Kaylo Littlejohn
GPU Lab 3

Part 1:

| Vector Size | Block Size | CPU Time | GPU Time | GPU Time (pinned) |
|---|---|---|---|---|
| 100 | 512 | 0.00 ms | 0.59 ms | 1.06 ms |
| 100 000 | 512 | 0.44 ms | 1.02 ms | 3.03 ms |
| 100 000 000 | 512 | 236.92 ms | 166.78 ms | 587.60 ms |
| 1 000 000 000 | 512 | 2372.91 ms | 1674.60 ms | 5670.10 ms |

Using nvprof, the call that had the most impact on the total percentage of execution time was the cudaEventCreate. However, when the vector size becomes very large (i.e. > 100,000,000), cudaMemcpy takes a greater time to complete. These statistics are beneficial because it allows us to see which API calls are having the greatest impact on performance. If the cudaEventCreate has the greatest impact, we know our program is performing better because most the execution time is spent simply measuring our performance. If the RAM on the CPU is filled with pinned memory, the execution time will be slowed down. Furthermore, not using cudaFree can result in memory leak for allocated GPU memory. We noticed CudaMallocHost uses a higher percentage of memory than the original malloc as the vector size increases.

Part 2:

OMP is short for OpenMP. These snippets of code are compiler instructions which allow for support for dynamic parallelism (assuming openmpi modules are loaded). The -Xcompiler flag tells the nvcc compiler to send the code to the native compiler (gcc) for compilation of the c portion of the code. Furthermore, the -fopenmp flag allows for the compiler to support OMP and dynamic parallelism. Using these flags, the set up time drastically decreased however the GPU execution time remained the same. Mixing OMP with CUDA will allow us to speed up total execution time for the GPU in combination with the CPU and make full use of dynamic parallelism.

When using dynamic parallelism to replace the for loop in the kernel, we get an error claiming the new kernel requires a separate compilation mode. Adding flag -rdc=true allows us to compile the code as is (despite launching kernels from kernels). Below is our results from the Baseline and Modified code.

| #Matrices | Size | Sequential | Baseline (All GPU) | Baseline (Kernal) | Modified (All GPU) | Modified (Kernal) |
|---|---|---|---|---|---|---|
| 100 | 100 | 0.66 | 21.79 | 17.60 | 25.60 | 21.47 |
| 100 | 1000 | 54.19 | 2339.63 | 2125.48 | 54.34 | 0.00 |
| 100 | 3000 | 280.42 | 21798.31 | 20195.82 | 445.56 | 0.00 |
| 1000 | 1000 | 300.72 | 26853.03 | 25075.85 | 2296.50 | 840.48 |
| 2000 | 750 | 344.38 | 16464.23 | 14465.47 | 2203.86 | 991.67 |
| 5000 | 500 | 382.04 | 11029.77 | 8804.65 | 6381.07 | 4549.12 |

One of the advantages of Dynamic Parallelism is it allows us to compute an entire loop at the same time within a kernel. Our results were unpredictable since we received calculation errors for the 100 1000 and 100 3000 inputs. The smaller the block size, the better. When we changed our block size from 256 to 512, the execution times on the GPU were much slower because each nested kernel launch from the original kernel now takes longer to finish. From our table, we can see that the best value for the number of matrices to be used lies between 100 and 1000. Dynamic parallelism is a great tool to use as long as the block sizes are minimized and assuming there are reasonable loops/calculations that would result in higher performance using a nested kernel launch. If all the loops were assigned as one single kernel call, we would need to increase the block size and number of blocks per grid.