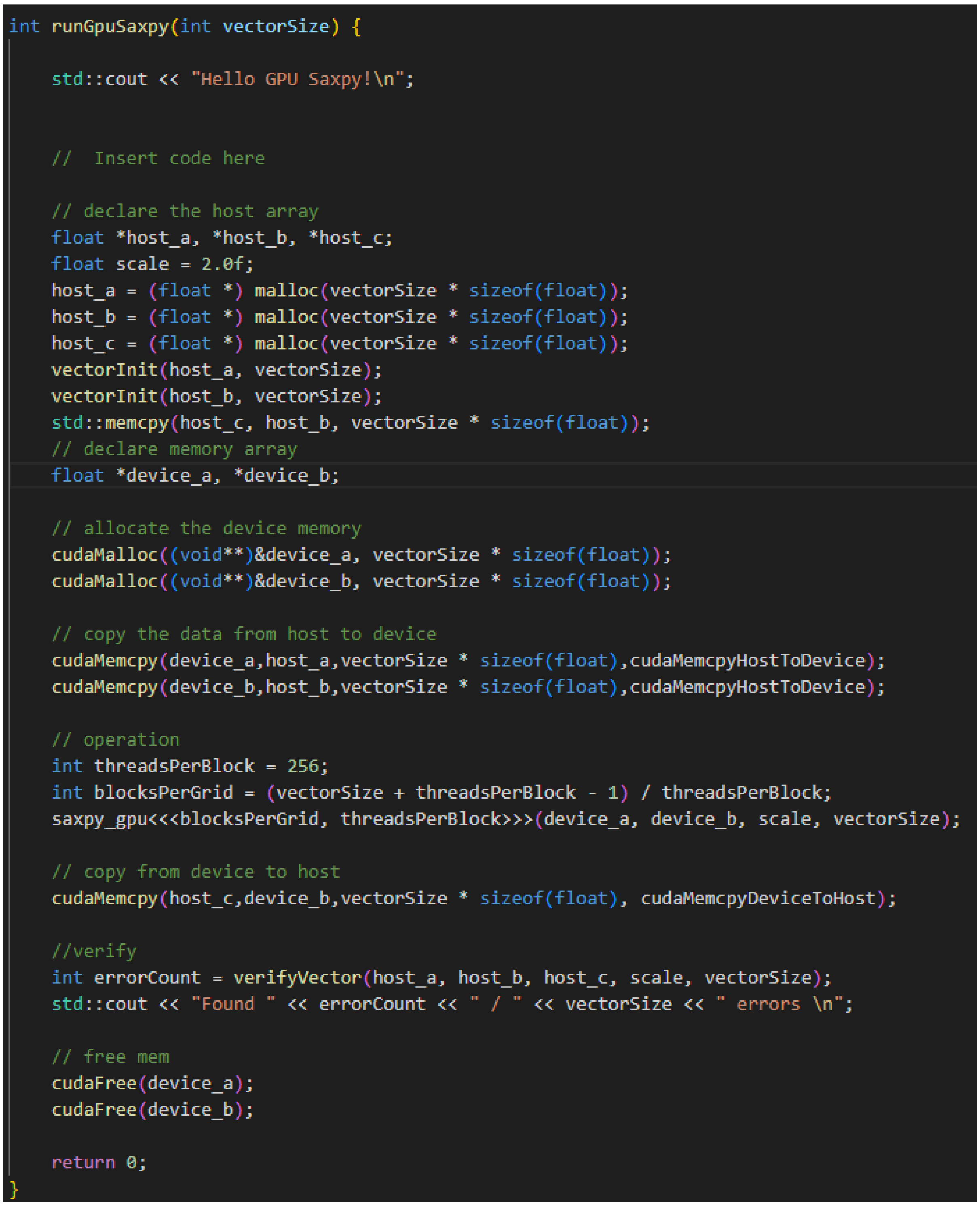
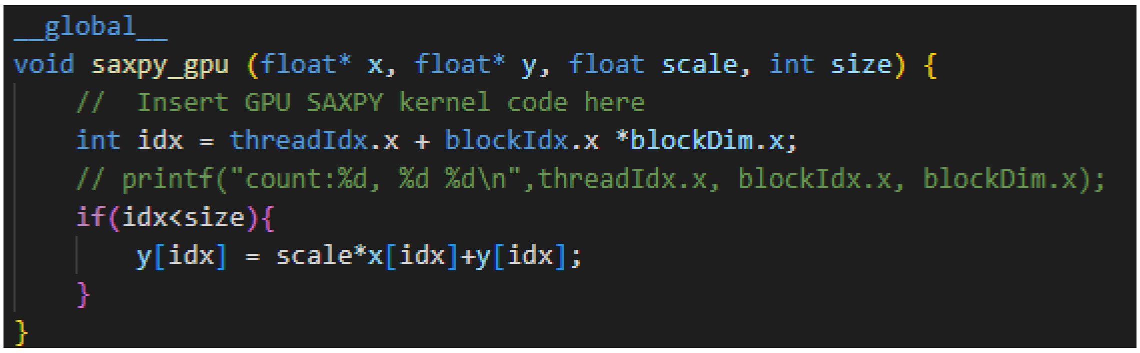
Lab1 Report
Name: Po-Wei Huang
Student ID: 0037713657
Github ID: Justin5567

Part A Saxpy
Design:
In the design of the saxpy, I allocate and init the vector in the host function, after that I pass the array into each thread for them to compute the corresponding value.

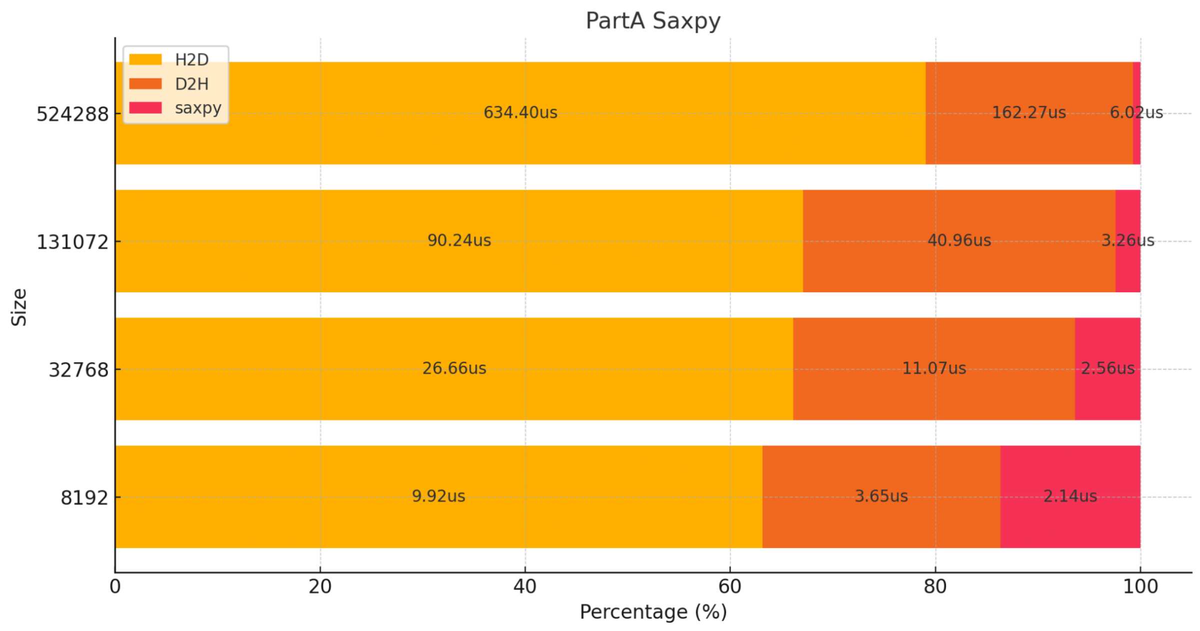


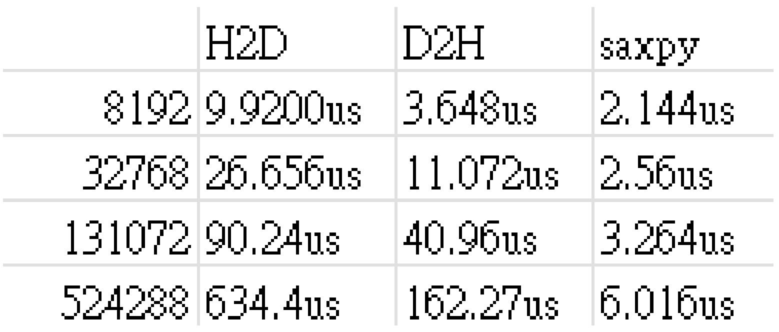
```
int runGpuSaxpy(int vectorSize) {

    std::cout << "Hello GPU Saxpy!\n";


    //  Insert code here

    // declare the host array
    float *host_a, *host_b, *host_c;
    float scale = 2.0f;
    host_a = (float *) malloc(vectorSize * sizeof(float));
    host_b = (float *) malloc(vectorSize * sizeof(float));
    host_c = (float *) malloc(vectorSize * sizeof(float));
    vectorInit(host_a, vectorSize);
    vectorInit(host_b, vectorSize);
    std::memcpy(host_c, host_b, vectorSize * sizeof(float));
    // declare memory array
    float *device_a, *device_b;

    // allocate the device memory
    cudaMalloc((void**)&device_a, vectorSize * sizeof(float));
    cudaMalloc((void**)&device_b, vectorSize * sizeof(float));

    // copy the data from host to device
    cudaMemcpy(device_a,host_a,vectorSize * sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(device_b,host_b,vectorSize * sizeof(float),cudaMemcpyHostToDevice);

    // operation
    int threadsPerBlock = 256;
    int blocksPerGrid = (vectorSize + threadsPerBlock - 1) / threadsPerBlock;
    saxpy_gpu<<<blocksPerGrid, threadsPerBlock>>>(device_a, device_b, scale, vectorSize);

    // copy from device to host
    cudaMemcpy(host_c,device_b,vectorSize * sizeof(float), cudaMemcpyDeviceToHost);

    //verify
    int errorCount = verifyVector(host_a, host_b, host_c, scale, vectorSize);
    std::cout << "Found " << errorCount << " / " << vectorSize << " errors \n";

    // free mem
    cudaFree(device_a);
    cudaFree(device_b);

    return 0;
}
```



```
__global__
void saxpy_gpu (float* x, float* y, float scale, int size) {
    //  Insert GPU SAXPY kernel code here
    int idx = threadIdx.x + blockIdx.x *blockDim.x;
    // printf("count:%d, %d %d\n",threadIdx.x, blockIdx.x, blockDim.x);
    if(idx<size){
        y[idx] = scale*x[idx]+y[idx];
    }
}
```

Test experiments:





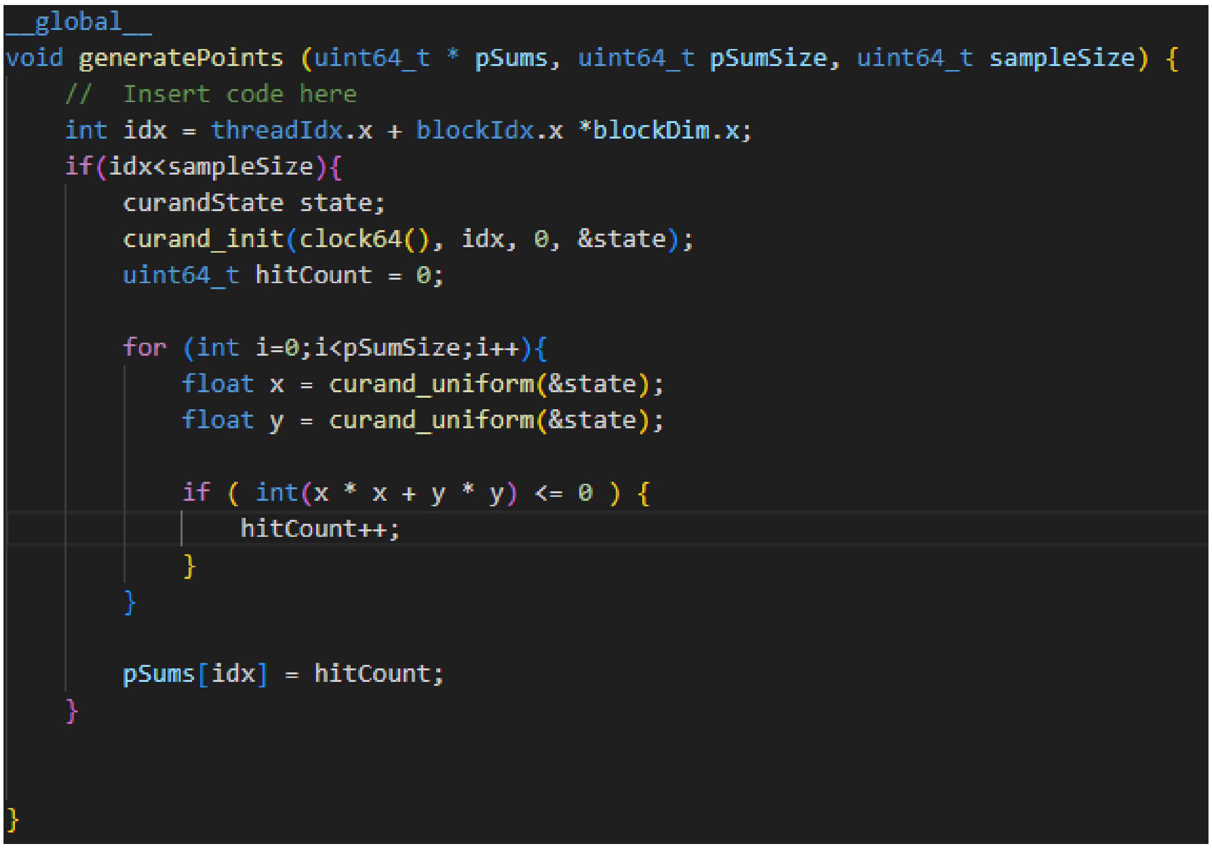
	H2D	D2H	saxpy
8192	9.9200us	3.648us	2.144us
32768	26.656us	11.072us	2.56us
131072	90.24us	40.96us	3.264us
524288	634.4us	162.27us	6.016us

For the Saxpy program, we could see the GPU activities. When the sampleSize keeps in low, CUDA program will not spend too much percentage of time passing the data to the kernel and reading it back(hostToDevice and deviceToHost). However, when the sample size increases, we can see that the time for transporting data will increase dramatically compared to the saxpy execution time. Thus, we can get a conclusion that when the sample size is not large enough, it is better to optimize the kernel code so that it could execute more efficiently; if the code is having a large enough dataset, it is better to optimize the data transport method to better have a lower execution time.

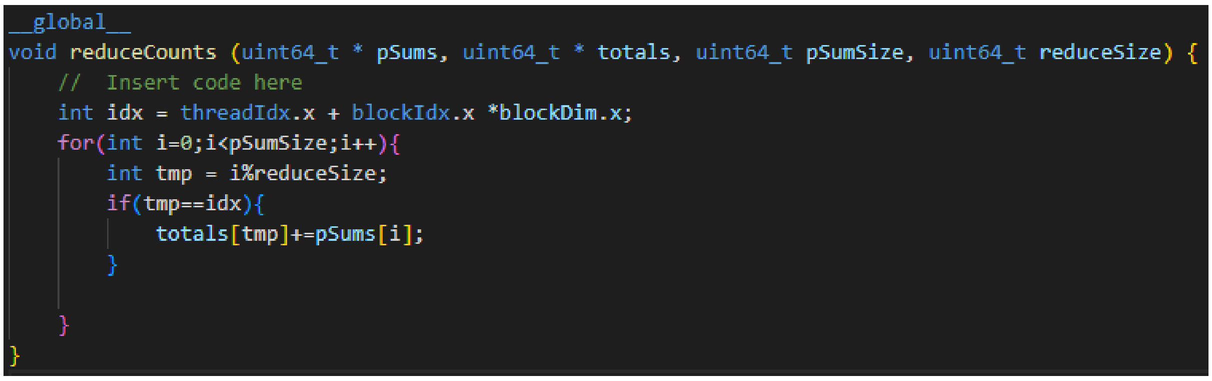
Part B Monte Carlo

Design:

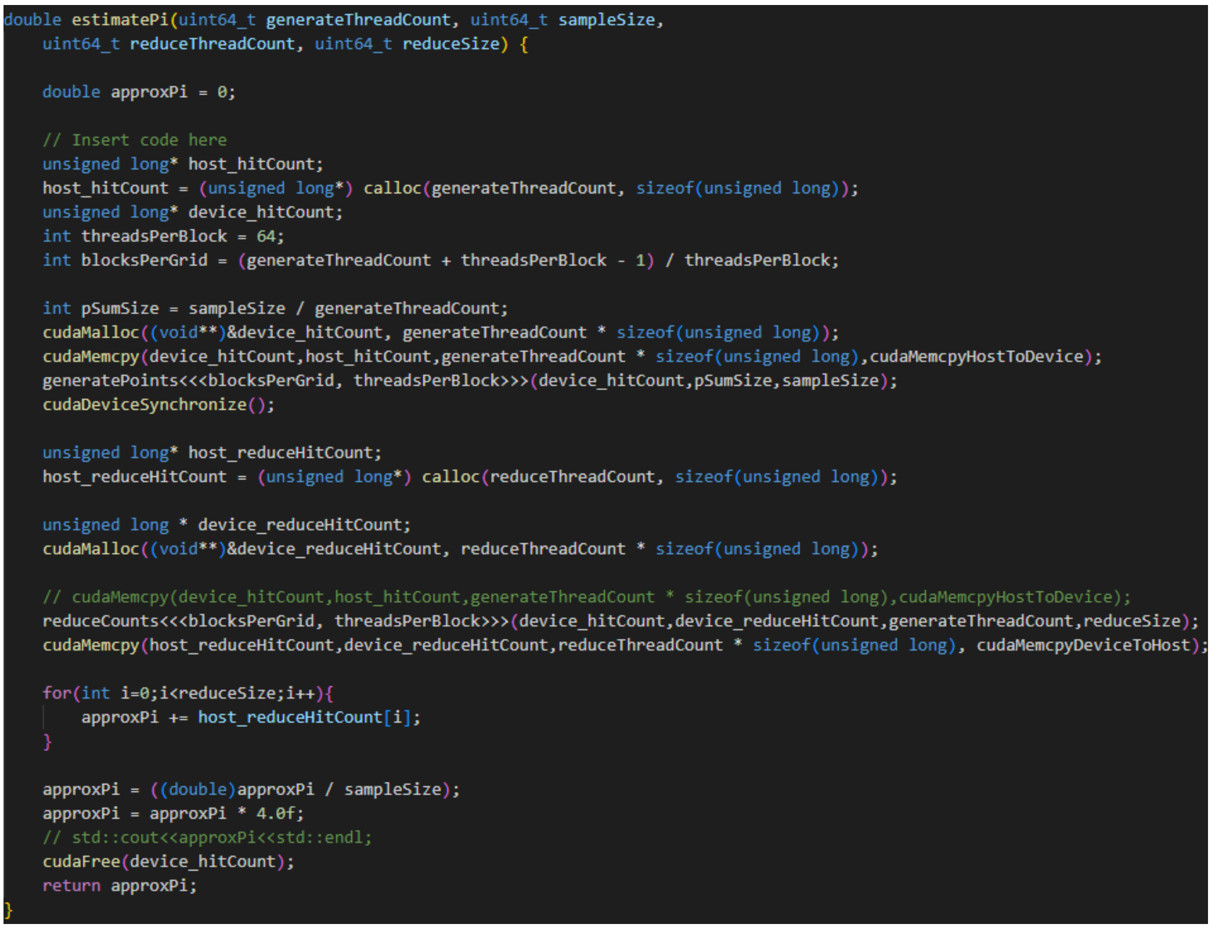
In part B, I implement the code with the sharing design. Every time when the code receives the sampleSize, it would equally split the workload to the existing thread. For example, if we got 32 threads then for each thread it will handle the sample id whose module is equal to its id number. So that we can ensure all of the task can equally split to all of the threads.



```
__global__
void generatePoints (uint64_t * pSums, uint64_t pSumSize, uint64_t sampleSize) {
    //  Insert code here
    int idx = threadIdx.x + blockIdx.x *blockDim.x;
    if(idx<sampleSize){
        curandState state;
        curand_init(clock64(), idx, 0, &state);
        uint64_t hitCount = 0;

        for (int i=0;i<pSumSize;i++){
            float x = curand_uniform(&state);
            float y = curand_uniform(&state);

            if ( int(x * x + y * y) <= 0 ) {
                hitCount++;
            }
        }

        pSums[idx] = hitCount;
    }


}
```

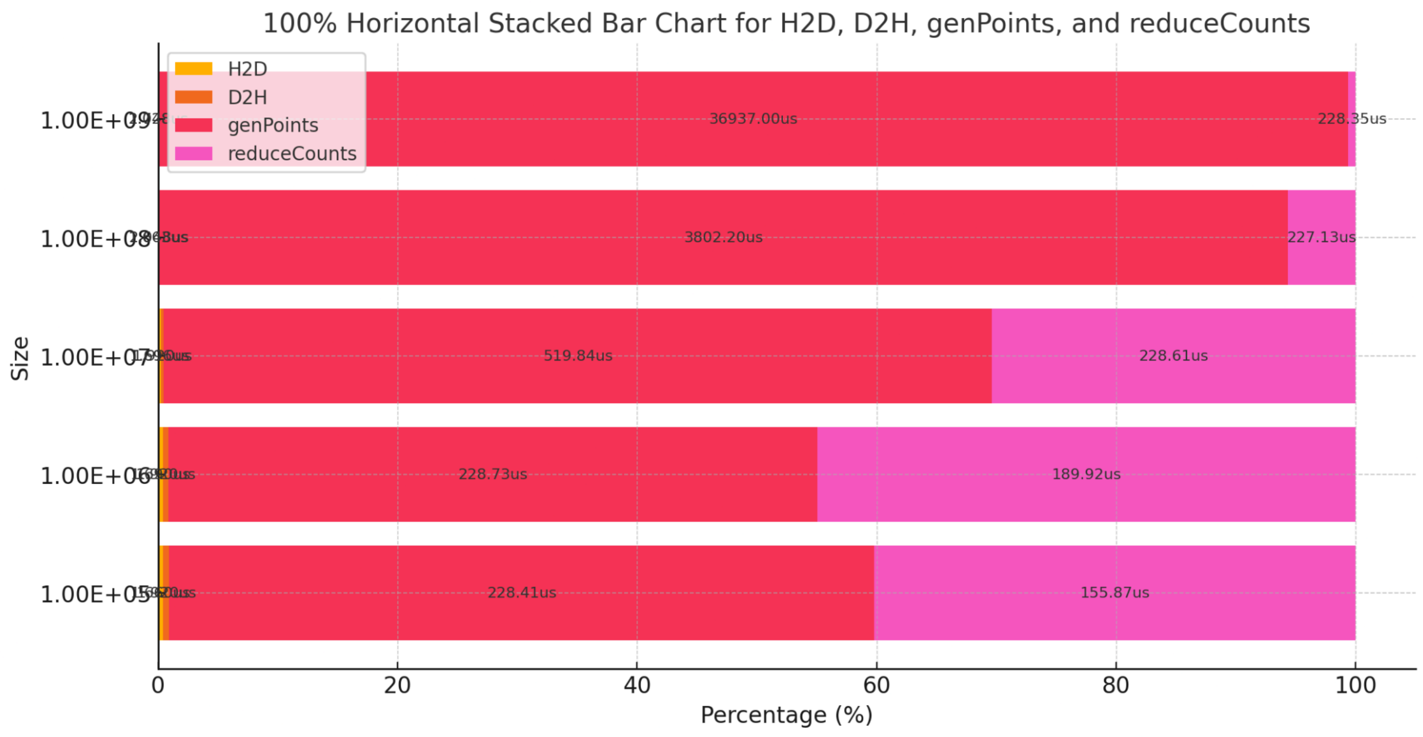


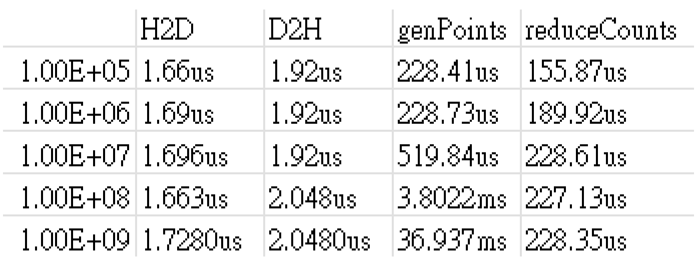
```
__global__
void reduceCounts (uint64_t * pSums, uint64_t * totals, uint64_t pSumSize, uint64_t reduceSize) {
    //  Insert code here
    int idx = threadIdx.x + blockIdx.x *blockDim.x;
    for(int i=0;i<pSumSize;i++){
        int tmp = i%reduceSize;
        if(tmp==idx){
            totals[tmp]+=pSums[i];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

    }
}
```



```
double estimatePi(uint64_t generateThreadCount, uint64_t sampleSize,
    uint64_t reduceThreadCount, uint64_t reduceSize) {

    double approxPi = 0;

    // Insert code here
    unsigned long* host_hitCount;
    host_hitCount = (unsigned long*) calloc(generateThreadCount, sizeof(unsigned long));
    unsigned long* device_hitCount;
    int threadsPerBlock = 64;
    int blocksPerGrid = (generateThreadCount + threadsPerBlock - 1) / threadsPerBlock;

    int pSumSize = sampleSize / generateThreadCount;
    cudaMalloc((void**)&device_hitCount, generateThreadCount * sizeof(unsigned long));
    cudaMemcpy(device_hitCount,host_hitCount,generateThreadCount * sizeof(unsigned long),cudaMemcpyHostToDevice);
    generatePoints<<<blocksPerGrid, threadsPerBlock>>>(device_hitCount,pSumSize,sampleSize);
    cudaDeviceSynchronize();

    unsigned long* host_reduceHitCount;
    host_reduceHitCount = (unsigned long*) calloc(reduceThreadCount, sizeof(unsigned long));

    unsigned long * device_reduceHitCount;
    cudaMalloc((void**)&device_reduceHitCount, reduceThreadCount * sizeof(unsigned long));

    // cudaMemcpy(device_hitCount,host_hitCount,generateThreadCount * sizeof(unsigned long),cudaMemcpyHostToDevice);
    reduceCounts<<<blocksPerGrid, threadsPerBlock>>>(device_hitCount,device_reduceHitCount,generateThreadCount,reduceSize);
    cudaMemcpy(host_reduceHitCount,device_reduceHitCount,reduceThreadCount * sizeof(unsigned long), cudaMemcpyDeviceToHost);

    for(int i=0;i<reduceSize;i++){
        approxPi += host_reduceHitCount[i];
    }

    approxPi = ((double)approxPi / sampleSize);
    approxPi = approxPi * 4.0f;
    // std::cout<<approxPi<<std::endl;
    cudaFree(device_hitCount);
    return approxPi;
}
```

Test experiment:





	H2D	D2H	genPoints	reduceCounts
1.00E+05	1.66us	1.92us	228.41us	155.87us
1.00E+06	1.69us	1.92us	228.73us	189.92us
1.00E+07	1.696us	1.92us	519.84us	228.61us
1.00E+08	1.663us	2.048us	3.8022ms	227.13us
1.00E+09	1.7280us	2.0480us	36.937ms	228.35us

Based on the graph and the results we can see that when the sampleSize increases, the generatePoints function will dominate the execution time due to increased distance calculations. When it comes to fewer sampleSize, others like reduceCount function or the HostToDevice and DeviceToHost will increase their percentages. Also, we can find out that by utilizing the multi-threads strategy, it can dramatically reduce the execution time when the same amount of sampleSize is executing in the cpu version.

Also, the reason why reduceCounts cost a lot is because I used the module function inside of it, if we want to further reduce the execution time we could optimize the function so that it will not take module action inside the for loop.