



VECTOR-ADDITION EXPERIMENT ANALYSIS

Sequential on CPU - CUDA on GPU

Author
Xiu Zhou

1/31/2024

Contents

1. Introduction	2
1.1. CUDA Overview	2
1.2. Key CUDA APIs and concepts	2
1.3. GEMM on Market.....	4
2. Experiments Introduction	5
2.1. Experiment Running Environment	5
2.2. Experiments Configure Introduction.....	6
3. Part 1 - GPU Hardware Specifications	6
3.1. "Part1_getGPUInfor.cu" to Get GPU Specifications	6
3.2. Program Output Screenshot	7
4. Part 2 - Sequential CPU Implementation	9
4.1. Sequential CPU Program	9
4.2. Program Partial Output Screenshot (size = 8)	9
5. Part 3 - CUDA Vector Addition (Experiments 1 - 2 - 3).....	10
5.1. CUDA Program	10
5.1.1. vectorAdditionKernel Function	10
5.1.2. vectorAdditionCUDA Function	11
5.1.3. Implementation in main() Part.....	11
5.1.4. Experiment 1 Partial Output Screenshot (size = 8)	13
5.1.5. Experiment 2 Partial Output Screenshot (size = 1024)	13
5.1.6. Experiment 3 Partial Output Screenshot (size = 32)	13
6. Experiment 4 Program Implementation	14
7. Experiment 5 – GPU_CPU_Compare.....	14
7.1. Experiment 5 - GPU Partial Output Screenshot	15
7.2. Experiment 5 – CPU Partial Output Screenshot.....	15
8. Plots.....	15
8.1. Plots for Experiment 1 - 3D	15
8.2. Plots for Experiment 2 - 3D	17
8.3. Plots for Experiment 3 - 3D	18
8.4. Plots for Experiment 4 - 3D	19
8.5. Plot for Experiment 5 – GPU_CPU_Compare by 2D.....	21
9. Part 4 - Report Findings.....	22
10. Conclusion and Brief Analysis	23
10.1. Based on Finding of Experiment 1 - 2 - 3 - 4	23
10.2. Based on Finding of Experiment 5	24

1. Introduction

The CUDA "Hello World Vector" assignment aims to familiarize students with basic CUDA programming, focusing on vector addition. This report presents the hardware specifications, sequential CPU implementation, CUDA vector addition, and findings from the experiments.

1.1. CUDA Overview

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs (Graphics Processing Units). CUDA enables developers to use NVIDIA GPUs for accelerating computation beyond graphics processing.

1.2. Key CUDA APIs and concepts

1) CUDA Runtime API

The CUDA Runtime API provides a set of functions for managing devices, allocating memory on the GPU, launching kernels, and more. It's the high-level API used by most CUDA developers.

```
#include <cuda_runtime.h>
```

2) CUDA Device Management

Functions like `cudaGetDeviceCount`, `cudaSetDevice`, and `cudaGetDeviceProperties` allow us to query and manage GPU devices.

```
int deviceCount;  
  
cudaGetDeviceCount(&deviceCount);
```

3) CUDA Memory Management

CUDA provides functions like `cudaMalloc`, `cudaMemcpy`, and `cudaFree` for managing memory on the GPU.

```
int* d_a;  
  
cudaMalloc(&d_a, size);  
  
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
```

4) CUDA Kernels

Kernels are functions that execute in parallel on the GPU. You define them using the `__global__` qualifier.

Launching a kernel is done using a triple-chevron syntax `<<<...>>>`.

```
__global__ void myKernel(int* a, int* b, int* result) {  
  
    // Kernel code  
  
}  
  
myKernel<<<numBlocks, numThreads>>>(d_a, d_b, d_result);
```

5) Thread Hierarchy

CUDA organizes threads into blocks, and blocks into a grid. Threads within a block can communicate using shared memory. The overall organization is grid -> block -> thread.

```
dim3 gridSize(10, 10);  
  
dim3 blockSize(16, 16);  
  
myKernel<<<gridSize, blockSize>>>(d_a, d_b, d_result);
```

6) CUDA Streams

CUDA streams allow for concurrent execution of multiple kernels and memory copies. They enable overlapping computation and communication to improve performance.

```
cudaStream_t stream;  
  
cudaStreamCreate(&stream);  
  
myKernel<<<gridSize, blockSize, 0, stream>>>(d_a, d_b, d_result);
```

7) Asynchronous Execution

CUDA operations can be asynchronous, meaning that control is returned to the CPU before the GPU finishes its work. This allows for better overlap of computation and communication.

```
cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, stream);
```

8) CUDA Events

CUDA events can be used to measure the time taken by a kernel or other GPU operations. They provide a way to synchronize the CPU with GPU activities.

```
cudaEvent_t start, stop;  
  
cudaEventCreate(&start);
```

```
cudaEventRecord(start, stream);
```

9) Cooperative Groups

Introduced in CUDA 9, cooperative groups provide a higher-level abstraction for managing groups of threads with a shared execution context. They allow for more flexible synchronization.

```
#include <cooperative_groups.h>
```

```
namespace cg = cooperative_groups;
```

10) Unified Memory

CUDA Unified Memory simplifies memory management by allowing the GPU and CPU to access the same memory space seamlessly.

```
int* data;
```

```
cudaMallocManaged(&data, size);
```

11) NVIDIA Nsight and Visual Profiler

NVIDIA provides tools like Nsight and Visual Profiler to analyze and optimize CUDA applications. They offer insights into GPU utilization, memory usage, and performance bottlenecks.

1.3. GEMM on Market

Several optimized GEMM implementations exist in the market, and they are often part of high-performance numerical libraries. Some notable implementations include:

1) BLAS (Basic Linear Algebra Subprograms)

BLAS is a widely-used collection of routines for linear algebra operations, including GEMM.

Different implementations of BLAS exist, and they are optimized for various hardware architectures.

Examples include OpenBLAS, Intel MKL (Math Kernel Library), and ATLAS (Automatically Tuned Linear Algebra Software).

2) OpenBLAS

OpenBLAS is an open-source implementation of BLAS that aims to provide high-performance linear algebra routines.

It is designed to be architecture-agnostic and can take advantage of hardware-specific optimizations.

3) Intel MKL (Math Kernel Library)

Intel MKL is a library of optimized mathematical routines for Intel processors.

It includes highly optimized GEMM implementations that leverage Intel's advanced hardware features.

4) cuBLAS (CUDA Basic Linear Algebra Subprograms)

cuBLAS is part of the NVIDIA CUDA toolkit and provides BLAS routines optimized for NVIDIA GPUs.

It allows developers to offload GEMM computations to the GPU for parallel processing.

5) ARM Performance Libraries

ARM provides optimized libraries for various mathematical operations, including GEMM, targeted for ARM architecture-based processors.

6) OpenMP and MPI Parallel Implementations

Many GEMM implementations leverage parallelization techniques such as OpenMP (for shared-memory systems) or MPI (Message Passing Interface) for distributed memory systems.

7) High-Performance Computing (HPC) Environments

GEMM implementations are often optimized for specific HPC architectures, such as those used in supercomputers and cluster environments.

8) Custom Hardware Accelerators

Some applications use custom hardware accelerators, like FPGAs (Field-Programmable Gate Arrays) or TPUs (Tensor Processing Units), to perform GEMM operations efficiently.

2. Experiments Introduction

2.1. Experiment Running Environment

- GPU computer: All the experiments about this report run on UW GPU Lab Computer, conducting remotely.
- Network: All the experiments about this report run at the same network, no locations change.

2.2. Experiments Configure Introduction

# Experiment	Configure Description (vector size unit (2 ⁿ))	Program Conducting Experiment
1	Experiment 1: Varying #Threads, #block = 1, increase Threads threads = {8, 64, 128, 512}; block = 1; vector size = {3,6,7,9}; outputFile: "experiment_thread_op_11.txt", "experiment_thread_m_12.txt";	Experiment 1, 2, 3 will be conducted in program: "Part3_CUDA_matrix_add1.cu".
2	Experiment 2: Varying #block, #Threads 1024, increase block threads = 1024 block = {1, 8,64,128}; vector size = {10,13,16,17}; outputFile: "experiment_block_op_21.txt", "experiment_block_m_22.txt";	
3	Experiment 3: Varying #block and #Threads with the same number threads = {1, 2, 4, 64}; block = {1, 2, 4, 64}; vector size = {5,8,10,15} outputFile: "experiment_thread_block_op_31.txt", "experiment_thread_block_m_32.txt";	
4	Experiment 4: Varying #block, #Threads 1024, descend blocks threads = 1024; block = {1024, 512,256,128, 32, 8, 1}; vector size = {4, 6,7, 9, 10, 11,12, 15,18,20}; outputFile: "experiment_block_op_41.txt", "experiment_block_m_42.txt";	Experiment 4 will be conducted in program: "Part3_CUDA_matrix_add_block.cu".
5	1) GPU: The numbers of blocks and thread is 1024, size changes from 2 ³ to 2 ²⁰ . threads = 1024; block = 1024; It will be implemented in program "compare_GPU_CPU.cu". Program generates 2 files, one is including memory time, the other is not. 2) CPU: Size changes form 2 ³ to 2 ²⁰ . It will be implemented in program "Part2_CPU_sequen_matrix_add.cpp". Program generates 1 file recording vector addition time consuming.	Experiment 5 – CPU part: It will be conducted in program "Part2_CPU_sequen_matrix_add.cpp". Experiment 5 – GPU part: It will be conducted in program "compare_GPU_CPU.cu".

3. Part 1 - GPU Hardware Specifications

This section provides detailed information about the GPU hardware used in the experiments. I use the CUDA APIs to gather specifications programmatically.

3.1. "Part1_getGPUInfor.cu" to Get GPU Specifications

In the Fig 1, there are two key functions in "Part1_getGPUInfor.cu" program.

cudaGetDeviceCount(&deviceCount) is to get the number of available CUDA devices.

cudaGetDeviceProperties(&prop, i) is to get the properties of the current CUDA device.

```

int deviceCount;

// Get the number of available CUDA devices
cudaGetDeviceCount(&deviceCount);

// Loop through each CUDA device
for (int i = 0; i < deviceCount; ++i) {
    cudaDeviceProp prop;

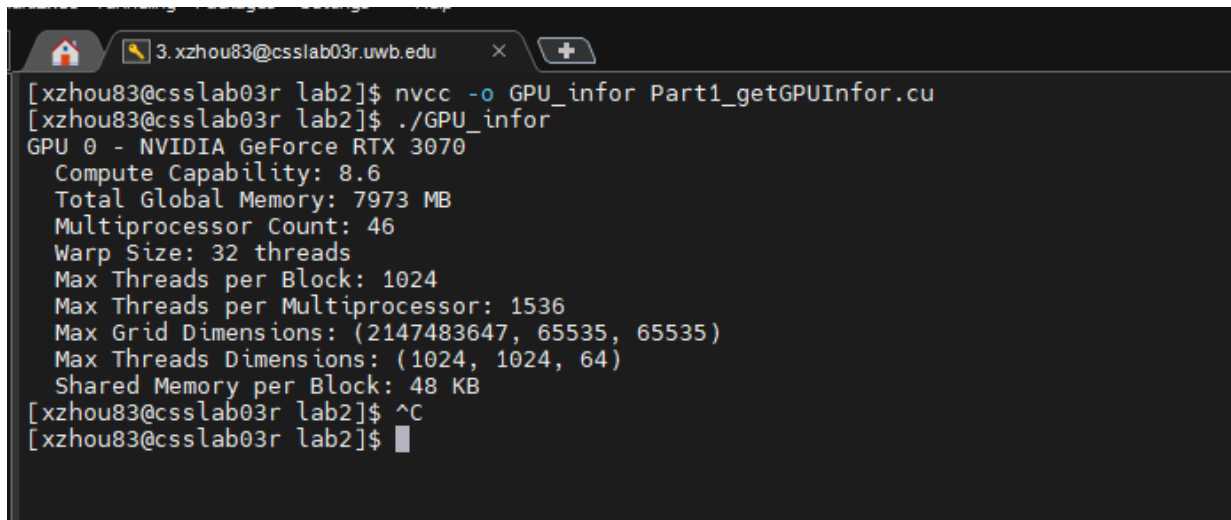
    // Get the properties of the current CUDA device
    cudaGetDeviceProperties(&prop, i);

    // Print information about the current CUDA device
    std::cout << "GPU " << i << " - " << prop.name << std::endl;
    std::cout << " Compute Capability: " << prop.major << "." << prop.minor << std::endl;
    std::cout << " Total Global Memory: " << prop.totalGlobalMem / (1024 * 1024) << " MB" << std::endl;
    std::cout << " Multiprocessor Count: " << prop.multiProcessorCount << std::endl;
    std::cout << " Warp Size: " << prop.warpSize << " threads" << std::endl;
    std::cout << " Max Threads per Block: " << prop.maxThreadsPerBlock << std::endl;
    std::cout << " Max Threads per Multiprocessor: " << prop.maxThreadsPerMultiProcessor << std::endl;
    std::cout << " Max Grid Dimensions: (" << prop.maxGridSize[0] << ", " << prop.maxGridSize[1] << ", " << prop.maxGridSize[2] << ")" << std::endl;
    std::cout << " Max Threads Dimensions: (" << prop.maxThreadsDim[0] << ", " << prop.maxThreadsDim[1] << ", " << prop.maxThreadsDim[2] << ")" << std::endl;
}

```

Fig 1, Key code module of "Part1_getGPUInfor.cu"

3.2. Program Output Screenshot



```

[xzhou83@csslab03r lab2]$ nvcc -o GPU_infor Part1_getGPUInfor.cu
[xzhou83@csslab03r lab2]$ ./GPU_infor
GPU 0 - NVIDIA GeForce RTX 3070
Compute Capability: 8.6
Total Global Memory: 7973 MB
Multiprocessor Count: 46
Warp Size: 32 threads
Max Threads per Block: 1024
Max Threads per Multiprocessor: 1536
Max Grid Dimensions: (2147483647, 65535, 65535)
Max Threads Dimensions: (1024, 1024, 64)
Shared Memory per Block: 48 KB
[xzhou83@csslab03r lab2]$ ^C
[xzhou83@csslab03r lab2]$

```

Fig 2, output screenshot of "Part1_getGPUInfor.cu"

In the Fig 2, there are several key specifications of GPU.

1) According GPU 0 - NVIDIA GeForce RTX 3070

This indicates that there is one GPU in the computer, and it is an NVIDIA GeForce RTX 3070. The "0" after "GPU" denotes the device index, which is 0-indexed. If the computer has multiple GPUs, you might see "GPU 0," "GPU 1," and so on.

2) Compute Capability: 8.6

The compute capability is a version number associated with the CUDA architecture of the GPU. It consists of two parts: major and minor. In this case, the major version is 8, and the minor version is 6. Compute capability 8.x GPUs are part of the NVIDIA Ampere architecture. The higher the compute capability, the more advanced features and capabilities the GPU supports.

3) Total Global Memory: 7973 MB

This indicates the total amount of global memory available on the GPU. Global memory is the primary memory space accessible to all threads in a CUDA kernel. Here, the GPU has 7973 megabytes (MB) of global memory. This memory is used for storing variables, data, and other resources during GPU computations.

4) Multiprocessor Count (46):

This indicates the number of multiprocessors on the GPU. Multiprocessors are individual processing units within the GPU that handle parallel execution of CUDA threads. A higher multiprocessor count generally leads to better parallel processing capabilities.

5) Warp Size (32 threads):

The warp is the basic unit of execution in a CUDA GPU. It consists of a fixed number of threads that are executed in lockstep. The warp size specifies the number of threads in a warp. In this case, the warp size is 32 threads, meaning that 32 threads are processed simultaneously.

6) Max Threads per Block (1024):

This indicates the maximum number of threads that can be accommodated within a single thread block. A thread block is a group of threads that can cooperate and synchronize. The limit of 1024 threads per block sets an upper boundary on the parallelism achievable within a block.

7) Max Threads per Multiprocessor (1536):

Specifies the maximum number of threads that can be executed concurrently on a single multiprocessor. Understanding this limit is crucial for optimizing parallelism and achieving efficient GPU utilization.

8) Max Grid Dimensions (2147483647, 65535, 65535):

Indicates the maximum dimensions that can be used when configuring a grid of thread blocks. The grid is the overall set of thread blocks. Here, the maximum grid dimensions are extremely large, allowing for flexibility in organizing parallel computations.

9) Max Threads Dimensions (1024, 1024, 64):

Specifies the maximum dimensions that can be used when configuring a thread block. It indicates the maximum number of threads along each dimension within a block.

10) Shared Memory per Block (48 KB):

Indicates the amount of shared memory available per thread block. Shared memory is a fast, on-chip memory that can be shared among threads within a block. Managing shared memory effectively is crucial for optimizing data sharing and communication among threads.

4. Part 2 - Sequential CPU Implementation

4.1. Sequential CPU Program

I implemented a sequential CPU program “Part2_CPU_sequen_matrix_add.cpp” to do vector addition in C++. The vectors were generated with random integers between -10 and 10 using generateRandomVector() function (see Fig 3). Timing measurements of vectorAddition() function (see Fig 3) were stored into file ("vector_sequen_addition_results.txt") to analyze the performance. Fig 4 provides the timing module. The vectorAddition() function takes charge of sequential vector addition.

```
21 // Function to generate vector with random integer between -10 and 10
22 void generateRandomVector(std::vector<int>& vec, size_t size) {
23     vec.resize(size);
24     for (size_t i = 0; i < size; ++i) {
25         // Generate random integer between -10 and 10
26         vec[i] = rand() % 21 - 10;
27     }
28 }
29
30 // Function to add two vector a and b, and store into result vector
31 void vectorAddition(const std::vector<int>& a, const std::vector<int>& b, std::vector<int>& result) {
32     for (size_t i = 0; i < a.size(); ++i) {
33         result[i] = a[i] + b[i];
34     }
35 }
```

Fig 3, generateRandomVector() and vectorAddition() functions

```
81 // Measure time
82 clock_t start = clock();
83
84 // Perform vector addition
85 vectorAddition(vectorA, vectorB, resultVector);
86
87 clock_t end = clock();
88
89 // Calculate time elapsed
90 double elapsedSeconds = static_cast<double>(end - start) / CLOCKS_PER_SEC;
91
92 // Write vectorSize, elapsedSeconds to the file
93 outputFile << vectorSize << "\t" << elapsedSeconds << std::endl;
94
```

Fig 4, timing vectorAddition() module

4.2. Program Partial Output Screenshot (size = 8)

Fig 5 shows vector addition results when size = 8, which proves the addition operation result is right.

```

[xzhou83@csslab03r lab2]$ g++ -o sequenAdd Part2_CPU_sequen_matrix_add.cpp
[xzhou83@csslab03r lab2]$ ./sequenAdd
Vector A Before experiments:
{
    4    3   -4   -4   -7   -7    7   -1
}
Vector B Before experiments:
{
    1    5    8   -7    2   -10   -7    3
}
resultVector Before experiments:
{
    0    0    0    0    0    0    0    0
}
Vector A after all the experiments:
{
    4    3   -4   -4   -7   -7    7   -1
}
Vector B after all the experiments:
{
    1    5    8   -7    2   -10   -7    3
}
resultVector after all the experiments:
{
    5    8    4   -11   -5   -17    0    2
}
[xzhou83@csslab03r lab2]$

```

Fig 5, output screenshot of "Part2_CPU_sequen_matrix_add.cpp"

5. Part 3 - CUDA Vector Addition (Experiments 1 - 2 - 3)

5.1. CUDA Program

The CUDA implementation included a kernel for vector addition `vectorAdditionKernel()` (see Fig 6). The vectors were generated similarly to the sequential CPU version(). Memory allocation, kernel launch, and data transfer were performed on the GPU. Timing measurements were collected, including inclusive and exclusive times.

Here are some key functions and modules.

5.1.1. vectorAdditionKernel Function

```

45 // Function to realize a kernel for vector addition
46 __global__ void vectorAdditionKernel(const int* a, const int* b, int* result, size_t size) {
47     size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
48
49     if (idx < size) {
50         result[idx] = a[idx] + b[idx];
51     }
52 }
53 }

```

Fig 6, vectorAdditionKernel() function

Function Explanation:

- `__global__ void vectorAdditionKernel(const int* a, const int* b, int* result, size_t size):` It is the kernel function for 1D vector addition. It takes two input vectors (a and b), an output vector (result), and the size of the vectors (size).
- `size_t idx = blockIdx.x * blockDim.x + threadIdx.x;` This line calculates the unique index of the current thread in the 1D grid. It takes into account both the block index (blockIdx.x) and the thread index within the block (threadIdx.x).
- `if (idx < size) { result[idx] = a[idx] + b[idx]; }` This condition ensures that the thread performs the addition only if its index is within the valid range of vector elements. Each thread adds the corresponding elements of vectors a and b and stores the result in the result vector.

5.1.2. vectorAdditionCUDA Function

```

70 // Function to perform 1D vector addition using CUDA and measure time
71 float vectorAdditionCUDA(const std::vector<int>& vectorA, const std::vector<int>& vectorB, std::vector<int>& resultVector,
72                          size_t numBlocks, size_t numThreadsPerBlock) {
73     size_t vectorSize = vectorA.size();
74     // Allocate memory on the device
75     int* d_a, *d_b, *d_result;
76     cudaMalloc(&d_a, vectorSize * sizeof(int));
77     cudaMalloc(&d_b, vectorSize * sizeof(int));
78     cudaMalloc(&d_result, vectorSize * sizeof(int));
79
80     // Copy input vectors to device
81     cudaMemcpy(d_a, vectorA.data(), vectorSize * sizeof(int), cudaMemcpyHostToDevice);
82     cudaMemcpy(d_b, vectorB.data(), vectorSize * sizeof(int), cudaMemcpyHostToDevice);
83     // Configure execution parameters
84     dim3 blockSize(numThreadsPerBlock);
85     dim3 gridSize(numBlocks);
86
87     // Start measuring time
88     cudaEvent_t start, stop;
89     cudaEventCreate(&start);
90     cudaEventCreate(&stop);
91     cudaEventRecord(start);
92     // Launch kernel
93     vectorAdditionKernel<<<gridSize, blockSize>>>(d_a, d_b, d_result, vectorSize);
94
95     // Stop measuring time
96     cudaEventRecord(stop);
97     cudaEventSynchronize(stop);
98     // Calculate elapsed time
99     float milliseconds = 0;
100     auto elapsedTime = cudaEventElapsedTime(&milliseconds, start, stop);
101     // Copy result back to host
102     cudaMemcpy(resultVector.data(), d_result, vectorSize * sizeof(int), cudaMemcpyDeviceToHost);
103
104     // Free device memory
105     cudaFree(d_a);
106     cudaFree(d_b);
107     cudaFree(d_result);
108
109     // Return elapsed time in milliseconds
110     return milliseconds;
111 }

```

Fig 7, vectorAdditionCUDA() function

Function Explanation:

- The vectorAdditionCUDA function manages the entire CUDA process for 1D vector addition.
- It allocates device memory, copies data from host to device, launches the kernel (vectorAdditionKernel), records execution time, and copies the result back to the host.
- `dim3 blockSize(numThreadsPerBlock);` and `dim3 gridSize(numBlocks);` define the CUDA grid and block dimensions based on the input parameters.
- `cudaEvent_t start, stop;` are used to measure the execution time.
- The elapsed time is calculated and returned as a float. It is just the adding operation consuming time.

5.1.3. Implementation in main() Part

```

156 // Experiment 1: Varying #Threads, #Block 1
157 for (size_t i = 0; i < numThreadsList1.size(); i++) {
158
159     // Get the threads number per block
160     size_t numThreads = numThreadsList1[i];
161
162     // Record the start time including memory and operation
163     auto startTime1 = std::chrono::high_resolution_clock::now();
164
165     // Calculate CUDA vector addition and return the adding operation time
166     float elapsedTime1 = vectorAdditionCUDA(vectorA, vectorB, resultVector, 1, numThreads);
167
168     // Record the end time just including memory
169     auto endTime1 = std::chrono::high_resolution_clock::now();
170
171     // Calculate CUDA vector addition time including memory and adding operation
172     std::chrono::duration<double, std::milli> elapsedMilliseconds1 = endTime1 - startTime1;
173
174     // Output the CUDA time just including adding operation time into file
175     outputFile11 << vectorSize << "\t" << numThreads << "\t1\t" << elapsedTime1 << "\n";
176
177     // Output the CUDA time including adding operation and memory time into file
178     outputFile12 << vectorSize << "\t" << numThreads << "\t1\t" << elapsedMilliseconds1.count() << "\n";
179 }
180

```

Fig 8, Experiment 1 module in main() part

Explanation:

The main part is conducting three different experiments to analyze the performance of CUDA vector addition by varying the number of threads and blocks.

1) Experiment 1: Varying #Threads, #Block 1

- Iterate through the list of thread numbers (numThreadsList1).
- For each thread number, record the start time.
- Perform CUDA vector addition (vectorAdditionCUDA) with 1 block and the current thread number.
- Record the end time.
- Output the elapsed time for just the addition operation (elapsedTime1) and the total elapsed time including memory and addition operation into separate files (outputFile11 and outputFile12).

2) Experiment 2: Varying #Block, #Threads 1024

- Iterate through the list of block numbers (numBlocksList2).
- For each block number, record the start time.
- Perform CUDA vector addition with 1024 threads per block and the current block number.
- Record the end time.
- Output the elapsed time for just the addition operation (elapsedTime2) and the total elapsed time including memory and addition operation into separate files (outputFile21 and outputFile22).

3) Experiment 3: Varying #Block and #Threads with the same number

- Iterate through the list of thread and block numbers (numThreadsList3 and numBlocksList3).
- For each combination of thread and block numbers, record the start time.
- Perform CUDA vector addition with the current thread and block numbers.
- Record the end time.
- Output the elapsed time for just the addition operation (elapsedTime3) and the total elapsed time including memory and addition operation into separate files (outputFile31 and outputFile32).

4) Vector Initialization and Output Part Result for verifying result correction

- For each experiment, initialize vectors (vectorA, vectorB, and resultVector) of the specified size.
- If the vector size is 8, print the vectors before kernel execution.
- After all experiments, if the vector size is 8, print the vectors after all experiments.

5) File Closing

- Close all the open output files.

5.1.4. Experiment 1 Partial Output Screenshot (size = 8)

```
[xzhou83@csslab03r lab2]$ nvcc -o addition1 Part3_CUDA_matrix_add1.cu
[xzhou83@csslab03r lab2]$ ./addition1
Vector A Before experiment 1, size = 8:
{
    6    -6    -5    -9    -1    1    -9    3
}
Vector B Before experiment 1, size = 8:
{
    -9    6    10    -8    5    -8    -2    -10
}
resultVector Before experiment 1, size = 8:
{
    0    0    0    0    0    0    0    0
}

Vector A after experiment, size = 8:
{
    6    -6    -5    -9    -1    1    -9    3
}
Vector B after experiment, size = 8:
{
    -9    6    10    -8    5    -8    -2    -10
}
resultVector after experiment, size = 8:
{
    -3    0    5    -17    4    -7    -11    -7
}
```

Fig 9, output screenshot of Experiment 1

5.1.5. Experiment 2 Partial Output Screenshot (size = 1024)

```
10    5    -5    -7    -2    1    8    -1    8    9    -3    8    6
0    -9    6    3    -10    -8    7    -10    1    8    -9    0    -9
3    -7    -9    2
}
resultVector after experiment 2, size = 1024:
{
    -9    -13    11    -3    -11    5    -2    -5    -5    1    -4    -5
    -4    1    -3    -11    -3    -5    -5    -16    8    9    -15    6    -1
    -6    4    15    -3    -10    2    5    -13    11    -12    -18    3    20
    18    -7    0    9    -8    -7    16    7    8    9    10    -6    -6
    4    18    -10    1    9    11    -7    3    1    -10    5    4    -2
    -14    -1    8    4    -9    -2    3    -1    -10    17    -9    -8    1
    3    2    13    7    11    7    -7    8    11    3    -4    11    2
    14    -6    4    -19    2    11    -4    5    -9    -14    3    13    5
    11    5    -6    -1    3    7    -3    -8    12    -16    17    -16    9
    -14    5    -7    7    10    -16    -12    11    3    15    6    7    -18
    2    -13    15    -7    -4    -6    5    12    -2    -11    7    8    0
    3    0    -3    -15    -13    0    -2    10    -14    0    -5    -6    -18
    -3    6    -12    -13    -8    0    -2    5    -9    0    -6    -14    -14
    2    -7    4    4    16    3    -3    17    6    15    -7    -9    -2
    10    6    9    -10    -12    4    16    -5    9    5    -10    -17    -14
    2    -2    7    6    -7    -5    9    16    6    9    -8    0    -7
    -9    3    0    -1    5    8    -13    -7    -12    -2    4    16    2
    6    -6    5    10    -5    0    -7    -2    -7    -14    4    -14    -9
    14    -4    0    3    -3    4    9    -13    5    -15    3    3    6
}
```

Fig 10, output screenshot of Experiment 2

5.1.6. Experiment 3 Partial Output Screenshot (size = 32)

```

Vector A Before experiment 3, size = 32:
{
    8      3     -5     -10    -4      2     -3      4      0      4      7      10
4     -4      9      4      -7     -9     -5      3     -7     -10     3      6      7
9     -9     10     -4     -6      7
}
Vector B Before experiment 3, size = 32:
{
    7     -3      7     -8      7     -9      6     -4      4      1     -7     10
-10     2      8     -9      5     -2     -6      4     -10     8      0      4      7
3     -10     4     -3      8
}
resultVector Before experiment 3, size = 32:
{
    0      0      0      0      0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0      0      0      0      0
0      0      0      0      0
}

Vector A after experiment 3, size = 32:
{
    8      3     -5     -10    -4      2     -3      4      0      4      7      10
4     -4      9      4      -7     -9     -5      3     -7     -10     3      6      7
9     -9     10     -4     -6      7
}
Vector B after experiment 3, size = 32:
{
    7     -3      7     -8      7     -9      6     -4      4      1     -7     10
-10     2      8     -9      5     -2     -6      4     -10     8      0      4      7
3     -10     4     -3      8
}
resultVector after experiment 3, size = 32:
{
    15      0      2     -18      3      -7      3      0      4      5      0      20
-14     11     12     -16     -4      -7      -3     -3     -20     11      6      11     12
-6      0      0     -9      15
}

```

Fig 11, output screenshot of Experiment 3

6. Experiment 4 Program Implementation

All the implementation is as same as “Part3_CUDA_matrix_add1.cu” except the vector size and the block number setting in the test cases. The Thread is 1024, the vector size and the block will change.

```

Vector A Before experiments 4, size = 16:
{
    9      9      3     -5     -8      10     -7     -10     5      3     -3      10     -
4     -6      7      2
}
Vector B Before experiments 4, size = 16:
{
    -5      9      2      8      2      8     -5     -9     -7      2      1      10     -
8      1      4      9
}
resultVector Before experiments 4, size = 16:
{
    0      0      0      0      0      0      0      0      0      0      0      0      0
0      0      0
}

Vector A experiments 4, size = 16: :
{
    9      9      3     -5     -8      10     -7     -10     5      3     -3      10     -
4     -6      7      2
}
Vector B experiments 4, size = 16: :
{
    -5      9      2      8      2      8     -5     -9     -7      2      1      10     -
8      1      4      9
}
resultVector experiments 4, size = 16:
{
    4      18      5      3     -6      18     -12     -19     -2      5     -2      20     -
12     -5      11      11
}

```

Fig 12, output screenshot of “Part3_CUDA_matrix_add_block.cu”

7. Experiment 5 – GPU_CPU_Compare

CPU part of this experiment will be implemented in program “Part2_CPU_sequen_matrix_add.cpp”, which calculate the sequential CPU vector-addition consuming time.

GPU part of this experiment will be implemented in program “compare_GPU_CPU.cu”, which calculate the GPU vector-addition consuming time, including with memory or not. The thread number is 1024, the block number is 1024.

7.1. Experiment 5 - GPU Partial Output Screenshot

```
[xzhou83@csslab03r lab2]$ nvcc -o gpu_cuda compare_GPU_CPU.cu
[xzhou83@csslab03r lab2]$ ./gpu_cuda
Vector A Before experiments 4, size = 16:
{
    -7    -1    8    -6    -7    -7    2    9    -6    -9    5    7    0
9    3    1
}
Vector B Before experiments 4, size = 16:
{
    1    -5    1    6    -3    2    -8    8    8    -10    -6    -8    -
9    6    10    -6
}
resultVector Before experiments 4, size = 16:
{
    0    0    0    0    0    0    0    0    0    0    0    0    0
0    0    0
}

Vector A experiments 4, size = 16: :
{
    -7    -1    8    -6    -7    -7    2    9    -6    -9    5    7    0
9    3    1
}
Vector B experiments 4, size = 16: :
{
    1    -5    1    6    -3    2    -8    8    8    -10    -6    -8    -
9    6    10    -6
}
resultVector experiments 4, size = 16:
{
    -6    -6    9    0    -10    -5    -6    17    2    -19    -1    -1    -
9    15    13    -5
}
```

Fig 13, output screenshot of Experiment 4 by “Part3_CUDA_matrix_add_block.cu”

7.2. Experiment 5 – CPU Partial Output Screenshot

See Fig 5.

8. Plots

8.1. Plots for Experiment 1 - 3D

Experiment 1: Time with Thread Changing With Memory (μs)				
Thread #	Size #			
	8	64	128	512
8	630806	95.861	99.504	130.933
64	106.912	108.83	105.213	105.126
128	110.388	106.666	106.512	105.353
512	131.37	109.298	104.439	107.981

Fig 14, raw data of Experiment 1 with memory

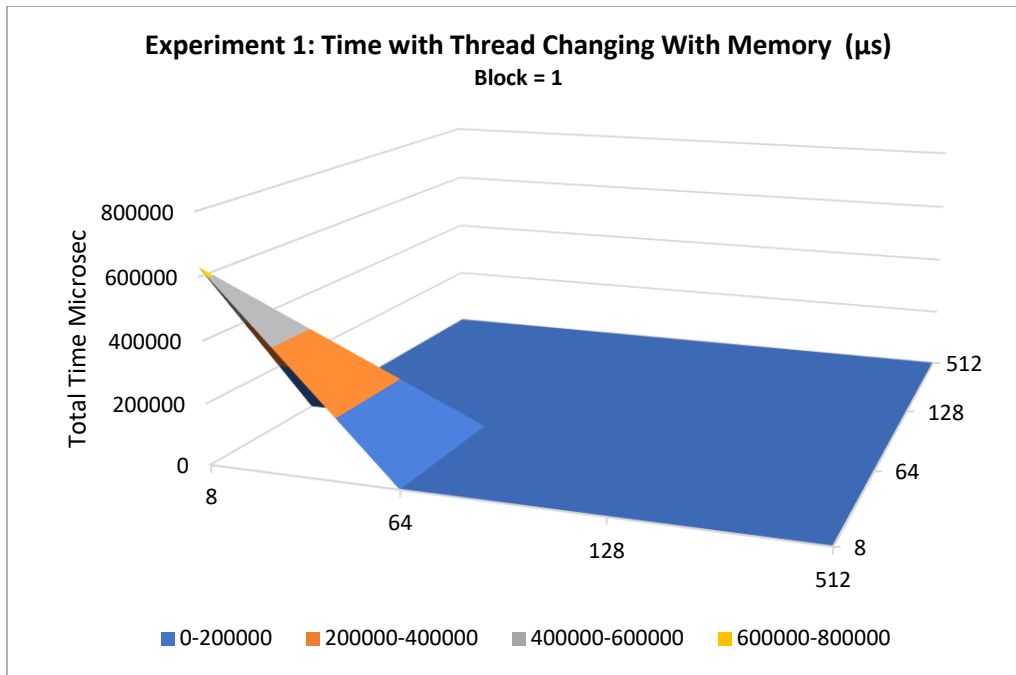


Fig 15, plot of Experiment 1 with memory

Experiment 1: Time with Thread Changing Without Memory (μ s)				
Thread #	Size #			
	8	64	128	512
8	16501.8	5.12	4.96	4.096
64	4.96	4.064	5.12	4.096
128	5.024	5.12	4.192	5.056
512	5.12	4.8	5.12	4.832

Fig 16, raw data of Experiment 1 without memory

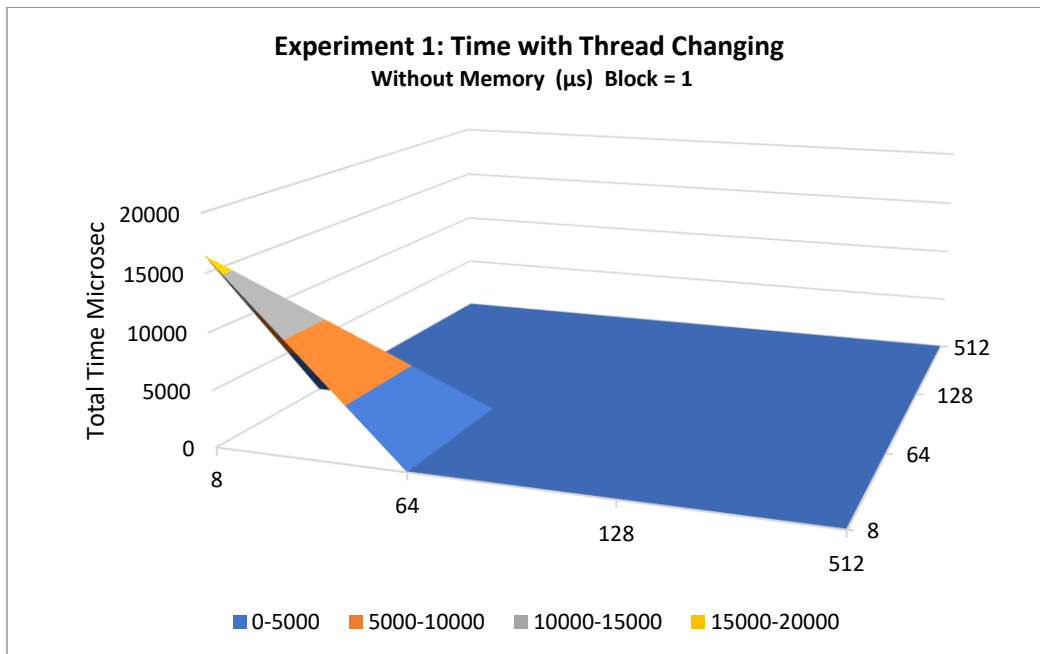


Fig 17, plot of Experiment 1 without memory

8.2. Plots for Experiment 2 - 3D

Experiment 2: Time with Block Changing With Memory (μ s)				
Block #	Size #			
	1024	8192	65536	131072
1	110.76	102.742	231.659	347.212
8	106.947	116.754	205.425	300.938
64	105.934	116.161	204.758	300.169
128	108.727	117.095	202.887	335.478

Fig 18, raw data of Experiment 2 with memory

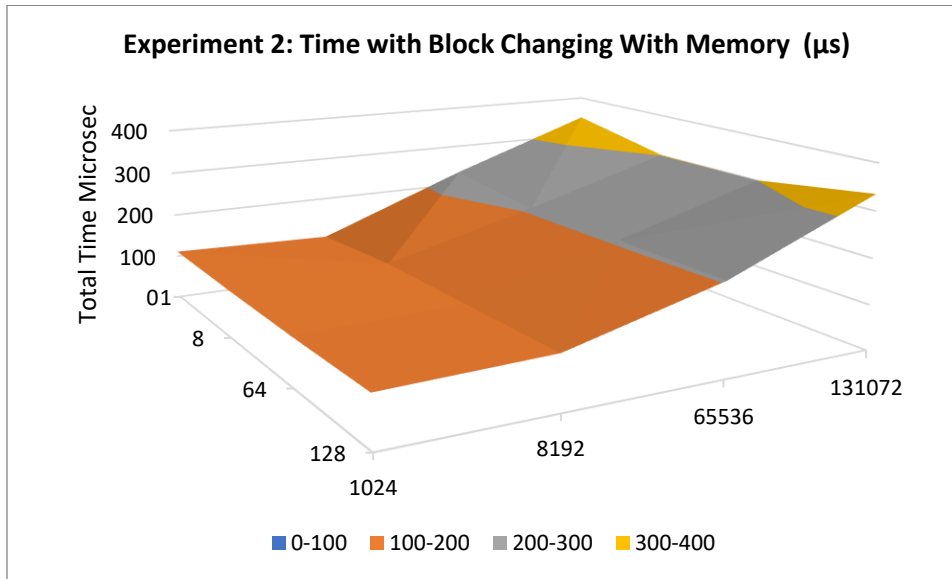


Fig 19, plot of Experiment 2 with memory

Experiment 2: Time with Block Changing Without Memory (μ s)				
Block #	Size #			
	1024	8192	65536	131072
1	5.088	4.096	6.752	10.944
8	5.12	3.072	6.144	11.456
64	4.096	2.048	8.352	12
128	4.896	3.072	7.168	15.104

Fig 20, raw data of Experiment 2 without memory

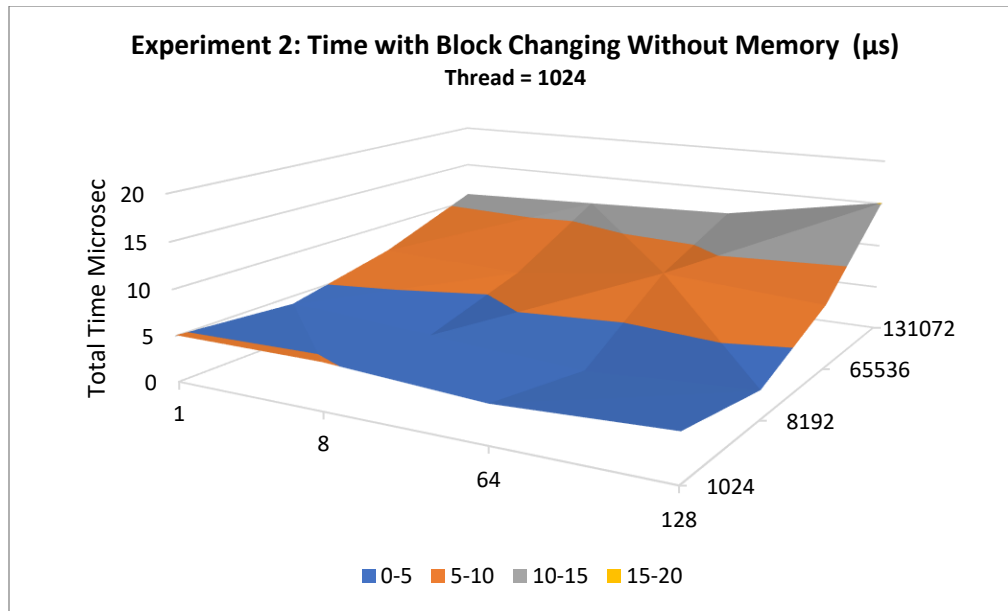


Fig 21, plot of Experiment 2 without memory

8.3. Plots for Experiment 3 - 3D

Experiment 3: Time with Thread and Block Changing With Memory (μ s)				
Thread/Block #	Size #			
	32	256	1024	32768
1	94.729	88.138	88.548	141.765
2	107.26	104.656	104.68	157.003
4	105.339	135.313	104.947	193.39
64	106.321	104.494	105.989	156.255

Fig 22, raw data of Experiment 3 with memory

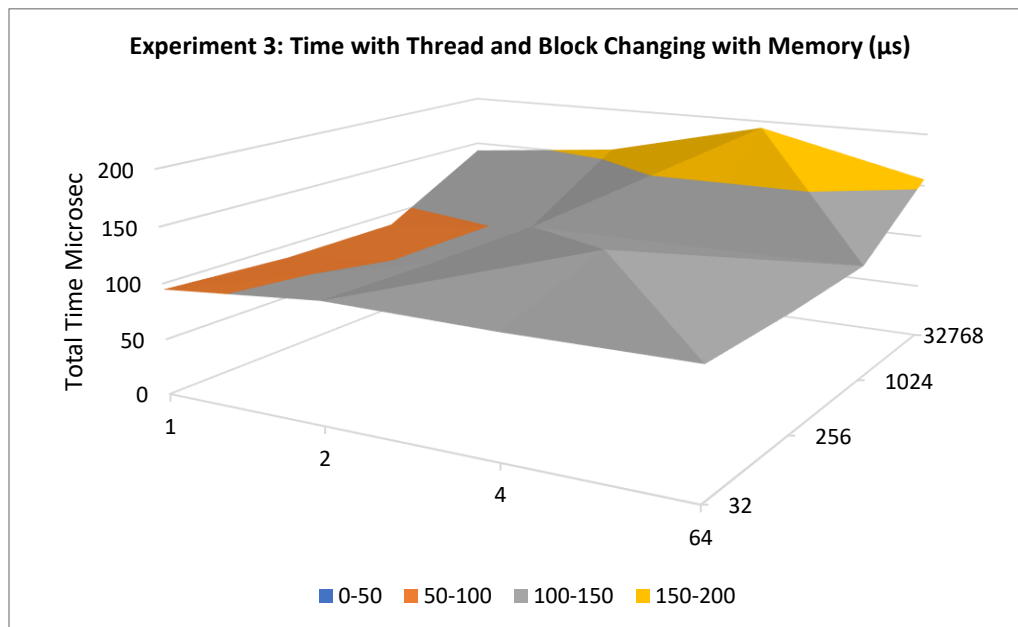


Fig 23, plot of Experiment 3 with memory

Experiment 3: Time with Thread and Block Changing Without Memory (μ s)				
Thread/Block #	Size #			
	32	256	1024	32768
1	5.12	5.12	4.768	7.584
2	4.096	4.896	4.736	8.288
4	4.544	4.992	4.096	8.16
64	5.12	4.992	4.096	7.872

Fig 24, raw data of Experiment 3 without memory”

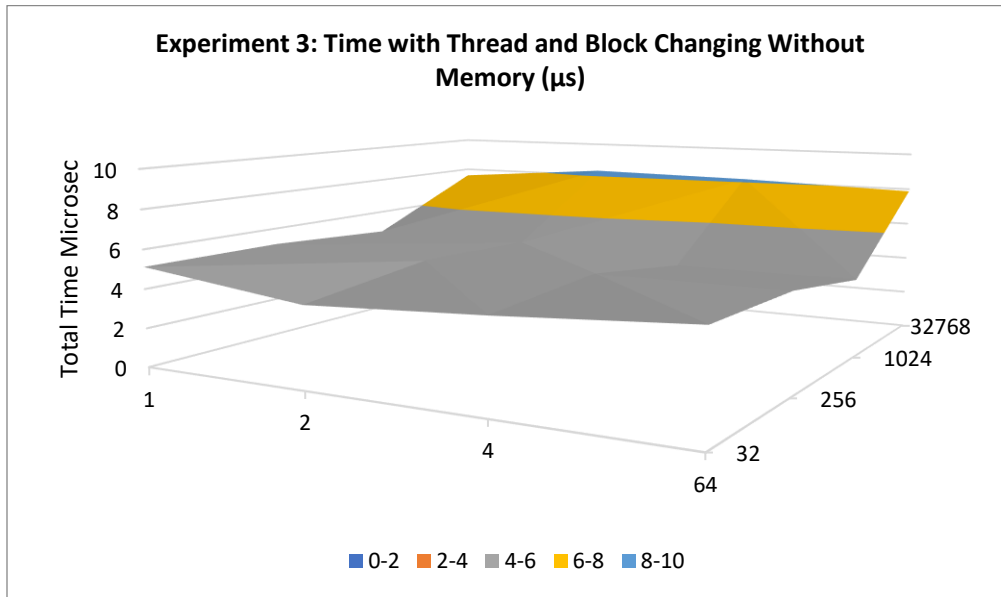


Fig 25, plot of Experiment 3 without memory

8.4. Plots for Experiment 4 - 3D

Experiment 4: Time with Block Changing with Memory (μ s)										
Block #	Size #									
	16	64	128	512	1024	2048	4096	32768	262144	1048576
1024	631682	95.271	103.418	105.217	89.772	92.249	95.517	148.083	633.493	1664.35
512	104.52	109.89	108.534	106.042	143.645	108.748	110.435	159.671	583.623	1624.74
256	110.687	106.508	104.177	106.645	100.545	106.414	109.205	157.719	575.287	1587.9
128	130.029	104.471	102.988	103.552	106.799	120.265	109.513	194.217	581.621	1583.42
32	107.932	110.702	105.653	103.091	104.507	106.681	125.775	155.752	569.492	1591.02
8	105.433	124.135	103.71	105.363	103.998	108.187	109.056	155.118	571.111	1576.33
1	106.838	104.793	102.123	103.596	105.428	105.749	109.687	153.52	568.795	1584.43

Fig 26, raw data of Experiment 4 with memory

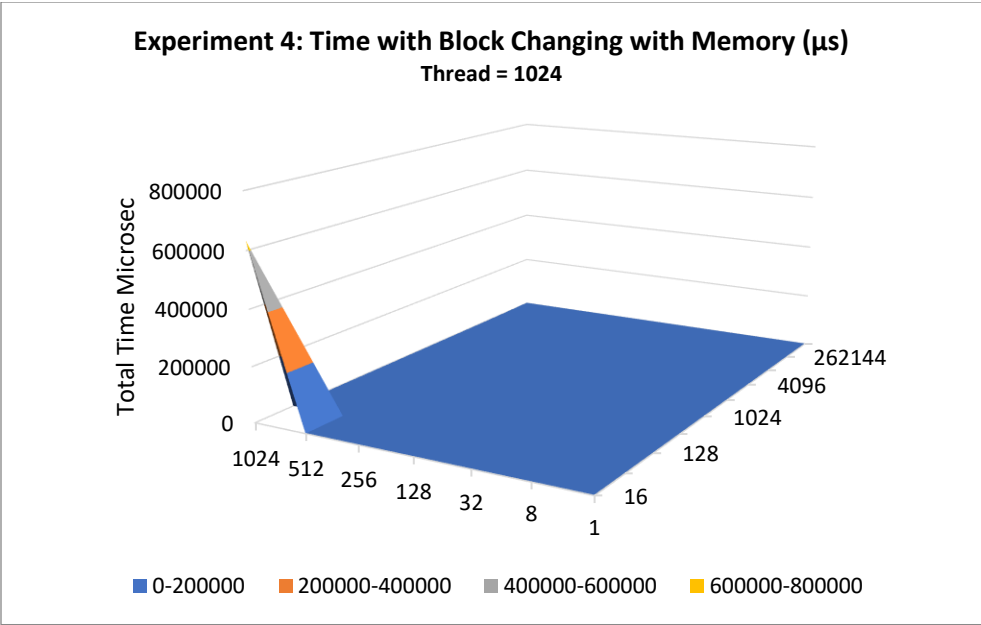


Fig 27, plot of Experiment 4 with memory

Experiment 4: Time with Block Changing without Memory (μ s)										
Block #	Size #									
	16	64	128	512	1024	2048	4096	32768	262144	1048576
1024	13387.7	9.216	8.992	9.024	8.192	8.192	8.192	13.024	13.984	39.36
512	6.56	6.144	5.12	6.848	6.496	6.912	5.12	10.816	7.648	22.432
256	5.12	5.056	5.12	4.448	5.12	6.144	5.12	8.8	6.464	12.512
128	4.096	4.096	5.12	4.864	4.096	5.12	4.096	9.152	12.288	7.968
32	4.032	3.744	4.096	4.096	4.992	5.12	4.096	9.056	4.192	11.264
8	4.96	4.096	4.096	4.096	4.32	4.096	4.096	8.736	4.128	4.768
1	4.064	4.32	3.904	4.096	4.16	4.096	5.12	8.224	4.8	10.944

Fig 28, raw data of Experiment 4 without memory

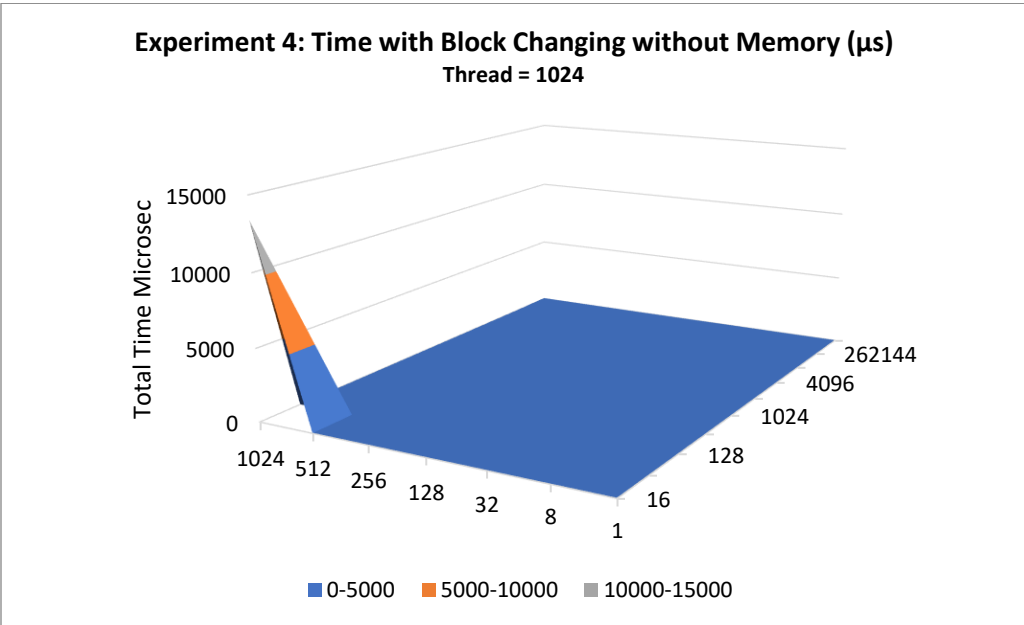


Fig 29, plot of Experiment 4 without memory

8.5. Plot for Experiment 5 – GPU_CPU_Compare by 2D

Size #	GPU_time_withMemory	GPU_time_opeartion	CPU_time
8	654.387	27.65	0.116
16	0.147584	0.009888	0.126
32	0.112304	0.008192	0.225
64	0.110985	0.009056	0.388
128	0.106506	0.008192	13.93
256	0.101966	0.00832	1.352
512	0.094205	0.009184	2.687
1024	0.094705	0.008896	5.381
2048	0.134259	0.022528	10.657
4096	0.096865	0.008192	21.565
8192	0.103159	0.007168	42.283
16384	0.119284	0.007168	84.922
32768	0.147302	0.013152	172.529
65536	0.197459	0.011872	356.545
131072	0.35141	0.017024	721.028
262144	0.617858	0.013344	1415.15
524288	0.982015	0.02016	2747.36
1048576	1.64542	0.03072	5493.86

Fig 30, raw data of Experiment 5

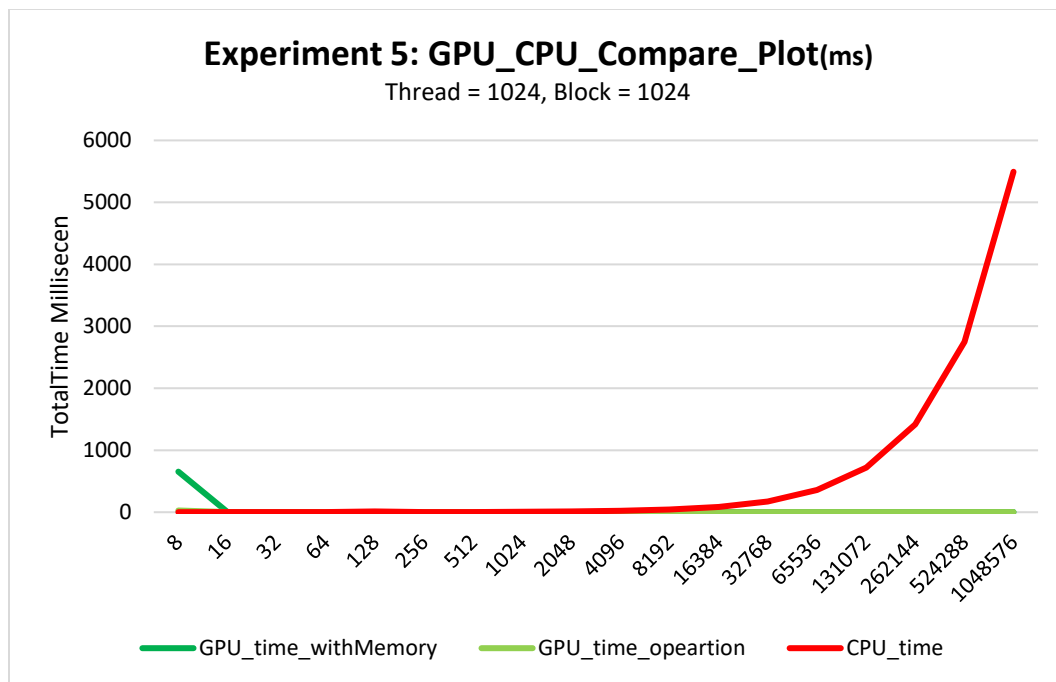


Fig 31, plot of Experiment 5

9. Part 4 - Report Findings

# Experiment	Finding From Data and Plot
1 – thread vary under different vector size, block = 1, thread ascending order	<p>Experiment 1 - with memory, called as E1-m, and Experiment 1 - just operation, called as E1-op, if mentioned later. From Fig 14 – 17, we can find:</p> <ol style="list-style-type: none"> 1. The threads number and the vector size are the same for each case. And they are very small, ≤ 512. 2. The peak time-consuming in Experiment 1-m is larger about 40 times than the one in Experiment 1-op. Except the first case call, for the corresponding same case, the time-consuming of E1-m is larger about 20-25 times than in E1-op. 3. For both of them, the plots have a very sharp hill like. After the first addition call (thread = 8, size = 8), the plots become plain. 4. For both of them, the first vector addition call (thread = 8 and size = 8) has the highest time-consuming. Then except this call, the left is very small comparing this highest one. 5. For both of them, when fixed thread, increasing the vector size won't bring obvious time-consuming increasing without case (thread = 8 and size = 8), here all the size is small. 6. For both of them, when vector size fixed, increasing the thread number won't bring time-consuming decreasing without considering case thread = 8 and size = 8. 7. A very interesting finding is that the (thread, size) combination: when (8, 512) or (512, 8) combination, it brings a second small time-consuming peak.
2 – block vary under different vector size, thread = 1024, block ascending order	<p>Experiment 2 - with memory, called as E2-m, and Experiment 2 - just operation, called as E2-op, if mentioned later. From Fig 18 – 21, we can find:</p> <ol style="list-style-type: none"> 1. The block number is small ≤ 128. The vector size is much bigger, increasing from thousands to hundred-thousand. 2. For both of them, there are no obvious big time-consuming, which leads there are no sharp hills and the plots are relatively plain comparing in Experiment 1. And the plots have several very low hills. 3. For the corresponding same case, time-consuming in E2-m is larger than in E2-op, and this gap is increasing as the size increase. 4. When vector size fixed, increasing the block number won't bring time-consuming saving. 5. When block number fixed, increasing the vector size will bring obvious time-consuming increase, when size increasing from thousand to ten thousand, and from ten-thousands to hundred-thousand.
3 – thread & block vary together under different vector size, ascending order	<p>Experiment 3 - with memory, called as E3-m, and Experiment 3 - just operation, called as E2-op, if mentioned later. From Fig 22 – 25, we can find:</p> <ol style="list-style-type: none"> 1. For each case, the block number = thread number, both are small ≤ 64. The vector size is much bigger, increasing from tens, hundreds, thousands, until ten-thousands. 2. Like in Experiment 2, for both of them, the plots have relatively plain and have several very low hills, and the plots have no sharp hills. 3. For the corresponding case, time-consuming in E3-m is larger than in E3-op, and this gap is increasing as the size increase. 4. When vector size fixed, increasing the block and thread number won't bring obvious time-consuming saving. 5. When block and thread number fixed, increasing the vector size will bring obvious time-consuming increase, when size increasing from thousand to ten thousand.
4 – block vary under different	<p>Experiment 4 - with memory, called as E4-m, and Experiment 4 - just operation, called as E4-op, if mentioned later. From Fig 26 – 29, we can find:</p>

vector size, thread = 1024, block descending order	<p>1. For each case, the block number decreases from 1024 to 1 by 2^{Δ}. The vector size increases from tens, hundreds, thousands, ten-thousands, hundred-thousands, until million.</p> <p>2. Like in Experiment 1, for both of them, the plots have very sharp hills, and after the first addition call (block = 1024, size = 16), the plots become plain.</p> <p>3. For both of them, the first case call (block = 1024 and size = 8) has the highest time-consuming, and the left cases have very small time-consuming comparing first case call.</p> <p>4. The case (block = 1024, size = 16) time-consuming in Experiment 4-m is larger about 40 times than the one in Experiment 4-op. Except the first case call, for the other corresponding same case, the time-consuming in E4-m is larger than in E4-op, and this gap is increasing as the size increase, just like in Experiment 2.</p> <p>3. When vector size fixed, decreasing the block number won't bring obvious time-consuming saving except considering the case (block = 1024 and size = 8).</p> <p>5. When block number fixed, increasing the vector size will bring obvious time-consuming increase, when size increasing from thousands to ten-thousands, ten- thousands to hundred-thousands, hundred-thousand to million.</p>
5 – GPU and CPU	<p>From Fig 31, we can find:</p> <p>1. The GPU time consuming is much larger than sequential CPU for the first case call, whose size is 8.</p> <p>2. The sequential CPU and GPU have the almost same little time consuming when vector size is small, like under about 10,000.</p> <p>3. When size > about 30,000, the CPU time consuming begins to increase, then, when size > about 1,000,000, its time-consuming increase very sharply. Opposite, the GPU's time-consuming is still keeping very low level even the vector size is over millions.</p>

10. Conclusion and Brief Analysis

10.1. Based on Finding of Experiment 1 - 2 - 3 - 4

Based on Experiment 1, 2, 3, 4 findings, we can get these conclusions:

1) About time-consuming

- When size is fixed, no matter block or thread number changes in both or just one, the time-consuming is almost similar. This suggests that the computational workload is not greatly affected by variations in block or thread numbers.
- The primary factor influencing time-consuming is the size of the vector. As the vector size increases, the time-consuming also increases. The noticeable increase in time-consuming becomes more apparent as the vector size grows from tens of thousands to hundreds of thousands and beyond. The increase in time-consuming with an increase in vector size is primarily related to the computational complexity and the nature of memory access patterns.

2) About the first case call's very large time-consuming comparing other cases

- GPU Start Call: When GPU programs start executing GPU operations, the first call (referred to as GPU Start Call) tends to have very large time-consuming. This is a common characteristic in GPU

programming. Initializing and setting up GPU resources for the first time can involve additional overhead. Once the GPU is "warmed up," subsequent calls become more efficient.

About time-consuming in different thread number and block number combination

- Thread and Block Number Variation: Whether the thread and block numbers vary together or separately, their time-consuming is similar. This suggests that the relationship between thread and block numbers does not significantly impact the overall time-consuming. The key driver appears to be the vector size.

About time-consuming in GPU operations

- In each case, it's evident that the primary bottleneck in GPU performance lies in memory access rather than the actual computational operations. The considerable time consumed by GPU operations is predominantly attributed to the challenges associated with memory access. To enhance the efficiency of GPU processing, a promising optimization avenue is to reduce the frequency of memory access and implementing techniques to streamline data transfer between the CPU and GPU.

10.2. Based on Finding of Experiment 5

Based on Experiment 1, 2, 3, 4 findings, we can get these conclusions:

1) About The First Case Call

- The GPU time-consuming is significantly larger than the sequential CPU time for the first case call. This initial high time-consuming on the GPU is likely due to the overhead associated with initializing and setting up the GPU (referred to as GPU Start Call in my terminology of "10.1.").
- After the initial setup, subsequent GPU calls become more efficient, resulting in lower time-consuming.

2) Similar Time-Consuming for Small Vector Sizes (<10,000)

- For small vector sizes (under about 10,000), the sequential CPU and GPU exhibit almost the same low level of time-consuming.
- This similarity is expected because, for small datasets, the overhead of transferring data between the CPU and GPU, as well as the overhead associated with parallelization on the GPU, may not provide a significant advantage over the sequential CPU execution.

3) CPU Time-Consuming Increase with Larger Vector Sizes

- When the vector size exceeds about 30,000, the CPU time-consuming begins to increase. This could be attributed to the fact that, as the vector size grows, the CPU's sequential execution becomes more computationally intensive and starts to experience the limitations of single-threaded performance.
- The sharp increase in CPU time-consuming when the vector size is over 1,000,000 suggests that the CPU is struggling to handle the increased computational workload efficiently. This behavior is typical when

the dataset outgrows the CPU cache, leading to more frequent cache misses and a subsequent slowdown in performance.

4) GPU's Robust Performance with Large Vector Sizes

- In contrast to the CPU, the GPU's time-consuming remains at a low level even when the vector size exceeds millions.
- GPUs are highly parallel processors, and their architecture is well-suited for handling large datasets with parallelizable workloads. The increase in vector size may not have a significant impact on the GPU's ability to efficiently parallelize the computation, leading to a more consistent and lower time-consuming.