CME213, Spring 2022
Parallel Computing
Julian Cooper

Homework 3: CUDA GPU Programming

Problem 1: Recurrence Implement a simple CUDA program for a recurrence relation (inspired by the Mandelbrot Set) for many different starting points.

- **1.1: Allocate GPU memory.** Idea: Use `cudaMalloc` to allocate memory on the GPU device for both the input and output arrays we will need for the recurrence implementation. Free memory with `cudaFree` at end of `main()`.

```
// Allocate num_bytes of memory to the device arrays
cudaMalloc(&device_input_array, num_bytes);
cudaMalloc(&device_output_array, num_bytes);
...
...

// Deallocate memory from both device arrays
cudaFree(device_input_array);
cudaFree(device_output_array);
```

- **1.2: Initialize array of random floats.** Idea: Use in-built rand() functor from the standard cpp library and scale it between -1 and 1 as required.

```
// Initialize an array of size arr_size in input_array with
//random floats between -1 and 1
void initialize_array(vec &input_array, size_t arr_size) {
    input_array.resize(arr_size);
    std::generate(input_array.begin(), input_array.end(), rand);

    for(int i=0; i<arr_size; i++){
    input_array[i] = static_cast<float>(input_array[i])/
                        (RAND_MAX/2)-1;
    }
}
```

- **1.3: Implement recurrence kernel.** Idea: Recurrence operations themselves are not independent and therefore not parallelizable, however, we can parallelize doing many of these recurrence loops for different starting points (constants). So, we want to parallelize over our 1-dim input array of constants.

  Since our kernel needs to handle cases where the number of threads is less than the number of entries in our input array, we need to use a grid-stride loop.

```
/**
* Implement the kernel recurrence.
* The CPU implementation is in host_recurrence() in main_q1.cu.
```

```
*/
__global__ void recurrence(const elem_type* input_array,
                           elem_type* output_array,
                           size_t num_iter,
                           size_t array_length) {

    for (int xid = blockIdx.x * blockDim.x + threadIdx.x;
        xid < array_length;
        xid += blockDim.x * gridDim.x) {

        elem_type z = 0;
        elem_type constant = input_array[xid];

        int it=0;
        while(it<num_iter) {
        z = z * z + constant;
        it++;
        }
        output_array[xid] = z;
    }
}
```

Console logs.

```
Starting at Fri Apr 29 00:28:45 UTC 2022

nvcc -O3 -std=c++11 -arch=compute_75 -code=sm_75 -o main_q1 main_q1.cu

Output from main_q1
----------------
Largest error found at pos: 15 error 7.81565e-08
    expected 1.52526 and got 1.52526

Largest error found at pos: 0 error 0
    expected 0.680375 and got 0.680375

Largest error found at pos: 439038 error 1.19193e-07
    expected 1.00014 and got 1.00014

Largest error found at pos: 142710 error 2.38333e-07
    expected 2.00072 and got 2.00072

Largest error found at pos: 482709 error 5.61004e-07
    expected 16.9994 and got 16.9994

Largest error found at pos: 482709 error 1.15797e-06
    expected 289.897 and got 289.898
```

```
Largest error found at pos: 482709 error 2.324e-06
    expected 84041.4 and got 84041.6

Largest error found at pos: 482709 error 4.63941e-06
    expected 7.06296e+09 and got 7.063e+09

Largest error found at pos: 482709 error 9.25711e-06
    expected 4.98854e+19 and got 4.98859e+19

Largest error found at pos: 138972 error 1.79297e-05
    expected 8.03779e+21 and got 8.03765e+21

Largest error found at pos: 905817 error 2.59306e-05
    expected 1.66519e+35 and got 1.66523e+35

Questions 1.1-1.3: your code passed all the tests!
```

- **1.4: Vary number of threads per block.** When we vary number of threads per block while holding number of iterations (40,000), array size (1e6) and grid size (72) constant, we get the plot shown below.

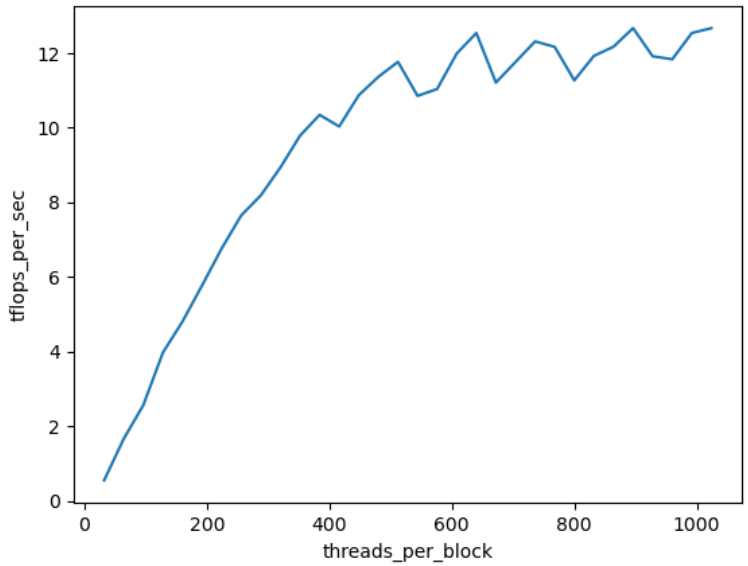

Figure 1: Performance as we vary number of threads per block

Performance improves roughly linearly as we increase threads per block from 32 to 384, after which we get asymptotic behaviour and a saw-tooth pattern.

The initial positive linear relationship comes from simply using more of the gpu's total threads and therefore running more recurrence relations in parallel.

We start to see diminshing returns (asymptotic behaviour) as we add too many threads for the problem size. Our overhead from spinning up additional threads approaches improvement from parallel computation.

The saw-tooth pattern reflects divisors of 1024, which is the maximum number of threads per SM as specified by our hardware specifications. If our number of total threads divides evenly into 1024, we will need one less multiprocessor (including overhead) than if we asked for just one additional thread.

- **1.5: Vary number of blocks.** When we vary number of blocks (grid size) while holding number of iterations (40,000), array size (1e6) and threads per block (128) constant, we get the plot shown below.

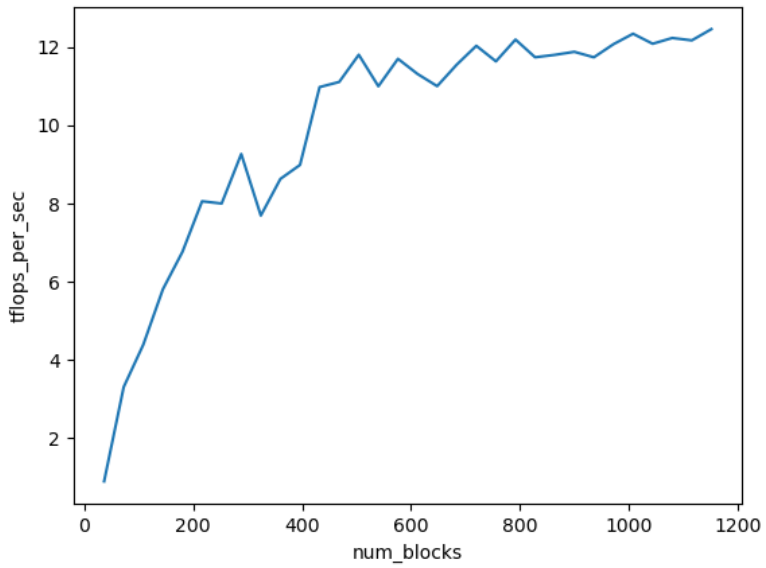

Figure 2: Performance as we vary number of blocks

Performance improves roughly linearly with number of blocks for the first 200 blocks, after which we get asymptotic behaviour.

Similarly to q1.4, the initial positive linear relationship comes from simply using more of the gpu's total threads and therefore running more recurrence relations in parallel.

We start to see diminshing returns (asymptotic behaviour) as we add too many blocks for the problem size. Our overhead from spinning up additional blocks approaches improvement from parallel computation.

- **1.6: Vary number of iterations.** When we vary number of iterations while holding number of threads per block (256), number of blocks (576) and array size (1e6) constant, we get the plot shown below.

  Iterations are done sequentially and so a higher number of iterations means that each thread takes longer to complete its "task" before being reassigned.

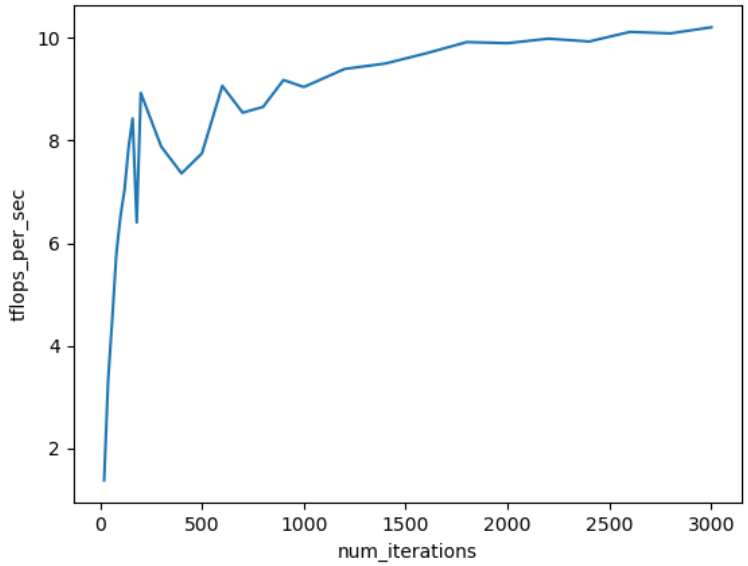

Figure 3: Performance as we vary number of iterations

Performance jumps quickly to 7 Tflops per second as we increase number of iterations from 20-120, after which we get asymptotic behaviour. This is because we are initially memory-bound (overhead of allocating threads outweighs benefits of parallelism) until our task size (number of sequential iterations) is large enough, after which point we approach our compute-bound limit.

It is very difficult to achieve the hardware compute-bound limit with non-trivial computation. In this case we appear to approach 10 Tflops/sec vs our theoretical hardware limit of 14.9 for the NVIDIA Quadro RTX 6000.

Problem 2: PageRank Implement a simplified PageRank link analysis algorithm that generates a score for every node in a graph by considering links to adjacent nodes.

- **2.1: Implement PageRank algorithm.** Idea: Want to parallelize update of each node across gpu threads since independent. Since recurrence relationship itself must be serial, we will need to sync our threads after each complete update. We do this by relaunching the kernel for each iteration step.

  Code snippet for kernel.

```
/*
* Each kernel handles the update of one pagerank score. In other
* words, each kernel handles one row of the update:
*
*       pi(t+1) = A pi(t) + (1 / (2N))
*/
```

```
__global__ void device_graph_propagate(
    const uint *graph_indices,
    const uint *graph_edges,
    const float *graph_nodes_in,
    float *graph_nodes_out,
    const float *inv_edges_per_node,
    int num_nodes
) {
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < num_nodes;
    i += blockDim.x * gridDim.x)
    {
        float sum = 0.f;

        for (int j = graph_indices[i]; j < graph_indices[i+1]; j++)
        {
        sum += graph_nodes_in[graph_edges[j]] *
            inv_edges_per_node[graph_edges[j]];
        }
        graph_nodes_out[i] = 0.5f / (float)num_nodes + 0.5f * sum;
    }
}
```

- **2.2: Compute bandwidth by hand.** Idea: Figure out by hand the number of bytes read from and written to global memory by the algorithm.

  For each iteration, for each node,

  - Read ints `graph_indices[i]` and `graph_indices[i+1]`
  - Read int `graph_edges[j]` * average number of edges
  - Read float `graph_nodes_in[]` * average number of edges
  - Read float `inv_edges_per_node[]` * average number of edges
  - Write float `graph_nodes_out[i]`

  Sub-total = sizeof(int) * (2 + edges) + sizeof(float) * (1 + 2 * edges)

  Total = sub-total * nodes * iterations

```
/**
* This function computes the number of bytes read from and written to
* global memory by the pagerank algorithm.
*
* nodes: the number of nodes in the graph
* edges: the average number of edges in the graph
* iterations: the number of iterations the pagerank algorithm was run
*/
uint get_total_bytes(uint nodes, uint edges, uint iterations)
{
    int subtotal = sizeof(int)*(2+1*edges) + sizeof(float)*(1+2*edges);
    return iterations * nodes * subtotal;
}
```

- **2.3: Plot bandwidth vs number of nodes.** Plot device memory bandwidth in Gb/sec as we increase number of nodes, while holding average number of edges constant.

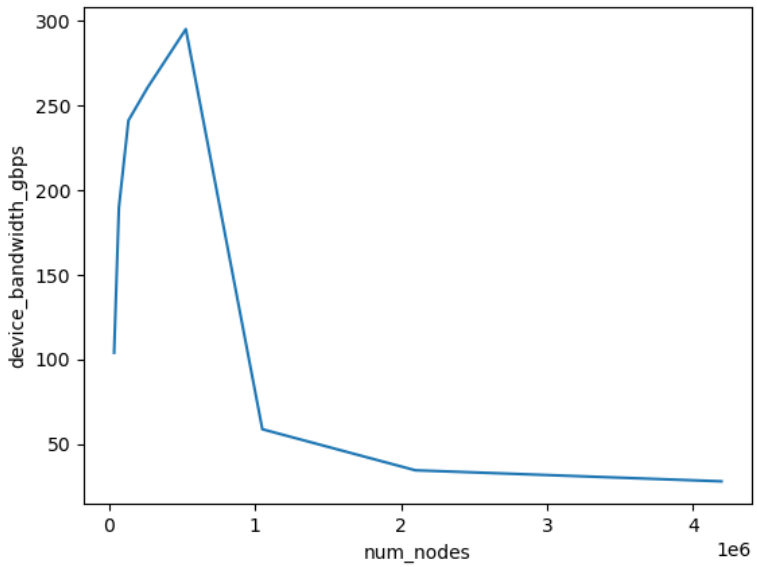

Figure 4: Device bandwidth as we increase number of nodes

- **2.4: Comment on memory bandwidth plot.** Our device bandwidth reaches an optimal of 300 Gb/sec near 500,000 nodes, with performance dropping sharply either side. This is the point where our memory access is most aligned and coalesced!

  What does the memory access pattern look like? Our PageRank program has a mix of sequential and random access patterns. For each node, it fetches 2 contiguous ints that specifies the relevant range of indices in our edge array, which we need access sequentially to find each adjacent node. However, the adjacent nodes themselves are not necessarily stored contiguously in memory within the input and inverse degree arrays, and so accessing these for each edge is roughly random access.

  Explain the difference between Problem 2 and maximum bandwidth of 480 GB/sec? Achieving maximum bandwidth performance would require that our memory access completely aligned and coalesced. We would need our access pattern to be completely sequential to maximize prefetching.

Problem 3: Strided Memory Access Benchmark our device by performing strided memory access in `benchmark.cu`.

- **3.1: Benchmark strided access.** Display semilogy plot of throughput in Gb/sec as a function of stride length.

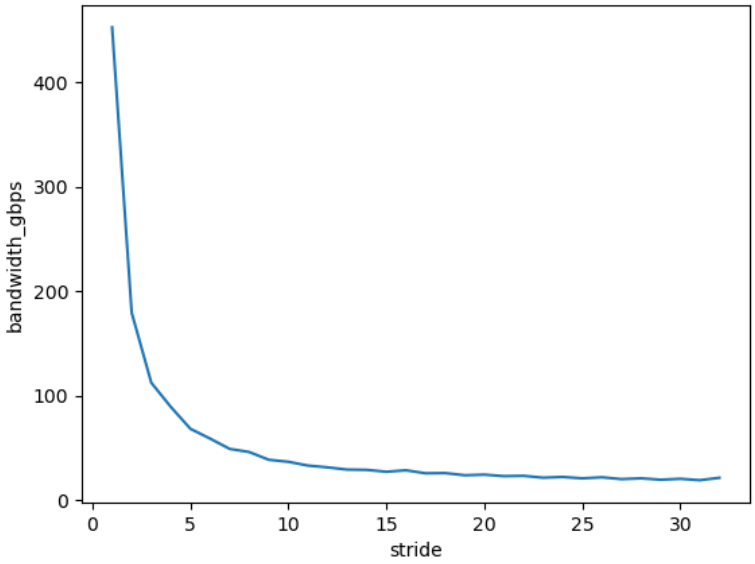

Figure 5: Device bandwidth as we increase stride

- **3.2: Comment on shape of graph.** As expected, we initially observe asymptotic declining performance as we increase stride length. This is because it becomes less and less common for us to "prefetch" memory relevant for a subsequent computation if the next required word does not have a nearby global memory address.

  After some point, our performance plateaus. This corresponds to a stride length of 32. Since cache requests are done at the warp level, and each warp has 32 threads, this corresponds to the point at which every subsequent thread requires global memory access (cache miss).

Submission information logs.

```
jelc@cardinal2:~$ /afs/ir.stanford.edu/class/cme213/script/submit.py hw3 private/cme213-
Submission for assignment 'hw3' as user 'jelc'
Attempt 2/10
Time stamp: 2022-05-01 21:36
List of files being copied:
    private/cme213-jelc53/hw3/main_q1.cu  [13253 bytes]
    private/cme213-jelc53/hw3/recurrence.cuh  [1589 bytes]
    private/cme213-jelc53/hw3/pagerank.cuh  [5894 bytes]
    private/cme213-jelc53/hw3/benchmark.cuh  [795 bytes]

Your files were copied successfully.
Directory where files were copied: /afs/ir.stanford.edu/class/cme213/submissions/hw3/jel
List of files in this directory:
```

```
    main_q1.cu  [13253 bytes]
    recurrence.cuh  [1589 bytes]
    pagerank.cuh  [5894 bytes]
    benchmark.cuh  [795 bytes]
    metadata  [137 bytes]

This completes the submission process. Thank you!

jelc@cardinal2:~$ ls /afs/ir.stanford.edu/class/cme213/submissions/hw3/jelc/2
benchmark.cuh  main_q1.cu  metadata  pagerank.cuh  recurrence.cuh
```