总 评 成 绩	

《高性能计算》实验报告册

学年学期： 2023 - 2024 学年第 2 学期

学生姓名： 高星杰、周松华

专业年级： 计算机科学与技术 2021 级

华中农业大学信息学院

2024 年 5 月

实 验 报 告 册

<table>
<tr><td>实验名称</td><td>基于 CUDA 实现二维卷积</td><td rowspan="2">实 验
成 绩</td><td rowspan="2"></td></tr>
<tr><td>实验日期</td><td>2023 年 5 月 14 日</td></tr>
</table>

一、实验目的（描述本实验的学习目的及你对本实验的学习预期。）

1、 了解 CUDA 并行程序的设计思想和原则。

2、 了解二维卷积的基本操作。

3、 学会用 CUDA 实现卷积操作。

4、 了解 CUDA 编程和 CPU 编程的联系与区别。

二、实验环境（请描述本实验教学活动所使用的实际环境。）

1. CPU：Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
2. GPU：Tesla V100-PCIE-32GB
3. OS：Ubuntu-18.04.1
4. GPU Driver：440.64.00
5. CUDA Version：10.2
6. cuDNN Version：v9.0.176

三、实验任务

用 CUDA 实现一个二维卷积网络的操作，并且和 CPU 下完成的卷积操作进行对比。

四、实验内容与过程

1. 实验原理

二维卷积计算原理:

我们采用是深度学习中卷积的定义(互相关 Cross-Correlation)

公式表示如下

$$H(i,j) = \sum_m \sum_n F(m,n)G(i+m,j+n)$$

其中我们每个H(i,j)之间的计算互不相关，所以我们对此可以采用并行计算的优化办法。我们每一线程可以计算一个H(i,j)最后就可以得出完整结果，但是其中仍然有一些可以优化的地方，以提高程序的性能。

2. 卷积算子优化的过程

为了详细的说明实验过程中我们对并行程序优化的过程，我们会逐个版本进行介绍，每个版本会在前一个版本的基础上进行优化。同时为了确定优化的效果，每次都会与上一版做出比较，取最优的算法与下一个版本进行比较（第一个版本首先与 cudnn 中自带的卷积操作进行比较）。

为了方便下面的分析，这里先给出用到的符号及其含义

符号	含义
N	batch size
C	channel number
H	数据高
W	数据宽
K	卷积核数量
R	卷积核高

S	卷积核宽
U	卷积在高方向上的步长
V	卷积在宽方向上的步长
P	卷积在高方向上的补边
Q	卷积在宽方向上的补边
Oh	卷积结果高
Ow	卷积结果宽

(1) 版本 0：基础卷积核实现

此版本使用的是全局内存的方式实现，每个线程块负责并行处理 m 和 n 维度循环，线程内执行 K 维度循环，并且我们为了方便，我们可以将二维数组存储的数据转换为一维数组。

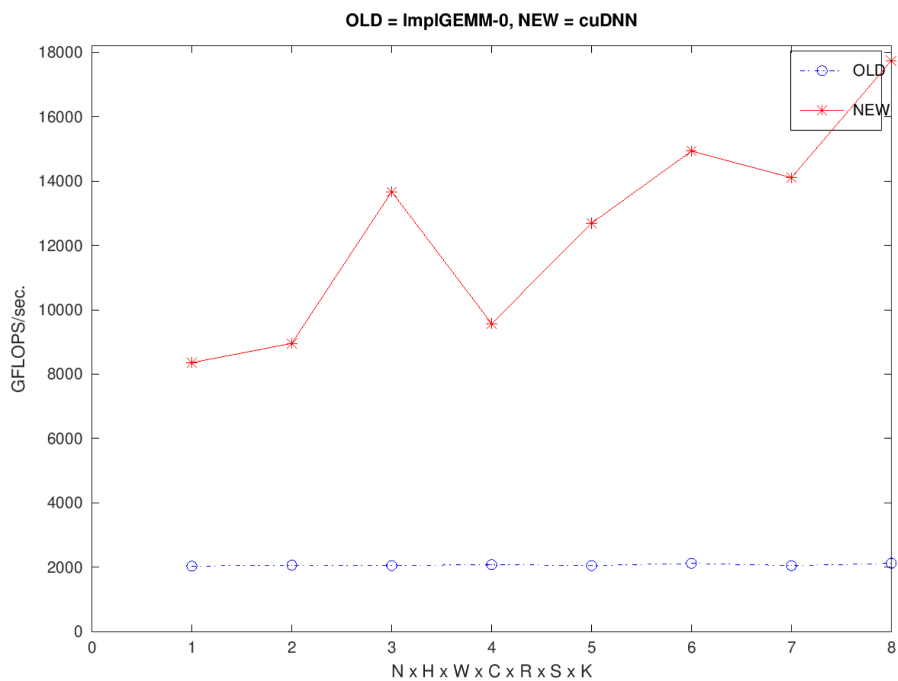
```
_global__ void conv(int M, int N, int K, float *A, float *B, float *C){
  int m = blockIdx.x * 16 + threadIdx.x;
  int n = blockIdx.y * 16 + threadIdx.y;
  if (m < M && n < N){
    float sum = 0.0;
    for (int i = 0; i < k; ++i){
      sum += A[m * K + i] * B[i * N + n];
    }
    C[m * N + n] = sum;
  }
}
int main(){
  dim3 block(16, 16);
  dim3 grid((M + 16 - 1) / 16, (N + 16 - 1) / 16);
  conv<<<grid, block>>>(M, N, K, A, B, C)
}
```

这里只是实现了简单的操作，但是我们还可以引入通道的概念，因为卷积一般都涉及通道数的问题，对此我们可以只把函数中的循环修改一下，修改成如下即可。

```
for (int i = 0; i < param.r; i++)
    {
        for (int j = 0; j < param.s; j++)
        {
```

```
            int posh_real = posh_ori + i;
            int posw_real = posw_ori + j;
            if (posh_real >= 0 && posw_real >= 0 && posw_real < param.w && posh_real < par
am.h)
            {
                int inOffsetTmp = inOffset;
                int weiOffsetTmp = weiOffset;
                for (int channel = 0; channel < param.c; channel++)
                {
                    sum += param.input[inOffsetTmp + i * param.w + j] * param.weight[weiOf
fsetTmp + i * param.s + j];
                    inOffsetTmp += inChannelOffset;
                    weiOffsetTmp += weightChannelOffset;
                }
            }
        }
    }
```

对比测试



可以发现 0 版的性能很低与 cudnn 带的卷积操作有显著差距

(2) 版本 1：利用共享内存减少访存次数

在这一版中，第一步先修改坐标映射为 K、R、S 三个维度的循环合并为一个循环，方便后续对其优化。由于卷积核的布局为 K*C*R*S，输入数据布局为 N*C*H*W，那么对于卷积核来说 C*R*S 维

度相邻，无需坐标映射，则此时的循环为

```
for (int i = 0; i < param.r * param.s * param.c; i++)
{
    int weiOffsetTmp = i;
    int curC = i / (param.r * param.s);              // input channel offset
    int curR = (i % (param.r * param.s)) / param.s; // kernel r offset
    int curS = (i % (param.r * param.s)) % param.s; // kernel s offset
    int curH = posh_ori + curR;                      // input real h
    int curW = posw_ori + curS;                      // input real w
    int inOffsetTmp = curC * inChannelOffset + curR * param.w + curS;
    if (curH >= 0 && curW >= 0 && curW < param.w && curH < param.h)
    {
        sum += param.weight[weiOffset + weiOffsetTmp] * param.input[inOffset+inOffsetTmp];
    }
}
```

而后对其进行线程块级优化，利用**共享内存**减少重复内存的读取。在线程块级别，数据从全局内存加载。但我们必须平衡多个相互冲突的 Block 策略。更大的线程块意味着从全局内存中获取的数据更少，从而确保 DRAM 带宽不会成为瓶颈。然而，大的线程块块可能无法很好地匹配问题的维度。对于这种情况的优化策略，将 K * R * S 维度跨多个 Block 或多个 Warp 进行**分区**。这些线程块或扭曲并行计算矩阵乘积；然后对乘积进行**规约**以计算结果。

我们可以通过计算来分析一下对性能的提高：

访存量：

$$修改前：(N*Oh*Ow)_{tile}*K_{tile}*C*R*S*4Bytes$$

$$修改后：((N*Oh*Ow)_{tile}+K_{tile})*C*R*S*4Bytes$$

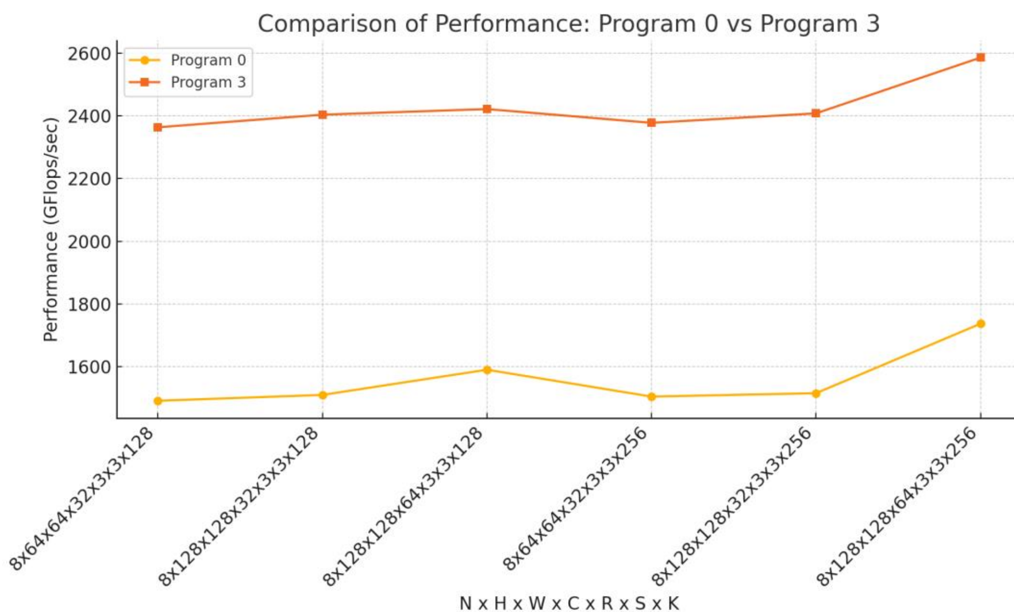
可以发现$(N*Oh*Ow)_{tile}=K_{tile}$是可以让访存量最小，此时$(C*R*S)_{tile}=(N*Oh*Ow)_{tile}=16$

所以程序中的循环可以修改为：

```
for (int i = 0; i < param.r * param.s * param.c; i += 16)
    {
        int weiOffsetTmp = i + tx;
        smemweight[ty][tx] = param.weight[weiOffset + weiOffsetTmp];

        int curC = (i + ty) / (param.r * param.s);              // channel offset
```

```
        int curR = ((i + ty) % (param.r * param.s)) / param.s; // kernel r offset
        int curS = ((i + ty) % (param.r * param.s)) % param.s; // kernel s offset
        int curH = posh_ori + curR;                            // input h
        int curW = posw_ori + curS;                            // input w
        int inOffsetTmp = curC * inChannelOffset + curH * param.w + curW;
        sminput[ty][tx] = param.input[inOffset + inOffsetTmp];

        __syncthreads();
#pragma unroll
        for (int subcrs = 0; subcrs < 16; ++subcrs)
        {
            sum += smemweight[ty][subcrs] * smeminput[subcrs][tx];
        }
        __syncthreads();
    }
```

对比测试



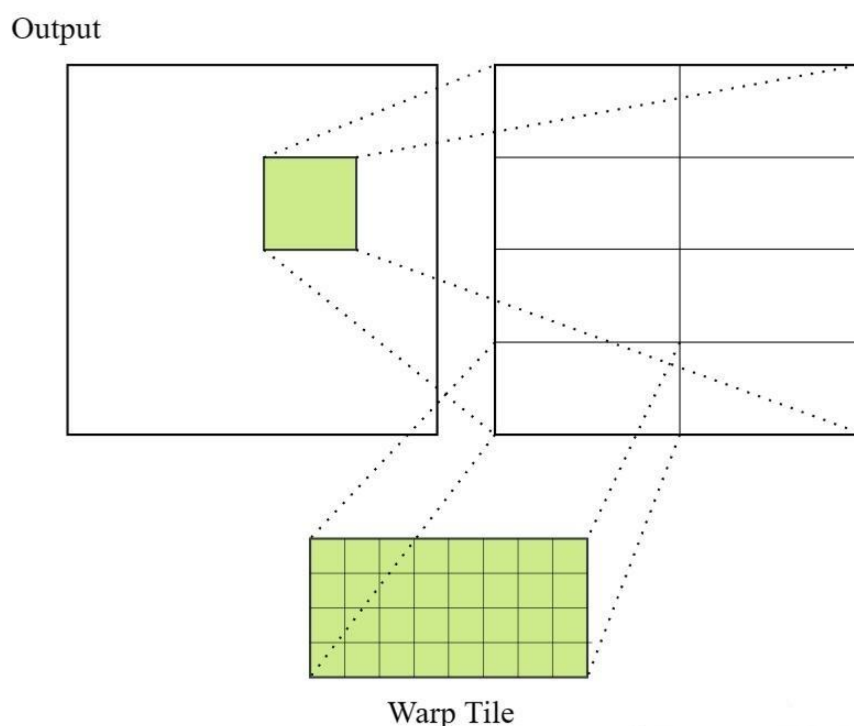
过这张图可以明显看到，在大部分配置中，程序 3 的性能显著高于程序 0 的执行时间。这里要说明的一点是，由于 KRS 循环的合并，模与除操作会加大 ALU 和 SLU 操作在计算中的占比，使得程序性能的优化不显著。

(3) 版本 2：使用 Warp Tile 技术优化

为了进一步的减少访存次数，我们可以使用 WarpTile 技术。Warp Tile 技术是 GPU 编程中的一种优化技术，通过合理组织和调度 Warp 内的线程以提高内存访问效率和计算效率。Warp 是 GPU 中并行计算的基本单位，通常由 32 个线程组成。Warp Tile 技术的目标是优化 Warp 内线程对共享内存和全局

内存的访问，从而提高计算的访存比（计算量与内存访问次数的比值）。

Warp 是 SM 调度和执行的基础概念。一个 threadblock 的访问 shared memory 的次数与 Warp 内的线程排布有关。单个 Warp 内的 32 个线程被排列为$Warp_x, Warp_y$，那么访问 shared memory 的次数就与$Warp_x$与$Warp_y$之和成正比，但计算量一定为$Warp_x * Warp_y$。那么考虑计算访存比的情况下，$Warp_x$与$Warp_y$的差值越小越好。这里采取 4 * 8 的排列方式。



代码实现部分，注意这只影响 lds(**load from shared memory**)/stg(**store to global memory**) 部分，所以 lds 与 stg 地址计算的变化如下，

```
//Warp tile
    const uint32_t lane_id = threadIdx.x % 32;
    const uint32_t warp_id = threadIdx.x / 32;
    const uint32_t mma_tid_x = lane_id % 8;
    const uint32_t mma_tid_y = lane_id / 8;
//lds addr
    uint32_t weight_lds_addr = (warp_id / 2) * 4 + mma_tid_y;
    uint32_t input_lds_addr = (warp_id % 2) * 8 + mma_tid_x;
//stg addr
    int x = bx * 16 + input_lds_addr;
    int y = by * 16 + weight_lds_addr;
```

对应实际的 lds 变为与上述 lds 地址相关

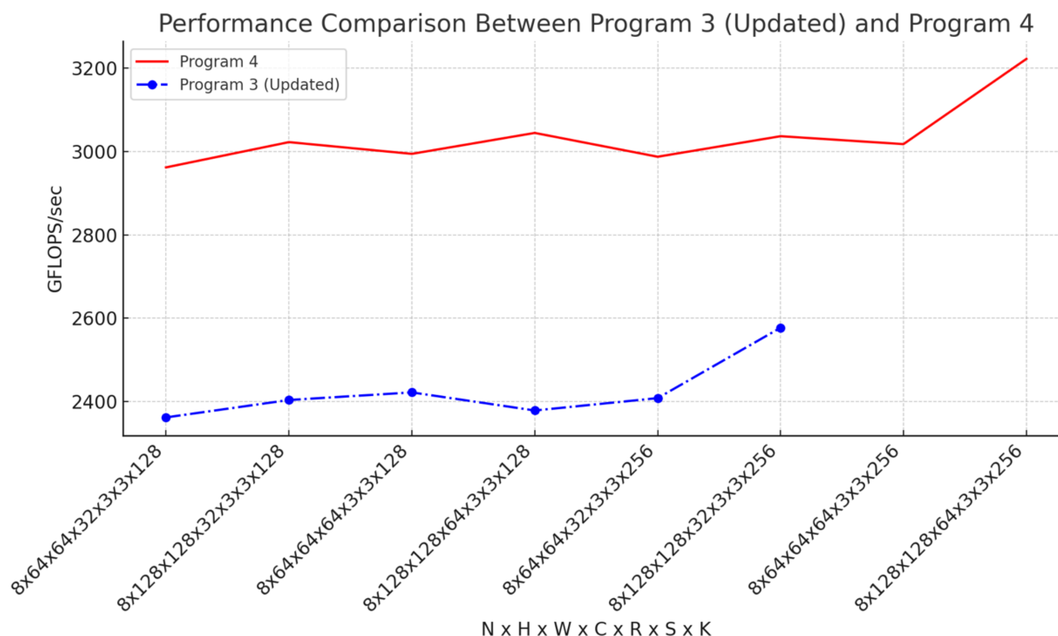
```
#pragma unroll
for (int subcrs = 0; subcrs < 16; ++subcrs)
{
    //load from shared memory
    sum += smemweight[weight_lds_addr][subcrs] * smeminput[subcrs][input_lds_addr];
}
```

此时的 x 与 y 为 lds/stg 的地址，ldg 的地址不能再沿用 x 与 y，也就是说当前处理数据的坐标运算还是用之前 x 与 y

```
// 当前线程处理的数据点在 oh、ow 上的坐标
    int posOh = (bx * 16 + tx) / param.Ow;
    int posOw = (bx * 16 + tx) % param.Ow;
    int weiOffset = (by * 16 + ty) * param.c * param.r * param.s;
```

在这一步后就可以利用实现 Warp 级 GEMM 实现卷积。可以通过发出 mma.sync 或 wmma 指令的 TensorCore 来实现（但要注意这里 wmma 的最小单位以及数据类型问题），也可以通过 CUDA core 做 thread level GEMM 来实现。

对比测试



可以发现改进后程序的性能增加显著，远超版本 2 的性能。

(4) 版本 3：使用 Thread Tile 技术优化

Thread Tile 是 GPU 编程中的一种优化技术，它将计算任务在每个线程内进一步细分，以提高计算效率和内存访问效率。在这种方法中，每个线程负责计算多个结果，而不是仅仅计算一个结果。通过这种方式，可以提高计算的访存比（计算量与内存访问次数的比值），并最大限度地利用共享内存和寄存器资源。

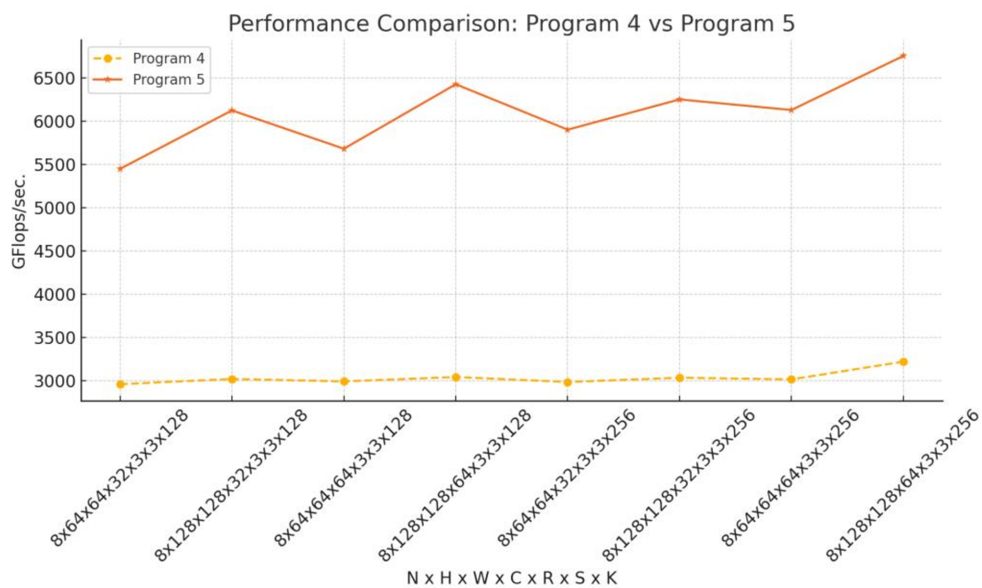
需要确定每个线程在 Warp 和 Thread Tile 中的位置。然后更改访问地址

```
//ldg
    posOh = (bx * 64 + tx % 64) / param.Ow;
    posOw = (bx * 64 + tx % 64) % param.Ow;
    weiOffset = (by * 64 + tx / 4) * param.c * param.r * param.s;
    curC = (crs + tx / 64) / (param.r * param.s);             // channel offset
    curR = ((crs + tx / 64) % (param.r * param.s)) / param.s; // kernel r offset
    curS = ((crs + tx / 64) % (param.r * param.s)) % param.s; // kernel s offset
```

sts/lds

```
// lds addr
    weight_lds_addr = (warp_id / 2) * 16 + mma_tid_y * 4;
    input_lds_addr = (warp_id % 2) * 32 + mma_tid_x * 4;
```

然后在使用向量外积方法通过计算一个矩阵的列向量和另一个矩阵的行向量的外积来构建结果矩阵。



首先版本 4 比版本 3 提高性能仅 1 倍，可以发现次方法提高性能非常显著，其次计算强度与性能强相

关，所以随着计算强度增加性能提高比较明显。

(5) 版本 4：提高计算强度

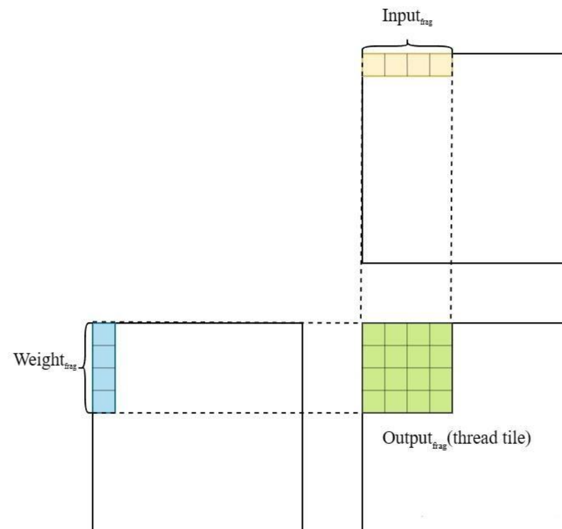
由于上一版发现，计算强度增加会让性能提高，所以本次选取了更大的$(N*Oh*Ow)_{tile}$，K_{tile}，$(C*R*S)_{tile}$

这里选取

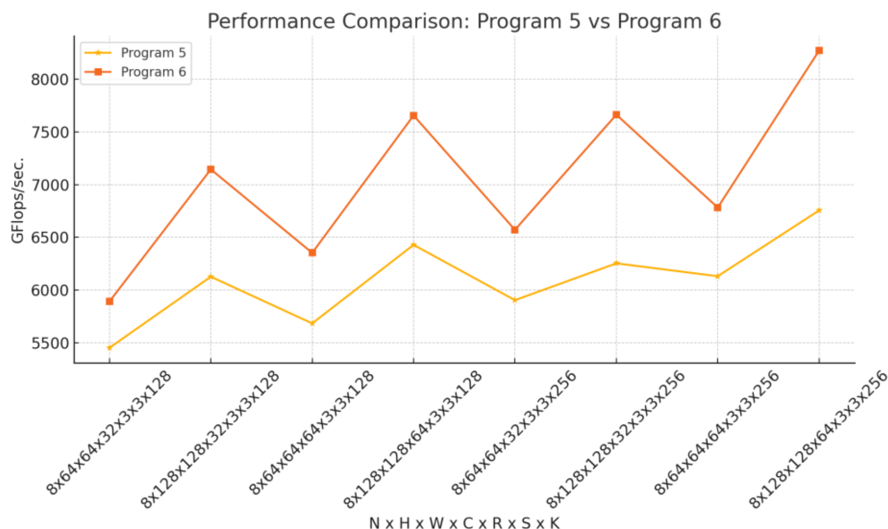
$$(N*Oh*Ow)_{tile} = K_{tile} = 128$$
$$(C*R*S)_{tile} = 8$$

其次我们还可以把卷次操作的矩阵乘改为向量外积，这样对共享内存的利用和寄存器的利用有帮助。

原理图：



对比测试：



程序 6 在所有测试参数下的性能均高于程序 5，所以版本 6 的改动是有效的。至此优化的性能已经接近 cudnn 自带包中的性能，也许还可以优化一下**共享内存的 Bank 冲突**，但是由于大家精力有限，已经从中学到了很多东西，所以就停手了。

五、测试结果

每一版的测试结果数据已经在实验过程中，所以这里就以第 5 版编译和测试为例来展示得到测试结果的过程。

为了测试程序方便，我们专门写了 makefile 文件进行编译和 sh 脚本进行测试，通过这样的方法可以自动化测试，现在把使用方法以截图的形式放在这里。

首先是 makefile 代码，通过修改 makefile 代码中的 TARGET 的值来实现对某一个版本的编译

```
CC=nvcc
TARGET := conv_3_2

CXXFLAGS += -arch=sm_60
INCLUDES   += -I./include
LDFLAGS = -lcudnn

#  定义源文件列表
SRCS := ./src/$(TARGET).cu ./main.cu

#将对应的 c 文件名转为 o 文件后放在下面的 CUR_OBJS 变量中
CUR_OBJS=${SRCS:.cu=.o}

EXECUTABLE=implgemm

all:$(EXECUTABLE)
```

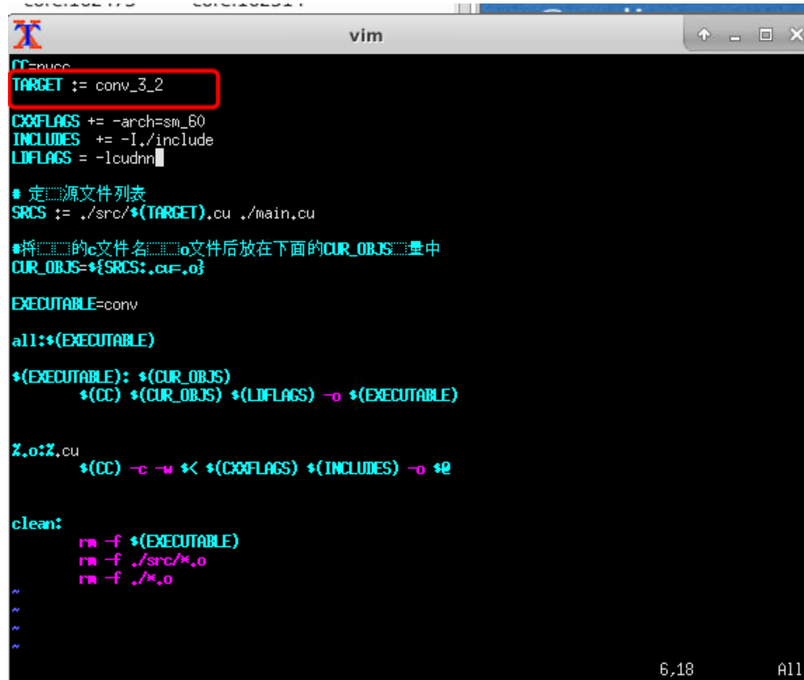
```
$(EXECUTABLE): $(CUR_OBJS)
	$(CC) $(CUR_OBJS) $(LDFLAGS) -o $(EXECUTABLE)



%.o:%.cu
	$(CC) -c -w $< $(CXXFLAGS) $(INCLUDES) -o $@



clean:
	rm -f $(EXECUTABLE)
	rm -f ./src/*.o
	rm -f ./*.o
```

然后就是 sh 脚本文件，通过直接 shell 脚本可以实现自动调用 makefile 编译和测试

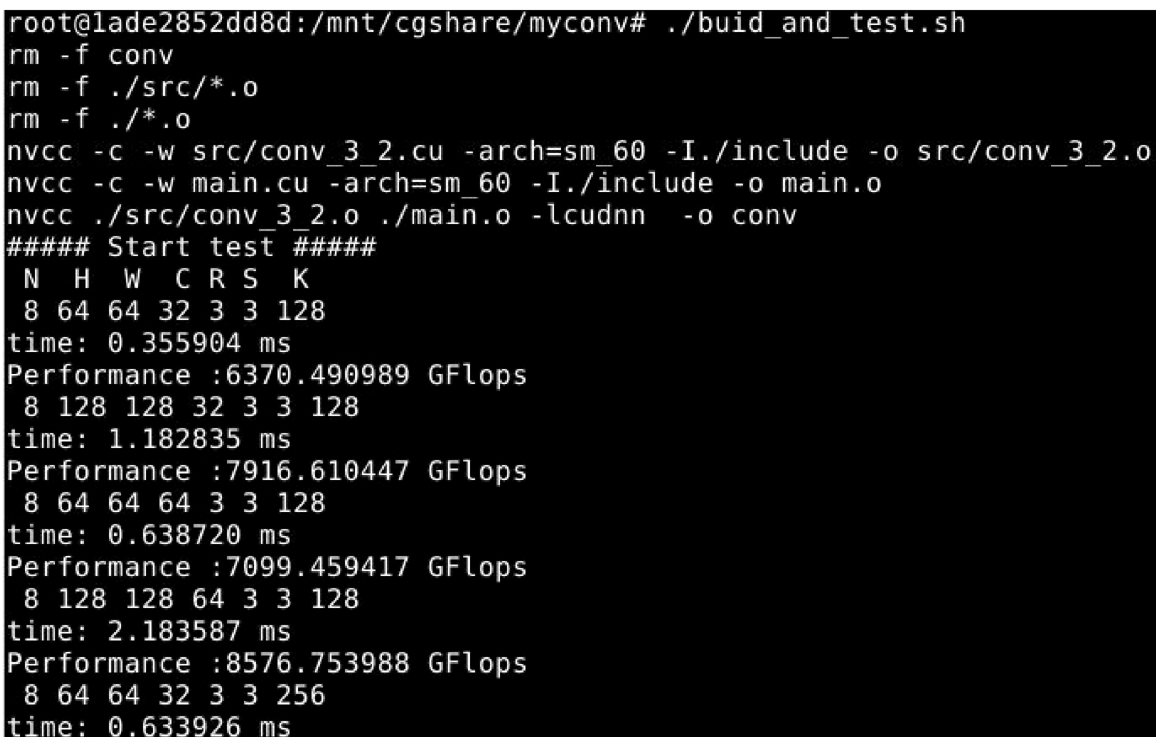
```
make clean
make
echo "##### Start test #####";
echo " N    H    W    C R S    K";
for case_k in {1..2}
do for case_c in {1..2}
	do for case_size in {1..2}
		do
		C=$[ $case_c * 32 ]
		H=$[ $case_size * 64 ]
		W=$[ $case_size * 64 ]
		K=$[ $case_k * 128 ]
		./conv 8 ${C} ${H} ${W} ${K} 3 3 1 1 0 0;
		done;
	done;
```

```
done;
echo "##### Test finish! #####";
```

所以每次的测试就是编写好代码，后修改 makefile



然后运行 shell 文件即可。



然后根据此数据使用 python 进行绘图即可。

六、实验问题及解决

1. 实验环境没有 cudnn

```
wget https://developer.download.nvidia.com/compute/cudnn/9.1.1/local_installers/cudnn-local-repo-ubuntu2004-9.1.1_1.0-1_amd64.debsudo
dpkg -i cudnn-local-repo-ubuntu2004-9.1.1_1.0-1_amd64.debsudo
cp /var/cudnn-local-repo-ubuntu2004-9.1.1/cudnn-*-keyring.gpg /usr/share/keyrings/sudo
apt-get updatesudo
apt-get -y install cudnn
```

通过以上命令配置完毕

2. 在编写版本 2 是运行后报错 segment fault

```
root@1ade2852dd8d:/mnt/cgshare/myconv# ./buid_and_test.sh
rm -f conv
rm -f ./src/*.o
rm -f ./*.o
nvcc -c -w src/conv_1_1.cu -arch=sm_60 -I./include -o src/conv_1_1.o
nvcc -c -w main.cu -arch=sm_60 -I./include -o main.o
nvcc ./src/conv_1_1.o ./main.o -lcudnn  -o conv
##### Start test #####
 N  H  W  C R S  K
 8 64 64 32 3 3 128
time: 0.959517 ms
Performance :2362.942709 GFlops
 8 128 128 32 3 3 128
time: 3.896214 ms
Performance :2403.370240 GFlops
./buid_and_test.sh: line 7:   973 Segmentation fault      (core dumped) ./conv 8
 ${C} ${H} ${W} ${K} 3 3 1 1 0 0
 8 128 128 64 3 3 128
time: 7.734656 ms
Performance :2421.321817 GFlops
 8 64 64 32 3 3 256
time: 1.909130 ms
Performance :2375.201143 GFlops
```

经过排查和讨论后发现，原来是申请的共享内存大小不够导致了越界的问题。

3. 编写编译和测试程序的问题

原本我们想直接用 python 来调用测试程序，但是发现不如 shell 调用更加方便，并且针对编译程序，每一版只是在上一版的基础上改了一下，所以我们是通过重写方法实现重载的，先用头文件声明，然后再重写方法即可，并且 main 函数可以不做修改就可调用。

然后针对编译指令比较复杂，所以我们编写了 makefile 文件，以方便的调用的编译程序，只用 make 即可，极大的简化了编译和测试的过程。

七、 实验总结

在本次基于 CUDA 实现二维卷积的实验中，我们经历了一个从基础实现到多次优化的过程。这不仅是对编程技能的一次提升，更是对并行计算思想的一次深刻理解。

首先，我们从基本的全局内存访问方式开始，逐步引入了共享内存、线程块优化、Warp Tile 技术以及 Thread Tile 技术。每一步优化都基于前一步的基础，不断提升性能。在每一个版本的优化中，我们通过对访存量和计算强度的分析，找到了瓶颈并提出了相应的解决方案。

通过对比测试结果，我们可以明显看到，每一次优化都带来了显著的性能提升。从最初的全局内存访问到引入共享内存，再到使用 Warp Tile 和 Thread Tile 技术，性能提升的幅度是巨大的。最终，程序 6 的性能已经接近 cuDNN 库中的实现，这不仅证明了我们的优化策略是有效的，也让我们对高性能计算有了更深的理解。

在这个过程中，我们不仅学会了如何使用 CUDA 编程，更体会到了并行计算中访存和计算平衡的重要性。通过对访存量的优化，我们有效减少了内存访问的开销；通过提高计算强度，我们充分利用了 GPU 的计算资源。这些都是在实践中得出的宝贵经验。

回顾整个实验过程，我们不仅完成了预定的实验任务，还通过不断尝试和改进，提升了自己的编程能力和问题解决能力。每一次测试和优化，都是一次学习和成长的机会。这次实验不仅让我们掌握了 CUDA 编程的技巧，更让我们懂得了科学研究的严谨态度和团队合作的重要性。

总之，这次实验不仅让我们在技术上有了质的飞跃，更在思想上有了深刻的认识。我们相信，这些收获将对我们未来的学习和研究产生深远的影响。

致谢

首先要感谢施以援手的同学，在实验中遇到问题与同学讨论后得到了很好的解决。其次感谢队友，能够坚持不懈，努力学习和合作交流解决问题。除此之外我们还要特别感谢课程老师给在课上教授的宝贵知识和提出的建议。老师的理论和建议为我们的实验提供了重要的指导，使我们在面对挑战时能

够有明确的方向和思路。虽然实验过程中的具体问题主要靠我们自己解决，但老师的总体知识和指导理论是我们完成实验的坚实后盾。

感谢老师在整个课程过程中的辛勤付出和耐心指导。您的教诲和支持让我们受益匪浅，我们将铭记在心，并在今后的学习和工作中不断努力，不辜负您的期望。再次感谢老师的悉心教导，愿您工作顺利，身体健康！