|  |  |
| --- | --- |
| **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* | *0m1.542s* | *0.86* | | 1000 | *2.39847 ms* | *9.13148 ms* | *0m10.144s* | *0.886* | | 5000 | *11.8805 ms* | *45.393 ms* | *0m48.627s* | *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? |
| *Code for the involved is below and in the stream-new-forward.cu file in the submission in the m3opts folder.*  *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 in the m3opts folder.*  #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));  *// register as pinned mem, not paged mem for GPU async transfers*  cudaHostRegister((void \*) host\_output, dop\_sz, 0);  cudaHostRegister((void \*) host\_input, dip\_sz, 0);  cudaHostRegister((void \*) host\_mask, dmp\_sz, 0);  *// 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);  cudaHostUnregister((void \*) host\_output);  cudaHostUnregister((void \*) host\_input);  cudaHostUnregister((void \*) host\_mask);  *// 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 loop unrolling 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* | *0m1.715s* | *0.86* | | 1000 | *63.838 ms* | *53.3827 ms* | *0m10.680s* | *0.886* | | 5000 | *176.503 ms* | *148.22 ms* | *0m50.561s* | *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). |
| *I’ll focus on 5000 batch size optime. For the baseline, we have 11.8805 + 45.393 = 57.2735. However, for the streams, the optimes were exceedingly low as due to how the stream opt needed to be done, it had to be outside of the testing fn. As such the layer times were 176.503 + 148.22 = 324.723 for the streams opt. There isn’t really a direct way to compare these values of layer time and optime though, and so I feel like to show the opt, we can look directly at reports and extrapolate from there.  Below is the screenshot for the timing graph of the nsys command for the stream opt as shown via the UI for NVIDIA Nsight Systems . As you can see, the relevant memcpys and kernel launches that occur are stacked, sharing execution time just as we needed. If we were to stretch out these transfers, they would significantly total up the time, as shown via the baseline pic below it. Additionally, the host code is ahead of the device by a decent margin, which ends up freeing the host thread much sooner to do other work: the result we were looking for. The kernel is unchanged and thus Nsight-Compute is not used here, but here is an example of the reduced kernel’s details screenshot, where you see the SM % util is up while the memory % went a bit down, although this is for a reduced batch size per kernel so comparisons aren’t 1-1.* |
|  |
|  |
| * 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. m3opts/ folder.*  *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 loop unrolling, as long as you swap over the ops used from a singlular to multiple hfma() calls.* |
| 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* | *0m2.825s* | *0.86* | | 1000 | *3.1836 ms* | *7.79319 ms* | *0m11.800s* | *0.887* | | 5000 | *15.7403 ms* | *38.359 ms* | *0m52.363s* | *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). |
| *So from the pov of the optimes,* *I’ll focus on 5000 batch size. For the baseline, we have 11.8805 + 45.393 = 57.2735. Comparing to the fp16, we have 15.7403 + 38.359 = 54.0993, a nearly 3ms improvement overall. As we can also see, the smaller layer pass went thru quicker on the baseline, meaning this approach isn’t as robust for smaller sizes, but the larger layer pass cut down by about 7ms, indicating these performance gains tend to scale. I will now show the comparison between the fp16 and the baseline kernel performance. As you can see, the SM% and the Memory% actually went down from the baseline; however, the timing of the actual kernel execution is actually way down from the baseline, meaning the kernel still outputs faster results despite the bandwidth hits. This tracks with this article I found on \_\_half arithmetic (**https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f) in which it states in the Performance section that “With \_\_half and 32 threads/warp — we only hit ~64B/Load, whereas to achieve good bandwidth we need to strive to hit at least 128B/Load”, meaning we want to try and pack in two \_\_half ops at the same time to save both on memory and SM performance. This could involve a refactoring to the \_\_half2 datatype if future optimizations wanted to go down that route, and I believe it would optimize this code even harder than what we have with just standard fp16 floats* |
| 1. What references did you use when implementing this technique? |
| *I utilized the half info section of the CUDA docs* [*https://docs.nvidia.com/cuda/cuda-math-api/group\_\_CUDA\_\_MATH\_\_\_\_HALF\_\_MISC.html*](https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html) *, and then combined this with the docs for the \_\_hfma() fn* [*https://docs.nvidia.com/cuda/cuda-math-api/group\_\_CUDA\_\_MATH\_\_\_\_HALF\_\_ARITHMETIC.html#group\_\_CUDA\_\_MATH\_\_\_\_HALF\_\_ARITHMETIC\_1gaec96bd410157b5813c940ee320175f2*](https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__ARITHMETIC.html#group__CUDA__MATH____HALF__ARITHMETIC_1gaec96bd410157b5813c940ee320175f2) *. I also utilized Ion Thrusters medium article* [*https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f*](https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f) *as a sort of post-hoc source for why the mem and SM bandwidth might be down even though we are using the \_\_half datatype.* |

|  |
| --- |
| 1. **Optimization 3: *Tuning with restrict and loop unrolling (considered as one optimization only if you do both) (3 points)***   ***(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. |
| *For this optimization, I used the \_\_restrict\_\_ ptrs along with loop unrolling within the kernel convolution fn for various kernel params. I have heard that loop unrolling is a classic C optimization to squeeze more performance out of the processor by stacking operations in larger batches over a total. As such, it would seem natural to try it out in order to try it over more than a couple threads on a CPU. The restrict keyword is also something I have seen as a contract or hint that helps out a lot with compiler optimization, and for similar reasons would make sense to try.* |
| * 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 loopunrollrestrict-new-forward.cu file in the submission. m3opts/ folder.*  *It involves tagging all 3 data pointers in the kernel fn with \_\_restrict\_\_ to hint to the compiler that this is non-aliased, read-only memory that doesn’t overlap. Further into the execution, the looping over the KxK mask is unrolled such that each loop iteration does multiple operations at once based on K. For sufficiently small K, it is unchanged as performance gain is minimal. For middling values, it does not perform as many unrolled ops due to overhead, but larger masks have more unrolled ops. There are a lot of local vars involved to split apart the loop unroll in order to organize the logic and remove redundant calculations. It should increase performance for the reasons listed in section A, in that \_\_restrict\_\_ hints to the compiler that you can go crazier on these optimizing pointers because we are being “nice” and not overlapping stuff, and the loop unrolling is meant to get away from the costly and sort of unneeded loop iteration on every single multiply-add operation. It synergizes with the streams and FP16 optimizations, as it changes the loop behavior, not the underlying operation. As long as the loops are intact in functionality, this optimization can help.* |
| * 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 | *0.252222 ms* | *0.84406 ms* | *0m4.207s* | *0.86* | | 1000 | *2.23091 ms* | *7.87841 ms* | *0m11.137s* | *0.886* | | 5000 | *11.0115 ms* | *39.0854 ms* | *0m51.149s* | *0.871* | |
| * 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). |
| *I wanted to start the optimization by tweaking the # of unrolled ops on the K cases between 3 and 8 (targeting the 5000 batch request we run after the test cases pass). I ran through various unroll factors for the 5000 batch size kernels, and these were the profiling results from the terminal output.*  *// unroll of 4* [*http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bb81edbc25168da9ef6db.tar.gz*](http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bb81edbc25168da9ef6db.tar.gz)  *Op Time: 12.29 ms*  *Op Time: 40.8108 ms*  *Test Accuracy: 0.871*  *real 0m52.019s*  *// unroll of 3* [*http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bb9b2dbc2516a4709382b.tar.gz*](http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bb9b2dbc2516a4709382b.tar.gz)  *Op Time:* *11.0115 ms*  *Op Time:* *39.0854 ms*  *Test Accuracy: 0.871*  *real* *0m51.149s*  *// unroll of 2* [*http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bbb56dbc2516bb4aa423b.tar.gz*](http://s3.amazonaws.com/files.rai-project.com/userdata/build-656bbb56dbc2516bb4aa423b.tar.gz)  *Op Time: 11.1923 ms*  *Op Time: 40.2075 ms*  *Test Accuracy: 0.871*  *real 0m51.068s*  *// unroll of 1 is just baseline so not tested*  *From this, it was clear that in terms of total optime, the unroll of 3 was the most performant. As such, I decided on that as my unroll factor to continue.*  *I’ll focus on 5000 batch size optime. For the baseline, we have 11.8805 + 45.393 = 57.2735. For this loop unrolling logic with the unroll factor of 3, we see* *11.0115 + 39.0854 = 50.0969, about 7ms less. From this, we can see this optimization does have the potential to enable the same amount of computation in less time. In all, it does seem like this optimization will shine more with larger masks as opposed to the rather small masks we have right now (3 or 7?) as that is the main variable to optimize against in this opt.*  *Below is the unroll report, followed by the baseline report. The last kernel launch (the one we care about) in the unroll report was 39.136ms, while the baseline was 45.586ms. This tracks as we see a huge ms drop off on the unroll due to that unroll factor being tuned in for the best gains in the timing department.*  *If we take a look at the kernels, we can see that the unroll kernel utilized 4.46% more of the memory in the SOL tests, compared to the baseline. And even more profound, the memory utilization jumped by a giant margin of 40.62%, showing this is nothing but a proper optimization that takes more advantage of the memory bandwidth. Results are overlayed with the green being the baseline and the blue being the unroll.* |
|  |
|  |
| * 1. What references did you use when implementing this technique? |
| *I was kickstarted by this forum post on the benefits of loop unrolling* [*https://forums.developer.nvidia.com/t/automatic-loop-unrolling/10502*](https://forums.developer.nvidia.com/t/automatic-loop-unrolling/10502) *which refers to a section in the best practices programming guide where this is esposed as a benefit. While the general idea was rather simple (split individual for loop iterations into batches), and even mirrored on large scale via the streams op, the real time spent was in tuning the # of ops per unrolled loop iteration in order to try and get optimal results by running the profiling queue again and again. Tooling the specifics of the manual loop unroll was done by myself after settling on a design.* |