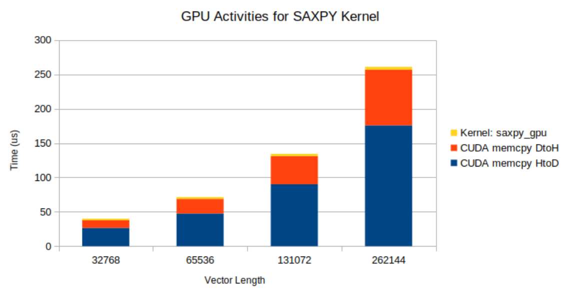
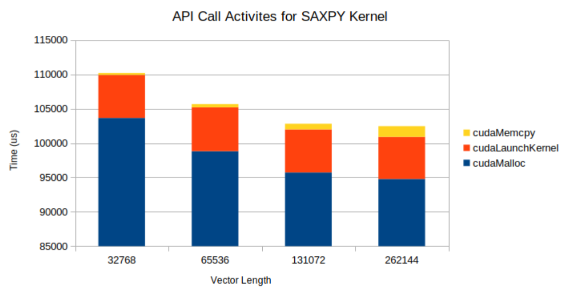
SAXPY Kernel

The SAXPY kernel only has one parameter to be varied, which is the vector size. This is the number of elements in the two x and y vectors that will be operated upon. Using the CPU, these vectors are filled with random numbers from 0 to 999. These are then copied to the GPU, operated upon, and copied back for validation.





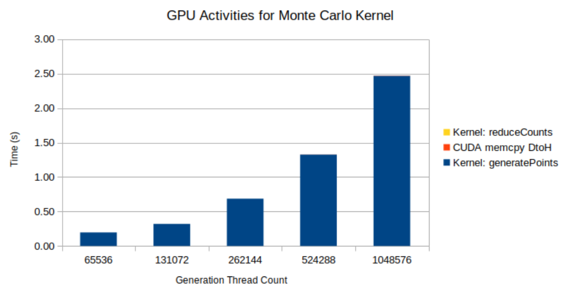
SAXPY Profiling Results

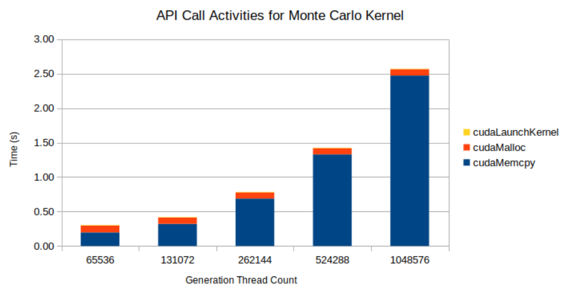
As expected, the SAXPY kernel is entirely memory bound, due to the low amount if processing that each individual thread does per byte of data operated upon. Therefore, the largest amount of time is copying the x and y vectors to the GPU, and the next largest is copying the y vector back to main memory (which takes about half of the time). All of the GPU activity times scale linearly with the size of the vector.

Surprisingly, the API call that takes the most amount of time is cudaMalloc. The kernel launch takes two orders of magnitude less time in comparison, and the memcpy takes 3 orders of magnitude less. The only API call runtime that scales with the vector length is the cudaMemcpy, which is linear. The kernel launch stays relatively consistent, and cudaMalloc decreases slightly with an increasing the vector length.

Monte Carlo Kernel

There are 4 different parameters for the Monte Carlo calculation: the amount of generate threads, the sample size for each thread, the amount of reduce threads, and the reduce size for each thread. For all of the profiling runs, the sample size was set to 1 million sample points. The reduce size was set to 128, and the reduce threads set to scale with the amount of generate threads to make sure they all get reduced.





Monte Carlo Profiling Results

Compared to the SAXPY kernel, the Monte Carlo kernel is mainly compute bound. This is due to the fact that all of the random points are generated on the GPU, and each thread of the kernel is generating and counting 1 million test points. This compute time scales linearly with the amount of generate threads, as does copying the results to the CPU. The reduction kernel does not increase substantially with an increase in the number of generation threads, pointing to most of this time being overhead, not calculation.

The API call that takes the most time is cudaMemcpy, again with a linear scaling with respect to the generate thread count. The cudaMalloc and cudaLaunchKernel calls take substantially less time, and are close to constant across the different test passes.