GPU Programming with CUDA and Triton

Assignment 2

V S Vishnu

23B3909

Overview

In this week's assignment we were asked to implement the following CUDA kernels: **Vector Addition**, **Element-wise Multiply & Scale** and the **ReLU activation** function. We were also asked to experiment with various input sizes and block dimensions.

Task 1: Implementing Basic CUDA Kernels

As all three functions were largely similar and only differed majorly in the actual arithmetic operation being implemented at kernel launch, I will be taking one example (Vector Addition) to explain the building of such a basic CUDA kernel.

CUDA Kernel Definition

```
__global__ void vector_add(const float* a,
                           const float* b,
                           float* c,
                           int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

The `__global__` keyword is an indication that the function is a CUDA kernel executed on the GPU and launched from the host CPU. Each GPU thread computes a unique global index [`i`] using the block [`blockIdx.x`] and thread [`threadIdx.x`] identifiers. The bounds check `if (i < n)` ensures correctness when the total number of launched threads exceeds the input size. Each active thread performs exactly one element-wise addition.

Host Memory Allocation

```
float *h_a = new float[n];
float *h_b = new float[n];
float *h_c = new float[n];

for (int i = 0; i < n; i++) {
    h_a[i] = 1.0f;
    h_b[i] = 2.0f;
}
```

Host memory is dynamically allocated for both input and output data sets. The input vectors are initialized with known constant values to allow easy verification of correctness after kernel execution. I have chosen `1.0f` and `2.0f` here for convenience.

Device Memory Allocation

```
cudaMalloc(&d_a, size);
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);

cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
```

The required GPU global memory is allocated using `cudaMalloc`. The input data is then transferred from host memory (CPU) to device memory (GPU) using `cudaMemcpy` with the `cudaMemcpyHostToDevice` flag.

Grid & Block Configuration

```
int threadsPerBlock = 256;
int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
```

The number of threads per block is chosen as 256. The grid size is computed using ceiling division to ensure that at least one thread is launched for every input element.

Kernel Launch & Synchronization

```
vector_add<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);
cudaDeviceSynchronize();
```

The kernel is launched with the provided grid and block dimensions. Since kernel launches are asynchronous, `cudaDeviceSynchronize` is used to ensure completion before copying results back to the host. I actually initially faced of problem of getting wrong outputs, which I later found out was because I forgot to add this instruction in my initial code.

Memory Cleanup

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
delete[] h_a;
delete[] h_b;
delete[] h_c;
```

All allocated host and device memory is explicitly released to prevent memory leaks and ensure proper program termination.

Task 2: Grid/Block Design Exploration

Table 1: Grid and Block Configuration Analysis for Vector Addition

n	**blockDim.x**	**gridDim.x**	**Total Threads**	**Active Threads**	**Idle Threads**
10^3	32	32	1024	1000	24
	128	8	1024	1000	24
	256	4	1024	1000	24
	512	2	1024	1000	24
10^5	32	3125	100000	100000	0
	128	782	100096	100000	96
	256	391	100096	100000	96
	512	196	100352	100000	352
10^7	32	312500	10000000	10000000	0
	128	78125	10000000	10000000	0
	256	39063	10000128	10000000	128
	512	19532	10000384	10000000	384

Each active thread performs exactly one element-wise operation, while idle threads arise due to ceiling-based grid dimension computation. All configurations produce **correct** output due to bounds checking in the kernel.