UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scienze Matematiche Fisiche e Naturali:
Laurea magistrale in Informatica

Corso di parallel computing

Relazione elaborati

Autore:
Andrea Moscatelli

Docente:
Marco Bertini

Anno accademico 2017-2018

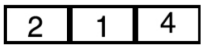
Contents

1 Elaborato 1: SAD in JAVA

Il progetto in questione tratta l'argomento del *pattern recognition*, ovvero il riconoscimento di un certo andamento in una serie di valori.
Il linguaggio scelto per lo sviluppo di tale elaborato è JAVA ed i dati utiliazzati (sia il pattern da riconoscere che il dataset analizzato) sono stati generati artificialemtne con valori random, ricreando serie numeriche di tipo random walk.

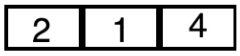
1.1 SAD - Sum of Absolute Differences

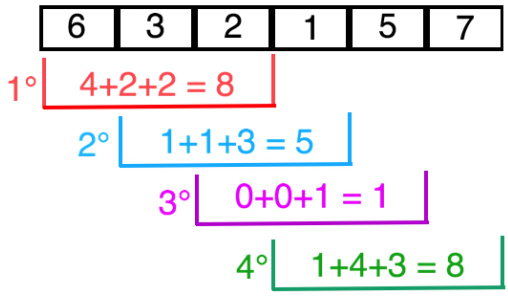
La tecnica utilizzata in questo elaborato per il riconoscimento di tale andamento è la *SAD - Sum of Absolute Differences*. Tale tecnica sfrutta le difference algebriche che ci sono fra il pattern ed ogni frammento dell'insieme di dati, selezionando il frammento che più "assomiglia" al pattern ricercato.
Ad esempio, se il pattern in questione è quello in figura



e il nostro insieme di valori (formato da una sola serie numerica) è il seguente

allora quello che otteremo con la tecnica SAD sarà





identificando il pattern nel terzo frammento, ovvero quello col SAD più basso.

1.2 Approccio sequenziale e multithread

La prima implementazione dell'algoritmo prevede una semplice scansione sequenziale di tutto il dataset fatta da un singolo processo, che analizza una ad una tutte le serie numeriche memorizzando il miglior valore SAD trovato fino a quel momento.
Questo approccio molto intuitivo ha evidentemente lo svantaggio di non essere particolarmente veloce, ma vedremo in seguito le performance più in dettaglio.
Il secondo approccio è quello di scansionare il dataset e calcolare il SAD di ogni frammento in modo parallelo, suddividento il dataset in n parti uguali, dove n è il numero totali di processi.
Questo approccio è sicuramente più veloce ma prevede una fase di divisione del dataset e una fase di ricongiungimento dei risultati ottenuti da ogni singolo thread.
Prima di analizzare i risultati e le tempistiche ottenute vediamo in dettaglio quali sono state le scelte effettuate nell'implementazione di tale versione.
Per la generazione e gestione dei differenti threads ho scelto di usare un *Executor*, nello specifico un Executor a numero fisso di processi. Il vantaggio di un Executor è quello di mantenere separati la creazione e la gestione dei threads dal resto del programma e di limitare la creazione di un numero eccessivo di threads mantenendo quindi basso l'overhead dell'applicazione.
Normalmente un numero di processi pari al numero dei cores della macchina sulla quale stiamo facendo girare il nostro programma è già una buona soluzione, ma il numero dei threads creati è stato fatto variare per capire meglio il loro effettivo vantaggio.
Se n è il numero di threads, ogni thread gestisce $\frac{1}{n}$ del dataset calcolando il suo miglior valore SAD. Alla terminazione di ogni thread, quest'ultimi comunicano il loro miglior risultato aggiornando in maniera sincronizzata il migliore globale.
Qui di seguito il codice in dettaglio della fase appena descritta.

```
1    /**
2     * The parallel version of the pattern recognition
          using the SAD technique
3     *
4     * @param numberOfThreadPerCore
5     *            the number of threads per core we
          want to use
6     *
7     * @return the best SAD between the generated
          series and pattern with the
8     *         indication of series index, series
          position and sad value.
9     */
10   private PatternRecognitionResult getBestSadParallel
         (float numberOfThreadPerCore) {
11      /*
```

```
12          * A good rule of thumb is to have approximately
                the same number of threads as
13          * available cores.
14          */
15         int threadPoolSize = new Integer((int) (Runtime.
               getRuntime().availableProcessors() *
               numberOfThreadPerCore));
16         /*
17          * I create an executor with a fixed size to
                avoid the generation of an
18          * excessive number of threads.
19          */
20         ExecutorService executor = Executors.
               newFixedThreadPool(threadPoolSize);
21         /*
22          * Each thread will take care of a portion of
                the whole matrix of series
23          */
24         int seriesSlice = new Double(Math.ceil((float)
               series.length / (float) threadPoolSize)).
               intValue();
25         PatternRecognitionResult bestSad = new
               PatternRecognitionResult(-1, -1, Float.
               MAX_VALUE);
26         for (int i = 0; i < threadPoolSize; i++) {
27            int startIndex = i * seriesSlice;
28            int endIndex;
29            if (i == threadPoolSize - 1) {
30               endIndex = series.length - 1;
31            } else {
32               endIndex = Math.min((i + 1) * seriesSlice
                     - 1, series.length - 1);
33            }
34            executor.execute(new Runnable() {
35
36               @Override
37               public void run() {
38                  PatternRecognitionResult bestLocalSad =
                        getBestSadSequential(series,
                       startIndex, endIndex);
39                  /*
40                   * Once the thread found its best local
                        SAD related to its portion of
                       series, in
41                   * a synchronized manner, it will
                       update the general best SAD (if it'
```

```
                    s the
42                  * case).
43                  */
44                 synchronized (bestSad) {
45                    if (bestLocalSad.getSadValue() <
                          bestSad.getSadValue()) {
46                       bestSad.setPositionIndex(
                             bestLocalSad.getPositionIndex
                             ());
47                       bestSad.setSeriesIndex(
                             bestLocalSad.getSeriesIndex())
                             ;
48                       bestSad.setSadValue(bestLocalSad.
                             getSadValue());
49                    }
50                 }
51              }
52           });
53        }
54        executor.shutdown();
55        try {
56           /*
57            * Manual barrier with timeout
58            */
59           executor.awaitTermination(60, TimeUnit.
                SECONDS);
60        } catch (InterruptedException e) {
61           e.printStackTrace();
62        }
63
64        return bestSad;
65     }
```

Per quanto riguarda il calcolo del miglior valore SAD, sia la versione sequenziale che quella parallela utilizzano lo stesso frammento di codice, ovvero scansionano in maniera sequenziale le serie numeriche a loro assegnate calcolando frammento per frammento il valore SAD. L'unica differenza fra i due approcci è il range di azione sul dataset: nel caso sequenziale il range è l'intero dataset che viene assegnato ad un unico processo mentre nel caso parallelo a ciascuno degli n threads verrà assegnato un range corrispondente ad un *n-esimo* del dataset.

```
1     /**
2      * It analyzes the series in the range [startIndex
          ; endIndex] calculating the
3      * best SAD for that range.
4      *
5      * @param seriesToAnalize
```

```
     *            the whole bunch of series generated
     * @param startIndex
     *            the first series to be analyzed
     * @param endIndex
     *            the last series to be analyzed
     * @return the best SAD calculated in the passed
        range, represented as a
     *         {@link PatternRecognitionResult}.
     */
    private PatternRecognitionResult
       getBestSadSequential(float[][] seriesToAnalize,
       int startIndex, int endIndex) {
       PatternRecognitionResult bestSad = new
          PatternRecognitionResult(-1, -1, Float.
          MAX_VALUE);
       for (int i = startIndex; i <= endIndex; i++) {
          for (int j = 0; j < seriesToAnalize[0].length
              - pattern.length; j++) {
             float sad = getSad(seriesToAnalize[i], j);
             if (sad < bestSad.getSadValue()) {
                bestSad = new PatternRecognitionResult(
                   i, j, sad);
             }
          }
       }

       return bestSad;
    }
```

1.3 Analisi delle prestazioni e conclusioni

Per capire meglio i vantaggi e svantaggi dei due approcci ho avviato l'algoritmo su un Quad-Core 2,6 GHz Intel i7 variando sia il numero di threads per core che il numero di serie numeriche da analizzare.

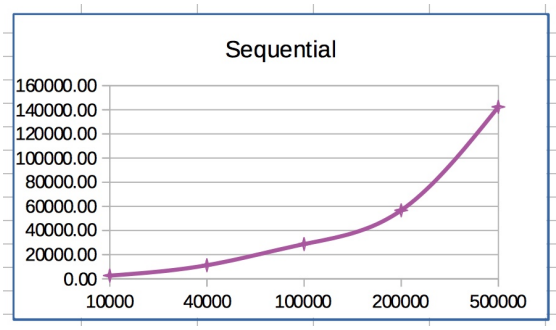
```
    private static int[] NUMBER_OF_SERIES = { 10000, 40
       000, 100000, 200000, 500000 };
    private static int[] NUMBER_OF_THREAD_PER_CORE = {
       1, 5, 20, 50, 100 };
```

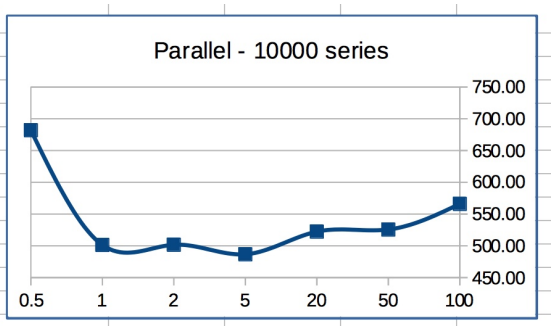
```
    public static void main(String[] args) {

       long seed = (long) (Math.random() * 10000);
       System.out.println("SEED: " + seed);
       RANDOM.setSeed(seed);

```

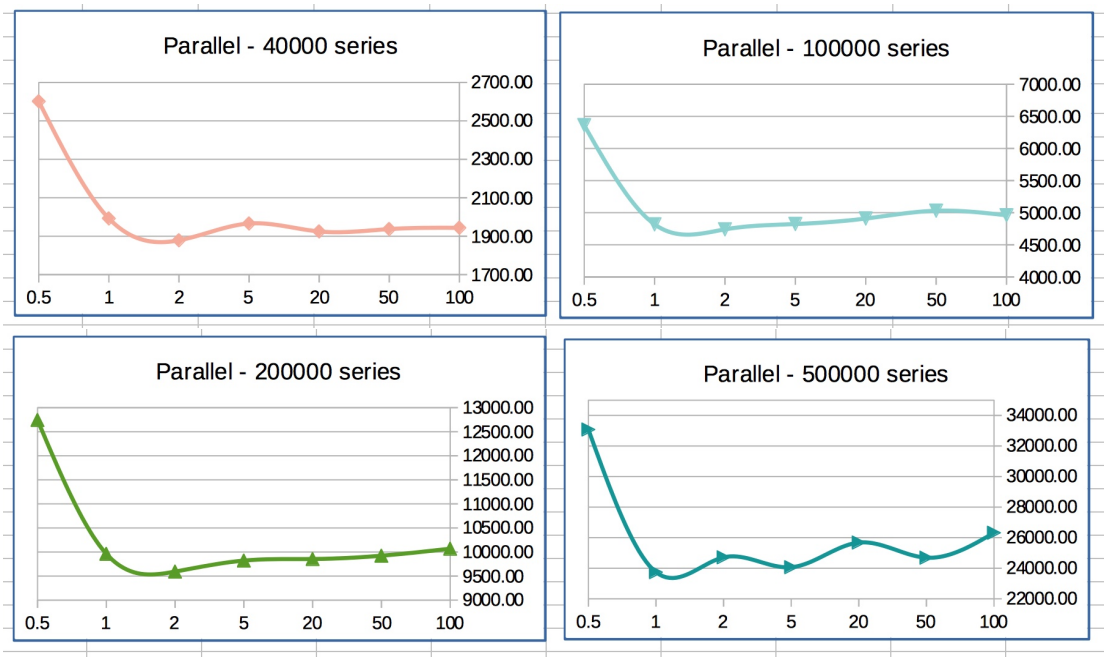
```
7       Pair<PatternRecognitionResult, Long>[]
            resultSequential = new Pair[NUMBER_OF_SERIES.
            length];
8       Pair<PatternRecognitionResult, Long>[][]
            resultParallel = new Pair[NUMBER_OF_SERIES.
            length][NUMBER_OF_THREAD_PER_CORE.length];
9       int row = 0;
10      for (int numberOfSeries : NUMBER_OF_SERIES) {
11          int column = 0;
12
13          System.out.println("\n\nNumber of series: " +
                numberOfSeries);
14          PatternRecognitionMain patternRecognition =
                new PatternRecognitionMain(numberOfSeries)
                ;
15          // Sequential method
16          System.out.println("Sequential version
                ongoing......");
17          long start = System.currentTimeMillis();
18          // PatternRecognitionResult bestSad = new
                PatternRecognitionResult(0, 0, 0);
19          PatternRecognitionResult bestSad =
                patternRecognition.getBestSadSequential(
                series);
20          long end = System.currentTimeMillis();
21          resultSequential[row] = new Pair<
                PatternRecognitionResult, Long>(bestSad,
                end - start);
22
23          System.out.println("##################
                SEQUENTIAL VERSION RESULT  ###############"
                );
24          printReport(start, bestSad, end);
25
26          for (float numberOfThreadPerCore :
                NUMBER_OF_THREAD_PER_CORE) {
27              System.out.println("\n\nNumber of series:
                    " + numberOfSeries + " - Number of
                    thread per core: "
28                      + numberOfThreadPerCore);
29
30              // Parallel method
31              System.out.println("\nParallel version
                    ongoing......");
32              long startP = System.currentTimeMillis();
33              PatternRecognitionResult bestSadP =
```

```
                patternRecognition.getBestSadParallel(
                    numberOfThreadPerCore);
34              long endP = System.currentTimeMillis();
35
36              System.out.println("#################
                    PARALLEL VERSION RESULT
                    #################");
37              printReport(startP, bestSadP, endP);
38
39              resultParallel[row][column] = new Pair<
                    PatternRecognitionResult, Long>(
                    bestSadP, endP - startP);
40              column++;
41              if (!bestSad.equals(bestSadP)) {
42                  System.out.println("!!!!!!!! ERROR ON
                        CALCULATION OF SAD !!!!!!!!!!!!");
43                  error = true;
44              }
45          }
46          row++;
47      }
48
49      printFullReport(resultSequential, resultParallel
            );
50
51  }
```

Per ogni combinazione ho ripetuto l'algoritmo 3 volte misurando il tempo trascorso. Quello che ho ottenuto sono i seguenti risultati:







Nel grafico "Sequential" viene messo in raffronto la durata dell'algoritmo in millisecondi (sull'asse Y) al variare del numero di serie numeriche (asse X) e come ci potevamo aspettare il rapporto fra queste due grandezze è piuttosto lineare.
Per quanto riguarda la versione parallela invece ho fatto variare non solo la quantità di serie numeriche ma anche il numero di thread per core: 0.5, 1, 2, 5, 20, 50 e 100.
Quello che si evince dai grafici è non solo una notevole diminuzione del tempo di esecuzione comparato alla versione sequenziale per svolgere lo stesso procedimento ma anche che, per ognuna delle quantità di serie numeriche trattate, la versione più veloce la si ottiene con 1-2 threads per core, valore oltre il quale il benificio di una versione multi-threads sembra diventare ininfluente e l'overhead dei processi aumenta risultando quindi in performance peggiori.
In conclusione, come ci aspettavamo, sfruttare la potenza del multithreading è senza dubbio un approccio da considerare se vogliamo migliorare le prestazioni del nostro algoritmo, ma esagerare con la creazione di threads non porta nessun beneficio. Il numero ottimale di thread per la nostra applicazione è solitamente **1-2 threads per core**.

2 Elaborato 2: Bigrammi e trigrammi in CUDA

Il secondo elaborato tratta l'analisi di bigrammi e trigrammi in un testo e più precisamente calcola le frequenze partendo da un file di testo qualsiasi.
Il linguaggio scelto è CUDA (Compute Unified Device Architecture) ovvero un linguaggio propietario NVIDIA che permette di sfuttare la velocissima GPU delle schede NVIDIA per i nostri processi. Vedremo nelle prossime sezioni quali sono state le scelte implementative per affrontare tale argomento ed i risultati ottenuti.

2.1 I bigrammi e i trigrammi

Prima di descrivere in dettaglio il lavoro svolto, è opportuno definire cosa sono i *bigrammi* e i *trigrammi*.
I bigrammi sono una coppia di parole o lettere adiacenti all'interno di un testo, mentre i trigrammi sono triplette di lettere o parole. L'analisi di queste sequenze di grafemi viene utilizzato in diversi ambiti linguistico-scientifici ed in particolar modo nell'ambito della decriptazione di password o testi cifrati.
In questo elaborato mi sono concentrato solo su bigrammi e trigrammi di lettere ed il calcolo di tali sequenze è molto semplice. Per capirlo meglio prendiamo un caso d'esempio.
Se dovessimo calcolare i bigrammi del testo "aabb bcc" otteremmo i risultati in tabella 1.

Bigramma	Frequenza
aa	1
ab	1
bb	2
bc	1
cc	1

Table 1: frequenze assolute di bigrammi nel testo "aabb bcc"

Mentre per un testo più articolato come questo:

"Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
che la diritta via era smarrita
E quanto a dir qual era è cosa dura
esta selva selvaggia ed aspra e forte
che nel pensier rinnova la paura".

Quello che otteremmo sarebbe la tabella dei bigrammi nella figura seguente:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	0	0	1	1	0	0	0	0	2	0	0	2	1	1	1	1	0	7	1	4	2	5	0	0	0	0	A
B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	B
C	1	0	0	0	2	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	C
D	3	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	D
E	6	0	0	1	0	0	0	2	1	0	0	1	1	2	0	2	0	0	3	1	0	0	0	0	0	0	E
F	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F
G	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	G
H	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	H
I	1	0	0	3	0	0	1	0	0	0	0	0	2	0	0	0	0	4	1	0	0	2	0	0	0	0	I
J	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	J
K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	K
L	2	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	L
M	2	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	M
N	1	0	0	0	2	0	0	0	3	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	N
O	1	0	1	0	0	1	0	0	0	0	0	0	0	2	0	0	0	1	0	1	0	0	0	0	0	1	O
P	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	P
Q	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	Q
R	1	0	0	0	4	0	0	0	3	0	0	0	0	0	1	1	0	2	0	2	3	0	0	0	0	0	R
S	5	0	0	0	1	0	0	0	0	0	0	0	0	1	3	0	0	0	0	0	0	0	0	0	0	0	S
T	0	0	0	0	0	0	0	0	4	0	0	0	0	1	0	0	0	1	2	1	0	0	0	0	0	0	T
U	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	U
V	2	0	0	0	0	0	0	0	0	0	0	3	0	0	2	0	0	0	0	0	0	0	0	0	0	0	V
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	W
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
Y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Y
Z	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Z
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	

Per quanto riguarda i trigrammi, il concetto è identico ma invece di coppie di lettere dobbiamo consierarne le triplette ottenendo quindi un cubo di frequenze.

2.2 CPU vs GPU

Per capire e quantificare un eventuale vantaggio nel gestire tale algoritmo con la GPU, ho sviluppato questa applicazione sia utilizzando la CPU che la GPU. La versione con la CPU, come si può vedere dal codice seguente, scansiona il testo lettera per lettera popolando una mappa di valori dove verranno conteggiate le frequenze di ogni singola sequenza (che sia bigramma o trigramma).

```
// this method finds the graphems (bigram or trigram) using the CPU
__host__ void findGraphemsWithCPU(string line, std::map<std::string,int> &graphems) {

    int tail = FIND_BIGRAM? 1 : 2;

    for(int i = 0; i < line.length()-tail; i++) {

        string key = std::string() + line[i] + line[i+1];
        if(!FIND_BIGRAM)
            key = key + line[i+2];

        std::map<std::string,int>::iterator it = graphems.find(key);
        if(it != graphems.end()){
            it->second++;
        }else{
            graphems.insert(std::make_pair(key, 1));
        }

    }

}
```

La versione che utilizza la GPU invece deve allocare sia sull'host che sul device lo spazio necessario a contenere i contatori di tutte le possibili combinazioni delle sequenze cercate, copiare il testo sul device e determinare la grandezza della griglia e dei blocchi sul device.

```
// this method finds the graphems (bigram or trigram) using the GPU
__host__ int* methodWithGPU(std::string line){

    // GRAPHEMS ARRAY
```

```
5    int lengthGraphems = FIND_BIGRAM? 26*26 : 26*26*26;
6    int *graphemsArrayDevice;
7    int *graphemsArrayHost=(int*)calloc(lengthGraphems,
        sizeof(int));
8
9
10   // allocate device memory
11   CUDA_CHECK_RETURN(
12         cudaMalloc((void ** )&graphemsArrayDevice,
13               sizeof(int) * lengthGraphems));
14
15   // copy from host to device memory
16   CUDA_CHECK_RETURN(
17         cudaMemcpy(graphemsArrayDevice,
                graphemsArrayHost, lengthGraphems * sizeof
                (int),
18               cudaMemcpyHostToDevice));
19
20   //  TEXT LINE
21   int lengthLine = line.length();
22   char *lineDevice;
23
24   //  allocate device memory
25   CUDA_CHECK_RETURN(
26         cudaMalloc((void ** )&lineDevice,
27               sizeof(char) * lengthLine));
28   //
29   // copy from host to device memory
30   CUDA_CHECK_RETURN(
31         cudaMemcpy(lineDevice, line.c_str(),
                lengthLine * sizeof(char),
32               cudaMemcpyHostToDevice));
33
34
35   // execute kernel
36   int totalthreadNumber = GRID_DIM * BLOCK_DIM;
37   int sliceLength = ceil(float(lengthLine)/float(
        totalthreadNumber));
38   findGraphemsWithGPU<<< GRID_DIM, BLOCK_DIM >>>(
        lineDevice, graphemsArrayDevice, sliceLength,
        lengthLine, FIND_BIGRAM);
39   //
40   cudaDeviceSynchronize();
41
42   // copy results from device memory to host
43   CUDA_CHECK_RETURN(
```

```
44          cudaMemcpy(graphemsArrayHost,
               graphemsArrayDevice, lengthGraphems *
               sizeof(int),
45                cudaMemcpyDeviceToHost));
46
47
48    // Free the GPU memory here
49    cudaFree(lineDevice);
50    cudaFree(graphemsArrayDevice);
51    return graphemsArrayHost;
52
53 }
```

Il device avvierà quindi gli N processi, dove N è dato da

$$N = \text{dimensione griglia} \times \text{dimensione blocco}$$

che in base al loro ID identificheranno la porzione di testo da analizzare in maniera parallela, aggiornando di volta volta in maniera sincronizzata i contatori delle sequenze trovate.
Una volta terminati gli N processi, si copia il risultato sull'host, si libera la memoria sul device e si procede, se richiesto, alla stampa delle sequenze.

```
1  __global__ void findGraphemsWithGPU(const char *line,
      int* graphemsArray, int sliceLength, int lineLength
      , bool findBigram) {
2
3     int startPoint =
4           blockDim.x * blockIdx.x +
5           threadIdx.x;
6
7     startPoint *= sliceLength;
8
9     int endPoint = startPoint + sliceLength - 1;
10    int tail = findBigram? 1 : 2;
11    endPoint += tail;
12
13    int index1;
14    int index2;
15    int index3;
16    if((startPoint+tail) < lineLength ){
17       index2 = getCharIndex(line[startPoint]);
18       if(!findBigram) {
19          index3 = getCharIndex(line[startPoint+1]);
20       }
21    }
22
```

```
23
24      while((startPoint+tail) <= endPoint && (startPoint+
            tail) < lineLength){
25          index1 = index2;
26          if(findBigram) {
27              index2 = getCharIndex(line[startPoint+tail]);
28              atomicAdd(&graphemsArray[index1 * 26 + index2
                    ], 1);
29          }else{
30              index2 = index3;
31              index3 = getCharIndex(line[startPoint+tail]);
32              atomicAdd(&graphemsArray[index1 * 26 * 26 +
                    index2 * 26 + index3], 1);
33          }
34
35          startPoint++;
36      }
37
38      return;
39  }
```

2.3 Analisi della prestazioni e conclusioni

Per quanto riguarda l'analisi delle prestazioni ho avviato l'algoritmo su 3 tipi di file di differente dimensione:

- 100KB
- 2MB
- 7MB

variando sia la quantita' di blocchi per griglia sia di processi per blocco, entrambi nel set di valori [1,2,4,8,16,32,128,256,512]. Per quanto riguarda l'analisi delle performance utilizzando la GPU quello che ho ottenuto è rappresentato nei grafici in figura 1, 2 e 3, mentre i risultati della versione che fa uso della CPU sono rappresentati in tabella 2 ovvero il risultato di una media di 10 ripetizioni.

Dimensione file	**Tempo d'esecuzione (ms)**
100KB	62.23
2MB	891.60
7MB	2261.67

Table 2: tempi di esecuzione della versione con CPU al variare della dimensione del file in analisi

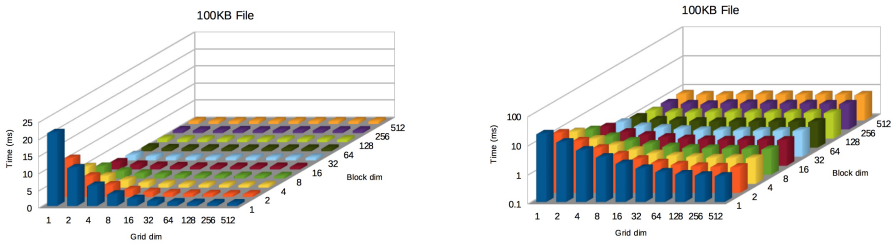


Figure 1: Risultato dell'analisi di un file di 100KB in scala normale (sinistra) e logaritmica (destra).

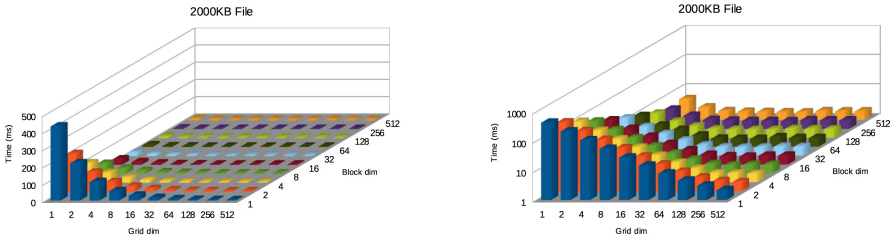


Figure 2: Risultato dell'analisi di un file di 2MB in scala normale (sinistra) e logaritmica (destra).

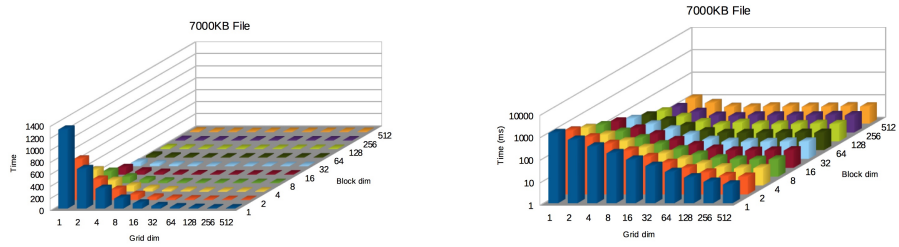


Figure 3: Risultato dell'analisi di un file di 7MB in scala normale (sinistra) e logaritmica (destra).

I risultati rappresentati in figura per la versione che fa uso della GPU sono stati ottenuti iterando l'algoritmo all'interno di un doppio ciclo *for* dedito a modificare la dimensione della griglia e dei blocchi utilizzati durante l'algoritmo. È necessario a questo punto specificare che la prima chiamata dell'algoritmo impiega notevolmente di piu' rispetto alle successive. Questo significa che "aprire la comunicazione" per la prima volta con la GPU della macchina sulla quale gira l'algoritmo ha un "costo" e questo costo sembra aggirarsi intorno ai **750**ms.

Alla luce di quanto ottenuto possiamo concludere che per quanto riguarda

la versione che utilizza la CPU, il tempo di esecuzione scala linearmente con la dimensione del testo trattato mentre per quanto riguarda la versione con la GPU, abbiamo performance migliori all'aumentare dei processi e dei blocchi creati che sembrano apportare un miglioramento in scala logaritmica. L'utilizzo della GPU e del linguaggio CUDA potrebbero pero' non essere sempre vantaggiosi, infatti per piccoli task aprire la comunicazione con la GPU risulterebbe di gran lunga più dispendioso dell'utilizzo della CPU. Per task piè onerosi invece, il costo di inizializzazione ($\approx$ 750ms) diventa sempre più ininfluente ed i guadagni apportati dal parallelismo diventano sempre più significativi.
In definitiva, come possiamo vedere dalla figura 4, sembrerebbe che, considerando anche il costo di inizializzazione per la comunicazione con la GPU, oltre una soglia di $\sim$1800KB conviene sempre utilizzare la GPU, mentre per dimensioni più piccole, l'utilizzo della CPU ha prestazioni migliori.

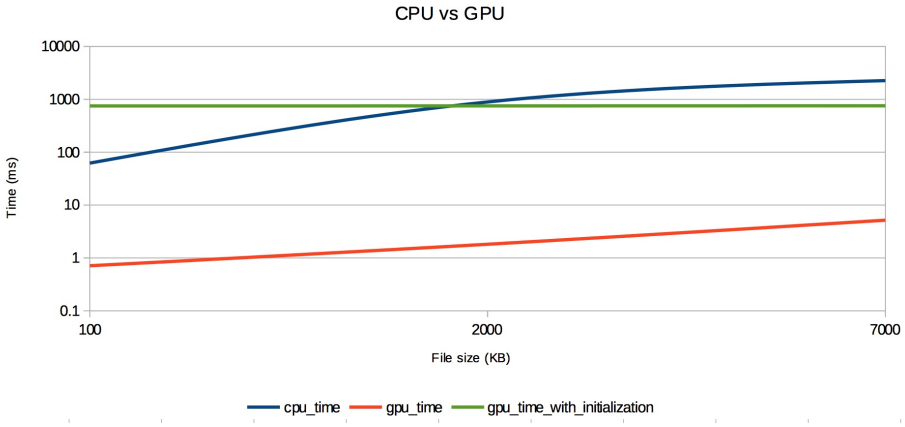


Figure 4: Prestazioni di CPU e GPU a confronto in relazione alla dimensione del file trattato