ResNet 推理优化与 GPGPU 模拟器

2022 年 12 月 31 日

目录

1 概述

本次作业是 Tensor Core 的原理与实现，实验内容包括两个部分，其一是基于 NVIDIA Volta 架构的 GPGPU（Tesla V100）中 Cuda Tensor Core 的硬件结构，使用 Cuda 与 C++ 语言编写算子、搭建网络模型并进行推理，完成 ResNet18 神经网络的 GPU 部署；其二是学习 Tensor Core 的架构原理尤其是 HMMA 指令实现矩阵乘加运算的方法，并利用伪指令函数实现矩阵乘法和 ResNet18 网络的推理。

在本次作业中，我们完成了所有的实验要求。具体来说，我们用 Cuda 语言和 C++ 完成了 ResNet18 在 GPU 端的部署，通过 Cuda 并行计算实现了推理加速，也借助论文和反编译工具理解了 WMMA Kernel 的计算原理，并在此基础上实现了通用矩阵乘法 GEMM 和 ResNet18 的前向传播。

1.1 文件结构

```
ResNet_1
├── bin
├── data
│   ├── input_data（请从 general_data 中复制）
│   ├── model_data（请从 general_data 中复制）
│   ├── result
│   └── time
├── include
│   └── cuda_kernel.cuh
├── src
│   └── cuda_kernel.cu
├── main.cpp
└── Makefile
ResNet_2
├── data
│   ├── input_data（请从 general_data 中复制）
│   ├── model_data（请从 general_data 中复制）
│   ├── result
│   └── time
├── resnet_test.cpp
├── tensor_core.h
└── Makefile
data_process（数据处理）
```

```
├── forward.py（计算中间层准确输出）
├── process.py（读取模型参数）
├── img_process.py（图片处理）
└── sass 指令文件分析.md
general_data（请在 https://jbox.sjtu.edu.cn/l/S1IbkL 中下载）
├── data_1（实验一结果）
│   ├── result
│   └── time
├── data_2（实验二结果）
│   ├── result
│   └── time
├── input_data（一万个 3 × 224 × 224 图片张量）
├── model_data（onnx 模型参数）
└── result_b（baseline 计算结果）
```

2 数据预处理

2.1 模型参数

本次实验中，ResNet18 神经网络以 onnx 格式存储在服务器端，由于没有 sudo 权限，所以需要首先在本地对模型进行解析，以一种中间格式存储模型参数，将其上传至服务器端，再由程序读取到内存中。

将卷积层权重、卷积层偏置、全连接层权重和全连接层偏置四种数据处理成 42 个 txt 文件，文件中按行保存浮点数。

模型参数包括四维（卷积层权重）、二维（全连接层权重）和一维（卷积层偏置和全连接层偏置）三种维度，将四维参数转化成二维，从而将这些参数都以矩阵的形式写入 txt 文件中。将这些 txt 文件上传到服务器端，从而在程序运行时读取模型参数。

2.2 图片数据

ResNet18 模型支持的输入是 3×224×224 ，选取 ImageNet 中验证集的若干图片，通过 torchvision.transforms 将图片裁剪成符合输入格式要求的张量，将数据转化成二维矩阵，写入 txt 文件中。在写入数据时，调用 numpy.savetxt 方法，这里保存了小数点后 8 位。

由于程序在读取文件数据时文件打开和关闭操作耗时较长，所以为提高程序运行效率，也可以将多张图片存放在一个 txt 文件中，从而减少文件打开和关闭的次数。

2.3 神经网络结构

在 https://netron.app 网站上可以对 onnx 模型进行可视化，并能查看全部参数，包括 stride, pad 以及输入输出维度等。

本次实验所选用的神经网络包括如下算子：

1. 卷积运算 Conv2d
2. 激活运算 ReLU
3. 张量相加 Add
4. 最大池化 MaxPool
5. 全局平均池化 GlobalAveragePool
6. 线性运算 Linear

3 基于 Cuda 的 ResNet 推理优化

3.1 文件读取加速

在 ResNet18 模型的部署中，首先需要读取模型参数和图片数据文件。这一部分可以通过 OpenMP 中的 parallel for 实施并行加速。服务器的 CPU 是 32 核 64 线程，实验中我选用了 60 线程实施加速。

3.2 算子实现

在模型的推理过程中，卷积运算是占比最多、运算量最大也最耗时的操作，所以卷积算子实现的速度决定了模型推理的整体速度。其余的算子也可以通过 Cuda 多线程加速来提高计算速度。

3.2.1 Conv

在当今大多数开源 AI 框架中（Caffe，PyTorch 等），卷积运算都是分 im2col 和 GEMM 两步实现的。其中 im2col 是将输入的特征图根据卷积运算涉及的区域（由 stride 和 kernel 决定）整理为矩阵的形式（每一行表示一个被卷积区域的参数，行数表示被卷积区域的数目），记为矩阵 A。此外还需将卷积权重和偏置整理为矩阵形式，记为矩阵 B 和 C。将这三个矩阵借助通用矩阵乘法 GEMM 执行乘加运算 D=A×B+C，得到的 D 即为输出特征图。再将其整理为（channel，a，b）的形式（记为 col2im），即完成了一次卷积运算。

GEMM 的实现利用了 Tensor Core 的结构，其运算数矩阵 A B C 理论上是形状一致的（Volta 架构的 Tesla V100 GPU 支持的通用矩阵乘法 m n k 是 4096），但根据 NVIDIA 官方给出的 GEMM 实现方式可知，k 的取值只需是 16 的倍数即可，m 和 n 的取值应为两个相等的数，且需是 64 的倍数。这是因为一个 Block 包括 4×4 个 Warp，每个 Warp 负责执行 16×16×16 的混合精度矩阵乘加运算。

GEMM 的实现是基于 WMMA Kernel。在 GEMM 的计算过程中，线程划分是依据矩阵 C 和 D 的形状（即 m 和 n），一个线程束 Warp （32 个线程）负责计算矩阵 D 中的一个 16×16 的区域。线程设置方式为

```
dim3 blockDim, gridDim;
blockDim.x = 4 * 32;
blockDim.y = 4;
gridDim.x = m / 64;
gridDim.y = n / 64;
```

```

simple_wmma_gemm <<<gridDim, blockDim>>> (...);
```

由于线程之间存在协作关系，所以在 GPU 函数中需要将这种关系考虑进去。具体而言，调用 WMMA Kernel 是按照 warpM 和 warpN 实施的，每个（warpM，warpN）负责沿 k 维度（A 的第二个维度和 B 的第一个维度）对矩阵 A 和 B 中相应的 16×16 部分实施乘法并累加，再加上矩阵 C 中对应的部分，从而得到结果。warpM 和 warpN 的计算方式如下：

```
#define warpSize 32

// Tile using a 2D grid  (0~64) * 128 + (0~128) = 8192 / 32 = 256
int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
// (0~64) * 4 + (0 ~4) = 256
// 256 * 16 = 4096
```

卷积运算所涉及的矩阵操作包括如下几种：

1. im2col
2. 矩阵分片和用 0 填充
3. GEMM
4. col2im

这些操作都可以通过 Cuda 多线程来实现。相比于仅仅在 GEMM 处使用 Cuda 加速，新的实现方式将每张图片的平均推理耗时从 2.5s 降到了 0.4s

3.2.2 其他算子

其他的算子包括 ReLU、Add、MaxPool、GlobalAveragePool 和 Linear。这些都可以按照通道划分多线程从而实现加速。Linear 的实现也可以采用 GEMM。

4 GPGPU 模拟器

本实验的第二个任务是学习 Tensor Core 架构原理，利用伪指令函数实现矩阵乘法。具体包括如下内容。

4.1 HMMA 指令分析

通用矩阵乘法是基于 WMMA Kernel 实现的，它负责计算一次 16×16×16 的矩阵乘加运算。WMMA Kernel 源代码的反编译结果 app.sass 由几十条 SASS 指令组成，其中的核心部分是 16 个 HMMA 指令，如图1所示。

论文[1]中详细分析了 HMMA 指令的计算原理。具体来说，16 条指令由 4 个 SET 组成，每个 SET 包括 4 个 STEP，每个 STEP 即一条 HMMA 指令。这 16 条指令定义了一个 Warp 中 32 个线程的操作方式。

[1]Modeling Deep Learning Accelerator Enabled GPUs

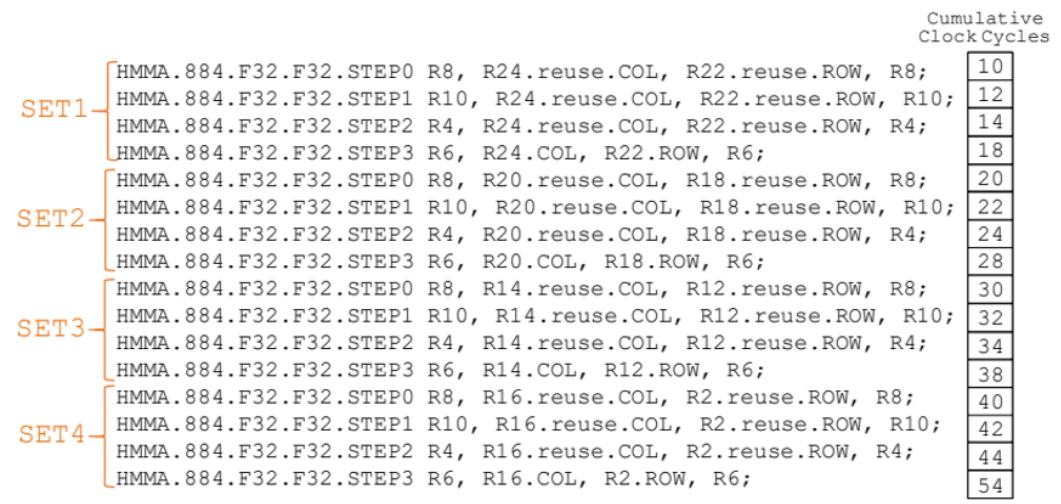


图 1: 16 条 HMMA 指令

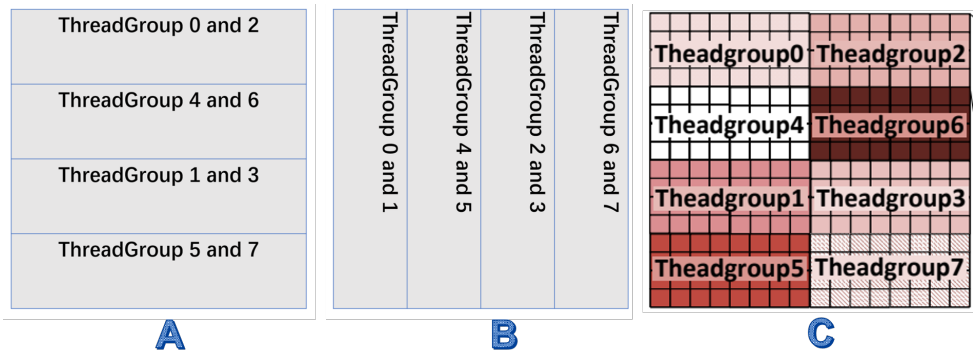


图 2: 操作数矩阵 A B C 在线程组间的分配方式

一个 Warp 的 32 个线程首先被划分成 8 个线程组 ThreadGroup，每个线程组由 4 个线程组成。三个操作数矩阵 A B C 存储在 global memory 中，由 32 个线程将其加载到寄存器中，再实施 HMMA 指令。具体的分配方式如图2所示。

图8展示了线程组中线程的分配方式[2]，由于 A B 矩阵中的元素是半精度浮点型 half，而 C D 矩阵中的元素是整精度浮点型 float，所以一个线程在一个寄存器中的 32 位一次能加载 A 或 B 矩阵中的两个元素，或者 C 矩阵中的一个元素。HMMA 指令的参数是四个寄存器，每个参数实际上表示相邻的两个寄存器，比如图1中第一行，R8 其实表示的是 <R8, R9>[3]。由于矩阵 A 和 B 中的每个元素都被加载了两次，这样 HMMA 指令的一个参数就表示了 A 和 B 在一个 SET 下的全部数据，以及 C 和 D 在一个 STEP 下所涉及的全部数据。

操作数矩阵的元素与寄存器的对应关系如图4、图5和图6所示。其中图4和图5中的小矩形表示 4 个半精度浮点数（共 64 位），上面的数字 $x-y$ 表示线程组编号-线程在组内的编号，其中 $x \in \{0, 1, ..., 7\}$，$y \in \{0, 1, 2, 3\}$。

矩阵 D 与矩阵 C 的线程分配方式是一致的。在这种分配方式下，线程组不能单独工作，需要两个线程组相配合，由 8 个线程组成的 Octet 是能够独立工作的最小单位，如图3所示。

[2]本图取自论文 Modeling Deep Learning Accelerator Enabled GPUs，但红圈部分所示的两个格子应该调换位置，此处应该是论文的笔误

[3]这里与论文 Modeling Deep Learning Accelerator Enabled GPUs 中所述不一致，应该是论文的笔误

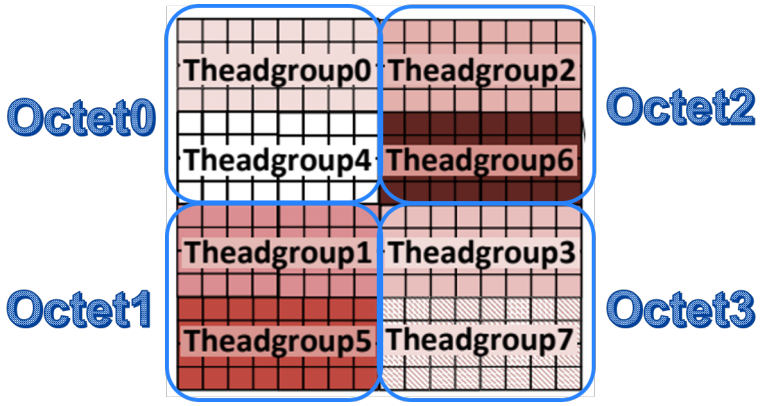


图 3: Octet 示意图

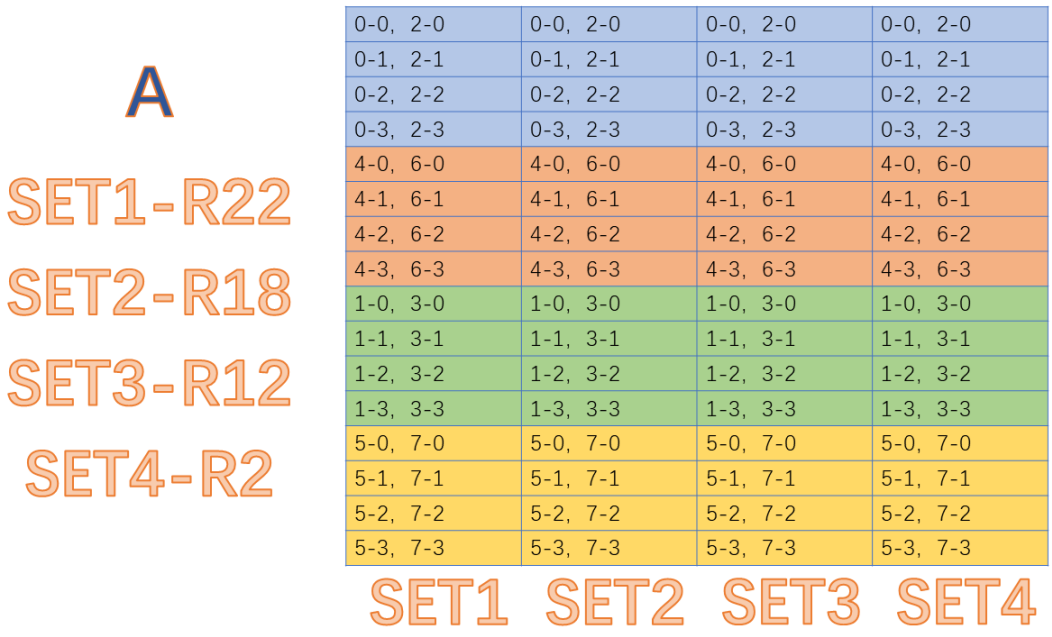


图 4: A 矩阵元素与线程和寄存器的对应关系

具体的计算过程如图7所示。4 个 SET 的作用是在 A 和 B 矩阵中遍历维度 k，每个 SET 完成一次 4×8×4 的混合精度乘加运算，经过 4 次计算后，就完成了 4×8×16 的运算（相当于是将维度 k 进行了拆分）。在一个 SET 中，4 个 STEP 分别计算出一个 2×4 的区域，从而完成对 4×8 区域的计算。至于为什么单个线程组不能独立工作，请参考图2中所示线程组加载矩阵数据的方式。以线程组 0 和 SET1 为例，前两个 STEP 涉及的 A 和 B 的数据都是由线程组 0 加载的，但是在后两个 STEP 中，A 矩阵的数据仍由线程组 0 加载，B 矩阵的数据却是由线程组 4 加载。所以，是两个线程组协作完成了 Octet0 的计算。

4.2 基于 SASS 伪指令实现 WMMA Kernel

仅仅通过论文和指令文件很难充分地理解清楚 WMMA Kernel 在指令级别上的具体工作原理。我通过 NVBit 工具中的 record_reg_vals 功能，截获了 WMMA Kernel 运行过程中各个寄存器的值，在此基础上理清了其工作原理。如图9和10所示。

在 app.sass 文件中，除去 16 条 sass 指令之外，其余的指令主要起到了如下的作用：

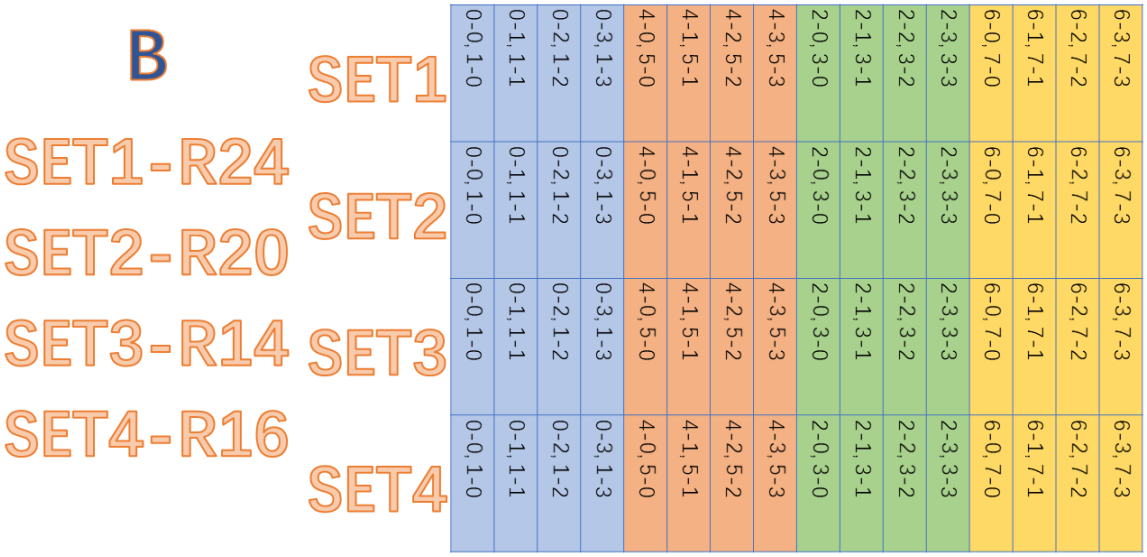


图 5: B 矩阵元素与线程和寄存器的对应关系

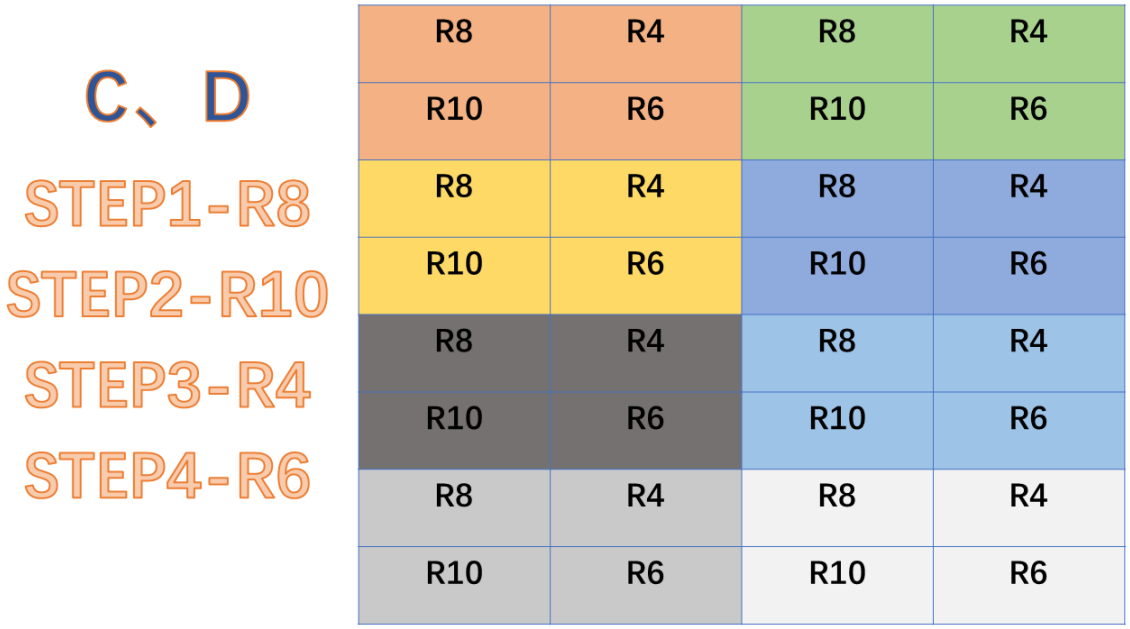


图 6: C D 矩阵元素与线程和寄存器的对应关系

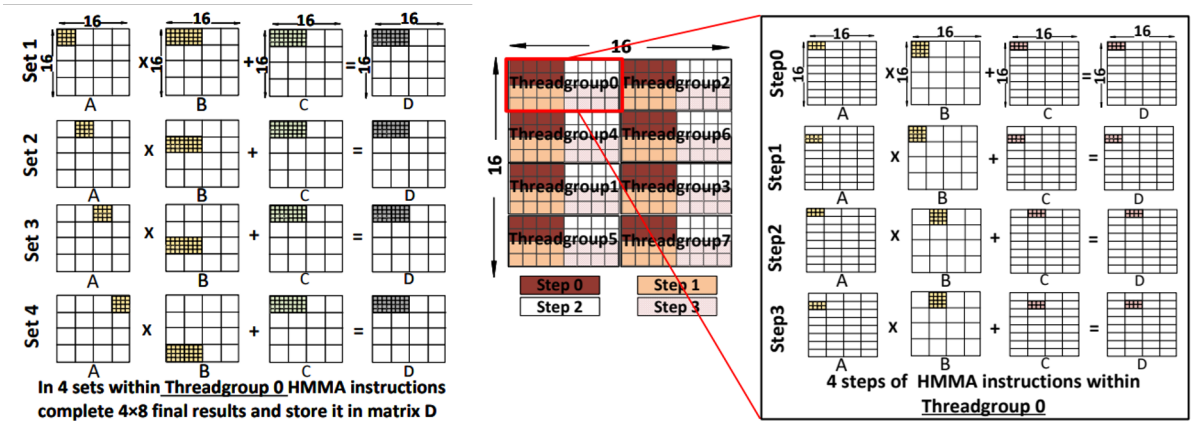
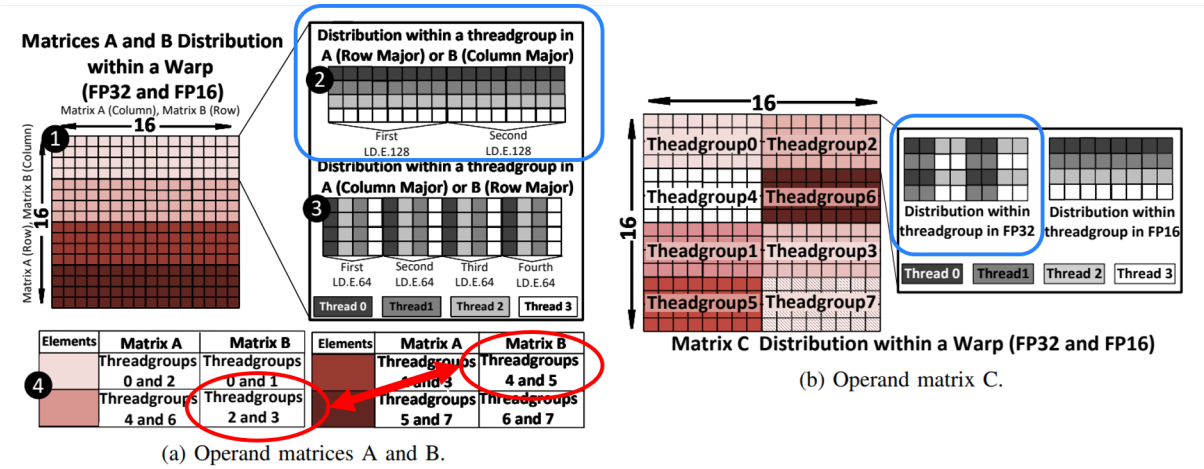


图 7: HMMA 指令计算过程



(a) Operand matrices A and B.

(b) Operand matrix C.

图 8: 线程组（ThreadGroup）中线程的分配方式

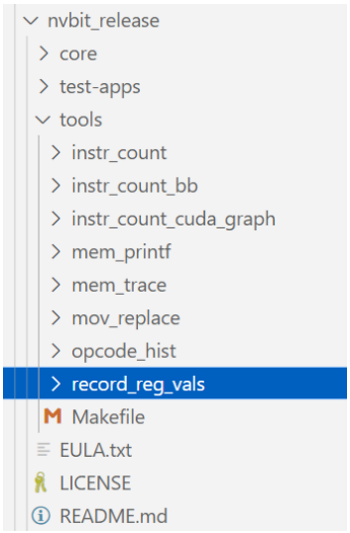


图 9: NVBit 工具

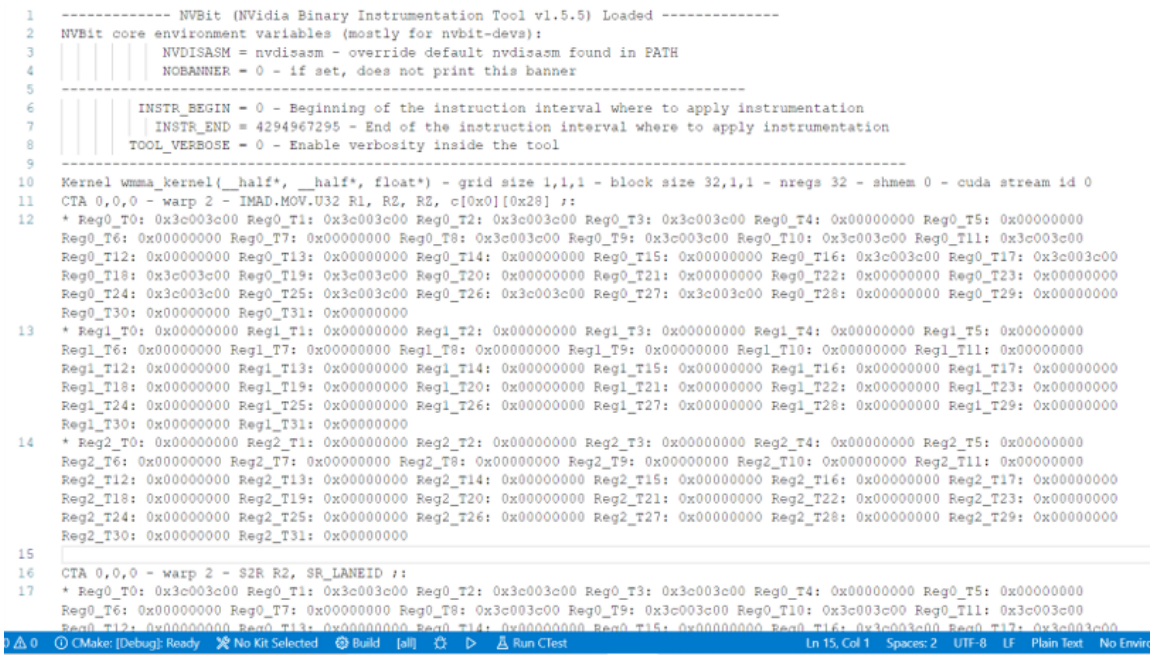


图 10: 程序运行过程中的寄存器值

1. 为寄存器在每个线程中对应的 32 位分配偏置值，从而使得传入函数的指针参数（即 global memory 中矩阵的起始地址）在加上偏置值后可以加载到线程所对应的值。
2. 将地址（函数的指针参数）加载到寄存器中。
3. 根据寻址结果将 global memory 中的值修改为寄存器值。

在分析完 SASS 指令的执行逻辑，并用 C++ 完成了伪指令的实现后，即可对伪指令进行调用从而实现 WMMA Kernel。

4.3 基于 GPGPU 模拟器重写 ResNet 推理

这一部分主要包括两个工作，一是将 Cuda 加速的部分都替换成 C++ 函数，以避免 GPGPU 模拟器和 nvcuda 命名空间发生关键字冲突；二是基于 GPGPU 模拟器重写 GEMM，同样也要把 Cuda 加速的部分替换掉。具体而言，替换掉 Cuda 加速即使用 for 循环替换掉 GPU 函数的调用。为加快推理，也可以使用 OpenMP 中的 parallel for 实施并行加速。

5 计算结果评估

完成了 ResNet18 神经网络模型在真实 GPU 和 GPU 模拟器上的部署之后，模型推理所得的 1000 维向量理论上应与 baseline 之间有较小的相对误差。

5.1 baseline

我在本地使用 Python+onnxruntime 对处理好的 3×224×224 张量计算前向传播，以此作为 baseline。任务一和任务二的推理结果都与之进行对比，并计算相对误差。这里相对误

差 $F_n(X, X')$ 的计算公式为

$$F_n(X, X') = \frac{\sum_{x \in X, x' \in X'} (\sum_{i=1}^{N} (x_i - x_i')^n)^{\frac{1}{n}}}{\sum_{x \in X} (\sum_{i=1}^{N} x_i^n)^{\frac{1}{n}}} \times 100\%$$

其中 n 表示以向量的 $l_n - norm$ 作为评价指标。

5.2 任务一

以任务一 GPU 端部署的 ResNet18 模型推理数据集的前 100 张图片，计算其输出向量的平均 l_2 范数和相对误差。结果如下：

baseline	推理结果	绝对误差	相对误差
89.4212	89.4169	2.12×10^{-5}	$2.37 \times 10^{-5}\%$

5.3 任务二

以任务二 GPU 模拟器上部署的 ResNet18 模型推理数据集的前 10 张图片，计算其输出向量的平均 l_2 范数和相对误差。结果如下：

baseline	推理结果	绝对误差	相对误差
82.7028	82.1664	0.1584	0.1915%

6 项目分工

7 总结与反思

这次大作业的工作量比较大，一方面需要结合论文和 NVIDIA 的官方工具理解清楚 SASS 指令文件尤其是 HMMA 指令的工作原理，另一方面需要编写代码完成 ResNet18 模型的部署，并通过 Cuda 实现推理优化。在写代码的过程中需要用到很多调试技巧，比如在部署模型时，可以通过 Python+onnxruntime 计算模型准确的中间层输出，以此作为基准进行调试；在编写矩阵运算的代码时，可以通过 C++ 将输出流重定向到文件中，从而得到程序运行的计算结果。在理解机器指令的过程中，也需要通过 Google 查找很多的相关资料，比如 StackOverFlow 的帖子，NVIDIA 的官方文档和官方论坛。

这次大作业也充分锻炼了我们的技术和能力，包括团队协作、文献阅读、工具使用和代码编写的能力。最后还要感谢梁老师生动幽默的课堂讲解和助教学长学姐们认真细致的作业指导！