Implementazione e studio di un algoritmo genetico 0-1 Knapsack su GPGPU

Attività progettuale per il corso di:
FONDAMENTI DI INTELLIGENZA ARTIFICIALE M
a cura di: Enrico Zamagni

Anno accademico: 2011/12

Prof.
Andrea Roli

Prof.ssa
Paola Mello

Indice

1. Introduzione

I principi ispiratori dell'*evolutionary computing*, oltre a costituire un ambito di ricerca particolarmente originale e stimolante, possono trovare un utile impiego in numerosi settori dell'informatica, dell'elettronica e della matematica. Tra queste varie applicazioni, gli algoritmi genetici (GA) offrono un approccio risolutivo euristico particolarmente semplice e innovativo a un'ampia classe di problemi che sono oggetto di studio della ricerca operativa e che prevedono di trovare la migliore soluzione possibile (o una sua valida approssimazione) per un'istanza di problema che ammette un campo estremamente vasto di soluzioni. Questa ricerca viene tradizionalmente affrontata mediante elaborate e costose procedure iterative studiate in maniera specifica per il particolare problema preso in esame. I GA promuovono invece un metodo di ricerca relativamente semplice da implementare e in grado di adattarsi con poco sforzo a diversi scenari di utilizzo.

L'idea chiave alla base degli algoritmi genetici è quella di considerare una possibile soluzione di un qualsiasi problema come un individuo le cui caratteristiche fisionomiche vengono descritte da una combinazione di più *geni* ognuno dei quali può assumere uno tra i valori del dominio (alleli). Se si considera come *popolazione* un insieme di più individui rappresentanti soluzioni per una stessa istanza di problema è possibile guidare un'*evoluzione* dell'intero patrimonio genetico attraverso procedure di selezione, ricombinazione e mutazione le quali costituiscono i principali operatori genetici. La *ricombinazione* produce un individuo figlio la cui configurazione genetica è ottenuta attraverso una combinazione dei due genotipi dei genitori che lo hanno generato; durante questo processo può intervenire con una certa probabilità l'operatore di *mutazione* che modifica in maniera imprevedibile il valore di un gene scelto casualmente. La mutazione, se usata nella giusta misura, aiuta la ricerca a convergere verso una soluzione migliore: è infatti estremamente probabile che la sola ricombinazione tenda a fermarsi in un punto di massimo (o minimo) locale, senza riuscire quindi a considerare uno spazio di soluzioni più ampio che potrebbe potenzialmente contenere individui con qualità migliori. L'operatore di *selezione* stabilisce ad ogni generazione quali individui dovranno prendere parte al processo di ricombinazione per generare nuovi individui; tale selezione viene effettuata principalmente in base al valore di *fitness* di ogni soggetto della popolazione, ovvero quel valore che quantifica la qualità di una soluzione dato il suo corredo genetico. Sfruttando questi e altri operatori si cerca di fare convergere una popolazione inizialmente ottenuta in maniera del tutto casuale verso un valore che approssimi adeguatamente la migliore soluzione ottenibile. È comunque facile intuire che quello appena esposto non è un metodo di ricerca ottimo: l'intero processo è regolato da un'abbondante componente di casualità e non esiste alcuna garanzia che si riesca a giungere alla soluzione ottima di un'istanza di problema, indipendentemente da quanti individui compongano la popolazione e per quante generazioni si faccia girare l'algoritmo. Numerosi studi e applicazioni pratiche hanno comunque dimostrato che i GA permettono – se accuratamente impostati – di raggiungere soluzioni molto buone in un tempo decisamente inferiore rispetto a quello necessario per una ricerca esaustiva e in molti casi anche con un'occupazione di memoria relativamente contenuta.

Una caratteristica particolarmente interessante degli algoritmi genetici riguarda il fatto che alcuni operatori tra cui – ad esempio quello di mutazione o di fitness – si applicano a ciascun soggetto in maniera completamente indipendente dagli altri individui della popolazione. Sfruttando adeguatamente questo parallelismo intrinseco dei GA, è possibile velocizzare considerevolmente l'esecuzione dell'algoritmo sovrapponendo nel tempo il calcolo di alcune funzioni ove possibile. Le architetture che si prestano particolarmente a questa ottimizzazione sono quelle di tipo SIMD (*Single Instruction Multiple Data*) le quali – come ricorda l'acronimo – sono in grado di svolgere una stessa sequenza di istruzioni (funzione) avvalendosi di più unità di calcolo, ognuna in grado di ricevere in ingresso un diverso set di parametri (dati). In alcune applicazioni di algoritmi genetici, il

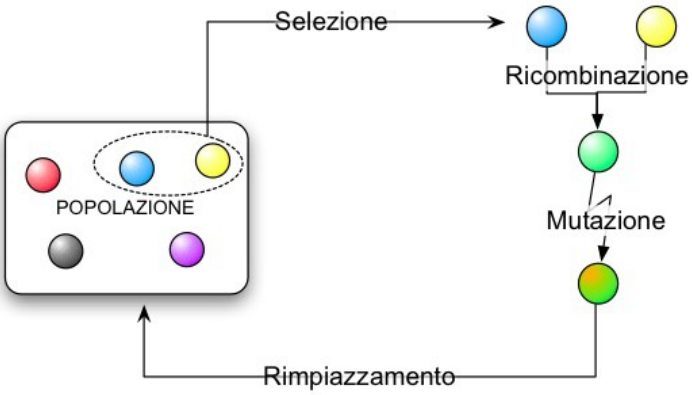

Figura 1.1: Una rappresentazione schematica dei principali operatori genetici in un ciclo generazionale.

calcolo della qualità di un individuo è svolto da funzioni di fitness che possono presentarsi anche molto complesse e la loro applicazione su ogni elemento della popolazione può risultare un compito particolarmente oneroso. In una architettura SIMD è possibile invece eseguire parallelamente queste funzione (ed eventualmente altre) su ogni individuo, con un conseguente aumento delle prestazioni.

Partendo da queste considerazioni, l'attività presentata si pone principalmente due obiettivi: si vuole innanzitutto realizzare un semplice algoritmo genetico e studiarne il comportamento in modo da verificare che esso costituisca un'euristica efficace a fronte di istanze di problemi particolarmente impegnative; in secondo luogo si cercherà di analizzare lo stesso algoritmo su di un'architettura SIMD in modo da poter stabilire se questa scelta porta a un sostanziale incremento della velocità di esecuzione.

Il primo passo affrontato è stato quello di scegliere quale architettura SIMD utilizzare per il progetto in esame. Escluse a priori soluzioni hardware ad-hoc e particolarmente costose, ci si è indirizzati verso il concetto di *GPGPU Computing* (General-Purpose computing on Graphics Processing Unit) che utilizza cioè l'elevatissimo parallelismo interno dei moderni device grafici per finalità di calcolo generiche. Soluzioni hardware di questo tipo offrono oggigiorno numerosissime unità di calcolo (*Stream Processors*) ad architettura unificata, grandi quantità di memoria dedicata e un elevato bandwidth di comunicazione con la CPU attraverso soluzioni di larghissima diffusione e prezzi estremamente accessibili. Per sfruttare questa tecnologia esistono due principali API molto diffuse: CUDA e OpenCL. Il primo è un framework proprietario di nVidia, mentre la seconda è una libreria aperta basata su C99 inizialmente proposta da Apple e poi ratificata da numerose aziende (tra cui nVidia, AMD e Intel). Entrambe offrono funzionalità del tutto simili e si distinguono per alcune importanti differenze: OpenCL consente l'esecuzione di codice su una molteplicità di piattaforme tra cui GPU, CPU e altri tipi di processori, mentre CUDA (per ora) offre supporto soltanto a device hardware nVidia. Tuttavia quest'ultimo framework ha riscosso negli ultimi anni un discreto successo ed è corredato di un'ampia documentazione, esempi e strumenti di sviluppo ed è estendibile ad altri linguaggi oltre a C (al momento C++, Python e Fortran). Per questi motivi si è quindi scelto di utilizzare CUDA per la realizzazione del progetto, all'inizio del prossimo capitolo verranno presentate le caratteristiche fondamentali di tale architettura.

Una seconda importante scelta riguarda la classe di problemi che si vuole risolvere. Come già accennato, la letteratura offre abbondanti esempi nei quali gli algoritmi genetici vengono applicati in numerosissimi campi. Si è deciso di prendere in esame il problema del Knapsack in quanto esso è ben noto dalla ricerca operativa, è di facile implementazione e si offre particolarmente bene agli scopi sopra descritti. In particolare, ci si concentra sulla sua formulazione più comune, ovvero il problema 0-1 Knapsack che permette una facile codifica delle soluzioni attraverso una stringa di bit. Adottare una tipologia di problema relativamente facile ne permette infatti una rapida realizzazione e consente di concentrarsi sullo studio delle sue performance e sulla sua parallelizzazione.

Nel prossimo capitolo verranno esposte nel dettaglio le problematiche affrontate nella realizzazione del progetto, soffermandosi in particolare sull'analisi svolta per poter adattare l'algoritmo scelto all'architettura CUDA e alle sue peculiarità. Seguirà un capitolo nel quale si esaminerà il comportamento del codice prodotto e si confronteranno i risultati attesi coi dati ottenuti.

2. Analisi e implementazione

Il primo importante problema che è stato necessario affrontare riguarda la possibilità di svolgere un'analisi del progetto che tenesse conto dei vincoli imposti dalla particolare piattaforma scelta: è infatti opportuno pensare di organizzare le strutture dati e l'impostazione logica dell'algoritmo in modo da poter trarre maggiori benefici dalla parallelizzazione degli individui sulle diverse unità computazionali della GPU. Viene pertanto esposta una breve presentazione delle principali caratteristiche architetturali di CUDA che è stato necessario approfondire prima di proseguire nella progettazione.

2.1 Il framework CUDA

Mediante l'utilizzo di particolari driver e apposite librerie, il framework CUDA permette di compilare alcune procedure scritte in uno dei diversi linguaggi di programmazione supportati (nel nostro caso si è fatto uso del C tradizionale) in un codice macchina direttamente eseguibile da una (o più) GPU CUDA-capable installata nel sistema. Il risultato della compilazione di un programma GPGPU consiste pertanto in un comune programma eseguito dalla CPU con un unico flusso di controllo (*host code*) che ha in ogni momento la possibilità di richiamare delle particolari funzioni (*kernels*) che vengono eseguite da un numero elevatissimo di unità di calcolo sul dispositivo grafico (*device code*). Mentre queste procedure vengono eseguite, la CPU è in grado di svolgere altri calcoli oppure può sospendersi in attesa dei risultati. Sfortunatamente il device code può supportare solo un ristretto sottoinsieme di funzionalità del C ed è obbligato a rispettare altri vincoli a causa della peculiare architettura sulla quale viene eseguito. Inoltre esso può utilizzare esclusivamente la memoria che risiede fisicamente sul chip grafico e tutte le comunicazioni tra GPU e CPU devono essere gestite attraverso apposite chiamate. Questa caratteristica rischia di portare verso un considerevole collo di bottiglia se le modalità e le tempistiche con le quali si effettuano i trasferimenti di memoria non vengono accuratamente organizzate visto che tali scambi devono avvenire necessariamente attraverso il bus PCIe.

La Figura 2.1 mostra in maniera schematica un confronto tra la struttura tradizionale di una CPU e una GPU. È immediato vedere come da una parte un comune processore moderno è dotato di poche ma sofisticate unità di calcolo orchestrate da una complessa rete di controllo e corredate da un'ampia cache multi-livello per i dati, mentre un device grafico dispone di numerosissime pipeline di elaborazione sprovviste però di grandi quantitativi di memoria cache, registri dedicati e logica di controllo. Il pattern di utilizzo più tipico di queste risorse prevede il trasferimento verso la GPU di grandi quantità di dati che vengono poi suddivisi e assegnati ai diversi stream processors per essere individualmente processati mediante funzioni e algoritmi appositi. Una volta terminata la loro

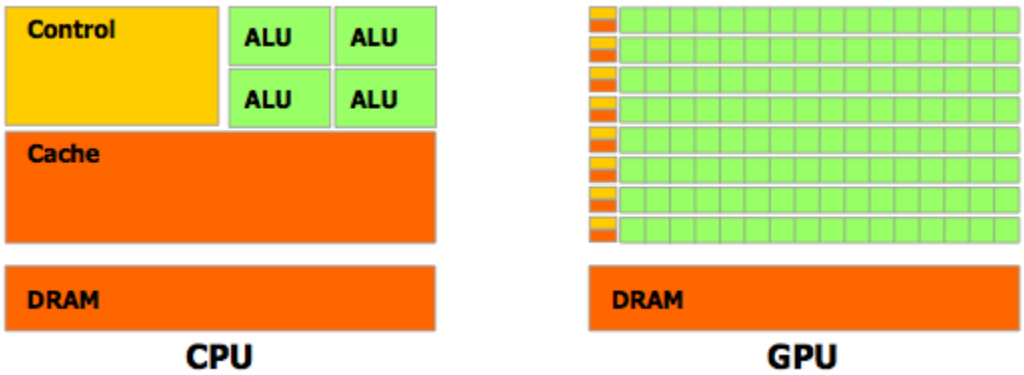

Figura 2.1: Principali differenze strutturali tra le due architetture in esame.

elaborazione – o comunque a intervalli prefissati – è necessario trasferire i risultati verso la CPU. Vengono anche previste delle semplici forme di collaborazione e/o sincronizzazione tra le varie unità di calcolo.

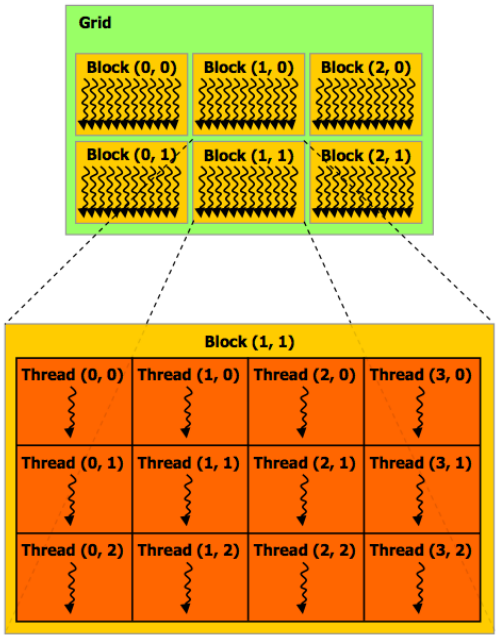

Figura 2.2: Organizzazione gerarchica delle unità di elaborazione in CUDA

Uno stesso kernel viene quindi eseguito da più thread contemporaneamente. Ognuno di questi viene indicizzato in un vettore o in una matrice a due o tre dimensioni che costituisce un'unica entità di calcolo detta *block*; tutti i thread all'interno di un blocco dispongono di una piccola memoria condivisa e possono pertanto scambiarsi dati e sincronizzarsi tra loro. Ogni blocco è a sua volta organizzato in un vettore o in una matrice bidimensionale che va a costituire la griglia (*grid*) di elaborazione per quel kernel. Grid diverse possono eventualmente eseguire kernel diversi contemporaneamente.

2.2 Analisi dell'algoritmo

È fondamentale tenere conto di queste peculiarità architetturali prima ancora di intraprendere la progettazione del codice al fine di sfruttare al meglio le potenzialità di calcolo offerte dal device grafico. Nel caso in esame, è necessario pensare a come organizzare le diverse strutture dati che rappresenteranno il patrimonio genetico della popolazione, a quali operatori implementare e come poterli parallelizzare il più efficacemente possibile e infine a come gestire la comunicazione tra i diversi soggetti. I numerosi esperimenti già svolti e documentati in letteratura offrono una ricca base di conoscenza dalla quale trarre ispirazione. L'attenzione principale è volta al fatto che si cerca di poter eseguire attraverso dei kernel CUDA la pressoché totalità dell'algoritmo: limitarsi ad eseguire su GPU delle semplici funzioni e mantenere il controllo principale sulla CPU vanificherebbe infatti ogni speedup, visto che il continuo scambio dei dati e il frequente passaggio da host code a device code introdurrebbe un overhead inaccettabile.

Si deve sopratutto cercare di sfruttare l'organizzazione dei thread imposta dal sistema in uso: ignorarla significherebbe impiegare solo una piccola porzione delle unità di calcolo messe a disposizione dalla GPU e lasciare inattive le altre. Il problema si traduce quindi nel pensare a un'organizzazione appropriata della popolazione di individui. L'approccio più tipico negli algoritmi genetici è quello di considerare un'unica popolazione panmittica nella quale ogni individuo può potenzialmente venire a contatto con un altro. Tale pattern è sicuramente semplice e di facile implementazione ma non è sfortunatamente utilizzabile nel caso in esame visto che una simile organizzazione prevederebbe di identificare un generico individuo con un thread e racchiudere tutti i thread in un unico blocco. In letteratura sono disponibili molti esempi di studio di popolazioni strutturate che superano questo semplice approccio; due in particolare sono particolarmente significative per questo contesto: le popolazioni distribuite (dGA) e cellulari (cGA). Le prime sono state proposte per sfruttare adeguatamente scenari di implementazione distribuiti, nei quali ogni host appartenente alla rete realizza una popolazione di individui locale che può periodicamente venire a contatto con soggetti adeguatamente selezionati provenienti da altri host collegati. Un simile modello è detto a *isole* e l'operatore che si occupa di selezionare e trasportare determinati individui da un'isola a un'altra secondo regole ben stabilite è detto operatore di *migrazione*. Questa tecnica presenta anche il notevole pregio di poter eseguire l'algoritmo con parametri diversi sulle diverse isole. I GA infatti sono notevolmente suscettibili a variazioni di quei valori che stabiliscono la probabilità o la frequenza con cui vengono applicati certi operatori (probabilità di mutazione, di crossover, etc..) e spesso è difficile individuare la configurazione ottimale per un determinato

problema o – addirittura – per una precisa istanza. Creando più popolazioni è pertanto possibile guidare la ricerca con impostazioni diverse dello stesso algoritmo su ciascuna di esse e cercare contemporaneamente di individuarne i parametri ottimali.

Le reti cellulari invece mantengono intatta l'idea di popolazione unica ma eliminano la proprietà di panmissia disponendo i vari soggetti in una griglia bidimensionale (eventualmente anche tridimensionale) e limitando le capacità di riproduzione di ogni individuo ai soli partner coi quali viene localmente a contatto, come visualizzato schematicamente nella Figura 2.3. Questo tipo di approccio è stato specificatamente pensato per trarre vantaggio da architetture di tipo SIMD in quanto cerca di semplificare e distribuire sui vari individui la logica di controllo degli operatori di selezione. Normalmente infatti, tali funzioni richiederebbero di operare sull'intero set di soluzioni (ad esempio per individuare i soggetti candidati per il crossover in base ad un ordinamento della loro qualità) e comprometterebbero irrimediabilmente il parallelismo dell'algoritmo. Con questo sistema invece, ogni entità di calcolo rappresenta un singolo individuo ed è in grado di prendere decisioni in maniera autonoma consultando soltanto un ristretto sottoinsieme della popolazione.

Entrambi i metodi offrono proprietà interessanti e utili: innanzitutto una popolazione cellulare si presta molto bene ad essere realizzata per l'architettura scelta, visto che il framework CUDA permette facilmente l'organizzazione dei vari thread in una griglia bidimensionale. Si decide quindi di adottare il modello cGA, consentendo ad ogni individuo di selezionare il proprio partner per il crossover scegliendolo tra i quattro individui che riesce a “toccare” orizzontalmente e verticalmente considerando la propria posizione locale. Questo accorgimento non è tuttavia sufficiente: una popolazione così strutturata verrebbe infatti mappata in un unico blocco CUDA e utilizzerebbe solo una piccola percentuale delle capacità di calcolo dell'elaboratore grafico. Per superare questo problema si potrebbe considerare un modello ibrido tra quelli precedentemente considerati: sono presenti in letteratura infatti numerose variazioni che cercano di unire i maggiori vantaggi delle più comuni tipologie di popolazione. Di particolare interesse per il nostro contesto è il modello cellulare a isole (vedi Figura 2.3) che mette in comunicazione più popolazioni cGA indipendenti utilizzando il già discusso operatore di migrazione. In questo modo si potrebbe allocare un'isola di individui disposti a griglia su diversi blocchi CUDA e prevedere che essi si scambino periodicamente dei soggetti selezionati per la migrazione, sfruttando così appieno l'architettura del framework. Chiaramente, gli scambi tra le diverse isole andrebbero effettuati ad intervalli non troppo ravvicinati per evitare di rallentare troppo l'esecuzione di un algoritmo parallelo con l'introduzione di logiche di

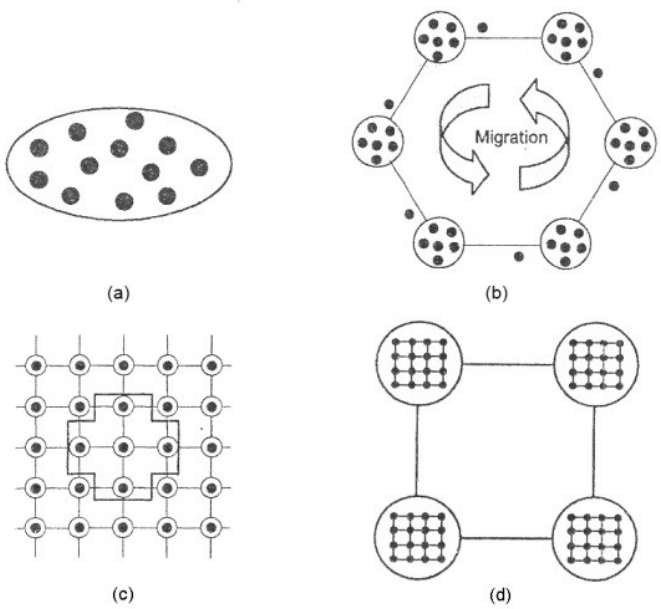

Figura 2.3: Diversi tipi di popolazione: a) popolazione panmittica; b) popolazione a isole; c) popolazione cellulare; d) soluzione ibrida con popolazioni cellulari organizzate a isole.

controllo che per loro natura risultano intrinsecamente sequenziali e quindi operabili esclusivamente dalla CPU. La progettazione dell'operatore di migrazione è stata però giudicata troppo onerosa per la portata del presente progetto e si è preferito concentrarsi su aspetti più rilevanti dal punto di vista didattico piuttosto che sul semplice ottenimento di maggiori performance. L'analisi tiene comunque conto delle scelte fatte e l'implementazione è stata realizzata in modo che tale caratteristica possa essere aggiunta in un secondo tempo. L'algoritmo supporta pertanto l'esecuzione su più isole (quindi su più blocks) ma allo stato attuale queste non hanno alcun modo di comunicare tra loro.

Una volta affrontato il problema di come parallelizzare l'algoritmo l'analisi non presenta notevoli difficoltà: la codifica delle soluzioni – ovvero la struttura dati per rappresentarla – è realizzabile mediante una semplice stringa di tanti bit quanti sono i possibili oggetti previsti dall'istanza: se quindi una soluzione presenta l'i-esimo bit con valore uno il corrispettivo oggetto 'i' risulterà presente nel knapsack e contribuirà al calcolo del peso e del fitness per quella soluzione.

2.3 Implementazione

Nel seguito verrà brevemente illustrata la realizzazione concreta dell'algoritmo, con particolare attenzione a come i diversi operatori genetici sono stati implementati. Una descrizione approfondita del sorgente esula dagli scopi di questo documento e verrà quindi esaminata la sola struttura di funzionamento dell'algoritmo genetico, senza dare troppo peso ai dettagli implementativi propri dell'architettura CUDA.

Come detto, l'intera popolazione viene organizzata in una griglia bidimensionale e ogni individuo è identificato univocamente dalle proprie coordinate di riga e colonna e rappresenta una soluzione del problema mediante una sequenza di bit. All'inizio dell'algoritmo l'intera popolazione viene determinata da una produzione causale e distribuita di bit, in questo modo la ricerca inizia con un patrimonio genetico generato casualmente, come previsto dalla pressoché totalità dei GA.

L'operatore di selezione implementato – come già discusso – è estremamente semplice: volendo sacrificare la complessità di una ricerca tra i migliori individui della popolazione a favore di un maggiore parallelismo, esso si limita a considerare per ogni soggetto i quattro elementi (*neighbours*) che esso è in grado di "toccare" partendo dalla propria posizione; in altre parole si selezionano i quattro individui collocati a una distanza Manhattan pari a uno a partire dalla propria cella della matrice. Con ciascuno dei quattro soggetti trovati si applica l'operazione di crossover e si verifica se l'individuo (*child*) generato è migliore del genitore e – in caso affermativo – esso andrà a sostituire il proprio progenitore nella matrice. Per non introdurre delle incongruenze nella popolazione, i figli vengono salvati in un buffer temporaneo e andranno eventualmente a sostituire i genitori solo quando l'iterazione corrente sarà terminata. In questo modo ogni individuo può prelevare i cromosomi corretti dal proprio vicinato e non delle stringhe già aggiornate o inconsistenti.

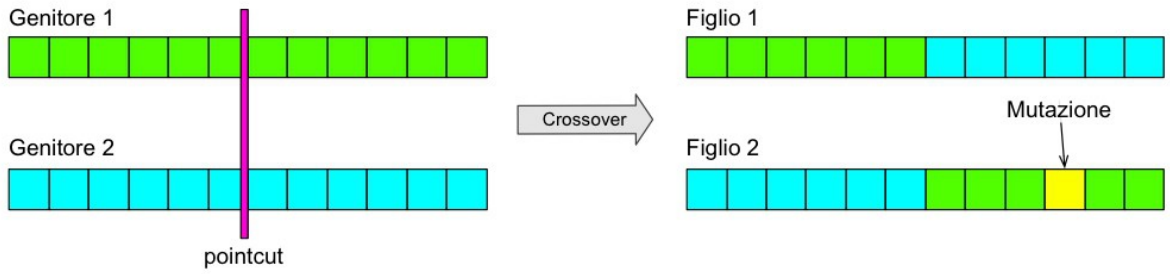

Figura 2.4: Illustrazione schematica degli operatori di crossover e mutazione impiegati.

L'operatore di crossover utilizzato è detto *single point*: esso consiste nell'individuare un singolo punto di taglio nel cromosoma di un individuo e copiare da quel punto in poi i geni di un secondo soggetto. In realtà con questo metodo è possibile produrre due soluzioni figlie da una coppia di genitori nelle quali si alterna l'ordine col quale vengono copiati i geni prima e dopo al punto di

taglio, come mostrato in Figura 2.4. Esistono in realtà procedure anche più sofisticate per il crossover ma con dei semplici esperimenti si dimostra che tale complessità non è spesso ripagata da un comportamento migliore dell'algoritmo, specialmente per problemi relativamente semplici quali il knapsack. Riassumendo quindi: ad ogni iterazione vengono individuati quattro neighbours per ogni singolo soggetto e viene applicato per ciascuno di essi l'operatore di crossover appena descritto per un totale complessivo di otto soluzioni figlie la migliore delle quali andrà a rimpiazzare il padre se presenta un fitness migliore del suo, oppure se presenta lo stesso fitness ma con un peso minore.

L'operatore di mutazione è stato implementato anch'esso in maniera molto semplice: viene eventualmente applicato immediatamente dopo il crossover tra due individui, sceglie un bit a caso nella stringa del soggetto figlio risultante e ne inverte il valore. Il suo funzionamento intuitivo è illustrato nella Figura 2.4.

Per valutare la qualità di una soluzione si utilizza una semplice funzione che produce come output due valori interi positivi: il *fitness* e il *weight* dell'individuo ovvero – rispettivamente – il valore complessivo degli oggetti selezionati e il loro peso cumulativo. È opportuno notare come durante l'esecuzione dell'algoritmo possa capitare frequentemente di avere a che fare con soluzioni non ammissibili, cioè che presentano un peso superiore alla capacità del knapsack. In questi casi la letteratura suggerisce due diverse possibilità: la prima prevede di scartare degli oggetti casualmente scelti fino a rendere la soluzione ammissibile, la seconda consiste nel penalizzare il suo valore di fitness per renderla meno appetibile rispetto a soluzioni valide. Si è deciso di implementare la seconda opzione in quanto si preferisce lasciare l'algoritmo libero di esplorare soluzioni non ammissibili visto che queste possono comunque dare luogo a individui migliori. Per ogni soluzione inammissibile quindi, la funzione di valutazione ne determina l'*overweight* rispetto alla capacità massima consentita e lo sottrae al valore per quell'individuo, moltiplicandolo prima per un fattore selezionabile dall'utente (vedi dopo). Sono stati fatti diversi esperimenti che prevedevano una svalutazione basata su funzioni non lineari calibrate in base allo specifico range dei valori dell'istanza, alla capacità disponibile e allo stesso valore di overweight, tuttavia tali implementazioni introducevano ulteriore complessità senza migliorare sensibilmente il comportamento dell'algoritmo.

```
N
  1 profit1 weight1
  2 profit2 weight2
  … …       …
  N profitN weightN
C
```

Figura 2.5: Formato utilizzato per la codifica dei file di istanza di problema.

Per codificare un'istanza di problema attraverso un file di testo è stato realizzato un semplice parser che interpreta il formato descritto in Figura 2.5: viene letto un intero N rappresentante il numero di oggetti selezionabili, vengono lette N righe contenenti un indice progressivo e due valori (interi) riguardanti rispettivamente il profitto e il peso per l'i-esimo oggetto e infine viene letta la capacità disponibile per il knapsack (C). Per generare le istanze utilizzate per le attività di testing e di raccolta dei risultati è stato utilizzato un generatore già realizzato da D. Pisinger, S. Martello e P. Toth e disponibile all'indirizzo http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html e compatibile con il formato sopra descritto.

Infine, per poter controllare i vari parametri utilizzati durante l'esecuzione sono stati previsti i seguenti switch che è possibile immettere da riga di comando, immediatamente dopo aver specificato il file di istanza da caricare (tutte le opzioni seguenti sono opzionali):

- **`-p`**
  Specifica il numero n di individui da includere nella popolazione, di default questo valore è assunto pari a 100.
- **`-r`**
  Indica il numero massimo di iterazioni da effettuare prima che l'algoritmo termini, indipendentemente dalla qualità della soluzione trovata. Normalmente l'algoritmo svolge 500 iterazioni.

- **`-m`**
  Specifica tramite un valore a precisione singola compreso tra 0 e 1 la probabilità che si verifichi una mutazione ad ogni applicazione dell'operatore di crossover. Di default si ha una probabilità pari a 0.1.
- **`-c`**
  Specifica tramite un valore a precisione singola compreso tra 0 e 1 la probabilità che ogni individuo ha – per ogni iterazione – di ricorrere all'operatore di crossover. Di default si ha una probabilità pari a 0.8.
- **`-z`**
  Specifica il fattore di penalizzazione da applicare alle soluzioni non ammissibili. Questo valore (a precisione singola) viene moltiplicato all'overweight della soluzione. E' possibile immettere il parametro **`-zz`** per evitare che l'algoritmo consideri soluzioni inammissibili.
- **`-s`**
  Permette di introdurre un valore long integer senza segno da utilizzare come seed per la generazione dei numeri casuali. Se non viene fornito dall'utente il programma ne recupererà automaticamente uno basandosi sull'ora di sistema.
- **`-v -x`**
  Abilitano rispettivamente la modalità verbose e step-by-step, utili per controllare l'evolvere della popolazione durante le iterazioni intermedie dell'algoritmo.
- **-h**
  Abilitano la visualizzazione delle stringhe di soluzione in formato esadecimale, particolarmente utile per istanze con molti oggetti.

La versione dell'algoritmo scritta con estensioni CUDA offre i seguenti switch aggiuntivi:

- **-i**
  Funzionalità sperimentale che specifica il numero di isole da utilizzare. È già stato spiegato come le isole siano tra di loro indipendenti e come ciascuna di esse venga mappata su un block della GPU. Di default viene utilizzata una sola isola.
- **-d**
  Permette di specificare uno specifico device CUDA da utilizzare per l'elaborazione, in caso di configurazioni con più GPU. Se non specificato dall'utente, questo valore viene mantenuto a 0 (default device).

La Figura 3.1 offre un riassunto completo del flusso di controllo dell'algoritmo fin qui descritto.

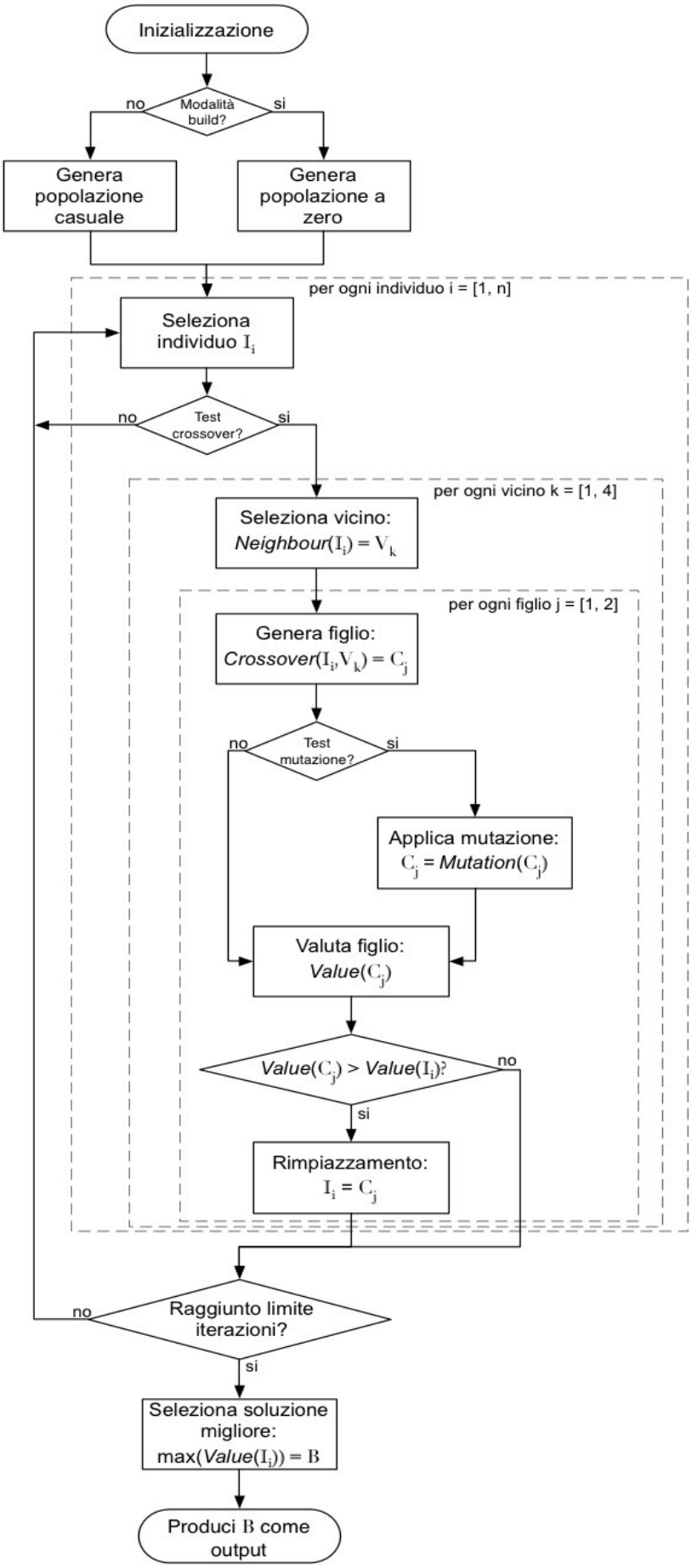

Figura 3.1: Flusso di controllo qualitativo dell'algoritmo implementato.

3. Verifica del comportamento

3.1 Confronto con euristica random search

Prima ancora di sperimentare come il software si comporta in presenza di diversi parametri o istanze è necessario verificare che quanto sviluppato sia effettivamente un algoritmo genetico funzionante ovvero in grado di guidare la ricerca verso soluzioni via via più apprezzabili; in altre parole è necessario dimostrarne l'utilità.

Per questo primo fondamentale test si è pensato di valutare il comportamento del GA realizzato confrontandolo con un algoritmo strutturato in maniera simile ma che procede la propria ricerca soltanto attraverso una produzione di soluzioni casuali. Si avrà quindi sempre una popolazione di individui disposti in una griglia ma non viene loro applicato nessuno degli operatori sopra illustrati: ad ogni iterazione ci si limita a generare una nuova soluzione casuale per ciascun soggetto, selezionare la migliore delle due e utilizzarla per l'iterazione successiva. Chiaramente – vista la sua semplicità – un simile test godrà di performance nettamente superiori rispetto al suo concorrente, tuttavia ci si aspetta che – in presenza di istanze sufficientemente complesse – il GA riesca a produrre soluzioni nettamente migliori in un minor numero di iterazioni (ciò non vuole dire però più velocemente rispetto all'altro). Si prevede inoltre che la ricerca random fatichi a trovare soluzioni ammissibili in istanze caratterizzate da un elevato numero di oggetti, mentre la ricerca genetica dovrebbe essere sempre in grado di trovare soluzioni quantomeno ammissibili.

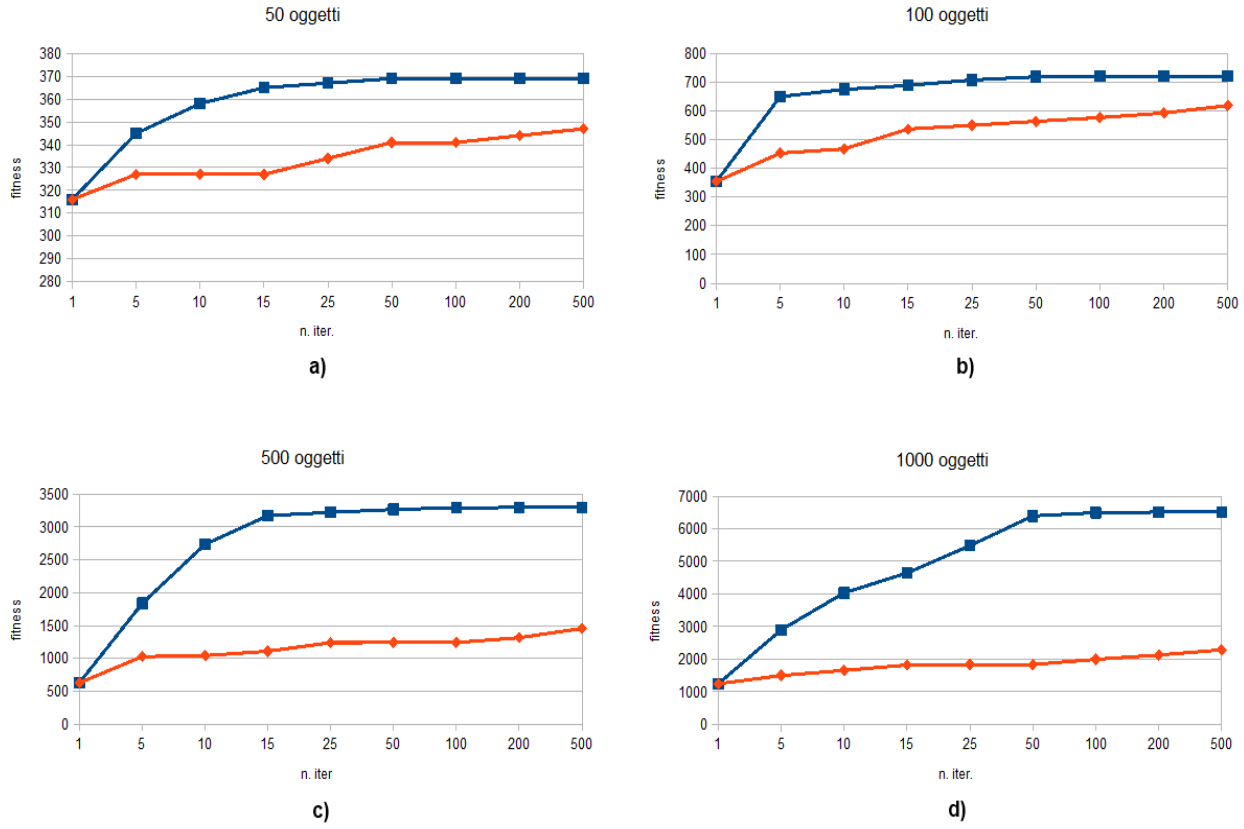

Figura 3.2: Algoritmo genetico (in blu) confrontato con un'euristica random search (in arancio) su quattro istanze di dimensioni crescenti. A parità di iterazioni il GA è sempre in grado di produrre soluzioni notevolmente migliori. I parametri utilizzati per l'esecuzione sono quelli standard indicati nel cap.2. Si noti che le soluzioni random prodotte in c) e d) sono sempre fortemente inammissibili.
N.B. L'ascissa non è da intendersi come asse temporale.

La Figura 3.2 mostra a colpo d'occhio come i risultati ottenuti corrispondano con quanto ci si aspettava: l'algoritmo genetico è sempre in grado di trovare soluzioni di qualità considerevolmente superiore rispetto al suo avversario. Più specificatamente, si vede come tale ricerca porti a un rapido innalzamento del fitness e quindi a una veloce saturazione, mentre la ricerca casuale procede irregolarmente con risultati molto inferiori e con una saturazione addirittura posteriore alle 500 iterazioni. Si deve prestare molta attenzione a non interpretare le ascisse dei grafici come indicatori temporali: una singola iterazione di GA è sensibilmente più lenta della corrispettiva random, a riprova del fatto che – per istanze semplici – è sempre preferibile utilizzare euristiche meno complesse e più veloci. L'algoritmo genetico realizzato mostra però la sua superiorità nelle istanze con un alto numero di oggetti e – sopratutto – in quelle istanze costruite in modo da risultare "difficili", ovvero che contengono molteplici oggetti da poter scegliere ma dispongono di capacità molto limitata. Le istanze utilizzate risultano di media complessità e presentano una correlazione debole dei valori di profitto e peso dei diversi oggetti rispetto alla qualità disponibile; nonostante ciò, se il numero di oggetti supera approssimativamente le 60-100 unità, la ricerca euristica utilizzata mostra una particolare inefficienza nel produrre soluzioni ammissibili. Si noti come nei grafici vengano considerate soluzioni già penalizzate ma non escluse in caso di inammissibilità. È pertanto importante evidenziare che nei grafici c) e d) le soluzioni ricavate dall'algoritmo random non sono soltanto di scarsa qualità ma sono anche tutte pesantemente inammissibili mentre il GA restituisce sempre soluzioni valide.

3.2 Prestazioni

Uno dei principali obiettivi del lavoro presentato riguarda il raggiungimento di prestazioni ottimali per mezzo di una parallelizzazione del codice e della sua esecuzione su GPU. Risulta quindi fondamentale verificare se gli sforzi spesi per l'implementazione mediante il framework CUDA si traducono effettivamente in un aumento sensibile delle performance. Per poter rispondere a questo quesito è stata realizzata una variante analoga dell'algoritmo che però esegue tutte le sue procedure esclusivamente su CPU. L'implementazione di questa seconda versione non introduce alcuna modifica alla struttura logica del programma e si differenzia sostanzialmente per una diversa fase di inizializzazione e per un differente pattern di accesso alla memoria.

Viene dapprima svolto un semplice test che prevede l'esecuzione dei due algoritmi con le stesse istanze e parametri e la misurazione del tempo che impiegano per completare un fissato numero di iterazioni; ci si aspetta che l'esecuzione su GPU possa completare molto prima della sua controparte. Il test viene eseguito su una macchina dotata di processore Intel i7 920 a 2.67 GHz e di scheda grafica GeForce GTX 560 Ti. Si noti che in questo primo esperimento, la versione del software per GPU esegue una popolazione composta da una sola isola e andrà quindi a occupare un solo blocco della GPU.

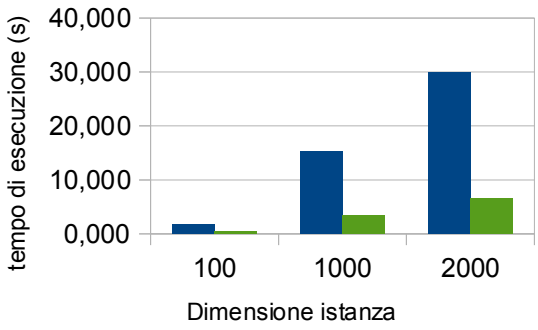

Figura 3.3: Confronto delle due versioni del GA: per CPU (in blu) e per GPU (in verde). Sono state svolte 1500 iterazioni con una popolazione di 200 individui.

Osservando i risultati del test esposti in Figura 3.3 si evince innanzitutto una buona superiorità della variante CUDA in termini di prestazioni. In entrambi i casi il tempo di esecuzione cresce linearmente con l'aumentare della dimensione del problema, tuttavia eseguire l'algoritmo sulla GPU permette di godere di uno speedup medio pari a 4.6. In realtà un simile guadagno non giustifica appieno gli sforzi che è necessario intraprendere per parallelizzarlo e svilupparlo in CUDA. Inoltre, lo stesso test eseguito su una macchina meno potente ha riportato risultati del tutto opposti, con un runtime CPU che risulta addirittura inferiore a quello della GPU.

Per verificare quanto lo speedup ottenibile risenta notevolmente della configurazione hardware in uso viene svolto un secondo test nel quale si mettono a confronto i tempi di esecuzione su diversi device: oltre ai componenti già enunciati vengono usati un processore Core 2 Duo a 2.4 GHz, una scheda grafica GeForce 8600M GT e una GeForce 8400GS.

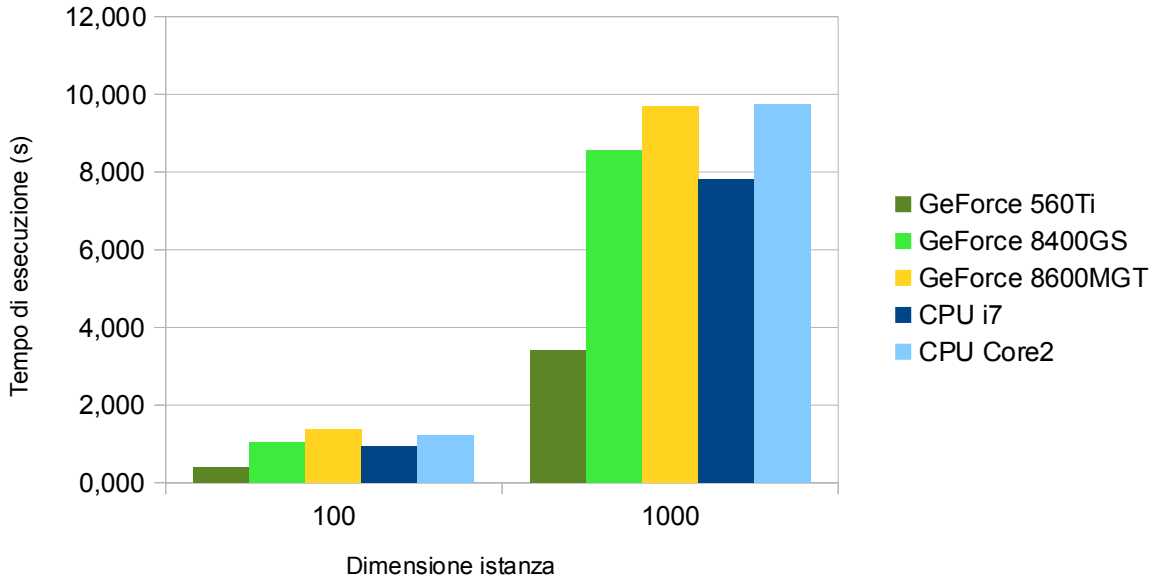

Figura 3.4: Confronto dei tempi di esecuzione ottenuti con diversi device grafici e processori. Sono state svolte 1500 iterazioni con una popolazione unica composta da 100 individui.

Osservando la Figura 3.4 si evince subito come i due device grafici meno potenti non riescano in generale a superare le performance dei due processori, mentre la GeForce di fascia alta si comporta molto meglio. Questo è probabilmente dovuto al fatto che quest'ultima – oltre ad essere di classe superiore – è costruita secondo un'architettura più recente e ottimizzata (architettura nVidia Fermi).

In ogni caso, la realizzazione proposta non pretende certo di essere la migliore possibile e lascia sicuramente spazio a ulteriori ottimizzazioni. Inoltre – come già detto – eseguire il test su un unico blocco non consente di sfruttare appieno le potenzialità di calcolo di un elaboratore grafico. Per questo è stato svolto un ulteriore esperimento che mira a verificare se è possibile ottenere risultati migliori eseguendo l'algoritmo su più isole, in modo da utilizzare più blocchi CUDA nella stessa esecuzione.

La Figura 3.5 mostra dei risultati più incoraggianti: non traendo alcun vantaggio dall'esecuzione di più isole, le versioni per CPU dell'algoritmo eseguono molto più lentamente. Si ottengono invece risultati eccellenti con l'elaboratore grafico di fascia più alta, in grado di raggiungere uno speedup medio pari a 6.9 con solo 10 isole e 48.9 utilizzandone 100, a conferma del fatto che utilizzare più blocchi GPU consente al codice di girare su più stream processors (SP) contemporaneamente. Anche le prestazioni della GPU 8600M (in giallo) sono decisamente migliori rispetto al caso precedente, mentre quelle del device di fascia più bassa restano ancora deludenti. Questa differenza è ben comprensibile se si tiene conto del numero di SP disponibili in ogni device e della loro frequenza di funzionamento: il componente GeForce560 mette a disposizione 384 CUDA Cores complessivi spinti a 1.66GHz, nel 8600M GT ne sono disponibili 32 funzionanti a 0.94GHz, mentre la GPU 8400GS dispone di appena 2 CUDA Cores alla frequenza di 1.35GHz. Avere a disposizione più unità di calcolo permette quindi di raggiungere uno speedup migliore in quegli scenari dove è possibile mappare l'algoritmo su più blocchi CUDA. Quando il calcolo si sposta su un unico SP quelli restanti vengono di fatto sprecati e la velocità di esecuzione è condizionata soltanto dalla frequenza di lavoro del dispositivo.

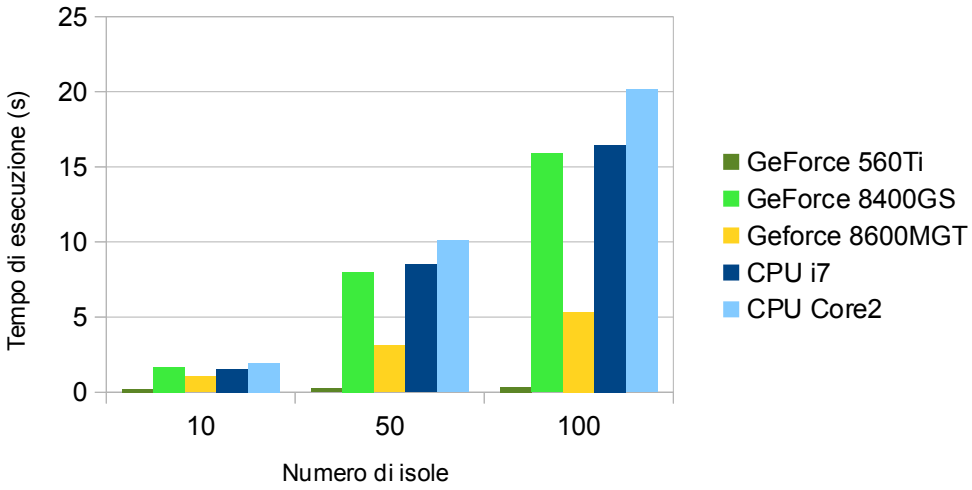

Figura 3.5: Confronto dei tempi di esecuzione ottenuti con popolazioni composte da più isole. Vengono svolte 500 iterazioni con isole da 50 individui usando una istanza da 100 oggetti.

Si deve comunque ricordare che – per come è stato realizzato il progetto – risolvere un'istanza utilizzando più isole non apporta significativi miglioramenti nella qualità della soluzione finale visto che non è stato implementato alcun operatore di migrazione e le diverse popolazioni sono del tutto indipendenti tra loro.

4. Conclusioni

Il progetto presentato si è dimostrato particolarmente utile dal punto di vista didattico per due motivi fondamentali: da un primo punto di vista ha guidato ad uno studio di applicazioni concrete dei concetti di *evolutionary computing* riguardanti gli algoritmi genetici; in via secondaria esso ha offerto l'occasione per esplorare alcuni principi fondamentali della programmazione parallela e – in particolare – ha permesso di ottenere una certa familiarità col framework CUDA di nVidia per la realizzazione di applicazioni SIMD su GPU.

La fase di analisi del GA preso in esame ha richiesto una fase preliminare di documentazione sullo stato dell'arte di algoritmi genetici realmente implementati e utilizzati, con particolare attenzione verso le diverse tipologie e metodologie dei principali operatori genetici. Le fasi successive di test e messa a punto hanno inoltre dimostrato quanto l'effettiva qualità di questo genere di algoritmi sia in realtà complessa da quantificare e quanto il loro comportamento dipenda enormemente dalle scelte fatte durante la loro progettazione, dalle singole istanze del problema e – in particolar modo – dai parametri di esecuzione riguardanti le probabilità legate ai vari operatori. Dal collaudo dell'applicazione è emerso che l'algoritmo realizzato è funzionante e offre soluzioni di buona qualità nella quasi totalità delle istanze generate. Risultati migliori si ottengono tuttavia soltanto dopo una prima fase in cui è necessario perfezionare la scelta dei parametri in input. Per questo motivo si sottolinea come l'aggiunta di un operatore di migrazione che permetta l'esecuzione dell'algoritmo su più isole con diverse impostazioni selezionabili per ciascuna popolazione possa portare ad un notevole incremento della qualità delle soluzioni per qualsiasi tipologia di istanze di problema. È stato anche evidenziato come questa estensione porterebbe ad un migliore utilizzo delle potenzialità di calcolo del device grafico e quindi a un sensibile aumento delle prestazioni.

Gli sforzi fatti per poter parallelizzare l'algoritmo e renderlo così eseguibile su architettura CUDA hanno rappresentato un utile (e talvolta arduo) banco di prova per meglio comprendere le problematiche caratteristiche di questa piattaforma e di architetture SIMD analoghe. I test di performance effettuati sul codice sviluppato hanno mostrato dei risultati in generale positivi ma non sempre eccellenti in presenza di hardware di fascia bassa. Nel precedente capitolo si è cercato di trovare una giustificazione a questo comportamento alla luce dei dati ottenuti su più device differenti.