CME 213 Parallel Programming **Spring 2018**

Final Report: June 10

Lecturer: Eric Darve *Student: Nurbek Tazhimbetov*

1.1 GEMM and Grading Mode 4

First, I tested the implementation of `myGEMM` using grading mode 4,

	`GEMM1`	`GEMM2`
Serial Time	0.0468626	0.00441485
Vanila Parallel	0.262259	0.00698588
Shared Parallel	0.15199	0.0065526
Parallel Shared A Plus	**0.0753727**	**0.00548108**
Relative difference	1.85746e-16	3.2505e-16

As you can see for `GEMM1`, I have started from vanilla parallel implementation and got around 0.26 seconds, while after implementing shared version, I nearly doubled the speed with 0.15 seconds. In the last algorithm, we nearly quadrupled the initial time and achieved 0.075 seconds. However, we are still far away from the serial code, which is 0.047 seconds. The shared A plus algorithm has been described in *Benchmarking GPUs to Tune Dense Linear Algebra* by Demmel and Volkov in 2008.

1.2 Grading Modes 3, 2 & 1

Table 1.1: Results for grading mode 3, 2 & 1

	`Grading mode 3`	`Grading mode 2`	`Grading mode 1`	`./main`
Serial	13.5539 s	50.2858 s	196.722 s	
Parallel	**12.4649 s**	**13.6078 s**	**52.6221 s**	**107.304 s**
Cross Validation accuracy (both)	0.845167	0.924167	0.894333	0.909833
Serial New Time	10.9739 s	31.4504 s	125.92 s	
Parallel New Time	**9.13504 s**	**4.13525 s**	**14.2701 s**	**37.5473 s**
Cross Validation New accuracy	0.845167	0.924167	0.894333	0.909833
$\|W_1^{(s)} - W_1^{(p)}\|_{\max}$	4.24529e-11	4.57298e-09	1.23397e-15	
$\|W_2^{(s)} - W_2^{(p)}\|_{\max}$	2.0301e-13	3.46816e-11	4.27981e-16	
$\|b_1^{(s)} - b_1^{(p)}\|_{\max}$	1.0906e-10	6.71967e-09	2.99972e-15	
$\|b_2^{(s)} - b_2^{(p)}\|_{\max}$	2.57021e-12	3.29032e-10	6.13302e-16	
$\|W_1^{(s)} - W_1^{(p)}\|_2$	4.04886e-11	3.47396e-09	1.09988e-15	
$\|W_2^{(s)} - W_2^{(p)}\|_2$	8.70559e-13	6.0723e-11	7.15398e-16	
$\|b_1^{(s)} - b_1^{(p)}\|_2$	2.98602e-11	2.08037e-09	1.50882e-15	
$\|b_2^{(s)} - b_2^{(p)}\|_2$	2.83175e-12	2.61242e-10	5.87586e-16	

Next, I used grading mode 3, 2 and 1 to run the neural net training and got the following results. As you can see, the time has dropped significantly. This has been achieved by copying and scattering the data outside of the epoch loop. We call `MPI_Scatterv()` only for one epoch for each batch iteration and store them on the device. This may seem unwise memory-wise, however, accessing to the stored X and y values is much faster from the device itself rather copying from the host every time.

1.3 Profiling

Table 1.2: Results from .err file for `./main`

Old Time (%)	**Old Time**	# of Old Calls	Old Name	**New Time (%)**	**New Time**	# of New Calls	New Name
36.59%	19.1606s	19040	`[CUDA memcpy HtoD]`	31.51%	5.68019s	5442	`[CUDA memcpy HtoD]`
21.98%	11.5084s	2720	`GEMMTransposeKernel`	28.94%	5.21721s	5440	`[CUDA memcpy DtoH]`
20.41%	10.6877s	2720	`GEMMKernel`	16.21%	2.92294s	2720	`GEMMTransposeKernel`
19.14%	10.0203s	10880	`[CUDA memcpy DtoH]`	15.83%	2.85349s	1360	`GEMMSigmoidKernel`
0.65%	342.03ms	5440	`GradientKernel`	4.23%	763.00ms	1360	`GEMMAdditionKernel`
0.59%	309.73ms	1360	`HadamardKernel`	1.72%	310.11ms	1360	`HadamardKernel`
0.46%	239.29ms	2720	`SumOfRowKernel`	1.43%	257.61ms	2720	`SumOfRowKernel`
0.14%	72.683ms	1360	`SigmoidKernel`	0.08%	13.700ms	1360	`SumOfExpColKernel`
0.03%	13.689ms	1360	`SumOfExpColKernel`	0.04%	6.6954ms	1360	`SoftmaxKernel`
0.01%	6.7247ms	1360	`SoftmaxKernel`	0.02%	4.0574ms	1360	`[CUDA memcpy DtoD]`

Another interesting topic to discuss is the difference between serial and parallel results for W_1, W_2, b_1 and b_2 b_2. This can be explained with the fact that some of the numerical computations are done on the GPU

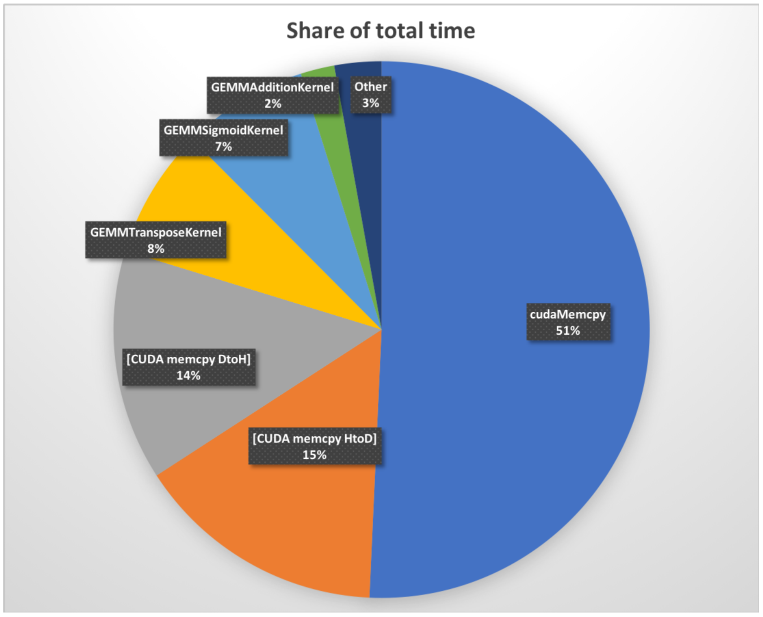

Figure 1.1: Share of total time

and CPU for parallel and serial algorithms, respectively. Therefore, even on hardware level we might have differences between these codes. The last interesting issue here is that for the grading mode 2, the error increases and is higher than for the other two. Perhaps, this is because the learning rate of this algorithm and number of iterations are not a good match.

Table 1.3: Results from .err file for `./main`

%	**Old Time**	# of Old Calls	Old Name	%	**New Time**	# of New Calls	New Name
98.81	54.8160s	29920	`cudaMemcpy`	96.75	19.0458s	12242	`cudaMemcpy`
0.71	394.53ms	13	`cudaMalloc`	2.16	425.68ms	11	`cudaMalloc`
0.34	187.74ms	19040	`cudaLaunch`	0.72	141.88ms	12240	`cudaLaunch`
0.05	29.384ms	364	`cuDeviceGetAttribute`	0.14	27.227ms	364	`cuDeviceGetAttribute`
0.04	22.668ms	104720	`cudaSetupArgument`	0.10	19.373ms	77520	`cudaSetupArgument`
0.02	12.384ms	13	`cudaFree`	0.07	13.786ms	11	`cudaFree`
0.02	8.4327ms	19040	`cudaConfigureCall`	0.04	6.9613ms	12240	`cudaConfigureCall`
0.01	2.8570ms	4	`cuDeviceGetName`	0.02	3.0728ms	4	`cuDeviceTotalMem`
0.00	2.7006ms	4	`cuDeviceTotalMem`	0.01	2.4341ms	4	`cuDeviceGetName`
0.00	19.218us	1	`cudaSetDevice`	0.00	20.598us	1	`cudaSetDevice`
0.00	4.8910us	3	`cuDeviceGetCount`	0.00	16.675us	2	`cudaFreeHost`
0.00	4.6690us	12	`cuDeviceGet`	0.00	6.1060us	12	`cuDeviceGet`
0.00	2.8930us	1	`cudaGetDeviceCount`	0.00	4.7760us	3	`cuDeviceGetCount`
				0.00	2.9900us	1	`cudaGetDeviceCount`

1.4 Optimization

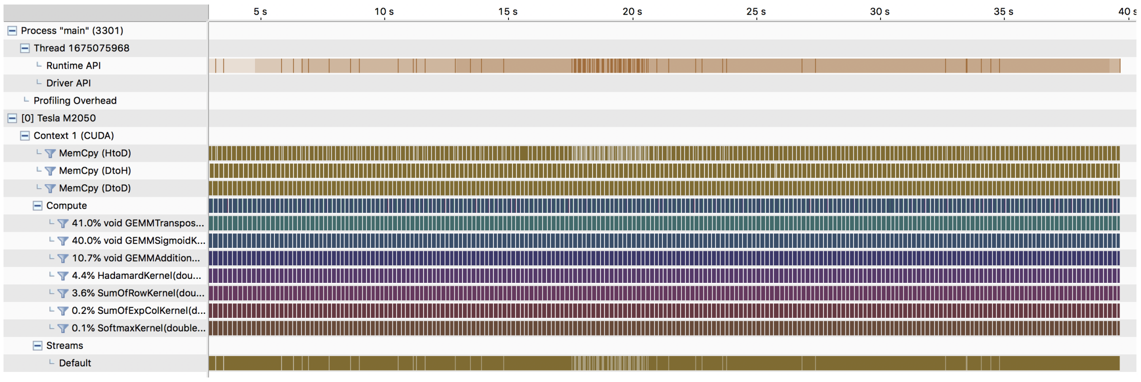

Figure 1.2: Visual Profiler Results

As it was expected the most time-consuming action is `memcpy`; both from the device to the host and vice-versa. Since the A Plus algorithm has been implemented, we have achieved (as mentioned before) four times faster solver; `GEMMTransposeKernel` dropped from over 11 seconds to just under 3 seconds and `GEMMKernel` dropped from over 10 seconds to almost 3 seconds. Furthermore, since we are storing X and y on the device, I minimized the number of calls of the copying function between the host and device (`memcpy`) from 29,920 to 12,242. This saved me 35 seconds.

Same can be said about the GEMM. As we have discussed earlier, the GEMM's fastest implementation is almost four times faster than the vanilla GEMM. That gave me almost 17 seconds.

1.5 Possible Further Optimization

After reducing the values of W_1, W_2, b_1 and b_2 on the host, I have used `arma`'s matrix-matrix addition instead of copying back to the device and write my own matrix-matrix addition, although for the preliminary test I had the GPU version of the gradient descent update. I tend to think this is a minor improvement, however, it might be important, if we have a bigger batch size, which increases number of computations. I could have also minimized the number of kernels, however, as you can see from Figure 1.2, the other kernels are not very time-consuming. However, this is still might give us more time.

- The percentage of time when `memcpy` is being performed in parallel with kernel is low. Therefore, we might want to copy in parallel.
- The percentage of of time when two kernels are being executed in parallel is low. In the back propagation part, we might actually be able to get more time.