LAB 4
Dennis Zang
704766877

Please explain the parallelization strategies you applied for each step (convolution, max pooling, etc) in this lab. How does it differ from your Lab 3 CPU parallelization strategy? What made you change your strategy?

The general idea of the parallelization in my implementation is dividing the final 112x112x256 output row-wise (ie into 2 blocks 56x112x256, 4 blocks 28x112x256, etc...), then calculating the values using the corresponding partitions ((224 / (# blocks)) + 4)x228x256 from the input and 256x5x5 from the weights. Because we are using a GPU this time, sharing memory between becomes much more significant, so we would also partition the blocks in multiple threads for each output 'i'.
Partition-wise, the blocks are partitioned in the same way as in lab 3, but with the GPU, we can instantiate a large number of threads in each block and share the input partition between multiple threads, making much better use of parallelism, particularly SPMD (single program, multiple data).
When programming with the GPU, we must make use of as many thread warps and shared memory as possible in order to get the best performance. In my case, I partitioned the output row-wise so that when we iteration through each of the j (((224 / (# blocks)) + 4)x228) grids, I can share the data between multiple threads by having each thread execute in parallel using the same data simultaneously.
Some of the strategies I used in this lab include loop unrolling, caching, and especially blocking the data so multiple threads can share and execute with at the same time. Loop unrolling

Please describe any optimization you have applied. (Optional, bonus +8(=2x4): Evaluate the performance of at least 4 different optimization techniques that you have incrementally applied and explain why such optimization improves the performance. Simply changing parameters does not count and sufficient code change is needed between versions. In your report, please include the most important changes you have applied to your code for each optimization.)

(The code is before the optimization is made compared to the final version of "nvidia.cl" and "params.sh".)

Caching:
One bottleneck we must consider is the constant readinf from __global memory. Because we much take into account the latency, one way to avoid this would be to cache the data of weight[][][][] and input[][][] into __private and __local memory. Without any form of caching and reading from __global constantly for input[][][], our average performance is about ~420 GFlops, which is about 200 GFlops lower than the final code.

Code:
```
__constant int kNum = 256;
__constant int kKernel = 5;
__constant int kImSize = 224;
__constant int kInImSize = 228;
__constant int kOutImSize = 112;

__kernel
void CnnKernel(__global const float* input, __global const float* weight,
        __global const float* bias, __global float* output) {

__private uint blockID = get_group_id(0);
__private uint threadID = get_local_id(0);
__private uint numBlocks = get_num_groups(0);
__private uint numThreadsInBlock = get_local_size(0);

for (int h = blockID; h < kOutImSize; h += numBlocks)
{
  for (int i = threadID; i < kNum; i += numThreadsInBlock)
  {
  __private float C[2][224] = {};
  for (int j = 0; j < kNum; ++j)
  {
    __private float4 mask0 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 0]);
    __private float maskRem0 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 4];
    __private float4 mask1 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 0]);
    __private float maskRem1 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 4];
    __private float4 mask2 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 0]);
    __private float maskRem2 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 4];
    __private float4 mask3 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 0]);
    __private float maskRem3 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 4];
    __private float4 mask4 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 0]);
    __private float maskRem4 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 4];

    __private float4 row0 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 0) * kInImSize) + 0]);
```

```
__private float4 row1 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + 0]);
__private float4 row2 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + 0]);
__private float4 row3 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + 0]);
__private float4 row4 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + 0]);
__private float4 ans0 = mask0 * row0;
__private float4 ans1 = mask1 * row1;
__private float4 ans2 = mask2 * row2;
__private float4 ans3 = mask3 * row3;
__private float4 ans4 = mask4 * row4;
__private float4 ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[0][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 0) * kInImSize) + 4] * maskRem0)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + 4] * maskRem1)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + 4] * maskRem2)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + 4] * maskRem3)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + 4] * maskRem4);
for (int w = 1; w < kImSize; ++w)
{
row0 = (float4) (row0.s1, row0.s2, row0.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 0) * kInImSize) + (w + 3)]);
row1 = (float4) (row1.s1, row1.s2, row1.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + (w + 3)]);
row2 = (float4) (row2.s1, row2.s2, row2.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + (w + 3)]);
row3 = (float4) (row3.s1, row3.s2, row3.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + (w + 3)]);
row4 = (float4) (row4.s1, row4.s2, row4.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + (w + 3)]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[0][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 0) * kInImSize) + (w + 4)] * maskRem0)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + (w + 4)] * maskRem1)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + (w + 4)] * maskRem2)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + (w + 4)] * maskRem3)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + (w + 4)] * maskRem4);
}

row0 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + 0]);
row1 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + 0]);
row2 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + 0]);
row3 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + 0]);
row4 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + 5) * kInImSize) + 0]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[1][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + 4] * maskRem0)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + 4] * maskRem1)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + 4] * maskRem2)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + 4] * maskRem3)
      + (input[(j * kInImSize * kInImSize) + (((h << 1) + 5) * kInImSize) + 4] * maskRem4);
for (int w = 1; w < kImSize; ++w)
{
row0 = (float4) (row0.s1, row0.s2, row0.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + (w + 3)]);
row1 = (float4) (row1.s1, row1.s2, row1.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + (w + 3)]);
row2 = (float4) (row2.s1, row2.s2, row2.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + (w + 3)]);
row3 = (float4) (row3.s1, row3.s2, row3.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + (w + 3)]);
row4 = (float4) (row4.s1, row4.s2, row4.s3, input[(j * kInImSize * kInImSize) + (((h << 1) + 5) * kInImSize) + (w + 3)]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[1][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 1) * kInImSize) + (w + 4)] * maskRem0)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 2) * kInImSize) + (w + 4)] * maskRem1)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 3) * kInImSize) + (w + 4)] * maskRem2)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 4) * kInImSize) + (w + 4)] * maskRem3)
            + (input[(j * kInImSize * kInImSize) + (((h << 1) + 5) * kInImSize) + (w + 4)] * maskRem4);
```

```
        }
    }
    for (int c = 0; c < kOutImSize; ++c)
    {
        __private union {
        float F[4];
        float4 f;
        } buf;
        buf.F[0] = C[0][c * 2] + bias[i];
        buf.F[1] = C[0][(c * 2) + 1] + bias[i];
        buf.F[2] = C[1][c * 2] + bias[i];
        buf.F[3] = C[1][(c * 2) + 1] + bias[i];
        buf.f = max(buf.f, (float4) (0.0f, 0.0f, 0.0f, 0.0f));
        output[(i * kOutImSize * kOutImSize) + (h * kOutImSize) + c] = max(
                                                    max(buf.F[0], buf.F[1]),
                                                    max(buf.F[2], buf.F[3]));
    }
    }
}
}
//INSTEAD OF storing to __private and __local, we read directly from __global
//~420 GFlops (112 x 256 = 28672)
//originally most similar to lab 3 approach
```

Utilization of Shared Memory: (2 optimizations here)

Another factor we must consider is that shared memory is actually faster than local memory for CUDA, assuming no memory bank conflicts (see https://devblogs.nvidia.com/using-shared-memory-cuda-cc/). This is because warps of the same instruction in the GPU can be executed in parallel, and storing into shared memory (and its memory banks) helps with this. If we are to cache the data into local memory, then we can promode SIMD and make our program run in parallel more efficiently.

In our shared memory implementation, another thing to note would be that by using shared memory, when loading the data, a single thread doesn't need to read all of the data into memory by itself; multiple threads can read parts of the memory and effectively increase the bandwidth in which data is being read.

Below, our non-shared implementation before optimization reads data into __private memory (or local memory in CUDA terms). This results in a much lower execution time as each thread needs to read the entiriety of the data, resulting in a performance of about ~130 GFlops, which is less than 1/4 of our original performance.

Code:

```
__constant int kNum = 256;
__constant int kKernel = 5;
__constant int kImSize = 224;
__constant int kInImSize = 228;
__constant int kOutImSize = 112;

__kernel
void CnnKernel(__global const float* input, __global const float* weight,
          __global const float* bias, __global float* output) {

__private uint blockID = get_group_id(0);
__private uint threadID = get_local_id(0);
__private uint numBlocks = get_num_groups(0);
__private uint numThreadsInBlock = get_local_size(0);

for (int h = blockID; h < kOutImSize; h += numBlocks)
{
    for (int i = threadID; i < kNum; i += numThreadsInBlock)
    {
    __private float C[2][224] = {};
    for (int j = 0; j < kNum; ++j)
    {
        __private float4 mask0 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 0]);
        __private float maskRem0 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 4];
        __private float4 mask1 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 0]);
        __private float maskRem1 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 4];
        __private float4 mask2 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 0]);
        __private float maskRem2 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 4];
        __private float4 mask3 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 0]);
        __private float maskRem3 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 4];
        __private float4 mask4 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 0]);
        __private float maskRem4 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 4];
```

```
/*
__local float inputBuffer[6][228];
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
for (int x = 0; x < 6; ++x)
for(int y = threadID; y < kInImSize; y += numThreadsInBlock)
   inputBuffer[x][y] = input[(j * kInImSize * kInImSize) + (((h << 1) + x) * kInImSize) + y];
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
*/
__private float inputBuffer[6][228];
for (int x = 0; x < 6; ++x)
{
__private float16 buf;
for(int y = 0; y < 224; y += 16)
{
   buf = vload16(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + x) * kInImSize) + y]);
   vstore16(buf, 0, &inputBuffer[x][y]);
}
__private float4 buf2 = vload4(0, &input[(j * kInImSize * kInImSize) + (((h << 1) + x) * kInImSize) + 224]);
vstore4(buf2, 0, &inputBuffer[x][224]);
}


__private float4 row0 = vload4(0, &inputBuffer[0][0]);
__private float4 row1 = vload4(0, &inputBuffer[1][0]);
__private float4 row2 = vload4(0, &inputBuffer[2][0]);
__private float4 row3 = vload4(0, &inputBuffer[3][0]);
__private float4 row4 = vload4(0, &inputBuffer[4][0]);
__private float4 ans0 = mask0 * row0;
__private float4 ans1 = mask1 * row1;
__private float4 ans2 = mask2 * row2;
__private float4 ans3 = mask3 * row3;
__private float4 ans4 = mask4 * row4;
__private float4 ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[0][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
      + (inputBuffer[0][4] * maskRem0)
      + (inputBuffer[1][4] * maskRem1)
      + (inputBuffer[2][4] * maskRem2)
      + (inputBuffer[3][4] * maskRem3)
      + (inputBuffer[4][4] * maskRem4);
for (int w = 1; w < kImSize; ++w)
{
row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[0][w + 3]);
row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[1][w + 3]);
row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[2][w + 3]);
row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[3][w + 3]);
row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[4][w + 3]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[0][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
            + (inputBuffer[0][w + 4] * maskRem0)
            + (inputBuffer[1][w + 4] * maskRem1)
            + (inputBuffer[2][w + 4] * maskRem2)
            + (inputBuffer[3][w + 4] * maskRem3)
            + (inputBuffer[4][w + 4] * maskRem4);
}

row0 = vload4(0, &inputBuffer[1][0]);
row1 = vload4(0, &inputBuffer[2][0]);
row2 = vload4(0, &inputBuffer[3][0]);
row3 = vload4(0, &inputBuffer[4][0]);
row4 = vload4(0, &inputBuffer[5][0]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[1][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
      + (inputBuffer[1][4] * maskRem0)
```

```
            + (inputBuffer[2][4] * maskRem1)
            + (inputBuffer[3][4] * maskRem2)
            + (inputBuffer[4][4] * maskRem3)
            + (inputBuffer[5][4] * maskRem4);
        for (int w = 1; w < kImSize; ++w)
        {
        row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[1][w + 3]);
        row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[2][w + 3]);
        row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[3][w + 3]);
        row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[4][w + 3]);
        row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[5][w + 3]);
        ans0 = mask0 * row0;
        ans1 = mask1 * row1;
        ans2 = mask2 * row2;
        ans3 = mask3 * row3;
        ans4 = mask4 * row4;
        ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
        C[1][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
                    + (inputBuffer[1][w + 4] * maskRem0)
                    + (inputBuffer[2][w + 4] * maskRem1)
                    + (inputBuffer[3][w + 4] * maskRem2)
                    + (inputBuffer[4][w + 4] * maskRem3)
                    + (inputBuffer[5][w + 4] * maskRem4);
        }
    }
    for (int c = 0; c < kOutImSize; ++c)
    {
        __private union {
        float F[4];
        float4 f;
        } buf;
        buf.F[0] = C[0][c * 2] + bias[i];
        buf.F[1] = C[0][(c * 2) + 1] + bias[i];
        buf.F[2] = C[1][c * 2] + bias[i];
        buf.F[3] = C[1][(c * 2) + 1] + bias[i];
        buf.f = max(buf.f, (float4) (0.0f, 0.0f, 0.0f, 0.0f));
        output[(i * kOutImSize * kOutImSize) + (h * kOutImSize) + c] = max(
                                                    max(buf.F[0], buf.F[1]),
                                                    max(buf.F[2], buf.F[3]));
    }
    }
}
}
//INSTEAD OF "__local float inputBuffer[6][228]", WE USE "__PRIVATE float inputBuffer[6][228]"
//~130 GFlops (112 x 256 = 28672)
//__local is faster than __private (if no memory bank conflicts) and is more memory efficient
```

Loop Unrolling:

Without this optimization, the performance come out to be around ~635 GFlops. However, although it didn't help too much with the execution speed, it did make our program more memory efficient (__local storage of ((224 / (# of blocks)) + 4)(228)). This increase our performance from ~630 GFlops to ~655 GFlops (not a too significant difference). Loop unrolling further lead to the same performance.

Code:

```
__constant int kNum = 256;
__constant int kKernel = 5;
__constant int kImSize = 224;
__constant int kInImSize = 228;
__constant int kOutImSize = 112;

__kernel
void CnnKernel(__global const float* input, __global const float* weight,
          __global const float* bias, __global float* output) {

__private uint blockID = get_group_id(0);
__private uint threadID = get_local_id(0);
__private uint numBlocks = get_num_groups(0);
__private uint numThreadsInBlock = get_local_size(0);

__private int offset = (112 / numBlocks); // = 2
__private int blockOffset = blockID * offset;
```

```
for (int i = threadID; i < kNum; i += numThreadsInBlock)
{
    __private float C[4][224] = {};   //2 * offset
    for (int j = 0; j < kNum; ++j)
    {
    __private float4 mask0 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 0]);
    __private float maskRem0 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + 0 + 4];
    __private float4 mask1 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 0]);
    __private float maskRem1 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (1 * kKernel) + 4];
    __private float4 mask2 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 0]);
    __private float maskRem2 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (2 * kKernel) + 4];
    __private float4 mask3 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 0]);
    __private float maskRem3 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (3 * kKernel) + 4];
    __private float4 mask4 = vload4(0, &weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 0]);
    __private float maskRem4 = weight[(i * kNum * kKernel * kKernel) + (j * kKernel * kKernel) + (4 * kKernel) + 4];

    __local float inputBuffer[8][228];    //(2 * blockOffset) + 4
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    for (int x = 0; x < 8; ++x)
        for(int y = threadID; y < kInImSize; y += numThreadsInBlock)
        inputBuffer[x][y] = input[(j * kInImSize * kInImSize) + (((blockOffset << 1) + x) * kInImSize) + y];
    barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);

    __private float4 row0 = vload4(0, &inputBuffer[0][0]);
    __private float4 row1 = vload4(0, &inputBuffer[1][0]);
    __private float4 row2 = vload4(0, &inputBuffer[2][0]);
    __private float4 row3 = vload4(0, &inputBuffer[3][0]);
    __private float4 row4 = vload4(0, &inputBuffer[4][0]);
    __private float4 ans0 = mask0 * row0;
    __private float4 ans1 = mask1 * row1;
    __private float4 ans2 = mask2 * row2;
    __private float4 ans3 = mask3 * row3;
    __private float4 ans4 = mask4 * row4;
    __private float4 ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
    C[0][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
               + (inputBuffer[0][4] * maskRem0)
               + (inputBuffer[1][4] * maskRem1)
               + (inputBuffer[2][4] * maskRem2)
               + (inputBuffer[3][4] * maskRem3)
               + (inputBuffer[4][4] * maskRem4);
    for (int w = 1; w < kImSize; ++w)
    {
        row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[0][w + 3]);
        row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[1][w + 3]);
        row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[2][w + 3]);
        row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[3][w + 3]);
        row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[4][w + 3]);
        ans0 = mask0 * row0;
        ans1 = mask1 * row1;
        ans2 = mask2 * row2;
        ans3 = mask3 * row3;
        ans4 = mask4 * row4;
        ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
        C[0][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
                   + (inputBuffer[0][w + 4] * maskRem0)
                   + (inputBuffer[1][w + 4] * maskRem1)
                   + (inputBuffer[2][w + 4] * maskRem2)
                   + (inputBuffer[3][w + 4] * maskRem3)
                   + (inputBuffer[4][w + 4] * maskRem4);
    }

    row0 = vload4(0, &inputBuffer[1][0]);
    row1 = vload4(0, &inputBuffer[2][0]);
    row2 = vload4(0, &inputBuffer[3][0]);
    row3 = vload4(0, &inputBuffer[4][0]);
    row4 = vload4(0, &inputBuffer[5][0]);
    ans0 = mask0 * row0;
    ans1 = mask1 * row1;
    ans2 = mask2 * row2;
    ans3 = mask3 * row3;
    ans4 = mask4 * row4;
    ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
```

```
C[1][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
          + (inputBuffer[1][4] * maskRem0)
          + (inputBuffer[2][4] * maskRem1)
          + (inputBuffer[3][4] * maskRem2)
          + (inputBuffer[4][4] * maskRem3)
          + (inputBuffer[5][4] * maskRem4);
for (int w = 1; w < kImSize; ++w)
{
   row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[1][w + 3]);
   row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[2][w + 3]);
   row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[3][w + 3]);
   row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[4][w + 3]);
   row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[5][w + 3]);
   ans0 = mask0 * row0;
   ans1 = mask1 * row1;
   ans2 = mask2 * row2;
   ans3 = mask3 * row3;
   ans4 = mask4 * row4;
   ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
   C[1][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
            + (inputBuffer[1][w + 4] * maskRem0)
            + (inputBuffer[2][w + 4] * maskRem1)
            + (inputBuffer[3][w + 4] * maskRem2)
            + (inputBuffer[4][w + 4] * maskRem3)
            + (inputBuffer[5][w + 4] * maskRem4);
}

row0 = vload4(0, &inputBuffer[2][0]);
row1 = vload4(0, &inputBuffer[3][0]);
row2 = vload4(0, &inputBuffer[4][0]);
row3 = vload4(0, &inputBuffer[5][0]);
row4 = vload4(0, &inputBuffer[6][0]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
ans3 = mask3 * row3;
ans4 = mask4 * row4;
ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
C[2][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
          + (inputBuffer[2][4] * maskRem0)
          + (inputBuffer[3][4] * maskRem1)
          + (inputBuffer[4][4] * maskRem2)
          + (inputBuffer[5][4] * maskRem3)
          + (inputBuffer[6][4] * maskRem4);
for (int w = 1; w < kImSize; ++w)
{
   row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[2][w + 3]);
   row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[3][w + 3]);
   row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[4][w + 3]);
   row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[5][w + 3]);
   row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[6][w + 3]);
   ans0 = mask0 * row0;
   ans1 = mask1 * row1;
   ans2 = mask2 * row2;
   ans3 = mask3 * row3;
   ans4 = mask4 * row4;
   ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
   C[2][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
            + (inputBuffer[2][w + 4] * maskRem0)
            + (inputBuffer[3][w + 4] * maskRem1)
            + (inputBuffer[4][w + 4] * maskRem2)
            + (inputBuffer[5][w + 4] * maskRem3)
            + (inputBuffer[6][w + 4] * maskRem4);
}

row0 = vload4(0, &inputBuffer[3][0]);
row1 = vload4(0, &inputBuffer[4][0]);
row2 = vload4(0, &inputBuffer[5][0]);
row3 = vload4(0, &inputBuffer[6][0]);
row4 = vload4(0, &inputBuffer[7][0]);
ans0 = mask0 * row0;
ans1 = mask1 * row1;
ans2 = mask2 * row2;
```

```
    ans3 = mask3 * row3;
    ans4 = mask4 * row4;
    ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
    C[3][0] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
               + (inputBuffer[3][4] * maskRem0)
               + (inputBuffer[4][4] * maskRem1)
               + (inputBuffer[5][4] * maskRem2)
               + (inputBuffer[6][4] * maskRem3)
               + (inputBuffer[7][4] * maskRem4);
    for (int w = 1; w < kImSize; ++w)
    {
       row0 = (float4) (row0.s1, row0.s2, row0.s3, inputBuffer[3][w + 3]);
       row1 = (float4) (row1.s1, row1.s2, row1.s3, inputBuffer[4][w + 3]);
       row2 = (float4) (row2.s1, row2.s2, row2.s3, inputBuffer[5][w + 3]);
       row3 = (float4) (row3.s1, row3.s2, row3.s3, inputBuffer[6][w + 3]);
       row4 = (float4) (row4.s1, row4.s2, row4.s3, inputBuffer[7][w + 3]);
       ans0 = mask0 * row0;
       ans1 = mask1 * row1;
       ans2 = mask2 * row2;
       ans3 = mask3 * row3;
       ans4 = mask4 * row4;
       ansTotal = ans0 + ans1 + ans2 + ans3 + ans4;
       C[3][w] += ansTotal.s0 + ansTotal.s1 + ansTotal.s2 + ansTotal.s3
                  + (inputBuffer[3][w + 4] * maskRem0)
                  + (inputBuffer[4][w + 4] * maskRem1)
                  + (inputBuffer[5][w + 4] * maskRem2)
                  + (inputBuffer[6][w + 4] * maskRem3)
                  + (inputBuffer[7][w + 4] * maskRem4);
    }
    }
    for(int r = 0; r < offset; r++)
    {
    for (int c = 0; c < kOutImSize; ++c)
    {
       __private union {
       float F[4];
       float4 f;
       } buf;
       buf.F[0] = C[(r << 1)][(c << 1)] + bias[i];
       buf.F[1] = C[(r << 1)][(c << 1) + 1] + bias[i];
       buf.F[2] = C[(r << 1) + 1][(c << 1)] + bias[i];
       buf.F[3] = C[(r << 1) + 1][(c << 1) + 1] + bias[i];
       buf.f = max(buf.f, (float4) (0.0f, 0.0f, 0.0f, 0.0f));
       output[(i * kOutImSize * kOutImSize) + ((blockOffset + r) * kOutImSize) + c] = max(
                                                                  max(buf.F[0], buf.F[1]),
                                                                  max(buf.F[2], buf.F[3]));
    }
    }
}
}
//STRAT: UNROLL LOOP h AS SHARED MEMORY inputBuffer SCALABLE
//HERE, "numBlocks" MUST BE SET AS 56
//~655 GFlops (56 x 256 = 14336)
```

Please report the number of work-groups (NOT global work size) and work-items per work-group that provides the best performance for your kernel. Then please look up the number of multiprocessors and CUDA cores per multiprocessor in the GPU of g3s.xlarge. Do the numbers match? If not, please discuss the probable reason. (Optional, bonus +2: Please include a table that shows the performance for different number of work-group and work-items per work group. Please report the performance for at least 3x3=9 different configurations including the one that provides the best performance. The spacing between different work-group/work-item should be around 2X different - e.g. #work-group- 8,16,32.)

For our implementation, a work-group count of 56 and a work-item count of 256 seems to work the best. We know that the GPU has 16 SM's and 128 cores/SM (with a maximum of 2048 threads per SM). With out block and thread count, we would have a total of 14336 threads in total. The number don't seem to match completely (even if we are to lower the block and thread count, the performance seems to stay relatively constant) (see below). Then again, we aren't given how many threads a single core is able to execute, and what is the optimum number of threads to assign to a single core.

(using the non loop-unrolled version)

(Work Group) x (Work-Items)	GFlops
28x256	631.379 GFlops
56x256	638.261 GFlops
112x256	639.187 GFlops
28x128	643.454 GFlops
56x128	638.569 GFlops
112x128	645.156 GFlops
28x64	514.259 GFlops
56x64	655.969 GFlops
112x64	635.992 GFlops