Tirocinio Curriculare
16/10/2017 – 31/12/2017

ALGORITMI GENETICI PARALLELI SULL'ARCHITETTURA CUDA

Tirocinante
Luka Micheletti

Tutor
Moreno Marzolla

Introduzione

Lo scopo del progetto è di implementare un algoritmo genetico parallelo.
Un algoritmo genetico è un algoritmo euristico ispirato al concetto di evoluzione darwiniana, utilizzato per la risoluzione di problemi di ottimizzazione. Gli algoritmi genetici partono da una popolazione di soluzioni pseudo-casuali al problema scelto (le soluzioni, in generale, possono essere anche non ammissibili, ma nel caso di studio preso in esame si è scelto di considerare solo soluzioni ammissibili), associate, nel paragone con la genetica naturale, agli individui della popolazione stessa.
Su tali individui sono applicate tecniche di selezione, incrocio e mutazione dei singoli "geni". Queste operazioni vengono ripetute un numero di volte pari al numero di "generazioni" desiderato.
L'obiettivo di ogni generazione dell'algoritmo è aumentare il "fitness" delle soluzioni, ovvero renderle più simili alla soluzione ottima del problema. Idealmente, selezionando e incrociando le soluzioni migliori di ogni generazione (in base al loro punteggio), è possibile raggiungere la soluzione ottima; in tal caso si dice che l'algoritmo raggiunge la convergenza.
Essendo l'approccio euristico, le soluzioni finali possono essere non ottime, ma ciò non pregiudica l'efficacia del programma, in quanto per molti problemi anche soluzioni sub-ottime possono risultare molto utili.
L'obiettivo delle versioni parallele è di fornire un aumento sensibile delle prestazioni dell'algoritmo seriale attraverso tecniche di programmazione parallela, così da ridurre il più possibile il tempo di computazione.
Il secondo obiettivo del tirocinio è far si che la versione parallela dell'algoritmo genetico sia anche generica, così che possa essere applicata al maggior numero possibile di problemi di ottimizzazione. Raggiunto questo obiettivo, l'implementazione parallela implementata potrebbe essere usata per risolvere problemi di tipo diverso, rendendosi un utile strumento per l'accelerazione del calcolo degli algoritmi genetici in qualunque ambito di ricerca.
Idealmente il programma implementato dovrebbe garantire un incremento di prestazioni e una estensibilità elevata.

Tecnologie

Il progetto è stato sviluppato sfruttando l'architettura CUDA, usata per la programmazione su GPU NVIDIA e distribuita dalla stessa azienda.
Per confronto è stata realizzata una versione che esegue su CPU e sfrutta il parallelismo fornito dal framework OpenMP per la parallelizzazione a memoria condivisa.
Il codice è scritto in linguaggio C++ e testato in ambiente Linux, sulle distribuzioni Ubuntu 16.04, 17.04, 17.10 e Mint 18.2, sfruttando diverse configurazioni di CPU e GPU: Intel Core i5-5200U con NVIDIA GeForce GT 920m, Intel Xeon e5-2603 v4 con NVIDIA GeForce GTX 1070, Intel Core i7-5820K con NVIDIA GeForce GTX

Titan X, Intel Core2 Duo E6750 con NVIDIA Tesla C870. I risultati sullo speedup sono stati calcolati eseguendo le varie versioni dell'algoritmo sulla macchina che monta Intel Core i7-5820K e NVIDIA GeForce GTX Titan X, ripetendo le misurazioni cinque volte per ogni caso esaminato e calcolando lo speedup sulla media dei valori ottenuti.

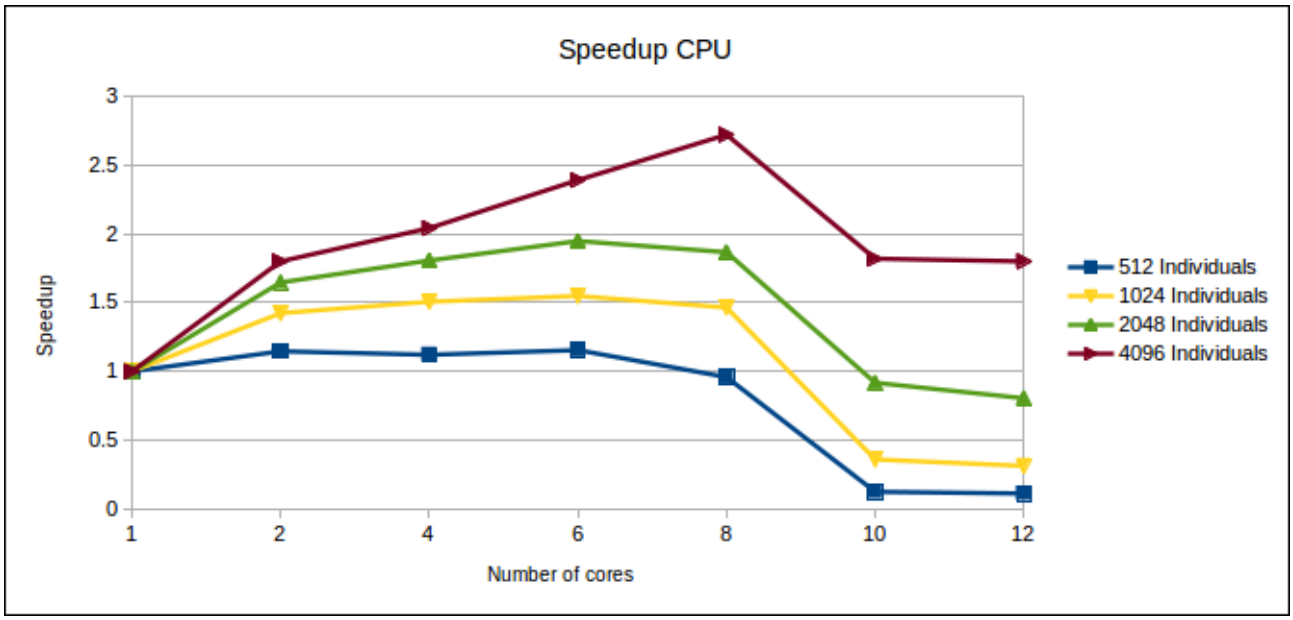
Attività

Il progetto prevede un confronto tra approcci diversi per il calcolo dello speedup; per tale ragione è stato necessario sviluppare diversi programmi che risolvessero lo stesso problema sfruttando metodi differenti.
Il caso di studio usato per testare le varie versioni sviluppate è il problema del commesso viaggiatore (TSP), essendo un problema NP-completo, cioè con costo computazionale non polinomiale per il calcolo della soluzione ottima.
Il problema del commesso viaggiatore consiste nel trovare il percorso più breve per visitare una serie di tappe distribuite in un area stabilita. Ogni tappa deve essere visitata una e una sola volta e alcune definizioni del problema, come quella adottata in questo progetto, specificano che il punto di partenza debba coincidere con il punto di arrivo.
Le soluzioni del problema preso in esame sono codificate tramite un array di punti 2D, a loro volta rappresentati da una struttura contenente due interi, che identificano le coordinate delle varie tappe del commesso viaggiatore. L'ordine dei punti 2D nell'array indica il percorso della soluzione, cioè i singoli "geni" di un individuo.
Sono state implementate tre versioni differenti dell'algoritmo genetico per la risoluzione del problema in esame: una che esegue unicamente su CPU e che sfrutta il parallelismo fornito dalle diverse unità di calcolo presenti all'interno di un moderno processore (core), una specifica per tale problema che esegue su GPU e una che esegue su GPU, ma generica rispetto al problema da risolvere.
Essendo l'architettura CUDA organizzata in thread, blocchi di thread e griglie di blocchi, è stato necessario, per le versioni su GPU, mappare il dominio del problema sulle strutture fornite dalla libreria (thread, blocchi e griglie). La mappatura pensata prevede una suddivisione della popolazione iniziale in sottopopolazioni dette "famiglie": viene creata una sola griglia, in cui ogni blocco è mappato su una famiglia e ogni thread del blocco su un singolo individuo della famiglia stessa. La mappatura del dominio su thread e blocchi consente di ottimizzare l'algoritmo sfruttando le memorie ad alta velocità presenti nel chip della GPU (shared memory), che sono condivise fra i thread di un blocco e non fra blocchi diversi.

-Versione CPU

La versione per CPU implementa un algoritmo genetico elitista classico:
creata la popolazione viene eseguito il loop che richiama le operazioni di selezione, incrocio e mutazione. L'individuo migliore di ogni generazione è mantenuto nella generazione successiva se ha un punteggio migliore del nuovo individuo con punteggio più alto.

Il numero di generazioni, la dimensione della popolazione, il numero di tappe delle soluzioni, la dimensione dello spazio delle tappe e le probabilità di incrocio e di mutazione sono modificabili dall'utente, in modo da permettere test con parametri diversi da quelli definiti in partenza.
Questa versione non è specifica per il problema del commesso viaggiatore, ma sufficientemente generica da poter essere estesa ad altri problemi di ottimizzazione.
Per ottenere la generalità necessaria è stato utilizzato il paradigma a oggetti messo a disposizione dal linguaggio C++, in particolare sono presenti diverse classi, una che rappresenta la popolazione di individui ("Population"), e che implementa il cuore dell'algoritmo genetico, cioè il ciclo evolutivo, e una classe astratta che rappresenta il singolo individuo ("Genome"). Tale classe offre un interfaccia che comprende metodi come "evaluate", necessario per la selezione, "crossover" e "mutate", che implementano rispettivamente l'incrocio tra due individui e la mutazione di singoli geni all'interno di un individuo.
L'estensione della classe "Genome" per il problema del commesso viaggiatore è "PathGenome" e implementa i metodi citati nel modo seguente:
"evaluate" calcola il percorso totale dell'individuo sommando le distanze tra i vari punti e usa il risultato ottenuto come punteggio dell'individuo (più il punteggio è alto, minore è la probabilità che l'individuo venga selezionato per eventuali incroci), "crossover" prende in ingresso due individui, cioè il "partner" nell'incrocio e il figlio e assegna al figlio i primi K geni da se stesso, con K generato pseudo-casualmente, e i restanti dal partner, avendo cura di evitare duplicazione di geni, che genererebbero soluzioni non ammissibili. "mutate" effettua un calcolo di probabilità per ogni gene dell'individuo (punti 2D) generando un numero pseudo-casuale compreso tra 0 e 1 e confrontandolo con la probabilità di mutazione specificata dall'utente, e scambiando il gene con uno selezionato pseudo-casualmente tra gli altri in caso di calcolo favorevole.
Nel grafico seguente è rappresentato lo speedup della versione CPU parallela rispetto a quella seriale al variare del numero di core usati.
Sono evidenziate le curve di speedup per diverse dimensioni della popolazione.



Si nota come lo speedup ottenuto non sia direttamente proporzionale al numero di thread utilizzati. Questo perché non tutto il programma può essere parallelizzato, e la parte intrinsecamente seriale limita l'aumento di prestazioni.
Un altro fattore che limita la performance è il numero di core fisici disponibili sul proccessore usato per le prove: la CPU usata ha 6 core fisici con hyperthreading, che risultano in 12 core virtuali. Nonostante l'hyperthreading consenta di migliorare le prestazioni, superati i 6 thread (corrispondenti ai core fisici del processore) è lecito aspettarsi un aumento dello speedup meno marcato.
Nonostante queste considerazioni, si riscontra un effetto inaspettato con l'esecuzione del programma su 11 e 12 thread: con una popolazione relativamente bassa le prestazioni dell'algoritmo peggiorano rispetto alla versione seriale, mentre con popolazioni relativamente elevate raggiungono quelle ottenute con 2 thread.

-Versione GPU non generica

La prima versione CUDA sviluppata è quella "cablata" sul problema del commesso viaggiatore. In questa versione non esistono classi (il paradigma usato è procedurale) e la gestione della memoria, soprattutto su GPU, è mirata all'ottimizzazione dell'algoritmo per questo specifico problema.
Le API CUDA, per gestire la programmazione su GPU, sfruttano l'astrazione host-device, intendendo con host la CPU e la memoria centrale (RAM), e con device la GPU e la VRAM e chiamando kernel una funzione che esegue su GPU. La copia dei dati tra host e device è relativamente lenta, richiedendo la comunicazione tramite bus PCI-e. Per questo motivo, il ciclo dell'algoritmo genetico è eseguito in un kernel e i dati vengono copiati da host a device prima della chiamata al kernel e da device a host a chiamata terminata.
Una volta creati e inizializzati casualmente gli individui, tutta la popolazione viene copiata in VRAM e il kernel "evolve" viene lanciato. La funzione "evolve" prende in ingresso la popolazione e il numero di generazioni da iterare e, prima di eseguire il ciclo dell'algoritmo, copia le singole famiglie in shared memory. Ogni thread copia l'individuo con il proprio indice in shared memory, così che possa essere letto e modificato più velocemente rispetto allo stesso in memoria globale. L'ultima operazione che il kernel esegue, dopo il ciclo dell'algoritmo, è la copia delle famiglie da shared memory a memoria globale, così che possano essere ricopiate successivamente nella memoria dell'host.
All'interno del ciclo evolutivo, le funzioni device richiamate sono: "evaluate", "select", "crossover" e "mutate".
Nella funzione "evaluate", ogni thread assegna all'individuo corrispondente un punteggio calcolato come somma delle distanze tra ogni tappa. Tale punteggio è sfruttato dalla funzione "select", nella quale ogni thread seleziona un individuo a caso nella propria famiglia e confronta il suo punteggio con il proprio. L'individuo con punteggio minore viene salvato per la fase di incrocio.
Nella funzione "crossover" ogni thread genera un nuovo individuo partendo da quello a sé associato e da un partner, scelto in base al proprio indice tra quelli salvati in fase di selezione. La prima parte della funzione genera un indice casuale, che identifica un gene all'interno dell'individuo; come per la versione CPU, i geni precedenti a quello

identificato sono presi dal primo individuo, mentra i seguenti dal partner, avendo cura di evitare duplicazioni e di mantenere l'ordine nell'inserimento.
Anche la funzione "mutate" implementa la mutazione in modo simile alla versione CPU: ogni thread cicla i singoli geni dell'individuo ad esso associato e ha una probabilità di scambiare ogni gene con uno a caso tra gli altri dell'individuo.
Il ciclo evolutivo nel kernel "evolve" è strutturato nel modo seguente:

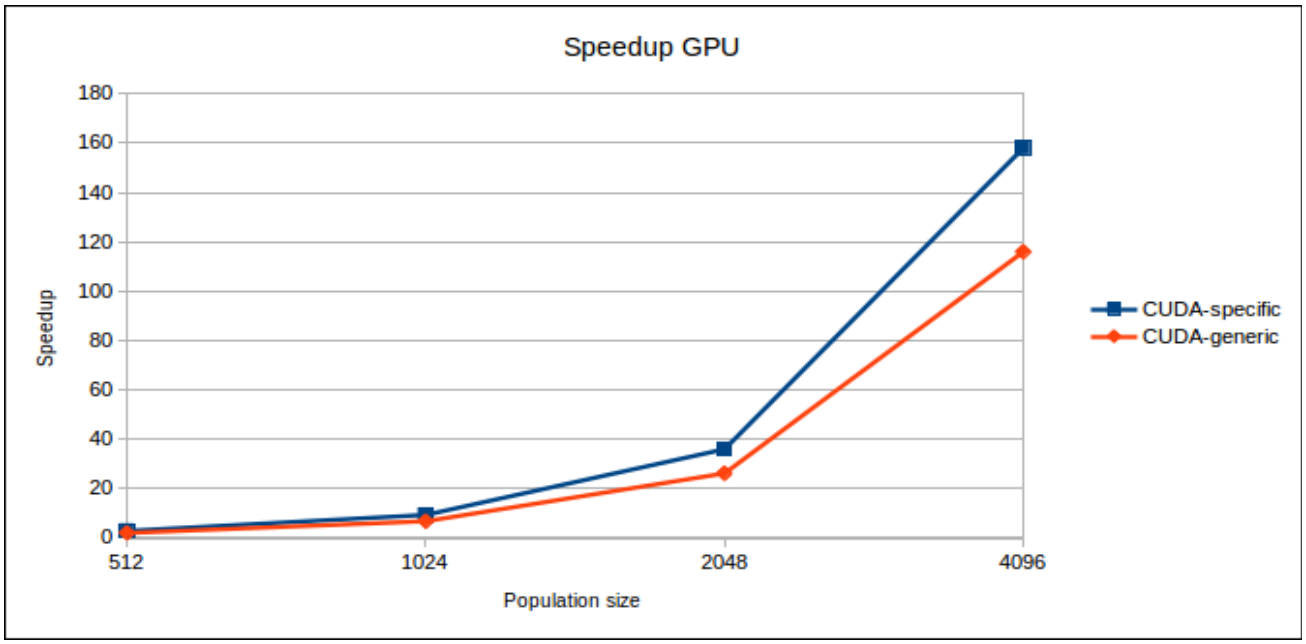
```
for (i = 0; i < NUM_GEN; i++) {
   evaluate(family);
   synchronize();

   select(family, fitters);
   synchronize();

   if (random(0, 1) < CROSS_PROB) {
      crossover(family, fitters);
   }
   synchronize();

   mutate(family);
   synchronize();
}
```

Si nota come la probabilità di incrocio e quella di mutazione siano usate in modo differente: per quanto riguarda l'incrocio, viene verificata la probabilità una sola volta per scegliere se eseguire la funzione o no, mentre per la mutazione, la probabilità è applicata per ogni singolo gene all'interno di un individuo.
Per questo motivo, essendo le operazioni di generazione di numeri casuali relativamente onerose computazionalmente, la funzione "mutate" rappresenta uno dei punti più dispendiosi, in termini di calcolo, soprattutto per istanze del problema che prevedono molte tappe.
Il parametro "fitters" indica un array contenente gli indici degli elementi selezionati dalla funzione select come partner per la funzione crossover.

-Versione GPU generica

La versione generica su GPU ha lo stesso funzionamento di quella specifica, ma sfrutta meccanismi messi a disposizione dal linguaggio C++ per generalizzare il genoma. Una soluzione troppo generica, tuttavia rischia di avere un impatto negativo sulle prestazioni del programma, perciò si è scelto di usare i template: come nella versione specifica, è presente una funzione kernel "evolve", che, però prende in ingresso una popolazione di un tipo di dato generico. La funzione "evolve" è implementata in un file esterno, così che possa essere portabile e applicabile a diversi problemi.
Per poter risolvere un problema di ottimizzazione sfruttando l'algoritmo genetico parallelo implementato in questa versione del programma è necessario mappare il

dominio delle soluzioni del problema in una struttura dati adatta e implementare le funzioni di valutazione, selezione, incrocio e mutazione su tale struttura dati. Le funzioni vengono richiamate dal kernel “evolve”, e hanno una signature definita, che deve essere rispettata da tali implementazioni specifiche.
Nel grafico seguente è mostrato lo speedup delle due versioni CUDA, calcolato rispetto alla versione seriale:



Come si nota, lo speedup quadruplica per ogni raddoppio della dimensione dell’input, il che indica un andamento lineare: la curva mostrata nel grafico suggerisce un andamento esponenziale a prima vista, ma tale effetto è dato dal fatto che la scala dell’asse delle ascisse è logaritmica.
Dal grafico si capisce che la versione non generica offre un aumento di prestazioni superiore rispetto alla versione generica, questo perché la prima ha un’implementazione simile alla seconda, ma ad hoc per il problema del commesso viaggiatore. Nonostante ciò, lo speedup fornito dalla versione generica è sufficientemente elevato da giustificarne l’utilizzo in molti casi di studio.
Nonostante l’andamento prometta un aumento dello speedup anche per dimensioni più grandi, c’è da aspettarsi che la curva subisca un rallentamento o addirittura un decremento, a causa della limitata dimensione delle shared memory sul device: la popolazione viene interamente copiata in shared memory, perciò per popolazioni troppo grandi la memoria condivisa può non bastare.

Le misurazioni, sia per le versioni GPU, sia per quella CPU, sono state effettuate mantenendo tutti i parametri costanti (15 tappe, dimensione del range delle tappe di 500x500, probabilità di incrocio del 100%, probabilità di mutazione dello 0,1% e 1000 generazioni) e su dimensioni della popolazione di 512, 1024, 2048 e 4096 individui. La scala logaritmica è un fattore da tenere in conto nelle valutazioni dello speedup.

Conclusioni

Gli obiettivi del progetto sono stati raggiunti, in quanto le tre versioni implementate forniscono i risultati aspettati e la versione parallela generica garantisce un sostanziale aumento di performance e un relativamente elevato grado di estensibilità, nonostante preveda una certa conoscenza dell'ambiente CUDA da parte dell'utente che intende utilizzare l'algoritmo per problemi diversi.
Le misurazioni sono state effettuate variando la dimensione della popolazione, parametro più indicativo della differenza di speedup, ma possono essere fatti test variando tutti gli altri parametri in combinazioni diverse. Variare il numero di tappe delle soluzioni, per esempio, può influire profondamente sull'aumento di performance, in quanto impatta direttamente sull'uso efficiente delle shared memory.