

**УФИМСКИЙ
УНИВЕРСИТЕТ
НАУКИ И ТЕХНОЛОГИЙ**

Курс «Технологии
параллельного
программирования»

Лабораторная работа №4. Параллельное сложение векторов на графическом процессоре

Юлдашев Артур Владимирович
art@ugatu.su

Спеле Владимир Владимирович
spele.vv@ugatu.su

Добровольцев Александр Сергеевич
dobrovolcev.as@ugatu.su

Сохатский Михаил Александрович
sohatskii.ma@ugatu.su

**Кафедра высокопроизводительных
вычислений и дифференциальных
уравнений (ВВиДУ)**

Цель работы

На примере задачи параллельного сложения векторов научиться разрабатывать простейшие параллельные программы средствами CUDA C.

Используемые элементы CUDA:

- реализация и вызов вычислительного ядра;
- функции выделения памяти;
- технология Unified Memory;
- функции явного копирования данных между CPU и GPU.

Команды для получения информации об установленных GPU (Linux)

➤ `lspci | grep -i nvidia`

14:00.0 3D controller: nVidia Corporation GF100 [Tesla S2050] (rev a3)

14:00.1 Audio device: nVidia Corporation GF100 High Definition Audio Controller (rev a1)

15:00.0 3D controller: nVidia Corporation GF100 [Tesla S2050] (rev a3)

15:00.1 Audio device: nVidia Corporation GF100 High Definition Audio Controller (rev a1)

➤ `nvidia-smi [-a]`

GPU 0000:15:00.0

Product Name : Tesla M2050

Display Mode : Disabled

Persistence Mode : Disabled

...

Memory Usage

Total : 2687 MB

Used : 6 MB

Free : 2681 MB

Compute Mode : Default

Utilization

Gpu : 0 %

Memory : 0 %

Ecc Mode

Current : Enabled

Pending : Enabled

...

Команда из комплекта компиляторов NVIDIA/PGI (Linux)

➤ nvaccelinfo


Device Number: 0
Device Name: Tesla M2050
Device Revision Number: 2.0
Global Memory Size: 2817982464
Number of Multiprocessors: 14
Number of Cores: 448
Concurrent Copy and Execution: Yes
Total Constant Memory: 65536
Total Shared Memory per Block: 49152
Registers per Block: 32768
Warp Size: 32
Maximum Threads per Block: 1024
Maximum Block Dimensions: 1024, 1024, 64
Maximum Grid Dimensions: 65535 x 65535 x 65535
Maximum Memory Pitch: 2147483647B
Texture Alignment: 512B
Clock Rate: 1147 MHz
Execution Timeout: No
Integrated Device: No

Can Map Host Memory: Yes
Compute Mode: default
Concurrent Kernels: Yes
ECC Enabled: Yes
Memory Clock Rate: 1546 MHz
Memory Bus Width: 384 bits
L2 Cache Size: 786432 bytes
Max Threads Per SMP: 1536
Async Engines: 2
Unified Addressing: Yes
Initialization time: 5087174 microseconds
Current free memory: 2748571648
Upload time (4MB): 1338 microseconds (754 ms pinned)
Download time: 1122 microseconds (688 ms pinned)
Upload bandwidth: 3134 MB/sec (5562 MB/sec pinned)
Download bandwidth: 3738 MB/sec (6096 MB/sec pinned)

Создание “CUDA Runtime” проекта в Visual Studio 2019 (Windows)

Создание проекта

Последние шаблоны проектов

 CUDA 10.2 Runtime

 CUDA 11.1 Runtime

 Пустой проект

 Консольное приложение

C++

C++

cuda

Все языки

Все платформы

Все типы



CUDA 10.2 Runtime

A project that uses the CUDA 10.2 runtime



CUDA 11.1 Runtime

A project that uses the CUDA 11.1 runtime

Не нашли то, что искали?
[Установка других средств и к...](#)

hello.c

Hello world!

```
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

```
> nvcc -o hello hello.c
> ./hello
```

Hello World!

hello.cu

Hello world!

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void mykernel(void) { }

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World from host!\n");
    return 0;
}
```

```
> nvcc -o hello hello.cu
> ./hello
```

Hello World from host!

hello2.cu

Hello world!

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void mykernel(void) {
    printf("Hello World from thread %d block %d on device!\n",
        threadIdx.x, blockIdx.x);
}

int main(void) {
    mykernel<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

> nvcc -o hello2 hello2.cu

> ./hello2

Hello World from thread 0 block 0 on device!

add.cu

Сложение двух чисел

```
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void add(float *a, float *b, float *c) { *c = *a + *b; }

int main(void) {
    float *a, *b, *c;
    int size = sizeof(float);

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

    *a = 2; *b = 6; //setup input values

    add <<<1, 1>>> (a, b, c); // Launch add() kernel on GPU
    cudaDeviceSynchronize();

    printf("c = %f\n", *c);

    // Cleanup
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

add2.cu

Сложение векторов (один блок нитей)

```
#define N 1024
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void add(float *a, float *b, float *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

int main(void) {
    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);
    add <<<1, N>>> (a, b, c); // Launch add() kernel on GPU
    cudaDeviceSynchronize();
    check_results(a, b, c, N);
    // Cleanup
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

Функции,
выделенные
красным, надо
реализовать
самостоятельно!

add4.cu

Сложение двух векторов (несколько блоков нитей) – v1

```
#define N 4096
#define M 1024 // THREADS_PER_BLOCK
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
__global__ void add(float *a, float *b, float *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
int main(void) {
    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);
    add <<<N / M, M>>> (a, b, c); // Launch add() kernel on GPU
    cudaDeviceSynchronize();
    check_results(a, b, c, N);
    // Cleanup
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

Вопросы

- Будет ли корректно работать программа при $N = 4097$ и других значения N , которые не делятся нацело на M (число нитей в блоке)?
- Что надо изменить?
- Только число блоков или что-то еще?
- Как универсальным образом задать требуемое число блоков?

add5.cu

Сложение двух векторов (несколько блоков нитей) – v2

```
#define N 4097
#define M 1024 // THREADS_PER_BLOCK
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
__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];
}

int main(void) {
    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);
    add <<<(N + M - 1) / M, M >>> (a, b, c); // Launch add() kernel on GPU
    cudaDeviceSynchronize();
    check_results(a, b, c, N);
    // Cleanup
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

Требования к программе сложения двух векторов

- Реализовать версию с использованием Unified Memory сложения на GPU двух векторов с элементами типа float.
- 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.
- ❖ Реализовать версию без использования Unified Memory сложения на GPU двух векторов с элементами типа float.
- ❖ Предусмотреть замер времени выполнения функции-ядра и копирования данных.

Оценка производительности разработанной программы

- Протестировать версию с использованием Unified Memory, заполнить таблицу 1:

№	Время (мс), производительность (гигафлопс) / размерность	10^5	10^6	10^7	10^8	MAX
1.1	Время выполнения на GPU					
1.2	Производительность					

- Протестировать версию без использования Unified Memory, заполнить таблицу 2:

№	Время (мс), производительность (гигафлопс) / размерность	10^5	10^6	10^7	10^8	MAX
1.1	Время выполнения на GPU (1.2 + 1.3)					
1.2	Время обмена данными с GPU					
1.3	Время расчета на GPU					
1.4	Производительность расчета					
1.5	Пропускная способность при работе с памятью GPU					

Требования к оформлению отчета

- В отчет по проделанной работе включить:
 - 1) параметры графического процессора, на котором проводились вычисления:
 - 1) архитектура, СС;
 - 2) объем памяти, пропускная способность;
 - 3) число CUDA-ядер, пиковая производительность;
 - 2) заполненные таблицы 1 и 2;
 - 3) графики зависимости времени, достигнутой производительности и пропускной способности от размерности векторов;
 - 4) скриншоты профиля из NVIDIA Visual Profiler для обеих версий программ;
 - 5) выводы о полученных результатах;
 - 6) листинг разработанных программ;

Базовые функции для динамической работы с памятью GPU

- **cudaError_t cudaMalloc (void ** devPtr, size_t size);**
выделение size байт памяти на GPU
- **cudaError_t cudaFree (void * devPtr);**
освобождение памяти по указателю devPtr
- **cudaError_t cudaMemcpy (void * dst, const void * src, size_t size, enum cudaMemcpyKind kind);**
копирование size байт памяти в направлении kind, которое может принимать следующие значения:
 - **cudaMemcpyHostToHost**
 - **cudaMemcpyHostToDevice**
 - **cudaMemcpyDeviceToHost**
 - **cudaMemcpyDeviceToDevice**
 - **cudaMemcpyDefault** (для GPU с поддержкой UVA)

Обработка ошибок в CUDA. Пример 1

```
#include <stdio.h>
#include <stdlib.h>

__global__ void foo(int *ptr) { *ptr = 7; }

int main(void) {
    foo<<<1,1>>>(0);
    // make the host block until the device is finished with foo
    cudaDeviceSynchronize();
    // check for error
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess) {
        // print the CUDA error message and exit
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }
    return 0;
}
```

Обработка ошибок в CUDA. Пример 2

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptr = 0;

    // gimme!
    cudaError_t error = cudaMalloc((void**) &ptr, UINT_MAX);
    if(error != cudaSuccess) {
        // print the CUDA error message and exit
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }
    return 0;
}
```

Обработка ошибок в CUDA.

Пример 3 (компилировать с **-DDEBUG**)

```
#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 foo(int *ptr) {
    *ptr = 7;
}
```

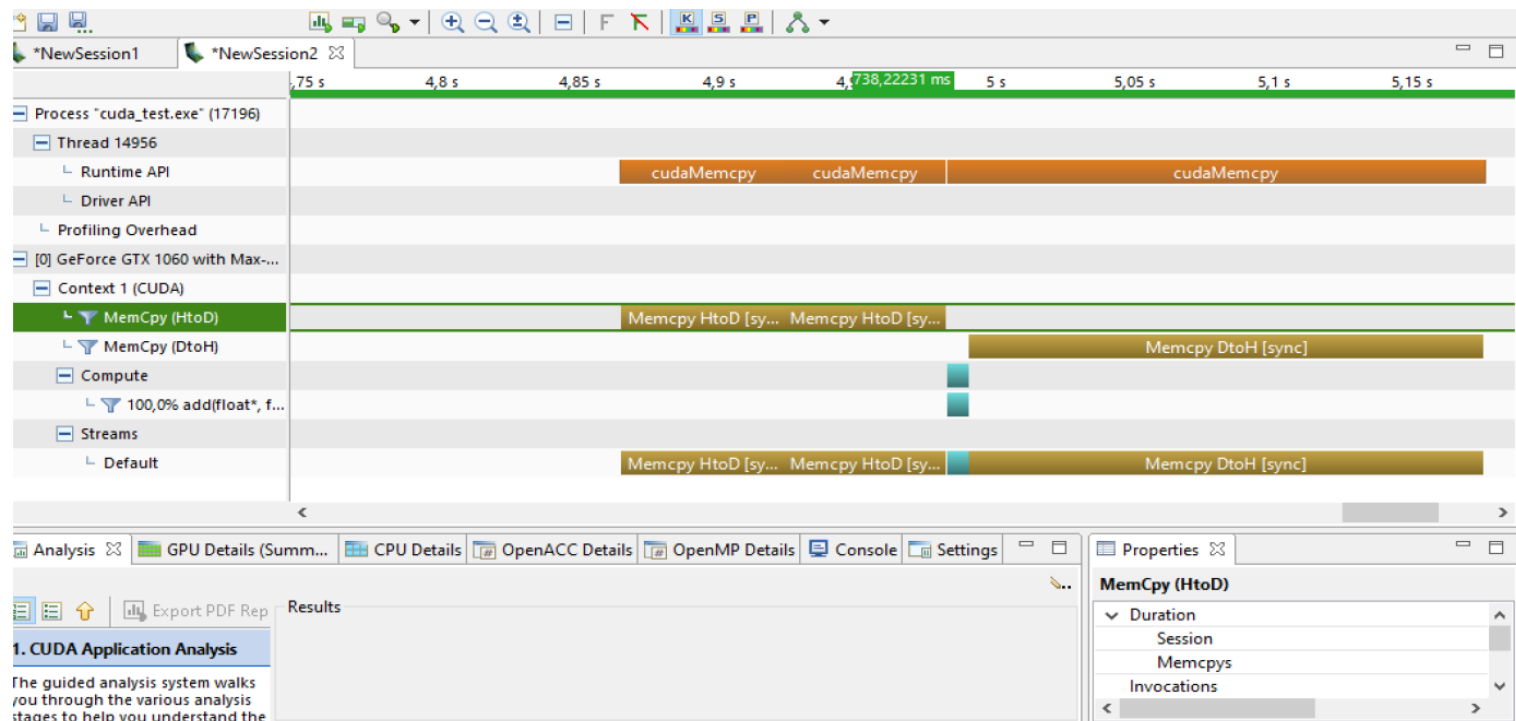
```
int main(void) {
    foo<<<1,1>>>(0);
    check_cuda_errors(__FILE__, __LINE__);
    return 0;
}
```

NVIDIA Visual Profiler

Под Linux запускается из командной строки:

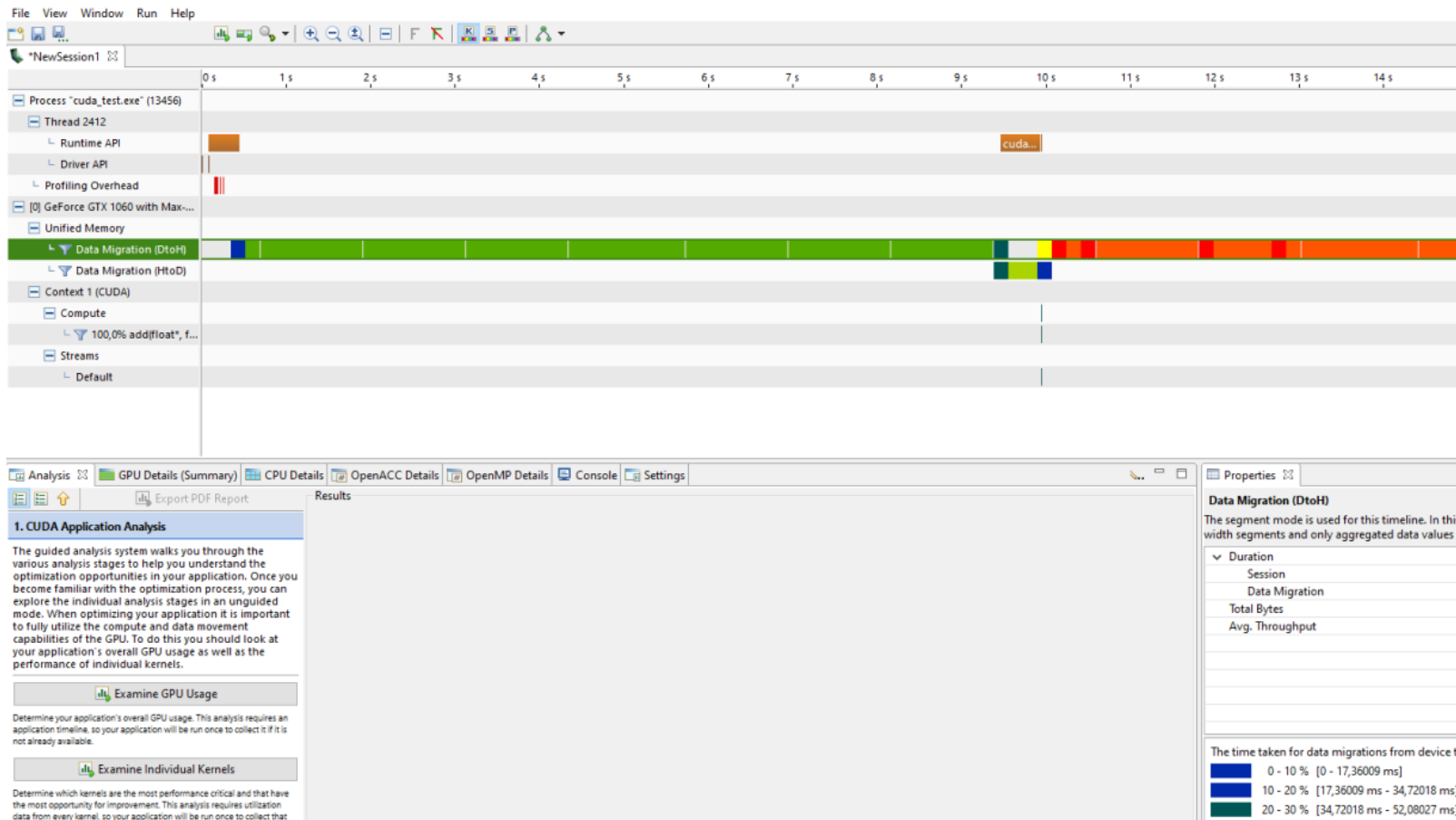
```
/usr/local/cuda-11.8/bin/nvvp -vm /usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java
```

Открываем меню File -> New Session. В пункте File указываем путь к профилируемой программе, нажимаем Next. Далее можно нажать Finish и после этого начнет генерироваться Timeline программы.



NVIDIA Visual Profiler

Профилировка с NVIDIA Visual Profiler



NVIDIA Visual Profiler