1. **EPCC使用docker**

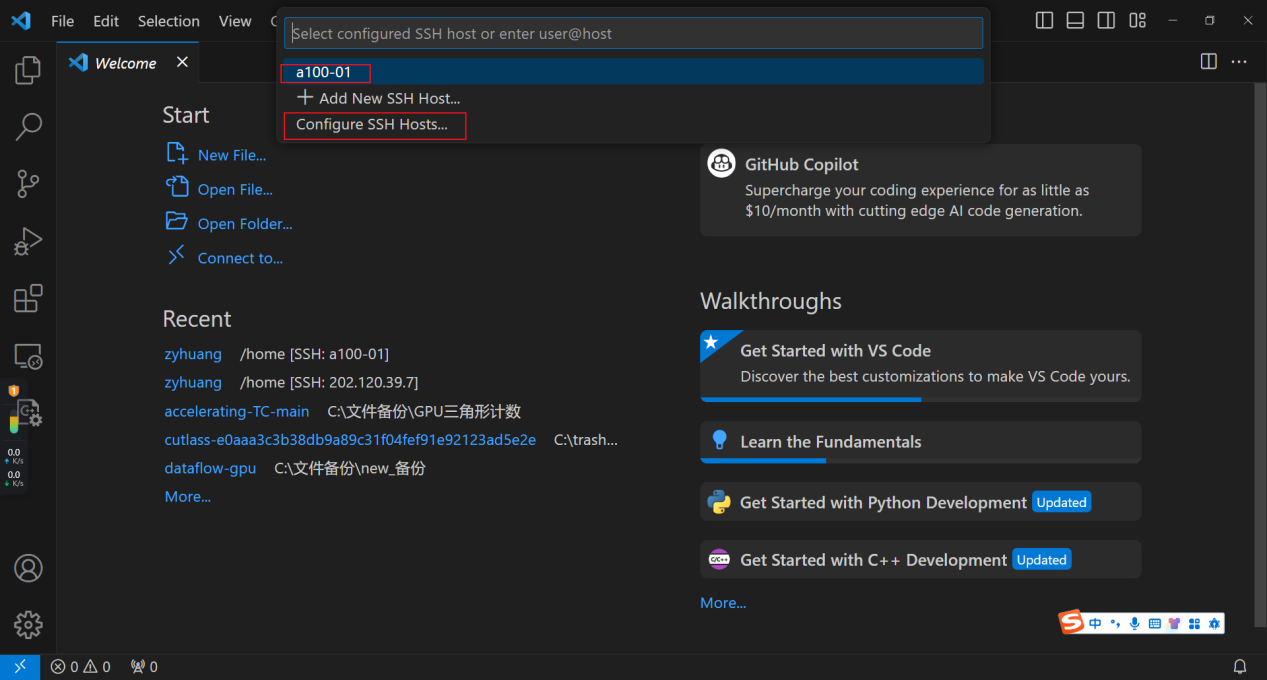
如果是centos就很卡，如果是ubuntu就好多了。能连上网，而且可以作为sudo权限。（不需要写sudo，而是直接作为sudo用户。）

1. **如何在VScode上连接远程服务器，并进入远程服务器的docker？**

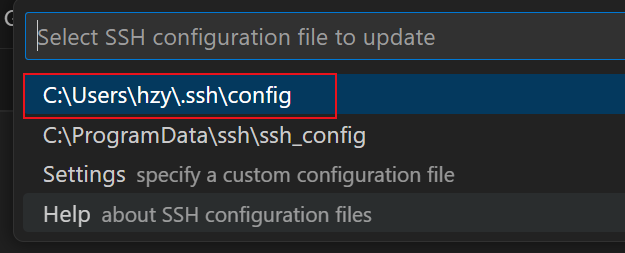
首先在VScode上安装remote - SSH插件。

然后点击左下角的><按钮，在上方搜索框会出现connect to Host remote - SSH的选项，点击！

然后如果是第一次连接，可以选择config SSH hosts



然后选一个文件去修改。其实这里点进去和去C盘文件地址修改是一样的。



这里我的文件写法有点独特，因为我们是直接登录进去是a40，然后再ssh a100-01，所以我需要跳转一下。所以写成这样：

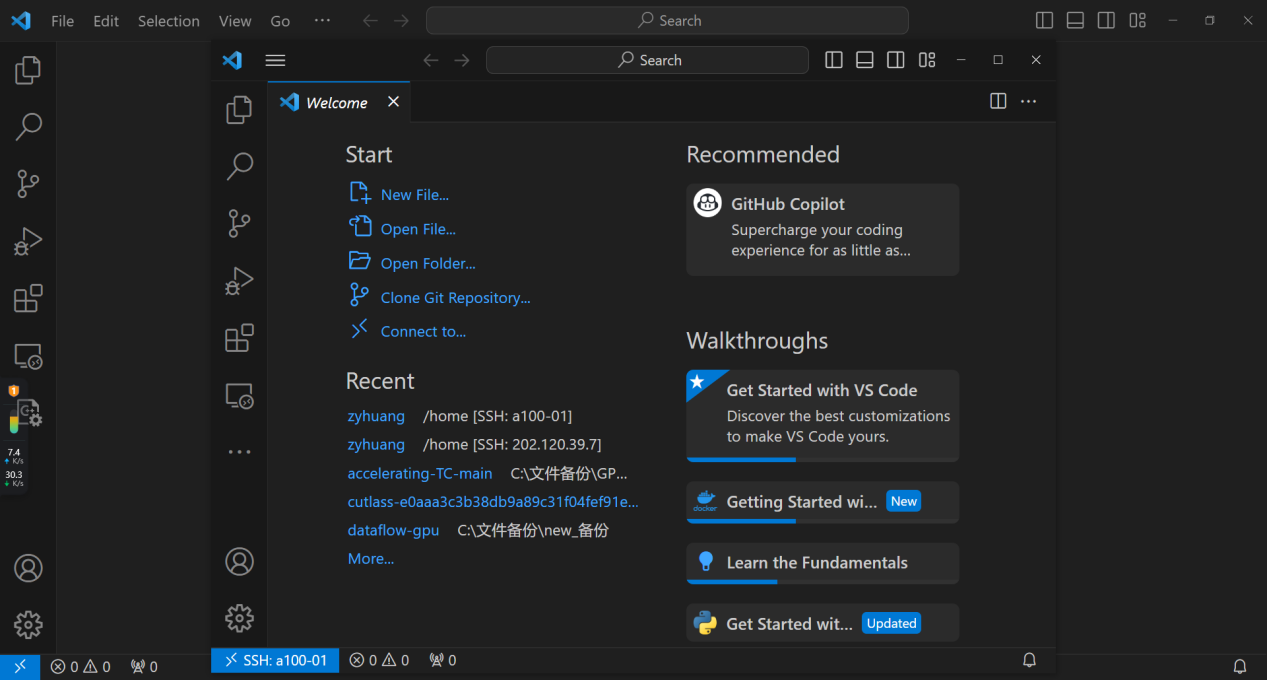
Host a100-01

    HostName a100-01

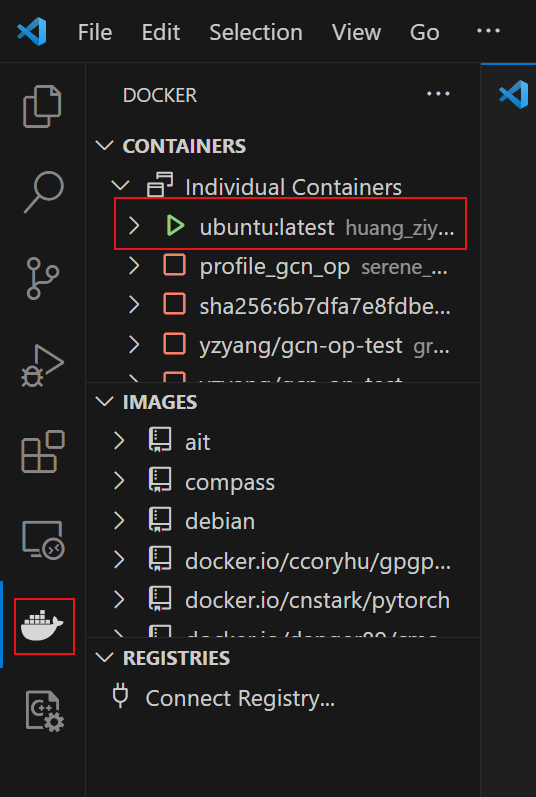
    User zyhuang

    ProxyJump zyhuang@202.120.39.7:8482

那么编辑好之后回到上一步，就会出现刚才那个a100-01选项。就需要输入两次密码（都一样啦）



然后在这个远程服务器的连接页面上安装extension：docker。



在mobaxterm登录的地方，连接a100之后创建docker。名字叫huang\_ziyu\_docker，然后此处就能显示出来！然后选择attach shell。那么就进去了。

为了使用GPU，需要使用nvidia-docker。那么方法是：

**~~docker run --name huang\_ziyu\_docker\_nvidia --privileged -v /home/zyhuang/:/home/zyhuang\_in\_docker/ -it --gpus all nvidia/cuda:latest /bin/bash~~**

docker run --name zyhuang\_docker\_2 --privileged -v /home/zyhuang/:/home/zyhuang/ -it nvcr.io/nvidia/pytorch:23.09-py3 /bin/bash

chmod 777 /home/zyhuang/flash-CUDA

首先，需要在宿主机上用上面这个777命令来给全部的权限。

然后，在docker内，如果是root创建或者下载或者修改的文件呢，那就是root身份，那么在宿主机上肯定就不是root了，就改不了。如果此时是用mobaxterm，也改不了（默认是用宿主机的身份来修改？）所以为了修改文件方便，需要在docker内创建另一个user，和外部宿主机的group\_ID和user\_ID一致，命令如下：

addgroup --gid 2084 zyhuang

adduser --disabled-password --uid 2084 --gid 2084 zyhuang （这里不需要有密码。。。所以disable了）

然后输入su zyhuang就可以激活

然后可以输入exit，这样就退出了zyhuang这个用户，就回到了root。如果再输入exit，就退出了docker。

--name huang\_ziyu\_docker\_nvidia 这个是命名。

--privileged加了这个，进去才能使用nvidia-smi。否则不行。

-v /home/zyhuang/:/home/zyhuang\_in\_docker/ 这个是为了挂载文件目录。

Bug记录：

有一次遇到这个报错

The following error occurred while trying to add or remove files in the

installation directory:

[Errno 13] Permission denied: '/usr/local/lib/python3.10/dist-packages/test-easy-install-2491.write-test'

The installation directory you specified (via --install-dir, --prefix, or

the distutils default setting) was:

/usr/local/lib/python3.10/dist-packages/

Perhaps your account does not have write access to this directory? If the

installation directory is a system-owned directory, you may need to sign in

as the administrator or "root" account. If you do not have administrative

access to this machine, you may wish to choose a different installation

directory, preferably one that is listed in your PYTHONPATH environment

variable.

然后可以加--user来指定安装位置以避免问题：

MAX\_JOBS=16 python setup.py install --user

MAX\_JOBS=16 pip install --user -e . 据说这个能够及时反映出代码的最新变化，有助于debug。。。。

为了让每次的值都是最新的，就是能够根据最新的代码来编译，我觉得应该删掉旧的文件：

rm -rf build/ dist/ \*.egg-info

MAX\_JOBS=16 python setup.py install --user

这样就不会有旧的代码的内容了。

或者放到一起执行：

rm -rf build/ dist/ \*.egg-info && MAX\_JOBS=16 python setup.py install --user

==================\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*==============\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

上面记录的方法是很有用的。下面是后来我又一次，为了使用cuda12.3的版本，在EPCC上的事情。自己是没办法在裸机上安装cuda的，需要使用docker。GPT截图如下：





docker pull nvidia/cuda:12.3.0-devel-ubuntu20.04

docker run --name zyhuang\_cuda\_12\_3\_devel --gpus all --privileged -v /home/zyhuang/:/home/zyhuang/ -it nvidia/cuda:12.3.0-devel-ubuntu20.04 /bin/bash

addgroup --gid 2084 zyhuang

adduser --disabled-password --uid 2084 --gid 2084 zyhuang

su zyhuang

其中，使用docker的base版本是没有cuda toolkit的，得用devel。以及还需要export PATH=/usr/local/cuda/bin:$PATH

这一句。然后就OK了，甚至还包含了ncu。

未经完全证实的，**给zyhuang用户赋予sudo权限**，以能够使用apt install等命令的方法：

总结一下，您现在应该在 Docker 容器的 root 用户下依次执行以下命令：

# 在 Docker 容器中作为 root 用户执行:

groupadd --gid 2084 zyhuang

useradd --uid 2084 --gid 2084 --create-home --shell /bin/bash zyhuang

usermod -aG sudo zyhuang

echo "zyhuang ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers

执行完这些之后，您可以用 su - zyhuang 来切换到 zyhuang 用户。

1. **学习安培架构的无寄存器读写。**

When this pattern occurs within an iterative algorithm, each thread block needs to synchronize after the shared[local\_idx] = global[global\_idx] assignment, to ensure all writes to shared memory have completed before the compute phase can begin. The thread block also needs to synchronize again after the compute phase, to prevent overwriting shared memory before all threads have completed their computations. This pattern is illustrated in the following code snippet.

这里说的第二种sync，意思应该是，写入共享内存，写完之后再同步，之后再读。避免有的还没写完，后面就开始读了。（写后需要同步）

第一种是全局内存读到共享内存，也是写完后同步。

1. **ncu-cli如何使用**

首先在远程服务器上建立nvidia-docker，然后下载compute。从宿主机修改权限（chmod +x）然后在docker里安装。至于如何运行文件：

<https://forums.developer.nvidia.com/t/nsight-profiling-crashes-with-error-code-9/230094/6>

ncu --set full --replay-mode application --app-replay-match grid --app-replay-buffer file -f --export output-file-full.nsight-cuprof-report ./a.out

sudo /usr/local/cuda-11.7/bin/ncu --set full --replay-mode application --app-replay-match grid --app-replay-buffer file -f --export output-file-full.nsight-cuprof-report ./delete

ncu --devices 0 --set roofline --replay-mode application --app-replay-match grid --app-replay-buffer file --app-replay-mode relaxed -f --export output-file-full.nsight-cuprof-report ./sgemm\_1（好像在多张卡上做测试需要加--devices 0这样。但是还没完全搞成功）

sudo /usr/local/cuda-12.2/nsight-compute-2023.2.2/ncu --set full --replay-mode application --app-replay-match grid --app-replay-buffer file -f --export output-file-full.nsight-cuprof-report ./sgemm3

如果想要更丰富的roofline图，可以把**--set full**改成**--set roofline**.

上面是普通执行文件（好像--app-replay-mode relaxed这个容易出问题）

ncu --set full --replay-mode application --app-replay-match grid --app-replay-buffer file --app-replay-mode relaxed -f --export output-file-full.nsight-cuprof-report python test\_delete.py

在SQZ上需要指明python具体位置：

sudo /usr/local/cuda-12.2/nsight-compute-2023.2.2/ncu --set roofline --replay-mode application --app-replay-match grid --app-replay-buffer file --app-replay-mode relaxed -f --export output-file-full.nsight-cuprof-report /home/zyhuang/anaconda3/bin/python blog\_test.py

上面是**python**代码里的kernel如何运行

--replay-mode application 避免RAM占用过高。

--app-replay-match grid --app-replay-buffer file --app-replay-mode relaxed 都很重要。否则就跑不起来。

a.out是之前编译得到的输出结果

然后把整个文件夹都下载下来，包括上delete.cu这个文件。否则查看source的时候PTX和source对应不上。

为了能对应上 ，在nvcc编译的时候加上-G。即：nvcc -G delete.cu -std=c++17 -arch sm\_80

如何使用PmSampling：：

ncu --section PmSampling --replay-mode application --app-replay-match grid --app-replay-buffer file -f --export output-file-full.nsight-cuprof-report --pm-sampling-interval 100 --pm-sampling-buffer-size 0 --pm-sampling-max-passes 100 C:\Users\hzy\anaconda3\envs\new\_torch\python.exe softmax.py

可以

1. **Ncu的lock-file如何解决**

==ERROR== Failed to open/create lock file /tmp/nsight-compute-lock. Please check that this process has write permissions on this file. See https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#faq for more details.

遇到上面这样的报错，怎么办？：

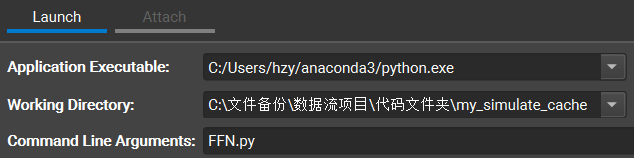
可以export TMPDIR=/tmp/other\_folder/ 来设置temp目录的地址，这样ncu会把临时文件放到新的temp 目录里。

export TMPDIR=/home/zyhuang/temp\_can/test\_FFN/new\_tmp

ncu --set detailed --replay-mode application --target-processes all -f --export output-file-full.nsight-cuprof-report /home/zyhuang/miniconda3/envs/py\_hzy\_new/bin/python /home/zyhuang/temp\_can/test\_FFN/pure\_FFN.py

遇到这个问题，先建立一个tmpdir，然后注意要制定完全的python和py文件地址。

1. **本地NCU如何执行python脚本**



1. **如何根据所需创建定制的NCU report**

ncu --section SpeedOfLight\_HierarchicalTensorRooflineChart --section MemoryWorkloadAnalysis\_Chart --section MemoryWorkloadAnalysis\_Tables --replay-mode application --target-processes all --clock-control=base --cache-control=all -f --export output-file-full.nsight-cuprof-report /home/zyhuang/miniconda3/envs/py\_hzy\_new/bin/python /home/zyhuang/temp\_can/test\_FFN/delete4.py

比如说这里，只选取我需要的section来测。

ncu --section SpeedOfLight\_HierarchicalTensorRooflineChart --section MemoryWorkloadAnalysis\_Chart --section MemoryWorkloadAnalysis\_Tables --replay-mode application --target-processes all **--clock-control=base --cache-control=all** -f --export /home/zyhuang/delete/47-GemmIdentityThreadblockSwizzle /home/zyhuang/cutlass/build/examples/47\_ampere\_gemm\_universal\_streamk/47\_ampere\_gemm\_universal\_streamk

1. **阅读李少侠矩阵乘法计算记录**

“Titan V L2 cache 带宽约 1.9TB/s，那么最乐观的结果即使 L2 cache 100% 命中，此方案的理论上限也只有 1.9T \* 0.48 = 0.912 tflops，远低于 14.9 tflops 的硬件算力。”

TB/s \* OP/byte = OP/s （byte和TB能约掉，flops就是OP/s）

“假设M\_tile，N\_tile都取 64，依上述表达式可得计算访存比为16，代入 Titan V FFMA 算力可得: 14.9tflops / 16 = 931GB/s，即对于 64\*64 大小的 thread block tile，平均访存带宽超过 931GB/s 后性能瓶颈就不在内存上了。”

OP/s / OP/byte = byte/s （就是当前计算访存比下最大能达到的最大内存流量，也就是说给更多的内存，计算也计算不过来。）

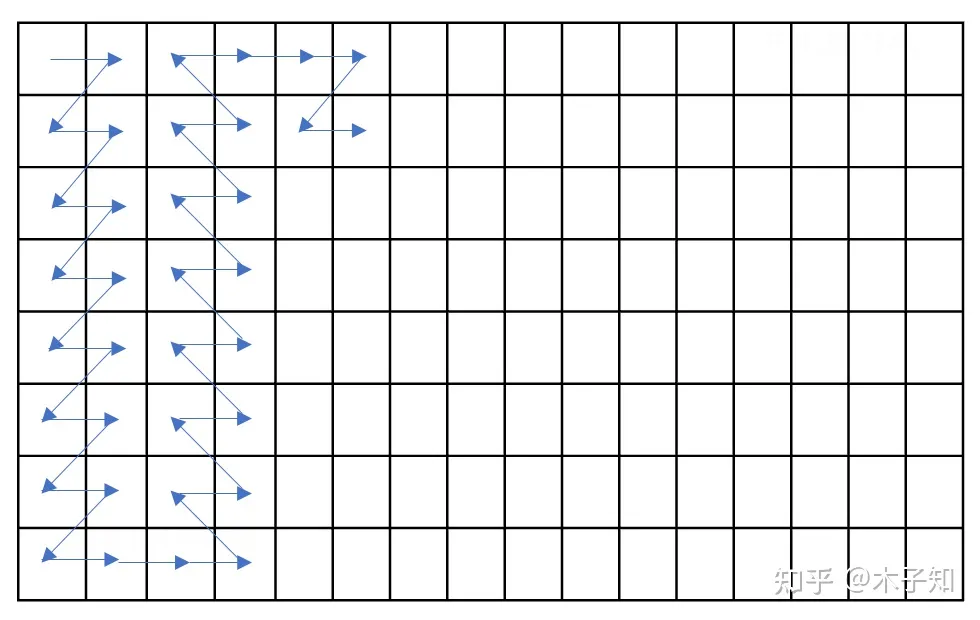
**对于计算访存比，一个是乘，一个是除，注意区分。**

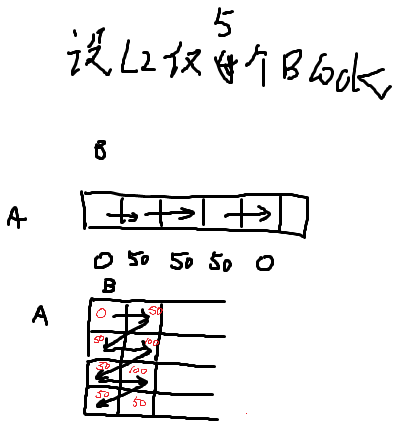
1. **提高L2 Cache命中率**

前面提到Block分块问题，假如Block分块已完成，那么在实际计算时这些Block分块的计算顺序该如何设计。最直接的方式是按行计算，这种计算方式会带来一个问题，显著降低L2 Cache命中率。

一般来说，对于同一行的C矩阵Block分块，其所需要的A矩阵的分块数据是相同的，同理对于同一列的C矩阵Block分块，其所需要的B矩阵的分块数据是相同的。如果按行计算，对于A矩阵来说，数据相同，L2 Cache命中率很高，但是对于B矩阵来说，数据都不相同，L2 Cache命中率很低，综合来看，考虑到L2 Cache的容量，L2 Cache命中率不会太高。

因此，可以采取如下图所示Swizzle计算方式，即“牛耕式”计算，兼顾A矩阵和B矩阵的L2 Cache命中率，提高整体的L2 Cache命中率。同时可以结合Block分块大小和L2 Cache容量，调整“牛耕”步长，获取最高的L2 Cache命中率。





我画了图来理解，确实是swizzle更好。假设每次只有一个block在计算。。。上面是A保有，但是B每次都是新的值，但是下面有可能爆出100%的命中率，以及哪怕L2占满了，丢失旧的，也有可能存在50%的命中率。

！！！而且，这个是控制block计算的精确顺序吗！！？？怎么做到的？？

1. **Mma如何用寄存器输入值**

<https://forums.developer.nvidia.com/t/can-we-directly-use-register-value-for-tensor-core-calculation/269808?u=202476410arsmart>

<https://forums.developer.nvidia.com/t/can-we-directly-use-register-value-for-tensor-core-calculation/269808?u=202476410arsmart>

\_\_global\_\_ void mma\_fp16\_acc\_fp32(float \*out) {

    float c[4] = {0., 0., 0., 0.};

    float d[4] = {0., 0., 0., 0.};

    half a[4] = {1., 1., 1., 1.};

    half b[2] = {1., 1.};

    // the above would set our input matrices to all 1

    // now lets modify some values

    if (threadIdx.x%4 == 0) {

    // set the first column of A to be 0, 1, 2, 3, ... 15

      a[0] = threadIdx.x/4; a[2] = threadIdx.x/4 + 8;

    // set the second row of B to 3,3,3, ... 3

      b[1] = 3;}

    unsigned const \*A = reinterpret\_cast<unsigned const \*>(&a);

    unsigned const \*B = reinterpret\_cast<unsigned const \*>(&b);

    float const \*C = reinterpret\_cast<float const \*>(&c);

    float \*D = reinterpret\_cast<float \*>(&d);

    asm(

      "mma.sync.aligned.m16n8k8.row.col.f32.f16.f16.f32 "

      "{%0,%1,%2,%3}, {%4,%5}, {%6}, {%7,%8,%9,%10};\n"

      : "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])

      :

        "r"(A[0]), "r"(A[1]),

        "r"(B[0]),

        "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])

    );

    memcpy(out+threadIdx.x\*2, D, 8);

    memcpy(out+8\*8+threadIdx.x\*2, D+2, 8);

}

这里为什么是a b c d每个寄存器只有4个或者2个值？因为总尺寸是A=16\*8, B=8\*8，又只有一个warp，所以平均下来，每个线程需要4个A的值，2个B的值。

为什么在asm里只有两个A：A[0], A[1]？ 因为这里都是32位的寄存器，而A是half，两个A才占一个寄存器。

为什么最后要memcpy两次？因为根据文档：

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#matrix-fragments-for-mma-m16n8k8>

这里的C是分两部分的，直接写一个表达式没法表示两段位置。（同理也可根据文档的标注理解  if (threadIdx.x%4 == 0) {

    // set the first column of A to be 0, 1, 2, 3, ... 15

      a[0] = threadIdx.x/4; a[2] = threadIdx.x/4 + 8;这一部分的逻辑。）

1. **如何跑GPU-dataflow**

首先我修改了，建了一个新的GPU-dataflow-new。里面似乎有很多原先代码跑不了的问题。。。各种import和cutlass-home的问题。。现在是改好了。然后进bert\_ffn去make，之后就可以运行。

1. **看258有感**

这里面的queue。比如后面几个案例，有多个mlp的，会第一个mlp读q1，写入q2，然后下一个mlp读q2，写入q3.。。能不能这时候写入q1呢？也许是因为q1还没算完。。。

这里还存在的问题是，分配block，有的block去算layer1，有的去算layer2，而有依赖关系先算layer1，结果要给layer2。那万一调度的时候block先去算layer2了，结果发现没有输入，然而layer1也没人去算，这不就死锁了吗？似乎可以用atomic的方法确定计算先后顺序。（去试试看！）

一个问题就是，所有的block的大小必须是一样的。哪怕可能算的是不同的功能。这一点可以通过更大的block来局部缓解，内部分配warp即可。不过前提是block数目足够多。

我的代码和258的不太一样。我的需要切分区域的每一行为界，探测到每个block都存好了，才能继续，但是也只需要负责自己的这一行。Acquire和release还得要。而且写出最好是能够按行来写出。

1. **如何使用tmux**

创建对话：

tmux new -s my\_session

重回对话：

tmux attach -t my\_session

上下滚动：

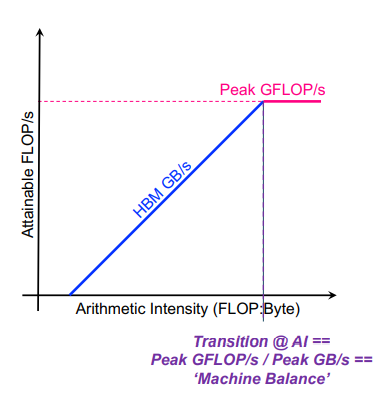
同时按住ctrl B（这里必须是大写的B哦！），然后再按[键。就可以了。要退出就按q键。

1. **树状和环状通信**

“当然,我们这里的通信时间只考虑传输带宽,而没有考虑每次传输都包含的延迟(latency)。当数据量V比较大时,延迟项可以忽略,上文的分析就是成立的。当V特别小,或者设备数p特别大时,带宽β就变得不重要了,反而是延迟比较关键,这时更好地实现就不是环状算法了,而应该使用树状通信。”

为什么？因为，环状需要的延迟是p-1个（总共p个处理器），每个之间通信都需要一个延迟，而树状是log2 P，一层需要一次。那么当P很大，环状的延迟p-1就远大于log2 P了。

1. **Roofline介绍与分析**



1. 为什么不可能超过蓝色线？

因为横坐标是FLOP/BYTE，纵坐标是FLOP/S，那么斜率就是BYTE/S，这个是由内存读取速度决定的，有上限。纵坐标显然是有上限的，这是硬件执行计算的极限速度，而横坐标是无上限的，那比如读进来两个数，我们反反复复的加减加减，也是可以的。

1. 为什么纵坐标一样：
2. 看起来不论是L1 L2还是DRAM，计算方式似乎都是用PE的计算量，除以本级缓存的时间（纵坐标）或者流过的数据量（横坐标）。因为PE计算量对大家都是一样的，所以纵坐标肯定是一样的。而本级缓存流过的数据量大家是不一样的，所以横坐标各有不同。
3. 那比如本图的L1和L2的横坐标也一样，那就说明所有L1的值都流过了L2。但是有些L2没有流回DRAM（L2更靠左，说明byte更大），说明L2对不同block起到了缓存的作用，有些直接在L2命中就走了，没有继续去DRAM。

FLOP是"Floating Point Operation"的缩写，意思是"浮点运算"。

FLOPS是"FLoating-point Operations Per Second"的缩写，意味着"每秒浮点运算次数"。

1. **在推理中，单GPU的显存不够用，是选择TP还是PP？**

TP能加速计算，但是需要更高的互联带宽，PP只需要低互联带宽，但是逐层计算带来的减速很明显。

从roofline角度来看，如果batch增大，那么同样多的矩阵乘法参数就能够执行更多的计算，那么在roofline上就会向右移动。但是推理常常batch很小，那么就会靠近memory bound

1. **If(thread==0)之后的sync的作用**

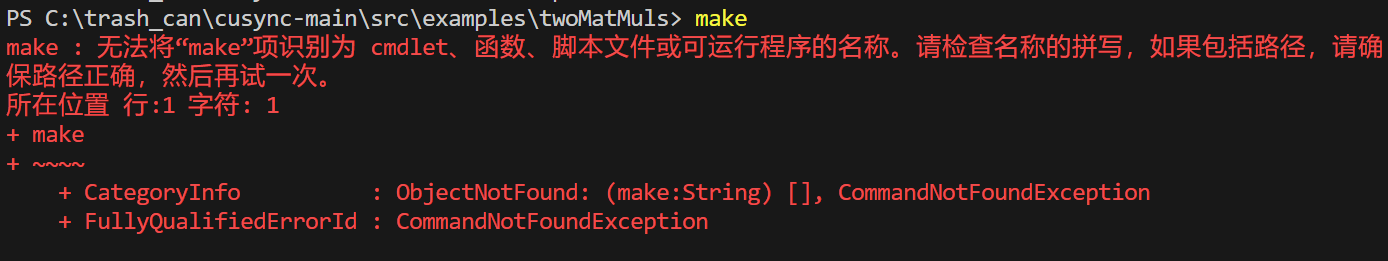
*// Synchronize to make sure that each thread reads* *// the correct value of isLastBlockDone.* \_\_syncthreads();

**if** (isLastBlockDone) {

首先有if(thread==0)的执行，然后在这个外面再sync一下，这样能够确保if内部写到全局内存的值能够被所有的线程都看到。这个操作的价值尤其在于，这个值在后续操作还要立刻用。

如果内存访问高度分散并且仅由单个线程访问，则执行未缓存的全局加载可能会节省带宽。可以使用内联 PTX 按指令控制缓存。还可以使用编译选项 -dlcm 禁用 L1 缓存。

1. **VScode里的环境变量和外部不同怎么办**



根据GPT的建议成功了：

尝试在 VSCode 中 手动设置终端的环境变量。你可以创建一个 .vscode 文件夹（如果它还不存在的话）在你的项目根目录下，然后在该文件夹中创建一个 settings.json 文件。在这个文件中，你可以添加以下设置：

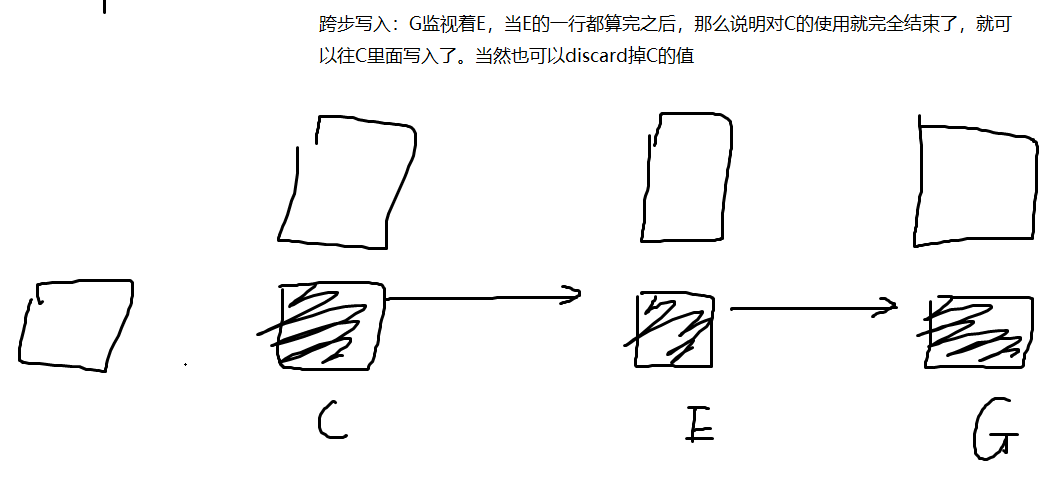
{

"terminal.integrated.env.windows": {

"PATH": "${env:PATH};C:\\cygwin64\\bin"

}

}



感觉好像是连续一整块比较好啊。。分立很多个不方便吧。。

1. **怎么用github进行版本回退**

首先将别人的github repo下载到自己的目录下。

然后git init

然后 git remote remove origin

然后在github自己的页面下创建一个私密的repo

git remote add origin https://github.com/ziyuhuang123/my\_cusync.git

然后就可以修改文件了。

然后就是

git add .

git commit -m "描述您的更改"

git push origin main

这里面很可能会要求输入用户名和密码。额。。。为了方便，密码最好记住：git config --global credential.helper store

然后密码一般是需要去生成一个的，PAT。。。有点麻烦。最好记下来。

/usr/local/cuda-11.7/bin/ncu --cache-control none --metrics lts\_\_t\_request\_hit\_rate.pct ./matrixMul

这个命令可以直接输出L2命中率。

1. **EPCC服务器反连本地VPN**

<https://www.cnblogs.com/hanX3/p/15834301.html>

问题描述：

  服务器用的是学校教育网，屏蔽了 github，但是笔记本连接热点或者挂 VPN 是可以访问 github 的。想要在笔记本上远程服务器时，可以访问 github。

解决：

  首先在笔记本设置中查看自己的网络代理转发端口，然后用 ssh -R 2080:127.0.0.1:10809 name@ipaddress，这里 127.0.0.1 是笔记本的地址，10809 是笔记本转发

2080" 和  export https\_proxy="127.0.0.1:2080" 来指定远程转发端口。如此，便可以访问 github.

我本地使用的是pigcha的VPN，进入“菜单”，“复制代理环境变量”，然后就得知了15732这个是本地的端口。然后登录我的EPCC的服务器：-p 8474 [zyhuang@202.120.39.7](mailto:zyhuang@202.120.39.7)

期智是：ssh -R 2080:127.0.0.1:15732 zyhuang@10.200.5.156

ssh -R 2080:127.0.0.1:15732 -p 30824 [root@10.210.22.155](mailto:zyhuang@202.120.39.7)

(base) C:\Windows\system32>**ssh -R 2080:127.0.0.1:15732 -p 8474 [zyhuang@202.120.39.7](mailto:zyhuang@202.120.39.7)**

**export http\_proxy="127.0.0.1:2080"**

**export https\_proxy="127.0.0.1:2080"**

**curl -I <https://www.google.com>**

zyhuang@202.120.39.7's password:

Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-88-generic x86\_64)

\* Documentation: https://help.ubuntu.com

\* Management: https://landscape.canonical.com

\* Support: https://ubuntu.com/advantage

\* Introducing Expanded Security Maintenance for Applications.

Receive updates to over 25,000 software packages with your

Ubuntu Pro subscription. Free for personal use.

https://ubuntu.com/pro

Expanded Security Maintenance for Applications is not enabled.

54 updates can be applied immediately.

1 of these updates is a standard security update.

To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.

See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.

To check for new updates run: sudo apt update

New release '22.04.3 LTS' available.

Run 'do-release-upgrade' to upgrade to it.

Your Hardware Enablement Stack (HWE) is supported until April 2025.

Web console: https://a100-04:9090/

Last login: Fri Jan 19 10:33:59 2024 from 10.2.0.1

(base) a100-04% **export http\_proxy="127.0.0.1:2080"**

(base) a100-04% **export https\_proxy="127.0.0.1:2080"**

(base) a100-04% **curl -I <https://www.google.com>**

**下面这个命令要输入的！否则不行的！**

**export http\_proxy="127.0.0.1:2080"**

**export https\_proxy="127.0.0.1:2080"**

HTTP/1.0 200 Connection established

HTTP/2 200

content-type: text/html; charset=ISO-8859-1

content-security-policy-report-only: object-src 'none';base-uri 'self';script-src 'nonce-PU8xU1L\_MoUz22XnqNNu9w' 'strict-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' https: http:;report-uri https://csp.withgoogle.com/csp/gws/other-hp

p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."

date: Fri, 19 Jan 2024 03:06:19 GMT

server: gws

x-xss-protection: 0

x-frame-options: SAMEORIGIN

expires: Fri, 19 Jan 2024 03:06:19 GMT

cache-control: private

set-cookie: 1P\_JAR=2024-01-19-03; expires=Sun, 18-Feb-2024 03:06:19 GMT; path=/; domain=.google.com; Secure

set-cookie: AEC=Ae3NU9N2aUZ\_WHE7f2C7wVJ9xM00ShDvXxKSQnR5QTqV8\_NTDx5WvYrrsw; expires=Wed, 17-Jul-2024 03:06:19 GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=lax

set-cookie: NID=511=pWumJkJRhckHpYPvu7IJI3ol5HNx10G1\_wbruzP0KhFZ6KLaPBWRpxeZTw0XBkQRPTaRHnr7AA1vuGdBUQFHh1TSvJRV6pSrCZGLuO8T3WdrpeXaym4z1\_OzpPJgoO4VlrgDoiGnzl7BeDldDp6B3tENZ7uddYQmQgzicJtnlDk; expires=Sat, 20-Jul-2024 03:06:19 GMT; path=/; domain=.google.com; HttpOnly

alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000

(base) a100-04% **curl -I https://www.github.com**

HTTP/1.0 200 Connection established

HTTP/2 301

content-length: 0

location: <https://github.com/>

1. **Cuda-gdb怎么用**

退出后用这个去查看多余的进程ps aux | grep cuda-gdb，然后kill -9 ....(进程号）

首先在编译的时候要在nvcc ..... 后面加上-g -G

然后就cuda-gdb delete1 （这里是编译完成的可执行文件）

有可能这里找不到cuda-gdb，那就去找。常见是在ls /usr/local/cuda/bin/cuda-gdb

那如果在，刚才这个命令会显示出具体地址。那然后加入path：

export PATH=/usr/local/cuda/bin/:$PATH && source ~/.zshrc

然后就进入了cuda-gdb。这时候要开始打断点了：break delete1.cu:30（第30行。这里是具体的.cu文件。）

然后就run，就会到断点处。

有可能到某个函数，想进去，上溯看看到底上层是哪里，那就step，或者s。

如果是想向下一行，就是next。简写是n

关于 finish 命令，它的用途是继续运行程序直到当前函数完成执行并返回。当您在函数内部调试并且想快速跳出该函数时，finish 非常有用。简写是fin

关于条件断点。可以写成break delete1.cu if threadIdx.x==0这样。虽然在主机端肯定还是会报错一堆，就是刚刚写完这个break之后。但是真的运行下去，还是OK的。不过注意条件要加在断点上。要是后面到地方了，比如print，再if，那就不行了。

向下运行直到某一行：until delete1.cu:60

1. **如何使用cmake？**

首先使用makefile会需要手动link很多库，如果用cmake，可以自动的寻找很多的库，会方便一点。我现在正在写libtorch（cpp版本的torch），这个拿makefile特别难写，用cmakelist就方便的多了。

文件结构一般是

---folder

|-----Cmakelist.txt

|-----folder.cpp

然后cd folder

mkdir build

cd build

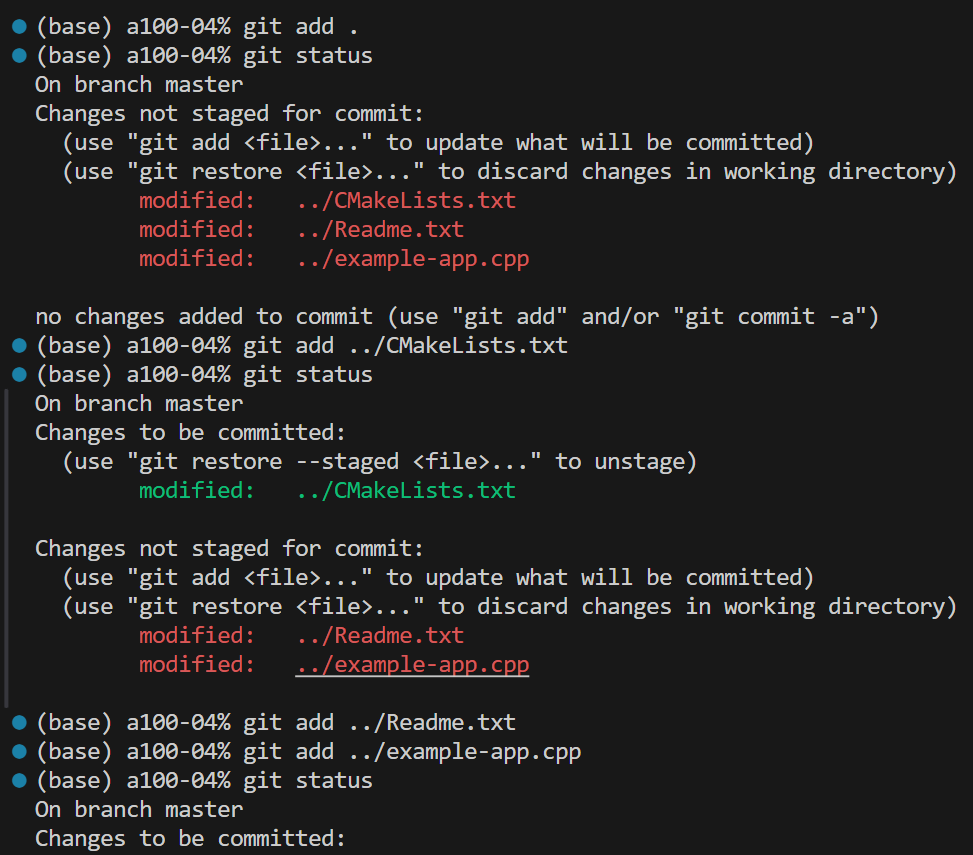
Cmake ..

cmake --build . --config Release -- -j 48

-j 48这个是多线程编译。总共有48个CPU内核，一般一个内核可以两个线程一起跑，似乎最大值是96。然后每次修改了代码之后的新的编译的时候不需要删掉旧的build，只需要cmake --build . --config Release -- -j 48

这一行就够了。

1. **为什么git add .失败？**



为什么这里git add .总失败？但是可以手动一个一个git add ../......这样？因为此时不在git仓库的根目录。

(base) a100-04% pwd

/home/zyhuang/temp\_can/example-app/**build**

1. **如何设置cutlass的环境变量？（否则flash attention报错了）**

**报错：**

In file included from /home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_launch\_template.h:11,

from /home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_split\_hdim256\_fp16\_sm80.cu:5:

/home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_kernel.h:7:10: fatal error: cute/algorithm/copy.hpp: No such file or directory

7 | #include <cute/algorithm/copy.hpp>

| ^~~~~~~~~~~~~~~~~~~~~~~~~

compilation terminated.

In file included from /home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_launch\_template.h:11,

from /home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_split\_hdim96\_fp16\_sm80.cu:5:

/home/zyhuang/temp\_can/extract\_cpp\_of\_flash/src/flash\_fwd\_kernel.h:7:10: fatal error: cute/algorithm/copy.hpp: No such file or directory

7 | #include <cute/algorithm/copy.hpp>

| ^~~~~~~~~~~~~~~~~~~~~~~~~

**解决办法：**

nano ~/.zshrc

export CPLUS\_INCLUDE\_PATH=$CPLUS\_INCLUDE\_PATH:/home/zyhuang/cutlass/include

export CUTLASS\_DIR=/home/zyhuang/cutlass

source ~/.zshrc

学长教我什么是矩阵乘法的multi-stage：就是

memcpy\_async(....)

memcpy\_async(....)

compute()

这样写两次读取，当然也得分配更多的shared memory，然后计算一次，这就算stage=3了。

1. **Flash attention学习日记**

新来的flash attention用python setup.py install编译，一堆报错，尤其是cute:....一堆的东西。就怀疑是cutlass版本不对，然后去删掉cutlass。更新环境变量（这些内容都是GPT教的）。然后就跑起来了。

再一次，遇到了cute等的版本错误。就是因为自己的cutlass太新了，一看flash支持的是3.4.1的@ bbe579a版本。然后使用该方法在cutlass文件夹下面更换到@ bbe579a：

git checkout bbe579a

然后注意自己的extract....文件夹的内容也得保持一致，和flash一样才行。

1. **如何在SQZ上连接外网**

export https\_proxy=http://127.0.0.1:7990 http\_proxy=http://127.0.0.1:7990 all\_proxy=socks5://127.0.0.1:7991

然后取消掉用：

unset http\_proxy

unset https\_proxy

unset all\_proxy

1. **让SQZ使用CUDA12.3**

export PATH=/usr/local/cuda-12.2/bin:$PATH

export LD\_LIBRARY\_PATH=/usr/local/cuda-12.2/lib64:$LD\_LIBRARY\_PATH

source ~/.bashrc

nvcc --version

1. **如何给华为项目发邮件**

收件人：huguipeng@huawei.com, jialiancheng@huawei.com, junsongwang@huawei.com, huangjiangle@huawei.com, hutianchi1@huawei.com

抄送: leng-jw@sjtu.edu.cn, altair.liu@sjtu.edu.cn, h.y.zhang-zdy@sjtu.edu.cn, [weiminghu@sjtu.edu.cn](mailto:weiminghu@sjtu.edu.cn)

1. **如何使用TileFlow**

首先需要在docker里用。按照本文档部分1介绍那样创建docker后，执行TileFlow的md文件介绍的初始化过程

如果没有gdb，可能需要在docker里安装一下gdb。

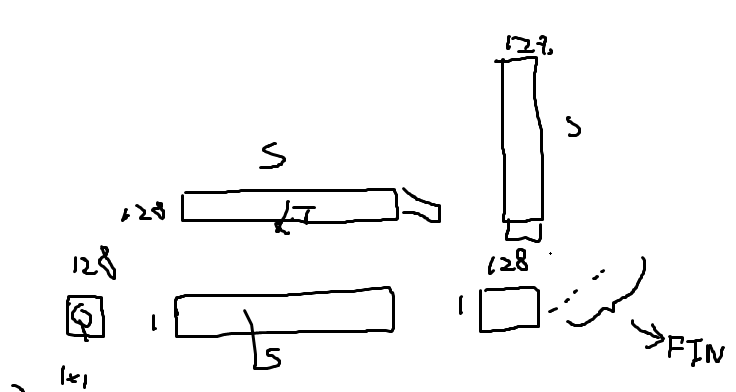
然后cd tutorials/00-GEMM

gdb ../../build/bin/tileflow 注意这里要能够补全出来tileflow，就是说从当前位置去寻找到tileflow可执行文件，路径要对。

break /home/zyhuang/temp\_can/TileFlow/src/application/main.cpp:169 这里没法被识别自动补全。。。但是后续也是可以的。。。

run arch/arch.yaml prob/prob.yaml map/map.yaml macro.yaml 然后这一行就会执行，并且停在上面所设置的break的位置。

1. **KV cache的执行细节**



M的宽度是1，但是长度是head-dim，然后多个attention拼接到一起。之后的FFN也是seq=1。

然后每次input\*W\_Q K V的时候，只生成一个很小的块，每次拼接到K和V上面，如上图最旁边的小块所示。

1. **Cutlass如何用tensor core和cuda core**

/home/zyhuang/cutlass/examples/21\_quaternion\_gemm--->cuda core

47\_ampere\_gemm\_universal\_streamk-->tensor core

上述两个代码可以执行。

以防以后找不到，打包压缩放到了my\_simulation\_cache文件夹里面

1. **学习cusync**

现在在学习cusync的cutlass是怎么写的。然后进行修改。

1. 研究cusync，这里面的cutlass是怎么实现的？怎么插入到里面去控制？

2. 基于这一套方法，我怎么消除掉其中的avoid\_kernel？

首先，**cusync是如何使用cutlass的**？我先研究了ml-bench的代码。跑起来之后再抽丝剥茧的删去无用部分。现在地址在/home/zyhuang/temp\_can/cusync/src/ml-bench/transformer下。

make build/mlp-eval-rowsync

build/mlp-eval-rowsync --batch 256 --check true --model gpt3 --split-k1 1 --split-k2 1 --policy cusync

CUDA\_LAUNCH\_BLOCKING=1 build/mlp-eval-rowsync --batch 128 --check true --model gpt3 --split-k1 1 --split-k2 1 --policy cusync --order-line 1 --file-path "/home/zyhuang/temp\_can/dataflow\_code/block\_swizzle/gen\_order/new\_gen\_order/orders\_512.txt"

build/mlp-eval-rowsync --batch 256 --check true --model gpt3 --split-k1 1 --split-k2 1 --policy cusync --order-line 1 --file-path "/home/zyhuang/temp\_can/dataflow\_code/block\_swizzle/gen\_two\_order/order\_256.txt"

swizzle duration(us) L2%

1 15,120 73.11 --->15712 73.54

2 9484 81.49 --->9007 85.03

4 9326 89.59 --->8993 86.70

8 9384 84.33 --->9117 82.26

16 9367 90.55 --->9067 83.90

terminate called after throwing an instance of 'cutlass\_cutlass::cuda\_exception'

what(): std::exception

遇到过这样的错误。然后nvidia-smi发现是GPU-0的内存被占满了。

然后用export CUDA\_VISIBLE\_DEVICES=1指定了使用GPU1，就可以跑起来了。

1. **如何用SQZ的GPU**

见管仁阳的聊天记录

我的密码都设为了123456

每次结束后都要在网页上点击备份。

开启前在网页上选终端，然后passwd

然后更新密码（好像每次都要重来一次）

备份的时候，要关掉其他VPN，否则会影响。即使不关VPN，也可以点击按钮，但是不会弹出状态栏，其实是没有备份成功。

1. **如何独占GPU做测试**

（首先需要将自己先设为root）

**将GPU设置为独占模式**

sudo nvidia-smi -i 7 -c EXCLUSIVE\_PROCESS

**设置GPU可见性**

export CUDA\_VISIBLE\_DEVICES=7

**恢复为共享模式**

sudo nvidia-smi -i 7 -c DEFAULT

**检查GPU模式**

nvidia-smi

1. **Github如何个性化搜索**

repo:NVIDIA/cutlass path:/^include\/cutlass\/epilogue\/collective\// content:/\bstore\b/

content:/\b \b/

在这个中间插入想要精确搜索的内容。

content:的意思是，只搜索内容，而不搜索标题。

/\b \b/的意思是前后不能带其他东西。比如我搜索store，那storeAdd就不行

/\bTmaWarpSpecialized\b/

1. **cutlass如何安装/编译**

首先git clone cutlass的文件夹。然后进入cutlass文件夹内。

export CUDACXX=${CUDA\_INSTALL\_PATH}/bin/nvcc ---->这地方需要根据自己的CUDA位置稍微改改。

mkdir build && cd build

cmake .. -DCUTLASS\_NVCC\_ARCHS=90a ---->根据自己的cc改 （这里遇到必须cmake --version》=3.19，不过其实安装cmake可以用.sh，而且是不需要sudo权限的。）

make ----->很慢！这会编译全部的文件

我们往往需要运行example文件夹里的具体某一个文件，如何编译呢？如果像上面那样make，会编译所有的文件，是非常非常慢的，可能要1小时以上。现在以编译./examples/cute/tutorial/sgemm\_1.cu这个文件为例。首先，虽然这里是有CMakelist的，但是不可以在本地直接make或者cmake。否则会出现：

**Unknown CMake command "cutlass\_example\_add\_executable".**

正确的方法是回到**cutlsss主目录**下，

mkdir build

cd build

然后执行cmake ..

然后此时应该make，但是具体make谁呢？可以make help。这样会列出所有可以make的目标。其实tutorial下面有很多个想make的，不过可能需要一个一个make吧。那我们现在就：make sgemm\_1

然后等个3分钟就差不多好了。生成的文件是在cutlass/build/examples/cute/tutorial下面的。不是在原先的cutlass/examples/cute/tutorial下面。

有时间去试试看ncu：docker挂载一个tmp，然后自己创建一个tmp，然后

touch /tmp/nsight-compute-lock

chmod 666 /tmp/nsight-compute-lock

这样是不是不需要root了？是不是也不需要docker了？好像是的）

在H100上没法联网，在cmake .. -DCUTLASS\_NVCC\_ARCHS=90a的时候会报错：googletest无法下载。解决办法是：

* 下载googletest并上传到服务器，比如此时放到位置：

/home/zyhuang/temp\_can/googletest

找到cutlass/cmake/googletest.cmake，将下面内容：

FetchContent\_Declare(

googletest

GIT\_REPOSITORY https://github.com/google/googletest.git

GIT\_TAG v1.14.0

)

更换为：

FetchContent\_Declare(

googletest

SOURCE\_DIR /home/zyhuang/temp\_can/googletest

)

再执行cmake .. -DCUTLASS\_NVCC\_ARCHS=90a即可成功。

1. **cutlass如何用nvcc单独编译**

nvcc ampere\_gemm.cu -o ampere\_gemm -arch=sm\_80 -std=c++17 -I ../cutlass/include -I ../cutlass/tools/util/include

用上述命令可以。代码是

https://github.com/NVIDIA/cutlass/tree/922fb5108b61c909ea3170d656f89ce151a246fd/examples/14\_ampere\_tf32\_tensorop\_gemm

往往会出现缺少helper.h的报错。那么从cutlass里面找到helper.h文件，放到当前同一目录下依然叫helper.h，然后编译就可以过了。

**另一个例子**：

zyhuang@ljwA100:~/cutlass/examples/cute/tutorial$ **nvcc -o sgemm\_nt\_1 sgemm\_nt\_1.cu -arch=sm\_80 -std=c++17 -I ../../../include -I ../../../tools/util/include --expt-relaxed-constexpr**

zyhuang@ljwA100:~/cutlass/examples/cute/tutorial$ ls

CMakeLists.txt sgemm\_nt\_1 sgemm\_nt\_1.cu

zyhuang@ljwA100:~/cutlass/examples/cute/tutorial$ ./sgemm\_nt\_1

Using device 0: NVIDIA A100 80GB PCIe (SM80, 108 SMs)

M = 5120

N = 5120

K = 4096

Verification by comparison with cuBLAS is disabled, either because the CMake option CUTLASS\_ENABLE\_CUBLAS was explicitly set to OFF, or because CMake could not find cuBLAS. If you would like to enable verification with cuBLAS, please set the CMake option CUTLASS\_ENABLE\_CUBLAS to ON, rerun CMake, and recompile this example.

CUTE\_GEMM: [16347.1]GFlop/s (13.1368)ms

这个代码是位于/cutlass/examples/cute/tutorial/sgemm\_nt\_1.cu，如果直接写cmake或者make之类的不能编译会报错，搞不明白为什么虽然。然后就研究自己写编译代码。

由于代码开头有#include <cute/tensor.hpp>，所以需要寻找到它的位置：./include/cute/tensor.hpp，然后就需要-I ../../../include

由于代码开头有

"cutlass/util/print\_error.hpp"

"cutlass/util/GPU\_Clock.hpp"

"cutlass/util/cublas\_wrappers.hpp"（如果定义了CUTLASS\_ENABLE\_CUBLAS）

"cutlass/util/helper\_cuda.hpp"

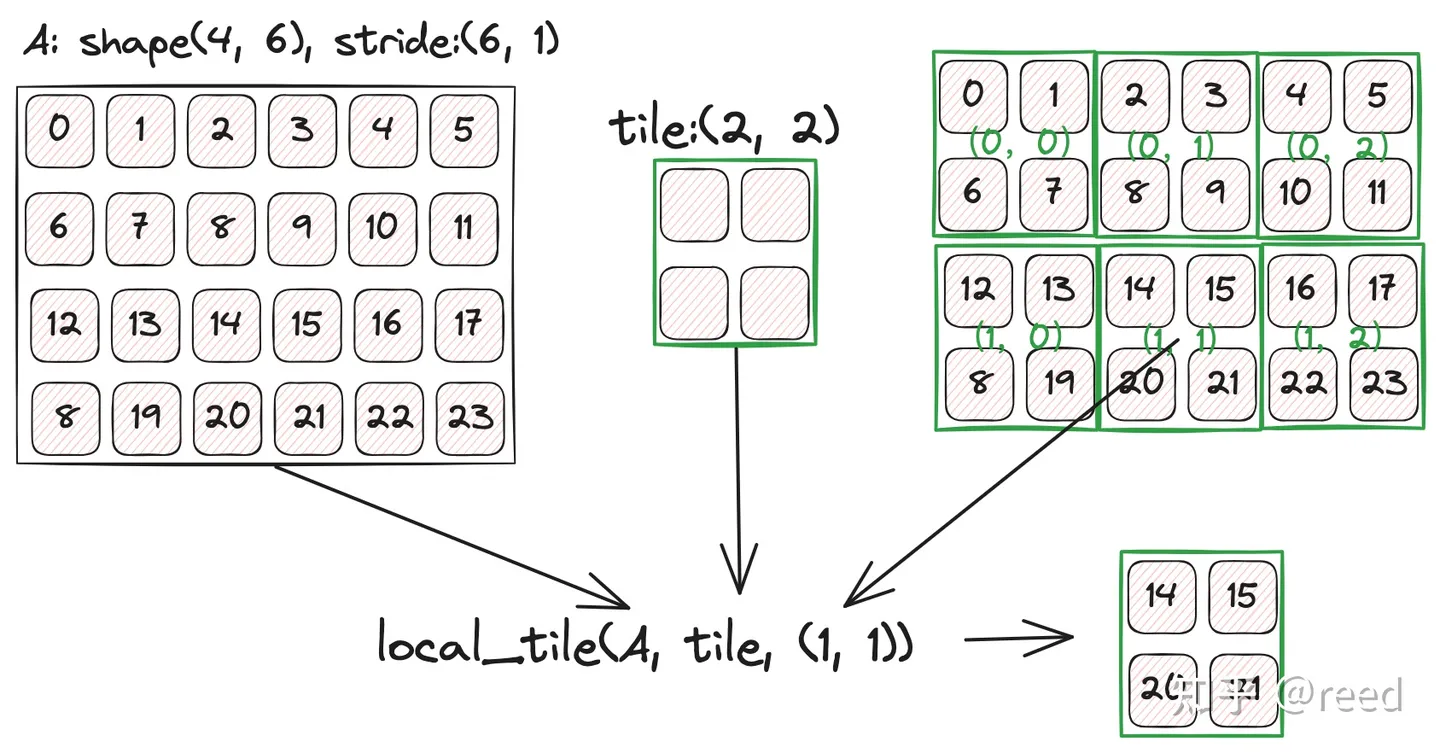
所以就需要-I ../../../tools/util/include

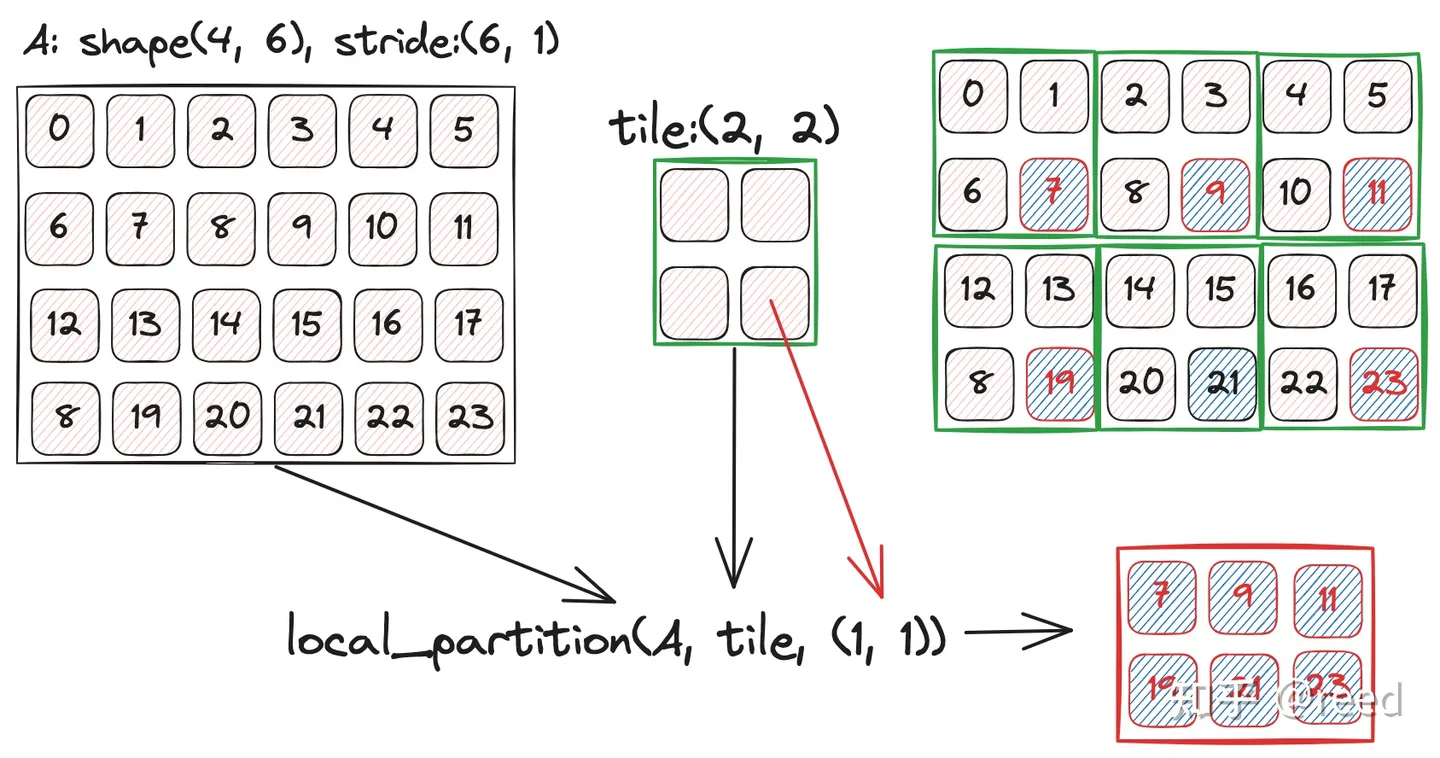
额，反正，这里的include是相对路径，也可以写成

nvcc -o sgemm\_nt\_1 sgemm\_nt\_1.cu -arch=sm\_80 -std=c++17 -I ../cutlass/include -I ../cutlass/tools/util/include --expt-relaxed-constexpr

1. **Cute学习日记**

什么是local\_tile和local\_partition？





  using T = cute::half\_t;

  using namespace cute;

  // 假设 self\_defined\_value 数组的维度为 16x16。

  int dim1 = 6;

  int dim2 = 6;

  T \*self\_defined\_value;

  self\_defined\_value = (T\*)malloc(sizeof(T) \* dim1 \* dim2);

  // 初始化 self\_defined\_value 数组

  for(int i = 0; i < dim1; ++i) {

      for(int j = 0; j < dim2; ++j) {

          self\_defined\_value[i \* dim2 + j] = i \* 10 + j / 10.0; // 根据给定公式赋值

      }

  }

  // 在打印时，将 half\_t 转换为 float

  for (int ii = 0; ii < dim1; ii++) {

      for (int jj = 0; jj < dim2; jj++) {

          // 将 half\_t 类型的值转换为 float 来打印

          printf("%2.3f\t", static\_cast<float>(self\_defined\_value[ii \* dim2 + jj]));

      }

      printf("\n");

  }

  printf("\n");

  // 现在可以使用 self\_defined\_value 来创建和操作 Tensor 对象

  Tensor tensor\_modified = make\_tensor(self\_defined\_value, make\_shape(dim1, dim2), make\_stride(dim2, 1));

  auto tile\_modified = make\_tile(2, 2);

  auto coor\_modified = make\_coord(1, 0);

  // Tensor tc1\_modified = local\_tile(tensor\_modified, tile\_modified, coor\_modified);

  auto tA = make\_layout(make\_shape(Int<3>{}, Int<3>{}), LayoutRight{});

  Tensor tc1\_modified = local\_partition(tensor\_modified, tA, 3);

  print\_tensor(tc1\_modified);

  // 注意tA的LayoutRight，这是行主序的意思，所以这里打印出来的就是第四个值，10.0。

  Tensor tc2\_modified = local\_partition(tensor\_modified, tA, 4); // 很明显了，实验表明，起码对于tA二维的情况，是列主序的，就是后面的index是优先沿着列往下走的。这是因为，默认的make\_layout创建出来就是列优先的。如https://github.com/NVIDIA/cutlass/blob/8f7d2789b8b04418ac01e515019dc3b74b249070/media/docs/cute/01\_layout.md这里所示。

  print\_tensor(tc2\_modified);

local\_tile是取一整块出来。而local\_partition是根据index，先按tA来切ptr，然后在每个tile里取第index个值（会按照列主序来取，除非tA写了是layoutright就是行主序）。并且注意这里的index只能是一个标量，哪怕你给了数组对，也会按某种方式理解为一个标量。

上面的输出如下：

tensor\_modified

0.000 0.100 0.200 0.300 0.400 0.500

10.000 10.102 10.203 10.297 10.398 10.500

20.000 20.094 20.203 20.297 20.406 20.500

30.000 30.094 30.203 30.297 30.406 30.500

40.000 40.094 40.188 40.312 40.406 40.500

50.000 50.094 50.188 50.312 50.406 50.500

ptr[16b](0x55bc83f628bc) o (2,2):(18,3): tc1\_modified

10.00 10.30

40.00 40.31

ptr[16b](0x55bc83f628be) o (2,2):(18,3): tc2\_modified

10.10 10.40

40.09 40.41

  using mma\_op = SM80\_16x8x8\_F16F16F16F16\_TN;

  using mma\_traits = MMA\_Traits<mma\_op>;

  using mma\_atom = MMA\_Atom<mma\_traits>;

using MMA = decltype(make\_tiled\_mma(mma\_atom{},

                      make\_layout(Shape<\_2, \_2, \_1>{}),

                      make\_layout(Shape<\_1, \_2, \_1>{})));

这个写法和下面是差不多的：

  using MMA = decltype(make\_tiled\_mma(mma\_atom{},

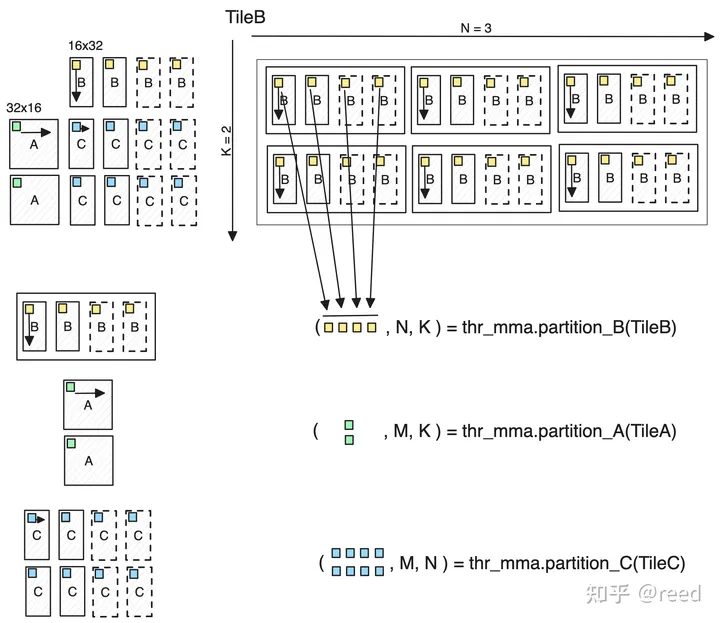
                      make\_layout(Shape<\_2, \_2, \_1>{}),

                      Tile<\_32,\_32,\_16>{}));

上面的写法是：mma\_atom{}, Layout, val\_layout，含义和reed的写法一致，第一个Layout影响线程数，在main函数中，我们选择使用Ampere架构提供的16x8x16的Tensor Core矩阵乘法指令，数据精度和计算精度都为fp16。然后通过MMA\_Traits得到mma\_traits，继续将traits转换成MMA\_Atom。我们知道SM80的Tensor Core执行是warp level的，也就是说这个MMA\_Atom是32个线程，我们对MMA\_Atom能力通过增加线程的方式进行M、N方向的重复，**同时我们让B矩阵C矩阵使用更多寄存器在N方向扩展2次**，得到main函数中的MMA类型。这样，我们便可以得到TiledMMA需要32x2x2 = 128线程，其能处理的矩阵的大小: M = 16 x 2 x 1 = 32, N = 8 x 2 x 2 = 32, K = 16 x 1 x 1 = 16, 即TiledMMA能处理的MNK为32x32x16。

下面的tile是新写法，直接可以指定尺寸为32,32,16。

上面加粗的，更多寄存器，与事实测试不符！很奇怪。甚至当用1,2,1相比于1,1,1还更少寄存器。

如上图所示，纵向是K=2，横向是N=3，而kTileK=32，我们这里的tile上面说了，在K维度上是16，所以这里的MMA\_K是2。

（MMA\_M=2, MMA\_N=3）这三个维度虽然没有明说，不过估计是在cute::gemm内部会进行迭代。显式迭代的只有大tile外侧的k，就是对128\*32这里的总K除以32得到的K\_num

1. **如何使用DSM**

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#distributed-shared-memory>

}

// cluster synchronization ensures that shared memory is initialized to zero in

// all thread blocks in the cluster. It also ensures that all thread blocks

// have started executing and they exist concurrently.

**cluster.sync(); <<<<-------要有这一步！在开始工作前就要有！**

for (int i = tid; i < array\_size; i += blockDim.x \* gridDim.x)

{

int ldata = input[i];

为了设置cluster尺寸，需要使用cudaLaunchKernelEx。如果使用传统的<<<>>>方式，则必须写死cluster尺寸，（此时可以以变量来调整gridDim和blockDim），如下：

\_\_global\_\_ void myKernel() \_\_cluster\_dims\_\_(2, 1, 1) {

// kernel code

}

// Launch kernel

myKernel<<<gridDim, blockDim>>>();

但是使用cudaLaunchKernelEx，就可以动态调整三个值，如下：

\_\_global\_\_ void myKernel() {

// kernel code

}

cudaLaunchConfig\_t config = {0};

config.gridDim = gridDim;

config.blockDim = blockDim;

cudaLaunchAttribute attribute[1];

attribute[0].id = cudaLaunchAttributeClusterDimension;

attribute[0].val.clusterDim.x = variableX; // 这里可以使用变量

attribute[0].val.clusterDim.y = variableY;

attribute[0].val.clusterDim.z = variableZ;

config.numAttrs = 1;

config.attrs = attribute;

cudaLaunchKernelEx(&config, myKernel, args...);

1. **如何使用TMA**

如何用cute来写TMA：

<https://research.colfax-intl.com/tutorial-hopper-tma/>

TMA的PTX页面：<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#data-movement-and-conversion-instructions-cp-async-bulk>

3. TMA的cuda API页面：https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#using-tma-to-transfer-one-dimensional-arrays（要用memcpy\_async加一些mbarrier的PTX）

TMA和mbarrier等非常相关！虽然我不熟悉这块。

可以从上面第三点提供的代码开始。我暂时还没跑通。

1. **研究cutlass/example/48**

**KernelScheduleAuto到底调用的是谁？**

首先要找到KernelScheduleAuto到底底层调用的是谁。现在找到其位置在：<https://github.com/NVIDIA/cutlass/blob/5b283c872cae5f858ab682847181ca9d54d97377/include/cutlass/gemm/collective/builders/sm90_gmma_builder.inl>

最终目标是获得KernelSchedule，根据cutlass代码，得到下面这样的判断树：

1. 判断是否支持TMA：

如果 IsTmaCompatible 为真，进入下一步判断；

否则，使用 KernelCpAsyncWarpSpecializedSchedule。

1. 判断CUDA版本是否>=12.1：

如果 CUDA 版本>=12.1，进入下一步判断；

否则，使用 KernelTmaWarpSpecialized。

1. 判断是否为Mixed Width Input：

如果 IsMixedWidthInput 为真，进入Mixed Width Input路径；

否则，进入Same Width Input路径。

3.1 Mixed Width Input路径：

判断 TileShape\_M 是否为64：

如果 TileShape\_M 为64，使用 KernelTmaWarpSpecializedPingpongMixedInput；

否则，使用 KernelTmaWarpSpecializedCooperativeMixedInput。

3.2 Same Width Input路径：

判断 TileShape\_M 是否为64：

如果 TileShape\_M 为64，使用 KernelTmaWarpSpecializedPingpong；

否则，使用 KernelTmaWarpSpecializedCooperative。

最终我们是使用KernelTmaWarpSpecializedCooperative

然后具体进入同一个文件的上面寻找，能找到最后可以是

// GMMA\_TMA\_WS\_RS

// GMMA\_TMA\_WS\_SS

这两者之一。。。暂时先研究GMMA\_TMA\_WS\_SS。

里面和TMA和warp specialization相关的是MainloopSm90ArrayTmaGmmaWarpSpecialized

GMMA\_TMA\_WS\_SS最底下定义了：

  using CollectiveOp = CollectiveMma<

      DispatchPolicy,

      TileShape\_MNK,

      ElementA,

      TagToStrideA\_t<GmemLayoutATag>,

      ElementB,

      TagToStrideB\_t<GmemLayoutBTag>,

      TiledMma,

      GmemTiledCopyA,

      SmemLayoutAtomA,

      SmemCopyAtomA,

      cute::identity,

      GmemTiledCopyB,

      SmemLayoutAtomB,

      SmemCopyAtomB,

      cute::identity

    >;

在外部cutlass/example里面调用的是

using CollectiveMainloop = typename cutlass::gemm::collective::CollectiveBuilder<

    ArchTag, OperatorClass,

    ElementA, LayoutA, AlignmentA,

    ElementB, LayoutB, AlignmentB,

    ElementAccumulator,

    TileShape, ClusterShape,

    cutlass::gemm::collective::StageCountAutoCarveout<

      static\_cast<int>(sizeof(typename CollectiveEpilogue::SharedStorage))>,

    cutlass::gemm::collective::KernelScheduleAuto

  >::CollectiveOp;

所以我们要进入CollectiveMma里面查看：

// WarpSpecialized Mainloop

template <

  int Stages,

  class ClusterShape,

  class KernelSchedule,

  class TileShape\_,

  class ElementA\_,

  class StrideA\_,

  class ElementB\_,

  class StrideB\_,

  class TiledMma\_,

  class GmemTiledCopyA\_,

  class SmemLayoutAtomA\_,

  class SmemCopyAtomA\_,

  class TransformA\_,

  class GmemTiledCopyB\_,

  class SmemLayoutAtomB\_,

  class SmemCopyAtomB\_,

  class TransformB\_>

struct CollectiveMma<

    MainloopSm90TmaGmmaWarpSpecialized<Stages, ClusterShape, KernelSchedule>,

    TileShape\_,

    ElementA\_,

    StrideA\_,

    ElementB\_,

    StrideB\_,

    TiledMma\_,

    GmemTiledCopyA\_,

    SmemLayoutAtomA\_,

    SmemCopyAtomA\_,

    TransformA\_,

    GmemTiledCopyB\_,

    SmemLayoutAtomB\_,

    SmemCopyAtomB\_,

    TransformB\_>

这里面有很多个不同的文件定义了CollectiveMma，依靠的是

MainloopSm90TmaGmmaWarpSpecialized来判断是否进入当前结构体。

为了确定进入的到底是哪个collectivemainloop，在这里插入：

/// Initialize operands to be used in the GEMM and reference GEMM

void initialize(const Options &options) {

  CollectiveMainloop op;

  op.print\_template\_name();

相应地，在sm90\_mma\_tma\_gmma\_ss\_warpspecialized.hpp这里插入：

  //

  // Methods

  //

  void print\_template\_name() const {

    printf("Enter sm90\_mma\_tma\_gmma\_ss\_warpspecialized\n");

  }

然后验证得知进入的确实是sm90\_mma\_tma\_gmma\_ss\_warpspecialized。这里区别的是rs和ss。判断标准在：

// We need to handle the tuples in this function since it is used in SFINAE dispatch in the CollectiveBuilder.

// At that point, it is not guaranteed that the tuples have been split out into the required parts.

template <class MaybeTupleElementA, class LayoutA, class MaybeTupleElementB, class LayoutB>

constexpr bool

is\_use\_rmem\_A() {

  using ElementA = detail::deduce\_mixed\_width\_dtype\_t<0, MaybeTupleElementA>;

  using ElementB = detail::deduce\_mixed\_width\_dtype\_t<0, MaybeTupleElementB>;

  constexpr bool IsABDifferentWidth = cute::sizeof\_bits\_v<ElementA> != cute::sizeof\_bits\_v<ElementB>;

  constexpr bool HasScales = cute::is\_tuple<MaybeTupleElementA>::value ^ cute::is\_tuple<MaybeTupleElementB>::value;

  constexpr bool IsInputSizeTwoBytes = is\_input\_size\_two\_bytes<ElementA, ElementB>();

  constexpr bool IsLayoutAkBk = cutlass::gemm::detail::is\_k\_major\_A<LayoutA>() &&

                                cutlass::gemm::detail::is\_k\_major\_B<LayoutB>();

  constexpr bool IsUseRmemA = (!IsInputSizeTwoBytes && !IsLayoutAkBk) || IsABDifferentWidth || HasScales;

  return IsUseRmemA;

}

综合所有条件，判断是否使用寄存器内存：

* 如果输入数据类型不是 2 字节宽且布局不是 k-major，则为真。
* 如果 A 和 B 的数据类型宽度不同，则为真。
* 如果 A 或 B 包含缩放因子，则为真。

似乎rs是一种新的优化方法，不过我暂时先不管。

在example48文件里面最重要的是这一行：

using GemmKernel = cutlass::gemm::kernel::GemmUniversal<

    Shape<int,int,int>, // Indicates ProblemShape

    CollectiveMainloop,

    CollectiveEpilogue

>;

这里的GemmUniversal决定了最内层的kernel到底是怎么写的。

这些GemmUniversal广泛的存在于

sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp/sm90\_gemm\_tma\_warpspecialized\_pingpong.hpp等等之中。在每个文件开头处有判断，比如：

template <

  class ProblemShape\_,

  class CollectiveMainloop\_,

  class CollectiveEpilogue\_,

  class TileScheduler\_

>

class GemmUniversal<

  ProblemShape\_,

  CollectiveMainloop\_,

  CollectiveEpilogue\_,

  TileScheduler\_,

  cute::enable\_if\_t<cute::is\_base\_of\_v<KernelTmaWarpSpecializedCooperative, typename CollectiveMainloop\_::DispatchPolicy::Schedule>>>

{

我们这里的输入就是KernelTmaWarpSpecializedCooperative，所以就进入到了sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp，打印结果也证实了这一点。

在include/cutlass/device\_kernel.h，里面调用了operator。所以在sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp里面真正被执行的是里面的operator。

/// Generic CUTLASS kernel template.

template <typename Operator>

CUTLASS\_GLOBAL

#ifdef \_\_CUDACC\_\_

// Enclosing this in \_\_CUDACC\_\_ suppresses MSVC warnings.

\_\_launch\_bounds\_\_(Operator::MaxThreadsPerBlock, Operator::MinBlocksPerMultiprocessor)

#endif // \_\_CUDACC\_\_

void device\_kernel(CUTLASS\_GRID\_CONSTANT typename Operator::Params const params)

{

  // Dynamic shared memory base pointer

  extern \_\_shared\_\_ char smem[];

  Operator op;

  op(params, smem);

}

**研究Operator的细节**

CUTE\_DEVICE void cluster\_arrive\_relaxed()

{

#if defined(CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED)

asm volatile("barrier.cluster.arrive.relaxed.aligned;\n" : : );

#else

CUTE\_INVALID\_CONTROL\_PATH("CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED is not defined");

#endif

}

CUTE\_DEVICE void cluster\_arrive()

{

#if defined(CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED)

asm volatile("barrier.cluster.arrive.aligned;\n" : : );

#else

CUTE\_INVALID\_CONTROL\_PATH("CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED is not defined");

#endif

}

CUTE\_DEVICE void cluster\_wait()

{

#if defined(CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED)

asm volatile("barrier.cluster.wait.aligned;\n" : : );

#else

CUTE\_INVALID\_CONTROL\_PATH("CUTE\_ARCH\_CLUSTER\_SM90\_ENABLED is not defined");

#endif

}

等等，这些的区别是什么？我自己怎么用？

ChatGPT的答案：（我也将信将疑。。。。）

· 初始化阶段：在集群中的所有线程开始执行主要任务之前，需要进行一些初始化操作。cluster\_arrive\_relaxed可以在这个阶段使用，以通知其他线程已经准备好，但不必等待所有线程都到达。

· · 主循环阶段：在集群中的所有线程都需要在某个关键点同步。使用cluster\_arrive和cluster\_wait确保所有线程都到达同步点，然后再继续执行。这通常用于确保数据一致性和正确性。

    auto cluster\_wait\_fn = [] () {

      // We need this to guarantee that the Pipeline init is visible

      // To all producers and consumer thread blocks in the Cluster

      if constexpr (size(ClusterShape{}) > 1) {

        cute::cluster\_arrive\_relaxed();

        return [] () { cute::cluster\_wait(); };

      }

      else {

        \_\_syncthreads();

        return [] () {}; // do nothing

      }

    } ();

假设 ClusterShape 的大小大于1，则执行流程如下：

定义 cluster\_wait\_fn：

立即执行 cute::cluster\_arrive\_relaxed()。

返回一个包含 cute::cluster\_wait() 的 lambda 表达式。

稍后调用 cluster\_wait\_fn：

执行返回的 lambda 表达式，也就是 cute::cluster\_wait()。

**共享内存使用量是多少？**

在文件

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp里有

  struct SharedStorage {

    struct PipelineStorage : cute::aligned\_struct<16> {

      using MainloopPipelineStorage = typename CollectiveMainloop::PipelineStorage;

      using EpiLoadPipelineStorage = typename CollectiveEpilogue::PipelineStorage;

      alignas(16) MainloopPipelineStorage mainloop;

      alignas(16) EpiLoadPipelineStorage epi\_load;

      alignas(16) typename LoadWarpOrderBarrier::SharedStorage load\_order;

    } pipelines;

    struct TensorStorage : cute::aligned\_struct<128> {

      using MainloopTensorStorage = typename CollectiveMainloop::TensorStorage;

      using EpilogueTensorStorage = typename CollectiveEpilogue::TensorStorage;

      EpilogueTensorStorage epilogue;

      MainloopTensorStorage mainloop;

    } tensors;

  };

这里的尺寸由下式求出：

static constexpr int SharedStorageSize = sizeof(SharedStorage);

在operator里输出SharedStorageSize可知是196864byte。Ncu测试也一样。（注意ncu的结果是kbyte，即1e3byte，而不是1024byte，那个是kb，不等同于kbyte）

每一项详细测出来是：

**PipelineStorage**

PipelineStorageSize=96:

PipelineStorage的总大小。这部分包括主循环管道存储、EpiLoad管道存储和LoadWarpOrderBarrier的共享存储。

**TensorStorage**

TensorStorageSize=196736:

TensorStorage的总大小。这部分包括主循环张量存储和Epilogue张量存储。

* MainloopPipelineStorage

MainloopPipelineStorageSize=64:

主循环管道存储的大小。这部分存储用于在主循环（Mainloop）中存储需要在不同线程之间共享的数据。

* EpiLoadPipelineStorage

EpiLoadPipelineStorageSize=2:

EpiLoadPipelineStorage的大小。这部分存储用于在Epilogue加载阶段存储需要在不同线程之间共享的数据。

* LoadWarpOrderBarrierSharedStorage

LoadWarpOrderBarrierSharedStorageSize=16:

LoadWarpOrderBarrier的共享存储大小。这部分存储用于协调不同线程之间的加载顺序。

* MainloopTensorStorage

**MainloopTensorStorageSize=196608:**

主循环张量存储的大小。这部分存储用于在主循环（Mainloop）中存储需要在不同线程之间共享的张量数据。（192KB）（smem\_A=64KB, smem\_B=128KB）

* EpilogueTensorStorage

EpilogueTensorStorageSize=1:

Epilogue张量存储的大小。这部分存储用于在Epilogue阶段存储需要在不同线程之间共享的张量数据。

/home/zyhuang/cutlass/include/cutlass/gemm/collective/sm90\_mma\_tma\_gmma\_ss\_warpspecialized.hpp

  // Tile along modes in a way that maximizes the TMA box size.

  using SmemLayoutA = decltype(tile\_to\_shape(

      SmemLayoutAtomA{},

      make\_shape(shape<0>(TileShape{}), shape<2>(TileShape{}), Int<DispatchPolicy::Stages>{}),

      cute::conditional\_t< ::cutlass::gemm::detail::is\_major<0,StrideA>(), Step<\_2,\_1,\_3>, Step<\_1,\_2,\_3>>{}));

打印出这里的SmemLayoutA和B来检查数值。方法是将其插入到同一个文件瞎的SharedStorage结构体里，然后去

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp的operator里printf出来。得到以下结果：

TileShape=(128, 256, 64)

stages\_value=4

SmemLayoutAtomA=(8, 64)

SmemLayoutA=(128, 64, 4)

和之前得到的结果相比，结果一致：

MNK=128,256,64

A矩阵是128(M)\*64(K)\*2/1024\*4(stage\_count)=64KB

B矩阵是256(N)\*64(K)\*2/1024\*4(stage\_count)=128KB

SmemLayoutAtomA是由ss\_smem\_selector得到的，底层是GMMA::Layout\_MN\_SW128\_Atom或者类似的东西，暂时不管。

**stage\_count是怎么得到的？**

因为SmemlayoutA是由SmemLayoutAtomA组合而得：

  // Tile along modes in a way that maximizes the TMA box size.

  using SmemLayoutA = decltype(tile\_to\_shape(

      SmemLayoutAtomA{},

      make\_shape(shape<0>(TileShape{}), shape<2>(TileShape{}), Int<DispatchPolicy::Stages>{}),

      cute::conditional\_t< ::cutlass::gemm::detail::is\_major<0,StrideA>(), Step<\_2,\_1,\_3>, Step<\_1,\_2,\_3>>{}));

所以需要上溯stage\_count的由来。可见我们需要的是DispatchPolicy::Stages，那么需要上溯DispatchPolicy。using DispatchPolicy = MainloopSm90TmaGmmaWarpSpecialized<Stages, ClusterShape, KernelSchedule>;让我们上溯MainloopSm90TmaGmmaWarpSpecialized

由此上溯

/home/zyhuang/cutlass/include/cutlass/gemm/collective/builders/sm90\_gmma\_builder.inl。

在此文件下找到

  using DispatchPolicy = cute::conditional\_t<IsArrayOfPointersGemm,

      MainloopSm90ArrayTmaGmmaWarpSpecialized<PipelineStages, ClusterShape\_MNK, KernelScheduleType>,

      /\* For FP8 use a separate mainloop compared to other datatypes \*/

      cute::conditional\_t<IsFP8Input,

          MainloopSm90TmaGmmaWarpSpecializedFP8<PipelineStages, ClusterShape\_MNK, KernelScheduleType>,

          MainloopSm90TmaGmmaWarpSpecialized<PipelineStages, ClusterShape\_MNK, KernelScheduleType>>>;

也就是找到PipelineStages，也就是

  static constexpr int PipelineStages = detail::compute\_stage\_count\_or\_override<detail::sm90\_smem\_capacity\_bytes,

      ElementAMma, ElementBMma, TileShape\_MNK>(StageCountType{});

compute\_stage\_count\_or\_override定义在同一个路径下，有三个override函数，区别在于输入参数，这里我们用到：

// Returns the maximum number of smem tiles that can be used with a given smem capacity, or overrides with manual count.

template<int CapacityBytes, class ElementA, class ElementB, class TileShapeMNK, int carveout\_bytes>

constexpr int

compute\_stage\_count\_or\_override(StageCountAutoCarveout<carveout\_bytes> stage\_count) {

  constexpr auto mainloop\_pipeline\_bytes = sizeof(typename cutlass::PipelineTmaAsync<1>::SharedStorage);

  constexpr auto a\_bits = cute::sizeof\_bits\_v<ElementA>;

  constexpr auto b\_bits = cute::sizeof\_bits\_v<ElementB>;

  constexpr int stage\_bytes =

    cutlass::bits\_to\_bytes(a\_bits \* size<0>(TileShapeMNK{}) \* size<2>(TileShapeMNK{})) +

    cutlass::bits\_to\_bytes(b\_bits \* size<1>(TileShapeMNK{}) \* size<2>(TileShapeMNK{})) +

    static\_cast<int>(mainloop\_pipeline\_bytes);

  return (CapacityBytes - carveout\_bytes) / stage\_bytes;

}

在operator里打印出计算需要的参数，分别是：

mainloop\_pipeline\_bytes: 16 bytes

a\_bits: 16 bits

b\_bits: 16 bits

TileShapeMNK = {128, 256, 64}

carveout\_bytes: 1 byte

a\_bits \* size<0>(TileShapeMNK{}) \* size<2>(TileShapeMNK{})

= 16 bits \* 128 \* 64

= 16 \* 128 \* 64

= 131072 bits=131072 / 8 = 16384 bytes

b\_bits \* size<1>(TileShapeMNK{}) \* size<2>(TileShapeMNK{})

= 16 bits \* 256 \* 64

= 16 \* 256 \* 64

= 262144 bits=262144 / 8 = 32768 bytes

stage\_bytes = 16384 + 32768 + 16

= 49168 bytes

return (232448 - 1) / 49168

= 232447 / 49168

≈ 4.728 所以取4。

1. **Setmaxnreg的特性**

the number of register is determined by:

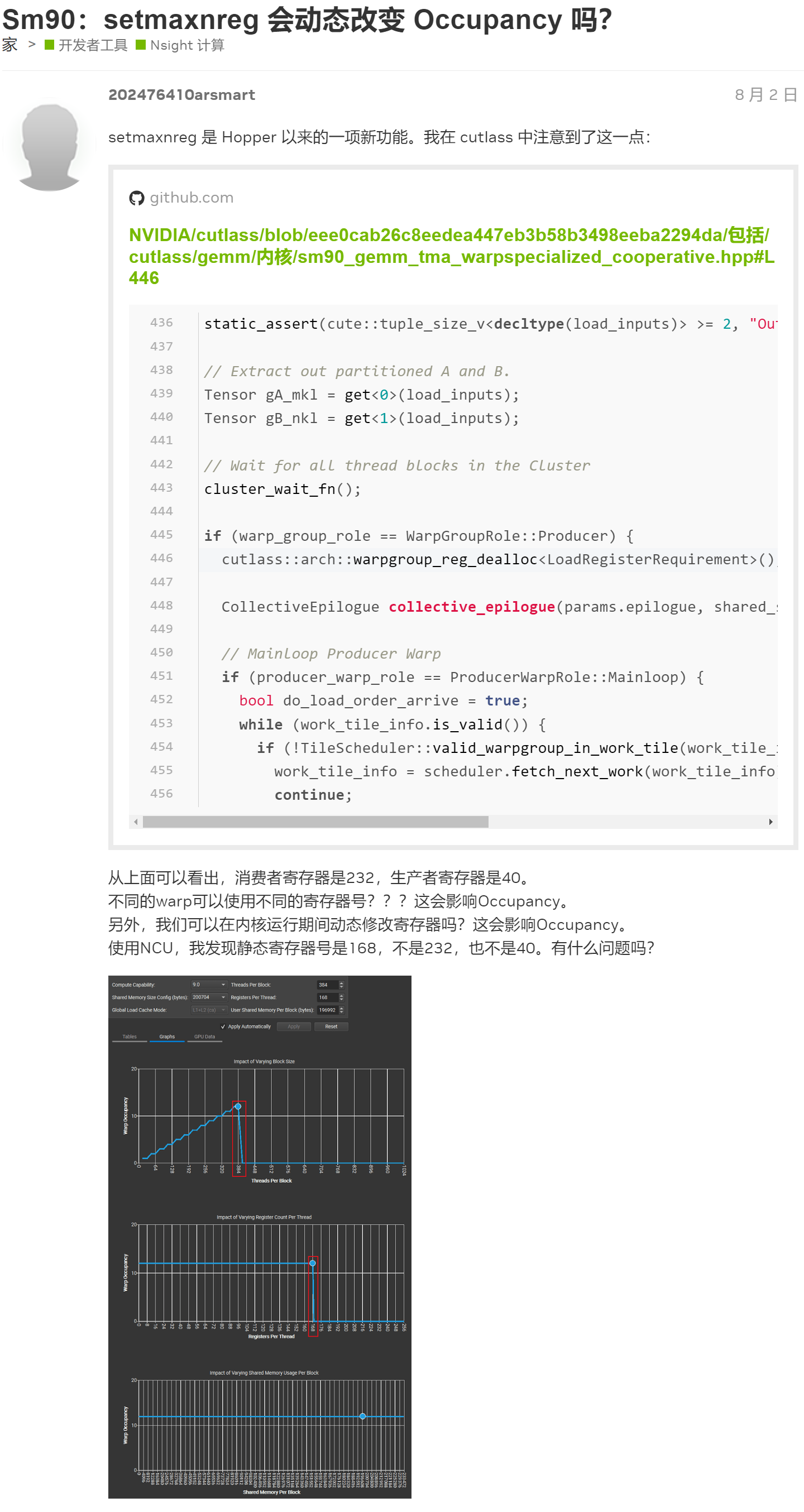
code shared by different warp (A registers per thread)

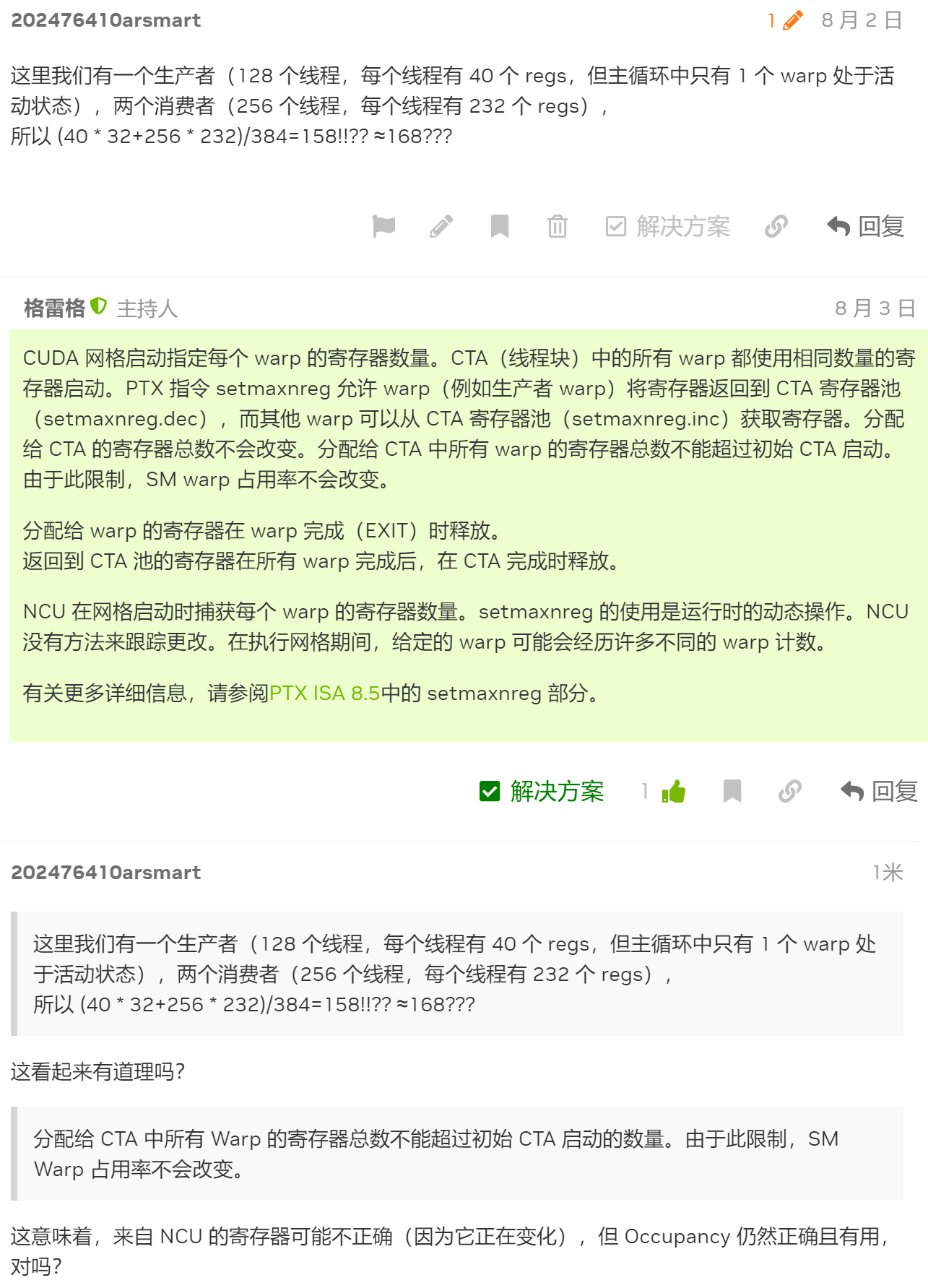
code snippet for warp(i, i+1…, i+K) (B registers per thread)------code snippet for warp(j, j+1, …, j+T)(C registers per thread)

In part 2, the register is num\_2 = (KB+CT)/(B+C)

So overall register number is max(A, num\_2)

如何获知producer和consumer的寄存器数量？可以用ncu的source code来看。找到最大寄存器数量即可。（有时间亲自试试看！）





1. **TileSchedulerSelector是怎么被调用的？**

首先是我们跑example48，从

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_gemm\_warpspecialized.hpp里看到scheduler.fetch\_next\_work，然后去找fetch\_next\_work，在include/cutlass/gemm/kernel/下面有三个tilescheduler文件：

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_tile\_scheduler.hpp

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_tile\_scheduler\_stream\_k.hpp

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_tile\_scheduler\_group.hpp

在其中都有fetch\_next\_work，在每个里面加打印函数，最后发现是进入了sm90\_tile\_scheduler。

所以去寻找该文件的类：class PersistentTileSchedulerSm90。搜索代码发现：

// Default (void) for Sm90 maps to PersistentTileSchedulerSm90

template <

  class ArchTag,

  class TileShape,

  class ClusterShape

>

struct TileSchedulerSelector<

  void,

  ArchTag,

  TileShape,

  ClusterShape

  > {

  using Scheduler = typename TileSchedulerSelector<

    PersistentScheduler,

    ArchTag,

    TileShape,

    ClusterShape

  >::Scheduler;

};

这里最关键的是TileSchedulerSelector后面紧跟着的是void。然后去寻找TileSchedulerSelector。

在include/cutlass/gemm/kernel/sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp里找到了

using TileScheduler = typename detail::TileSchedulerSelector<

    TileScheduler\_, ArchTag, TileShape, ClusterShape>::Scheduler;

再上溯去找TileScheduler\_

template <

  class ProblemShape\_,

  class CollectiveMainloop\_,

  class CollectiveEpilogue\_,

  class TileScheduler\_

>

class GemmUniversal<

  ProblemShape\_,

  CollectiveMainloop\_,

  CollectiveEpilogue\_,

  TileScheduler\_,

  cute::enable\_if\_t<cute::is\_base\_of\_v<KernelTmaWarpSpecializedCooperative, typename CollectiveMainloop\_::DispatchPolicy::Schedule>>>

这一点，在

/home/zyhuang/cutlass/examples/48\_hopper\_warp\_specialized\_gemm/48\_hopper\_warp\_specialized\_gemm.cu里没有直接体现：

using GemmKernel = cutlass::gemm::kernel::GemmUniversal<

    Shape<int,int,int>, // Indicates ProblemShape

    CollectiveMainloop,

    CollectiveEpilogue

>;

缺失了。然后去找GemmUniversal的定义函数：

/home/zyhuang/cutlass/include/cutlass/gemm/kernel/gemm\_universal\_decl.h

template <

  class ProblemShapeOrThreadblockMma\_, // (m, n, k) or (m, n, k, l)

  class CollectiveMainloopOrEpilogue\_,

  class CollectiveEpilogueOrThreadblockSwizzle\_,

  class TileScheduler\_ = void,

  class Enable = void

>

class GemmUniversal;

得知如果不写，就默认是void。然后最终使用PersistentTileSchedulerSm90。

1. **Sm90上的block执行顺序是什么？**

首先sm90必然使用persistent策略。所以一个CTA就会计算多个128\*256的block位置。到底计算哪些？

其次，每次都是以cluster为集体来计算的。一个cluster是怎么排布的？

首先由41确定下来，我们用的是PersistentTileSchedulerSm90，位于/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_tile\_scheduler.hpp

然后这里面计算当前cluster/CTA具体负责的block的函数是get\_work\_idx\_m\_and\_n，这将被/home/zyhuang/cutlass/include/cutlass/gemm/kernel/static\_tile\_scheduler.hpp调用。每次在operator里scheduler.fetch\_next\_work的时候，底层都会调用到get\_work\_idx\_m\_and\_n。要调整每个cluster/CTA的计算位置，也只需要修改get\_work\_idx\_m\_and\_n即可。

一个测试结果是，在代码中，限定blockIdx.x=.y=0时，输出当前CTA负责的block\_coord：

|  |  |  |
| --- | --- | --- |
| blk\_M | blk\_N | blk\_K |
| 128 | 256 | 64 |
| cluster0 | cluster1 | cluster2 |
| 1 | 2 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4096-4096-256 |  |  |  |
| M | N | M\*16+N | 上下差值 |
| 0 | 0 | 0 |  |
| 8 | 4 | 132 | 132 |
| 16 | 8 | 264 | 132 |
| 24 | 12 | 396 | 132 |

（16是4096/256）

|  |  |  |  |
| --- | --- | --- | --- |
| 4096-8192-256 |  |  |  |
| M | N | M\*32+N | 上下差值 |
| 0 | 0 | 0 |  |
| 4 | 4 | 132 | 132 |
| 8 | 8 | 264 | 132 |
| 12 | 12 | 396 | 132 |
| 16 | 16 | 528 | 132 |
| 20 | 20 | 660 | 132 |
| 24 | 24 | 792 | 132 |
| 28 | 28 | 924 | 132 |

（32是8192/256）

暂未破译132是什么。。。后来查明了，在

cutlass/include/cutlass/gemm/kernel/static\_tile\_scheduler.hpp的advance\_to\_next\_work，每次block的增量是total\_block\_size，也就是gridDim.x\*gridDim.y\*gridDim.z。所以这里如果是132，那就是cluster\_size=1或者2，就是总共能发射132个block（因为SM也是132个）。不过我测过，当cluster\_size=4，最大block\_per\_wave是116，cluster\_size=16, max\_block\_per\_wave是112。

当我们深入/home/zyhuang/cutlass/include/cutlass/gemm/kernel/static\_tile\_scheduler.hpp，会看到scheduler\_params后面附带很多参数，不知道具体是在哪里定义的，下面去寻找具体定义的位置。

    auto [work\_idx\_m, work\_idx\_n] = Subclass::get\_work\_idx\_m\_and\_n(blk\_per\_grid\_dim, scheduler\_params.divmod\_cluster\_shape\_major\_, scheduler\_params.divmod\_cluster\_shape\_minor\_, scheduler\_params.divmod\_cluster\_blk\_major\_, scheduler\_params.log\_swizzle\_size\_, scheduler\_params.raster\_order\_);

在/home/zyhuang/cutlass/include/cutlass/gemm/kernel/sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp里面有：

  // Convert to underlying arguments. In this case, a simple copy for the aliased type.

  static

  Params

  to\_underlying\_arguments(Arguments const& args, void\* workspace) {

    CUTLASS\_TRACE\_HOST("to\_underlying\_arguments():");

    auto problem\_shape = args.problem\_shape;

    if constexpr (detail::Has\_SwapAB\_v<CollectiveMainloop>) {

      // swap M/N

      get<0>(problem\_shape) = get<1>(args.problem\_shape);

      get<1>(problem\_shape) = get<0>(args.problem\_shape);

    }

    auto problem\_shape\_MNKL = append<4>(problem\_shape, 1);

    // Get SM count if needed, otherwise use user supplied SM count

    int sm\_count = args.hw\_info.sm\_count;

    if (sm\_count <= 0) {

      CUTLASS\_TRACE\_HOST("  WARNING: Arguments do not include a valid SM count.\n"

          "  For optimal performance, populate the arguments KernelHardwareInfo struct with the SM count.");

      sm\_count = KernelHardwareInfo::query\_device\_multiprocessor\_count(args.hw\_info.device\_id);

    }

    CUTLASS\_TRACE\_HOST("to\_underlying\_arguments(): Setting persistent grid SM count to " << sm\_count);

    KernelHardwareInfo hw\_info{args.hw\_info.device\_id, sm\_count};

    // Calculate workspace pointers

    uint8\_t\* workspace\_ptr = reinterpret\_cast<uint8\_t\*>(workspace);

    size\_t workspace\_offset = 0;

    void\* scheduler\_workspace = workspace\_ptr;

    workspace\_offset += TileScheduler::template get\_workspace\_size<ProblemShape, ElementAccumulator>(

      args.scheduler, args.problem\_shape, args.hw\_info, NumMmaWarpGroups);

    workspace\_offset = round\_nearest(workspace\_offset,  MinWorkspaceAlignment);

    void\* epilogue\_workspace = workspace\_ptr + workspace\_offset;

    workspace\_offset += CollectiveEpilogue::get\_workspace\_size(args.problem\_shape, args.epilogue);

    workspace\_offset = round\_nearest(workspace\_offset,  MinWorkspaceAlignment);

    void\* mainloop\_workspace = nullptr;

    // Precompute the sub tiles numbers in epilogue, pass into tile scheduler.  Therefore it will be used

    // in separate reduction scheme for streamk case, NumEpilogueSubTiles default value is 1, which means

    // subtile will not be used, therefore separate reduction will not be enabled.

    constexpr uint32\_t NumEpilogueSubTiles = CollectiveEpilogue::get\_store\_pipe\_increment(TileShape{});

    TileSchedulerParams scheduler = TileScheduler::to\_underlying\_arguments(

      problem\_shape\_MNKL, TileShape{}, ClusterShape{}, hw\_info, args.scheduler, scheduler\_workspace, NumEpilogueSubTiles);

    return {

      args.mode,

      problem\_shape,

      CollectiveMainloop::to\_underlying\_arguments(args.problem\_shape, args.mainloop, mainloop\_workspace),

      CollectiveEpilogue::to\_underlying\_arguments(args.problem\_shape, args.epilogue, epilogue\_workspace),

      hw\_info,

      scheduler,

      workspace

    };

  }

注意这里：

    TileSchedulerParams scheduler = TileScheduler::to\_underlying\_arguments(

      problem\_shape\_MNKL, TileShape{}, ClusterShape{}, hw\_info, args.scheduler, scheduler\_workspace, NumEpilogueSubTiles);

TileScheduler::to\_underlying\_arguments是cutlass/include/cutlass/gemm/kernel/static\_tile\_scheduler.hpp里面的：

  template <class ProblemShapeMNKL, class TileShape, class ClusterShape>

  static Params

  to\_underlying\_arguments(

    ProblemShapeMNKL problem\_shape\_mnkl,

    TileShape tile\_shape,

    ClusterShape cluster\_shape,

    [[maybe\_unused]] KernelHardwareInfo const& hw\_info,

    Arguments const& arguments,

    [[maybe\_unused]] void\* workspace=nullptr,

    [[maybe\_unused]] const uint32\_t epilogue\_subtile = 1) {

    // We only need the tile and cluster shape during scheduler setup, so let FTAD do the magic

    static\_assert(cute::is\_static<TileShape>::value);

    static\_assert(cute::is\_static<ClusterShape>::value);

    dim3 problem\_blocks = get\_tiled\_cta\_shape\_mnl(problem\_shape\_mnkl, tile\_shape, cluster\_shape);

    Params params;

    params.initialize(

      problem\_blocks,

      to\_gemm\_coord(cluster\_shape),

      hw\_info,

      arguments.max\_swizzle\_size,

      arguments.raster\_order

    );

    return params;

  }

注意这里的Params是PersistentTileSchedulerSm90Params，这定义在：cutlass/include/cutlass/gemm/kernel/tile\_scheduler\_params.h

这里面定义了很多很多东西。这就完全找到最底层了。

      int cta\_per\_device = sm\_count;

      /\*

      \* Optimal grid size calculation is based on

      \* GH100: 8 GPCs, 72 TPCs (9 TPCs/GPC), 2 SMs/TPC, 144 SMs per full GPU

      \* Hence, maximum SMs per GPC = 18

      \*/

      constexpr int max\_sm\_per\_gpc = 18;

      // Provided SM count could possibly be less than the assumed maximum SMs per GPC

      auto cluster\_size = cluster\_shape.m() \* cluster\_shape.n();

      int const min\_num\_gpc = sm\_count < max\_sm\_per\_gpc ? 1 : sm\_count / max\_sm\_per\_gpc;

      int const max\_cta\_occupancy\_per\_gpc = max\_sm\_per\_gpc - (max\_sm\_per\_gpc % cluster\_size);

      cta\_per\_device = min\_num\_gpc \* max\_cta\_occupancy\_per\_gpc;

回到block的执行顺序上。在cutlass/include/cutlass/gemm/kernel/sm90\_tile\_scheduler.hpp里这个函数是开始：

  // Kernel helper function to get next work tile

  CUTLASS\_DEVICE

  auto

  fetch\_next\_work(WorkTileInfo work\_tile\_info) {

    if (continue\_current\_work(work\_tile\_info)) {

      return work\_tile\_info; // 在static\_tile\_scheduler里定义了continue\_current\_work函数必然返回false。所以是绝对不会进入这里的。

    }

    advance\_to\_next\_work();

    return get\_current\_work();

  }

也就是advance\_to\_next\_work()和get\_current\_work()。

advance\_to\_next\_work()是对current\_work\_linear\_idx\_（private变量）每次进行增加一个wave的block数量（假设是112个）。

get\_current\_work()是调用get\_work\_idx\_m\_and\_n，对current\_work\_linear\_idx\_进行计算，比如假如是100，那么就除以problem\_shape<1>，比如是横向有15个block每行，那么第100位置对应的就是第六行的第十个位置。

然后下次advance\_to\_next\_work()再增加到212，也就是14行第二列。

直到https://github.com/NVIDIA/cutlass/blob/7192f4ab230bb721fa8d4d3df33886dbe86cdc59/include/cutlass/gemm/kernel/static\_tile\_scheduler.hpp#L169-L172

  get\_current\_work\_for\_linear\_idx(uint64\_t linear\_idx) const {

    if (linear\_idx >= scheduler\_params.blocks\_per\_problem\_) {

      return WorkTileInfo::invalid\_work\_tile();

    }

这里结束外层的循环https://github.com/NVIDIA/cutlass/blob/7192f4ab230bb721fa8d4d3df33886dbe86cdc59/include/cutlass/gemm/kernel/sm90\_gemm\_tma\_warpspecialized\_cooperative.hpp#L453。

注意现在因为我只改写了raster=N的情况。所以要想跑通，得写全，类似于：

./48\_hopper\_warp\_specialized\_gemm --m=4096 --n=20480 --k=5120 --raster=N

1. **计算结果是怎么存出到global的？**

首先看到在producer里面有epilogue。怀疑是不是在这里存出。但是第一，producer并不拥有每次的计算结果。其次是每个warp\_group算完后就应该存出去，因为共享内存也没有分配地方来存这些中间结果（也没必要）。所以存出不可能放在producer。

其次看到consumer里面有epilogue\_store，注明了是存到global。

        if (TileScheduler::compute\_epilogue(work\_tile\_info, params.scheduler)) {

          // Epilogue and write to gD

          auto [epi\_load\_pipe\_consumer\_state\_next, epi\_store\_pipe\_producer\_state\_next] =

          collective\_epilogue.store(

            epi\_load\_pipeline,

            epi\_load\_pipe\_consumer\_state,

            epi\_store\_pipeline,

            epi\_store\_pipe\_producer\_state,

            problem\_shape\_MNKL,

            blk\_shape,

            blk\_coord,

            accumulators,

            tiled\_mma,

            mma\_thread\_idx,

            shared\_storage.tensors.epilogue,

            work\_tile\_info.reduction\_subtile\_idx()

          );

经过打印证实，确实进入了这里面。collective\_epilogue上溯到源头，来自于example cu文件里面的：

using CollectiveEpilogue = typename cutlass::epilogue::collective::CollectiveBuilder<

    cutlass::arch::Sm90, cutlass::arch::OpClassTensorOp,

    TileShape, ClusterShape,

    cutlass::epilogue::collective::EpilogueTileAuto,

    ElementAccumulator, ElementAccumulator,

    ElementC, LayoutC, AlignmentC,

    ElementC, LayoutC, AlignmentC,

    cutlass::epilogue::collective::EpilogueScheduleAuto

  >::CollectiveOp;

核心在EpilogueScheduleAuto。

在cutlass/include/cutlass/epilogue/collective/builders/sm90\_builder.inl里面找到了EpilogueScheduleAuto。

最后追溯到：

  using CollectiveOp = cute::conditional\_t<

    cute::is\_same\_v<Schedule, NoSmemWarpSpecialized>,

    cutlass::epilogue::collective::detail::Sm90TmaWarpSpecializedAdapter<

      cutlass::epilogue::collective::DefaultEpilogue<

        cutlass::detail::TagToStrideC\_t<GmemLayoutTagC>,

        cutlass::detail::TagToStrideC\_t<GmemLayoutTagD>,

        ThreadOp,

        cutlass::gemm::EpilogueDefault>>,

    // Epilogue for Ptr-Array and Grouped Gemm

    cutlass::epilogue::collective::detail::Sm90TmaWarpSpecializedAdapter<

      cutlass::epilogue::collective::DefaultEpilogueArray<

        cutlass::detail::TagToStrideC\_t<GmemLayoutTagC>,

        cutlass::detail::TagToStrideC\_t<GmemLayoutTagD>,

        ThreadOp,

        Schedule>>

    >; // 一开始定义的是auto，然后里面确定会使用NoSmemWarpSpecialized，所以这里就会选择DefaultEpilogue（第一个）

也就是cutlass/include/cutlass/epilogue/collective/default\_epilogue.hpp里面的operator：

  template<

    class ProblemShapeMNKL,

    class BlockShapeMNK,

    class BlockCoordMNKL,

    class FrgEngine, class FrgLayout,

    class TiledMma,

    class ResidueMNK

  >

  CUTLASS\_HOST\_DEVICE void

  operator()(

      ProblemShapeMNKL problem\_shape\_mnkl,

      BlockShapeMNK blk\_shape\_MNK,

      BlockCoordMNKL blk\_coord\_mnkl,

      cute::Tensor<FrgEngine, FrgLayout> const& accumulators,

      TiledMma tiled\_mma,

      ResidueMNK residue\_mnk,

      int thread\_idx,

      [[maybe\_unused]] char\* smem\_buf)

  {

    using namespace cute;

    using X = Underscore;

    static\_assert(cute::rank(ProblemShapeMNKL{}) == 4, "ProblemShapeMNKL must be rank 4");

    static\_assert(is\_static<BlockShapeMNK>::value, "ThreadBlock tile shape must be static");

    static\_assert(cute::rank(BlockShapeMNK{}) == 3, "BlockShapeMNK must be rank 3");

    static\_assert(cute::rank(BlockCoordMNKL{}) == 4, "BlockCoordMNKL must be rank 3");

    // Separate out problem shape for convenience

    auto M = get<0>(problem\_shape\_mnkl);

    auto N = get<1>(problem\_shape\_mnkl);

    auto L = get<3>(problem\_shape\_mnkl);

    auto stride\_c = detail::get\_epilogue\_stride<EpilogueSchedule>(params.dC);

    auto stride\_d = detail::get\_epilogue\_stride<EpilogueSchedule>(params.dD);

    // Represent the full output tensor

    Tensor mC\_mnl = make\_tensor(make\_gmem\_ptr(params.ptr\_C), make\_shape(M,N,L), stride\_c);                 // (m,n,l)

    Tensor mD\_mnl = make\_tensor(make\_gmem\_ptr(params.ptr\_D), make\_shape(M,N,L), stride\_d);                 // (m,n,l)

    Tensor gC\_mnl = local\_tile(mC\_mnl, blk\_shape\_MNK, make\_coord(\_,\_,\_), Step<\_1,\_1, X>{});    // (BLK\_M,BLK\_N,m,n,l)

    Tensor gD\_mnl = local\_tile(mD\_mnl, blk\_shape\_MNK, make\_coord(\_,\_,\_), Step<\_1,\_1, X>{});    // (BLK\_M,BLK\_N,m,n,l)

    // Slice to get the tile this CTA is responsible for

    auto [m\_coord, n\_coord, k\_coord, l\_coord] = blk\_coord\_mnkl;

    Tensor gC = gC\_mnl(\_,\_,m\_coord,n\_coord,l\_coord);                                                 // (BLK\_M,BLK\_N)

    Tensor gD = gD\_mnl(\_,\_,m\_coord,n\_coord,l\_coord);                                                 // (BLK\_M,BLK\_N)

    // Partition source and destination tiles to match the accumulator partitioning

    auto thr\_mma = tiled\_mma.get\_thread\_slice(thread\_idx);

    Tensor tCgD = thr\_mma.partition\_C(gD);                                       // (VEC,THR\_M,THR\_N)

    Tensor tCgC = thr\_mma.partition\_C(gC);                                       // (VEC,THR\_M,THR\_N)

    static\_assert(is\_static<FrgLayout>::value, "Accumulator layout must be static");

    CUTE\_STATIC\_ASSERT\_V(size(tCgC) == size(tCgD),

        "Source and destination must have the same number of elements.");

    CUTE\_STATIC\_ASSERT\_V(size(tCgD) == size(accumulators),

        "Accumulator count must have the same destination element count.");

    // Make an identity coordinate tensor for predicating our output MN tile

    auto cD = make\_identity\_tensor(make\_shape(unwrap(shape<0>(gD)), unwrap(shape<1>(gD))));

    Tensor tCcD = thr\_mma.partition\_C(cD);

    // source is needed

    if (epilogue\_op.is\_source\_needed()) {

      CUTLASS\_PRAGMA\_UNROLL

      for (int i = 0; i < size(accumulators); ++i) {

        if (elem\_less(tCcD(i), make\_coord(get<0>(residue\_mnk), get<1>(residue\_mnk)))) {

          tCgD(i) = epilogue\_op(accumulators(i), tCgC(i));

        }

      }

    }

    // source is not needed, avoid load

    else {

      CUTLASS\_PRAGMA\_UNROLL

      for (int i = 0; i < size(accumulators); ++i) {

        if (elem\_less(tCcD(i), make\_coord(get<0>(residue\_mnk), get<1>(residue\_mnk)))) {

          tCgD(i) = epilogue\_op(accumulators(i));

        }

      }

    }

  }

在这里，直接将值从寄存器写到global。一来没经过共享内存，二来没有使用TMA。以后可以换其他epilogue试试看。

1. **sm90\_epilogue\_tma\_warpspecialized是怎么使用的？**

（背景：为了将计算的中间结果存储到共享内存，所以要使用default之外的epilogue。这就是TMA，而且要修改，以避免使用reuseC，reuseSMEM，因为我们需要同时保存所有的C计算结果在共享内存。。。反之倒也可以，但是对后续计算就会引入很多麻烦。。以后再细想吧）

cutlass/include/cutlass/epilogue/collective/sm90\_epilogue\_tma\_warpspecialized.hpp

这是最底层的执行文件。Class叫CollectiveEpilogue。上溯得到

cutlass/include/cutlass/epilogue/collective/builders/sm90\_builder.inl

struct Sm90TmaBuilderImpl {

using CollectiveOp = cutlass::epilogue::collective::CollectiveEpilogue

然后同一个文件上溯得到

struct CollectiveBuilder<

typename detail::Sm90TmaBuilderImpl<

而这里的CollectiveBuilder是很多类，比如TMA，auto，default的定义位置。

1. **Consumer-epilogue里是如何accumulator的reg-->smem的**

代码内容在：

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L510-L820>

1. Tensor Core 计算结果存储在寄存器中

Tensor Core 计算的结果通常存储在寄存器中，并且在这段代码中使用了 accumulators 表示寄存器中的数据。

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L574>

    Tensor tRS\_rAcc = thread\_r2s.retile\_S(accumulators); // ((R2S,R2S\_V),MMA\_M,MMA\_N)

这行代码将 accumulators 张量映射到线程级别的 tRS\_rAcc，表示每个线程对应的寄存器数据。

2. 处理寄存器中的数据

接下来，这段代码创建了一个向量化的视图，以便对寄存器中的数据进行批处理：

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L586-L589>

    Tensor tRS\_rAcc\_frg = recast<Array<ElementAccumulator, FragmentSize>>(tRS\_rAcc);

    // Vectorized fragment view

    constexpr int FragmentSize = DispatchPolicy::FragmentSize;

    Tensor tRS\_rAcc\_frg = recast<Array<ElementAccumulator, FragmentSize>>(tRS\_rAcc);

    Tensor tRS\_rD\_frg   = recast<Array<SmemElementD      , FragmentSize>>(tRS\_rD);

tRS\_rAcc\_frg 是寄存器中的向量化片段视图。

tRS\_rD\_frg 是目标共享内存的向量化片段视图。

3. 将寄存器中的数据处理并存储到共享内存

下面这部分代码处理了寄存器中的数据，并将其写入共享内存：

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L769C1-L771C10>

        Tensor tRS\_rAcc\_frg\_mn = tRS\_rAcc\_frg(\_,mma\_m,mma\_n);

        CUTLASS\_PRAGMA\_UNROLL

        for (int epi\_v = 0; epi\_v < size(tRS\_rCompute\_frg); ++epi\_v) {

          tRS\_rCompute\_frg(epi\_v) = cst\_callbacks.visit(tRS\_rAcc\_frg\_mn(r2s\_v + epi\_v), epi\_v, epi\_m, epi\_n);

        }

tRS\_rAcc\_frg是tRS\_rAcc的切片。要是写入了tRS\_rAcc\_frg，也就等于tRS\_rAcc有了内容。tRS\_rAcc\_frg\_mn又是tRS\_rAcc\_frg的切片。然后把tRS\_rAcc\_frg\_mn写入到tRS\_rCompute\_frg。（这都是reg到reg）

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L787C1-L790C10>

        CUTLASS\_PRAGMA\_UNROLL

        for (int i = 0; i < size(tRS\_rD\_frg); ++i) {

          tRS\_rD\_frg(i) = cutlass::NumericArrayConverter<SmemElementD, RegisterElementD, FragmentSize>{}(tRS\_rCompute\_frg(i));

        }

然后再从tRS\_rCompute\_frg写到tRS\_rD\_frg（这都是reg到reg）

cst\_callbacks.visit: 这个回调函数在这里处理了寄存器中的数据，并存储到 tRS\_rCompute\_frg 中。

tRS\_rD\_frg(i) = ...: 这一行代码将处理过的结果从 tRS\_rCompute\_frg 复制到共享内存视图 tRS\_rD\_frg 中，即将结果写入共享内存的 tRS\_rD。

4. 从共享内存复制到全局内存

接下来的代码负责将共享内存中的数据存储到全局内存（D 矩阵）中。

<https://github.com/NVIDIA/cutlass/blob/4e5a8f6853817e6595189e712a8018e1b71e4380/include/cutlass/epilogue/collective/sm90_epilogue_tma_warpspecialized.hpp#L792C1-L795C10>

        // Copy tile from register to smem

        if constexpr (is\_destination\_supported) {

          copy(tiled\_r2s, tRS\_rD, tRS\_sD(\_,\_,\_,store\_pipe\_producer\_state.index()));

        }

上面写来写去，就是写到了tRS\_rD，所以就可以写到SMEM里去。

1. **当beta=0，如何避免C矩阵浪费共享内存空间？**

tensor core计算得到的accumulator结果是放在reg的。我需要将其存到SMEM里。但是要注意swizzle，避免bank conflict。但是原先example48存出的策略，是直接从reg到global并且没有用TMA。后来找到example49，使用了TMA，将中间结果存到SMEM，再存出到global。但是问题在于，明明我的beta等于0，但是还是留给C和D各一份共享内存空间，对C来说是完全没用的。仔细寻找之后发现是cutlass的疏漏，修改stagesC=0，节省了C的共享内存空间。

此外，我还修改让reuseC，reuseSMEM为0。（有点强行。。）