UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II
Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

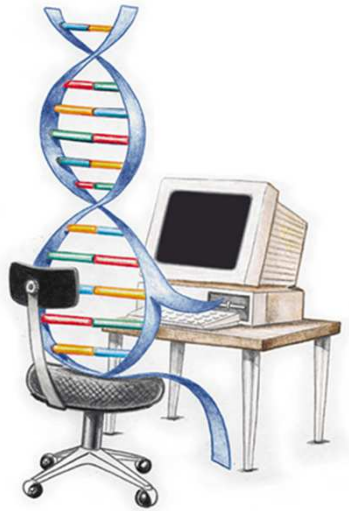
GPGPU for machine learning algorithms

Anno Accademico 2010/2011

relatori
Ch.mo prof. Giorgio Ventre
Ch.mo prof. Antonio Pescapè

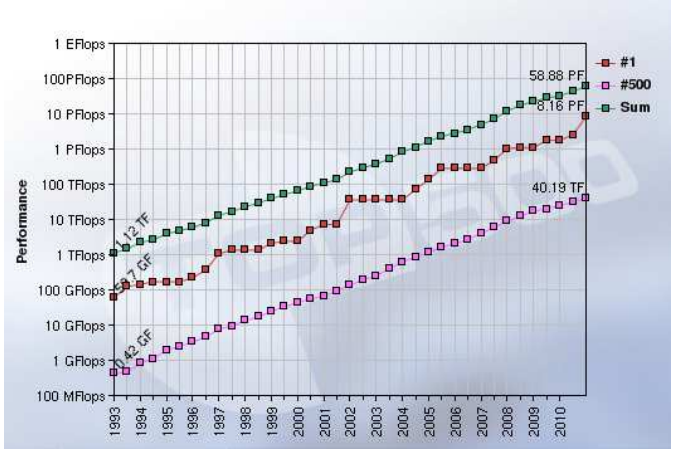
correlatore
Ch.mo dott. Massimo Brescia

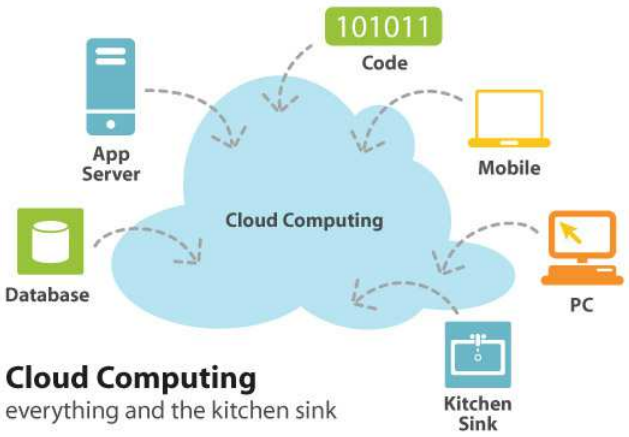
candidato
Mauro Garofalo
Matr. 041/2780

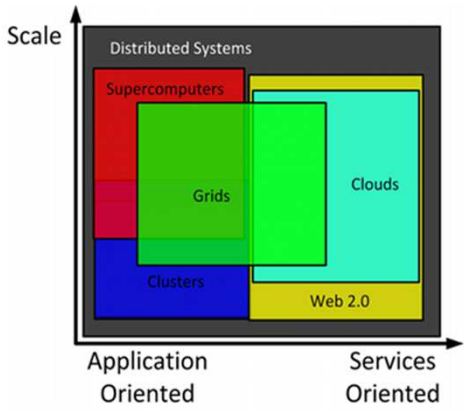


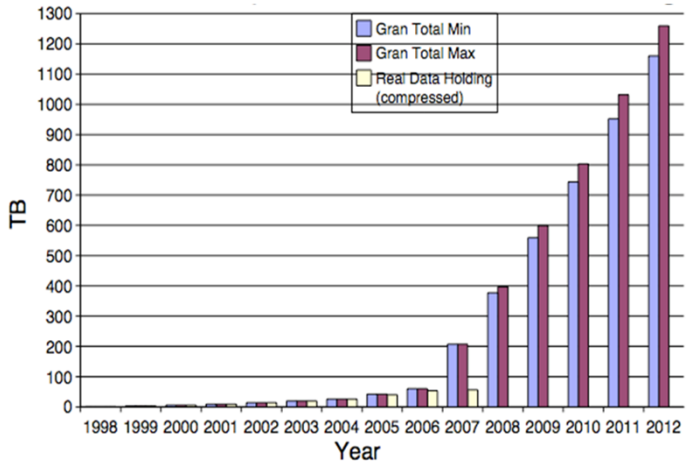
Problema

- I dati derivanti da problemi scientifici sono aumentati in modo esponenziale negli ultimi anni, a differenza della potenza di calcolo che è aumentata linearmente.
- Questo «tsunami» di dati ha comunque portato alla nascita di nuove architetture per il calcolo ad alte prestazioni.
- Le tecnologie e i servizi messi a disposizione da HPC, Grid e Cloud sono diventati strumenti fondamentali per la ricerca scientifica.





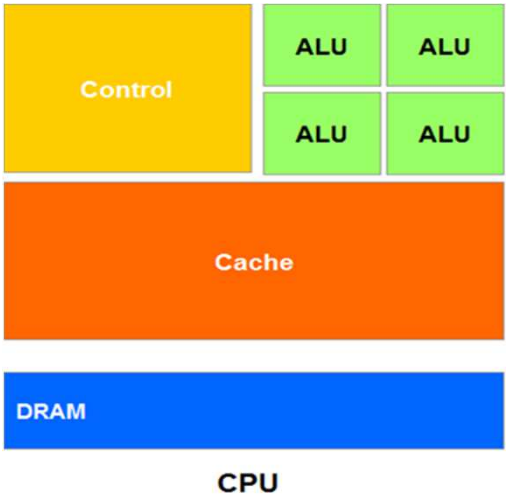




Contributo del lavoro di tesi/Sommario

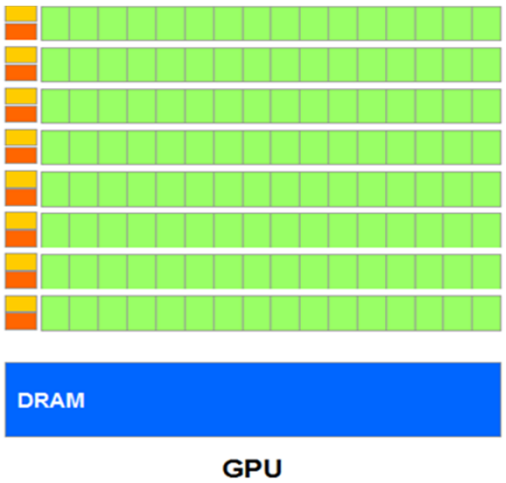
- **Studio dello stato dell'arte delle tecnologie di computing più utilizzate in campo scientifico**
  - **HPC (Cluster), sistemi distribuiti (Grid e Cloud) e GPGPU**
- **Studio delle moderne architetture GPU, delle loro capacità prestazionali e delle attuali tecniche di GPU Computing**
  - **In particolare GPU con tecnologia NVIDIA CUDA**
- **Design e implementazione di un algoritmo genetico su architettura CUDA**
  - **Derivato da GAME, un algoritmo genetico elaborato all'interno del Programma DAME (Data Mining & Exploration, http://dame.dsf.unina.it), su un'infrastruttura distribuita ibrida** (Brescia et al. 2011b)

CPU vs GPU



Multi-core CPU

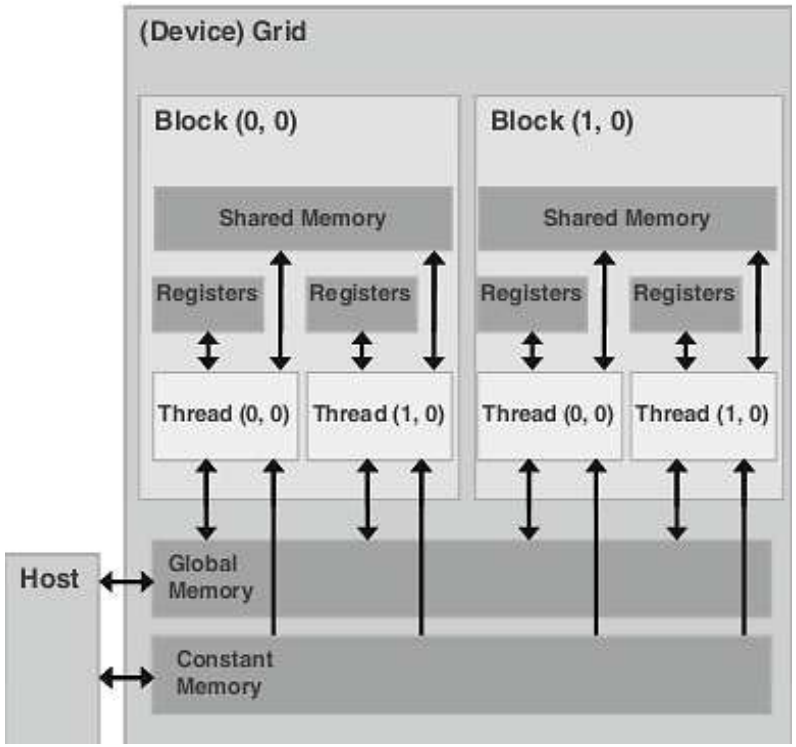
- Composta da pochi core è progettata per massimizzare l'efficienza di codice sequenziale;
- Memoria cache di grandi dimensioni utile a ridurre i tempi di latenza per l'accesso ai dati o l'esecuzione di istruzioni complesse;
- Logica di controllo sofisticata, per la gestione avanzata del flusso delle istruzioni (pipelining e multi-threading ed esecuzione out-of-order).



Many-core GPU

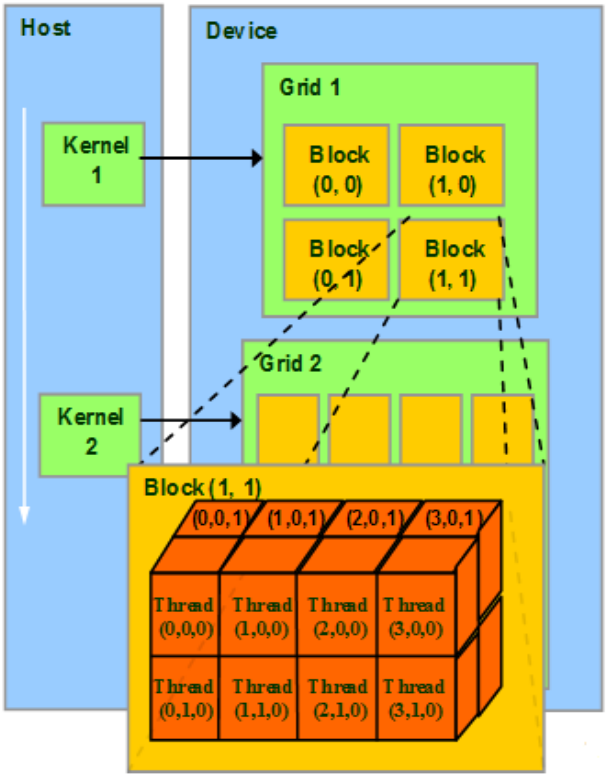
- Composta da molti core (anche centinaia) è progettata per l'esecuzione di applicazioni parallele;
- Effettua contemporaneamente numerose operazioni semplici;
- Strutture di memoria con tempi di accesso spesso trascurabili;
- Logica di controllo più semplice (esecuzione sempre in-order);

CUDA – Modello architetturale



- Una GPU CUDA è organizzata in array di Streaming Multiprocessor (SM);
- Ogni SM è formato da Stream Processor (SP) che condividono logica di controllo e cache delle istruzioni;
- Unità di memoria:
  - *Registri*: read/write dal proprio thread;
  - *Memoria locale*: read/write dal proprio thread;
  - *Shared Memory*: read/write dai thread di uno stesso blocco;
  - *Constant Memory* (cached): read only dai thread di una stessa grid;
  - *Global Memory*: può essere letta e scritta da tutti i thread nello stesso grid;
- Global e Constant Memory sono memorie persistenti tra differenti esecuzioni di kernel della stessa applicazione;

CUDA - Modello di Esecuzione



Esempio di una struttura a 5 dimensioni: *griglia* 2D (2x2) e *blocchi* 3D di *thread (4x2x2)*;

- Un'applicazione CUDA è composta da parti seriali, eseguite dall'***host***, e da parti parallele (i ***kernel)***, eseguite dal ***device***;
- Un *kernel* è definito come una ***grid***, decomposta a sua volta in *blocchi* assegnati, sequenzialmente, ai vari ***SM***;
- Ogni *blocco* è formato da ***thread***, l'unità computazionale fondamentale, ed eseguono tutti la stessa istruzione (Single-Program Multiple-Data);
- Ciascun *thread* può appartenere ad un solo *blocco,* ed è identificato da un indice univoco per tutta la durata del *kernel*;
- I *kernel* sono eseguiti sequenzialmente tra loro mentre *block* e *thread* sono eseguiti in parallelo;

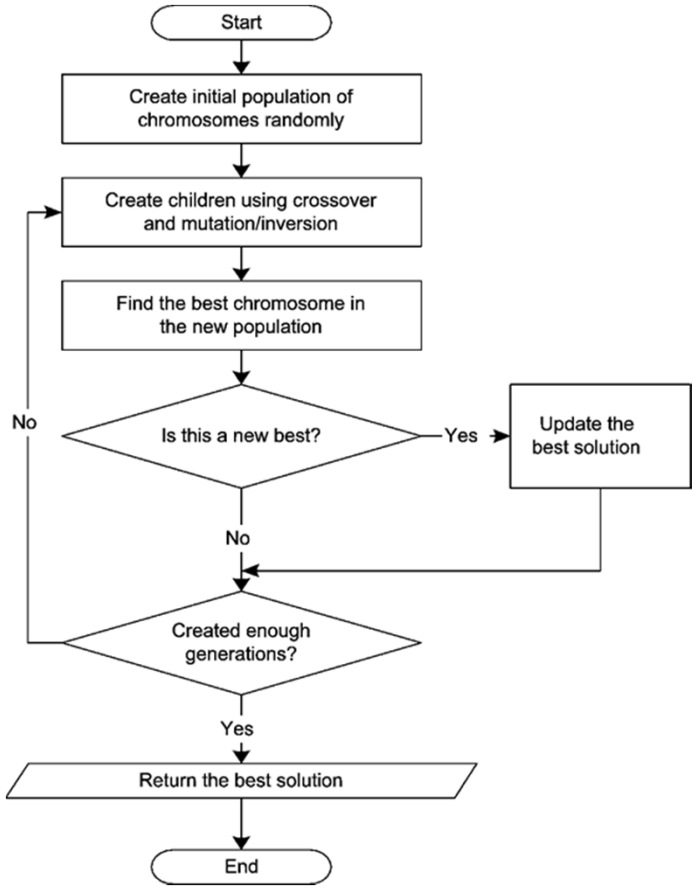
L'Algoritmo Genetico

- **I GA sono algoritmi di ricerca basati sui meccanismi di selezione naturale introdotti da Charles Darwin;**
- **Simulano l'evoluzione di una popolazione di individui, favorendo sopravvivenza e riproduzione per meglio adattarsi alle proprietà di un determinato ambiente;**
- **GAME è concepito per risolvere problemi di ottimizzazione per classificazione e regressione;**
- **La loro struttura funzionale si presta naturalmente ad essere implementata su architetture parallele.**

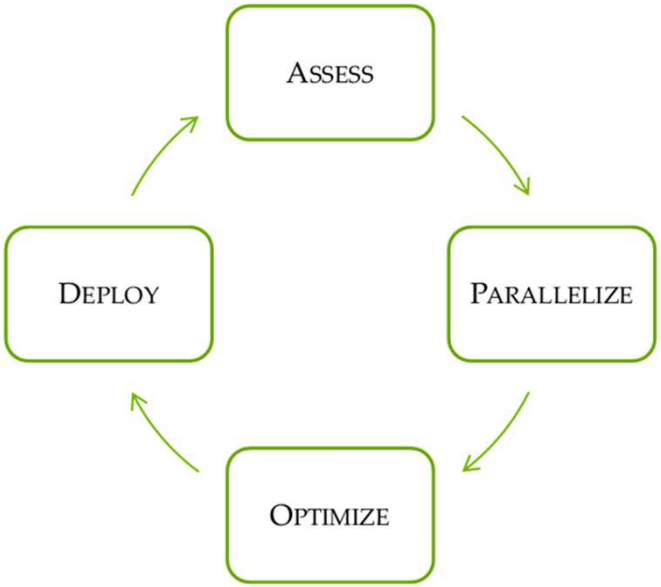
Corrispondenza tra modello biologico e matematico

Biologia	Matematica
Individuo	Vettore **x**
Adattamento all'ambiente	Funzione di fitness **f(x)**
Competizione	Operatori di Selection
Riproduzione	Crossover e Mutation
Individuo migliore	Ottimo del problema

Flow Chart d'esecuzione del GA



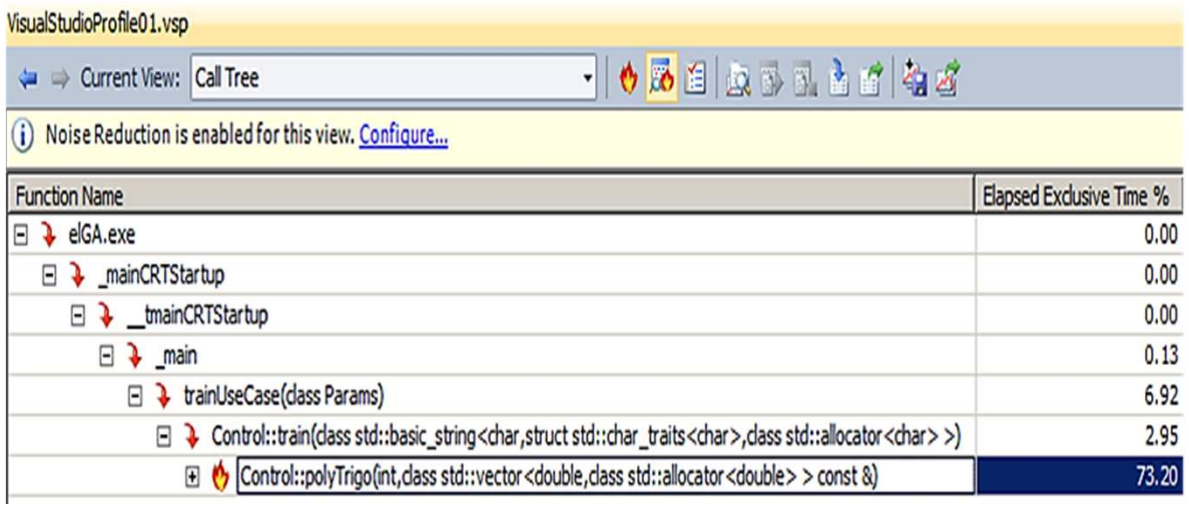
Modello di sviluppo - APOD



- **Processo di produzione ciclico a 4 passi:**
  - **Assess**
  - **Parallelize**
  - **Optimize**
  - **Deploy**
- **Mira a identificare le porzioni di codice che maggiormente possono beneficiare dall'esecuzione su GPU;**
- **Identificata la porzione di codice da ottimizzare, una prima versione viene sviluppata, testata e rilasciata rapidamente;**
  - **ogni ciclo successivo ricomincia identificando ulteriori opportunità di ottimizzazione.**

Assess

- Valutiamo il codice multi-core per identificare le parti di codice da parallelizzare;
  - Per fare ciò creiamo un profilo dell'applicazione utilizzando gli strumenti di profiling;
- Si redige un elenco di candidati per la parallelizzazione ordinata per tempo di esecuzione;
  - Verificheremo il comportamento dell'applicazione per il caso d'uso di TRAIN (addestramento del GA), che è quello maggiormente time consuming;
- Solo il «miglior» candidato viene parallelizzato;
  - Nel nostro caso la funzione `polyTrigo()`



Per generare il profilo dell'applicazione ed identificare gli **hotspot**, abbiamo utilizzato Visual Profiler, tool fornito con Visual Studio 2010.

Parallelize

- In questa fase bisogna esplicitare il parallelismo della funzione candidata, `polyTrigo();`
- Un modulo esterno conterrà il codice parallelo, utilizzabile dall'host tramite chiamate a funzioni esterne;
  - Il compilatore CUDA necessita di file «.cu»
- Le funzioni sono scritte utilizzando Thrust, una libreria template C++ simile alla C++ STL;
  - Permette di manutenere più semplicemente codice seriale e parallelo;
  - Include una collezione di algoritmi come sort e reduce;
  - la sua interfaccia astrae i parametri di lancio dei kernel.

C++ code

```
for (int i = 0; i < lenght; i++){
   Z[i] = X[i] + Y[i]
}
```

Thrust code

```
thrust::transform(X.begin(),X.end(),Y.begin(),Z.begin(),
     thrust::plus<float>());
```

C++ code

```
for (int i = 0; i < num_features; i++) {
   for (int j = 1; j <= poly_degree; j++) {
      ret +=  v[j] * cos(j * input[i]) +
         v[j + poly_degree] * sin(j * input[i]);
   }
}
```

Thrust code

```
struct sinFunctor {
 __host__ __device__
 double operator()(tuple <double, double> t) {
  return sin(get < 0 > (t) * get < 1 > (t));
 }
};
…
thrust::transform
 (thrust::make_zip_iterator
  (thrust::make_tuple(j.begin(), input.begin())),
 thrust::make_zip_iterator
  (thrust::make_tuple(j.end(), input.end()),
 vSin.begin(),
 sinFunctor());

double sinComp= reduce(vSin.begin(), vSin.end());
```

Optimize

- E' un processo iterativo:
  - valutare le possibilità di ottimizzazione;
  - applicare e verificare l'ottimizzazione;
  - verificare l'aumento di velocità raggiunta;
  - Reiterare il processo;

Tecniche di ottimizzazione ad alto livello utilizzate per ottenere incrementi delle prestazioni in Thrust sono:

- **Fusion**: fondere due funzioni successive in una sola operazione di g (f(x)) e dimezzando il numero di transazioni di memoria;
  - L'esecuzione fusa verrà eseguita circa due volte più velocemente;
- **Structure of Arrays** (**SoA**): è un modo alternativo di memorizzare le strutture dati rispetto al classico Array of Structures (AoS). Migliora l'efficienza degli accessi in memoria;
- **Implicit Sequences**: intervalli i cui valori sono definiti a livello di programmazione e non memorizzati da nessuna parte in memoria.

- L'ottimizzazione delle prestazioni si basa su:
  - Massimizzare l'esecuzione parallela;
  - Ottimizzare l'utilizzo della memoria per ottenere il massimo bandwidth;
  - Ottimizzare l'uso di istruzioni per raggiungere il massimo throughput delle istruzioni.

```
for (int i = 0; i < N; i++)
   Y[i] = F(X[i]); (y=f(x))
```

+

```
for (int i = 0; i < N; i++)
   sum += Y[i];   (z=g(y))
```

=

```
for (int i = 0; i < N; i++)
   sum += F(X[i]); z=g(f(x)))
```

AoS: `device_vector<float3> s;`

SoA: `device_vector<float> x,y,z;`

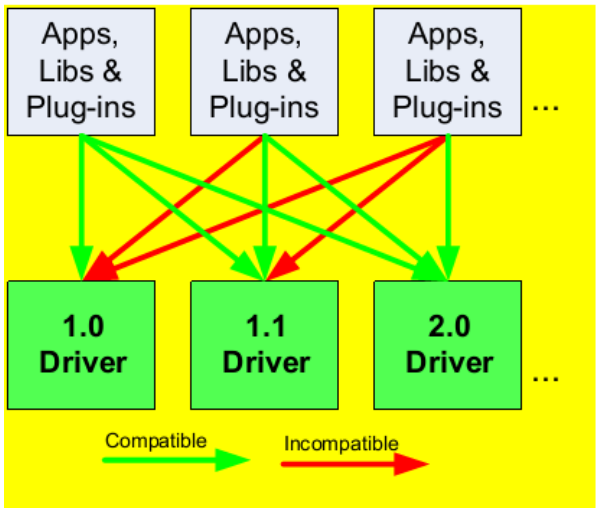
```
device_vector<int> indices(vec.size());
sequence(indices.begin(), indices.end());
```

```
counting_iterator<int> begin(0);
counting_iterator<int> end(vec.size());
```

Deploy

- Dopo il primo ciclo di accelerazione, l'implementazione corrente (parzialmente parallelizzata) viene distribuita, questo ci permette di:
  - beneficiare subito dei miglioramenti ottenuti;
  - avere un controllo più semplice tra le versioni;
  - Il processo di parallelizzazione è evolutivo anziché rivoluzionario;
- Confrontiamo il codice prodotto (GPU) con la versione legacy (CPU)
  - L'esecuzione della versione seriale non è ottimizzata per il calcolo su tutti core della CPU;
  - I benchmark per la versione seriale sono eseguiti su una CPU Intel Core i7 2630QM quad-core;
  - I benchmark per la versione GPU sono stati eseguiti su una GPU NVIDIA Tesla C1060 a 240 core (compute capability 1.3);
- Il codice parallelo è stato compilato utilizzando CUDA toolkit ver. 4.1 (compute compatibility 1.3);

- La *Compute Capability* descrive le funzionalità hardware ed il set di istruzioni;
- Le versioni delle API CUDA sono backward ma non forward compatibili;
- Dalla compute capability 1.3 sono disponibili le operazioni aritmetiche in doppia precisione;



Testing

- Per misurare le performance abbiamo eseguito su entrambe le applicazioni uno stesso esperimento di classificazione;

  La classificazione è una procedura in cui vengono inseriti i singoli elementi in gruppi, sulla base delle informazioni quantitative su una o più caratteristiche inerenti agli elementi (features), nonchè sulla base di un training set di elementi precedentemente etichettati;

- Funzione di output (polyTrigo):

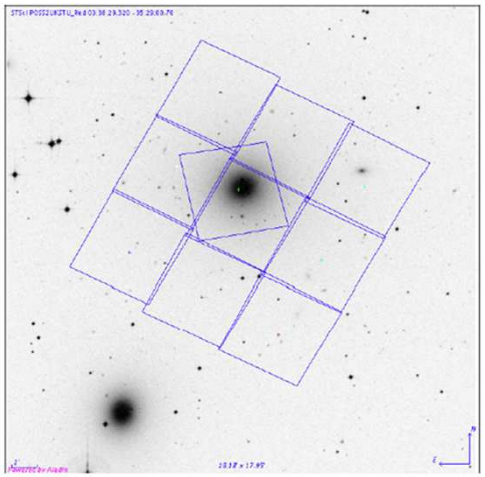
$$Out(pat_k) \cong a_0 + \sum_{i=1}^{m}\sum_{j=1}^{d} a_j \cos(j\, f_i) + \sum_{i=1}^{m}\sum_{j=1}^{d} b_j \sin(j\, f_i)$$

  - m = numero di features e d = grado del polinomio

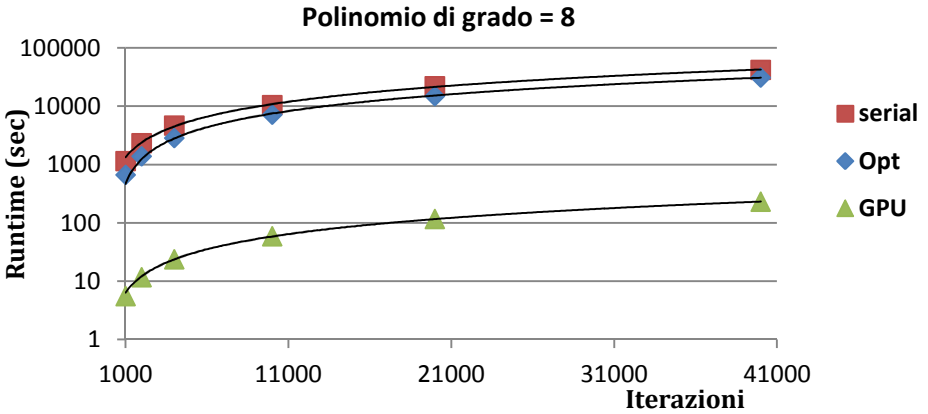
- Funzione di fitness**:**

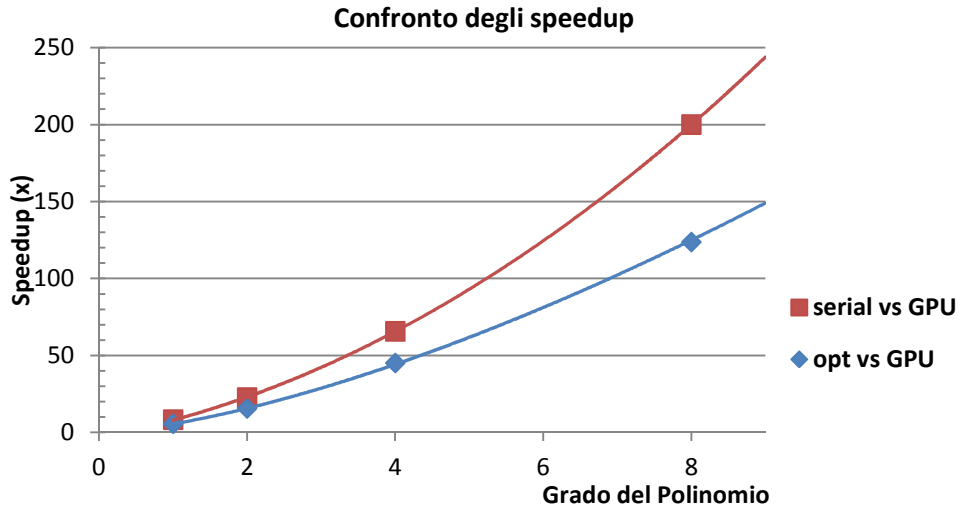
$$f(x) = 1 - E_k \quad \text{con} \quad E_k = (t_k - Out(pat_k))^2$$

- Il dataset di input scelto per la classificazione si chiama GCSearch
  - consiste di 2100 righe (input patterns) e 11 colonne (features);
  - dati reali che si riferiscono ad ampie osservazioni HST sul campo della gigante ellittica NGC1399 nel cluster Fornax (Brescia et al. 2011b);
  - Lo scopo ultimo è classificare oggetti nel campo osservato, tra GC (Globular Clusters) e oggetti di tipo diverso (classificazione crispy).
  - La versione non GPU ha fornito una performance di classificazione di circa l'86%. La scelta del caso d'uso è stata anche dettata dalle dimensioni contenute dei dati.



Andamento dell'algoritmo





- All'aumentare delle iterazioni non c'è deterioramento delle prestazioni;
- L'aumento del grado del polinomio porta ad un aumento delle prestazioni, l'accelerazione è tanto maggiore quanto più dati si devono elaborare;

Speedup				
degree	vs. Serial	step	vs. Opt	step
1	8x		6x	
2	23x	2.9	16x	2.7
4	66x	2.9	45x	2.8
8	200x	3.0	125x	2.8

La «qualità» dell' output della applicazione parallela rispetto a quella seriale, non subisce variazioni dato che non è stato modificato la sequenza delle operazioni dell'algoritmo.

Conclusioni

- **L'utilizzo di tecnologie GPGPU consente un elevato guadagno prestazionale a costi estremamente inferiori rispetto alle classiche infrastrutture di calcolo ad alte prestazioni;**
- **L'aumento delle prestazioni è legato al tipo di problema;**
  - **Bisogna prestare particolare attenzione alle fasi iniziali di analisi e design;**
  - **Problemi non SIMD non otterranno benefici.**

Sviluppi Futuri

- **Studio di possibili ottimizzazioni dell'algoritmo sfruttando codice misto Thrust/CUDA C;**
- **Implementazione di altri algoritmi di machine learning sviluppati e testati dal gruppo DAME, come SVM o PPS;**