|  |  |
| --- | --- |
| **Name:** | *Mohnish Kalia* |
| **NetID:** | *mkalia2* |
| **Section:** | *CS 483 AL1 (76324)* |

**ECE 408/CS483 Milestone 3 Report**

|  |
| --- |
| 1. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai\_amd64\_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.262705 ms* | *0.957244 ms* | *0m0.160s* | *0.86* | | 1000 | *2.39847 ms* | *9.13148 ms* | *0m0.257s* | *0.886* | | 5000 | *11.8805 ms* | *45.393 ms* | *0m0.804s* | *0.871* | |
| 1. **Optimization 1: *Using Streams to overlap computation with data transfer (4 points)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose to implement the streams to overlap computation with data transfer. In my mind, this done as I felt this could have real efficacy when scaled up to larger batch sizes, as it utilizes GPU resources even harder than a single kernel launch, while also freeing up the host code to get its end of the bargain (memory control) out of the way. The way we implemented it in class examples was very similar to how I did it, as I noticed the input and output data is able to be segmented over the image batches with minimal changes to the actual kernel, allowing for progressive data loads and reads over multiple kernel launches in a sort of batches of batches approach.* |
| * 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 starts with modifying the kernel to be launched multiple times by adding a number of streams to the host code using the cudaStreamCreate fns. Thru this, we are able to use the cudaMemcpyAsync api to achieve asynchronism with memcpys in HtoD direction right before a kernel launch is needed, and a DtoH direction right afterwords. This is while other sections of the overall Project Milestone 2 (PM2) kernel are also running at the same time. By splitting up the B batch processed images over NUM\_STREAMS streams, I can run multiple kernel independently. By computing the exact amount of data needed for each one of those kernel launches, I was able to organize a set of memcpys in the loop to insert/extract data from the kernels while they are running asynchronously. This involved a rather involved piece of host code logic to account for leftover (not nicely sized) workloads. The kernel required a small adjustment to take in a streamNum in order to figure out which B image batch a thread block is running on, replacing the standard blockIdx.z identifier. Code for the involved is below and in the stream-new-forward.cu file in the submission.*  #include <cmath>  #include <iostream>  #include "gpu-new-forward.h"  #define TILE\_WIDTH 16  #define NUM\_STREAMS 3  \_\_global\_\_ void conv\_forward\_kernel(const int *streamNum*, float \* \_\_restrict\_\_ *output*, const float \* \_\_restrict\_\_ *input*, const float \* \_\_restrict\_\_ *mask*, const int *B*, const int *M*, const int *C*, const int *H*, const int *W*, const int *K*,const int *S*)  {  */\**  *Modify this function to implement the forward pass described in Chapter 16.*  *We have added an additional dimension to the tensors to support an entire mini-batch*  *The goal here is to be correct AND fast.*  *Function paramter definitions:*  *output - output*  *input - input*  *mask - convolution kernel*  *B - batch\_size (number of images in x)*  *M - number of output feature maps*  *C - number of input feature maps*  *H - input height dimension*  *W - input width dimension*  *K - kernel height and width (K x K)*  *S - stride step length*  *\*/*  const int H\_out = (H - K)/S + 1;  const int W\_out = (W - K)/S + 1;  *// We have some nice #defs for you below to simplify indexing. Feel free to use them, or create your own.*  *// An example use of these macros:*  *// float a = in\_4d(0,0,0,0)*  *// out\_4d(0,0,0,0) = a*  #define out\_4d(*i3*, *i2*, *i1*, *i0*) output[(i3) \* (M \* H\_out \* W\_out) + (i2) \* (H\_out \* W\_out) + (i1) \* (W\_out) + i0]  #define in\_4d(*i3*, *i2*, *i1*, *i0*) input[(i3) \* (C \* H \* W) + (i2) \* (H \* W) + (i1) \* (W) + i0]  #define mask\_4d(*i3*, *i2*, *i1*, *i0*) mask[(i3) \* (C \* K \* K) + (i2) \* (K \* K) + (i1) \* (K) + i0]  *// Insert your GPU convolution kernel code here*  *// same as grid setup*  int W\_size = ceil(1.0f\*W\_out/TILE\_WIDTH); *// number of horizontal tiles per output map*  int H\_size = ceil(1.0f\*H\_out/TILE\_WIDTH); *// number of vertical tiles per output map*  int b = blockIdx.z + (ceil(1.0f \* B / NUM\_STREAMS) \* streamNum); *// batch num is based on streamNum iteration*  int m = blockIdx.x;  int h = (blockIdx.y / W\_size) \* TILE\_WIDTH + threadIdx.y; *// target h of output*  int w = (blockIdx.y % W\_size) \* TILE\_WIDTH + threadIdx.x; *// target w of output*  *// each thread ran should be within output bounds, otherwise return*  *// b >= B check because splitting of grid Z may not be bounded by B*  if (w < 0 || w >= W\_out || h < 0 || h >= H\_out || b >= B)  return;  float acc = 0.0f;  for (int c = 0; c < C; c++) { *// sum over all input channels*  for (int p = 0; p < K; p++) { *// loop over KxK filter*  for (int q = 0; q < K; q++) {  int h\_idx = (h \* S + p);  int w\_idx = (w \* S + q);  if (!(w\_idx < 0 || w\_idx >= W || h\_idx < 0 || h\_idx >= H)) {  acc += in\_4d(b, c, h\_idx, w\_idx) \* mask\_4d(m, c, p, q);  }  }  }  }  *// after accumulating, set to output value*  out\_4d(b, m, h, w) = acc;  *// atomicAdd(&(out\_4d(b, m, h, w)), acc);*  #undef out\_4d  #undef in\_4d  #undef mask\_4d  }    \_\_host\_\_ void GPUInterface::conv\_forward\_gpu\_prolog(const float \**host\_output*, const float \**host\_input*, const float \**host\_mask*, float \*\**device\_output\_ptr*, float \*\**device\_input\_ptr*, float \*\**device\_mask\_ptr*, const int *B*, const int *M*, const int *C*, const int *H*, const int *W*, const int *K*, const int *S*)  {  *// Allocate memory and copy over the relevant data structures to the GPU*  *// We pass double pointers for you to initialize the relevant device pointers,*  *// which are passed to the other two functions.*  *// Useful snippet for error checking*  *// cudaError\_t error = cudaGetLastError();*  *// if(error != cudaSuccess)*  *// {*  *// std::cout<<"CUDA error: "<<cudaGetErrorString(error)<<std::endl;*  *// exit(-1);*  *// }*  #define wbCheck(*stmt*) \  do { \  cudaError\_t err = stmt; \  if (err != cudaSuccess) { \  std::cout<<"Failed to run stmt: "<<#stmt<<std::endl; \  std::cout<<"CUDA error: "<<cudaGetErrorString(err)<<std::endl; \  exit(-1); \  } \  } while (0)  const int H\_out = (H - K)/S + 1;  const int W\_out = (W - K)/S + 1;  size\_t dop\_sz = B \* M \* H\_out \* W\_out \* sizeof(float);  size\_t dip\_sz = B \* C \* H \* W \* sizeof(float);  size\_t dmp\_sz = M \* C \* K \* K \* sizeof(float);  *// device mask stream/event*  cudaStream\_t dms;  cudaStreamCreate(&dms);  cudaEvent\_t dme;  cudaEventCreate(&dme);  *// would be async but we need CUDA 11.3+ for that. Minimal impact anyways*  wbCheck(cudaMalloc((void \*\*)device\_output\_ptr, dop\_sz));  wbCheck(cudaMalloc((void \*\*)device\_input\_ptr, dip\_sz));  wbCheck(cudaMalloc((void \*\*)device\_mask\_ptr, dmp\_sz));  *// async memcpy here, record event after done to signal before kernel*  wbCheck(cudaMemcpyAsync(\*device\_mask\_ptr, host\_mask, dmp\_sz, cudaMemcpyHostToDevice, dms));  cudaEventRecord(dme, dms);  int streamSize = ceil(1.0f \* B / NUM\_STREAMS); *// proportion of B to batch process in each stream*  int W\_size = ceil(1.0f\*W\_out/TILE\_WIDTH); *// number of horizontal tiles per output map*  int H\_size = ceil(1.0f\*H\_out/TILE\_WIDTH); *// number of vertical tiles per output map*  int tileNums = H\_size \* W\_size; *// total number of tiles per map*  dim3 DimBlock(TILE\_WIDTH, TILE\_WIDTH, 1); *// output tile for untiled code*  dim3 DimGrid(M, tileNums, streamSize);  std::cout<<"DimBlock: "<<DimBlock.x<<"x"<<DimBlock.y<<"x"<<DimBlock.z<<std::endl;  std::cout<<"DimGrid: "<<DimGrid.x<<"x"<<DimGrid.y<<"x"<<DimGrid.z<<std::endl;  cudaStream\_t\* streams = (cudaStream\_t\*) malloc(NUM\_STREAMS \* sizeof(cudaStream\_t));  for (int streamNum = 0; streamNum < NUM\_STREAMS; streamNum++) {  cudaStreamCreate(&(streams[streamNum]));  }  bool ended = false;  for (int streamNum = 0; streamNum < NUM\_STREAMS; streamNum++) {  int offset = streamNum \* streamSize; *// number of Bs to skip for start of stream*  int outStreamBytes = streamSize \* M \* H\_out \* W\_out \* sizeof(float); *// how many Bs to transfer back to host*  int inStreamBytes = streamSize \* C \* H \* W \* sizeof(float); *// how many Bs to transfer to kernel*  *// compute if the stream bytes on top of the offset will overflow the max bounds sizes for input and output*  int outDiff = dop\_sz - ((offset \* M \* H\_out \* W\_out \* sizeof(float)) + outStreamBytes);  int inDiff = dip\_sz - ((offset \* C \* H \* W \* sizeof(float)) + inStreamBytes);  *// if we already ended, just skip this stream*  if (ended)  continue;  *// if we are at or past the max input or output bounds, and we have not reached the end, mark as ended*  if((outDiff <= 0 || inDiff <= 0) && !ended)  ended = true;  *// logging*  *// std::cout<<"Starting stream: "<<streamNum<<", offset: "<< offset << ", outB: "<< outStreamBytes << ", inB: "<< inStreamBytes <<", Ended: "<<ended<<std::endl;*  *// if we go past max output bounds, add the negative diff back (clamping op)*  if(outDiff < 0)  outStreamBytes += outDiff;  *// same for input bounds*  if(inDiff < 0)  inStreamBytes += inDiff;  *// general "stacking" idea from Mark Harris of Nvidia https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/*  *// changed up drastically to account for not-nice parameters and leftover work conditions, along with wait event*  cudaMemcpyAsync(&(\*device\_input\_ptr)[offset \* C \* H \* W], &host\_input[offset \* C \* H \* W], inStreamBytes, cudaMemcpyHostToDevice, streams[streamNum]);  cudaStreamWaitEvent(streams[streamNum], dme, 0);  conv\_forward\_kernel<<<DimGrid, DimBlock, 0, streams[streamNum]>>>(streamNum, \*device\_output\_ptr, \*device\_input\_ptr, \*device\_mask\_ptr, B, M, C, H, W, K, S);  cudaMemcpyAsync((void \*) &host\_output[offset \* M \* H\_out \* W\_out], &(\*device\_output\_ptr)[offset \* M \* H\_out \* W\_out], outStreamBytes, cudaMemcpyDeviceToHost, streams[streamNum]);  }  for (int streamNum = 0; streamNum < NUM\_STREAMS; streamNum++) {  cudaStreamDestroy(streams[streamNum]);  }  cudaEventDestroy(dme);  cudaStreamDestroy(dms);  *// not needed because we sync after anyways*  *// cudaDeviceSynchronize();*  free(streams);  #undef wbCheck  }  \_\_host\_\_ void GPUInterface::conv\_forward\_gpu(float \**device\_output*, const float \**device\_input*, const float \**device\_mask*, const int *B*, const int *M*, const int *C*, const int *H*, const int *W*, const int *K*, const int *S*)  {  *// Set the kernel dimensions and call the kernel*  *// due to limitations of this fn definition, the logic for running the kernels is all in the prolog fn*  }  \_\_host\_\_ void GPUInterface::conv\_forward\_gpu\_epilog(float \**host\_output*, float \**device\_output*, float \**device\_input*, float \**device\_mask*, const int *B*, const int *M*, const int *C*, const int *H*, const int *W*, const int *K*, const int *S*)  {  #define wbCheck(*stmt*) \  do { \  cudaError\_t err = stmt; \  if (err != cudaSuccess) { \  std::cout<<"Failed to run stmt: "<<#stmt<<std::endl; \  std::cout<<"CUDA error: "<<cudaGetErrorString(err)<<std::endl; \  exit(-1); \  } \  } while (0)  *// All we need to do here is cleanup*  *// Free device memory*  wbCheck(cudaFree(device\_input));  wbCheck(cudaFree(device\_output));  wbCheck(cudaFree(device\_mask));  #undef wbCheck  }  \_\_host\_\_ void GPUInterface::get\_device\_properties()  {  int deviceCount;  cudaGetDeviceCount(&deviceCount);  for(int dev = 0; dev < deviceCount; dev++)  {  cudaDeviceProp deviceProp;  cudaGetDeviceProperties(&deviceProp, dev);  std::cout<<"Device "<<dev<<" name: "<<deviceProp.name<<std::endl;  std::cout<<"Computational capabilities: "<<deviceProp.major<<"."<<deviceProp.minor<<std::endl;  std::cout<<"Max Global memory size: "<<deviceProp.totalGlobalMem<<std::endl;  std::cout<<"Max Constant memory size: "<<deviceProp.totalConstMem<<std::endl;  std::cout<<"Max Shared memory size per block: "<<deviceProp.sharedMemPerBlock<<std::endl;  std::cout<<"Max threads per block: "<<deviceProp.maxThreadsPerBlock<<std::endl;  std::cout<<"Max block dimensions: "<<deviceProp.maxThreadsDim[0]<<" x, "<<deviceProp.maxThreadsDim[1]<<" y, "<<deviceProp.maxThreadsDim[2]<<" z"<<std::endl;  std::cout<<"Max grid dimensions: "<<deviceProp.maxGridSize[0]<<" x, "<<deviceProp.maxGridSize[1]<<" y, "<<deviceProp.maxGridSize[2]<<" z"<<std::endl;  std::cout<<"Warp Size: "<<deviceProp.warpSize<<std::endl;  }  }  *I thought it would increase performance simply on the fact that we are utilizing more of the GPUs resources in order to produce the same outputs. The overhead did not seem too large in theory, but I supposed the smaller batch sizes left the optimization in a tough spot of not being able to show its efficacy.*  *It is able to synergize with any other optimization that happens on the level of the base batch image units, so fp16 optimizations or constant memory would work with this approach as well.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *7.37061 ms* | *6.46393 ms* | *0m0.204s* | *0.86* | | 1000 | *63.838 ms* | *53.3827 ms* | *0m0.320s* | *0.886* | | 5000 | *309.101 ms* | *263.341 ms* | *0m0.772s* | *0.871* | |  | *NOTE: layer time used, optimes are super small* |  |  |  | |
| * 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). |
| *<answer here>* |
| * 1. What references did you use when implementing this technique? |
| *The refs here were NVIDIA based. The general "stacking" idea of overlaying the memcpys and kernels were from Mark Harris of Nvidia* [*https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/*](https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/) *. Runtime api docs were also used here* [*https://docs.nvidia.com/cuda/cuda-runtime-api/group\_\_CUDART\_\_STREAM.html#group\_\_CUDART\_\_STREAM\_1g7840e3984799941a61839de40413d1d9*](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html#group__CUDART__STREAM_1g7840e3984799941a61839de40413d1d9) *. And finally, this do/don’ts of streams by* Justin Luitjens for debugging and optimizing the code <https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf> |
| 1. **Optimization 2: *FP16 arithmetic. (note this can modify model accuracy slightly) (4 points)*** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose the FP16 arithmetic optimization because since the GPU is such a compute oriented device, it make sense to avoid working with abstractions such as FP32 in order to get more lower level control of the ops and utilize some more primitive instructions.* |
| 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 is within the fp16-new-forward.cu file in the submission.*  *The optimization starts with modifying the host code to convert all the float input and mask data from float to \_\_half data type via the \_\_**float2half() fn. After that, I took memcpyed the data to the device ptrs and fed it to the kernel, which now accepts \_\_restrict\_\_ \_\_half ptrs, the \_\_restrict\_\_ to hopefully squeeze out more optimization over our assurance that the ptrs do not interleave. From there, the main operation to optimize the the multiply from the input to the mask, then the add to the output. This was able to be done in one fused op instruction via the \_\_hfma() fn, which deals with the FP16 floats. After that, some more host code to copy back the \_\_half ptr output and then a set of \_\_half2float() in order to convert back into the host\_output arr used for the actual submission. NOTE: conversion from floats and halfs in host code was oked by this instructor comment, otherwise a simple kernel like from MP7 could be used to do the conversions* [*https://campuswire.com/c/G184FB646/feed/636*](https://campuswire.com/c/G184FB646/feed/636) *.*  *The performance should be increased due to the lessening of data volume, along with playing by the FP16 primitives the GPU actually uses under the hood (especially that super fast fused multipy add instruction). It synergizes with streams as it doesn’t care about the underlying data types, as long as you swap over to \_\_half in all relevant places. Same with the constant memory optimization, as long as you swap over the mask to use the \_\_half data type.* |
| 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.351344 ms* | *0.837059 ms* | *0m0.204s* | *0.86* | | 1000 | *3.1836 ms* | *7.79319 ms* | *0m0.292s* | *0.887* | | 5000 | *15.7403 ms* | *38.359 ms* | *0m0.909s* | *0.8712* | |
| 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). |
| *<answer here>* |
| 1. What references did you use when implementing this technique? |
| *<answer here>* |

|  |
| --- |
| 1. **Optimization 3: *<optimization name>***   ***(Delete this section blank if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *<answer here>* |
| * 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? |
| *<answer here>* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 1000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 5000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | |
| * 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). |
| *<answer here>* |
| * 1. What references did you use when implementing this technique? |
| *<answer here>* |
| 1. **Optimization 4: *<optimization name>***   ***(Delete this section blank if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *<answer here>* |
| * 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? |
| *<answer here>* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 1000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 5000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | |
| * 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). |
| *<answer here>* |
| * 1. What references did you use when implementing this technique? |
| *<answer here>* |
| 1. **Optimization 5: *<optimization name>***   ***(Delete this section if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *<answer here>* |
| * 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? |
| *<answer here>* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 1000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 5000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | |
| * 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). |
| *<answer here>* |
| * 1. What references did you use when implementing this technique? |
| *<answer here>* |
| 1. **Optimization 6: *<optimization name>***   ***(Delete this section if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *<answer here>* |
| * 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? |
| *<answer here>* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 1000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | | 5000 | *<op\_time>* | *<op\_time>* | *<exec\_time>* | *<accuracy>* | |
| * 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). |
| *<answer here>* |
| * 1. What references did you use when implementing this technique? |
| *<answer here>* |