involving mathematical calculations and flow control, on the other hand, will be referred to as *Computation Effort* [14]. *Memory Access* is typically slower than the in-register *Computation Effort*, due to the hierarchical storage architecture. These terms highlight specific aspects of *AMM*'s performance profile that are often underrepresented in conventional big O notation analyses, especially when considering the advanced capabilities like SIMD and out-of-order execution prevalent in modern processor architectures.

3 Studied Algorithms

Based on how to handle the matrix information carried by input matricies A, B, we broadly classify *AMM* algorithms into three categories: pruning-based, extraction-based, and hybrid. We summarize representative *AMM* algorithms and the *MM* baselines in Table 2.

3.1 Pruning-based *AMM*

The pruning-based *AMM* involves the judicious removal of redundant information from A, B, aiming to alleviate memory constraints and computational demands, either at the level of individual bits or entire elements. Bit-wise pruning compresses each matrix element by employing

Table 2. *AMM* algorithms and *MM* baselines investigated

Category	*Algorithm Name*	*Descriptions*
Purning-based *AMM*	INT8 [35]	Purning 32-bit into 8-bit
	CRS [25]	Purning elements by sampling
	CS [29]	Purning elements by sketching
Extraction-based *AMM*	CoOFD [58]	Extracting singular value, for entire matrices
	BlockLRA [44]	Extracting singular value, for blocks
	FastJLT [12]	Extracting JL embeddings
	VQ [40]	Extracting KNN centroids
	PQ [18]	Similar to VQ, more efficient codebook
Hybrid *AMM*	RIP [36]	Randomized JL embeddings extraction
	SMP-PCA [56]	Similar to RIP, scaling values for higher accuracy
	WeigthedCR [21]	Extract the weight information during sampling
	TugOfWar [47]	Extract the median and select the optimal after sketching
Baseline *MM*	NLMM [58]	The manual, brute-force, nested loop implementation of *MM*
	LTMM [8]	Libtorch's optimized implementation of *MM*

fewer binary bits. For instance, INT8 [35] quantizes 32-bit floating-point elements into 8-bit signed integers. On the other hand, element-wise pruning preserves the full binary representation for selected elements and discards the rest completely. This process of element removal can be accomplished through methods like CRS (**c**olumn **r**ow **s**ampling) [25] or count sketching structures within CS (**c**ount **s**ketch-based *AMM*) [29]. Pruning-based *AMM* algorithms are usually lightweight, but they rely on certain assumptions of data distributions to maintain the accuracy.

3.2 Extraction-based *AMM*

Extraction-based *AMM* captures inherently higher-level characteristics from matrix elements, maintains the extracted characteristics in their intermediate structures, and leverages the intermediate structures for carrying out *AMM* computations. This strategy is adopted as computing on these intermediate structures is notably faster than processing $MM(A, B)$. Diverse matrix attributes are available to achieve this objective. Frequent Direction [39]-based methods, exemplified by CoOFD [42], iteratively extract significant singular values from the input matrices themselves. The BlockLRA [44] algorithm makes use of the singular value decomposition of disjoint blocks within input matrices. This approach introduces a trade-off between precision and reduced computational load compared to performing singular value decomposition on the whole input matrices as CoOFD. Another distinct approach is adopted by FastJLT [12], which discretely extracts Johnson-Lindenstrauss (JL) embedding characteristics of the input matrices by employing Walsh-Hadamard transformations. Besides intermediate structures based on matrices, integrating a codebook that represents K-nearest neighbor (KNN) centroids of matrix rows or columns proves advantageous for information extraction in *AMM* computations. This concept is realized by VQ (**v**ector **q**uantization)[40] and PQ (**p**roduct **q**uantization)[18]. PQ generally outperforms VQ in speed for large input matrices due to its further optimizations, i.e., the cartesian product of subspaces [40], in reducing the number of KNN centroids. Compared with pruning-based algorithms, extraction-based approaches offer superior theoretical accuracy as they retain more valuable information during higher-level feature extraction. Nevertheless, the amount of computation effort and memory access in the process of feature extraction might even exceed that in $MM(A, B)$.

3.3 Hybrid *AMM*

Hybrid *AMM* heuristically leverage pruning and extraction to trade off processing latency and accuracy. Some of the hybrid *AMM* methods seek to expedite the feature extraction process by allowing information pruning. Notably, RIP [36] and SMP-PCA [56] randomizes the JL embeddings extraction process, which is deterministic in FastJLT, for the speed-up. SMP-PCA involves an extra value scaling step after the randomly pruned JL transform compared with RIP, leading to higher accuracy theoretically. The rest of hybrid *AMM* explores feature extraction to refine element-wise pruning. For instance, WeigthedCR [21] potentially enhances the accuracy by utilizing the extracted weight information to conduct sampling, instead of a random one as CRS does. Similarly, TugOfWar [47] approach determines the optimal sketch that lost the least amount of information among a set of random sketches, rather than the arbitrary sketch in CS. By taking advantage of both pruning and extraction, hybrid *AMM* potentially achieves improved overall performance.

3.4 Baseline *MM*

For a comprehensive evaluation, we select two *MM* implementations as the baselines. The first one is a crude, brute-force, nested loop implementation, it explicitly aggregates the outer products of the rows of A and B without any further optimizations. We call it NLMM (**n**ested **l**oop *MM*), which has been used as the baseline in a number of prior work [28, 42, 58]. We are aware that *MM* is highly optimized in state-of-art statistic and machine learning libraries. Therefore, we also select the LibTorch's implementation, specifically the *torch::matmul* C++ API as the second baseline, namely LTMM (**L**ib**T**orch's *MM*). In particular, LTMM involves LibTorch's comprehensive optimizations of cache utilization and SIMD instructions.

3.5 Implementation Details

All studied *AMM* algorithms have been integrated into a unified C++ codebase, employing static compilation techniques, as opposed to Just-In-Time (JIT) compilation. We adopt the IEEE 754 32-bit floating-point (FP32) standard as the default binary format for matrix elements. The implementations, barring NLMM and INT8 (i.e., which computes by 8-bit integers instead of FP32), leverage the C++ API of LibTorch [8], which inherently utilizes AVX-512 for FP32 operations.

We acknowledge the existence of additional ad-hoc optimization techniques for certain *AMM* algorithms, such as the incorporation of Bernoulli sampling probability in CRS [11] and the utilization of so-called *MADDNESS* hash function in PQ [18]. Nevertheless, they enforce additional strong assumptions that lack generality and lead to restricted applicability, such as "independent input matrix rows and presciently bounded singular value" for using the *MADDNESS* hash function [18]. Therefore, we exclude these scenario-specific techniques from our implementation.

4 Evaluation Methodology

In this section, we introduce our evaluation methodology.

4.1 Tuning Knob ω of *AMM* Algorithms

The adaptability inherent in numerous *AMM* algorithms allows for a configurable balance between l and ϵ. This balance is typically managed via the tuning knob ω. We summarize the meaning of tuning knob ω for specific evaluated *AMM* algorithms as follows.

- ω has no impact on INT8, LTMM and NLMM, since these algorithms do not involve any parameter tuning.
- CRS, CS, WeigthedCR, and TugOfWar utilize ω to control the proportion of elements that are not pruned [25, 29, 44, 58]. For instance, $\omega = 10\%$ refers to preserving 10% elements in both A, B.
- CoOFD and FastJLT use ω to adjust the column volume of the extracted features [44, 58] in proportion to that of input matrix A. The same principle applies to hybrid *AMM* RIP and SMP-PCA, which involve randomized optimizations during a similar feature extraction process [36, 56].
- BlockLRA first calculates the eigenspace of A, B, and then the feature extraction matrix is sized in proportion to ω relative to the calculated eigenspace [44].
- In VQ and PQ, ω adjusts the number of KNN centroids in the proportion of rows in the input matrix A [18].

In the following, we set ω as 10% if not otherwise specified.

4.2 Hardware platform

For the experiments, we used an Intel Xeon Silver 4310 processor based on the Ice Lake micro-architecture. This processor serves as an apt platform for our study, focusing on the intrinsic computational efforts and memory access patterns of *AMM*, as outlined in Section 2.2. The server has a L3 cache size of 18MB and a memory capacity of 32GB. It operates on the Ubuntu 22.04 system and uses the g++ 11.3.0 compiler for the compilation of the source codes. Note that, the processor features AVX-512-based SIMD and involves an out-of-order execution manner of instructions.

We utilize PAPI [20] to measure the native performance counters of the processor. While our analysis does not extend to Xeon-specific or Ice-Lake-specific micro-operation efficiencies [49], the chosen processor offers valuable insights due to its two key features. First, the processor is able to categorize and count executed instructions with high granularity. It provides a detailed view of the computational processes at play, crucial for understanding the intricacies of *AMM* execution. Second, the processor's ability to trace running cycles comprehensively, including key components like the memory subsystem, allows for an in-depth analysis of functional bottlenecks in *AMM* execution.

Table 3. Real-world Workloads for Comparing *AMM* Algorithms

Name	Application Field	Size	#Matricies
ECO	Economics	207×260	2
DWAVE	Integrated Circuit	512×512	2
AST	Astrophysics	765×765	1
UTM	Plasma Physics	1700×1700	2
RDB	Chemical Engineering	2048×2048	2
ZENIOS	Air Traffic	2873×2873	1
QCD	Quantum Physics	3072×3072	2
BUS	Land Traffic	4929×10595	1

4.3 Workloads

Our comprehensive benchmark includes a diverse set of **real-world workloads** from MatrixMarket [1], such as *ECO*, *DWAVE*, *AST*, *UTM*, *RDB*, *ZENIOS*, *QCD*, and *BUS* (see Table 3). All workloads have their matrix element normalized into $-1 \sim +1$. These matrices span different domains and sizes, from the economic model of *ECO* (a 207×260 matrix) to the larger scale power flow problem represented by *BUS* (sized 4929×10595). The workloads cover a variety of applications, including circuits, physics, chemistry, and traffic control problems. Notably, *ECO* and *UTM* have a biased distribution of non-zero values, with *UTM* exhibiting higher skewness in these values. Note that, we perform $MM(A, A^T)$ or $AMM(A, A^T)$ on single-matrix workloads for self-correlation studies. For two-matrix workloads with matrices A and B, we execute $MM(A, B^T)$ or $AMM(A, B^T)$ for mutual correlation. To further test *AMM* under controlled conditions, we generate **synthetic workloads** using LibTorch functions [8], such as *torch::rand* to generate random matrices with adjustable data scale.

5 How Does AMM Perform?

This section presents the comprehensive evaluation of *AMM* algorithms. Section 5.1 provides a detailed comparison of *AMM* algorithms across a range of real-world workloads, presenting a broad perspective on their performance. In

Section 5.2, the focus shifts to an in-depth, hardware-conscious profiling study of *AMM*. Section 5.3 is dedicated to a sensitivity analysis, examining how variations in workload characteristics and the tuning parameter ω affect *AMM* algorithms. We summarize the key findings in Section 5.4.

5.1 Overall Performance

We first show the overall performance of all algorithms, concerning both processing latency and accuracy. We additionally report the energy consumption for a more comprehensive comparison.

5.1.1 Processing Latency Comparison. We illustrate the resulting processing latency (l) through Figure 2(a). We made the following three observations:

First, pruning-based *AMM* methods such as CRS and INT8 consistently demonstrate superior performance or at least comparability at l compared to both other *AMM* categories and the two baseline *MM* methods across most datasets. Notably, their advantages over *MM* become more pronounced in larger datasets like *QCD* and *BUS*. For instance, on *ECO*, CRS reduces l by 56.3% compared to NLMM and 14.3% compared to LTMM. This benefit becomes much more substantial on *BUS*, i.e., 99.9% and 98.7%.

Second, extraction-based *AMM* strategies often yield unacceptably high l, unless dealing with sufficiently large datasets. For instance, CoOFD always lags behind the stronger *MM* baseline LTMM and only surpasses the weaker baseline NLMM in scenarios like *QCD* and *BUS*. PQ is perhaps the most effective extraction-based *AMM*, but still always slower than the pruning-based *AMM* CRS.

Third, hybrid *AMM* algorithms yield moderate l values compared to pruning-based and extraction-based counterparts, and they are more appealing to adopt with matrix element volume increases. A notable example is SMP-PCA, which leads to 5× more l compared with LTMM on *ECO* but emerges to be 97% less l on *BUS*.

5.1.2 Accuracy Comparison. Figure 2(b) shows the error (ϵ) of each algorithm, revealing three crucial insights.

First, we observe that extraction-based *AMM* (i.e., CoOFD, BlockLRA, FastJLT, VQ, and PQ) usually outperforms the pruning-based *AMM* which prunes the elements (i.e., CRS and CS) in terms of accuracy across all evaluated datasets. For instance, CRS results in an ϵ surpassing 0.99 in the *BUS* dataset, while extraction-based *AMM* demonstrates an ϵ as low as 3×10^{-6} (BlockLRA). However, the bit-wise pruning approach INT8 can also achieve a low error among all datasets stably.

Second, hybrid *AMM* methods usually attain a balanced level of accuracy as anticipated, particularly benefiting from the judicious fusion of pruning and extraction strategies. In particular, the ϵ of SMP-PCA is especially low in *ZENIOS* (1×10^{-5}), which is even better than some extraction-based *AMM*, such as CoOFD (0.03) and PQ (0.01).

Finally, in addition to the salient distinctions above, all of our evaluated *AMM* algorithms except INT8, yield elevated errors (i.e., more than 1.0) in *UTM* datasets. This is because *UTM* involves both 1) a highly skewed distribution of numeric value and 2) a high level of biased non-zero region, or equivalently, low symmetry of matrices. Please refer to Sections 5.3.3 and 5.3.4 to delve deeper into these two factors.

5.1.3 Energy Consumption Comparison. We record the energy consumption of each algorithm through MSR registers [9]. The default policies of frequency scaling in the Linux 5.15 kernel are applied, and static energy consumption related to the operating system and other software is excluded.

The result is depicted in Figure 2(c), and the trend is similar to the l comparison in Figure 2(a) in general. Specifically, both pruning-based *AMM* and hybrid *AMM* significantly reduce energy consumption when the data scale is sufficiently large (i.e., thousands of rows and columns or more). For instance, CRS and SMP-PCA reduce 97% and 82% energy consumption on *BUS* compared with LTMM, respectively. Nevertheless, extraction-based *AMM* is much less energy efficient than LTMM, with up to 51× more energy consumption for CoOFD on *QCD*.

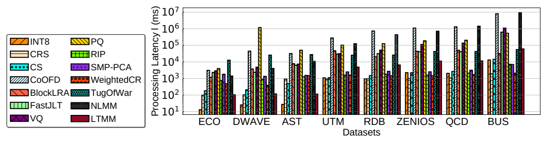
Additionally, minor variations in patterns between energy consumption and processing latency are observed (e.g., CoOFD vs. PQ on *DWAVE*), as computational efforts typically consume more power than memory access [60]. The evaluated algorithms involve different proportions of these components (refer to Section 5.2.1 for further details). It's worth noting that more fine-grained energy modeling and energy management for *AMM* are anticipated as part of future research endeavors.

5.2 Hardware-Conscious Profiling

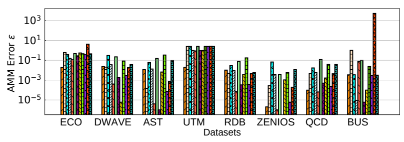
Following the processor model outlined in Section 2.2, we delve into software-hardware interaction of *AMM*, specifically the ISA utilization, performance bottlenecks, and execution efficiency.

5.2.1 ISA Utilization. We present a comparative analysis of instruction counts and a detailed breakdown of running *AMM* on the modern processor in Figure 3. We use the *BUS* dataset as the example and other datasets share similar outcomes to *BUS*. We are concerned about ISA utilization of memory load (*Mem-load*), mathematical calculations (*Math*), memory-store (*Mem-store*), and flow control (*Flow*). We further break the *Math* part into scalar (*Math-S*) and vector execution (*Math-V*), and separate the branch instructions (*Flow-B*) from other flow control efforts (*Flow-O*) such as unconditional function call and pointer update.

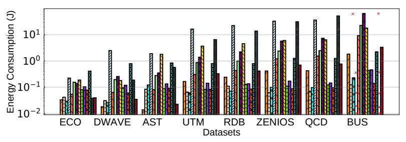
There are five major takeaways as follows. First, a reduced O complexity does not necessarily translate to a reduced instruction count (Figure 3(a)), especially for extraction-based *AMM* with matrix-based feature extraction data



(a) Processing latency l.

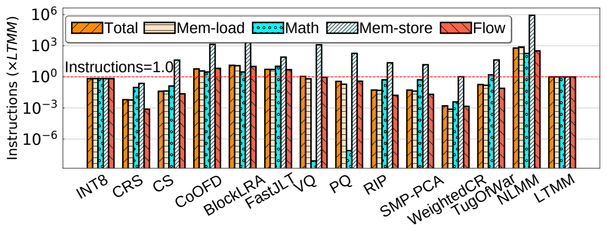


(b) *AMM* Error ϵ.

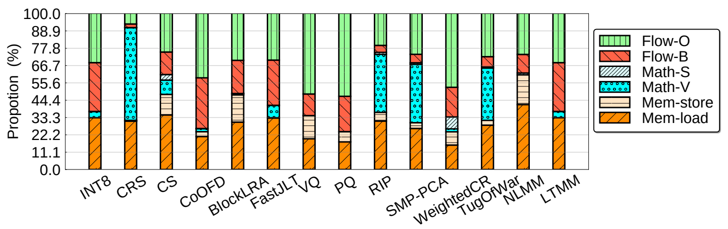


(c) Energy Consumption.

Figure 2. Overall performance comparison. Unable to measure the energy consumption of CoOFD and NLMM under *BUS* due to value overflow in MSR registers.

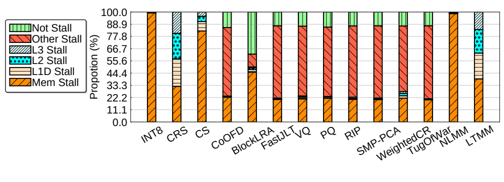


(a) Comparative analysis of instructions, LTMM is the '1' reference.

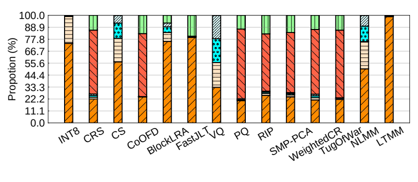


(b) Detailed breakdown of instruction propotions.

Figure 3. Analysis of ISA utilization, using *BUS* as example.



(a) Smallest dataset *ECO*


(b) Moderate dataset *UTM*

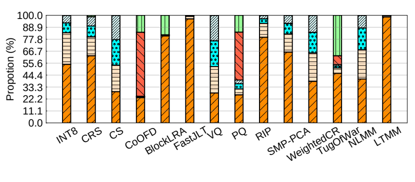

(c) Largest dataset *BUS*

Figure 4. Cycles breakdown.

structures (CoOFD, BlockLRA, and FastJLT). Notably, BlockLRA involves 11.5× more total instructions than LTMM. Specifically, the resource-intensive mathematical transformations and complex intermediate data structure maintainous lead to unexpected intensive *Math* and *Mem-load*. Conversely, all pruning-based *AMM* algorithms (INT8, CRS, and CS), all hybrid *AMM* (RIP, SMP-PCA, WeigthedCR, and TugOfWar), and two novel extraction-based *AMM* (PQ and VQ) converge into reducing *Math* and *Mem-load* and achieves more efficient computation.

Second, memory access, particularly the *Mem-load* instructions, takes a large proportion of all the evaluated *AMM* algorithms (Figure 3(b)), e.g., as high as 35.0% in CS and at least 17.8% in PQ. Such a large amount of *Mem-load* is caused by reading both input matrices and large intermediate data structures, as discussed in Section 2.2.

Third, most of our evaluated *AMM* algorithms, except INT8, CRS, and WeigthedCR, involve a higher amount of *Mem-store* than LTMM. Particularly, BlockLRA leads to at least 4 × more *Mem-store* than LTMM. This is primarily due to the additional need of most *AMM* to update intermediate data structures for pruning or feature extraction, which often exceed the capacity of registers.

Fourth, we notice that all extraction-based *AMM* (CoOFD, BlockLRA, FastJLT, VQ, and PQ), additionally exhibit a complex control flow, as their *Flow-B* and *Flow-O* instructions consistently account for more than 50% of the total instructions. For instance, PQ and VQ involve massive pointer updating and jumps (about 56% among total instructions), which are covered by *Flow-O*, to conduct their codebook manipulation. On the contrary, the pruning-based and hybrid counterparts typically exhibit notably lower proportions of *Flow-B* and *Flow-O* instructions, indicating simpler and more streamlined control flows. For instance, in the case of CRS, *Flow-B* and *Flow-O* collectively contribute to a mere 14.5% of the total instructions.

Lastly, SIMD utilization of mathematical calculations is still an important factor for further optimization. While AVX512 capabilities are inherently provided by LibTorch, and some algorithms like CRS and SMP-PCA can indeed conduct 90% ∼ 99% of their mathematical calculations under the vectorized and SIMD manner (i.e., *Math-V*), CS and WeigthedCR are suffering from poorer SIMD utilization (e.g., 72% *Math* are *Math-V* in CS). This limitation arises from their tendency to fragment the contiguous memory space of large tensors into separate vectors or lists, thereby compelling LibTorch to downgrade its vectorized functions to scalar operations, leading to poorer SIMD utilization.

5.2.2 Performance Bottlenecks.

We break down the processor cycle of conducting *AMM* to investigate the performance bottlenecks. Using the Silver 4310 processor, we can evaluate six distinct categories of cycles: 1) *Mem Stall*, encompasses cycles stalled in the main memory subsystem of the processor, primarily caused by waiting for TLB or DDR4 operation. 2) *L1D Stall*, 3) *L2 Stall*, and 4) *L3 Stall* refers to the stalled cycles resulting from cache miss in L1D, L2, and L3, respectively. 5) *Other Stall* includes cycles stalled in other parts of the processor, such as those due to ALU (integer operations) or FPU (floating-point operations) stalls. 6) *Not Stall* covers cycles when the processor is not experiencing any stalls. The cycle breakdown is influenced by the data volume, and to illustrate this, we use datasets *ECO*, *UTM*, and *BUS*, representing varying sizes from smallest to largest, as shown in Figure 4.

There are four major observations as follows. First, cache and memory stalls pose a significant challenge for all *AMM* algorithms and all *MM* baselines. E.g., they occupy up to 99.9% of the processor cycles for FastJLT algorithm in *BUS*. This is because *AMM* algorithms inherently involve intensive memory accesses (Figure 3(b)), which are typically much slower than in-register computation (Section 2.2).

Second, the cache and memory stalls are generally more pronounced with larger datasets. This is because memory access scales super-linearly with data scale, and memory stalls can also cascade into cache performance and increase cache stall hierarchically. For example, cache and memory stalls account for 58% proportion of the cycles when using CS on *ECO*, and this proportion increases significantly to 99.9% for larger datasets like *UTM* and *BUS*. Please also refer to our sensitivity study on scalability to data volume (Figure 7).

Third, for most *AMM* algorithms, except CS and VQ, *Mem Stall* is more dominant than all kinds of cache stalls, often surpassing them by a significant margin, such as up to 39.7× more than L1D Stalls in FastJLT algorithm on *BUS* dataset. This phenomenon can be attributed to the way these algorithms organize data structures, typically tensors, in a continuous manner of addressing. Libtorch's cache-aware optimizations, such as *cache blocking* and *spatial locality* [8], perform well in this context. In contrast, CS and VQ face challenges as they iteratively access a set of physically isolated and logically dependent tensors, reducing the benefits of LibTorch's optimizations.

Lastly, CoOFD and PQ suffer from *Other Stall*, even in the largest dataset *BUS* (59.3% and 45.3%, respectively). This is because CoOFD involves an intensive mathematical calculation and usually makes the FPU stalled, and PQ

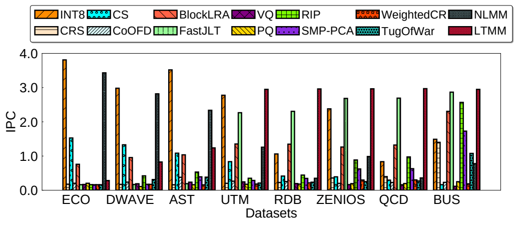


Figure 5. IPC comparison.

requires a significant proportion of flow control efforts to search through its codebooks and look-up tables (Figure 3(b)), which slows down the ALU and flow control logic.

5.2.3 Execution Efficiency.

We conduct the Instructions Per Cycle (IPC) comparison of *AMM*, as shown in Figure 5. IPC is a metric that quantifies an algorithm's capacity to utilize the capabilities of a modern processor fully [45]. A higher IPC value indicates more efficient execution and more effective instruction-level parallelism, i.e., it matches better to the out-of-order execution capability of the Silver 4310 processor. Overall, *AMM* algorithms involve a wide range of IPC from about 0.2 to 3.5, and the IPC of the same algorithm varies a lot to the specific datasets. More importantly, the IPC is affected a lot by 1) the matrix elements' width, 2) the number of independent data structures (e.g., potential tensor parallelism) in an algorithm, and 3) the data volume of the matrix.

For datasets such as *ECO*, *DWAVE*, *AST*, and *UTM*, which are all constrained to dimensions not exceeding 1700×1700, the INT8 algorithm stands out by achieving the highest IPC. This is primarily due to its predominant use of 8-bit integer manipulation for matrix elements. However, as we shift our focus to larger datasets like *BUS*, a different trend emerges. In this context, the significance of independent tensors is more pronounced, and the FastJLT algorithm takes the lead by achieving the highest IPC among all *AMM* algorithms. Specifically, FastJLT leverages up to 3 independent tensors during its feature extraction phase, surpassing the rest of the algorithms due to such a tensor parallelism. Regarding PQ and VQ, which rely on codebooks of tensors, their IPC performance is consistently limited due to the necessity of sequentially accessing the codebook, i.e., generally not surpassing 0.55. The consistently low IPC is also found in CoOFD, as it also requires a sequential and iterative update on a complex data structure during its frequent directions computation [42].

It is important to note that IPC itself can not reflect the resulting latency in Figure 2(a), and the number of instructions (Figure 3(a)) should also be considered. For instance, even though FastJLT has the highest IPC on *BUS*, its intensive instructions can still lead to a higher latency compared with pruning-based algorithms INT8, CRS, and CS.

5.3 Sensitivity Study

In this section, we investigate the effectiveness of *AMM* methods under various settings of data properties and algorithm configurations.

5.3.1 Scalability to Data Volume. We utilize the synthetic dataset to investigate the data scalability of *AMM* algorithms. Specifically, the two multiplicable matrices A, B are generated by *torch::rand* function, and elements of A, B are randomly valued within $0 \sim 1$. We keep the number of columns in A, that of rows and columns in B to 2500, and vary the number of rows in A from $100 \sim 50000$. The resulting processing latency l, *AMM* error ϵ are demonstrated in Figures 6(a) and 6(c), respectively, and we further demonstrate the memory stall cycles in Figure 6(b).

Four significant findings are made. First, all of the evaluated *AMM* algorithms and *MM* baselines, involve an increasing l as the data scale grows. As expected, such a l growth aligns closely with the rise in memory stall cycles, as depicted in Figure 6(b). This observation reaffirms that memory stalls represent a significant bottleneck (Section 5.2). For instance, there is a more than 40× growth of l in LTMM and INT8 when the rows in matrix A increase from 1000 to 2500. This is because they reach the peak of memory bandwidth utilization within this data scaling range ($1000 \sim 2500$ rows), and the penalty for memory access becomes significantly more pronounced once the bandwidth capacity is fully utilized.

Second, there is a static overhead of *AMM*, and such overhead is even more time-consuming than the streamlined operation of LTMM when the data scale is small (e.g., around 100 rows) for most *AMM* except INT8. For instance, CRS and CS require setting up the element indices of pruning, which makes it slower than LTMM when the data scale is within 1000 rows. Additionally, the static overhead is generally larger for hybrid *AMM* like SMP-PCA or RIP.

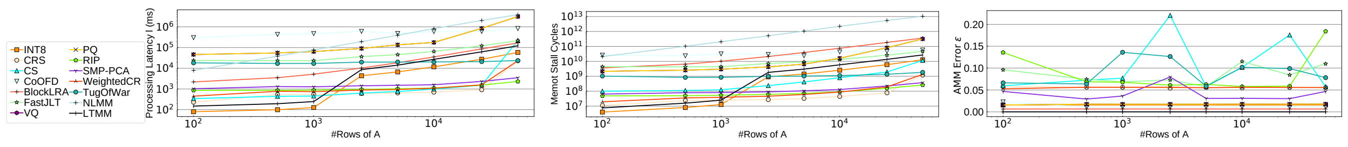
Third, while pruning-based *AMM* (INT8, CRS, and CS) exhibit an appealing reduction of l under a moderate data scale (i.e., $1000 \sim 10000$ rows here), they are found to be less scalable compared to two hybrid *AMM* approaches, RIP and SMP-PCA, primarily due to the constraints on memory bandwidth. In particular, INT8, CRS, and CS experience a sharp increase in memory stall cycles as the data scale exceeds a certain threshold, resulting in a significant spike in l. This threshold is comparable to that of LTMM for INT8 (approximately $1000 \sim 2500$ rows) and substantially higher for CRS and CS (25000 rows or more). In contrast, RIP and SMP-PCA demonstrate a smooth increase in memory stall cycles and l without surge growth, signifying that they have not fully saturated the available memory bandwidth.

Fourth, when it comes to the *AMM* error ϵ, most algorithms keep ϵ no more than 0.1. Nevertheless, the ϵ of CS, RIP, FastJLT, and TugOfWar display a notable degree of variability in their ϵ values. Notably, CS can exhibit ϵ that exceeds 0.2 during our data scaling evaluation. This variability in ϵ can be attributed to the fact that the theoretical error guarantees of CS, RIP, FastJLT, and TugOfWar algorithms, as documented in [12, 29, 36, 47], are weaker than bounding the Frobenius Norm (as discussed in Section 2.1). Consequently, this theoretical flexibility contributes to increased uncertainty in ϵ as data scales up.

5.3.2 Impacts of Tuning Knob ω. We let both input matrices A, B sized 2500×2500 and generated by *torch::rand*. We vary ω from $0.04\% \sim 50\%$, and demonstrate the resulting processing latency l, memory stall cycles, and AMM error ϵ are demonstrated in Figures 7(a), 7(b), and 7(c), respectively.

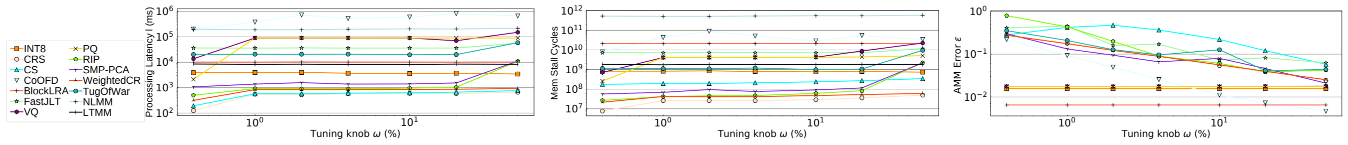
Our findings can be summarized as follows. First, for most tunable AMM algorithms, except for CS, increasing ω usually leads to a reduction in ϵ at the expense of higher l, and the specific trade-off varies significantly among different algorithms. For example, CRS allows to reduce about 60% l by increasing ϵ from 0.02 to 0.28 during tuning the ω. Compared with CRS, BlockLRA exhibits a narrow trade-off space between ϵ and l, consistently maintaining an ϵ of less than 0.01. On the contrary, CS does not necessarily lead to a lower ϵ with larger ω due to its relatively weak error bound, as also demonstrated in Figure 6(c). Second, as depicted in Figure 7(b), a higher ω value results in increased memory stall cycles, which is a primary contributing factor to larger l. As discussed in Sections 2.2 and 4.1, a larger ω preserves or extracts more information, necessitating a larger intermediate data structure and more intensive memory access to maintain it.

5.3.3 Impacts of Skewed Numeric Values. We let both A, B sized 2500×2500 with ω set to 10%. The matrix B is still generated by *torch::rand*. In contrast, matrix A is intentionally structured to exhibit a skewed pattern of numeric values. To achieve this, we apply the *torch::pow* and *torch::multinomial* functions to shape the elements of matrix A to follow a Zipf distribution. Subsequently, we scaled the values within matrix A to a range of 0 to 1 to align with matrix B. We explored varying the distribution factor of Zipf from 1 to 2, where a larger factor indicates a higher degree of value skewness, making some randomly selected values more likely to appear than others. Given that processing latency exhibited only minor fluctuations for each algorithm (approximately ±5%), and the latency comparison of different algorithms is the same as Figure 6(a)'s case when both A, B are sized 2500×2500, our primary focus was on the resulting *AMM* error (ϵ), as depicted in Figure 8. In summary, all of the evaluated *AMM* algorithms displayed sensitivity to value skewness. Notably, when the Zipf factor exceeded 1.6, only INT8, BlockLRA, CoOFD, VQ and PQ managed to maintain ϵ within a 0.5 margin, while the remaining *AMM* algorithms exceeded 0.8. This sensitivity stems from the fact that the extraction of singular values, as utilized in BlockLRA and CoOFD, inherently



(a) Processing latency l. (b) Memory stall cycles. (c) *AMM* error ϵ.

Figure 6. Scalability to Data Volume.



(a) Processing latency l. (b) Memory stall cycles. (c) AMM error ϵ.

Figure 7. Impacts of tuning knob ω. Note that ω does not affect LTMM, NLMM, or INT8.

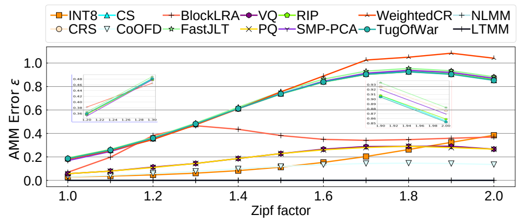


Figure 8. Impacts of skewed numeric values. A larger Zipf factor refers to a higher level of skewness. We have zoomed in on two areas where lines are intensive in embedded images.

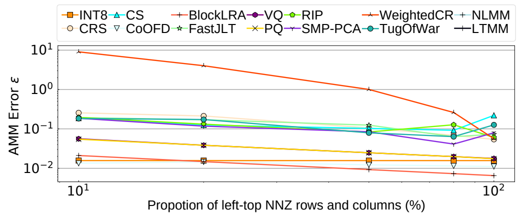


Figure 9. Impacts of biased NNZ region. A smaller proportion refers to a more biased NNZ region. The ϵ of LTMM and NLMM baselines are zero and can not be displayed in the log scale.

exhibits robustness to value skewness. The mathematical procedures involved in singular value extraction, such as centering and regularization [58], are effective treatments for handling value skewness. However, it's worth noting that this robustness is somewhat compromised by the block partitioning of matrices, as BlockLRA yielded up to a 5.5x higher ϵ compared to CoOFD. Extracting KNN centroids (i.e., VQ and PQ) also inherently involves centering of data [18] and heals value skewness to some extent, but it is less robust compared with singular values extraction due to the mathematical lack of a regularization process. On the other hand, the straightforward bit-wise pruning technique employed in INT8 (Section 3.1) also demonstrates resilience to value skewness. However, the scalar quantization error of values is amplified by value skewness as the Zipf factor increases, leading to ϵ exceeding 0.2 when the Zipf factor surpasses 1.5.

5.3.4 Impacts of Biased Non-zero (NNZ) Region.

We follow all of the settings used in Figure 8, with the exception that matrix A was generated to exhibit a biased Non-Zero (NNZ) region. To achieve this, we initially created a 2500×2500 zero matrix using the *torch::zero* function. Subsequently, we selectively replaced a portion of the top-left sub-matrix with random values generated by *torch::rand*. For example, when the proportion is set to 10%, the NNZ region consists of the first 250 rows and the first 250 columns of matrix A. A smaller proportion indicates a more highly biased NNZ region, as the non-zero values become concentrated within a smaller area. We varied the proportion of NNZ rows and columns in the top-left region from 10% $\sim$ 100%, and displayed the resulting ϵ in Figure 9. It's worth noting that varying the biased NNZ region had a negligible impact on processing latency, with fluctuations of around ±6%, so we have omitted those specific details. In general, all of the evaluated *AMM* algorithms, except for INT8, exhibited larger ϵ when the NNZ region was highly biased. Notably, the ϵ of WeigthedCR even exceeds 10 when only 10% of NNZ rows and columns were concentrated at the top-left. WeigthedCR, being a hybrid *AMM* algorithm, incorporates sampling for information pruning and median feature extraction (as discussed in Section 3.3). It is evident that neither sampling nor median feature extraction is resilient to biased NNZ regions, and their combination can further amplify inaccuracies. Specifically, the pruning-based *AMM* algorithm CRS, which exclusively employs sampling, consistently achieved ϵ within 0.3. In contrast to the other algorithms, INT8 consistently maintained a low ϵ, staying within 0.0016. This resilience can be attributed to the bit-wise bounded scalar quantization error of INT8 [35], as supported by information theory. Notably, this error was not exacerbated by other factors, such as value skewness, as shown in Figure 8. Additionally, BlockLRA and CoOFD also demonstrated resilience to biased NNZ regions. Although their ϵ increased with more biased NNZ locations, it consistently remained below 0.002. This resilience can be attributed to their singular value extraction process. Moreover, the downgrade effects of block partitioning (i.e., BlockLRA vs. CoOFD) on resilience was less significant than in Figure 8. BlockLRA resulted in up to 0.6× more ϵ than CoOFD, and it could even achieve less ϵ than CoOFD when more than 50% of NNZ rows and columns were located in the top-left region.

5.4 Key Findings

Our main findings of this section are summarized as follows:

Recommendation of AMM Algorithms. For scenarios with moderate data scales, involving thousands of rows and columns, we endorse two pruning-based *AMM* algorithms, INT8, and CRS, for accelerating *MM* processes (refer to Figure 2). However, for objectives prioritizing lower error rates and enhanced scalability in the context of larger datasets, hybrid *AMM* algorithms SMP-PCA emerges as the preferred choice. Another hybrid *AMM* RIP involves similar data scalability with SMP-PCA and it's slightly faster than SMP-PCA, but it involves a relatively loose error bound. Conversely, extraction-based *AMM* algorithms generally fall short of the practical benefits offered by pruning-based or hybrid alternatives.

Common Bottlenecks. A common bottleneck across all assessed *AMM* algorithms and *MM* baselines is the intensive memory load operations that lead to significant memory stalls (detailed in Figure 3). These stalls contribute to performance penalties that outstrip the projections of O complexity analysis, particularly as data scales increase (Figure 6(b)) and with higher ω values (Figure 7(b)). Thus, future *AMM* optimizations may need to prioritize heuristic approaches to mitigate memory pressure over simply addressing theoretical computational complexity. In addition to memory pressure, extraction-based *AMM* algorithms also suffer from intricate flow control issues, which impede practical performance — another nuance not captured by O complexity metrics.

Implications for Complex Workloads. Despite most state-of-the-art studies providing a well-defined error bound [11, 12, 18, 21, 25, 28, 29, 36, 40, 41, 44, 47, 56, 58], it is imperative to acknowledge that workloads with complex characteristics (e.g., *ECO* and *UTM*), such as a highly skewed distribution of numeric value and a high level of biased non-zero region, can exacerbate the *AMM* error rates for many algorithms, as illustrated in Figure 2(b).

Table 4. Dataset and Latency Proportion of *AMM*-Replaceable *MM* in Downstream Tasks

Downstream Task	Dataset	Matrix Size	Proportion
PCA	*SIFT10K*	128×10000	13.2%
Machine Learning Training	*MNIST*	$392 \times 60000 \cdot 60000 \times 392$	21.4%
Machine Learning Inference	*CIFAR100*	$10000 \times 512 \cdot 512 \times 100$	86.4%
Unitary Transformation	*QCD*	3072×3072	100%

6 What Is the Application Impact of AMM?

We now explore the impact of *AMM* in various downstream applications detailed in Section 6.1. We provide a detailed report encompassing the processing latency (l), error (ϵ), and the application error (e) in Section 6.2.

6.1 Application Overview

We examine a suite of downstream tasks where the integration of *AMM* is particularly appealing. These tasks include *Principal Component Analysis* (PCA), *Canonical Correlation Analysis* (CCA), *Machine Learning Training* and *Inference* phases, and *Unitary Transformation*. In Table 4, we present an analysis of the datasets utilized, highlighting the proportion of latency attributed to *AMM*-replaceable *MM* within different downstream tasks. All tasks are integrated into the LibTorch ecosystem and utilize its LTMM functionality for *MM*. Machine learning training and inference are implemented through the PyTorch frontend, with bindings to our static compilation ensuring that the JIT does not impact the execution of *AMM* or *MM*. Further elaboration on these downstream tasks is presented in the subsequent discourse.

Principal Component Analysis (PCA). PCA is a popular statistical function for dimensionality reduction [46, 54]. It computes the rank-r approximation of a matrix A as $\hat{A_r}$, and the application error e is defined as $e = ||A - \hat{A_r}||/||A||$. PCA is required to compute the covariance matrix, and the involved *MM* can be replaced by *AMM*. We conducted the PCA task on the *SIFT10K* dataset following the methodology outlined in [56]. Because the number of rows is exceptionally small (128) in comparison to the substantial column count (10000), the tuning parameter ω is configured at 10% for PQ and VQ, as their adjustment is row-relevant (Section 4.1). For the remaining *AMM*, we set ω to 1%.

Canonical Correlation Analysis (CCA). CCA quantifies the linear relationships between two matrices A, B, and it outputs three matrices denoted as U, S, V to represent the corresponding linear correlations of A, B [22, 30, 50]. Similar to PCA's case, CCA requires the covariance calculation where *MM* is involved and can be replaced by *AMM*. The e of CCA is defined as $e = max\{abs(diag(S - \hat{S}))\}$, where S is the original result computed with *MM*, and $\hat{S}$ refers to the outcome with *AMM* utilization. We conducted the CCA task using the *Mediamill* dataset, following the methodology outlined in [13]. ω

setting is the same as PCA's case, as *Mediamill* also involves extremely small row volume compared with its column volume.

Machine Learning Training. We implement the methodology outlined in prior work [11], incorporating *AMM* techniques into three fully connected layers of an MLP model during the machine learning training. We use different configurations of hidden layer dimensions in our evaluation, i.e., 500-D and 2000-D, which involve relatively smaller and larger weight matrices, respectively. There are thousands of Stochastic Gradient Descent (SGD) iterations in training, and each SGD iteration utilizes *AMM* to forward the training loss before randomly updating the weight matrices. We report the average ϵ of first 10 SGD iterations during training, as they are most meaningful for the training task. We exclude VQ and PQ, because they necessitate costly rebuilding of the entire codebook from scratch for each new version of the weight matrix, and it requires 10^7 ms l for building codebook 1000 times even in 500-D hidden layer case.

Machine Learning Inference. For inference, we apply *AMM* to the final dense layer in the pre-trained, VGG-like model discussed in [18] and set ω to 2.56%. The machine learning model also works as classifiers in [18], and the meaning of e is the same as that in the case of machine learning training. we employ *CIFAR100* datasets as the illustration example.

Unitary Transformation. Unitary transformation is one of the key operations in the scientific computing of quantum physics [23, 27], and it transforms the quantum state matrix q into Q by two consecutive *MM* with a unitary gate. All of these *MM* are possible to be replaced by *AMM*, which transforms q into $\hat{Q}$ instead of Q. We report the average ϵ of these two multiplications. Q^2 is proportional to the probability of collapsing into specific classic states when measured, and e is hence formulated as $e = ||Q^2 - \hat{Q}^2||/||Q^2||$. For illustration purposes, we instantiate q as one of the *QCD* matrices, let the unitary gate exhibit Gaussian distributions [16], and ignore the normalization constants.

6.2 Evaluation Results

The evaluation results of deploying *AMM* in the four downstream applications are shown in Figure 10.

PCA Task. We observe two pruning-based *AMM* approaches, namely INT8 and CRS, exhibit a significant reduction in l compared to LTMM, i.e., by about 1/3 and 1/2, respectively. As anticipated, the use of INT8 and CRS incurs a tolerable increase in both ϵ and e. Furthermore, it's important to note three extraction-based *AMM* algorithms, namely BlockLRA, VQ, and PQ, demonstrate significantly higher *AMM* error (ϵ) than the others, and their resulting application error e is also high, e.g., reaching up to 0.99 under PQ. This is due to the mismatch of their extracted features and principle information of PCA.

CCA Task. The l comparison of *AMM* algorithms under CCA closely mirrors that of PCA, except that only the pruning-based *AMM* INT8 outperforms LTMM. ϵ is no more than 0.14 for all of the evaluated *AMM* methods, and hybrid *AMM* SMP-PCA achieved the smallest ϵ of about 0.001. However, e is usually larger than 0.4 for most *AMM* except INT8, and the most significant discrepancy was observed in the case of CoOFD with e reaching 0.84 but ϵ being only 0.01. This is because CCA involves calculating the inverse after utilizing *MM* or *AMM* [50], which is particularly sensitive to information loss of matrix elements.

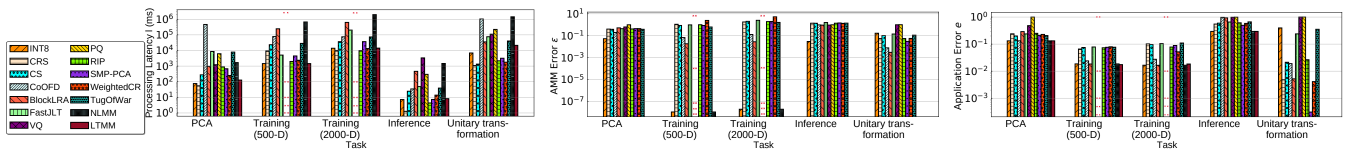
Machine Learning Training. We have two-fold key observations of applying *AMM* in training. On the one hand, more *AMM* approaches (e.g., CRS and RIP) outperform the stronger *MM* baseline LTMM in l under 2000-D hidden layer compared with the 500-D hidden layer setting (i.e., only INT8). This is because larger weight matrices involved in the 2000-D case make the static overhead of *AMM* less dominant. Consequently, opting for pruning-based or hybrid *AMM* approaches becomes more meaningful as they alleviate memory stress and, in turn, lead to a reduction in l. On the other hand, a larger weight matrix is more sensitive to information loss of *AMM*, leading to higher e when using *AMM*. For instance, CRS attains a higher e by 0.01 compared to LTMM at 500-D hidden layer, but this difference in e is further enlarged to 0.08 at 2000-D hidden layer.

Machine Learning Inference. *AMM* is more attractive for adoption in inference compared with training. Particularly, four *AMM* methods (INT8, CRS, RIP, and SMP-PCA) outperform LTMM in l, with CRS achieving the most substantial l reduction, approximately by 80%. Additionally, there is a disparity between ϵ and e. Specifically, several extraction-based *AMM* methods, including CoOFD, BlockLRA, VQ, and PQ, exhibit elevated e (e.g., 0.9 e of using CoOFD). This is because they fail to preserve the semantic information hidden behind the numeric patterns.

Unitary Transformation. All of the pruning-based and most hybrid *AMM* (except TugOfWar) outperform *MM* in l in the unitary transformation. Notably, CRS reduces 94% l compared with *MM*. Besides, the consecutive multiplications involved in the unitary transformation lead to an amplification of error and a high e for some *AMM* like INT8, VQ, and PQ. On the contrary, SMP-PCA is resilient to such error amplification and achieves the lowest e among all *AMM* (i.e., 0.0003) due to its extra scaling (Section 3.3). We acknowledge that such an error amplification may also occur in more complicated scientific computing applications like Finite Element Analysis [17] and leave it as a future work.

6.3 Key Findings

We have confirmed the practical usefulness of *AMM*, especially in machine learning inference and unitary transformation. This confirmation basically aligns with Section 5.4's recommendation. However, there exists a



(a) Processing latency l of *AMM*

(b) *AMM* Error ϵ.

(c) Application error e

Figure 10. Evaluation on applying *AMM* to downstream tasks. VQ and PQ are excluded in training due to the $\geq 10^7 ms$ overhead of codebook rebuilding (Section 6.1).

discrepancy between the *AMM* error (ϵ) and the application error (e), as well as the error amplification, in some downstream tasks. Therefore, incorporating *AMM* into new applications typically requires a thorough and discerning selection process. It is also beneficial if the error bounds in the future *AMM* can be strengthened and more tailored to the specific needs of downstream tasks.

7 Related Work

The spectrum of *AMM* algorithms spans scalar quantization to complex transforms, as documented in a range of studies from Khudia et al. [35] to Mroueh et al. [42]. Our benchmark encapsulates these methods under pruning, extraction, and hybrid categories and extends the analysis to machine learning-enhanced feature extraction techniques [18, 40]. These contributions, pivotal in theory, now meet an assessment of their practical efficiency on contemporary processors through our work.

Research endeavors such as those by Drineas et al. [26] and Wu et al. [56] provide valuable insights into *AMM*'s utility in statistical tasks, but often lack comprehensive benchmarks that account for both latency and accuracy. Our study distinguishes itself by thoroughly evaluating *AMM* across a spectrum of use cases, debunking common misconceptions about computation and memory access patterns with a meticulous hardware profiling that prior studies have overlooked [11, 18, 44].

Profiling studies like those by Sridharan et al. [49] and Panda et al. [45] have illuminated the impact of hardware-specific operational patterns on algorithmic performance. Our research contributes to this dialogue by focusing on *AMM*, i.e., to understand the nuanced interplay of memory access, mathematical operations, and flow control in conducting *AMM* on modern hardware.

8 Conclusion

Our study of *AMM* algorithms on a modern processor reveals key insights into their practical performance. Pruning-based (INT8, CRS) and hybrid *AMM* (SMP-PCA, RIP) outshine extraction-based methods, particularly in real-world scenarios. We uncover that theoretical analyses often fail to account for critical software-hardware interactions, such as *memory access overhead* and *flow control complexities*, which significantly impact algorithm efficiency. Furthermore, we note discrepancies between theoretical error bounds and actual workload behaviors, suggesting a need for algorithms that better anticipate complex data characteristics. Moving beyond the limitations of theoretical analysis, future works may pay more attention to the hardware consciousness in optimizing *AMM*, such as reducing memory pressure in algorithm design, agglomerating input data for less static overhead, and better exploring novel high-memory bandwidth (HBM) architectures.

References

[1] 2007. *Matrix Market, http://https://math.nist.gov/MatrixMarket//*. Last Accessed: 2023-08-09.

[2] 2021. *Arm Architectures, https://developer.arm.com/architectures*. Last Accessed: 2021-05-12.

[3] 2022. *TMS320C6000 Technical Brief, https://www.ti.com/lit/ug/spru197d/spru197d.pdf*. Last Accessed: 2022-05-10.

[4] 2023. *Cortex-M7, https://developer.arm.com/Processors/Cortex-M7*. Last Accessed: 2023-05-10.

[5] 2023. *ESP32 Series, https://www.espressif.com/en/products/socs*. Last Accessed: 2023-05-10.

[6] 2023. *MIPS Architectures, https://www.mips.com/products/architectures/*. Last Accessed: 2023-10-12.

[7] 2023. *NVIDIA Technologies, https://www.nvidia.com/en-us/technologies/*. Last Accessed: 2023-05-10.

[8] 2023. *Pytorch homepage, https://pytorch.org/*.

[9] 2023. *Reading and Writing Model Specific Registers (MSRs) in Linux, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/reading-writing-msrs-in-linux.html*. Last Accessed: 2023-05-12.

[10] 2023. *RISC-V, https://riscv.org/*. Last Accessed: 2023-10-12.

[11] Menachem Adelman, Kfir Levy, Ido Hakimi, and Mark Silberstein. 2021. Faster neural network training with approximate tensor operations. *Advances in Neural Information Processing Systems* 34 (2021), 27877–27889.

[12] Nir Ailon and Bernard Chazelle. 2009. The fast Johnson–Lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on computing* 39, 1 (2009), 302–322.

[13] Haim Avron, Christos Boutsidis, Sivan Toledo, and Anastasios Zouzias. 2013. Efficient dimensionality reduction for canonical correlation analysis. In *International conference on machine learning*. PMLR, 347–355.

[14] P Bachmann. 1894. Analytische Zahlentheorie (Bd. 2) Teubner. *Leipzig, Germany* (1894).

[15] Austin R Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices* 50, 8 (2015), 42–53.

[16] Reinhold A Bertlmann, G Launer, and E De Rafael. 1985. Gaussian sum rules in quantum chromodynamics and local duality. *Nuclear Physics B* 250, 1-4 (1985), 61–108.

[17] SS Bhavikatti. 2005. *Finite element analysis.* New Age International.

[18] Davis Blalock and John Guttag. 2021. Multiplying matrices without multiplying. In *International Conference on Machine Learning.* PMLR, 992–1004.

[19] Christos Boutsidis and Alex Gittens. 2013. Improved matrix algorithms via the subsampled randomized Hadamard transform. *SIAM J. Matrix Anal. Appl.* 34, 3 (2013), 1301–1340.

[20] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. 2000. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications* 14, 3 (2000), 189–204.

[21] Neophytos Charalambides, Mert Pilanci, and Alfred O Hero. 2021. Approximate weighted CR coded matrix multiplication. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 5095–5099.

[22] Kamalika Chaudhuri, Sham M Kakade, Karen Livescu, and Karthik Sridharan. 2009. Multi-view clustering via canonical correlation analysis. In *Proceedings of the 26th annual international conference on machine learning.* 129–136.

[23] Xie Chen, Zheng-Cheng Gu, and Xiao-Gang Wen. 2010. Local unitary transformation, long-range quantum entanglement, wave function renormalization, and topological order. *Physical review b* 82, 15 (2010), 155138.

[24] Helin Cheng, Wenxuan Li, Yuechen Lu, and Weifeng Liu. 2023. HASpGEMM: Heterogeneity-Aware Sparse General Matrix-Matrix Multiplication on Modern Asymmetric Multicore Processors. In *Proceedings of the 52nd International Conference on Parallel Processing.* 807–817.

[25] Petros Drineas, Ravi Kannan, and Michael W Mahoney. 2006. Fast Monte Carlo algorithms for matrices I: Approximating matrix multiplication. *SIAM J. Comput.* 36, 1 (2006), 132–157.

[26] Petros Drineas, Ravi Kannan, and Michael W Mahoney. 2006. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM Journal on computing* 36, 1 (2006), 158–183.

[27] Lei Feng, Or Katz, Casey Haack, Mohammad Maghrebi, Alexey V Gorshkov, Zhexuan Gong, Marko Cetina, and Christopher Monroe. 2023. Continuous symmetry breaking in a trapped-ion spin chain. *Nature* (2023), 1–5.

[28] Deena P Francis and Kumudha Raimond. 2022. A practical streaming approximate matrix multiplication algorithm. *Journal of King Saud University-Computer and Information Sciences* 34, 1 (2022), 1455–1465.

[29] Vipul Gupta, Shusen Wang, Thomas Courtade, and Kannan Ramchandran. 2018. Oversketch: Approximate matrix multiplication for the cloud. In *2018 IEEE International Conference on Big Data (Big Data).* IEEE, 298–304.

[30] Hotelling Harold. 1936. Relations between two sets of variables. *Biometrika* 28, 3 (1936), 321–377.

[31] Jianyu Huang, Devin A Matthews, and Robert A van de Geijn. 2018. Strassen's algorithm for tensor contraction. *SIAM Journal on Scientific Computing* 40, 3 (2018), C305–C326.

[32] Jianyu Huang, Tyler M Smith, Greg M Henry, and Robert A Van De Geijn. 2016. Strassen's algorithm reloaded. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 690–701.

[33] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[34] Intel. 2019. *Intel 64 and IA-32 Architectures optimization Reference Manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf*. Last Accessed: 2023-10-12.

[35] Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. 2021. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615* (2021).

[36] Felix Krahmer and Rachel Ward. 2011. New and improved Johnson–Lindenstrauss embeddings via the restricted isometry property. *SIAM Journal on Mathematical Analysis* 43, 3 (2011), 1269–1281.

[37] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–22.

[38] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 747–761. https://doi.org/10.1145/3575693.3575706

[39] Edo Liberty. 2013. Simple and deterministic matrix sketching. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.* 581–588.

[40] Yusuke Matsui and Shin'ichi Satah. 2018. Revisiting Column-Wise Vector Quantization for Memory-Efficient Matrix Multiplication. In *2018 25th IEEE International Conference on Image Processing (ICIP).* IEEE, 1158–1162.

[41] Youssef Mroueh, Etienne Marcheret, and Vaibahava Goel. 2017. Co-Occurring Directions Sketching for Approximate Matrix Multiply. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Jerry Zhu (Eds.). PMLR, 567–575. https://proceedings.mlr.press/v54/mroueh17a.html

[42] Youssef Mroueh, Etienne Marcheret, and Vaibahava Goel. 2017. Co-occurring directions sketching for approximate matrix multiply. In *Artificial Intelligence and Statistics.* PMLR, 567–575.

[43] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023).* Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[44] Kazuki Osawa, Akira Sekiya, Hiroki Naganuma, and Rio Yokota. 2017. Accelerating matrix multiplication in deep learning by using low-rank approximation. In *2017 International Conference on High Performance Computing & Simulation (HPCS).* IEEE, 186–192.

[45] Reena Panda, Christopher Erb, Michael Lebeane, Jee Ho Ryoo, and Lizy Kurian John. 2015. Performance characterization of modern databases on out-of-order cpus. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* IEEE, 114–121.

[46] Rafael do Espírito Santo. 2012. Principal Component Analysis applied to digital image compression. *Einstein (São Paulo)* 10 (2012), 135–139.

[47] Tamas Sarlos. 2006. Improved approximation algorithms for large matrices via random projections. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06).* IEEE, 143–152.

[48] Kyungrak Son, Aditya Ramamoorthy, and Wan Choi. 2021. Distributed matrix multiplication using group algebra for on-device edge computing. *IEEE Signal Processing Letters* 28 (2021), 2097–2101.

[49] Shriram Sridharan and Jignesh M. Patel. 2014. Profiling R on a Contemporary Processor. *Proc. VLDB Endow.* (2014).

[50] Ya Su, Yun Fu, Xinbo Gao, and Qi Tian. 2011. Discriminant learning through multiple principal angles for visual recognition. *IEEE transactions on image processing* 21, 3 (2011), 1381–1390.

[51] Elahe Vedadi and Hulya Seferoglu. 2021. Adaptive coding for matrix multiplication at edge networks. In *2021 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 1064–1069.

[52] John Von Neumann. 1993. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing* 15, 4 (1993), 27–75.

[53] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 215–228.

[54] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[55] Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2013. On-line soft error correction in matrix–matrix multiplication. *Journal of Computational Science* 4, 6 (2013), 465–472.

[56] Shanshan Wu, Srinadh Bhojanapalli, Sujay Sanghavi, and Alexandros G Dimakis. 2016. Single pass PCA of matrix products. *Advances in Neural Information Processing Systems* 29 (2016).

[57] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 94–105.

[58] Qiaomin Ye, Luo Luo, and Zhihua Zhang. 2016. Frequent direction algorithms for approximate matrix multiplication with applications in CCA. *computational complexity* 1, m3 (2016), 2.

[59] Brandon Yue et al. 2023. *Optimizing Out-Of-Memory Sparse-Dense Matrix Multiplication*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[60] Xianzhi Zeng and et al. 2023. Parallelizing Stream Compression for IoT Applications on Asymmetric Multicores. In *Proceedings of the 39th IEEE International Conference on Data Engineering (ICDE2023)*. IEEE.

[61] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702