Выполнила: Белоусова Е., ИП-911

Задача

Лабораторная: написать программу для сложения двух векторов, выполняемую на GPU (использовать программные конструкции из примера в первой лекции). Построить графики зависимости времени вычисления от размерности векторов N в диапазоне от 1<<10 до 1<<23, при различных конфигурациях нитей. Оформить результаты в форме отчета (электронные документы, без распечатывания).

Описание работы программы

Узнаем характеристики девайса для того, чтобы правильно сконфигурировать нити. Информация о возможностях GPU возвращается в виде структуры *cudaDeviceProp,* используется функция cudaGetDeviceProperties.

```
sonya@sonya-S551LB:~/cuda/lab1$ ./11.o
Found 1 devices
Device 0
Compute capability     : 3.5
Name                   : NVIDIA GeForce GT 740M
Total Global Memory    : 2101739520
Shared memory per block: 49152
Registers per block    : 65536
Warp size              : 32
Max threads per block  : 1024
Total constant memory  : 65536
Max Thread Dimensions: 1024 x 1024 x 64
Max Block Dimensions: 2147483647 x 65535 x 65535
sonya@sonya-S551LB:~/cuda/lab1$
```

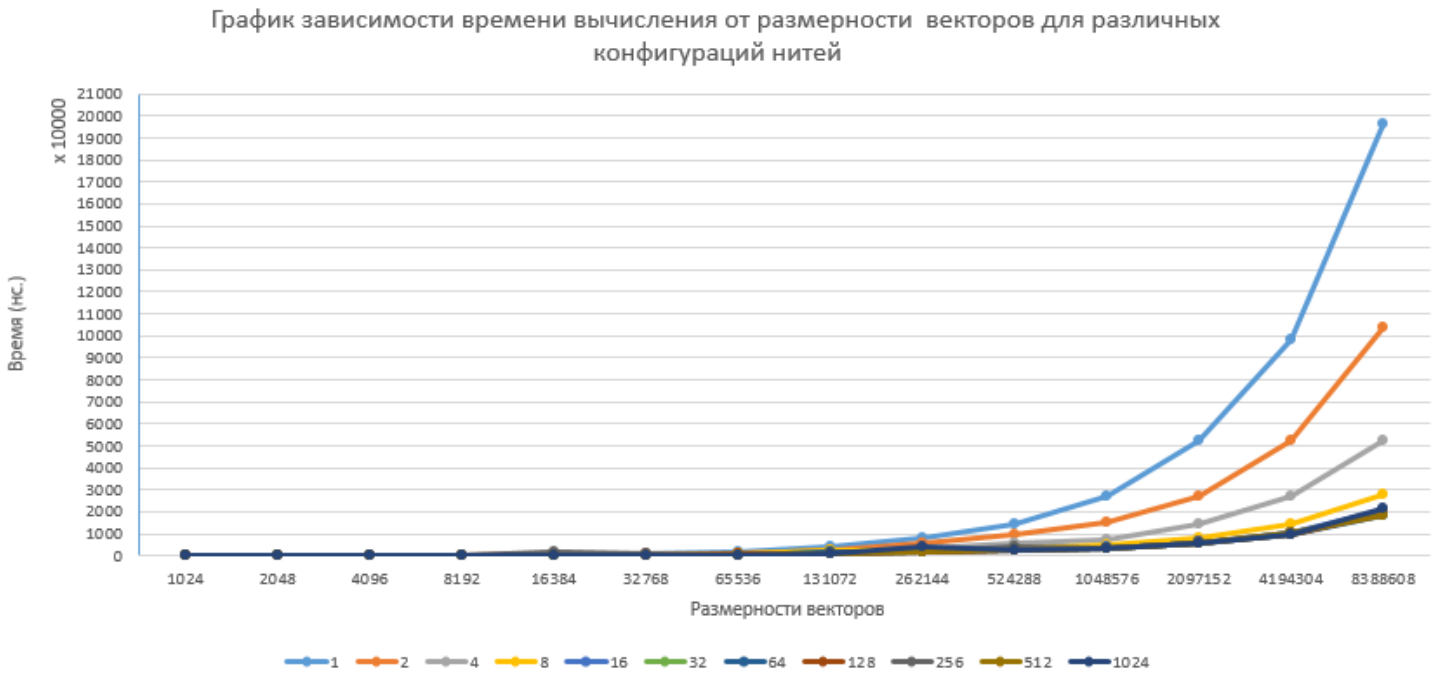
Максимальное количество нитей в блоке – 1024.

Функция addVec() есть ядро (атрибут __global__) и будет выполняться на GPU по одной независимой нити для каждого набора элементов. Нить имеет собственные координаты, поэтому функция начинается с определения глобального индекса массива, зависящего от координат нити. Каждая нить складывает элементы векторов и завершает работу.

В основной функции в циклах перебираются различные варианты конфигураций нити и размеров векторов. После выбора очередных значений программа выделяет память на CPU, GPU, заполняет вектора на CPU и копирует их на GPU. Вызывается ядро с заданной конфигурацией. Далее ожидается завершение работы ядра. После завершения результат копируется в память CPU и освобождается выделенная память.

size/threads	1	2	4	8	16	32	64	128	256	512	1024
1024	327114	48969	40357	27584	40492	26460	29565	42874	37850	26968	36354
2048	58666	41452	27332	25410	32311	21005	44718	42188	23204	35874	38833
4096	89886	57041	42749	36208	33286	31427	30571	50036	30170	29999	32078
8192	170032	106063	95083	64274	57126	54596	52869	50132	52562	52838	52996
16384	474103	332667	280128	263378	236740	240759	229933	845516	1853417	228192	231730
32768	1051242	626840	436793	341060	307014	304276	300693	298879	1073998	294305	300514
65536	1931108	1139982	941377	552303	505292	482782	473098	774326	509452	474577	471633
131072	4055834	2377933	1689225	2404038	1180217	1696653	1442975	1132962	1132516	1183260	1129509
262144	8377232	5346196	2977293	2232869	1977567	1917550	1928605	1847622	1775186	1910751	4302972
524288	14312993	9512996	5390306	3041312	2582666	3036825	2536301	2427756	4433493	2675656	2451889
1048576	27152578	14878311	7620244	4644284	3708688	3522974	3503449	3438507	3530032	3486151	3506648
2097152	52043991	26825862	14341057	8182659	6074450	5700498	5678020	5675252	6073126	5685814	6064489
4194304	98258423	51966425	26910558	14373864	10792766	9981283	9971219	9986169	10159270	10322035	9960786
8388608	196381164	103764854	52559947	27823384	20686836	19028969	18620251	18986192	19020882	18535771	21253221

Для измерения времени используется std::chrono.



Листинг

//1.cu

```
#include <bits/stdc++.h>

#include <stdio.h>

#include <malloc.h>

#include <time.h>


__global__ void addVec(int* a, int* b, int* c, int N){

        int i = threadIdx.x + blockIdx.x * blockDim.x;

        if(i >= N) return;

        c[i] = a[i] + b[i];

}


void addVectors(int N, int j)

{

        int *a, *b, *c, *d, *cuA, *cuB, *cuC;

        //timespec start, end;

        a = (int*)calloc(N, sizeof(int));

        b = (int*)calloc(N, sizeof(int));

        c = (int*)calloc(N, sizeof(int));

        d = (int*)calloc(N, sizeof(int));

        cudaMalloc((void**)&cuA, N*sizeof(int));
```

```
cudaMalloc((void**)&cuB, N*sizeof(int));
cudaMalloc((void**)&cuC, N*sizeof(int));

for(int k = 0; k < N; k++)
{
	a[k] = k;
	b[k] = k;
	c[k] = 0;
	d[k] = k + k;
}

cudaMemcpy(cuA, a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(cuB, b, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(cuC, c, N * sizeof(int), cudaMemcpyHostToDevice);
//clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);
auto start = std::chrono::high_resolution_clock::now();
addVec << <N / j, j >> > (cuA, cuB, cuC, N);
cudaDeviceSynchronize();
auto elapsed = std::chrono::high_resolution_clock::now() - start;
//clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);
//double time = (double)(end.tv_nsec-start.tv_nsec);
//printf("%0.9f\n", time);
cudaMemcpy(c, cuC, N * sizeof(int), cudaMemcpyDeviceToHost);

		/*for(int k = 0; k < N; k++)
		{
			if(c[k] != d[k])
			{
				printf("Wrong!\n");
				free(a);
				free(b);
				free(c);
				cudaFree(cuA);
				cudaFree(cuB);
				cudaFree(cuC);
				return;
			}
		}
```

```
                                */
        std::cout << N << " size  \t\t"<< N/j << " blocks \t\t" <<
std::chrono::duration_cast<std::chrono::nanoseconds>(elapsed).count() << " ns" <<std::endl;
        free(a);
        free(b);
        free(c);
        cudaFree(cuA);
        cudaFree(cuB);
        cudaFree(cuC);
}

void changeThreads()
{
        int N = 0;
        for(int j = 1<<0; j <= 1<<10; j<<=1)
        {
                std::cout << "Threads per block " << j << std::endl;
                for(int i = 10; i <= 23; i++)
                {
                        N = 1 << i;
                        addVectors(N, j);
                }
        }
}

int main(int argc, char* argv[])
{
        if (argc == 1)
        {
                changeThreads();
        }
        else if(argc == 3)
        {
                int N = std::stoi(argv[1]);
                int threads = std::stoi(argv[2]);
                addVectors(N, threads);
        }

        return 0;
```

```
}
```

//11.cu

```
#include <stdio.h>

int main ( int argc, char *  argv [] )
{
   int          deviceCount;
   cudaDeviceProp devProp;

   cudaGetDeviceCount ( &deviceCount );

   printf ( "Found %d devices\n", deviceCount );

   for ( int device = 0; device < deviceCount; device++ )
   {
      cudaGetDeviceProperties ( &devProp, device );

      printf ( "Device %d\n", device );
      printf ( "Compute capability     : %d.%d\n", devProp.major, devProp.minor );
      printf ( "Name                   : %s\n", devProp.name );
      printf ( "Total Global Memory    : %d\n", devProp.totalGlobalMem );
      printf ( "Shared memory per block: %d\n", devProp.sharedMemPerBlock );
      printf ( "Registers per block    : %d\n", devProp.regsPerBlock );
      printf ( "Warp size              : %d\n", devProp.warpSize );
      printf ( "Max threads per block  : %d\n", devProp.maxThreadsPerBlock );
      printf ( "Total constant memory  : %d\n", devProp.totalConstMem );
      printf("Max Thread Dimensions: %i x %i x %i\n", devProp.maxThreadsDim[0], devProp.maxThreadsDim[1],
devProp.maxThreadsDim[2]);
      printf("Max Block Dimensions: %i x %i x %i\n", devProp.maxGridSize[0], devProp.maxGridSize[1], devProp.maxGridSize[2]);
   }

   return 0;
}
```