ECE60827: CUDA Assignment 1

Name: Nikhil Reddy Mummadi
GitHub username: mniksr786

Abstract
The C++ programs for SAXPY and Monte Carlo estimation of Pi were parallelized and implemented on the NVIDIA Tesla V100 GPU in CUDA. The programs were profiled across multiple input parameters to compare performance and compared with their CPU implementation.

Results Discussion

SAXPY Implementation

The SAXPY loop was executed on the GPU for multiple vector and block sizes.

Cases	VECTOR_SIZE	Block Size	Runtime (seconds)
Case 1	32768	256	0.388122
Case 2	524288	256	0.411404
Case 3	1048576	256	0.440131
Case 4	1048576	1024	0.438491

Table 1: Input conditions for SAXPY

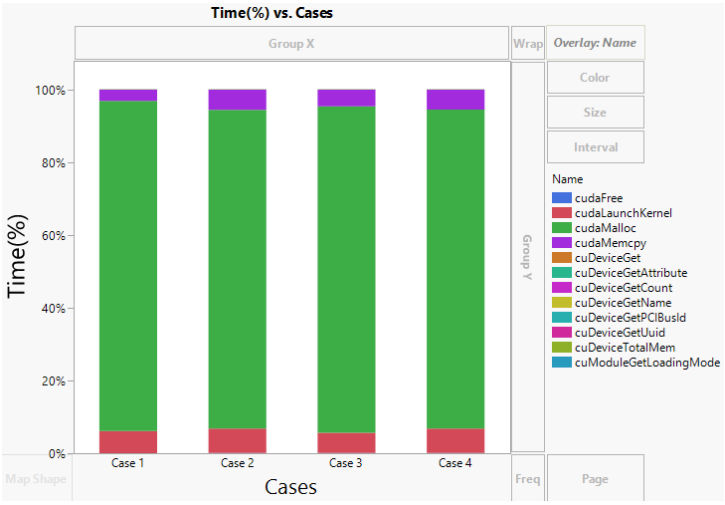

Figure 1: Time(%) for CUDA API calls in SAXPY

Figure 2: Time(%) for GPU activities in SAXPY

An increase in the vector sizes would increase the runtime while using larger block sizes helped reduce the runtime due to better resource utilization (shown in Table 1). Due to the slow transfer speed over PCIe, the majority of the GPU activity in all three input cases was used in data transfer between the CPU and GPU (CUDA memcpy). In the SAXPY code, the majority of the API calls were from cudaMalloc, cudaLaunchKernel, and cudaMemcpy which together constituted about 99% of the total API calls.

Monte Carlo Estimation of Pi

The value of Pi was estimated by running Monte Carlo simulations across multiple sample sizes and iterations on CPU and GPU.

Case	MC_SAMPLE_SIZE	Thread Count	Thread Blocks	Estimated Pi	Runtime (secs)
Case 1	1.00E+06	1024	8	3.141567707	0.316368045
Case 2	1.00E+08	1024	8	3.141591072	10.22964496
Case 3	1.00E+06	2048	16	3.14161253	0.312127991
Case 4	1.00E+08	2048	16	3.141588449	10.26251101
Case 5	1.00E+06	1024	2	3.141620159	0.501812368

Table 2: Program runtime for different inputs to Monte Carlo simulation using CUDA

For the CPU implementation, the Monte Carlo simulations were run for 32 iterations with each loop computing over a sample size of 1.0E+6 randomly generated points (same as case 1 in Table 2). The program runtime on the CPU was 41.9515 seconds. In the CUDA implementation of this program, data points for all the random samples were generated on the GPU using the cuRAND library. This eliminates the hassle of transferring the large samples over PCIe between the CPU and GPU, thus saving plenty of execution time and power. The program was executed with different sample sizes and threads spawned, where each thread would iterate over the sample size to count the hits of the random data points. For the sample size in case 1 which is the same as the input for the CPU implementation, the program runtime on the GPU was just 0.3163 seconds! The parallelization of the program helped bring down the runtime from almost 42 seconds to less than 0.5 seconds. Spawning more threads did not help reduce the runtime by much. However, increasing the random sample size increased the runtime to almost 10 seconds which is a 33x slowdown.

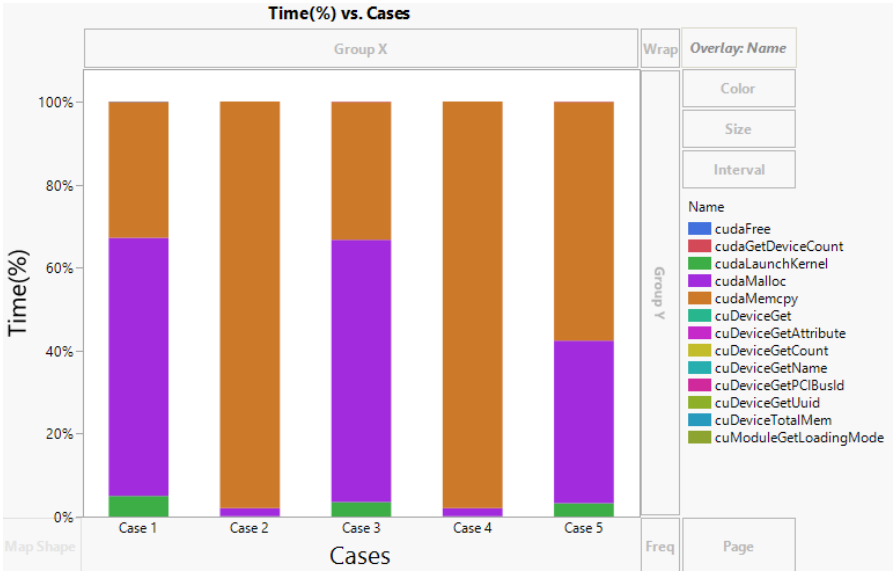

Figure 3: Time(%) for CUDA API calls in Monte Carlo Pi estimation

For a larger sample size (cases 2 and 4), the API cudaMemcpy runs for about 97% of the program runtime to transfer the large number of hits (resulting from a large sample size) counted by all the kernel threads back to the CPU for the Pi calculation. In the remaining cases, the APIs cudaMalloc and cudaMemcpy called for the majority of the program runtime.

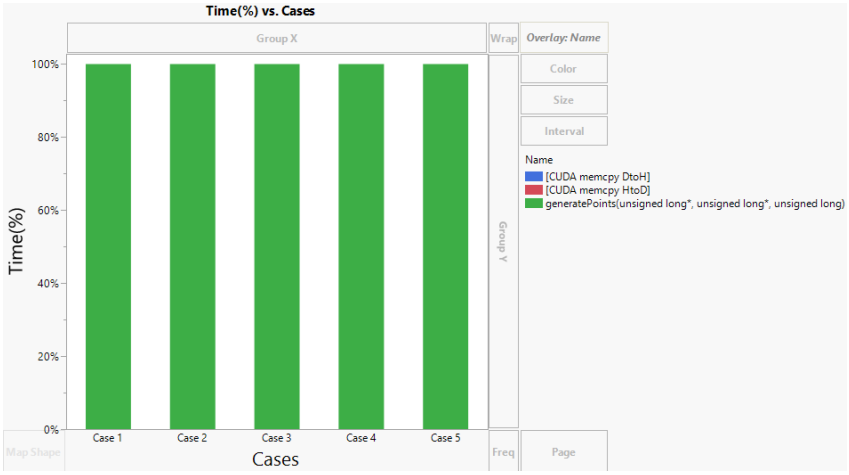

Figure 4: Time(%) for GPU activities in Monte Carlo Pi estimation

Case	Type	Time(%)	Time	Calls	Name
Case 1	GPU activities	100.00%	99.221ms	1	generatePoints(unsigned long*, unsigned long*, unsigned long)
Case 1	GPU activities	0.00%	1.7590us	1	[CUDA memcpy DtoH]
Case 1	GPU activities	0.00%	1.4720us	1	[CUDA memcpy HtoD]
Case 2	GPU activities	100.00%	10.0166s	1	generatePoints(unsigned long*, unsigned long*, unsigned long)
Case 2	GPU activities	0.00%	1.7600us	1	[CUDA memcpy DtoH]
Case 2	GPU activities	0.00%	1.4720us	1	[CUDA memcpy HtoD]
Case 3	GPU activities	100.00%	99.358ms	1	generatePoints(unsigned long*, unsigned long*, unsigned long)
Case 3	GPU activities	0.00%	1.7920us	1	[CUDA memcpy DtoH]
Case 3	GPU activities	0.00%	1.4720us	1	[CUDA memcpy HtoD]
Case 4	GPU activities	100.00%	10.0489s	1	generatePoints(unsigned long*, unsigned long*, unsigned long)
Case 4	GPU activities	0.00%	1.7920us	1	[CUDA memcpy DtoH]
Case 4	GPU activities	0.00%	1.4720us	1	[CUDA memcpy HtoD]
Case 5	GPU activities	100.00%	284.29ms	1	generatePoints(unsigned long*, unsigned long*, unsigned long)
Case 5	GPU activities	0.00%	1.7600us	1	[CUDA memcpy DtoH]
Case 5	GPU activities	0.00%	1.1200us	1	[CUDA memcpy HtoD]

Table 3: GPU Activity for Monte Carlo Pi Estimation

For all input cases of the Monte Carlo Pi estimation, the majority of GPU activity is coming from the CUDA kernel as shown in table 3.

In conclusion, minimizing data transfer between the CPU and GPU over the PCIe interface would reduce the overall program runtime since slower data transfer speeds are the main bottleneck for overall performance.