UNIVERSITÀ DEGLI STUDI DI MILANO

Dipartimento di Informatica

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

PROGETTO DI GPU COMPUTING

Parallelizzazione dell'algoritmo Advanced Encryption Standard (AES) su architettura CUDA

A cura di
Andrea Ceccarelli
Tommaso Celata

Docente
Giuliano Grossi

Anno Accademico 2015/2016

Indice

Capitolo 1

Introduzione

Il progetto da noi realizzato si pone l'obiettivo di verificare i vantaggi che l'architettura CUDA può portare nella parallelizzazione di algoritmi che presentano una sostanziosa parte seriale che può essere eseguita parallelamente. Per tale motivo si è scelto l'algoritmo Advanced Encryption Standard **(AES)** che cifra stati di dimensione fissa senza concatenare i risultati tra loro dandoci la possibilità di ottenere un buon livello di parallelizzazione.

In una prima fase si è implementato l'algoritmo in linguaggio c verificandone la corretta esecuzione tramite vettori di test trovati online, poi si è modificato il codice per adattarsi e sfruttare l'architettura **CUDA**. Una prima implementazione non è stata sufficiente per ottenere dei vantaggi rispetto alla versione **C**, questo dovuto al fatto che venivano lanciati **troppi kernel** che creando un collo di bottiglia rendevano inutile il vantaggio portato dalla parallelizzazione. Andando avanti con le varie versioni si è diminuito sostanzialmente il numero di kernel lanciati arrivando ad ottenere buoni risultati.

Spiegheremo quindi le caratteristiche delle varie versioni implementate concentrandoci sull'argomento **parallelizzazione** e su quali vantaggi (o svantaggi) si sono ottenuti da una versione all'altra.

Capitolo 2

Advanced Encryption Standard (AES)

2.1 Caratteristiche

Sviluppato dai due crittografi belgi Joan Daemen e Vincent Rijmen l'Advanced Encryption Standard (AES), conosciuto anche come Rijndael, di cui più propriamente è una specifica implementazione, è un algoritmo di cifratura a blocchi utilizzato come standard dal governo degli Stati Uniti d'America.

Data la sua sicurezza e le sue specifiche pubbliche si presume che in un prossimo futuro venga utilizzato in tutto il mondo come è successo al suo predecessore, il Data Encryption Standard (DES) che ha perso poi efficacia per vulnerabilità intrinseche. AES è stato adottato dalla National Institute of Standards and Technology (NIST) e dalla US FIPS PUB nel novembre del 2001 dopo 5 anni di studi, standardizzazioni e selezione finale tra i vari algoritmi proposti.

2.1.1 La chiave di sessione

AES usa un **key schedule** per espandere una chiave primaria corta in un certo numero di chiavi di **ciclo** differenti. La lunghezza della chiave ricavata dipende dal livello di sicurezza con cui vogliamo criptare i dati.

Per il livello SECRET è sufficiente una chiave a 128 bit mentre per il livello Top secret si consigliano chiavi a 192 o 256 bit. Questo significa che per la prima volta il pubblico ha accesso ad una tecnologia crittografica che NSA ritiene adeguata per proteggere i documenti TOP SECRET. Si è discusso sulla necessità di utilizzare chiavi lunghe (192 o 256 bit) per i documenti TOP SECRET.

Alcuni ritengono che questo indichi che l'NSA ha individuato un potenziale attacco che potrebbe forzare una chiave relativamente corta (128 bit), mentre la maggior parte degli esperti ritiene che le raccomandazioni della NSA siano basate principalmente sul volersi garantire un elevato margine di sicurezza per i prossimi decenni contro un potenziale attacco esaustivo.

2.1.2 Lo stato

A differenza di **Rijndael**, algoritmo da cui AES deriva e che utilizza blocchi multipli di 32 bit a partire da 128 fino ad arrivare a 256, AES utilizza matrici di 4x4 byte chiamate **stati**.

Dati 16 bit di informazione $b_0, b_1, ..., b_15$, verranno disposti nella matrice come segue

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

2.2 Descrizione dell'algoritmo

L'algoritmo sfrutta diverse funzioni che modificano i valori della matrice di stato mantenendone però la dimensione.

L'AES effettua:

- 10 round per la chiave a 128 bit
- 12 round per la chiave a 192 bit
- 14 round per la chiave a 256 bit

Ogni round consiste in diversi step, ognuno dei quali contiene a sua volta quattro diverse fasi, inclusa una che sfrutta la chiave di ciclo. Questo consente di passare da un testo in chiaro *plaintext* a un testo cifrato *ciphertest*.

1. **KeyExpansions** La chiave di cifratura viene espansa dal key schedule per generare una chiave più grande contenente tutte le chiavi di ciclo.
2. **Round iniziale**
   (a) *AddRoundKey* Ogni byte dello stato viene combinato con il byte corrispondente della chiave di ciclo tramite uno XOR

3. **Rounds**
   (a) *SubBytes* Ogni byte viene sostituito con un altro secondo delle tabelle.
   (b) *ShiftRows* Una trasposizione dove le ultime tre righe dello stato sono shiftate a sinistra un certo numero di volte.
   (c) *MixColumns* Opera sulle colonne combinandone i byte.
   (d) *AddRoundKey*
4. **Round Finale (senza MixColumns)**
   (a) *SubBytes*
   (b) *ShiftRows*
   (c) *AddRoundKey*

2.2.1 AddRoundKey

Nella fase AddRoundKey, la chiave di sessione viene combinata con lo stato. Per ogni round viene derivata una sottochiave dalla chiave originaria usando il key schedule; ogni sottochiave è della stessa dimensione dello stato. La sottochiave è aggiunta allo stato combinando ogni byte di questo con il corrispondente byte della chiave con uno **XOR**.

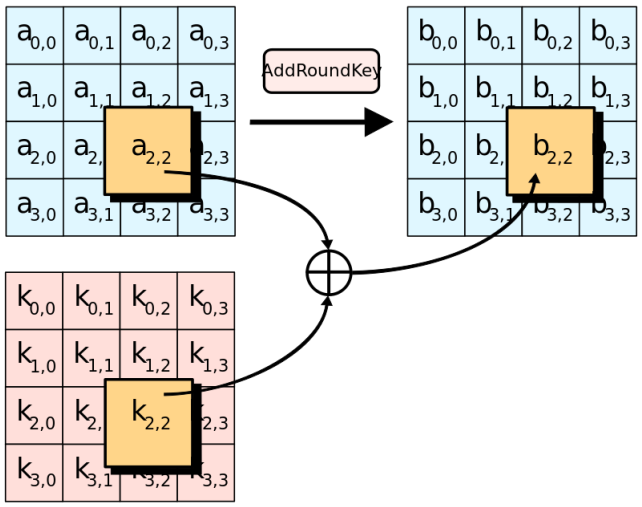

Figura 2.1: Nel passaggio AddRoundKeys ogni byte della matrice viene combinato con la sua sottochiave tramite un'operazione di XOR.

2.2.2 SubBytes

In SubBytes ogni byte $a_{i,j}$ della matrice di stato è sostituito con il corrispondente byte di una matrice chiamata **S-box**.

$$b_{i,j} = S(a_{i,j})$$

Questa operazione garantisce la non-linearità della cifratura. La S-box utilizzata è derivata da una funzione inversa nel campo finito GF(2^8), conosciuta per avere delle ottime proprietà di non linearità. Per evitare un potenziale attacco basato sulle proprietà algebriche la S-box è costruita combinando la funzione inversa con una trasformazione affine invertibile. La S-box è stata scelta con cura per non possedere né punti fissi né punti fissi opposti.

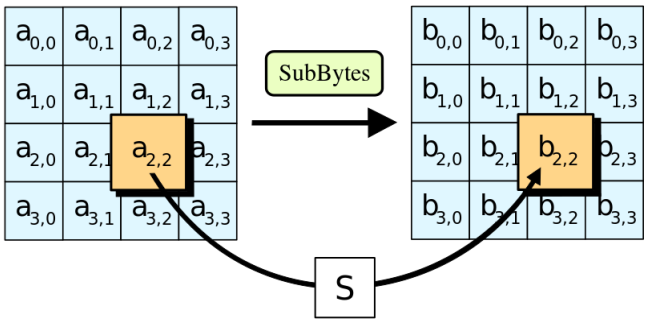

Figura 2.2: Nel passaggio SubBytes, ogni byte della matrice è sostituito con i dati contenuti nella trasformazione S; $b_{ij} = S(a_{ij})$.

2.2.3 ShiftRows

ShiftRows opera sulle righe dello stato shiftando ciclicamente i byte di ogni riga di un certo offset dipendente dal numero di riga lasciando la prima riga invariata. Ogni byte della seconda riga è ciclicamente spostato a sinistra di una posizione. Similmente i byte della terza riga sono shiftati ciclicamente a sinistra di due posizioni e quelli della quarta di tre.

In questo modo l'ultima colonna dei dati in ingresso andrà a formare la diagonale della matrice in uscita. (Rijndael utilizza un disegno leggermente diverso per via delle matrici di lunghezza non fissa.)

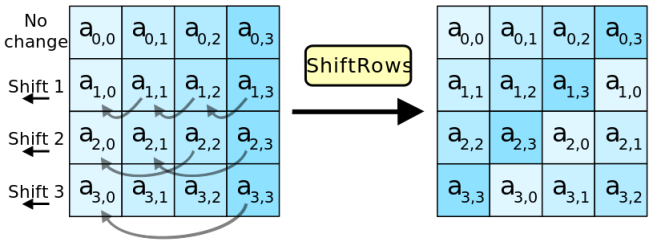

Figura 2.3: Nel passaggio ShiftRows, i byte di ogni riga vengono spostati verso sinistra dell'ordine della riga. Vedi figura per i singoli spostamenti.

2.2.4 MixColumns

Il passaggio MixColumns prende i quattro byte di ogni colonna e li combina utilizzando una trasformazione lineare invertibile. Utilizzati in congiunzione, ShiftRows e MixColumns provvedono a far rispettare il criterio di confusione e diffusione nell'algoritmo (teoria di Shannon). Ogni colonna è trattata come un polinomio in $GF(2^8)$ e viene moltiplicata modulo $x^4 + 1$ per un polinomio fisso $c(x) = 3x^3 + x^2 + x + 2$.

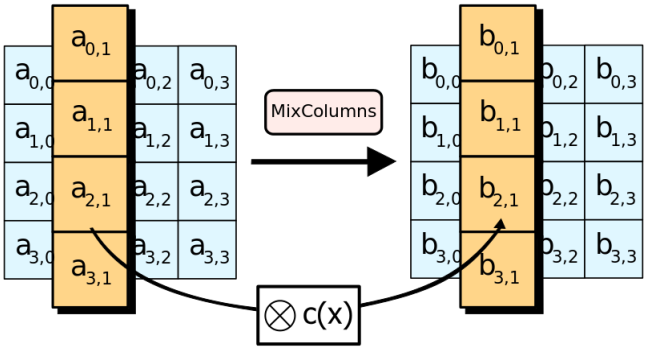

Figura 2.4: Nel passaggio MixColumns ogni colonna di byte viene moltiplicata per un polinomio fisso $c(x)$.

Capitolo 3

Architettura CUDA

3.1 Cos'è CUDA

CUDA (Compute Unfied Device Architecture) è un architettura hardware creata da NVIDIA e pensata per l'elaborazione parallela che si basa dunque sulla suddivisione di un problema in sottoproblemi e lo svolgimento di ognuno di questi concorrentemente. Tutto ciò è possibile sfruttando la potenza di calcolo della GPU (unità di elaborazione grafica) che assieme alla CPU permette di rendere molto efficiente in termini di tempo l'esecuzione di molte applicazioni. Il calcolo parallelo su GPU offre prestazioni senza precedenti caricando porzioni **compute-intensive** dell'applicazione sulle GPU, mentre la porzione rimanente di codice continua ed eseguire su CPU. Pertanto, le GPU devono operare congiuntamente con le CPU mediante bus PCI-Express

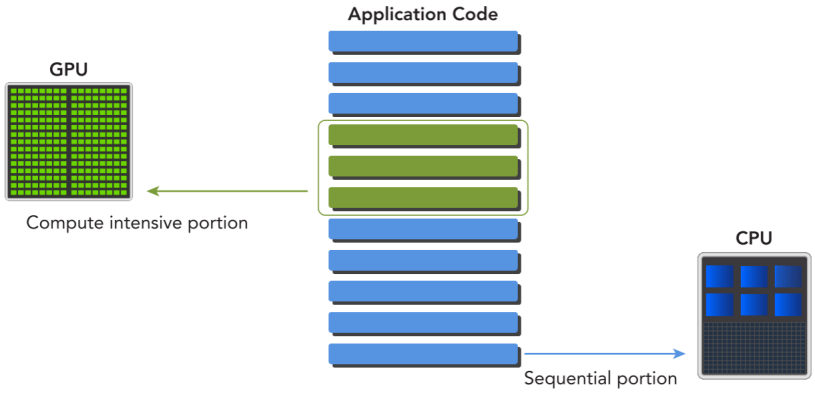

Figura 3.1: Divisione delle operazioni tra GPU e CPU in ambiente CUDA

Le applicazioni in cui CPU e GPU lavorano insieme consistono di due parti:

- codice Host
- codice Device

Il codice Host esegue su **CPUs** e il codice device esegue su **GPUs**. Il codice CPU è responsabile della gestione dell'ambiente, dell'IO e della gestione dei dati per il device stesso, prima di caricare task intensivi sul device. La GPU è usata per accelerare l'esecuzione di questa porzione di codice basandosi sul parallelismo dei dati. La CPU è ottimizzata per sequenze di operazioni in cui il controllo del flusso è impredicibile; le GPU lo sono per carichi dominati da semplice flusso di controllo.

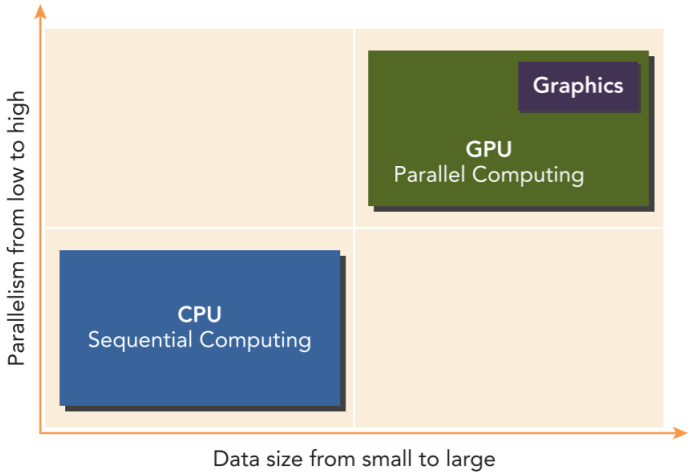

Figura 3.2: Prestazioni tra GPU e CPU in relazione a grado di parallelismo e quantità dei dati da elaborare.

3.2 Il modello di programmazione CUDA

CUDA come tutti i modelli di programmazione fornisce un' **astrazione** della macchina e si pone come ponte tra il software e l'hardware. Il programma, scritto seguendo il modello di programmazione, definisce come condividere le informazione e coordinare le attività mentre fornisce una visione logica del computer attraverso strumenti come il compilatore, le librerie, sistema operativo e primitive hardware.

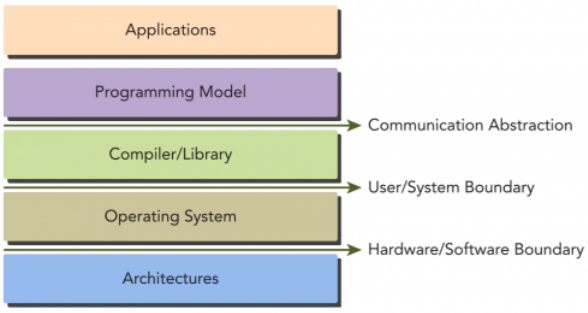

Figura 3.3: Prestazioni tra GPU e CPU in relazione a grado di parallelismo e quantità dei dati da elaborare.

In particolare il modello CUDA offre una visione dell'architettura GPU per sfruttarne a pieno la potenza di calcolo:

- Struttura gerarchica dell'organizzazione dei Threads
- Struttura gerarchica dell'organizzazione della memoria

3.2.1 Struttura di un programma CUDA

La struttura di un programma CUDA evidenzia la coesistenza di un host (CPU) e **uno o più** devices (GPU). Le funzioni che devono essere eseguite dal device sono integrate nel sorgente C e indicano al device quali operazioni effettuare, su quali dati e con quale grado di parallelismo svolgerle. Il modello CUDA quindi permette l'esecuzione di applicazioni su modelli di calcolo **eterogenei** (CPU + GPU) semplicemente annotando il codice con un piccolo insieme di estensioni del linguaggio C separando:

- *HOST*: CPU+ Memoria (**Host** memory)
- *DEVICE*: GPU + Memoria (**Device** memory)

Kernel

Il codice eseguito sul device è chiamato kernel ed è, come detto, denotato da opportune keyword CUDA. Di fatto si tratta di una procedura sequenziale che CUDA gestisce in modo concorrente mediante lo scheduling dei thread sui vari core. Il modello CUDA è **asincrono** permettendo così che le computazioni su GPU e CPU siano sovrapponibili e gestite da comunicazioni host-device

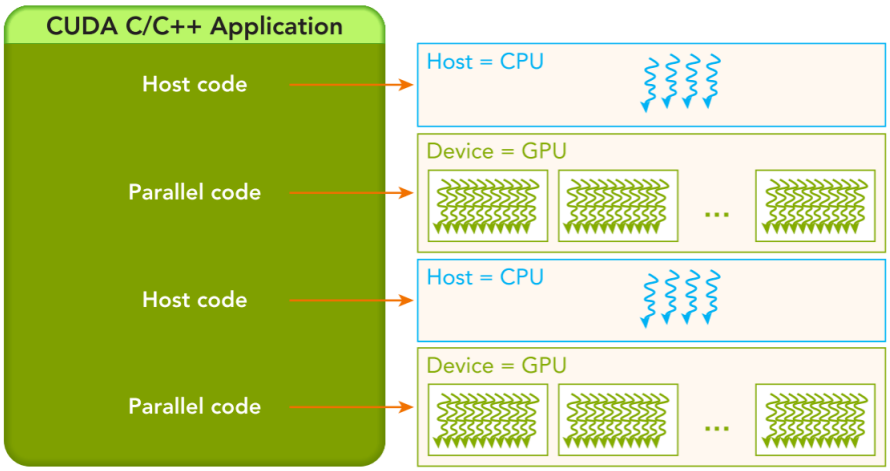

Figura 3.4: Prestazioni tra GPU e CPU in relazione a grado di parallelismo e quantità dei dati da elaborare.

3.2.2 Flusso tipico

Il tipico flusso di un programma CUDA è composto di tre fasi:

- Copia dei dati dalla CPU memory alla GPU memory
- Invocazione dei kernel per operare su dati memorizzati in GPU memory
- Copa dei dati dalla GPU memory alla CPU memory

Il codice per il device viene compilato a runtime dal compilatore NVCC mentre il codice C/C++ dal compilatore C ed eseguito su CPU. Quando un kernel viene lanciato, viene eseguito da un elevato numero di thread, suddivisi in blocchi che collettivamente vengono chiamati **griglia**.

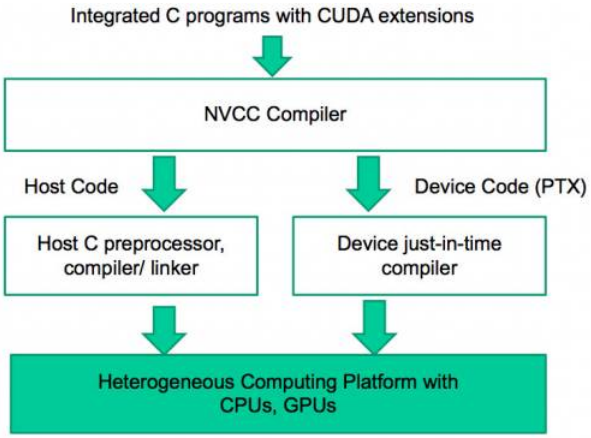

Figura 3.5: Prestazioni tra GPU e CPU in relazione a grado di parallelismo e quantità dei dati da elaborare.

Capitolo 4

Parallelizzazione AES in CUDA

Per poter ottenere buoni risultati nel campo del GPU Computing c'è bisogno di un algoritmo che ci consenta un sufficiente grado di parallelizzazione. La scelta di AES infatti non è casuale: basti pensare a come l'algoritmo opera **indipendentemente** su uno stato o su un altro senza concatenare le due cose e subito ci accorgiamo che consente un alto grado di **parallelizzazione**. L'obiettivo del nostro progetto è quindi quello di dimostrare come l'alto grado di parallelizzazione offerta dall'architettura CUDA possa essere utile nell'algoritmo AES.

Implementazione in C In una prima fase abbiamo sviluppato l'algoritmo in C per poterlo usare poi come confronto con la versione in CUDA. Sia la versione C che la versione CUDA seguono le seguenti fasi:

- Lettura del **plainText**, il testo in chiaro scritto in esadecimale
- Lettura di una chiave da 16 caratteri esadecimali e generazione di una chiave **estesa** da 176 sufficente per 11 round
- Separazione del plainText in stati da 16 elementi.
- Esecuzione dell'algoritmo su ogni stato
- Controllo della corretta esecuzione tramite confronto con vettore di test
- Salvataggio dei risultati per fattore tempo

Per quanto riguarda i risultati in fattore tempo questi vengono calcolati comprendendo non solo la fase vera e propria dell'esecuzione di AES ma anche quella di separazione degli stati visto che nel caso GPU questa comprende

anche il passaggio dei dati da **host memory** (memoria centrale) a **GPU memory**. Abbiamo voluto includere nelle prestazioni questa fase considerando che la memoria è un noto *bottleneck* da considerare.

4.1 Prima implementazione in CUDA - Un solo stato

Una prima implementazione dell'algoritmo in CUDA non considerava una pluralità di stati ma si concentrava solo sulla parallelizzazione delle **funzioni interne** quali subBytes, addRoundKey ecc... Purtroppo i risultati furono scoraggianti, infatti nonostante fossimo riusciti a replicare il funzionamento di AES in CUDA verificandone il successo con il vettore di test, i tempi erano di molto peggiori rispetto alla versione C.

Perchè? Abbiamo pensato che un solo stato non fosse sufficiente per evidenziare un miglioramento nella nostra versione e che questo fosse dovuto proprio a un bottleneck della memoria.

4.2 Seconda implementazione in CUDA - Più stati

Sulla considerazione precedente abbiamo applicato l'algoritmo a più stati contemporaneamente criptando porzioni di testo maggiori e non più limitate a 16 caratteri esadecimali. In questo modo non solo abbiamo aumentato le dimensioni dei test ma abbiamo anche ottenuto una parallelizzazione più "esterna", in aggiunta a quella già implementata.

Vediamo un esempio di come abbiamo parallelizzato le funzioni sugli stati utilizzate da AES.

```
1  void addRoundKey(uChar** state, int cur_round) {
2    for (int i = 0; i < columns; i++) {
3      for (int j = 0; j < rows; j++) {
4        int val = (cur_round * 16) + ((i * 4) + j);
5        state[j][i] = state[j][i] ^ expanded_key[val];
6
7      }
8    }
9
10
```

Listing 4.1: Versione sequenziale di addRoundKey

```
__device__ void addRoundKey(uChar* state, int cur_round, uChar*
    expanded_keyGPU) {
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int index = Col * M + Row;
  int val = (cur_round * 16) + index;
  state[index] = state[index] ^ expanded_keyGPU[val];
}
```

Listing 4.2: Versione parallela di addRoundKey

Nella versione sequenziale lo xor tra stato e chiave espansa viene effettuato con due cicli for con un costo di $O(n)$ dove n è la dimensione dello stato quindi 16. Nella versione sequenziale, sfruttando i thread arriviamo invece ad un $O(1)$

Purtroppo neanche questa versione è risultata conveniente dando risultati peggiori della versione C.

4.3 Ultima implementazione

Utilizzando Nsight ci siamo resi conto dell'alto numero di kernel che lanciavamo per ogni test considerando che ad ogni stato veniva lanciato un kernel e che eseguivamo test da un minimo di 15 mila stati circa fino a un massimo di 2 milioni circa. L'idea è stata quella di utilizzare un solo kernel con un sufficiente numero di thread da servire tutti gli stati. Questa soluzione si è poi rivelata quella vincente consentendoci di arrivare a un tempo più o meno **costante** di esecuzione e superando di parecchio la versione C, come vedremo nel prossimo capitolo.

Capitolo 5

Risultati

Capitolo 6

Considerazioni

Bibliografia

[1] Kenneth Price , Rainer M. Storn , Jouni A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, 2005