Program Assignment 1: CUDA Programming and GPGPU-Sim

Brian Park

North Carolina State University, Computer Engineering 786

February 2023

1 Part A: CUDA Programming

Here we do matrix matrix multiplication of the following equation in CUDA:
Defined as:

$$\begin{pmatrix} a'_{b_{n-1},\ldots,b_{t+1},0,b_{t-1},\ldots,b_0} \\ a_{b'_{n-1},\ldots,b_{t+1},1,b_{t-1},\ldots,b_0} \end{pmatrix} = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix} \begin{pmatrix} a_{b_{n-1},\ldots,b_{t+1},0,b_{t-1},\ldots,b_0} \\ a_{b_{n-1},\ldots,b_{t+1},1,b_{t-1},\ldots,b_0} \end{pmatrix}$$

The serial code for CPU is as follows:

```
void quantum_simulation_cpu(float* U, float* a, float* output, size_t qubit, size_t N) {
    // Perform quantum simulation on qubit
    for (size_t i = 0; i < N; i++) {
        if ((i & (1 << qubit)) == 0) {
            output[i] = U[0] * a[i] + U[1] * a[i + (1 << qubit)];
        } else {
            output[i] = U[2] * a[i - (1 << qubit)] + U[3] * a[i];
        }
    }
}
```

And this is the transformed CUDA code:

```
__global__ void quantum_simulation_gpu(const float* U, const float* a, float* output, int qubit,
                                       int N) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    register size_t qid = 1 << qubit;

    if (tid > N)
        return;

    if (tid & qid)
        output[tid] = U[2] * a[tid - qid] + U[3] * a[tid];
    else
        output[tid] = U[0] * a[tid] + U[1] * a[tid + qid];
}
```

I divide the work in the kernel simply by observing patterns in the formula. I notice that per every $1 << qubit$, there are contiguous dot products done on the elements. By simply doing boolean algebra, I can filter out the indices to perform dot products on. I can add an if else statement to make sure that elements don't collide when doing computations, as the first a 0s out the a_{qubit} and the second a fills out a_{qubit} with 1. Note that this will probably incur branch divergence due to conditional statement. Some further optimization sare done such as pinning qid to a register. We could replace the dot product operations with FMADD (fused multiply add) to double performance. Since the compiler flag is already set to `-O3`, observing the PTX assembly using `nvcc -ptx -O3 quamsimV1.cu` we see that compiler already handled that for us by using `fma.rn.f32` instruction.

1.1 Benchmarks

The physical hardware used for the GPUs were the Hydra clusters, which are equipped with a mix of NVIDIA TITAN V and NVIDIA TITAN Xp.

I benchmarked the CUDA kernel for 100 trials on the NVIDIA TITAN V GPU.

On `input.txt` the execution time for version 1 is 5.532 μs for the quantum simulation kernel using `cudaMalloc`. For version 2 using CUDA unified memory semantics via `cudaMallocManaged`, the speed is 10.010 μs, which is 1.81× slower than version 1.

The performance can also be defined not only in latency, but also FLOPS (floating point operations per second). Thus, the performance for version 1 is 69.42 GFLOPS and for version 2 is 38.36 GFLOPS. We see that the second version really takes a hit on the execution time with a tradeoff of programmability. Explicitly managing memory between the CPU and GPU has substantial benefits over letting CUDA handle it via their unified memory programming model.

2 Part B: GPGPU-Sim

When running the quantum simulation code using GPGPU simulator the IPC of the program is 1.29 instructions per cycle. This is calculated using the `gpgpu_simulation_rate` instructions per second and cycles per second reported, which were 261 (inst/sec) and 202 (cycle/sec) respectively.

The data cache miss rate is 100% for L1 and for L2, it is 48.57%. The rates are reported via `L1D_total_cache_miss_rate` and `L2_total_cache_miss_rate`, but can also be calculated by dividing the reported misses by total cache accesses per each level.