CPU 架构下的 GEMM 软硬件优化

姓名：　　学号：
上海交通大学微纳电子学系

日期：2024 年 12 月 24 日

1　前言

通用矩阵乘法（General Matrix Multiplication, GEMM）作为一种重要的线性代数操作，是科学计算、高性能计算以及深度学习等领域的核心计算任务之一。无论是在深度神经网络的前向传播、后向传播过程中，还是在数值模拟与数据处理等高性能计算应用中，GEMM 操作的效率都直接影响整个系统的性能表现。然而，GEMM 的高计算密集性和内存带宽需求对计算硬件的架构设计提出了极高的挑战，尤其是在面对现代 CPU 复杂的缓存层次结构和指令流水线优化时，性能瓶颈的暴露往往显得尤为突出。

本实验基于 gem5 仿真平台，探讨特定 CPU 架构下的 GEMM 软硬件优化策略。gem5 作为一个灵活、模块化的计算机体系结构仿真工具，能够模拟微架构细节，包括缓存行为、指令执行路径和分支预测等，为性能分析和优化提供强有力的支持。通过在 gem5 环境中搭建二级缓存的 CPU 系统，并结合对 GEMM 算法的改进，本实验旨在提升 GEMM 性能，分析优化的关键因素，并为实际硬件设计提供参考。

实验的意义在于，通过仿真与优化实践，不仅加深对 CPU 架构特性和 GEMM 操作的理解，还能够为高性能计算领域提供具有指导意义的优化方案。

2　gem5 环境搭建

本实验采用 Docker Desktop 部署 gem5 运行环境，以提高配置的便捷性和一致性。使用的镜像为 gem5 官方提供的 ghcr.io/gem5/ubuntu-20.04_all-dependencies:v23-0 版本，该镜像预装了运行 gem5 所需的依赖和工具。通过 Docker 技术，可以快速拉取镜像并创建容器，同时将本地工作目录与容器挂载，方便实验数据的管理与持久化。

基于该镜像的环境配置简化了传统的安装流程，不需要手动解决依赖冲突或编译问题。镜像内的操作系统基于 Ubuntu 20.04，适配性强且支持广泛的工具和库，为实验提供了灵活的开发环境。搭建完成后，用户即可在容器内运行 gem5，进行仿真和性能分析，为后续的 GEMM 优化实验奠定基础。

3 Baseline 的具体实现

3.1 硬件架构

本实验的 Baseline 硬件架构基于典型的两级缓存系统设计，如下图所示：

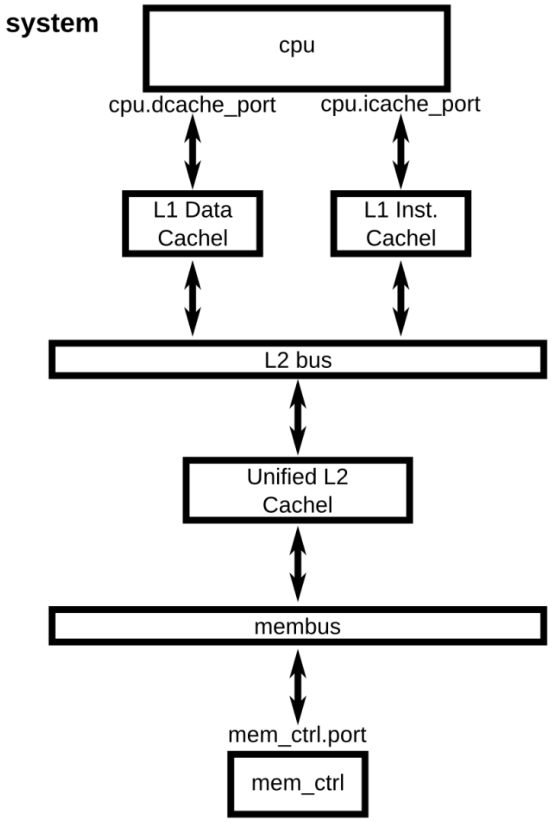

图 1: Baseline 硬件架构

- **CPU：**CPU 是系统的核心计算单元，通过两个独立的端口连接到一级缓存：`cpu.dcache_port` 负责数据访问，`cpu.icache_port` 负责指令访问。
- **一级缓存（L1 Cache）：**一级缓存分为数据缓存（L1 Data Cache）和指令缓存（L1 Inst. Cache），分别用于存储数据和指令，以减少 CPU 访问主存的延迟。L1 缓存与 CPU 直接连接，具有较低的访问延迟。
- **L2 总线（L2 Bus）：**一级缓存与二级缓存通过 L2 总线相连，用于数据和指令在不同缓存级别之间的传输。该总线为系统内存层次结构提供了高效的数据交互路径。
- **二级缓存（Unified L2 Cache）：**二级缓存为统一缓存结构，既存储数据又存储指令。相比一级缓存，二级缓存容量更大但访问延迟稍高，主要用于提高缓存命中率，减少对主存的访问需求。
- **内存总线（membus）：**二级缓存与主存控制器通过内存总线连接。内存总线在缓存失效时将数据请求转发到主存，完成数据的加载或写回。
- **主存控制器（mem_ctrl）：**主存控制器负责管理与主存之间的交互，协调数据的读取与写入操作。其 `mem_ctrl.port` 与内存总线相连，为 CPU 提供对主存的最终访问路径。

该架构的特点是采用分级缓存系统，通过 L1 缓存与 L2 缓存的组合减少内存访问延迟，提高系统性能。同时，统一的二级缓存设计简化了硬件结构，适合于多种应用场景的仿真与性能分析，是一个广泛使用的基准硬件配置。

我们将 CPU 设置为基于顺序执行的 X86TimingSimpleCPU，l1 dcache 的大小设为 4kB，l2 cache 的大小设为 16kB，时钟信号为 1GHz。

3.2 软件实现

作为 Baseline 的 GEMM 软件实现采取了最简单的 3 重循环的方式来计算。即通过三重嵌套的 for 循环依次计算矩阵中每个元素的值。这种方法虽然实现简单，但性能较为低效，因为它没有利用现代处理器的缓存结构和并行计算能力。在该实现中，内存访问模式通常是按行或按列顺序依次读取和写入数据，这可能导致频繁的缓存未命中，影响执行效率。因此，在优化 GEMM 计算时，通常会考虑通过矩阵分块、转置存储等技术来提高性能。在本次实验中 A、B、C 均为 128*128 大小的矩阵，即 (M, N, K)=(128, 128, 128) 。

具体代码如下所示：

```
#define SIZE 128
// Matrix multiplication A * B = C
void matmul(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE])
    {
    int i, j, k;

    // Matrix multiplication operation
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            for (k = 0; k < SIZE; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

4 优化策略的具体实现

4.1 软件优化

4.1.1 循环展开

首先我们考虑到，在原始的三重循环中，每次迭代都需要进行条件判断和跳转操作，这会带来一定的性能开销。这是因为 CPU 遇到分支指令时，要么中断流水线，造成性能损失；要么

为了不中断流水线，进行分支预测，而预测错误就需要冲刷掉多条已经执行的指令，也会造成性能的损失。通过展开循环，可以减少这些跳转指令的数量，因为同一层次的循环可以在一个单一的迭代中处理多个操作，从而减小了循环控制的频率。

具体的代码如下所示：

```
#define SIZE 128
// Matrix multiplication A * B = C
void matmul_loop_unroll(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[
    SIZE][SIZE]) {
     int i, j, k;

     // Matrix multiplication operation
     for (i = 0; i < SIZE; i++) {
         for (j = 0; j < SIZE; j++) {
             for (k = 0; k < SIZE; k=k+8) {
                 C[i][j] += A[i][k] * B[k][j];
         C[i][j] += A[i][k+1] * B[k+1][j];
         C[i][j] += A[i][k+2] * B[k+2][j];
         C[i][j] += A[i][k+3] * B[k+3][j];

         C[i][j] += A[i][k+4] * B[k+4][j];
         C[i][j] += A[i][k+5] * B[k+5][j];
         C[i][j] += A[i][k+6] * B[k+6][j];
         C[i][j] += A[i][k+7] * B[k+7][j];

             }
         }
     }
}
```

4.1.2 矩阵分块

对于 128*128*128 的大矩阵 GEMM 而言，直接计算的话，会导致由于 B 矩阵的列数据频繁从内存加载到缓存，而新的数据又不断替换旧的数据，导致缓存抖动。因此，我们可以采用矩阵分块的方法去缓解这一现象。矩阵分块后，计算时只需要访问子块中的数据，较少触及整个大矩阵。这样，连续的数据访问模式能够更好地利用缓存，提升了空间局部性。同时，分块算法使同一块数据在短时间内被多次访问，从而增加了缓存命中的概率，减少了代价更大的主存访问，提升了时间局部性。考虑到我们的 l1 dcache 的大小为 4kB，而 int 类型的数据大小为 4B，因此可以将矩阵的分块大小设为 32*32，来契合 cache 的大小。

具体的代码如下所示：

```
#define SIZE 128
#define BLOCK_SIZE 32  // Use the sub block size of BLOCK_SIZE x
   BLOCK_SIZE
// Matrix multiplication A * B = C
void matmul_block(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][
   SIZE]) {
    int i, j, k, ii, jj, kk;

    // Block matrix multiplication
    for (ii = 0; ii < SIZE; ii += BLOCK_SIZE) {
        for (jj = 0; jj < SIZE; jj += BLOCK_SIZE) {
            for (kk = 0; kk < SIZE; kk += BLOCK_SIZE) {
                for (i = ii; i < ii + BLOCK_SIZE && i < SIZE; i++) {
                    for (j = jj; j < jj + BLOCK_SIZE && j < SIZE; j
                       ++) {
                        for (k = kk; k < kk + BLOCK_SIZE && k < SIZE
                           ; k++) {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
                }
            }
        }
    }
}
```

4.1.3 转置存储

在常规的 GEMM 计算过程中，计算矩阵 C 的每个元素时，通常需要获取矩阵 A 的某一行数据和矩阵 B 的对应列数据，进行逐元素相乘并累加的操作。然而，由于计算机内存中的二维数组实际上是按照一维数组的线性方式存储的，矩阵 A 按行访问时，其数据存储是连续的，可以充分利用内存的空间局部性，提高缓存命中率。然而，矩阵 B 按列访问时，由于列内数据存储在内存中的地址通常不连续，可能导致频繁的缓存未命中，从而影响性能。因此，为了优化矩阵 B 的访问模式，可以在计算前对矩阵 B 进行转置存储。通过转置操作，将原本需要按列访问的数据重新排列为按行访问的形式，使得每次计算时矩阵 B 的数据访问变为按行读取。这种调整能够将原本不连续的内存访问模式转换为连续访问，从而显著提高其空间局部性，减少缓存未命中率，提升整体计算性能。

具体代码如下所示：

```
#define SIZE 128
// Matrix multiplication A * B = C
void matmul_transpose(int A[SIZE][SIZE], int B[SIZE][SIZE], int C[
    SIZE][SIZE]) {
     int i, j, k;

     // Matrix multiplication operation
     for (i = 0; i < SIZE; i++) {
         for (j = 0; j < SIZE; j++) {
             for (k = 0; k < SIZE; k++) {
                 C[i][j] += A[i][k] * B[j][k];
             }
         }
     }
}
```

4.1.4 外积计算

传统的 GEMM 计算方法采用内积的方式，即将矩阵 C 中的每个元素视为矩阵 A 的某一行向量与矩阵 B 的对应列向量的内积结果。然而，矩阵 C 也可以通过外积的方式进行构建。具体而言，外积计算将矩阵 A 的第 i 列（列向量）与矩阵 B 的第 i 行（行向量）相乘，生成一个大小为 128*128 的矩阵（即外积矩阵）。这种计算会依次进行 128 次，产生 128 个外积矩阵。最终，将这些外积矩阵按对应位置逐元素相加即可得到矩阵 C。这种方法等价于逐步累积所有外积的结果，从而构建出最终的矩阵 C。这样做的好处在于，对于 A 的每列数据和 B 的每行数据，都只需要参与 1 次计算，而传统的 GEMM 计算方法中需要 128 次，造成频繁跨行或跨列的数据访问。因此外积的计算方式减少了直接访问大矩阵 A 和 B 的频率，从而优化了 GEMM 的效率。

具体代码如下所示：

```
#define SIZE 128

// Outer product of two vectors
void outer_product(int A_col[SIZE], int B_row[SIZE], int C[SIZE][
    SIZE]) {
     for (int i = 0; i < SIZE; i++) {
         for (int j = 0; j < SIZE; j++) {
             C[i][j] += A_col[i] * B_row[j];
         }
     }
```

```
}

// Matrix multiplication A * B = C using outer product
void matmul_outer_product(int A[SIZE][SIZE], int B[SIZE][SIZE], int
    C[SIZE][SIZE]) {
  int A_col[SIZE];
  int B_row[SIZE];

  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      A_col[j] = A[j][i];
      B_row[j] = B[i][j];
    }
    outer_product(A_col, B_row, C);
  }
}
```

4.2 硬件优化

4.2.1 缓存参数调整

从硬件优化的角度来考虑，调整 L1 数据缓存（l1 dcache）和 L2 缓存（l2 cache）的大小是一项重要的优化策略。这种优化的核心目标是充分利用缓存层次结构，提高数据访问的局部性，降低主存访问延迟，从而提升整体性能。

GEMM 涉及大量的矩阵元素重复访问，具有显著的数据复用特性。默认配置下，Baseline 的 l1 dcache 仅有 4kB，而 l2 cache 为 16kB，这种容量限制可能导致数据块频繁溢出缓存，增加主存的访问次数。在这种情况下，扩大 l1 dcache 和 l2 cache 的容量能够更好地容纳工作集，减少缓存失效率。

由于本次实验所计算的 GEMM 规模为 128*128*128，所以很显然，通过调整缓存参数来提升 GEMM 性能的做法是有极限的，当缓存大到一定程度（例如可以将矩阵完全读取进 cache）之后，GEMM 的性能就不会有明显的提升了。

在本次实验中，我们首先尝试调整 l2 cache 的大小，固定 l1 dcache 的大小。将其从 16kB 逐渐增大到 128kB。当收效变得不明显后，我们再调整 l1 dcache 的大小，将其从 4kB 逐渐增大到 64kB，直到收效也不明显为止。

可以通过调整命令中的参数来设置缓存的大小，例如，对于 Baseline 的设置命令如下：

```
build/X86/gem5.opt configs/tutorial/part1/two_level.py tests/test-
    progs/gemm/src/gemm --l2_size='16kB' --l1d_size='4kB'
```

4.2.2 缓存替换策略 (Optional)

我们 Baseline 的缓存替换策略是 LRU。这是因为在 gem5 中，BaseCache 类默认设置的缓存替换策略（Replacement Policy）是 LRU。如下所示的是 gem5/src/mem/cache 路径下的 Cache.py 文件内的相关设置：

```
replacement_policy = Param.BaseReplacementPolicy(
    LRURP(), "Replacement policy"
)
```

同时，在 gem5/src/mem/cache/replacement_policies 路径下的 ReplacementPolicies.py 文件内定义了 gem5 自带的一些缓存替换策略，包括但不限于以下的策略：

- **随机替换策略（Random）：**最简单的替换策略；它不需要替换数据，因为它会在候选条目中随机选择一个受害者进行替换。
- **最近最少使用（Least Recently Used, LRU）：**替换数据由“最后访问时间戳”组成，受害者是基于时间戳选择的：时间戳越旧，该条目越可能被替换。
- **伪最近最少使用（Tree Pseudo Least Recently Used, TreePLRU）：**一种 LRU 的变体，通过使用二叉树和 1 位指针记录条目的最近使用情况。
- **双模插入策略（Bimodal Insertion Policy, BIP）：**双模插入策略类似于 LRU，但块被插入为“最近最常使用”（MRU）的概率由双模节流参数（btp）控制。btp 越大，新块作为 MRU 插入的可能性越高。
- **LRU 插入策略（LRU Insertion Policy, LIP）：** LRU 插入策略是一种 LRU 替换策略，但它将块插入为“最近最少使用”（LRU）条目，而不是具有最新时间戳的条目。在块后续被访问时，其时间戳会更新为“最近最常使用”（MRU），与 LRU 类似。可以将其视为一种特殊的 BIP，其中新块作为 MRU 插入的概率为 0%。
- **最近最常使用（Most Recently Used, MRU）：**最近最常使用策略根据条目的最近使用情况选择受害者，与 LRU 相反，条目越新，其被替换的可能性越大。
- **最少使用（Least Frequently Used, LFU）：**受害者是基于引用频率选择的。引用次数最少的条目将被驱逐，而与访问次数或上次访问的时间无关。
- **先进先出（First-In, First-Out, FIFO）：**受害者是基于插入时间戳选择的。如果不存在无效条目，则最早插入的条目将被替换，而与访问次数无关。
- **二次机会（Second-Chance）：**二次机会替换策略类似于 FIFO，但条目在被替换之前会被赋予“第二次机会”。如果一个条目是下一个替换目标，但其“第二次机会位”已设置，则清除此位并将条目重新插入 FIFO 的末尾。在发生未命中时，条目以其“第二次机会位”清除状态插入。
- **近期未使用（Not Recently Used, NRU）：**近期未使用策略是一种 LRU 的近似算法，使用单个位来判断一个块是否会在近期或远期被再次引用。如果该位为 1，则该块可能不会很快被引用，因此被选为替换受害者。当一个块被替换时，其所有的替换候选条目的“再次引用位”都会增加。
- **再次引用间隔预测（Re-Reference Interval Prediction, RRIP）：**再次引用间隔预测是一种

NRU 的扩展算法，它使用再次引用预测值（RRPV）来判断块是否会在近期被再次引用。RRPV 的值越高，块距离下次访问的时间越远。从原始论文来看，该实现被称为静态 RRIP（Static RRIP, SRRIP），因为它总是以相同的 RRPV 值插入块。

- **双模再次引用间隔预测（Bimodal Re-Reference Interval Prediction, BRRIP）：**双模再次引用间隔预测是 RRIP 的扩展，它有一定概率不将块插入为“最近最少使用”（LRU），类似于双模插入策略（BIP）。这种概率由双模节流参数（btp）控制。

我们同样可以自定义实现非 gem5 自带的缓存替换策略。在本次实验中，我们自定义了如下所示的两种缓存替换策略：代际替换策略（Generational Replacement Policy, GenRP）和鹰眼缓存替换策略（Hawkeye Cache Replacement Policy, HawkeyeRP）。

- **GenRP：** GenRP 是一种基于缓存条目访问的" 代" 来进行替换的策略。它的核心思想是将缓存条目分配到不同的代，每个代的缓存条目会被赋予不同的优先级，以此来决定替换的顺序。
- **HawkeyeRP：** HawkeyeRP 是一种基于预测的替换策略，它结合了访问历史和预测模型来确定最合适的替换受害者。该策略通过分析访问模式，特别是基于访问间隔的预测，来优化数据块的替换决策。

在 gem5/src/mem/cache/replacement_policies 路径下，以 GenRP 为例，自定义缓存策略的流程如下：

1. 实现 gen_rp.cc 和 gen_rp.hh 文件，自行编写 invalidate(), touch(), reset(), getVictim() 等函数逻辑以及声明；
2. 在 SConscript 脚本文件内修改/添加如下代码：

```
    SimObject('ReplacementPolicies.py', sim_objects=[
    'BaseReplacementPolicy', 'DuelingRP', ... , 'GenRP'])

    Source('gen_rp.cc')
```

3. 在 ReplacementPolicies.py 的最后添加如下代码：

```
class GenRP(BaseReplacementPolicy):
    type = 'GenRP'
    cxx_class = 'gem5::replacement_policy::GenRP'
    cxx_header = "mem/cache/replacement_policies/gen_rp.hh"
```

4. 在 terminal 中运行如下命令重新编译 X86 指令集。

```
 scons build/X86/gem5.opt -j {cpus}
```

5. 在自定义的 caches.py 文件中给 L1Cache 类与 L2Cache 类添加如下代码，之后重新运行仿真即可。

```
replacement_policy = GenRP()
```

在本次实验中，除了作为 Baseline 的 LRU 策略，我们还测试了如上所示的其他 11 种 gem5 自带策略以及自定义的 GenRP 和 HawkeyeRP 对 GEMM 性能的影响。

4.2.3 RISC-V 架构 (Optional)

Baseline 的硬件架构使用的是基于 X86 指令集的 CPU，是 CISC 指令集架构。我们可以将 CPU 替换为基于 RISC-V 指令集的 CPU（RiscvTimingSimpleCPU），将其做对比，探究 CISC 和 RISC 在 GEMM 这一应用上的差异。

值得注意的是，在将 CPU 架构更换为 RISC-V 之后，需要更换工具链对软件代码重新进行编译。在使用 X86 架构时，编译的命令是：

```
gcc gemm.c -o gemm -static
```

而在使用 RISC-V 架构时，由于 Docker 内缺乏 riscv64-linux-gnu-gcc 工具链，所以需要先安装，命令是：

```
apt install gcc-riscv64-linux-gnu
```

然后进行编译，命令是：

```
riscv64-linux-gnu-gcc -o gemm gemm.c  – static
```

4.2.4 乱序执行模型 (Optional)

在 Baseline 中，我们使用的 X86TimingSimpleCPU 是顺序执行（In Order）模型，我们可以将其更换为乱序执行（Out of Order）模型进行对比，探究不同处理器特性对性能的影响。而 O3CPU（也称为 DerivO3CPU）是一个功能强大的乱序执行 CPU 模型，模拟了指令、功能单元、内存访问和流水线阶段之间的依赖关系，适用于 SE 模式。我们可以将 CPU 更换为 O3CPU 来运行仿真分析，如下所示：

```
system.cpu = DerivO3CPU()
```

5 优化策略的性能分析

优化策略的性能分析主要使用的是 gem5/m5out 文件夹下的 config.ini 与 stats.txt 文件。其中，config.ini 文件是 gem5 的主配置文件，包含了各种参数和设置，用于定义模拟器如何运行。而 stats.txt 文件存储模拟器运行时的性能统计信息，用于分析和研究系统性能行为。

5.1 软件优化的性能分析

首先，我们进行运行时间的分析。我们使用 time.h 头文件中的 clock_t 类对矩阵计算的过程进行计时，控制变量，从而得到如下的结果。可以观察到，循环展开方法收效甚微，这是因为对于简单 CPU 而言，虽然减少了跳转指令的数目，却无法充分利用展开的指令间的并行性，所以尽管有收益，但是并不明显。而矩阵分块方法实现了约 20% 的速度提升，转置存储与外积计算实现了约 40% 的速度提升。

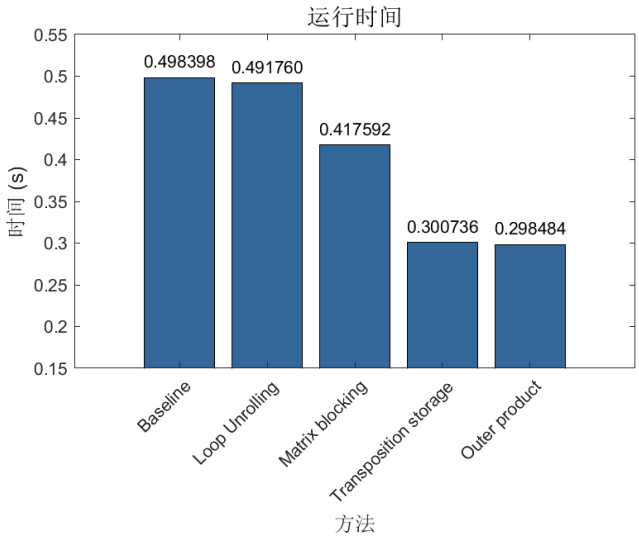

图 2: 软件优化运行时间分析

我们可以从 stats.txt 文件中提取各个方法对应的分支（Branch）数量，结果如下所示。从数据中可以观察到，循环展开方法的分支数量显著低于其他几种方法，这与我们的预期一致。此外，由于矩阵分块方法的循环结构为六层嵌套，因此该方法产生了更多的分支跳转。

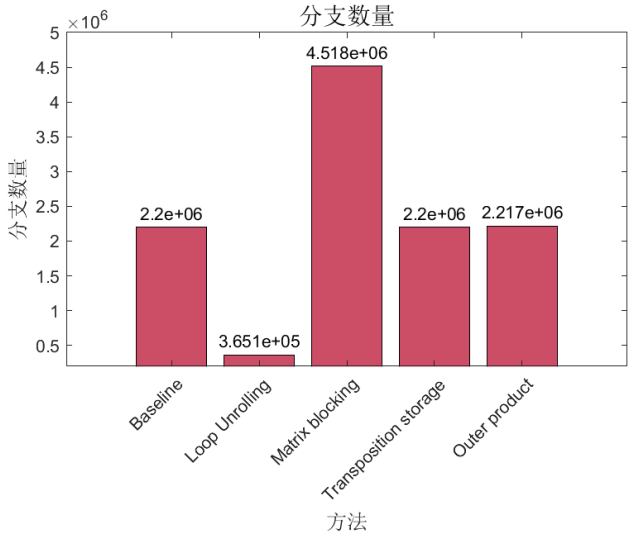

图 3: 软件优化分支数量分析

同样，从 stats.txt 文件中，我们可以分析各个方法下 l1 dcache 与 l2 cache 的访问总数与缺失率（Miss Rate），结果如下所示：

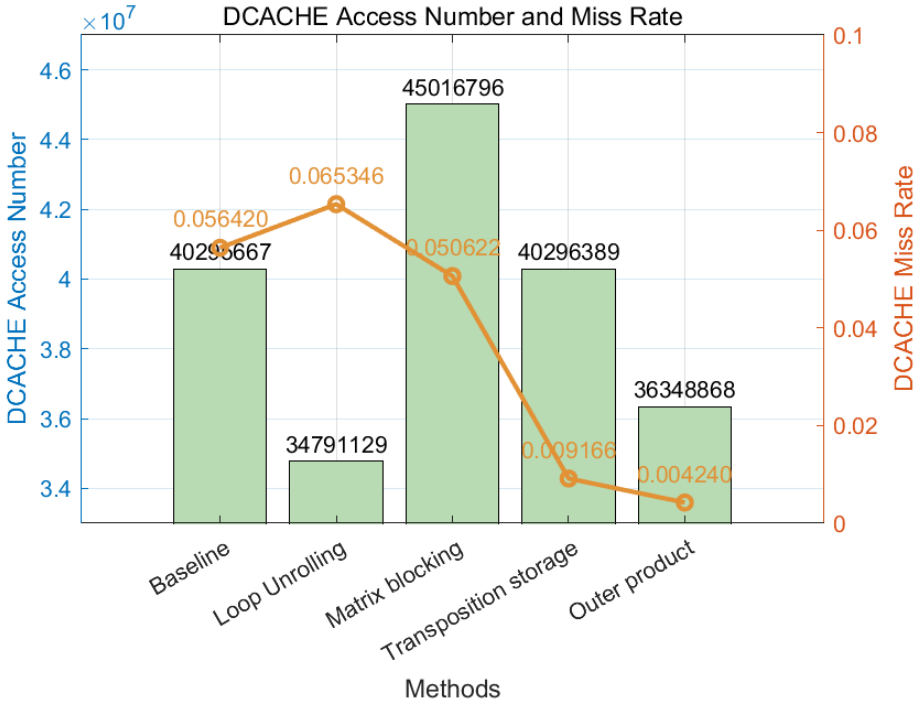

图 4: 软件优化 dcache 访问数与缺失率分析

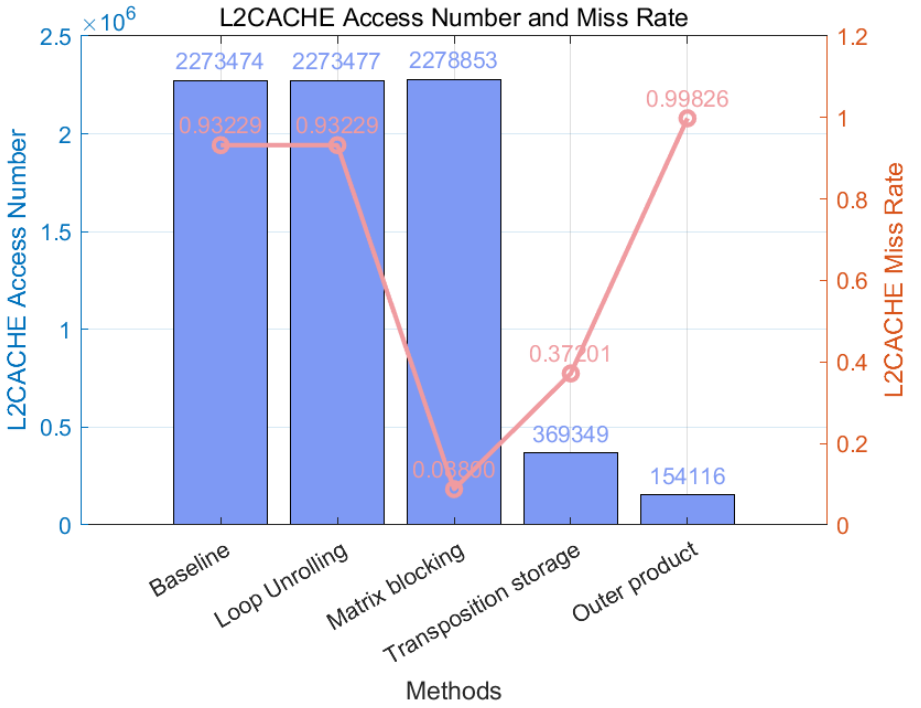

图 5: 软件优化 l2cache 访问数与缺失率分析

从以上两幅图可以看出：

- 循环展开对 l2 cache 的访问优化效果有限，但显著减少了 l1 dcache 的访问次数；
- 矩阵分块显著降低了 l2 cache 的缺失率；
- 转置存储同时降低了 l1 dcache 和 l2 cache 的缺失率，并减少了 l2 cache 的访问次数；

- 外积计算有效减少了 l1 dcache 的访问次数和缺失率，并显著降低了 l2 cache 的访问次数。

由此可见，这些优化方法是从不同的角度对 GEMM 进行了改进。

5.2 硬件优化的性能分析

5.2.1 缓存参数调整的性能分析

首先，我们将 (16, 4) 定义为：l2 cache 的大小为 16kB，l1 dcache 的大小为 4kB。本章节后续将以这样的方式来表示缓存的参数。(16, 4) 是 Baseline 的设置。

首先，我们对运行时间进行了分析。如图所示，随着参数增大的变化，访问时间呈现出下降的趋势。然而，这一下降并非平缓，而是呈现出明显的阶跃性特征，这个特征与 GEMM 问题的规模密切相关。

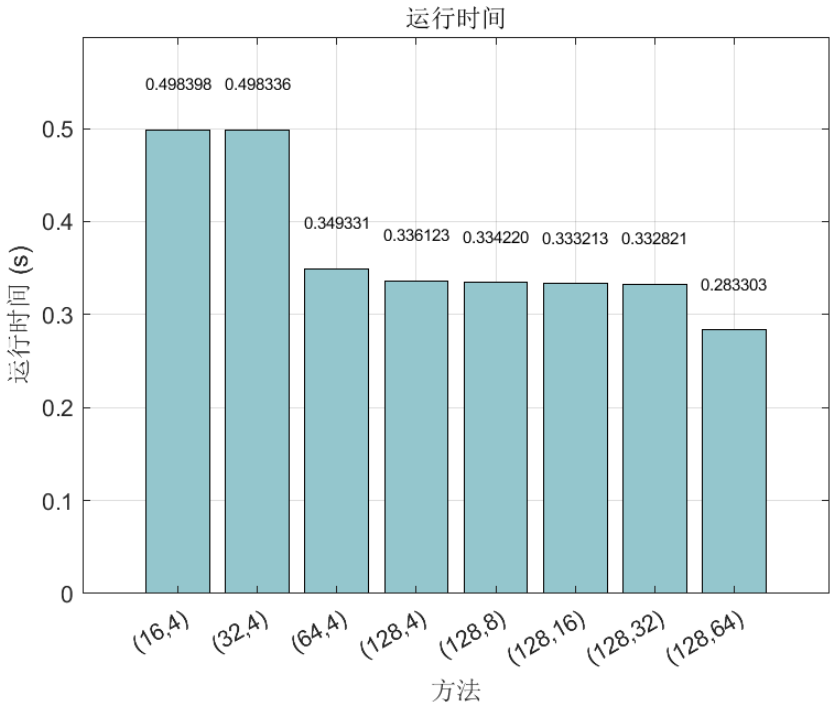

图 6: 缓存参数调整运行时间分析

根据 stats.txt 文件内的信息，在调整缓存参数大小时，对 l1 dcache 的访问数是保持不变的。这是自然的，因为在软件保持不变的情况下，对 1 级 cache 的数据访问次数是不变的。

接下来，我们分析了在“固定 l1 dcache 大小为 4kB，增大 l2 cache”时，l2 cache 的缺失率变化情况。如图所示，随着 l2 cache 大小的增大，我们观察到当 l2 cache 大小超过 64kB 时，缺失率明显下降。这个现象的原因在于，当 l2 cache 达到 64kB 时，它恰好能够容纳一个 128*128 的整数型矩阵的数据量。由于该矩阵的数据可以完全存储在 l2 cache 中，避免了频繁的缺失，因此 l2 cache 的缺失率显著降低。这表明 64kB 的 l2 cache 大小在处理这一规模的矩阵时，能够有效提高缓存命中率，减少数据访问延迟。这也告诉我们，在对一个特定应用进行优化时，如果缓存参数调整不当，不但会造成缓存资源的浪费，同时对性能提升也不明显。

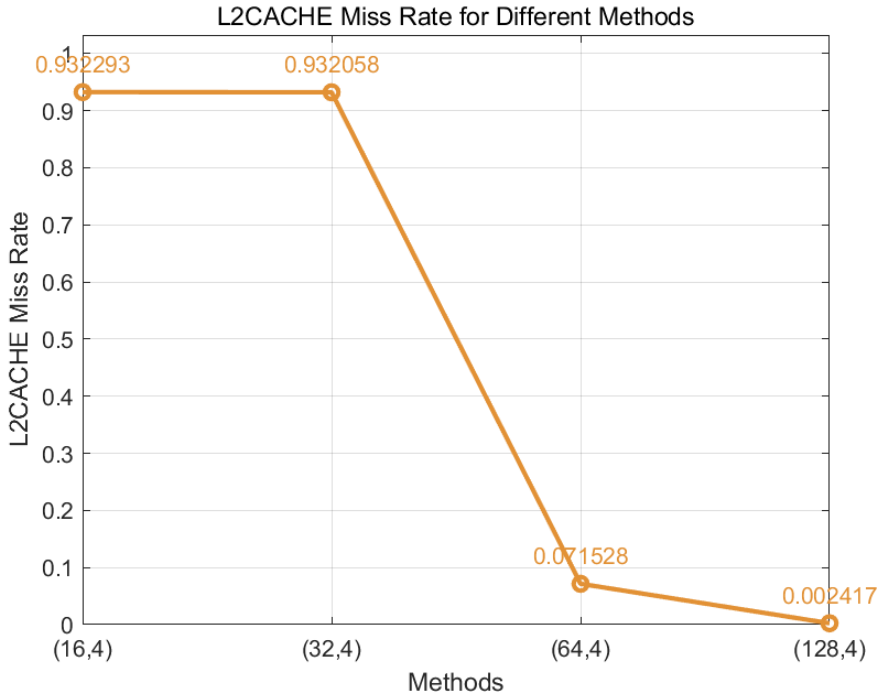

图 7: l2cache 参数调整时的缺失率分析

然后我们分析分析了在“固定 l2 cache 大小为 128kB，增大 l1 dcache”时，l1 dcache 的缺失率变化，如下图所示。从下图我们可以观察到，同样是在增大到 64kB 后，l1 dcache 的缺失率得到了显著的下降，理由同上。

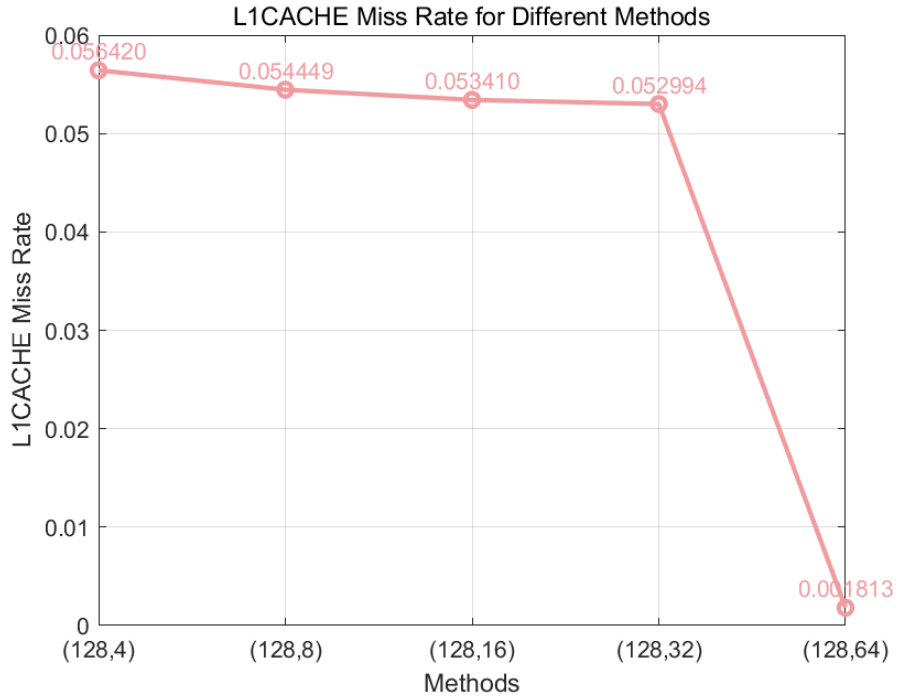

图 8: l1dcache 参数调整时的缺失率分析

同时，我们发现“固定 l2 cache 大小为 128kB，增大 l1 dcache”的过程中，l2 cache 的访问

次数和缺失率数据如下表所示。当 l1 dcache 增大到 64kB 后，l2 cache 的访问次数骤减，说明此时大部分数据都可以在 l1 dcache 内访问到。同时我们也观察到 l2 cache 的缺失率有所上升，这是访问次数减少造成的正常上升，因为访问次数大约降为了原来的 $\frac{1}{30}$，因此缺失率上升了大约 30 倍。这一事实再次告诉我们，在对特定应用做优化时，如果缓存参数调整的不合适，即便缓存资源变多，性能也不会有明显提升。

Parameter	(128,4)	(128,8)	(128,16)	(128,32)	(128,64)
L2 Cache Access Num	2273474	2194062	2152196	2135419	73045
L2 Cache Miss Rate	0.002417	0.002504	0.002552	0.002571	0.073845

5.2.2 不同缓存替换策略的性能分析

Baseline 的默认缓存替换策略是 LRU。我们测试了 11 种其他自定义的策略与自定义的两种 Gen 与 Hawkeye 策略，相关数据如下表所示：

RP	Run Time/s	L1 DCache Miss Rate	L2 Cache Miss Rate
LRU	0.498398	5.642%	93.2293%
Random	0.501929	6.3317%	82.2236%
TreePLRU	0.498374	5.642%	93.2068%
BIP	0.467574	5.4278%	81.0817%
LIP	0.467105	5.4944%	77.6704%
MRU	0.540666	8.7295%	64.6209%
LFU	0.525785	7.6299%	75.8262%
FIFO	0.502684	6.3233%	83.2052%
SecondChance	0.499813	5.7188%	91.9805%
NRU	0.496828	5.5066%	95.5215%
RRIP	0.496337	5.4871%	95.8563%
BRRIP	0.467276	5.3935%	81.7344%
Gen	0.517448	7.0457%	80.3791%
Hawkeye	0.557914	8.7297%	71.336%

在上图中可以观察到，BIP，LIP，BRRIP 三种策略对 GEMM 的适配性最好，三种方法都很契合 GEMM 中强局部性的访问模式，这是因为在未经软件优化的 GEMM 过程中，经常需要在不同的时间点重复访问某些数据（如矩阵的某一行或列），这三种方法能够对重复访问的数据进行保留，从而提升性能。而其他策略在本次实验中，并不契合 GEMM 的数据流，因此会造成性能的略微下降。

从上述数据中还可以观察到，l1 dcache 的缺失率增加所导致的性能损失，明显大于 l2 cache 缺失率降低所带来的性能提升。这表明，相较于提高 l2 cache 的命中率，降低更接近 CPU 的缓存（即 l1 dcache）的缺失率对性能优化的影响更加显著。因此，优化 l1 dcache 的命中率比优化 l2 cache 更为关键。

同时，从上表中我们也可以注意到，相较于软件优化与缓存参数调整，改变缓存替换策略收益很小，这是因为在 Baseline 的情况下，性能的瓶颈主要还是在软件的组织方式与缓存大小上。

5.2.3 CISC vs. RISC 性能分析

Baseline 的 CPU 设置为 X86TimingSimpleCPU。通过将 CPU 模型替换为 RiscvTimingSimpleCPU，并在 RISC-V 工具链下重新编译并运行 GEMM 程序，我们能得到两种不同指令集架构下的性能信息对比，如下所示（部分差距不明显的数据未列出）：

ISA	X86(Baseline)	RISC-V
CPU	X86TimingSimpleCPU	RiscvTimingSimpleCPU
Run Time/s	0.498398	0.531086
CPI	5.646112	5.818515
Inst. Num	88785126	92313612
FP Inst. Num	1321	26
Int Inst. Num	108283989	92296571
L1 DCache Miss Latency/Tick	221810946000	220598799000
L2 Cache Miss Latency/Tick	207211045000	206093187000

从运行时间分析，X86 的运行时间略低，表现出更高的性能。这可能是由于 X86 在硬件优化（如缓存设计和指令调度）上的优势导致的。

从 CPI 分析，X86 的 CPI 稍低，表明其每条指令的执行效率更高。这可能与其更复杂的硬件流水线设计和指令并行性优化有关，而 RISC-V 作为一种精简指令集架构，硬件可能较为简单，因此指令执行的效率稍低。

从提交的总指令数分析，RISC-V 提交的指令数更多，这表明在执行相同的工作量时，RISC-V 所需的指令数量高于 X86。这是因为 X86 的 CISC 指令集架构可以用更少的复杂指令完成相同的任务，而 RISC-V 的 RISC 架构使用更多的简单指令完成任务。

从浮点指令数分析，X86 的浮点指令数显著多于 RISC-V，说明两种架构对浮点运算的实现方式不同。X86 可能使用了更多的浮点指令来完成计算，而 RISC-V 的浮点指令数较少，可能是因为其浮点计算由其他整数指令间接实现，或者浮点单元的使用效率更高。

从整数指令数分析，RISC-V 的整数指令数少于 X86，主要原因是 CISC 架构的 X86 指令集的微操作分解。在 CPU 执行过程中，X86 程序中一些复杂指令（如乘除法、条件分支等）会被分解成多个微操作（Micro-Ops）。这也是为什么 X86 指令集的整数指令数会出现大于总指令数的情况。

从 l1 dcache 和 l2 cache 的数据缺失延迟分析，RISC-V 的 L1 和 L2 数据缓存缺失延迟均略低于 X86。这表明 RISC-V 的内存访问模式更高效，减少了缓存缺失的频率。

综上所述，X86 的优势是在浮点指令和复杂计算任务中表现更优，运行时间较短，CPI 更低。适合复杂任务，其复杂的硬件架构优化了指令并行性。RISC-V 的优势是内存访问效率较高，L1 和 L2 缓存缺失延迟都更低，简单的指令集设计使其对缓存更友好。对于计算密集型任务（如 GEMM），X86 的硬件优化使其运行速度略优。而 RISC-V 由于其缓存性能较好，可能在内存受限的场景中表现更优。

5.2.4 In Order vs. OoO 性能分析

Baseline 的 CPU 设置为 X86TimingSimpleCPU，是顺序执行模型。将 CPU 模型替换为乱序执行模型 O3CPU（也称为 DerivO3CPU），我们无需重新编译 GEMM 程序，因为 O3CPU 也是 X86 指令集。运行 GEMM，我们可以得到两种模型的性能信息。首先。我们观察 config.ini 文件内 CPU 的相关组件信息，对两种 CPU 进行微架构分析，如下所示：

```
[system.cpu]
type=BaseTimingSimpleCPU
children=dcache decoder icache interrupts isa mmu power_state tracer
    workload
```

```
[system.cpu]
type=BaseO3CPU
children=branchPred dcache decoder fuPool icache interrupts isa mmu
   power_state tracer workload
```

从上面的信息可以看出，DerivO3CPU 相较于 X86TimingSimpleCPU 而言多出了 branchPred (分支预测器) 和 fuPool (功能单元池)，这两个模块是实现乱序执行的关键组成部分。

- 分支预测器 (branchPred)：分支预测器在乱序执行中起着重要作用，它通过预测分支指令的执行路径，减少流水线停顿的频率。相比于简单 CPU，增加分支预测器使得 DerivO3CPU 能在执行条件分支和循环时显著提高效率，从而更好地适应复杂程序。
- 功能单元池 (fuPool)：功能单元池是支持乱序执行的核心组件，它允许多个功能单元并行执行指令，包括整数运算单元、浮点运算单元、加载/存储单元等。相较于简单的顺序执行 CPU，DerivO3CPU 的功能单元池可以调度更多指令，并在硬件资源允许的情况下同时处理多个操作。这种并行执行的能力显著提升了吞吐量和性能。

此外，DerivO3CPU 的设计中可能还伴随着更复杂的调度逻辑和寄存器重命名机制，以支持乱序指令的发射和提交。这些改进使得 DerivO3CPU 更适合模拟现代高性能 CPU 的行为，而 X86TimingSimpleCPU 则主要用于更简单的顺序执行场景。

通过这些额外模块的支持，DerivO3CPU 在处理复杂的计算密集型工作负载（如 GEMM）时，能够更有效地利用硬件资源，显著提高执行效率。

基于以上的微架构分析，我们观察 stats.txt 文件内的统计信息，可以观察到如下所示的性能信息对比（部分差距不明显的数据未列出）：

Feature	In Order	Out of Order
CPU	X86TimingSimpleCPU	DerivO3CPU
Run Time/s	0.498398	0.120917
CPI	5.646112	1.368881
l1 dcache Miss Latency/Tick	221810946000	217612026000
l2 cache Miss Latency/Tick	207211045000	200913513000

从运行时间分析，DerivO3CPU 的运行时间仅为 0.120917 秒，而 X86TimingSimpleCPU 的运行时间为 0.498398 秒。DerivO3CPU 的运行时间约为 X86TimingSimpleCPU 的 1/4，这表明乱序执行架构在处理相同工作负载时效率明显更高。原因在于乱序执行能够充分利用硬件资源，实现指令级并行，减少流水线停顿时间。

从 CPI 分析，DerivO3CPU 的 CPI 为 1.368881，而 X86TimingSimpleCPU 的 CPI 为 5.646112。CPI 的显著降低表明，DerivO3CPU 在同样的指令集上能够更快地完成指令执行，得益于分支预测器、功能单元池以及乱序调度的支持，从而显著提升了吞吐量。

从 l1 dcache 和 l2 cache 的数据缺失延迟分析，DerivO3CPU 的 L1 和 L2 数据缓存缺失延迟均低于 X86TimingSimpleCPU。这种差异可能反映了 DerivO3CPU 在缓存访问路径上的优化设计。

DerivO3CPU 显示了乱序执行架构相对于顺序执行架构的显著性能优势，尤其是在运行时间和指令执行效率上。L1 和 L2 缓存的缺失延迟对两种架构的影响相似，但 DerivO3CPU 能够更好地缓解缓存缺失对性能的影响，充分体现了乱序执行架构的优势。通过这组数据，可以看出 DerivO3CPU 更适合处理复杂的计算密集型任务，而 X86TimingSimpleCPU 可能更适合用于简单任务或功耗敏感的场景。

6 总结

在本次实验中，我们基于 gem5 仿真平台，对 CPU 架构下的 GEMM 进行了全面的软硬件优化研究。其中，软件优化部分主要探索了四种方法：循环展开（Loop Unrolling），通过减少循环控制开销以提升性能；矩阵分块（Matrix Blocking），通过优化数据局部性减少缓存缺失；转置存储（Transposition Storage），通过调整数据存储方式进一步提高缓存利用率；外积计算（Outer Product），通过改进计算模式减少数据传输和缓存访问压力。

在硬件优化部分，我们着重研究了四种方法：调整缓存参数（缓存大小），以分析缓存容量对性能的影响；改变缓存替换策略（如 BIP、LIP 和 BRRIP 等），以优化缓存命中率和数据访问效率；采用 RISC-V 架构，探索其 CISC 架构与 RISC 架构的差异；以及引入乱序执行模型，通过提升指令级并行性和资源利用率进一步优化计算性能。

通过对 GEMM 的软硬件优化，我们不仅验证了不同优化方法对 GEMM 的实际性能提升效果，还深入分析了其内在的机制，对 CPU 架构与 GEMM 的计算模式都有了更深入的了解。