ФГБОУ ВО «Национальный исследовательский университет «МЭИ»

**Кафедра вычислительных машин, систем и сетей**

**Лабораторная работа №4**

по курсу «Программное обеспечение высокопроизводительных систем»

по теме «Размещение данных в глобальной, разделяемой и константной памяти»

Выполнили:

Певченко Сергей

Локотков Борис

Салий Екатерина

Бригада № 6

Группа: А-07м-18

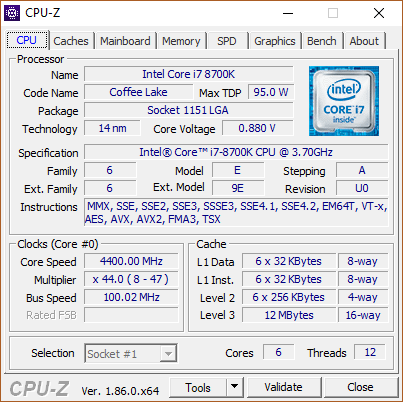
**Цель работы.**

Получение навыков в использовании графических процессоров в качестве сопроцессора для ускорения вычислений.

**Формула вычисления. Вариант задания №6.**

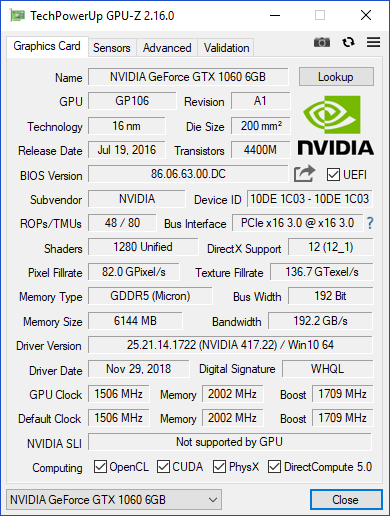
Характеристики ВС, на которой проводились вычисления:

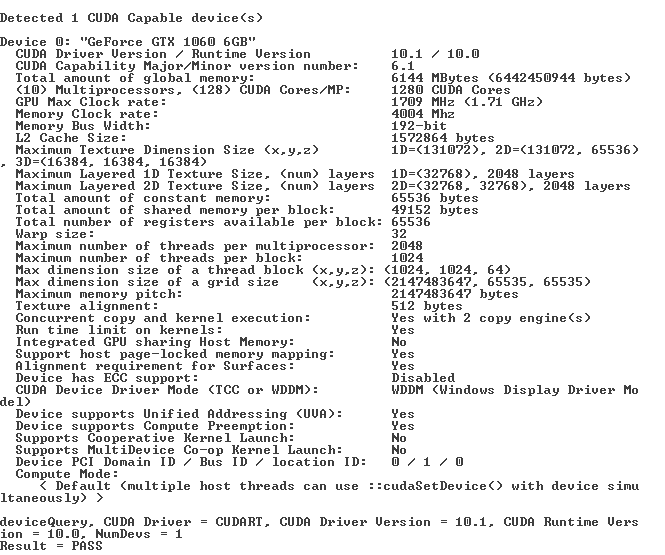
Процессор: Intel Core i7 8700K; Число ядер/потоков: 6/12; Тактовая частота: 4400 МГц;



Операционная система: Windows 10 Pro.

Видеокарта: Nvidia Geforce GTX 1060





3. Программа на CUDA, где все массивы в процессе вычислений хранятся в глобальной памяти.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#define M 381 //21\*7 + 234

#define N 512

\_\_global\_\_ void func\_Kernel(char \*A, char \*B, char \*C, double \*D, double \*X, int s)

{

int idx\_thread = blockIdx.x \* blockDim.x + threadIdx.x;

int i, j;

for (i = 0; i < s; i++)

{

for (j = 0; j < 2 \* M; j++)

{

X[idx\_thread\*s + i] = (double)A[idx\_thread\*s + i] \* X[idx\_thread\*s + i] \* (X[idx\_thread\*s + i] \* C[idx\_thread\*s + i] + B[idx\_thread\*s + i]) / D[idx\_thread\*s + i];

}

}

}

int main()

{

char A[N], B[N], C[N];

double X[N], D[N];

char \*dev\_a, \*dev\_b, \*dev\_c;

double \*dev\_d, \*dev\_x;

int steps, amIt;

int blocks[3] = { 1, 2, 4 };

int blocksize[4] = { 1,4,32,256 };

float srd = 0;

float elapsedTime;

int i, j;

while (true)

{

printf("Number of iterations: ");

scanf("%d", &amIt);

int numIt = amIt;

for (int numB = 0; numB < 3; numB++)

{

for (int numT = 0; numT < 4; numT++)

{

amIt = numIt;

while (amIt > 0)

{

cudaEvent\_t start, stop; //ids of events

cudaEventCreate(&start); //inits of events

cudaEventCreate(&stop);

for (i = 0; i < N; i++)

{

A[i] = i + 1;

B[i] = i + 2;

C[i] = i + 3;

D[i] = i + 4;

}

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for five vectors

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_c, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_d, N \* sizeof(double));

cudaMalloc((void\*\*)&dev\_x, N \* sizeof(double));

//blocks = 1; blocksize = 1;

steps = (int)N / (blocks[numB]\*blocksize[numT]);

cudaEventRecord(start, 0); //capture of event start

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_a, A, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, B, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_c, C, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_d, D, N \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_x, X, N \* sizeof(double), cudaMemcpyHostToDevice);

// Launch a kernel on the GPU with one thread for each element.

func\_Kernel << <blocks[numB], blocksize[numT] >> > (dev\_a, dev\_b, dev\_c, dev\_d, dev\_x, steps);

// cudaDeviceSynchronize waits for the kernel to finish

cudaDeviceSynchronize();

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(X, dev\_x, N \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

cudaFree(dev\_d);

cudaFree(dev\_x);

cudaEventRecord(stop, 0); //capture of event stop

cudaEventSynchronize(stop); //waits for an event to complete

cudaEventElapsedTime(&elapsedTime, start, stop); //computes the elapsed time between events

srd += elapsedTime;

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaDeviceReset();

amIt--;

}

srd /= numIt;

printf("[blocks: %d, threads: %d]Average elapsed time of GPU computing = %f ms\n", blocks[numB], blocksize[numT], srd);

printf("============================================================\n");

srd = 0;

}

}

}

return 0;

}

5. Программа на CUDA, где все массивы в процессе вычислений на device хранятся в разделяемой памяти.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#define M 381 //21\*7 + 234

#define N 512

\_\_global\_\_ void func\_Kernel(char \*A, char \*B, char \*C, double \*D, double \*X)

{

\_\_shared\_\_ char as[N], char bs[N], char cs[N];

\_\_shared\_\_ double xs[N], double ds[N];

int i,j;

int i\_thread = threadIdx.x;

int idx\_thread = blockIdx.x \* blockDim.x + threadIdx.x; // номер потока

int threadCountGlobal = gridDim.x \* blockDim.x;

for (i = idx\_thread; i < N; i += threadCountGlobal)

{

as[i\_thread] = A[i];

bs[i\_thread] = B[i];

cs[i\_thread] = C[i];

ds[i\_thread] = D[i];

xs[i\_thread] = X[i];

\_\_syncthreads();

for (j = 0; j < 2 \* M; j++)

{

xs[i\_thread] = (double)A[i] \* xs[i\_thread] \* (xs[i\_thread] \* C[i] + B[i]) / D[i];

}

\_\_syncthreads();

X[i] = xs[i\_thread];

}

int main()

{

char A[N], B[N], C[N];

double X[N], D[N];

char \*dev\_a, \*dev\_b, \*dev\_c;

double \*dev\_d, \*dev\_x;

int steps, amIt;

int blocks[3] = { 1, 2, 4 };

int blocksize[4] = { 1,4,32,256 };

float srd = 0;

float elapsedTime;

int i, j;

while (true)

{

printf("Number of iterations: ");

scanf("%d", &amIt);

int numIt = amIt;

for (int numB = 0; numB < 3; numB++)

{

for (int numT = 0; numT < 4; numT++)

{

amIt = numIt;

while (amIt > 0)

{

cudaEvent\_t start, stop; //ids of events

cudaEventCreate(&start); //inits of events

cudaEventCreate(&stop);

for (i = 0; i < N; i++)

{

A[i] = i + 1;

B[i] = i + 2;

C[i] = i + 3;

D[i] = i + 4;

}

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for five vectors

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_c, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_d, N \* sizeof(double));

cudaMalloc((void\*\*)&dev\_x, N \* sizeof(double));

//blocks = 1; blocksize = 1;

steps = (int)N / (blocks[numB]\*blocksize[numT]);

cudaEventRecord(start, 0); //capture of event start

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_a, A, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, B, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_c, C, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_d, D, N \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_x, X, N \* sizeof(double), cudaMemcpyHostToDevice);

// Launch a kernel on the GPU with one thread for each element.

func\_Kernel << <blocks[numB], blocksize[numT] >> > (dev\_a, dev\_b, dev\_c, dev\_d, dev\_x, steps);

// cudaDeviceSynchronize waits for the kernel to finish

cudaDeviceSynchronize();

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(X, dev\_x, N \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

cudaFree(dev\_d);

cudaFree(dev\_x);

cudaEventRecord(stop, 0); //capture of event stop

cudaEventSynchronize(stop); //waits for an event to complete

cudaEventElapsedTime(&elapsedTime, start, stop); //computes the elapsed time between events

srd += elapsedTime;

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaDeviceReset();

amIt--;

}

srd /= numIt;

printf("[blocks: %d, threads: %d]Average elapsed time of GPU computing = %f ms\n", blocks[numB], blocksize[numT], srd);

printf("============================================================\n");

srd = 0;

}

}

}

return 0;

}

7. Программа на CUDA, где все массивы кроме X в процессе вычислений на

device хранятся в константной памяти, а массив X в разделяемой памяти.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#define M 381 //21\*7 + 234

#define N 512

char A[N], B[N], C[N];

double X[N], D[N];

\_\_constant\_\_ char Ac[N], char Bc[N], char Cc[N], double Dc[N];

\_\_global\_\_ void func\_Kernel(double \*X, int s)

{

\_\_shared\_\_ double xs[N];

int i, j;

int i\_thread = threadIdx.x; //номер потока в задаче

int idx\_thread = blockIdx.x \* blockDim.x + threadIdx.x; // номер нити

int threadCountGlobal = gridDim.x \* blockDim.x; //всего нитей во всех блоках

for (i = idx\_thread; i < N; i += threadCountGlobal)

{

xs[i\_thread] = X[i];

\_\_syncthreads();

for (j = 0; j < 2 \* M; j++)

{

xs[i\_thread] = (double)Ac[i] \* xs[i\_thread] \* (xs[i\_thread] \* Cc[i] + Bc[i]) / Dc[i];

}

\_\_syncthreads();

X[i] = xs[i\_thread];

}

}

int main()

{

double \*dev\_x;

int steps, amIt;

int blocks[3] = { 1, 2, 4 };

int blocksize[4] = { 1,4,32,256 };

float srd = 0;

float elapsedTime;

int i, j;

while (true)

{

printf("Number of iterations: ");

scanf("%d", &amIt);

int numIt = amIt;

for (int numB = 0; numB < 3; numB++)

{

for (int numT = 0; numT < 4; numT++)

{

amIt = numIt;

while (amIt > 0)

{

cudaEvent\_t start, stop; //ids of events

cudaEventCreate(&start); //inits of events

cudaEventCreate(&stop);

for (i = 0; i < N; i++)

{

A[i] = i + 1;

B[i] = i + 2;

C[i] = i + 3;

D[i] = i + 4;

}

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for five vectors

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_c, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_d, N \* sizeof(double));

cudaMalloc((void\*\*)&dev\_x, N \* sizeof(double));

//blocks = 1; blocksize = 1;

steps = (int)N / (blocks[numB]\*blocksize[numT]);

cudaEventRecord(start, 0); //capture of event start

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_x, X, N \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Ac, A, sizeof(char)\*N, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Bc, B, sizeof(char)\*N, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Cc, C, sizeof(char)\*N, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Dc, D, sizeof(double)\*N, cudaMemcpyHostToDevice); // Launch a kernel on the GPU with one thread for each element.

func\_Kernel << <blocks[numB], blocksize[numT] >> > (dev\_x, steps);

// cudaDeviceSynchronize waits for the kernel to finish

cudaDeviceSynchronize();

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(X, dev\_x, N \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(dev\_x);

cudaEventRecord(stop, 0); //capture of event stop

cudaEventSynchronize(stop); //waits for an event to complete

cudaEventElapsedTime(&elapsedTime, start, stop); //computes the elapsed time between events

srd += elapsedTime;

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaDeviceReset();

amIt--;

}

srd /= numIt;

printf("[blocks: %d, threads: %d]Average elapsed time of GPU computing = %f ms\n", blocks[numB], blocksize[numT], srd);

printf("============================================================\n");

srd = 0;

}

}

}

return 0;

}

10. Программы на CUDA пункта 5 и 7, но в них считается доступной для хранения массивов емкость разделяемой памяти по 16 Кб на блок, в котором размещаются и эффективно считаются данные их массивов.

5:

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#define M 381 //21\*7 + 234

#define N 65536

#define SHARE\_SIZE 1024

\_\_global\_\_ void func\_Kernel(char \*A, char \*B, char \*C, double \*D, double \*X, int s)

{

\_\_shared\_\_ char as[SHARE\_SIZE], char bs[SHARE\_SIZE], char cs[SHARE\_SIZE];

\_\_shared\_\_ double xs[SHARE\_SIZE], double ds[SHARE\_SIZE];

int i,j;

int i\_thread = threadIdx.x;

int idx\_thread = blockIdx.x \* blockDim.x + threadIdx.x; // номер потока

int threadCountGlobal = gridDim.x \* blockDim.x;

for (i = idx\_thread; i < N; i += threadCountGlobal)

{

as[i\_thread] = A[i];

bs[i\_thread] = B[i];

cs[i\_thread] = C[i];

ds[i\_thread] = D[i];

xs[i\_thread] = X[i];

\_\_syncthreads();

for (j = 0; j < 2 \* M; j++)

{

xs[i\_thread] = (double)A[i] \* xs[i\_thread] \* (xs[i\_thread] \* C[i] + B[i]) / D[i];

}

\_\_syncthreads();

X[i] = xs[i\_thread];

}

int main()

{

char A[N], B[N], C[N];

double X[N], D[N];

char \*dev\_a, \*dev\_b, \*dev\_c;

double \*dev\_d, \*dev\_x;

int steps, amIt;

int blocks[3] = { 1, 2, 4 };

int blocksize[4] = { 1,4,32,256 };

float srd = 0;

float elapsedTime;

int i, j;

while (true)

{

printf("Number of iterations: ");

scanf("%d", &amIt);

int numIt = amIt;

for (int numB = 0; numB < 3; numB++)

{

for (int numT = 0; numT < 4; numT++)

{

amIt = numIt;

while (amIt > 0)

{

cudaEvent\_t start, stop; //ids of events

cudaEventCreate(&start); //inits of events

cudaEventCreate(&stop);

for (i = 0; i < N; i++)

{

A[i] = i + 1;

B[i] = i + 2;

C[i] = i + 3;

D[i] = i + 4;

}

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for five vectors

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_c, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_d, N \* sizeof(double));

cudaMalloc((void\*\*)&dev\_x, N \* sizeof(double));

//blocks = 1; blocksize = 1;

steps = (int)N / (blocks[numB]\*blocksize[numT]);

cudaEventRecord(start, 0); //capture of event start

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_a, A, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, B, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_c, C, N \* sizeof(char), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_d, D, N \* sizeof(double), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_x, X, N \* sizeof(double), cudaMemcpyHostToDevice);

// Launch a kernel on the GPU with one thread for each element.

func\_Kernel << <blocks[numB], blocksize[numT] >> > (dev\_a, dev\_b, dev\_c, dev\_d, dev\_x, steps);

// cudaDeviceSynchronize waits for the kernel to finish

cudaDeviceSynchronize();

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(X, dev\_x, N \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

cudaFree(dev\_d);

cudaFree(dev\_x);

cudaEventRecord(stop, 0); //capture of event stop

cudaEventSynchronize(stop); //waits for an event to complete

cudaEventElapsedTime(&elapsedTime, start, stop); //computes the elapsed time between events

srd += elapsedTime;

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaDeviceReset();

amIt--;

}

srd /= numIt;

printf("[blocks: %d, threads: %d]Average elapsed time of GPU computing = %f ms\n", blocks[numB], blocksize[numT], srd);

printf("============================================================\n");

srd = 0;

}

}

}

return 0;

}

7:

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#define M 381 //21\*7 + 234

#define N 65535

#define SHARE\_SIZE 1024

char A[N], B[N], C[N];

double X[N], D[N];

\_\_constant\_\_ char Ac[SHARE\_SIZE], char Bc[SHARE\_SIZE], char Cc[SHARE\_SIZE],

double Dc[SHARE\_SIZE];

\_\_global\_\_ void func\_Kernel(double \*X, int step)

{

\_\_shared\_\_ double xs[SHARE\_SIZE];

int i, j;

int i\_thread = threadIdx.x; //номер потока в задаче

int idx\_thread = blockIdx.x \* blockDim.x + threadIdx.x; // номер потока

int threadCountGlobal = gridDim.x \* blockDim.x;

for (i = idx\_thread; i < SHARE\_SIZE; i += threadCountGlobal)

{

xs[i\_thread] = X[step+i];

\_\_syncthreads();

for (j = 0; j < 2 \* M; j++)

{

xs[i\_thread] = (double)Ac[i] \* xs[i\_thread] \* (xs[i\_thread] \* Cc[i] + Bc[i]) / Dc[i];

}

\_\_syncthreads();

X[step+i] = xs[i\_thread];

}

}

int main()

{

double \*dev\_x;

int steps, amIt;

int blocks[3] = { 1, 2, 4 };

int blocksize[4] = { 1,4,32,256 };

float srd = 0;

float elapsedTime;

int i, j;

while (true)

{

printf("Number of iterations: ");

scanf("%d", &amIt);

int numIt = amIt;

for (int numB = 0; numB < 3; numB++)

{

for (int numT = 0; numT < 4; numT++)

{

amIt = numIt;

while (amIt > 0)

{

cudaEvent\_t start, stop; //ids of events

cudaEventCreate(&start); //inits of events

cudaEventCreate(&stop);

for (i = 0; i < N; i++)

{

A[i] = i + 1;

B[i] = i + 2;

C[i] = i + 3;

D[i] = i + 4;

}

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for five vectors

cudaMalloc((void\*\*)&dev\_a, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_b, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_c, N \* sizeof(char));

cudaMalloc((void\*\*)&dev\_d, N \* sizeof(double));

cudaMalloc((void\*\*)&dev\_x, N \* sizeof(double));

//blocks = 1; blocksize = 1;

steps = (int)N / (blocks[numB]\*blocksize[numT]);

cudaEventRecord(start, 0); //capture of event start

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_x, X, N \* sizeof(double), cudaMemcpyHostToDevice);

char AforConst[SHARE\_SIZE], BforConst[SHARE\_SIZE], CforConst[SHARE\_SIZE];

double DforConst[SHARE\_SIZE];

for (int stepInArr = 0; stepInArr < N; stepInArr += SHARE\_SIZE)

{

for (i = 0; i < SHARE\_SIZE; i++)

{

AforConst[i] = A[stepInArr + i];

BforConst[i] = B[stepInArr + i];

CforConst[i] = C[stepInArr + i];

DforConst[i] = D[stepInArr + i];

}

cudaMemcpyToSymbol(Ac, AforConst, sizeof(char)\*SHARE\_SIZE, 0, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Bc, BforConst, sizeof(char)\*SHARE\_SIZE, 0, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Cc, CforConst, sizeof(char)\*SHARE\_SIZE, 0, cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(Dc, DforConst, sizeof(double)\*SHARE\_SIZE, 0, cudaMemcpyHostToDevice);

// Launch a kernel on the GPU with one thread for each element.

func\_Kernel << <blocks[numB], blocksize[numT] >> > (dev\_x, stepInArr);

// cudaDeviceSynchronize waits for the kernel to finish

cudaDeviceSynchronize();

}

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(X, dev\_x, N \* sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(dev\_x);

cudaEventRecord(stop, 0); //capture of event stop

cudaEventSynchronize(stop); //waits for an event to complete

cudaEventElapsedTime(&elapsedTime, start, stop); //computes the elapsed time between events

srd += elapsedTime;

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaDeviceReset();

amIt--;

}

srd /= numIt;

printf("[blocks: %d, threads: %d]Average elapsed time of GPU computing = %f ms\n", blocks[numB], blocksize[numT], srd);

printf("============================================================\n");

srd = 0;

}

}

}

return 0;

}

*Таблица 1. Время выполнения программ пунктов 3, 5, 7 в мс*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Времена выполнения вычислений на GPU при N = 512 | | | | | | | | | | | | |
|  | Число блоков / число потоков в блоке | | | | | | | | | | | |
| 1/1 | 1/4 | 1/32 | 1/256 | 2/1 | 2/4 | 2/32 | 2/256 | 4/1 | 4/4 | 4/32 | 4/256 |
| Global | 281.578 | 70.779 | 10.325 | 2.045 | 139.252 | 35.258 | 5.204 | 1.206 | 69.206 | 17.897 | 2.737 | 0.413 |
| Shared | 234.457 | 61.063 | 9.007 | 2.281 | 116.543 | 30.722 | 4.743 | 1.311 | 60.024 | 16.175 | 2.866 | 1.313 |
| Shared and constant | 259.241 | 62.104 | 8.098 | 1.720 | 123.654 | 31.196 | 4.066 | 1.021 | 59.143 | 15.948 | 2.773 | 1.195 |

*Таблица 2. Время выполнения программ пункта 10 в мс*

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Времена выполнения вычислений на GPU при N = 65536 | | | | | | | | | | | | |
|  | Число блоков / число потоков в блоке | | | | | | | | | | | |
| 1/1 | 1/4 | 1/32 | 1/256 | 2/1 | 2/4 | 2/32 | 2/256 | 4/1 | 4/4 | 4/32 | 4/256 |
| Global | 35371.4 | 8898.0 | 1259.8 | 158.0 | 18337.8 | 4523.3 | 635.9 | 93.6 | 8970.9 | 2239.8 | 316.8 | 48.2 |
| Shared | 29974.1 | 6929.3 | 882.4 | 149.8 | 13987.8 | 3501.2 | 432.6 | 75.3 | 6941.9 | 1741.0 | 216.7 | 38.2 |
| Shared and constant | 27659.1 | 6869.0 | 888.0 | 162.8 | 13815.6 | 3452.1 | 442.9 | 83.9 | 6924.2 | 1744.7 | 230.8 | 44.9 |

*Таблица 3. Сравнительная таблица времен, пункт 12*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | число блоков / число потоков в блоке | | | | | | | | | | | |
| N | Память | 1/1 | 1/4 | 1/32 | 1/256 | 2/1 | 2/4 | 2/32 | 2/256 | 4/1 | 4/4 | 4/32 | 4/256 |
| 512 | Glob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Shar | 1.20098 | 1.15911 | 1.14633 | 0.89654 | 1.19486 | 1.14765 | 1.09720 | 0.91991 | 1.15298 | 1.10646 | 0.9550 | 0.31455 |
| S&C | 1.08616 | 1.13968 | 1.27501 | 1.18895 | 1.12614 | 1.13021 | 1.27988 | 1.18119 | 1.17015 | 1.12221 | 0.98702 | 0.34561 |
| 65536 | Glob | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Shar | 1.18007 | 1.28411 | 1.42770 | 1.05474 | 1.31099 | 1.29193 | 1.46995 | 1.24303 | 1.29228 | 1.28650 | 1.46193 | 1.26178 |
| S&C | 1.27883 | 1.29539 | 1.41869 | 0.97052 | 1.32733 | 1.31030 | 1.43576 | 1.11561 | 1.29559 | 1.28377 | 1.37262 | 1.07350 |