**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**"Уфимский государственный авиационный технический университет"**

**Кафедра искусственного интеллекта и перспективных математических исследований**

**Дисциплина:** Параллельное программирование

**Отчет по лабораторной работе № 5**

**Тема:** «Параллельное сложение векторов на графическом процессоре

средствами CUDA»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа ПМ-357 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Лонщакова А.А |  |  |  |
| Принял | Юлдашев А.В. |  |  |  |

**Уфа 2022**

**Цель:** на примере задачи параллельного сложения векторов научиться разрабатывать простейшие параллельные программы средствами CUDA C.

**Теоретический материал**

**Квалификаторы**

В CUDA C введены квалификаторы типа функции и типа переменной. Первые определяют, что запускает и что выполняет функцию – устройство или хост. Вторые определяют область памяти GPU, в которой хранится переменная.

Квалификаторы функций:

\_\_device\_\_ указывает, что функция:

* Выполняется на устройстве,
* Вызывается только с устройства.

\_\_global\_\_ указывает, что функция является ядром. Эта функция:

* Выполняется на устройстве,
* Вызывается с хоста,
* Для GPU с Compute Capability 3.0 и выше может вызываться также и с устройства,
* Имеет тип void.

\_\_host\_\_ указывает, что функция:

* Выполняется на хосте,
* Вызывается только с хоста.

Объявление функции только с квалификатором \_\_host\_\_ равносильно объявлению без какого-либо квалификатора.

Объявление функции с квалификаторами \_\_global\_\_ и \_\_host\_\_ или \_\_global\_\_ и \_\_device\_\_ одновременно не допускается.

Объявление функции с квалификаторами \_\_host\_\_ и \_\_device \_\_ одновременно допускается. В этом случае функция будет скомпилирована и для хоста, и для устройства.

Основные квалификаторы переменных:

\_\_device\_\_ указывает, что переменная хранится в памяти устройства. Все нижеперечисленные квалификаторы переменных могут использоваться вместе с \_\_device\_\_ для более точного указания, в какой памяти располагается переменная. Если же никакого другого квалификатора, кроме \_\_device\_\_ не указано, то переменная:

* Располагается в глобальной памяти устройства,
* Существует на протяжении выполнения программы,
* Доступна для любой нити в сетке, а также для хоста через функции библиотеки среды выполнения.

Переменная, объявленная с квалификатором \_\_constant\_\_, подчиняется тем же правилам, что и \_\_device\_\_-переменная, с одной оговоркой, что располагается в константной памяти устройства.

\_\_shared\_\_ указывает, что переменная:

* Располагается в разделяемой памяти блока нитей,
* Существует на протяжении выполнения блока,
* Доступна только для нитей блока.

**Функция-ядро**

Функция-ядро объявляется со спецификатором \_\_global\_\_ и имеет тип void.

\_\_global\_\_ void kernel\_function (float\* data) {

int tid = blockIdx.x \* blockDim.x + threadIdx.x;

float x = data[tid];

...

return;

}

Простейший вызов функции-ядра происходит с использованием специального синтаксиса.

kernel\_function <<<GridDim, BlockDim>>> (data);

GridDim задает общее количество блоков (размерность сетки), а BlockDim – количество нитей в блоке (размерность блока). Функция-ядро выполняется множеством запущенных нитей в параллельном режиме.

Вызов функции-ядра является асинхронным: управление возвращается на хост до момента завершения выполнения ядра на устройстве.

**Нити и блоки**

Все порождаемые нити группируются в блоки заданного размера, которые образуют сетку блоков. Как блок нитей, так и сетка блоков могут быть одномерными, двумерными или трехмерными. Поэтому размерность блоков и сетки в вызове функции-ядра может быть задана и скалярной величиной (integer), и структурой dim3 (трехмерный вектор). Конструкторы структуры dim3, в которых опускаются одно или несколько измерений, записывают в соответствующие переменные единицы.

Для доступа к информации о размерностях блока или сетки, а также о нумерации нитей и блоков из функции-ядра существуют встроенные переменные:

* uint3 threadIdx – индекс нити в блоке;
* uint3 blockIdx – индекс блока в сетке;
* dim3 blockDim – размер блока;
* dim3 gridDim – размер сетки.

Основное требование к блокам нитей заключается в том, что они должны исполняться независимо: необходимо иметь возможность выполнять блоки в любом порядке как параллельно, так и последовательно. Это требование позволяет писать код, выполняющийся корректно на GPU с любым числом мультипроцессоров.

**Работа с памятью**

Так как память хоста и устройства разделена физически, а функция-ядро оперирует с памятью на устройстве, возникает потребность управлять памятью устройства с хоста, например, копировать данные из одной памяти в другую. Библиотека среды выполнения поддерживает различные функции работы с памятью на устройстве и копирования данных. Перечислим наиболее широко употребляющиеся:

cudaError\_t cudaMalloc(void \*\*devPtr, size\_t size) – выделяет size байт на устройстве и возвращает в \*devPtr указатель на выделенную область памяти.

cudaError\_t cudaFree(void \*devPtr ) – освобождает область памяти, на которую указывает devPtr.

cudaError\_t cudaMemset(void \*devPtr, int value, size\_t count) – заполняет каждый байт из первых count байтов в области памяти, на которую указывает devPtr, значением value.

cudaError\_t cudaMemcpy(void \*dst, const void \*src, size\_t count, enum cudaMemcpyKind kind) – копирует count байтов из области памяти, на которую указывает src в область памяти, на которую указывает dst. При этом kind указывает направление копирования и может принимать значения:

* cudaMemcpyHostToHost – из памяти хоста в память хоста,
* cudaMemcpyHostToDevice – из памяти хоста в память устройства,
* cudaMemcpyDeviceToHost – из памяти устройства в память хоста,
* cudaMemcpyDeviceToDevice – из памяти устройства в память устройства.

Приведем пример работы с памятью в программе.

...

int n = 5;

int size = n \* sizeof(int);

int \*a = (int\*)malloc(size); // Allocate space on host

int \*d\_a, // device copies of a

cudaMalloc((void \*\*)&d\_a, size); // Allocate space on device

dataInit(a); // Some data initialization;

cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice); // Copy data to GPU

someKernel<<<GridDim, BlockDim>>>(d\_a); // Manipulate data on GPU

cudaMemcpy(a, d\_a, size, cudaMemcpyDeviceToHost); // Copy back to CPU

someFunction(a); Manipulate data on CPU

cudaFree(d\_a); // Cleanup

...

В CUDA 4.0 была представлена технология Unified Virtual Addressing (UVA), поддерживаемая устройствами с Compute Capability 2.0 и выше. Суть этой технологии в том, что память хоста и устройства, оставаясь физически разделенной, имеет общее адресное пространство. Это позволяет определить расположение области памяти по значению указателя на эту область.

Данная технология упрощает использование cudaMemcpy(), заменяя все возможные значения kind одним – cudaMemcpyDefault.

В CUDA 6.0 появилась еще одна технология, значительно упрощающая работу с памятью – Unified Memory (поддерживается устройствами с Compute Capability 3.0 и выше). Эта технология избавляет программиста от использования cudaMemcpy() и дублирования указателей для данных на хосте и девайсе. Память, выделенная с помощью:

cudaError\_t cudaMallocManaged(void \*\*devPtr, size\_t size),

доступна по \*devPtr и с хоста, и с устройства. Все задачи по копированию ложатся на драйвер CUDA. От программиста при этом требуется следить за тем, чтобы не было одновременных изменений данных и на хосте, и на устройстве. Так, приведенный выше пример, в случае использования Unified Memory, будет выглядеть следующим образом.

...

int n = 5;

int size = n \* sizeof(int);

int \*a;

cudaMallocManaged((void \*\*)&a, size); // Allocate space

dataInit(a); // Some data initialization

someKernel<<<GridDim, BlockDim>>>(a); // Manipulate data on GPU

cudaDeviceSynchronize(); // Waiting for finish manipulate data on GPU

someFunction(a); Manipulate data on CPU

cudaFree(a); // Cleanup

...

Для использования Unified Memory в случае статического объявления, в CUDA 6.0 был добавлен квалификатор типа переменной \_\_managed\_\_.

**Синхронизация хоста и устройства**

Так как вызовы ядер, а также некоторые других функций CUDA, являются асинхронными, необходимо иметь функцию-барьер, которая заставит хост ожидать завершения работы на устройстве. В CUDA для этого используется функция:

cudaError\_t cudaDeviceSynchronize(void).

**Получение информации об устройстве**

На вычислительной системе может быть установлено несколько различных графических процессоров, поддерживающих CUDA. Основные функции для оперирования устройствами:

cudaError\_t cudaGetDeviceCount(int\* count) – возвращает в count число доступных для вычислений устройств.

cudaError\_t cudaGetDevice(int\* device) – возвращает в device номер устройства, которое используется для вычислений.

cudaError\_t cudaSetDevice(int device) – устанавливает устройство, которое будет использоваться для вычислений.

cudaError\_t cudaGetDeviceProperties(cudaDeviceProp\* prop, int device) – возвращает в prop структуру, содержащую описание свойств устройства с номером device.

**Замер времени вычислений на устройстве**

В CUDA для мониторинга выполнения вычислений на устройстве и наиболее точного замера времени используются события (events). Событие совершается, когда все задачи на устройстве (или команды в указанном CUDA-потоке), запущенные до его записи, выполнены. Перечислим команды работы с событиями.

cudaError\_t cudaEventCreate(cudaEvent\_t\* event) – создает объект события.

cudaError\_t cudaEventDestroy(cudaEvent\_t event) – уничтожает объект события.

cudaError\_t cudaEventRecord(cudaEvent\_t event, cudaStream\_t stream = 0) – запись события.

cudaError\_t cudaEventSynchronize(cudaEvent\_t event) – ожидает совершения события.

cudaError\_t cudaEventElapsedTime (float\* ms, cudaEvent\_t start, cudaEvent\_t end) – возвращает в ms время между совершением событий start и end в миллисекундах.

Приведем пример кода, замеряющего время выполнения некоторой функции-ядра.

...

float elapsedTime;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

someKernel<<<GridDim, BlockDim>>>(data);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime, start, stop);

cudaEventDestroy(start);

cudaEventDestroy(stop);

...

**Индивидуальное задание**

Задание:

1. Реализовать версию №1 сложения на GPU двух векторов с элементами типа float с использованием. Инициализацию элементов векторов провести явно на CPU. При вычислениях на GPU взять 1024 нити в блоке. Предусмотреть проверку корректности вычислений, обработку ошибок и замер времени выполнения функции-ядра.
2. Реализовать версию №2 сложения на GPU двух векторов с элементами типа float без использования Unified Memory. Предусмотреть замер времени выполнения функции-ядра и копирования данных.

Результаты:

Тесты проводились на видеокарте со следующими характеристиками:

1. Архитектура Quadro P2000
2. CUDA Capability Major/Minor version number 6.1
3. Total amount of global memory 5057 Mbytes (5GB)
4. Пропускная способность памяти 140 GB/s
5. Число СUDA-ядер 1024
6. Пиковая производительность 3.0 TFLOPS

Протестируем версию №1:

Таблица 1. Результаты тестирования версии №1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Время (мс), производительность (гигафлопс) / размерность |  |  |  |  |  |
| 1.1 | Время выполнения на GPU | 0,55 | 2.33 | 19,87 | 198,406 | 589.146 |
| 1.2 | Производительность | 0,1818 | 0,429 | 0,503 | 0,504 | 0,509 |

Протестируем версию №2:

Таблица 2. Результаты тестирования версии №2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Время (мс), производительность (гигафлопс) / размерность |  |  |  |  |  |
| 1.1 | Время выполнения на GPU (1.2 + 1.3) | 0,392 | 3,06 | 27,5 | 272 | 815,674 |
| 1.2 | Время обмена данными с GPU | 0,37 | 2,95 | 26,6 | 262,49 | 786,569 |
| 1.3 | Время расчёта на GPU | 0,022 | 0.11 | 0,98 | 9,68 | 29,105 |
| 1.4 | Производительность расчёта на GPU | 4.55 | 9.09 | 10.20 | 10.33 | 10.31 |
| 1.5 | Пропускная способность при работе с памятью GPU, GB/S | 50,80 | 101,60 | 114,04 | 115,45 | 115,20 |

Исходя из результатов тестирования, можно построить графики зависимости времени и производительности от размерности векторов (оранжевая линия – версия №2, синяя – версия №1):

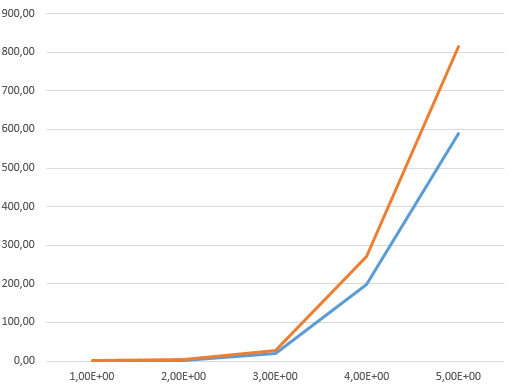


Рисунок 1. График зависимости времени от размерности векторов.

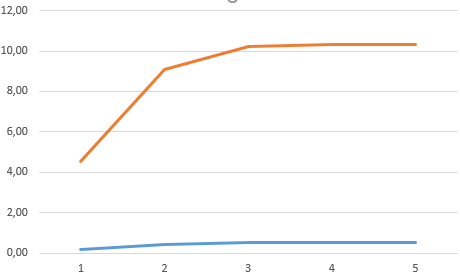


Рисунок 2. График зависимости производительности от размерности векторов

Также была проведена профилировка программы:

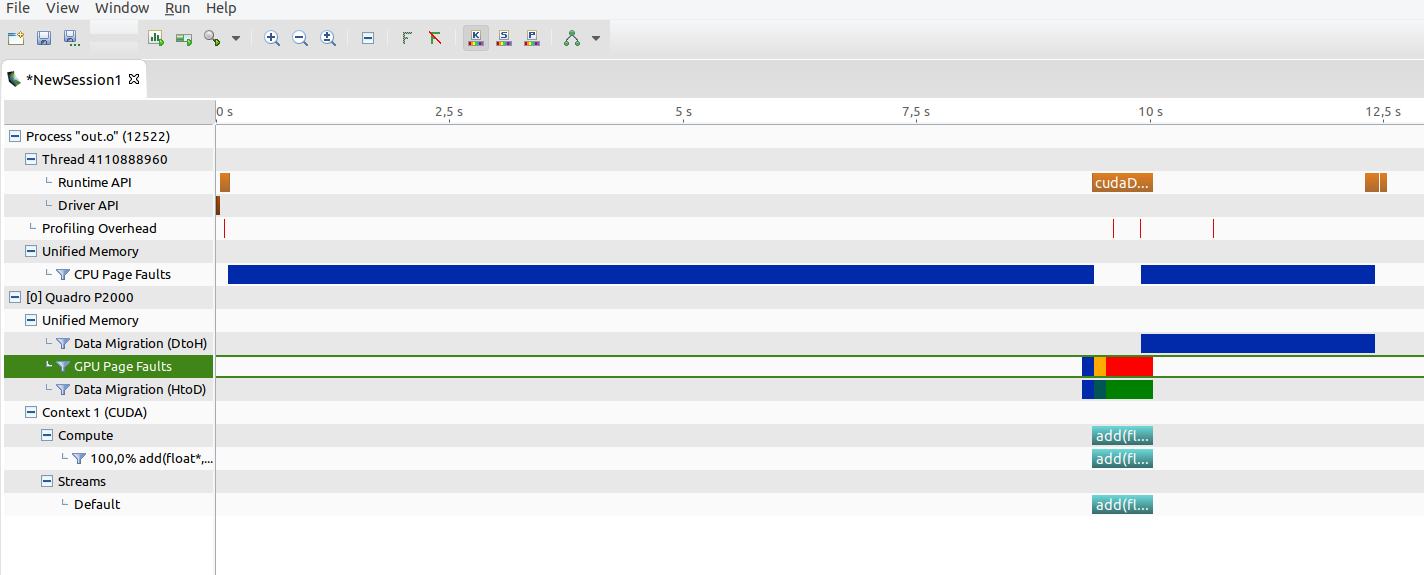


Рисунок 3. Профилировка программы для версии №1.

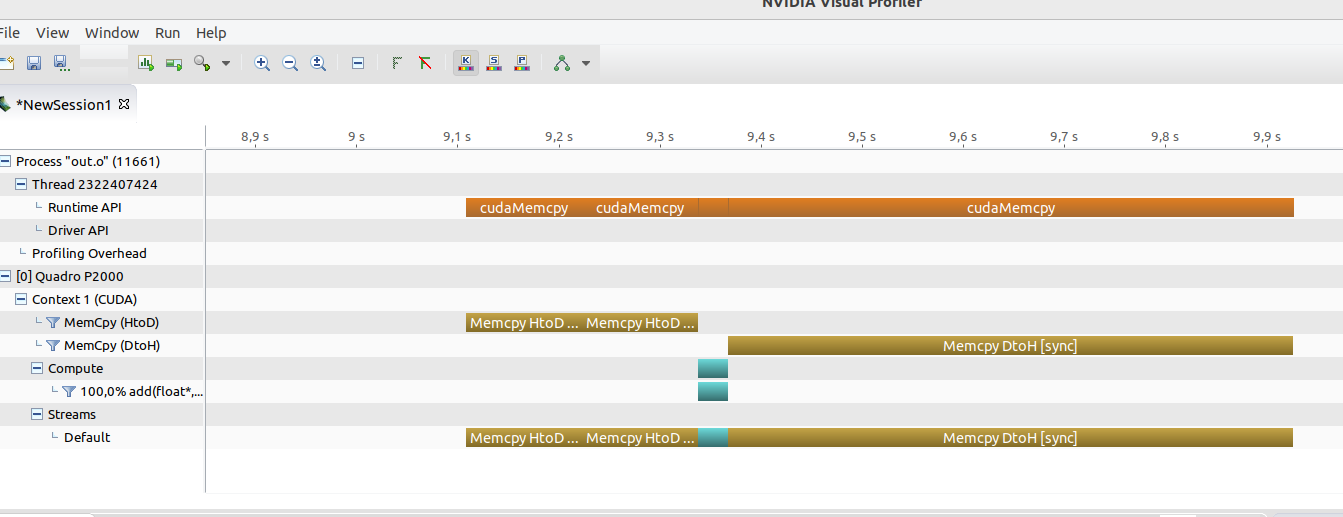


Рисунок 4. Профилировка программы для версии №2.

**Вывод**

В ходе выполнения лабораторной работы на примере задачи параллельного сложения векторов была изучена разработка простейших параллельных программ средствами CUDA C.

**Приложение**

**hello.cu (Unified-memory)**

/// Hello1.cu

#define N 300000000

#define M 1024 // THREADS\_PER\_BLOCK

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

inline void check\_cuda\_errors(const char\* filename, const int line\_number) {

 //#ifdef DEBUG

    cudaDeviceSynchronize();

    cudaError\_t error = cudaGetLastError();

    if (error != cudaSuccess) {

        printf("CUDA error at %s:%i: %s\n", filename, line\_number, cudaGetErrorString(error));

        exit(-1);

    }

 //#endif

}

\_\_global\_\_ void add(float\* a, float\* b, float\* c) {

    int index = threadIdx.x + blockIdx.x \* blockDim.x;

    if (index < N) c[index] = a[index] + b[index];

}

void random\_floats(float\* a, int n) {

    for (int i = 0; i < n; ++i) {

        a[i] = rand();

    }

}

void check\_results(float\* a, float\* b, float\* c, int n) {

    int success = 1;

    for (int i = 0; i < n; ++i) {

        if (!(a[i] + b[i]) == c[i]) {

            success = 0;

            break;

        }

    }

    if (success) {

        printf("Ok\n");

    }

    else {

        printf("Not ok\n");

    }

}

int main(void) {

    srand(time(0));

    float\* a, \* b, \* c;

    int size = N \* sizeof(float);

 // Allocate space

    cudaMallocManaged((void\*\*)&a, size);

    cudaMallocManaged((void\*\*)&b, size);

    cudaMallocManaged((void\*\*)&c, size);

    random\_floats(a, N); //setup input values

    random\_floats(b, N);

 //

    cudaEvent\_t start, stop;

    float time = 0;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    add << <(N + M - 1) / M, M >> > (a, b, c); // Launch add() kernel on GPU

    cudaDeviceSynchronize();

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&time, start, stop);

    printf("Elapsed time : %.2f ms\n", time);

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    check\_cuda\_errors(\_\_FILE\_\_, \_\_LINE\_\_);

    check\_results(a, b, c, N);

    // Cleanup

    cudaFree(a);

    cudaFree(b);

    cudaFree(c);

    return 0;

}

**hello2.cu**

// Hello2

#define N 300000000

#define M 1024 // THREADS\_PER\_BLOCK

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

inline void check\_cuda\_errors(const char\* filename, const int line\_number) {

//#ifdef DEBUG

    cudaDeviceSynchronize();

    cudaError\_t error = cudaGetLastError();

    if (error != cudaSuccess) {

        printf("CUDA error at %s:%i: %s\n", filename, line\_number, cudaGetErrorString(error));

        exit(-1);

    }

//#endif

}

\_\_global\_\_ void add(float\* a, float\* b, float\* c) {

    int index = threadIdx.x + blockIdx.x \* blockDim.x;

    if (index < N) c[index] = a[index] + b[index];

}

void random\_floats(float\* a, int n) {

    for (int i = 0; i < n; ++i) {

        a[i] = rand();

    }

}

void check\_results(float\* a, float\* b, float\* c, int n) {

    int success = 1;

    for (int i = 0; i < n; ++i) {

        if (!(a[i] + b[i]) == c[i]) {

            success = 0;

            break;

        }

    }

    if (success) {

        printf("Good\n");

    }

    else {

        printf("Not good\n");

    }

}

int main(void) {

    srand(time(0));

    int size = N \* sizeof(float);

    float\* a = new float[N], \* b = new float[N], \* c = new float[N];

 //Allocate space

    random\_floats(a, N); //setup input values

    random\_floats(b, N);

    float\* dev\_a;

    float\* dev\_b;

    float\* dev\_c;

    cudaMalloc((void\*\*)&dev\_a, size);

    cudaMalloc((void\*\*)&dev\_b, size);

    cudaMalloc((void\*\*)&dev\_c, size);

    check\_cuda\_errors(\_\_FILE\_\_, \_\_LINE\_\_);

    cudaEvent\_t start\_exchange1, stop\_exchange1;

    float time\_exchange1 = 0;

    cudaEventCreate(&start\_exchange1);

    cudaEventCreate(&stop\_exchange1);

    cudaEventRecord(start\_exchange1, 0);

    cudaMemcpy(dev\_a, a, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_b, a, size, cudaMemcpyHostToDevice);

    cudaEventRecord(stop\_exchange1, 0);

    cudaEventSynchronize(stop\_exchange1);

    cudaEventElapsedTime(&time\_exchange1, start\_exchange1, stop\_exchange1);

    cudaEventDestroy(start\_exchange1);

    cudaEventDestroy(stop\_exchange1);

    check\_cuda\_errors(\_\_FILE\_\_, \_\_LINE\_\_);

    cudaEvent\_t start, stop;

    float time = 0;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);

    add <<<(N + M - 1) / M, M >>> (dev\_a, dev\_b, dev\_c); // Launch add() kernel on GPU

    cudaDeviceSynchronize();

    cudaEventRecord(stop, 0);

    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&time, start, stop);

    printf("Elapsed time : %.2f ms\n", time);

    cudaEventDestroy(start);

    cudaEventDestroy(stop);

    check\_cuda\_errors(\_\_FILE\_\_, \_\_LINE\_\_);

    cudaEvent\_t start\_exchange2, stop\_exchange2;

    float time\_exchange2 = 0;

    cudaEventCreate(&start\_exchange2);

    cudaEventCreate(&stop\_exchange2);

    cudaEventRecord(start\_exchange2, 0);

    cudaMemcpy(c, dev\_c, size, cudaMemcpyDeviceToHost);

    cudaEventRecord(stop\_exchange2, 0);

    cudaEventSynchronize(stop\_exchange2);

    cudaEventElapsedTime(&time\_exchange2, start\_exchange2, stop\_exchange2);

    cudaEventDestroy(start\_exchange2);

    cudaEventDestroy(stop\_exchange2);

    printf("Elapsed time for exchange: %.2f ms\n", time\_exchange1 + time\_exchange2);

    check\_cuda\_errors(\_\_FILE\_\_, \_\_LINE\_\_);

    check\_results(a, b, c, N);

 //Cleanup

    delete[] a;

    delete[] b;

    delete[] c;

    cudaFree(dev\_a);

    cudaFree(dev\_b);

    cudaFree(dev\_c);

    return 0;

}