上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

计算机体系架构实验报告

课题 基于 gem5 的 GEMM 算法优化

学生姓名 汪子尧、蔡昂欣、张博文

学号 124039910018、124039910003、124039910094

学院 集成电路与学院

2025 年 1 月 18 日

目录

1 实验内容

在现代计算架构中，GEMM（General Matrix Multiplication，通用矩阵乘）算法是深度学习和其他科学计算领域中消耗计算资源较大的操作之一。随着计算需求的不断增长，对 GEMM 算法的优化变得尤为重要。GEMM 涉及两个矩阵的乘法，形式为 $C = AB$，其中 A 是 $M \times K$ 的矩阵，B 是 $K \times N$ 的矩阵，而 C 是 $M \times N$ 的结果矩阵。传统的 GEMM 算法具有 $O(n^3)$ 的时间复杂度，但通过算法优化和软件优化，可以显著提高其性能。

Gem5 是一个开源的计算机系统架构模拟器，提供了丰富的硬件模型和灵活的配置选项。其主要特点包括可配置的 CPU 模型（如 Atomic、Timing、In-Order 和 O3）、系统模型（SE 和 FS）以及存储器模型（Classic 和 Ruby）。Gem5 通过 Python 脚本进行配置和初始化，支持详细的性能分析和硬件配置研究。

本实验旨在利用 Gem5 仿真平台对 GEMM 算法进行软硬件的优化。通过 Gem5 的统计工具，详细分析 GEMM 算法在不同硬件配置下的性能表现，研究不同 CPU 型号、缓存大小和内存类型对算法性能的影响。此外，实验还将在 Gem5 平台上实现和验证各种 GEMM 软件优化策略，以验证其有效性并提高算法性能。

本次实验内容完成情况如 表 1 所示：

表 1: 实验内容及完成情况

要求	**项目**	**完成度**
必做	搭建环境并成功运行 gem5	完成
必做	C/C++ 实现通用矩阵乘法计算（GEMM）作为 Baseline 进行优化	完成
必做	软件优化：通过修改数据流、矩阵分块、稀疏化等手段实现 GEMM 加速	完成
必做	硬件优化：通过 gem5 上的架构优化、缓存优化等手段实现 GEMM 加速	完成
选做	实现非 gem5 自带的缓存替换策略，实现 GEMM 优化	完成
选做	指令集差异分析：在 RISC-V 架构上进行性能评估	完成
选做	处理器结构差异分析：评估不同处理器特性对性能的影响（In order vs 3O）	完成

本次实验分工如 表 2 所示：

表 2: 实验分工表

姓名	主要工作	工作量占比
汪子尧	缓存优化分析、缓存替换策略实现、整体硬件架构优化实现	1/3
蔡昂欣	多种软件优化策略：矩阵分块、存储转置等策略实现与分析	1/3
张博文	环境搭建及 baseline 编写、指令集架构优化与处理器结构优化分析	1/3

2 实验环境

2.1 Gem5 仿真器

本实验旨在利用 Gem5 仿真平台对 GEMM 算法进行软硬件的优化。实验环境搭建在 Linux 操作系统上，使用 Docker 容器技术来管理和运行 Gem5。选择 Docker 的原因是其能够提供一个隔离的、一致的实验环境，确保实验结果的可重复性和可移植性。Gem5 的版本为 v23.0，该版本提供了最新的架构模拟功能和性能分析工具。Docker 容器内的操作系统基于 Ubuntu 20.04 LTS，这是一个轻量级且功能齐全的 Linux 发行版，适合运行 Gem5。实验环境如 表 3 所示：

表 3: 实验环境详细配置

组件	配置
主机操作系统	Ubuntu 18.04 LTS
Docker 版本	Docker 24.0.2
Docker 镜像	gem5/ubuntu-20.04_all-dependencies:v23-0
容器内操作系统	Ubuntu 20.04 LTS
容器内 Python 版本	Python 3.8.10

2.2 实验性能基线（Baseline）

Baseline 的硬件部分,我们参考了 Gem5 官方提供的二级缓存架构配置文件(two_level.py)。这一配置文件详细定义了处理器型号，缓存层次结构，包括 L1 和 L2 缓存的大小以及缓存的替换策略等关键参数，确保了实验的起始点具有广泛的适用性和可比性，同时也为后续的优化工作提供基础。官方提供的二级缓存架构原理图如图 1所示：

架构中我们比较关心的部件及参数配置主要有：

1) CPU 型号（default：X86TimingSimpleCPU）

2) L1 Data Cache 与 Unified L2 Cache 的容量（default：L1 Data Cache 4kB L1 Data Cache 16kB）

3) L1 Data Cache 与 Unified L2 Cache 的缓存替换策略（default：LRU）

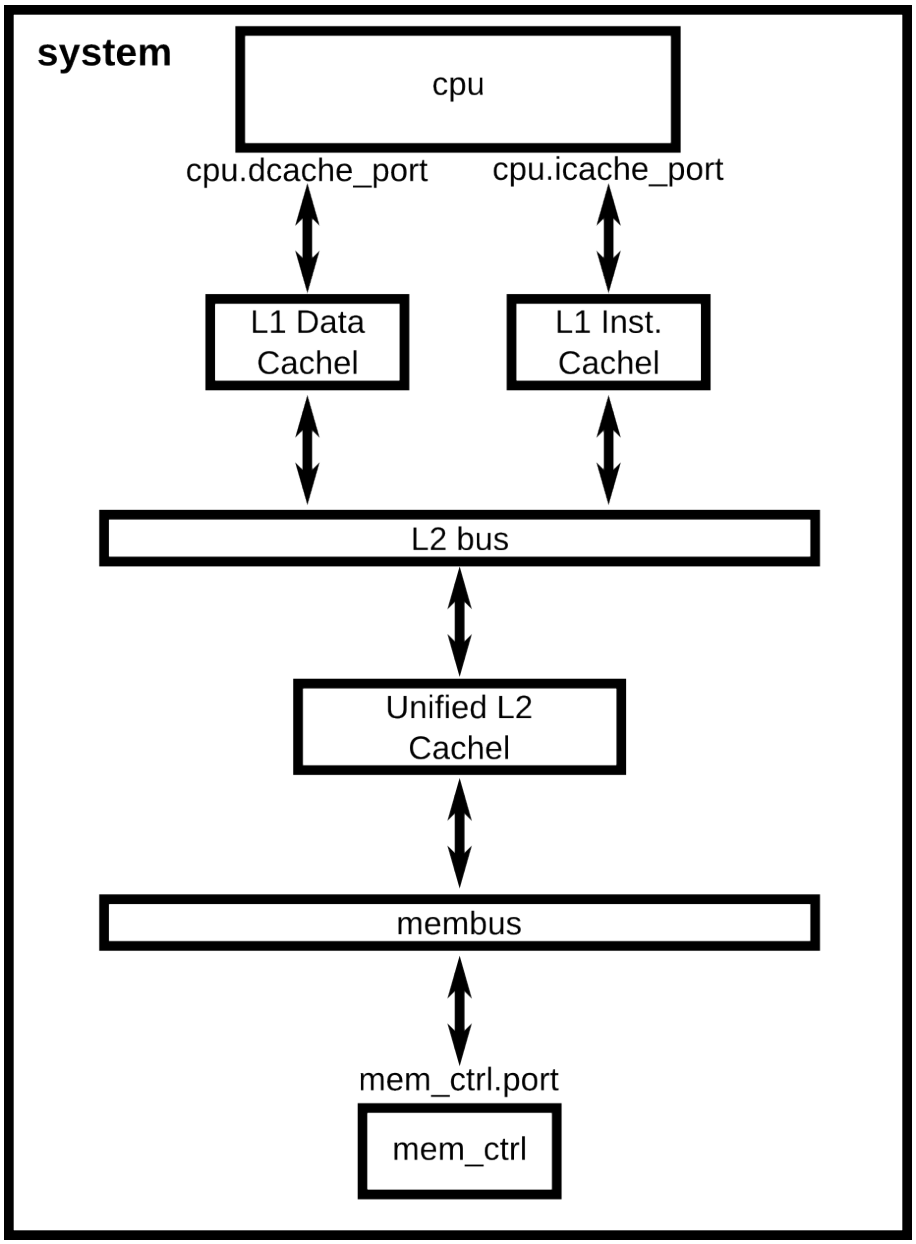


图 1: 2-level cache architecture

Baseline 的软件部分，我们实现了未经过优化的三重嵌套循环的通用矩阵乘法（GEMM）算法。这一实现遵循了 GEMM 算法的基本形式，即通过三个嵌套循环遍历输入矩阵的行和列，逐个计算结果矩阵中的每个元素。具体而言，算法的伪代码可以表示为：

Algorithm 1 Baseline GEMM 算法

初始化矩阵 C 为 $M \times N$ 的零矩阵
for $i = 1$ **to** M **do**
    for $j = 1$ **to** N **do**
        $C_{ij} \leftarrow 0$
        for $k = 1$ **to** K **do**
            $C_{ij} \leftarrow C_{ij} + A_{ik} \times B_{kj}$
        end for
    end for
end for

这种未优化的实现方式直接反映了 GEMM 算法的基本计算模式，没有考虑任何特定的硬件特性或软件优化技术。因此，它为我们提供了一个纯粹的算法性能基线，使得我们能够清晰地观察到在未进行任何优化的情况下，算法在不同硬件配置下的原始性能表现。这一基线对于后续评估各种优化策略的效果具有重要的参考价值，帮助我们准确地衡量优化措施带来的性能提升。

3 硬件优化策略

为了简化硬件仿真流程，我们对原有的 python 配置脚本进行修改，通过 argparse 库实现命令行通过指定 cpu 型号、各 Cache 容量与各 Cache 替换策略参数简化 python 配置脚本的频繁修改。进一步地，基于此构建了批量化进行参数修改的 bash 脚本，实现了自动化参数调整，调用 Gem5 仿真以及关键参数提取的工作，这一部分 bash 脚本代码见**附录 A**。脚本定义及输出文件如下：

1） **autorun_cache.sh**：cache 容量的测试，关键参数提取及结果输出在 autorun_cache.txt

2） **autorun_cpu.sh**：cpu 型号的测试，关键参数提取及结果输出在 autorun_cpu.txt

3） **autorun_rp.sh**：cache 缓存替换策略的测试（包含非 Gem5 自带的缓存替换策略），关键参数提取及结果输出在 autorun_rp.txt 与 autorun_my_rp.txt

3.1 缓存优化

GEMM 算法通常被归类为计算-访存密集型算法。这种分类反映了 GEMM 算法在执行过程中对计算资源和内存访问的高需求。缓存容量和缓存替换策略是影响 GEMM

算法性能的两个主要因素，它们共同决定了算法在访问内存时的效率。

3.1.1 缓存参数调整

在 GEMM 算法中，矩阵 A、B 和 C 的数据需要频繁地被访问。如果缓存容量足够大，能够容纳这些矩阵的大部分或全部数据，那么缓存命中率将显著提高，从而减少对主内存的访问次数。这可以显著降低内存访问延迟，提高算法的整体性能。相反，如果缓存容量较小，无法容纳足够的数据，缓存未命中率将增加，导致频繁的内存访问，从而降低算法的性能。

由于这里我们主要对运算中数据的缓存行为进行分析优化，因此这里只考虑 L1 Data Cache 与 L2 Cache 容量，而忽略 L1 Inst Cache 变量。我们简单定义**测试向量：(L1 Cache Size , L2 Cache Size)**。通过修改测试向量，探究两级 Cache 对 GEMM 的行为影响以及优化分析。在批量运行脚本中指定测试向量如下：

```
# 定义要测试的L1d和L2缓存容量组合
declare -a l1d_sizes=("4kB" "4kB" "4kB" "4kB" "8kB" "16kB" "32kB" "64kB"
    "128kB")
declare -a l2_sizes=("16kB" "32kB" "64kB" "128kB" "128kB" "128kB" "128kB"
    "128kB" "256kB")
```

这里需要说明的一点是:int 类型 128*128 矩阵乘运算，单矩阵的总 size 应该为 64kB，因此 cache 总容量最大不需要超过 256kB，同时 l2 cache size/l1 cache size 的比值为 2 的幂次较为合理。

不同测试向量下的部分实验结果如表 9所示：

表 4: 不同缓存配置下的矩阵乘法性能

L1d Size (kB)	L2 Size (kB)	执行时间 (秒)	L2 Cache Miss Rate	D-Cache Miss Rate
4	16	0.506695	0.935581	0.055031
4	32	0.506665	0.935347	0.055031
4	64	0.363563	0.078919	0.055031
4	128	0.349067	0.002125	0.055031
8	128	0.347262	0.002198	0.053202
16	128	0.346313	0.002241	0.052235
32	128	0.345933	0.002257	0.051852
64	128	0.296492	0.059292	0.001853
128	256	0.295467	0.110253	0.000855

由于测试向量中存在两个变量，为了更加直观地观察分析 cache size 对 GEMM 性能的影响，我们每次固定其中一个 cache size，将执行时间，cache 的 Miss Rate 进行制图，结果如图 2，图 3，图 4所示：

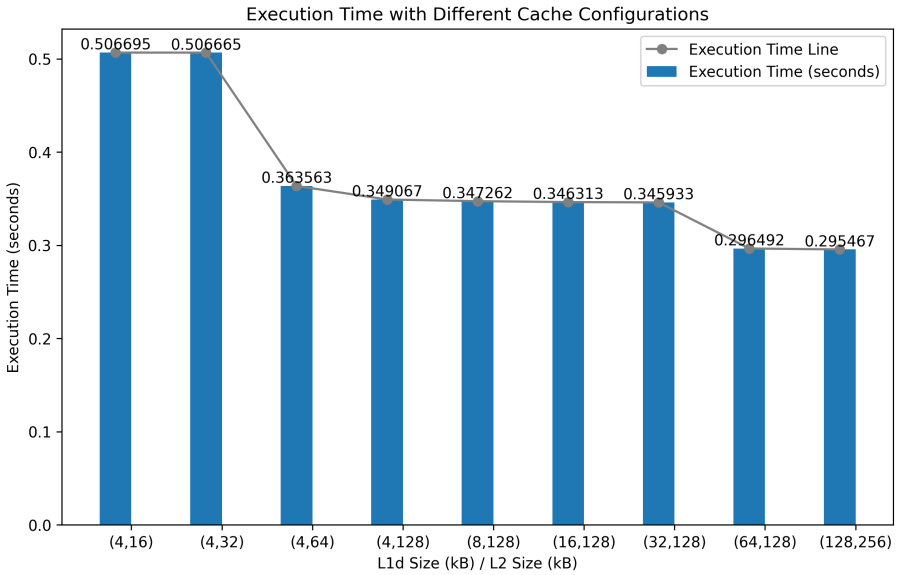


图 2: 不同 cache size 下的 execution time

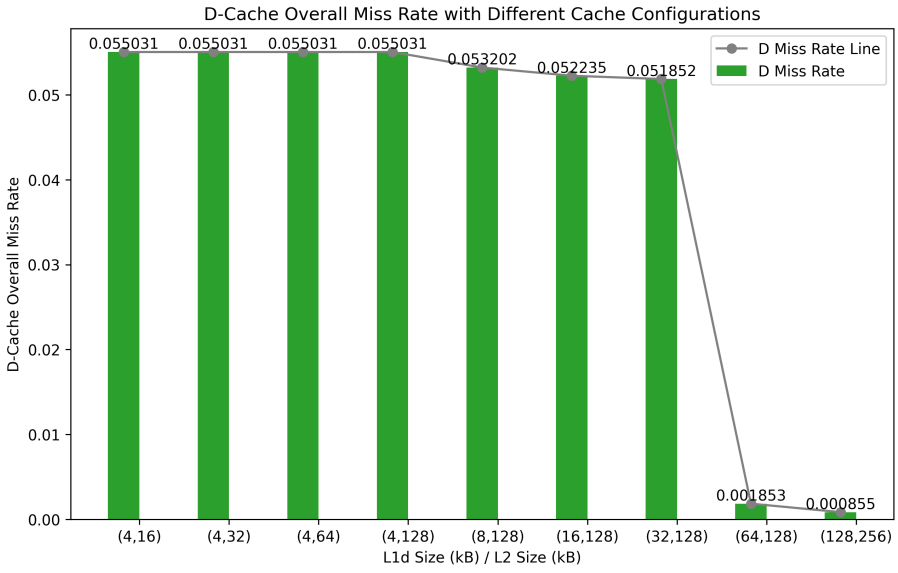


图 3: 不同 cache size 下的 l1 miss rate

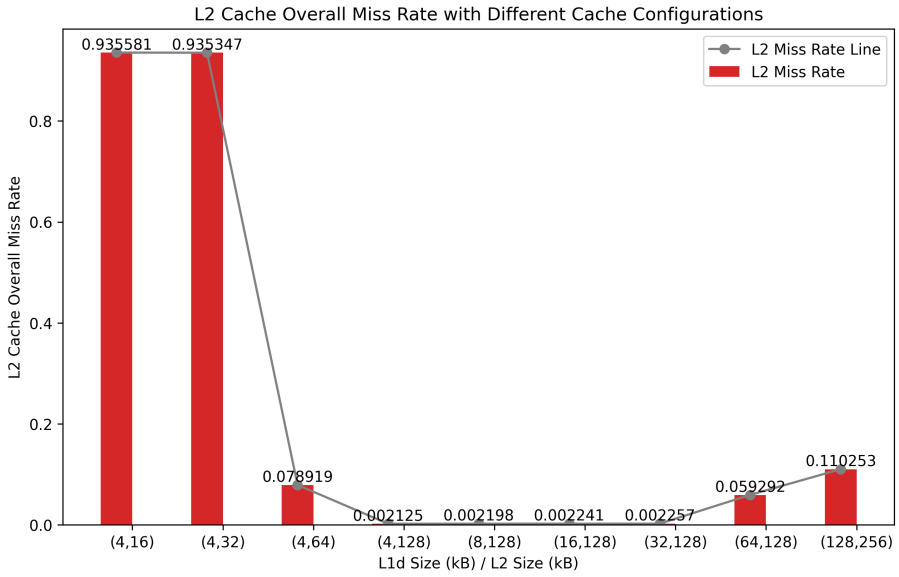


图 4: 不同 cache size 下的 l2 miss rate

从图 2我们容易发现，随着 cache 容量的增加，执行时间呈下降趋势，这一点是显而易见的。cache 容量增加，GEMM 算法在获取数据时的等待时间减少，因为更多的数据可以存储在速度更快的 cache 中，从而减少了对主内存的访问次数。这种减少的内存访问延迟直接导致了整体执行时间的降低。从图 2中我们也可以发现，在固定 L1 cache 为 4kB 时，L2 cache size 在 16kB -> 32kB 与 64kB -> 128kB 时，执行时间的优化效果并不显著；但 L2 cache size 在 32kB -> 64kB 时，执行时间呈现阶跃式下降，减少了近 28%。同样的，L2 cache 增加带来的执行时间降低基本保持稳定时，固定 L2 cache size，调整 L1 cache 时也出现了类似的现象，L1 cache size 在 4kB -> 8kB -> 16kB -> 32kB 时，执行时间的优化效果并不显著，但 L1 cache size 在 32kB->64kB 时，执行时间同样呈现阶跃式下降，减少了近 14.7%。

这一变化并非偶然，我们通过简单计算易知，一块 int 类型 128*128 矩阵的大小恰为 64kB!

因此，当 cache 的容量增加到 64kB 时，恰能容纳完整矩阵，而依据 GEMM 算法访存的空间局部性效应，cache 的命中率大大增加，从图 3与图 4中，我们也可以清除验证这一点。

此外，我们这里最后一个测试组将 cache 的 size 设置完全饱和，即 L2 cache 能完整容纳 A、B、C 三块矩阵，L1 cache 能完整容纳参与计算的 A、B 矩阵，从执行时间上我们可以看出来，在 size 饱和的情况下，继续增加 cache 的容量带来的收益甚微，反而会大幅度增加面积、功耗、价格开销。而图 4中 L2 cache miss rate 甚至出现回弹，这是因为缓存的数据分布不均匀，或者数据访问模式导致某些缓存行频繁被替换，而没有足够的时间被重复访问以证明其保留在缓存中的价值。

3.1.2 缓存替换策略

常见的缓存替换策略包括最近最少使用（LRU）、先进先出（FIFO）和随机替换（RAND）等。在 GEMM 算法中，数据访问模式通常是高度规律的，这意味着某些缓存行可能会被频繁访问，而其他缓存行则较少被访问。一个有效的缓存替换策略可以确保频繁访问的数据始终保留在缓存中，从而提高缓存命中率。

gem5 仿真环境提供了多种缓存替换策略，这里我们选取了多种缓存替换策略进行评估：

```
# 定义要测试的替换策略
declare -a replacement_policies=("lru" "mru" "plru" "lfu" "bip" "lip" "random" "fifo"
    "nru" "rrip")
```

为了尽量最大化缓存替换策略对执行时间的影响因子，我们特意将缓存大小设置为默认值，即 L1 缓存大小为 4kB，L2 缓存大小为 16kB。**这样的设置旨在模拟一个较为拥堵的缓存配置环境，缓存会频繁发生替换，因此替换策略在此时显得尤为重要**。

下面对测试的缓存替换策略的逻辑进行简单解释：

1) **LRU (Least Recently Used)**: 替换最久未被访问的数据块。适用于大部分通用场景，尤其是当数据访问模式呈现局部性时。

2) **MRU (Most Recently Used)**: 替换最近被访问的数据块。

3) **PLRU (Pseudo Least Recently Used)**: 一种近似 LRU 的实现，通常用于硬件中。

4) **LFU (Least Frequently Used)**: 替换访问次数最少的数据块。适用于数据访问模式较为固定，且某些数据块访问频率较低的场景。

5) **BIP (Bimodal Insertion Policy)**: 新数据块以一定概率作为最近最常使用 (MRU) 插入。适用于数据访问模式较为复杂，需要平衡 LRU 和 MRU 的场景。

6) **LIP (LRU Insertion Policy)**: 替换重要性评分最低的数据块，重要性基于访问频率和时间。适用于需要根据数据块重要性进行替换的场景。

7) **RandomRP**: 从候选数据块中随机选择一个进行替换。适用于数据访问模式较为随机的场景。

8) **FIFO (First In First Out)**: 缓存满时替换最早进入的数据块。适用于数据访问模式较为简单，且数据块生命周期较为均匀的场景。

9) **NRU (Not Recently Used)**: 替换最近未使用的数据块，结合了 LRU 和 MRU 的特点。适用于数据访问模式较为复杂，需要平衡 LRU 和 MRU 的场景。

10) **RRI (Recency Recency Insertion)**: 新数据块作为最近最少使用 (LRU) 或最近最常使用 (MRU) 插入，取决于最近访问情况。适用于数据访问模式较为复杂，需要根据最近访问情况动态调整替换策略的场景。

不同缓存策略下的 gem5 仿真结果对比如表 10所示：

表 5: 不同缓存替换策略下的 GEMM 的性能比较

Replacement Policy	Execution Time (s)	D-Cache Miss Rate	L2 Miss Rate
LRU	0.506695	0.055031	0.935581
MRU	0.553967	0.085088	0.646769
PLRU	0.506663	0.055031	0.935385
LFU	0.540417	0.074118	0.770182
BIP	0.480627	0.053039	0.812913
LIP	0.475470	0.053621	0.778739
Random	0.515646	0.061876	0.823562
FIFO	0.517659	0.061788	0.833527
NRU	0.505182	0.053848	0.956191
RRI	0.504813	0.053611	0.960413

这里我们重点关注在该实验场景下表现较为优异的 LIP、表现最劣势的 MRU 缓存替换策略。

考虑 GEMM 算法的访问模式：对于 GEMM 算法中的一个基本操作：

$$C_{ij} = \sum_{k=1}^{K} A_{ik} \times B_{kj}$$

其中，A、B 和 C 是大小分别为 $m \times k$、$k \times n$ 和 $m \times n$ 的矩阵。在这个操作中，数据访问模式具有以下特点：

- **行优先访问**：对于矩阵 A 和 B，数据是按行连续访问的。
- **重复访问**：在计算 C 的每个元素时，A 的同一行和 B 的同一列会被重复访问。

LIP 是一种改进的替换策略，它基于 LRU 替换策略的基本原则，但在插入新块时有所不同。在 LIP 策略下，新块被插入到最近最少使用的位置（LRU），即最不可能被访问的位置。当新块被插入到 LRU 位置时，它们的时间戳会被更新为 MRU，以便在后续访问中正确反映它们的使用频率。这样，当新块被重复访问时，它们将逐渐向 MRU 位置移动，直到它们成为最近访问的块。由于 GEMM 算法的空间局部性，刚插入的数据块很可能在短时间内被重复访问。每次重复访问时，LIP 策略会更新这些数据块的时间戳，使其向 MRU 位置移动。随着算法的执行，它们将逐渐移动到缓存的 MRU 位置，从而减少未来的缓存未命中，减少了 GEMM 算法的执行时间。

而 MRU 与 LRU 相反，LRU 策略选择的是最久未被使用的条目进行替换，而 MRU 策略选择的是最近被使用的条目进行替换，这会导致较旧但仍然频繁访问的条目被替换出缓存，从而降低缓存的命中，结果从 L1 cache Miss Rate 一列也验证了这一点。**同时，从表项对比中我们也不难发现，GEMM 算法的访存中，L1 cache 命中/未命中对执行时间的影响远大于 L2 cache**。

除了 gem5 内置的缓存替换策略以外，我们额外自行实现了一种基于随机的加权随机（WeightedRandom）缓存替换策略。WeightedRandom 的实现详见附录二，修改 SConscript 编译配置脚本文件，以及增加 WeightedRandom 的声明定义，重新编译构建 gem5，即可在仿真时同样通过指定 RP 参数调用 WeightedRandom 的缓存替换策略。

WeightedRandom 策略结合了随机性和权重机制。随机替换策略对于所有 block 的替换是完全公平，即每一个 block 都有完全相等的概率被替换出去，而不考虑算法的无论是空间还是时间的局部性。因此，基于此，我们提出了加权随机，每次访问一个 block 时，这个 block 的权重就会增加，最终依据权重再进行随机替换。在保证一定的公平性的基础上，即每一个 block 都有机会被替换出去，又能对最近使用的块产生一定的排斥机制，即更容易被替换出去。类似地，如果我们对每一个 block 权重赋初值后，每次访问时减少这个 block 的权重，最终依据权重随即替换，在保证一定的公平性的基础上，会对最近使用的块产生一定的保护机制，即更难被替换出去。

从更加极限的角度来考虑，当每次访问 block，权重增加无穷大时，算法会收敛为 MRU；而每次访问 block，权重减小无限大时，算法就比较特殊了，block 相当于会被锁死，整体上更趋于 LRU 算法，这里我们在实现时定义为每次访问权重自增 **1×** 与 **10×**，将算法与 Random 缓存替换策略对比，对比结果如表 11所示：

表 6: WeightedRandom 策略与 Random 策略

Replacement Policy	**Execution Time (s)**	**D-Cache Miss Rate**	**L2 Miss Rate**
Random	0.515646	0.061876	0.823562
WeightedRandom weight:1×	0.523583	0.068558	0.748128
WeightedRandom weight:10x	0.528192	0.071743	0.721079

3.2 架构优化

3.2.1 RISC 架构优化

RISC-V 是一种开源的精简指令集架构（RISC），由加州大学伯克利分校开发。该架构以其简洁性和可扩展性为设计目标，支持模块化设计，允许根据特定需求添加新的指令集扩展。RISC-V 的指令集数量相对较少，通常不超过 300 条，这不仅使其在实现上更为简洁，而且更易于理解和维护。在本节中，我们将对 RISC-V 架构上的 GEMM 算法进行性能评估，并将其结果与 CISC x86 架构进行对比分析。

为了在 RISC-V 架构上运行 GEMM 算法，我们需要重新编译生成适用于 RISC-V 架构的可执行文件。由于之前生成的可执行文件是基于 x86 架构的，因此这一步骤是必要的。为了确保对比的公平性，我们将继续使用相同的基线 C 源代码，但这次将采用 RISC-V 的交叉编译工具链进行编译。具体的编译命令如下

```
riscv64−linux−gnu−gcc −o test test.c –static
```

将仿真结果与 X86 架构下的 baseline 进行对比，为了更加全面直观的比较两种指令集架构下的差异，例如**汇编指令数目**、**Load/Store Architecture** 带来的影响。我们额外从 stats 中提取了更多我们关心的比较有趣的实验数据，数据对比如表 12所示：

表 7: X86 架构与 RISC-V 架构仿真结果对比

架构	**CISC-X86**	**RISC-V**
执行时间 (秒)	0.506695	0.506105
L2 Miss Rate	0.935581	0.914345
D-Cache Miss Rate	0.055031	0.056489
CPU CPI	5.815975	5.526771
simInsts	88682623	93321418
simOps	109910810	93354195
total dram read	2120825	2120491
total dram write	3275	3189

从数据结果中我们可以看到，RISC-V 架构的执行时间略短于 X86 架构，但这种差异基本可以忽略不计，差异在 0.16‰ 我们可以得出结论：在 CPU 均为非流水线、时序模型的访存架构下时，并且时钟频率设置相同的情况下，RISC-V 与 X86 架构在 GEMM 算法下的性能基本保持一致。下面我们关心两种不同架构下其他几个参数的差异化结果。RISC-V 的 CPI（5.526771）低于 X86（5.815975），这表明 RISC-V 在每个周期内执行的指令数更多，这一点也是显而易见的。**由于 RISC-V 作为一种精简指令集，单指令的复杂程度相对是低于 X86 架构指令的，因此指令执行的平均 cycle 是少于 X86 架构的，但单条指令复杂程度的降低并非是无条件的，这依赖于指令总数的增加**，从 simInsts 的对比中我们也验证了这一点。**下面我们进行了一个有趣的计算，我们将 CPI 与 simInsts 相乘的结果进行对比，X86 与 RISCV 的乘积结果分别为 515775918 与 515766107，从这个维度上我们也可以看出来他们的性能差异是不大的**。

最后，在 DRAM 读取次数上，两种架构几乎相同（RISC-V 为 2120491，X86 为 2120825），而在 DRAM 写入次数上，X86 架构略高（3275 对比 3189）。这可能反映了两种架构在内存访问模式上的微小差异。

3.2.2 处理器架构优化

在本小节中，我们探究不同处理器架构特性对性能的影响。Baseline 采用的是 TimingSimple，gem5 提供了功能更加强大的 O3（out-of-order）处理器架构，下面我们对两种不同处理器架构下 GEMM 算法进行仿真，对比结果如表 13所示：

表 8: 不同处理器架构性能对比

CPU 类型	(D-cache，l2-cache)	执行时间（秒）	CPU CPI
TimingSimple	(4kB，16kB)	0.506695	5.815975
	(128kB，256kB)	0.295467	3.434031
O3	(4kB，16kB)	0.262001	2.970501
	(128kB，256kB)	0.057787	0.667700

这里我们设置了两个对比组，分别在缓存拥堵与缓存饱和的情况下对两种架构 CPU 性能进行对比，从表中数据我们容易获得以下结论：**执行时间基本与 CPU CPI 保持一致，在缓存拥堵的情况下，O3 处理器相较于 TimingSimple 的性能提升一倍，CPI 降低为原来的 50%；在缓存饱和的情况下，这种提升更为明显，性能一度提升 5 倍以上，同时 CPI 也减少到原先的 20%。**

GEMM 算法在 O3 处理器上的性能提升是可预见的，因为 O3 处理器是一种高级的超标量乱序执行（Out-of-Order，OoO）处理器。它能够模拟乱序执行和超标量执行的指令间依赖，以及在多 CPU 上并发执行的多线程。这种处理器架构设计显著提高了指令的吞吐吐率，从而大幅提升了性能，并缩短了执行时间。

O3 处理器拥有 7 级流水线，包括取值（Fetch）、译码（Decode）、重命名（Rename）、发射（Dispatch）、执行（Execute）、写回（Write-back）和提交（Commit）等阶段。这种流水线设计通过并行处理多个指令来提高吞吐吐率。理论上，7 级流水线可以达到最高 7 倍的加速比，但在缓存完全饱和的情况下，实际加速比可能在 5 倍以上。这是合理的，因为 GEMM 算法同样存在控制冒险（control hazard），这些冒险可能导致流水线的冲刷，从而降低 CPI。**但尽管如此，O3 的性能仍旧是非常强大的，通过其高级的超标量乱序执行能力和流水线设计，为 GEMM 算法提供了显著的性能提升。这种提升主要体现在减少执行时间和提高指令吞吐吐率上，使得 O3 处理器成为执行计算密集型任务如 GEMM 算法的理想选择。**

4 软件优化策略

本章旨在探索多种软件优化方法对 GEMM 性能的提升效果。我们选取了**循环展开**、**分块矩阵**、**外积展开**、**矩阵转置**、**混合方法以及软硬件协同**等六种优化策略。特别的软硬件协同优化方法通过**修改行缓存大小**来适配软件代码需求实现。这些方法从不同的角度出发，针对 GEMM 运算中的数据访问模式、计算粒度、缓存利用效率等问题进行优

化。通过对比优化前后的性能数据，分析各优化方法的优势与局限性，为实际应用中选择合适的优化方案提供参考依据，进而推动高性能计算在更广泛领域的高效应用。

4.1 基本软件优化

4.1.1 循环展开

在矩阵乘法中，原始的三重循环是按照 i,j,k 的顺序进行的，即先固定行索引 i 和列索引 j，然后遍历 k 来计算 C[i][j]。这种顺序会导致在访问矩阵 A 和 B 时，存在大量的缓存未命中情况。因为每次改变 k 时，都需要访问矩阵 A 和 B 中相隔较远的元素，这与缓存的局部性原理相违背。

在提供的代码中，循环展开是针对 k 循环进行的。将 k 循环展开后，可以使得在计算过程中，对矩阵 A 和 B 的访问更加集中。具体来说，每次固定 k 值后，会连续计算多个 C[i][j] 的值。这样在访问矩阵 A 时，会连续访问 A[i][k] 这一列中的多个元素；在访问矩阵 B 时，会连续访问 B[k][j] 这一行中的多个元素。这种访问模式更符合缓存的局部性，能够提高缓存的利用率。

考虑到有两级缓存，一级缓存为 4KB，二级缓存为 16KB。循环展开后，可以使得矩阵 A 和 B 中参与计算的元素在缓存中有更高的命中率。因为每次计算时，会尽量利用已经在缓存中的数据，减少从主内存中加载数据的次数。例如，当 k 固定时，矩阵 A 中的一列数据和矩阵 B 中的一行数据会在缓存中被保留较长时间，用于计算多个 C[i][j] 的值，而不是频繁地替换缓存中的数据。

```
void matrix_multiply(int A[M][K], int B[K][N], int C[M][N]) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < K; k += 8) {
                C[i][j] += A[i][k] * B[k][j] + A[i][k+1] * B[k+1][j] +
                           A[i][k+2] * B[k+2][j] + A[i][k+3] * B[k+3][j] +
                           A[i][k+4] * B[k+4][j] + A[i][k+5] * B[k+5][j] +
                           A[i][k+6] * B[k+6][j] + A[i][k+7] * B[k+7][j];
            }
        }
    }
}
```

分析上述代码，优化原因如下：

1) **减少缓存未命中次数**: 由于循环展开使得数据访问更加局部化，缓存未命中次数显著减少。在原始的循环顺序下，每次改变 k 值，都需要从主内存中加载新的 A[i][k] 和 B[k][j] 数据。而循环展开后，这些数据在缓存中的驻留时间变长，能够被多次复用。

例如，当计算 C[i][j] 时，A[i][k] 和 B[k][j] 已经在缓存中，可以直接使用，不需要再次访问主内存，从而减少了内存访问时间。

2) **提高数据复用率**: 在循环展开的计算过程中，矩阵 A 和 B 中的元素被多次用于计算不同的 C[i][j] 值。这种数据复用减少了数据传输的开销。因为数据只需要从主内存加载到缓存一次，就可以在多个计算步骤中使用。例如，矩阵 A 中的一列数据在计算多个 C[i][j] 时都被用到，而不是每次计算一个 C[i][j] 就加载一次，这样充分利用了缓存中的数据，提高了计算效率。

3) **减少循环迭代次数**: 虽然代码中看起来循环的嵌套结构没有改变，但循环展开实际上减少了每次循环迭代中需要进行的内存访问次数。因为每次迭代可以处理更多的计算任务，所以整体上减少了循环的迭代次数。这使得 CPU 可以更快地完成矩阵乘法的计算，减少了总的计算时间。

4.1.2 分块矩阵

分块矩阵方法是将原始的大矩阵划分为多个小的子矩阵（块），然后以块为单位进行计算。在上述代码中，通过引入 block_size 变量，将矩阵 A、B 和 C 分别划分为大小为 block_size × block_size 的子矩阵块。这种划分方式使得每次计算都在一个较小的数据范围内进行，更符合缓存的工作原理。

代码中的循环嵌套结构为 ii - jj - kk - i - j - k。外层的 ii、jj 和 kk 循环负责遍历各个子矩阵块，内层的 i、j 和 k 循环则负责在当前子矩阵块内进行具体的乘法和累加操作。这种结构确保了在计算每个子矩阵块时，数据访问具有较高的局部性。例如，当固定 ii、jj 和 kk 时，内层循环会集中访问矩阵 A、B 和 C 中对应子矩阵块内的元素，减少了缓存未命中的概率。

考虑到两级缓存的配置，一级缓存为 4KB，二级缓存为 16KB。分块矩阵方法通过合理设置 block_size，使得每个子矩阵块的大小能够适应缓存的容量。当 block_size 设置为 8 时，每个子矩阵块的大小为 8 × 8 × sizeof(int)（假设 int 类型占用 4 字节），即 256 字节。这样，多个子矩阵块可以同时驻留在一级或二级缓存中，提高了缓存的利用率。在计算过程中，可以充分利用缓存中的数据，减少对主内存的访问，从而提高计算效率。

```
void matrix_multiply(int A[M][K], int B[K][N], int C[M][N]) {
    int block_size = 8; // 假设块大小为16
    for (int ii = 0; ii < M; ii += block_size) {
        for (int jj = 0; jj < N; jj += block_size) {
            for (int kk = 0; kk < K; kk += block_size) {
                for (int i = ii; i < ii + block_size && i < M; i++) {
                    for (int j = jj; j < jj + block_size && j < N; j++) {
                        for (int k = kk; k < kk + block_size && k < K; k++) {
```

```
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

分析上述代码，优化原因如下：

1) **减少缓存未命中次数**: 由于分块矩阵方法使得数据访问具有较高的局部性，缓存命中率显著提高。在计算每个子矩阵块时，矩阵 A、B 和 C 中对应子矩阵块内的元素会在缓存中被频繁访问。例如，当计算 C[i][j] 时，所需的 A[i][k] 和 B[k][j] 数据很可能已经在缓存中，可以直接从缓存中读取，而不需要从主内存中加载。这减少了内存访问的延迟，加快了计算速度。

2) **提高数据复用率**: 分块矩阵方法减少了数据在主内存和缓存之间的传输次数。在传统的矩阵乘法实现中，数据可能会频繁地在主内存和缓存之间来回传输，导致大量的时间浪费在数据搬运上。而通过分块计算，每个子矩阵块的数据只需加载到缓存一次，就可以在多个计算步骤中复用。例如，矩阵 A 中的一块数据在计算多个 C[i][j] 值时都被用到，而不是每次计算一个 C[i][j] 就重新加载一次，从而减少了数据传输的开销，提高了计算效率。

3) **充分利用缓存容量**: 合理设置 block_size 使得子矩阵块的大小能够充分利用缓存的容量。一级缓存为 4KB，二级缓存为 16KB，通过将 block_size 设置为 8，每个子矩阵块的大小为 256 字节，多个子矩阵块可以同时驻留在缓存中。这样，在计算过程中，可以同时利用缓存中的多个子矩阵块进行计算，提高了缓存的利用率。相比于未分块的情况，缓存中的数据能够更好地服务于计算需求，减少了因缓存容量不足而导致的数据频繁替换和加载，从而优化了总时间。

4.1.3 外积展开

外积展开是将矩阵乘法中的内积计算转化为外积计算的一种方法。在矩阵乘法中，传统的计算方式是通过内积来计算结果矩阵 C 中的每个元素，即对于每个 C[i][j]，需要遍历 k 来计算 A[i][k] 和 B[k][j] 的乘积之和。而外积展开则是先计算矩阵 A 的一行与矩阵 B 的一列的外积，得到一个中间矩阵，然后将这个中间矩阵累加到结果矩阵 C 中。

在上述代码中，循环结构为 k - i - j。外层的 k 循环负责遍历矩阵 A 的列和矩阵 B 的行，内层的 i 和 j 循环则负责计算中间矩阵并累加到结果矩阵 C 中。这种结构使得每次计算都是在一个较小的数据范围内进行，更符合缓存的工作原理。具体来说，当固定

k 时，内层循环会集中访问矩阵 A 的第 i 行和矩阵 B 的第 k 列，以及结果矩阵 C 的第 i 行第 j 列，减少了缓存未命中的概率。

考虑到两级缓存的配置，一级缓存为 4KB，二级缓存为 16KB。外积展开方法通过合理安排数据访问顺序，使得每次计算的数据量能够适应缓存的容量。在计算过程中，矩阵 A 的第 i 行和矩阵 B 的第 k 列的数据会被频繁访问，这些数据可以驻留在缓存中，减少了对主内存的访问。同时，结果矩阵 C 的更新操作也是在缓存范围内进行，进一步提高了缓存的利用率。

```
void matrix_multiply(int A[M][K], int B[K][N], int C[M][N]) {
    for (int k = 0; k < K; ++k) {
        for (int i = 0; i < K; ++i){
            for (int j = 0; j < K; ++j){
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

分析上述代码，优化原因如下：

1) **减少缓存未命中次数**: 外积展开方法使得数据访问具有较高的局部性，缓存命中率显著提高。在计算过程中，矩阵 A 的第 i 行和矩阵 B 的第 k 列的数据会被多次访问，这些数据可以驻留在缓存中。例如，当计算 C[i][j] 时，所需的 A[i][k] 和 B[k][j] 数据很可能已经在缓存中，可以直接从缓存中读取，而不需要从主内存中加载。这减少了内存访问的延迟，加快了计算速度。

2) **提高数据复用率**: 外积展开方法减少了数据在主内存和缓存之间的传输次数。在传统的矩阵乘法实现中，数据可能会频繁地在主内存和缓存之间来回传输，导致大量的时间浪费在数据搬运上。而通过外积展开，每次计算的数据量较小，可以充分利用缓存中的数据进行计算。例如，矩阵 A 的一行数据和矩阵 B 的一列数据在计算多个 C[i][j] 值时都被用到，而不是每次计算一个 C[i][j] 就重新加载一次，从而减少了数据传输的开销，提高了计算效率。

3) **充分利用缓存容量**: 合理安排数据访问顺序使得外积展开方法能够充分利用缓存的容量。一级缓存为 4KB，二级缓存为 16KB，通过将数据访问集中在较小的范围内，多个数据块可以同时驻留在缓存中。在计算过程中，可以同时利用缓存中的多个数据块进行计算，提高了缓存的利用率。相比于未优化的情况，缓存中的数据能够更好地服务于计算需求，减少了因缓存容量不足而导致的数据频繁替换和加载，从而优化了总时间。

4) **减少计算复杂度**: 虽然外积展开方法在代码结构上看起来与传统的矩阵乘法类似，但

实际上通过集中处理外积计算，减少了不必要的计算步骤。每次计算一个外积并直接累加到结果矩阵 C 中，避免了多次重复计算中间结果。这种计算方式在处理大规模矩阵时尤其有效，能够显著减少计算复杂度，提高计算效率。

4.1.4 矩阵转置

在传统的矩阵乘法中，计算 C[i][j] 时需要遍历矩阵 A 的第 i 行和矩阵 B 的第 j 列。然而，这种访问模式在内存中并不是连续的，尤其是对于矩阵 B 的列访问，会导致大量的缓存未命中。为了改善这一点，代码中引入了矩阵 B 的转置版本 B_transposed。这样，原本需要访问矩阵 B 的列的操作，现在变成了访问 B_transposed 的行，从而使得内存访问更加连续。

代码中的循环结构为 i - j - k。外层的 i 循环遍历矩阵 A 的行，中间的 j 循环遍历矩阵 C 的列（同时也是 B_transposed 的行），内层的 k 循环负责具体的乘法和累加操作。这种结构使得在计算过程中，对矩阵 A 的行和 B_transposed 的行的访问都是连续的，符合缓存的局部性原理。

考虑到两级缓存的配置，一级缓存为 4KB，二级缓存为 16KB。通过矩阵转置，每次计算时可以将矩阵 A 的一行和 B_transposed 的一行加载到缓存中。由于这些行在内存中是连续存储的，它们可以高效地利用缓存的容量。这样，在计算过程中，可以充分利用缓存中的数据，减少对主内存的访问，从而提高计算效率。

```
void matrix_multiply(int A[M][K], int B_transposed[N][K], int C[M][N]) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            for (int k = 0; k < K; ++k) {
                C[i][j] += A[i][k] * B_transposed[j][k];
            }
        }
    }
}
```

分析上述代码，优化原因如下：

1) **减少缓存未命中次数**: 矩阵转置方法使得数据访问具有较高的局部性，缓存命中率显著提高。在计算过程中，矩阵 A 的第 i 行和 B_transposed 的第 j 行的数据会被频繁访问，这些数据可以驻留在缓存中。例如，当计算 C[i][j] 时，所需的 A[i][k] 和 B_transposed[j][k] 数据很可能已经在缓存中，可以直接从缓存中读取，而不需要从主内存中加载。这减少了内存访问的延迟，加快了计算速度。

2) **提高数据复用率**: 矩阵转置方法减少了数据在主内存和缓存之间的传输次数。在传统的矩阵乘法实现中，访问矩阵 B 的列会导致频繁的缓存未命中，因为列访问在内存

中不是连续的。而通过转置矩阵 B，每次计算的数据量较小，可以充分利用缓存中的数据进行计算。例如，矩阵 A 的一行数据和 B_transposed 的一行数据在计算多个 C[i][j] 值时都被用到，而不是每次计算一个 C[i][j] 就重新加载一次，从而减少了数据传输的开销，提高了计算效率。

3) **充分利用缓存容量**：合理安排数据访问顺序使得矩阵转置方法能够充分利用缓存的容量。一级缓存为 4KB，二级缓存为 16KB，通过将数据访问集中在较小的范围内，多个数据块可以同时驻留在缓存中。在计算过程中，可以同时利用缓存中的多个数据块进行计算，提高了缓存的利用率。相比于未优化的情况，缓存中的数据能够更好地服务于计算需求，减少了因缓存容量不足而导致的数据频繁替换和加载，从而优化了总时间。

4) **减少计算复杂度**：虽然矩阵转置方法在代码结构上看起来与传统的矩阵乘法类似，但实际上通过转置矩阵 B，减少了不必要的计算步骤。每次计算一个 C[i][j] 时，访问的是连续的内存区域，避免了多次重复计算中间结果。这种计算方式在处理大规模矩阵时尤其有效，能够显著减少计算复杂度，提高计算效率。

4.1.5 实验结果分析

基础软件优化策略效果如下图：本实验旨在评估不同优化策略对处理器性能的影响，具体通过比较不同策略下的执行时间、L2 缓存未命中率和 L1 缓存未命中率来进行分析。以下是对实验结果的详细分析：

从图 (a) 可以看出，与基线（Baseline）相比，所有优化策略均能显著减少执行时间。具体来说：

1) 4 次循环展开策略减少了 22.12% 的执行时间。

2) 8 次循环展开策略减少了 24.26% 的执行时间。

3) 16 次循环展开策略减少了 26.76% 的执行时间。

4) 8 级分块策略减少了 37.69% 的执行时间，是所有策略中效果最显著的。

5) 16-Level Tiling（16 级分块）策略减少了 23.95% 的执行时间。

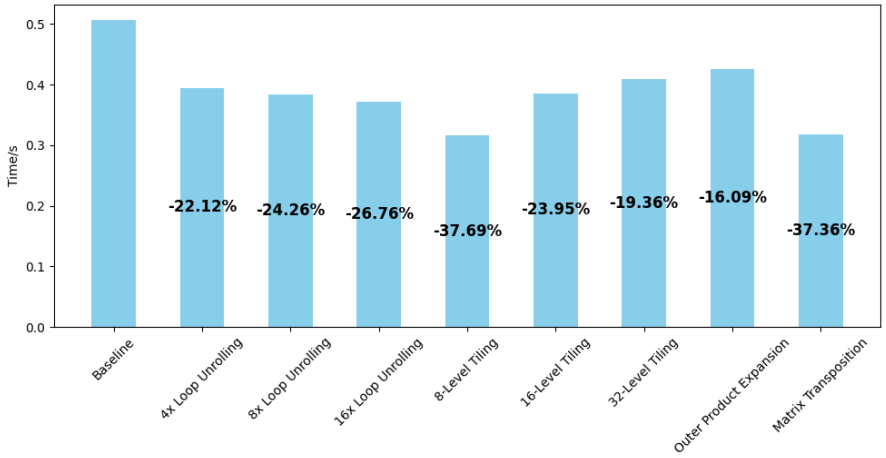
6) 32 级分块策略减少了 19.36% 的执行时间。

7) 外积展开策略减少了 16.09% 的执行时间。

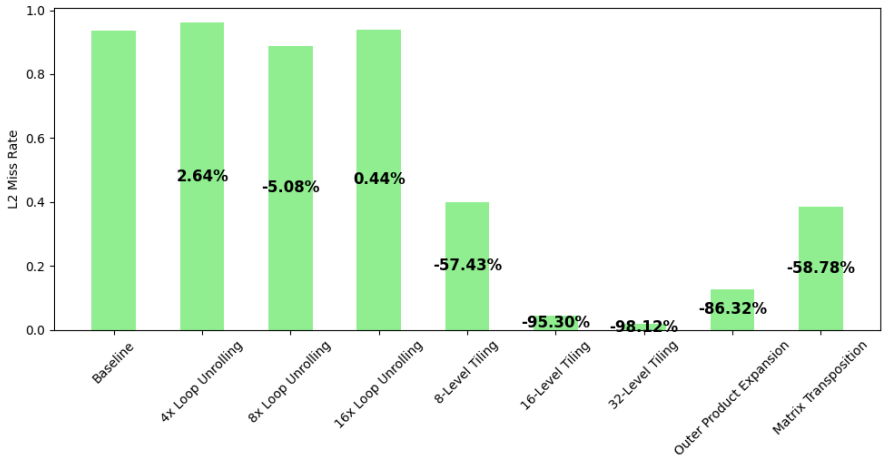
8) 矩阵转置策略减少了 37.36% 的执行时间，效果仅次于 8 级分块策略。

图 (b) 展示了不同优化策略对 L2 缓存未命中率的影响：

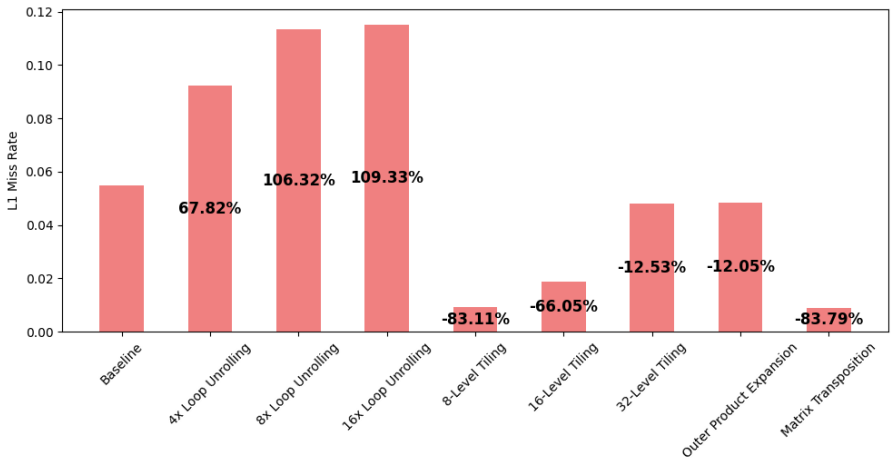
1) 基线的 L2 缓存未命中率为 0.9。



(a)Time/s Comparison



(b)L2 Miss Rate Comparison



(c)L1 Miss Rate Comparison

图 5: 不同基础软件优化策略的优化结果

2) 4 次循环展开策略略微增加了 2.64% 的未命中率。

3) 8 次循环展开减少了 5.08% 的未命中率。

4) 16 次循环展开策略减少了 0.44% 的未命中率。

5) 8 级分块策略策略减少了 57.43% 的未命中率。

6) 16 级分块策略策略减少了 95.30% 的未命中率，效果最为显著。

7) 32 级分块策略策略减少了 98.12% 的未命中率，效果最佳。

8) 外积展开策略减少了 86.32% 的未命中率。

9) 矩阵转置策略减少了 58.78% 的未命中率。

图 (c) 展示了不同优化策略对 L1 缓存未命中率的影响：

1) 基线的 L1 缓存未命中率为 0.05。

2) 4 次循环展开策略增加了 67.82% 的未命中率。

3) 8 次循环展开策略增加了 106.32% 的未命中率。

4) 16 次循环展开策略增加了 109.33% 的未命中率。

5) 8 级分块策略策略减少了 83.11% 的未命中率。

6) 16 级分块策略策略减少了 66.05% 的未命中率。

7) 32 级分块策略减少了 12.53% 的未命中率。

8) 外积展开策略减少了 12.05% 的未命中率。

9) 矩阵转置策略减少了 83.79% 的未命中率。

综合以上分析，可以得出以下结论：

- 执行时间：8 级分块和矩阵转置策略在减少执行时间方面效果最为显著。
- L2 缓存未命中率：32 级分块策略在降低 L2 缓存未命中率方面效果最佳。
- L1 缓存未命中率：8 级分块和矩阵转置策略在降低 L1 缓存未命中率方面效果最为显著。

4.2 混合软件优化

4.2.1 设计思路分析

通过观察我们可以发现，矩阵转置解决的问题在其他几种优化方案中仍然存在，所以通过转置矩阵的方式可以与其余几种基本优化策略混合，形成新的混合软件优化策略，可以达到更优的优化的效果。转置矩阵 + 其余基本优化策略的混合软件优化策略代码实现如下。

```
//循环展开+矩阵转置
void matrix_multiply(int A[M][K], int B_transposed[N][K], int C[M][N]) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < K; k += 8) {
                C[i][j] += A[i][k] * B_transposed[j][k] + A[i][k+1] *
                    B_transposed[j][k+1] +
                        A[i][k+2] * B_transposed[j][k+2] + A[i][k+3] *
                            B_transposed[j][k+3] +
                        A[i][k+4] * B_transposed[j][k+4] + A[i][k+5] *
                            B_transposed[j][k+5] +
                        A[i][k+6] * B_transposed[j][k+6] + A[i][k+7] *
                            B_transposed[j][k+7];
            }
        }
    }
}
//分块矩阵+矩阵转置
void matrix_multiply(int A[M][K], int B_transposed[N][K], int C[M][N]) {
    int block_size = 32; // 假设块大小为16
    for (int ii = 0; ii < M; ii += block_size) {
        for (int jj = 0; jj < N; jj += block_size) {
            for (int kk = 0; kk < K; kk += block_size) {
                for (int i = ii; i < ii + block_size && i < M; i++) {
                    for (int j = jj; j < jj + block_size && j < N; j++) {
                        for (int k = kk; k < kk + block_size && k < K; k++) {
                            C[i][j] += A[i][k] * B_transposed[j][k];
                        }
                    }
                }
```

```
            }
        }
    }
}
//外积展开+矩阵转置
void matrix_multiply(int A[M][K], int B_transposed[N][K], int C[M][N]) {
    for (int k = 0; k < K; ++k) {
        for (int i = 0; i < K; ++i){
            for (int j = 0; j < K; ++j){
                C[i][j] += A[i][k] * B_transposed[j][k];
            }
        }
    }
}
```

4.2.2 实验结果分析

混合软件优化策略效果如下图：本实验通过对比不同混合优化策略对处理器性能的影响，具体分析了执行时间、L2 缓存未命中率以及 L1 缓存未命中率的变化情况。

从图 (a) 可以看出，与基线（Baseline）相比，所有优化策略均能显著减少执行时间。具体结果如下：

1) 矩阵转置 +4 次循环展开策略减少了 59.62% 的执行时间。

2) 矩阵转置 +8 次循环展开策略减少了 64.01% 的执行时间。

3) 矩阵转置 +16 次循环展开策略减少了 65.90% 的执行时间。

4) 矩阵转置 +8 级分块策略减少了 28.42% 的执行时间。

5) 矩阵转置 +16 级分块策略减少了 29.77% 的执行时间。

6) 矩阵转置 +32 级分块策略减少了 30.23% 的执行时间。

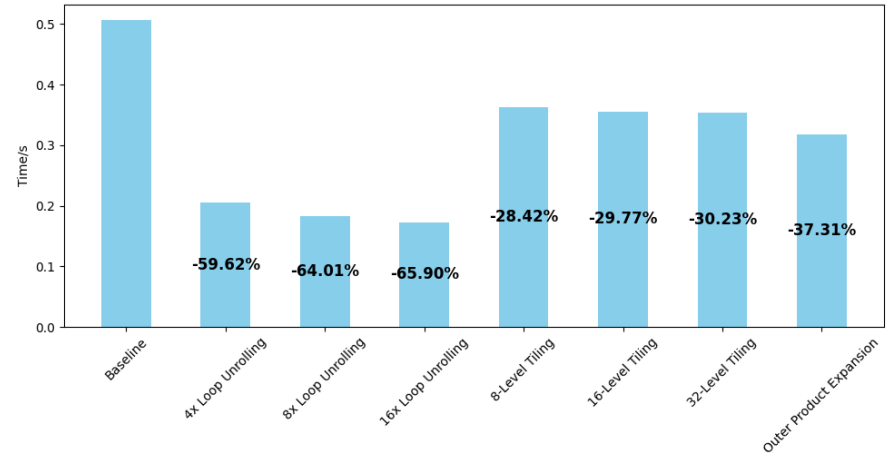
7) 矩阵转置 + 外积展开策略减少了 37.31% 的执行时间。

图 (b) 展示了不同优化策略对 L2 缓存未命中率的影响：

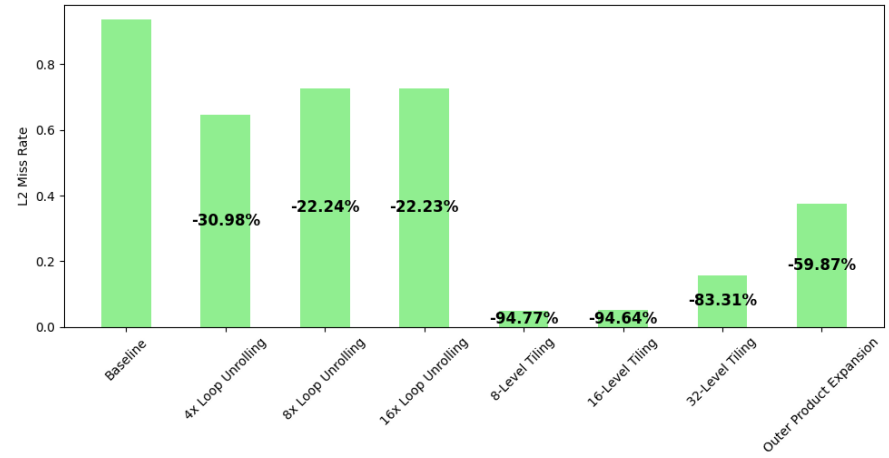
1) 基线（Baseline）的 L2 缓存未命中率较高。

2) 4 次循环展开策略减少了 30.98% 的未命中率。

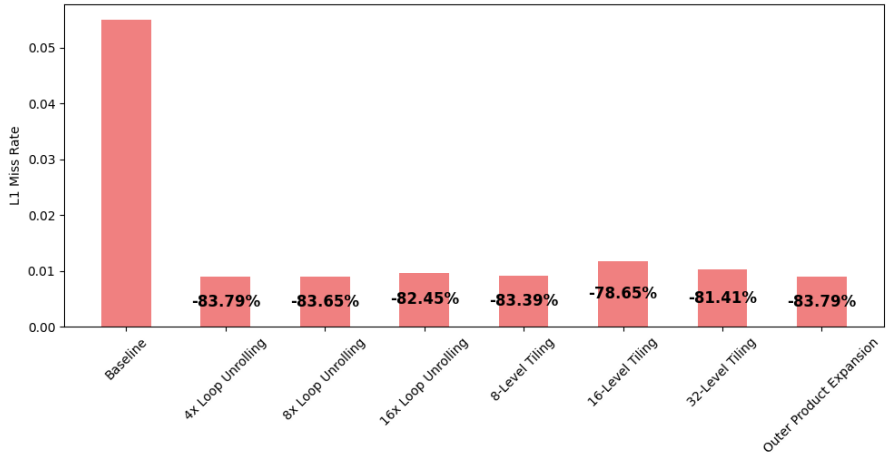
3) 8 次循环展开策略减少了 22.24% 的未命中率。



(a)Time/s Comparison



(b)L2 Miss Rate Comparison



(c)L1 Miss Rate Comparison

图 6: 不同混合软件优化策略的优化结果

4) 16 次循环展开策略减少了 22.23% 的未命中率。

5) 8 级分块策略减少了 94.77% 的未命中率，效果最为显著。

6) 16 级分块策略减少了 94.64% 的未命中率。

7) 32 级分块策略减少了 83.31% 的未命中率。

8) 外积展开策略减少了 59.87% 的未命中率。

图 (c) 展示了不同优化策略对 L1 缓存未命中率的影响：

1) 矩阵转置 + 基线（Baseline）的 L1 缓存未命中率为 0.056。

2) 矩阵转置 +4 次循环展开策略减少了 83.79% 的未命中率。

3) 矩阵转置 +8 次循环展开策略减少了 83.65% 的未命中率。

4) 矩阵转置 +16 次循环展开策略减少了 82.45% 的未命中率。

5) 矩阵转置 +8 级分块策略减少了 83.39% 的未命中率。

6) 矩阵转置 +16 级分块策略减少了 78.65% 的未命中率。

7) 矩阵转置 +32 级分块策略减少了 81.41% 的未命中率。

8) 矩阵转置 + 外积展开策略减少了 83.79% 的未命中率。

综合以上分析，可以得出以下结论：

- 在减少执行时间方面，矩阵转置 +16 次循环展开策略效果最佳，减少了 65.90% 的执行时间。
- 在降低 L2 缓存未命中率方面，矩阵转置 +8 级分块策略效果最为显著，减少了 94.77% 的未命中率。
- 在降低 L1 缓存未命中率方面，矩阵转置 + 外积展开策略效果最佳，减少了 83.79% 的未命中率。

4.3 软硬件协同优化

4.3.1 设计思路分析

在多级缓存体系中，缓存行大小（cache line size）是指缓存中每个数据块的大小。调整缓存行大小是为了更好地匹配矩阵乘法（GEMM）操作中的数据访问模式，从而提高缓存的利用率。通过软件和硬件的协同优化，可以使得数据在缓存中的存储和访问更加高效。

软件优化需要与硬件配置相结合，以实现最佳性能。通过调整缓存行大小，可以使得软件中的分块矩阵大小与硬件缓存的配置相匹配。例如，可以使用编译器指令或系统调用调整缓存行大小，或者在软件中通过合理的数据访问模式来适应硬件的缓存行大小。

我们基于在上述实验中提到的提出策略中相对最优的优化策略矩阵转置 +16 次循环展开策略，来研究的不同缓存行大小对性能与缓存未命中率的影响，以平衡两级缓存的未命中率，以获得最低的执行时间。

修改 system.py 文件中的以下代码，修改行缓存大小。

```
cache_line_size = Param.Unsigned(64, "Cache line size in bytes")
```

优化原因如下：

1) **减少缓存未命中次数**: 通过调整缓存行大小，使得分块矩阵的大小与缓存行大小相匹配，可以显著提高缓存命中率。在计算过程中，矩阵 A 和 B 的分块数据可以高效地加载到缓存中，并且在计算过程中多次复用，减少了对主内存的访问。

2) **提高数据复用率**: 调整缓存行大小后，每次加载到缓存中的数据块可以被充分利用，减少了数据在主内存和缓存之间的传输次数。这不仅减少了内存访问的延迟，还提高了数据传输的效率。

3) **充分利用缓存容量**: 合理设置分块矩阵的大小，使得多个数据块可以同时驻留在缓存中。这样，在计算过程中可以同时利用缓存中的多个数据块进行计算，提高了缓存的利用率。相比于未优化的情况，缓存中的数据能够更好地服务于计算需求，减少了因缓存容量不足而导致的数据频繁替换和加载。

4) **减少计算复杂度**: 通过分块矩阵方法，每次计算的数据量较小，可以充分利用缓存中的数据进行计算。这不仅减少了不必要的计算步骤，还提高了计算效率。每次计算一个分块矩阵时，访问的是连续的内存区域，避免了多次重复计算中间结果。

4.3.2 实验结果分析

软硬件协同优化策略效果如下图：

本实验旨在评估不同缓存行大小对处理器性能的影响，具体通过比较不同缓存行大小下的执行时间、L2 缓存未命中率以及 L1 缓存未命中率来进行分析。

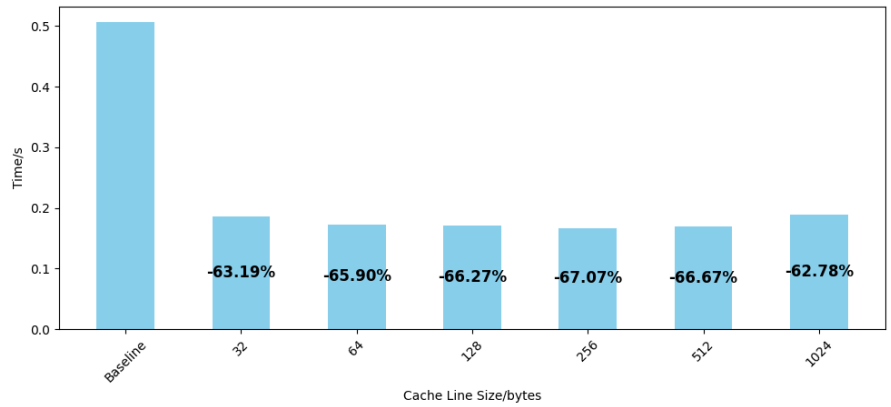
图 (a) 展示了不同缓存行大小对执行时间的影响：

1) 基线（Baseline）的执行时间为最高。

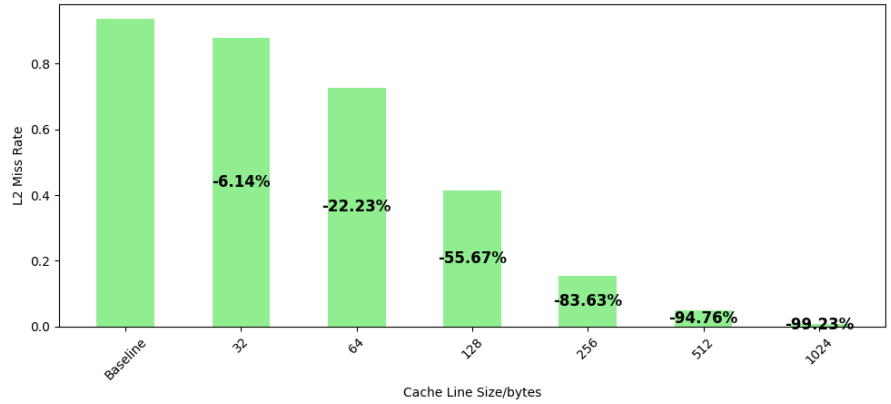
2) 当缓存行大小为 32 字节时，执行时间减少了 63.19%。

3) 当缓存行大小为 64 字节时，执行时间减少了 65.90%。

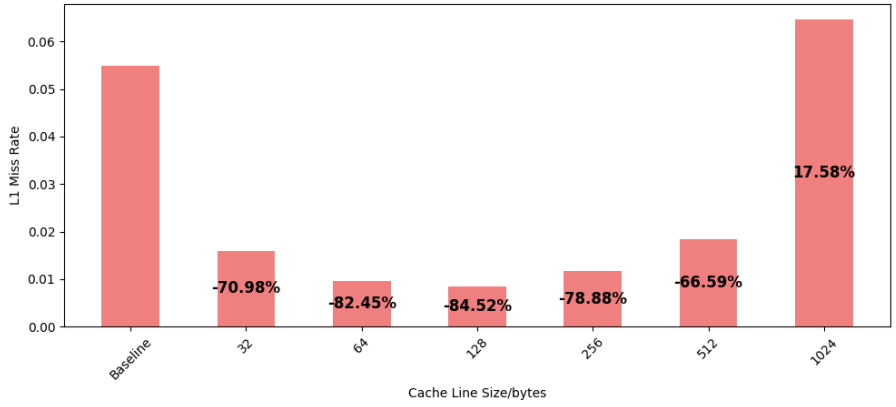
4) 当缓存行大小为 128 字节时，执行时间减少了 66.27%。



(a)Time/s Comparison



(b)L2 Miss Rate Comparison



(c)L1 Miss Rate Comparison

图 7: 不同缓存行尺寸的软硬件协同优化结果

5) 当缓存行大小为 256 字节时，执行时间减少了 67.07%。

6) 当缓存行大小为 512 字节时，执行时间减少了 66.67%。

7) 当缓存行大小为 1024 字节时，执行时间减少了 62.78%。

图 (b) 展示了不同缓存行大小对 L2 缓存未命中率的影响：

1) 基线（Baseline）的 L2 缓存未命中率为最高。

2) 当缓存行大小为 32 字节时，未命中率减少了 6.14%。

3) 当缓存行大小为 64 字节时，未命中率减少了 22.23%。

4) 当缓存行大小为 128 字节时，未命中率减少了 55.67%。

5) 当缓存行大小为 256 字节时，未命中率减少了 83.63%。

6) 当缓存行大小为 512 字节时，未命中率减少了 94.76%。

7) 当缓存行大小为 1024 字节时，未命中率减少了 99.23%，效果最佳。

图 (c) 展示了不同缓存行大小对 L1 缓存未命中率的影响：

1) 基线（Baseline）的 L1 缓存未命中率为 0.056。

2) 当缓存行大小为 32 字节时，未命中率减少了 70.98%。

3) 当缓存行大小为 64 字节时，未命中率减少了 82.45%。

4) 当缓存行大小为 128 字节时，未命中率减少了 84.52%。

5) 当缓存行大小为 256 字节时，未命中率减少了 78.88%。

6) 当缓存行大小为 512 字节时，未命中率减少了 66.59%。

7) 当缓存行大小为 1024 字节时，未命中率增加了 17.58%，效果最差。

综合以上分析，可以得出以下结论：

- 在减少执行时间方面，缓存行大小为 256 字节时效果最佳，减少了 67.07% 的执行时间。
- 在降低 L2 缓存未命中率方面，缓存行大小为 1024 字节时效果最佳，减少了 99.23% 的未命中率。
- 在降低 L1 缓存未命中率方面，缓存行大小为 64 字节时效果最佳，减少了 82.45% 的未命中率。

5 实验结果总结

本章对实验的结果进行总结，**相关软硬件优化策略、实现以及全部实验结论已在第三章、第四章详细阐述**，这里不做过多赘述，下面进行实验结果进行数据整理与总结。

5.1 硬件优化实验结果

表 9: 不同缓存配置下的矩阵乘法性能

L1d Size (kB)	L2 Size (kB)	执行时间 (秒)	L2 Cache Miss Rate	D-Cache Miss Rate
4	16	0.506695	0.935581	0.055031
4	32	0.506665	0.935347	0.055031
4	64	0.363563	0.078919	0.055031
4	128	0.349067	0.002125	0.055031
8	128	0.347262	0.002198	0.053202
16	128	0.346313	0.002241	0.052235
32	128	0.345933	0.002257	0.051852
64	128	0.296492	0.059292	0.001853
128	256	0.295467	0.110253	0.000855

表 10: 不同缓存替换策略下的 GEMM 的性能比较

Replacement Policy	**Execution Time (s)**	**D-Cache Miss Rate**	**L2 Miss Rate**
LRU	0.506695	0.055031	0.935581
MRU	0.553967	0.085088	0.646769
PLRU	0.506663	0.055031	0.935385
LFU	0.540417	0.074118	0.770182
BIP	0.480627	0.053039	0.812913
LIP	0.475470	0.053621	0.778739
Random	0.515646	0.061876	0.823562
FIFO	0.517659	0.061788	0.833527
NRU	0.505182	0.053848	0.956191
RRI	0.504813	0.053611	0.960413

表 11: WeightedRandom 策略与 Random 策略

Replacement Policy	**Execution Time (s)**	**D-Cache Miss Rate**	**L2 Miss Rate**
Random	0.515646	0.061876	0.823562
WeightedRandom weight:1×	0.523583	0.068558	0.748128
WeightedRandom weight:10x	0.528192	0.071743	0.721079

表 12: X86 架构与 RISC-V 架构仿真结果对比

架构	**CISC-X86**	**RISC-V**
执行时间 (秒)	0.506695	0.506105
L2 Miss Rate	0.935581	0.914345
D-Cache Miss Rate	0.055031	0.056489
CPU CPI	5.815975	5.526771
simInsts	88682623	93321418
simOps	109910810	93354195
total dram read	2120825	2120491
total dram write	3275	3189

表 13: 不同处理器架构性能对比

CPU 类型	**(D-cache，l2-cache)**	**执行时间 (秒)**	**CPU CPI**
TimingSimple	(4kB，16kB)	0.506695	5.815975
	(128kB，256kB)	0.295467	3.434031
O3	(4kB，16kB)	0.262001	2.970501
	(128kB，256kB)	0.057787	0.667700

5.2 软件优化实验结果

5.2.1 基础软件优化方法

不同基础软件优化方法的 gem5 仿真结果对比如表 14所示：

表 14: 不同基础软件优化方法的性能比较

Optimization Methods	Execution Time (s)	D-Cache Miss Rate	L2 Miss Rate
Baseline	0.506544	0.054967	0.93561
4 次循环展开	0.394497	0.092246	0.960326
8 次循环展开	0.38367	0.113408	0.888106
16 次循环展开	0.370989	0.115061	0.93974
8 阶分块矩阵	0.385225	0.018659	0.043953
16 阶分块矩阵	0.408459	0.048077	0.017554
32 阶分块矩阵	0.425039	0.048344	0.127964
外积展开	0.315632	0.009282	0.398254
矩阵转置	0.317306	0.008911	0.385626

从表格中可以看出，执行时间的优化情况：

1) 4 次循环展开策略减少了 22.12% 的执行时间。

2) 8 次循环展开策略减少了 24.26% 的执行时间。

3) 16 次循环展开策略减少了 26.76% 的执行时间。

4) 8 级分块策略减少了 37.69% 的执行时间，是所有策略中效果最显著的。

5) 16-Level Tiling（16 级分块）策略减少了 23.95% 的执行时间。

6) 32 级分块策略减少了 19.36% 的执行时间。

7) 外积展开策略减少了 16.09% 的执行时间。

8) 矩阵转置策略减少了 37.36% 的执行时间，效果仅次于 8 级分块策略。

L2 缓存未命中率优化情况：

1) 基线的 L2 缓存未命中率为 0.9。

2) 4 次循环展开策略略微增加了 2.64% 的未命中率。

3) 8 次循环展开减少了 5.08% 的未命中率。

4) 16 次循环展开策略减少了 0.44% 的未命中率。

5) 8 级分块策略策略减少了 57.43% 的未命中率。

6) 16 级分块策略策略减少了 95.30% 的未命中率，效果最为显著。

7) 32 级分块策略策略减少了 98.12% 的未命中率，效果最佳。

8) 外积展开策略减少了 86.32% 的未命中率。

9) 矩阵转置策略减少了 58.78% 的未命中率。

L1 缓存未命中率优化情况：

1) 基线的 L1 缓存未命中率为 0.05。

2) 4 次循环展开策略增加了 67.82% 的未命中率。

3) 8 次循环展开策略增加了 106.32% 的未命中率。

4) 16 次循环展开策略增加了 109.33% 的未命中率。

5) 8 级分块策略策略减少了 83.11% 的未命中率。

6) 16 级分块策略策略减少了 66.05% 的未命中率。

7) 32 级分块策略减少了 12.53% 的未命中率。

8) 外积展开策略减少了 12.05% 的未命中率。

9) 矩阵转置策略减少了 83.79% 的未命中率。

综合以上分析，可以得出以下结论：

- 执行时间：8 级分块和矩阵转置策略在减少执行时间方面效果最为显著。
- L2 缓存未命中率：32 级分块策略在降低 L2 缓存未命中率方面效果最佳。
- L1 缓存未命中率：8 级分块和矩阵转置（Matrix Transposition）策略在降低 L1 缓存未命中率方面效果最为显著。

5.2.2 混合软件优化方法

不同混合软件优化方法的 gem5 仿真结果对比如表 15所示：
从表格中可以看出，执行时间的优化情况：

1) 矩阵转置 +4 次循环展开策略减少了 59.62% 的执行时间。

2) 矩阵转置 +8 次循环展开策略减少了 64.01% 的执行时间。

表 15: 不同混合软件优化方法的性能比较

Optimization Methods	**Execution Time (s)**	**D-Cache Miss Rate**	**L2 Miss Rate**
Baseline	0.506544	0.054967	0.93561
矩阵转置 +4 次循环展开	0.204561	0.008911	0.645764
矩阵转置 +8 次循环展开	0.182287	0.008989	0.727564
矩阵转置 +16 次循环展开	0.172753	0.009649	0.727583
矩阵转置 +8 阶分块矩阵	0.362576	0.009129	0.048961
矩阵转置 +16 阶分块矩阵	0.355766	0.011733	0.050124
矩阵转置 +32 阶分块矩阵	0.353405	0.010219	0.156119
矩阵转置 + 外积展开	0.317559	0.008909	0.375458

3) 矩阵转置 +16 次循环展开策略减少了 65.90% 的执行时间。

4) 矩阵转置 +8 级分块策略减少了 28.42% 的执行时间。

5) 矩阵转置 +16 级分块策略减少了 29.77% 的执行时间。

6) 矩阵转置 +32 级分块策略减少了 30.23% 的执行时间。

7) 矩阵转置 + 外积展开策略减少了 37.31% 的执行时间。

L2 缓存未命中率优化情况：

1) 基线（Baseline）的 L2 缓存未命中率较高。

2) 4 次循环展开策略减少了 30.98% 的未命中率。

3) 8 次循环展开策略减少了 22.24% 的未命中率。

4) 16 次循环展开策略减少了 22.23% 的未命中率。

5) 8 级分块策略减少了 94.77% 的未命中率，效果最为显著。

6) 16 级分块策略减少了 94.64% 的未命中率。

7) 32 级分块策略减少了 83.31% 的未命中率。

8) 外积展开策略减少了 59.87% 的未命中率。

L1 缓存未命中率优化情况：

1) 矩阵转置 + 基线（Baseline）的 L1 缓存未命中率为 0.056。

2) 矩阵转置 +4 次循环展开策略减少了 83.79% 的未命中率。

3) 矩阵转置 +8 次循环展开策略减少了 83.65% 的未命中率。

4) 矩阵转置 +16 次循环展开策略减少了 82.45% 的未命中率。

5) 矩阵转置 +8 级分块策略减少了 83.39% 的未命中率。

6) 矩阵转置 +16 级分块策略减少了 78.65% 的未命中率。

7) 矩阵转置 +32 级分块策略减少了 81.41% 的未命中率。

8) 矩阵转置 + 外积展开策略减少了 83.79% 的未命中率。

综合以上分析，可以得出以下结论：

- 在减少执行时间方面，矩阵转置 +16 次循环展开策略效果最佳，减少了 65.90% 的执行时间。
- 在降低 L2 缓存未命中率方面，矩阵转置 +8 级分块策略效果最为显著，减少了 94.77% 的未命中率。
- 在降低 L1 缓存未命中率方面，矩阵转置 + 外积展开策略效果最佳，减少了 83.79% 的未命中率。

5.3 软硬件协同优化实验结果

基于矩阵转置 +16 次循环展开的不同缓存行大小的 gem5 仿真结果对比如表 16所示：从表格中可以看出，执行时间的优化情况：

表 16: 不同缓存行大小的性能比较

Cache Line Size (bytes)	Execution Time (s)	D-Cache Miss Rate	L2 Miss Rate
Baseline	0.506544	0.054967	0.93561
32	0.186459	0.015949	0.878192
64	0.172753	0.009649	0.727583
128	0.170842	0.008509	0.414715
256	0.166814	0.01161	0.153146
512	0.168839	0.018366	0.048999
1024	0.188528	0.007159	0.064628

1) 基线（Baseline）的执行时间为最高。

2) 当缓存行大小为 32 字节时，执行时间减少了 63.19%。

3) 当缓存行大小为 64 字节时，执行时间减少了 65.90%。

4) 当缓存行大小为 128 字节时，执行时间减少了 66.27%。

5) 当缓存行大小为 256 字节时，执行时间减少了 67.07%。

6) 当缓存行大小为 512 字节时，执行时间减少了 66.67%。

7) 当缓存行大小为 1024 字节时，执行时间减少了 62.78%。

L2 缓存未命中率优化情况：

1) 基线（Baseline）的 L2 缓存未命中率为最高。

2) 当缓存行大小为 32 字节时，未命中率减少了 6.14%。

3) 当缓存行大小为 64 字节时，未命中率减少了 22.23%。

4) 当缓存行大小为 128 字节时，未命中率减少了 55.67%。

5) 当缓存行大小为 256 字节时，未命中率减少了 83.63%。

6) 当缓存行大小为 512 字节时，未命中率减少了 94.76%。

7) 当缓存行大小为 1024 字节时，未命中率减少了 99.23%，效果最佳。

L1 缓存未命中率优化情况：

1) 基线（Baseline）的 L1 缓存未命中率为 0.056。

2) 当缓存行大小为 32 字节时，未命中率减少了 70.98%。

3) 当缓存行大小为 64 字节时，未命中率减少了 82.45%。

4) 当缓存行大小为 128 字节时，未命中率减少了 84.52%。

5) 当缓存行大小为 256 字节时，未命中率减少了 78.88%。

6) 当缓存行大小为 512 字节时，未命中率减少了 66.59%。

7) 当缓存行大小为 1024 字节时，未命中率增加了 17.58%，效果最差。

综合以上分析，可以得出以下结论：

- 在减少执行时间方面，缓存行大小为 256 字节时效果最佳，减少了 67.07% 的执行时间。
- 在降低 L2 缓存未命中率方面，缓存行大小为 1024 字节时效果最佳，减少了 99.23% 的未命中率。
- 在降低 L1 缓存未命中率方面，缓存行大小为 64 字节时效果最佳，减少了 82.45% 的未命中率。

A 自动化仿真脚本

```
#!/bin/bash

# 定义gem5的路径和配置文件
GEM5_PATH="build/X86/gem5.opt"
CONFIG_FILE="configs/learning_gem5/part1/my_two_level.py"
BINARY_PATH="tests/gemm/baseline"
OUTPUT_FILE="autorun_cache_rs.txt"
STATS_FILE="m5out/stats.txt"

> $OUTPUT_FILE

# 定义要测试的L1d和L2缓存容量组合
declare -a l1d_sizes=("4kB" "4kB" "4kB" "4kB" "8kB" "16kB" "32kB" "64kB"
    "128kB")
declare -a l2_sizes=("16kB" "32kB" "64kB" "128kB" "128kB" "128kB" "128kB"
    "128kB" "256kB")

# 循环执行gem5仿真
for i in "${!l1d_sizes[@]}"; do
    l1d_size=${l1d_sizes[$i]}
    l2_size=${l2_sizes[$i]}

    # 将提取的统计数据追加到输出文件
    echo "L1d size: $l1d_size, L2 size: $l2_size" >> $OUTPUT_FILE

    #提取时间
    $GEM5_PATH $CONFIG_FILE $BINARY_PATH --l1d_size=$l1d_size
        --l2_size=$l2_size 2>&1 | \
    grep -E "Matrix multiplication took" >> $OUTPUT_FILE

    #提取stats的Missrate
    l2_miss_rate=$(grep "system.l2cache.overallMissRate::total" $STATS_FILE |
        awk '{print $2}')
    dcache_miss_rate=$(grep "system.cpu.dcache.overallMissRate::total"
        $STATS_FILE | awk '{print $2}')
```

```
    echo "L2 Cache Overall Miss Rate: $l2_miss_rate" >> $OUTPUT_FILE
    echo "D-Cache Overall Miss Rate: $dcache_miss_rate" >> $OUTPUT_FILE

    # 添加空行以分隔不同的测试组输出
    echo "" >> $OUTPUT_FILE
done

echo "Simulation outputs recorded in $OUTPUT_FILE"
```

B 自行实现 WeightedRandom 核心代码

```
ReplaceableEntry*
WeightedRandom::getVictim(const ReplacementCandidates& candidates) const
{
    // There must be at least one replacement candidate
    assert(candidates.size() > 0);
    ReplaceableEntry* victim = nullptr;
    //------------------------------ weighted random
        ----------------------------------------
    int totalWeight = 0;
    for (const auto& candidate : candidates) {
        totalWeight += std::static_pointer_cast<WeightedRandomReplData>(
            candidate->replacementData)->weight;
    }
    int randomValue = random_mt.random<int>(0,totalWeight);
    int accumulatedWeight = 0;
    for (const auto& candidate : candidates) {
        accumulatedWeight +=
            std::static_pointer_cast<WeightedRandomReplData>(
            candidate->replacementData)->weight;
        if (accumulatedWeight >= randomValue) {
            victim = candidate;
            break;
        }
    }
    //------------------------------------------------------------
    // Choose one candidate at random
```

```
    // ReplaceableEntry* victim = candidates[random_mt.random<unsigned>(0,
    // candidates.size() - 1)];

    // Visit all candidates to search for an invalid entry. If one is found,
    // its eviction is prioritized
    for (const auto& candidate : candidates) {
        if
            (!std::static_pointer_cast<WeightedRandomReplData>(candidate->replacementData)-
            {
            victim = candidate;
            break;
        }
    }

    return victim;
}
```