2025 秋编译原理 (H)
课程探索题目

(第 1 版)

张昱

(2025.12.23 更新)

目录

第 1 章　2025 秋编译原理 (H) 课程探索题目

1.1　面向鲲鹏 CPU 的动态图调度优化

- 背景赛题信息：动态算子图编译与并行调度
- 实验的公共仓库：基于 Clang Driver 的驱动示例程序仓库 以及助教提供的分块 cholesky 分解算法程序及正确性验证脚本。
- 运行环境：鲲鹏服务器，已提供账号信息。环境使用详情参考 基于 Clang Driver 的驱动示例程序仓库。

1.1.1　实验信息

本实验的核心目标是在确保程序功能正确且误差符合要求的前提下，利用编译优化技术最大化目标程序在鲲鹏 920（ARM v8）平台上的并行计算性能。核心目标是在输入矩阵维度 N 和分块大小 B 编译期未知的情况下，通过 LLVM Pass 插桩和运行时动态构图，将串行的分块 Cholesky 分解算法转换为高效的并行执行流，最大化多核利用率。具体要求如下：

- **算子依赖分析**：仅限于在 LLVM 层面增加 Pass 的方式，对给定的应用程序代码（主要是分块 Cholesky 分解）进行算子依赖关系分析。
- **并行调度生成**：生成二进制可执行程序，使得原始程序中的 `cholesky`、`trsm`、`madd` 等核心算子能够高效并行执行。
- **运行时支持**：允许设计并实现并行运行时库（Runtime Library），供编译器插入相关的接口调用，以支持动态图的构建与调度。
- **正确性验证**：输出结果必须通过双精度误差检验（Scaled Residual），否则性能得分无效。

1.1.2 算法原理与算子定义

分块 Cholesky 分解算法

Cholesky 分解将对称正定矩阵 A 分解为 $A = LL^T$。为了适配鲲鹏 920 的 Cache 层级结构并提高并行度，采用分块算法。输入矩阵 A 是一个 $N \times N$ 的正定对称矩阵。分块则是为了优化访存性能和并行度，人为将大矩阵 $N \times N$ 切割成的子矩阵的边长，记作 B。即每个分块的大小为 $B \times B$.

算法伪代码如下：

算法 1 Blocked Cholesky Decomposition

```
for k = 0 to N step B do
    L_kk ← cholesky(A_kk) {对角线块分解}
    for i = k + B to N step B do
        L_ik ← trsm(A_ik, L_kk) {列块更新 (Panel Factorization)}
    end for
    for j = k + B to N step B do
        for i = j to N step B do
            A_ij ← madd(A_ij, L_ik, L_jk) {尾部子矩阵更新 (Trailing Update)}
        end for
    end for
end for
```


核心算子定义与形状关系

构建动态依赖图的基础是准确理解算子的输入输出内存区域。假设当前处理的块基准大小为 $B \times B$（边缘块可能小于 B）：

1. **cholesky**:
   - **公式**: $L_{kk} = \text{Chol}(A_{kk})$。
   - **形状**: 输入/输出均为 $B \times B$。
   - **依赖特征**: 读写同一块内存 A_{kk}，具有强顺序约束。
2. **trsm**:
   - **公式**: $L_{ik} = A_{ik} L_{kk}^{-T}$。

- **形状**: 更新目标 A_{ik} $(h \times B)$，依赖输入 L_{kk} $(B \times B)$。
- **依赖特征**: 必须等待 L_{kk} 的 `cholesky` 完成。

3. **madd**:

- **公式**: $A_{ij} = A_{ij} - L_{ik}L_{jk}^T$。
- **形状**: 更新目标 A_{ij} $(h_1 \times h_2)$。依赖输入 L_{ik} $(h_1 \times B)$ 和 L_{jk} $(h_2 \times B)$。
- **依赖特征**: 是计算量最大的部分，并行度最高，需等待对应的两个 `trsm` 任务完成。

1.1.3 基于 mClang 驱动程序的开发架构

为了支持 Pass 的快速调试与集成，我们设计了一个基于 Clang/LLVM 库的轻量级驱动程序 mClang。它负责协调前端解析、中间代码优化（Pass 执行）以及与运行时库的链接。

mClang 驱动程序设计

`mClang` 是一个编译器前端，且集成了 MCJIT（Machine Code JIT）引擎，允许运行时编译运行，方便开发时测试新开发 Pass 在测例上的优化效果。其核心流程如下：

1. **前端解析 (Frontend Action)**: 利用 `clang::ASTConsumer` 解析 C 源码。虽然赛题主要要求在 IR 层操作，但 AST 上的分析可能方便提取高层语义，从而方便做特定类型的代码转换。

2. **Pass 管道构建 (Pass Pipeline)**: 在生成 LLVM Module 后，驱动程序需显式注册我们的自定义 Pass。这是比赛的关键集成点。

代码 1.1: mClang 中注册自定义 Pass 的逻辑

```
// main.cpp 核心逻辑
void main(int argc, const char **argv) {
    // 1. 初始化，创建 Passmanger
    TheDriver.InitializePasses();

    // 2. 注册基础分析 Pass (如 Alias Analysis)
    TheDriver.addPass(llvm::createBasicAAWrapperPass());

    // 3. 注册我们的核心 Pass: 动态算子图生成器
    // 该 Pass 负责识别 cholesky/trsm/madd 并进行插桩
    TheDriver.addPass(new DynamicOperatorGraphPass());

    // 4. 运行 Pass，分析和转换 LLVM IR
    TheDriver.run();
}

```

3. **JIT 执行环境**: `mClang` 使用 `llvm::EngineBuilder` 构建执行引擎。可以通过 `LoadLibraryPermanently` 方法设置 Engine 加载自行编写的并行运行时库 (Runtime Library), 并将其符号映射到 JIT 空间中，使得插桩后的 IR 代码可以正确调用运行时接口（如 `submit_task`）。

1.1.4 大致实验思路：从串行到动态图并行

由于输入规模不确定，传统的静态指令调度无法处理循环次数未知的依赖关系。一种较为朴素的解决办法是修改成运行时动态构图 + 基于动态图的运行时调度：

编译时（构建对应的 LLVM Pass）

代码转换：将命令式执行逻辑改成运行时动态构图 在 LLVM IR 层编写 Pass，识别主循环结构（三层循环：i, j, k of 1）中的关键算子调用。

- **依赖识别**：分析 `cholesky`, `trsm`, `madd` 之间的数据依赖。
- **代码变换**：将原始的三个核心算子的函数定义进行修改（如有必要），将核心算子的调用语句替换为通过运行时库的调度器 API 提交计算任务，同时提交依赖的输入数据用于调度器构建依赖图。
- **示例变换**：

```
// 原始代码
madd(&L[...], ...);

// 变换后代码（伪代码）
runtime_submit_task(madd, dependence_list={...}, data_ptr=&L[...]);
```

代码优化：静态分析依赖关系，执行算子融合等优化 简单地提交任务由运行时调度通常难以达到预期的高性能。可以考虑开展算子融合等优化。

举例：分析算子依赖图，发现 `trsm` 和 `madd` 函数存在数据局部性，`trsm` 的输出是 `madd` 的输入。那么，可以考虑将 `trsm` 和 `madd` 融合，合成一个函数。这个函数内部可以先为 `madd` 计算依赖的两个分块 L_{ik} 和 L_{jk}，再计算 `madd`. 这样做虽然可能导致重计算，但是在一些访存开销是瓶颈的场景下潜在也有收益。大致如下：

```
// 原始代码
trsm(&L{...}, ...);
```

```
for i in (j, N, B):
    madd(&L[...], ...);

// 变换后代码（伪代码）
for i in (j, N, B):
    runtime_submit_task(trsm_madd, dependence_list={...}, data_ptr=&L[...]);
```

上面的优化只是一种示例，同学们可以根据实际评测的性能情况，针对性优化程序的瓶颈部分。

运行时调度器（Scheduler）

实现一个轻量级的运行时系统来管理任务执行：

- **动态构图**：当主线程执行到插桩代码时，不立即执行计算，而是创建一个“任务节点”并加入任务图。
- **依赖追踪**：根据传入的地址或块索引（Block ID）记录节点间的依赖关系（如使用引用计数或拓扑前驱列表）。
- **并行发射**：维护一个线程池。当某个任务节点的前驱依赖全部满足时，将其分发给空闲线程执行。

1.2 基于进化算法的 Triton 算子自动优化

1.2.1 实验目标

本实验旨在设计并实现一套基于进化算法（Evolutionary Algorithms）的自动代码优化系统，针对 **Triton-CPU 算子**进行性能调优。学生将在给定的 Agent 框架中，利用进化算法自动搜索并生成优化后的 Triton 代码，在确保功能正确的前提下，最大化算子的执行效率。

1.2.2 实验内容

进化算法框架

本实验需根据参考文献自主选择和搭建进化算法框架，本节仅以 **EvoPrompt** 为例进行介绍，其伪代码如算法2 所示，核心流程如下：

算法 2 EvoPrompt: 基于 LLM 的进化提示优化框架

Require: 初始提示集合 $P_0 = \{p_1, p_2, \ldots, p_N\}$，种群大小 N，评估数据集 D，评估函数 $f_D(\cdot)$，迭代次数 T，进化操作函数 Evo$(\cdot)$

初始评估得分：$S_0 \leftarrow \{s_i = f_D(p_i) \mid i \in [1, N]\}$
for $t = 1$ **to** T **do**
  选择：从当前种群 P_{t-1} 中选择若干提示作为父代 $p_{r_1}, \ldots, p_{r_k}$
  进化：基于父代提示，利用 LLM 执行进化操作生成新提示 $p'_i \leftarrow \text{Evo}(p_{r_1}, \ldots, p_{r_k})$
  评估：计算新提示得分 $s'_i \leftarrow f_D(p'_i, D)$
  更新：根据得分更新种群 $P_t \leftarrow \{P_{t-1}, p'_i\}$ 与得分集合 $S_t \leftarrow \{S_{t-1}, s'_i\}$
end for
返回最终种群中得分最高的提示 $p^* \leftarrow \arg\max_{p \in P_T} f_D(p, D)$

进化操作函数 Evo$(\cdot)$ 主要包括交叉（Crossover）与突变（Mutation）两种操作，均通过调用大语言模型实现语义级别的代码变换。

代码优化对象

- 优化目标为 **Triton-CPU** 代码库中的算子。
- Triton 是一种面向 GPU/CPU 的高性能内核编程语言，语法接近 Python。
- 学生需通过 Agent 自动对 Triton 代码进行变换与优化，**禁止直接手动修改源码**。

- 为便于开展实验，课程助教已预先配置 **Triton-CPU 实验环境**，学生可参照以下教程完成环境配置与验证：实验环境配置教程链接

支持调用的大语言模型

实验允许调用以下大语言模型进行代码生成与优化：

- DeepSeek-V3、DeepSeek-V3.1、DeepSeek-R1
- Qwen3 系列（含 Qwen3 Coder）
- Kimi-K2-Instruct/Thinking
- GLM-4.5

实验输出要求

- Agent 应返回至多 **5 个**优化后的算子版本。
- 其中至少一个版本必须能够**成功编译并通过所有功能测试**。
- 若所有版本均未通过测试，或所有版本与原代码完全相同，则视为未完成实验目标。

1.2.3 实验评价标准

功能正确性

- 至少一个优化后的算子须通过全部功能测试。
- 此为实验通过的基本前提。

性能加速比

性能提升通过 **加速比（Speedup Ratio）**衡量，计算公式如下：

$$\text{Speedup Ratio} = \max\left(\frac{T_{\text{baseline}}}{T_{\text{current}}} - 1,\ 0\right)$$

其中 T_{baseline} 为原代码运行时间，T_{current} 为优化后代码运行时间。加速比越高，代表性能优化效果越显著。

资源使用限制

- 每个算子的优化过程最多运行 **20 分钟**或消耗 **20 万 token**（任一条件触发即停止）。
- 若采用多 Agent 协作，总 token 消耗量为所有 Agent 之和。

1.2.4 参考文献

以下文献为实验提供算法基础与扩展思路，鼓励调研其他相关工作以深入理解进化算法与代码生成的结合方法。

1. Guo, Q., *et al.* "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers." *arXiv preprint arXiv:2309.08532* (2023).
2. Novikov, A., *et al.* "AlphaEvolve: A coding agent for scientific and algorithmic discovery." *arXiv preprint arXiv:2506.13131* (2025).
3. Lange, R. T., Imajuku, Y., & Cetin, E. "Shinkaevolve: Towards open-ended and sample-efficient program evolution." *arXiv preprint arXiv:2509.19349* (2025).
4. Lange, R. T., *et al.* "Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization." *arXiv preprint arXiv:2509.14279* (2025).

1.3 基于 AscendNPU IR 的自动算子融合代码

1.3.1 实验目标

本实验基于 Ascend NPU 编译体系，围绕 AscendNPU IR 展开，目标是设计并实现一种**NPU 自动融合代码生成与调优的编译 Pass**，以提升融合算子在 ascend NPU 平台上的执行性能。赛题地址。

实验的总体目标如下：

- 基于 AscendNPU IR，实现算子自动融合与代码生成；
- 通过编译 Pass 的方式对融合算子进行性能调优；
- 生成可在 ascend NPU 上高效执行的二进制文件。

实验要求采用 AscendNPU IR 中的融合 Pass 进行自动代码生成与优化，具体包括：

- 构建面向 ascend A2/A3 平台的自动融合代码生成与调优 Pass；
- 在 IR 层面对融合算子进行分析、划分与优化；
- 可结合规则驱动、Auto-tuning 或 Auto-learning 等策略提升性能。

输入： 融合算子的原始计算逻辑 IR，包含算子计算描述、调度信息以及动态 shape 信息。

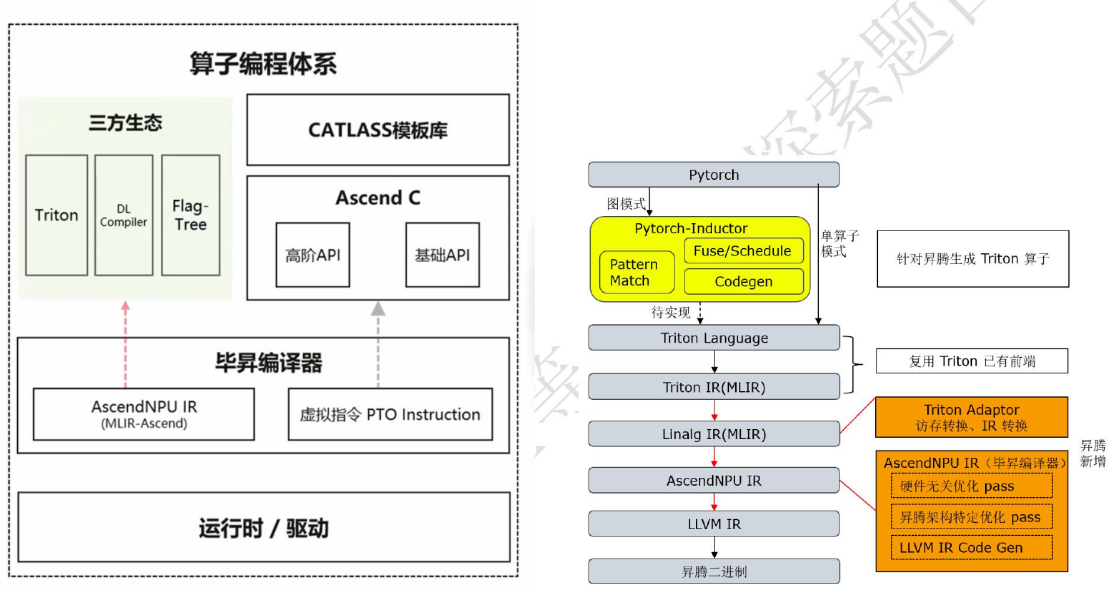
输出： 1. 融合并优化后的 AscendNPU IR；2. 最终可在 ascend NPU 上执行的高性能二进制文件。

实验需完成以下具体内容：

- 在 AscendNPU IR 层面实现算子融合 pass；
- 支持常见算子组合及动态 shape 场景；
- 适配 ascend 异构大模型平台提供的模型；
- 将所设计的融合与优化 Pass 集成至 AscendNPU IR 的 Pass Pipeline；
- 在比赛主办方提供的完整的测试用例上验证功能正确性和性能提升效果。

1.3.2 介绍 AscendNPU IR

- 如 1.1 所示，支持单算子模式编译
  - 复用 Triton 已有前端，将 Python AST 转为 Triton IR
  - Triton Adaptor 将离散访存转为连续线性访存，并将 Triton IR 转成 Linalg IR
  - BiSheng 编译器通过各类 Pass 将 Linalg IR 转成优化后的 AscendNPU IR ，并生成 LLVM IR 代码，最终编译得到 kernel.o
- Pytorch 默认的通过 inductor 进行的图编译流程暂时没有支持



(a): Ascend 编译整体流程　　(b): AscendNPU IR 在 Triton 中的位置

图 1.1: Ascend 编译流程与 AscendNPU IR 位置示意

1.3.3 HFusion IR 算子运行示例

运行环境

学校提供的 910B 集群平台：选择此题目的同学请联系助教开通平台账号。

- 镜像 triton-ascend:latest 。镜像的环境仅能运行下面列出的命令，如果需要增加新功能、修改或者增加 HFusion pass，那么可能需要自己安装所需依赖，并保存镜像从而持久化

- 普通开发建议选择 NPU: 4 核 8G 内存单卡 910B 套餐，包含单卡 910B3、Kunpeng-920 虚拟化出的 4 核 CPU、8GB 内存。注意，如果需要源码编译 Triton，可能需要选择包含更多的 CPU 核心的 NPU 套餐从而加速编译。

RMSNorm 算子的编译与性能评测

使用下面的命令编译运行 `rms_norm` 算子并评测性能。

编译 `rms_norm` 算子的测试程序

```
mkdir build
cd build

cmake -G Ninja .. \
-DCMAKE_C_COMPILER=clang \
-DCMAKE_CXX_COMPILER=clang++ \
-DCMAKE_BUILD_TYPE=Release

ninja -j 6  # 6个线程并行编译
```

编译 `rms_norm` 算子

```
bishengir-compile -enable-hfusion-compile=true -block-dim=40 \
../compile-rms-norm-fp16-split-n.mlir -o kernel.o
```

运行算子

```
./test_rms_norm
```

性能评测 使用 msprof 测量性能，从而获取详细的性能数据，比如 vector 和 cube 占用率、icache miss 率。为了等待 AI Core 频率拉起从而测量出精准的耗时，可以尝试运行 30 次 kernel 取后 15 次平均，`msprof` 会自动统计到 30 次 kernel 调用。

```
msprof --application="./test_rms_norm" --output=.
```

性能统计结果在 `op_summary_*.csv` 中，总结如下：

Op Name	Task Type	Task Duration (us)	aiv_total_cycles	aiv_vec_ratio	aiv_scalar_ratio	aiv_mte2_ratio	aiv_mte3_ratio	aiv_icache_miss_rate
rms_norm_f16	AI_VECTOR_CORE	22.321	1326192	0.249	0.295	0.307	0.332	0.014

表 1.1: RMSNorm 算子性能统计结果

1.3.4 修改或增加 HFusion Pass

根据 赛题要求，需要针对比赛主办方给出的融合算子 IR（推测是 HFusion MLIR），对于已有 HFusion Pass 进行优化，并编译得到新的 `bishengir-compile`，从而正确生成性能更优的 `kernel.o`。

可以尝试先以 `rms_norm` 算子为例，对不同输入形状 $[B, H]$，制定不同的切分调度策略。还可以尝试将

$$y = \mathrm{RMSNorm}(x), \quad [q, k, v] = y[W_q, W_k, W_v]$$

这两个算子进行算子融合。

- 参考 文档 从源码编译得到 `bishengir-compile`
- 参考 HFusion 的已有 编译 pipeline ，修改或增加 HFusion Pass
- 修改或增加 HFusion Pass 后，编译得到 `bishengir-compile`，参考上一节内容，运行并评测算子的正确性和性能

1.4 Triton-Ascend 高性能算子实现

- 背景赛题信息：基于 Triton-Ascend 构建 Ascend 亲和算子
- 实验的公共仓库：Triton 矩阵乘法（Matmul）NPU 性能优化案例
- 相关教程：CANN 社区-矩阵乘法、triton-ascend

1.4.1 实验概览

本实验旨在利用 Triton 编程框架在 Ascend NPU 上实现高性能矩阵乘法。核心任务是利用**分块（Tiling）技术**解决片上存储限制，结合 **Autotune 机制**自动搜索最优超参数，并利用 **vector 核并行**掩盖写回延迟。实验评价以 **V1 (Baseline)** 版本耗时为基准，通过**加速比**（$SpeedupRatio = T_{V1}/T_{current}$）来衡量对硬件性能的压榨程度，加速比越高代表优化方案越优。

1.4.2 优化原理与关键机制

算子分块技术（Tiling）

本小节简单介绍 Ascend 上矩阵乘法的分块技术，详细说明可以参考CANN 社区-矩阵乘法。

矩阵乘法（Matmul）的基本计算公式为 $C = A \times B$。由于 NPU 的 Local Memory（片上缓存）存储空间有限，无法完整容纳大型矩阵，必须将全局内存（GM）中的数据切割为子矩阵（Tiles）进行处理。

存储层级与数据分流

- **操作数定义**：A 为左矩阵 $(M \times K)$，B 为右矩阵 $(K \times N)$，C 为结果矩阵 $(M \times N)$。
- **存储层级架构**：数据流向通常遵循 $GM \to A1/B1 \to A2/B2$。A1/B1（二级缓存）存放整块矩阵，A2/B2（一级缓存）存放切分后的小块矩阵，通过分级缓存减少计算前的等待时间。

Tiling 切分策略（参考 图 1.2）

- **多核并行切分**：为了实现多核并行，需要将矩阵分配到不同核上处理。如 图 1.2 所示，将 A 沿 M 轴切分为 $singleCoreM$，B 沿 N 轴切分为 $singleCoreN$。例如 8 个核参与计算时，A 沿 M 轴划分为 4 块，B 沿 N 轴切分 2 块，每个核仅处理特定的分块。
- **核内切分与迭代**：在单核内部进一步将 $singleCoreM$ 与 $singleCoreN$ 切分为基本块 $base$，$baseN$，$baseK$。结果矩阵中的一个基本块是通过 A 沿 K 轴切分的多个分片与 B 对应分片进行多次相乘并累加得到的。

- **迭代顺序**：计算完一个 $[baseM, baseN]$ 分片后，Matmul 自动偏移到下一个位置的顺序。

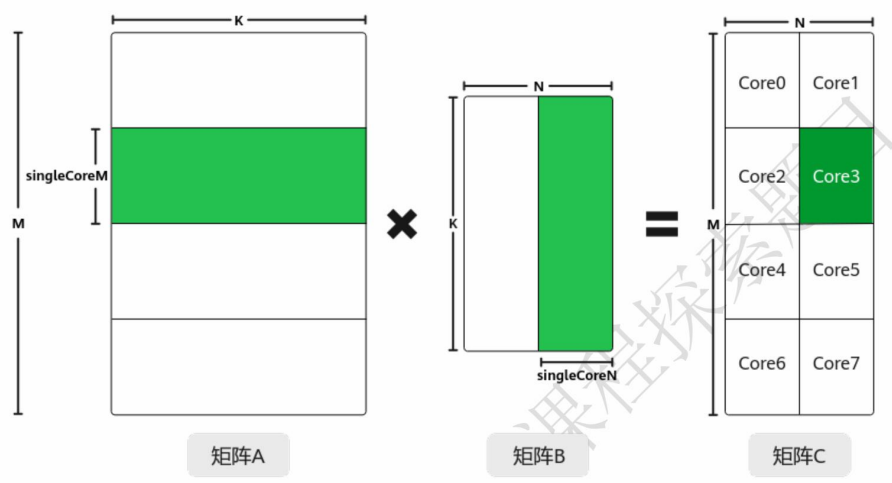


图 1.2: Matmul 矩阵乘法多核 Tiling 切分策略示意图

Triton 编程框架与 autotune

Triton 是一种面向高性能内核的编程语言，旨在简化硬件算子的开发与调优流程。其核心优势在于自动化处理复杂的硬件细节，特别是通过 Autotune（代码 1.2）机制寻找最优的分块超参数。详细文档可以看triton-ascend。

Triton 自动调优 (autotune) 矩阵乘法的性能高度依赖于分块超参数($BLOCK\_SIZE\_M$ 等)。Triton 通过装饰器提供了一套动态搜索机制：

代码 1.2: Triton Autotune 配置示例

```python
# 定义候选配置空间
AUTOTUNE_CONFIGS_FULL = [
    triton.Config({"BLOCK_SIZE_M": 128, "BLOCK_SIZE_N": 128, "BLOCK_SIZE_K": 32}),
    triton.Config({"BLOCK_SIZE_M": 64, "BLOCK_SIZE_N": 256, "BLOCK_SIZE_K": 64}),
    # ... 更多配置
]

# 自动调优装饰器：根据矩阵形状 (M, N, K) 自动搜索并缓存最优配置
@triton.autotune(
    configs=AUTOTUNE_CONFIGS_FULL,
    key=["M", "N", "K"],
)
@triton.jit
def kernel(..., BLOCK_SIZE_M: tl.constexpr,
           BLOCK_SIZE_N: tl.constexpr,
           BLOCK_SIZE_K: tl.constexpr):
    # Kernel 实现逻辑，参数将由 autotune 自动注入
    pass
```

- **配置搜索**：`@triton.autotune` 会在程序首次执行时遍历候选列表，通过测量实际执行耗时选出该硬件环境下针对当前输入形状的最优参数。
- **vector 核并行**：利用 `tl.parallel` 指令显式调用多个 vector 核并行执行结果写回操作，可以有效掩盖访存延迟，提高整体计算效率。

1.4.3 运行环境

学校提供的 910B 集群平台：选择此题目的同学请联系助教开通平台账号。

- 镜像 triton-ascend:latest。注意：如果需要修改根目录下的环境，那么需要保存镜像
- 普通开发建议选择 NPU: 4 核 8G 内存单卡 910B 套餐，包含单卡 910B3、Kunpeng-920 虚拟化出的 4 核 CPU、8GB 内存。注意，如果需要源码编译 Triton，可能需要选择包含更多的 CPU 核心的 NPU 套餐从而加速编译。

1.4.4 实验评价标准

本实验以实验仓库中 V1 (Baseline) 版本的运行耗时为基准，通过加速比（Speedup Ratio）来衡量优化效果：

$$SpeedupRatio = \frac{T_{V1\_Baseline}}{T_{current}} \tag{1.1}$$

其中 $T_{V1\_Baseline}$ 为基准版本耗时，$T_{current}$ 为当前优化版本的耗时。加速比越大，说明优化方案对硬件性能的压榨越充分。

除了上述 Autotune 与并行写回外，鼓励尝试针对实际模型（如 Transformer 结构）中的特定长瘦矩阵进行优化，或探索其他优化方法以进一步突破性能瓶颈。