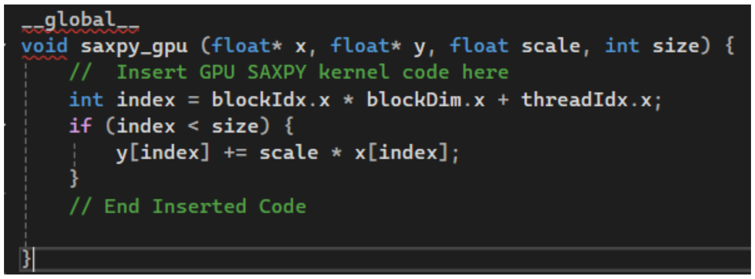
ECE60827 – SP’25 Programming Assign#1

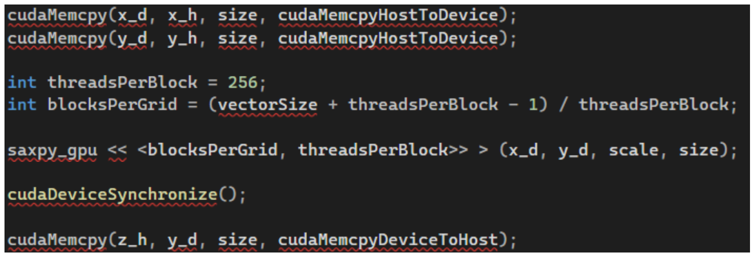
Conor X Devlin, PUID: 0035599728

Part A: SAXPY GPU

For Part A of the assignment we needed to convert the CPU implementation of SAXPY to a parallelized GPU version (short code snippet shown below). Notably, we needed to take *vectorSize* from the *runGpuSaxpy* function which sets the size of vectors *x* and *y.* Inside our CUDA kernel we then assign each thread a value inside vectors *x* and *y* and then *scale* said value of *x* and add it to *y*. In short, GPU SAXPY gets an integer *vectorSize* and then randomly generates *vectorSize* amount of values for two different vectors and then uses a CUDA function to calculate the new values of the vector by individually assigning each’s elements’ scaling action to a single thread. Effectively *vectorSize* amount of threads are generated.



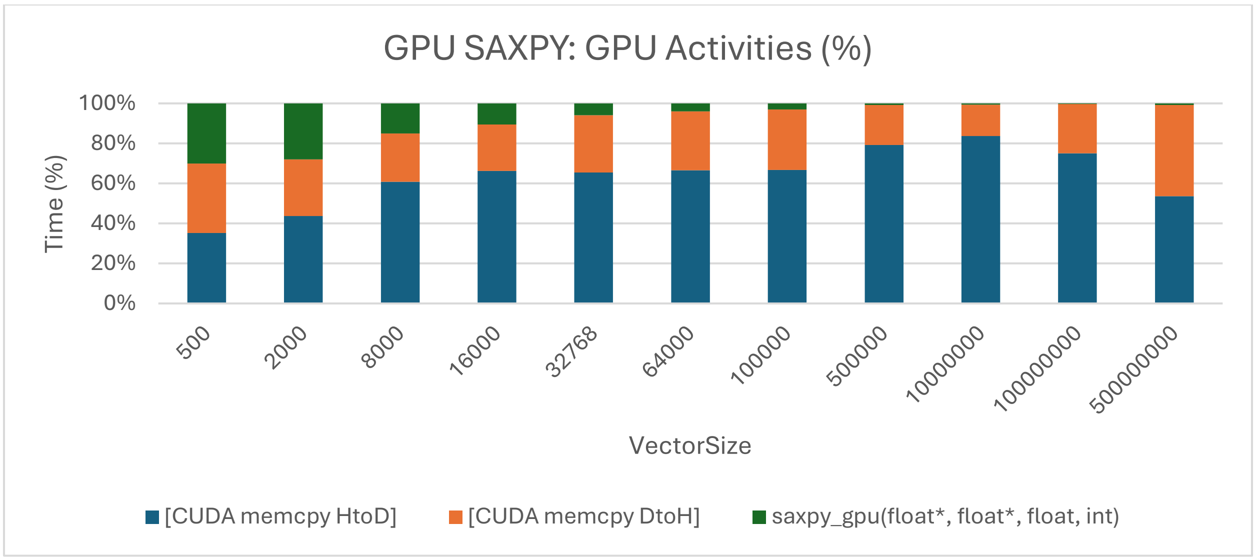
```
__global__
void saxpy_gpu (float* x, float* y, float scale, int size) {
    //  Insert GPU SAXPY kernel code here
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size) {
        y[index] += scale * x[index];
    }
    // End Inserted Code

}
```



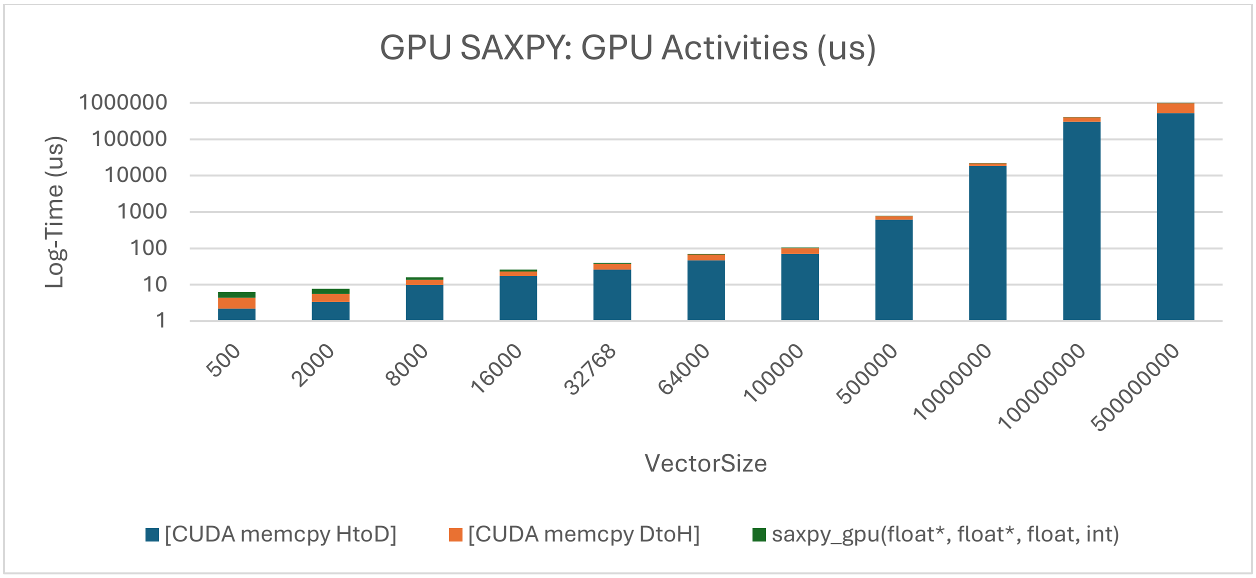
```
cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;
int blocksPerGrid = (vectorSize + threadsPerBlock - 1) / threadsPerBlock;

saxpy_gpu << <blocksPerGrid, threadsPerBlock>> > (x_d, y_d, scale, size);

cudaDeviceSynchronize();

cudaMemcpy(z_h, y_d, size, cudaMemcpyDeviceToHost);
```

Figure 1: GPU SAXPY CUDA kernel and kernel call.

Depicted below are Graphs 1 and 2 which show the percentage and raw time (microseconds) spent in each of the three GPU activities related to GPU SAXPY: copying memory from the host to device, device to the host and the actual function itself.

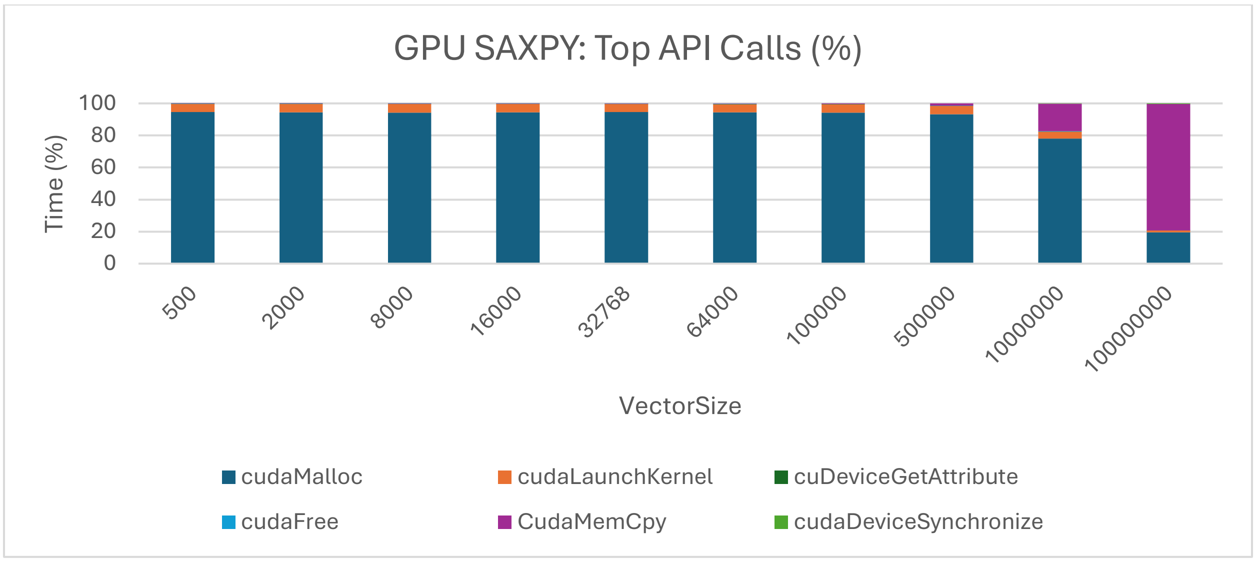


Graph 1: GPU SAXPY - GPU Activities time (%) spent with varying vectorSize.

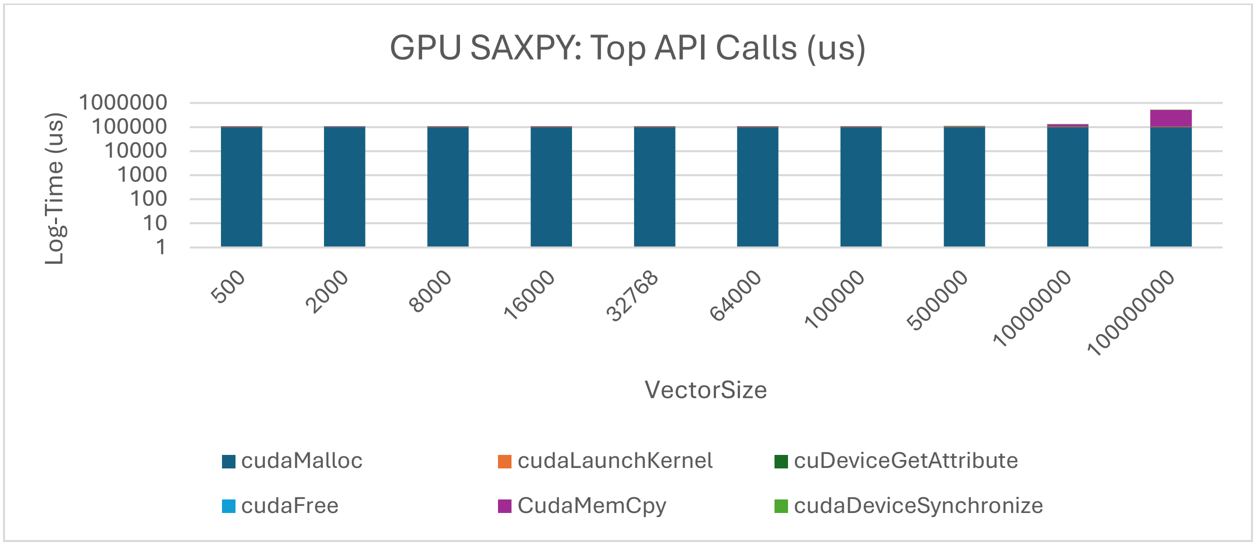


Graph 2: GPU SAXPY - GPU Activities time (us) spent with varying vectorSize.

Further depicted in Graphs 3 and 4 are the API calls and their percentage and raw time (microseconds) spent for GPU SAXPY, I recorded the five largest API calls: cudaMalloc, cudaLaunchKernel, cuDeviceGetAttribute, cudaFree and cudaMemCpy.



Graph 3: GPU SAXPY – API Calls time (%) spent with varying vectorSize.

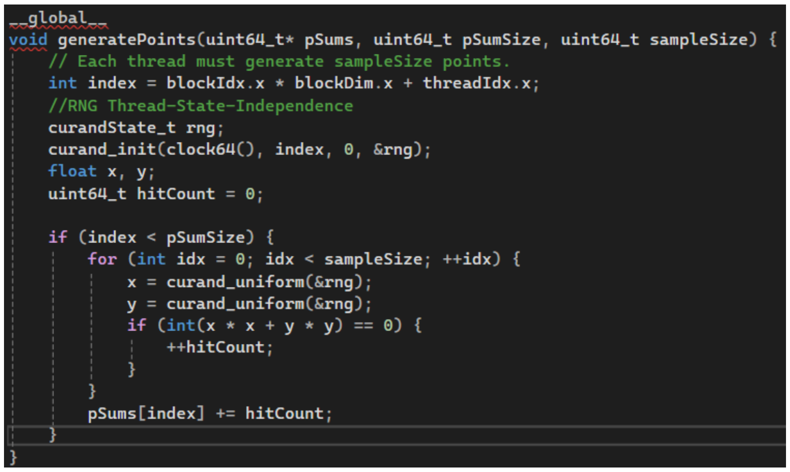


Graph 4: GPU SAXPY – API Calls time (us) spent with varying vectorSize.

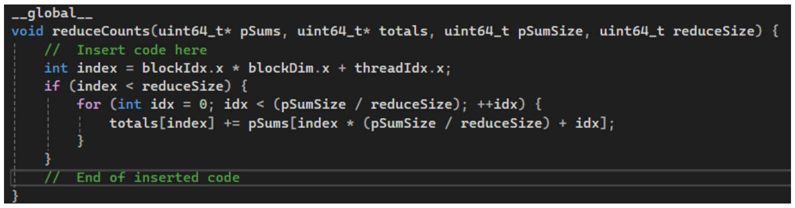
Considering GPU activities we find that the actual execute of the function is really only noticeable with small vector sizes <100K elements and even then it's share of the overall execution time is small (aside from 500 elements) and instead is dominated by the memory copy of the host to device and device to host. This is further confirmed in the API calls where the allocation of memory is the leading factor in time spent and the only other meaningful API call is CudaMemCpy which is only relevant at 10^8 elements and beyond. Overall as this function is rather just a multiply and add and since we're using every thread to hold just a single element of *x* and *y* the expectation to see the majority of the activity and API calls spent as it pertains to memory is confirmed.

Part B: Monte Carlo Pi Estimation GPU

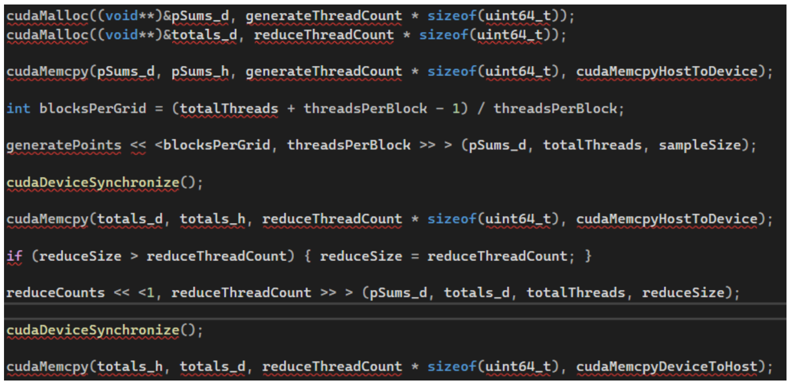
For the second problem we needed to write the kernels such that each GPU thread, defined by *generateThreadCount* will process a predefined number of random points, defined by *sampleSize* on the x-y plane. Then we must reduce those threads filled with samples using a smaller selection of threads, defined by *reduceThreadCount*, and then sum and save those values to a vector of size *reduceSize*. This results in two kernels *generatePoints* and *reduceCounts*, the first generates random points of the amount of *sampleSize* per *generateThreadCount* then adds them up, the second takes the summed vector of size generateThreadCount and reduces it using *reduceThreadCount* threads to a vector of size *reduceSize*. We then add up those final values and perform the Pi approximation on the host (code snippet below).



```
__global__
void generatePoints(uint64_t* pSums, uint64_t pSumSize, uint64_t sampleSize) {
    // Each thread must generate sampleSize points.
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    //RNG Thread-State-Independence
    curandState_t rng;
    curand_init(clock64(), index, 0, &rng);
    float x, y;
    uint64_t hitCount = 0;

    if (index < pSumSize) {
        for (int idx = 0; idx < sampleSize; ++idx) {
            x = curand_uniform(&rng);
            y = curand_uniform(&rng);
            if (int(x * x + y * y) == 0) {
                ++hitCount;
            }
        }
        pSums[index] += hitCount;
    }
}
```



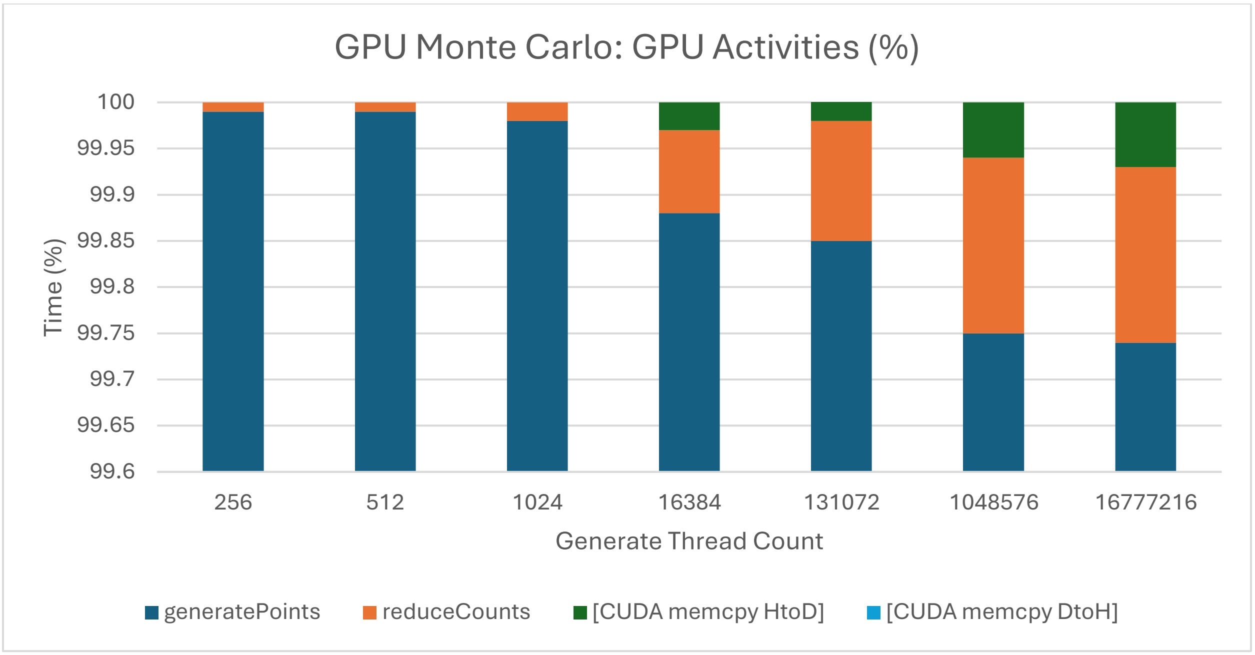
```
__global__
void reduceCounts(uint64_t* pSums, uint64_t* totals, uint64_t pSumSize, uint64_t reduceSize) {
    //  Insert code here
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < reduceSize) {
        for (int idx = 0; idx < (pSumSize / reduceSize); ++idx) {
            totals[index] += pSums[index * (pSumSize / reduceSize) + idx];
        }
    }
    //  End of inserted code
}
```



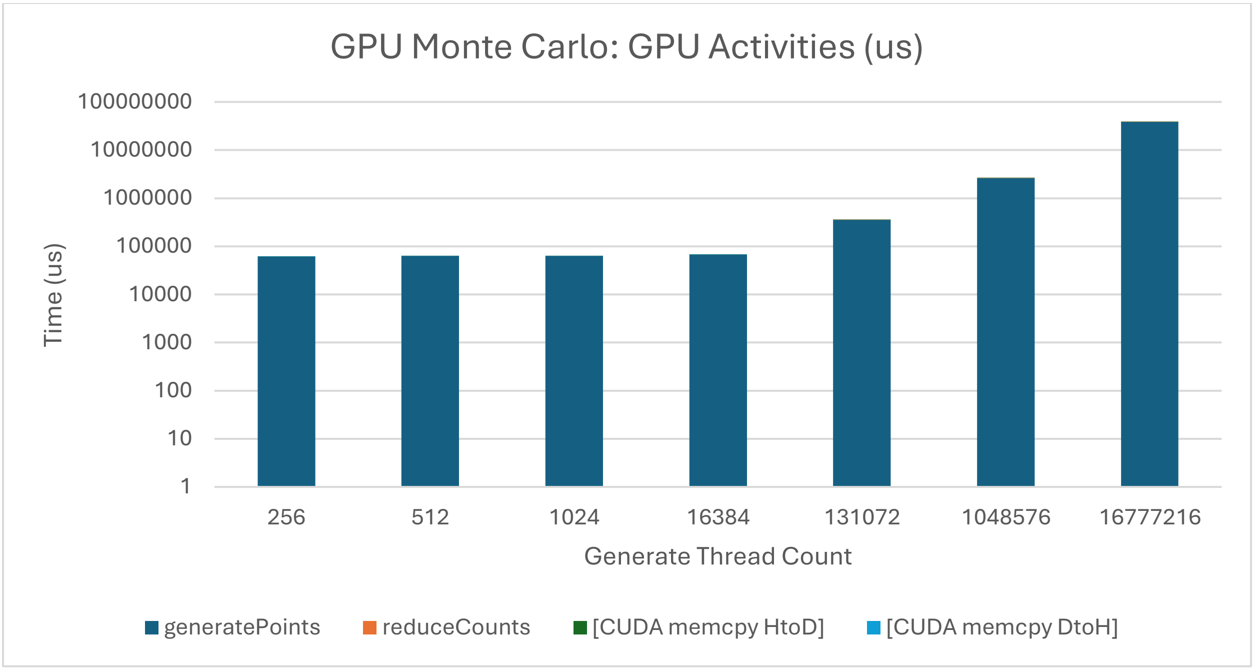
```
cudaMalloc((void**)&pSums_d, generateThreadCount * sizeof(uint64_t));
cudaMalloc((void**)&totals_d, reduceThreadCount * sizeof(uint64_t));

cudaMemcpy(pSums_d, pSums_h, generateThreadCount * sizeof(uint64_t), cudaMemcpyHostToDevice);

int blocksPerGrid = (totalThreads + threadsPerBlock - 1) / threadsPerBlock;

generatePoints << <blocksPerGrid, threadsPerBlock >> > (pSums_d, totalThreads, sampleSize);

cudaDeviceSynchronize();

cudaMemcpy(totals_d, totals_h, reduceThreadCount * sizeof(uint64_t), cudaMemcpyHostToDevice);

if (reduceSize > reduceThreadCount) { reduceSize = reduceThreadCount; }

reduceCounts << <1, reduceThreadCount >> > (pSums_d, totals_d, totalThreads, reduceSize);

cudaDeviceSynchronize();

cudaMemcpy(totals_h, totals_d, reduceThreadCount * sizeof(uint64_t), cudaMemcpyDeviceToHost);
```

Figure 2: GPU Monte Carlo CUDA kernels and kernel calls.

In Graphs 5 and 6 below, the percentage and raw time (microseconds) spent in each of the three GPU activities related to GPU Monte Carlo: copying memory from the host to device, device to the host and the generatePoints and reduceCounts functions are depicted.

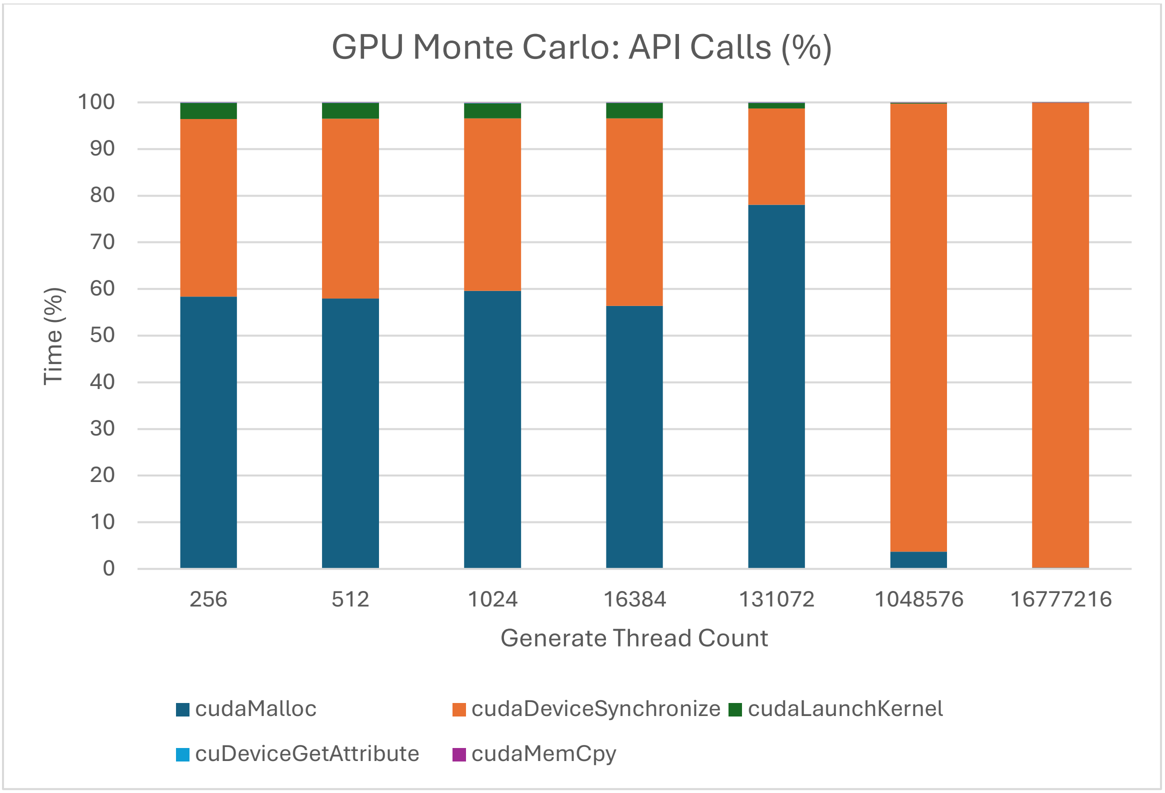


Graph 5: GPU Monte Carlo – GPU activities time (%) spent with varying generateThreadCounts.

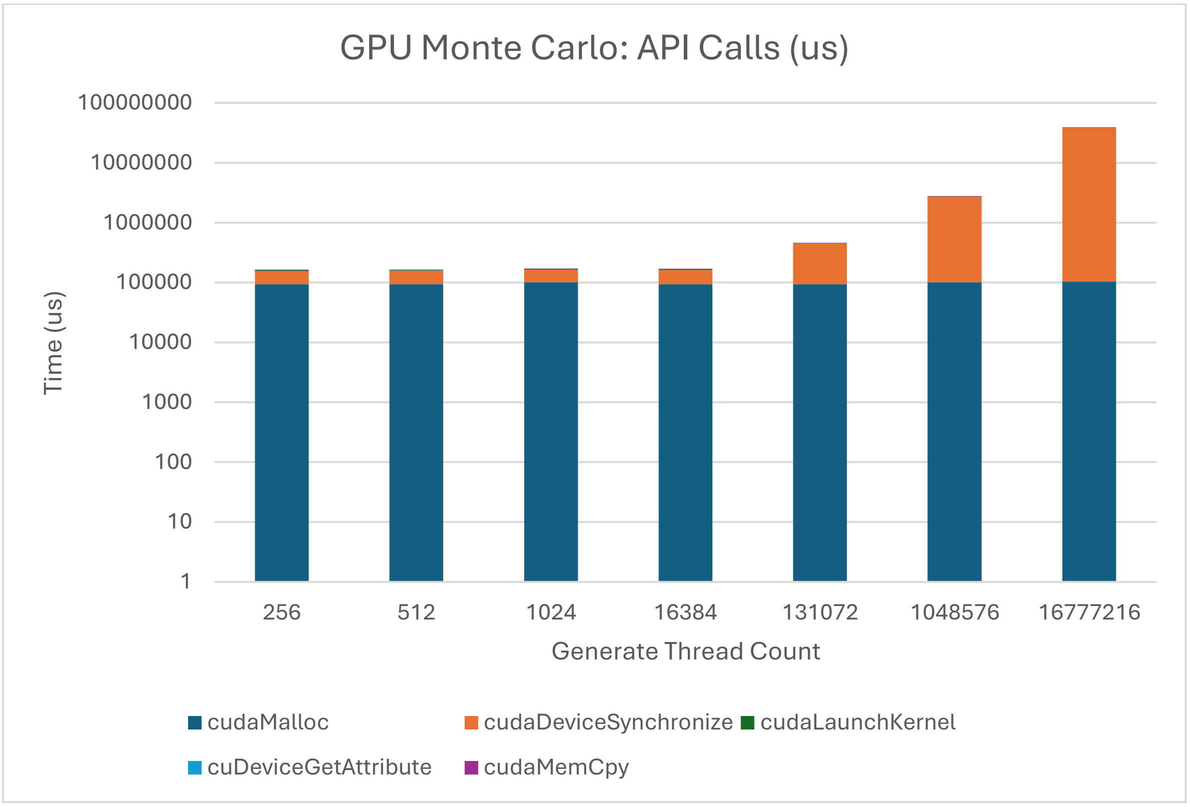


Graph 6: GPU Monte Carlo – GPU activities time (us) spent with varying generateThreadCounts.

In Graphs 7 and 8 below, the percentage and raw time (microseconds) spent in each of the five largest API calls related to GPU Monte Carlo: cudaMalloc, cudaLaunchKernel, cuDeviceGetAttribute, cudaFree and cudaMemCpy are depicted.



Graph 7: GPU Monte Carlo – API calls time (%) spent with varying generateThreadCounts



Graph 8: GPU Monte Carlo – API calls time (%) spent with varying generateThreadCounts

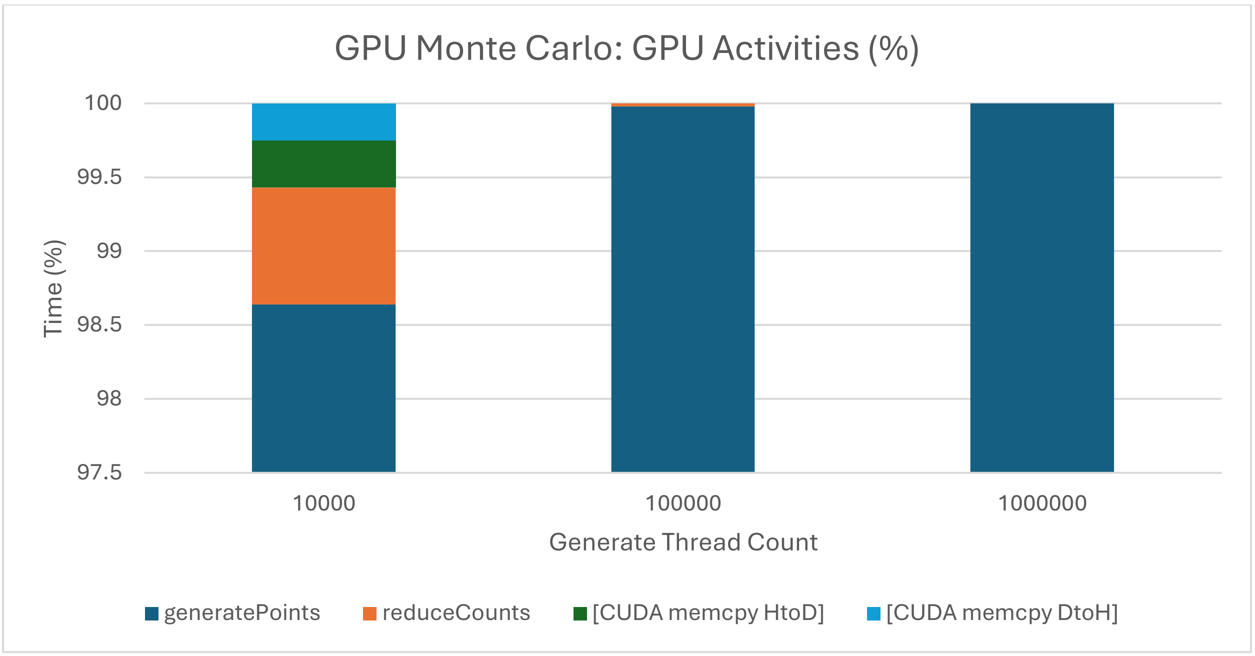
We depict further graphs below which vary additional parameters but there are several interesting things to note about the above four graphs. First is that in the GPU activities (contrasted with SAXPY), the kernels themselves, specifically *generatePoints*, dominate the time spent this time around, largely because the kernel consists of much more than a multiply and add. Moreover, as the sheer amount of threads increases for *generateThreadCount* the cudaDeviceSynch API call takes up more time increasing significantly after 16,384 threads, even to the point of practically dominating the API calls with greater than 95% of the time spent for threads of size: 2^20 and 2^24.

In Graphs 9-12 below I vary the sample from 10K, 100K and 1M and we see the recurring trend of the generatePoints function dominating throughout and then the API call cudaDeviceSynchronize taking up the majority share after 100K samples.

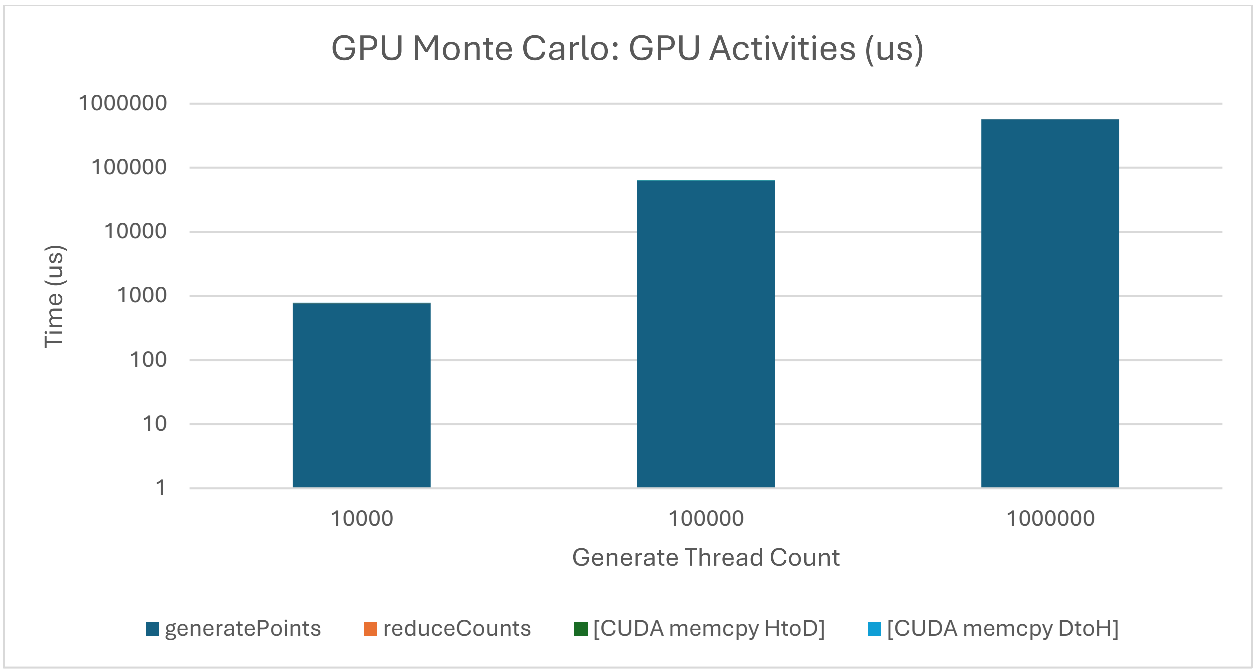
In Graphs 13-16 I vary the reduce thread count from 4, 32 to 128 and we once again repeat the observed trends

Conclusion

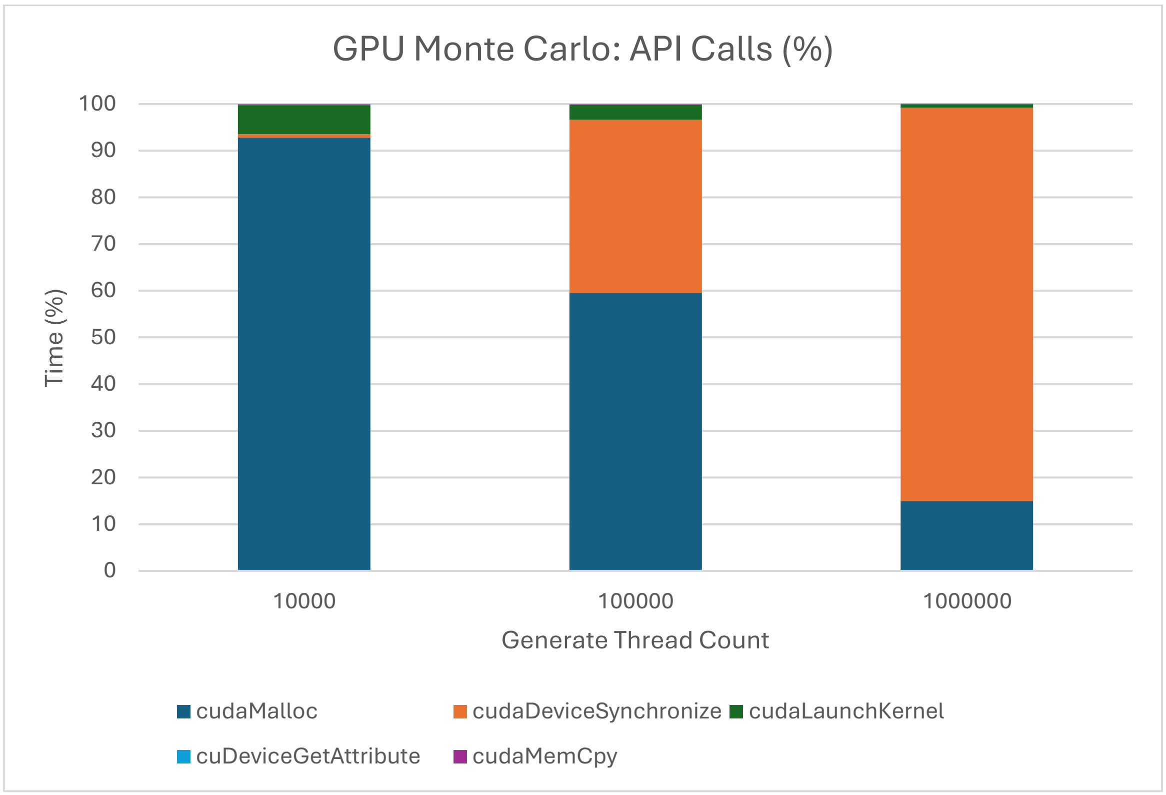
Both parts A and B allow us to see two overarching themes. Part A is a case of memory-bandwidth constraints where the kernel function is performing relatively trivial work but the overall program is largely based on moving lots of information between host and the device. Part B is as case of compute as each thread is independent but performs a more rigorous function generating its own random values and having to individually execute over *sampleSize* amounts of operations.



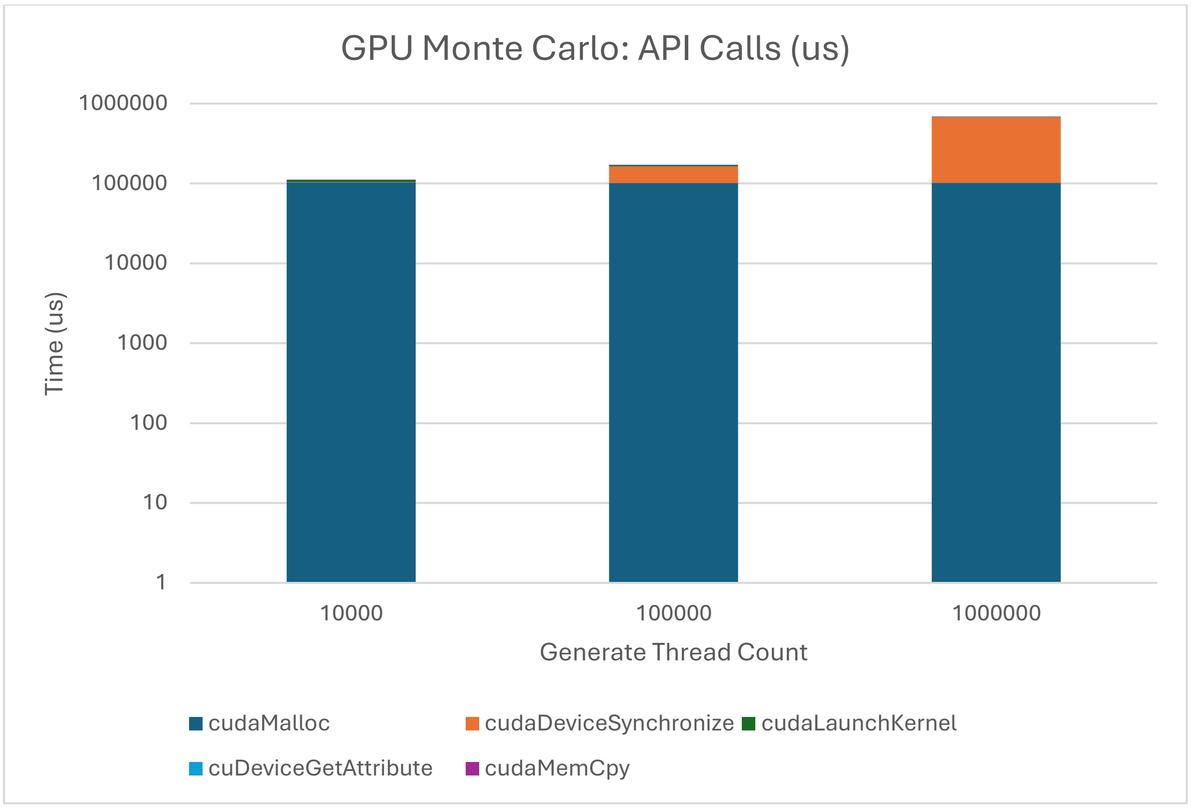
Graph 9: GPU Monte Carlo – GPU Activities time (%) spent with varying sampleSize.



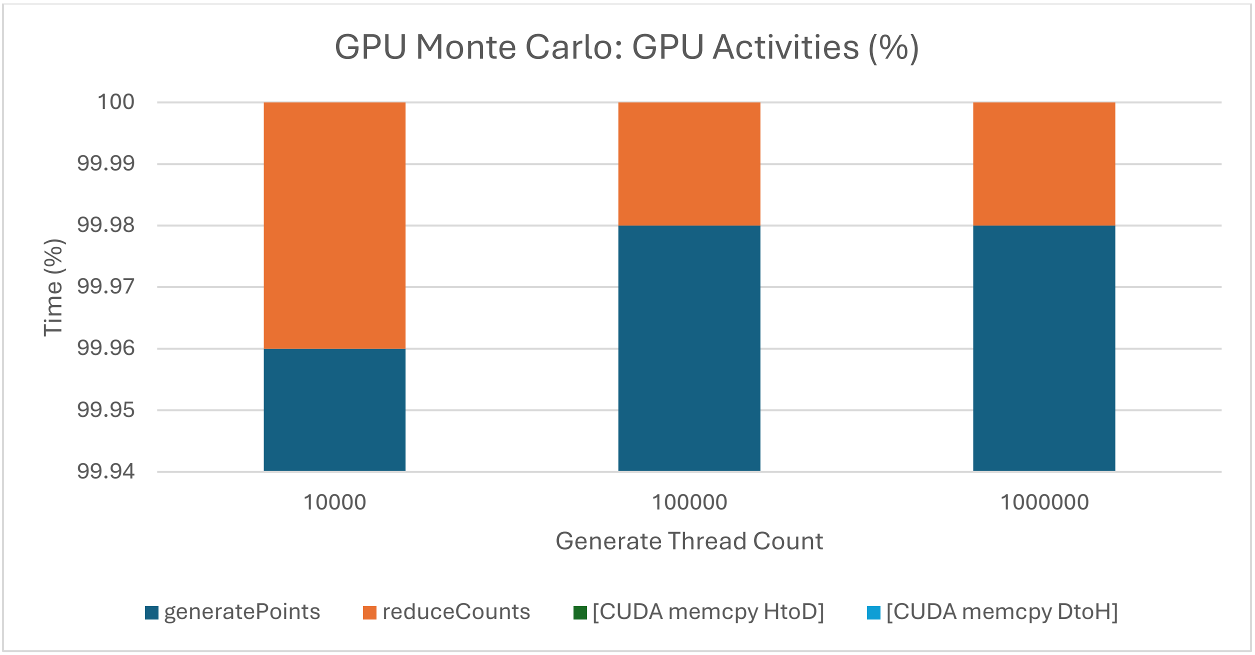
Graph 10: GPU Monte Carlo – GPU Activities time (us) spent with varying sampleSize.



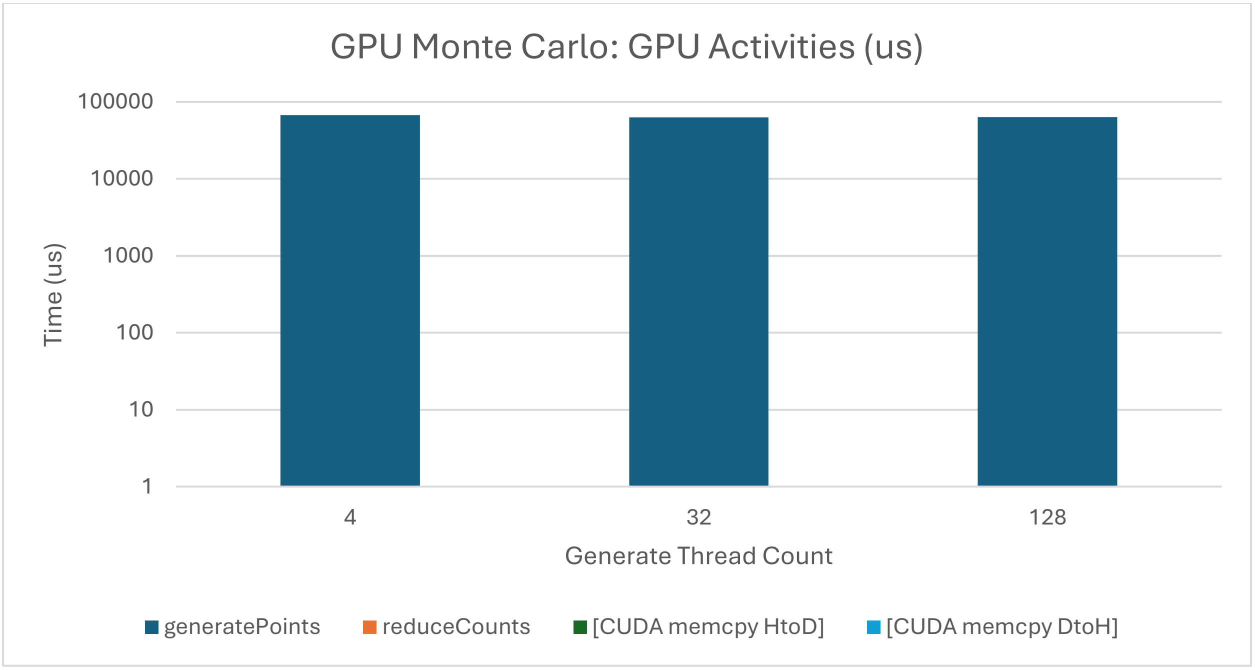
Graph 11: GPU Monte Carlo – API calls time (%) spent with varying sampleSize.



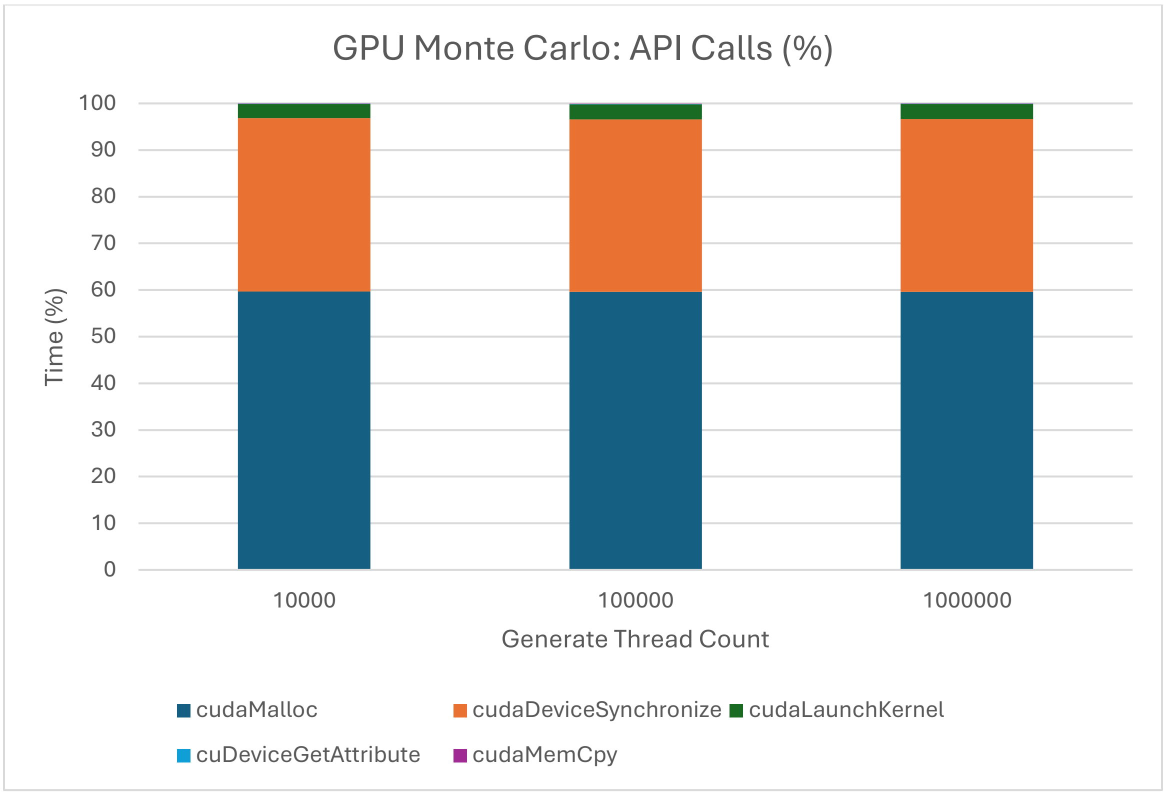
Graph 12: GPU Monte Carlo – API calls time (us) spent with varying sampleSize.



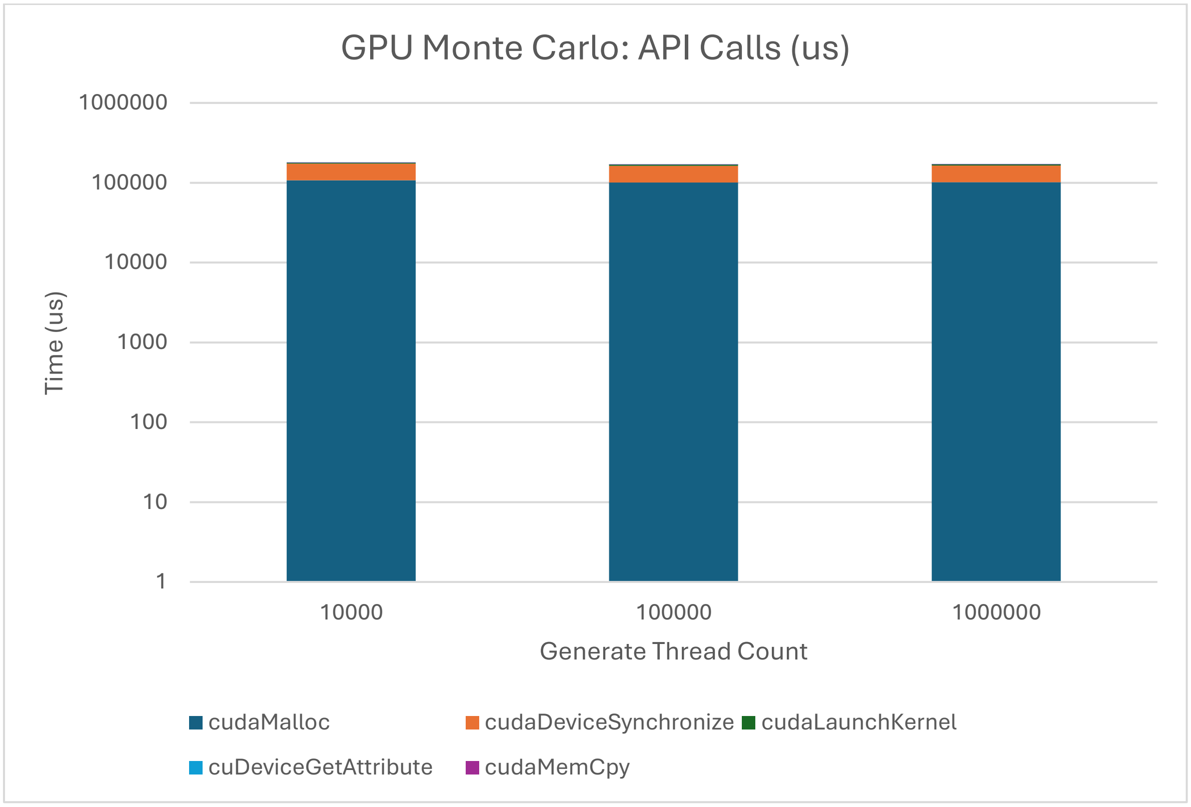
Graph 13: GPU Monte Carlo – GPU Activities time (%) spent with varying reduceThreadCount.



Graph 14: GPU Monte Carlo – GPU Activities time (us) spent with varying reduceThreadCount.



Graph 15: GPU Monte Carlo – API calls time (%) spent with varying reduceThreadCount.



Graph 16: GPU Monte Carlo – API calls time (us) spent with varying reduceThreadCount.