Name: Ander Liu NetID: anderdl2 Section: AL1

ECE 408/CS483 Milestone 3 Report

0. 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.

			Total	
Batch Size	Op Time 1	Op Time 2	Execution	Accuracy
			Time	
100	0.21207	0.618524	0m1.158s	0.86
	ms	ms		
1000	1.92372	5.83225 ms	0m11.235s	0.886
	ms			
10000	18.7866	57.072 ms	1m39.627s	0.8714
	ms			

1. Optimization 1: Weight matrix (kernel values) in constant memory (1 point)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Weight matrix in constant memory, because using constant memory can reduce global memory access.

b. 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?

In host, we put the weight matrix into constant memory first. Then, whenever we need access to the weight matrix, we can directly get it from constant memory instead of global memory. I think the optimization could reduce OP time because the global memory bandwidth would decrease.

c. 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	0.16978	0.509486	0m1.484s	0.86
	ms	ms		
1000	1.62142	5.23384 ms	0m9.797s	0.886
	ms			
10000	15.9062	51.8366 ms	1m36.829s	0.8714
	ms			

d. 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 slightly improves the performance. Before the optimization, the total time of conv_forward_kernel is 846ms. After the optimization, the total time of conv_forward_kernel is 732ms

• Before (baseline)

CUDA API Statistics (nanoseconds)

` '			ls Average		num Maximum Name
93.8	185336005	8	23167000.6	5015	5 184778945 cudaMalloc
5.4	10657368	8	1332171.0	17636	5635700 cudaMemcpy
0.4	876302	6	146050.3	2918	633192
cudaDevi	iceSynchroniz	е			
0.3	506914	6	84485.7	5696	144198 cudaFree
0.1	117269	6	19544.8	14623	24389 cudaLaunchKernel
CUDA Ke	rnel Statistics	(nanos	seconds)		
Time(%)	Total Time	Instai	nces Avera	age Mir	nimum Maximum Name
00.4					
		2	422973.0	213822	632124
	845946 ward_kernel	2	422973.0	213822	632124
	ward_kernel			213822 1280	632124 1408 prefn_marker_kernel
conv_for	ward_kernel 2688	2	1344.0		
conv_for 0.3 0.3	ward_kernel 2688	2	1344.0	1280	1408 prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Tir	ne Op	erations	Average	Minimur	n Maximum Name

93.3 8178665	2	4089332.5	3461417	4717248 [CUDA memcpy					
DtoH] 6.7 588541 HtoD]	6	98090.2	1376 3.	37726 [CUDA memcpy					
	itions	Average		Maximum Name					
DtoH]	2	8612.0		10000.0 [CUDA memcpy 889.0 [CUDA memcpy					
HtoD]	O	300.0	0.004 2	.885.0 [CODA Memcpy					
	e Cal	ls Averag	e Minimun	n Maximum Name 					
35.1 113482054 sem_timedwait 31.0 100091711 31.0 100022267 pthread_cond_timed	31.0 1000917110 25 40036684.4 35079 100219931 poll 31.0 1000222676 2 500111338.0 500093718 500128958 pthread_cond_timedwait								
	e Cal	ls Averag		n Maximum Name					
6.2 11376831 0.4 748391	. 6 6 6	28408548. 1896138.5	11753	169917476 cudaMalloc 6135749 cudaMemcpy 559688					
cudaDeviceSynchron 0.3 629120 0.2 407238 cudaMemcpyToSyml	6 2	104853.3 203619.0		179366 cudaFree 204409					
0.1 115539	6	19256.5	13755	23838 cudaLaunchKernel					
CUDA Kernel Statistic									
Time(%) Total Tim	e Insta	nces Aver	_	um Maximum Name					
Time(%) Total Tim	e Insta 2	nces Aver							

Time(%)		e Operat	ions Ave	erage N		Maximum	
DtoH]						64052 [CUDA 9 [CUDA mem	
HtoD] CUDA M	emory Opera	ation Stati tions	istics (KiB) Average	Minim	num	Maximum N	
177 DtoH]	 225.0	2	8612.0	7225.000	1000	00.0 [CUDA m	
Time(%)	ng System Ru Total Tim	e Calls	Averag	e Minii	mum	Maximum Na	ıme
34.6 sem_tim 31.5 31.5		4 25 5 25 3 2	43968052 40078980	0.6 38	452 10	0166372 0217631 pol 500147304	

e. What references did you use when implementing this technique? Lab4 from UIUC ECE408

2. Optimization 2: Shared memory matrix multiplication and input matrix unrolling + Shared memory matrix multiplication and input matrix unrolling (5 points)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Shared memory matrix multiplication and input matrix unrolling, because the baseline algorithm is not efficient in global memory bandwidth. Each input tile will be loaded M times, where M is number of output features.

b. 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?

We unroll the input matrix so that the convolution has been transform into matrix multiplication, which could reduce more global memory bandwidth. I think the optimization could reduce OP time because the global memory bandwidth would decrease. I synergize this optimization with Weight matrix in constant memory.

c. 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).

			Total	
Batch Size	Op Time 1	Op Time 2	Execution	Accuracy
			Time	
100	0.687337	1.8069 ms	0m1.218s	0.86
	ms			
1000	5.27139	13.8351 ms	0m9.790s	0.886
	ms			
10000	51.212 ms	595.715 ms	1m36.738s	0.8714

d. 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 decreases the performance instead. Before the optimization, the total time of conv_forward_kernel is 732ms. After the optimization, the total time of conv_forward_kernel is 2.486s. I think the reason is memory coalescing. Although we reduce the total access of memory, the implementation does not fit with memory burst, which is important when memory optimization as well.

• Before (Weight matrix in constant memory)

			_		um Maximum Name	
92.8	170451291	6	28408548.5	5 74487	7 169917476 cudaMall	ос
6.2	11376831	6	1896138.5	11753	6135749 cudaMemcp	У
0.4	748391	6	124731.8	2980	559688	
	iceSynchronize					
0.3	629120	6	104853.3	57668	179366 cudaFree	
0.2	407238	2	203619.0	202829	204409	
	mcpyToSymbo					
0.1	115539	6	19256.5	13755	23838 cudaLaunchKern	el
CUDA Ke	rnel Statistics	(nanos	seconds)			
				_	imum Maximum Nan 	
	732190		366095.0	174175	558015	
	ward_kernel					
					1376 prefn_marker_kerne	el
	2592			1248	1344	
do_not_ı	remove_this_l	kernel				
CUDA Me	emory Operati	ion Sta	tistics (nanose	econds)		
Time(%)		Opera	tions Ave	rage Mi	nimum Maximum Na 	
Time(%) 90.8	Total Time	Opera	tions Ave	rage Mi 		
Time(%) 90.8 DtoH] 9.2	Total Time	Opera 2	tions Ave	rage Mi 3792471		 ncpy
Time(%) 90.8 DtoH] 9.2 HtoD] CUDA Mo	Total Time 8956523 904446 emory Operatotal Operati	Opera 2 6 ion Sta	tions Ave	3792471 1216 Minimu	5164052 [CUDA mer	ncpy
Time(%) 90.8 DtoH] 9.2 HtoD] CUDA Mo	Total Time 8956523 904446 emory Operatotal Operati	Opera 2 6 ion Sta ons	tions Ave	3792471 1216 Minimu	5164052 [CUDA mer 480479 [CUDA memcpy Maximum Name	ncpy
Time(%) 90.8 DtoH] 9.2 HtoD] CUDA Me	Total Time 8956523 904446 emory Operational Operational	Opera 2 6 ion Sta ons	tions Ave	3792471 1216 Minimu	5164052 [CUDA mer 480479 [CUDA memcpy Maximum Name	ncp
Time(%) 90.8 DtoH] 9.2 HtoD] CUDA Mo To 172 DtoH] 540 HtoD]	Total Time	Opera 2 6 ion Sta ons 2	tions Ave	Minimu 7225.000 0.004	5164052 [CUDA memcpy Maximum Name 10000.0 [CUDA memcpy	ncţ

31.5	1001974516	25	40078980.6	38452	100217631 poll
31.5	1000252958	2	500126479.0	500105654	500147304
pthread_	cond_timedwait				

After (Weight matrix in constant memory + Shared memory matrix multiplication and input matrix unrolling)

CUDA AP	I Statistics (na	nosec	onds)	.	
Time(%)	Total Time	Cal	ls Averag		um Maximum Name
					2 190662214 cudaMalloc
					5902541 cudaMemcpy
	2502647		41/10/.8	3089	1804983
	iceSynchroniz		400504.0	64700	162020
					163830 cudaFree
	397966		198983.0	198455	199511
	ncpyToSymbo				
0.1	117895	6	19649.2	14743	23014 cudaLaunchKernel
	rnel Statistics	-	•		
					imum Maximum Name
99.8	2486478	2	1243239.0	682779	1803699
conv_for	ward_kernel				
0.1	2528	2	1264.0	1184	1344 prefn_marker_kernel
	2400				1216
do_not_i	remove_this_	kernel			
CUDA M	emory Operat	ion Sta	itistics (nanos	econds)	
				· · ·	nimum Maximum Name
				3616356	4987674 [CUDA memcp
DtoH]					
	904409	6	150734.8	1184	480764 [CUDA memcpy
HtoD]					
CUDA M	emory Operat	ion Sta	itistics (KiB)		
	otal Operati		Average	Minimu	m Maximum Name
	· 				
173	 225.0 2	 ว	8612.0	7225.000	10000.0 [CUDA memcpy
	223.0	<u> </u>	3012.0	1223.000	10000.0 [CODA Memcpy
DtoH]	02.0 6		900.0	0.004	2000 O [CLIDA momeny
	02.0 6		900.0	0.004	2889.0 [CUDA memcpy
HtoD]					

Operating System Runtime API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum Name
34.8	1119405357	26	43054052.2	38820	100151786
sem_tim	edwait				
31.2	1002479586	25	40099183.4	52837	100227092 poll
31.1	1000268842	2	500134421.0	500107029	500161813
pthread	cond timedwait	-			

e. What references did you use when implementing this technique? Slides from UIUC ECE408 Lecture 12

3. Optimization 3: FP16 arithmetic (4 points)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

FP16 arithmetic, because using half-precision floating-point might reduce the time of calculation.

b. 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?

There are two types of FP16 in CUDA, half and half2. half is single 16-bit floating point quantity/type. half2 is a vector type, consisting of two 16-bit floating point quantities packed into a single 32-bit type. In kernel, I transform two floats into half2 first. And then, multiple half2 and transform back into floats again.

I think the optimization could reduce OP time because the calculation reduces.

I synergize this optimization with Weight matrix in constant memory.

 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).

			Total	
Batch Size	Op Time 1	Op Time 2	Execution	Accuracy
			Time	
100	1.88773	2.45859 ms	0m1.205s	0.86
	ms			
1000	2.40648	7.88971 ms	0m9.994s	0.886
	ms			
10000	23.8918	76.3115 ms	1m39.224s	0.8714
	ms			

d. 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 does not improve the performance as well. Before the optimization, the total time of conv_forward_kernel is 732ms. After the optimization, the total time of conv_forward_kernel is 1.056s. I think the reason is that the conversion time between float and FP16 is larger than the reduced time of calculation

• Before (Weight matrix in constant memory)

			_		ım Maximum Name
					169917476 cudaMalloc
6.2	11376831	6	1896138.5	11753	6135749 cudaMemcpy
0.4	748391	6	124731.8	2980	559688
	iceSynchroniz				
0.3	629120	6	104853.3	57668	179366 cudaFree
			203619.0		
	mcpyToSymbo				
			19256.5	13755	23838 cudaLaunchKernel
CUDA Ke	rnel Statistics	(nanos	seconds)		
				_	mum Maximum Name
99.3	732190	2	366095.0	174175	558015
	ward_kernel				
			1360.0		.376 prefn_marker_kernel
0.4	2592	2	1296.0	1248 1	.344
	remove_this_l				
CUDA M	emory Operat	ion Sta	itistics (nanose	conds)	
Time(%)	Total Time	Opera	itions Aver	age Mir	nimum Maximum Name
	8956523	2	4478261.5	3792471	5164052 [CUDA memcpy
DtoH]					
	904446	6	150741.0	1216	480479 [CUDA memcpy
HtoD]					
•					
	emory Operat	ion Sta	itistics (KiB)		

17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]
5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]
Operating System Runtime API Statistics (nanoseconds) Time(%) Total Time Calls Average Minimum Maximum Name
34.6 1099201304 25 43968052.2 30586 100166372
sem_timedwait 31.5 1001974516 25 40078980.6 38452 100217631 poll 31.5 1000252958 2 500126479.0 500105654 500147304 pthread_cond_timedwait
 After (Weight matrix in constant memory + FP16) CUDA API Statistics (nanoseconds)
Time(%) Total Time Calls Average Minimum Maximum Name
93.1 176129476 6 29354912.7 75957 175497364 cudaMalloc 5.7 10815388 6 1802564.7 12235 5799626 cudaMemcpy 0.6 1072154 6 178692.3 3162 810449
cudaDeviceSynchronize 0.3 592857 6 98809.5 61519 149900 cudaFree 0.2 398766 2 199383.0 194448 204318
cudaMemcpyToSymbol 0.1 127542 6 21257.0 15305 27471 cudaLaunchKernel
CUDA Kernel Statistics (nanoseconds) Time(%) Total Time Instances Average Minimum Maximum Name
99.5 1056024 2 528012.0 247358 808666
conv_forward_kernel 0.2 2528 2 1264.0 1184 1344 prefn_marker_kernel 0.2 2432 2 1216.0 1216 1216 do_not_remove_this_kernel
CUDA Memory Operation Statistics (nanoseconds) Time(%) Total Time Operations Average Minimum Maximum Name
90.3 8458909 2 4229454.5 3545988 4912921 [CUDA memcpy
DtoH] 9.7 904824 6 150804.0 1216 480988 [CUDA memcpy HtoD]

CUDA Memory Operation St Total Operations				Minim	um Maximum Name 	
172 DtoH]	225.0	2	8612.0	7225.000	10000.0 [CUDA memcpy	
54 HtoD]	02.0	6	900.0	0.004	2889.0 [CUDA memcpy	
•	Operating System Runtime API Statistics (nanoseconds) Time(%) Total Time Calls Average Minimum Maximum Name					
34.7 sem_tim	 11615054 edwait	90 26	 5 4467328	8.1 274	192 100162685	
29.9	11014196 10002687 cond_time_	76 2			943 100219958 poll 15683 500153093	

e. What references did you use when implementing this technique?

https://docs.nvidia.com/cuda/cuda-mathapi/group CUDA MATH INTRINSIC HALF.html

4. Optimization 4: Tuning with restrict and loop unrolling (3 points)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Restrict pointer and loop unrolling, because they can optimize the compiler more.

b. 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?

Restrict pointer means that the pointer is the only way to access the object pointed by it. By utilizing restrict pointer, compiler can produce better optimized code.

The first benefit of loop unrolling is that it reduces the calculation of index. The second benefit of loop unrolling is the enhancement of Instruction-Level Parallelism. In the unrolled version, there would possibly be more operations for the processor to push into processing pipeline without being worried about the for loop condition in every iteration.

I think the optimization could reduce OP time because of the complier optimization.

I synergize this optimization with Weight matrix in constant memory.

c. 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).

			Total	
Batch Size	Op Time 1	Op Time 2	Execution	Accuracy
			Time	
100	0.194829	0.615394	0m1.274s	0.86
	ms	ms		
1000	1.75325	5.76556 ms	0m9.707s	0.886
	ms			
10000	17.2728	57.4893 ms	1m39.408s	0.8714
	ms			

d. 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 does not affect the performance. Before the optimization, the total time of conv_forward_kernel is 732ms. After the optimization, the total time of conv_forward_kernel is 790ms. I think the reason is that the complier is advanced enough that it already implements Tuning with restrict and loop unrolling.

• Before (Weight matrix in constant memory)

CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Calls	s Average	Minimum	n Maximum Name
92.8	170451291	6	28408548.5	74487	169917476 cudaMalloc
6.2	11376831	6	1896138.5	11753	6135749 cudaMemcpy

0.4	748391	6	124731.8	2980	559688		
cudaDev	viceSynchroniz	<u>ze</u>					
0.3	629120	6	104853.3	57668	179366	cudaFree	
0.2	407238	2	203619.0	202829	204409		
	mcpyToSymbo						
	115539		19256.5	13755	23838 ci	udaLaunchKernel	
0		•		20700			
CUDA K	ernel Statistics	(nanos	econds)				
		-		rage Mii	nimum	Maximum Name	
				_			
00.3	732190	2	366095.0	17/175	552015		
	rward_kernel		300033.0	174175	336013	•	
0.4	2720	2	1260.0	1244	1276 profr	narkar karnal	
						n_marker_kernel	
	2592		1296.0	1248	1344		
do_not_	_remove_this_	kernei					
	lemory Opera						
						Maximum Name	
90.8	8956523	2	4478261.5	379247	1 51640)52 [CUDA memc	Эγ
DtoH]							
9.2	904446	6	150741.0	1216	480479 [CUDA memcpy	
HtoD]							
CUDA M	lemory Opera	tion Stat	tistics (KiB)				
				Minim	um M	aximum Name	
17	225.0	2	8612 N	7225 000	10000	0 [CUDA memcpy	,
DtoH]	223.0	2	8012.0	7223.000	10000.	o (coba memcp)	
-	102.0	-	000.0	0.004	2000 0 [0	CLIDA mamany	
	102.0)	900.0	0.004	2889.U [C	CUDA memcpy	
HtoD]							
					,		
-	ng System Run						
Time(%)	Total Time	Call	s Averag	ge Minin	num Ma	aximum Name	
34.6	1099201304	25	4396805	2.2 305	86 1001	66372	
sem_tim	nedwait						
31.5	1001974516	25	4007898	0.6 384	152 1002	17631 poll	
31.5	1000252958	2	50012647	9.0 50010		00147304	
	_cond_timedv						

CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Cal	ls Average	Minimu	ım Maximum Name
93.2	195806043	6	32634340.5	85007	195167318 cudaMalloc
5.9	12480644	6	2080107.3	13348	7255342 cudaMemcpy
0.4	806017	6	134336.2	2709	607979
cudaDev	iceSynchronize)			
0.3	572078	6	95346.3	57150	129742 cudaFree
0.2	399348	2	199674.0	198119	201229
cudaMer	mcpyToSymbol				
0.1	121053	6	20175.5	14292	24991 cudaLaunchKernel

After (Weight matrix in constant memory + Tuning with restrict and loop unrolling)

CUDA API Statistics	(nanoseconds)
----------------------------	---------------

Total Time	Cal	ls Average	Minimu	ım Maximum Name
195806043	6	32634340.5	85007	195167318 cudaMalloc
12480644	6	2080107.3	13348	7255342 cudaMemcpy
806017	6	134336.2	2709	607979
iceSynchronize				
572078	6	95346.3	57150	129742 cudaFree
399348	2	199674.0	198119	201229
ncpyToSymbol				
121053	6	20175.5	14292	24991 cudaLaunchKernel
	195806043 12480644 806017 iceSynchronize 572078 399348 ncpyToSymbol	195806043 6 12480644 6 806017 6 iceSynchronize 572078 6 399348 2 ncpyToSymbol	195806043 6 32634340.5 12480644 6 2080107.3 806017 6 134336.2 iceSynchronize 572078 6 95346.3 399348 2 199674.0 ncpyToSymbol	195806043 6 32634340.5 85007 12480644 6 2080107.3 13348 806017 6 134336.2 2709 iceSynchronize 572078 6 95346.3 57150 399348 2 199674.0 198119 ncpyToSymbol

CUDA Kernel Statistics (nanoseconds)

Time(%)	Total Time	Insta	nces Ave	rage M	inimum Maximum Name
99.3	790493	2	395246.5	184767	605726
conv_forv	vard_kernel				
0.3	2624	2	1312.0	1248	1376 prefn_marker_kernel
0.3	2592	2	1296.0	1280	1312

do_not_remove_this_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Opera	ations Av	erage M	linimum	Maximum Name	
							-
91.6 DtoH]	9861593	2	4930796.5	377118	5 6090	0408 [CUDA memcp	у
8.4 HtoD]	903996	6	150666.0	1216	480446	[CUDA memcpy	

CUDA Memory Operation Statistics (KiB)						
num Maximum Name						
10000.0 [CUDA memcpy						
2889.0 [CUDA memcpy						
s)						
num Maximum Name						
007 100160391						
192 100218490 poll						
20288 500150074						

e. What references did you use when implementing this technique?

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#pragma-unroll

5. Optimization 5: Sweeping various parameters to find best values (1 point)

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Sweeping various parameters, because finding correct parameters might improve the performance as well.

b. 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?

I test with different parameter, such as (4, 4, 64), (8, 8, 16), (16, 16, 4), and find that (8, 8, 16) is the best. However, in PM2, I already tune the parameter into (8, 8, 16), so it will not change the performance.

c. 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	0.194829	0.615394	0m1.274s	0.86
	ms	ms		
1000	1.75325	5.76556 ms	0m9.707s	0.886
	ms			
10000	17.2728	57.4893 ms	1m39.408s	0.8714
	ms			

- d. 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).
 - In PM2, I already tune the parameter into (8, 8, 16), so it will not change the performance.
- e. What references did you use when implementing this technique? PM3 github of UIUC ECE408