UNIVERSITÀ DEGLI STUDI DI PERUGIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

Terza esercitazione:

CALCOLO DISTRIBUITO E PARALLELO

MOLTIPLICAZIONI DI MATRICI, CONFRONTANDO LE PRESTAZIONI DI CPU E GPU

Studente: Manuel Severi

INDICE

1 | INTRODUZIONE

Con la presente relazione si vuole dimostrare come, per la moltiplicazione tra matrici di grandi dimensioni, il calcolo parallelo sfruttando la gpu sia migliore che il calcolo utilizzando la classica cpu.

DEFINIZIONE GPGPU

Con la sigla GPGPU(general-purpose computing on graphics processing units letteralmente "calcolo a scopo generale su unità di elaborazione grafica"), si intende l'uso di un'unità di elaborazione grafica (GPU) per scopi diversi dal tradizionale utilizzo nella grafica computerizzata.
Il GPGPU viene impiegato per elaborazioni estremamente esigenti in termini di potenza di elaborazione, e per le quali le tradizionali architetture di CPU non hanno una capacità di elaborazione sufficiente. Per loro natura tali elaborazioni sono di tipo altamente parallelo e in grado quindi di beneficiare ampiamente dell'architettura tipica delle GPU. A tale caratteristica intrinseca a partire dal 2007 si è aggiunta l'estrema programmabilità offerta da varie soluzioni commerciali, che al succedersi delle generazioni ha aumentato non solo la propria potenza elaborativa ma anche la propria versatilità.

Le applicazioni che sono in grado di avvantaggiarsi significativamente della potenza di calcolo delle moderne GPU sono solo una ristretta parte dell'intero panorama software, in quanto per sfruttare le caratteristiche di tali architetture è necessaria un'elevata parallelizzazione del codice, una

caratteristica tipica di alcuni problemi scientifici ma non tutti; alcuni settori scientifici che beneficiano del GPGPU sono:

- Fisica: Equazioni di Eulero, Equazioni di Navier-Stokes, meccanica quantistica, Metodi reticolari di Boltzmann.
- Chimica e biologia: Simulazioni Monte-Carlo, bioinformatica, modellazione proteica.
- Matematica: Trasformate di Fourier, crittografia.
- Astronomia e astrofisica: radioastronomia, decodifica segnali spaziali, compensazione immagini.
- Medicina: Immagini mediche, supporto alle decisioni diagnostiche.
- Sicurezza informatica: crittografia, MD6.
- Criptovalute: utilizzo delle GPU per il mining.
- Calcolo distribuito: accelerazione delle applicazioni BOINC attraverso le GPU consumer.

Diversi programmi fanno uso del GPGPU, tra cui: Adobe Photoshop, Autodesk Maya, GIMP, Autodesk Autocad, Blender, Matlab, LibreOffice, OpenMM.

2 | STRUMENTI UTILIZZATI

Il seguente progetto è stato realizzato sfruttando un a scheda grafica Invidia e un processore intel core i5 di 6th generazione.

Microsoft Visual Studio

Si è scelto di utilizzare Visual Studio 2019 per la realizzazione del progetto perché questo IDE sviluppato da Microsoft è gratuito nella sua versione comunity. Visual Studio integra la tecnologia IntelliSense che permette di correggere eventuali errori sintattici, e anche alcuni logici, senza compilare l'applicazione, possiede un debugger interno per il rilevamento e la correzione degli errori logici nel codice in runtime e fornisce diversi strumenti per l'analisi delle prestazioni. Visual Studio consente di reperire e installare template e componenti aggiuntivi di terze parti dal Web per ottenere ulteriori funzionalità rispetto a quelle già presenti all'interno del suo codice.

C++

Nato nel 1983 dall'ingegno di Bjarne Stroustrup, allora ricercatore presso AT&T, C++ è tra i primi 5 linguaggi più utilizzati al mondo. I campi di applicazione sono i più svariati: dal gaming alle applicazioni real-time, dai

componenti per sistemi operativi ai software di grafica e musica, dalle app per cellulari ai sistemi per supercomputer. Praticamente, C++ è ovunque. Negli anni si sono avvicendate diverse versioni di questo linguaggio di programmazione, introducendo via via diverse modifiche ed alcune caratteristiche che ne fanno ancora un linguaggio assolutamente importante e moderno. Il C++ è un linguaggio pensato per la programmazione orientata agli oggetti, che rende questo linguaggio una piattaforma ideale per realizzare progetti di grosse dimensioni, favorendo l'astrazione dei problemi. Ciò ci consente di sviluppare software seguendo i più moderni pattern di progettazione: esistono moltissime librerie già pronte e riutilizzabili, integrabili con i progetti grazie ad opportune strategie e design pattern come Adapter o Facade. C++ mette insieme l'espressività dei linguaggi orientati agli oggetti con l'efficienza e la compattezza del linguaggio C dal quale discende e di cui eredita potenza, flessibilità e la possibilità di programmare a basso livello. Questo ci permette di scrivere programmi di tutte le dimensioni, utilizzando ricchi stack di framework per dialogare ad alto livello, oppure lavorare a basso livello fino ad arrivare all'intergrazione di istruzioni assembly.

CUDA

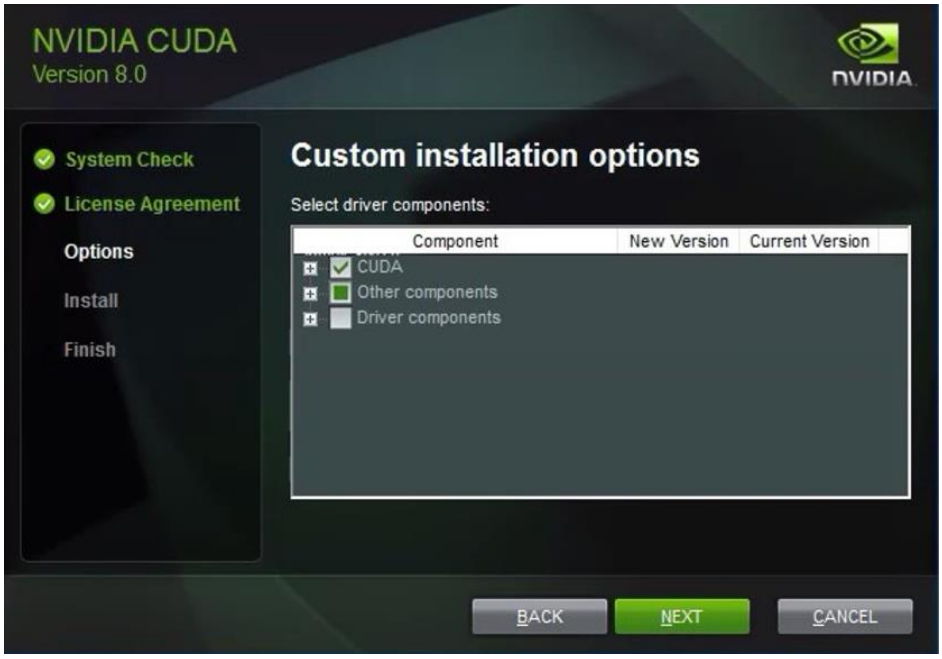
CUDA (acronimo di Compute Unified Device Architecture) è un'architettura hardware per l'elaborazione parallela creata da NVIDIA. Tramite l'ambiente di sviluppo per CUDA, i programmatori di software

possono scrivere applicazioni capaci di eseguire calcolo parallelo sulle GPU delle schede video NVIDIA. I linguaggi di programmazione disponibili nell'ambiente di sviluppo CUDA sono estensioni dei linguaggi più diffusi per scrivere programmi. Il principale è 'CUDA-C' (C con estensioni NVIDIA), altri sono estensioni di Python, Fortran, Java e MATLAB. Programmi che sfruttano l'architettura CUDA possono essere scritti anche utilizzando le librerie software OpenCL e DirectCompute. CUDA da accesso agli sviluppatori ad un set di istruzioni native per il calcolo parallelo di elementi delle GPU CUDA. Usando CUDA, le ultime GPU Nvidia diventano in effetti architetture aperte come le CPU. Diversamente dalle CPU, le GPU hanno un'architettura parallela con diversi core, ognuno capace di eseguire centinaia di processi simultaneamente: se un'applicazione è adatta per questo tipo di architettura, la GPU può offrire grandi prestazioni e benefici. Questo approccio alla risoluzione dei problemi è noto come GPGPU.
CUDA ha parecchi vantaggi rispetto alle tradizionali tecniche di calcolo sulle GPU che usano le API grafiche tra cui:

- l codice può essere letto da posizioni arbitrarie in memoria.
- Memoria condivisa: CUDA espone una regione di 16kB di grandezza che può essere condivisa velocemente fra i thread. Questa può essere usata come una cache gestita dall'utente, rendendo disponibili grandi larghezze di banda che è possibile usare per strutture texture.
- Letture e scritture veloci, verso e dalla GPU.
- Supporto completo per divisioni intere e operazioni bit-a-bit, tra cui l'accesso a texture intere.

3 | INIZIALIZZAZIONE DEL PROGETTO

L'intento del progetto è quello di creare un programma scritto in C++ che effettui la moltiplicazione tra due matrici di grandi dimensioni. La moltiplicazione sarà effettuata in seriale e in parallelo. Nello specifico in seriale utilizzando la CPU mentre in parallelo utilizzando la GPU. Verranno inoltre riportati i dati delle varie moltiplicazioni (in secondi) per evidenziare le divergenze dei tempi delle stesse operazioni svolte da due componentistiche hardware diverse. Per poter fare ciò, è stato installato CUDA nella macchina scaricando il Toolkit di CUDA all'interno del sito NVIDIA (avendo nel mio computer una scheda video dell'NVIDIA, come precedentemente accennato).



Per controllare se l'installazione è terminata con successo, basta digitare il comando: "*nvcc -V*" all'interno di un prompt dei comandi. La risposta del prompt sarà la verisone installata.

4 CREAZIONE DEL PROGRAMMA PER LA MOLTIPLICAZIONE TRA MATRICI

Una volta che tutte le proprietà sono state inserite correttamente, ho scritto un programma in C++ che svolgesse la moltiplicazione tra due matrici, di dimensioni variabili, in parallelo e in serie.

Il file prende il nome di "*esercitazione.cpp*" ed inizia con queste inclusioni e definizioni:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#ifndef _CUDACC_RTC_
#define _CUDACC_RTC_
#endif // !(_CUDACC_RTC_)
#include <device_functions.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <ctime>

#define BLOCK_SIZE 16
```

Oltre alle librerie "standard" come stdio.h,stdlib.h, assert.h, e ctime poniamo particolare attenzione su:

- cuda.h
- cuda_runtime_api.h
- device_fuctions.h
- device_lauch_parameters.h

queste librerie sopra menzionate sono quelle, che appunto, ci permettono di sfruttare la potenza di calcolo della gpu parallelizzando il codice.

FUNZIONE MAIN

Come in ogni programma in c o c++, l'esecuzione parte dalla funzione main() che nel nostro caso ha la struttura che verrà descritta nel presente capitolo (parte del codice verrà omessa nel seguente capitolo ma consultabile a fine relazione della sezione "codice").
Come primo passo viene chiesto all'utente di scegliere la dimensione delle matrici.

```
int m, n, k;
/* Fixed seed for illustration */
srand(3333);
printf("please type in m n and k\n");
scanf("%d %d %d", &m, &n, &k);
```

Una volta scoperte le dimensioni delle matrici si alloca memoria a sufficienza per eseguire i calcoli.

```
// allocate memory in host RAM, h_cc is used to store CPU result
int* h_a, * h_b, * h_c, * h_cc;
cudaMallocHost((void**)&h_a, sizeof(int) * m * n);
cudaMallocHost((void**)&h_b, sizeof(int) * n * k);
cudaMallocHost((void**)&h_c, sizeof(int) * m * k);
cudaMallocHost((void**)&h_cc, sizeof(int) * m * k);
```

Ora si procede a popolare le due matrici con valori randomici compresi tra 0 e 1024, visualizzando a video il valore ottenuto.

```
// random initialize matrix A
        for (int i = 0; i < m; ++i) {
                for (int j = 0; j < n; ++j) {
                        h_a[i * n + j] = rand() % 1024;
                        printf("[%d][%d]:%d ", i, j, h_a[i * n + j]);
                }
                printf("\n");
        }
        printf("----------------------------------------------------------\n");
        // random initialize matrix B
        for (int i = 0; i < n; ++i) {
                for (int j = 0; j < k; ++j) {
                        h_b[i * k + j] = rand() % 1024;
                        printf("[%d][%d]:%d ", i, j, h_b[i * n + j]);
                }
                printf("\n");
        }
```

Ora viene definita la variabile per contare i millisecondi che trascorrono dalla presa in carico del lavoro fino alla sua conclusione.

```
float gpu_elapsed_time_ms, cpu_elapsed_time_ms;

// some events to count the execution time
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// start to count execution time of GPU version
cudaEventRecord(start, 0);
```

Si fa partire il “timer” e si copiano le due matrici nella memoria della gpu.

```
// copy matrix A and B from host to device memory
cudaMemcpy(d_a, h_a, sizeof(int) * m * n, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeof(int) * n * k, cudaMemcpyHostToDevice);

unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

Si controlla se le matrici hanno la medesima dimensione in caso affermativo il calcolo viene svolto dalla funzione gpu_square_matrix_mult mentre se il test risulta falso, la funzione verrà data impasto a gpu_matrix_mult.

Infine si passa il risultato alla macchina, si libera la memoria della scheda grafica e si stampa il tempo impiegato a svolgere tutto il lavoro utilizzando la GPU.

Ora si passa lo stesso lavoro, con i medesimi dati alla cpu per controllare il tempo che impiega a svolgerlo passando le due matrici alla funzione *cpu_matrix_mult*, nel frattempo ho avviato il timer per controllare i tempi. Una volta ottenuto il risultato e stoppato il timer, visualizzo il tempo di risposta della CPU. Ora eseguo un controllo sui risultati ottenuti dalle due diverse

modalità di calcolo, e mi aspetto che tornino uguali. Per fare quello appena descritto uso un ciclo for come si evince dal codice sottostante.

```
int all_ok = 1;
	for (int i = 0; i < m; ++i)
	{
		for (int j = 0; j < k; ++j)
		{
			printf("[%d][%d]:%d == [%d][%d]:%d, "
				,i, j, h_cc[i*k + j], i, j, h_c[i*k + j]);
			if (h_cc[i * k + j] != h_c[i * k + j])
			{
				all_ok = 0;
			}
		}
		printf("\n");
	}
```

Questo controllo, forse un po' eccessivo, se va a buon fine mi stampa la matrice risultato della moltiplicazione e permette di proseguire il codice del main stampando lo speedup ottenuto utilizzando la GPU al posto della CPU oppure mi informa della presenza di qualche errore. Alla fine del main vado a liberare tutte le memorie che sono andato a occupare durante l'esecuzione del programma tramite le funzioni **cudaFree** e **cudaFreeHost.**

```
// free memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
cudaFreeHost(h_a);
cudaFreeHost(h_b);
cudaFreeHost(h_c);
cudaFreeHost(h_cc);
```

FUNZIONE GPU_SQUARE_MATRIX_MULT E GPU_MATRIX_MULT

come precedentemente ho sviluppato due codici diversi in base alle dimensioni delle matrici per migliorare le prestazioni della gpu. Se le due matrici hanno la medesima dimensione, andrò ad utilizzare gpu_square_matrix_mult che è stata così implementata:

```
__global__ void gpu_square_matrix_mult(int* d_a, int* d_b, int* d_result, int n)
{
	__shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
	__shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

	int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
	int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
	int tmp = 0;
	int idx;

	for (int sub = 0; sub < gridDim.x; ++sub)
	{
		idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
		if (idx >= n * n)
		{
			// n may not divisible by BLOCK_SIZE
			tile_a[threadIdx.y][threadIdx.x] = 0;
		}
		else
		{
			tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
		}

		idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
		if (idx >= n * n)
		{
			tile_b[threadIdx.y][threadIdx.x] = 0;
		}
		else
		{
			tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
		}
		__syncthreads();

		for (int k = 0; k < BLOCK_SIZE; ++k)
		{
			tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
		}
		__syncthreads();
	}
	if (row < n && col < n)
	{
		d_result[row * n + col] = tmp;
	}
}
```

Mentre nel caso in cui la matrice A abbia dimensioni diverse dalla matrice B, naturalmente per come abbiamo implementato l'input della dimensione è già verificata la condizione necessaria per la moltiplicazione delle matrici date due matrici ovvero che per poter effettuare il prodotto righe per colonne A X B il numero delle colonne di A deve essere uguale al numero delle righe di B. Per eseguire la moltiplicazione la funzione gpu_matrix_mult è stata così sviluppata

```
__global__ void gpu_matrix_mult(int* a, int* b, int* c, int m, int n, int k)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        int sum = 0;
        if (col < k && row < m)
        {
                for (int i = 0; i < n; i++)
                {
                        sum += a[row * n + i] * b[i * k + col];
                }
                c[row * k + col] = sum;
        }
}
```

Naturalmente il codice è molto più semplice anche se sicuramente meno performante del precedente, non tanto per come è stato programmato ma per la struttura stessa della matrice. Basta immaginare di moltiplicare una
matrice A (1 x 10000) e una matrice B (10000 x 500)
Si nota facilmente come alcune operazioni non sono parallelizzabili allo stesso momento.

FUNZIONE CPU_MATRIX_MULT

La funzione che verrà descritta nel seguente capitolo è quella che permetta alla cpu di eseguire le moltiplicazioni delle matrici. Si può notare come non vi sono tentativi di parallelizzazione anzi le operazioni vengo svolte una alla volta in sequenza grazie all'utilizzo di due for

```
void cpu_matrix_mult(int* h_a, int* h_b, int* h_result, int m, int n, int k) {
        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < k; ++j)
                {
                        int tmp = 0;
                        for (int h = 0; h < n; ++h)
                        {
                                tmp += h_a[i * n + h] * h_b[h * k + j];
                        }
                        h_result[i * k + j] = tmp;
                }
        }
}
```

Si nota come nel primo for vengono eseguite le moltiplicazioni mentre poi le adizioni e nel secondo for il valore viene poi salvato in una variabile temporanea.

5 | TEST DEL PROGETTO

Andiamo a testare il codice appena descritto. Notiamo che l'IDE ci segnala due errori ma comunque compila ed esegue il programma. Questi "finti errori" sono dati dal fatto, che come detto in precedenza, Visual Studio controlla il nostro codice pensando che è un file C++ ma la sintassi di cuda è leggermente diversa in alcune parti facendo crede al software che è stato commesso un errore. La parte di codice incriminato è:

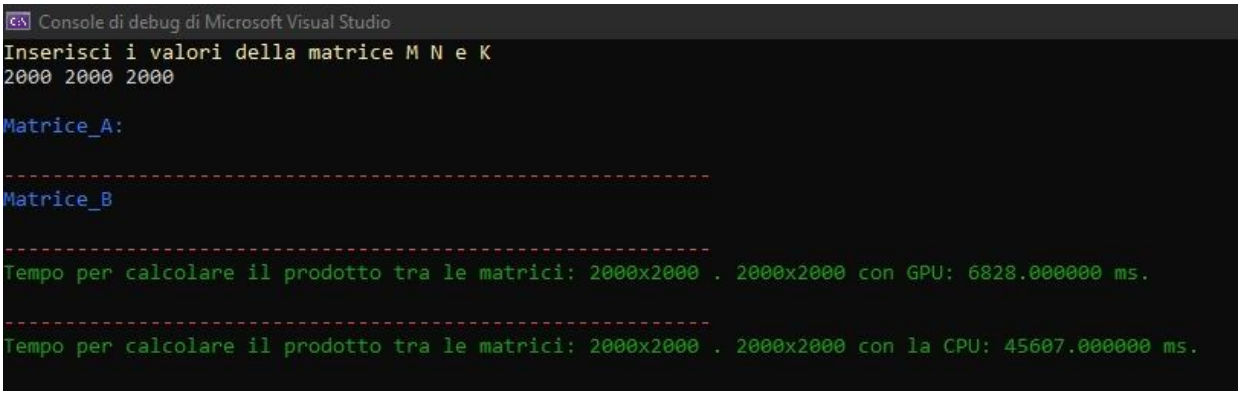
```
gpu_square_matrix_mult << <dimGrid, dimBlock >> > (d_a, d_b, d_c, n);
```

```
gpu_matrix_mult<<< dimGrid, dimBlock>>> (d_a, d_b, d_c, m, n, k);
```

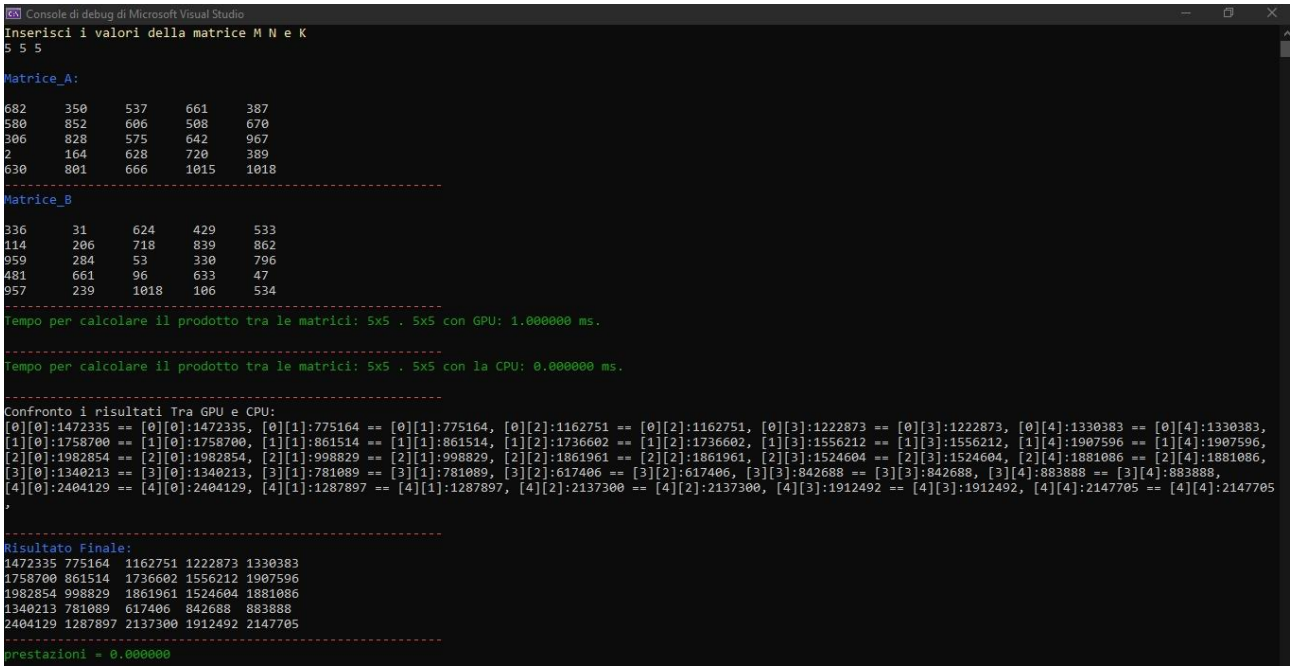
Sì aprirà una finestra come quella che vediamo qua sotto dove inizialmente andremo a specificare la dimensione della variabile.

Ora eseguiamo un test con delle matrici molto grandi (A è una 2000x2000, B è una 2000x2000) e come possiamo notare dalla prossima schermata la differenza di tempo di elaborazione tra la GPU e la CPU è veramente sostanziale. Nella schermata sottostante sono stati nascosti i risultati delle variabili perché in

questo caso la nostra attenzione era focalizzata soprattutto sul tempo impiegato.

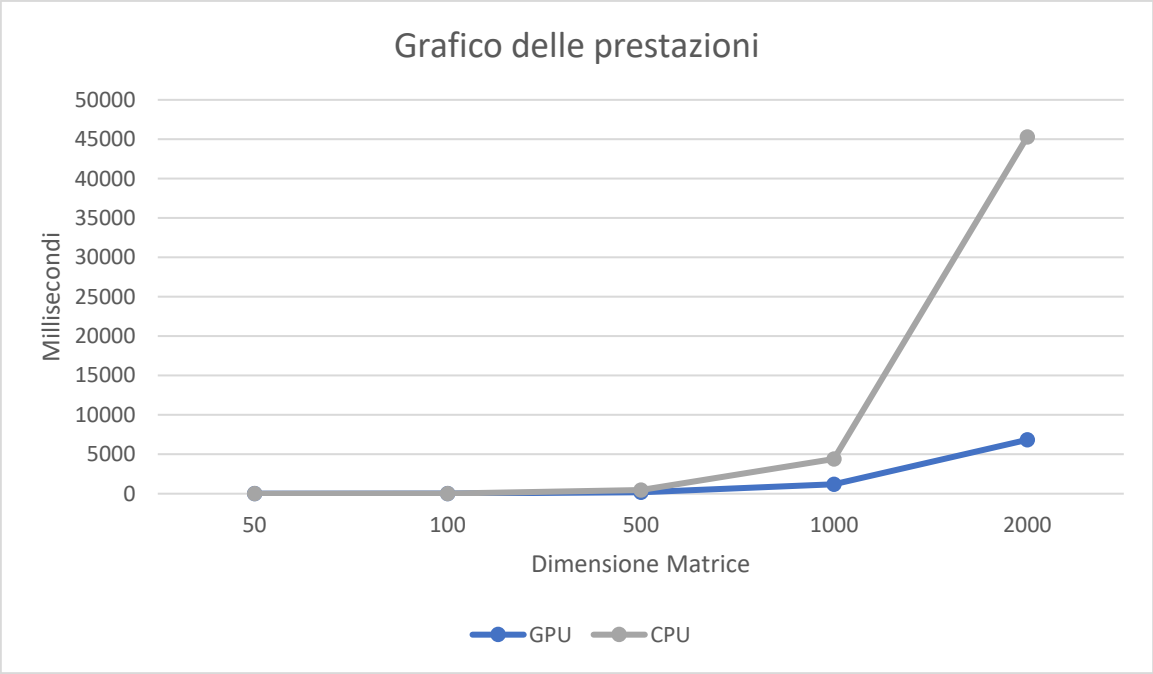


Ora eseguiamo un nuovo test con delle matrici più piccole (A da 5x5, B da 5x5) per controllare se effettivamente il risultato è calcolato in modo corretto.



Se abbiamo programmato tutto in modo coretto, come in questo caso, il risultato è coretto come è dimostrato dalla schermata superiore. Notiamo come in questo caso il tempo della GPU sia maggiore di quello della CPU. Non c'è da preoccuparsi, perché, viste le esigue dimensioni della matrice il device "spreca" più tempo a parallelizzare rispetto a calcolare. È un paradosso

paragonabile ad avere un compito facile e facilmente risolvibile e cercare di suddividerlo in tanti problemi semplicissimi perdendo così tempo nella suddivisione quando bastava poco per risolvere il problema. Se vogliamo anche questo paradosso conferma la nostra tesi che il calcolo parallelo per grandi matrici sia migliore di quello classico. Come possiamo notare dal grafico sottostante che rappresenta i tempi per calcolare le moltiplicazioni di matrici dalle dimensioni uguali, abbiamo un evidente guadagno di tempo solo se le matrici hanno dimensioni elevate altrimenti il guadagno è pressoché nullo.



Ma notiamo come dopo la dimensione 1000 il guadagno di tempo diventa vertiginoso, infatti la linea grigia delle prestazione della cpu schizza verso l'alto mentre quella della GPU ha un incremento decisamente minore.

6 | CONCLUSIONE

La seguente relazione è stata molto utile per mettere alla prova le mie conoscenze della programmazione parallela, acquisite solo in maniera teorica durante la triennale in informatica. Inoltre grazie al seguente progetto è possibile comprendere come qualsiasi problema, che comprenda della grafica (ricordiamo che le immagini sono salvate come delle matrici), è preferibile usare la GPU e la programmazione parallela per sfruttare questo componente hardware che rende al meglio proprio in queste situazioni.

CODICE DEL PROGETTO

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#ifndef _CUDACC_RTC_
#define _CUDACC_RTC_
#endif // !(_CUDACC_RTC_)

#include <device_functions.h>
#include <cuda.h>
#include <cuda_runtime_api.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <ctime>

#define BLOCK_SIZE 16

 __global__ void gpu_matrix_mult(int* a, int* b, int* c, int m, int n, int k)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;
        int sum = 0;
        if (col < k && row < m)
        {
                for (int i = 0; i < n; i++)
                {
                        sum += a[row * n + i] * b[i * k + col];
                }
                c[row * k + col] = sum;
        }
}

__global__ void gpu_square_matrix_mult(int* d_a, int* d_b, int* d_result, int n)
{
        __shared__ int tile_a[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ int tile_b[BLOCK_SIZE][BLOCK_SIZE];

        int row = blockIdx.y * BLOCK_SIZE + threadIdx.y;
        int col = blockIdx.x * BLOCK_SIZE + threadIdx.x;
        int tmp = 0;
        int idx;

        for (int sub = 0; sub < gridDim.x; ++sub)
        {
                idx = row * n + sub * BLOCK_SIZE + threadIdx.x;
                if (idx >= n * n)
                {
                        // n may not divisible by BLOCK_SIZE
                        tile_a[threadIdx.y][threadIdx.x] = 0;
                }
                else
                {
                        tile_a[threadIdx.y][threadIdx.x] = d_a[idx];
                }

                idx = (sub * BLOCK_SIZE + threadIdx.y) * n + col;
                if (idx >= n * n)
```

```
                {
                        tile_b[threadIdx.y][threadIdx.x] = 0;
                }
                else
                {
                        tile_b[threadIdx.y][threadIdx.x] = d_b[idx];
                }
                __syncthreads();

                for (int k = 0; k < BLOCK_SIZE; ++k)
                {
                        tmp += tile_a[threadIdx.y][k] * tile_b[k][threadIdx.x];
                }
                __syncthreads();
        }
        if (row < n && col < n)
        {
                d_result[row * n + col] = tmp;
        }
}
void cpu_matrix_mult(int* h_a, int* h_b, int* h_result, int m, int n, int k) {
        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < k; ++j)
                {
                        int tmp = 0;
                        for (int h = 0; h < n; ++h)
                        {
                                tmp += h_a[i * n + h] * h_b[h * k + j];
                        }
                        h_result[i * k + j] = tmp;
                }
        }
}
int main(int argc, char const* argv[])
{
        int m, n, k;
        /* Fixed seed for illustration */
        srand(3333);
        printf("\033[1;33m");
        printf("Inserisci i valori della matrice M N e K\n");
        printf("\033[0m");
        scanf("%d %d %d", &m, &n, &k);
        printf("\n");
        // allocate memory in host RAM, h_cc is used to store CPU result
        int* h_a, * h_b, * h_c, * h_cc;
        cudaMallocHost((void**)&h_a, sizeof(int) * m * n);
        cudaMallocHost((void**)&h_b, sizeof(int) * n * k);
        cudaMallocHost((void**)&h_c, sizeof(int) * m * k);
        cudaMallocHost((void**)&h_cc, sizeof(int) * m * k);

        // random initialize matrix A
        printf("\033[1;34m");
        printf("Matrice_A:\n\n");
        printf("\033[0m");
        for (int i = 0; i < m; ++i) {
                for (int j = 0; j < n; ++j) {
                        h_a[i * n + j] = rand() % 1024;
                        //printf("%d\t",h_a[i * n + j]);
                }
                //printf("\n");
        }
```

```
        printf("\033[1;31m");
        printf("-------------------------------------------------------------\n");
        printf("\033[0m");
        // random initialize matrix B
        printf("\033[1;34m");
        printf("Matrice_B\n\n");
        printf("\033[0m");
        for (int i = 0; i < n; ++i) {
                for (int j = 0; j < k; ++j) {
                        h_b[i * k + j] = rand() % 1024;
                        //printf("%d\t ",h_b[i * n + j]);
                }
                //printf("\n");
        }

        float gpu_elapsed_time_ms, cpu_elapsed_time_ms;

        // some events to count the execution time
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);

        // start to count execution time of GPU version
        cudaEventRecord(start, 0);
        // Allocate memory space on the device
        int* d_a, * d_b, * d_c;
        cudaMalloc((void**)&d_a, sizeof(int) * m * n);
        cudaMalloc((void**)&d_b, sizeof(int) * n * k);
        cudaMalloc((void**)&d_c, sizeof(int) * m * k);

        // copy matrix A and B from host to device memory
        cudaMemcpy(d_a, h_a, sizeof(int) * m * n, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, h_b, sizeof(int) * n * k, cudaMemcpyHostToDevice);

        unsigned int grid_rows = (m + BLOCK_SIZE - 1) / BLOCK_SIZE;
        unsigned int grid_cols = (k + BLOCK_SIZE - 1) / BLOCK_SIZE;
        dim3 dimGrid(grid_cols, grid_rows);
        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

        // Launch kernel
        clock_t t;
        if (m == n && n == k)
        {

                        t = clock();
                        gpu_square_matrix_mult << <dimGrid, dimBlock >> > (d_a,
d_b, d_c, n);
                        cudaDeviceSynchronize();
                        t = clock() - t;
                                        printf("\033[1;31m");
                        printf("-------------------------------------------------
---------\n");
                        printf("\033[0m");
                        printf("\033[0;32m");
                        printf("Tempo per calcolare il prodotto tra le matrici:
%dx%d . %dx%d con GPU: %f ms.\n\n", m, n, n, k, (((double)t) / CLOCKS_PER_SEC) *
1000);
                        printf("\033[0m");
                        gpu_elapsed_time_ms = (((double)t) / CLOCKS_PER_SEC) *
1000;

        }
```

```
        else
        {
                t = clock();
                gpu_matrix_mult<<< dimGrid, dimBlock>>> (d_a, d_b, d_c, m, n, k);
                cudaDeviceSynchronize();
                t = clock() - t;
                //printf("It took GPU %f ms.\n", (((double)t) / CLOCKS_PER_SEC) *
1000);
                printf("\033[1;31m");
                printf("-------------------------------------------------------------
-\n");
                printf("\033[0m");
                printf("\033[0;32m");
                printf("Tempo per calcolare il prodotto tra le matrici: %dx%d .
%dx%d con GPU: %f ms.\n\n", m, n, n, k, (((double)t) / CLOCKS_PER_SEC) * 1000);
                printf("\033[0m");
                gpu_elapsed_time_ms = (((double)t) / CLOCKS_PER_SEC) * 1000;

        }
        // Transefr results from device to host
        cudaMemcpy(h_c, d_c, sizeof(int) * m * k, cudaMemcpyDeviceToHost);
        //cudaThreadSynchronize();
        cudaDeviceSynchronize();
        // time counting terminate
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        // compute time elapse on GPU computing
        cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
        //printf("Time elapsed on matrix multiplication of %dx%d . %dx%d on GPU:
%f ms.\n\n", m, n, n, k, gpu_elapsed_time_ms);

        // start the CPU version
        cudaEventRecord(start, 0);
        t = clock();
        cpu_matrix_mult(h_a, h_b, h_cc, m, n, k);
        cudaDeviceSynchronize();
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&cpu_elapsed_time_ms, start, stop);
        t = clock() - t;
        cpu_elapsed_time_ms = (((double)t) / CLOCKS_PER_SEC) * 1000;
        //printf("It took CPU %f ms.\n", (((double)t) / CLOCKS_PER_SEC) * 1000);
        printf("\033[1;31m");
        printf("-------------------------------------------------------------\n");
        printf("\033[0m");
        printf("\033[0;32m");
        printf("Tempo per calcolare il prodotto tra le matrici: %dx%d . %dx%d
con la CPU: %f ms.\n\n", m, n, n, k, (((double)t) / CLOCKS_PER_SEC) * 1000);
        printf("\033[0m");
        // validate results computed by GPU
        printf("\033[1;31m");
        printf("-------------------------------------------------------------\n");
        printf("\033[0m");
        printf("Confronto i risultati Tra GPU e CPU:\n");
        int all_ok = 1;
        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < k; ++j)
                {
                        //printf("[%d][%d]:%d == [%d][%d]:%d, ", i, j, h_cc[i*k +
j], i, j, h_c[i*k + j]);
                        if (h_cc[i * k + j] != h_c[i * k + j])
```

```
                        {
                                all_ok = 0;
                        }
                }
                //printf("\n");
        }
        printf("\n");
        printf("\033[1;31m");
        printf("-------------------------------------------------------------\n");
        printf("\033[0m");
        printf("\033[1;34m");
        printf("Risultato Finale:\n");
        printf("\033[0m");
        printf("");
        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < k; ++j)
                {
                        //printf("%d\t", h_cc[i * k + j]);
                }
                //printf("\n");
        }
        printf("\033[1;31m");
        printf("-------------------------------------------------------------\n");
        printf("\033[0m");
        // roughly compute speedup
        if (all_ok)
        {
                printf("\033[0;32m");
                printf("prestazioni = %f\n", cpu_elapsed_time_ms /
gpu_elapsed_time_ms);
                printf("\033[0m");
        }
        else
        {
                printf("incorrect results\n");
        }

        // free memory
        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_c);
        cudaFreeHost(h_a);
        cudaFreeHost(h_b);
        cudaFreeHost(h_c);
        cudaFreeHost(h_cc);
        return 0;
```

link al file liberamente consultabile

https://mega.nz/#!tBE0TYbD