CS1645 Project Report: CUDA

I. Introduction

Graphics processing units (GPUs) have multiple cores that can very quickly and efficiently process large data blocks in parallel. Compute Unified Device Architecture (CUDA) is a platform for parallel computing and programming model that utilizes GPUs in order to fully make use of their capacity and better performance. Through this software layer, software engineers can configure and run programs on GPUs for general-purpose mathematical calculations with sizable performance improvements over serial or even parallel programs employing central processing units (CPUs).

While it was certainly possible to utilize the GPU for calculations before the advent of CUDA with preexisting application programming interfaces (APIs) such as Direct3D and OpenGL, they demanded advanced graphics programming skills. The CUDA platform provides compiler directives, CUDA-accelerated libraries, and additions to well-used programming languages like C and C++ that enable parallel programming specialists to employ the resources of the GPU even if they have little graphics programming knowledge. The focus of this project is most specifically CUDA C.

II. Syntax

CUDA C kernels, functions that N different CUDA threads execute N times in parallel, are declared using the __global__ declaration specifier. Execution configuration syntax specifies the number of kernel threads for a given call. These functions run on the GPU

and can be called from CPU code. This code, for example, adds two vectors (A and B) and stores the result into vector C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Memory is allocated using cudaMallocManaged() and deallocated using cudaFree(). This unified memory can be accessed by every CPU and GPU in a system. Here is a simple example displaying these functions:

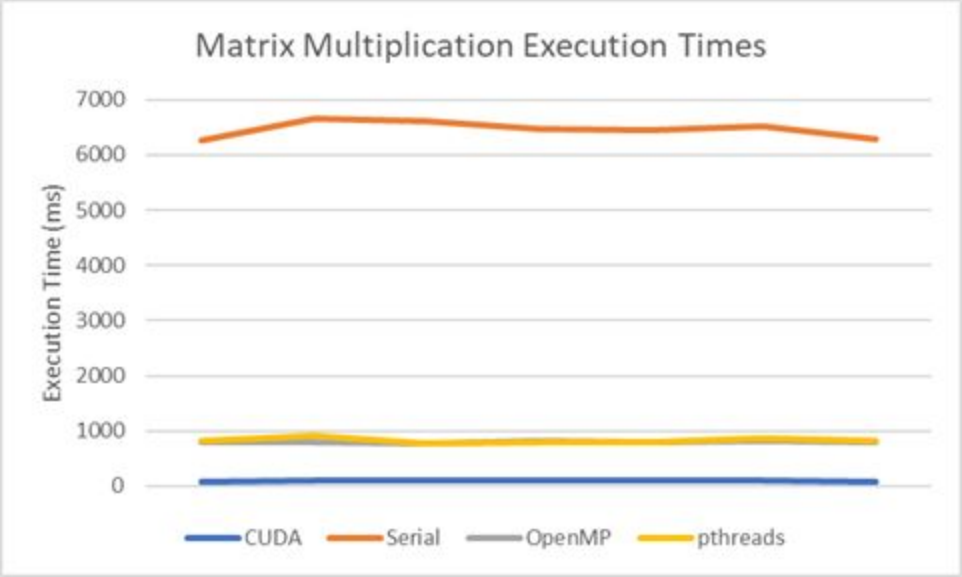
```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

The keyword gridDim.x provides the number of blocks in the grid. Another, blockIdx.x, contains the index of the current thread block in the grid. The block size is encapsulated in blockDim.x. The code below shows idiomatic CUDA in which each thread is getting

its index by calculating the offset to the start of its block and adding the index of the thread within the block.

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

III. Comparison



CUDA C executed matrix multiplication approximately 66 times faster than serial execution and approximately 8 times faster than both other implementations of OpenMP and POSIX Threads. Implementations of matrix multiplication of OpenMP and POSIX Threads did not have a sizable difference in run times comparatively, but

they were both approximately eight times faster than serial execution. Whereas OpenMP is an application programming interface for shared-memory multiprocessing programming that offers pragmas, directives, clauses, and work-sharing constructs and POSIX Threads offer a set of types, functions, and constants, CUDA C is an entire programming language.

IV. Discussion

The most difficult part of the project was fully conceptualizing the CUDA execution model, e.g., the mapping of thread blocks to Streaming Multiprocessors (SMs). CUDA has a higher learning curve than POSIX Threads or OpenMP. The performance of CUDA C is many times more efficient on average than OpenMP or POSIX Threads because it utilizes the GPU, a highly parallel and efficient multi-core system. The implementation of a parallel program using CUDA C is extremely difficult. It should not be added to the CS1645 curriculum because its exceptionally high learning curve would not only unnecessarily burden students but also demand enough time teaching it that other crucial facets of high-performance computing would not be properly edified.

V. Conclusions

While it might be difficult to learn and employ in everyday projects, CUDA C is undeniably powerful and rewarding. As a programming language, it is crafted to purposefully and intricately utilize parallelism more efficiently than either serial or parallel C programs can. The fine-tuned control and excellent performance provided by

CUDA C makes it a formidable choice when high performance is absolutely crucial. That being said, the ease of using OpenMP and POSIX Threads still might make them more preferable candidates for utilizing parallelism in many cases. CUDA C is more susceptible to logical errors because of its difficulty. CUDA C code also demands a greater amount of production time.

CUDA C is wonderful to learn if the primary goal is to familiarize oneself with parallel computing and the underlying concepts. It should be used when programming logic and abstractions are mostly unnecessary. Discussion of improvements to CUDA C is beyond the scope of this paper. Altogether, for better or worse, it is the epitome of high-performance computing.