Jarred Parr -- October 17, 2018

CUDA Human Anatomy Modeling

The cuda compiled and executed code provided a significant speedup compared to the CPU bound process. The times for 2 thousand and 20 thousand dimensional vectors were extremely slow by comparison and when I ran with a 2 million entry vector it simply took too long to even bother calculating.

Speedup Analysis

The speed of each GPU calculation is as follows, all trials varied in size with a constant block size of 1024:

- 20000 Dimensional Vector: 134.4 ms
- 200000 Dimensional Vector: 135.68 ms
- 2000000 Dimensional Vector: 141.05 ms
- 20000000 Dimensional Vector: 127.61 ms
- 20000000000 Dimensional Vector: 312.25 ms
- 2000000000000 Dimensional Vector: 1812.86 ms

- 2000: < 1 s
- 200000: < 1 s
- 2000000: 100 ms
- 20000000000 Dimensional Vector Sequential: 75850 ms
- 2000000000000 Dimensional Vector Sequential: 850082 ms

After running with a 20 million all the way up to 2 billion entries, it exhibited the following speedups:

- 2000: < 1
- 20000: < 1
- 200000: ~1
- 20 Million: 594.4
- 2 Billion: 1277.6

Conclusions

It was surprising the astounding amount of overhead that can be accidentally applied to CUDA code when you get things up and running. There were a few times where I had been seeing pretty slow results for the first few implementations of my code when I was still getting familiar with how things were working. I had a few frustrating moments getting more of a handle on how to do some of the memory management, but once I got things under control it was pretty smooth sailing. This was a very interesting and insightful intro to CUDA.

Source Code

```
#include <stdio.h>
#include <iostream>



__global__
```

```
void dot_product(
    unsigned int n,
    unsigned int* force,
    unsigned int* distance,
    unsigned int* product) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride) {
    product[i] += force[i] * distance[i];
  }
}

int main(int argc, char** argv) {
  if (argc < 2) {
    std::cerr << "usage: muscle vector_size threads_per_block" << std::endl;
    return EXIT_FAILURE;
  }

  unsigned int vector_size = atoi(argv[1]);
  unsigned int block_size = atoi(argv[2]);
  int num_blocks = (vector_size * block_size - 1) / block_size;

  unsigned int *force, *distance, *output;

  // Allocated unified memory
  cudaMallocManaged(&force, vector_size * sizeof(unsigned int));
  cudaMallocManaged(&distance, vector_size * sizeof(unsigned int));
  cudaMallocManaged(&output, vector_size * sizeof(unsigned int));

  for (unsigned int i = 0; i < vector_size / 2; ++i) {
    force[i] = (i + 1);
  }

  int val = vector_size / 2;
  for (unsigned int i = vector_size / 2; i < vector_size; ++i) {
    force[i] = val + 1;
    --val;
  }

  for (unsigned int i = 0; i < vector_size; ++i) {
    distance[i] = ((i % 10) + 1);
  }

  dot_product <<< num_blocks, block_size >>>(vector_size, force, distance, output);
  cudaDeviceSynchronize();

  unsigned int sum = 0;
  for (int i = 0; i < vector_size; ++i) {
    sum += output[i];
  }

  std::cout << "output: " << sum << std::endl;
```

```
  cudaFree(force);
  cudaFree(distance);

  return EXIT_SUCCESS;
}
```