Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Una applicazione per la valutazione delle prestazioni di architetture parallele a basso consumo

Relatore:
Dott.
MORENO MARZOLLA

Candidato:
ALESSANDRO PETRELLA

Correlatore:
Dott.
DANIELE CESINI

Sessione III
Anno Accademico 2013/2014

Sommario

In questa tesi si descrive il lavoro svolto presso l'istituto INFN-CNAF, che consiste nello sviluppo di un'applicazione parallela e del suo utilizzo su di un'architettura a basso consumo, allo scopo di valutare il comportamento della stessa, confrontandolo a quello di architetture ad alta potenza di calcolo. L'architettura a basso consumo utilizzata è un system on chip mutuato dal mondo mobile e embedded contenente una cpu ARM quad core e una GPU NVIDIA, mentre l'architettura ad alta potenza di calcolo è un sistema x86_64 con una GPU NVIDIA di classe server. L'applicazione è stata sviluppata in C++ in due differenti versioni: la prima utilizzando l'estensione OpenMP e la seconda utilizzando l'estensione CUDA. Queste due versioni hanno permesso di valutare il comportamento dell'architettura a basso consumo sotto diversi punti di vista, utilizzando nelle differenti versioni dell'applicazione la CPU o la GPU come unità principale di elaborazione.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Le architetture fino ad ora utilizzate dalle comunità HPC (high performance computing) prevedono un insieme di nodi CPU(central processing unit)/GPU(graphics processing unit)[1], connessi tra di loro tramite una rete telematica a bassa latenza. Questo insieme di nodi viene chiamato *cluster*. Lo scopo di un cluster è quello di riuscire a distribuire un'elaborazione complessa sui vari nodi, ottenendo una diminuzione dei tempi di completamento dell'elaborazione. L'idea di cluster è nata per sostituire i supercalcolatori: elaboratori progettati per ottenere potenze di calcolo estremamente elevate. Col tempo i cluster sono riusciti ad ottenere le stesse prestazioni dei supercalcolatori, pur avendo un risparmio notevole sui componenti hardware. In realtà col tempo, sono stati raggiunti anche dei limiti fisici che i progettisti hardware hanno dovuto considerare: un processore è un insieme di transistor; maggiore è il numero di transistor di cui è composto un processore, tanto più sarà la sua velocità di elaborazione. Il problema è legato al fatto che più la velocità di elaborazione del processore si alza, più l'energia consumata è maggiore e questo può portare ad un innalzamento della temperatura del processore, temperatura fisicamente non dissipabile: in sostanza il processo-

[1]Particolare tipologia di coprocessore che si contraddistingue per essere specializzata nel rendering di immagini grafiche. Il suo tipico utilizzo è come coprocessore della CPU e da alcuni anni viene anche utilizzata in generiche elaborazioni dati

re è troppo piccolo per il calore che deve riuscire a dissipare. Questo limite è stato inoltre, il motivo della nascita di una nuova tendenza, che ha portato allo sviluppo di nuovi processori dual-core e multi-core. Questi tipi di processori consentono di aumentare la potenza del processore, senza andarne ad aumentare la frequenza di calcolo, quindi senza raggiungere la temperatura limite descritta in precedenza. La stessa idea è quella che sta alla base dei SoC (system on chip): si tratta di architetture sulle quali vengono racchiusi su di un unico chip una unità di calcolo avanzata e la maggior parte dei componenti necessari a controllare il sistema(CPU, GPU, Bus di comunicazione ecc.). Le architetture SoC hanno come principale scopo la trasportabilità ed il basso consumo di energia elettrica: per questo vengono anche chiamate architetture a basso consumo. I modelli SoC garantiscono un ottimo rapporto *potenza di calcolo / costo (investimento iniziale e consumo energia elettrica)*: questa caratteristica li ha posti all'attenzione di molti possessori di cluster HPC tradizionali.

Lo scopo di questo lavoro è dunque quello di valutare le prestazioni di un'architettura a basso consumo, misurandone le prestazioni e andandole a confrontare con quelle di un'architettura HPC. Per fare ciò è stata implementata una applicazione parallela, la quale verrà eseguita sulle due architetture.

In questo elaborato vengono innanzitutto presentate le architetture sulle quali sono eseguiti i test e si presentano i linguaggi con i quali è stata implementata l'applicazione parallela. Dopodiché andremo nei dettagli dell'applicazione implementata, mostrandone sia gli aspetti matematici che stanno alla base dell'algoritmo, sia gli aspetti implementativi, quindi in che modo è stato attuato il parallelismo. Al termine di questo elaborato sono mostrati i risultati dell'esperimento e vengono confrontate le prestazione delle diverse architetture.

Capitolo 2

Architetture Hardware

In questo capitolo si presentano in dettaglio le architetture sulle quali viene eseguita l'applicazione e misurate le prestazioni. La prima è una architettura di un nodo di un cluster HPC, la seconda è una architettura SoC a basso consumo.

2.1 Architettura cluster

Per i nostri esperimenti è stato utilizzato un nodo del cluster disponibile presso l'INFN-CNAF, nodo il quale presenta una GPU Tesla K20 e 2 processori Xeon Phi 3120; andiamo quindi a mostrare le caratteristiche di queste due componenti. Per riferirci a questa architettura durante la tesi sarà utilizzato il nome *architettura cluster*.

2.1.1 Xeon Phi 3120

Xeon Phi 320 è un coprocessore, quindi un processore ausiliario in aiuto al processore principale dell'elaboratore. Avendone a disposizione 2, l'architettura presenta 12 core fisici con possibilità di hyper-threading (12 x 2 = 24 core HT). Il prezzo di mercato attualmente si aggira intorno a 400 Euro. Vengono qui riportati i dettagli di uno Xeon Phi 320:

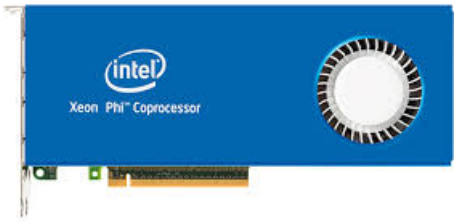

Figura 2.1: Illustrazione di una scheda con processore Xeon Phi 3120

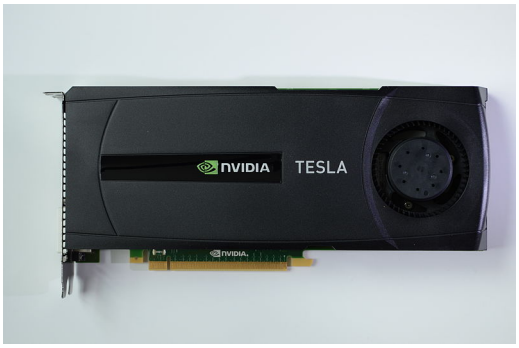

Figura 2.2: Illustrazione di una scheda NVIDIA Tesla K20

- *Nome:* Intel Xeon Processor E5-2620
- *Cache:* 15 MB
- *Instruction Set:* 64 bit
- *N° core:* 6
- *N° thread:* 12
- *Processsor Frequency:* 2 Ghz (2.5 Ghz Max Turbo Frequency)
- *N° core:* 6

2.1.2 Tesla K20

Tesla K20 è una scheda grafica di gamma alta prodotta da NVIDIA; è basata sulla architetture NVIDIA Kepler e attualmente il suo prezzo si aggira intorno a 2500 Euro. Tesla K20 è una GPU ad elevate prestazioni e riesce a raggiungere 1.17 TFLOPS[1] se utilizzata per operazioni richiedenti precisione in virgola mobile. Vengono ora presentate le caratteristiche tecniche principali:

- *Nome GPU:* GK110
- *Velocità di clock:* 706 Mhz
- *Memoria:* 5210 MB GDDR5
- *Core CUDA:* 2496

2.2 Architettura a basso consumo

L'architettura a basso consumo utilizzata per questo esperimento è il Jetson TK1, piattaforma sulla quale è presente il SoC Tegra K1 sviluppato da NVIDIA. Esso è stata pensata in due varianti: a 32 e a 64 bit; per questo esperimento è stato utilizzato la versione a 32 bit. Il modello è chiamato precisamente col nome Tegra K1 Logan (32 bit) ed il suo prezzo attuale si aggira intorno a 180 Euro. Viene fornito con pre-installato Linux4Tegra OS (Ubuntu 14.04 con installati tutti i driver necessari). Tegra K1 presenta una CPU ARM quad-core a 2.3 GHZ ed una GPU Tegra k1 a 192 core CUDA. Vengono ora elencate le caratteristiche principale dell'architettura:

[1]FLOPS (FLoating point Operations Per Second) è una unità di misura che serve a misurare il numero di operazioni in virgola mobile eseguite in un secondo dalla CPU/GPU, può essere calcolato tramite la seguente equazione: FLOPS = n° core X clock X (FLOPs / cycle). La maggior parte dei microprocessori moderni possono eseguire 4 FLOPs per ciclo di clock.

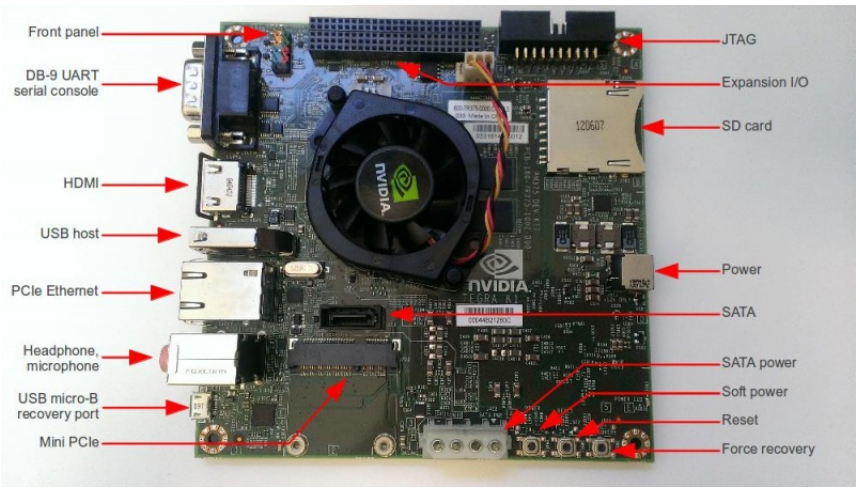

Figura 2.3: Illustrazione della piattaforma Jetson TK1, con installato il SoC NVIDIA Tegra K1

Caratteristiche tecniche Tegra K1

- *Dimensione:* 12,7 cm x 12,7 cm
- *CPU:* NVIDIA 4-Plus-1 2.32GHz ARM quad-core Cortex-A15 CPU
- *GPU:* NVIDIA Kepler GK20a GPU with 192 SM3.2 CUDA core
- *DRAM:* 2GB DDR3L 933MHz
- *Storage:* 16GB fast eMMC 4.51
- *ROM:* 4MB SPI boot flash
- *Power:* 12V DC

API supportate Tegra K1

- *CUDA:* 6.0
- *OpenGL:* 4.4
- *OpenGL ES:* 3.1

- *NPP:* Primitive CUDA per l'ottimizzazione prestazioni NVIDIA
- *OpenCV4Tegra:* Neon + GLSL + ottimizzazione della quad-core CPU

Test mostrano come la GPU della Tegra k1 possa superare i 300 GFLOPS, consumando circa 15 Watt di potenza. Già di per sé questo è un buon risultato pensando che l'architettura cluster precedentemente presentata, con l'utilizzo della GPU K20, consuma circa 1 Watt ogni 9 GFLOPS.

Capitolo 3

Programmazione Parallela

La *programmazione parallela* è un metodo di programmazione che consente l'esecuzione di uno stesso codice simultaneamente su diversi elaboratori; in questo modo si possono eseguire calcoli in parallelo in maniera tale da diminuire i tempi di esecuzione di un programma. Vi sono diverse tecniche di attuazione del parallelismo, la scelta di una di esse dipende quasi totalmente dal dominio del problema e dall'architettura hardware che si ha a disposizione. I sistemi che compiono calcolo parallelo sono classificati in due categorie: *SIMD* (single instruction-multiple data) dove il sistema esegue le stesse istruzioni su di un dominio di dati differenti, *MIMD* (muliple instruction- multiple data) dove invece non vi sono legami tra i processi paralleli né per quanto riguarda le istruzioni eseguite né per quanto riguarda i dati.

In un classico programma parallelo sono presenti parti del codice che vengono eseguite serialmente, quindi da un singolo thread, che gira su di un unico processore. Altre parti invece vengono eseguite da più thread contemporaneamente, che vengono girati sui processori a disposizione della macchina. La figura 3.1 mostra un esempio d'esecuzione di un programma parallelo dove sono presenti segmenti d'esecuzione effettuati da un unico *Master-Thread*, che ha il compito di eseguire quelle parti di codice che non necessitano o dove è impossibilitato il parallelismo. Ad esempio l'inizializzazione dei primi dati o la stampa dei risultati finali in un file. Sono presenti altri segmenti

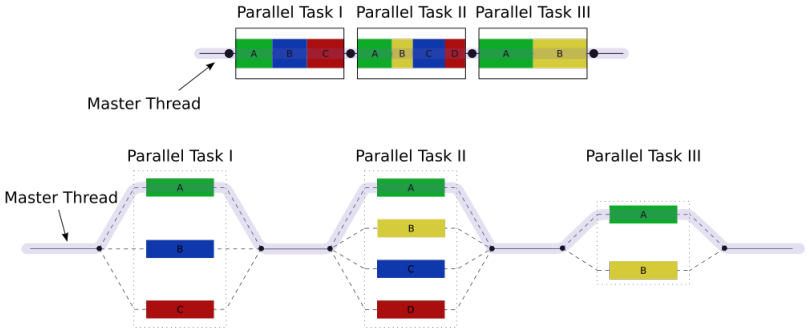

Figura 3.1: Struttura generale di una computazione parallela

chiamati *Parallel-Task* i quali compiono calcoli simultaneamente: ad esempio con il compito di effettuare una semplice elaborazione di una matrice o di un attacco ad una password per forza bruta. Questo passaggio da un unico thread a più thread in termine tecnico è chiamato *fork*.

Mostrato a grandi linee che cosa si intende per programmazione e calcolo parallelo, vediamo ora dei semplici esempi che ci mostrano come può essere realmente realizzata una applicazione parallela. Si pensi ad esempio ad un semplice algoritmo che prenda in ingresso un array *ArrayInt* di interi, una variabile intera n con il numero degli elementi dell'array e che debba modificare lo stesso array moltiplicando tutti gli elementi per 2. Una semplice soluzione seriale potrebbe essere:

```
moltiplicaPer2( int arrayInt[], int n){

  for( int i=0; i<n; i++ ){
    arrayInt[i] *= 2;
  }
}
```

Da questa implementazione, che chiameremo "moltiplicaPer2", è facile notare che il costo computazionale dell'operazione è proporzionale alla lunghezza dell'array in ingresso, quindi $O(n)$. Supponiamo ora di avere a disposizione p processori sul quale elaborare i dati, l'idea è quella di dividere le n

moltiplicazioni per il numero di processori, ottenendo un algoritmo di questo tipo:

```
moltiplicaPer2_parallel( int arrayInt[], int n){

  for all ( int i=0; i<n; i=i+p ){  Do For All p Processors
    arrayInt[i] *= 2;
  }
}
```

Con la pseudoistruzione *for all*, ogni processore elabora n/p moltiplicazioni riuscendo a far diminuire il costo computazionale dell'algoritmo da $O(n)$ ad $O(n/p)$. Ad esempio con soli 2 processori si riuscirebbero a dimezzare i tempi di esecuzione, mentre avere il numero di processori uguale al numero degli elementi dell'array significherebbe un costo computazionale pari ad $O(1)$. Ovviamente stiamo tenendo conto, esclusivamente, del tempo impiegato dall'algoritmo per arrivare alla soluzione perché a livello di uso di risorse il costo computazionale non cambierebbe; infatti nella versione seriale si ha $O(n) * O(1)$ e nella versione parallela si ha $O(n/p) * O(p)$: entrambi risultano essere $O(n)$.

L'intero processo può apparire a prima vista immediato e di facile implementazione ma ottenere un buon sistema parallelo non è così semplice; dietro un sistema parallelo si nascondono tanti fattori e variabili da tenere in considerazione: i tempi di spostamento dei dati da una memoria ad un'altra, la duplicazione dei dati su memorie differenti, la gestione delle zone critiche, la sincronizzazione tra i thread ecc..

Vengono ora presentate le due estensioni del linguaggio C++ utilizzate in questo lavoro per ottenere il parallelismo: OpenMP e CUDA. Per ulteriori dettagli rimandiamo a [1] per la parte di OpenMP e [2] per la parte in CUDA.

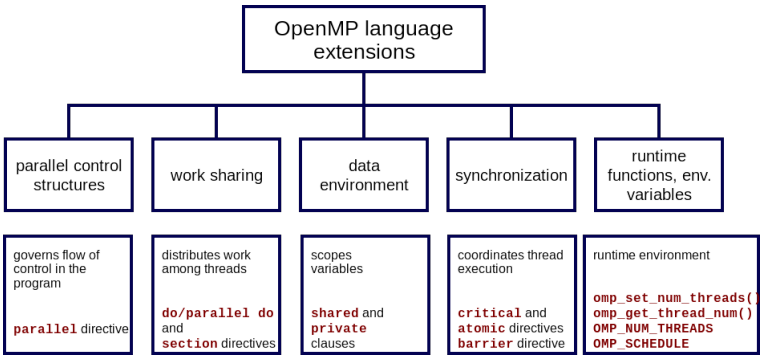

Figura 3.2: Principali costrutti di OpenMP

3.1 OpenMP

OpenMP è un'API (Application Programming Interface) multi piattaforma che consente di creare applicazioni su processori multi-core a memoria condivisa ed è composto: da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente. La figura 3.2 ne mostra i principali costrutti.

Viene ora presentato un esempio di applicazione della programmazione parallela attraverso OpenMP, modificando l'algoritmo moltiplicaPer2 visto in precedenza.

```
#include<omp.h>
moltiplicaPer2_OMP( int arrayInt[], int n){
  #pragma omp parallel for
  for (int i=0; i<n; i++){
    arrayInt[i] *= 2;
  }
}
```

In questo codice si notano due nuove istruzioni, proprio grazie ad esse è stato possibile implementare il parallelismo.

```
#include<omp.h>
```

Questa è l'istruzione con la quale si include l'intestazione della libreria OpenMP. La libreria nella maggior parte dei casi non va installata perché già inclusa

nelle librerie del compilatore; ad esempio è supportata già dalla versione 4.7 di Gcc.

```
#pragma omp parallel for
```

Questa è l'istruzione con la quale si applicano le direttive per il calcolo parallelo; in questo caso l'istruzione dentro al ciclo viene suddivisa per i core disponibili sulla macchina con schedulazione "core-iterazioni ciclo" prefissata: se ad esempio l'array avesse 12 elementi e il processore avesse 4 core, ogni core sarebbe responsabile delle operazioni sugli $idCore*(i+n°Core)$ elementi dell'array; dove $idCore$ è uguale al numero identificativo del core (da 0 a 3) e i è compreso tra 0 e 2((numero elementi array / numero core)-1). Questa tecnica risulta ottimale quando tutte le iterazioni del ciclo effettuano esattamente le stesse operazioni. In alcuni casi però, le iterazioni di uno stesso ciclo potrebbero avere tempi diversi di elaborazione; in questo caso non risulterebbe essere la scelta migliore quella di assegnare a priori le iterazioni del ciclo ai core, ma piuttosto sarebbe ottimale assegnarli dinamicamente: ogni volta che un core è libero controlla qual è la prossima iterazione del ciclo da eseguire e se ne prende carico. Questa tecnica è implementabile grazie alla seguente direttiva:

```
#pragma omp parallel for schedule(dynamic)
```

Oltre ai costrutti già mostrati, sono degne di nota le direttive per la gestione delle zone critiche. É possibile infatti inserire la clausola *critical* prima di un'istruzione, per specificare che si sta andando ad effettuare un'operazione su di una variabile condivisa tra i vari thread: così facendo si evita il rischio di imbattersi nel classico problema della *race condition* [1]. In questo modo quando un thread riesce ad accedere alla sezione critica, blocca automaticamente l'accesso alla variabile condivisa ai possibili thread contendenti. Altri metodi per evitare la medesima situazione sono: utilizzare

[1]La race condition si verifica nei sistemi concorrenti e avviene quando, in un sistema basato su processi multipli, il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti. Spesso questa situazione produce un malfunzionamento del sistema.

la direttiva *Atomic* o differenziare per ogni thread variabili locali e globali, all'interno del ciclo, con le direttive *shared* e *private*. OpenMP inoltre, mette a disposizione variabili d'ambiente con diverse funzioni. Ad esempio con *omp_set_num_threads* è possibile specificare il numero dei core massimo da utilizzare per l'elaborazione, con *omp_get_thread_num* è possibile conoscere l'identificativo del thread su cui è richiamata la variabile.

Per compilare un programma scritto in C++, con aggiunta di libreria OpenMP, è sufficiente aggiungere la direttiva *-fopenmp* alla riga di comando. Ad esempio se il file sorgente si chiamasse arrayx2.cpp, l'istruzione in ambiente linux potrebbe essere:

```
gcc -fopenmp -o array2x array2x.cpp
```

Si riceve così in uscita un file eseguibile di nome array2x.

3.2 CUDA

CUDA (Compute Unified Device Architecture) è una architettura hardware sviluppata da NVIDIA, che permette l'elaborazione parallela su GPU. Con CUDA è possibile parallelizzare un codice scritto in vari linguaggi ad alto livello come C, C++, Java, ecc. (per ogni linguaggio esiste una estensione ad hoc, l' estensione utilizzata in questo progetto è la CUDA-C/C++), utilizzando direttive che creano thread di elaborazione paralleli sulla GPU. Un codice scritto in C++ con estensione CUDA avrà una parte che sarà eseguita serialmente su di un Host, quindi sulla CPU della macchina, ed una parte che sarà eseguita in parallelo su di un Device, quindi sulla GPU della macchina.

Mostriamo ora come l'algoritmo moltiplicaPer2 può essere parallelizzato grazie a CUDA:

```
__global__ kernel(int* array){

   int idx =  threadIdx.x;
   array[idx] *= 2;
```

```
}

moltiplicaPer2_CUDA(){
 int n = 100;
 int* arrayInt_h;
 int* arrayInt_d;

 arrayInt = (float*)malloc(n * sizeof(Int));
 cudaMalloc((void**)& arrayInt_h, n * sizeof(Int) );

 for(i = 0; i<n; i++ ){
     arrayInt_h[i] = i;
 }

 cudaMemcpy(arrayInt_d, arrayInt_h, n * sizeof(Int),
                                     cudaMemcpyHostToDevice );

 dim3 DimGrid(n,1); dim3 DimBlock(1,1,1);
 kernel<<<DimGrid,DimBlock>>>(arrayInt_d);

 cudaMemcpy(arrayInt_h, arrayInt_d, n * sizeof(Int),
                                    cudaMemcpyDeviceToHost );

 for(i = 0; i<n; i++ ){
     print (arrayInt_h[i]);
 }
}
```

All'occhio di un qualsiasi lettore che non ha mai avuto a che fare con CUDA, il codice potrebbe risultare di difficile interpretazione. Viene di seguito spiegata ogni parte del codice.

É presente una funzione che è al di fuori della funzione main (moltiplicaPer2_CUDA). Questa è la parte del codice che viene eseguita sulla GPU e che tratteremo in seguito.

```
int n = 100;
int* arrayInt_h;
int* arrayInt_d;
```

```
arrayInt = (float*)malloc(n * sizeof(Int));
cudaMalloc((void**)& arrayInt_h, n * sizeof(Int) );

for(i = 0; i<n; i = 100){
     arrayInt_h[i] = i;
}
```

Questa è la parte di inizializzazione del programma dove viene settata una variabile n che servirà a dare la lunghezza agli array ed inizializzati due puntatori ad interi "arrayInt_h" e "arrayInt_d". I due saranno utilizzati rispettivamente nella parte del codice per l'host e per il device. Questa istruzione serve ad assegnare la parte di memoria di grandezza $n*sizeof(Int)$ ai due array: la prima nella memoria dell'host attraverso la classica chiamata alla funzione malloc di C++, la seconda nella memoria del device attraverso la chiamata alla funzione cudaMalloc, che altro non è che il rispettivo di malloc in CUDA. Il ciclo for serve a settare i valori per ogni cella di arrayInt_h.

```
cudaMemcpy(arrayInt_d, arrayInt_h, n * sizeof(Int),
                                    cudaMemcpyHostToDevice );
```

Con questa istruzione si copiano tutti i valori di arrayInt_h in arrayInt_d, quindi nell'array memorizzato nella memoria del device. La funzione cudaMemcpy richiede come primo parametro il puntatore di destinazione, come secondo il puntatore sorgente, come terzo la grandezza di memoria occupata dall'array puntato e come ultimo parametro il tipo di operazione: in questo caso il significato è copia dalla memoria host alla memoria device. In questo caso, per memoria del device, si è intesa la memoria detta globale della GPU ed è in effetti il principale mezzo di comunicazione tra l'host e il device. Il contenuto di questa memoria è visibile a tutti i thread eseguiti sulla GPU. Il problema della memoria globale sono gli elevati tempi di accesso; un elevato numero di letture e scritture su questa memoria infatti può andare a rallentare fortemente il processo di elaborazione. Oltre alla memoria globale, la GPU dispone di memorie condivise tra i thread eseguiti su di uno stesso blocco o

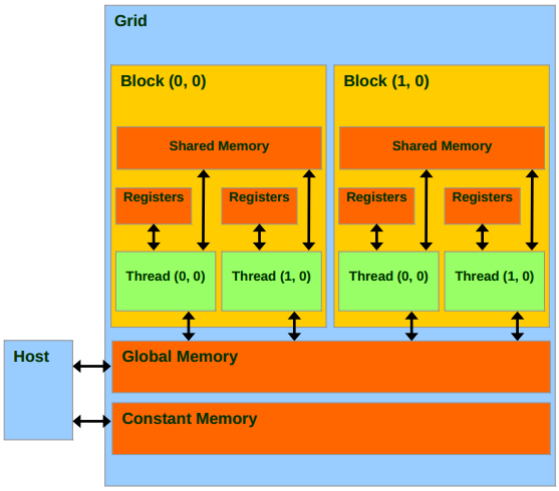

Figura 3.3: Modello della memoria nelle architetture CUDA

memorie completamente private per ogni thread. La figura 3.3 mostra un modello della memoria nelle architetture CUDA.

```
dim3 DmGrid(n,1);
dim3 DimBlock(1,1,1);
```

In questa parte si definiscono i blocchi d'esecuzione del codice device, cioè si decide la modalità di parallelizzazione del codice, quanti thread eseguire e con quale struttura. Ogni funzione chiamata dalla CPU ed eseguita dalla GPU ha bisogno di una configurazione di esecuzione. Una configurazione di esecuzione definisce una griglia ed il numero di thread eseguiti su ogni blocco della griglia. É possibile definire una griglia di esecuzione a 2 dimensioni, con all'interno una struttura thread a 3 dimensioni. Ad esempio le istruzioni:

```
dim3 DmGrid(3,2);
dim3 DimBlock(4,3,1);
```

creano una griglia di esecuzione $3*2 = 6$ blocchi ($DimGrid(3,2)$), al cui interno di ogni blocco vengono eseguiti $4*3*1 = 12$ thread ($DimBlock(4,3,0)$). La figura 3.4 mostra il risultato che si otterrebbe con le istruzioni appena presentate. Ritornando al nostro codice d'esempio, la configurazione di ese-

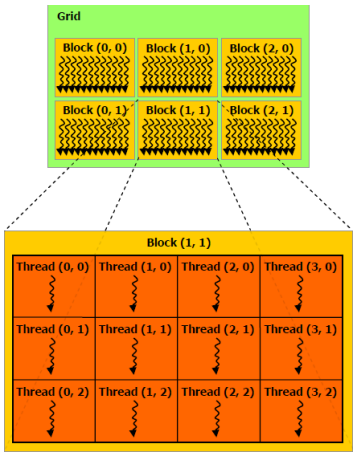

Figura 3.4: Griglia dei thread eseguiti in blocchi sui Core CUDA

cuzione crea una semplice griglia $n * 1$ ($DimGrid(n, 1)$), al cui interno di ogni blocco viene eseguito un singolo thread ($DimBlock(1, 1, 1)$). Una volta definita la configurazione di esecuzione, è possibile richiamare la funzione "kernel" del nostro esempio, dandogli in ingresso i parametri (tra le <<< >>>) e l'array inizializzato per il codice device:

```
kernel<<<DimGrid,DimBlock>>>(arrayInt_d);
```

La funzione così chiamata va dunque a creare tanti thread quanti sono gli elementi dell'array.

```
cudaMemcpy(arrayInt_h, arrayInt_d, n * sizeof(Int),
                                    cudaMemcpyDeviceToHost );

  for(i = 0; i<n; i++ ){
      print (arrayInt_h[i]);
  }
```

In questa ultima parte vengono copiati i nuovi valori dell'array dalla memoria del device a quella dell'host; infine vengono stampati a video i risultati.

```
__global__ kernel(int* array){

    int idx =  threadIdx.x;
```

```
    array[idx] *= 2;
}
```

La funzione kernel viene definita prima della funzione main. La direttiva "__global__" significa che questa funzione deve essere compilata per l'esecuzione su GPU, ed è richiamabile da una funzione eseguita su CPU. Una funzione che vuole essere eseguita su GPU e chiamata da una funzione eseguita sempre su GPU, deve essere dichiarata con la direttiva "__device__". L'istruzione "idx = threadIdx.x" assegna alla variabile *idx* l'id del thread in cui viene eseguito il calcolo; in questo modo ogni thread avrà idx settata con un numero diverso e nella seconda istruzione ogni thread modificherà la cella dell'array che ha come indice il numero dell'id del thread. CUDA mette a disposizione altre variabili d'ambiente che permettono ad ogni thread di essere distinto da un altro nella struttura creata con la configurazione di esecuzione. Andiamo a presentarle:

- threadIdx, threadIDy, threadIDz = Identificativo x,y e z, del thread all'interno del blocco
- blockIdx, blockIDy = identificativo x e y del blocco all'interno alla griglia
- blockDim.x, blockDim.y, blockDim.z = numero di thread all'interno del blocco, per le assi x, y e z
- gridDim.x, gridDim.y = Dimensione della griglia in numero di blocchi, per lea assi x,y

Per eseguire un codice CUDA è necessaria un server che abbia installata una GPU NVIDIA che supporti il linguaggio (CUDA è supportata dalle architetture Kepler in poi), invece per compilarlo è necessario scaricare dal sito ufficiale NVIDIA i CUDA repository ed il CUDA Toolkit; fatto ciò è possibile usare il compilatore *nvcc*.

Capitolo 4

Quadratic Sieve

L'algoritmo selezionato per l'applicazione del progetto è il *Quadratic Sieve* [3]. Questo algoritmo mira a risolvere il classico problema della fattorizzazione dei numeri RSA.

4.1 Numeri RSA e fattorizzazione

I numeri RSA sono un insieme di semiprimi; un semiprimo è il prodotto tra due numeri primi. Un numero primo è un numero intero maggiore di 1 che può essere diviso solamente da 1 e da se stesso. Un numero RSA di grandi dimensioni, composto da numeri primi abbastanza grandi, può essere utilizzato nella crittografia asimmetrica per generare chiavi personali utilizzabili per cifrare o firmare informazioni. Il numero RSA è di solito reso pubblico, cioè, chiunque conosce il numero ed a chi appartiene. Per poter però cifrare e decifrare informazioni, vanno conosciuti anche entrambi i numeri primi che lo compongono; è dunque chiaro che se una persona esterna venisse a conoscenza di questi ultimi, potrebbe facilmente impersonare il proprietario del numero RSA.

Il processo che porta alla scomposizione del numero RSA si chiama fattorizzazione. Una tecnica di fattorizzazione base è quella di provare per ogni intero n primo minore della radice quadrata del numero RSA, il test

$RSA \% n == 0$; se la risposta fosse positiva n sarebbe uno dei due numeri che compongono RSA. Il secondo numero è calcolabile grazie ad una semplice divisione: RSA/n. Tutto a prima vista potrebbe apparire abbastanza semplice,; il problema nasce quando il numero RSA da fattorizzare è di enormi dimensioni: gli standard attuali prevedono chiavi asimmetriche di almeno 600 cifre decimali. Queste enormi dimensioni rendono il test computazionalmente non fattibile su scale temporali accettabili data la quantità elevata di fattori da provare. Inoltre non abbiamo considerato il fatto di non essere sempre a conoscenza di tutti i numeri primi minori della radice del numero RSA. Capire se un numero è primo o no è un operazione computazionalmente molto onerosa.

Negli anni diversi studiosi si sono occupati del caso, un importante risultato è stato ottenuto da Carl Pomerance nel 1981 con lo sviluppo dell' algoritmo di nostro interesse chiamato Quadratic Sieve. Questo algoritmo riesce a fattorizzare i numeri RSA con costi computazionali relativamente bassi rispetto alla tecnica base.

4.2 Descrizione dell'algoritmo

L'algoritmo base del Quadratic Sieve consta di 8 passi. Viene ora brevemente presentato e si andranno in seguito a esplicarne i dettagli:

1. Si riceve in input il numero RSA da fattorizzare, lo chiameremo $nRSA$, si pone $X0 = sqrt(nRSA)$.

2. Si sceglie un intero $k > 0$.

3. Si crea un insieme chiamato FactorBase, contenente ogni numero p che rispetta le seguenti caratteristiche: $0 < p < k$, p è numero primo, $\left(\frac{n}{p}\right) \neq -1$. Con $\left(\frac{n}{p}\right)$ si intende il simbolo di Legendre (vedi dopo).

4. Si sceglie un intero $r > 0$, si calcolano le relazioni Y dove $Y_j = (X0 + j)^2 - nRSA$, per ogni j compresa nell'intervallo $[-r, r]$. Di queste

relazioni Y si mantengono quelle fattorizzabili con i soli elementi della FactorBase. Il numero dei valori mantenuti deve essere maggiore della $FactorBaseSize$ (Numero degli elementi della FactorBase).

5. Per ognuna delle relazioni $Y_1, Y_2, .., Y_t$ mantenute, si calcola il vettore $Z_2^t : v_2(Y_i) = (e_1, e_2, ..e_{FactorBaseSize})$, dove e_i è la riduzione a modulo 2 dell'esponente p_i, nella fattorizzazione di Y_i. Con tutti i vettori si crea una matrice.

6. Con il metodo della eliminazione di Gauss, si ricavano quei vettori $v_2(Y_i)$ che fanno parte dello spazio nullo della matrice.

7. Per ogni vettore trovato nel passo precedente, si calcola a uguale al prodotto delle radici delle relazioni y_i corrispondenti agli 1 del vettore soluzione, si calcola b uguale al prodotto delle potenze di $p_1, p_2, ..., p_t$ con esponenti uguali alla divisione per 2 degli esponenti della fattorizzazione delle stesse y_i.

8. Si calcolano i massimi comun divisori GCD(a+b, nRSA) e GCD(a-b, nRSA), se uno delle due operazioni dà un risultato compreso tra 0 e nRSA, è stato trovato un fattore non banale del numero RSA, altrimenti si torna al passo 2 effettuando una scelta di k più grande.

Descriviamo ora in dettaglio gli aspetti meno chiari dell'algoritmo.

4.2.1 Creazione FactorBase

Per la creazione della FactorBase è necessario effettuare valutazioni sui numeri candidati a farne parte: in primo luogo occorre verificare se si tratta di un numero primo poi verificare l'eventuale superamento del test di Legendre. Come è stato accennato alla sezione precedente, la verifica della caratteristica di primalità del numero primo può non essere una operazione computazionalmente leggera, è necessario quindi adottare un qualche metodo che venga in aiuto nell'operazione.

In questo lavoro è stato utilizzato il Test di Miller-Rabin [4], un test di primalità probabilistica altamente affidabile e computazionalmente efficiente. Vengono qui di seguito presentati i passi dell'algoritmo tralasciando lo specifico di prestazioni, lemmi e dimostrazioni matematiche.

Passi dell'algoritmo del test di Miller-Rabin:

1. Si riceve in input un numero $n > 1$.
2. Viene scelto a caso un numero $b1$ tale che $1 < b1 < n$ e se $MCD(b1, n) > 1$ allora n non è primo e termina l'algoritmo, altrimenti si prosegue.
3. Si calcola $s = n - 1$, e si divide s per 2 tante volte quanto possibile.
4. Si calcola $mod = b1^s \ \% \ p$.
5. Finché sono verificate le seguenti condizioni: s != $p-1$, mod != 1, mod != $p - 1$; si calcola $mod = mod^{mod} \ \% \ p$ e $s = s * 2$.
6. Se mod != $p - 1$ e $s \ \% \ 2 == 0$ allora n non è primo, altrimenti n è probabilmente primo.

Se un numero n passa il test Miller-Rabin, quindi n è considerato dal test un numero probabilmente primo, la probabilità che n non sia primo è 1/4. Occorre quindi reiterare il test per un numero sufficiente di volte per abbassare questa probabilità. Se, per esempio, il test verrebbe reiterato per 20 volte e il numero risultasse probabilmente primo tutte le volte, la probabilità che n non sia primo diventerebbe $1/(4^{20})$.

Una volta eseguito il test Miller-Rabin su di un numero per un certo numero di iterazioni, appurato che il numero in questione si possa considerare primo, è necessario confrontarlo con il numero RSA da fattorizzare attraverso un test chiamato simbolo di Legendre [5].

Simbolo di Legendre:

Il simbolo di Legendre è una funzione definita come segue: dati p numero primo ed a numero intero:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{se } p \text{ divide } a; \\ 1 & 1 \text{ se } a \text{ è un quadrato modulo } p; \\ -1 & \text{altrimenti.} \end{cases}$$

a è un quadrato modulo p se esiste un intero k tale che $k^2 \equiv a (\text{mod } p)$ o a è un residuo quadratico modulo p.

Se un numero n passa anche il test di Legendre, il numero verrà inserito tra gli elementi della FactorBase.

4.2.2 Eliminazione di Gauss modulo 2 e vettori nulli

L'eliminazione di Gauss[6] è usata in algebra lineare per risolvere sistemi di equazioni lineari ed invertire matrici. Nel nostro caso prende in ingresso una matrice N x M qualsiasi e compie operazioni del tipo: scambiare due righe, moltiplicare una riga per un numero diverso da zero, sommare una riga ad un multiplo di un'altra riga. Ciò che ne ritorna è una matrice a scalini N x M, quindi con la seguente proprietà: il primo elemento diverso da zero di ogni riga è più a destra del primo elemento diverso da zero della riga precedente. I primi elementi da sinistra di ogni riga diversi da 0 sono chiamati *pivot*. Se una riga ha tutti gli elementi uguali a zero; quella riga non ha pivot, e non lo avranno neanche tutte le righe al di sotto. Nel caso di nostro interesse le matrici lavorano in Z^2.

Esempio di una matrice modulo 2:

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Risultato dopo la eliminazione di Gauss:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Dalla matrice A, ora con forma a scalini, è possibile ricavare i suoi vettori nulli, cioè quei vettori che moltiplicati alla matrice restituiscono valore 0. I vettori sono calcolabili risolvendo il sistema lineare dove: i coefficienti sono rappresentati dai valori della matrice, vi sono un numero di equazioni pari al numero di righe di A ed un numero di incognite pari al numero di colonne. Nel nostro caso le colonne sono 5 e le variabili saranno chiamate $x1, x2, .., x5$. Denotando quindi:

$$x = \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \end{pmatrix} \textit{ vettore delle incognite}$$

va risolta la seguente equazione $Ax = 0$. La tecnica è individuare innanzitutto le colonne nella matrice A dove non è presente un pivot. Queste colonne rappresentano le variabili chiamate *variabili libere*. Nel nostro caso ne è presente una: $x3$. Ora ogni riga va trasformata in una equazione andando ad ottenere il seguente sistema:

$$\begin{cases} x1 = 0 \\ x2 + x3 = 0 \\ x4 = 0 \\ x5 = 0 \end{cases} \tag{4.1}$$

Ora basta assegnare un valore qualsiasi (nel nostro caso 1 o 0 perché stiamo lavorando in Z^2) alla variabile libera, e risolvere il sistema, andando ad ottenere un vettore nullo. Ogni volta che si assegnerà un valore diverso alle variabili libere si avrà come risultato un diverso vettore nullo. Nel nostro caso, avendo una sola variabile libera, i vettori nulli sono esattamente 2. Assegnando ad esempio ad $x3$ il valore 1, si ottiene il seguente vettore nullo:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

4.2.3 Setaccio

Il setaccio, ultima parte del passo 4 dell'algoritmo, è una delle parti più importanti ed allo stesso tempo meno immediate da comprendere. In questo passo dell'algoritmo viene innanzitutto scelto un numero $r > 0$, in secondo luogo viene creato una lista di relazioni Y, dove ogni elemento Y_j della lista è uguale ad $(X0 + j)^2 - nRSA$, con j compreso nell'intervallo $[-r, r]$. Fatto ciò bisogna andare a selezionare tra tutte le relazioni Y, quelle fattorizzabili con i soli valori presenti nella FactorBase. É chiaro che eseguire un test di divisibilità per ogni relazione Y, per ogni multiplo di ogni elemento della FactorBase rallenterebbe in modo disastroso il processo. Il trucco sta nel fatto che noi conosciamo la forma che hanno le relazioni Y.

Poniamo ad esempio $nRSA = 1100017$, quindi $X0 = sqrt(1100017) = 1048$, scegliamo $r = 25$. Le relazioni Y avranno dunque la forma $Y_j = (1048 + j)^2 - 1100017$ dove j è compreso tra $[-25, 25]$.

La domanda che ci si pone ora è : quali relazioni Y sono divisibili per 2?
$(X0 + j)^2 - n \bmod 2$ è uguale a 0 se e solo se $(X0 + j)^2 \bmod 2$ è uguale a

n, ossia se e solo se $X0 + j$ e' una radice quadrata di n mod 2; quindi per $X0 + j = 1 + 2k$, per k intero. Poiché $X0 = 0$ mod 2, le relazioni Y divisibili per 2 sono gli elementi dell'array Y_j, con $j = 1 + 2m$ intero dispari, quindi:

$$.... \ ,Y_{-5},\ Y_{-3}\ ,\ Y_{-1}\ ,\ Y_1\ ,\ Y_3\ ,\ Y_5\ ,\ ....$$

setacciamo l'array e dividiamo tutti questi numeri per 2.

Altro esempio: quando è che $(X0 + j)^2 - n$ e' divisibile per 49?
$(X0 + j)^2 - n =$ modulo 49 è uguale a n, se e solo se $X0 + j$ è una radice quadrata di n modulo 49 e per $n = 16$ modulo 49, le radici quadrate di n in Z^{49} sono 4 e 45. Quindi $X0 + j$ e' una radice quadrata di n, modulo 49 se e solo se $X0 + j = 4 + 49k$, con k in Z^{49} oppure $X0 + j = 45 + 49h$, con h in Z^{49}. Poiché $X0 = 19$ modulo 49, le relazioni divisibili per 2 sono gli elementi dell'array Y_j per $j = 34 + 49k$, con k in Z^{49} oppure Y_j, per $j = 26 + 49h$, con h in Z^{49}

$$Y_{-15},\ Y_{-23}$$

setacciamo l'array e dividiamo questi numeri per 7.

Bisogna ripetere questa operazione di setaccio per tutti i multipli di tutti gli elementi della FactorBase che sono minori della metà dell'elemento più grande nella lista delle relazioni: nel nostro esempio è $a(-25) = 1023^2 - n = -53488$. Una volta eseguite tutte le divisioni, le relazioni che ci interessano sono facilmente riconoscibili perché saranno a valore 1.

4.3 Esempio di elaborazione

Viene ora presentato un semplice esempio di elaborazione del Quadratic Sieve per rendere più chiaro il suo funzionamento, seguendo i passi descritti precedentemente.

Passo 1: viene preso in input il numero RSA, nRSA = 8129, e viene calcolata la sua radice quadrata intera, $X0 = sqrt(8129) = 90$[1].

Passo 2: viene scelto il parametro $k = 10$

Passo 3: viene generata la $FactorBase = \{2, 5, 7\}$

Passo 4: viene scelto il parametro $k = 20$, e vengono generate le relazioni Y.

$$
\begin{aligned}
Y_{-20} &= (X\_0 - 20)^2 - nRSA = (90 - 20)^2 - 8129 = -3229 \\
Y_{-19} &= (X\_0 - 19)^2 - nRSA = (90 - 19)^2 - 8129 = -3088 \\
Y_{-18} &= (X\_0 - 18)^2 - nRSA = (90 - 18)^2 - 8129 = -2945 \\
Y_{-17} &= (X\_0 - 17)^2 - nRSA = (90 - 17)^2 - 8129 = -2800 \\
Y_{-16} &= (X\_0 - 16)^2 - nRSA = (90 - 16)^2 - 8129 = -2653 \\
Y_{-15} &= (X\_0 - 15)^2 - nRSA = (90 - 15)^2 - 8129 = -2504 \\
Y_{-14} &= (X\_0 - 14)^2 - nRSA = (90 - 14)^2 - 8129 = -2353 \\
Y_{-13} &= (X\_0 - 13)^2 - nRSA = (90 - 13)^2 - 8129 = -2200 \\
Y_{-12} &= (X\_0 - 12)^2 - nRSA = (90 - 12)^2 - 8129 = -2045 \\
Y_{-11} &= (X\_0 - 11)^2 - nRSA = (90 - 11)^2 - 8129 = -1888 \\
Y_{-10} &= (X\_0 - 10)^2 - nRSA = (90 - 10)^2 - 8129 = -1729 \\
Y_{-9} &= (X\_0 - 9)^2 - nRSA = (90 - 9)^2 - 8129 = -1568 \\
Y_{-8} &= (X\_0 - 8)^2 - nRSA = (90 - 8)^2 - 8129 = -1405 \\
Y_{-7} &= (X\_0 - 7)^2 - nRSA = (90 - 7)^2 - 8129 = -1240 \\
Y_{-6} &= (X\_0 - 6)^2 - nRSA = (90 - 6)^2 - 8129 = -1073 \\
Y_{-5} &= (X\_0 - 5)^2 - nRSA = (90 - 5)^2 - 8129 = -904 \\
Y_{-4} &= (X\_0 - 4)^2 - nRSA = (90 - 4)^2 - 8129 = -733 \\
Y_{-3} &= (X\_0 - 3)^2 - nRSA = (90 - 3)^2 - 8129 = -560 \\
Y_{-2} &= (X\_0 - 2)^2 - nRSA = (90 - 2)^2 - 8129 = -385 \\
Y_{-1} &= (X\_0 - 1)^2 - nRSA = (90 - 1)^2 - 8129 = -208
\end{aligned}
$$

[1]Applichiamo ad sqrt(n), la radice quadrata intera di n, la seguente definizione: $[\ sqrt(n)sqrt(n)\ +\ q\ =\ n\ ]\ \&\ [\ 0\ <=\ q\ <=\ 2n\ ]$

$$Y_0 = (X\_0 + 0)^2 - nRSA = (90 + 0)^2 - 8129 = -29$$
$$Y_1 = (X\_0 + 1)^2 - nRSA = (90 + 1)^2 - 8129 = 152$$
$$Y_2 = (X\_0 + 2)^2 - nRSA = (90 + 2)^2 - 8129 = 335$$
$$Y_3 = (X\_0 + 3)^2 - nRSA = (90 + 3)^2 - 8129 = 520$$
$$Y_4 = (X\_0 + 4)^2 - nRSA = (90 + 4)^2 - 8129 = 707$$
$$Y_5 = (X\_0 + 5)^2 - nRSA = (90 + 5)^2 - 8129 = 896$$
$$Y_6 = (X\_0 + 6)^2 - nRSA = (90 + 6)^2 - 8129 = 1087$$
$$Y_7 = (X\_0 + 7)^2 - nRSA = (90 + 7)^2 - 8129 = 1280$$
$$Y_8 = (X\_0 + 8)^2 - nRSA = (90 + 8)^2 - 8129 = 1475$$
$$Y_8 = (X\_0 + 9)^2 - nRSA = (90 + 9)^2 - 8129 = 1672$$
$$Y_{10} = (X\_0 + 10)^2 - nRSA = (90 + 10)^2 - 8129 = 1871$$
$$Y_{11} = (X\_0 + 11)^2 - nRSA = (90 + 11)^2 - 8129 = 2072$$
$$Y_{12} = (X\_0 + 12)^2 - nRSA = (90 + 12)^2 - 8129 = 2275$$
$$Y_{13} = (X\_0 + 13)^2 - nRSA = (90 + 13)^2 - 8129 = 2480$$
$$Y_{14} = (X\_0 + 14)^2 - nRSA = (90 + 14)^2 - 8129 = 2687$$
$$Y_{15} = (X\_0 + 15)^2 - nRSA = (90 + 15)^2 - 8129 = 2896$$
$$Y_{16} = (X\_0 + 16)^2 - nRSA = (90 + 16)^2 - 8129 = 3107$$
$$Y_{17} = (X\_0 + 17)^2 - nRSA = (90 + 17)^2 - 8129 = 3320$$
$$Y_{18} = (X\_0 + 18)^2 - nRSA = (90 + 18)^2 - 8129 = 3535$$
$$Y_{19} = (X\_0 + 19)^2 - nRSA = (90 + 19)^2 - 8129 = 3752$$

Di queste relazioni vengono selezionate quelle che sono scomponibili con solamente i fattori facenti parte della nostra FactorBase:

$$Y_{-17} = -2800 = 2^4 * 5^2 * 7$$
$$Y_{-9} = -1568 = 2^5 * 7^2$$
$$Y_{-3} = -560 = 2^4 * 5 * 7$$
$$Y_5 = 896 = 2^7 * 7$$
$$Y_7 = 1280 = 2^8 * 5$$

Passo 5: Dalle relazioni trovate si crea una matrice N * M dove N è uguale al numero degli elementi della FactorBase più uno, per mantenere il segno del valore della relazione, ed M è uguale al numero delle relazioni mantenute.

Il valore di ogni cella rappresenta l' esponente di un certo fattore per una certa relazione.

$$\left(\begin{array}{c|ccccc} & Y_{-17} & Y_{-9} & Y_{-3} & Y_5 & Y_7 \\ \hline -1 & 1 & 1 & 1 & 0 & 0 \\ 2 & 4 & 5 & 4 & 7 & 8 \\ 5 & 2 & 0 & 1 & 0 & 1 \\ 7 & 1 & 2 & 1 & 1 & 0 \end{array}\right)$$

Di questa matrice se ne calcola il modulo 2:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Passo 6: Con il metodo della eliminazione di Gauss ricaviamo innanzitutto la matrice triangolare, e da essa i vettori nulli.

Matrice a scalini:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Vettore Nullo:

$$\begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Passo 7: il vettore nullo trovato corrisponde al prodotto tra la prima, seconda e quarta relazione, e determina la relazione quadratica:

$$(73 * 81 * 95)^2 = 2^16 * 5^2 * 7^2$$

Da queste informazioni riusciamo a calcolare a e b:

$$a = 73 * 81 * 95 \bmod 8129 = 834$$
$$b = 2^8 * 5 * 7 \bmod 8129 = 5817$$

Passo 8: ora si tenta di trovare un fattore non banale di nRSA cercando il massimo comune divisore tra nRSA e a+b e tra nRSA e a-b

gcd (a+b,n) = 1 (Fallito)
gcd (a-b, n) = 11 (Trovato)
nRSA = 8129 è scomponibile in 11 * 8129/11 = 11 * 739.

Capitolo 5

Implementazione Parallela

Come primo passo è stata implementata una versione seriale dell'algoritmo: ciò ha permesso di capirne gli aspetti meno chiari e permesso una conoscenza più ampia del problema. Uno dei primi aspetti considerati è stato dover lavorare con numeri interi di grandi dimensioni; per risolvere il problema la prima scelta è stata quella di utilizzare la libreria GNU Multiple Precision Arithmetic Library [7], la quale ha permesso di utilizzare numeri di enormi dimensioni come fossero normali interi. Inoltre la libreria mette a disposizione funzioni matematiche utili al nostro problema. Presto, ci si è posto il problema di dover utilizzare la libreria anche in CUDA: dopo essermi documentato ho scartato la libreria per impossibilità di integrazione con il linguaggio CUDA. Ho dunque deciso di utilizzare per le variabili che lo necessitavano il tipo int64_t, il quale potenzialmente ha permesso di lavorare con numeri di 18 cifre: non un numero così grande per il problema della fattorizzazione dei numeri RSA, ma abbastanza grande per il raggiungimento dello scopo dell'esperimento.

Una volta implementata la versione seriale, ho studiato il codice per capire quali parti fossero adatte ad un'implementazione parallela. Le parti più importanti sono risultate 3: il setaccio, l'eliminazione di Gauss, e la valutazione dei vettori nulli.

5.1 Setaccio parallelo

Come precedentemente descritto, il setaccio serve a individuare quali delle relazioni possono essere fattorizzate utilizzando i soli valori contenuti nella FactorBase. L'operazione viene riassunta in questo pseudocodice:

```
fb = num elementi FactorBase;
limit = max(elemento FactorBase) / 2;

for( i = 0 ; i < fb; i++ ){
  j = 1;

  while( true ){

    r = num relazioni divisibile da FactorBase[fb] ^ j;
    relazioni[r] = relazioni divisibili da da FactorBase[fb]^j;

    for( k = 0; k < r; k++  ){
        relazioni[r] /= FactorBase[fb];
    }

    j++;
    if (FactorBase[fb] ^ j > limit) break;
  }
}
```

La funzione consta di tre cicli annidati, dove il più esterno scorre tutti i valori all'interno della FactorBase, il secondo serve ad elevare a potenza il fattore della FactorBase per le volte necessarie. Il ciclo più interno scorre tutte le relazioni che sono divisibili per il fattore elevato a potenza. I cicli non sono intercambiabili perché ognuno ha bisogno di informazioni derivate dal ciclo più esterno e l'implementazione parallela è risultata abbastanza banale; non vi sono fondamentali differenze tra la versione in OpenMP ed in CUDA.

```
p = numero processori;
fb = num elementi FactorBase;
limit = max(elemento FactorBase) / 2;
```

```
for all( i = 0 ; i < fb; i = i + p ){     Do For All p Processors
  j = 1;

  while( true ){

    r = num relazioni divisibile da FactorBase[fb] ^ j;
    relazioni[r] = relazioni divisibili da da FactorBase[fb]^j;

    for( k = 0; k < r; k++  ){

        acquisisci mutex (relazioni[r]);
        relazioni[r] /= FactorBase[fb];
        rilascia mutex (relazioni[r]);
    }
    j++;
    if (FactorBase[fb] ^ j > limit) break;
  }
}
```

La parallelizzazione viene effettuata sul ciclo più esterno ed ogni thread è quindi responsabile di un elemento della FactorBase. È introdotto un *mutex*[1] al momento in cui è effettuata l'operazione di divisione di una relazione per l'elemento della FactorBase, perché la stessa relazione potrebbe essere sottoposta alla divisione in più thread contemporaneamente. Il costo computazionale in termini di tempo della funzione seriale $O(n)$ cala ad $O(n/p)$ dove, n è calcolato moltiplicando il numero degli elementi della FactorBase per le volte in cui il fattore viene elevato a potenza, per il numero delle relazioni divisibili per il fattore. p è uguale al numero dei thread paralleli. La stima dell'esatto valore di n risulta essere una operazione complessa e non influente per i nostri scopi, pertanto viene tralasciata.

[1]Con il termine mutex (mutual exclusion) si indica un procedimento di sincronizzazione fra processi o thread concorrenti, con il quale si impedisce che più task paralleli accedano contemporaneamente ai dati in memoria o ad altre risorse soggette a race condition

5.2 Eliminazione di Gauss parallela

L'algoritmo dell'eliminazione di Gauss in Z^2, data la matrice NxM in ingresso, restituisce una matrice NxM a scalini. L'algoritmo viene descritto nel seguente pseudocodice:

```
matrice[N][M] = matrice data in input;
riga_marcata = -1;
for ( i = 0; i < N; i++) {                          //Ciclo 1
   riga_pivot = -1;
   for ( j = i; j < M; j++) {                       //Ciclo 2
       if ( matrice[i][j] == 1 ) {
           riga_marcata++;
           riga_pivot = j;
       }
   }
   if (riga_pivot != -1){
     for (k = 0; k < N; k++){                        //Ciclo 3

         scambio riga matrice[k][riga_marcata]
         con riga matrice[k][riga_pivot];
     }
     for ( j = riga_marcata+1; j < M; j++) {     // Ciclo 4

       operazioni tra righe per rendere a scalini; // ciclo 5
     }
   }
}
```

L'algoritmo prevede quindi un ciclo esterno ed altri quattro interni ad esso. Il ciclo più esterno (ciclo 1) scorre le colonne della matrice dove, per ognuna di esse, vengono eseguite le seguenti operazioni: si cerca la riga col pivot (ciclo 2), la si scambia con la riga successiva all'ultima marcata (ciclo 3), vengono eseguite le operazioni classiche delle matrice per rispettare la regola del pivot (ciclo 4 e ciclo 5). La parallelizzazione dell'algoritmo non è banale come lo è stato per il caso precedente, infatti esaminando il ciclo 1 si nota che ognuna delle sue iterazioni per essere eseguita ha bisogno dello stato della

matrice derivante dall'iterazione precedente. Anche il ciclo 2 non può essere parallelizzato in modo banale perché si ha la necessità di trovare un solo pivot(quello più in alto): infatti se la riga col pivot risultasse esattamente la successiva di quella marcata nell'iterazione precedente, il ciclo 3 non verrebbe eseguito. I cicli 3 e 4 sembrano i più adatti alla parallelizzazione, infatti nel ciclo 3 le operazioni di scambio riga può essere effettuato in più thread paralleli indipendenti, stessa cosa vale nel ciclo 4 per le operazioni tra righe.

```
p = numero processori;
riga_marcata = -1;
for ( i=0; i<N; i++) {
    riga_pivot = -1;
    for ( j=i; j<M; j++) {
        if ( matrice[i][j] == 1 ) {
            riga_marcata++;
            riga_pivot = j;
        }
    }
    if (riga_pivot != -1){
      for ( k=0; k< N; k=k+p){        Do For All p Processors

          scambio matrice[k][riga_marcata]
          con matrice[k][riga_pivot];
      }
      for ( j=riga_marcata+1; j<M; j=j+p) {   Do For All p
                                                    Processors

        operazioni tra righe per rendere a scalini;
      }
    }
}
```

Dopo alcune versioni realizzate ed alcuni test, per le dimensioni delle nostre matrici, questa è risultata essere la versione più veloce, per la versione OpenMP, quindi la parallelizzazione costruita sul ciclo 4 e ciclo 5. Per la versione in CUDA invece, questa parallelizzazione avrebbe richiesto un trasferimento di dati tra la memoria CPU e la memoria GPU ad ogni iterazione

del Ciclo 1, sia in andata che in ritorno. Questo ha fatto sì che il tempo di trasferimento dati andasse a peggiorare la situazione, rispetto ad eseguire l'algoritmo serialmente. Dopo essersi documentati, si sono trovate soluzioni alternative che potessero risolvere il problema: queste soluzioni avrebbero impiegato un tempo notevole per essere studiate a fondo ed implementate. Per la versione CUDA quindi si è deciso di non adottare il calcolo parallelo nella eliminazione di Gauss.

5.3 Valutazione dei vettori nulli parallela

La valutazione dei vettori nulli è l'ultima operazione del quadratic sieve e consiste nel valutare tutti i vettori derivati dallo spazio nullo della matrice a scalini, in uscita dall'algoritmo precedente. Per ogni variabile libera si crea un vettore, dove tutte le variabili libere sono settate a 0 tranne quella presa in considerazione settata a 1. Successivamente si valuta un primo vettore così composto e se ne calcola un secondo addizionando ogni elemento del primo vettore ad 1. Questa operazione viene eseguita in alcune versioni del Quadratic Sieve. Nel nostro caso questa operazione ha fatto si che in alcuni casi venga data una soluzione finale in tempi più brevi, quindi si è deciso di adottarla. Il numero dei vettori da valutare perciò, è esattamente il numero delle variabili libere moltiplicato per 2. Il seguente pseudocodice presenta l'algoritmo:

```
matrice_scalini[N][M] = matrice data in input;
for ( i = 0; i < num_var_libere; i++) {

  risultato = valuta( matrice_scalini , i);

  risultato2 = valuta_inverso( matrice_scalini , i);
}
```

Da questo algoritmo sono state create 2 versioni parallele, una per OpenMP e una per CUDA. Andiamo a presentare la versione per OpenMP:

```
matrice_scalini[N][M] = matrice data in input;
```

```
p = numeri thread paralleli;
for ( i=0; i<num_var_libere; i=i+p ) { Do For All p Processors

  vettore[N] = vettore(matrice_scalini, i)

  risultato = valuta( vettore );

  risultato2 = valuta_inverso( vettore );
}
```

La versione per OpenMP è semplice e intuitiva: i thread paralleli sono stati organizzati utilizzando l'unico ciclo presente ed ogni thread è responsabile della valutazione di uno dei vettori e del suo inverso. Questa versione per CUDA non è stata valutata come la migliore perché, a differenza di OpenMP che utilizza i core della CPU, solitamente di numero limitato (nel nostro caso 1-12), CUDA è a disposizione di molti più core. Nel nostro caso infatti il numero dei core CUDA supera sempre il doppio del numero delle variabili libere. Questa considerazione ha portato all'elaborazione del seguente algoritmo:

```
matrice_scalini[N][M] = matrice data in input;
matrice_soluzioni[N*2][M];
p = numero thread paralleli;

for ( i=0; i<num_var_libere; i=i+p ){ Do For All p Processors

  matrice_soluzioni[N+num_var_libere][M] =
                              inverso( matrice_scalini[N][M] );
}
for ( i=0; i<num_var_libere*2; i=i+p ){ Do For All p Processors

  vettore[N] = vettore(matrice_soluzioni, i);

  risultato = valuta( vettore );
}
```

Nella versione in CUDA viene quindi creata una matrice aggiuntiva, la quale riporta sia i vettori originali che tutti gli inversi. Così facendo si riesce

ad effettuare la valutazione di entrambi i gruppi di vettori in parallelo. Per correttezza viene riportato il caso generale in cui il numero di thread creati può essere inferiore al numero delle variabili libere per 2, anche se come già detto nel nostro caso è una condizione mai verificata. Questo significa che la valutazione di tutti i vettori viene effettuata in tempo $O(1)$.

Capitolo 6

Valutazioni Delle Prestazioni

Le due versioni dell'applicazione (OpenMp e CUDA) sono state eseguite sulle due architetture precedentemente presentate. La versione con OpenMP è stata eseguita un numero di volte pari al numero dei core per ogni architettura, sfruttando ogni volta un numero di core differente. Ad esempio se l'architettura avesse a disposizione 2 core di elaborazione, il test verrebbe ripetuto 2 volte: prima utilizzando un solo core poi utilizzandoli entrambi. Con ognuna delle versioni si sono misurati i tempi di risoluzione dei differenti numeri RSA. Ogni numero RSA è composto da due numeri primi di differente grandezza in termini di numeri di cifre. Per l'esattezza delle misurazioni ogni numero è stato risolto 10 volte da ogni implementazione e se n'è calcolato il tempo di risoluzione medio. Oltre ai tempi di risoluzione sono stati misurati i consumi di energia necessari alla macchina per completare l'elaborazione. Gli intervalli di confidenza misurati sono talmente ridotti da essere considerati non influenti sui risultati, per questo motivo si è deciso di ometterli, allo scopo di rendere i grafici più chiari.

Alcuni dei grafici dei risultati delle elaborazioni con la versione dell'applicazione in OpenMP, presentano una funzione chiamata *speedup*, la quale viene ora definita.

Definiamo:

- p = numero di processori o core

- T_{serial} = tempo di esecuzione del programma seriale
- $T_{parallel}(p)$ = tempo di esecuzione del programma parallelo su p processori o core

Allora lo speedup $S(p)$ è definito come segue:

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)}$$

Ad esempio, se un numero RSA venisse risolto in un secondo con l'utilizzo del programma seriale e in 0,5 secondi con il programma parallelo utilizzando 2 core, lo speedup relativo all'uso di 2 core sarebbe $1/0,5 = 2$. In questo esperimento il tempo di esecuzione seriale è in realtà il tempo di esecuzione del programma parallelo su un core.

Prima di presentare i risultati si mostrano i numeri RSA sottoposti alle elaborazioni.

Numeri RSA:
RSA7 = 3445889
RSA8 = 25734383
RSA9 = 522480911
RSA10 = 1521954233
RSA11 = 11702621389
RSA12 = 310181376187
RSA13 = 3481959031631
RSA14 = 31322658991751
RSA15 = 153682055133511
RSA16 = 2331870651543353

Soluzioni RSA:

RSA7 = 4481 * 769 (4 * 3 = 7 cifre)
RSA8 = 4481 * 5743 (4 * 4 = 8 cifre)

RSA9 = 90977 * 5743 (5 * 4 = 9 cifre)
RSA10 = 90977 * 16729 (5 * 5 = 10 cifre)
RSA11 = 16729 * 699541 (6 * 5 = 11 cifre)
RSA12 = 699541 * 443407 (6 * 6 = 12 cifre)
RSA13 = 699541 * 4977491 (7 * 6 = 13 cifre)
RSA14 = 4977491 * 6292861 (7 * 7 = 14 cifre)
RSA15 = 24421651 * 6292861 (8 * 7 = 15 cifre)
RSA16 = 46761881 * 49866913 (8 * 8 = 16 cifre)

I tempi di risoluzione di RSA7, RSA8 e RSA9 sono risultati minori di $0,1$ secondi in ogni versione dell'applicazione, per ogni diversa architettura. I tempi misurati non avevano differenze neanche utilizzando OpenMP con numeri di core diversi, per questo si è deciso di tralasciare queste misurazioni e di non inserirli nei grafici.

6.1 Test su CPU

In questa sezione vengono riportati i grafici dei test relativi alla versione dell'applicazione implementata in OpenMP.

6.1.1 Architettura cluster

Dal grafico 6.1 possiamo notare un tempo di risoluzione crescente al crescere della lunghezza del numero RSA; infatti si passa da RSA14 dove si impiega circa un secondo per la risoluzione con 12 core, a RSA16 dove se ne impiega circa 3,5 secondi. Un altro aspetto rilevante è la buona scalabilità dell'applicazione parallela fino a 6 core, dopodiché, i tempi di risoluzione diventano più alti. Ciò può essere causato dal fatto che i due processori Xeon hanno 6 core ognuno, dividendo quindi la computazione fino a 6 thread in realtà si utilizza uno solo dei due processori; dividendo invece la computazione per un numero maggiore di 6, l'elaborazione va ad utilizzare tutti e due i processori: questo probabilmente crea un costo di accesso alla memoria dei 2 processori più alto

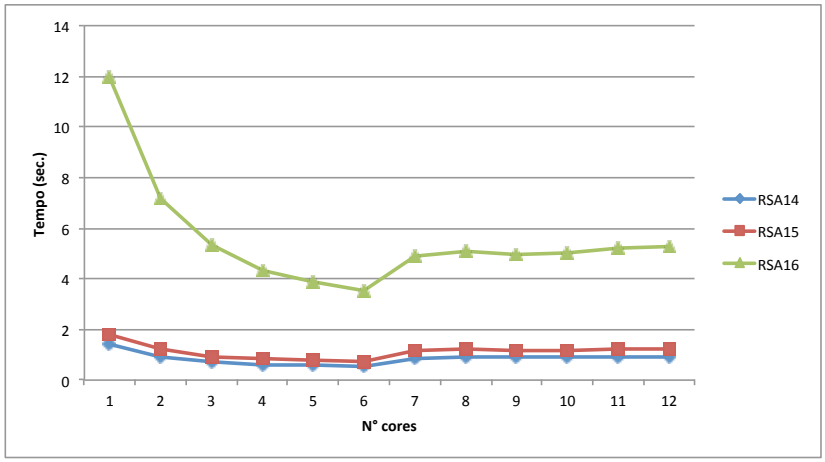

Figura 6.1: Tempi di risoluzione in funzione del diverso numero di core utilizzati, eseguendo la versione dell'applicazione in OpenMP, su architettura cluster per RSA14, RSA15 e RSA16

del vantaggio di utilizzare un numero di thread maggiore di 6. Anche dal grafico 6.2, il quale riporta la funzione di speedup dell'applicazione eseguita su diversi core(da 1 a 12), possiamo notare il calo di prestazioni dall'utilizzo di 7 core in poi. Sempre dal grafico 6.2 notiamo come lo speedup delle 3 curve abbia migliori prestazioni con numeri RSA più difficili da risolvere; questo probabilmente è dovuto al fatto che le iterazioni dell'algoritmo si ripetono più volte, più il numero è difficile da risolvere. Ad ogni nuova iterazione i parametri in ingresso all'algoritmo aumentano e le strutture dei dati vanno in crescendo proporzionalmente ai parametri. Più le strutture dati da gestire sono grandi più viene in aiuto la parallelizzazione.

6.1.2 Architettura a basso consumo

Dal grafico 6.3 mostra come anche in questo caso, il tempo di risoluzione sia crescente al crescere della lunghezza del numero RSA. Dal grafico 6.4 notiamo che l'applicazione riesce a scalare bene fino all'utilizzo di tutti e 4 i core, inoltre che le curve degli speedup sono quasi rette a 45 gradi, il

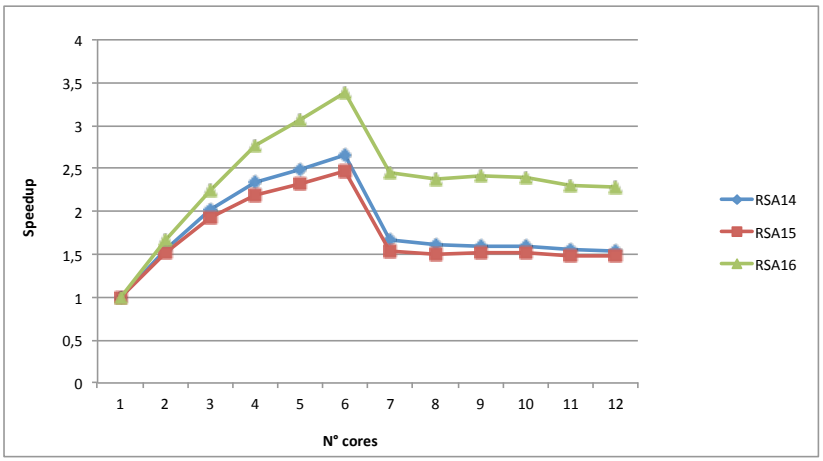

Figura 6.2: Speedup in funzione di diverso numero di core utilizzati, eseguendo la versione dell'applicazione in OpenMP, su architettura cluster per la risoluzione di RSA14, RSA15 e RSA16

che significa una parallelizzazione quasi ottimale. Prendendo ad esempio lo speedup di risoluzione di RSA16, notiamo come la retta arrivi a fattore 3 con l'utilizzo di 4 core: una scalabilità accettabile e più alta della stessa misurata nel test con l'architettura cluster; quest'ultima con quattro core raggiungeva un fattore di speedup di circa 2.7.

6.2 Test su GPU

In questa sezione vengono riportati i grafici dei test relativi alla versione dell'applicazione implementata in CUDA. Il grafico 6.5 riporta i tempi di risoluzione di vari numeri RSA nelle due diverse architetture. Anche in questa versione dell'applicazione, i tempi di risoluzione aumentano al crescere del numero di cifre dei numeri RSA. Le due architetture riescono a risolvere i numeri RSA utilizzando circa lo stesso tempo fino ad RSA15; in realtà si nota come l'architettura a basso consumo riesce a risolvere i numeri RSA più piccoli in tempi leggermente più bassi dell'architettura cluster. Al contrario

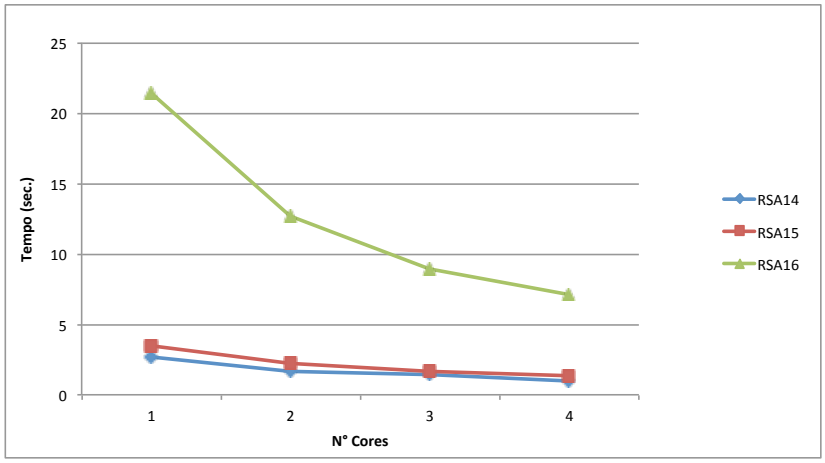

Figura 6.3: Tempi di risoluzione in funzione del diverso numero di core utilizzati, eseguendo la versione dell'applicazione in OpenMP, su architettura a basso consumo per RSA14, RSA15 e RSA16

con RSA16, il numero più difficile da risolvere, l'architettura cluster impiega circa 2 secondi in meno rispetto all'architettura a basso consumo.

6.3 Confronto tra le due architetture

In questa sezione si vogliono confrontare le architetture, non solo secondo il punto di vista delle tempistiche di risoluzione, ma misurando anche il loro consumo energetico al momento dell'elaborazione. Per eseguire questo test si sono misurati i consumi di energia delle architetture durante la risoluzione dei numeri RSA. Per l'architettura cluster si è misurato un assorbimento di 350 +/- 20 watt durante l'esecuzione dell'applicazione, sia nella versione per CPU che per GPU. Nel seguito useremo il valore di 350 watt. Per l'architettura a basso consumo è stato utilizzato un multimetro collegato direttamente tra la scheda Jetson k1 e il suo alimentatore. La figura 6.6 mostra come è stato effettuato il test: a sinistra vi è il computer, collegato con un account sulla scheda Jetson K1, con cui viene lanciata l'applicazione, a destra vi è il collegamento tra la scheda Jetson K1, il multimetro con cui si stano misurando gli

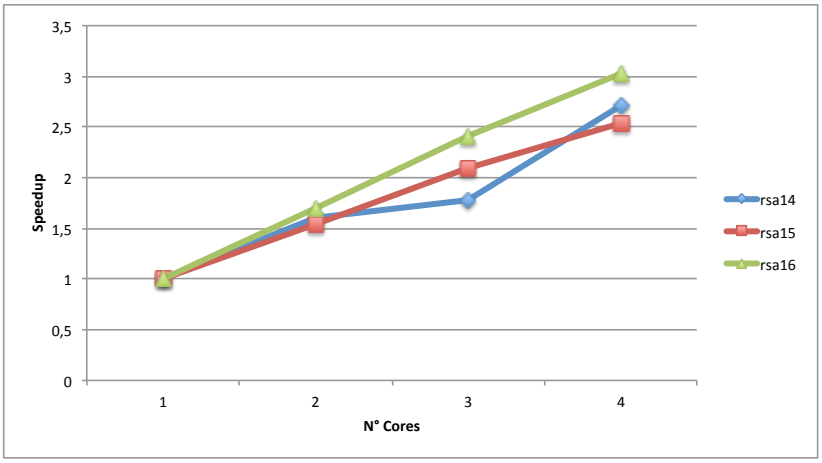

Figura 6.4: Speedup in funzione di diverso numero di core utilizzati, eseguendo la versione dell'applicazione in OpenMP, su architettura a basso consumo per la risoluzione di RSA14, RSA15 e RSA16

ampere[1] assorbiti dalla scheda durante l'elaborazione e l'alimentatore utilizzato per fornire la corrente necessaria. Noti i volt[2] necessari per alimentare la scheda (12 V) e gli ampere assorbiti durante l'elaborazione, per calcolare la potenza necessaria alla scheda Jetson K1 si è applicata la seguente formula:

$$P(watt) = V(volt) * I(ampere)$$

Il grafico 6.7 ha lo scopo di confrontare le due architetture, riportando i tempi di risoluzione di RSA16 più veloci per ogni architettura, nelle due versioni dell'applicazione. Il tempo più basso è dato dai processori Xeon dell'architettura cluster che, con l'utilizzo di 6 core, riesce a fattorizzare RSA16 in 3.5 secondi circa. Utilizzando invece la GPU Tesla K20 l'architettura impiega circa il doppio del tempo. L'architettura a basso consumo impiega invece circa 7 secondi, stesso tempo della K20, utilizzando tutti e 4 core che ha a disposizione nel processore e circa 9 secondi con l'utilizzo della GPU. I risultati sono sorprendenti se si pensa che la Tegra K1 ha attualmente un

[1]Unità base utilizzata per misurare l'intensità della corrente elettrica
[2]Unità di misura del potenziale elettrico e della differenza di potenziale

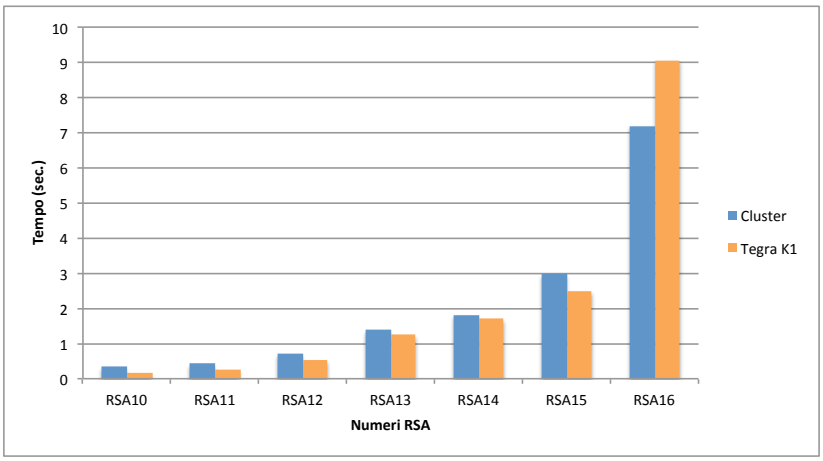

Figura 6.5: Tempi di risoluzione in funzione di diversi numeri RSA, eseguendo la versione dell'applicazione in CUDA, su architettura cluster e architettura a basso consumo

prezzo di commercio di circa 180 Euro; mentre per assemblare una macchina con caratteristiche simili all'architettura cluster, si arriva a spendere più di 3000 Euro solo per il processore Xeon e la GPU Tesla K20.

Il grafico 6.8 vuole confrontare l'effettiva energia consumata durante l'elaborazione della risoluzione di RSA16, per entrambi le architetture. Da esso si evince come le architetture a basso consumo sembrano realizzare ciò che promettono, avendo consumi molto inferiori a quelli dell'architettura cluster; fare attenzione alla scala logaritmica dell'asse y del grafico. Con l'architettura a basso consumo, per la risoluzione di RSA16 si assorbono circa, 8.4 watt impiegando circa 7 secondi utilizzando la CPU e 7 watt impiegando circa 9 secondi con l'utilizzo della GPU. Consumi che sono circa di due ordini di grandezza inferiori a quelli dell'architettura cluster.

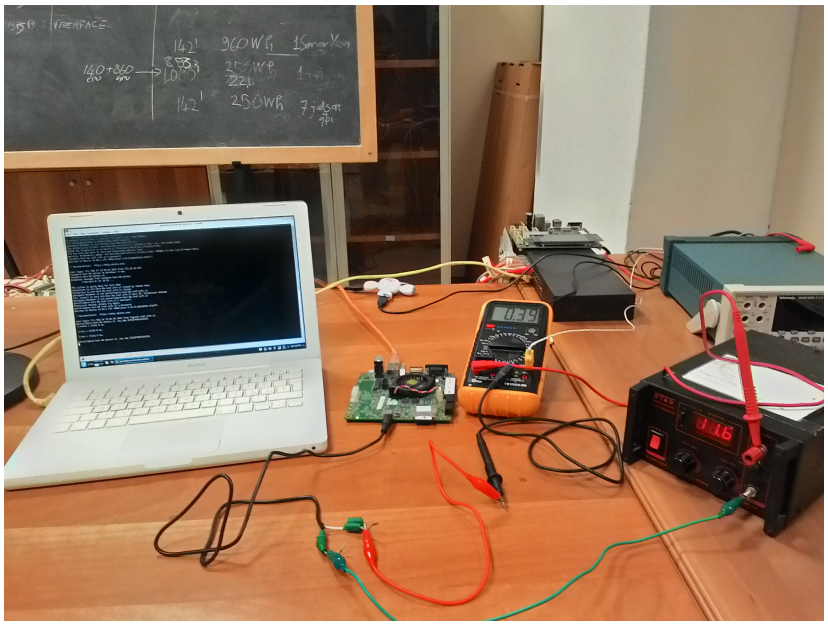

Figura 6.6: Test dell'applicazione su architettura a basso consumo

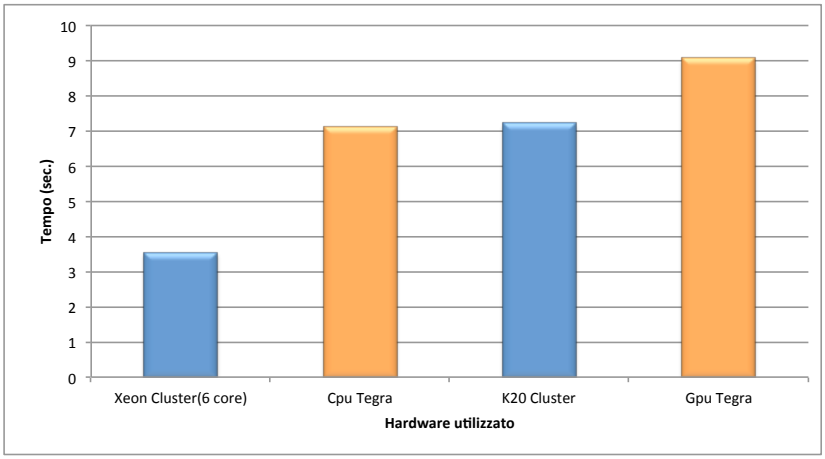

Figura 6.7: Tempo di risoluzione in secondi di RSA16 delle diverse tecnologie hardware (in blu dell'architettura cluster e in arancione dell'architettura a basso consumo)

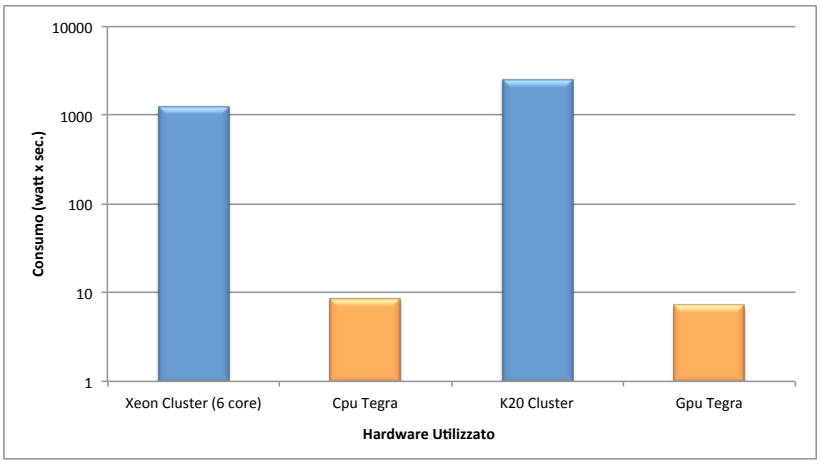

Figura 6.8: Consumi in watt * secondi necessari alla risoluzione di RSA16 delle diverse tecnologie hardware

Conclusioni e Sviluppi Futuri

In questo lavoro è stata realizzata un'applicazione parallela per la fattorizzazione di numeri RSA in due differenti versioni: la prima versione, realizzata con l'estensione OpenMP, è servita a studiare il comportamento del processore dell'architettura a basso consumo Tegra K1, andandolo a confrontare con quello di un cluster ad alta potenza di calcolo. La seconda versione, realizzata in CUDA, ha permesso lo studio del comportamento delle GPU presenti nelle due architetture. I risultati hanno mostrato come la Tegra K1 riesca ad avere prestazioni più basse rispetto ad architetture più potenti, con ridotti consumi di energia. Inoltre la Tegra K1, avendo prezzi di mercato molto bassi, consente all'utente un notevole risparmio. Ovviamente queste considerazioni valgono per l'utilizzo del nostro tipo di applicazione. Questa applicazione non è in grado di saturare i core delle GPU, quindi di utilizzarle al massimo delle loro potenzialità e sarebbe infatti utile effettuare uno studio simile, utilizzando applicazioni di differente tipo.

Questo lavoro potrà essere uno spunto per altri studi che abbiano come soggetto architetture a basso consumo; ad esempio sarebbe interessante realizzare dei piccoli cluster con l'utilizzo di più Tegra K1 in parallelo. Inoltre lo stesso studio potrebbe essere fatto su altri sistemi SoC presenti sul mercato: altri produttori oltre alla NVIDIA stanno proponendo diversi prodotti con differenti prezzi e prestazioni.

Il lavoro effettuato ha mostrato alcune lacune in CUDA, le quali potrebbero essere motivo di studio; la prima lacuna è l'assenza di una libreria solida e completa che permetta l'utilizzo di BigInteger, la seconda è l'impossibilità

di chiamata ad una funzione parallela da una funzione già interna al Device, nei processori ARM [8].

Bibliografia

[1] http://openmp.org/wp/

[2] http://www.nvidia.com/object/cuda_home_new.html

[3] Carl Pomerance, Analysis and Comparison of Some Integer Factoring Algorithms, in Computational Methods in Number Theory, Part I, H.W. Lenstra, Jr. and R. Tijdeman, eds., Math. Centre Tract 154, Amsterdam, 1982, pp 89-139

[4] Gary Miller, Riemann's Hypothesis and Tests for Primality in J. Comput. System Sci., vol. 13, nº 3, 1976, pp. 300-317

[5] A. M. Legendre Essai sur la theorie des nombres Paris 1798, p 186

[6] Bathe, Klaus-Jürgen, and Edward L. Wilson. Numerical methods in finite element analysis. (1976): 6-12.

[7] https://gmplib.org/

[8] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz3SwfksQ1E