GPU Computing

Project

담당교수 : 공영호

제출일 : 2021. 11. 16.

학과 : 컴퓨터정보공학부

학번 : 2017202088

이름 : 신해담

1. Introduction

- GPU에서 224*224*300의 입력 데이터와 3*3 2D weight의 convolution 연산을 수행하는 커널 함수를 작성한다.
- GPU에서 224*224*300의 입력 데이터와 1*1*300 3D weight 900개의 convolution 연산을 수행하는 커널 함수를 작성한다.
- GPU에서 224*224*300의 입력 데이터와 3*3*300 3D weight 900개의 convolution 연산을 수행하는 커널 함수를 작성한다.

2. Background

- **GPU 구조**

  GPU는 그래픽 처리 및 각종 병렬연산을 처리하는 장치이다. 따라서 병렬연산의 성능을 높이기 위해서 많은 양의 코어를 가지고, 많은 수의 thread를 실행할 수 있다.

  GPU는 texture(thread) processor cluster(TPC)의 집합으로 구성된다. TPC에는 pixel shading에 특화된 TEX와 streaming multiprocessor(SM)이 들어있다. 각 SM은 캐시, 공유 메모리, streaming processor(SP), special function unit(SFU) 등을 가지며, CUDA의 thread block을 처리하는 기본 단위이다. SP는 단일 thread에 대한 scalar ALU를 가지며, 실질적인 코어의 역할을 한다.

  GPU의 thread가 실행될 때 메모리 접근 등의 큰 지연을 가지는 명령어를 실행하는 경우 그 지연시간동안 대기하는 것이 아니라 다른 thread로 context switching해서 실행하게 된다. Warp는 한 번에 실행할 수 있는 thread 수로 CUDA에서 32 threads를 의미하며, GPU는 어떤 warp를 실행할 것인지 scheduling하게 된다.

  CUDA에서 thread block을 지정해서 최대 1024의 thread가 동일한 계산을 실행할 수 있다. 각 thread는 thread ID로 구분하며, 이를 사용해서 작업을 하거나 서로 다른 데이터에 접근할 수 있다. Thread block들이 모여서 grid를 구성한다.

  Grid는 SPA에서 수행되며, thread block들이 SM에 분배된다. 각 SM은 thread를 warp단위로 스케줄링하여 실행시키고, 모든 warps가 완료되면 resource를 해제한다.

- **GPU 메모리 계층구조**

  프로그램 실행시간에서 I/O의 overhead가 차지하는 비중이 매우 크다. GPU에서는 I/O overhead를 줄이기 위해서 여러 종류의 메모리를 사용한다.

  Registers는 가장 빠르지만 크기가 가장 작다.

  Shared memory는 thread block에 존재하는 메모리이며, block 안의 thread들이 공동으로 사용한다. register보다는 느리지만 global memory보다 빠르다.

  Global memory는 VRAM으로 shared memory보다 느리며 크기가 매우 크다.

그 외에 읽기전용인 constant memory와 local memory 등이 있다.

- **GPU에서 convolution 연산의 장점**

  요즘 사용하는 CPU는 multicore를 사용함으로써 성능향상을 꾀한다. 하지만 GPU는 manycore로 CPU보다 훨씬 많은 코어를 사용한다. 또한 코어당 수행 가능한 thread 수도 차이가 난다. CPU 코어는 한 번에 multiple threads를 수행할 수 있지만, GPU 코어는 한 번에 many threads를 수행할 수 있다. 그 외에도 공유 메모리나 코드 공유 등의 전략을 사용해서 GPU는 CPU에 비해 병렬처리를 효과적이고 빠르게 수행할 수 있다.

  Convolution 연산은 매우 큰 input array에 동일한 연산을 반복해서 수행한 다음 마찬가지로 매우 큰 output array에 저장한다. 따라서 GPU를 사용해서 병렬처리하면 CPU에서보다 빠르게 convolution 연산을 수행할 수 있다.

3. Code explanation

- **Conv3x3_2d**

```
dim3 dimGrid_2d(7, 7, 300);
dim3 dimGrid_3d(7, 7, 900);
dim3 dimBlock(32, 32, 1);
```

32*32의 thread block과 7*7*300의 grid에서 수행한다.

```
__global__ void Conv3x3_2d(double* input, double* weight, double* output)
{
    // shared mem
    __shared__ double s_input[1024];
    __shared__ double s_weight[9];

    // index
    unsigned int tx = threadIdx.x;  // 0~31
    unsigned int ty = threadIdx.y;  // 0~31
    unsigned int bx = blockIdx.x;   // 0~6
    unsigned int by = blockIdx.y;   // 0~6
    unsigned int bz = blockIdx.z;   // 0~299

    // get shared data
    s_input[ty*32 + tx] = input[bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)];
    if(ty == 0 && tx < 9) s_weight[tx] = weight[tx];

    // barrier
    __syncthreads();
```

공유 메모리에 s_input과 s_weight를 선언한다. s_input은 224*224*300입력에서 index를 사용해서 해당하는 위치의 input을 가져온다. s_weight는 index보다 작으므로, read/write가 반복 발생하지 않도록 if문으로 제한했다. Global memory 값들이 공유 메모리에 들어오도록 __syncthreads 함수로 대기한다.

```
// calculate
double sum = 0.0;
if(tx > 0 && tx < 31 && ty > 0 && ty < 31)
{
    sum += s_input[ (ty-1)*32 + tx-1] * s_weight[0];
    sum += s_input[ (ty-1)*32 + tx  ] * s_weight[1];
    sum += s_input[ (ty-1)*32 + tx+1] * s_weight[2];

    sum += s_input[ ty*32     + tx-1] * s_weight[3];
    sum += s_input[ ty*32     + tx  ] * s_weight[4];
    sum += s_input[ ty*32     + tx+1] * s_weight[5];

    sum += s_input[ (ty+1)*32 + tx-1] * s_weight[6];
    sum += s_input[ (ty+1)*32 + tx  ] * s_weight[7];
    sum += s_input[ (ty+1)*32 + tx+1] * s_weight[8];
}
else
{
    sum += s_weight[0] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx)];
    sum += s_weight[1] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 1)];
    sum += s_weight[2] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 2)];

    sum += s_weight[3] * input[ bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx)];
    sum += s_weight[4] * s_input[ty*32 + tx];
    sum += s_weight[5] * input[ bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 2)];

    sum += s_weight[6] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx)];
    sum += s_weight[7] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 1)];
    sum += s_weight[8] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 2)];
}

output[bz*224*224 + (by*32 + ty)*224 + (bx*32 + tx)] = sum;
}
```

convolution 연산을 수행하는 코드이다. 공유 메모리에 가져온 input값만 사용하는 경우 s_input을 사용해서 계산한다. 공유 메모리에 없는 값을 사용해야 하는 경우, 경우의 수가 많아서 코드가 복잡해지고 직관성이 떨어지므로 항상 공유 메모리를 사용하는 하나를 제외하고 일관적으로 global memory 값을 가져오도록 구현했다.

- **Conv1x1_3d**

```
dim3 dimGrid_2d(7, 7, 300);
dim3 dimGrid_3d(7, 7, 900);
dim3 dimBlock(32, 32, 1);
```

32*32의 thread block과 7*7*900의 grid에서 수행한다.

```
__global__ void Conv1x1_3d(double* input, double* weight, double* output)
{
    // shared mem
    __shared__ double s_weight[300];

    // index
    unsigned int tx = threadIdx.x;  // 0~31
    unsigned int ty = threadIdx.y;  // 0~31
    unsigned int bx = blockIdx.x;   // 0~6
    unsigned int by = blockIdx.y;   // 0~6
    unsigned int bz = blockIdx.z;   // 0~899

    // get shared data
    if(ty*32 + tx < 300)    s_weight[ty*32 + tx] = weight[bz*300 + (ty*32 + tx)];

    // barrier
    __syncthreads();
```

1x1 3d convolution의 경우, input값은 weight마다 한 번씩만 쓰이므로 s_weight만 선언하고 index를 사용해서 초기화한다.

```
    // calculate
    double sum = 0.0;
    for(int i = 0; i < 300; i=i+6)
    {
        sum += input[(i  )*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i];
        sum += input[(i+1)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+1];
        sum += input[(i+2)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+2];
        sum += input[(i+3)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+3];
        sum += input[(i+4)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+4];
        sum += input[(i+5)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+5];
    }

    output[(bz*224*224) + (by*32 + ty)*224 + (bx*32 + tx)] = sum;
}
```

convolution 연산을 수행하는 코드이다. 해당하는 input과 weight를 곱한 값을 곱한 다음 해당 output에 더하도록 구현했다.

- **Conv3x3_3d**

```
dim3 dimGrid_2d(7, 7, 300);
dim3 dimGrid_3d(7, 7, 900);
dim3 dimBlock(32, 32, 1);
```

32*32의 thread block과 7*7*900의 grid에서 수행한다.

```
__global__ void Conv3x3_3d(double* input, double* weight, double* output)
{
    // shared mem
    __shared__ double s_input[1024];
    __shared__ double s_weight[900];

    // index
    unsigned int tx = threadIdx.x;  // 0~31
    unsigned int ty = threadIdx.y;  // 0~31
    unsigned int bx = blockIdx.x;   // 0~6
    unsigned int by = blockIdx.y;   // 0~6
    unsigned int bz = blockIdx.z;   // 0~899
```

공유 메모리에 input값을 저장할 s_input과 weight값을 저장할 s_weight를 선언한다.

```
// calculate
double sum = 0.0;
for(int i = 0; i < 300; i++)
{
    // get shared data
    s_input[ty*32 + tx] = input[i*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)];
    //if(ty == 0 && tx < 9) s_weight[tx] = weight[bz*300*9 + i*9 + tx];
    if(i%100 == 0 && ty*32+tx < 900)
        s_weight[ty*32 + tx] = weight[bz*300*9 + ty*32 + tx];

    // barrier
    __syncthreads();
```

사용하는 데이터들이 channel에 따라 바뀌므로, 해당 데이터들을 loop 안에서 업데이트한다. 공유메모리가 다 업데이트 된 후에 실행하도록 __syncthreads 함수로 대기한다.

```
        if(tx > 0 && tx < 31 && ty > 0 && ty < 31)
        {
            sum += s_input[ (ty-1)*32 + tx-1] * s_weight[0];
            sum += s_input[ (ty-1)*32 + tx  ] * s_weight[1];
            sum += s_input[ (ty-1)*32 + tx+1] * s_weight[2];

            sum += s_input[ ty*32     + tx-1] * s_weight[3];
            sum += s_input[ ty*32     + tx  ] * s_weight[4];
            sum += s_input[ ty*32     + tx+1] * s_weight[5];

            sum += s_input[ (ty+1)*32 + tx-1] * s_weight[6];
            sum += s_input[ (ty+1)*32 + tx  ] * s_weight[7];
            sum += s_input[ (ty+1)*32 + tx+1] * s_weight[8];
        }
        else
        {
            sum += s_weight[0] * input[ i*226*226 + (by*32 + ty    )*226 + (bx*32 + tx)];
            sum += s_weight[1] * input[ i*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 1)];
            sum += s_weight[2] * input[ i*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 2)];

            sum += s_weight[3] * input[ i*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx)];
            sum += s_weight[4] * s_input[ty*32 + tx];
            sum += s_weight[5] * input[ i*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 2)];

            sum += s_weight[6] * input[ i*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx)];
            sum += s_weight[7] * input[ i*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 1)];
            sum += s_weight[8] * input[ i*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 2)];
        }

        // barrier
        __syncthreads();
    }

    output[(bz*224*224) + (by*32 + ty)*224 + (bx*32 + tx)] = sum;
}
```

실제 convolution 연산을 처리하는 코드이다. 3x3_2d연산과 동일하게, 공유메모리만 사용하는 경우와 global memory도 사용하는 경우로 나누어서 연산한다. 연산이 끝난 다음에 데이터가 업데이트되도록 __syncthreads 함수로 대기한다.

4. Optimization

- **실행환경**

```
Device 0: "NVIDIA GeForce GTX 1660 SUPER"
  CUDA Driver Version / Runtime Version          11.4 / 11.4
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 6144 MBytes (6442450944 bytes)
  (22) Multiprocessors, ( 64) CUDA Cores/MP:     1408 CUDA Cores
  GPU Max Clock rate:                            1830 MHz (1.83 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               zu bytes
  Total amount of shared memory per block:       zu bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          zu bytes
  Texture alignment:                             zu bytes
  Concurrent copy and kernel execution:          Yes with 6 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

GPU 는 GTX 1660 SUPER 를 사용한다.

CUDA core 가 1408 개 있다.

multiprocessor 당 최대 trhead 수는 1024, block 당 최대 thread 수는 1024 다.

- **Grid 및 thread block 설정**

```
dim3 dimGrid_2d(7, 7, 300);
dim3 dimGrid_3d(7, 7, 900);
dim3 dimBlock(32, 32, 1);
```

output array 에 값을 써야하므로 output index 를 총 thread 수가 따른다.

2d 의 경우 224*224*300, 3d 의 경우 224*224*900 의 output index 를 가진다.

GPU 스펙을 통해 block 당 최대 thread 수가 1024 임을 확인했으므로, thread block 은 32*32(=1024)로 설정했다.

224*224 를 만족하기 위해 grid 의 width, height 는 7*7 로 선택했고(32*7*32*7 = 224*224), 2d/3d 의 output channel 에 따라 channel 을 각각 300/900 으로 설정했다.

- **Memory 사용**

```
// shared mem
__shared__ double s_input[1024];
__shared__ double s_weight[9];
```

shared memory 는 각 thread block 들이 공유할 데이터를 반영하도록 구현했다. Input 은 3x3 연산에서 공유될 수 있으며, weight 는 모든 경우에서 공유될 수 있다.

```
// get shared data
s_input[ty*32 + tx] = input[bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)];
if(ty == 0 && tx < 9) s_weight[tx] = weight[tx];
```

weight 는 thread index 보다 수가 작으므로, 중복실행이 발생하지 않도록 if 문으로 일부 thread 만 값을 가져오도록 제한했다.

```
// calculate
double sum = 0.0;
```

output 에 write 할 값은 register 에 할당해서 빠르게 쓰일 수 있도록 구현했다.

```
// shared mem
__shared__ double s_input[1024];
__shared__ double s_weight[900];
```

```
// get shared data
s_input[ty*32 + tx] = input[i*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)];
//if(ty == 0 && tx < 9) s_weight[tx] = weight[bz*300*9 + i*9 + tx];
if(i%100 == 0 && ty*32+tx < 900)
    s_weight[ty*32 + tx] = weight[bz*300*9 + ty*32 + tx];
```

3x3_3D 연산에서 for loop 내부에서 shared memory 값을 계속 업데이트한다. Input 이 업데이트되는 것은 어쩔 수 없지만, weight 는 한 번의 loop 마다 9 개의 값만 shared memory 로 가져온다면 다른 thread 들이 놀게 된다. 따라서 이 점을 완화하기 위해 s_weight index 를 900 으로 선언하고, loop 100 번마다 900(3*3*300 / 3)의 weight 를 가져오도록 구현했다.

- **Loop unrolling**

```
for(int i = 0; i < 300; i=i+6)
{
    sum += input[(i  )*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i];
    sum += input[(i+1)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+1];
    sum += input[(i+2)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+2];
    sum += input[(i+3)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+3];
    sum += input[(i+4)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+4];
    sum += input[(i+5)*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 1)] * s_weight[i+5];
}
```

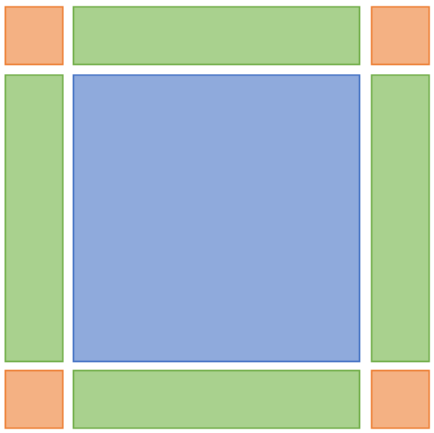
```
if(tx > 0 && tx < 31 && ty > 0 && ty < 31)
{
    sum += s_input[ (ty-1)*32 + tx-1] * s_weight[0];
    sum += s_input[ (ty-1)*32 + tx  ] * s_weight[1];
    sum += s_input[ (ty-1)*32 + tx+1] * s_weight[2];

    sum += s_input[ ty*32     + tx-1] * s_weight[3];
    sum += s_input[ ty*32     + tx  ] * s_weight[4];
    sum += s_input[ ty*32     + tx+1] * s_weight[5];

    sum += s_input[ (ty+1)*32 + tx-1] * s_weight[6];
    sum += s_input[ (ty+1)*32 + tx  ] * s_weight[7];
    sum += s_input[ (ty+1)*32 + tx+1] * s_weight[8];
}
```

위와 같이 loop 를 사용해서 구현할 수 있는 부분에서 loop 를 사용하지 않거나 loop 구간을 6 개씩 풀어서 약간의 성능향상을 꾀했다.

5. Conclusion

- **3x3 convolution**

```
// calculate
double sum = 0.0;
if(tx > 0 && tx < 31 && ty > 0 && ty < 31)
{
    sum += s_input[ (ty-1)*32 + tx-1] * s_weight[0];
    sum += s_input[ (ty-1)*32 + tx  ] * s_weight[1];
    sum += s_input[ (ty-1)*32 + tx+1] * s_weight[2];

    sum += s_input[ ty*32     + tx-1] * s_weight[3];
    sum += s_input[ ty*32     + tx  ] * s_weight[4];
    sum += s_input[ ty*32     + tx+1] * s_weight[5];

    sum += s_input[ (ty+1)*32 + tx-1] * s_weight[6];
    sum += s_input[ (ty+1)*32 + tx  ] * s_weight[7];
    sum += s_input[ (ty+1)*32 + tx+1] * s_weight[8];
}
else
{
    sum += s_weight[0] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx)];
    sum += s_weight[1] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 1)];
    sum += s_weight[2] * input[ bz*226*226 + (by*32 + ty    )*226 + (bx*32 + tx + 2)];

    sum += s_weight[3] * input[ bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx)];
    sum += s_weight[4] * s_input[ty*32 + tx];
    sum += s_weight[5] * input[ bz*226*226 + (by*32 + ty + 1)*226 + (bx*32 + tx + 2)];

    sum += s_weight[6] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx)];
    sum += s_weight[7] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 1)];
    sum += s_weight[8] * input[ bz*226*226 + (by*32 + ty + 2)*226 + (bx*32 + tx + 2)];
}

output[bz*224*224 + (by*32 + ty)*224 + (bx*32 + tx)] = sum;
}
```



3x3 convolution 계산하는 과정에서 위 그림과 같이 경우의 수가 9 가지 발생한다. 이 경우를 모두 고려해서 if 문으로 제어하면서 input 과 s_input 의 index 를 관리하면 좋겠지만, 코드가 복잡해지고 직관성이 떨어지게 된다. 게다가 thread scheduling 과 컴파일러 성능을 고려하면 모든 경우를 고려하는 것보다 global memory 접근이 필요한 경우와 그렇지 않은 경우로 나누어서 계산하는 것이 효율적이라 판단하고 구현하게 되었다.