

HPC Experiment 8

Aim: Accelerate Vector Addition Application

Theory:

At this point in time you have accomplished all of the following lab objectives:

1. Write, compile, and run C/C++ programs that both call CPU functions and launch GPU kernels.
2. Control parallel thread hierarchy using execution configuration.
3. Refactor serial loops to execute their iterations in parallel on a GPU.
4. Allocate and free memory available to both CPUs and GPUs.
5. Handle errors generated by CUDA code.

Now you will complete the final objective of the lab:

1. Accelerate CPU-only applications.

Lab Experiment to be performed:

The following challenge will give you an opportunity to use everything that you have learned thus far in the lab. It involves accelerating a CPU-only vector addition program, which, while not the most sophisticated program, will give you an opportunity to focus on what you have learned about GPU-accelerating an application with CUDA. After completing this exercise, if you have time and interest, continue on to the Advanced Content section for some challenges that involve more complex code bases.

01-vector-add.cu contains a functioning CPU-only vector addition application. Accelerate its `addVectorsInto` function to run as a CUDA kernel on the GPU and to do its work in parallel. Consider the following that need to occur, and refer to the solution if you get stuck.

1. Augment the `addVectorsInto` definition so that it is a CUDA kernel.
2. Choose and utilize a working execution configuration so that `addVectorsInto` launches as a CUDA kernel.
3. Update memory allocations, and memory freeing to reflect that the 3 vectors `a`, `b`, and `result` need to be accessed by host and device code.
4. Refactor the body of `addVectorsInto`: it will be launched inside of a single thread, and only needs to do one thread's worth of work on the input vectors. Be certain the thread will never try to access elements outside the range of the input vectors, and take care to note whether or not the thread needs to do work on more than one element of the input vectors.
5. Add error handling in locations where CUDA code might otherwise silently fail.

!nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005–2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0

pip install nvcc4jupyter

Collecting nvcc4jupyter
Downloading nvcc4jupyter-1.2.1-py3-none-any.whl.metadata (5.1 kB)
Downloading nvcc4jupyter-1.2.1-py3-none-any.whl (10 kB)
Installing collected packages: nvcc4jupyter
Successfully installed nvcc4jupyter-1.2.1
Note: you may need to restart the kernel to use updated packages.

```
[ ] !python --version
!nvcc --version
!pip install nvcc4jupyter
%load_ext nvcc4jupyter
```

Python 3.10.12
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005–2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
Requirement already satisfied: nvcc4jupyter in /usr/local/lib/python3.10/dist-packages (1.2.1)
Detected platform "Kaggle". Running its setup...
Updating the package lists...
Installing nvidia-cuda-toolkit, this may take a few minutes...
Source files will be saved in "/tmp/tmpjhwkwbx".

```
%%writefile vector_add.cu
#include <iostream>
#include <cuda_runtime.h>

// CUDA kernel for vector addition
__global__ void addVectorsInto(float *a, float *b, float *result, int N) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        result[idx] = a[idx] + b[idx];
    }
}

// Error checking macro
#define cudaCheckError(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line) {
    if (code != cudaSuccess) {
        std::cerr << "CUDA Error: " << cudaGetErrorString(code)
        << " at " << file << ":" << line << std::endl;
        exit(code);
    }
}

int main() {
    const int N = 1000;
    const int size = N * sizeof(float);

    float *a, *b, *result;

    // Allocate unified memory
    cudaCheckError(cudaMallocManaged(&a, size));
    cudaCheckError(cudaMallocManaged(&b, size));
    cudaCheckError(cudaMallocManaged(&result, size));

    // Initialize inputs
    for (int i = 0; i < N; ++i) {
        a[i] = static_cast<float>(i);
        b[i] = static_cast<float>(2 * i);
    }
}
```

```
[ ] // Choose execution configuration
    int threads_per_block = 256;
    int number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

    // Launch kernel
    addVectorsInto<<<number_of_blocks, threads_per_block>>>(a, b, result, N);

    // Wait for GPU to finish
    cudaCheckError(cudaDeviceSynchronize());


    // Display first 10 results
    for (int i = 0; i < 10; ++i) {
        std::cout << a[i] << " + " << b[i] << " = " << result[i] << std::endl;
    }

    // Free memory
    cudaCheckError(cudaFree(a));
    cudaCheckError(cudaFree(b));
    cudaCheckError(cudaFree(result));

    return 0;
}
```

 Overwriting vector_add.cu

```
[ ] !nvcc -o vector_add vector_add.cu
    !./vector_add
```

 0 + 0 = 0
 1 + 2 = 3
 2 + 4 = 6
 3 + 6 = 9
 4 + 8 = 12
 5 + 10 = 15
 6 + 12 = 18
 7 + 14 = 21
 8 + 16 = 24
 9 + 18 = 27

Conclusion:

In this experiment, we accelerated a CPU-only vector addition program by converting the `addVectorsInto` function into a CUDA kernel that executes in parallel on the GPU. We allocated unified memory for the input and output vectors to ensure accessibility from both the host and device, selected an appropriate execution configuration, and ensured each thread handled one element of the vectors without exceeding bounds. Error handling was added for robustness. This exercise demonstrated how to apply all key CUDA concepts learned so far to transform a serial CPU program into a parallel GPU-accelerated application.