Programming Assignment

(Lab 1 Report)

- ECE 60827 -

(Joel Taina)

1/28/2025

PART A: Single-precision A · X Plus Y (SAXPY)

SAXPY is an "embarrassingly parallel" task for the GPU to perform, so much so that for floating-point accuracy reasons, NVIDIA GPUs support SAXPY with a single operation, the fused multiply-add operation (FMA). Introduced by IEEE 754-2008, it approximates the mathematical answer by one rounding step. Compilers use this operation for CUDA C++ by default, while CPU code implements this as separate add and multiply, with two rounding steps total. This might explain why CPU vs GPU result comparisons are off by the least significant bit in the mantissa of both floats.

Execution time for GPU activities

All results use scale = 2.0, with vector sizes within tester range, from 1 (2^0) to 2^{29}-1. Thread blocks are 16x16. The program ran without crashes, with zero errors. Vectors are initialized by the CPU, then transferred to be SAXPY'd by the GPU, before returning results back. Notable observations for GPU activities:

- For small vector sizes (size < 1024), math operations (saxpy_gpu) are on par with data transfer between host and GPU (HtoD, DtoH) (see Figure 1). Run time remains constant throughout for all operations (see Figure 2). The GPU has enough cores to dedicate a thread for each slot. Only one memory block is transferred (back-and-forth), which is large enough to hold all data.
- Around the middle (1024 < size < 2^20), run time begins to increase for all operations, though math operations lag by a time shift (see Figure 2), resulting in a decrease in the percentage (see Figure 1). Multiple memory blocks are transferred across, and GPU cores begin to saturate.
- At large enough vector sizes (size > 2^20), the rate of change of run time for math operations becomes parallel with that of the total run time, leveling off at a constant yet slim percentage. It is likely that we were allowed access to an entire NVIDIA V100 GPU 32 GB per student, or perhaps at least 4 GB of it (2^29 floats per vector * 4 bytes per float * 2 input vectors).

Execution time for CUDA API calls

Looking at CUDA API calls, data transfer run times (similar to total time) between host and GPU correlates with use of cudaMemcpy(), while other API calls remain constant. Around vector size 2^25, runtime for this API call overtakes that of all other CUDA API calls combined (see Figure 3).

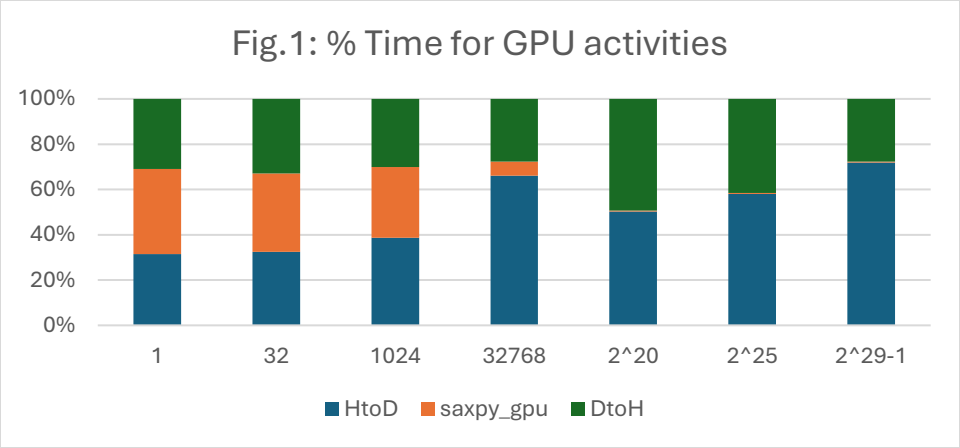
Fig.1: % Time for GPU activities
100%
80%
60%
40%
20%
0%
1
32
1024
32768
2^20
2^25
2^29-1
HtoD
saxpy_gpu
DtoH

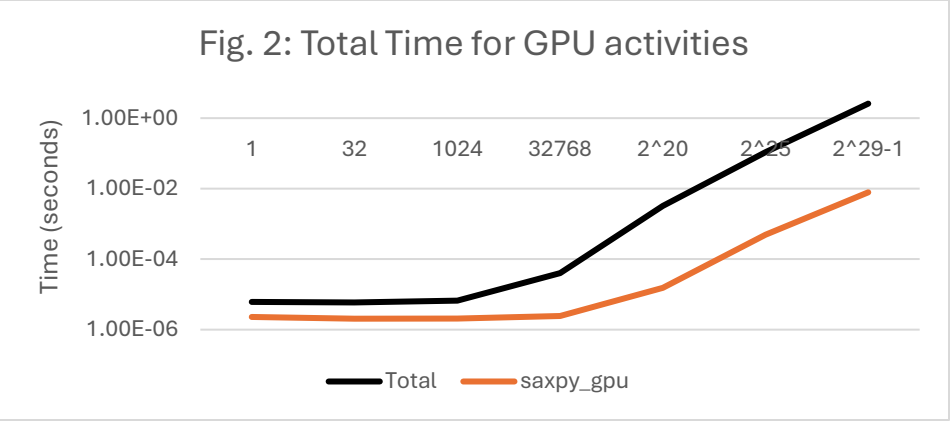
Fig. 2: Total Time for GPU activities
Time (seconds)
1.00E+00
1.00E-02
1.00E-04
1.00E-06
1
32
1024
32768
2^20
2^25
2^29-1
Total
saxpy_gpu

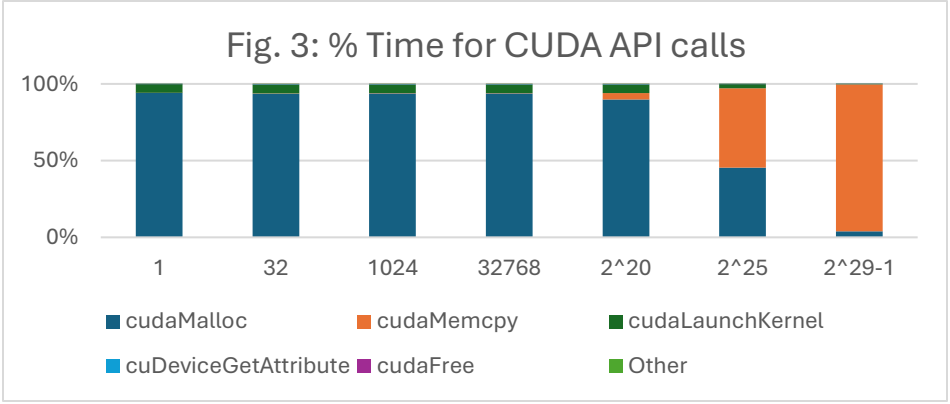
Fig. 3: % Time for CUDA API calls
100%
50%
0%
1
32
1024
32768
2^20
2^25
2^29-1
cudaMalloc
cudaMemcpy
cudaLaunchKernel
cuDeviceGetAttribute
cudaFree
Other

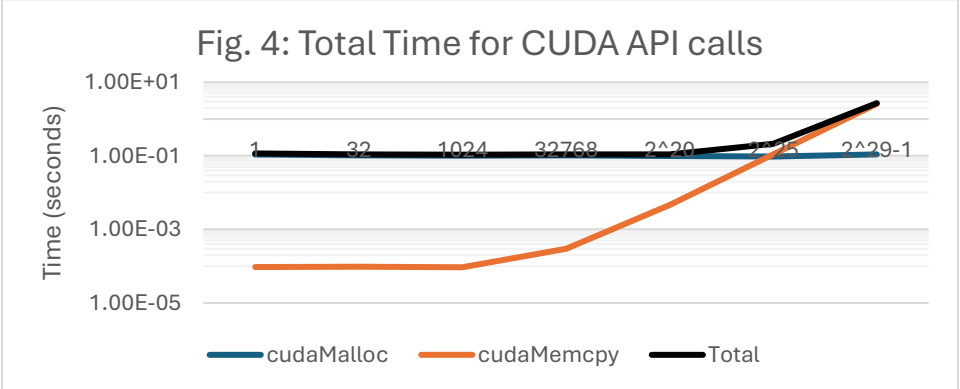
Fig. 4: Total Time for CUDA API calls
Time (seconds)
1.00E+01
1.00E-01
1.00E-03
1.00E-05
1
32
1024
32768
2^20
2^25
2^29-1
cudaMalloc
cudaMemcpy
Total

PART B: Monte Carlo estimation of π

Monte Carlo methods rely on repeated random sampling to get results; more points should yield a more accurate result. This was used here to estimate the value of pi. Note that the ratio of the area of a quarter circle with radius r to a square of side length r is π/4. This program estimates this ratio by counting which randomly sampled points withing the 1x1 square fall within the quarter circle.

Execution time for kernel, memory activity

The program was tested with inputs generateThreadCount and sampleSize, with sizes ranging from 1 (2^0) to 2^{29}-1, with no crashes. Test plots below (Fig. 5-8) narrow the testing ranges to focus on interesting results. Thread blocks are 16x16. The XORWOW random number generator was used to generate numbers on the GPU, no CPU-to-GPU transfer needed. `generateThreadCount` is the number of threads spawned (may exceed max concurrent threads for GPU), and `sampleSize` is the number of sample points tested by each thread. Each thread's hit count is then transferred to the CPU as an array, one element per thread, which is ultimately summed up by the CPU to make a pi estimate. Notable observations:

- GPU-to-CPU (DtoH) transfer of final hit counts take a small portion of the total `cudaMemcpy` run time; the majority is taken by calls within curand library functions that transfer generated numbers within the GPU memory hierarchy. Two notes:
    - DtoH runtime is a function of the size of the hit count array transferred via GPU-to-CPU. Below, 2^12 threads, DtoH runtime becomes constant, as only one memory block is transferred (Figs 5, 7). This agrees with SAXPY observations.
    - Holding thread count constant, `cudaMemcpy` remains constant until about 2^20 sample size, which it then accelerates until around 2^13, where the runtime increase is 1:1 with sample size increase (Figs 6, 8). This gradual acceleration is likely due to differing transfer speeds between different levels in the GPU memory hierarchy.
    - Holding sample size constant, `cudaMemcpy` runtime remains constant of increases gradually until 2^14 threads, where it accelerates instantly to a 1:1 increase as a function of thread count (Figs 5, 7). The GPU is fully occupied, and past this point, adding threads have the same effect as adding sample size per thread.
- Runtime for kernel `generatePoints()` is mainly determined by the `cudaMemcpy` subcalls withing curand library functions (Figs 5-8).
- The CUDA API call `cudaOccupancyMaxActiveBlocksPerMultiprocessor()`, which is used to return the max number of concurrent threads in the GPU, takes the largest constant runtime among all calls, at about 100 ms. This pegs the total runtime of the program at a constant 100 ms for small total sample size (# samples per thread * # threads).

- With a fully occupied GPU (thread count > 2^14), total runtime increases once the total sample size exceeds 2^20 * 2^14 = 2^34 (Figs 5, 8).

Fig 5: generatePoints, thread count @ 2^20 sample size

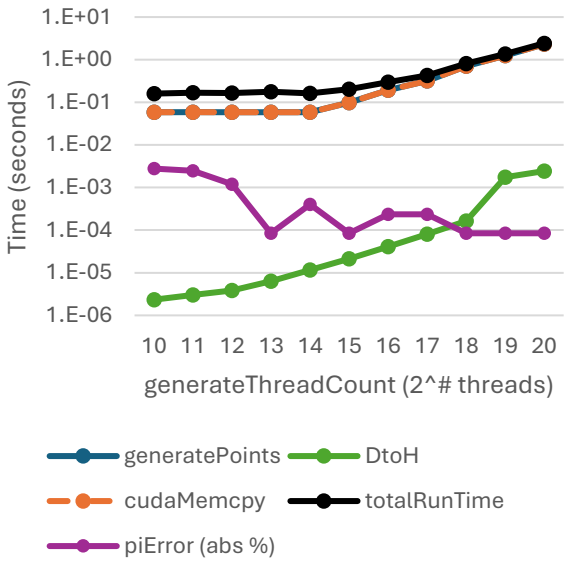

Fig 6: generatePoints sample size @ 2^10 thread count

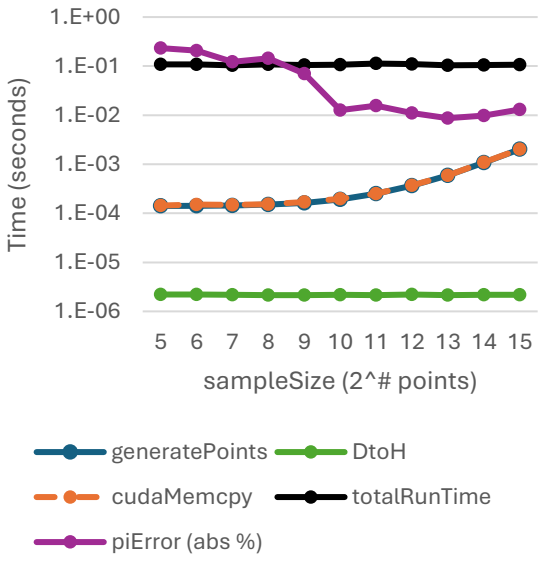

Fig 7: generatePoints, thread count @ 2^5 sample size

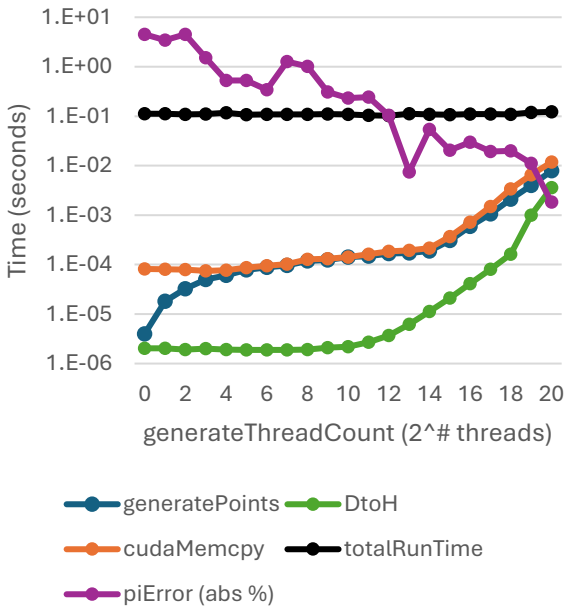

Fig 8: generatePoints, sample size @ 2^20 thread count