Metody Programowania Równoległego
Cuda - VectorAdd

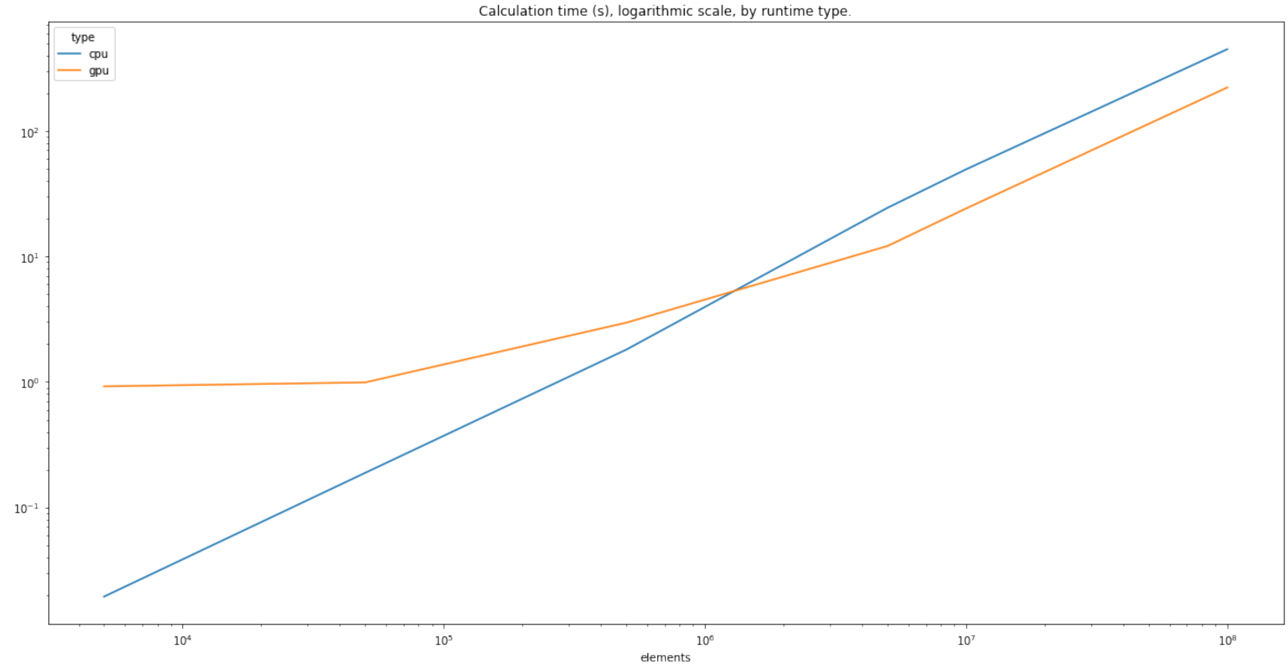
Maciej Trątnowiecki

AGH, Semestr Letni, 2022

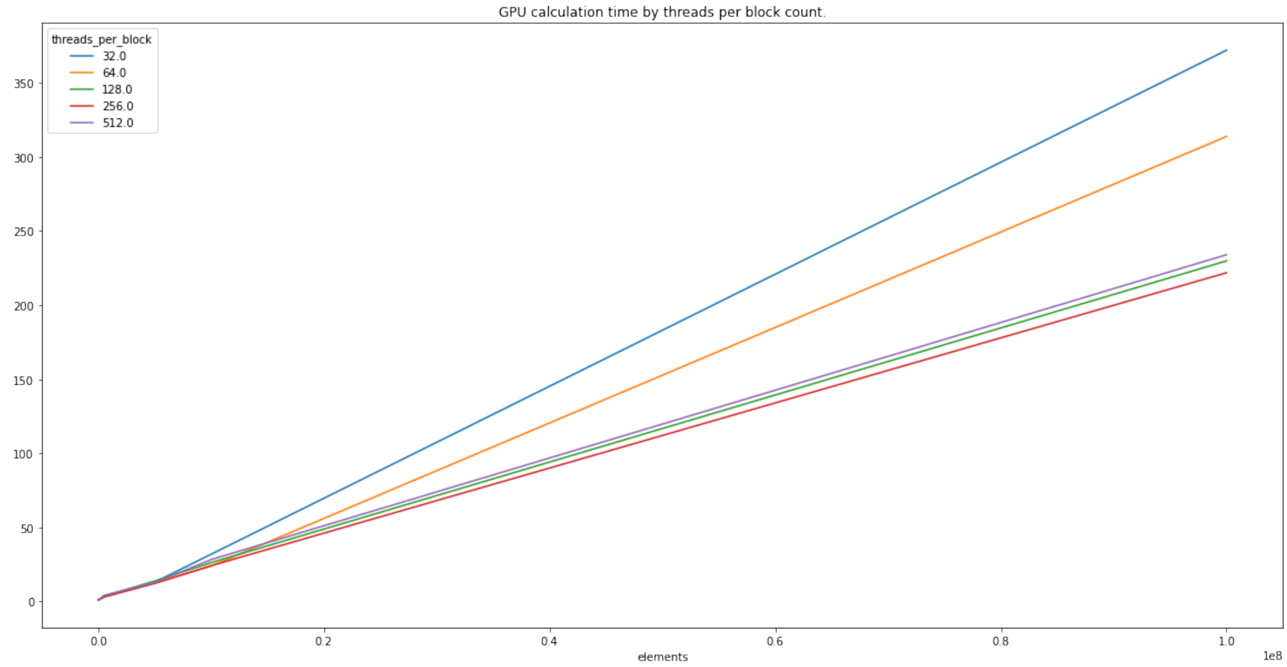
1 Pomiary i wyniki

W ramach laboratorium przygotowałem odpowiednio zmodyfikowany kod programu implementującego dodawanie wektorów (opierając się na przygotowanym przez nvidię programie przykładowym) i dokonałem pomiarów czasu wykonania. Wyniki zebrałem w poniższej tabeli oraz zilustrowałem za pomocą poniższych wykresów. Analizując wykresy widzimy, że dla odpowiednio małych rozmiarów wektorów czas wykonania programu może być mniejszy bez użycia akceleracji za pomocą karty graficznej. Wynika to z narzutu czasowego generowanego przez obsługę pamięci karty graficznej i kopiowanie danych. Obserwując wykres wpływu zmiany ilości wątków na blok dla obliczeń wykonywanych z wykorzystaniem karty graficznej zauważyć możemy, że ma on ten sam kształt (w skali logarytmicznej) dla każdego rozmiaru dodawanych wektorów.

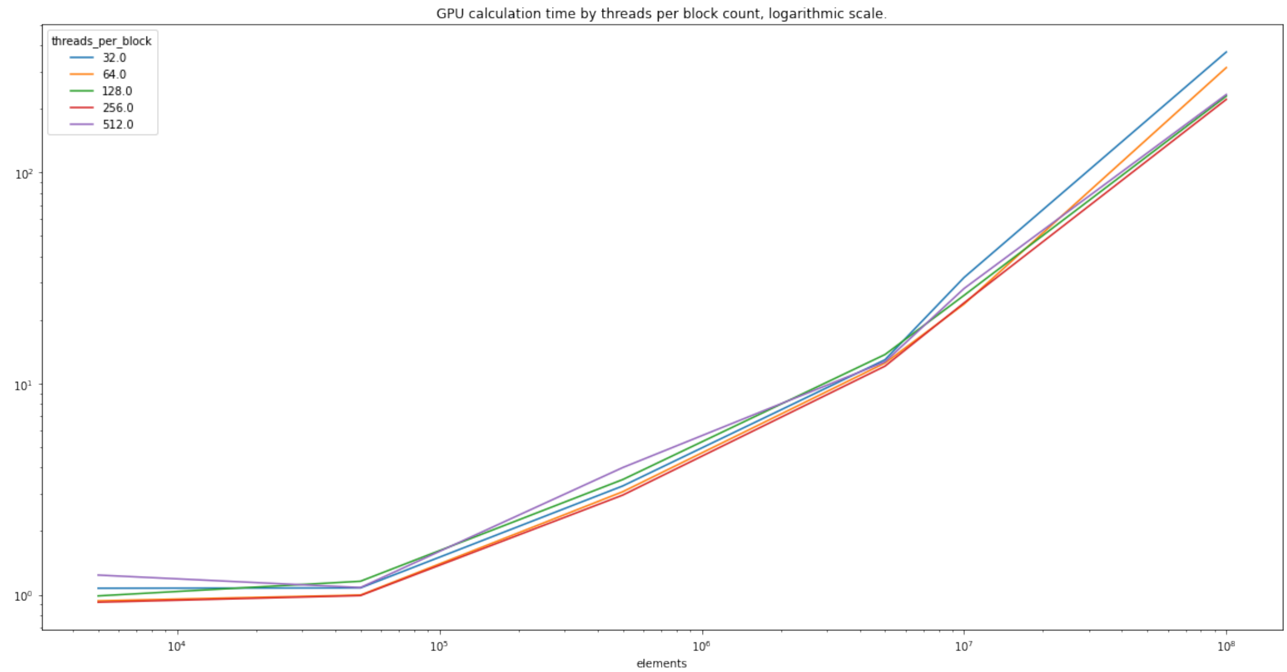
	elements	time	type	threads_per_block	blocks_per_grid
0	5000	0.019456	cpu	nan	nan
1	5000	1.073056	gpu	32.000000	157.000000
2	5000	0.935136	gpu	64.000000	79.000000
3	5000	0.986880	gpu	128.000000	40.000000
4	5000	0.921024	gpu	256.000000	20.000000
5	5000	1.238368	gpu	512.000000	10.000000
6	50000	0.188416	cpu	nan	nan
7	50000	1.054368	gpu	32.000000	1563.000000
8	50000	1.076416	gpu	32.000000	1563.000000
9	50000	0.995840	gpu	64.000000	782.000000
10	50000	1.157600	gpu	128.000000	391.000000
11	50000	0.990336	gpu	256.000000	196.000000
12	50000	1.080864	gpu	512.000000	98.000000
13	500000	1.808384	cpu	nan	nan
14	500000	3.273312	gpu	32.000000	15625.000000
15	500000	3.073728	gpu	64.000000	7813.000000
16	500000	3.510112	gpu	128.000000	3907.000000
17	500000	2.962688	gpu	256.000000	1954.000000
18	500000	4.000768	gpu	512.000000	977.000000
19	5000000	24.389631	cpu	nan	nan
20	5000000	12.969472	gpu	32.000000	156250.000000
21	5000000	12.517504	gpu	64.000000	78125.000000
22	5000000	13.730528	gpu	128.000000	39063.000000
23	5000000	12.097664	gpu	256.000000	19532.000000
24	5000000	12.685152	gpu	512.000000	9766.000000
25	10000000	49.402882	cpu	nan	nan
26	10000000	31.789600	gpu	32.000000	312500.000000
27	10000000	23.785185	gpu	64.000000	156250.000000
28	10000000	26.165279	gpu	128.000000	78125.000000
29	10000000	24.098751	gpu	256.000000	39063.000000
30	10000000	28.189184	gpu	512.000000	19532.000000
31	100000000	447.545349	cpu	nan	nan
32	100000000	372.130005	gpu	32.000000	3125000.000000
33	100000000	313.858063	gpu	64.000000	1562500.000000
34	100000000	229.868637	gpu	128.000000	781250.000000
35	100000000	221.865829	gpu	256.000000	390625.000000
36	100000000	234.037064	gpu	512.000000	195313.000000



Rysunek 1: Czas wykonania programu (w sekundach) w zależności od jednostki wykonującej obliczenia, skala logarytmiczna.



Rysunek 2: Czas wykonania programu (w sekundach) w zależności od jednostki wykonującej obliczenia, skala logarytmiczna.



Rysunek 3: Czas wykonania programu (w sekundach) w zależności od jednostki wykonującej obliczenia, skala logarytmiczna.

2 Kod programu

```
/**
 * Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
 *
 * Please refer to the NVIDIA end user license agreement (EULA) associated
 * with this source code for terms and conditions that govern your use of
 * this software. Any use, reproduction, disclosure, or distribution of
 * this software and related documentation outside the terms of the EULA
 * is strictly prohibited.
 *
 */

/**
 * Vector addition: C = A + B.
 *
 * This sample is a very basic sample that implements element by element
 * vector addition. It is the same as the sample illustrating Chapter 2
 * of the programming guide with some additions like error checking.
 */

#include <stdio.h>

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include "gputimer.h"
#define VERBOSE false

/**
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C. The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

void run_on_gpu(size_t size, int numElements, float *h_A, float *h_B, float *h_C, int threadsPerBlock, int blocksPerGrid) {
    // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    // Allocate the device input vector A
    float *d_A = NULL;
    err = cudaMalloc((void **)&d_A, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Allocate the device input vector B
    float *d_B = NULL;
    err = cudaMalloc((void **)&d_B, size);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Allocate the device output vector C
    float *d_C = NULL;
    err = cudaMalloc((void **)&d_C, size);
```

```
    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy the host input vectors A and B in host memory to the device input vectors in
    // device memory
    #if (VERBOSE == true)
        printf("Copy input data from the host memory to the CUDA device\n");
    #endif
    err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Launch the Vector Add CUDA Kernel
    #if (VERBOSE == true)
        printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid,
    threadsPerBlock);
    #endif
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
    err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Copy the device result vector in device memory to the host result vector
    // in host memory.
    #if (VERBOSE == true)
        printf("Copy output data from the CUDA device to the host memory\n");
    #endif
    err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy vector C from device to host (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    // Free device global memory
    err = cudaFree(d_A);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector A (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_B);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector B (error code %s)!\n",
    cudaGetErrorString(err));
```

```
        exit(EXIT_FAILURE);
    }

    err = cudaFree(d_C);

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to free device vector C (error code %s)!\n",
    cudaGetErrorString(err));
        exit(EXIT_FAILURE);
    }
}

void run_on_cpu(size_t size, int numElements, float *h_A, float *h_B, float *h_C) {
    for (int i = 0; i < numElements; ++i)
        h_C[i] = h_A[i] + h_B[i];
}


/**
 * Host main routine
 */
int
main(int argc, char *argv[]){
    // Print the vector length to be used, and compute its size
    int numElements = 0;
    if (argc > 1) {
        numElements = atoi(argv[1]);
        #if (VERBOSE == true)
            printf("Number of elements: %d\n", numElements);
        #endif
    } else {
        printf("Please pass number of elements as command line argument.\n");
        exit(EXIT_FAILURE);
    }

    bool cpu_only = false;

    if (argc > 2 && strcmp(argv[2], "true") == 0){
        cpu_only = true;
        #if (VERBOSE == true)
            printf("Running on CPU only!\n");
        #endif
    }

    size_t size = numElements * sizeof(float);
    // printf("[Vector addition of %d elements]\n", numElements);

    // Allocate the host input vector A
    float *h_A = (float *)malloc(size);

    // Allocate the host input vector B
    float *h_B = (float *)malloc(size);

    // Allocate the host output vector C
    float *h_C = (float *)malloc(size);

    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL || h_C == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    int threadsPerBlock = 256;
    if(argc > 3)
        threadsPerBlock = atoi(argv[3]);

    int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
```

```
    if(argc > 4)
        blocksPerGrid = atoi(argv[4]);

    GpuTimer timer;
    timer.Start();

    if(cpu_only)
        run_on_cpu(size, numElements, h_A, h_B, h_C);
    else
        run_on_gpu(size, numElements, h_A, h_B, h_C, threadsPerBlock, blocksPerGrid);

    timer.Stop();
    #if (VERBOSE == true)
        printf("Elapsed: %f\n", timer.Elapsed());
    #endif

    // Verify that the result vector is correct
    for (int i = 0; i < numElements; ++i)
    {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5) {
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    #if (VERBOSE == true)
        printf("Test PASSED\n");
    #endif

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    #if (VERBOSE == true)
        printf("Done\n");
    #endif

    // printf("elements, time, type, threads_per_block, blocks_per_grid\n");
    printf("%d, %f, %s, %d, %d\n", numElements, timer.Elapsed(),(cpu_only ? "cpu" : "gpu"),
    threadsPerBlock, blocksPerGrid);
    return 0;
}
```

../0_Simple/vectorAdd/vectorAdd.cu