|  |  |
| --- | --- |
| **Name:** | *Dongming Liu* |
| **NetID:** | *dl35* |
| **Section:** | *AL* |

**ECE 408/CS483 Milestone 3 Report**

|  |
| --- |
| 1. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.175719 ms* | *0.633705 ms* | *0m1.220s* | *0.86* | | 1000 | *1.59055 ms* | *6.10214 ms* | *0m9.758s* | *0.886* | | 10000 | *15.7009 ms* | *60.7863 ms* | *1m35.228s* | *0.8714* | |
| 1. **Optimization 1: Weight matrix in constant memory (1 points)** |
| * 1. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique. |
| Tiled shared memory convolution (**2 points**)  Shared memory matrix multiplication and input matrix unrolling (**3 points**)  Kernel fusion for unrolling and matrix-multiplication (**2 points**)  Weight matrix in constant memory (**1 point**)  Tuning with restrict and loop unrolling (**3 points**)  Sweeping various parameters to find best values (**1 point**)  Multiple kernel implementations for different layer sizes (**1 point**)  Input channel reduction: tree (**3 point**)  Input channel reduction: atomics (**2 point**)  Fixed point (FP16) arithmetic. (**4 points**)  Using Streams to overlap computation with data transfer (**4 points**)  An advanced matrix multiplication algorithm (**5 points**)  Using Tensor Cores to speed up matrix multiplication (**5 points**)  Overlap-Add method for FFT-based convolution (**8 points**)  Other optimizations: please explain  *I select weight matrix in constant memory as one of my optimizations. I picked this one because it is very straightforward, and this technique appears in one of the previous MPs.*   * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?   *The optimization works by transferring the weight mask to the constant memory and reducing global memory access. Since the convolution kernel never updates the mask, it is better to store the weight mask on constant memory. I thought that this technique would increase the performance of the forward convolution. Constant memory is faster than global memory in general, so using constant memory can reduce global memory access. This is the first optimization.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.166936* | *0.570333 ms* | *0m1.204s* | *0.86* | | 1000 | *1.47756 ms* | *5.6494 ms* | *0m9.715s* | *0.886* | | 10000 | *14.6025 ms* | *56.799 ms* | *1m35.093s* | *0.8714* |  * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of)   *The optimization was successful in improving performance.*      *Kernel statistics in m2 implementation*    *Kernel statistics in optimization 1*  *Based on the result from nsys, the conv\_forward\_kernel total time of optimization 1 is 72.47 ms in total which is less than that of m2 implementation, 76.77 ms.*    *Left: m2 implementation, right: optimization 1*  *The SM utilization of optimization 1 is also higher than the SM utilization in m2. This indicates that optimization 1 can use SM more efficiently to compute the convolution output.* |
| * 1. What references did you use when implementing this technique? |
| *MP4: 3D convolution*  *Lectures: lecture 7 and lecture 8*     * 1. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.   For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling  *Code can be found under a folder called optimize in code submission.*  *Declaration of constant memory, line 6*    *Copy the mask to constant memory, line 87*    *Kernel code* |

|  |
| --- |
| 1. **Optimization 2: tile shared memory convolution (2 points)** |
| * 1. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique. |
| Tiled shared memory convolution (**2 points**)  Shared memory matrix multiplication and input matrix unrolling (**3 points**)  Kernel fusion for unrolling and matrix-multiplication (**2 points**)  Weight matrix in constant memory (**1 point**)  Tuning with restrict and loop unrolling (**3 points**)  Sweeping various parameters to find best values (**1 point**)  Multiple kernel implementations for different layer sizes (**1 point**)  Input channel reduction: tree (**3 point**)  Input channel reduction: atomics (**2 point**)  Fixed point (FP16) arithmetic. (**4 points**)  Using Streams to overlap computation with data transfer (**4 points**)  An advanced matrix multiplication algorithm (**5 points**)  Using Tensor Cores to speed up matrix multiplication (**5 points**)  Overlap-Add method for FFT-based convolution (**8 points**)  Other optimizations: please explain  *I chose Tile shared memory convolution because it appeared on one of the previous MPs. In addition, previous lectures talked about this technique. I thought it was easy to implement.*   * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?   *Optimization 2 works by preloading data from global memory to shared memory. During the calculations of the convolution output, the threads can directly retrieve the data from shared memory rather than from the global memory. I thought this optimization would increase the performance of the forward convolution because shared memory is much faster than global memory. This can reduce global memory access and optimize memory utilization. This optimization synergizes with optimization 1: Weight Matrix in constant memory.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.164547 ms* | *0.657649 ms* | *0m1.244s* | *0.86* | | 1000 | *1.55923 ms* | *6.53511 ms* | *0m9.735s* | *0.886* | | 10000 | *15.415 ms* | *64.9177 ms* | *1m34.864s* | *0.8714* |  * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of   *The optimization was not successful in improving performance.*    *Optimization 1 on baseline m2, nsys statistics*    *Optimization 1 & 2 on baseline m2, nsys statistics*  *Compared with optimization 1, optimization 1&2 spent less time on cudaMemcpy and cudaMalloc. However, the total time of conv\_foward\_kernel in optimization 1&2 is higher than that of conv\_forward\_kernel in optimization 1.*    *Optimiztion 1 on baseline m2, memory statistics*    *Optimization 1 & 2 on baseline: memory statistics*  *Compared with Optimization 1, the memory throughput in optimization 1&2 (273.91 GB/s) is lower than that of optimization 1 (286.69 GB/s). Even though optimization 2 has a higher L2 Hit Rate (44.56%), it has a lower L1 Hit Rate (23.34%). This is one factor that this optimization did not have a great performance.*    *Optimization 1 on baseline: warp statistics*    *Optimization 1&2 on baseline: warp statistics*  *Compared with optimization 1, optimization 1 & 2 has fewer average active threads per warp. Optimization 1 has an average of 32, and optimization 2 has an average of 30.64. The warp cycles per issued/executed instruction in optimization 2 are not as good as the ones in optimization 2. This might be caused by the kernel code that transfers data from global memory to shared memory. Not all threads in a warp would transfer the data to the shared memory, which introduced another set of control divergences to my kernel and lowered the number of active threads. This would also bring down the performance.* |
| * 1. What references did you use when implementing this technique? |
| *MP4: 3D convolution*  *Lectures: lecture 7 and lecture 8*  *Using Shared Memory in CUDA C/C++: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/*   * 1. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.   For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling  *Code can be found under a folder called optimize in code submission.*  *Dynamically shared memory allocation, line 109.*    *Kernel code:*  #define TILE\_WIDTH 16  \_\_constant\_\_ float kernel\_mask[1 \* 7 \* 7 \* 4 \* 16];  \_\_global\_\_ void conv\_forward\_kernel(float \*output, const float \*input, const float \*mask, const int Batch, const int Map\_out, const int Channel, const int Height, const int Width, const int K)  {      const int Height\_out = Height - K + 1;      const int Width\_out = Width - K + 1;      const int W\_grid = ceil((Width\_out\*1.0)/TILE\_WIDTH);      const int blocksize = TILE\_WIDTH + K - 1;      extern \_\_shared\_\_ float tileMem[]; // shared memory      #define out\_4d(i3, i2, i1, i0) output[(i3) \* (Map\_out \* Height\_out \* Width\_out) + (i2) \* (Height\_out \* Width\_out) + (i1) \* (Width\_out) + i0]      #define in\_4d(i3, i2, i1, i0) input[(i3) \* (Channel \* Height \* Width) + (i2) \* (Height \* Width) + (i1) \* (Width) + i0]      #define mask\_4d(i3, i2, i1, i0) kernel\_mask[(i3) \* (Channel \* K \* K) + (i2) \* (K \* K) + (i1) \* (K) + i0]      #define shareMem(i2, i1, i0) tileMem[(i2)\*(blocksize\*blocksize) + (i1) \* blocksize + (i0)] // use for 3d shared memory      // Insert your GPU convolution kernel code here      int tx = threadIdx.x;      int ty = threadIdx.y;      int c;  // image\_index, map\_inedx, specific\_height, specific\_width, channel      int h\_topleft= (blockIdx.z / W\_grid) \* TILE\_WIDTH;      int w\_topleft = (blockIdx.z % W\_grid) \* TILE\_WIDTH;      int n = blockIdx.x;      int m = blockIdx.y;      int h = h\_topleft + ty; // the output height index      int w = w\_topleft + tx; // the output width index      float acc = 0.0;        // with shared tiles, may be able to reduce sync\_threads      for(c = 0; c < Channel; ++c){          for(int i = ty; i < blocksize; i+= TILE\_WIDTH){              for(int j = tx; j < blocksize; j += TILE\_WIDTH){                  if(h\_topleft + i < Height && w\_topleft + j < Width){                      shareMem(c,i,j) = in\_4d(n, c, h\_topleft + i, w\_topleft + j);                  }                  else{                      shareMem(c,i,j) = 0.0f;                  }              }          }      }      \_\_syncthreads();      if (h < Height\_out && w < Width\_out){          for(c = 0; c < Channel; ++c){              for(int p = 0; p < K; ++p){     // for loop, the mask K x K                  for(int q = 0; q < K; ++q){                      acc += shareMem(c ,ty+p , tx+q) \* mask\_4d(m,c,p,q);                      // acc += in\_4d(n, c, h+p, w+q) \* mask\_4d(m,c,p,q);                  }              }          }          out\_4d(n,m,h,w) = acc;      }      #undef out\_4d      #undef in\_4d      #undef mask\_4d      #undef shareMem  } |

|  |
| --- |
| 1. **Optimization 3: Fixed point (FP16) arithmetic (4 points)** |
| * 1. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique. |
| Tiled shared memory convolution (**2 points**)  Shared memory matrix multiplication and input matrix unrolling (**3 points**)  Kernel fusion for unrolling and matrix-multiplication (**2 points**)  Weight matrix in constant memory (**1 point**)  Tuning with restrict and loop unrolling (**3 points**)  Sweeping various parameters to find best values (**1 point**)  Multiple kernel implementations for different layer sizes (**1 point**)  Input channel reduction: tree (**3 point**)  Input channel reduction: atomics (**2 point**)  Fixed point (FP16) arithmetic. (**4 points**)  Using Streams to overlap computation with data transfer (**4 points**)  An advanced matrix multiplication algorithm (**5 points**)  Using Tensor Cores to speed up matrix multiplication (**5 points**)  Overlap-Add method for FFT-based convolution (**8 points**)  Other optimizations: please explain  *I chose this optimization because I thought this optimization would be interesting since it would change the accuracy according to the final project GitHub. In addition, I found relevant documents on FP16 arithmetic. I thought that FP16 might have some advantage over float in calculation and size.*   * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?   *The optimization would convert the floating points number into a new type \_\_half (FP16). Instead of using floating point arithmetic in calculating the convolution output, the kernel calculates the convolution output with FP16 arithmetic. In the end, the result will convert back to float. I think this will improve the performance of the forward convolution. FP16 is smaller than float, which means FP16 may have a faster computation speed due to the advantage of a smaller data size. Also, FP16 requires less memory than float. I thought this optimization synergizes with optimization 1.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.213373 ms* | *0.791051 ms* | *0m1.248s* | *0.86* | | 1000 | *1.99508 ms* | *7.73802 ms* | *0m10.106s* | *0.887* | | 10000 | *19.7247 ms* | *77.7143 ms* | *1m36.354s* | *0.8716* |  * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off)     *This optimization was not successful in improving performance. This optimization was built on top of optimization 1, and the Op Time for optimization 1 is much better. The accuracy was slightly higher than the accuracy in the previous implementation. This was caused by the precision of calculations and rounding in FP16.*    *Optimization 1 & 3 on baseline m2, took 98.33ms*    *Optimization 1 on baseline m2, took 72.47ms*  *Based on the data from nsys, the time that conv\_forward\_kernel spent in optimization 1 & 3 is larger than that of optimization 1.*    *Blue: Optimization 1 & 3 on baseline m2, Orange: Optimization 1 on baseline m2*    *Memory statistics of Optimization 1&3 on baseline m2*  *Based on the result from the Nsight Compute, optimization 1&3 has a lower memory utilization and higher SM utilization. This indicates that optimization 1&3 does less memory in the kernel. However, the memory throughput of optimization 1&3 is 25.37% lower than the memory throughput of optimization 1.*    *Warp statistics and instruction statistics of Optimization 1&3 on baseline*  *The warp cycles per issue/execute instruction in optimization 1&3 are higher than that of optimization 1, so optimization 1&3 would have a longer instruction execution time. From the instruction statistics, the number of instructions in optimization 1&3 is 16.78% higher than the number of instructions in optimization 1. This is mainly caused by the conversion between float and FP16 (\_\_half) data type. Although the FP16 data type may have some advantages over float type, the overheads of float to FP16 conversions and FP16 to float conversions cannot be avoided in optimization 1&3. Thus, the performance of optimization 1&3 is lower.* |
| * 1. What references did you use when implementing this technique? |
| *CUDA Toolkit documentation:*  *https://docs.nvidia.com/cuda/cuda-math-api/index.html*   * 1. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.   For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling  *Code can be found under a folder called optimize in code submission.*  *Using a new library, cuda\_fp16.h*    *Kernel code*  #define TILE\_WIDTH 16  \_\_constant\_\_ float kernel\_mask[1 \* 7 \* 7 \* 4 \* 16];  \_\_global\_\_ void conv\_forward\_kernel(float \*output, const float \*input, const float \*mask, const int Batch,      const int Map\_out, const int Channel, const int Height, const int Width, const int K)  {      const int Height\_out = Height - K + 1;      const int Width\_out = Width - K + 1;      const int W\_grid = ceil((Width\_out\*1.0)/TILE\_WIDTH);      #define out\_4d(i3, i2, i1, i0) output[(i3) \* (Map\_out \* Height\_out \* Width\_out) + (i2) \* (Height\_out \* Width\_out) + (i1) \* (Width\_out) + i0]      #define in\_4d(i3, i2, i1, i0) input[(i3) \* (Channel \* Height \* Width) + (i2) \* (Height \* Width) + (i1) \* (Width) + i0]      #define mask\_4d(i3, i2, i1, i0) kernel\_mask[(i3) \* (Channel \* K \* K) + (i2) \* (K \* K) + (i1) \* (K) + i0]      // Insert your GPU convolution kernel code here        int n, m, h, w, c;  // image\_index, map\_inedx, specific\_height, specific\_width, channel      n = blockIdx.x;      m = blockIdx.y;      h = (blockIdx.z / W\_grid) \* TILE\_WIDTH + threadIdx.y;      w = (blockIdx.z % W\_grid) \* TILE\_WIDTH + threadIdx.x;      \_\_half acc = \_\_float2half\_rn(0.0f);      if(h < Height\_out && w < Width\_out){          for(c = 0; c < Channel; ++c){              for(int p = 0; p < K; ++p){     // for loop, the mask K x K                  for(int q = 0; q < K; ++q){                      acc = \_\_hadd(acc, \_\_hmul(\_\_float2half\_rn(in\_4d(n, c, h+p, w+q)), \_\_float2half\_rn(mask\_4d(m,c,p,q))));                  }              }          }          out\_4d(n,m,h,w) = \_\_half2float(acc);      }        #undef out\_4d      #undef in\_4d      #undef mask\_4d  } |

|  |
| --- |
| 1. **Optimization 4: Using Streams to overlap computation with data transfer (4 point)** |
| * 1. Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique. |
| Tiled shared memory convolution (**2 points**)  Shared memory matrix multiplication and input matrix unrolling (**3 points**)  Kernel fusion for unrolling and matrix-multiplication (**2 points**)  Weight matrix in constant memory (**1 point**)  Tuning with restrict and loop unrolling (**3 points**)  Sweeping various parameters to find best values (**1 point**)  Multiple kernel implementations for different layer sizes (**1 point**)  Input channel reduction: tree (**3 point**)  Input channel reduction: atomics (**2 point**)  Fixed point (FP16) arithmetic. (**4 points**)  Using Streams to overlap computation with data transfer (**4 points**)  An advanced matrix multiplication algorithm (**5 points**)  Using Tensor Cores to speed up matrix multiplication (**5 points**)  Overlap-Add method for FFT-based convolution (**8 points**)  Other optimizations: please explain  *I choose using streams to overlap computation with data transfer because there is a lecture talk about this. Implementing streams can increase the overall performance and data transfer rate.*   * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?   *This optimization works by creating multiple streams to overlap kernel execution and data transfer. A large chunk of data is broken into smaller chunks and distributed among the streams. I thought this would increase the overall performance. Some streams might execute the kernel whereas other streams transfer the data and prepare for kernel execution. This creates parallelism among the streams and reduces a certain amount of latency in data transfer. I think the optimization synergizes with optimization 1 and produces better performance.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.003278 ms* | *0.002823 ms* | *0m1.236s* | *0.86* | | 1000 | *0.002481 ms* | *0.003544 ms* | *0m9.889s* | *0.886* | | 10000 | *0.003199 ms* | *0.003446 ms* | *1m38.379s* | *0.8714* |  * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of   *The optimization was not very successful in improving performance. Based on the nsys output data below, even though optimization 1&4 have better total time in memory operation (454.43 ms) than the total time in memory operation of optimization 1 (1040.49 ms), the total time in conv\_forward\_kernel of optimization 1&4*  *(86.61 ms) did not beat the total time of optimization 1 (72.47ms).*    *Optimization 1&4 on baseline m2*    *Optimization 1 on baseline m2*    *Blue: optimization 1&4 on baseline m2, Red: optimization 1 on baseline m2*    *Optimization 1&4 on baseline m2 compared with Optimization 1 on m2*  *Based on the information from Nsight Compute, both utilization on SM and memory in optimization decreases. The memory throughput of each kernel launch in optimization 1&4 is not as good as the one in optimization 1. The good things are that the L1 and L2 hit rate in optimization 1&4 are high. The decrease in the performance may come from the number of streams that I used in my code and data access patterns. The number of streams that I picked may not be optimized one for this convolution kernel. The amount of overlapping between kernel execution and data transfer is not sufficient.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture 22*   * 1. Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.   For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling  *Code can be found under a folder called optimize in code submission.*  *Kernel code*  #define TILE\_WIDTH 16  #define STREAM\_NUM  4  \_\_constant\_\_ float kernel\_mask[1 \* 7 \* 7 \* 4 \* 16];  \_\_global\_\_ void conv\_forward\_kernel(float \*output, const float \*input, const float \*mask, const int Batch,      const int Map\_out, const int Channel, const int Height, const int Width, const int K)  {      const int Height\_out = Height - K + 1;      const int Width\_out = Width - K + 1;  const int W\_grid = ceil((Width\_out\*1.0)/TILE\_WIDTH);      #define out\_4d(i3, i2, i1, i0) output[(i3) \* (Map\_out \* Height\_out \* Width\_out) + (i2) \* (Height\_out \* Width\_out) + (i1) \* (Width\_out) + i0]      #define in\_4d(i3, i2, i1, i0) input[(i3) \* (Channel \* Height \* Width) + (i2) \* (Height \* Width) + (i1) \* (Width) + i0]      #define mask\_4d(i3, i2, i1, i0) kernel\_mask[(i3) \* (Channel \* K \* K) + (i2) \* (K \* K) + (i1) \* (K) + i0]      // Insert your GPU convolution kernel code here        int n, m, h, w, c;  // image\_index, map\_inedx, specific\_height, specific\_width, channel      n = blockIdx.x;      m = blockIdx.y;      h = (blockIdx.z / W\_grid) \* TILE\_WIDTH + threadIdx.y;      w = (blockIdx.z % W\_grid) \* TILE\_WIDTH + threadIdx.x;      float acc = 0;      if(h < Height\_out && w < Width\_out){          for(c = 0; c < Channel; ++c){              for(int p = 0; p < K; ++p){     // for loop, the mask K x K                  for(int q = 0; q < K; ++q){                      acc += in\_4d(n, c, h+p, w+q) \* mask\_4d(m,c,p,q);                  }              }          }          out\_4d(n,m,h,w) = acc;      }        #undef out\_4d      #undef in\_4d      #undef mask\_4d  }  *create streams and declare some useful variables*  *The for loop for data transfer and kernel launch with streams*    *Other changed part of the code* |