现代多核 DSP 上的通用矩阵乘优化

Kainan Yu[1,§], Xinxin Qi[1,§], Peng Zhang[1,§], Jianbin Fang[1,*], Dezun Dong[1,*], Ruibo Wang[1,*],
Tao Tang[1], Chun Huang[1], Yonggang Che[1], and Zheng Wang[2]
1 *School of Computer Science and Technology*, *National University of Defense Technology*, Changsha, China
2 *School of Information Science and Technology*, *Northwest University*, Xi'an, China
{j.fang, dong, ruibo}@nudt.edu.cn

摘要—General Matrix Multiplication (GEMM) is a key subprogram in high-performance computing (HPC) and deep learning workloads. With the rising significance of power and energy consumption in HPC systems, accelerators based on Digital Signal Processors (DSPs) have been integrated into general-purpose HPC systems. Due to the architecture disparities, the GEMM optimization techniques used on conventional multi-core CPUs and GPGPUs are not always applicable to DSPs. This paper shares our experience in optimizing GEMM on multi-core GPDSPs, using a CPU-DSP processor as a case study. Our approach employs a range of techniques to optimize performance for DSP architectures. These include data partitioning, three-level pipelining, dedicated micro-kernel design, and improved vector reduction. These optimizations maximize the overlap between computation and communication while fully exploiting the capabilities of floating-point arithmetic units to achieve high performance. Our experimental results demonstrate that the performance attained by our optimization is up to 96% of the theoretical peak performance of the hardware.

摘要——通用矩阵乘 (GEMM) 是高性能计算 (HPC) 和深度学习应用中的关键子程序。随着能效和能耗逐渐成为 HPC 系统需要考虑的重要指标，基于数字信号处理器 (DSP) 的加速器被集成到了通用 HPC 系统中。由于体系结构的差异，用于常规的多核 CPU 和 GPGPU 的 GEMM 优化方法不总能适用于 DSP 平台。这篇文章以一个 CPU-DSP 处理器为例，分享了我们在 GPDSP 上优化 GEMM 的经验。我们的方法运用了一系列技巧来优化 DSP 架构的性能，包括数据分块，三层流水，专用的微内核设计，以及一种改进的向量规约。这些优化最大限度地重叠了计算和通信，并充分利用了浮点部件的算力。我们的实验结果表明，优化后的性能可以达到硬件峰值性能的 96

***Index Terms*—Matrix Multiplication, GPDSP, Performance Model, Performance Optimization**

关键词——矩阵乘，通用数字信号处理器，性能模型，性能优化

I. Introduction

General matrix multiplication (GEMM) is a crucial software component within high-performance computing (HPC) applications, spanning from traditional computational fluid dynamics (CFD) [1] to emerging workloads in deep learning [2]–[4]. Previous research efforts have delved into the optimization of GEMM for various platforms, including multi-core CPUs [5], [6] and GPGPUs [7]–[10].

通用矩阵乘法 (GEMM)[1] 是高性能计算 (HPC) 应用中的一个关键的软件组件。无论是传统的计算流体动力学 (CFD) [1]，还是深度学习的新兴工作负载 [2]–[4] 都需要频繁调用 GEMM。以往的研究工作已经深入讨论了 GEMM 在不同平台上的优化，包括多核 CPU [5], [6] 和 GPGPU [7]–[10]。

As power and energy consumption become increasingly critical concerns in HPC systems, there is a growing trend of integrating processor architectures based on digital signal processors (DSPs) into general-purpose HPC systems [11], [12]. These DSP-based processors incorporate very long instruction words (VLIW) and single instruction multiple data (SIMD) features, offering substantial opportunities to achieve both instruction-level and data-level parallelism. Unlike CPUs and GPUs, DSPs typically lack an out-of-order execution pipeline, making most optimizations developed for CPU and GPU no longer applicable. While prior research efforts have focused on optimizing general matrix multiplication [11], [13], currently there is no consensus on the most effective optimization strategies for this kernel on modern multi-core DSPs.

随着功耗和能耗成为 HPC 系统中日益关注的问题，将基于数字信号处理器（DSP）的体系结构集成到通用 HPC 系统中的趋势越来越明显 [11], [12]。这些基于 DSP 的处理器包含了超长指令字（VLIW）和单指令多数据（SIMD）的特性，为实现指令级并行和数据级并行提供了机会。与 CPU 和 GPU 不同，DSP 通常不支持指令乱序执行，这使得为 CPU 和 GPU 开发的大多数优化方法不再适用。虽然以前的研究工作已经致力于优化通用矩阵乘法 [11], [13]，但目前还没有一个公认的最有效的面向现代多核 DSP 的优化策略。

[1]A GEMM is defined as $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are input matrices, α and β are scalar factors, and $\mathbf{C}$ is a matrix overwritten by the output. In this work, we follow the naming convention of linear algebra libraries, where matrix $\mathbf{A}$ is a $M \times K$ matrix with M rows and K columns, matrix $\mathbf{B}$ is sized $K \times N$, and matrix $\mathbf{C}$ is sized $M \times N$.

GEMM 的定义是 $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$，其中 $\mathbf{A}$ 和 $\mathbf{B}$ 是输入矩阵，α 和 β 是标量因子，$\mathbf{C}$ 是被输出覆盖的矩阵。在本文中，我们遵循线性代数库的命名惯例，其中矩阵 A 是一个 $M \times K$ 矩阵，有 M 行和 K 列，矩阵 $\mathbf{B}$ 的大小是 $K \times N$，矩阵 $\mathbf{C}$ 的大小是 $M \times N$。

[§]Equal contribution *Corresponding author

This paper shares our experience of optimizing GEMM on multi-core general-purposed DSPs (GPDSPs) by taking the FT-M7032 CPU-DSP platform as a case study (see Section II-A). The FT-M7032 DSP implements a sophisticated VLIW instruction set with 1024-bit SIMD components and features a hierarchical on-chip scratchpad memory. Like other DSPs, this memory is bare metal, which requires careful software management. Based on these typical features, optimization techniques developed for the FT-M7032 can be applied to many other DSPs of its like [11], [12], [14], [15].

本文分享了我们在多核通用 DSP（GPDSP）上优化 GEMM 的经验，以 FT-M7032 CPU-DSP 平台作为案例来研究（见第 II-A 节）。FT-M7032 DSP 实现了一个复杂的 VLIW 指令集，搭配 1024 位的 SIMD 部件，并具有层次化的片上存储器。与其他 DSP 架构类似，片上缓存需要软件来管理。基于这些典型的特性，面向 FT-M7032 的 GEMM 优化技巧也可以应用到其他多种 DSP 平台上 [11], [12], [14], [15]。

We have identified two key limitations of prior work in optimizing GEMM for DSPs: the inefficient use of the memory hierarchy and the SIMD components. On the one hand, modern GPDSPs often integrate a memory system consisting of off-chip DDR RAM shared across different DSP cores, and on-chip multi-level scratchpad memories managed by each processor. Despite previous efforts to minimize memory access through dedicated data partitioning and optimize the communication timing with the help of multi-buffer strategies [13], these approaches fail to achieve a perfect match between the GEMM algorithm and the specific hardware, leading to a suboptimal allocation of memory resources. On the other hand, while GPDSPs often provide SIMD components to accelerate floating-point operations, prior works incur unnecessary overhead of vector reduction [16], which is expensive in compute-intensive kernels like GEMM. Even though fully utilizing the vectorized floating-point arithmetic units, the reduction of long vectors extends the execution time of micro-kernel and significantly impedes the overall performance of GEMM.

已有的面向 DSP 的 GEMM 优化工作存在两个局限：多层次片上缓存和 SIMD 部件的低效利用。一方面，现代 GPDSP 通常集成了由不同 DSP 核共享的片外 DDR RAM 和由每个处理器单独管理的片上多级缓存。尽管以前的工作通过专门设计的数据划分方式降低了访存量，并利用多缓冲策略优化了数据通信的时序 [13]，但这些方法未能在 GEMM 算法和特定硬件之间实现完美的匹配，导致了存储资源的非最优分配。另一方面，虽然 GPDSP 通常提供 SIMD 部件来加速浮点运算，但先前的工作引入了不必要的向量归约开销 [16]，这在 GEMM 这类计算密集型算子中代价高昂。尽管充分利用了向量化的浮点算术单元，但长向量归约延长了微核的执行时间，从而显著降低了 GEMM 的整体性能。

Our work aims to address the two aforementioned drawbacks when optimizing GEMM on GPDSPs. To take advantage of the hierarchical memory, we revise the classical Goto algorithm [17] for GEMM with a multi-level pipelining strategy. It reduces memory access overhead by overlapping data movement with computation. We then develop a data partitioning method to divide the large matrix into small sub-matrices, optimizing memory access patterns and reducing data loaded from the off-chip DDR memory. Building upon the multi-level pipeline, we introduce high-performance micro-kernels with an efficient vector reduction approach specifically designed to leverage the SIMD capabilities. Our goal is to optimize the utilization of arithmetic units and minimize the overhead of related operations. To achieve this, we employ a series of analytical models to design the optimal shape of GEMM micro-kernels and their entire instruction pipelines.

我们的工作旨在解决在 GPDSP 上优化 GEMM 时遇到的上述两个不足。为了利用层次化存储结构，我们采用改进自经典 Goto 算法 [17] 的多级流水线策略。它通过将数据移动与计算重叠进行来降低访存开销。然后，我们开发了一种数据分块方法，将大矩阵划分为多个较小的子矩阵读入片上缓存，优化内存访问模式并减少从片外 DDR 内存加载的数据总量。在多级流水线的基础上，我们引入了高性能的微内核，采用了一种高效的向量归约方法，充分利用了平台的 SIMD 特性。我们的优化目标是提高算术单元的利用率并最小化相关操作的开销。为了实现这一目标，我们采用了一系列分析模型，用于来设计 GEMM 微核的最优形状和它们的整个指令流水线。

We evaluate our techniques on the FT-M7032 DSP platform with a 16-core ARM CPU and four multi-core GPDSP clusters each with 8 cores. Experimental results show that our techniques deliver up to 96% of the hardware peak performance.

我们在 FT-M7032 DSP 平台上评估了我们的优化技术，该平台具有 16 核 ARM CPU 和四个 DSP 簇，每个簇有 8 个 DSP 核。实验结果表明，经过优化以后，GEMM 算子的实际性能达到了硬件峰值性能的 96%。

This paper makes the following contributions:

1) It introduces a new multi-level pipeline algorithm to utilize the memory hierarchy of a typical multi-core GPDSP design (Section III);
2) It presents a high-performance GEMM micro-kernel to utilize the GPDSP vectorization components with an improved vector reduction strategy (Section IV);

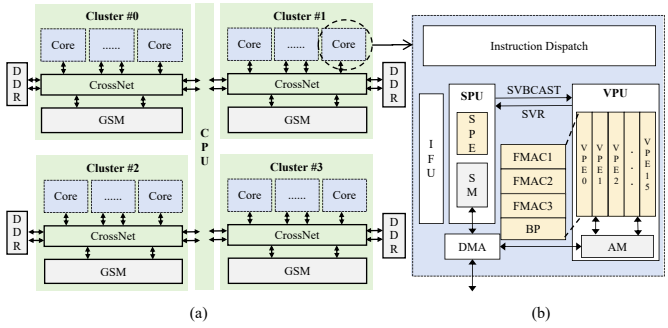


图 1. A high-level view of the FT-M7032 architecture (a) and the micro-architecture of a DSP core in FT-M7032 (b).

3) It demonstrates how analytical methods can be employed to automatically determine the key parameters of the GEMM kernel (Section III and Section IV).

本文做出了以下贡献：

1) 引入了一种新的多级流水线算法，用于 DSP 的多级存储结构 (Section III);
2) 提出了一种高性能的 GEMM 微内核，充分利用了 GPDSP 的向量化部件，以及一种改进的向量归约方法 (Section IV);
3) 展示了如何使用基于分析模型的方法来自动确定 GEMM 算子的关键参数 (Section III and Section IV).

II. Background and Motivation

This section introduces the FT-M7032 CPU-DSP architecture targeted in this work, the classical GEMM algorithm, and the observations which motivate our work.

这一章介绍了本文工作面向的硬件平台 FT-M7032，经典的 GEMM 算法，以及本文研究动机的来源。

A. FT-M7032 CPU-DSP Architecture

Our work targets a CPU-DSP heterogeneous platform (FT-M7032) with a 16-core ARMv8 general-purpose CPU and 4 multi-core GPDSP clusters, which are shown in Fig. 1(a). Each cluster includes 8 DSP cores, a global shared memory (GSM), and an off-chip DDR memory shared between the general-purpose CPU and the DSPs. The GSM is a software-managed memory, private to each cluster but shared among the 8 DSP cores within that cluster. The general-purpose CPU cores can access the entire DDR space across different DSP clusters, while the DSP cores can solely access the DDR within their respective acceleration cluster.

我们的工作是基于一个 CPU-DSP 异构平台（FT-M7032 ）进行的，它包含一个 16 核的 ARMv8 通用 CPU 和 4 个多核 GPDSP 簇，如图 1(a) 所示。每个簇包括 8 个 DSP 核心，一个全局共享内存（GSM），和一个在通用 CPU 和 DSP 之间共享的片外 DDR 内存。GSM 是一个软件管理的缓存，私有于每个簇，但在簇内由 8 个 DSP 核共享。通用 CPU 可以访问每个 DSP 簇的 DDR 空间，但是 DSP 核只能访问它们各自簇内的 DDR。

The CPU cores operate at a frequency of 2.0 GHz and have a double-precision floating-point peak performance of 140.8 GFlops. Each DSP core runs at 1.8 GHz with a double-precision floating-point peak performance of 172.8 GFlops, delivering a peak performance of 1.38 TFlops for a cluster with 8 DSP cores. Given that the four clusters run independently, we only focus on optimizing GEMM on an individual cluster.

CPU 核以 2.0 GHz 的时钟频率运行，双精度浮点峰值性能为 140.8 GFlops。每个 DSP 核以 1.8 GHz 的时钟频率运行，双精度浮点峰值性能为 172.8 GFlops。一个包含 8 个 DSP 核的簇可提供 1.38 TFlops 的峰值算例。考虑到四个簇是独立运行的，因此我们只关注如何在单个簇上优化 GEMM。

Fig. 1(b) shows the micro-architecture of a DSP core, which implements the VLIW architecture. It includes an instruction scheduling unit (IFU), a scalar processing unit (SPU), a vector processing unit (VPU), and a DMA engine. The IFU can issue 11 instructions per cycle, including 5 scalar instructions and 6 vector instructions. The SPU contains an SPE (Scalar Processing Element) and a SM (Scalar Memory) of 64KB. It is designed for instruction flow control and scalar computations. The VPU has a 768KB AM (Array Memory) and 16 VPEs (Vector Processing Elements) that work in a SIMD fashion. Each VPE has 64 64-bit registers, 3 floating-point multiply-add units (FMAC), one BP unit, and two Load/Store vector units. Data transfers between SPU and VPU via the scalar broadcast instruction (SVBCAST) and the scalar-vector register (SVR). Physical DMA engines are used to move data among different levels of memory.

图 1(b) 显示了单个 DSP 核的内部结构。它包含一个指令调度单元（IFU）、一个标量处理单元（SPU）、一个向量处理单元（VPU）和一个 DMA 引擎。IFU 每个周期可以发射 11 条指令，包括 5 条标量指令和 6 条向量指令。SPU 包括一个 SPE（标量处理元件）和一个 64KB 的 SM（标量片上缓存）。它是为指令流控制和标量计算而设计的。VPU 包括一个 768KB 的 AM（向量片上缓存）和 16 个以 SIMD 方式工作的 VPE（向量处理元件）。每个 VPE 有 64 个 64 位寄存器，3 个浮点乘加运算单元（FMAC），一个 BP 单元和两个 Load/Store 单元。SPU 和 VPU 之间通过标量广播指令（SVBCAST）和标向量寄存器（SVR）进行数据传输。DMA 引擎用于在不同层次的内存之间搬运数据。

We note that the DSP cluster of FT-M7032 is a bare-metal machine, and thus it is of significant challenge for compilers and programmers to tap the po-

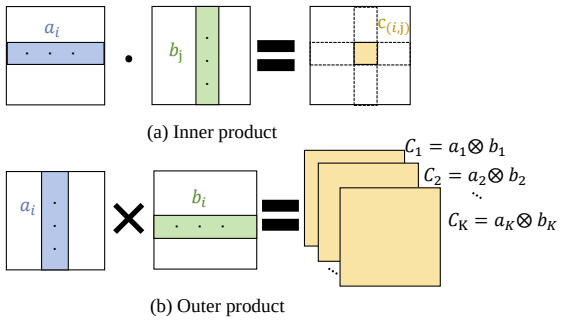


图 2. The inner product and outer product formulations.

tential. To alleviate the need for developers to directly interface with the underlying bare-metal hardware, a high-level heterogeneous multi-threaded programming model, referred to as *hthreads*, has been devised to enhance the programmability of the multi-core DSP [16]. Its primary objective is to manage the interactions between the CPU cores and DSP cores. By utilizing logical thread instances rather than directly addressing physical cores, programmers are afforded an intuitive approach to their programming tasks.

注意到，FT-M7032 的 DSP 簇是一台裸机，因此对于编译器和程序员来说，挖掘其潜力是一个巨大的挑战。为了减轻开发者直接与底层硬件交互的需求，一个高级的异构多线程编程模型，叫作 *hthreads*，被设计出来用于提高多核 DSP 的可编程性 [16]。它的主要目标是管理 CPU 核和 DSP 核之间的交互。通过使用逻辑线程实例而不是直接操作物理核心，程序员可以使用一种更直观的编程方法。

B. GEMM Algorithm

The GEMM algorithm can be divided into the inner product approach and the outer product approach, based on the loop index traversal order. Fig. 2(a) shows the process of computing dot products using the inner product formulation, which involves multiplying corresponding elements of two vectors and summing the products to obtain a scalar value. Fig. 2(b) shows that the outer product approach calculates the cross-product of two vectors, resulting in a matrix.

GEMM 算法根据循环索引的遍历顺序可以分为内积方法和外积方法。图 2(a) 显示了使用内积公式计算点积的过程，可概括为将两个向量的对应元素相乘并求和，得到一个标量数值。图 2(b) 显示了使用外积方法计算两个向量叉积的过程，结果是得到一个矩阵。

Our GEMM micro-kernels are built upon the outer product approach for two reasons. First, it minimizes the need for vector reduction. As for the inner product approach, multiplication of a row vector from **A** and a column vector from **B** results in a new vector, which is then reduced to a scalar value through SVR. Given that each vector register of FT-M7032 stores 1024-bit data, such a reduction operation consumes at least 16 cycles, accounting for a large overhead for the micro-kernel. With the outer product approach, the need for vector reduction can be eliminated in some cases. The second reason is that using the outer product approach takes full advantage of DSP's long vector components. In the normal case without the need for matrix transposition, the row-major storage order perfectly aligns with the vector direction of submatrices **B** and **C**, unleashing the potential of SIMD parallelism. To summarize, with the outer product approach, both avoiding vector reduction and increasing data-level parallelism can be achieved.

我们的 GEMM 微内核基于外积方法进行设计，原因有二。首先，它尽可能规避了向量规约操作。对于内积方法，A 中的行向量和 B 中的列向量相乘得到一个新的向量，然后需要借助 SVR 寄存器归约为一个标量值。考虑到 FT-M7032 的每个向量寄存器存储 1024 位的数据，这样的归约操作至少需要消耗 16 个周期，为微核带来了很大的开销。而使用外积方法则可以在某些情况下省去向量规约步骤。第二个原因是，使用外积方法可以充分利用 DSP 的长向量部件。在不需要矩阵转置的情况下，行优先的存储顺序与子矩阵 B 和 C 的向量方向完美重合，释放了 SIMD 特点带来的并行潜力。总之，使用外积方法，既可以避免向量归约，又可以增加数据级并行性。

C. Motivation

We have observed two issues of optimizing GEMM on DSPs which motivate our work.

我们的研究动机来自于现有的 DSP 矩阵乘优化工作的两点不足。

1) Incomplete computation-communication overlap: In this article, communication refers to data movement between the main memory (DDR) and the on-chip scratchpad memory. The previous approach [13] uses two buffers for matrix **C**, resulting in incomplete overlap between computation and communication. To address this limitation, we propose the adoption of three buffers for matrix **C**. This is because matrix **C** is related to both read and write operations. With two buffers, only two out of the three stages (read, write and compute) can execute in parallel, as shown in Fig. 3. The overhead of memory access can not be fully hidden. By using three buffers, bubbles between adjacent compute stages disappear. While the data of the current step are being computed, the results of the previous step are stored back, and the data for the next step are simultaneously loaded. This method achieves full overlap between computation and data movement.

在本文中，通信指的是主存（DDR）和片上缓存之间的数据搬运。现有的方法 [13] 仅为矩阵 C 分配了两个片上缓冲区，导致计算和通信之间的无法完全重叠。为了解决这个不足，我们提出了为矩阵 C 分配三个片

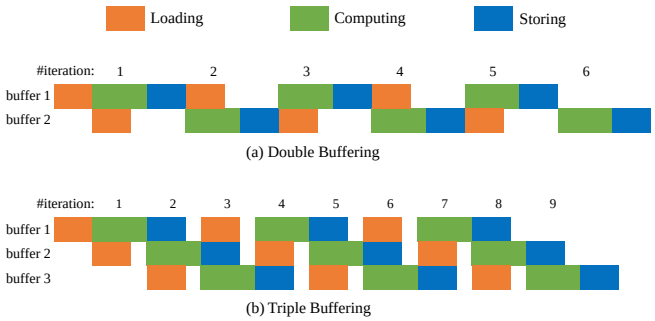


图 3. Comparison of the impact of double buffering and triple buffering on computation-communication overlap.

上缓冲区的方法。这是因为矩阵 C 与读写操作都有关。使用两个缓冲区时，三个阶段（读、写和计算）只有其中的两个可以并行执行，如图 3 所示，访存开销因此无法完全隐藏。通过使用三个缓冲区，相邻计算阶段之间的气泡会消失。在当前迭代步的数据被计算的同时，前一步的结果将被写回，同时下一步要用到的数据会被加载。这种方法实现了计算和数据移动之间的完全重叠。

2) Inefficient vector reduction method: The conventional vector reduction on FT-M7032 is implemented with SVR. This process involves writing vectors into SVR, individually reading each element of SVR into scalar registers (with each vector taking up 16 cycles), performing scalar addition, and finally writing the results back to the on-chip memory (requiring an additional 14 cycles). Our empirical results have shown that even with full pipelining, reducing n vectors requires at least $3 + 16n + 14$ cycles. Based on this approach, a micro-kernel with a shape of 6×512×8 can only reach 39% of the hardware peak performance. Thus, we argue for a better vector reduction method that consumes less cycles.

FT-M7032 上传统的向量归约是借助 SVR 实现的。这个过程包括把向量写入 SVR，将 SVR 的每个元素分别读入标量寄存器（每个向量占用 16 个周期），执行标量加法，最后将结果写回片上缓存（需要额外的 14 个周期）。经验结果表明，即使做到了完全流水化的指令执行，归约 n 个向量也至少需要 3+16n+14 个周期。基于这种方法，一个形状为 6×512×8 的微内核只能达到硬件峰值性能的 39%。因此，我们希望提出一种开销更小的长向量规约方法。

III. Optimizing General Matrix Multiplication

A. Overview

Fig. 4 gives a high-level overview of our GEMM implementation for computing $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$, based on the Goto GEMM algorithm [17], [18] used by mainstream linear algebra libraries, including OpenBLAS [19] and BLIS [20]. Our algorithm partitions matrices **A**, **B** and **C** into submatrices (Fig. 5) and moves them to specified on-chip memory locations.

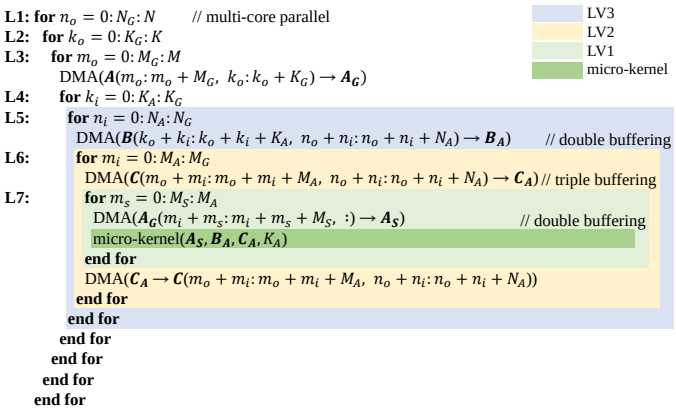


图 4. Our GEMM implementation for computing $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$. The matrix data locations are denoted by subscripts G, S and A, representing GSM, SM and AM, respectively. Variables M_*, N_* and K_* indicate the sizes of submatrices in their respective locations.

Then, a manually implemented micro-kernel is invoked to perform computation. The data partitioning aims to maximize data reuse and on-chip cache locality, with DMA transfers cleverly hidden within a three-level, multi-stage pipeline. The micro-kernel aims to fully exploit the computational capabilities of the FMAC units. These optimization strategies collectively ensure the high performance of our GEMM implementations. The implementation of $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B}^T + \beta\mathbf{C}$ (i.e. the NT mode) is almost identical to that of Fig. 4, except that the storage locations of submatrices are changed.

图 4 给出了我们基于 Goto GEMM 算法 [17], [18] 计算 $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$ 的程序流程。Goto 算法被包括 OpenBLAS [19] 和 BLIS [20] 在内的主流线性代数库采用。我们的算法将矩阵 **A**, **B** 和 **C** 划分为子矩阵（图 5）并将它们移动到指定的片上缓存位置。然后，调用一个手工实现的微核来执行计算。数据划分旨在最大化数据重用率和片上缓存局部性，并在一个三级、多阶段的流水线中巧妙地隐藏 DMA 传输开销。微核的目标是充分利用 FMAC 单元的浮点算力。这些优化策略共同确保了我们的 GEMM 可以实现高性能。$\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B}^T + \beta\mathbf{C}$（即 NT 模式）的实现与图 4 的流程几乎相同，只是子矩阵的缓存位置发生了变化。

The workflow of our GEMM implementation for computing $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$ is as follows: To begin with, we partition matrices **B** and **C** along the column direction (with the n_o as index) into submatrices of sizes $K \times N_G$ and $M \times N_G$, respectively. These blocks are distributed to different DSP cores for computation. For each core, we employ a block-wise approach. Specifically, matrices **A** and **B** are divided into column panels of size $M \times K_G$ and row panels of size $K_G \times N_G$, respectively (with the k_o as index). Matrix **A** is further partitioned along the row dimension into submatrices of size $M_G \times K_G$ and loaded to the A_G buffer in GSM. We assume that the submatrices of **A** always reside entirely in GSM. Panel B of size $K_G \times N_G$ is

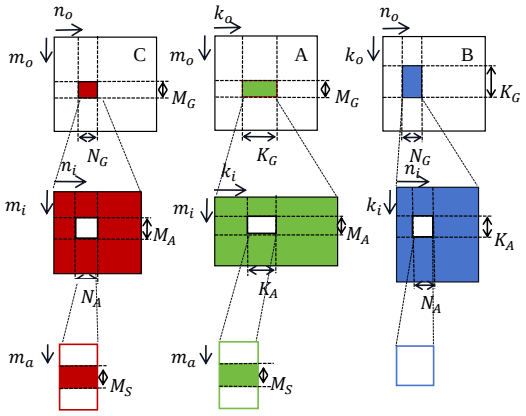


图 5. Data Partitioning.

divided into submatrices of size $K_A \times N_A$ and loaded to the B_A buffer in AM. Similarly, panel C, with dimensions $M_G \times N_G$, is partitioned into submatrices of size $M_A \times N_A$, which are then loaded to the C_A buffer in AM. We further divide the data of **A** in the A_G buffer into tiles of size $M_S \times K_A$, which are loaded to the A_S buffer in SM. Then, we call a dedicated micro-kernel, which will be detailed in Section IV. Once the micro-kernel computation is completed, the resultant submatrix of size $M_A \times N_A$ is stored back to the main memory.

我们计算 $\mathbf{C} = \alpha\mathbf{A} \cdot \mathbf{B} + \beta\mathbf{C}$ 的步骤如下：首先，沿着列维度（以 n_o 为索引）将矩阵 **B** 和 **C** 分割成大小为 $K \times N_G$ 和 $M \times N_G$ 的子矩阵，分配给不同的 DSP 核进行计算。对于每个 DSP 核，我们进一步进行数据切分。具体来说，矩阵 **A** 和 **B** 分别被划分为大小为 $M \times K_G$ 和 $K_G \times N_G$ 子矩阵（以 k_o 为索引）。**A** 沿着行维度进一步切分成大小为 $M_G \times K_G$ 的子矩阵，并加载到 GSM 中的 A_G 缓冲区。我们假设 **A** 的子矩阵总是完全驻留在 GSM 中。大小为 $K_G \times N_G$ 的子矩阵 B 被切分为 $K_A \times N_A$ 的数据块，并加载到 AM 中的 B_A 缓冲区。类似地，大小为 $M_G \times N_G$ 的子矩阵 C 被切分为 $M_A \times N_A$ 的数据块，然后加载到 AM 中的 C_A 缓冲区。我们进一步将 A_G 缓冲区中数据切分成大小为 $M_S \times K_A$ 的数据块，加载到 SM 中的 A_S 缓冲区。然后，我们调用一个专门设计并优化的微内核，它将在第 IV 节中被详细介绍。一旦微核计算完成，大小为 $M_A \times N_A$ 的结果子矩阵就立即被写回主存。

B. Manual Cache Management

FT-M7032 features a hierarchical on-chip scratch-pad memory, i.e. SM, AM and GSM, which we have to carefully manage to maximize the data locality of GEMM.

FT-M7032 具有层次化的片上缓存，分别是 SM、AM 和 GSM。为了最大化 GEMM 的数据局部性，必须精细地管理这些片上缓存。

First, we partition matrices **B** and **C** based on the number of DSP cores (P). Each DSP core handles N_G columns under the assumption of even divisibility of N by P.

首先，我们根据启动的 DSP 核数（P）切分矩阵 **B** 和 **C**。在假设 N 能被 P 整除的情况下，每个 DSP 核处理 N_G 列数据。

Second, directly accessing matrix **A**, which is shared by multiple DSP cores, will contend for the valuable DDR bandwidth. Thus, we first load a single copy of submatrix **A** into GSM, and then broadcast these data to the private cache of each core without occupying the bandwidth of DDR.

接着，考虑到直接访问由多个 DSP 核共享的矩阵 **A** 将会竞争有限的 DDR 带宽，我们首先将矩阵 **A** 的一部分数据加载到 GSM 中，再将 GSM 中的数据广播到每个核私有的缓存中，而不让重复的数据传输占用 DDR 带宽。

Third, we decide the allocation of space for sub-matrices within an individual DSP core. In the NN mode, when data are stored in row-major order, the outer product approach accesses contiguous memory addresses of matrices **B** and **C**, but discontinuous data units from matrix **A**. As long vector components treat contiguous 1024-bit data as an indivisible vector unit, we load submatrices of **B** and **C** into AM, while those of matrix **A** into SM.

然后，我们决定单个 DSP 核心中片上缓存的分配方式。在 NN 模式下，当数据以行优先的顺序存储时，外积方法除了访问矩阵 **B** 和 **C** 的连续内存地址，也访问矩阵 **A** 中不连续的数据单元。由于长向量组件将连续的 1024 位数据视为不可分割的向量单元，我们将 **B** 和 **C** 的子矩阵加载到 AM 中，而将 **A** 的子矩阵加载到 SM 中。

In the NT mode, we can take matrix $\mathbf{B}^T$ as **B** stored in column-major order, while matrix **A** is still stored in row-major order. Thus, we multiply a column of row-major vectors of **A** with a row of column-major vectors of **B**. The outcome of each vector-vector multiplication is accumulated and finally reduced to a scalar value. Accordingly, both submatrices of **A** and **B** should be placed into AM, while those of **C** should be moved to SM.

在 NT 模式下，我们可以将 $\mathbf{B}^T$ 矩阵视为以列优先的顺序存储的 **B** 矩阵，而矩阵 **A** 仍然以行优先的顺序存储。于是，我们将 **A** 的一列行优先的向量与 **B** 的一行列优先的向量相乘。每个向量-向量乘法的结果被累加，最后归约为一个标量值。因此，**A** 和 **B** 的子矩阵都应该放入 AM 中，而 **C** 的子矩阵应该暂存到 SM 中。

C. Three-level, Multi-stage Pipelining

Fig. 6 shows the design of our three-level, multi-stage pipeline, aiming to overlap the communication tasks with the computation tasks shown in Fig. 4.

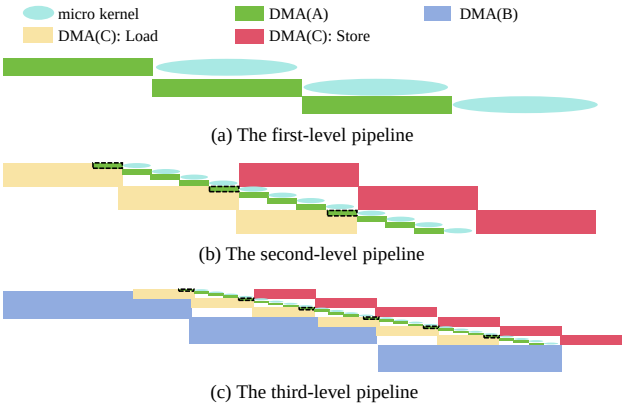


图 6. Three-level, multi-stage pipelining. The diagram does not depict the DMA transfers from the main memory to GSM, as they are considered an intrinsic and unavoidable part of the process.

图 6 显示了我们的三级多阶段流水线的设计，目的是将图 4中的计算任务和通信任务重叠进行。

1) The first-level pipeline: The first-level (LV1) pipeline consists of two stages: loading and computation. In the loading stage, a tile of **A** sizing $M_S \times N_A$ is moved from GSM to SM. In the computation stage (i.e. the micro-kernel), these data are fetched from SM into scalar registers and participate in computation. The specific procedure is detailed as follows. Two pre-allocated buffers (a_1 and a_2) are used to store the tiles of **A** in SM. After the first tile is loaded from GSM into a_1, the micro-kernel using these data is immediately launched. Meanwhile the next tile starts to be loaded into a_2. Then, computation is performed using data in a_2 as the next tile is loaded into a_1 again. This process repeats for a total of M_A/M_S times until all the tiles of **A** in GSM have been iterated. By overlapping the loading of tile **A** with the computation, the LV1 pipeline can effectively hide the memory access overhead.

第一级（LV1）流水线由两个阶段组成：数据加载和计算。在数据加载阶段，来自 **A** 矩阵大小为 $M_S \times N_A$ 的数据块被从 GSM 移动到 SM。在计算阶段（即微内核），这些数据被从 SM 读取到标量寄存器，并参与浮点运算。具体的过程如下。在 SM 中预先分配两个缓冲区（a_1 和 a_2）用于存储来自 **A** 矩阵的数据块。在第一块数据从 GSM 加载到 a_1 后，立即启动微内核开始计算。同时，下一块数据开始向 a_2 中加载。这之后，a_2 中的数据开始参与计算，同时下一块数据被加载到已经空闲的 a_1 中。这个过程会重复 M_A/M_S 次，直到 GSM 中的所有数据块都被遍历一轮。通过将 **A** 数据块的加载与计算相重叠，LV1 流水线可以有效地隐藏访存开销。

2) The second-level pipeline: The second-level (LV2) pipeline consists of three stages: loading, computation, and storing. During the loading stage, a submatrix of **C** sizing $M_A \times N_A$ is moved from the main memory to AM. In the computation stage, data of size $M_A \times N_A$ is fetched from AM, and all the operations of LV1 pipeline are performed. In the storing stage, the computed results are stored back into the main memory from AM. Note that unlike matrix **A**, which only involves read operations, accessing matrix **C** requires both read and write operations. When only two buffers are provided for matrix **C**, we can not achieve a full overlap of loading, storing and computation.

第二级（LV2）流水线由三个阶段组成：数据加载、计算和写回。在数据加载阶段，来自矩阵 **C** 大小为 $M_A \times N_A$ 的数据块被从主存加载到 AM 中。在计算阶段，程序将大小为 $M_A \times N_A$ 的数据块从 AM 取出，并执行 LV1 流水线的所有操作。在数据写回阶段，微内核的计算结果被从 AM 写回主存。注意到，与只涉及读操作的矩阵 **A** 不同，访问矩阵 **C** 既需要读操作，又需要写操作。若只为矩阵 **C** 分配两个缓冲区，我们就无法实现数据加载、存储和计算的完全重叠。

To this end, we propose a three-stage pipeline. Three specific buffers (c_1, c_2 and c_3) are pre-allocated in the AM. Fig. 6(b) shows that these buffers can be used in different stages simultaneously. For example, while the LV1 pipeline is executing using data in c_2, the computation results from the previous iteration in c_1 is stored back to the main memory. Meanwhile, data for the next iteration is being loaded into c_3. At the next time step, tasks of computation, storing and loading switch to c_3, c_2 and c_1, respectively. This design enables enhanced parallelism and fully hide the overhead of memory access.

为此，我们提出了一种三阶段的流水策略。在 AM 中预先分配三个缓冲区（c_1、c_2 和 c_3）。图 6(b) 显示了这些缓冲区可以在不同的阶段被同时使用。例如，当 c_2 中的数据被用于 LV1 流水线的执行时，c_1 中来自前一次迭代的计算结果会被写回主存。同时，下一次迭代需要的数据也正在被加载到 c_3 中。等到了下一个时间步，计算、存储和加载的任务就分别切换到了 c_3、c_2 和 c_1。这种设计实现了很高的并行性，并完全隐藏了内存访问的开销。

3) The third-level pipeline: The third-level (LV3) pipeline includes loading submatrices of B and executing the LV2 inner loop. During the loading stage, a submatrix of size $K_A \times N_A$ is moved from DDR to one of the b_1, b_2 buffers in AM. Then, these data are fetched by the micro-kernel during the execution of LV2 pipeline. To minimize the communication overhead, we apply the same operations as before by rescheduling the loading stage to an earlier time step. Similar to the LV1 pipeline, the primary focus here lies on the scenario where the time spent in LV2 equals or exceeds the delay of loading submatrix **B**. In such cases, the communication overhead can be effectively hidden at each level of the entire pipeline.

第三级（LV3）流水线包括加载 **B** 的子矩阵和执行 LV2 的内层循环。在数据加载阶段，将大小为 $K_A \times N_A$

表 I
CONSTRAINTS ON DATA PARTITIONING IN NN AND NT MODES.

Capacity constraints (NN Mode)	①: $M_G \times K_G \times d \leq S_1$ ②: $(2 \times K_A + 3 \times M_A) \times N_A \times d \leq S_2$ ③: $2 \times M_S \times K_A \times d \leq S_3$
Capacity constraints (NT Mode)	①: $M_G \times K_G \times d \leq S_1$ ②: $(2 \times M_S + 2 \times N_A) \times K_A \times d \leq S_2$ ③: $3 \times M_A \times N_A \times d \leq S_3$
Bandwidth constraints	①: $P \times \frac{K_A \times N_A + 2 \times M_G \times N_A}{T_1 + T_5 + \frac{M_G}{M_A} \times (T_2 + T_5 + T_3 + T_5)} \leq BW_{DDR}$ ②: $P \times \frac{M_A \times K_A}{\frac{M_A}{M_S} \times (T_6 + T_5)} \leq BW_{GSM}$

的一个子矩阵从 DDR 移动到 AM 中的 b_1, b_2 缓冲区之一。然后，这些数据在 LV2 流水线的执行过程中被微内核取用。为了最小化访存开销，我们类似于之前的操作，将数据加载阶段提前一个时间步开始。与 LV1 流水线类似，这里关注的重点是 LV2 的执行时间应该等于或超过加载子矩阵 **B** 的延时。在这种情况下，才能确保流水线每一层的通信开销能被有效隐藏。

D. Data Partitioning

To determine the right block sizes at each memory level, the following capacity and bandwidth constraints listed in Table I, have to be satisfied.

为了确定每一级片上缓存容纳的数据块大小，表 I 中列出的容量和带宽约束必须得到满足。

1) Capacity constraints: Based on Fig. 4, we observe that the submatrices of **A** are moved from the main memory to GSM. To ensure that their sizes do not exceed the capacity (S_1) of GSM, we need to carefully manage the allocation of these submatrices. Similarly, the submatrices of **B** and **C** are moved from the main memory to AM in the NN mode. Their total sizes must be smaller than the capacity (S_2) of AM. When further tiling the submatrix of **A** and moving these tiles from GSM to SM, the size of each should remain below the capacity (S_3) of SM. In the NT mode, the difference is that the submatrices of **A** and **B** are moved from the main memory to AM, while the submatrices of **C** are sent to SM. Despite being allocated to different parts of the on-chip memory, the total number of buffers remains the same, which imposes the capacity constraints specified in Table I.

根据图 4，我们观察到 **A** 的子矩阵需要先从主存加载到 GSM。为了确保其大小不超过 GSM 的容量（S_1），我们需要限制加载的数据规模。类似地，在 NN 模式下，**B** 和 **C** 的子矩阵从主存加载到 AM 中，它们的总大小必须小于 AM 的容量（S_2）。在进一步把 **A** 的子矩阵切分为更小的数据块，并从 GSM 移动到 SM 时，每个数据块的大小应该保证在 SM 的容量（S_3）之内。在 NT 模式下，不同的是，**A** 和 **B** 的子矩阵会从主存加载到 AM 中，而 **C** 的子矩则被加载到 SM 中。尽管分配到片上内存的不同部分，但缓冲区的总数保持不变，如此就得到了表 I 中的容量约束。

2) Bandwidth constraints: Since accessing AM and SM provide a much larger bandwidth than accessing GSM, which is far faster than accessing the main memory, we have to consider the bandwidth limitations of the main memory (BW_{DDR}) and GSM (BW_{GSM}).

由于 AM 和 SM 的物理带宽高于 GSM，而 GSM 的物理带宽又远远高于主存，所以我们需要同时考虑主存储器（BW_{DDR}）和 GSM（BW_{GSM}）的物理带宽限制。

Bandwidth constraint ① limits the data transfer rate between the main memory and the on-chip memory of each core. At any given time, the actual rate at which all cores transfer data between DDR and their respective on-chip memory should not exceed the physical bandwidth limit of DDR. In the NN mode, these data include the submatrices of **B** loaded to AM and the submatrices of **C** first loaded to AM then stored back to the main memory. In the NT mode, the data volume remains the same, while the submatrices of **C** are transferred between DDR and SM instead. With the sizes of submatrices **B** and **C**, the physical bandwidth of DDR, and the composition of data transfer overhead, we can conclude the constraint ①. Specifically, T_1 represents the loading time for a submatrix of **B** ($K_A \times N_A$) from DDR to AM, while T_2 and T_3 respectively represent the loading and storing time for a submatrix of **C** ($M_A \times N_A$) between DDR and AM/SM. T_5 denotes the overhead incurred by DMA launch and wait. The number of cores (P) is multiplied as a factor, because the DDR bandwidth is shared by the DMA requests of all cores.

带宽约束 ① 限制了主存和每个 DSP 核片上缓存之间的数据传输速率。在任意时刻，DDR 与片上所有核的实际数据传输速率不应超过 DDR 物理带宽的限制。在 NN 模式下，这些数据包括加载到 AM 的来自 **B** 的子矩阵和先加载到 AM 然后存回主存储器的来自 **C** 的子矩阵。在 NT 模式下，数据量保持不变，而 **C** 的子矩阵改为在 DDR 和 SM 之间传输。根据 **B** 和 **C** 的子矩阵大小、DDR 物理带宽和数据传输开销的组成，我们可以得到约束 ①。其中，T_1 表示从 DDR 向 AM 加载 **B** 的一个子矩阵（$K_A \times N_A$）的传输时长，而 T_2 和 T_3 分别表示在 DDR 和 AM/SM 之间加载和写回 **C** 的一个子矩阵（$M_A \times N_A$）的传输时长。T_5 表示 DMA 启动和等待所产生的开销。DSP 核心的数量（P）被作为一个因子相乘，这是因为 DDR 带宽是由所有核的 DMA 请求所共享的。

Bandwidth constraint ② limits the data transfer rate between GSM and the on-chip memory of each core. At any given time, the actual rate at which all cores load data from GSM to their respective on-chip memory should not exceed the physical bandwidth limit

of GSM. In the NN mode, each time before the micro-kernel is invoked, a tile of submatrix **A** sizing $M_S \times K_A$ is loaded from GSM to the SM of each core. In the NT mode, the destination changes to AM, while the data volume remains the same. Considering that DMA transmission occurs M_A/M_S times in the LV1 loop, with T_6 as the single execution time of the micro-kernel, we can conclude the constraint ②.

带宽约束 ② 限制了 GSM 和每个 DSP 核片上缓存之间的数据传输速率。在任意时刻，GSM 与簇内所有核的实际数据传输速率不应超过 GSM 物理带宽的限制。在 NN 模式下，每次调用微内核之前，会将大小为 $M_S \times K_A$ 的来自 **A** 的一个数据块从 GSM 加载到簇内每个核的 SM 中。在 NT 模式下，数据加载的目的地变为 AM，而数据量保持不变。考虑到 DMA 传输在 LV1 循环中发生了 M_A/M_S 次，以 T_6 为微核的单次执行时间，我们可以得到约束 ②。

3) The shape of submatrices: Under the premise of satisfying the capacity constraints, to reduce the memory bandwidth pressure, the on-chip memory should be fully utilized to improve data reuse. The total amount of data flowing in and out of DDR can be calculated as

在满足容量约束的前提下，为了减轻内存带宽压力，应该充分利用片上缓存来提高数据重用率。流入和流出 DDR 的数据总量可以通过下式计算

$$Access_{DDR} = M \times K + M \times K \times N \times (\frac{1}{M_G} + \frac{2}{K_G}) \quad (1)$$

We see that larger M_G and K_G can reduce the total communication volume between the main memory and the on-chip memory, thus making the bandwidth constraint ① more likely to be satisfied. Although other block parameters (e.g. M_A, N_A) are unrelated to data reuse, larger values of them can reduce the number of DMA calls, thereby reducing the overhead of DMA launch and wait.

可见，更大的 M_G 和 K_G 能够降低主存和片上缓存之间通信的数据量，从而使带宽约束 ① 更容易得到满足。尽管其他块参数（例如 M_A、N_A）与数据重用无关，但它们取较大值可以减少 DMA 调用的次数，从而减少 DMA 启动和等待的开销。

Given the capacity of each level of the on-chip memory and the shape of micro-kernels (to be discussed in Section IV), the remaining block parameters can be determined based on the capacity constraints, while the validity of these parameters is verified by the bandwidth constraints. For the FT-M7032 platform, S_1=6MB, S_2=768KB, S_3=64KB. In the NN mode, if we select a micro-kernel with M_S=6 and N_A=48, we can obtain the block parameters for each level using the equations shown in Table I. Based on capacity constraint ③ and the fact that M_S is 6, the value of K_A should not exceed 682. To make K_A a power of 2, we set it as 512. Furthermore, according to the maximum value of K_A, K_G is also set as 512 from the perspective of data blocking. Therefore, based on capacity constraint ①, we calculate that M_G should not exceed 1536. By applying capacity constraint ②, we can obtain the maximum value of M_A as 252 ($42 \times M_S$).

根据每一级片上缓存的容量和微核的形状（将在第 IV 节中讨论），可以根据容量约束确定剩余的分块参数，而这些参数的有效性则由带宽约束来验证。对于 FT-M7032 平台，S_1=6MB，S_2=768KB，S_3=64KB。在 NN 模式下，如果我们选择一个 M_S=6 和 N_A=48 的微核，我们可以使用表 I 中的不等式得到每一级的分块参数。根据容量约束 ③ 和 $M_S = 6$，可知 K_A 的值不应超过 682。为了使 K_A 成为 2 的幂，我们将它设为 512。此外，根据 K_A 的最大值，K_G 从数据分块的角度来说也该设置为 512。因此，根据容量约束 ①，我们计算出 M_G 不应超过 1536。通过应用容量约束 ②，我们可以得到 M_A 的最大取值为 252（$42 \times M_S$）。

In the NT mode, if we choose a micro-kernel that achieves the best performance when sizing M_S=6 and N_A=48 (composed of 6 small micro-kernels with a shape of 6×8), we find according to capacity constraint ③ and ②, the maximum value of M_A is 56 and the value of K_A should not exceed 910. To make K_A a power of 2, we also set it as 512. Note that the loop of k_o is somehow redundant, we eliminate it by setting $K_G = K_A$. Therefore, based on capacity constraint ①, we conclude M_G with a maximum value of 1536 .

在 NT 模式下，如果我们选择一个在大小为 M_S=6 和 N_A=48 时达到最佳性能的微内核（由 6 个形状为 6 × 8 的小微内核组成），我们发现根据容量约束 ③ 和 ②，M_A 的最大值是 56，而 K_A 的值不应超过 910。为了使 K_A 成为 2 的幂，我们也将它设置为 512。注意到 k_o 那层循环有一些多余，我们通过设置 $K_G = K_A$ 将其消除。于是，根据容量约束 ① ，我们设置 K_G 为 1536。

4) Expansion to similar architectures: Our data partitioning approach based on multi-level pipelining is also applicable to other DSP architectures. For processors whose on-chip memory has a scalar section and a vector section, the tiling parameters can be determined according to Table I. For processors with homogeneous on-chip memory, the capacity constraints ② and ③ need to be merged into one constraint. For non-DSP architecture hardwares, such as CPU, our method is inapplicable any more, but still has reference significance. To serve the target platform, the destination of data prefetching can be replaced with the multilevel cache of CPU. Meanwhile, the DMA transfer model can be replaced by a data packing model to adapt to the hardware-managed cache.

我们基于多级流水线的数据分块方法也适用于其他 DSP 架构。对于片上缓存分为标量区和向量区的处理器，可以根据表 I 确定分块参数。对于具有同构片上

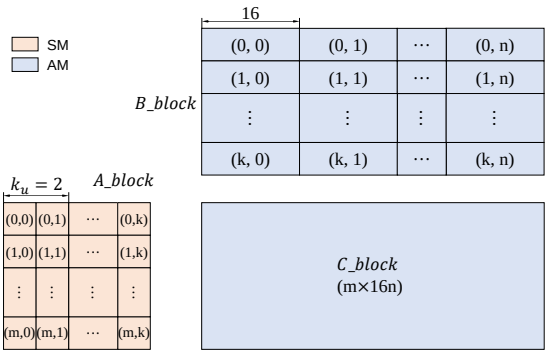


图 7. The data shape of micro-kernel in the NN mode

缓存的处理器，容量约束 ② 和 ③ 需要合并为一个约束。对于非 DSP 架构的硬件，例如 CPU，我们的方法不再适用，但仍具有参考意义。为适配目标平台，数据预取的目的地可以用 CPU 的多级缓存来替换。同时，DMA 传输模型可以用数据打包模型来替换，以适应硬件管理的片上缓存。

IV. Micro-kernel Design

To maximize the utilization of VPU and its three FMAC units, we have to answer two questions when implementing our micro-kernel in assemblies. The first is to design the shape of the micro-kernel, which determines how many registers to use and whether each cycle of the main loop can be fully filled with multiply-add instructions. The second is to design the entire instruction pipeline of the main loop, including choosing suitable instructions, packing them correctly, and scheduling them properly to eliminate their latency.

为了最大化 VPU 及其三个 FMAC 单元的利用率，我们在用汇编代码实现微内核时必须回答两个问题。第一个问题是如何确定微内核的形状，它决定了要使用多少个寄存器，以及主循环的每一拍是否能够完全填充乘加指令。第二个问题是怎样设计主循环的指令流水线，包括选择合适的指令，正确地打包它们，以及合理地调度它们以便隐藏指令延迟。

A. NN Mode Micro-kernel

1) The shape of the micro-kernel: Here, shape refers to the sizes of data processed in each micro-kernel call. Fig. 7 shows the 4 parameters we use to denote the micro-kernel computation, i.e., m denotes the height of block A and C, n denotes the number of vectors per row in block B and C, k denotes the length of the common dimension of block A and B, and k_u denotes the unroll factor of the main loop. While k is passed by the outer algorithm with the value of K_A (Section III-D3), we have to specify m, n, k_u at the micro-kernel level. We compute them according to three specific constraints: available vector registers, instruction packet filling and pipeline utilization.

这里，形状指的是每次调用微内核处理的数据规模。图 7 显示了我们用来描述微内核计算的 4 个参数，即 m 表示数据块 A 和 C 的高度，n 表示数据块 B 和 C 每行的向量数，k 表示数据块 A 和 B 的公共维度的长度，以及 k_u 表示主循环的循环展开因子。由于参数 k 是由外层算法传递的，等于 K_A (第 III-D3 节)，我们只需在微内核的层级确定 m，n，k_u 这三个参数。确定它们依据三类约束，分别是：可用的向量寄存器资源，指令包的完全填充，以及指令流水线的充分利用。

a) Constraint on register resources: During each iteration of the main loop, $m \times k_u$ scalars from block A in SM are read and broadcasted into vectors, which requires $m \times k_u$ vector registers. $n \times k_u$ registers are used to store vectors read from block B in AM. Besides, another $m \times n$ vector registers are required to store the intermediate results of $A \times B$ (i.e. the result registers). Given that each DSP core contains 64 vector registers, the shape of the micro-kernel has to satisfy Eq.(2).

在主循环的每一轮迭代中，大小为 $m \times k_u$ 的数据块 A 将从 SM 中被读取，并逐个广播成向量，占用 $m \times k_u$ 个向量寄存器。另外 $n \times k_u$ 个向量寄存器将用于暂存从 AM 中读取的数据块 B 。此外，还需要另外 $m \times n$ 个向量寄存器来存储 $A \times B$ 的中间结果（即结果寄存器）。考虑到每个 DSP 核心包含 64 个向量寄存器，微内核的形状必须满足方程 (2)。

$$\textbf{C1:}\quad (m+n) \times k_u + m \times n \leq 64 \tag{2}$$

b) Constraint on filling instruction packets: To fully utilize all the FMAC units, three multiply-add instructions need to be packed and issued per cycle. Each iteration of the unrolled main loop, a total of $m \times n \times k_u$ multiply-add instructions are issued, taking $(m \times n \times k_u)/3$ cycles. Consider that k represents the largest dimension of the buffer and tends to be a power of 2, k_u should be set as

为了充分利用所有的 FMAC 单元，每一拍需要打包并发射三条乘加指令。在循环展开后的每一轮迭代中，总共会发射 $m \times n \times k_u$ 条乘加指令，消耗 $(m \times n \times k_u)/3$ 个时钟周期。考虑到 k 是缓冲区的最长维度，并且倾向于是 2 的幂，所以 k_u 也该满足

$$\textbf{C2:}\quad k_u = 2^x \quad (x \geq 0) \tag{3}$$

Therefore, at least one of the dimensions, either m or n, should be a multiple of 3.

因此，至少有一个维度，要么是 m，要么是 n，应该是 3 的倍数。

$$\textbf{C3:}\quad m \times n = 3y \quad (y \geq 1) \tag{4}$$

c) Constraint on pipeline utilization: We should also design the micro-kernel to minimize pipeline stalls due to data dependencies. Before loop unrolling, the multiply-add instructions conduct read-then-write

表 II
FEASIBILITY ANALYSIS OF VARIOUS PARAMETER COMBINATIONS

m, n	$n=3$	$n=4$	$n=6$	$n=8$
$k_u=1$				
$m=3$	Violate C4	Violate C4	Violate C5	Violate C5
$m=4$	Violate C4	Violate C3	Violate C5	Violate C3
$m=6$	Violate C5	Violate C5	Feasible	Feasible
$m=8$	Violate C5	Violate C3	Feasible	Violate C3
$k_u=2$				
$m=3$	Violate C4	Violate C4	Feasible	Feasible
$m=4$	Violate C4	Violate C3	Feasible	Violate C3
$m=6$	Feasible	Feasible	Feasible	Violate C1
$m=8$	Feasible	Violate C3	Violate C1	Violate C3

operations on each vector from the $m \times n$ result registers. This sequence is repeated in the subsequent iteration. To guarantee that the next read of a result register occurs only after the current write is completed, we have to delay the issuing of the next multiply-add instruction until the current one is finished. Therefore, m and n should satisfy Eq.(5).

我们还需要最小化由于数据依赖而导致的流水线停顿。在循环展开之前，乘加指令会对 $m \times n$ 个结果寄存器中的每个向量进行先读后写的操作。这一操作过程在后续的迭代中将不断重复。为了保证下一次对结果寄存器的读取只在当前的写入完成后发生，我们必须延迟发出下一条乘加指令，直到当前的指令完成。因此，m 和 n 的取值应该满足 Eq.(5)。

$$\textbf{C4:} \quad m \times n \geq 18 \tag{5}$$

The entire process includes loading scalars from block A in SM (using SLDW/SLDDW with 7-cycle latency), broadcasting them into vectors (using SVBCAST with 4-cycle latency), and multiplying them with the vectors loaded from block B (using VFMULAD with 6-cycle latency) before being added to result registers. These steps take a total of $7+4+(m\times n\times k_u)/3$ cycles. Since we can prefetch data for the next iteration while simultaneously computing data for the current one, this entire process only needs to be completed within two iterations. So Eq.(6) should be satisfied.

每一轮迭代包括从 SM 中的 A 数据块加载标量（使用 SLDW/SLDDW，延迟 7 个周期），将它们广播成向量（使用 SVBCAST，延迟 4 个周期），并在累加到结果寄存器之前，将它们与从 B 数据块中加载的向量相乘（使用 VFMULAD，延迟 6 个周期）。这些步骤总共需要 $7+4+(m\times n\times k_u)/3$ 个时钟周期。由于我们可以在计算当前步数据的同时预取下一个迭代步需要的数据，所以这个过程只需要在两轮迭代内完成。所以，(6) 式应该被满足。

$$\textbf{C5:} \quad 11 + m \times n \times k_u/3 \leq 2 \times (m \times n \times k_u/3) \tag{6}$$

2) Assembly Pipeline: Before designing the instruction pipeline, a feasibility analysis of various parameter combinations based on the 5 constraints above is conducted, shown in Table II, where k_u is taken as 1 or 2. m, n is chosen from 3, 4, 6, and 8. While multiple parameter combinations can satisfy the constraints, we choose the smaller data size with balanced height and width. Thus, the micro-kernel shape in the NN mode is $k_u = 2$, $m = 6$, and $n = 3$.

1	VFMULAD A[0][0], B[0][0,1,2]		SVBCAST A[4][0]
2	VFMULAD A[1][0], B[0][0,1,2]	SLDDW A_next[0][0,1]	SVBCAST A[5][0]
3	VFMULAD A[2][0], B[0][0,1,2]	SLDDW A_next[1][0,1]	SVBCAST A[0][1]
4	VFMULAD A[3][0], B[0][0,1,2]	SLDDW A_next[2][0,1]	SVBCAST A[1][1] | VLDW B_next[0][0,1,2]
5	VFMULAD A[4][0], B[0][0,1,2]	SLDDW A_next[3][0,1]	SVBCAST A[2][1]
6	VFMULAD A[5][0], B[0][0,1,2]	SLDDW A_next[4][0,1]	SVBCAST A[3][1]
7	VFMULAD A[0][1], B[1][0,1,2]	SLDDW A_next[5][0,1]	SVBCAST A[4][1]
8	VFMULAD A[1][1], B[1][0,1,2]		SVBCAST A[5][1]
9	VFMULAD A[2][1], B[1][0,1,2]		SVBCAST A_next[0][0]
10	VFMULAD A[3][1], B[1][0,1,2]	VLDW B_next[1][0,1,2]	SVBCAST A_next[1][0]
11	VFMULAD A[4][1], B[1][0,1,2]		SVBCAST A_next[2][0]
12	VFMULAD A[5][1], B[1][0,1,2]		SVBCAST A_next[3][0]

图 8. The assembly pipeline of the NN mode micro-kernel

在设计指令流水线之前，我们根据上述 5 个约束对不同参数组合进行了可行性分析，结果如表 II 所示，其中 k_u 取自于 1 或 2。m，n 取自于 3，4，6，8 中选择。虽然有多个参数组合可以满足约束条件，但我们选择了规模较小且数据高度和宽度相平衡的组合。因此，NN 模式下的微内核形状是 $k_u = 2$，$m = 6$，$n = 3$。

We analyze that each iteration takes 12 cycles after loop unrolling, during which two columns of A and two rows of B are multiplied successively. Thanks to the row-major storage, two adjacent scalars in the same row of block A can be loaded simultaneously with a single SLDDW instruction. We show the instruction pipeline of each iteration in Fig. 8, where $A(B)[i][j]$ represents the scalar(vector) located at row i and column j in block $A(B)$, and $\_next$ denotes the data to be used in the next iteration.

可以分析出，循环展开之后，每轮迭代需要 12 个周期。这期间，A 的两列数据和 B 的两行数据依次相乘。由于采用了行优先存储，数据块 A 中同一行的两个相邻标量可以用一条 SLDDW 指令同时加载。图 8 中展示了主循环每轮迭代的指令流水线，其中 $A(B)[i][j]$ 表示块 $A(B)$ 中位于第 i 行第 j 列的标量（向量），$_next$ 表示下一轮迭代需要使用的数据。

3) Performance Analysis: At the beginning and end of the micro-kernel, the register initialization phase and data write-back phase take 14 and 12 cycles respectively. In the middle, the main loop includes $k/2$ iterations, consuming a total of $6k$ cycles. So the theoretical performance is $\frac{6k}{14+6k+12} = \frac{3k}{3k+13}$ of the hardware peak. When k is set as 512 according to K_A, the predicted efficiency is 99.2%.

在微内核的开始和结束阶段，寄存器初始化和数据写回分别需要 14 和 12 个周期。位于中间的主循环有 $k/2$ 次迭代，总共消耗 $6k$ 个周期。因此，微内核的理论性能是硬件峰值的 $\frac{6k}{14+6k+12} = \frac{3k}{3k+13}$。当 k 根据 K_A 设置为 512 时，微内核的预期效率为 99.2%。

4) Expansion to other platforms: Our micro-kernel design is also applicable to platforms other than FT-M7032. For other DSP architectures that support VLIW and SIMD, we still have to use the outer product

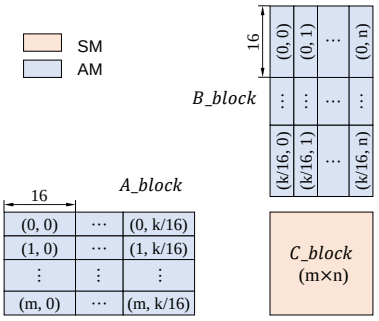


图 9. The data shape of micro-kernel in the NT mode

implementation, and the shape of micro-kernel can be determined according to the hardware resources and their corresponding instruction latency. Then, the selected instructions are reordered and packed to overlap between computation and on-chip memory access. For CPU architectures that support out-of-order execution, constraints on instruction latency can be relaxed, while the scarcity of register resources needs to be considered.

我们的微内核设计也适用于 FT-M7032 以外的平台。对于支持 VLIW 和 SIMD 的其他 DSP 架构，我们仍然必须使用外积实现，并根据硬件资源和相应的指令延迟确定微内核的形状。然后，选择的指令被重新排序和打包，以重叠计算和片上访存的开销。对于支持乱序执行的 CPU 架构，可以放宽对指令延迟的约束，但寄存器资源的稀缺性需要考虑。

B. NT Mode Micro-kernel

In the NT mode, several changes are made to accommodate the column-major storage of $\mathbf{B^T}$. To multiply a column of row-major vectors in block A with a row of column-major vectors in block B, we store block A, B in AM and block C in SM, as shown in Fig. 9. The outcome of each vector multiplication is still a vector, which is accumulated and temporarily stored in the result registers. Once the entire main loop is finished, each vector result is reduced to a scalar value and written back to SM.

在 NT 模式下，为了适应 $\mathbf{B^T}$ 的列优先存储，我们做了一些改变。为了将数据块 A 中的一列行优先向量与数据块 B 中的一行列优先向量相乘，我们将块 A，B 存储在 AM 中，将块 C 存储在 SM 中，如图 9 所示。每次向量乘法的结果仍然是一个向量，它被累加并临时存储在结果寄存器中。当整个主循环结束后，每个向量结果会被规约为一个标量值，并写回到 SM 中。

1) Vector Reduction: We propose a novel vector reduction approach that combines modulo vector loading and vector adding. Modulo vector loading refers to a special type of instructions, including VLDDWM2 and VLDDWM4, which are used to load data from AM to vector registers at an address interval of 2 or 4. VLDDWM4 is used in our implementation.

我们提出了一种新颖的向量规约方法，它结合了模取数和向量加法。模取数指的是一种特殊类型的指令，包括 VLDDWM2 和 VLDDWM4，它们用于以 2 或 4 的地址间隔从 AM 加载数据到向量寄存器中。我们的实现中使用了 VLDDWM4。

Suppose there are 4 vectors to be reduced, our approach involves the following five steps, as shown in Fig. 10.

假设有 4 个向量需要规约，我们的方法包括以下五个步骤，如图 10 所示。

Step ①: The vectors are written to AM using the VSTDW instruction and then read back into vector registers using the VLDDWM4 instruction. This step ensures that the elements originally residing in a single vector register are now distributed across four different vector registers.

步骤 ①：使用 VSTDW 指令将向量写入 AM，然后使用 VLDDWM4 指令将它们读回到向量寄存器中。这一步确保了最初位于单个向量寄存器中的元素现在分布到了四个不同的向量寄存器中。

Step ②: An accumulation is performed on these four vector registers, resulting in a single vector where every four elements form a group to be further reduced.

步骤 ②：对这四个向量寄存器进行累加，得到一个向量，其中相邻的四个元素作为一组，有待进一步规约。

Step ③: A step similar to ① is performed. The vector obtained from the previous step is moved to AM and loaded into vector registers again with the VLDDWM4 instruction. As a result, the four elements from each group are distributed across four different vector registers.

步骤 ③：类似于 ①，将从上一步得到的向量移动到 AM，并再次使用 VLDDWM4 指令将其加载到向量寄存器中。这样一来，之前每组中的四个元素此刻分布到了四个不同的向量寄存器中。

Step ④: An accumulation is once again performed on these four vector registers, yielding a vector where each element represents the final result of the reduction.

步骤 ④：再次对这四个向量寄存器进行累加，得到一个向量，其中每个元素即为规约的最终结果。

Step ⑤: The reduced vector is moved to the scalar-vector register (SVR), where each element is then read individually into a scalar register. Afterwards, these values are added to the original data from the C block and written back to SM.

步骤 ⑤：将规约后的向量移动到标量-向量寄存器（SVR），其中每个元素被逐个读入标量寄存器。之后，这些值与数据块 C 中的原始数据相加，并写回到 SM 中。

During the reduction process, we have to consider the efficiency of these vector operations. In steps ① and ②, each vector register is fully utilized. But in steps ③ and ④, only the first 4 elements within each vector

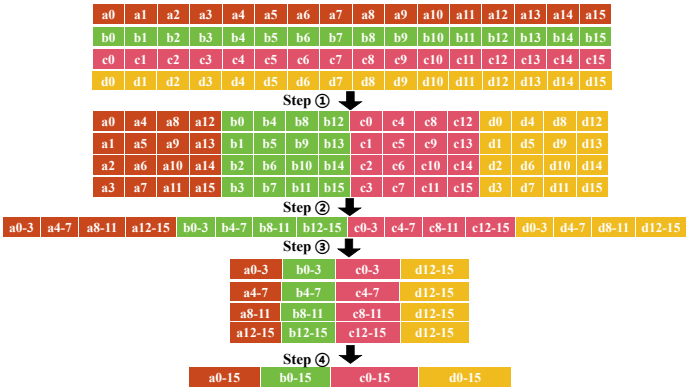


图 10. The workflow of our vector reduction method.

register contribute to useful operations. To utilize the remaining part, we should reduce 16 vectors in a batch. Besides, the reduction process involves frequent AM access using the VLDDW instruction. Each cycle, two VLDDWs are issued to load 4 vectors, with a latency of 9 cycles. To hide this latency, a consecutive loading of at least 36 vectors is required. Considering these requirements, we set the number of vectors for reduction (also the size of block C) as 48.

在规约过程中，我们必须考虑这些向量操作的效率。在步骤 ① 和 ② 中，每个向量寄存器都被充分利用。但是在步骤 ③ 和 ④ 中，每个向量寄存器只有前 4 个元素参与了有效的操作。为了利用剩余的部分，我们应该一次规约 16 个向量。此外，规约过程涉及使用 VLDDW 指令频繁地访问 AM。每个周期，发射两条 VLDDW 指令来加载 4 个向量，延迟为 9 拍。为了隐藏这个延迟，需要连续加载至少 36 个向量。考虑到这些要求，我们将规约的向量数（也是数据块 C 的大小）设置为 48。

2) Assembly Pipeline: In the NT mode, designing the micro-kernel shape is similar to the NN case. The only difference is that block A is loaded in vector granularity with a latency of 9 cycles. Thus, the C5 constraint is replaced by C6, while the C1-C4 constraints remain the same.

在 NT 模式下，确定微内核形状的方法与 NN 情况类似。唯一的区别是数据块 A 以向量粒度加载，延迟为 9 拍。因此，C5 约束被 C6 替代，而 C1-C4 约束保持不变。

$$\textbf{C6:}\quad m \times n \times k_u/3 \geq 9 \tag{7}$$

We calculate that $k_u = 1, m = 6, n = 8$ satisfies these five constraints and make the size of block C exactly 48. The instruction pipeline of the main loop is shown in Fig. 11.

我们计算发现，参数组合 $k_u = 1, m = 6, n = 8$ 满足这五个约束，并使数据块 C 的大小恰好为 48。主循环的指令流水线如图 11 所示。

```
1   VFMULAD A[2], B[3,2,1]          | VLDW A[4]
2   VFMULAD A[1], B[3,2,1]          | VLDW A[3]
3   VFMULAD A[0], B[3,2,1]
4   VFMULAD A[2,1,0], B[0]
5   VFMULAD A[2,1,0], B[7]          | VLDW A_next[2]
6   VFMULAD A[2,1,0], B[6]          | VLDW B_next[3]
7   VFMULAD A[2,1,0], B[5]          | VLDW B_next[2]
8   VFMULAD A[2,1,0], B[4]          | VLDW B_next[1]
9   VFMULAD A[5], B[3,2,1]          | VLDW A_next[1]
10  VFMULAD A[4], B[3,2,1]          | VLDW A_next[0]
11  VFMULAD A[3], B[3,2,1]          | VLDW B_next[0]
12  VFMULAD A[5,4,3], B[0]          | VLDW B_next[7]
13  VFMULAD A[5,4,3], B[7]          | VLDW B_next[6]
14  VFMULAD A[5,4,3], B[6]          | VLDW B_next[5]
15  VFMULAD A[5,4,3], B[5]          | VLDW B_next[4]
16  VFMULAD A[5,4,3], B[4]          | VLDW A_next[5]
```

图 11. The assembly pipeline of the NT mode micro-kernel.

表 III
THE STEPS AND CORRESPONDING OVERHEADS OF VECTOR REDUCTION.

Step	Instruction	Lat	Function	#Insts	#Cycs
①	VSTDW	4	write 48 vectors into AM in sequence	24	12
	VLDDWM4	9	read 48 vectors into vector registers in modulo 4 order	24	12
②	VFMULAD	6	reduce 48 vectors to 12 vectors after 3 rounds of addition	36	18
③	VSTDW	4	write 12 vectors into AM in sequence	6	3
	VLDDWM4	9	read 12 vectors into vector registers in modulo 4 order	6	9
④	VFMULAD	6	reduce 12 vectors to 3 vectors after 3 rounds of addition	9	12
⑤	VMVCGC	3	write the 3 vectors into SVR	3	3
	SMVCCG	2	read each element from SVR into a scalar register	48	48
	SFADDD	3	add 48 scalars to original data from the C block	48	2
	SSTDW	4	write the results back into the C block in SM	48	3
total cycles of reduction + write-back					122

3) Performance Analysis: The register initialization phase takes 16 cycles. As for the main loop phase, whose step size is based on the vector length, the total number of iterations is $k/16$ when $k_u = 1$. Since each iteration takes 16 cycles, the main loop consumes a total of k cycles. In the data write-back phase, all 48 vectors are divided into three batches, conducting the five-step vector reduction in a pipeline. Table III lists the specific steps with their actual overheads after instruction packing and pipelining.

寄存器初始化阶段需要 16 个周期。对于主循环阶段，步长大小基于向量长度，当 $k_u = 1$ 时，主循环总共有 $k/16$ 轮迭代。由于每轮迭代需要 16 个周期，主循环总共消耗 k 个周期。在数据写回阶段，所有 48 个向量被分成三批，以流水线的方式进行上述的五步向量规约。表 III 列出了指令打包和流水线后的具体步骤及其实际开销。

To sum up, the theoretical performance of the NT mode micro-kernel is $\frac{k}{16+k+122} = \frac{k}{138+k}$ of the hardware peak performance. When k is set to 512, 1024, 2048, the predicted efficiency is 78.8%, 88.1% and 93.7% respectively.

总之，NT 模式下微内核的理论性能是硬件峰值的 $\frac{k}{16+k+122} = \frac{k}{138+k}$。当 k 设置为 512, 1024, 2048 时，微内核的预期效率分别为 78.8%，88.1% 和 93.7%。

C. Upper Bound Analysis

Now we analyze the theoretical time complexity of Fig. 4 in the best-case scenario. The delays that are

difficult to hide include the DMA transmission latency (T_{D1}) and the DMA launch/wait latency (T_{D2}). The former consists of five parts: (1) DMA transmissions from the main memory to GSM (T_0), (2) the first-time loading of submatrix B from the main memory to AM (T_1), (3) the first-time loading of submatrix C from the main memory to AM (T_2), (4) the last-time storing of submatrix C from AM to the main memory (T_3), and (5) the first-time loading of tile A from GSM to SM (T_4).

现在我们在最理想的情况下分析图 4 的理论时间复杂度。其中难以隐藏的延迟包括 DMA 传输延迟 (T_{D1}) 和 DMA 启动/等待延迟 (T_{D2})。前者包括五个部分：(1) 从主存到 GSM 的 DMA 传输 (T_0)，(2) 第一次从主存加载子矩阵 B 到 AM(T_1)，(3) 第一次从主存加载子矩阵 C 到 AM (T_2)，(4) 最后一次从 AM 写回子矩阵 C 到主存 (T_3)，和 (5) 第一次从 GSM 加载数据块 A 到 SM (T_4)。

T_{D1} can be calculated with Eq.(8).

T_{D1} 可以用等式 (8) 来计算。

$$T_{D1} = \frac{K \times M}{K_G \times M_G} \times (T_0 + T_1 + T_2 + T_3 + T_4) \tag{8}$$

T_{D2} represents the total time consumed by DMA launch and wait operations, which can be calculated in Eq(9),

T_{D2} 表示 DMA 启动和等待操作消耗的总时间，可以用等式 (9) 计算，

$$T_{D2} = (\frac{KM}{K_G M_G} + \frac{KMN_G}{K_G M_G N_A} + \frac{2MKN_G}{M_A K_A N_A} + \frac{MKN_G}{M_S K_A N_A}) \times T_5 \tag{9}$$

where T_5 represents the time consumed by a single pair of DMA launch and wait operation.

其中 T_5 表示一对 DMA 启动和等待操作消耗的时间。

With the single execution time (T_6) and the number of executions of the micro-kernel, the total computation time T_{comp} is calculated as

根据微内核的单次执行时间 (T_6) 和执行次数，可以得到用于计算的总时长为 T_{comp}。

$$T_{comp} = \frac{K \times M \times N_G}{K_A \times M_S \times N_A} \times T_6 \tag{10}$$

By considering the ratio of computation time to communication overhead, we can further calculate the maximum attainable floating-point performance of our algorithm.

通过考虑计算时长与通信开销的比例，我们可以进一步计算我们的算法能达到的最大浮点性能。

$$\frac{maxFP}{FP} = \frac{T_{comp}}{T_{D1} + T_{D2} + T_{comp}} \tag{11}$$

Based on our observations, we know that T_{D1} can be ignored. But the latency of DMA launch and wait is a significant factor impacting the overall performance, with the constant T_5 constituting approximately 2% of T_6. To exemplify it, let us consider a micro-kernel in the NN mode with the shape of $M_S = 6$, $N_A = 48$, and $K_A = 512$. Taking into account the bandwidth and capacity constraints, we have determined the optimal configuration as $M_G = 1512$, $M_A = 252$, $K_G = 512$, and $N_G = 1152$. Assuming that the shape of the input matrices is $M = M_G$, $K = K_G$, and $N = N_G$, we calculate the maximum efficiency as 97.4%. With the micro-kernel of 99.2% efficiency, our method can achieve 96.6% of the hardware peak performance. Note that we only take the example of K_A=512 to illustrate how we analyze the upper bound of performance. As K_A increases, the performance of both the micro-kernel and the GEMM implementation will correspondingly increase. In the NT mode, the equation remains the same, and the maximum efficiency is 76.75%, 85.8%, and 91.26% when K_A is set to 512, 1024, and 2048.

根据我们的观察，可知 T_{D1} 可以忽略。但是 DMA 启动和等待的延迟却是影响总体性能的一个重要因素，其中常数 T_5 占 T_6 的大约 2%。为了举例说明，让我们考虑 NN 模式下的一个微内核，其形状为 $M_S = 6$，$N_A = 48$，和 $K_A = 512$。考虑到带宽和容量的限制，我们确定了最优的参数配置为 $M_G = 1512$，$M_A = 252$，$K_G = 512$，和 $N_G = 1152$。假设输入矩阵的形状为 $M = M_G$，$K = K_G$，和 $N = N_G$，我们计算出 GEMM 的最大效率为 97.4%。在微内核效率为 99.2% 的情况下，我们的方法可以达到硬件峰值性能的 96.6%。注意，我们只是以 K_A=512 为例来说明我们如何分析性能的上限。随着 K_A 的增加，微内核和 GEMM 的性能都会相应地提高。在 NT 模式下，等式保持不变，当 K_A 设置为 512，1024，和 2048 时，最大效率分别为 76.75%，85.8%，和 91.26%。

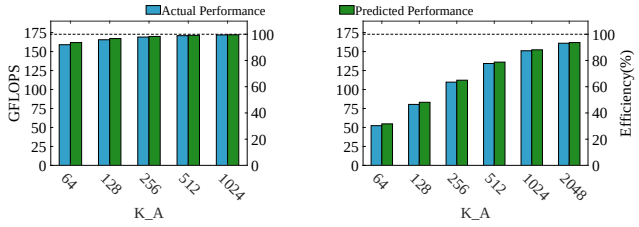
V. Performance Results

This section evaluates our GEMM implementation with a range of input matrices of varying shapes. We analyze the performance of matrix multiplication in micro-kernel, single-core, and multi-core scenarios for both NN and NT modes. Additionally, we compare the performance of our approach to state-of-the-art. Note that, we run each experiment 10 times, and report the geometric mean of the actual performance.

本节使用不同形状的一系列输入矩阵评估我们的 GEMM 实现。我们分析了在 NN 和 NT 模式下，微内核、单核和多核场景下矩阵乘计算的性能。此外，我们还将本文的方法与最近的方法进行了性能对比。我们运行每个实验 10 次，并报告实际性能的几何平均值。

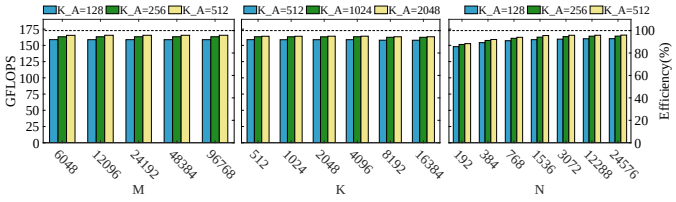
A. Micro-Kernel Performance

As stated in Section IV, the performance of the micro-kernel depends on K_A. We evaluate the actual



(a) NN Mode (b) NT Mode

图 12. The micro-kernel performance. The dashed line represents the hardware peak performance of a single DSP core. The maximum value of K_A in (a) is limited to 1024 due to the capacity constraint of SM.



(a) K=2048, N=1536 (b) M=6048, N=1536 (c) M=6048, K=2048

图 13. The NN Mode performance on a single DSP core.

performance of the micro-kernel with various K_A values while keeping the remaining two dimensions unchanged. For the NN and NT mode, we set the shape of micro-kernel as $6\times K_A\times 48$ and $6\times K_A\times 8$ respectively.

如第 IV 节所述，微内核的性能取决于 K_A 的大小。我们在保持其余两个维度不变的情况下，用不同的 K_A 值评估微内核的实际性能。对于 NN 和 NT 模式，我们将微内核的形状分别设置为 $6\times K_A\times 48$ 和 $6\times K_A\times 8$。

Fig. 12 shows that the actual performance closely aligns with the predicted values and improves as K_A increases. The difference between the two is mainly attributed to the overhead of outer loops. The reason for performance increase with the K_A is that a bigger K_A generates a larger proportion of the total cycle count allocated to the computation phase, leading to better efficiency of the micro-kernel.

图 12 显示了实际性能与预测值紧密对应，并随着 K_A 的增加而提高。两者之间的差异主要是因为外层循环的开销。微内核性能随 K_A 增加的原因是，较大的 K_A 使得微内核中计算阶段占据了更大的时间比例，从而提高了微内核的效率。

In the NN mode, setting K_A to 1024 achieves a performance of 171.9 GFlops, which is 99.5% of the hardware peak performance and close to the predicted value of 99.57%. Reducing K_A to 512 lowers the actual performance to 170.9 GFlops, accounting for 98.9% of the hardware peak. It proofs that our model can accurately predict the achieved performance.

在 NN 模式下，将 K_A 设置为 1024，微内核的性能可以达到 171.9 GFlops ，是硬件峰值性能的 99.5%，接近预测值的 99.57%。将 K_A 降低到 512 会使微内核的实际性能下降到 170.9 GFlops，仍然有硬件峰值的 98.9%。这证明我们的模型可以准确地预测微内核的实际性能。

From Fig. 12 (b), we observe that the measured performance closely aligns with the predicted values in the NT mode. When K_A is set to 2048, the actual performance is 161.0 GFlops, reaching 93.2% of the hardware peak performance. Our model accurately predicts an efficiency of 93.6%, with only a 0.4% difference from the measured one.

从图 12 (b) 中可以看出，NT 模式下的实际性能与预测值很接近。当 K_A 设置为 2048 时，实际性能为 161.0 GFlops，达到了硬件峰值性能的 93.2%。我们的模型预测值为 93.6% 的效率，与测量值之间只有 0.4% 的误差。

We also observe that the micro-kernel can achieve high performance even if K_A is very small in the NN mode, but not in the NT mode. When K_A is set to 64, the micro-kernel in the NN mode achieves a performance of 159.0 GFlops, reaching 92.0% of the hardware peak performance. Meanwhile, the micro-kernel in the NT mode only achieves 52.4 GFlops, which is 30.3% of the hardware peak. The significant gap between the two can be attributed to the differences in their micro-kernel designs. Based on our performance analysis in Sec. IV-A3 and Sec. IV-B3, the predicted efficiency in the NN mode is given by $3k/(3k+13) \rightarrow k/(k+13/3)$, while in the NT mode it is $k/(k+138)$. Here, $k = K_A$. The NT mode includes a time consuming phase of vector reduction, leading to low efficiency when K_A is small.

我们还观察到，即使当 K_A 很小时，NN 模式下的微内核也可以实现高性能，而 NT 模式则不行。当 K_A 设置为 64 时，NN 模式下的微内核达到了 159.0 GFlops 的性能，是硬件峰值的 92.0%。与此同时，NT 模式下的微内核只达到了 52.4 GFlops，是硬件峰值的 30.3%。两者之间性能的显著差距可以归结于它们的微内核设计差异。根据我们在第 IV-A3 节和第 IV-B3 节的性能分析，NN 模式下微内核效率的计算公式为 $3k/(3k+13) \rightarrow k/(k+13/3)$ ，而 NT 模式下微内核效率的计算公式为 $k/(k+138)$。这里的 k 即为 K_A。NT 模式的微内核因为包含一个耗时的向量规约阶段，导致在 K_A 很小时效率低下。

B. Single-core Performance

We evaluate the GEMM performance of various input matrices on a single DSP core. The input matrices are generated by fixing the lengths of two dimensions and changing the third one.

我们在单个 DSP 核上评估了在不同输入矩阵情况下 GEMM 的性能。输入矩阵是通过固定两个维度的长度并改变第三个维度来生成的。

1) NN Mode: In the NN mode, we conduct tests with three values of K_A (128, 256, 512). To satisfy the

capacity constraints, we set the remaining parameters to N_A=192, 96, 48, M_S=24, 12, 6, M_A=72, 144, 252, M_G=6048, 3024, 1512, respectively. The results are shown in Fig. 13.

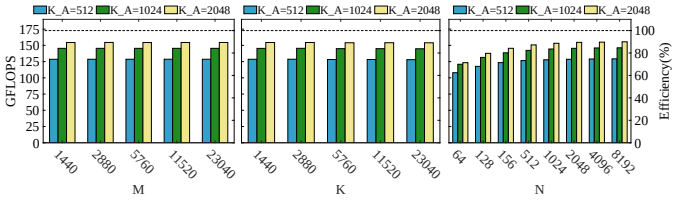
在 NN 模式下，我们用三个 K_A 值 (128, 256, 512) 进行测试。为了满足容量约束，我们分别将其余参数设置为 N_A=192，96，48，M_S=24，12，6，M_A=72，144，252，M_G=6048，3024，1512。结果如图 13 所示。

As seen in Fig. 13 (a) and (b), for the three values of K_A, the performances are very close, namely 158, 163 and 165 GFlops. Each represents approximately 92%, 94% and 96% of the hardware peak performance. For $K_A = 512$, compared with the performance of micro-kernel, the kernel performance only drops by 3%, indicating a good computation-communication overlap. The reason why the actual performance is insensitive to M and K is that they only determine the number of blocks traversed in the outermost two loops, but do not affect the overlap between computation and DMA transmissions in the inner loops. Furthermore, considering that the total amount of communication for matrix **A** is $M \times K$, while for matrix **B** and **C**, it is $M \times K \times N \times (\frac{1}{M_G} + \frac{2}{K_G})$. When N is large enough, the latter quantity significantly surpasses the former. Therefore, the communication overhead associated with matrix **A** becomes negligible.

从图 13 (a) 和 (b) 中可以看出，在三种 K_A 的取值下，单核性能非常接近，分别为 158，163 和 165 GFlops，分别是硬件峰值性能的 92%，94% 和 96%。对于 $K_A = 512$，与微内核的性能相比，GEMM 的整体性能只下降了 3%，说明计算和通信重叠得好。M 和 K 对实际性能影响很小，原因是它们只决定了在最外两层循环中遍历的数据块数，但不影响内层循环中计算操作和 DMA 传输操作之间的重叠。此外，考虑到矩阵 **A** 的总通信量为 $M \times K$，而矩阵 **B** 和 **C** 的总通信量为 $M \times K \times N \times (\frac{1}{M_G} + \frac{2}{K_G})$。当 N 足够大时，后者显著超过前者。因此，与矩阵 **A** 相关的通信开销可以忽略不计。

On the other hand, Fig. 13 (c) shows that with an increase of N from 192 to 24576, the performance rises slightly from 148, 152, 153 GFlops and soon stabilizes at 160, 163, 165 GFlops, respectively. The influence of length N on performance arises from its role in determining the number of subsequent **B** and **C** blocks loaded immediately after an **A** block. Despite the utilization of double buffering techniques for loading **B** and **C** blocks, the initial stage solely involves data loading without any computation, while the final stage exclusively focuses on computation without loading new data. Thus, a sufficient quantity of **B** and **C** blocks facilitates a better overlap between computation and communication.

另一方面，从图 13 (c) 可以看出，随着 N 从 192 增加到 24576，单核性能始于 148，152，153 GFlops，逐渐上升，并很快分别稳定在 160，163，165 GFlops。N 的长度对性能的影响源于它决定了在加载完一个 **A** 块之后会紧接着加载多少个 **B** 块和 **C** 块。尽管加载 **B** 块和 **C** 块都使用了双缓冲技术，但在初始阶段只有数据加载而没有计算，在终了阶段只有计算而无数据加载。因此，只有当 **B** 块和 **C** 块的数量足够，才能更好地实现计算和通信之间的重叠。



(a) K=2048, N=2048 (b) M=2880, N=2048 (c) M=2880, K=2048

图 14. The NT Mode performance on a single DSP core.

2) NT Mode: Similarly, we conduct tests using three values of K_A (512, 1024, 2048) to evaluate its impact on single-core performance in the NT mode. To satisfy the capacity constraints, we set the remaining parameters to N_A=64, 32, 16, M_S=12, 12, 6, M_A=36, 72, 144, M_G=1440, 720, 288, respectively.

类似地，我们采用 K_A 的三个取值 (512，1024，2048) 来评估 NT 模式下的单核性能。为了满足容量约束，我们将其余参数分别设置为 N_A=64，32，16，M_S=12，12，6，M_A=36，72，144，M_G=1440，720，288。

As seen in Fig. 14, the performance characteristics in the NT mode bear a strong resemblance to those in the NN mode, suggesting a common underlying cause. Despite varying sizes of M and K, the performance remains relatively stable. When K_A is set to 512, 1024, and 2048, the performance stabilizes at approximately 128, 145, and 154 GFlops. This translates to efficiencies of approximately 74%, 84%, and 89% of the hardware peak performance, respectively.

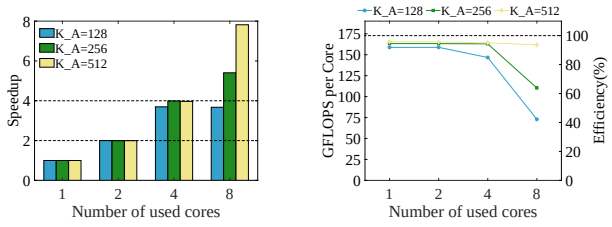
从图 14 中可以看出，NT 模式下的性能特征与 NN 模式下的非常相似，具有共同的潜在原因。尽管 M 和 K 有较大变化，但性能基本不变。当 K_A 设置为 512，1024，和 2048 时，性能分别稳定在约 128，145，和 154 GFlops，是硬件峰值性能的 74%，84%，和 89% 的效率。

C. Multi-core Performance

We evaluate the multi-core scalability of our GEMM implementations with different numbers of cores P, including 1, 2, 4, and 8.

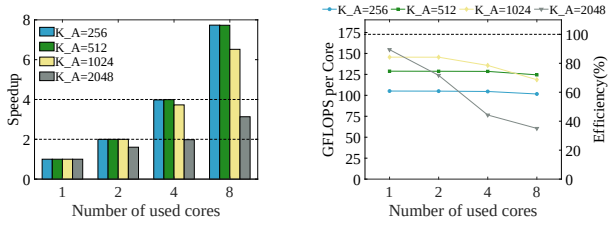
我们启动不同数量的 DSP 核心，用来评估我们 GEMM 实现的多核扩展性。

1) NN Mode: We evaluate the impact of different numbers of cores and various K_A values (128, 256, and 512), while keeping the input matrix size fixed



(a) Speedup (b) Performance per Core

图 15. The NN Mode performance on multiple DSP Cores (M=6048, K=2048, N_G=1536).



(a) Speedup (b) Performance per Core

图 16. The NT Mode performance on multiple DSP Cores (M=2880, K=2048, N_G=2048).

at M=6048, K=2048, and $N = 1536 \times P$. Fig. 15(a) shows the achieved speedup and Fig. 15(b) shows the average single-core performance in multi-core scenarios for different K_A values.

我们评估了不同的核数和不同的 K_A 值（128, 256, 和 512）对多核性能的影响，保持输入矩阵的大小固定为 M=6048, K=2048, 和 $N = 1536 \times P$。图 15(a) 显示了加速比，图 15(b) 显示了不同的 K_A 多核运行的平均单核性能。

We observe that a larger value of K_A ensures the GEMM implementations achieve linear speedup in multi-core scenarios. For instance, when K_A is set to 512, 8-core speedup is close to 8, indicating excellent scalability. When K_A is small, the scalability of GEMM is poor. This is because when K_A is set to 128 and 256, the execution time of the micro-kernel accounts for a small proportion of the total time. In this case, as the number of cores increases, multiple cores will contend the DDR bandwidth, resulting in a larger DMA transfer latency and decreased single-core performance. But when K_A is set to 512, the micro-kernel execution time is long enough, and the impact of increased DMA transfer latency is minimal.

我们观察到，较大的 K_A 值使得 GEMM 在多核并行计算时可以达到接近线性的加速比。例如，当 K_A 设置为 512 时，8 核并行的加速比接近 8，体现了优异的扩展性。当 K_A 很小时，GEMM 的多核扩展性很差。这是因为当 K_A 设置为 128 和 256 时，微内核的执行时间占总时间的比例小。这种情况下，随着核数的增加，多核将竞争 DDR 带宽，导致更大的 DMA 传输延迟，并显著降低单核的性能。但是当 K_A 设置为 512 时，微内核的执行时间足够长，DMA 传输延迟的影响很小。

2) NT Mode: We evaluate the impact of different numbers of cores and various K_A on the NT mode GEMM. We set K_A with values ranging from 256 to 2048 while keeping the input matrix size fixed at M=2880, K=2048, and $N = 2048 \times P$.

我们评估了不同的核数和不同的 K_A 取值在 NT 模式下对 GEMM 性能的影响。我们将 K_A 设置为从 256 到 2048 的不同取值，同时保持输入矩阵的大小固定为 M=2880，K=2048，和 $N = 2048 \times P$。

Fig. 16 demonstrates that the increase in K_A has a negative impact on scalability in the NT mode. When K_A is set to 256 or 512, we have observed a nearly linear speedup. When K_A becomes larger, e.g. K_A=1024, the speedup decreases to 3.73× in 4-core scenario and 6.52× in 8-core scenario. When K_A=2048, the scalability becomes even worse.

图 16 表明，K_A 的增加对 NT 模式的多核扩展性有负面影响。当 K_A 设置为 256 或 512 时，我们观察到了接近线性的加速比。当 K_A 变大，例如 K_A=1024，加速比在 4 核并行时仅为 3.73×，在 8 核并行时仅为 6.52×。当 K_A=2048 时，多核扩展性变得更差。

The reason for the performance decrease over the number of cores is the presence of the DMA transfer overhead. As K_A increases, N_A needs to be reduced due to the capacity constraint of AM. Since the **C** matrix is stored in row-major order, a smaller value of N_A results in poor continuity of data partitioning, leading to an increased latency of DDR access during DMA transmissions. Meanwhile, the rising number of cores linearly intensifies the competition for DDR bandwidth, causing each core to stop computing until its required data has been delivered. In this situation, it is impractical to fully hide the communication overhead, and a significant decline in the average single-core performance is inevitable.

性能随核数增加而降低是由 DMA 的传输开销导致的。随着 K_A 的增加，N_A 需要根据 AM 的容量限制设得更小。由于 **C** 矩阵是行主序存储，较小的 N_A 值会导致数据分块的连续性较差，使 DMA 传输过程中 DDR 访问的延迟增加。同时，核数的增加线性加剧了 DDR 带宽的竞争，导致每个核在其所需数据被传输到位之前只能停止计算。在这种情况下，完全隐藏通信开销是不切实际的，平均单核性能也就不可避免地显著下降。

D. Comparison with State-of-the-art

This section compares our approach to the prior work [13] known as ftIMM, which is targeted to irregular-shaped matrix multiplications. Given that ftIMM is in single-precision, we have to revise our approach by replacing those double-precision floating-point instructions in the micro-kernel with corresponding single-precision ones. In this case, a long vector

is composed of 32 32-bit scalars, and the theoretical performance is thus doubled.

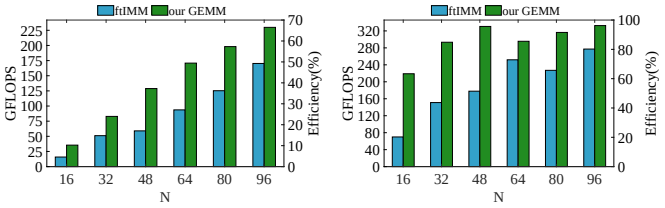
本节中，我们将我们的方法与之前的工作: ftIMM [13] 进行对比，该工作针对的是不规则矩阵乘法。考虑到 ftIMM 是单精度的，我们通过使用相应的单精度浮点指令替换微内核中的双精度浮点指令来修改我们的方法，用于单精度不规则矩阵乘。在这种情况下，一个长向量由 32 个 32 位标量组成，理论性能因此翻倍。

With a range of input matrices of varying shapes, the performance in the single-core scenario is shown in Fig. 17. When M is large but N and K are small, although both implementations have poor performance, our approach is 35%~124% better than ftIMM. As N and K increase from 16 to 96, the performance of our implementation improves significantly, rising from 10% to 67% of the hardware peak. In the case that M and N are both large (i.e. 20480), our approach achieves more than 80% of the hardware peak performance when N exceeds 32, and even reaches 95% when N=48 or 96, which is 20% faster than ftIMM.

对于不同形状的一系列输入矩阵，单核场景下的性能如图 17 所示。当 M 很大但 N 和 K 很小时，尽管两种实现的性能都很差，但我们的方法比 ftIMM 好 35%~124%。随着 N 和 K 从 16 增加到 96，我们的方法性能显著提高，从硬件峰值的 10% 上升到 67%。在 M 和 N 都很大（即 20480）的情况下，当 N 超过 32 时，我们的方法可以达到硬件峰值性能的 80% 以上，甚至当 N=48 或 96 时，可以达到 95%，比 ftIMM 快 20%。

Considering a typical feature of the input matrices in [13] that the N dimension is very short, we adjust our tiling strategy in the multi-core scenario. Specifically, we partition the **A** and **C** matrices along the M dimension, and distribute these submatrices to each core for computation. This adjustment can leverage the potential of multi-core scalability due to a more regular data shape. The actual performances are shown in Fig. 18. When M is large but N and K are small (shown in Fig. 18(a)(c)), the performance of these two implementations are poor, both below 20% of the hardware peak. This is because the small data size reduces the computation-to-memory ratio, which leads to severe bottleneck of DDR memory access in the case of multi-core bandwidth contention. Nevertheless, our implementation still runs around 40 GFlops faster than ftIMM. When M and K are both large (shown in Fig. 18(b)(d)), the parallel performance of both methods increases significantly with the increase of N. As N increases to 96, Our approach achieves 57% of the hardware peak performance, which is 7.8% better than ftIMM.

考虑到 [13] 中输入矩阵的一个典型特征是 N 维度很短，因此在多核并行场景下我们调整了核间的数据分块策略。具体来说，我们沿着 M 维度切分 **A** 和 **C** 矩阵，并将切分后的子矩阵分配给每个核进行计算。这样调整可以改善多核扩展性，因为数据形状更加规则。实际性能如图 18 所示。当 M 很大但 N 和 K 很小时（如图 18(a)(c) 所示），这两种实现的性能都很差，都低于硬件峰值的 20%。这是因为过小的数据规模降低了计算访存比，导致在多核竞争带宽的情况下 DDR 访存遇到了严重的瓶颈。尽管如此，我们的实现仍然比 ftIMM 要快大约 40 GFlops。当 M 和 K 都很大时（如图 18(b)(d) 所示），随着 N 的增加，两种方法的并行性能都显著提高。当 N 增加到 96 时，我们的方法达到了硬件峰值性能的 57%，比 ftIMM 好 7.8%。



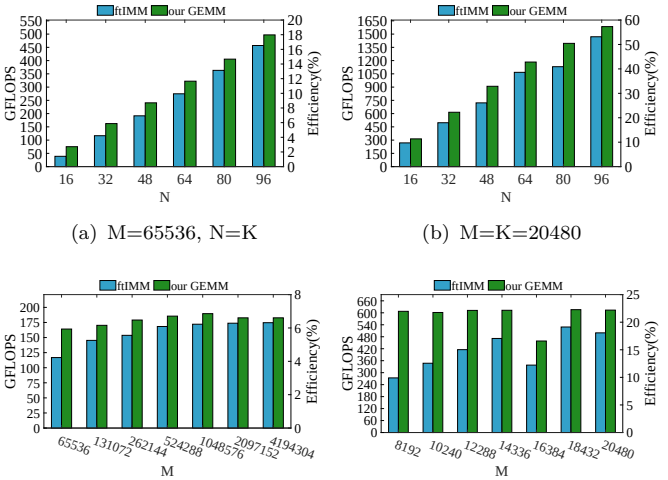
(a) M=65536, N=K (b) M=K=20480

图 17. The performance of single-precision irregular-shaped matrix multiplication in the single-core scenario.

E. Evaluation of Optimization Techniques

We assess the impact of our multilevel pipelining and vector reduction strategies on GEMM performance. We first compare the performance between double-buffering and triple-buffering for matrix **C**. We find that the double-buffering approach yields performance below 157 GFlops, while our triple-buffering approach can consistently achieve performance above 165 GFlops. This proofs that our approach achieves a better overlap between computation and data movement, further improving the per-core performance. We then evaluate the effectiveness of our proposed vector reduction algorithm. The micro-kernel using the previous vector reduction method only reaches 39% of the hardware peak performance. However according to Fig. 12, our vector reduction approach can achieve up to 79% of the hardware peak performance, showcasing its notable enhancement.

我们评估了我们的多级流水线和向量规约方法对 GEMM 性能的影响。我们首先比较了双缓冲策略和三缓冲策略用于矩阵 **C** 后的性能。我们发现双缓冲策略的性能低于 157 GFlops，而我们的三缓冲策略可以稳定地达到 165 GFlops 以上的性能。这证明了我们的方法能够更好地重叠计算和数据移动，并进一步提高了每个 DSP 核心的性能。然后，我们评估了我们提出的向量规约算法的有效性。使用之前的向量规约方法的微内核只能达到硬件峰值性能的 39%。但是根据图 12可知，我们的向量规约方法可以达到硬件峰值性能的 79%，称得上是效果显著的改进。



(a) M=65536, N=K (b) M=K=20480

(c) N=K=32 (d) N=32, M=K

图 18. The performance of single-precision irregular-shaped matrix multiplication in the 8-core scenario.

VI. Related Work

GEMM is a critical software component in HPC applications. Various implementations of GEMM algorithms have been proposed to improve its performance. Agarwal *et al.* [21] optimized GEMM implementation with a double-buffering mechanism, which reduces the overhead of memory access. Lee *et al.* [22] introduced the Cannon algorithm, specifically designed for toroidal interconnected systems to achieve high memory utilization.

GEMM 是 HPC 应用中的一个关键子程序。为了提高它的性能，各种 GEMM 算法的实现方式被提了出来。Agarwal *et al.* [21] 用双缓冲机制优化了 GEMM 的实现，减少了存储器的访问开销。Lee *et al.* [22] 引入了 Cannon 算法，专门用于环形互连系统，以实现更高的存储器利用率。

The GotoBLAS approach by Goto *et al.* [17] has become a widely adopted modern implementation of GEMM. Libraries like OpenBLAS [19] and BLIS [20] are designed based on this approach. Smith *et al.* [23] systemically parallelize this approach in many-threaded architectures, leading to excellent scalability. Heinecke *et al.* [24] and Guney *et al.* [25] apply the GotoBLAS approach on the Intel Xeon Phi (KNC) processor and the KNL processors.

Goto *et al.* [17] 提出的 GotoBLAS 方法已经成为了一种被广泛采用的现代 GEMM 实现。像 OpenBLAS [19] 和 BLIS [20] 这样的库都是基于这种方法设计的。Smith *et al.* [23] 在多线程体系结构中系统地实现了 GotoBLAS 的并行化，并具备优异的扩展性。Heinecke *et al.* [24] 和 Guney *et al.* [25] 将 GotoBLAS 方法应用在了 Intel Xeon Phi (KNC) 处理器和 KNL 处理器上。

Many research efforts have focused on optimizing GEMM implementations for specific processor architectures based on the GotoBLAS approach. Jiang *et al.* [26] explore the micro-architecture of the heterogeneous many-core SW26010 processor to optimize DGEMM performance. They employed a three-level data blocking algorithm to manage the memory hierarchy and adopted an architecture-aware approach to enhance data locality. Smith *et al.* [23] design a parallelization method for BLIS based on the multi-threaded architectures of Intel Xeon Phi and IBM PowerPC A2. Volko *et al.* [7] perform a thorough benchmarking on GPUs, providing new insights into GPU programming. They implemented GEMM, with a better performance than the vendor libraries.

许多研究工作都致力于基于 GotoBLAS 方法为特定的处理器架构优化 GEMM 的实现。Jiang *et al.* [26] 面向异构多核 SW26010 处理器优化了 DGEMM 的性能。他们采用了三级数据分块算法来管理存储器层次，并采用了一种感应架构的方法来提高数据局部性。Smith *et al.* [23] 基于 Intel Xeon Phi 和 IBM PowerPC A2 的多线程架构，为 BLIS 设计了一种并行化方法。Volko *et al.* [7] 在 GPU 上进行了全面的基准测试，提供了关于 GPU 编程的新见解。他们实现的 GEMM 性能优于厂商提供的数学库。

There are several efforts on optimizing GEMM on DSPs. Igual *et al.* [11] investigate the micro-architecture of TI DSPs and optimizes GEMM using a BLAS-based programming approach. Despite careful consideration of data placement, their implementation fails to fully overlap computation and communication. Besides, they compare the power consumption with other HPC platforms, and find that DSPs have significant energy-saving effects. Different from their research purpose, our study only focuses on optimizing GEMM for better performance, without considering power consumption. Yin *et al.* [13] propose ftIMM to the FT-M7032 platform, as a solution for single-precision irregular-shaped matrix multiplication. ftIMM is capable of automatically generating micro-kernels and dynamically adjusting blocking parameters and parallel strategies. Different from their work, we focus on optimizing GEMM to the extreme for DSP architectures, specifically in the context of double-precision large-scale GEMM, by introducing a new computation pipeline and a dedicated micro-kernel. Furthermore, we have revised our GEMM implementation and compared its performance with the ftIMM in Section V. The result has shown that our approach can yield a much better performance even in the case of irregular-shaped matrix multiplications.

有一些工作致力于在 DSP 上优化 GEMM。Igual *et al.* [11] 面向 TI DSP 架构使用基于 BLAS 的编程方法

优化了 GEMM。尽管他们对数据的缓存方式做了仔细的考虑，但并没有完全实现计算和通信的重叠。此外，他们还与其他 HPC 平台进行了功耗对比，发现 DSP 有显著的节能效果。与他们的研究目的不同，我们的研究只关注如何优化 GEMM 以获得更好的性能，而不考虑功耗问题。Yin *et al.* [13] 提出了 ftIMM 方法，用于作为 FT-M7032 平台的单精度不规则矩阵乘的解决方案。ftIMM 能够自动生成微内核，并动态调整分块参数和并行策略。与他们的工作不同，我们专注于面向 DSP 架构的 GEMM 极致性能优化，特别是针对双精度大规模 GEMM 的应用场景，为此引入一种新的计算流水线和一个专用的微内核。此外，我们还略加修改了我们的 GEMM 实现，并在第 V 节中与 ftIMM 进行了性能比较。结果表明，即使用于计算不规则矩阵乘法，我们的方法也能获得更好的性能。

VII. Conclusions

This paper shares our experience in optimizing GEMM on FT-M7032 DSP. Our approach incorporates a three-level pipelining strategy to overlap data movement with computation and a data partitioning method to enhance memory access patterns. We also introduce improved micro-kernels with a better vector reduction strategy to fully exploit the potential of SIMD features. This method effectively reduces non-vectorization computation overhead, thereby improving the overall performance of the micro-kernel. The micro-kernel shape is determined by a series of analytical models, and we carefully design the assembly pipeline with perfect instruction-level parallelism. Extensive experimental results demonstrate that the performance attained by our optimization is up to 96% of the theoretical peak performance of the hardware.

这篇论文分享了我们在 FT-M7032 DSP 上优化 GEMM 的经验。我们的方法采用了一种三级流水线策略，实现了计算和数据搬运的重叠，并通过一种数据分块方法改善了访存模式。此外，我们引入了手工优化的微内核，它充分利用平台的 SIMD 特性改良了长向量规约方法。这种方法有效减少了非向量化计算的开销，提升了微内核的整体性能。我们用一系列分析模型来确定微内核的形状，并基于此设计了最大化指令集并行的汇编流水线。广泛的实验结果表明，经过我们的优化，GEMM 的性能可以达到硬件峰值性能的 96%。

参考文献

[1] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proc. SC*, pages 981–991, 2016.

[2] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Proc. ICFHR*, pages 1–7, 2006.

[3] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proc. SC*, pages 66:1–66:12, 2018.

[4] Evangelos Georganas, Kunal Banerjee, Dhiraj D. Kalamkar, Sasikanth Avancha, Anand Venkat, Michael J. Anderson, Greg Henry, Hans Pabst, and Alexander Heinecke. Harnessing deep learning via a single building block. In *Proc. IPDPS*, pages 222–233, 2020.

[5] Feng Wang, Hao Jiang, Ke Zuo, Xing Su, Jingling Xue, and Canqun Yang. Design and implementation of a highly efficient dgemm for 64-bit armv8 multi-core processors. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP '15, page 200–209, USA, 2015. IEEE Computer Society.

[6] Arslan Munir, Ann Gordon-Ross, and Sanjay Ranka. *High-Performance Optimizations on Tiled Manycore Embedded Systems: A Matrix Multiplication Case Study**, pages 287–342. 2016.

[7] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08. IEEE Press, 2008.

[8] Vasilios Kelefouras, A. Kritikakou, Iosif Mporas, and Vasilios Kolonias. A high-performance matrix—matrix multiplication methodology for cpu and gpu architectures. *J. Supercomput.*, 72(3):804–844, mar 2016.

[9] Rajib Nath, Stanimire Tomov, Tingxing "Tim" Dong, and Jack Dongarra. Optimizing symmetric dense matrix-vector multiplication on gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Cody Rivera, Jieyang Chen, Nan Xiong, Jing Zhang, Shuaiwen Leon Song, and Dingwen Tao. Tsm2x: High-performance tall-and-skinny matrix–matrix multiplication on gpus. *Journal of Parallel and Distributed Computing*, 151:70–85, 2021.

[11] Francisco D. Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert A. van de Geijn. Unleashing the high-performance and low-power of multi-core dsps for general-purpose HPC. In Jeffrey K. Hollingsworth, editor, *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 26. IEEE/ACM, 2012.

[12] Yaohua Wang, Chen Li, Chang Liu, Sheng Liu, Yuanwu Lei, Jian Zhang, Yang Zhang, and Yang Guo. Advancing DSP into hpc, ai, and beyond: challenges, mechanisms, and future directions. *CCF Trans. High Perform. Comput.*, 3(1):114–125, 2021.

[13] Shangfei Yin, Qinglin Wang, Ruochen Hao, Tianyang Zhou, Songzhu Mei, and Jie Liu. Optimizing irregular-shaped matrix-matrix multiplication on multi-core dsps. In *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*, pages 451–461. IEEE, 2022.

[14] Sheng Ma, Zhong Liu, Shenggang Chen, Libo Huang, Yang Guo, Zhiying Wang, and Meidi Zhang. Coordinated DMA: improving the DRAM access efficiency for matrix multiplication. *IEEE Trans. Parallel Distributed Syst.*, 30(10):2148–2164, 2019.

[15] Chao Yang, Shuming Chen, Jian Zhang, Zhao Lv, and Zhi Wang. A novel DSP architecture for scientific computing and deep learning. *IEEE Access*, 7:36413–36425, 2019.

[16] Jianbin Fang, Peng Zhang, Chun Huang, Tao Tang, Kai Lu, Ruibo Wang, and Zheng Wang. Programming bare-metal accelerators with heterogeneous threading models: a case study of matrix-3000. *Frontiers Inf. Technol. Electron. Eng.*, 24(4):509–520, 2023.

[17] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), may 2008.

[18] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1), jul 2008.

[19] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, page 684–691, USA, 2012. IEEE Computer Society.

[20] Field G. Van Zee and Robert A. van de Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Trans. Math. Softw.*, 41(3), jun 2015.

[21] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[22] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. Generalized cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, page 44–51, New York, NY, USA, 1997. Association for Computing Machinery.

[23] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1049–1059, 2014.

[24] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Anirudha G. Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 126–137, 2013.

[25] Murat Efe Guney, Kazushige Goto, Timothy B. Costa, Sarah Knepper, Louise Huot, Arthur Mitrano, and Shane Story. Optimizing matrix multiplication on intel® xeon phi th x200 architecture. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 144–145, 2017.

[26] Lijuan Jiang, Chao Yang, Yulong Ao, Wanwang Yin, Wenjing Ma, Qiao Sun, Fangfang Liu, Rongfen Lin, and Peng Zhang. Towards highly efficient dgemm on the emerging sw26010 many-core processor. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 422–431, 2017.