Declaration

I hereby declare that the thesis entitled *Hardware-aware algorithm optimization on AI processors: Huawei Ascend as an example* represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Yifeng Tang
December 2024

Abstract of thesis entitled

Hardware-aware algorithm optimization of AI processors: Huawei Ascend as an example

submitted by

Yifeng Tang

for the degree of Doctor of Philosophy
at the University of Hong Kong
in December 2024

The field of Artificial Intelligence (AI) has become one of the most prominent areas of research and industry application due to its vast impact and versatility. The development of complex AI models, which often involve extensive parameters, necessitates substantial computational power. This is where specialized hardware, known as AI processors, becomes essential. This thesis focuses on the architecture and performance of Huawei Ascend processors, a representative AI processor, and introduces novel optimization strategies to enhance algorithmic efficiency.

AI processors primarily rely on matrix multiplier-accumulators (MACs), which execute matrix multiplications with remarkable computational capability. Matrix multiplications serve as the foundation for various AI algorithms, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers. However, complete AI applications require a broader range of operations beyond matrix multiplication. Therefore, AI processors include additional hardware units that facilitate essential operations and data transfers to enable comprehensive AI functionalities.

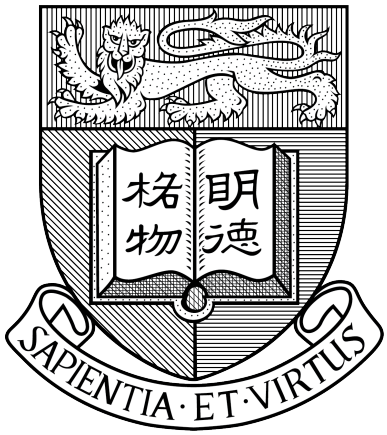
Huawei Ascend processors incorporate four key types of hardware units: Matrix MACs for matrix multiplications, IO units for intra- and inter-core

data transfers, vector units for performing vectorized calculations, and scalar units for address calculations and branch conditions. Each unit has specific strengths and constraints that influence the overall performance and optimization of AI algorithms on Ascend processors.

To gain in-depth insights into the structure and functionality of Ascend processors, we developed specialized micro-benchmarks to examine their hardware characteristics, including IO contention, bandwidth sharing, and runtime behavior. The empirical data collected enabled us to construct a performance model, Verrocchio, which accurately predicts the execution time of real-world Ascend kernels. Verrocchio's predictions achieve an average error rate of 2.62% for single-core and 2.30% for double-core executions.

It is noteworthy that non-MAC units, compared to Matrix MACs, often exhibit limited performance, potentially bottlenecking overall application efficiency. In response, we introduce two primary optimization strategies alongside Verrocchio to enhance algorithmic implementation on Ascend processors: (1) replacing suboptimal scalar or vectorized operations with more efficient alternatives, and (2) mapping certain operations to matrix multiplications where feasible. For the first optimization, we exemplify with the k-nearest neighbors (k-NN) algorithm, proposing SelB-k-NN (Selection-Bitonic-k-NN), which mitigates the need for suboptimal operations in large-scale datasets. SelB-k-NN delivers a 2.01× speedup over bitonic k-selection, 23.93× over the heap method, and 78.52× over the CPU-based approach. For the second optimization, we propose Cube-fx, a mapping algorithm of Taylor expansion for multiple functions onto Matrix MACs. Performance evaluations show that Cube-fx surpasses the standard Taylor expansion implementation by 2.73×, CORDIC by 6.06×, and Horner's Method by 1.64×. While this second strategy achieves significant performance gains by fully utilizing Matrix MACs, it is limited to operations that can be reformulated as matrix multiplications. Therefore, the first strategy remains essential for optimizing a wider range of computations on AI processors. Together, these strategies offer a comprehensive approach to maximize efficiency on this architecture. (466 words)

Hardware-aware algorithm optimization on AI processors: Huawei Ascend as an example



Yifeng Tang

Department of Computer Science
The University of Hong Kong

Thesis submitted to the University of Hong Kong,
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy.

December 2024

Acknowledgement

First and foremost, I extend my deepest gratitude to my primary supervisor, Prof. Cho-li Wang, for his invaluable guidance and encouragement throughout my research journey. Prof. Wang has a clear vision of cutting-edge technologies and offers insightful and practical guidance for feasible exploration paths. His unwavering commitment and high standards for my work have been a constant source of inspiration. I am also sincerely grateful to my co-supervisor, Prof. Heming Cui, whose support with University paperwork has allowed me the freedom to focus intensively on my research.

I would like to thank my labmates: Zhuoran Ji, Jiuzhou Zhao, Xueyu Wu, Zhaorui Zhang, Xin Yao, Dong Huang, Frank Qing, Qi Hu, and Yu Tian. The time we spent together at the University will always be a cherished memory.

I am deeply grateful to my parents, Zhongchun Tang and Ping Yan, my wife, Yihan Mei, my relatives, and friends for their unwavering support during the most challenging moments of my Ph.D. journey. Finally, special thanks go to GPT-4, which assisted in refining much of the writing in the final version of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Neural Networks (DNNs), as one of the most prominent models in Artificial Intelligence (AI), have captured global attention. DNNs have transformed numerous fields, including medical biology [1], renewable energy [2], and autonomous vehicles [3], by enabling complex and intelligent tasks that were once impractical or unattainable. Achieving high performance in DNNs relies heavily on massive matrix multiplications, demanding substantial computational power. General-purpose processors are not optimized for such tasks, leading to high energy consumption, prolonged execution times, and limited scalability. In response, hardware manufacturers have developed domain-specific hardware, known as AI processors [4–10], which include specialized units to accelerate AI computations. Most of these units are designed to optimize matrix multiplications, known as matrix multiplier-accumulators (MACs), which substantially enhance DNN performance by reducing the execution time for these operations.

Beyond matrix multiplications, complete AI applications also involve various other algorithms and operations, such as the Top-K algorithm used in K-Nearest Neighbors (KNN) [11] and Taylor expansions in DNNs [12]. Compared to matrix multiplications, AI processors typically offer less robust support for scalar or vector operations in terms of (1) quantity (e.g., only one 256-bit vector unit per core in Huawei Ascend processors [9]); (2) computa-

tional power (e.g., 22 TFLOPS for FP32 vector units versus 362 TFLOPS for BF16/CFP8 Matrix MACs in Tesla Dojo [4]); and (3) microarchitecture (e.g., the absence of sophisticated features like branch prediction, speculative prefetching, caches, and address coalescing in TPUs [13]). These design choices and trade-offs in AI processors have a substantial impact on algorithm optimization. Therefore, an in-depth understanding of the hardware structure and performance characteristics is essential. This thesis examines Huawei Ascend processors [9] as a case study, thoroughly analyzing its architecture and proposing targeted hardware-aware algorithm optimization techniques.

Huawei Ascend architecture [9] introduces a novel approach to AI processing. Huawei has released two versions of the Ascend processor: the Ascend 310 for edge devices [9] and the Ascend 910B for data center servers [10]. Both feature the DaVinci Core, which provides up to 4 TFLOPS of FP16 computational power for matrix multiplications per core. Recently, Huawei also introduced a comprehensive framework called CANN [14], allowing developers to design custom DNN operators on Ascend processors, thereby enabling algorithm-specific optimizations [15, 16]. However, without a precise performance model that captures the working details and implications, optimizing algorithms for this novel hardware remains challenging.

To address this, the thesis first presents Verrocchio, a performance model specifically designed for the DaVinci Core of Huawei Ascend AI processors. Verrocchio predicts execution times for Ascend Cube-based Compute Engine (CCE) instructions using discrete-event simulation based on a given kernel program. In developing this model, we perform not only standard benchmarks for individual hardware unit performance but also micro-benchmarks to explore bandwidth contention along data paths. The results of this performance model and benchmarking reveal a critical design trade-off in AI processors: substantial performance gains in Matrix MACs come at the expense of reduced support for other essential operations, such as IO, vector, and scalar operations. As a result, applications that require diverse oper-

ations, beyond matrix multiplications, often suffer performance limitations, especially in adapting non-DNN applications.

Taking into account these specific hardware characteristics, this thesis proposes two key approaches for hardware-aware algorithm optimization on AI processors. The first approach involves replacing inefficient scalar and vector operations with alternative scalar and vector operations. Benchmark results indicate that scalar operations exhibit similar low performance across the board, while vectorized comparisons and selections are the most costly. To demonstrate this approach, we introduce SelB-k-NN, a k-nearest neighbors algorithm that minimizes reliance on these less efficient operations. The second approach maps scalar and vector operations onto Matrix MACs by utilizing matrix multiplications. We illustrate this approach with Cube-fx, a mapping algorithm that maps Taylor expansions for multiple functions onto the Matrix MACs. While this second approach significantly enhances performance by leveraging underutilized hardware units, its applicability remains restricted to algorithms that can be efficiently transformed to utilize Matrix MACs. Therefore, both approaches provide valuable strategies for achieving hardware-aware algorithm optimization on AI processors.

1.1 Background

1.1.1 AI Processors

AI applications have rapidly expanded across numerous domains, including natural language generation with models such as GPT-3 [20], GPT-4 [21], and Llama 2 [22], as well as image generation using models like Stable Diffusion [23], Midjourney [24], and DALL·E 3 [25]. Underlying the impressive performance of these applications is a reliance on fundamental matrix multiplications, which require substantial computational power. To address this demand, hardware manufacturers have developed AI processors equipped with specialized AI acceleration units, referred to as Matrix MACs in this

Table 1.1: Specifications of current popular AI processors

AI Processor	**Exec. Model**	**Memory Hierarchy**	**AI Acceleration Unit**	**Other Computation Unit**
Qualcomm Hexagon 780 [17]	SIMT	Seamless shared memory for all acceleration units	Fused AI Accelerator including Scalar, Tensor & Vector	
Google TPU V4 [13]	SIMD	Unified buffer as local storage DRAM Chips as weights inputs	Four (128×128) Matrix Multiply Units SparseCore (composed by specific VPUs)	128-lane Vector Processing Units No complex microarchitectural features
Cambricon MLU290-M5 [18]	SIMD	Scratchpad memories for units separately w/ DMA to IO	Neural Functional Units succeeding to DianNao [19]	Unrevealed
Nvidia H100 GPU [7]	SIMT	GPU memory hierarchy w/ Tensor Core register files	640 ($16 \times 16 \times 16$) Tensor Cores (support sparse matrix)	18,432 CUDA cores 80 Streaming Multiprocessors (SMs)
Baidu Kunlun K200 [8]	SIMD	Separate buffers for inputs, and outputs after activation units	One MAC array with activation and element-wise units	Scalar ALU, Special Function Unit (log, exp, etc.), 256b vector unit
Huawei Ascend [9]	SIMD	Separate L1, L0 buffers for Cube Unit, Unified Buffer for others	Cube Unit based on 16×16 tree-like MACs	256-bit Vector Unit per core, Scalar Unit for addressing and basic arithmetic

thesis, to perform matrix multiplications efficiently.

Table 1.1 summarizes the hardware specifications of several popular AI processors. While GPU manufacturers such as Nvidia and Qualcomm have adopted the Single Instruction Multiple Threads (SIMT) model to extend their established GPU architectures, other AI processor manufacturers have based their designs on the parallelism inherent in SIMD. Compared with the SIMT execution model, SIMD reduces power consumption and minimizes hardware complexity but sacrifices programming flexibility. AI processors generally employ two types of memory hierarchy: shared and separate. Shared memory aims to reduce data transfer costs among hardware units, whereas separate memory architectures provide varied bandwidths and capacities tailored to specific functions.

The designs of Matrix MACs differ significantly across AI processors. Typically, each Matrix MAC has a defined operational "shape" (e.g., $(8 \times 8 \times 4)$ in Tesla Dojo), representing the size of the matrix multiplication block that the MAC can compute in a single step, usually within 100 cycles [6]. Using accumulators, the Matrix MACs then complete larger matrix multiplications through block matrix operations. Most AI processors in the table, except for Google TPU V4, opt for smaller Matrix MAC sizes, such as $(16 \times 16 \times 16)$, to improve utilization rates within neural networks [9]. A smaller systolic array size requires more loop iterations, increasing the complexity of the systolic array's design [26]; hence, Google has chosen a larger (128×128) configuration for their TPUs. However, since the matrix size in many applications may not be perfectly divisible by 128, this configuration can lead to an additional tail computation step, impacting overall efficiency.

Regarding other computation units, Nvidia and AMD GPUs continue to expand their mature parallel computation ecosystems, incorporating a substantial number of compute units for parallelism beyond Matrix MACs. In contrast, other AI processor designers focus on minimizing the cost of additional units to allocate more resources to the Matrix MACs. Compared with GPUs, most AI processors are equipped with only one or two SIMD

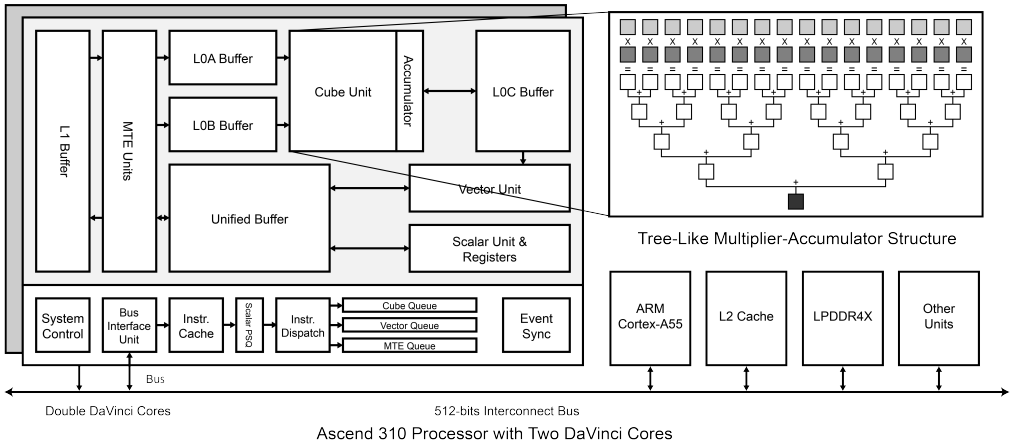


Figure 1.1: The hardware structure of an Ascend 310 processor

vector units and a limited number of scalar units, which provide essential but modest computational power.

1.1.2 Huawei Ascend Processors

The Huawei Ascend architecture builds upon the strengths of existing designs, enhancing the SIMD execution model with a powerful Cube Unit for matrix multiplications, referred to as the Matrix MAC. The DaVinci Core in Huawei Ascend processors integrates several separate memory buffers to support the Cube Unit and a shared Unified Buffer for other computational units. This core is composed of multiple components, including compute, storage, and control units. Figure 1.1 displays the hardware structure of an Ascend 310 processor, while Figure 1.2 shows the logic diagrams for Huawei Ascend 310 and 910B processors.

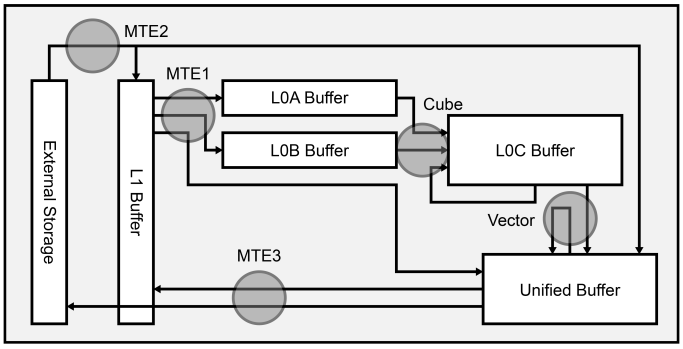
The compute units within the DaVinci Core include the Scalar Unit, Vector Unit, and Cube Unit. The Scalar Unit mainly handles address calculations and conditional branching, while the Vector Unit performs vectorized arithmetic or logical operations, such as data preprocessing and activation functions. At the heart of the DaVinci Core is the Cube Unit, which enhances the efficiency of matrix multiplications. Each Cube Unit consists

of 4096 FP16 MACs and 8192 INT8 MACs, organized as a (16×16) tree-like structure of multiplier-accumulators, as shown in Figure 1.1. Matrix multiplications are divided into multiple smaller units of $(16 \times 16 \times 16)$ for processing, with intermediate results accumulated in the L0C Buffer.

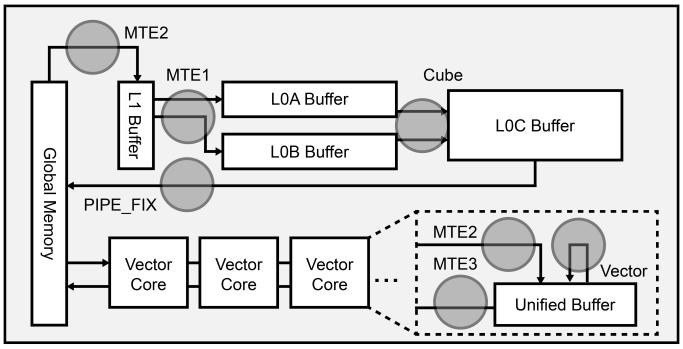
The storage units encompass external and internal storage, as well as Memory Transfer Engines (MTEs) that manage data paths. External storage includes L2 Buffer, DDR, and other storage accessed via the bus, while internal storage comprises five primary buffers (L1, L0A, L0B, L0C, and the Unified Buffer) and various registers. The MTEs (MTE1, MTE2, MTE3, and PIPE_FIX for Ascend 910B only) control data transfer between these internal storage buffers. These five buffers and MTEs work together to bridge the gap between lower memory bandwidth and high compute speed.

The control units manage the coordination between compute and storage units and consist of System Control, the Scalar Program Scheduling Queue (PSQ), Instruction Dispatch, instruction queues (for Cube, Vector, and MTE operations), and Event Synchronization. The Scalar PSQ receives and decodes instructions from the Instruction Cache, and Instruction Dispatch then routes these instructions to the appropriate queues. Event Synchronization handles data dependencies, ensuring correct execution order using a binary semaphore mechanism with eight registers. For Ascend 910B, an additional inter-core synchronization mechanism, based on a slower global memory-supported semaphore, is used to manage coordination across multiple cores.

One major difference in hardware structure between Ascend 310 and Ascend 910B lies in how Cube Units connect with Vector Units. Figure 1.2(a) shows that Ascend 310 features a direct data path between the L0C Buffer and the Unified Buffer, facilitating efficient data transfer from Cube Units to Vector Units. In contrast, Ascend 910B lacks this direct path; instead, Cube and Vector Units must interact through global memory, which decouples these computation units. Consequently, Ascend 910B has a single Cube Core supporting multiple Vector Cores, as illustrated in Figure 1.2(b). This configuration changes the one-to-one mapping of Cube and Vector Units in



(a) Logic Diagram of Ascend 310 Storage Units and Data Path Control



(b) Logic Diagram of Ascend 910B Storage Units and Data Path Control

Figure 1.2: The logic diagrams of Huawei Ascend 310 and 910B hardware structure

Ascend 310 to a one-to-many mapping in Ascend 910B.

Although Ascend 910B's design allows greater flexibility by supporting multiple Vector Cores per Cube Unit, the reliance on global memory for data exchange introduces additional latency, potentially offsetting the benefits of increased parallelism. The design differences mean that Ascend 310's configuration may offer more efficient synchronization and data flow within a single core, while Ascend 910B's structure provides scalability but incurs higher data communication costs.

1.1.3 Exploration and Optimization on AI processors

With the rapid development of AI processors, researchers have shown growing interest in understanding and optimizing this novel hardware across various domains. Similar to previous efforts focused on traditional GPUs [27], researchers have initiated detailed investigations into AI processor architectures, such as Tensor Cores [28]. These investigations have facilitated performance modeling on AI processors, typically divided into two main approaches: white-box and black-box models. White-box models require in-depth knowledge of the hardware, offering high interpretability but limited generalizability across different architectures [29]. Conversely, black-box models rely on historical performance data, requiring little to no hardware-specific information and allowing for broader application, albeit at the cost of reduced transparency [30, 31].

Optimization research on AI processors generally follows two main directions. The first focuses on improving the efficiency of core AI operations, particularly matrix multiplications, which are central to most AI workloads [32–35]. This research aims to fully leverage the native strengths of AI processors by refining computational techniques, optimizing memory access patterns, and improving data management to reduce latency and maximize throughput.

The second research direction explores ways to expand the application scope of AI processors, adapting them for general-purpose computations beyond AI-specific tasks. For example, some studies have adapted reduction and scan operations to Matrix MACs on Nvidia GPUs by segmenting data and processing it across warps and blocks within the CUDA compute hierarchy [36, 37]. Other work has focused on designing dense matrix multiplication techniques tailored to narrow matrix operations, which significantly enhances Matrix MAC utilization [38], or applied matrix multiplications for polynomial convolutions in lattice-based cryptography [39]. Additionally, research by Hu *et al.* [40] investigated using Matrix MACs to accelerate join operations in database processing. These efforts typically involve restructur-

ing data layouts and computation hierarchies to align with the architectural strengths of Matrix MACs, thereby increasing utilization and demonstrating the versatility of AI processors for non-AI applications.

Overall, these studies underscore the potential for AI processors to play a broader role in high-performance computing. By extending their application range through innovative adaptation techniques, researchers continue to explore the boundaries of what AI processors can achieve beyond their primary design.

1.2 Motivations

1.2.1 Hidden Hardware Details of AI Processors

Although hardware manufacturers have published various architectures for AI processors, the public and research community remain largely unfamiliar with these hardware specifics beyond a few general parameters, such as FLOPS. These high-level specifications offer only a broad overview of hardware performance, lacking the detailed information needed for effective algorithm migration and optimization. One reason for this knowledge gap is that most AI processors are not available as consumer-grade physical products but are primarily accessible through cloud platforms. Consequently, performance evaluations conducted in virtualized cloud environments are often influenced by resource-sharing policies, potentially leading to inaccurate assessments that obscure architectural details.

Another factor contributing to this unfamiliarity is the limited functionality of toolchains provided for most AI processors. Many toolchains restrict access to high-level APIs, typically integrated within deep learning frameworks, which prevent developers from performing instruction-level benchmarks to analyze essential hardware characteristics such as bandwidth and throughput. Table 1.2 outlines the availability and API level provided by various popular AI processor manufacturers.

Table 1.2: Availability and API level of AI processor manufacturers

Manufacturer	**Public Availability**	**API Level**
Qualcomm	Mobile SOC [17]	Operations in QNN [41]
Google	Cloud platforms [13]	Operations in TensorFlow or PyTorch XLA [42]
Cambricon	Cloud platforms [43]	Low-level C & assembly [44]
Nvidia	Physical cards [45]	Low-level CUDA & PTX [46]
Baidu	Internal usage	Applications in Baidu AI Cloud [47]
Huawei	Physical cards [9]	Low-level C & assembly [14]

In contrast to Nvidia GPUs, which have been extensively studied [27–29, 48–50], the limited transparency in AI processor details poses significant challenges for algorithm migration and optimization. For example, when implementing a matrix multiplication of size ($256 \times 256 \times 256$), Huawei Ascend processors require more loop iterations to execute the operation compared to Google TPUs, since the Ascend's Matrix MAC size (16×16) is considerably smaller than that of the TPU (128×128). This discrepancy necessitates different tiling parameters for optimal performance. Without access to such hardware details, programmers are unable to determine the most efficient tiling strategies, making algorithm optimization highly challenging.

To address this gap, this thesis begins with a comprehensive dissection of the Huawei Ascend AI processor, utilizing various benchmarks to uncover its specific hardware characteristics.

1.2.2 Inefficient Hardware Units on AI Processors

As standalone processors, AI processors are equipped with additional units beyond Matrix MACs to support necessary operations, such as instruction

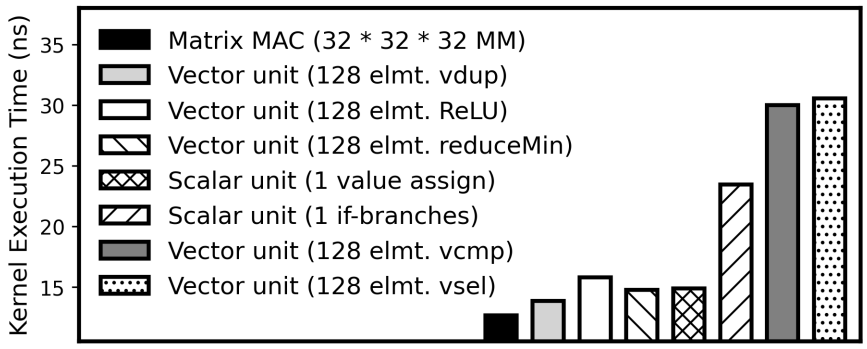


Figure 1.3: The benchmark results on Huawei Ascend 310 AI processors

dispatch and vectorized computation. As discussed in Table 1.1, aside from GPUs, most AI processor designs prioritize hardware resources for enhancing matrix multiplication performance, often at the expense of other units.

To examine the efficiency of these additional units on the Huawei Ascend 310 AI processor, we conducted an instruction-level benchmark. Each test involves repeated executions of specific operations, such as a $(32 \times 32 \times 32)$ matrix multiplication for the Matrix MACs, 128-element ReLU operations for the vector units, and single-operand if-branch instructions for the scalar units. By increasing the number of operations in each test, we derived the execution time per individual instruction based on the measured increase in total execution time.

Figure 1.3 shows the benchmark results. A scalar conditional branch (if-branch) operation exhibits significantly lower efficiency, taking on average 1.48 $\times$ longer than a vectorized `ReLU` operation, 1.69 $\times$ longer than `vecDup`, and 1.60 $\times$ longer than `reduceMin` on 128 elements in parallel. An immediate assignment operation in scalar units takes approximately 0.94 $\times$ the execution time of `ReLU`, despite `ReLU` involving data read, compute, and write operations. Vectorized comparison and selection operations (`vcmp` and `vsel`) are particularly inefficient, taking 1.90 $\times$ and 1.93 $\times$ longer than `ReLU`, respectively.

For other AI processors, TPUs remove all sophisticated microarchitec-

tural features, including branch prediction, speculative prefetching, caches, and address coalescing [13], which were supposed to enhance the scalar operation performance. Meanwhile, although Cambricon instruction set contains vectorized comparisons and selections [44], MLU 100 processors have no built-in hardware support [43].

These findings highlight the limited efficiency of non-MAC units, particularly for scalar and certain vectorized operations. This hardware design constraint poses challenges for migrating traditional algorithms from other platforms to AI processors. Even with successful migration, implementations often suffer from significant performance degradation due to the limited efficiency of these auxiliary units.

1.2.3 Low Usage Rate of Matrix MACs

Since Matrix MACs in AI processors are designed exclusively for matrix multiplications, many algorithms and applications cannot fully leverage this high-performance hardware. Even for algorithms that inherently involve matrix multiplications, the actual usage rate of Matrix MACs is often lower than expected.

Previous studies and official documentation [51, 52] have highlighted the underutilization of Cube Units and the limited performance of Vector Units in Ascend 310. To further investigate this issue, we evaluated two representative applications on Ascend processors: matrix multiplications on Ascend 310 and attention operations on Ascend 910B [53]. These tests provided insight into the performance of Matrix MACs across different matrix dimensions.

Figure 1.4(a) presents the results for matrix multiplications of various sizes on Ascend 310, showing a consistently low utilization rate of Matrix MACs (below 5%). For Ascend 910B, the evaluation includes a breakdown of execution times for Cube Units (denoted as AIC Time) and Vector Units (denoted as AIV Time). As shown in Figure 1.4(b), AIC Time represents the execution time dedicated to matrix multiplications, which is comparable to the results observed on Ascend 310. Even though the attention operation

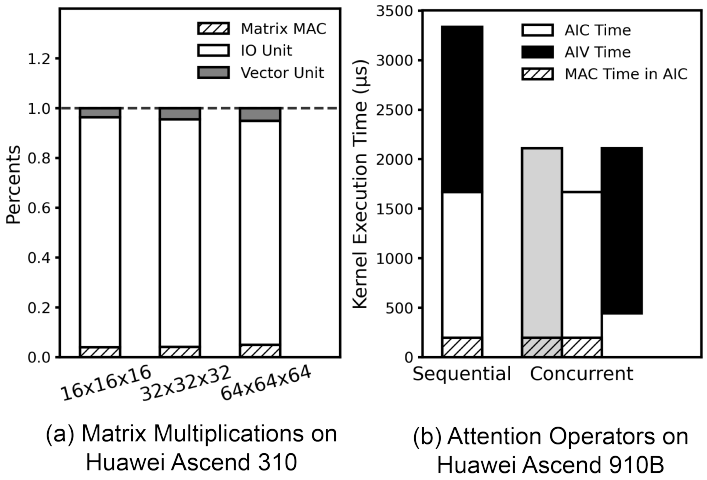


Figure 1.4: The percents of matrix multiplication rates on Ascend 310 and 910B processors

involves substantial matrix calculations, such as $(128 \times 128 \times 128)$ and $(256 \times 128 \times 256)$ multiplications, the Matrix MAC usage rate remains low in both naive sequential and optimized concurrent executions.

These results indicate that on both Ascend 310 and Ascend 910B processors, Matrix MACs are not fully utilized, even in applications that are heavily reliant on matrix multiplications. This underutilization suggests potential opportunities for developing more efficient algorithms that can better exploit the cooperative capabilities of multiple units.

1.3 Main contributions

1.3.1 Performance Modeling on Huawei Ascend

We begin by demystifying the Huawei Ascend 310 processor through micro-benchmarks that reveal key characteristics of the DaVinci Core, which serves as the central AI processing unit in the Huawei Ascend architecture. Particular focus is placed on the IO units, as they play a critical role in influencing

execution time. Our analysis examines how the Memory Transfer Engines (MTEs) manage and compete for data paths that connect the separate memory units. In addition to benchmarks evaluating bandwidth sharing and contention ratios, we designed specialized benchmarks to identify the sources of contention, primarily the Interconnect Bus, and to analyze its runtime behavior under contention.

Building on these insights, we developed a discrete-event-based performance model, **Verrocchio**. Verrocchio captures essential architectural features critical to DaVinci Core performance, including bandwidth contention and concurrent execution of hardware units with synchronization. Evaluation results demonstrate that Verrocchio achieves average error rates of 2.62% and 2.30% for single-core and dual-core executions of sample kernels, respectively. By leveraging Verrocchio, we optimized a matrix multiplication algorithm, enabling search space exploration in tiling size selection. This optimization achieved a speedup of 1.70× at the operator level and 1.53× at the application level compared with CANN's [14] native operators, with prediction error rates of 5.06% and 5.25% for single-core and dual-core execution, respectively.

1.3.2 (1) Replacing Inefficient Operations

SelB-*k*-NN (**Sel**ection-**B**itonic-***k***-**NN**) is a mini-batch k-nearest neighbors algorithm optimized for high-performance AI processors. For each mini-batch, SelB-k-NN incorporates an efficient distance computation step alongside a specialized k-selection method. Inspired by selection sort, SelB-k-NN employs a bitonic 1-selection technique to maintain the min-k result array, comparing minimized values of the computed distances with the initial element of this array [54]. Compared to previous approaches, SelB-k-NN minimizes the reliance on hardware-inefficient scalar operations and vectorized comparisons & selections.

To enhance portability across different AI processor architectures, we designed SelB-k-NN with minimal hardware dependencies. Only basic op-

erations like `vecDup`, `ReLU`, and `reduceMin` (`reduceMax`), without indexing, are required, ensuring compatibility with most AI processors. For all mini-batches, an early exit policy dynamically reduces k-selection workloads, and our quantification shows that this workload reduction is inversely proportional to the data size. We formulated the tiling shape selection as an optimization problem, aiming to improve SelB-k-NN's overall performance. Additionally, an offline pruning method was introduced during preprocessing to reduce the search space of the optimization problem.

1.3.3 (2) Mapping Computations to Matrix Multiplications

Cube-fx is a mapping technique that utilizes the computational power of Matrix MACs to parallelize Taylor expansion. Cube-fx is particularly effective when a dataset requires computation across multiple independent functions [55–57]. It formulates the construction and evaluation of Taylor polynomials as two consecutive matrix multiplications. While matrix multiplication for polynomial evaluation is straightforward, Cube-fx innovates in its approach to matrix generation. During this step, Cube-fx reads input data as contiguous memory segments from external memory, preparing it directly for computation based on Matrix MAC output.

Unlike previous optimization approaches for Matrix MACs, Cube-fx avoids extra matrix generation or data layout conversion, relying solely on a few small constant matrices. Based on precision requirements, Cube-fx adjusts the order of expansion to approximate the original functions effectively. For lower precision requirements, Cube-fx enhances data parallelism by combining inputs and coefficient matrices as direct sums, enabling multiple inputs to be computed within a single set of basic matrix multiplications. Cube-fx is adaptable across various AI processors and is particularly efficient on devices with high-speed communication between vector units and Matrix MACs.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 begins by presenting a detailed analysis and benchmarking of Huawei Ascend AI processors. It then introduces the performance model Verrocchio and provides a comprehensive evaluation of its accuracy. Chapter 3 explores the first optimization approach, which focuses on replacing inefficient scalar and vectorized operations with more suitable alternatives. Chapter 4 presents the second optimization approach, which involves mapping certain scalar and vectorized operations into matrix multiplications for enhanced performance. Finally, Chapter 5 concludes the thesis and outlines directions for future research. Although multi-core execution is considered, the scope of this thesis is limited to a single-node, in-memory computing environment, without addressing distributed platforms or large-scale storage and network IO bottlenecks.

Chapter 2

Performance Modeling on Huawei Ascend

This chapter mainly introduces a performance model, Verrocchio, with complete benchmarking works. For simplicity, the model focuses on the Ascend 310 processors, the simplest version of the architecture. Sec. 2.1 gives an overview of the performance model. Sec. 2.2 reports the benchmark works in detail. Sec. 2.3 discusses the modeling work for both single- and multi-core situations. Sec. 2.4 evaluates the accuracy and acceleration of the performance model with the help of the performance model in matrix multiplication optimization. Sec. 2.5 gives the conclusion.

2.1 Verrocchio, A Performance Model

The specific hardware structure of the DaVinci Core, discussed in Sec. 1.1.2, guarantees high performance for AI applications while leaving several challenges for performance modeling and further optimization. This section first discusses the key challenges we address, how to model the storage unit hardware performance and the concurrent execution synchronization. Then we give an overview of our performance model, Verrocchio, which aims to overcome the challenges for accurate performance modeling.

2.1.1 Key Challenges

1) Storage Unit Hardware Performance

Since different MTE Units control different data paths, the bandwidth of each path requires benchmarking respectively. Especially, multiple MTE Units are allowed to share a single data path concurrently, which would bring bandwidth contentions and significantly influence the hardware performance. As shown in Fig. 1.2 (a), the data path controlled by the on-core bus is the only access for the DaVinci Cores to the off-core 512-bits Interconnect Bus. Fig. 1.2 (b) illustrates that the MTE2 and MTE3 Units both transfer data between the external storage and the DaVinci Core. The architecture suggests a potential contention between the MTE2 and MTE3 Units. However, from the figure of the hardware structure, we cannot postulate the existence of the contention and figure out the contention source: the MTE Units, the on-core bus unit, or the off-core Interconnect Bus. Without the knowledge of the source, simply fitting the performance of the contentions could be inappropriate.

Furthermore, existing studies on the hardware resource contentions [58, 59] mainly focus on the performance variations caused by the different contention ratios when the bandwidths are saturated. In addition to the analyses of the contention ratio, the benchmark of the MTE Unit contention in our performance modeling should pay further attention to the entire runtime of the contention and not only to the stable saturated status. One of the most common methods to monitor the entire runtime of a kernel is sampling [60–62]. However, since the DaVinci Cores have no breakpoint or in-kernel timestamp mechanism, we can hardly insert sampling points into the DaVinci kernels. Therefore, designing a benchmark to measure the runtime of the contention becomes a challenge.

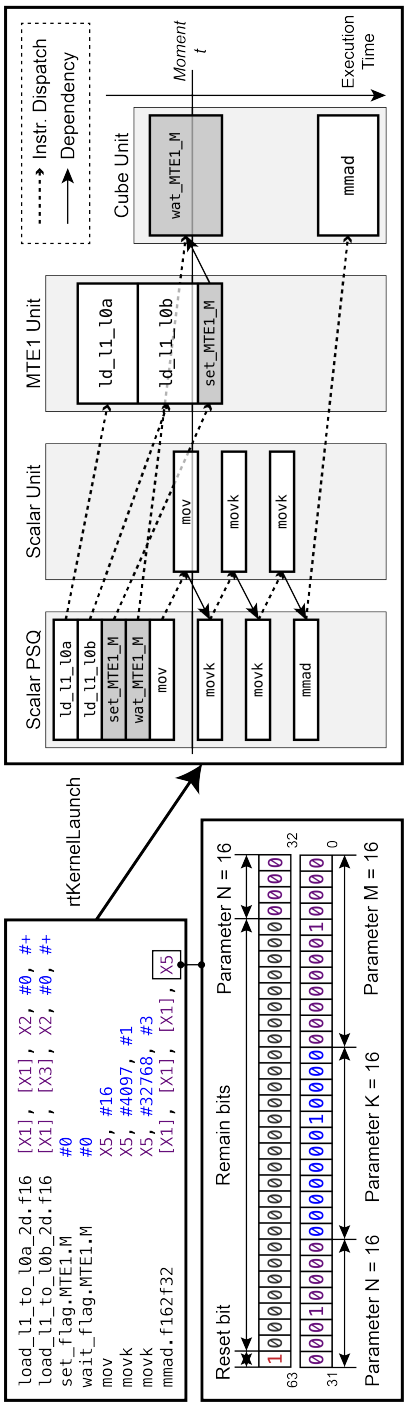


Figure 2.1: The DaVinci Core working details

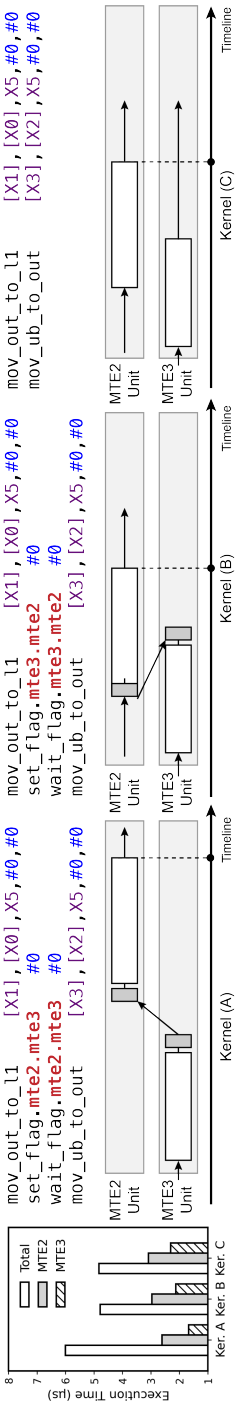


Figure 2.2: The examples of kernels influenced by the binary semaphore

2) Concurrent Execution and Synchronization

As we demystify in Fig. 2.1, all the instruction queues, including the Scalar PSQ, strictly follow the FIFO policy. The Scalar PSQ maintains input kernel instructions and identifies the type of each instruction (Scalar, Vector, Cube, MTE1/2/3), which performs as Instruction Decoder in CPUs. The types of instruction determine the instruction queue to which the instruction is sent, as well as the concurrent unit on which the instruction is executed. For the Scalar instructions, the Scalar PSQ directly sends them to the Scalar Unit. For others, the Scalar PSQ sends them to the corresponding instruction queues via Instruction Dispatch. Each compute or storage unit executes one single instruction at every moment. When a unit is idle, the corresponding queue dequeues the oldest instruction to the unit for execution. The Scalar PSQ asynchronously dispatches the instructions except for the Scalar instructions.

While the DaVinci Core adopts the single-thread execution model, the concurrency of the hardware units makes the Instruction-Level Parallelism (ILP) supported. To guarantee the dependency among the units, the programmers must manually insert the `set_flag` and `wait_flag` instructions, acting as the binary semaphore mechanism. The fundamental operations of the semaphore mechanism are PV Operations [63]. P Operation decrements the value of the semaphore variable by 1. If the new variable is negative, it blocks the caller process. Meanwhile, V Operation increments the value of the semaphore variable by 1. If the old variable is negative, it wakes up a blocked process and lets it access a resource unit. In DaVinci Cores, the `wait_flag` acts as P Operation and the `set_flag` performs as V Operation. The Scalar PSQ assigns the two semaphore operations to the corresponding instruction queue just as it does with other instructions. In the example of the moment `t`, following the FIFO policy, the `set_flag` (`set_MTE1_M`) in the MTE1 queue shall not start due to the incompleteness of the last instruction `ld_l1_l0b`. The corresponding `wait_flag` (`wat_MTE1_M`) keeps blocking the

Cube Unit until the execution of `set_flag`. Therefore, the instruction `mmad` shall not start before the end of the `ld_l1_l0b`.

The idleness of the hardware unit caused by the synchronization is one of the most critical factors influencing the execution time. Each instruction could prolong or block the execution based on its programming logic. Furthermore, the concurrent execution can be intentionally activated or deactivated, which is decided by how the kernel programs schedule the multiple semaphore registers and control flows. Fig. 2.2 illustrates a simple DaVinci kernel containing two data transfer operations (on MTE2 and MTE3 Units) with different binary semaphore operations. As a result, Kernel (A) takes 1.26× more time than Kernel (B) and 1.24× more than Kernel (C), while the actual two data transfer operations take similar time among all kernels. For Kernel (A) and Kernel (B), the parameter order (`mte2.mte3` & `mte3.mte2`) of the semaphore operations determines the execution time. The parameter order `mte2.mte3` prohibits the concurrent execution while the reversed `mte3.mte2` does not, which reports a similar result to Kernel (C). Therefore, the DaVinci Core working details make the modeling methods based on kernel metadata analysis or statistics ineffective, e.g., the most straightforward execution time computed by $DataSize/Bandwidth$. From the view of those modeling methods without analysis in the instruction logic, Kernel (A) and (B) should have a similar total execution time, which is false in our evaluations. Then it is necessary to model the performance with the instruction-level analyses in the kernel source codes.

2.1.2 Overview of Verrocchio

Verrocchio is a discrete-event-based performance model, where events occur due to occurrence time with execution periods at the discrete points in time. Fig. 2.3 illustrates the structure of our DaVinci Core model. Verrocchio has two inputs, the target program source codes and the hardware benchmark results. It predicts and outputs the execution period of the kernel program.

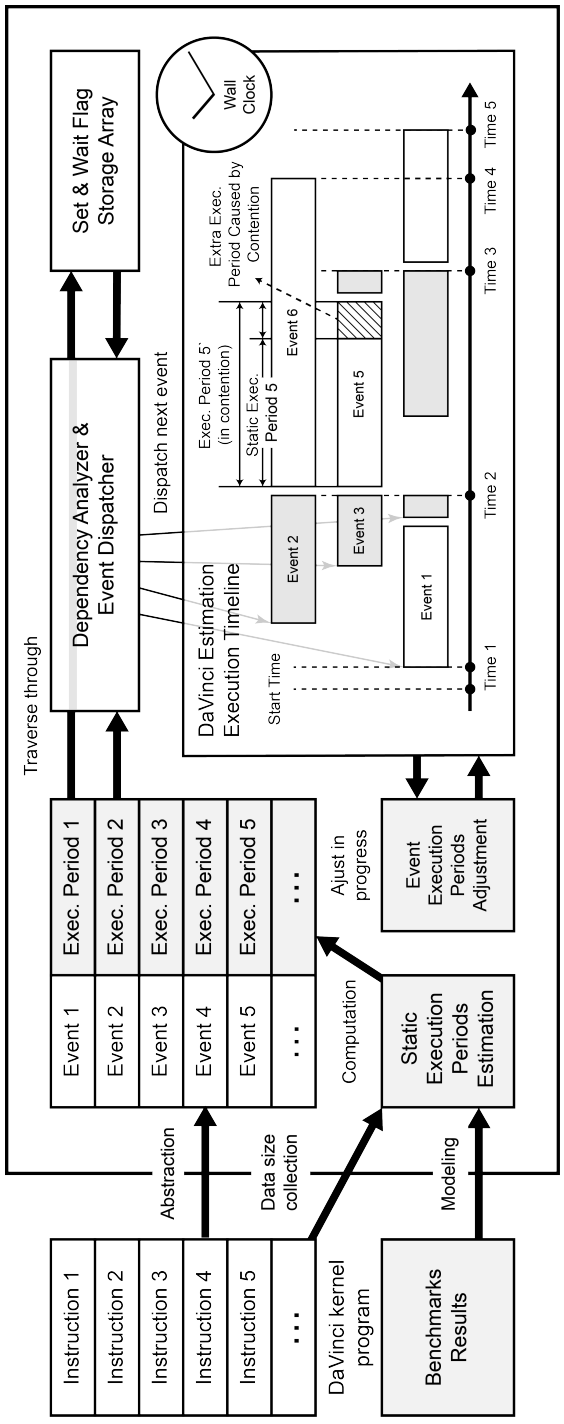


Figure 2.3: Verrocchio, a discrete-event-based performance model

To provide instruction-level modeling for the kernel program, Verrocchio first scans the instructions of the kernel program to abstract them into discrete events. It also collects the data size for each instruction. Verrocchio applies the results from our instruction-level micro-benchmarks, which target to expose the hardware performance, especially the DaVinci Core storage unit performance. Combining the data size and the benchmark results, Verrocchio estimates the static execution period of each instruction, which is the execution period when the instruction executes alone.

Two core components of Verrocchio are the dependency analyzer and event dispatcher, which manage the prepared discrete events and place them on a DaVinci execution timeline at the correct timestamp with a wall clock. To handle the synchronization supported by the binary semaphore, Verrocchio maintains two arrays from the analysis of the prepared events. The two sorted arrays store the Set Flag and Wait Flag operations and guide Verrocchio correctly start an event when meeting the binary semaphore conditions. As we discussed, during the procedure, the event execution periods can be updated because of the bandwidth contention, as the example of Event 5 shown in Fig. 2.3. To model the contentions, Verrocchio offers the execution periods adjustment, which updates the new execution periods each time the contentions begin and end.

2.2 Benchmarking DaVinci

To reveal the performance of the storage unit, we benchmark the Ascend 310 processor by launching a series of instruction-level micro-benchmarks. For the hardware units, whose performance reports a regular and stable trend, the benchmark increases the processed data size and computes the bandwidth by the least-square method. For the bus bandwidth, which connects to the external storage and involves the contention by the MTE2 and MTE3 Units, we first design a specifically-crafted benchmark to identify the contention source and expose the runtime behaviors. Then we adopt further benchmarks

Table 2.1: Ascend 310 Cube Unit benchmark results

Benchmark Types	Double-Core	Single-Core	Percentages
MTE1 bw. (L1 Buffer to L0A Buffer)	695.97 GB/s	347.99 GB/s	99.99%
MTE1 bw. (L1 Buffer to L0B Buffer)	348.79 GB/s	174.37 GB/s	100.00%
Vector bw. (None)	348.16 GB/s	174.06 GB/s	100.00%
Vector bw. (F32 to F16)	348.2 GB/s	174.09 GB/s	100.00%
Cube Unit practical FLOPS (RESET off)	10796.92 GFLOPS	5390.32 GFLOPS	100.00%
Cube Unit practical FLOPS (RESET on)	10789.17 GFLOPS	5397.34 GFLOPS	99.99%
Ascend 310 documented FLOPS	-	4096.00 GFLOPS	-
Kernel launch time	2293.5 ns	2354.5 ns	-

with different contention ratios to quantify the bandwidth sharing under contentions.

2.2.1 Benchmarks for Hardware Units

Methodology

For the MTE1 Unit, we directly adopt the micro-benchmark kernels. For the Vector Unit, we list the data movement bandwidth here and with a precision converting function from FP32 to FP16. For the Cube Unit, we benchmark the FLOPS by modifying the `mmad` instruction parameters. The computation amount of each minimized $16 \times 16 \times 16$ matrix multiplication is $(15 + 16) \times (16 \times 16) = 7936$ FLOPs. After the single-core benchmarks, we activate and benchmark two DaVinci Cores on the Ascend 310 processor. We compare the results with the single-core results to check whether the double-core execution can approximately double the bandwidths or FLOPSs. In addition, we also benchmark the kernel launch time of double-core execution compared with the single-core execution.

Benchmark Results

We list the benchmark results in Table 2.1. We summarize two main observations of the results as follows:

- As we expected, the independent hardware units report perfectly double acceleration. These hardware units require on-core resources only (bandwidths and computation power) and shall not suffer performance loss in the multi-core execution situation because of the Interconnect Bus.
- The kernel launch time results report that the double-core execution shows no apparent kernel launch time extension compared with the single-core execution. We assert that the kernel launch time does not suffer from any contentions in double-core execution. As DaVinci Cores have no inter-core communication mechanism, we assume that the two DaVinci Cores on Ascend 310 processors start and execute the kernel synchronically.

2.2.2 Benchmarks for Bus Contention Source and Runtime Behaviors

While the bandwidth of the individual bus execution can be evaluated naturally with the benchmark in Sec. 2.2.1, the contention caused by the multiple units is non-trivial to be measured. In this subsection, we design a benchmark to study the source and the runtime behaviors of the contentions observed at the DaVinci Cores bus, which are caused by the MTE2 and MTE3 Units.

Benchmark Design

The objective of the benchmark is to monitor the behaviors of the entire contention runtime, from the start to the end of it, with a fixed contention ratio of two (the MTE2 and MTE3 Units). As shown in Fig. 2.4, on one of the hardware units (e.g., the MTE2 Unit), we first let the unit process

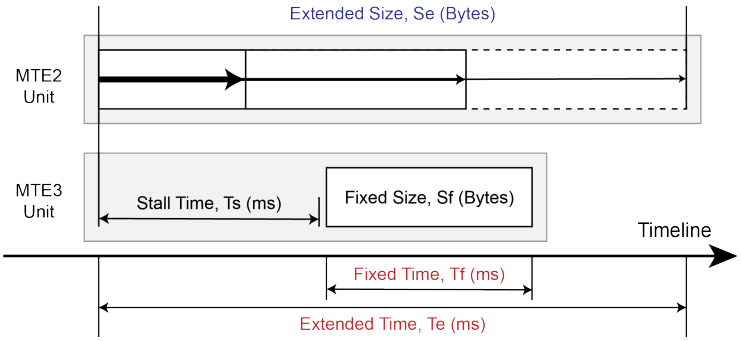


Figure 2.4: Bandwidth contention monitoring benchmark (e.g., MTE2 & MTE3 Units)

Algorithm 1: Bandwidth Contention (e.g., MTE2 & MTE3 Units)

input : Stall time, T_s; Fixed size, S_f

output: Extended time array: A_e; Fixed time array: A_f

1 **for** $Se : 0 \rightarrow MAX$ **do**
2 **Barrier call**
3 **(MTE2)** Process data of size S_e
4 **(Scalar)** Dummy instruction
5 **(Scalar)** $\cdots$ $\cdots$ (Block instruction 7 for T_s)
6 **(Scalar)** Dummy instruction
7 **(MTE3)** Process data of size S_f

an increasing size of data, S_e, and then keep it idle until the end. For the other hardware unit (e.g., the MTE3 Unit), which is concurrently executing alongside, it first stays idle for a short period T_s and then processes a segment of data with the size of S_f until the end. On the DaVinci Cores, we insert dummy Scalar instructions between the instructions of the former unit and the latter, as annotated in Alg. 1. As discussed, the Scalar PSQ must wait until the end of the Scalar instructions and proceed to the next instruction. Therefore, the dummy instructions block the Scalar PSQ and keep the latter unit idle. When the former unit has been executing for a period, the latter unit finally receives its instruction from the Scalar PSQ and then executes with the designed contention. We measure and record the execution time of two units, T_e and T_f with the Huawei official profiler respectively, with the increment of S_e.

Divided by T_e, the two hardware units experience three contention statuses: before, under, and after contention. When $T_e < T_s$, where S_e is not large enough, the two hardware units stay the status before contention. When $T_e < T_s + T_f$, the two units are under contention. When $T_e > T_s + T_f$, the two units finish the contention periods at the status after contention. Therefore, after finishing the benchmark in Alg. 1, the recorded execution time T_e and T_f can expose the entire runtime behaviors during the bandwidth contention. In addition, we vary the input parameters T_s and S_f to study the potential influences by the contention beginning time and processed data size under contention.

Contention Source and Runtime Behavior Results

Based on the fundamental benchmark designed in Sec. 2.2.2, we illustrate the benchmark results of the MTE2 and MTE3 Units with the different corresponding data paths in Fig. 2.5. We insert NOP instructions as dummy instructions, the number of which (related to T_s) are annotated in the legend of figures.

We first call the instructions for transferring data between the external

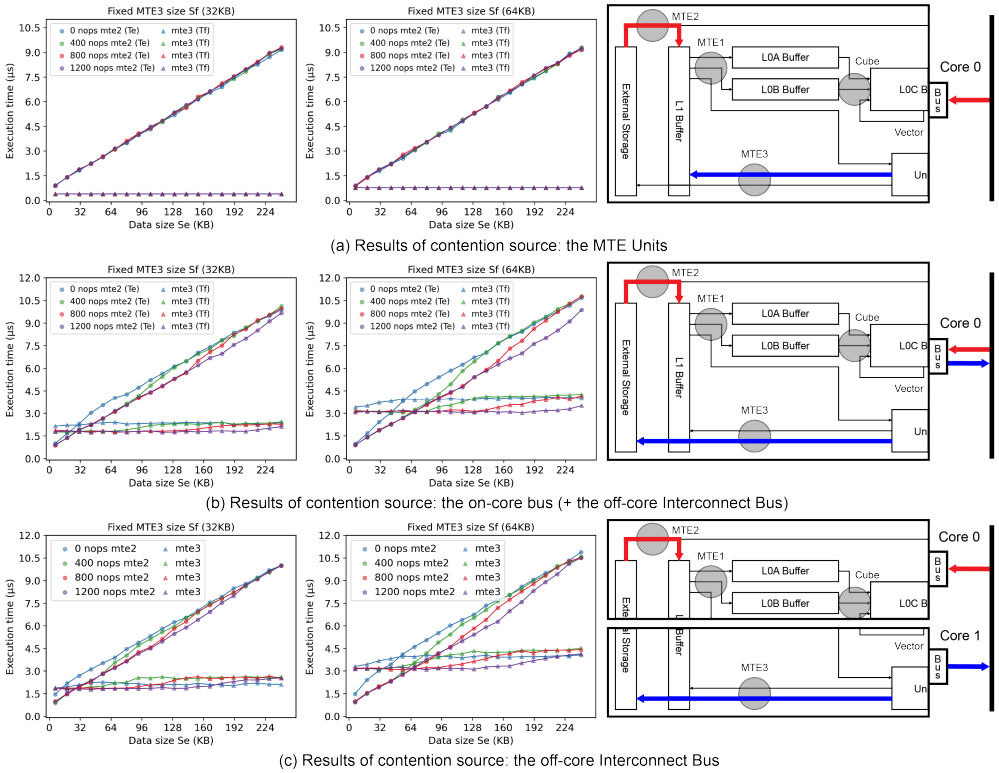


Figure 2.5: Contention source results with the corresponding data paths

storage and the L1 Buffer (MTE2) and that between the Unified Buffer and L1 Buffer (MTE3) concurrently, which does not involve other data paths. The results in Fig. 2.5 (a) show that with the variation of S_f, S_e, and T_s, the evaluated execution time T_f and T_e remain unchanged, which indicates no contention happens between the two MTE Units.

By contrast, from the results in Fig. 2.5 (b) and (c), we observe apparent irregular execution time variations, which present clear bandwidth contention runtime behaviors. For both the MTE2 and MTE3 Units, it is easy to identify the turning points of the three contention statuses we described in Sec. 2.2.2. With the increment of the data size S_e, the MTE2 Units report two different slopes in three-segment polylines, where the first segment is before contention, the second is under contention, and the third is after

contention. For the MTE3 Units with a fixed data size S_f, which were supposed to take a stable execution time, the contentions bring an extra execution time increment and also result in three-segment polylines. The number of the NOP instructions determines T_s and the length of the first segment. The increment of the MTE3 fixed data size S_f extends the second segment, which shows a lower bandwidth because of the sharing of the limited bandwidth under contention. The third segment's slopes are the same as the first for both the MTE2 and MTE3 Units, indicating that the bandwidths recover from the contention without extra side effects.

For the contention source, since it is impracticable to transfer the data without the off-core Interconnect Bus independently, the results shown in Fig. 2.5 (b) are introduced by both the on-core buses and the off-core Interconnect Bus. Meanwhile, Fig. 2.5 (c) involves the off-core Interconnect Bus only by placing the two instructions on two DaVinci Cores separately. In this way, in Fig. 2.5 (c), the on-core bus of Core 0 transfers data to the DaVinci Core with the MTE2 Unit individually and avoid any possibilities for contentions. Comparing the results of Fig. 2.5 (b) and (c), the execution time, the variation trends, and the slopes are similar, which suggests they activate the same contention source. Therefore, we assert that the contention source of the Ascend 310 processor bus bandwidth is the common part of Fig. 2.5 (b) and (c), the off-core Interconnect Bus.

Fig. 2.6 shows the results of the second benchmark, where we repeat the benchmark shown in Fig. 2.5 (b) but increase T_s with tiny steps to explore the execution patterns of the MTE Units under contention. Fig. 2.6 (a) plots the fixed-sized MTE2 Unit execution results (S_f = 32 KB) with the increasing MTE3 Unit data size S_e while Fig. 2.6 (b) is symmetric. With the increment of T_s, the beginning of the contention delays, which increases the start points of the second segment we discuss in Sec. 2.2.2 accordingly. However, from the results for the two cases, we observe that although T_s grows evenly, the start points of the second segments are not averagely distributed along the x-axis but gather at specific points, especially

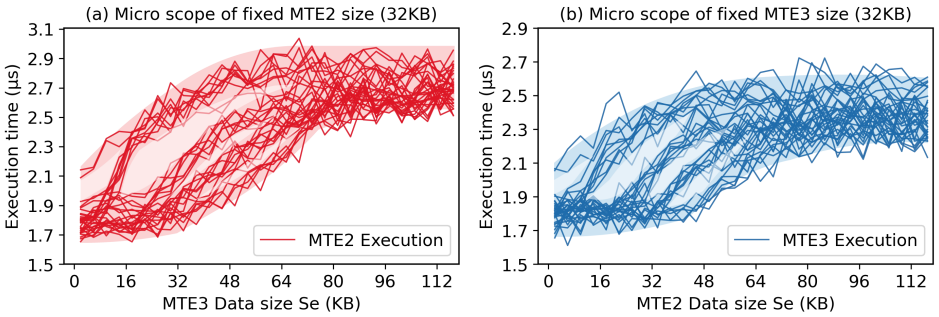


Figure 2.6: The execution of the MTE Units with tiny T_s increments

for the MTE3 Unit. The execution patterns show that the MTE Unit does not start the data transfer and forms a contention instantly after execution, suggesting the data is transferred in blocks or segments. We assert that before block transferring, the Interconnect Bus must manage and decide how to share the bandwidth based on the number of executing Units each time.

2.2.3 Benchmarks for Bandwidth Sharing under Contention

After identifying the contention source and runtime behaviors, the maximum contention ratio ($R_{max} = 4$) and the participated hardware units (the MTE2 & MTE3 Units on two DaVinci Cores) are also determined. We further analyze the detailed policy for the potential participants to share the bandwidth.

Methodology

We follow the benchmark used in Sec. 2.2.1, where we increase the processed data sizes and compute the bandwidths under contention using the least-square method. We control the active hardware units, which are involved in the contentions of each case, with different instructions on two DaVinci Cores, respectively. Especially for the MTE3 Units, the benchmarks transfer

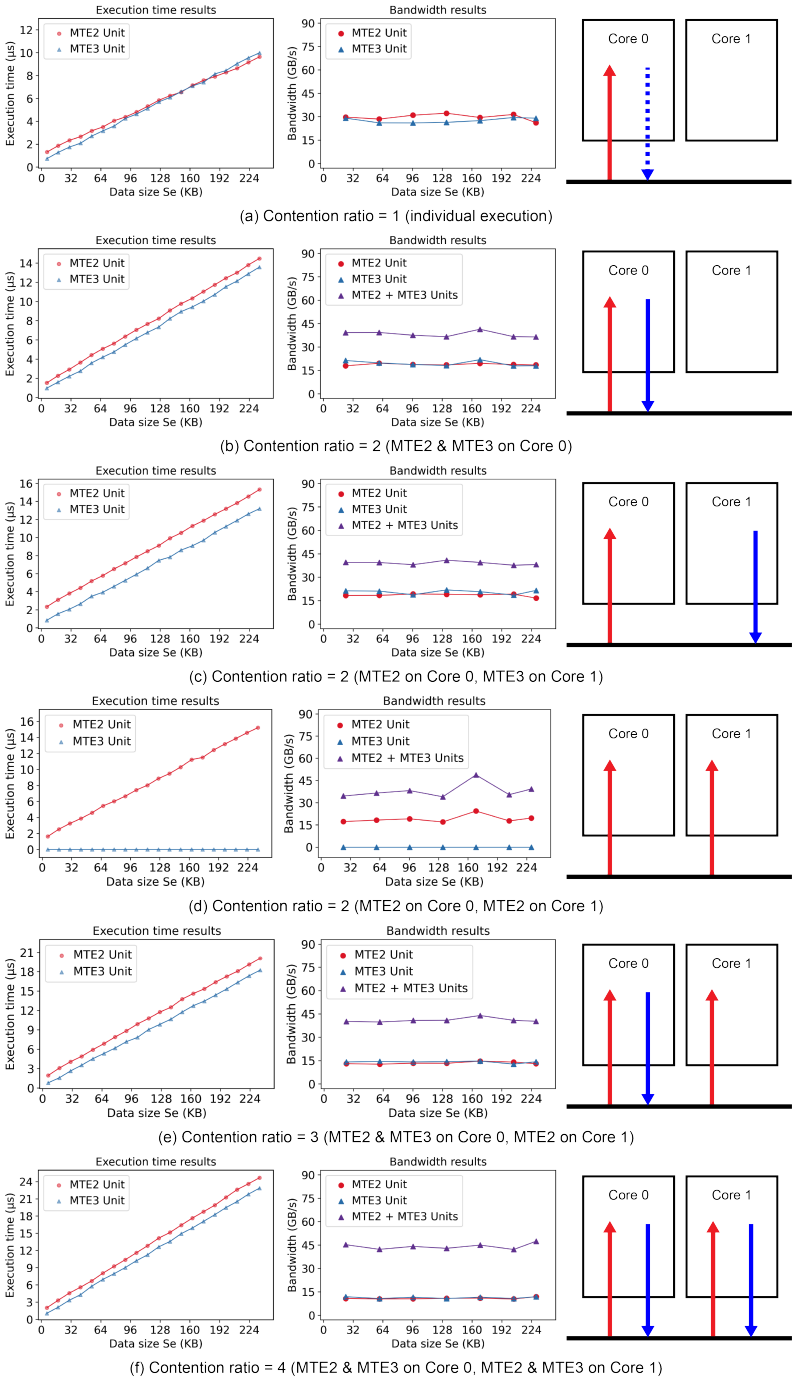


Figure 2.7: Bandwidth sharing of the Ascend 310 processors under contention

the data from the Unified Buffer to the external storage, similar to Fig. 2.5 (b).

Bandwidth Sharing Results

Fig. 2.7 illustrates the benchmark results for six cases, which include all potential cases under contention. The bandwidth result of each MTE Unit plotted in the chart is the bandwidth of a single operation. Therefore, for the example in Fig. 2.7 (e), the total bandwidth is computed by $2 \times bw_{MTE2} + bw_{MTE3}$. Generally, with the increment of the contention ratio, the total bandwidth (MTE2 + MTE3 Units) also grows with a maximum bandwidth of about 42 GB/s when the contention ratio is 4, shown in Fig. 2.7 (f).

The bandwidths of writing and reading operations, controlled by the MTE2 and MTE3 Units, are almost equal in all cases while sharing. When the contention ratio is 2, Fig. 2.7 (b, c, d) present three different combinations of the MTE operations, MTE2 and MTE3 operations on Core 0, MTE2 and MTE3 operations on Core 0 and 1, and two MTE2 operations on Core 0 and 1. In contrast to the different units participating, the bandwidth sharing results are the same. Either operation occupies half the total bandwidth in each case, no matter what kind of operation it is or which core the operation is executing. Fig. 2.7 (e) reports the bandwidth results when the contention ratio is 3, where we place two MTE2 operations on two DaVinci Cores with an MTE3 operation on Core 0. The bandwidth of the MTE2 operation is still equal to the MTE3 operation, which implies that the three operations trisect the bandwidth of the Interconnect Bus. Fig. 2.7 (f) also shows a similar averagely sharing result. Therefore, we assert that the Interconnect Bus is a half-duplex bus, where the writing and reading operations can execute concurrently but influence each other. Our later bandwidth contention modeling work is based on the conclusion of the benchmark, where the bandwidth is shared equally by each Unit.

2.3 Modeling DaVinci

After benchmarking the hardware units, this section formally describes the performance model, Verrocchio, especially how it manages the concurrent execution and synchronization.

2.3.1 Verrocchio Notation and Parameter Definitions

We list the notations used in Table 2.2 and part of the hardware parameters used in Table 2.3. Each instruction of the kernel program C is considered as a discrete event $E_{u,n}$ with an operated data size $Ds_{u,n}$. Since the kernel program is determined, the order of the instructions on each unit is confirmed. Verrocchio uses the pair of u, n to locate instructions. The wall clock updates the death time $T_{u,n}$ of each event.

2.3.2 Verrocchio Main Performance Modeling Procedure

We show the Main Performance Modeling Procedure in Alg. 2. Since the wall clock time when the kernel program finishes is the wall clock time when the last event finishes, which is the largest death time $T_{u,n}$. We assert that the execution time of the kernel program C is the maximum death time of the last Event E_{u,N_u} among all Units, as shown in Alg. 2 line 12.

For all Events $E_{u,n}$, we divide them into three categories, normal computation or memory operations, Set Flag operations for `set_flag` instructions, and Wait Flag operations for `wait_flag` instructions. The last two categories are the fundamental concepts of the binary semaphore mechanism. Different categories have different procedures to update its death time $T_{u,n}$. We give $E_{u,n}$ a 3-tuple (u_a, u_b, r) as its value to classify the categories, where u_a, u_b are the related Units of a Set Flag or Wait Flag operation, r is the corresponding semaphore register, as shown in Fig. 2.8 (a).

Table 2.2: Notations used in Verrocchio Performance Model

Notations	Description	Value
u	The unit number	$0 < u \leq U$
N_u	The instruction number of the unit u for a kernel program	$0 \leq N_u$
n	The order number of an instruction on the unit u	$0 < n \leq N_u$
$E_{u,n}$	A discrete event, an abstraction of the DaVinci instruction, which is the n-th event executes on the Unit u	(u_a, u_b, r)
$T_{u,n}$	The death time of $E_{u,n}$, which is the wall clock time when $E_{u,n}$ finishes	$0 \leq T_{u,n}$
$P_{u,n}$	The execution period of $E_{u,n}$	$0 \leq P_{u,n}$
C	The input instructions, an ordered array of a series of $E_{u,n}$	$\langle E_{u0,n0}, E_{u1,n1}, ... \rangle$
$Ds_{u,n}$	The operated data size of $E_{u,n}$	$0 \leq Ds_{u,n}$
$S_{E_{u,n}}$	The sorted array to store the subscripts (u, n) of all Set Flag operations with the same value (u_a, u_b, r)	$\langle (u0, n0), (u1, n1), ... \rangle$
$W_{E_{u,n}}$	The sorted array to store the subscripts (u, n) of all Wait Flag operations with the same value (u_a, u_b, r)	$\langle (u0, n0), (u1, n1), ... \rangle$

Table 2.3: Verrocchio hardware parameter definitions of the Ascend 310 processors

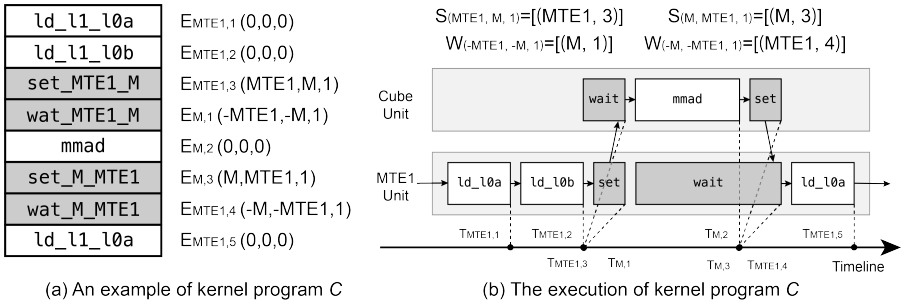
Parameters	Description	Value
U	The total unit number	5
r_{max}	The maximum binary semaphore register number	8
$Init$	The initialization time of each instruction	40 ns
T_{kernel}	The initialization time of DaVinci kernel	2050 ns (inc. 1000 ns for MTE2, 350 ns for MTE3)
Th_u	The throughput (or bandwidth) of the Unit u	Partially Listed in Table 2.1

Algorithm 2: Verrocchio Main Performance Modeling Procedure

input : Kernel program $C = \langle E_{u0,n0}, E_{u1,n1}, ... \rangle$
Operated data size $Ds = \langle Ds_{u0,n0}, Ds_{u1,n1}, ... \rangle$

output: Kernel wall clock finish time: T_{end}

$(T_{u_0,n_0}, E_{u,n}) \leftarrow (T_{kernel}, E_{u_0,n_0})$
while $E_{u,n}$ ***is not*** *the last event in* C **do**
  case $E_{u,n} = (0,0,0)$ **do** // normal operations
    $P_{u,n} \leftarrow Ds_{u,n} \,/\, Th_u + Init$
    $T_{u,n} \leftarrow T_{u,n-1} + P_{u,n}$
  case $E_{u,n} > (0,0,0)$ **do** // set flag operations
    $T_{u,n} \leftarrow T_{u,n-1}$
  case $E_{u,n} < (0,0,0)$ **do** // wait flag operations
    $E_{u',n'} \leftarrow -E_{u,n}$
    $T_{u,n} \leftarrow T_{u',n'} \leftarrow T_{S_{-E_{u,n}}[Index((u,n),W_{E_{u,n}})]}$
  $E_{u,n} \leftarrow$ *the next event in* C
Bandwidth_Contention_Check$(E_{u,n})$
$T_{end} \leftarrow \max_{0<u\leq U} T_{u,N_u}$



(a) An example of kernel program C (b) The execution of kernel program C

Figure 2.8: An example of Verrocchio main performance modeling procedure

Normal Computation Or Memory Operations

For the normal operations, we set $E_{u,n} = (0, 0, 0)$, which means $E_{u,n}$ is not related to the binary semaphore mechanism. For each normal computation or memory operation $E_{u,n}$, we first compute the corresponding static execution period $P_{u,n}$ to quantify how long the $E_{u,n}$ could occupy the Unit u. In general, $P_{u,n}$ is computed as shown in Alg. 2 line 4. According to the FIFO policy, the incidence of $E_{u,n}$ should be immediately after the death of $E_{u,n-1}$. Hence, we compute the death $T_{u,n}$ of $E_{u,n}$ as the sum of the last death time $T_{u,n-1}$ on the same Unit u and the event execution period $P_{u,n}$, as shown in Alg. 2 line 5.

Set Flag Operations

For the Set Flag operations, we set $E_{u,n} > (0, 0, 0)$. For the example in Fig. 2.8 (a), the instruction `set_MTE1_M` is converted to $E_{MTE1,3} = (MTE1, M, 1)$, which means the MTE1 Unit has finished its work, and the Cube Unit should terminate its corresponding Wait Flag operation and continue its next operation with Register 1. Since the Set Flag operations do not suspend or finish any workloads, the death time $T_{u,n}$ equals the last death time $T_{u,n-1}$.

Before the Main Performance Modeling Procedure, we create and maintain an array $S_{E_{u,n}}$ to store the Set Flag operations with the subscript of the 3-tuple (u_a, u_b, r). We store the 2-tuple (u, n) of $E_{u,n}$, the identification of

the Set Flag operation. We have the definition of $S_{E_{u,n}}$ shown below:

$$\begin{aligned} S_{E_{u_1,n_1}} =& S_{E_{u_2,n_2}} = S_{(u_a,u_b,r)} = \langle (u_1,n_1),(u_2,n_2),...\rangle \\ & \forall E_{u,n} > (0,0,0), n_i < n_j, \forall i < j \end{aligned} \tag{2.1}$$

where the array $S_{E_{u,n}}$ is sorted according to the operation order number n.

Wait Flag Operations

For the Wait Flag operations, we set $E_{u,n} < (0,0,0)$, which is symmetric to the Set Flag operation definition. For the example in Fig. 2.8 (a), the instruction `wait_MTE1_M` is converted to $E_{M,1} = (-MTE1, -M, -1)$. We also create and maintain an array $W_{E_{u,n}}$ to store the 2-tuple (u,n), as shown below:

$$\begin{aligned} W_{E_{u_1,n_1}} = W_{E_{u_2,n_2}} =& W_{(-u_a,-u_b,-r)} = \langle (u_1,n_1),(u_2,n_2),...\rangle \\ & \forall E_{u,n} < (0,0,0), n_i < n_j, \forall i < j \end{aligned} \tag{2.2}$$

For a Wait Flag operation $E_{u,n}$, its corresponding Set Flag operation $E_{u',n'}$ must exist, where $E_{u,n} = -E_{u',n'}$, as shown in Alg. 2 line 9. The array S and W are used to locate the related Set Flag and Wait Flag operation pair. Since DaVinci Cores have no memory heap or stack structure but only registers to store the semaphores, the Wait Flag operation order number in $W_{E_{u,n}}$ equals the order number of its corresponding Set Flag operation in $S_{E_{u',n'}}$. We introduce a function $Index(x, Array)$ to return the position of x in an array $Array$ without duplicated elements, then we have the relation:

$$Index((u,n), W_{E_{u,n}}) = Index((u',n'), S_{E_{u',n'}}) = Index((u',n'), S_{-E_{u,n}}) \tag{2.3}$$

Hence, we have:

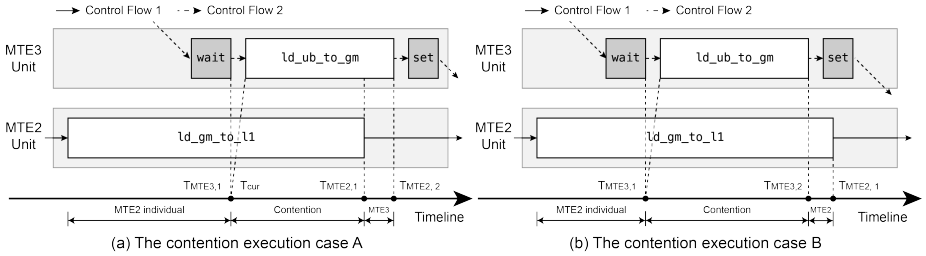


Figure 2.9: Examples of Verrocchio contention execution

$$(u', n') = S_{-E_{u,n}}[Index((u, n), W_{E_{u,n}})] \tag{2.4}$$

Since a Wait Flag operation blocks the Unit until the corresponding Set Flag operation is done, the death time of the Wait Flag operation equals the Set Flag operation death time. For the example in Fig. 2.8 (b), we compute $T_{M,1}$ as follows:

$$T_{M,1} = T_{S_{-E_{M,1}}[Index((M,1),W_{E_{M,1}})]} = T_{S_{(MTE1,M,1)}[0]} = T_{MTE1,3} \tag{2.5}$$

Bandwidth Contention Updating

As we discussed, the bandwidth contention of the Interconnect Bus influences the performance of the MTE2 and MTE3 Units significantly. The runtime behaviors we expose in Sec. 7 show that each time the two Units start to run concurrently, the contention occurs, and the related bandwidth changes; when either of the Units finishes, the contention ends, and the Units regain the original bandwidth. For the example in Fig. 2.9 (a), at $T_{MTE3,1}$, when the MTE3 starts to execute the instruction `ld_ub_to_gm`, the contention begins; when the MTE2 Unit finishes its work at $T_{MTE2,1}$, the contention is finished. Therefore, the contention status changes at the event death time. Verrocchio checks whether a contention appears or disappears when an event is finished to update the death time of the related events.

We first compute the left data size of the event, which is influenced by the contention status change. For an event $E_{u,n}$, we compute the left data size as shown below:

$$Ds'_{u,n} = Ds_{u,n} \times \frac{T_{u,n} - T_{cur}}{P_{u,n}} \tag{2.6}$$

where T_{cur} is the current global time, also the death time of the latest event triggering the bandwidth contention, e.g., $T_{MTE3,1} = T_{cur}$ in Fig. 2.9 (a), $Ds'_{u,n}$ is the data size left for the new contention status. Then we update the event death time as follows:

$$T'_{u,n} = T_{cur} + Ds'_{u,n} \,/\, Th'_u \tag{2.7}$$

where Th'_u is the bandwidth of the Unit u at the new contention status, $T'_{u,n}$ is the new death time. For Th'_u, we strictly follow our conclusions in Sec. 2.2.3, where multiple Units equally share the bandwidth. The potential executions with a contention are divided into two cases shown in Fig. 2.9, based on the relationship of the two events' execution period. For the case in Fig. 2.9 (a), both units suffer one contention status change; for Fig. 2.9 (b), one unit suffers two contention status changes, experiencing the three-segment polyline of bandwidths discussed in Sec. 7.

2.3.3 Multi-core Execution Performance Modeling

According to our benchmark results, the DaVinci hardware units can be classified into two categories, the one which relies on the independent on-core resources and the other which occupies the Interconnect Bus and causes contentions. For the first category, the throughput Th_{single} of every single core does not change (as a const value C_{single}) in the multi-core execution, and the total unit throughput Th_{total} of all cores increases proportionally.

$$Th_{single} = C_{single} \qquad Th_{total} = N \times Th_{single} \tag{2.8}$$

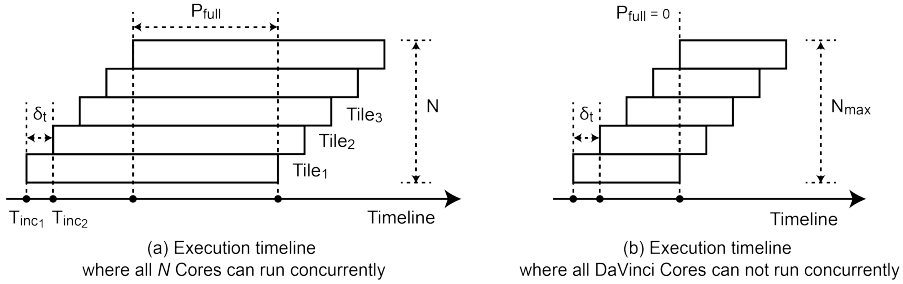


Figure 2.10: Execution timeline of $E_{u,n}$, which is tiled and submitted to multiple DaVinci Cores

where N is the number of the activated DaVinci Cores. For the second category, as our benchmarks report in Sec. 2.2.3, the throughputs of all single core Th_{single} averagely share the total throughput.

$$Th_{single} = \frac{Th_{total}}{N} \qquad Th_{total} = C_{total} \tag{2.9}$$

Like most multi-core accelerators, we assume that multi-core DaVinci processors continue to adopt the SIMD execution model, which means the programs are tiled and distributed to multiple cores. We focus on a general normal computation or memory operation $E_{u,n}$, submitted to multiple DaVinci Cores, as shown in Fig. 2.10, where each bar is a tile of $E_{u,n}$, e.g. $Tile_1$, $Tile_2$. To model the differences in the $E_{u,n}$ incident time $T_{u,n_{inc}}$ in all DaVinci Cores (e.g. $T_{inc_2} - T_{inc_1}$), we introduce δ_t, which can be measured in our benchmarks. It represents the average interval between the two nearest incident times among all DaVince Cores.

For each tile of $E_{u,n}$ among all DaVinci Cores, the workloads (data sizes) Ds_{single} are the same. Because of δ_t, the tile of $E_{u,n}$ on each core shows a stepped execution pattern with a start-up time. The execution period $P_{u,n}$ of $E_{u,n}$ for case (a) in Fig. 2.10 is calculated as below:

$$P_{u,n} = 2(N-1)\delta_t + P_{full} \tag{2.10}$$

where P_{full} is the period when all the cores are running concurrently. The execution period $P_{u,n}$ of $E_{u,n}$ for case (b) is calculated as below:

$$P_{u,n} = 2(N_{max} - 1)\delta_t \tag{2.11}$$

where N_{max} is the maximized all-busy core number for the event $E_{u,n}$. Then, to calculate $P_{u,n}$, we must determine P_{full} and N_{max} first. For the first category of the units, which relies on the independent on-core resources only, we have Eq. 2.12 for the case (a) in Fig. 2.10:

$$\begin{aligned} 2Th_{single}\delta_t \times \sum_{i=1}^{N-1} i + N \times Th_{single}P_{full} &= N \times Ds_{single} \\ \implies P_{full} &= \frac{Ds_{single} - (N-1)Th_{single}\delta_t}{Th_{single}} \end{aligned} \tag{2.12}$$

Fig. 2.10 shows that, the condition converting the case (a) to the case (b) is $P_{full} = 0$, which means that all DaVinci Cores cannot run concurrently and then the core number N is N_{max}. Then we have the relation shown below:

$$P_{full} = \frac{Ds_{single} - (N_{max} - 1)Th_{single}\delta_t}{Th_{single}} = 0 \implies N_{max} = \frac{Ds_{single}}{Th_{single}\delta_t} + 1 \tag{2.13}$$

We follow a similar idea for the second category of units as for the first category units. Hence, we list the equation below:

$$\begin{gathered} 2Th_{total}\delta_t \times (N-1) + Th_{total}P_{full} = N \times Ds_{single} \\ P_{full} = \frac{NDs_{single} - 2(N-1)Th_{total}\delta_t}{Th_{total}} \; N_{max} = \frac{2Th_{total}\delta_t}{2Th_{total}\delta_t - Ds_{single}} \end{gathered} \tag{2.14}$$

The Ascend 310 processors are the simplest DaVinci architecture processors and have two DaVinci Cores, and then we have $N = 2$. According to our

benchmark results, as shown in Table 2.1, $\delta_t \approx 61ns$, which is 2.5% of the kernel launch time and can be ignored. Hence, in the double-core execution, the total time can be simplified to $P_{u,n} = P_{full} = Ds_{single}/Th_{single}$ for the first category of units or $P_{u,n} = N \times Ds_{single}/Th_{total}$ for the second category.

2.4 Evaluation

This section evaluates the accuracy of Verrocchio. We first implement several basic kernels for the accuracy evaluations. Then as a complicated test case and a demonstration of the usage of Verrocchio, we show a matrix multiplication optimizing process guided by Verrocchio. All the experiment results are collected by Huawei's official profiler on the Ascend 310 processor (two DaVinci Cores) with Huawei Kunpeng 920 2.6GHz CPU equipped.

2.4.1 Sample Kernel Evaluation

Table 2.4 lists the sample kernels we implement for evaluations. The cases include memory transfers, single computations, and several basic applications. We evaluate the accuracy results for both single-core and double-core situations. Fig. 2.11 illustrates the evaluated error rate results for each case.

Generally, Verrocchio reports high accuracy results. For the single-core execution, the average error rate is 2.62%; for the double-core execution, the result is 2.30%. The error rates among all cases do not show a system error but are randomly scattered around the baseline. The single-core kernel *mte1B* reports the maximum error rate of 7.33%. On the other hand, for the double-core execution, *mte1B* reports a much lower error rate of 3.14%. Since the MTE1 Unit is the independent on-core unit we discussed in Sec. 2.3.3, which does not suffer the performance loss in the multi-core execution, *mte1B* is expected to show similar error rates in the two situations. Therefore, the practical results suggest that the MTE1 Unit is not stable as other independent on-core units, which may require deeper analyses. In addition, we observe that the error rates of the Application kernels and the Memory

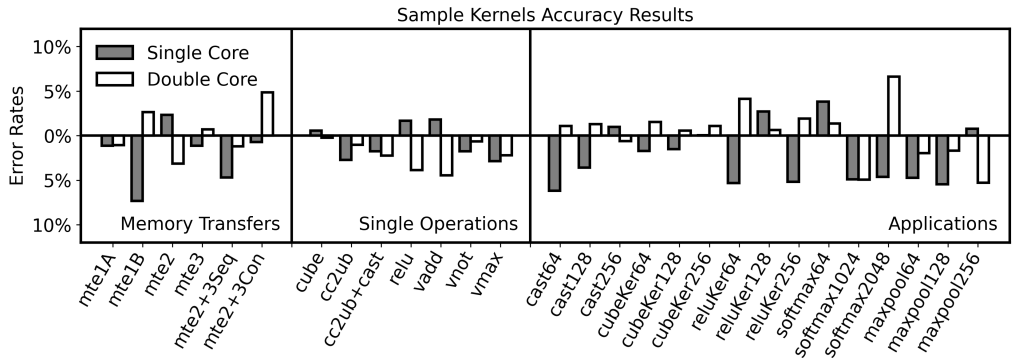


Figure 2.11: The Verrocchio error rate results for sample kernels

Transfer kernels are higher than the Single Operation kernels, which indicates that the memory transfers, especially between the on-core buffers and the external storage, are the main prediction error sources of Verrocchio.

2.4.2 Matrix Multiplication Optimizing Process

This section demonstrates an optimizing process of matrix multiplication with Verrocchio, one of the most significant applications for AI processors.

Improved Matrix Multiplication and Tiling Parameter Selection

Alg. 3 shows the target improved matrix multiplication algorithm for optimization. The matrix multiplications require four tiling parameters, $mTiles$, $kTiles$, $nTiles$, and $bNum$, which represents the tile number from the m, k, and n directions with the multiple control flow number.

The matrix multiplication adopts a policy called Double Buffer or Ping-Pong Buffer, shown in Alg. 3 line 3. DaVinci Core buffers allow reading and writing on different addresses at the same buffer concurrently. Meanwhile, the binary semaphore mechanism allows at most eight independent control flows. Hence, the Double Buffer policy splits the buffers and the operations in half and activates the second control flow. Fig. 2.12 illustrates how the Double Buffer policy fills the idleness gap of the MTE1 Unit to increase the

Algorithm 3: Target Improved Matrix Multiplication

input : matrix **A**, **B** of size $(m \times k)$, $(k \times n)$
tiling parameter *mTiles*, *kTiles*, *nTiles*, *bNum*

output: matrix **C** of size $(m \times n)$

```
1  (mSlice, kSlice, nSlice) ← (m, k, n) / (mTiles, kTiles, nTiles)
2  for i ← 0, j ← 0 to mTiles, nTiles do
3      for l ← 0, bf ← 0 to kTiles / bNum, bNum do
4          if j = 0 then
5              load A tile mSlice × kSlice to L1 from External ;
                // MTE2
6          load A tile mSlice × kSlice to L0A from L1 ;          // MTE1
7          if i = 0 then
8              load B tile kSlice × nSlice to L1 from External ; // MTE2
9          load B tile kSlice × nSlice to L0B from L1 ;          // MTE1
10         do matrix multiplication of A tile × B tile to L0C ; // Cube
11     load C tile mSlice × nSlice to UB from L0C ;            // Vector
12     load C tile mSlice × nSlice to External from UB ;       // MTE3
```

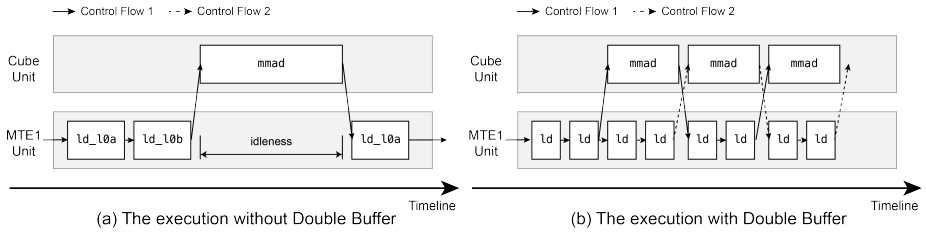


Figure 2.12: The effect of Double Buffer policy on the execution performance and concurrency

Table 2.4: Verrocchio sample kernels for accuracy evaluation

Categories	Kernel Name	Description
Memory transfers	mte1A	Transfer data from L1 Buffer to L0A Buffer
	mte1B	Transfer data from L1 Buffer to L0B Buffer
	mte2	Transfer data from external storage to L1 Buffer
	mte3	Transfer data from Unified Buffer to external storage
	mte2+3Seq	MTE2 and MTE3 Units execute sequentially
	mte2+3Con	MTE2 and MTE3 Units execute concurrently
Single computations	cube	Do a matrix multiplication w/ Cube Unit
	cc2ub	Transfer from L0C Buffer to Unified Buffer w/ Vector Unit
	cc2ub+cast	Same as above plus type conversion
	relu	ReLU activation function
	vadd	Vectorized addition
	vnot	Vectorized boolean NOT operation
	vmax	Vectorized element-wise Max operation
Applications (Full kernels with I/O)	castXxx	Type conversion operator
	cubeKerXxx	Matrix multiplication operator without tiling
	reluKerXxx	ReLU activation operator
	softmaxXxx	Softmax operator
	maxpoolXxx	MaxPooling operator

concurrency and ILP. However, the Double Buffer policy sometimes introduces too much cost and lowers the performance because of the imbalanced execution period.

To formulate an optimization problem, we conduct the tiling parameter selection procedure as a permutation of all four parameters, which are the variables of the problem. The improved matrix multiplication has no further requirements or constraints for the variables, but the tiling numbers must be less than the $16 \times 16 \times 16$ block numbers along the same direction. For each parameter combination in the feasible domain, Verrocchio predicts the execution time and records, which performs the objective function of the optimization problem. To solve the problem, we implement the exhaustive search, the most straightforward solution. Finally, we find the optimized

solution as the parameter combination with the least execution time for our matrix multiplication.

Verrocchio Accuracy Results

To evaluate the Verrocchio accuracy, we compare our predicted results from Verrocchio with the practical execution time results, which adopt the optimized tiling parameters, as shown in Fig. 2.13. For a better visibility, we define the matrix multiplication scale as the number of the basic blocks operated by the Cube Unit ($m \times k \times n/16^3$). As some matrix multiplications can have the same matrix multiplication scale, we use the average values as the results.

Fig. 2.13 (a) shows the total execution time error rates. The average error rate among all test cases is 5.06% for the single-core execution and 5.25% for the double-core execution. The predicted results are lower than the experiment results when the scale is small. One of the potential reasons is that we do not consider the Scalar Unit and the corresponding instructions in Verrocchio. As we discussed in Sec. 2.2.2, the Scalar instructions could block the execution of other units. When the scale gets larger, the effect of the Scalar instructions declines apparently, and the error rates rise above the baseline. When the scale is larger than 5000, the error rates get stable at around 5%, meaning the prediction of Verrocchio is more accurate and stable in large-scale cases.

We also show their prediction error rate for each hardware unit separately in the following figures of Fig. 2.13. The Vector Unit and the Cube Unit are well-predicted, reporting the average error rates of 0.69% and 1.17% for the single-core execution, 0.51% and 0.85% for the double-core execution, shown in Fig. 2.13 (a) and (b) respectively. The MTE1 Unit performs slightly worse with the average error rates of 9.78% and 9.46%. We observe apparent error rates when the scale is small. One of the potential reasons is that the initialization time of the MTE1 instruction is less than other instructions. The MTE2 and MTE3 Units report the worst average error rates of 18.89%

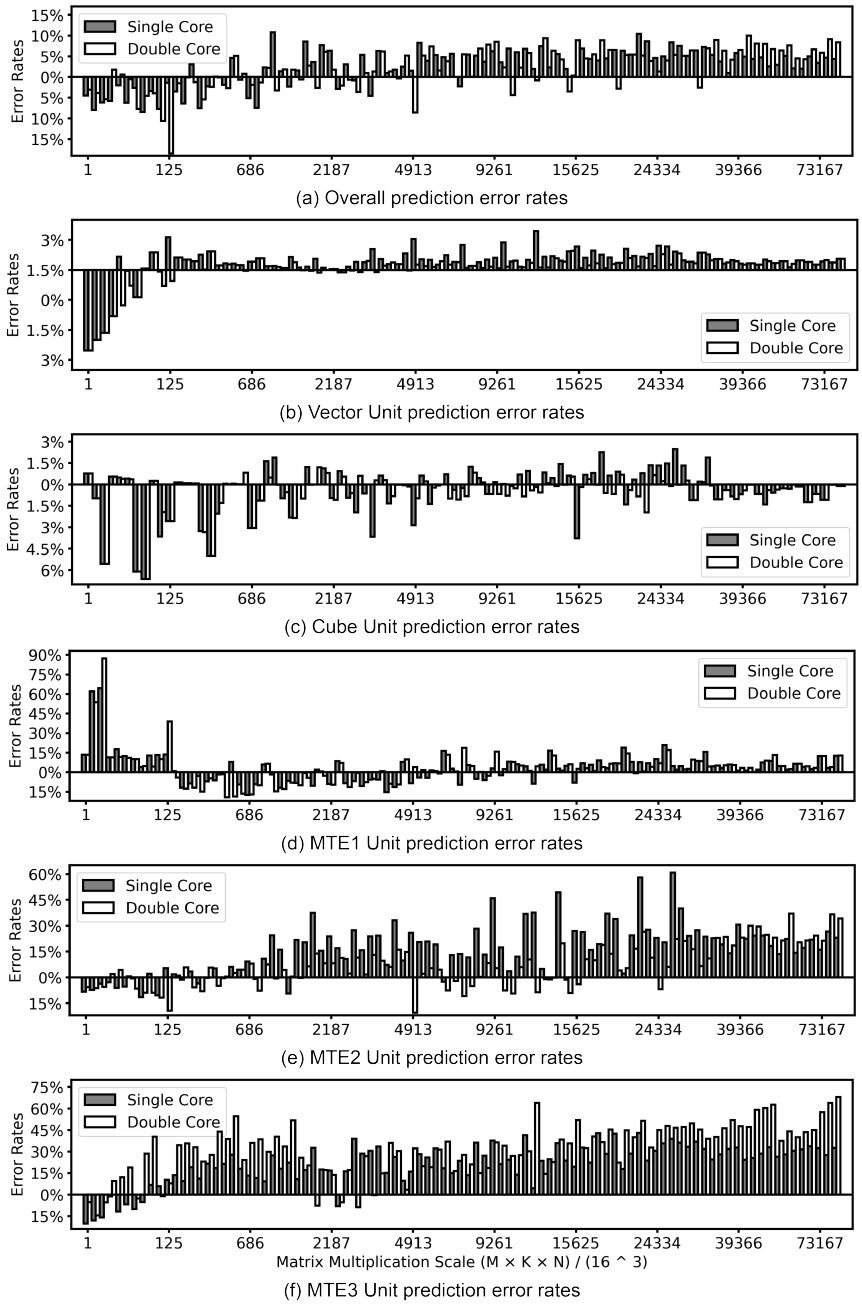


Figure 2.13: The Verrocchio error rate results for the improved matrix multiplications

and 23.72% for the single-core execution, 14.22% and 33.72% for the double-core execution. Furthermore, the two MTE Units report a system error when the scale is large, where the predicted execution time is always larger than the practical experiment time, especially the MTE3 Unit results. One of the reasons is that, as shown in Fig. 2.6, the bus contentions of the MTE Units do not start immediately but wait for the finish of a block. Verrocchio considers all contentions starting at once when the instructions are called, which brings more contentions than the practical execution. In addition, we notice the two MTE Units perform worse in the double-core execution. According to the data collected in Table 2.1, the two DaVinci Cores of the Ascend 310 processors are not synchronized perfectly. Therefore, Verrocchio could bring false contentions in the double-core execution when two respective MTE Units on two cores do not fully overlap.

Matrix Multiplications Acceleration Results

Operator Level We compare the improved matrix multiplications with the Huawei CANN operators. As the CANN operators require different logical core numbers, we keep the active core numbers as their respective demands. Our improved matrix multiplication always uses two cores, so we set the core number to two during the experiments of our algorithm. We test matrix multiplications in 12 shapes, from $W \times W \times 2W$ to $W \times W \times 6W$, as shown in Fig. 2.14.

The results show how Verrocchio helps the matrix multiplications to achieve a significant speedup. The improved matrix multiplications achieve an average speedup of 1.70× among all test cases compared with the Huawei official CANN operators. Generally, the accelerations grow steadily with the increment of the matrix multiplication scale (or value W). For the regular-shaped matrix multiplications like $W \times W \times 2W$, our matrix multiplications show similar performance compared with the CANN operators, where the two approaches follow similar tiling strategies. However, our matrix multiplications perform much better for those irregular-shaped matrix multiplications

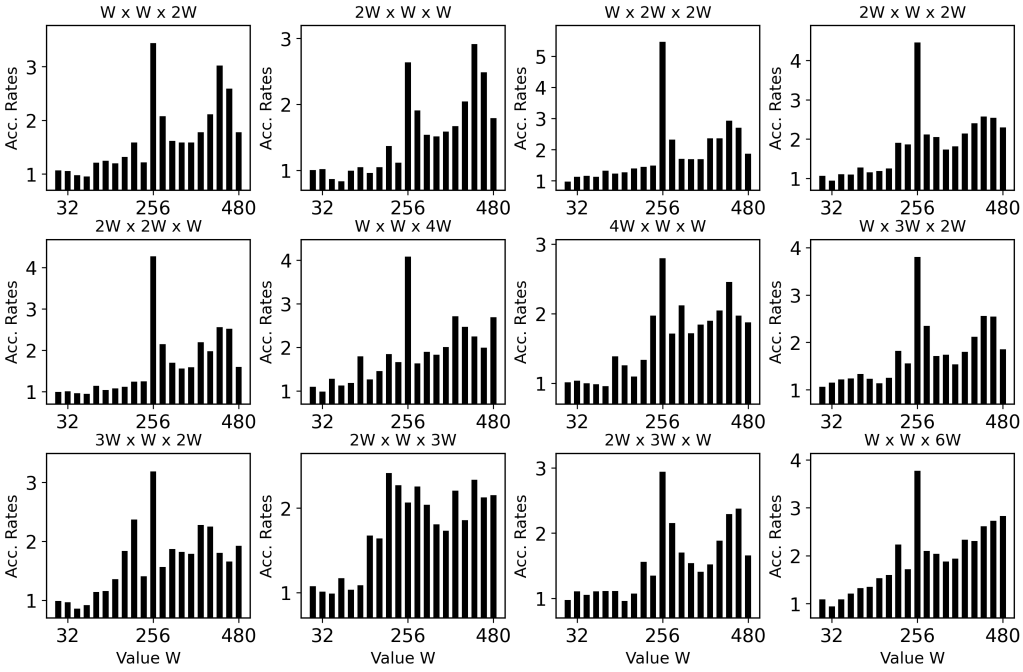


Figure 2.14: Accelerations of the improved matrix multiplication compared with the CANN operators

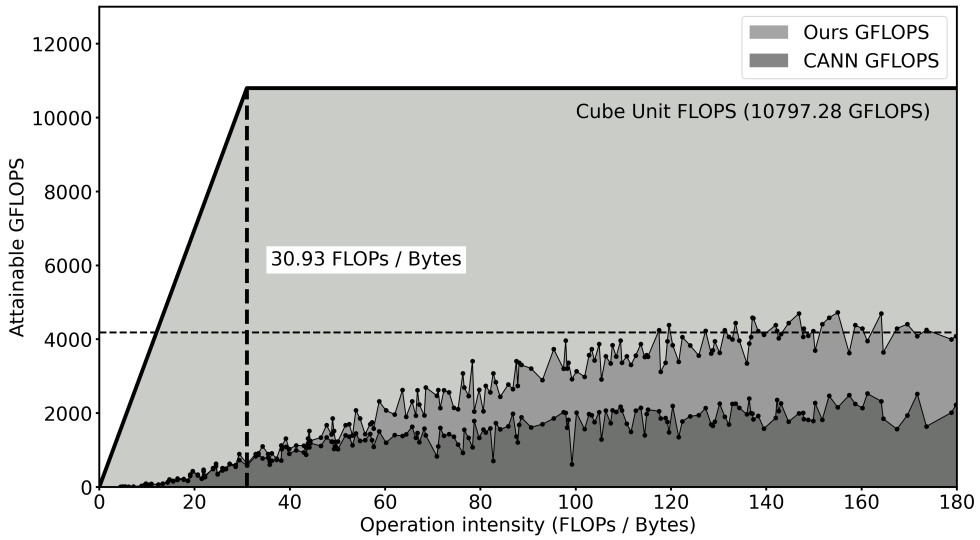


Figure 2.15: The roofline model with the matrix multiplication results annotated

like $W \times W \times 4W$. The main reason is that the CANN operators abuse the multiple logical cores with inefficient tilings in these irregular-shaped cases, even when the Ascend 310 processors have only two physical cores. Meanwhile, our improved matrix multiplications focus on the two physical DaVinci Cores with the most appropriate and efficient tilings suggested by Verrocchio, which make great improvements.

Furthermore, we plot our matrix multiplications in the roofline model [64], as shown in Fig. 2.15, which considers the Cube Unit FLOPS and the bandwidth between the L0 and L1 Buffer. Compared with the CANN operators, our matrix multiplication algorithm reports a smaller performance gap to the attainable roofline results. The average ratio of the peak hardware performance we achieve is 23.64%, 2.09X higher than that of the CANN operators. With the increment of the operation intensity, the ratio of our algorithm becomes higher than that of the CANN operators, and finally stops at about 38.78%, which further proves the better performance in larger scales we observed in Fig. 2.14.

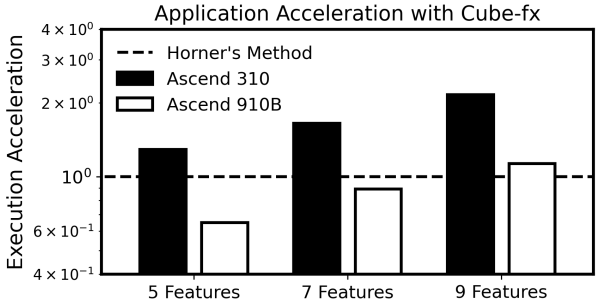


Figure 2.16: The estimation results of the popular DNN applications

Application level Currently, executing a DNN model on Ascend 310 processor needs an official ATC offered by Huawei, which converts the Caffe or TensorFlow model to the Huawei custom model. However, Huawei does not offer the structures and formats of the custom model. Hence, we propose an estimated method to evaluate the acceleration brought by the improved matrix multiplications for popular DNN applications. For each layer of a DNN application, we evaluate the total execution time and the Cube Unit execution time. We compute the matrix multiplication scales defined in the last subsection with the collected Cube Unit execution time. Therefore, with the average acceleration rates of the scale, we have an approximate acceleration result of this matrix multiplication operator. By estimating all operators, we have final results of the DNN application [65–69], as shown in Fig. 2.16.

Among all evaluation results, AlexNet shows the most significant speedup of 2.18×. The reason is that AlexNet is one of the earliest Convolutional Neural Networks (CNNs) with large Fully Connected (FC) layers. It requires matrix multiplications of a large scale. As we show in the last subsection, the CANN operators perform worse than our operators on a large scale. Generally, the DNN applications report an average speedup of 1.53×. Although YOLO V4 is a complex model, the matrix multiplication scales are not large, reporting the lowest acceleration rate of 1.15×.

2.5 Conclusion

For the first time, we focus on building a performance model, Verrocchio, for Huawei DaVinci AI Cores. The DaVinci Core has complex data paths and distinct working details, which significantly influences the performance and brings a series of modeling challenges. Our discrete-event-based model reports high accuracy for various kernel programs but involves some system errors of the MTE Unit bandwidths, which are caused by the simplification of their runtime behaviors.

Chapter 3

SelB-k-NN: Replacing the Inefficient Operations

After giving deep investigation and benchmarks in Chapter 2, we indentify the two key challenges of the algorithm optimization on the Ascend processors. As discussed in Chapter 1, we propose two optimization approaches for the challenges. In this chapter, we discuss the first optimization approach with the example of SelB-k-NN. Sec 3.1 discusses the background and motivation of the algorithm. Sec 3.2 introduces the central part of the single-core SelB-k-NN algorithm. Sec 3.3 proposes two significant methods to minimize the hardware support requirements of the algorithm. In Sec 3.4, we discuss how we can quantify and achieve the optimized tiling shapes for large datasets. Sec 3.5 evaluates the performance of SelB-k-NN, and Sec 3.6 concludes this chatper.

3.1 Background & Motivation

3.1.1 *K*-Nearest Neighbors Algorithm

K-NN is one of the most classical and well-studied algorithms for classification problems. The general idea of k-NN is uncomplicated: those points in

the same category mostly have high similarity and are located close to one another in high-dimensional space. To classify a test point, k-NN computes the similarity between the test point and all the training points and then selects the k nearest neighbors in the k-selection phase. K-NN finally chooses the category of the most neighbors in the k nearest neighbor results as the test point's category.

In the early days, the distance computation took most of the k-NN's execution time. With the improvement of the hardware, GPUs with massive parallelism successfully mitigate the bottleneck. The k-selection phase remains a new bottleneck with several solutions. In general, the k-selection algorithms are divided into two categories, heap-based [54,70–74] and bitonic-based [54,75,76]. However, the two approaches rely on a considerable number of weakly supported operations on the AI processors. For 16-selection of 16 test points with 4096 training points on the Huawei Ascend 310 processors, the dynamic addressing during sibling selections in the re-heap procedure occupies 94.10% of the heap-based k-selection execution time. The vectorized comparisons & selections, which support compare-and-swap operations, occupy 75.98% of the bitonic-based k-selection. A novel algorithm to reduce these operations on the AI processors is urgently needed.

3.1.2 Tiling Shape Mismatch of *K*-NN

Tiling is an effective technique for matrix multiplication optimization, which benefits from the data locality and is widely studied on the different platforms including the AI processors [77–82]. It combines sufficient knowledge of the hardware structures and the fixed specific matrix multiplication execution patterns for maximized hardware performance.

However, most of their works separate the matrix multiplications from the applications. While the matrix multiplications prefer regular tiling shapes ($m \approx n$) on the Matrix MACs [80], k-selections prefer irregular ones ($m \gg n$) on the vector units, which brings a tiling shape mismatch. For example, the heap-based k-selection is expanded efficiently along the m-direction, main-

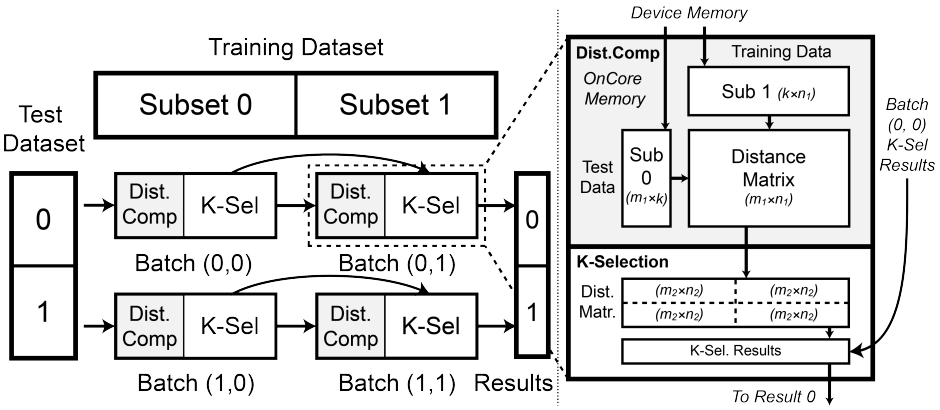


Figure 3.1: An overview of SelB-k-NN with a detailed view of the batch (0, 1)

taining m heaps on the vector units for parallel element-wise comparison. Increasing m would not prolong the execution time seriously, while increasing n raises the loop iterations and makes the execution longer. However, the matrix multiplication output restricts the k-selection's tiling shape. When selecting a regular tiling shape for the matrix multiplication, the relation $m \gg n$ cannot be true for the k-selection. On other platforms, selecting the regular tiling shapes for k-NN may perform well. On AI processors, the matrix multiplication does not dominate the k-NN execution time. Directly applying matrix multiplication's tiling shapes works poorly on the k-selection and harms the k-NN performance.

3.2 SelB-k-NN Algorithm

Fig. 3.1 illustrates an overview of SelB-k-NN. It first tiles the test and training datasets along the m-direction and n-direction respectively. A batch is assigned a tile of the workload with the distance computation tiling shape (m_1, n_1). It receives data from the datasets, computes and updates the previous min-k results from the last adjacent batch along the n-direction. The k-selection of each batch is further tiled with the tiling shape (m_2, n_2), as

shown on the right side of Fig. 3.1. The processed data of each batch is transferred from the large but slow device memory, while the intermediate min-k results can be maintained constantly at the fast but tiny on-core memory until the finish of the current test dataset tile along the m-direction. The mini-batches are executed sequentially on a single processor and easily applied to multi-processors [83]. This section mainly describes a single batch of SelB-k-NN.

3.2.1 Distance Computation on AI Processors

By accelerating the matrix multiplications with the Matrix MACs, the AI processors efficiently improve the performance of cosine distance computations.

The cosine distance of points $\boldsymbol{p}_1, \boldsymbol{p}_2$ is:

$$D_{cos} = 1 - \frac{\boldsymbol{p}_1 \cdot \boldsymbol{p}_2}{\|\boldsymbol{p}_1\| \, \|\boldsymbol{p}_2\|} = 1 - \boldsymbol{p}_1 \cdot \boldsymbol{p}_2 \text{ (for } \ell 2\text{-norm pts.)} \tag{3.1}$$

Therefore, the essence of the distance computation is the dot product of two vectors. For $m_1 \times n_1$ data, we compute the dot products following the matrix multiplication below:

$$\begin{bmatrix} \boldsymbol{p}_1 \\ \vdots \\ \boldsymbol{p}_{m_1} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{q}_1^T & \cdots & \boldsymbol{q}_{n_1}^T \end{bmatrix} = \begin{bmatrix} \boldsymbol{p}_1 \cdot \boldsymbol{q}_1 & \cdots & \boldsymbol{p}_1 \cdot \boldsymbol{q}_{n_1} \\ \vdots & \ddots & \vdots \\ \boldsymbol{p}_{m_1} \cdot \boldsymbol{q}_1 & \cdots & \boldsymbol{p}_{m_1} \cdot \boldsymbol{q}_{n_1} \end{bmatrix} \tag{3.2}$$

where $\boldsymbol{p}_i$ and $\boldsymbol{q}_i$ are d dimensional vectors. For simplicity, we assume that all the data in this work are $\ell 2$-normalized, and the term *distance* in the chapter is cosine distance.

3.2.2 *K*-Selection on AI Processors

SelB-k-NN k-selection algorithm relies on two groups of common vectorized operations, which are used in the most significant operations in DNN ap-

plications and supported by AI processors [14, 84, 85]. The first includes `reduceMin` (and `reduceMax`), which find the minimized value in a given vector. Some AI processors further support returning the corresponding indices [14]. The two operations form the central part of the pooling layer, frequently used after convolutional layers. The other includes `vecCmp` and `vecSel`, which compare two given vectors by elements to build a mask and select the elements based on the mask. The two operations support an elementary branch for data, which is necessary for training like the backpropagation computation of `ReLU`. However, as discussed, existing works [14,16,85] indicate inefficiency or lack of the two operations on some AI processors.

Our k-selection algorithm finds indices of the min-k results for m_2 separate n_2-sized vectors in at most k steps. For each vector, the general idea is to ensure that after the k-selection phase, the minimum value remaining in the vector is greater than the maximum value in the min-k results. Based on the idea, we propose the k-selection algorithm in Alg. 4.

The main procedure of the algorithm is a k-step loop. Fig. 3.2 illustrates one *step* example of the batch $(0, 1)$ of the k-selection with the tiling shape of $m_2 = 3, n_2 = 10$, which process the data range of $m \in [0, 3), n \in [10, 20)$. The algorithm first find the minimum value res_{min} with its index idx_{min} in the target vector $\mathbf{D}[ln]$. Then for m target vectors, we compare whether res_{min} is less than the first element of the min-k results $\mathbf{K}[0]$ in parallel. With the compared results, we update $\mathbf{K}[0]$ and $\mathbf{I}[0]$ accordingly. For the example in Fig. 3.2, $\mathbf{K}[0][0]$ and $\mathbf{K}[0][1]$ are updated since $-0.2 < 0.2$ and $-0.3 < 0.4$. The next step is the most significant step, where we maintain the min-k results $\mathbf{K}$ to ensure the first element is always the maximum element with the bitonic 1-selection. When $k = 1$, the bitonic k-selection is degenerated to a reduction but not only return the reduced value as `reduceMax` does. Instead, it rearranges the vector and puts the top-1 value at the first element. Compared with directly invoking `reduceMax`, the bitonic 1-selection avoids scalar dynamic addressing that locates the reduction results and variable assignments that update values for each target vector respectively. Instead, it calls

Algorithm 4: SelB-k-NN K-Selection Algorithm

Input : distance matrix $\mathbf{D}$ of size $(m_2 \times n_2)$
last min-k results $\mathbf{K}$ of size $(k \times m_2)$
last indices of results $\mathbf{I}$ of size $(k \times m_2)$

Output: min-k results $\mathbf{K}'$ of size $(k \times m_2)$
indices of results $\mathbf{I}'$ of size $(k \times m_2)$

```
1  mask ← 0⃗
2  for step ← 0 to k do
3      for ln ← 0 to m do
4          if mask[ln] ≠ 0 then
5              continue
6          (res_min, idx_min)[ln] ← reduceMin(D[ln])
7          D[ln][idx_min[ln]] ← +∞ ;                          // Scalar
8      cmp ← vecCmpLt(res_min, K[0]) ;                        // Cmp
9      mask ← vecSel(cmp, mask, 1⃗) ;                          // Sel
10     mask_min ← reduceMin(mask)
11     if mask_min ≠ 0 then
12         return K as K', I as I'
13     K[0] ← vecSel(cmp, res_min, K[0]) ;                    // Sel
14     I[0] ← vecSel(cmp, idx_min, I[0]) ;                    // Sel
15     bitonicKSel(k = 1, (K, I)) ;                  // O(k) Cmp&Sel
16 return K as K', I as I'
```

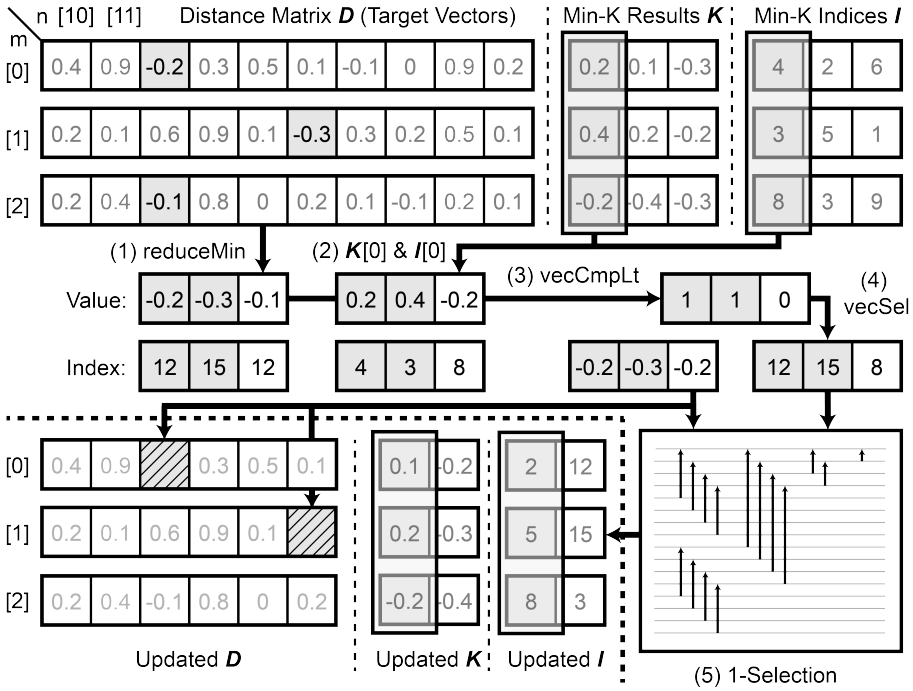


Figure 3.2: One step of the batch $(0, 1)$ of the k-selection ($m_2 = 3, n_2 = 10$)

$O(k)$ vectorized comparisons & selections. On the other hand, the approach should not replace `reduceMin` for the n-sized $\mathbf{D}$, which would bring $O(n)$ vectorized comparisons & selections as the direct bitonic-based approach does for the entire k-selection. With the compared results, we also update value res_{min} in the target vector with a large constant.

For each test point, the time complexity of SelB-k-NN k-selection algorithm is $O(k + k^2 + nk)$, which consists of three parts, the scalar operation part, the vectorized comparison & selection part, and other well-supported vectorized operations part. Although the overall time complexity seems to be higher than the heap-based k-selection of $O(nlogk)$ and the bitonic-based k-selection of $O(nlog^2k)$, our algorithm significantly reduces the scalar operations from $O(nlogk)$ to $O(k)$, and the vectorized comparisons & selections from $O(nlog^2k)$ to $O(k^2)$, as annotated in Alg. 4. SelB-k-NN k-selection algorithm prevents the less powerful operations of AI processors from growing with the scale of training dataset n. Existing works [86, 87] report that in k-NN algorithm, a larger parameter k does not contribute to higher accuracy. Therefore, we can safely assume that $n \gg k$, where our algorithm restrains the number of the scalar operations and vectorized comparisons & selections for higher performance with acceptable accuracy.

The second feature of the algorithm is an early exit policy. For m_2 test points, we initialize an m_2-length vector $mask$ to '0's. The vector is updated each step based on the parallel comparison results by selecting the elements between $mask$ itself with '1's, as shown in line 10. All the values of those target vectors, whose minimum values are greater than the maximum values in their corresponding min-k results, are greater than all the values in the min-k results, as Eq. 3.3 shows.

$$\begin{aligned} &\forall x \in \mathbf{D}, res_{min} \leq x;\ \forall y \in \mathbf{K}, res_{max} \geq y \\ &\quad res_{min} > res_{max} \implies \forall x > \forall y \end{aligned} \tag{3.3}$$

Therefore, once $res_{min} > res_{max}$, all the remaining values in the target vector are not the candidates of the min-k results and can be rejected directly.

Then the related element of $mask$ is updated to '1', and the algorithm skips the corresponding target vector, as shown in line 4. In the example of Fig. 3, the $\mathbf{D}[2]$ is rejected in this step since $-0.1 > -0.2$. When the minimum value of $mask$ is not '0', which means all the elements are updated to '1', the algorithm ends in advance, as shown in line 11. For all batches after tiling the test dataset along m-direction and the training dataset along n-direction, each batch of SelB-k-NN additionally receives the min-k results from the last adjacent tile along n-direction, as shown in **Input K** of Alg. 4. The early exit policy can be activated by the last results and reduces most workloads during the execution. We will discuss the details of the workload reduction later in Sec. 3.4.

3.3 Minimizing Hardware Support Requirements

To minimize the hardware support as much as possible for portability of SelB-k-NN, this section proposes two algorithms to replace the uncommon functions in DNNs with the most common functions on the AI processors [14, 84, 85].

3.3.1 ReLU-Based Operations for VecCmp and VecSel

For the AI processors without `vecCmp` and `vecSel` hardware supports, we propose a novel `ReLU`-based bitwise algorithm specifically for the AI processors. We do not involve bit-shifting or integer division to avoid extra requirements for hardware components on the AI processors.

We show the algorithm in Alg. 5 for `vecCmpLt` and `vecSel`, which generates a mask based on the smaller elements of the two vectors and then selects values from the other two target vectors with the mask. For any vector $\mathbf{A}$ and $\mathbf{B}$, Alg. 5 first computes the the vectorized subtraction results sub_1.

Algorithm 5: Bitwise Ops for VecCmpLt and VecSel

Input : compared vector **A, B** of FP16 numbers
target vector **X, Y** of 16-bits numbers

Output: result vector **R** from **X** and **Y**, based on the element-wise minimum between **A** and **B**

1 ## **VecCmpLt:** Computing *mask* from **A, B**
2 $\text{sub}_1 \leftarrow \mathbf{A} - \mathbf{B}$
3 $\text{sub}_2 \leftarrow (\text{sub}_1 \;\&\; \mathbf{0x8\vec{0}00}) \mid \mathbf{0x0\vec{0}01}$
4 $\text{mask}_1 \leftarrow \mathbf{ReLU}(\text{sub}_2)$
5 $\text{mask}_2 \leftarrow$ reinterpret mask_1 as INT16 vector
6 $\text{mask} \leftarrow \text{mask}_2 - \vec{\mathbf{1}}$
7 ## **VecSel:** Computing **R** with *mask* from **X, Y**
8 $\mathbf{R} \leftarrow \mathbf{X}$ & mask
9 $\mathbf{R} \leftarrow \mathbf{R} \mid (\mathbf{Y} \;\&\; \sim \text{mask})$

Then it masks out the exponent and mantissa part (last 15 bits) of the FP16 number sub_1 and then sets the last bit to '1' by a bitwise inclusive OR operation. For the element where $A < B$, the sub_2 is 0x8001, while for $A > B$, the sub_2 is 0x0001. The essence and the most novel part of the algorithm is line 3, where it computes the comparison mask $mask_1$ with `ReLU`, which is one of the most representative activation functions. The binary form of `ReLU` function is shown below:

$$\mathbf{ReLU}(t) = \begin{cases} 0\text{b0xxx} & t = 0\text{b0xxx } (t \geq 0) \\ 0\text{b0000} & t = 0\text{b1xxx } (t \leq 0) \end{cases} \tag{3.4}$$

where 'x' is any bit ('0' or '1'). For both floating-point and integer numbers, whose most significant bit (the 1st bit) is the sign bit, `ReLU` checks the sign bit and returns the result values according to its formula. Therefore, after line 3, the $mask_1$ for $A < B$ is set to 0x0000, while for $A > B$ it is set to 0x0001. Alg. 5 reinterprets $mask_1$ as INT16 number, which requires

no instructions but only indicates that the following operation is integer operations. Then Alg. 5 subtracts INT16 number 1 from the result following the integer arithmetic rules. After that, the final comparison mask *mask* for $A < B$ becomes 0xffff (-1 in INT16), while for $A > B$, it becomes 0x0000. To compute the mask of $A > B$ (vecCmpGt), swap the input **A** and **B** and the rest part remains the same. For the mask of $A = B$ (vecCmpEq), the sub_1 is modified to be computed by the formula below with other parts unchanged:

$$sub_1 \leftarrow (\mathbf{A} - \mathbf{B}) \mid (\mathbf{B} - \mathbf{A}) \tag{3.5}$$

With the comparison masks, Alg. 5 then selects the values from the target vector **X** and **Y**. Line 6 masks the values in **X** vector where $A > B$ with the corresponding mask of 0x0000, and keeps values where $A < B$ with the mask of 0xffff. Line 7 finishes the symmetric jobs and combines the two results with a bitwise inclusive OR operation.

The replaced `vecCmpLt` and `vecSel` involves only several efficient vectorized operations on the AI processors, including `ReLU`. `ReLU` is innately designed to contain a branch operation but wrapped by its original algorithm. Our algorithm widens `ReLU` with its branch operation to general purposes. The algorithm contains no scalar operations and can achieve high performance on the AI processors.

3.3.2 Indexing with Vectorized Operations

While `reduceMin` (`reduceMax`) is irreplaceable in the deep learning applications, returning the corresponding indices is not necessary to be supported on the AI processors. To migrate the algorithm to the AI processors without vectorized hardware indexing supports, we propose an efficient algorithm for indexing on those AI processors shown as Alg. 6. The algorithm has one more vectorized operations than the k-selection algorithm, `vecDup`, which

Algorithm 6: Indexing w/ vectorized operations

Input : indexing vector **K** of size (n)
indexing target value V

Output: index of value V in vector **K**

1 idxMap $\leftarrow [0, 1, 2, \ldots, (\mathrm{n}-1)]$
2 vArray $\leftarrow [\mathrm{V}, \mathrm{V}, \mathrm{V}, \ldots, \mathrm{V}]$
3 cmp $\leftarrow$ **vecCmpEq**(vArray, **K**)
4 idxArray $\leftarrow$ **vecSel**(cmp, idxMap, $\vec{\textbf{0xfc00}}$)
5 V $\leftarrow$ **reduceMax**(idxArray)

assigns a value to the whole vector and is also commonly used in the initialization of each layer.

The algorithm first assigns the target value V to an n-sized array $vArray$ with `vecDup`. Then it compares the $vArray$ with the indexing vector **K** to build a bitmask with `vecCmpEq` for the position where V locates. With the comparison mask cmp, we select the indices data between a prepared indices map $idxMap$ and an array full of 0xfc00 to build a result array $idxArray$. With `reduceMax`, the algorithm finds the result index of indexing target value V. Since the indexing algorithm has no scalar operations, it would not be the bottleneck of the whole k-selection algorithm.

One tricky part is the value 0xfc00. Most AI processors [9, 13, 88, 89] adopt FP16 as the computation data type to save the storage and improve the performance. Therefore, the vectorized operations usually fully support FP16 data on these AI processors but support INT16 data partially, e.g., subtraction we used in Sec. 3.3.1. For `vecCmpEq` operations without INT16 data supports, since it does not rely on the numerical value of the data, we can directly cast the INT16 indices data to FP16 with no bit modifications. However, for `reduceMax`, we need to ensure that the final index we find is numerically greater than others in FP16 format. Therefore, we select the value 0xfc00 as the initialized value, which is the negative infinity of FP16

and is less than all other 16-bit data in FP16 format. In this way, when we still cast our INT16 indices data to FP16 without any format conversions, `reduceMax` still works no matter how the bits represent in FP16 format. One potential issue is that when multiple target values V exist in the vector, we cannot ensure which index will be returned. However, it makes no trouble in our k-selection algorithm. In addition, on the AI processors that support INT16 vectorized operations, the value 0xfc00 still works well. The value 0xfc00 is negative in INT16 format (-1023) and less than all possible indices.

3.4 Optimal Tiling Shape Search

As illustrated in Fig. 3.1, the parameters (m_1, n_1) and (m_2, n_2) determine the tiling (or batch) shapes of the distance computation and the k-selection, which significantly influences the entire performance. This section discusses how we find the optimial tiling shapes with an optimization problem.

3.4.1 *K*-selection Workload Quantification

Along the n-direction, while each tile of the matrix multiplications has stable and equal workloads, the workload of each tile in SelB-k-NN k-selection varies based on its assigned datasets and last min-k results. During the runtime of the algorithm, the early exit policy of the k-selection is continually activated by the last adjacent tile's results. It dynamically rejects the for-loop steps of the k-selection as shown in Alg. 4, identified as the primary workloads. Intuitively, the later tiles always have fewer workloads. To accurately model the algorithm performance to an optimization problem, we present an analysis to quantify the k-selection workload of the tile t.

Let us assume that for a specific test data m, the distance computation results of all training data D obey some random distribution $\mathcal{F}(X)$. Therefore, after the computation of tile t during the execution, the distance computation results that have been computed $D_{1\to t}$ and are computed just at the computation of tile t, D_t, also obey the distribution $\mathcal{F}(X)$.

$$D \sim \mathcal{F}(X) \implies D_{1\to t}, D_t \sim \mathcal{F}(X) \tag{3.6}$$

Then let us consider that after the tile t, $x'_{1\to t}$ is the k-th minimum value of $D_{1\to t}$, which is also the maximum value of the k-min results. We can compute the cumulative distribution function (CDF) of the distribution $\mathcal{F}_X(x'_{1\to t})$ as follows:

$$\mathcal{F}_X(x'_{1\to t}) = P(X \leq x'_{1\to t}) = \frac{k}{nt} \tag{3.7}$$

where n is the number of distance results of each tile (also the number of k-selection inputs). The CDF computes the probability of randomly selecting a value from $D_{1\to t}$, which is one of the k-min results. Let us call that for the tile t, the values, which belong to the distance results D_t and are going to replace another value in the current k-min results, are *updating* values. Then for the tile t, when one value x is updating at this stage, it means that $x \leq x'_{1\to t}$. Therefore, for the tile t of size n, the expected number of the updating value can be computed by the CDF as follows:

$$E_t(X \leq x'_{1\to t}) = n\mathcal{F}_X(x'_{1\to t}) = \frac{k}{t} \tag{3.8}$$

Since each for-loop *step* of SelB-k-NN k-selection in Alg. 4 updates one value, the expected *updating* value is also the expected *step* number for tile t. Hence, for tile t, the workload of the k-selection phase is quantified to k/t theoretically.

3.4.2 Execution Time Optimization Problem Formulation

For the entire test dataset of $(M \times d)$ and the training dataset of $(d \times N)$, let us consider the tiling shape (m_1, n_1) for the distance computation phase and

(m_2, n_2) for the k-selection phase. As we discussed in Sec. 3.1.2, since the tiling shapes of k-selection are restricted by those of distance computation, we have $m_2 \leq m_1, n_2 \leq n_1$. Our goal is to minimize the overall execution time by a space exploration of the four tiling shapes (m_1, n_1) and (m_2, n_2). Formally, we formulate the following optimization problem.

$$
\begin{aligned}
min \quad & T_{mm} + T_{kSel} \\
s.t. \quad & T_{mm} = \frac{MN}{m_1 n_1} f(m_1, n_1) \\
& T_{kSel} = \frac{M}{m_2} \sum_{t=1}^{\frac{N}{n_2}} (\frac{k}{t} g(m_2, n_2) + g_0(m_2, n_2)) \\
& d(m_1 + n_1) + m_2(n_2 + k) < Size_{mem} \\
& m_2 \leq m_1, n_2 \leq n_1 \qquad (*)
\end{aligned}
\tag{3.9}
$$

where $f(m_1, n_1)$ is the profiled execution time of the matrix multiplication with the tiling shape (m_1, n_1), $g(m_2, n_2)$ is the execution time of a *step* in k-selection (line 3 to 15 in Alg. 4) with the tiling shape (m_2, n_2) with constant cost $g_0(m_2, n_2)$, $Size_{mem}$ is the size the on-core memory.

3.4.3 Search Space Pruning

The search space size of the optimization problem in Eq. 3.9 is $O(m^2 n^2)$, which can be very large and time-consuming to search exhaustively. This subsection adds one more constraint to prune its search space to $O(m^2 n)$.

For the optimized solutions, the search procedure contains a four-layer loop to search the parameters. It firsts select a set of (m_1, n_1) in the outer loops, and then traverse all (m_2, n_2) in the inner loops with the constraint $(*)$. At the innermost loop for n_2 search, while (m_1, n_1, m_2) are selected, let we consider $pn_2 = n_1$, where $p \geq 1$. Then we have:

$$T_{kSel}(m_2, n_2) = \frac{M}{m_2} \sum_{t=1}^{\frac{pN}{n_1}} (\frac{k}{t} g(m_2, \frac{n_1}{p}) + g_0(m_2, \frac{n_1}{p})) \tag{3.10}$$

If $T_{kSel}(m_2, n_2) > T_{kSel}(m_2, n_1)$ always holds, it means that for any $p > 1$ (or for any $n_2 < n_1$), the execution time is longer compared with the exeuction time when $n_2 = n_1$. Therefore, let us consider:

$$\begin{gathered} T_{kSel}(m_2, n_2) > T_{kSel}(m_2, n_1) \\ \sum_{t=1}^{\frac{pN}{n_1}} (\frac{k}{t} g(\frac{n_1}{p}) + g_0(\frac{n_1}{p})) > \sum_{t=1}^{\frac{N}{n_1}} (\frac{k}{t} g(n_1) + g_0(n_1)) \end{gathered} \tag{3.11}$$

where we simplify $g(m_2, n)$ to $g(n)$. Then we have:

$$\frac{N}{n_1}(p g_0(\frac{n_1}{p}) - g_0(n_1)) + k(\sum_{t=1}^{\frac{pN}{n_1}} \frac{g(\frac{n_1}{p})}{t} - \sum_{t=1}^{\frac{N}{n_1}} \frac{g(n_1)}{t}) > 0 \tag{3.12}$$

Notice that:

$$k(\sum_{t=1}^{\frac{pN}{n_1}} \frac{g(\frac{n_1}{p})}{t} - \sum_{t=1}^{\frac{N}{n_1}} \frac{g(n_1)}{t}) > -k \sum_{t=1}^{\frac{N}{n_1}} \frac{g(n_1)}{t} > -\frac{kN}{n_1} g(n_1) \tag{3.13}$$

Then Eq. 3.12 is converted to:

$$p g_0(\frac{n_1}{p}) - (g_0(n_1) + k g(n_1)) > 0 \tag{3.14}$$

Since the inequality above does not involve input N, after profiling the execution time (e.g., $g(n_1)$) on the AI processors, the search space can be reduced in advance in the preprocessing stage. For a set of selected (m_1, n_1, m_2), when n_2 satisfies the inequality above, the search for n_2 should

be pruned directly as $n_1 = n_2$. In addition, the inequality can be explained qualitatively. It suggests that for k-selection with the tiling shape n_1, we further divide n_1 into p smaller tiling shapes n_2. If the constant cost of the p smaller tiling shapes' execution time ($pg_0(\frac{n_1}{p})$) is larger than the maximum execution time of the original tiling shape n_1 ($g_0(n_1) + kg(n_1)$), the division of the p tiles is ineffective.

3.5 Evaluation

We implement SelB-k-NN algorithm on Huawei Ascend AI processors based on DaVinci [9] architecture for evaluation. The essence of the Ascend AI processors is DaVinci AI Cores, which provide 4 TFLOPS (FP16) or 8 TOPS (INT8) computation power per core. For the matrix multiplication phase, we directly use the official native functions in CANN libraries. The detailed hardware model we used is the Huawei Ascend 310 processor with 2 vCPUs of Intel Cascade Lake 6278 2.6GHz. All experiment results are wall-clock execution period from the start of the distance computation kernel to the end of the k-selection kernel (or the end of k-selection computation in the CPU approaches). For the evaluation dataset, since SelB-k-NN is insensitive to the data distribution, we build randomly generated synthetic datasets. The data points are 128-dimensional with the value of each dimension in the range of $[-1, 1]$ on normal distribution.

3.5.1 Single Execution of SelB-k-NN

We compare SelB-k-NN with the bitonic k-selection approach, heap-based approach on the AI processors, and host CPU C++ STL quick sort approach. The distance computations of all approaches are accelerated by the Matrix MACs. We offer the implementation results without hardware indexing and `vecCmp` & `vecSel` supports for the SelB-k-NN algorithm. Fig. 3.3 shows the results of the test cases where the size of the test dataset is (32×128), and

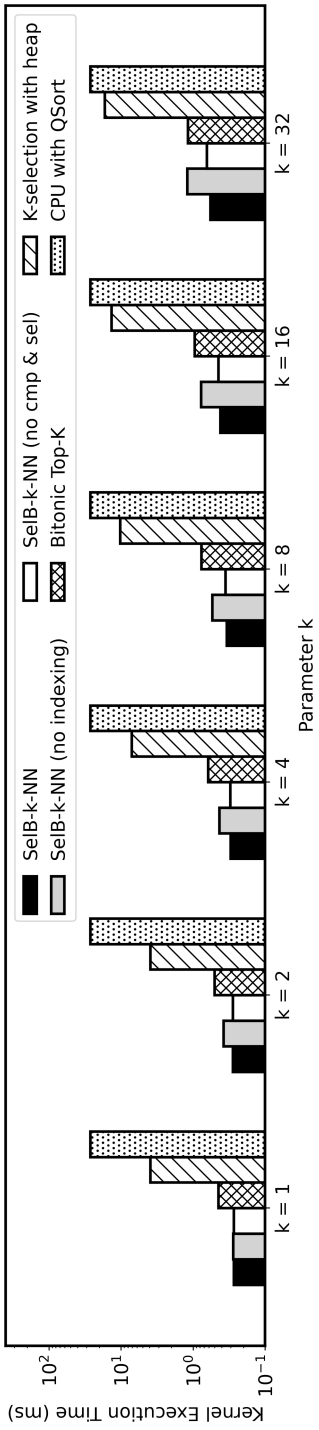


Figure 3.3: The single execution results of several k-NN algorithm kernels

the size of the training dataset is (128×4096). We analyze the performance changes with the variation of the parameter k of k-NN.

Overall Performance

Generally, SelB-k-NN algorithm reports incredible accelerations. It achieves an average acceleration of 2.01× compared with the bitonic approach, 23.93× compared with the heap approach, and 78.52× compared with the CPU quicksort implementation. With the increment of the parameter k, except for the CPU quicksort, which has a constant workload, the execution time of the other implementations increases accordingly. The execution time of SelB-k-NN when $k = 32$ grows to 1.55× compared with that when $k = 1$, while the bitonic approach increases to 2.13× and the heap approach rises to 3.45×. Although the overall time complexity of SelB-k-NN is $O(k+k^2+nk)$, which theoretically increases faster than the other approaches with that of $O(nlog^2k)$ and $O(nlogk)$, the practical experiments report the opposite results. The abnormal behavior is mainly because of the inefficient performance of the weakly-supported scalar operations, vectorized comparisons & selections on the AI processors, as we discussed. Time complexity assumes that all operations are equal, which is false on the AI processors. It evidences that the design principle of our SelB-k-NN algorithm, to reduce the two operations as more as possible, meets the special hardware requirements of the AI processors. For the implementation without hardware indexing supports, the results show that the software indexing methods extend the execution time of the SelB-k-NN algorithm to 1.56× averagely. For the implementation without hardware `vecCmp` and `vecSel` supports, our algorithm reports similar results with an average number of 1.04×. At least in SelB-k-NN, our software algorithm suffers a slight performance loss, which still achieves high efficiency on the AI processors.

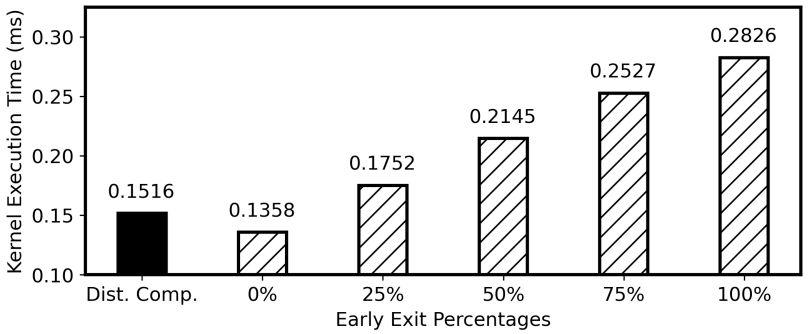


Figure 3.4: The k-selection results with different early exit percentages

K-Selection Early Exit

By modifying the input datasets and the initial k-min values, the number of updating values can be controlled intentionally. Therefore, we practically control and evaluate the execution time reduced by the early exit policy. Fig. 3.4 illustrates the execution results for the case when $k = 16$ with the size of test dataset (32×128) and training dataset (128×4096). The early exit percentage represents the percentages of the workloads that have been done when the algorithm exits. We also add the execution time of the distance computation kernel to compute the exact proportion of the whole algorithm execution time being reduced.

The results report a linear growth of the k-selection algorithm with the increment of the workloads. The k-selection execution time of the case with all workloads is 2.08× than that with no workloads. For the whole algorithm execution time with the distance computation, the acceleration is 1.51×. The results show that our early exit policy works well and significantly reduces the workload. In addition, even though there is no computation workload assigned to the kernel, the k-selection algorithm still takes 0.1516 ms. The results show that the simple but slow scalar conditional branch operations take 48.22% of the k-selection kernel, which further shows the low performance of the scalar units on the AI processors.

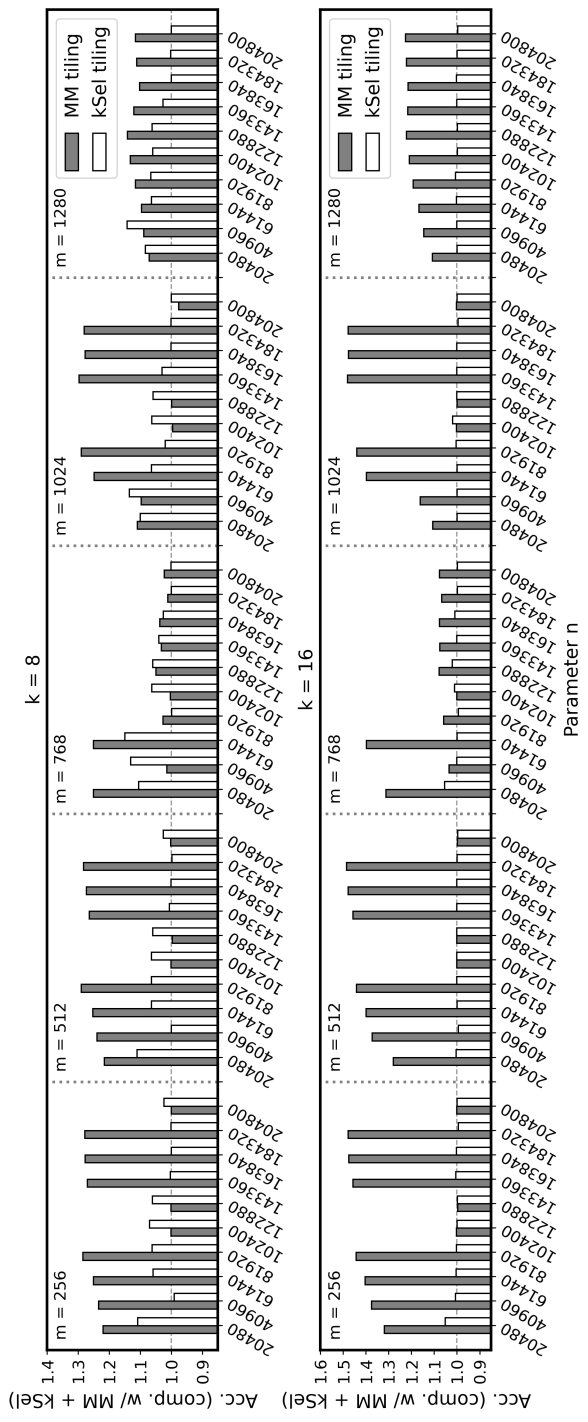


Figure 3.5: The accelerations of the overall best best tiling shapes (MM + kSel) compared with the individual best tiling shapes (MM & kSel)

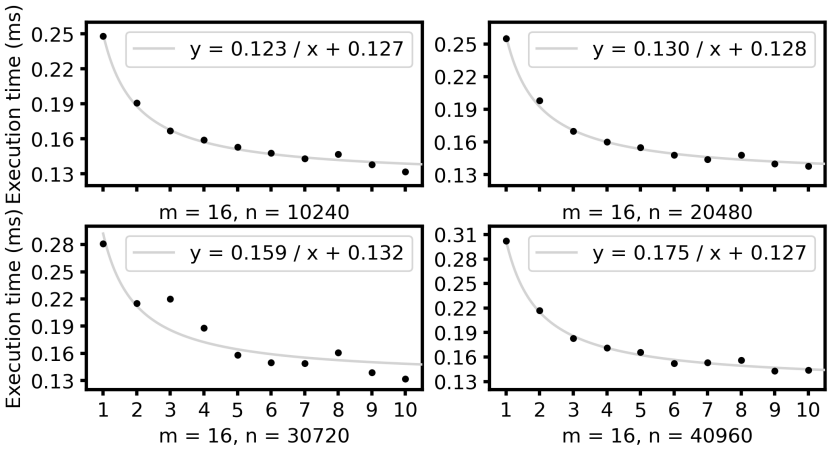


Figure 3.6: The k-selection execution time of each tile and the fitting curves

3.5.2 Mini-batch Execution of SelB-k-NN

K-selection Workload Quantification

We evaluate the execution time of sequential k-selection tiles after tiling, where each tile receives the min-k results from the last adjacent tile to reduce its workloads. Fig. 3.6 shows the evaluation results for four examples. The entire training dataset is divided into ten tiles along the n-direction. Each point in the figures represents the k-selection execution time of the t-th tile.

For all four examples, the execution time decreases in an inverse proportion, as discussed in Sec. 3.4.1. The curves in the figures are fitted by the non-linear least squares method with an average coefficient of determination, $R^2 = 0.97$. With the increment of n, the execution time of each t-th tile increases plus a fixed constant cost of about 0.13 ms. For the example of $m = 16, n = 40960$, a tile is the single execution we shown in Sec. 3.5.1. Compared with the early exit results in Fig. 3.4, the workload being executed on each tile is apparent to be looked up, which also follows the inverse proportion. The curves and the workload variations evidence the correctness of our k-selection qualification.

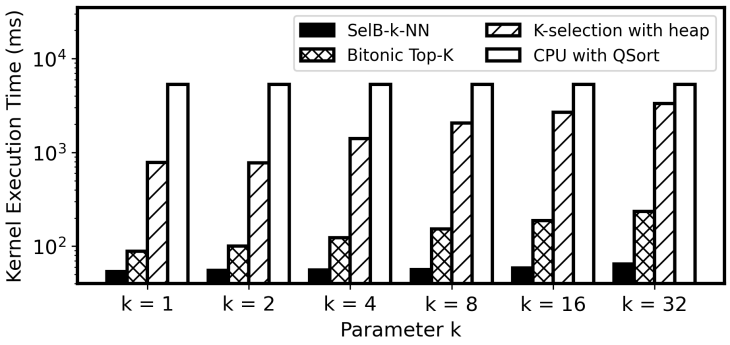


Figure 3.7: The k-selection execution time of each tile and the fitting curves

Mini-batch SelB-k-NN Performance

We compare the performance of mini-batch SelB-k-NN with other approaches with a larger dataset, where the test dataset is (1280×128) and the training dataset is (128×204800). The tiling shape we choose for SelB-k-NN is (16×128) and (128×4096).

Fig. 3.7 shows the results for the comparison with different k-values. Generally, compared with the single execution results, mini-batch SelB-k-NN achieves higher accelerations. It reports an average speedup of 2.53× compared with the bitonic approaches, 31.24× with the heap approaches, and 92.75× with the CPU quicksort implementations. Meanwhile, with the increment of the k-value, we observe the accelerations of SelB-k-NN compared with other algorithms also grow significantly. The main reason for the higher accelerations is the early exit policy in SelB-k-NN, which skipped most of the mini-batches in the large training dataset.

Tiling Shapes Results

We evaluate the performance of SelB-k-NN with different tiling shapes. We first profile the execution time of the matrix multiplication, $f(m_1, n_1)$, and that of the k-selection, $g(m_2, n_2)$ and $g_0(m_2, n_2)$. Then with the collected performance data, we solve the optimization problem in Eq. 3.9 for the

optimal tiling shape (m_1, n_1, m_2, n_2). In addition, we also solve the individual optimization problem of the matrix multiplication (m_1, n_1) and k-selection (m_2, n_2) to apply them to the both phases for comparisons.

Fig. 3.5 illustrates the acceleration rates of the optimal tiling shapes (MM + kSel) compared with the other two tiling shapes (MM & kSel) for the cases $k = 8$ and $k = 16$. Since the execution time of the CANN native matrix multiplication does not increase linearly with the expansion of the tiling shapes, the acceleration rates vary irregularly. Compared with the matrix multiplication tiling shapes, the optimal tiling shapes achieve an acceleration of $1.29\times$ when $k = 8$ and $1.48\times$ when $k = 16$. The acceleration is $1.14\times$ when $k = 8$ and $1.05\times$ when $k = 16$ compared with the k-selection tiling shapes. The figure shows that the matrix multiplication tiling shapes significantly lower the performance in most cases. In contrast, the performance with the k-selection tiling shapes is closer to that with the optimal tiling shapes. Moreover, the optimal tiling shape acceleration compared with the best matrix multiplication tiling shapes when $k = 16$ is larger than $k = 8$. The results are opposite for the acceleration compared with the best k-selection tiling shapes. It is because when $k = 16$, the matrix multiplication occupies more proportions of the algorithm execution time. The whole execution is mainly determined by the k-selection phase, which performs poorly with the matrix multiplication tiling shapes. Although the matrix multiplication also has a performance loss with the k-selection tiling shapes, the execution time increment is far less than the decrement by the k-selection phase. For the case when $k = 8$, the reasons are symmetric. On the other hand, in some cases, the two tiling shapes both cause a worse performance than the performance with overall tiling shapes, where two phases have similar influences on the whole execution time.

In addition, with our pruning method, for the case when $k = 8$, 72.80% of the optimization problem's search space is reduced; for $k = 16$, 55.23% of the space is reduced. The percentage of the pruned domain is related to parameter k, which determines the proportion of the constant cost to the

execution time related to the workloads of the k-selection.

3.6 Conclusion

This chapter introduced SelB-k-NN, which is a novel k-NN algorithm specifically designed for the AI processors. Based on the fact that the scalar operations, vectorized comparisons & selections perform unexpectedly poorly on the AI processors, the main idea of SelB-k-NN is to decrease the two operations as much as possible by combining the selection sort and bitonic k-selection. Since the AI processors produced by different manufacturers have various designs, we propose two algorithms to reduce hardware support requirements to the most critical operations of deep learning. Especially, the involvement of `ReLU` in the bitwise operations brings new thoughts for the famous Bit Twiddling Hack problems. We believe our first step of this algorithm could inspire more brilliant ideas for those classical solutions when applied to the AI processors. For large datasets, we set up an optimization problem for the optimal performance based on accurate quantification with a pruning method working in the preprocessing stage.

Chapter 4

Cube-fx: Mapping to Matrix Multiplications

Last chapter introduce the first optimization approach, replacing the inefficient operations. However, the approach still leave the Matrix MACs, the most significant hardware unit of the AI processors, idle. Therefore, in this chapter, we discuss the second optimization approach with the example of Cube-fx. Sec 4.1 introduces the existing algorithms to evaluate special functions as the background. Sec 4.2 discusses the algorithm Cube-fx in detail, which includes two main stages: the computation stage in Sec. 4.2.1 and the preparation stage in Sec. 4.2.2. Sec. 4.3 introduces an enhanced mapping, which aims to enhance Cube-fx for more general cases. Sec. 4.4 evaluates Cube-fx and Sec. 4.5 gives the conclusions.

4.1 Background & Motivation

4.1.1 Taylor Expansion

Taylor expansion is a powerful mathematical tool for evaluating a function. It expands infinitely differentiable functions as a series named Taylor series that only contains additions and multiplication. Eq. 4.1 shows the definition

of Taylor series, where $f(x)$ is expanded to n degrees at point a. The polynomial part of Taylor series is called Taylor polynomial. In the interval of convergence, Taylor polynomial approximates the functions with ignorable errors.

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k + o[(x-a)^n] \tag{4.1}$$

One of the common approaches to evaluate a Taylor polynomial is Horner's Method. Evaluating an n-degree polynomial in its original form, such as $p(x)$ in Eq. 4.2, requires $(n^2+n)/2$ multiplications and n additions. Horner's Method reduces this to n multiplications and additions by transforming the polynomial into a combination of n linear polynomials. Estrin's Scheme [90] further decomposes the polynomial into independent sub-polynomials that can be computed in parallel, requiring $(n-1)$ multiplications and additions, along with an additional $\log_2 n$ squarings, when n is a power of 2. Motzkin's Method [91], using coefficient pre-processing, divides the polynomial into several sub-polynomials. For a 4th-degree polynomial, Motzkin's Method requires only 2 multiplications and 5 additions, compared to 4 multiplications and additions with Horner's Method, making it more efficient on hardware where multiplications are costly. These methods, as injective functions, can naturally be adapted for parallel implementation using SIMD on vectorized units when dealing with large inputs. However, as discussed in Sec. 1.1.2, current AI processors perform poorly on vectorized operations, leading to performance losses. Therefore, instead of reducing the number of vectorized operations, leveraging Matrix MACs for computation will be more effective on AI processors.

$$\begin{aligned} f_a(x) &= a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0 \\ &= (((a_n x + a_{n-1})x + a_{n-2})x + ... + a_1)x + a_0 \end{aligned} \tag{4.2}$$

Furthermore, in applications such as optimization problems [55], feature extraction [56], or combinations of multiple activations [57], data often need

to be processed by multiple functions. Methods like Horner's Method, Estrin's Scheme, and Motzkin's Method handle each function independently, requiring redundant calculations. For example, consider two functions, $f_a(x)$ in Eq. 4.2 and $f_b(x)$, which share the same input x. Taking Horner's Method as an example, the intermediate result from computing $f_a(x)$, such as $(a_n x + a_{n-1})$, is discarded immediately after computing $f_a(x)$ and cannot be reused for $f_b(x)$. This results in repeating the evaluation process for each function. In contrast, using the original form of Taylor polynomials allows for common intermediate results, such as the sequence of powers $(x^n, x^{n-1}, \ldots, x)$, to be reused by multiple functions. If $f_a(x)$ generates this sequence, $f_b(x)$ and other functions can reuse it for their computations and reduce redundancy. However, generating this sequence is time-consuming with the vectorized units on AI processors, potentially offsetting the benefits of intermediate result reuse because of the inefficiencies in vectorized operations.

4.2 Cube-fx Algorithm

Fig. 4.1 gives an overview of Cube-fx. It contains two main stages: the preparation stage to generate the intermediate sequence $(x, x^2, x^3, ...)$ from input x and the computation stage to evaluate multiple functions in parallel. Compared with previous implementations composed of vectorized operations, either stage of Cube-fx contains one matrix multiplication as the key mapping step accelerated by the Matrix MACs. Each matrix of the matrix multiplications is specially designed and fixed, which aims to avoid extra matrix generation.

4.2.1 Computation Stage

The computation stage is mainly a matrix multiplication, shown in Eq. 4.3, where m is the number of inputs $(x_1, x_2, ..., x_m)$, k is the order number, and

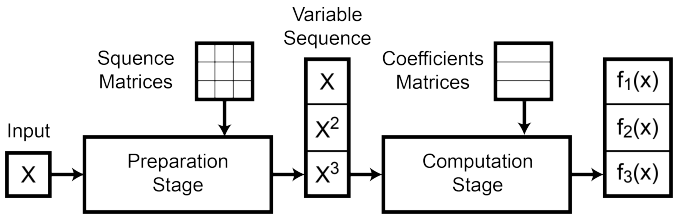


Figure 4.1: An overview of Cube-fx Algorithm for a single input

j is the number of evaluated functions. The left matrix of the matrix multiplication is a matrix of the Variable Sequence $(x, x^2, x^3, ...)$, which is built from the inputs and listed in the row-major order. The right matrix is the Coefficients Matrix, which is expanded from multiple functions $(f_a, f_b, ..., f_j)$ to k order and listed in the column-major order. For each evaluated function $(f_a, f_b, ..., f_j)$, the coefficients of Taylor polynomials are $(a_1, a_2, ..., a_k)$, $(b_1, b_2, ..., b_k)$, ..., $(j_1, j_2, ..., j_k)$. In this way, each element of the result matrix is a function evaluation result based on its related Taylor polynomial. For the example of $f_t(x)$ in the lower part of Eq. 4.3, which is expanded at point p, the elements in column t of the result matrix are the evaluation results of $f_t(x)$ with inputs from $(x_1, x_2, ..., x_m)$ and the coefficients $(t_1, t_2, ..., t_k)$. Therefore, the computation stage evaluates $(m \times j)$ Taylor polynomials in one step with the Matrix MACs.

$$\begin{bmatrix} x_1 & x_1^2 & \dots & x_1^k \\ x_2 & x_2^2 & \dots & x_2^k \\ \vdots & & \ddots & \vdots \\ x_m & x_m^2 & \dots & x_m^k \end{bmatrix} \begin{bmatrix} a_1 & b_1 & \dots & j_1 \\ a_2 & b_2 & \dots & j_2 \\ \vdots & & \ddots & \vdots \\ a_k & b_k & \dots & j_k \end{bmatrix} \approx \begin{bmatrix} f_a(x_1) & \dots & f_j(x_1) \\ f_a(x_2) & \dots & f_j(x_2) \\ \vdots & \ddots & \vdots \\ f_a(x_m) & \dots & f_j(x_m) \end{bmatrix} \tag{4.3}$$

$$f_t(x) = \sum_{s=0}^{k} \frac{f_t^{(s)}(p)}{s!}(x-p)^s + o[(x-p)^k]$$
$$\approx t_1 x + t_2 x^2 + \cdots + t_k x^k$$

Since the evaluated functions are chosen before launching the kernel, the values of the right matrix, the Coefficient Matrix, are predetermined and

constant. Therefore, we consider the Coefficient Matrix a read-only memory segment that can be directly written in kernel codes. On the other hand, the data layout requirement of the left matrix, the Variable Sequence Matrix, is more stringent. For most of the AI processors, the left matrix of the Matrix MACs adopts the row-major, including Nvidia GPUs with tensor cores [80], Cambricon MLUs [85], and Huawei Ascend AI processors in this paper [9]. With the right matrix in column-major, the design can significantly improve the efficiency of matrix multiplications. Then, for a variable sequence $(x_i, x_i^2, x_i^3, ...)$ mapped by the input x_i in the Variable Sequence Matrix, as shown in 4.3, it should be stored contiguously in the memory. In other words, after the preparation stage, the data at the memory address x_{i+1} in the kernel inputs should be moved to x_{i+k} with its powers at $(x_{i+k+1}, x_{i+k+2}, ...)$. Then, its neighbor, data at x_{i+2}, should be moved to x_{i+2k} with its powers at $(x_{i+2k+1}, x_{i+2k+2}, ...)$. Without a built-in vectorized *scatter* instruction, the scattering process could only be done by weak scalar operations for each input x_i, which significantly ruins the kernel performance of the AI processors. Therefore, the primary objective of the preparation stage is to complete the scattering process with the Matrix MACs instead of the vector or scalar units.

4.2.2 Preparation Stage

$$e^{k \ln v} = e^{\ln v^k} = v^k \tag{4.4}$$

The preparation stage of Cube-fx is based on two mathematical formulas. The first combines the logarithmic power law and the logarithmic definition, given in Eq. 4.4, where k is a constant, $v \in \mathbb{R}$, converting the product to power.

$$\begin{bmatrix} \ln v_{1,1} & \dots & \ln v_{1,s} \\ \vdots & \ddots & \vdots \\ \ln v_{m,1} & \dots & \ln v_{m,s} \end{bmatrix} \cdot mat = \begin{bmatrix} \ln {v_{1,i_0}}^{k_1} & \dots & \ln {v_{1,i_0}}^{k_{j_0}} \\ \vdots & \ddots & \vdots \\ \ln {v_{m,i_0}}^{k_1} & \dots & \ln {v_{m,i_0}}^{k_{j_0}} \end{bmatrix} \tag{4.5}$$

$$mat = (b_{i,j}) \in \mathbb{F}^{s \times j_0}, b_{i,j} = \begin{cases} 0, i \neq i_0 \\ k_j, i = i_0 \end{cases}$$

The second formula is a special case of matrix multiplications, given in Eq. 4.5, where the left matrix is an input matrix of $(m \times s)$, $p_{i,j} \in \mathbb{R}$, the right matrix mat is a matrix of $(s \times j_0)$, $k_j \in \mathbb{R}$. Except for a non-zero vector $(k_1, k_2, \dots, k_{j_0})$ at the row i_0, all other elements $b_{i,j}$ of mat are 0. Therefore, the matrix multiplication, shown as Eq. 4.5, computes the products of the column i_0 of the left matrix $(\ln v_{1,i_0}, \ln v_{2,i_0}, ..., \ln v_{m,i_0})^T$, the exponents of Eq. 4.4, and the row i_0 of the right matrix $(k_1, k_2, ..., k_{j_0})$ to form a full result matrix. Therefore, the required variable sequences are computed with an extra logarithmic operation.

Based on the two formulas, Alg. 7 presents the preparation stage of Cube-fx, where we consider the shape of the Matrix MACs is $(s \times s)$. In the beginning, Alg. 7 initializes an s-element array of $(s \times k)$ Sequence Matrices $mat[s]$, as we described in Eq. 4.5. For the idx-th Sequence Matrix $mat[idx]$, the non-zero row is at row idx, whose value is an increasing vector $(1, 2, \dots, k)$. Therefore, for each row idx of the $(s \times s)$ Matrix MACs, we construct a unique Sequence Matrix with a non-zero vector at that row. Then, Alg. 7 computes the logarithmic results $\mathbf{V}_{ln}$ of the input $\mathbf{V}$ with the built-in vectorized operation. The logarithmic results are reinterpreted as an $(\frac{N}{s} \times s)$ matrix $\mathbf{V}_{mat}$ and transferred to the Matrix MAC. At the mathematical level, the reinterpretation constructs the left matrix in Eq. 4.5, where $m = \frac{N}{s}$. At the system level, Huawei Ascend processors adopt a row-major order for the (16×16) left matrix of the basic matrix multiplication as discussed in Sec. **??**. Therefore, the continuous memory segment $\mathbf{V}_{ln}$ requires no additional data layout conversion, as it is inherently indexed in Z-order as an $(\frac{N}{s} \times s)$ through a single data transfer operation to the Matrix MAC.

Algorithm 7: Preparation stage of Cube-fx

Input : input 1D array $\mathbf{V}$ of the size N

Output: result sequences $\mathbf{Sq}$ of the size $(N \times k)$

1 $mat[s] \leftarrow \underbrace{\begin{bmatrix} 1\,2\,3\ldots k \\ 0\,0\,0\ldots 0 \\ \ldots \\ 0\,0\,0\ldots 0 \end{bmatrix} \ldots \begin{bmatrix} \ldots \\ 1\,2\,3\ldots k \\ \ldots \\ 0\,0\,0\ldots 0 \end{bmatrix} \ldots \begin{bmatrix} 0\,0\,0\ldots 0 \\ 0\,0\,0\ldots 0 \\ \ldots \\ 1\,2\,3\ldots k \end{bmatrix}}_{An\ s-array\ of\ (s\times k)\ matrices}$

2 $\mathbf{V}_{ln} \leftarrow \mathbf{ln}(\mathbf{V})$

3 $\mathbf{V}_{mat} \leftarrow$ reinterpret $\mathbf{V}_{ln}$ as a $(N/s) \times s$ matrix

4 **for** $idx \leftarrow 0$ **to** s **do**

5 $\quad \mathbf{Sq}_{mat}[idx] \leftarrow \mathbf{V}_{mat} \cdot mat[idx]$

6 $\quad \mathbf{Sq}_{ln}[idx] \leftarrow$ reinterpret $\mathbf{Sq}_{mat}[idx]$ as an 1D array

7 $\quad \mathbf{Sq}[idx] \leftarrow \mathbf{exp}(\mathbf{Sq}_{ln}[idx])$

8 **return Sq**

The next step is the central step of the preparation stage, which is an s-step loop. In the loop step idx, the logarithmic inputs $\mathbf{V}_{mat}$ take matrix multiplications with the Sequence Matrix $mat[idx]$. As discussed in Eq. 4.5, the idx-th matrix multiplication generates the $(\frac{N}{s} \times s)$ Variable Sequence matrix for the column idx of the logarithmic inputs. For the example of the second matrix multiplication in Fig. 4.2, the non-zero row of the second Sequence Matrix $mat[1]$ is at the second row $mat[1][1, :]$. Therefore, the result matrix is formed by the product of the second column of the logarithmic inputs $\mathbf{V}_{mat}[:, 1]$ and the increasing vector $(1, 2, \ldots, k)$. Since the logarithmic results are shaped as $(\frac{N}{s} \times s)$, s columns need s loop steps and the Sequence Matrice in total.

After the matrix multiplication in each loop step, the preparation stage applies an exponential function to the result matrices, which is reinterpreted as an array. As shown in Eq. 4.4, it restores the logarithmic product results

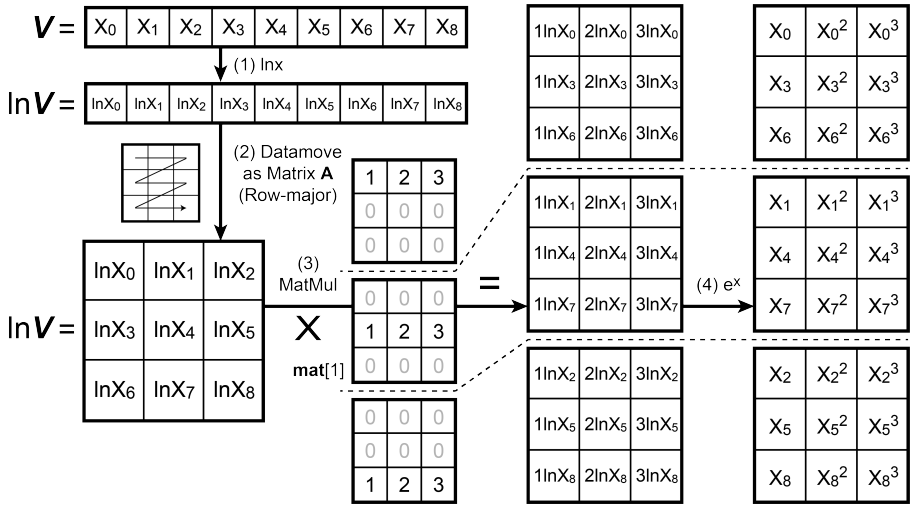


Figure 4.2: An example for 9 inputs with 3×3 Matrix MACs

$\mathbf{Sq}_{ln}$ to the original power results. Finally, the preparation stage completes the construction of the Variable Sequence, which perfectly meets the data layout requirement of the computation stage. In addition, although the results are currently shaped as an array, similar to $\mathbf{V}_{ln}$ without extra operations, they can be directly passed to the computation stage as $(\frac{N}{s} \times k)$ matrices.

The time complexity of the preparation stage is $O(Nk) + O(Nk/s^2)$ for sequential execution, where N is the size of input data, k is the order number of Taylor expansion, and s is the side length of the Matrix MACs. The first part $O(Nk)$ is the time complexity of the vectorized operations, which equals that of Horner's Method. The second part $O(Nk/s^2)$ is the time complexity of the matrix multiplications on the Matrix MACs. Theoretically, the execution time of the preparation stage is slightly longer than Horner's Method for a single-function evaluation. In detail, the number of vectorized operations of the preparation stage is $Nk+N$, while that of Horner's Method is $2Nk$. Therefore, although the vectorized operations of Cube-fx are exp and ln, the preparation stage still performs similarly to Horner's Method. In addition, the execution of the Matrix MACs and Vector Units can be scheduled concurrently. In theory, the longer execution can entirely hide the

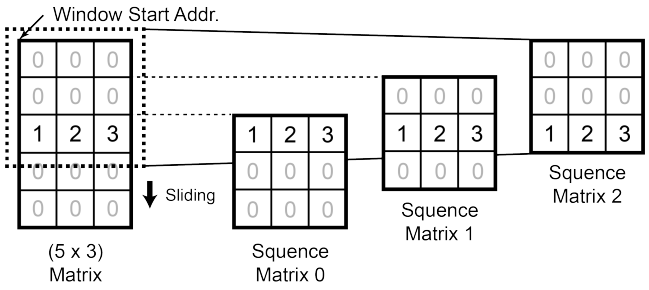


Figure 4.3: An example for the sliding window for 9 inputs with 3×3 Matrix MACs

execution of the other. Therefore, the concurrent time complexity can be reduced to $max(O(Nk), O(Nk/s^2))$ in the best situation. We will describe the concurrent execution in the following subsection. As a result, the time complexity of the preparation stage is acceptable since it is built for multiple functions instead of only one function in the later computation stage.

For the space complexity, the extra memory needed by the preparation stage is $O(ks^2)$, which stores the preparation matrices. However, a sliding window policy can reduce the space complexity to $O(ks)$, which allocates $((2/s-1) \times n)$ matrix. As shown in 4.3, the non-vector is at the second row, while other rows are filled with 0. Therefore, when the (3×3) windows slide down from row 0 to row 2, the three required Sequence Matrices are offered respectively.

4.2.3 Concurrent Execution

Since Cube-fx executes on the Matrix MACs and vectorized units, a naive sequential implementation could keep one unit idle while the other is busy, as well as the related data movement. Therefore, this subsection demonstrates a concurrent implementation on Ascend 910B processors.

To achieve concurrent execution on Ascend 910B processors, a technology named Double Buffer (or Ping Pong Buffer [92]) is used. Given an algorithm, we first split the source codes into the parts of Cube Units and Vector Units

Algorithm 8: Concurrent Cube-fx on Ascend 910B

Input : input 1D array V of the size N
input 2D array $\mathbf{C}$ of the size $(k \times j)$

Output: results **Res** of the size $(N \times j)$

1 **Vec:**
  $V_{ln} \leftarrow \mathbf{ln}(V)$
  Cube:
  $mat[s] \leftarrow Array\ of\ (s \times k)\ matrices$
2 **Cube:**
  $\mathbf{V}_{mat} \leftarrow$ reinterpret $\mathbf{V}_{ln}$ as a $(N/s) \times s$ matrix
  for $idx \leftarrow 0$ **to** $s/2$ **do**
    $\mathbf{Sq}_{mat}[idx] \leftarrow \mathbf{V}_{mat} \cdot mat[idx]$
3 **Vec:**
  for $idx \leftarrow 0$ **to** $s/2$ **do**
    $\mathbf{Sq}_{ln}[idx] \leftarrow$ reinterpret $\mathbf{Sq}_{mat}[idx]$ as an 1D array
    $\mathbf{Sq}[idx] \leftarrow \mathbf{exp}(\mathbf{Sq}_{ln}[idx])$
  Cube:
  for $idx \leftarrow s/2$ **to** s **do**
    $\mathbf{Sq}_{mat}[idx] \leftarrow \mathbf{V}_{mat} \cdot mat[idx]$
4 **Vec:**
  for $idx \leftarrow s/2$ **to** s **do**
    $\mathbf{Sq}_{ln}[idx] \leftarrow$ reinterpret $\mathbf{Sq}_{mat}[idx]$ as an 1D array
    $\mathbf{Sq}[idx] \leftarrow \mathbf{exp}(\mathbf{Sq}_{ln}[idx])$
  Cube:
  $\mathbf{Sq}_{mat_0} \leftarrow$ reinterpret $\mathbf{Sq}[0 : N/2][:]$ as a $(N/2) \times k$ matrix
  $\mathbf{Res}[0 : N/2][:] \leftarrow \mathbf{Sq}_{mat_0} \cdot \mathbf{C}$
5 **Cube:**
  $\mathbf{Sq}_{mat_1} \leftarrow$ reinterpret $\mathbf{Sq}[N/2 : N][:]$ as a $(N/2) \times k$ matrix
  $\mathbf{Res}[N/2 : N][:] \leftarrow \mathbf{Sq}_{mat_1} \cdot \mathbf{C}$
6 **return Res**

and mark the continuous segments with two units alternately appearing. Then, we cut the inputs of the marked segments into several pieces and then execute the pieces one by one to manually create a pipeline. The key is that while executing the second piece of the first part on one unit (Cube Unit), the first piece of the second part, which follows the first part of codes, is also started on the other unit (Vector Unit). As given in Alg. 8, each step contains the concurrent operations on two units (**Cube** for Cube Units and **Vec** for Vector Units). As shown in Fig. 1.2(b), Cube Units and Vector Units are separately located in Cube Cores and Vector Cores. While Cube Units compute the matrix multiplications as discussed in Sec. 1.2.2, a Vector Unit computes a fixed-size and continuous data segment in the pattern of SIMD, similar to most vectorized units on CPUs [93]. In addition to the computation, Alg. 8 also includes multiple data transfers through various data paths, connecting the global memory to the computation units. An inter-core synchronization is required between every two steps. For the example of Alg. 8, in Step 2, the Cube Unit first computes the first segment of $\mathbf{Sq}_{ln}$ between 0 and $s/2$. Then, in Step 3, the result is transferred to the Vector Unit for the later exponent computation. Meanwhile, the Cube Unit continues to compute the second segment of $\mathbf{Sq}_{ln}$ between $s/2$ and s. This way, the Cube and Vector Unit run concurrently in Step 3 and leave neither unit idle.

An inter-core synchronization is required between every two steps. For the example of Alg. 8, in Step 2, the Cube Unit first computes the first segment of $\mathbf{Sq}_{ln}$ between 0 and $s/2$. Then, in Step 3, the result is transferred to the Vector Unit for the later exponent computation. Meanwhile, the Cube Unit continues to compute the second segment of $\mathbf{Sq}_{ln}$ between $s/2$ and s. This way, the Cube and Vector Unit run concurrently in Step 3 and leave neither unit idle.

Especially, it is noticed that in Step 2, 5 of Alg. 8, the Vector Units are still idle. Theoretically, by splitting the segments as small as possible, the idleness of the Vector Units could be ignorable. The concurrent time com-

plexity is approaching $max(O(Nk), O(Nk/s^2))$ as we discussed. However, because of the slow inter-core binary semaphore, as discussed in Sec. 1.2.2, too many synchronizations cause significant overheads, which would cancel the benefits from the concurrent execution. Therefore, on Ascend 910B processors, we only split the data into two segments. In contrast, on Ascend 310 processors, which have a much faster binary semaphore and lower synchronization overheads, we split the data into 16 segments in our implementation to reduce the idleness.

4.3 Cube-fx for General Cases

The last section introduces Cube-fx in the case where the Matrix MACs are always fully utilized. In this section, we extend the algorithm to more general instances where the Matrix MACs are not saturated ($k \mod s \neq 0$).

We first define a metric, Matrix MAC Count, to evaluate the scale of matrix multiplications. As mentioned in Sec. 1.2.2, in one basic step, a Matrix MAC completes a fixed shape of matrix multiplication. Then, it accumulates the results from multiple steps as a whole matrix multiplication. Therefore, the Matrix MAC Count indicates how many basic fix-shaped matrix multiplications it takes for a Matrix MAC to finish a large matrix multiplication with the standard $O(n^3)$ block matrix multiplication algorithm. For example, for a Matrix MAC of the shape $(s \times s)$, the Matrix MAC Count of matrix multiplication of the shape $(m \times k \times n)$ is (mkn/s^3).

4.3.1 Enhanced Mapping of Cube-fx

Since the evaluation precision is not always the most significant target, the Taylor series's order requirement is flexible to improve efficiency with sight precision loss [94]. Therefore, in these situations, the order number does not have to equal the slide length of the Matrix MACs. Fig. 4.4 shows a naive application of Cube-fx where the order number is two, the side length is four, and the evaluated function number is two. In this case, we can easily

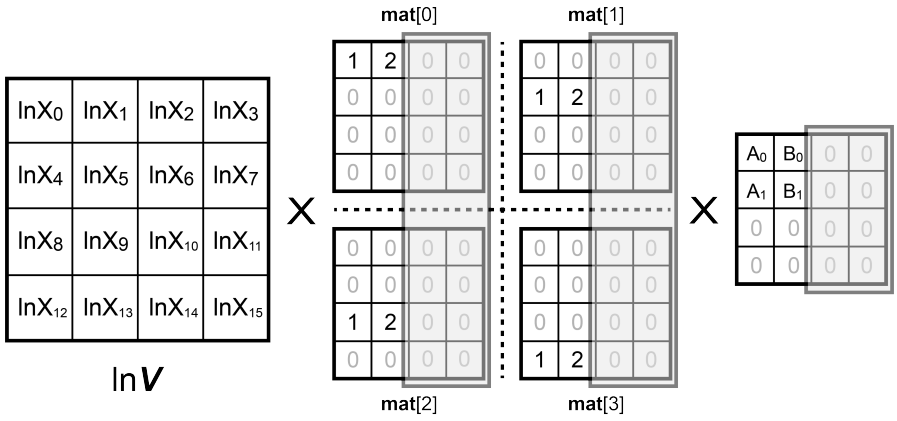


Figure 4.4: Naive Cube-fx where $s = 4, k = 2, j = 2$

observe that half of the Matrix MAC units we shade are full of zeros in the preparation and computation stages. Since all the units of the Matrix MACs must be utilized in each step of the matrix multiplication, the shaded units' computation power and results are wasted, which generates meaningless zeros.

To avoid the computation power waste, we propose an enhanced mapping for Cube-fx, which merges multiple matrices to the block diagonal matrices to reduce the number of Cube-fx loop steps. As shown in Fig. 4.5, in the preparation stage, the Sequence Matrix $mat[0]$ and $mat[1]$ are merged into one single block-diagonal Sequence Matrix $mat[0] + mat[1]$, where the two non-zero columns of $mat[1]$ are moved to the third and fourth column of the $mat[0]$. Therefore, the result Variable Sequence matrix is a combination of the columns of the two related result matrices. Symmetrically, to fit the Variable Sequence matrix generated in the preparation stage, the block-diagonal Coefficient Matrix also combines the two original Coefficient Matrices but combines the rows of the matrices instead of the columns in the computation stage. As a result, all the elements of the result matrix are non-zero and valid, which means no computation power is wasted to compute zeros. Furthermore, the step number of the main loop is decreased to two, and the Matrix MAC Count is cut from eight to four, significantly reducing the exe-

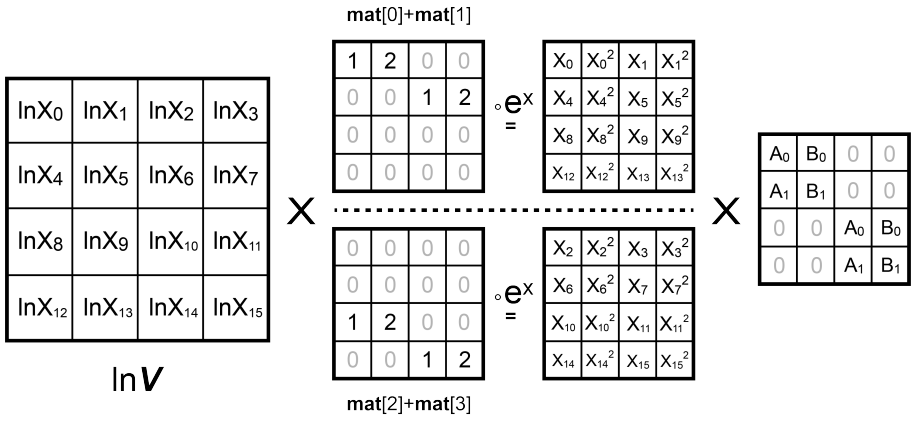


Figure 4.5: Enhanced Cube-fx where $s = 4, k = 2, j = 2$

cution time.

However, the mapping cannot improve the efficiency of Cube-fx in all cases. As the example given in Fig. 4.6, we increase the order number to three. Then, in the preparation stage, one Matrix MAC, whose side length is four, cannot simultaneously contain six non-zero columns of two Sequence Matrices. Therefore, to fully utilize the computation power, we can only fuse four original Sequence Matrices into three basic units of the Matrix MACs, whose total column number is 12. Since the matrices of a matrix multiplication should share the same k-dimension, the Coefficients Matrix must combine six matrices in the computation stage, whose row number is 12. The Matrix MAC Count of the enhanced Cube-fx increases to $(3+3\times 2 = 9)$ compared to $(4 + 4 = 8)$ with naive Cube-fx. Hence, in this case, the enhanced Cube-fx mapping does not decrease the Matrix MAC Count but increases a lot instead. Therefore, in the following subsection, we analyze and quantify when Cube-fx needs further mapping and how to decide the number of merged Sequence Matrices.

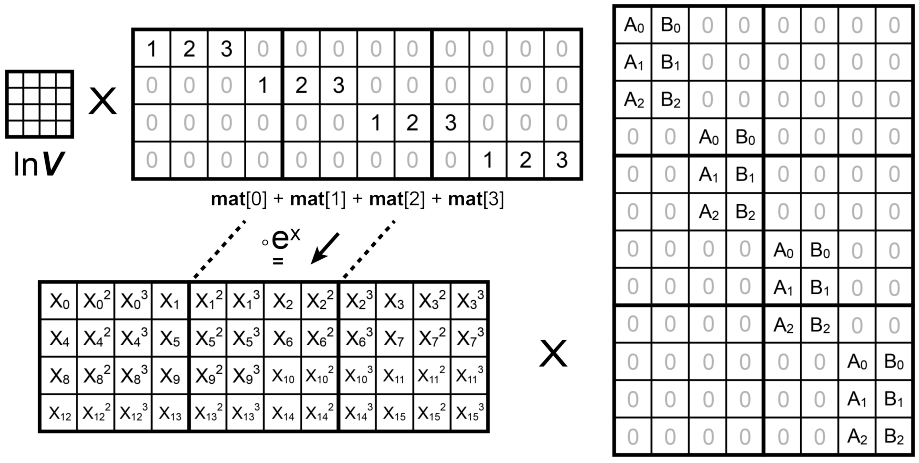


Figure 4.6: Enhanced Cube-fx where $s = 4, k = 3, j = 2$

4.3.2 Quantification

For a Matrix MAC of the shape $(s \times s)$, let us consider an application of Cube-fx, where the Taylor polynomial order number is k, the evaluated function number is j, and the input data length is $(s \times s)$.

In the preparation stage, since the order number is k, the shape of the non-zero data from each original Sequence Matrix is $(1 \times k)$. Let us merge t original Sequence Matrices, as shown in Fig. 4.7, where $t \in \mathbb{N}^+, 1 \leq t \leq s$. Then, the loop step L of Cube-fx is:

$$L = \left\lceil \frac{s}{t} \right\rceil \tag{4.6}$$

where $\lceil x \rceil$ is the ceiling function. The Matrix MAC Count of the preparation stage M_p for one loop step is:

$$M_p = \left\lceil \frac{kt}{s} \right\rceil \cdot s \cdot \frac{s^2}{s^3} = \left\lceil \frac{kt}{s} \right\rceil \tag{4.7}$$

Similarly, in the computation stage, the Matrix MAC Count of the computation stage M_c for one loop step is:

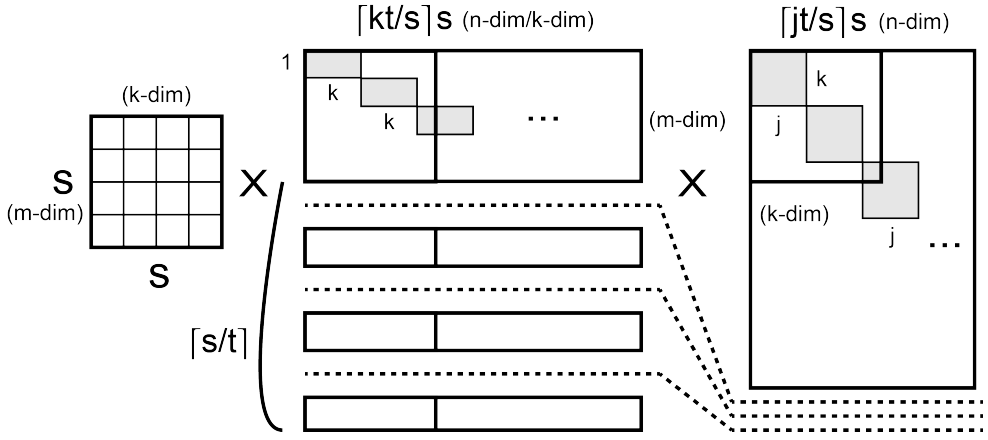


Figure 4.7: Enhanced Cube-fx for General Cases

$$M_c = \left\lceil \frac{jt}{s} \right\rceil \cdot \left\lceil \frac{kt}{s} \right\rceil \cdot s^2 \cdot \frac{s}{s^3} = \left\lceil \frac{jt}{s} \right\rceil \cdot \left\lceil \frac{kt}{s} \right\rceil \tag{4.8}$$

Hence, the Matrix MAC Count for L steps M is:

$$M = L \cdot (M_p + M_c) = \left\lceil \frac{s}{t} \right\rceil \cdot \left\lceil \frac{kt}{s} \right\rceil \cdot (1 + \left\lceil \frac{jt}{s} \right\rceil) \tag{4.9}$$

In addition, the size of the three matrices in Cube-Fx transferred from the global memory is formulated as:

$$\begin{aligned} S_z &= s^2 + \left\lceil \frac{s}{t} \right\rceil \cdot (s^2 \cdot \left\lceil \frac{kt}{s} \right\rceil + s^2 \cdot \left\lceil \frac{kt}{s} \right\rceil \cdot \left\lceil \frac{jt}{s} \right\rceil) \\ &= s^2 + \left\lceil \frac{s}{t} \right\rceil \cdot \left\lceil \frac{kt}{s} \right\rceil \cdot s^2 \cdot (1 + \left\lceil \frac{jt}{s} \right\rceil) \end{aligned} \tag{4.10}$$

We formulate an integer programming (IP) problem:

$$\begin{aligned} min \quad & M + bS_z = \left\lceil \frac{s}{t} \right\rceil \cdot \left\lceil \frac{kt}{s} \right\rceil \cdot (1 + \left\lceil \frac{jt}{s} \right\rceil) \cdot (1 + bs^2) + bs^2 \\ s.t. \quad & t \in \mathbb{N}^+, 1 \leq t \leq s \end{aligned} \tag{4.11}$$

where b is a constant value combining the Matrix MAC Count with the data movement. Since the data movement size is proportional to the Matrix MAC Count, the actual value of b is trivial in the optimization problem. Intuitively, increasing t can decrease the loop step L but raise the n-dimention in both the preparation and computation stages.

4.4 Evaluation

In this section, we evaluate the precision and efficiency performance of Cube-fx on Huawei Ascend 310 and Ascend 910B AI processors. Since Cube-fx can naturally execute in parallel, we only activate a single core during the evaluation for simplicity. Therefore, for Ascend 310 processors, we activate one Cube Unit with one Vector Unit; for Ascend 910B processors, we activate one Cube Unit with two Vector Units (one Cube Core with two Vector Cores), the most elementary structure. All the experiment results are evaluated from Huawei's official profiler tool, *msprof*, which reports the detailed runtime information of a DaVinci Kernel. The data type we used during the evaluation is FP16, one of AI applications' most popular data types.

4.4.1 Computation Precision Evaluation

We first evaluate the computation precision of the Cube-fx. We use six sample functions with three different inputs, including trigonometric functions and activation functions in neural networks, to compare the precision of Cube-fx with CORDIC and Horner's Method, respectively. CORDIC (COordinate Rotation DIgital Computer) [95] is an iterative algorithm used to efficiently calculate trigonometric, hyperbolic, exponential, and logarithmic functions. The core idea of CORDIC is to represent calculations as a sequence of vector rotations in a plane, where each rotation is chosen to progressively bring the resulting vector closer to the desired angle. This iterative process avoids the need for multiplications, making it particularly well-suited for hardware implementations where computational efficiency is critical. The

Table 4.1: Precision Evaluation for Float16

Function	Input	Numpy	CORDIC	Horner's Order 8	Cube-fx Order 8	Horner's Order 16	Cube-fx Order 16
sin(x)	0.14	0.1395	0.0%	0.012%	0.012%	0.012%	0.012%
	0.52	0.4969	0.0%	0.011%	0.011%	0.011%	0.011%
	2.08	0.8731	0.0%	0.178%	0.178%	0.046%	0.046%
cos(x)	0.14	0.9902	0.0%	0.002%	0.002%	0.002%	0.002%
	0.52	0.8678	0.0%	0.017%	0.017%	0.017%	0.017%
	2.08	-0.4875	0.0%	1.917%	1.917%	0.214%	0.214%
tan(x)	0.14	0.1409	0.0%	0.049%	0.049%	0.049%	0.049%
	0.52	0.5726	0.0%	0.052%	0.052%	0.034%	0.034%
	2.08	-1.791	0.0%	1181%	1180%	11160%	11160%
tanh(x)	0.14	0.1391	0.0%	0.049%	0.049%	0.049%	0.049%
	0.52	0.4777	0.0%	0.017%	0.017%	0.017%	0.017%
	2.08	0.9693	0.0%	597.3%	595.3%	5649%	5620%
Sigmoid(x)	0.14	0.5349	0.0%	0.040%	0.040%	0.040%	0.040%
	0.52	0.6271	0.0%	0.031%	0.031%	0.031%	0.031%
	2.08	0.8889	0.0%	1.239%	1.184%	0.134%	0.134%
GELU(x)	0.14	0.0778	-	0.034%	0.034%	0.049%	0.049%
	0.52	0.3632	-	0.021%	0.021%	0.021%	0.021%
	2.08	2.0410	-	14.17%	14.17%	0.289%	0.194%

two algorithms based on Taylor polynomials are expanded around $x = 0$. The error rates are calculated as the percentage difference between the output of the algorithm and the output of Numpy, a Python library for scientific computing.

First, CORDIC reports 0.000% error rates with the iteration of 8, which shows extremely high precision. The reason is that CORDIC is accurate to the bit of its iteration number, which is even higher than the precision of FP16. While the CORDIC implementations of $tanh(x)$ and $Sigmoid(x)$ are proposed by former researchers [96], CORDIC for $GELU(x)$ is still absent, which we do not list in the table. The other two algorithms based on Taylor

polynomials, including Cube-fx, report a few higher error rates. Generally, except for the outliers, the average error rates of Cube-fx are 1.112% and 0.058% for the order numbers 8 and 16, respectively. For comparison, the average error rates of Horner's Method are 1.115% and 0.063% for the order numbers 8 and 16. As for $tan(x)$ and $tanh(x)$ with the input 2.08, the high error rates are caused by the fact that 2.08 is out of the radius of convergence $(-\frac{\pi}{2}, \frac{\pi}{2})$ and diverges.

The evaluation results indicate that Cube-fx has comparable or lower error rates than Horner's method for most cases. Although the precision of Cube-fx is much lower than CORDIC, the errors are caused by the theoretical limit of Taylor expansion but not Cube-fx itself. In other words, compared to the pure multiplications and additions of Horner's method, the logarithm and exponent operations of Cube-fx do not bring extra errors to the computation and even reduce the errors for some cases. Therefore, for applications where Taylor expansion has met the precision requirements, Cube-fx would be safe to replace other implementations, including Horner's Method.

4.4.2 Full Matrix MAC Evaluation

In this subsection, we compare the execution time of Cube-fx with the naive Taylor expansion, CORDIC, Horner's Method, Estrin's Scheme, and Motzkin's Method. The naive Taylor implementation is used in SymPy [97] for its `series()` function, which builds the variable sequences and performs the multiplications with the Taylor polynomial coefficients. All implementations are SIMD-optimized using vectorized operations. For Taylor-based algorithms, we use an order of 16, matching the side length of the Matrix MACs ($k = s = 16$) to utilize the Matrix MACs on Ascend processors fully. For CORDIC, we use 4 iterations, providing high precision as discussed in Sec. 4.4.1. To ensure a fair comparison, all functions evaluated are trigonometric, avoiding the need for extra pre- and post-processing for CORDIC. We analyze the kernel execution time for varying numbers of evaluated functions, using input data sizes of 16384 and 32768 on Ascend 310 processors,

and 1638400 and 3276800 on Ascend 910B processors, which represent the length of the input data sequence being processed.

Results

Fig. 4.8 illustrates the evaluation results. Generally, all the algorithms grow with the increment of the evaluated function number. Since all the algorithms traverse the data for execution, the execution time linearly increases with the data sizes. Cube-fx shows significant speedups compared to other implementations when the number of the evaluated functions is large enough. For the results on Ascend 310 processors, Cube-fx shows an average result of 2.73× speedup compared to the naive Taylor expansion implementation, 6.06× speedup compared to CORDIC, 1.64× speedup compared to Horner's Method, 1.53× speedup compared to Estrin's Scheme and 1.85× speedup compared to Motzkin's Method. For the results on Ascend 910B processors, Cube-fx reports 1.63× speedup compared to the naive Taylor expansion implementation, 3.52× speedup compared to CORDIC, 1.28× speedup compared to Horner's Method, 1.20× speedup compared to Estrin's Scheme and 1.42× speedup compared to Motzkin's Method.

When the evaluation function number is 1, Horner's Method is 2.68× and 6.02× faster than Cube-fx, Estrin's Scheme is 2.87× and 6.38× faster, and Motzkin's Method reaches 2.38× and 4.76× on Ascend 310 and 910B processors, respectively. However, when the number is 9, Cube-fx reversely reports 2.48× and 1.05×, 2.48× and 1.19×, and 2.80× and 1.42× speedup instead. In detail, as shown in Fig. 4.8, on Ascend 310 processors, when the evaluation function number is larger than 3, Cube-fx reports better efficiency than other implementations and keeps enlarging the performance. On Ascend 910B processors, the turning point is 8. The main reason for the results is that compared to the time complexity we discussed in Sec. 4.2 for Cube-fx, the time complexity of general Cube-fx is $O(Nk + \lceil N/s \rceil \lceil k/s \rceil + \lceil N/s \rceil \lceil k/s \rceil \lceil j/s \rceil)$, where the execution time of Cube-fx increases every s evaluated functions. The ceiling functions are added here because the Matrix MAC must complete

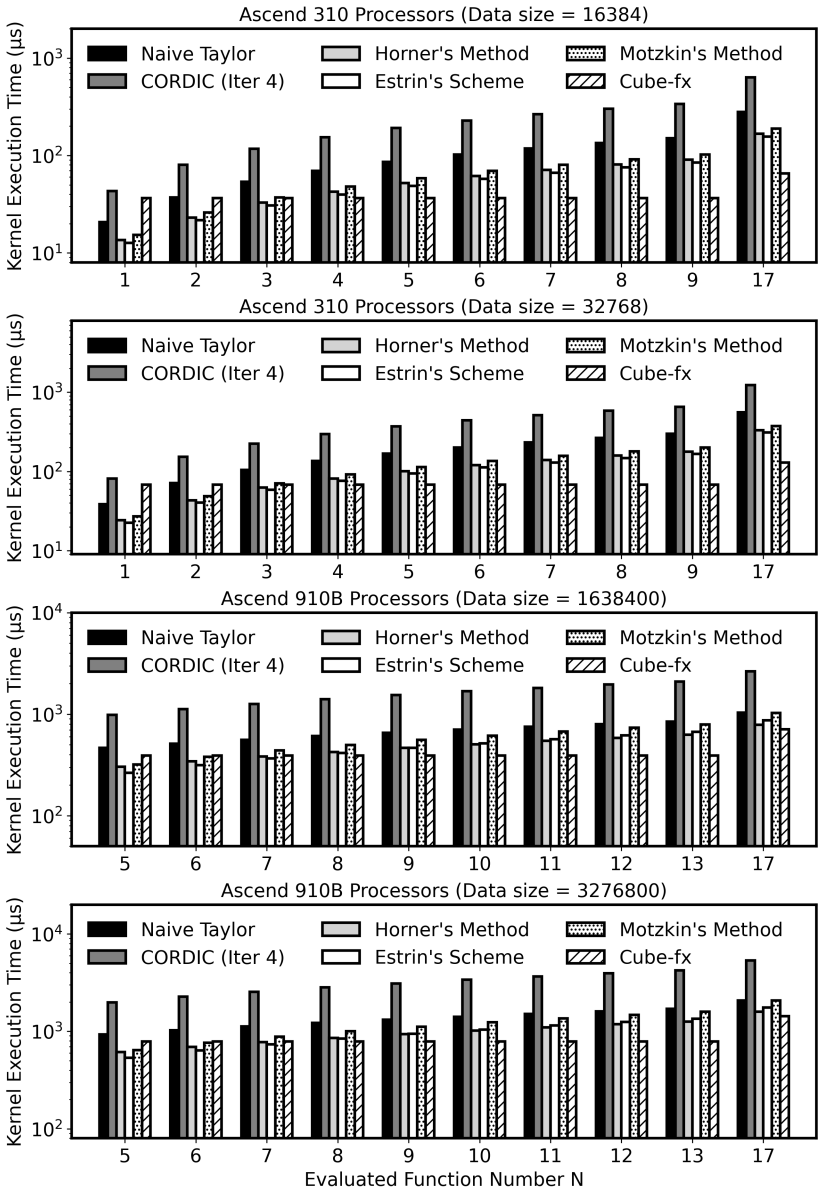


Figure 4.8: The execution time evaluations (order number = 16)

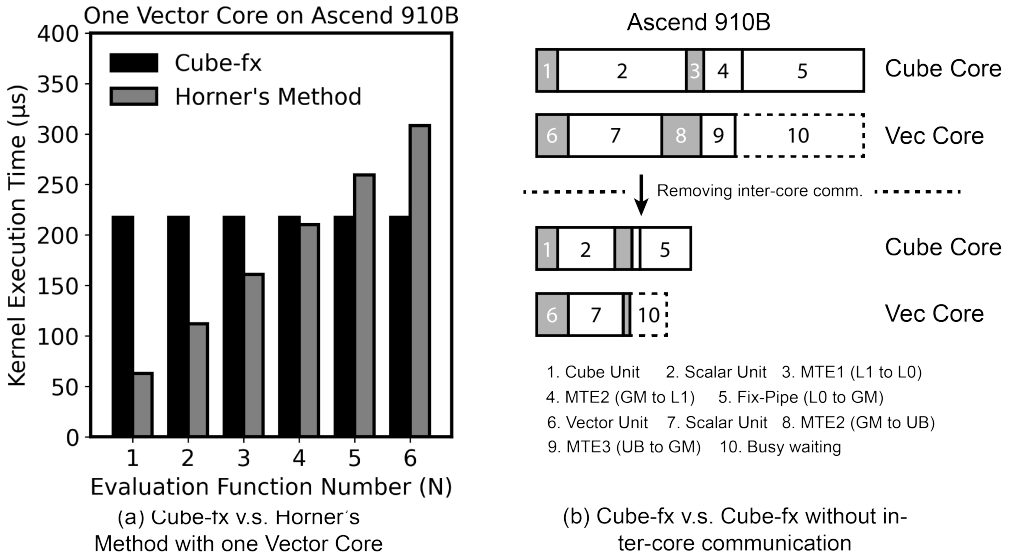


Figure 4.9: Performance comparison of modified implementations on Ascend 910B

a basic unit ($s \times s \times s$) of matrix multiplication each time, as mentioned in Sec. 4.3.2. Therefore, for the number of evaluated functions within $[1, 16]$, Cube-fx reports a nearly constant execution time, while the other implementations increase linearly. As for the evaluated function number of 17, where $\lceil 17/16 \rceil = 2$, the execution time of Cube-fx finally increases to 1.79× of the previous results on Ascend 310 processors. On Ascend 910B processors, the execution time of Cube-fx with the evaluated function number 17 increases to 1.81×. In addition, by activating the Matrix MACs, we reduce the utilization rate of the weak Vector Units to 56.38% on Ascend 310 and 34.95% on Ascend 910B, which move most of the workloads to the Matrix MACs.

Discussion

The results above reveal that Cube-fx performs much better on Ascend 310 processors than on Ascend 910B processors. Based on the hardware structure, this subsection discusses the reasons and the potential improvements.

First, the most significant reason is that Ascend 910B processors have two Vector Units, which provide double vectorized computation power compared to Ascend 310 processors. As discussed in Sec. 1.2.2, the Cube Unit of Ascend 310 processors is more dominant than Ascend 910B, which empowers our Cube-fx more than other implementations. When we disable one Vector Unit of Ascend 910B processors, Cube-fx reports similar accelerations compared with Horner's Method on Ascend 310 processors. When the evaluated function number is 4, Cube-fx performs only 1.03× of execution time compared with Horner's Method, as shown in Fig. 4.9(a). Since Cube-fx only contains a small part to be executed on Vector Units, the increment of the Vector Units cannot improve to the majority of the algorithm. The performance of Cube-fx is much more sensitive to the number of the Cube Units, as it highly utilizes the units. Therefore, Cube-fx performs better on Ascend 310 processors, the edge-computing hardware without rich vectorized computation resources.

Second, since Cube-fx heavily relies on the cooperation of the two computation units, the data communication between Cube Core and Vector Core heavily influences the performance of Cube-fx. As shown in Fig. 1.2, to transfer data between the Cube Units and Vector Units, Ascend 910B processors must write the results back to the global memory and then read from it. The IO operations contribute to most of the execution time compared with the execution on Ascend 310 processors. Fig. 4.9(b) shows the results of removing the data communication between Cube Core and Vector Core by manually deleting the related instructions. We illustrate the execution of Cube Core and Vector Core separately in Fig. 4.9(b). Each bar refers to the execution time of each hardware unit on the according core. As a result, the modified version without data communication (lower subgraph) reports a 43.46% reduction in the execution time compared with the original version (upper subgraph). Notice that in addition to the reduction of Fix-Pipe and MTE3 Units, the execution time of the Scalar Unit also reduces, which computes the memory address of the data communication. Furthermore, the

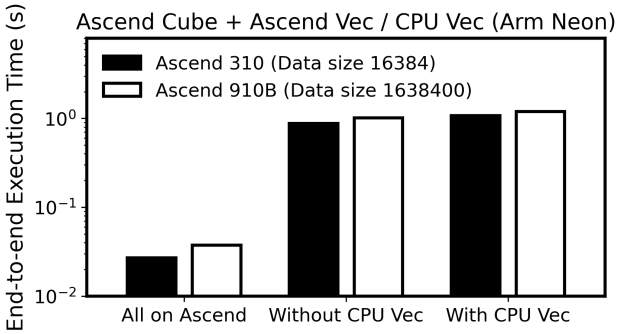


Figure 4.10: The comparison between (AI Cube + AI Vector) and (AI Cube + CPU Vector)

busy waiting time of Vector Core is significantly reduced to 28.55% of that in the original Cube-fx. The busy waiting time comprises the idleness and inter-core synchronization we discussed in Sec. 4.2.3. Even though we optimize the algorithm for potential concurrent execution, the extra inter-core data communication severely extends the execution periods on both units, which the concurrent execution cannot overlap and hide.

AI Cube + AI Vector vs AI Cube + CPU Vector

Since previous experiments identified vector units as a potential bottleneck, this subsection investigates the performance of integrating CPU vector units with Ascend Cube Units. We evaluate the performance of Cube-fx by combining CPU Vector Units (Arm Neon [98]) with Ascend Cube Units and compare it against using only Ascend processors. The input data sizes are configured as 16384 for Ascend 310 and 1638400 for Ascend 910B processors. In contrast to the previous experiments, where the Huawei profiler, *msprof*, was used for performance measurement, we use an end-to-end wall clock approach here. This change is because of the limitation that the profiler cannot be used to measure execution times on the IO and CPU sides.

As shown in Fig. 4.10, performing all the computations on the AI processors leads to much faster execution compared to using CPUs for vectorized

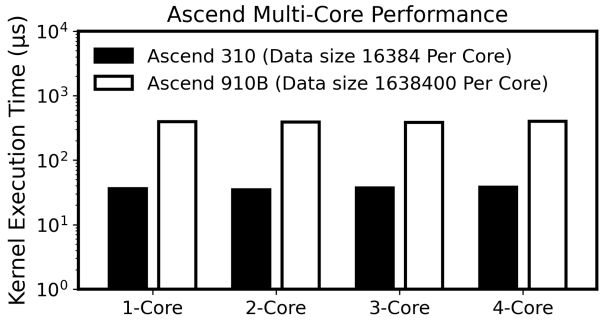


Figure 4.11: The multi-core execution of Cube-fx

computations, achieving speedups of 39.81× and 31.67× on Ascend 310 and 910B processors, respectively. We also included a scenario without using the CPUs for vectorized computations, labeled as 'Without CPU Vec' in Fig. 4.10. These results indicate that the execution time was still 32.63× and 26.89× longer compared to using only the AI processors. The main reason for this is that transferring data between the AI processors and CPUs takes too much time. Additionally, doing vectorized computations on the CPU is inherently slower. Firstly, Arm Neon Instructions do not directly support exponentiation or logarithmic functions, requiring extra processing and evaluation. Secondly, the CPU SIMD units (Huawei Kunpeng) are much narrower compared to Ascend Vector Units (8 float16 [99] vs 128 float16). Therefore, combining CPU Vector Units with Ascend Cube Units ends up being less efficient, making it more advantageous to conduct all computations on the AI processors alone.

Results for Multi-Cores

This subsection evaluates Cube-fx in multi-core cases on both Ascend 310 and 910B processors. For different core numbers, the data size per core is fixed at 16384 for Ascend 310 and 1638400 for Ascend 910B, respectively.

As shown in Fig. 4.11, the performance of Cube-fx in multi-core environments shows no clear performance loss compared to the single-core case,

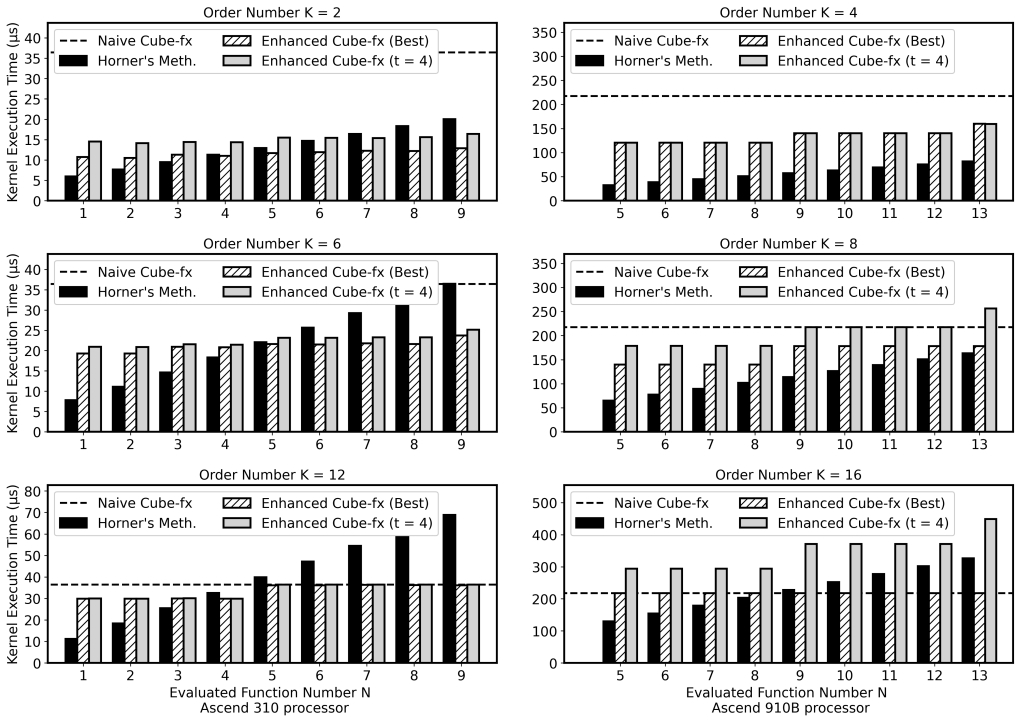


Figure 4.12: The accelerations of the enhanced Cube-fx

which suggests that Cube-fx runs efficiently when multiple cores are used. This result can be attributed to Cube-fx being a data parallelism-based algorithm, allowing it to distribute workloads effectively among different cores and fully utilize the processors' resources. This ensures scalability and maintains high performance across multiple cores, taking full advantage of the parallel processing capabilities and maximizing overall system performance.

4.4.3 Unfilled Matrix MAC Evaluation

This subsection evaluates the performance of the enhanced Cube-fx on both Ascend 310 processors and Ascend 910B processors. For cases where $k = \{2, 6, 12\}$, the results are evaluated on Ascend 310 processors, while for cases where $k = \{4, 8, 16\}$, the results are evaluated on Ascend 910B processors. To

indicate the performance improvement of the enhanced Cube-fx, we compare its execution time with those of Horner's Method, naive Cube-fx, and the enhanced Cube-fx with a fixed number of merged matrices $t = 4$. The data size used in this subsection is 16384 and 1638400, respectively, on Ascend 310 and Ascend 910B.

Fig. 4.12 illustrates our collected evaluation results. Since the execution time of naive Cube-fx for all cases is the same, we plot it as a dashed line in all charts, which reports significant performance gaps between the other two implementations in most cases. Generally, the enhanced Cube-fx reports an average speedup of 1.83× compared to naive Cube-fx on Ascend 310 processors and 1.63× on Ascend 910B processors. While the best case reports a 3.39× speedup on Ascend 310 processors and 2.68% speedup on Ascend 910B processors, the worst reports an equivalent execution time compared to naive Cube-fx. Compared with the results of the enhanced Cube-fx when $t = 4$, the enhanced Cube-fx with the optimized t reports an average speedup of 1.12× and 1.28×. These results mean that applying the enhanced mapping to Cube-fx for all cases is riskless, and our optimized problem helps find the optimized execution of Cube-fx. Compared with Horner's Method on Ascend 310 processors, the enhanced mapping empowers Cube-fx again and offers the capability to beat Horner's Method. The enhanced Cube-fx reports lower execution time when the number of evaluated functions is larger than about 4, similar results to the saturated Matrix MAC results in Sec. 4.4.2. However, on Ascend 910B processors, the enhanced Cube-fx cannot work as powerfully as on Ascend 310 processors. The reason is that Horner's Method works so well on Ascend 910B processors, as discussed in Sec. 4.4.2, that Cube-fx can only outperforms when k is large enough.

4.4.4 Case Study: Audio Signal Feature Extraction

In this subsection, we demonstrate how Cube-fx enhances the efficiency of feature extraction for real applications in audio signal processing [56]. One-dimensional audio signals can be transformed using the Fourier transform

Table 4.2: Sample features of audio signal

Feature	**Taylor Expansion of** $x(t) = \cos(t)$
Instantaneous Amplitude	$A(t+\Delta t) = \lvert\cos(t)\rvert - \sin(t)\Delta t - \cos(t)\Delta t^2/2 + \sin(t)\Delta t^3/6 + \cos(t)\Delta t^4/24 + \ldots$
Log-Amplitude	$L(t+\Delta t) = \log(\lvert\cos(t)\rvert) - \tan(t)\Delta t - \Delta t^2/(2\cos^2(t)) + \tan(t)\Delta t^3/(3\cos^2(t)) - \Delta t^4/(4\cos^4(t))\ldots$
Instantaneous Power	$P(t+\Delta t) = \cos^2(t) - 2\cos(t)\sin(t)\Delta t - \sin^2(t)\Delta t^2 + 2\cos(t)\sin(t)\Delta t^3/3 + \sin^4(t)\Delta t^4/24 + \ldots$
Instantaneous Gradient	$G(t+\Delta t) = -\sin(t) - \cos(t)\Delta t + \sin(t)\Delta t^2/2 + \cos(t)\Delta t^3/6 - \sin(t)\Delta t^4/24 + \ldots$
Short-Time Autocorrelation	$R(\tau+\Delta t) = \sum_{t=0}^{T-1}\cos(t)\cdot (\cos(t+\tau) - \sin(t+\tau)\Delta t - \cos(t+\tau)/2\Delta t^2 + \cdots)$

to represent them as trigonometric functions, enabling Taylor expansion to evaluate the impact of small perturbations. For simplicity, our experiments directly use basic cosine signals as inputs. Table 4.2 presents several sample features used in the experiment. For features requiring multiple input values, such as Short-Time Autocorrelation, we employ a sliding window approach to generate input vectors from the original data for each window. All input data sizes are set to 16M for evaluation consistency.

Fig. 4.13 presents the evaluation results for feature extraction in audio signal processing. Beacuse of the input vector generation required in this application, Cube-fx demonstrates slightly lower acceleration compared to previous experiments. Across the three experiments, the average speedup over Horner's Method is 1.70× on the Ascend 310 processor and 0.89× on the Ascend 910B processor. Specifically, while the previous benchmarking results indicated nearly identical performance for extracting 7 features in both implementations, the real application results reveal a noticeable performance drop caused by the input vector generation. These findings reaffirm

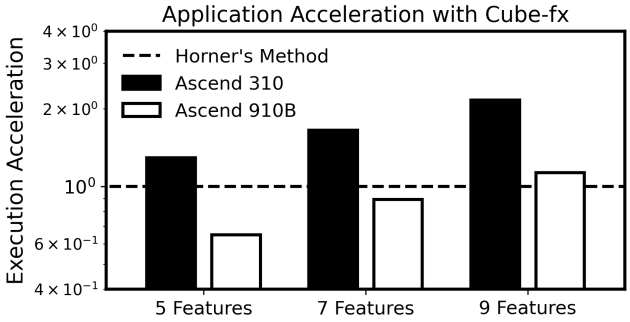


Figure 4.13: Feature extraction with Cube-fx acceleration in audio signal processing

our previous analysis in Sec 4.4.2: Cube-fx, with its reliance on extensive communication, performs efficiently on the Ascend 310 processor but faces challenges on the Ascend 910B processor.

4.5 Conclusion

This chapter introduced Cube-fx, a novel algorithm to map Taylor expansion for function evaluation onto the powerful Matrix MACs of the AI processors, especially the Huawei Ascend processors we focus on in this paper. Based on the fact that the AI processors usually equip poor vector units and a piece of data often needs to be computed by multiple functions in real-world applications, Cube-fx converts the building and computation of Taylor polynomials to specific matrix multiplications supported by the Matrix MACs. Since the Matrix MACs must complete a fixed shape of matrix multiplication each time, Cube-fx wastes a part of the computation power when the order number of Taylor polynomial is low. Therefore, we propose another further mapping technology to fuse the matrix multiplications of Cube-fx, which again empowers Cube-fx and exploits the computation power as much as possible. Although the application and performance of Cube-fx are still imperfect, our attempt to extend the programmability of the AI processors

would inspire more researchers to work on this novel hardware with more valuable results.

Chapter 5

Conclusion and Future work

This thesis explores a specific type of AI processor, the Huawei Ascend, by proposing two hardware-aware optimization solutions tailored to its architecture. Through detailed benchmarking, we analyze hardware characteristics and develop a performance model that provides insights into its operation (Chapter 2). Subsequently, two main optimization strategies are introduced: (1) replacing inefficient scalar or vectorized operations with alternative implementations (Chapter 3), and (2) using matrix multiplications as substitutes for scalar or vectorized operations (Chapter 4). These findings offer an in-depth examination of current AI processors, shedding light on their strengths and limitations, opportunities for algorithmic optimizations, and directions for future hardware improvements. The primary results and contributions of this work are as follows:

First, an accurate performance model is constructed based on detailed dissections of the Huawei Ascend processors. Alongside general hardware unit benchmarks, the analysis focuses specifically on bus contention, which significantly influences the runtime behavior of IO operations. Using the collected hardware parameters, the performance model, named Verrocchio, takes program source code as input to accurately predict execution time. Benchmark results also highlight a key design trade-off in AI processors: optimized matrix multiplication performance comes at the

expense of scalar and vector operation efficiency.

This trade-off presents a significant challenge when migrating algorithms from other platforms. While AI processors substantially accelerate matrix multiplications, full AI applications generally require other essential operations, such as scalar or vector operations for dynamic addressing or activations. Thus, two main approaches are proposed to alleviate these bottlenecks.

(1) The first approach replaces inefficient scalar and vector operations with alternative implementations. Taking SelB-k-NN as an example, we aim to replace the most computationally expensive scalar operations and vector comparisons & selections with optimized vectorized operations on Huawei Ascend processors. This approach not only replaces inefficient operations but also enhances the programmability of AI processors. For instance, on AI processors lacking specific `vecCmp` and `vecSel` hardware support, this approach suggests a `ReLU`-based bitwise algorithm as a substitute.

This approach delivers significant performance gains in our experiments. However, it does not leverage the primary hardware feature of AI processors, the Matrix MACs. Consequently, it could be applied across different platforms following extensive benchmarking. To address this, we propose the second approach, which is specifically optimized for AI processors.

(2) The second approach maps scalar and vector operations to matrix multiplications on Matrix MACs. The Cube-fx method exemplifies this approach, utilizing the Matrix MACs for computations that do not typically involve matrix multiplication. Since matrix multiplications inherently include additional mapping and reduction steps compared to vector operations, Cube-fx applies this feature to map single inputs across multiple functions for Taylor expansion and reduce each multiplication result to produce the final polynomial output.

Compared to the first approach, this second approach fully utilizes the Matrix MACs, leading to substantial performance gains specific to AI processor hardware. However, it is applicable to a more limited range of appli-

cations. The primary challenge lies in the Matrix MACs' restricted functionality, as they only perform multiply-and-add operations. While some applications, such as Cube-fx proposed in this thesis, can be adapted to these operations, others requiring more complex operations (e.g., logical operations, exponentiation) cannot be easily transformed. Therefore, both approaches are necessary for algorithm optimizations on the current AI processors. Together, these two strategies provide a comprehensive solution that addresses both general-purpose and MAC-specific optimization needs.

Furthermore, at the software level, while traditional optimization methods (e.g., instruction scheduling or data reuse) remain valuable, further software optimizations on current AI processor architectures are expected to be feasible only for specific, narrow application cases. Future work should therefore prioritize the evolution of next-generation hardware architectures to achieve broader and more effective optimizations.

A promising direction involves advancements in the architecture of Matrix MACs. Initial work by Zhang et al. [100] has begun to address this by proposing a novel hardware unit to perform operations of the form $(A \odot B \oplus C)$. This unit allows programmers to customize operations, selecting specific $\odot$ and $\oplus$ operations suited to their algorithms. While this is a significant step, more complex applications will require core computations that go beyond two binary operations. Thus, our future work will focus on enhancing Matrix MACs within next-generation AI processors to support higher degrees of customization and programmability, enabling more complex and varied algorithmic implementations. Such advancements are expected to expand the range of AI applications that can be efficiently supported, further pushing the boundaries of AI processor capabilities.

In addition, a more accurate performance modeling framework beyond traditional Big O notation will be explored. Since Big O notation does not capture low-level hardware behavior or instruction-level inefficiencies, an extension of the Roofline model [64] is proposed. This extended model will incorporate multiple rooflines representing different hardware units (e.g.,

vector units, scalar units, memory bandwidth), offering a more detailed and architecture-aware performance visualization. With such a model, it will be possible to analyze algorithms not only by their theoretical complexity, but also by their interaction with heterogeneous hardware resources, thereby guiding software, hardware co-optimization more effectively.

Bibliography

[1] Marinka Zitnik, Francis Nguyen, Bo Wang, Jure Leskovec, Anna Goldenberg, and Michael M Hoffman. Machine learning for integrating data in biology and medicine: Principles, practice, and opportunities. *Information Fusion*, 50:71–91, 2019.

[2] André Gensler, Janosch Henze, Bernhard Sick, and Nils Raabe. Deep learning for solar power forecasting—an approach using autoencoder and lstm neural networks. In *2016 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 002858–002865. IEEE, 2016.

[3] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[4] Emil Talpes, Douglas Williams, and Debjit Das Sarma. Dojo: The microarchitecture of tesla's exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, 2022.

[5] Norman P. Jouppi, George Kurian, Sheng Li, Peter C. Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clif Young, Xiang Zhou, Zongwei Zhou, and David A. Patterson. TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In Yan Solihin

and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 82:1–82:14. ACM, 2023.

[6] AMD. Amd instinct mi300x accelerators. `https://www.amd.com/en/products/accelerators/instinct/mi300.html`. Accessed: 2024-01-07.

[7] Nvidia. Nvidia h100 tensor core gpu architecture. `https://resources.nvidia.com/en-us-tensor-core`. Accessed: 2023-12-13.

[8] Jian Ouyang, Xueliang Du, Yin Ma, and Jiaqiang Liu. Kunlun: A 14nm high-performance AI processor for diversified workloads. In *IEEE International Solid-State Circuits Conference, ISSCC 2021, San Francisco, CA, USA, February 13-22, 2021*, pages 50–51. IEEE, 2021.

[9] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, August 18-20, 2019*, pages 1–44. IEEE, 2019.

[10] Huawei. Ascend-910b. `https://item.jd.com/100073373192.html`. Accessed: 2023-12-13.

[11] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[12] Gaojie Jin, Xinping Yi, Dengyu Wu, Ronghui Mu, and Xiaowei Huang. Randomized adversarial training via taylor expansion. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 16447–16457. IEEE, 2023.

[13] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, et al. In-datacenter performance

analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.

[14] Huawei. Cann - a unified heterogeneous compute architecture. https://www.hiascend.com/software/cann. Accessed: 2023-10-23.

[15] Caio S. Rohwedder, João P. L. de Carvalho, José Nelson Amaral, Guido Araújo, Giancarlo Colmenares, and Kai-Ting Amy Wang. Pooling acceleration in the davinci architecture using im2col and col2im instructions. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*, pages 46–55. IEEE, 2021.

[16] Zhuoran Ji and Cho-Li Wang. Accelerating DBSCAN algorithm with AI chips for large datasets. In Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen, editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 51:1–51:11. ACM, 2021.

[17] Qualcomm. Snapdragon 888 ai presentation. https://www.qualcomm.com/media/documents/files/snapdragon-888-ai-presentation.pdf. Accessed: 2021-10-02.

[18] cambricon. Mlu290-m5 (in chinese). https://www.cambricon.com/index.php. Accessed: 2021-12-10.

[19] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Diannao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, 2016.

[20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh,

Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[21] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[23] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent dif-

fusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10674–10685. IEEE, 2022.

[24] Jonas Oppenlaender. The creativity of text-to-image generation. In *25th International Academic Mindtrek conference, Academic Mindtrek 2022, Tampere, Finland, November 16-18, 2022*, pages 192–202. ACM, 2022.

[25] OpenAI. Improving image generation with better captions. https://cdn.openai.com/papers/dall-e-3.pdf. Accessed: 2023-11-01.

[26] Rui Xu, Sheng Ma, Yang Guo, and Dongsheng Li. A survey of design and optimization for systolic array-based DNN accelerators. *ACM Comput. Surv.*, 56(1):20:1–20:37, 2024.

[27] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Ke-ren Zhou, and Mingyu Chen. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 31–43. ACM, 2017.

[28] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[29] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. Modeling deep learning accelerator enabled gpus. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019*, pages 79–92. IEEE, 2019.

[30] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, et al. Learning to optimize tensor programs. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman,

Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3393–3404, 2018.

[31] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, and Mike Burrows. A learned performance model for the tensor processing unit. *CoRR*, abs/2008.01040, 2020.

[32] GCC. Accelerating ml workloads using new gcc builtins. `https://lpc.events/event/7/contributions/719/attachments/524/1105/LPC_GNU_builtins.pdf`. Accessed: 2024-03-05.

[33] Intel. Oneapi. `https://www.oneapi.io/`. Accessed: 2024-03-05.

[34] Nicholai Tukanov, Rajalakshmi Srinivasaraghavan, José E. Moreira, and Tze Meng Low. Modeling matrix engines for portability and performance. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1173–1183, 2022.

[35] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.

[36] Roberto Carrasco, Raimundo Vega, and Cristobal A. Navarro. Analyzing GPU tensor core potential for fast reductions. In *37th International Conference of the Chilean Computer Science Society, SCCC 2018, Santiago, Chile, November 5-9, 2018*, pages 1–6. IEEE, 2018.

[37] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-Mei W. Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, pages 46–57. ACM, 2019.

[38] Hao Tang, Kazuhiko Komatsu, Masayuki Sato, and Hiroaki Kobayashi. An efficient skinny matrix-matrix multiplication method by folding input matrices into tensor core operations. In *Eighth International Symposium on Computing and Networking Workshops, CANDAR 2020 Workshops, Naha, Japan, November 24-27, 2020*, pages 164–167. IEEE, 2020.

[39] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. *IEEE Access*, 10:20616–20632, 2022.

[40] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. Tcudb: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1360–1374, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Qualcomm. Qualcomm ai engine direct. https://docs.qualcomm.com/bundle/publicresource/topics/80-63442-50/SupportedOps.html. Accessed: 2023-11-01.

[42] Google. Pytorch/xla. https://github.com/pytorch/xla. Accessed: 2023-11-01.

[43] Cambricon. Dev platform. http://devplatform.hengqinai.com:30080/server. Accessed: 2023-11-01.

[44] Cambricon. Bang forum. https://forum.cambricon.com/list-33-1.html. Accessed: 2023-11-01.

[45] Nvidia. Nvidia ada gpu architecture. https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf. Accessed: 2023-11-01.

[46] NVIDIA. Parallel thread execution isa version 7.4. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2023-11-01.

[47] Baidu. Baidu console. https://console.bce.baidu.com/bml-app/overview. Accessed: 2023-11-01.

[48] Ke-ren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. A performance analysis framework for exploiting GPU microarchitectural capability. In William D. Gropp, Pete Beckman, Zhiyuan Li, and Francisco J. Cazorla, editors, *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, pages 15:1–15:10. ACM, 2017.

[49] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. GPU code optimization using abstract kernel emulation and sensitivity analysis. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 736–751. ACM, 2018.

[50] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John M. Mellor-Crummey. GPA: A GPU performance advisor based on instruction sampling. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 115–125. IEEE, 2021.

[51] Salli Moustafa. Accelerating sparse matrix-matrix multiplication with the ascend ai core. In *AccML, 2023*, pages 50–51. HiPEAC, 2023.

[52] Huawei. Atlas 200 ai accelerator module 1.0.0 application software development guide. `https://support.huawei.com/enterprise/en/`

`doc/EDOC1100107776/d6ebdd49/operator-usage-suggestions`. Accessed: 2024-01-07.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[54] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1557–1570. ACM, 2018.

[55] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[56] John R Deller Jr, John G Proakis, and John H Hansen. *Discrete time processing of speech signals*. Prentice Hall PTR, 1993.

[57] Franco Manessi and Alessandro Rozza. Learning combinations of activation functions. In *24th International Conference on Pattern Recognition, ICPR 2018, Beijing, China, August 20-24, 2018*, pages 61–66. IEEE Computer Society, 2018.

[58] Cristina Hristea, Daniel Lenoski, and John S. Keen. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1997, November 15-21, 1997, San Jose, CA, USA*, page 45. ACM, 1997.

[59] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 529–540. USENIX Association, 2015.

[60] IBM. Sample-based profiling. https://www.ibm.com/docs/en/mondiag-tools?topic=perspective-sample-based-profiling. Accessed: 2022-07-14.

[61] DelphiTools. Sampling profiler. https://www.delphitools.info/samplingprofiler/. Accessed: 2022-07-14.

[62] Google. Profiling concepts. https://cloud.google.com/profiler/docs/concepts-profiling. Accessed: 2022-07-14.

[63] Edsger W. Dijkstra. Over seinpalen. http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF, n.d.

[64] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[66] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[67] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern*

Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018, pages 4510–4520. Computer Vision Foundation / IEEE Computer Society, 2018.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[69] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[70] Polychronis Velentzas, Michael Vassilakopoulos, and Antonio Corral. Gpu-based algorithms for processing the k nearest-neighbor query on disk-resident data. In *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings*, volume 12732 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2021.

[71] Kimikazu Kato and Tikara Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010, 17-20 May 2010, Melbourne, Victoria, Australia*. IEEE Computer Society, 2010.

[72] Kimikazu Kato and Tikara Hosino. Multi-gpu algorithm for k-nearest neighbor problem. *Concurr. Comput. Pract. Exp.*, 24(1):45–53, 2012.

[73] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Proceedings of the International Conference on Image Processing, ICIP 2010, September 26-29, Hong Kong, China*, pages 3757–3760. IEEE, 2010.

[74] Shengren Li, Lance Simons, Jagadeesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D. Owens, and Nina Amenta. kann on the GPU with shifted sorting. In *Proceedings of the EUROGRAPHICS Conference on High Performance Graphics 2012, Paris, France, June 25-27*, pages 39–47. Eurographics Association, 2012.

[75] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Trans. Big Data*, 7(3):535–547, 2021.

[76] Xiaoxin Tang, Zhiyi Huang, David M. Eyers, Steven Mills, and Minyi Guo. Efficient selection algorithm for fast k-nn search on gpus. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29*, pages 397–406, 2015.

[77] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. A coordinated tiling and batching framework for efficient GEMM on gpus. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 229–241. ACM, 2019.

[78] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 90–106. ACM, 2022.

[79] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multi-

plication. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 300–314. ACM, 2019.

[80] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 634–643. IEEE, 2020.

[81] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 278–291. ACM, 2021.

[82] Jie Zhao and Peng Di. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 427–441. IEEE, 2020.

[83] Antal van den Bosch and Ko van der Sloot. Superlinear parallelization of k-nearest neighbor retrieval. In *Proceedings of the 19th Belgian-Dutch Artificial Intelligence Conference*, pages 65–72. Citeseer, 2007.

[84] Google. Run a calculation on a cloud tpu vm by using jax. https://cloud.google.com/tpu/docs/run-calculation-jax. Accessed: 2023-10-23.

[85] Cambricon. Cambricon bang c developer guide. https://www.cambricon.com/docs/bangc/developer_guide_html/13BuiltinFuction/index.html. Accessed: 2023-10-23.

[86] Ahmad Basheer Hassanat, Mohammad Ali Abbadi, Ghada Awad Altarawneh, and Ahmad Ali Alhasanat. Solving the problem of the k parameter in the knn classifier using an ensemble learning approach. *arXiv preprint arXiv:1409.0919*, 2014.

[87] Haneen Arafat Abu Alfeilat, Ahmad BA Hassanat, Omar Lasassmeh, Ahmad S Tarawneh, Mahmoud Bashir Alhasanat, Hamzeh S Eyal Salman, and VB Surya Prasath. Effects of distance measure choice on k-nearest neighbor classifier performance: a review. *Big data*, 7(4):221–248, 2019.

[88] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. NVIDIA A100 tensor core GPU: performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[89] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, et al. Cambricon: An instruction set architecture for neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 393–405. IEEE Computer Society, 2016.

[90] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In R. M. Bennett, editor, *Papers presented at the 1960 western joint IRE-AIEE-ACM computer conference, IRE-AIEE-ACM 1960 (Western), San Francisco, California, USA, May 3-5, 1960*, pages 33–40. ACM, 1960.

[91] Theodore Samuel Motzkin. Evaluation of polynomials and evaluation of rational functions. *Bull. Amer. Math. Soc*, 61(163):10, 1955.

[92] Youngmi Joo and Nick McKeown. Doubling memory bandwidth for network buffers. In *Proceedings IEEE INFOCOM '98, The Conference on Computer Communications, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, Gateway to the*

21st Century, San Francisco, CA, USA, March 29 - April 2, 1998, pages 808–815. IEEE Computer Society, 1998.

[93] Intel. Intel® avx-512 instructions. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html`.

[94] Sanjeev Arora. Toward theoretical understanding of deep learning. `https://www.cs.princeton.edu/courses/archive/fall18/cos597G/lecnotes/lecture3.pdf`.

[95] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electron. Comput.*, 8(3).

[96] Hui Chen, Lin Jiang, Yuanyong Luo, Zhonghai Lu, Yuxiang Fu, Li Li, and Zongguang Yu. A cordic-based architecture with adjustable precision and flexible scalability to implement sigmoid and tanh functions. In *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020*, pages 1–5. IEEE, 2020.

[97] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[98] Arm. Arm neon. `https://www.arm.com/technologies/neon`.

[99] Huawei. Simd instruction set. `https://www.hikunpeng.com/document/detail/en/kunpenggrf/progtuneg/kunpengprogramming_05_0017.html`.

[100] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. Simd2: A generalized matrix instruction set for accelerating tensor computation beyond gemm. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 552–566, New York, NY, USA, 2022. Association for Computing Machinery.