Tensor Core的使用与原理实现 ⭐⭐⭐

一、实验内容

1. 利用CUDA TENSOR CORE硬件结构，使用cuda与c++语言编写算子、搭建网络模型以及进行推理，完成ResNet18神经网络的GPU部署。
2. 学习TENSOR CORE架构原理，利用伪指令函数实现矩阵乘法。
3. 分组：有 A、B、C、D、E，5组标准一样

二、背景介绍

Tensor Core

Tensor Core在 NVIDIA 的 Volta 架构中首次被提出。如图显示了 Volta 架构一个 SM 的结构组成图。可以看到，每个可编程多处理器 SM 内的 8 个Tensor Core在逻辑上与CUDA核心（包含INT32、FP32、FP64 等计算单元）是等同的，它们均匀地分布在每个 SM 内的4 个处理子块（processing block）中，每个处理子块内含有两个Tensor Core。

根据 NVIDIA 对 V100 的介绍，每个Tensor Core可以在一个周期内完成 64 = 4×4×4 次乘加运算，其中两个输入矩阵为 FP16 格式，累加矩阵与结果矩阵为 FP16 或 FP32 格式。硬件上，V100 GPGPU 包含了 80 个 SM，即 640 个Tensor Core，因此在 1.53GHz 的工作频率下，整个 GPGPU 能够达到125TFlops（FP16）的算力水平。

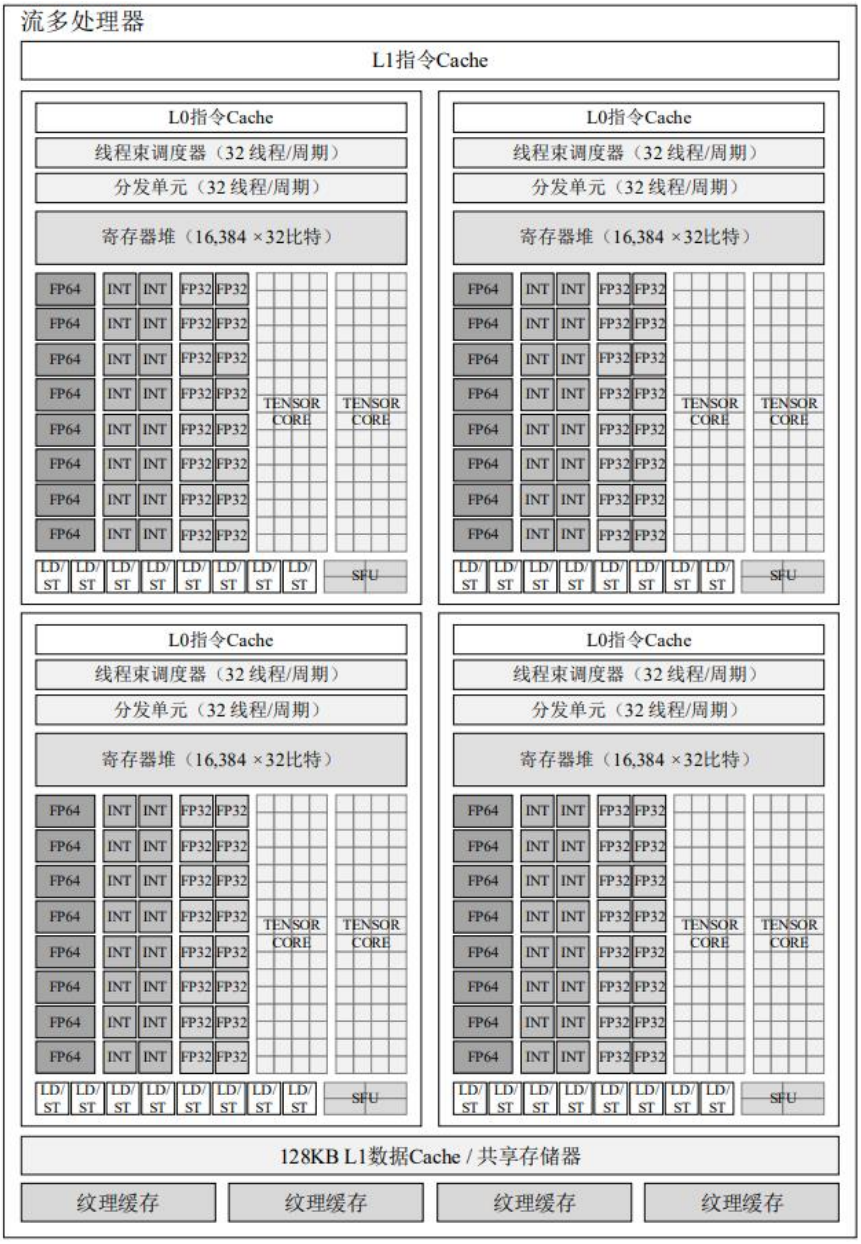
在指令集上，Volta 架构为基于Tensor Core矩阵乘法操作提供了新的 wmma 指令，能够完成16×16×16 的矩阵乘加运算。这个指令由上述 4×4×4 的矩阵乘加通过拼接组合完成。对于其他矩阵形状，如 15×15×15 的矩阵乘法，需要由编程人员填充至 16×16×16 才可计算；亦或是 17×17×17的矩阵乘法，编程人员需要自行拆分成若干个 16×16×16 的矩阵乘法。对于其他的数据类型，例如INT32、FP64 等，则不能使用Tensor Core。

在传统的 SIMT 计算模型中，编程人员通过编写每个线程的代码，每个线程独立完成各自的计算，包括操作数读取、计算、写回，再通过多线程并发来完成整个计算任务。例如可以为 16×16×16 的矩阵乘法运算声明 16×16 个线程，每个线程负责计算结果矩阵中的每个元素，最终是由这 256 个线程完成了 16×16×16 的矩阵乘法。

然而在 Tensor Core 中，硬件为了达到更高的计算效率进行了定制化的设计。为了

屏蔽大量的硬件细节，SIMT 模型发生了改变，即 16×16×16 的矩阵乘法不再由声明线程的方式完成，而是由特定的 API 来完成。这些 API 以warp为粒度，编程人员需要以warp为粒度控制Tensor Core的操作数读取、计算、写回。对于warp内 32 个线程如何具体执行，编程人员不可见也不受编程人员控制。

编程人员在使用 API 编写程序时，虽然形式上仍是 SIMT 模型，即线程块内的每个线程都执行该语句，但实际上要求归属于一个warp内的线程在执行 API 时的参数必须完全一致。在使用Tensor Core进行计算时，数据类型从标量数据变为一种新的专用于Tensor Core的数据类型 fragment。虽然仍然是读取、计算、写回的整体流程，但是执行粒度发生了改变。这是Tensor Core所带来的编程模型上的改变。



三、实验平台

本次实验在提供的NVIDIA V100服务器上进行

IP端口：202.120.38.28:43322

用户名为group+组号(比如第五组就是group5)

参考文件: `/home/cs433/files/Lab1`

数据集：`/home/cs433/files/benchmark/`

四、推荐实验步骤

（一） CUDA Tensor Core 编程

1. 随机选择10000张ImageNet中的图片
2. 使用CUDA编程，正确实现tensor core完成矩阵乘
3. 使用矩阵乘实现卷积操作
4. 使用cuda core实现的其他算子功能
5. 最终完成resnet18的推理过程，得到正确分类。

（二）简单GPU功能模拟器

1. 正确初始化GPU资源，合理模拟regfile、pregfile和gpu_memory。
2. 可参考Lab1中提供的tensor_core.cu指定Volta架构，编译得到可执行文件，并使用cuobjdump工具对其抽取SASS指令。
3. 学习FP16的数据格式，并实现模拟
4. 根据指令文档完成每个指令函数，保证正确性，（HMMA最重要）
5. 使用伪指令函数的方式实现gemm_kernel，参考代码为Lab1中提供的tensor_core.cpp。为了简化模拟流程仅模拟一个warp，warp内认为线程并行，完成小规模矩阵乘
6. 实现gemm函数，应对大矩阵进行分片,调用gemm_kernel函数
7. 测试gemm函数核对计算结果保证正确性
8. 将（一）中tensor core矩阵乘替换成模拟器中的矩阵乘函数
9. 推理10张图片，得到正确分类，对比最后一层的结果，若存在误差是否可以合理解释；
10. 模板中的函数仅供参考，在文档中说明设计思路，得到正确结果即可。

> 为了避免出现bra和simt stack等指令，kernel函数仅作(16x16x16)矩阵乘法，分片在cpu端分批传入kernel函数即可(仅在(二)模拟器作此要求，(一)实卡执行可随意构造kernel函数)

五、验收要求

1、代码+展示+文档

2、将源代码文件打包提交

2、展示时间为10分钟Pre+5分钟问答，Pre需要现场运行程序，实验步骤（一）（二）的执行结果。PPT展示应包括实现的算子，结合指令简述tensor core计算矩阵乘法的过程。

3、将所有文件打包压缩到 `学号_姓名_Lab1.zip` 并发送至kyehan3@sjtu.edu.cn

六、实验评分

1、保证正确性(50%)

2、展示时现场运行实验(一)推理程序记录时间，根据相对排名给分(20%)

3、展示+代码+文档质量(30%)

七、硬件功能建模及指令简述

寄存器文件

这一部分不需要大家对硬件进行建模，按以下的描述实现寄存器文件行为仿真即可。

理论上每个线程可以最大包含255个寄存器，其中R255表示为RZ，值恒为0。寄存器的位宽为32位，在某些指令中要求读写64/128位值时，要读写相邻的寄存器，例如 `IMAD.WIDE R2, R4, R6, R8;` 这样一条指令时，要求Sc为64位数，那么要读取R8，R9两个寄存器值，将其拼凑成64位数，其中R9为高位，R8为低位，存储与上述类似，64位结果要存储在R2和R3两个寄存器，R3为高位，R2为低位。

谓词寄存器

Pg为谓词寄存器，每个线程有7个谓词寄存器，P7为固定值PT,恒表示为True。Pg常在指令中用作bool判断，例如 `@!P3 IMAD.WIDE R2, R4, R6, R8;` 指令，首先获取P3的值，!表示取非，!P3为真时该指令才会执行，!P3为假时，指令不执行，指令未出现@Pg,默认执行。

常量内存

c[0x??][0x??]表示constant memory地址，实验中不做此memory的仿真。kernel函数中，c[0x0][0x160]表示第一个函数参数的值，根据第一个参数的位宽，向后依次排列，例如gemm_kernel(int *a,int *b,...)中，指针占八个字节，c[0x0][0x160]-c[0x0][0x167]表示a

的值，，因此c[0x0][0x168]-c[0x0][0x16f]表示b的值，以此类推。由于实验中不仿真constant memory地址，故在遇到此类值时直接将所需值作为函数参数传入即可。

指令简述

LOP3、LEA指令已经为大家实现，可作为参考。其余指令可根据描述自行实现。对指令的执行结果不确定时可参考Nvbit对指令前后进行数值的查看。

IMAD

IMAD将源操作数Ra与源操作数Sb相乘，得到的结果与源操作数Sc相加，最终结果存储到目的寄存器中

{@{!}Pg}　　IMAD{.WIDE}{.fmt} Rd, {-}Ra, {-}Sb, {-}Sc
.fmt　　源操作数数据类型
　　.U32 32位无符号数
　　.S32 32位有符号数(默认数据类型)
.WIDE Sc为64位数，32位Ra与32位Sb相乘后保留64位结果，与Sc相加后将64位结果保存到目的寄存器中

LOP3

LOP3将三个源寄存器做逻辑操作，结果存储到目的寄存器中
{@{!}Pg}　　LOP3{.LUT} Rd, Ra, Sb, Sc
.LUT　　查表

S2R

S2R将特殊寄存器的结果存储到目的寄存器中
{@{!}Pg}　　S2R Rd, SR
SR　　SR_LANEID　线程在该warp内线程号
　　SR_TID.X　threadIdx.x
　　SR_TID.Y　threadIdx.y
　　SR_CTAID.X　blockIdx.x
　　SR_CTAID.Y　blockIdx.y

CS2R

S2R将特殊寄存器的结果存储到目的寄存器中
{@{!}Pg}　　CS2R Rd, SR
SR　　SRZ　64位0

SHF

左右移动寄存器值，将结果存储到目的寄存器中, 由Sc和Ra组合64位值val（val=Sc<<32|Ra），val向左/右移动(HI?Sb+32:Sb)位，最终将val低32位存储到Rd中。
{@{!}Pg}　　SHF.dir{.maxshift}{.Xmode} Rd, Ra, Sb, Sc
.dir
　　.L 左移
　　.R 右移
.maxshift
　　.U32 应使用无符号（逻辑）移位
　　.S32 应使用有符号（算术）移位
.Xmode
　　.HI具有比从Sb移位多32位的效果。

LEA

LEA缩放偏移加法,通常用来计算有效地址
{@{!}Pg}　　LEA{.HI}{.X} Rd,{Pd0}, Ra, Sb, IMM, {Ps0}
.HI
　　计算高32位地址
.X
　　计算要附加Ps0的值

EXIT

退出指令

LDG

LDG从指定global memory地址获取指定位宽数据，结果存储到目的寄存器
{@{!}Pg}　　LDG{.E}{.sz} Rd, [Ra+IMM]
.E

表示地址为64位

.sz

数据位宽

[Ra+IMM]为global memory地址值

STG

STG将源操作数Sb的值存储到指定global memory地址

{@{!}Pg}　　STG{.E}{.sz} [Ra+IMM], Sb

.E

表示地址为64位

.sz

数据位宽

[Ra+IMM]为global memory地址值

八、参考

1. cuda编程 https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
2. Tensor core API https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma
3. T**ensor Core 参考论文** https://arxiv.org/pdf/1811.08309v2.pdf
4. 模型可视化工具 https://netron.app/
5. nvbit 指令执行结果查看工具 https://github.com/NVlabs/NVBit/releases
6. cuobjdump https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#cuda-binary
7. GPU相关书籍 https://item.jd.com/13199025.html