SAPIENZA
UNIVERSITÀ DI ROMA

Uno Sguardo al Futuro della Programmazione

Facoltà di Ingegneria
Corso di laurea in Informatica
Cattedra di Ingegneria degli Algoritmi

Candidato
Umberto Sonnino
Matricola 1237312

Relatore
Camil Demetrescu

Sessione Invernale

Anno Accademico 2011/2012

A Tati e Setta, Maria e Giannina le mie Guide.
A Luigi e Guido ispirazione da sempre.
A Renato e Marina.

Capitolo Pagina

Introduzione

Da vari anni ormai l'aumento della potenza di calcolo delle CPU e' in declino: non potendo piu' aumentare in modo considerevole la frequenza di clock e avendo seri problemi a superare l'ostacolo degli accessi alla memoria, ormai si sente il bisogno di trovare nuove risposte alle crescenti esigenze.

A differenza delle CPU, le unita' di processamento grafico (GPU - *Graphics Processing Unit)* rappresentano la vera avanguardia, ricevono una grossa spinta da un'industria milionaria come quella dei videogiochi: ormai le schede grafiche riescono a gestire grandi quantita' di dati e sfruttano i meccanismi delle pipeline in parallelo, permettendo cosi di nascondere la latenza dovuta agli accessi in memoria.

E' dunque evidente che la strada da intraprendere e' quella della programmazione in parallelo: un contributo importante nell'industria e' stato dato negli ultimi anni da NVIDIA, ATI e Apple, con la pubblicazione di librerie e ambienti di sviluppo che hanno permesso ai piu' audaci di intraprenderla. Ormai sono decenni che i programmatori di tutto il mondo si occupano di sviluppare efficienti algoritmi lineari, l'esecuzione dei quali comporta una serie di istruzione eseguite consecutivamente. Ora grazie all'avvento delle architetture hardware, ma soprattutto di quelle software, e' necessario tornare indietro per riformularli e riconsiderare tale approccio.

In questa tesi di laurea di primo livello sperimentale e' stato sfruttato l'ambiente di sviluppo in CUDA, acronimo di *Compute Unified Device*

Architecture, creato da NVIDIA nel 2007. L'architettura di CUDA include in linguaggio assembly (PTX) e una tecnologia di compilazione che e' la base su cui molte tecnologie di calcolo parallelo sono sviluppate sulle GPU NVIDIA; in particolare e' stata utilizzata l'estensione di C denominata CUDA-C. Il modello di programmazione in CUDA e' un modello eterogeneo in cui sia la CPU che la GPU vengono utilizzate. In CUDA il termine *host* si riferisce alla CPU e la sua memoria, mentre il termine *device* si riferisce alla GPU e la sua memoria. Il codice che gira sul *host* dovra' gestire la memoria anche sul dispositivo, e lancera' i cosidetti *kernel,* che sono funzioni eseguite sul dispositivo: questi *kernel* vengono eseguiti da numerosi thread della GPU in parallelo.

Il progetto di tesi ha comportato lo sviluppo di un clone del celeberrimo gioco di Tetris come *case-study*, per analizzare le prestazioni di un'intelligenza artificiale che lo gioca. Tale intelligenza artificiale e' stata prima sviluppata soltanto sul *host* e poi, sfruttando il nuovo paradigma, con CUDA-C sul *device*; i risultati vengono di seguito esposti e rappresentano un banale esempio di come la programmazione in parallelo potra' essere usata in futuro. Non banale invece e' stata l'architettura vera e propria del progetto, che ha richiesto un nuovo modo di ragionare e di rapportarsi al problema della programmazione in parallelo.

Threads, Blochi, Griglie..?!

Dal punto di vista di uno studente, studiare C e' sempre stato un piacere poiche', in quanto un linguaggio di basso livello, ha permesso di ottenere risultati soddisfacenti, scrivendo codice che fossi in grado di controllare in ogni sua minima parte. Arrivando a lavorare con CUDA, tutto cio' stato stravolto.

Quando si utilizza un singolo processore, si segue un unico filo d'esecuzione, gestendo in modo facile e chiaro cio' che sta avvenendo. Quando si impiega un dispositivo basato su CUDA, tutto cio' cambia, e dalla singola nozione di thread, che esiste in ogni sistema operativo, si amplia l'idea arrivando ad utilizzare blocchi e griglie.

In CUDA quelle che in C vengono chiamate funzioni e in Java metodi, sono detti kernels; pertanto, scrivere un programma che sia eseguito in parallelo sulla GPU vorrà dire scrivere un kernel. Poiche' l'architettura su cui si lavora e' in grado di processare, grazie alla presenza di numerosi core fisici, una quantita' impressionante di threads in parallelo, ogni chiamata a kernel dovra' specificare il numero di blocchi e la quantita' di threads che possono girare in un singolo blocco. Come poter allora accedere ad ogni thread nel singolo blocco? Una variabile di stato all'interno del Kernel ci permette di identificare il thread su cui il programma sta girando al momento. Il tutto si calcola come se ci si trovasse all'interno di una matrice row-major, per cui l'ID del thread si ottiene moltiplicando l'indice del blocco del thread (blockIdx.x) per il numero di threads in ogni blocco (blockDim.x); a questo numero si somma, dunque, l'indice del thread (threadIdx.x).

A questo punto risulta percio' importante chiarire qual e' l'idea dietro a blocchi, griglie e threads su cui si fonda l'esecuzione in parallelo di CUDA. Diamo una definizione piu' precisa:

- *Thread:* rappresenta soltato l'esecuzione del kernel(funziona CUDA) grazie al suo indice;

- *Blocco:* e' un gruppo di threads. Questo gruppo di thread puo' funzionare sia in concorrenza che serializzato, in nessun ordine particolare. Tale processo puo' essere controllato tramite la chiamata alla funzione `_syncthreads()`, che si occupa di fermare l'esecuzione di un thread ad un certo punto del kernel(funzione CUDA) fino a quando tutti i thread nello stesso blocco non risultano sincronizzati.

- *Griglia:* e' un raggruppamento dei blocchi. Non ci sono in questo caso opzioni di sincronizzazione.

Tale rappresentazione e' molto simile a quella dei vettori in C, in quanto si hanno vettori unidimensionali(threads), bidimensionali(matrici - blocchi), tri-dimensionali (cubi - griglie).

La cosa importante da capire e' come tali griglie, blocchi e thread vengono eseguiti:

- *Griglia:* un'intera griglia viene eseguita da un singolo chip della GPU;

- *Blocco:* la GPU ha a disposizione una quantita' di multiprocessori(MP), ognuno dei quali e' responsabile di uno o piu' blocchi all'interno di una griglia. Non accadra' mai che un blocco venga diviso su diversi multiprocessori;

- *Thread:* i multiprocessori sopracitati sono composti da un certo numero di *Stream Processors* (SP - processori di flusso), ognuno dei quali si occupa della gestione di uno o piu' threads in un blocco.

Ogni SP, che da qui in avanti chiameremo più semplicemente *core*, può eseguire pertanto un thread in modo sequenziale, anche se i core si comportano secondo lo schema che NVIDIA chiama SIMT(Single Instruction, Multiple Threads); tutti i core nello stesso blocco si occupano dell'esecuzione della stessa istruzione allo stesso tempo (uno schema molto simile al piu' classico SIMD - Single Instruction, Multiple Data, che si e' dimostrato essere molto efficiente in termini di parallelismo). Il codice viene eseguito in gruppi di 32 threads, che NVIDIA chiama *warp* (distorsione). Un warp puo' essere definito piu' semplicemente come il numero di thread che sta eseguendo istruzioni in un multiprocessore.

NVIDIA ha progettato anche una zona di memoria condivisa gestibile tramite software che e' a disposizione di tutti i core: e' una memoria bassa latenza, dotata di ampia banda ed e' incidizzabile. Tutto ciò vuol dire che tale memoria riesce a raggiungere le velocita' dei registri(le più recenti schede NVIDIA supportano 64KB di memoria).

Uno dei più noti colli di bottiglia e' legato alle operazioni in memoria, che ovviamente richiedono grandi tempi di attesa dovuti alla lunga latenza, che portano spesso allo spreco di centinaia di cicli di clock. Poiché, inoltre, le GPU sono state disegnate per stream o throughput computing, l'utilizzo di cache risulta inutile. Per tollerare dunque queste alte latenze, le GPU sfruttano al massimo il multithreading: quando un warp si trova in attesa in un'operazione in memoria, il multiprocessore seleziona un altro warp che e' pronto all'esecuzione. In questo modo i cores saranno produttivi fino a quando ci sarà abbastanza parallelismo da mantenerli occupati.

Se tramite terminale facciamo partire il programma di prova `deviceQuery` possiamo visualizzare le prestazioni della scheda video:

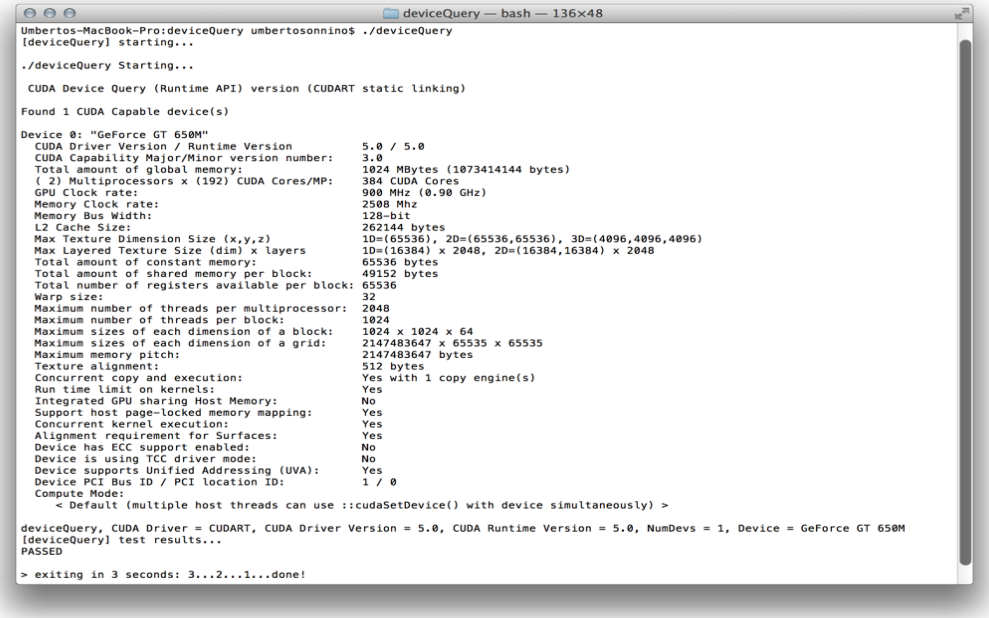

Figura 1. Prestazioni in Cuda

La GPU dispone di 1024 MB di SDRAM, ovvero di memoria globale. Ogni multiprocessore nella GPU ha accesso a 48 KB di memoria condivisa e 65536 registri; sono anche presenti 64KB di memoria per tutti i multiprocessori nella GPU.

Possiamo anche vedere le dimensioni dei blocchi e delle griglie. I blocchi possono essere in uno, due o tre dimensioni (x, y, z) dove x, y e z risultano essere pari a 1024, 1024 e 64 rispettivamente, cosi che $x \times y \times z \leq 1024$, che rappresenta il massimo numero di thread per ogni blocco. Abbiamo gia' visto che i blocchi sono organizzati a loro volta in griglie, che possono arrivare, nuovamente, fino a tre dimensioni[1], fino a 65535 $(2^{16}-1)$blocchi in ogni dimensione.

Il limite inferiore risulta dunque essere la presenza di 1024 threads per blocco, dovuti evidentemente alla presenza del piccolo numero di registri che possono essere allocati durante l'esecuzione dei threads in tutti i blocchi assegnati al multiprocessore. Ovviamente questo limite si manifesta

[1] Al momento NVIDIA supporta soltanto 1 come terza dimensione nella griglia, rendendo di fatto tutte le griglie al massimo di 2 dimensioni.

proprio perche' solo i thread all'interno di un singolo blocco possono sincronizzarsi e scambiarsi dati attraverso la memoria condivisa in un MP.

Alla luce di queste osservazioni, possiamo concludere che per ottenere le migliori prestazioni a livello di GPU e' necessario:

- Trovare il modo parallelizzare tante piu' operazioni possibili cosi da far lavorare tutti i multiprocessori;
- trovare il modo ulteriore di parallelizzare tante piu' operazioni possibli per far si che il multithreading mantenga costantemente i core occupati;
- ottimizzare gli accessi alla memoria del dispositivo (la GPU) per i dati contigui, limitando cosi gli elevati tempi di attesa dovuti alle operazioni di lettura/scrittura.

Prestazioni in Memoria

Ogni multiprocessore, come illustrato in figura, contiene quattro tipi di memoria:

1. Un set di registri locali per ogni thread;
2. della memoria condivisa da tutti i thread e implementa lo spazio di memoria condivisa;
3. una cache di sola lettura costante che viene condivisa da tutti i thread e aumenta la velocita' in lettura dalla memoria globale *(constant memory)*;
4. una cache di sola lettura per le texture che e' condivisa da tutti i processori e aumenta la velocita' in lettura dalla memoria globale (*texture memory)*.

La memoria 'locale' in figura viene concepita soltanto come un'astrazione per caratterizzare il modello, non una reale cache hardware a disposizione di ogni singolo multiprocessore. La memoria locale generalmente viene allocata in memoria globale dal compilatore e ha, dunque, le sue stesse prestazioni. E' pertanto usata dal programmatore come uno spazio dove vengono allocate variabili locali che non riuscirebbero altrimenti ad essere allocate nella memoria piu' veloce, come i registri, perche' evidentemente risultano gia' occupati.

Bisogna essere molto prudenti, quindi, nell'uso della memoria locale, perche' puo' essere la ragione di prestazioni lente. Compilando con

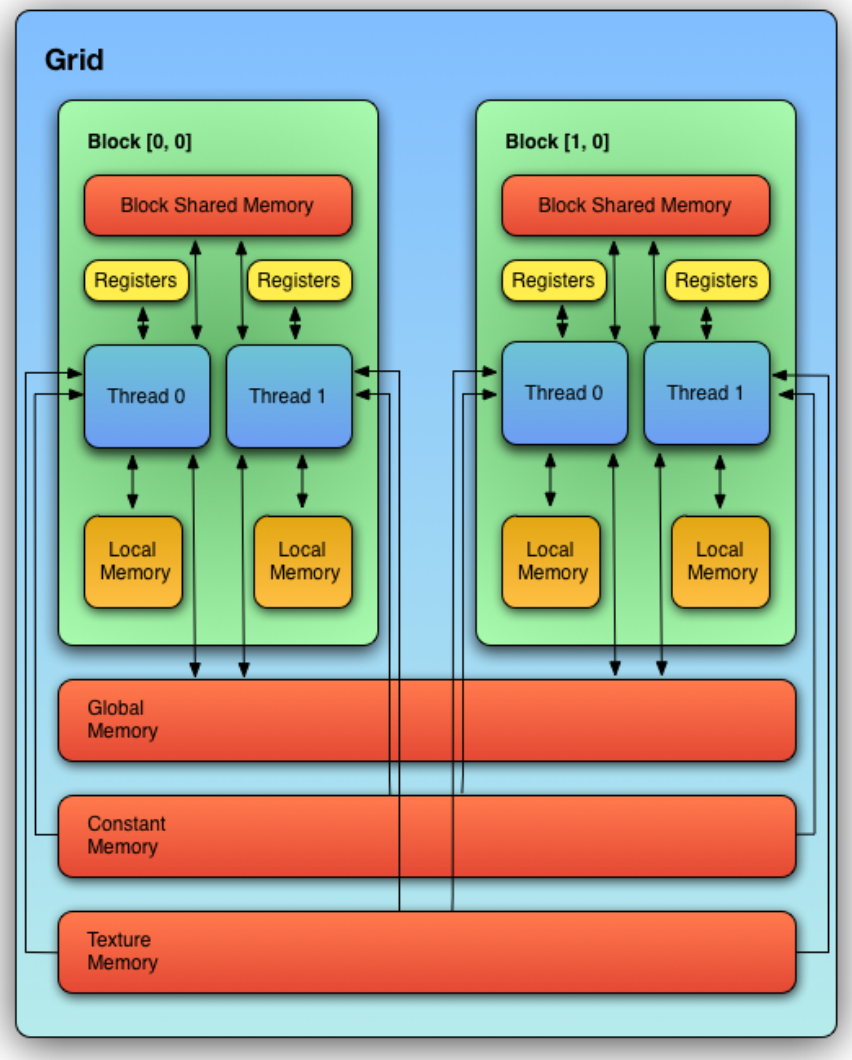

Figura 2. Struttura della memoria in CUDA

l'opzione -ptx possiamo ispezionare il codice assembly PTX per rivedere il nostro schema di allocazione all'interno dell'applicazione.

Analizziamo ora le prestazioni: un multiprocessore impiega 4 cicli di clock per mandare un'istruzione in memoria ad un *warp*. L'accesso alla memoria locale o globale provoca un'addizionale latenza di 400-600 cicli di clock.

Vediamo un esempio:

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

Con il codice sopra, l'operatore di assegnazione impiega 4 cicli di clock per far eseguire la lettura, 4 cicli di clock per la far eseguire scrittura sulla memoria condivisa, e 400-600 cicli di clock per leggere il *float* dalla memoria globale. (`__device__` e' un tipo che denota una variabile che risiede in memoria globale. Le variabili di tipo `__device__` non possono essere accedute dal codice eseguito sull'host).

Ovviamente con un fattore che varia tra il 100x e i 150x risulta essere ovvio che gli sviluppatori hanno bisogno di minimizzare gli accessi in memoria globale e riutilizzare i dati all'interno della memoria dei singoli multiprocessori. Dal punto di vista di CUDA, gli ingegneri di NVIDIA hanno fatto un ottimo lavoro per quanto riguarda lo scheduler dei thread.

La memoria condivisa (shared memory) in CUDA e' divisa in moduli di uguale grandezza chiamati *memory banks*. Ognuno di essi puo' contenere un valore a 32 bit --che sia un int o un float-- cosi che array consecutivi possono essere acceduti da thread consecutivi molto velocemente. Gli errori avvengono quando si mandano richiesti di accesso ad uno stesso *bank* . In questo caso l'hardware tende a serializzare tutte le operazioni che conseguentemente risultano essere rallentate, poiche' i thread devono attendere fino a quando tutte le richieste in memoria non vengono soddisfatte.

Quindi per riassumere:

- Registri:
  - il tipo piu' veloce di memoria sul multiprocessore;

        - soltanto accessibile dal thread;
        - ha il tempo di vita di un thread.
    - Memoria Condivisa
        - Quando non ci sono conflitti sui *memory banks* risulta essere veloce quanto i registri;
        - accessibile da qualunque thread all'interno di un blocco;
        - ha il tempo di vita del blocco.
    - Memoria Globale
        - Potenzialmente x150 piu' lenta dei registri o della memoria condivisa;
        - accessibile sia dall'host, che dal device(GPU);
        - ha il tempo di vita dell'applicazione
    - Memoria Locale
        - Potenzialmente molto lento;
        - accessibile soltanto dal thread;
        - ha il tempo di vita di un thread;

Poiché la memoria condivisa risulta essere decisamente più veloce della memoria globale o locale, e' importante sapere come poterla sfruttare. Quando un *kernel* viene invocato in CUDA, abbiamo visto che sono presenti due parametri; NVIDIA ha messo a disposizione anche un terzo parametro (opzionale, in quanto la configurazione di default assume che non venga usata memoria condivisa), per specificare la quantita' di memoria condivisa utilizzabile nel lancio del *kernel*.

Programmazione in CUDA

Basandoci sull'architettura descritta nei paragrafi precedenti, CUDA ci offre un linguaggio di programmazione denominato CUDA-C (a tutti gli effetti una derivato di ANSI C) che ci da la possibilita' di interagire con la GPU e ormai ha raggiunto anche una discreta maturita' per quanto riguarda le librerie che la interfacciano. Scopo di questa tesi e' ovviamente di capire il funzionamento di CUDA e il suo modo di interagire con la scheda grafica, pertanto in questa sede non ci occuperemo delle eventuali librerie.

Per interfacciarci con l'intera architettura di calcolo, che si tratti della CPU o della GPU, CUDA-C mette a disposizione dei qualificatori di linguaggio:

- `__device__` definisce una funzione che viene eseguita sulla GPU e che puo' essere chiamata soltanto dalla GPU;
- `__host__` definisce una funzione che e' eseguita e puo' essere chiamata soltanto dal programma in esecuzione *(host)*. Rappresenta il valore di default di una funzione che non ha un qualificatore;
- `__global__` definisce un *kernel*, ovvero una funzione che e' chiamabile soltanto *dall'host* ma che puo' essere eseguito soltanto sul *device*. Una funzione `__global__` puo' avere soltanto tipo di ritorno *void*.

Una particolarita' interessante e' che `__host__` e `__device__` possono essere utilizzati congiuntamente, per ottenere una funzione che sia compilata per entrambi.

```
__host__ __device__ int number(){
    return 1;
}

__global__ void foo(int a){
    a = number();
}
```

Abbiamo visto precedentemente come gli accessi in memoria siano il collo di bottiglia prestazionale dell'architettura parallelizzata. CUDA ci mette a disposizione dei qualificatori per definire variabili che si trovano sull'*host* o sul *device*.

- `__device__` dichiara delle variabili che risiedono sul *device*. Se non viene usato un qualificatore piu' specifico la variabile cosi dichiarata risiedera' in memoria globale, ha tempo di vita dell'applicazione e potra' essere acceduta da tutti i thread in una griglia.
- Per utilizzare la memoria condivisa, CUDA utilizza oltre a `__device__`, il qualificatore `__shared__`. Queste variabili saranno accedute soltanto dai thread all'interno del loro blocco e avranno tempo di vita del blocco, pertanto alla fine dell'esecuzione del thread, tali variabili non saranno piu' accessibili.

Le variabili che non sono state qualificate, ma definite all'interno di una funzione che si trova sul *device* sono allocate nella memoria locale di un thread. Variabili condivise *(shared)* possono essere definite anche senza `__device__` in questo caso. Puntatori alla memoria globale e a quella condivisa sono supportati, ma solo se e' possibile indirizzare lo spazio di indirizzamento in memoria a tempo di compilazione, altrimenti il puntatore indirizzera' alla memoria globale.

```
//Un array che risiede in memoria globale
__device__ int array[10];

__global__ void kernel(float shared_variable){
     __shared__ int sh_array[10];
     int *pointer;

     if(shared_variable != 1){
          pointer = sh_array;
     }else{
          pointer = array;
     }
}
```

Infine sono presenti delle variabili *built-in,* che ci permettono di lavorare con threads, blocchi e griglie. Alcuni sono stati visti gia precedentemente nella descrizione del funzionamento della meccanica degli stessi e sono `blockIdx` e `threadIdx` che sono di tipo `uint3`. `uint3` e' un vettore tri-dimensionale che contiene tre valori interi e senza segno accessibili tramite i campi `x, y, z`. Le altre variabili built-in sono `gridDim` e `blockDim` che sono invece di tipo `dim3`; `dim3` e' identico a

`uint3`, con la differenza che i componenti non specificati vengono inizializzati a 1.

`gridDim` contiene le dimensioni della griglia e al momento puo' essere espressa soltanto in due dimensioni. `gridDim.z` quindi sara' sempre pari a 1. In maniera analoga `blockDim` contiene la dimensione del blocco. `blockIdx` ci restituisce l'indice del blocco all'interno della griglia e `threadIdx` invece rappresenta l'indice del thread all'interno del blocco.

Per controllare l'esecuzione di un *kernel* abbiamo due opzioni:

1. Invocare il *kerkel* tramite `<<<numBlocks, threadsPerBlock>>>` indicando cosi la dimensione della griglia che conterra' `numBlocks` blocchi e la dimensione di ogni singolo blocco che avra' `threadsPerBlock` threads;
2. una sequenza di chiamate a funzione:
    a. `cudaConfigureCall(dim3 gridDim, dim3 blockDim)` che configura il prossimo lancio del kernel con i parametri specificati;
    b. tramite `cudaSetupArguments(T arg, size_t offset)` si passano i parametri di lancio alla stack del kernel[1], `offset` definisce la posizione sullo stack a cui sara' posizionato l'argomento;
    c. `cudaLaunch(T kernel_p)` ci permette lanciare tramite un puntatore al kernel `__global__` che deve essere eseguito.

La memoria in CUDA viene gestita in maniera lineare[2], ovvero viene allocata in indirizzi di 32-bit che si possono indirizzare tramite puntatori. Per mantere la vicinanza a C, gli ingegneri di NVIDIA hanno implementato due funzioni: `cudaMalloc(void** devPtr, size_t size)` e `cudaFree(void* devPtr)`. Entrambe si comportano esattamente come in C, allocando memoria sul *device* e poi liberandola. Per gestire i passaggi in memoria fra *host* e *device* e' anche presente una funzione

[1] Kernel stack e' simile allo stack di funzione, ma si trova sul *device*

[2] Vi sono anche altri modi per gestire la memoria in CUDA, ma non verranno trattati

`cudaMemCpy( void *destination, const void *source, size_t count, enum cudaMemcpyKind kind)` che permette, tramite il *flag* rappresentato da `kind`, passaggi sia da host a device che viceversa.

Infine bisogna parlare del *parallelismo dinamico*, che permette ad un *kernel* CUDA di creare e sincronizzare del lavoro nidificato. In modo piu' semplice, possiamo dire che un *kernel* puo' essere chiamato da un *kernel* 'genitore' che puo' essere, ad esempio, sincronizzato sul completamento della funzione 'figlio'. Il *kernel* 'genitore' puo' quindi consumare i dati generati dal 'figlio' senza alcuna interazione della CPU.

```
__global__ ChildKernel(void* data){
    //Operate on data
}
__global__ ParentKernel(void *data){
    ChildKernel<<<16, 1>>>(data);
}
// In Host Code
ParentKernel<<<256, 64>>(data);
```

Il modello di esecuzione in CUDA e' basato su threads, blocchi e griglie, con i *kernel* che definiscono le operazioni in esecuzione in un singolo thread. Il *parallelismo dinamico* estende l'abilita' di configurare e lanciare griglie. Nell'esempio di cui sopra abbiamo parlato di 'genitori' e 'figli': la terminologia puo' essere riutilizzata nel caso delle griglie. Un thread che e' parte di una griglia in esecuzione appartiene ad una griglia 'genitore'; la griglia che si crea all'invocazione del nuovo *kernel* annidato sara' detta griglia 'figlio'.

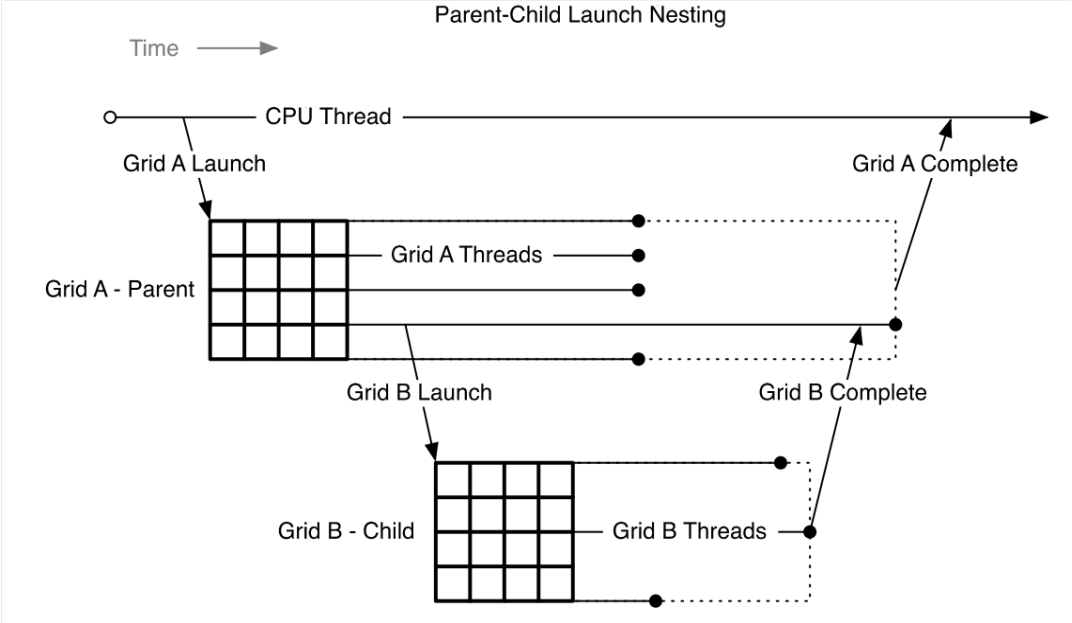

Figura 3. Annidamento delle griglie in CUDA

L'invocazione e il completamento delle griglie e' propriamente annidato, ovvero la griglia 'genitore' non viene considerata completa fino a quando tutte le griglie 'figlio' create dai suoi thread hanno terminato la loro esecuzione. Anche se i therad non richiedono la sincronizzazione esplicita, vi e' a tempo di esecuzione una sincronizzazione *implicita* tra 'genitore' e 'figlio'.

La ricorsione e' stata supportata ed introdotta nelle ultime versioni del SDK (*Software Development Kit)* e quindi un *kernel* puo' chiamare se stesso:

```
__global__ RecursiveKernel(void* data){
    if(continueRecursion == true)
        RecursiveKernel<<<64, 16>>>(data);
}
```

Il *parallelismo dinamico* quindi aiuta notevolmente il lavoro del programmatore nel dover processare i dati per ordinarli e trasferirli; abbiamo tra l'altro gia' visto come la gestione dei dati tra memoria centrale e dispositivo sia il principale collo di bottiglia del lavoro in parallelo. Non sara' piu' necessario dover riadattare e riscrivere tutti quegli algoritmi che richiedevano delle modifiche non banali per rimpiazzare la ricorsione o altri costrutti che non erano stati disegnati su misura: il fparallelismo ora puo' essere espresso in modo piu' trasparente e chiaro e il lavoro del programmatore risulta notevolmente semplificato.

Lo studio dell'architettura non e' stato fine a se stesso. Come detto nell'introduzione, e' stato la scusa per la costruzione di un progetto di piu' ampio respiro, cosi da poter mettere in pratica tutte le nozioni teoriche apprese fino ad ora. Tale progetto, come e' stato anticipato, ha previsto la produzione di un Tetris, ma soprattutto, il *case-study* ha condotto alla realizzazione di un'intelligenza artificiale che si occupa di giocare il Tetris stesso. Il gioco, l'artilligenza artificiale e la successiva implementazione della stessa in CUDA vengono di seguito esposte.

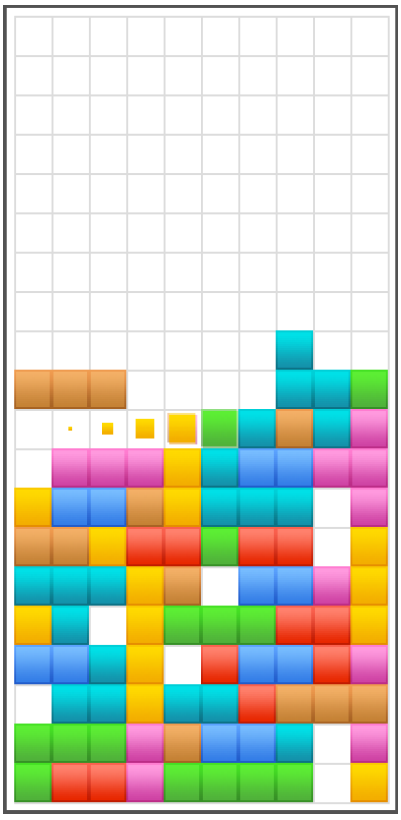

Figura 4. Tetris in azione

Tetris!

Il gioco di Tetris e' piuttosto famoso: in una griglia 10x20 scendono una serie di pezzi che man mano si vanno a poggiare sul fondo. L'obiettivo del gioco e' di far allineare i pezzi sulla griglia cosi da riuscire a completare le linee, senza pero' far riempire la griglia oltre la cima. Quando una linea viene completata (ovvero riempita da pezzi in tutta la sua lunghezza), la si puo' eliminare. L'eliminazione di linee e' quindi il vero e proprio obiettivo del gioco, perche' riuscendo a togliere quante piu' linee possibile si sopravvive piu' a lungo. I pezzi che cadono dall'alto hanno varie forme, e sono state ribattezzate per la somiglianza nella forma con le lettere Z, S, T, O, L, I e J.

Per il sistema di windowing e' stato utilizzato SFML(*Simple and Fast Multimedia Library*), che e' un API in C++ e ha permesso la semplificazione dell'interfacciamento con il sistema operativo, rendendo inoltre il progetto multipiattaforma.

L'implementazione del Tetris comporta dunque il disegno a schermo della griglia, che in memoria sara' una matrice *row-major* di grandezza 200. I calcoli vengono fatti quindi sul riposizionamento dei pezzi, anche essi rappresentati da griglie che avranno un *flag* laddove il pezzo prende parte nel disegno.

Ad esempio, il pezzo a J sara' la matrice (sempre da considerarsi *row-major*):

```
[0, 1,
0, 1,
1, 1]
```

e a seconda delle rotazioni che il pezzo puo' assumere, la posizione dei *flag* sara' cambiata.

Il giocatore puo' interagire con il Tetris premendo le frecce direzionali per far ruotare il pezzo, che proseguira' la sua discesa verso il fondo della griglia, e premendo *space* potra' far cadere il pezzo dall'alto e interagire con il prossimo. Il punto chiave di tale implementazione sta nel fatto che il sistema registrera' le mosse aggiunte in un `std::vector<Piece*>`: tali mosse, seguendo l'ordine First In First Out (FIFO), verranno processate dal Tetris per essere eseguite in modo sequenziale. Tale gestione risulta molto efficace per la successiva implementazione dell'intelligenza artificiale.

Un altro punto cruciale risiede nel fatto che i pezzi del Tetris di per se sono delle *macchine a stati (state machine);* questo implica che a seconda del tasto che viene premuto ad un certo istante, il `TetrisPiece` al momento sulla griglia assumera' uno stato diverso. Ad esempio, tornando al pezzo J di prima:

```
[1, 0, 0,       [1, 1,     [1, 1, 1,
1, 1, 1]        1, 0,            0, 0, 1]
                1, 0]
```

Gli stati diversi che il pezzo puo' assumere sono i seguenti: `MoveLeft`, `MoveRight`, `RotateLeft`, `RotateRight`, `SlamDown`, `SlamLeft`, `SlamRight`. Nuovamente, tale logica risulta molto efficace per l'intelligenza artificiale, che potra' sfruttare l'intera architettura cosi concepita cosi come e', senza richiedere nuove modifiche, ma andando ad agire soltanto su ogni singolo pezzo.

L’Intelligenza Artificiale

Il fulcro vero e proprio di questa tesi di laurea triennale e’ concentrata nel concepimento e costruzione dell’intelligenza artificiale che ‘gioca’ Tetris. Si e’ potuto sfruttare a fondo la struttura precedenemente costruita e, partendo da essa, analizzare il comportamento di un programma che prima viene ideato sull’*host*, e successivamente e’ stato adattato per funzionare sul *device*, ovvero l’adattamento ha richiesto un nuovo approccio algoritmico per la paralellizzazione. In questa parte si analizzera’ l’idea dietro l’intelligenza artificiale, la sua implementazione in C++ in modalita’ iterativa e, nel tentativo di renderla piu’ scaltra, quella ricorsiva.

Per poter concepire un’intelligenza artificiale che potesse giocare Tetris e’ stato necessario tornare indietro per analizzare quali siano le regole che governano il Tetris, in modo da sfruttarle per poter ottenere dei risultati migliori.

Il giocatore puo’ spostare e far rotare i pezzi che stanno cadendo dall’alto verso il basso: l’obiettivo e’ sempre quello di completare le linee per poterle cosi rimuovere dalla griglia e fare spazio per nuovi pezzi; se non si riescono a rimuovere i pezzi prima che raggiungano la cima della griglia si perde. *Quindi come si gioca Tetris?*

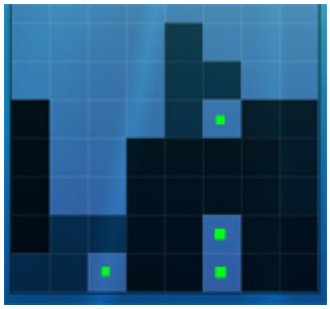

Figura 5. Creazione di buchi nella griglia

L'obiettivo del gioco e' sopravvivere quanto piu' a lungo possibile, quindi il problema che si pone per giocare e' come fare a stare in gioco quanto piu' a lungo possibile. Poiche' perdiamo quando i blocchi raggiungono un'altezza eccessiva, un primo parametro da considerare quando si lascia cadere un pezzo e' quello di penalizzare l'altezza, poiche' un pezzo che e' posizionato piu' in alto aumentera' il rischio di perdere. La strategia complementare e' quella di far abbassare l'altezza eliminando linee. Quindi per ogni linea eliminata facendo cadere un determinato pezzo si otterra' un bonus sul punteggio. Un altro importante parametro da tenere in considerazione e' di rendere la griglia tanto piu' *densa* possibile. Questo vuol dire che vogliamo evitare che ci siano buchi tra le varie linee, altrimenti rischiamo di rendere piu' difficile il completamento delle stesse: il terzo parametro e' dunque la *penlizzazione di buchi*. Sempre in relazione alla densita' della griglia, si e' deciso di includere un quarto ed ultimo parametro per premiare un pezzo che ha le facce a contatto con quanti piu' altri pezzi possibili.

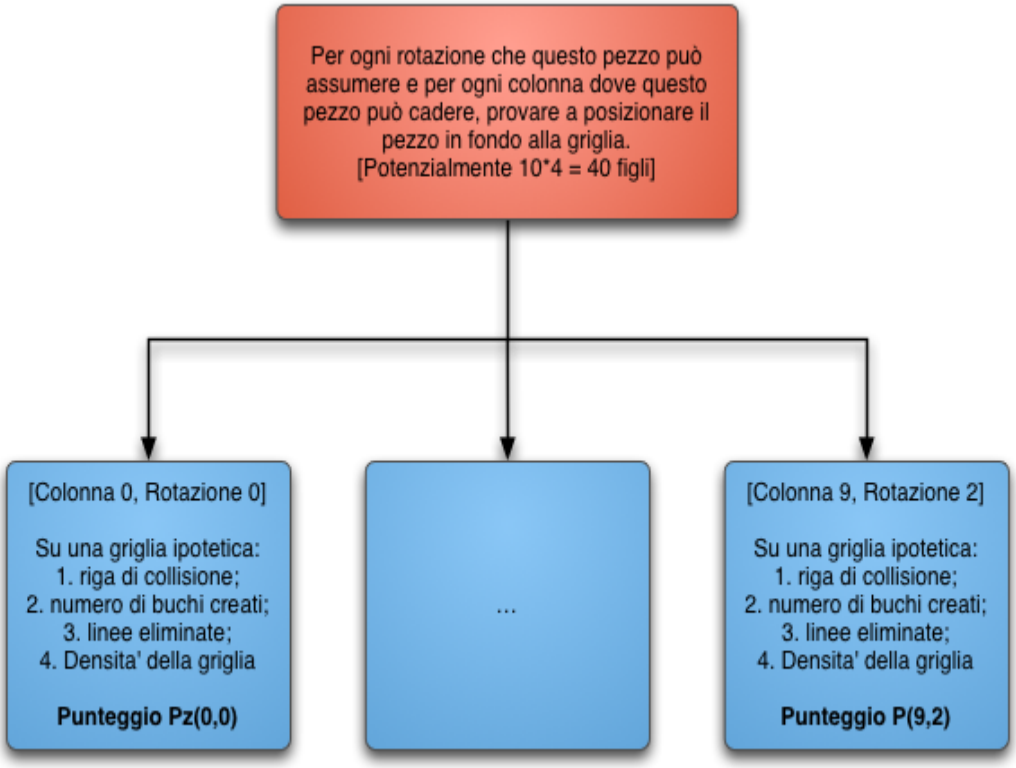

Figura 6. Albero di Computazioni

Sulla base dei principi sopracitati, l'intelligenza artificiale e' stata inizialmente concepita come un albero di computazioni che ci permette di analizzare le mosse possibili sulla griglia di gioco, e procedere con la migliore, a seconda di quali siano i pezzi gia' caduti e di quale sia la situazione della griglia stessa: il problema si e' trasformato in una ricerca su ogni possibile nodo dell'albero.

L'albero sara' composto da nodi ognuno contente i calcoli associati al pezzo in gioco in una determinata rotazione; percio' ad ogni rotazione del pezzo, lo trasleremo sulla griglia, per poi cercare il punto di collisione. A questo punto si potra' generare un punteggio in base ai parametri:

- riga di collisione;
- numero di buchi creati;
- linee eliminate;
- densita' della griglia.

Poiche' un pezzo puo' potenzialmente trovarsi su n colonne diverse e assumere m possibili rotazioni, il costo dell'algoritmo sara' O($n * m$), dove n e m rispettivamente $1 \leq n \leq 10$e $1 \leq m \leq 4$, per un totale di 40 possibili combinazioni.

Alla fine di questa prima parte di progetto, il problema e' stato risolto in modo semplice con una ricerca a partire dalla radice in tutti i figli. I risultati sono stati discreti, ottenendo un buon metodo di gioco, coerente ed efficace.

Per rendere il tutto *piu' intelligente,* si e' deciso di far conoscere alla CPU i prossimi pezzi che sarebbero arrivati: viene passato come parametro un `std::vector<Pieces*>` nel quale sono inclusi i prossimi k pezzi in arrivo. In questo modo non solo si puo' calcolare la migliore posizione in base al prossimo pezzo in gioco, ma la migliore combinazione delle prossime k mosse: piu' pezzi l'IA avra' a dispozione, migliore sara' il risultato. Stavolta l'albero si complica, e per ogni possibile rotazione e traslazione vi saranno altri possibili O($n * m$) figli. Il totale risulta essere un albero con O($n^2 * m^2$) nodi, ovviamente il risultato quadratico e' indice di un algoritmo poco efficiente, per quanto efficace nel giocare.

Lo studio di questa intelligenza artificiale si basa su un recente algoritmo di tipo *best-first*, che si chiama **Monte-Carlo Tree Search** (MCTS). Le implicazioni che la scelta di questa algoritmo hanno avuto nella risoluzione del problema sono interessanti, sia perche' ci danno una buona logica per risolvere il problema in maniera ricorsiva, sia perche' MCTS e' un algoritmo che risulta facilmente parallelizzabile: non poteva esserci scelta migliore per il nostro *case-study*.

Le ricerche *best-first* sono un tipo di ricerca sugli alberi di ricerica che sfruttano la conoscenza dei diversi stati del problema da risolvere. Ogni nodo del problema rappresenta uno stato ben preciso del problema e i suoi figli rappresentano lo "stato obiettivo"; la radice, ovviamente, e' lo "stato iniziale". Ogni possibile soluzione del problema la sia ha quando il cammino giunge al termine, ovvero quando si e' arrivati a visitare le foglie tramite

l'esplorazione di un determinato ramo. La strategia *best-first* implementa un'apposita funzione $f(n)$che ha il compito di selezionare il prossimo nodo da espandere ad ogni passo di ricerca; l'algoritmo sceglie, ad ogni passo, tutti i nodi possibili da espandere, e sceglie il nodo con la valutazione *piu' alta (o piu' bassa* se la funzione dev'essere minimizzata), da cui *best-first*. Una funzione di questo tipo viene detta eurisitica e ha il compito di selezionare, di volta in volta, il nodo che sembra condurra alla soluzione ottimale del problema.

Monte-Carlo Tree Search utilizza i risultati delle precedenti esplorazioni; l'algoritmo costruisce gradualmente un albero in memoria, e diventa quindi piu' preciso nella stima dei valori delle mosse con l'evolversi dei calcoli. MCTS e' diviso in quattro fasi:

1. nel passo di *selezione* l'albero viene attraversato dal nodo radice fino alla foglia;
2. la *strategia di espansione* viene usata per raggiungere una foglia dell'albero;
3. la *strategia di simulazione* mette in atto le mosse disponibili fino a quando non si e' raggiunta la fine del gioco;
4. la *strategia di retropropagazione* restituisce il risultato migliore.

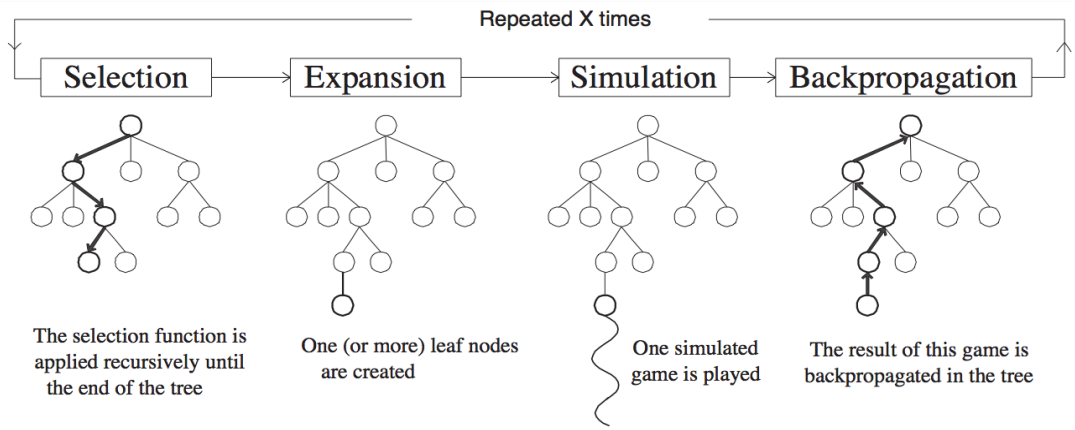

Figura 7. Schema del Monte-Carlo Tree Search

Quando termina il tempo a disposizione dell'algoritmo, la mossa che viene messa in atto e' il figlio della radice che e' stato visitato piu' volte; ovviamente, piu' a lungo il programma viene eseguito, migliore sara' il risultato.

Come e' stato anticipato sopra, l'implementazione concreta dell'algoritmo e' ricorsiva e procedera' in questo modo: ad ogni livello dell'albero si calcoleranno tutti i punteggi alle diverse traslazioni associate alla griglia *ipotetica* di quel livello. Il miglior possibile risultato viene riportato nel nodo che sta costruendo il risultato, e dopo aver calcolato il risultato corrente potra':

- continuare la computazione, se la profondita' del nodo e' piu' piccola della dimensione del vettore;
- terminare la computazione e tornare sul ramo che ha garantito il risultato corrente. Conseguemente costruire il punteggio dalla foglia fino alla radice: il miglior risultato ad ogni livello sara' il miglior risultato al livello corrente + il miglior risultato su tutti i figli.

L'albero sara' di profondita' k e garantira' il miglior risultato su tutti i livelli.

CUDA: le difficolta' della Programmazione in Parallelo

Ora passiamo al punto cruciale del nostro *case-study*, ovvero lo studio di un algoritmo parallelo e nuovamente la nostra analisi parte dalla ricerca sugli alberi Monte-Carlo (MCTS) e i possibili tipi di parallelizzazione che si possono adottare. Si presuppone che il programma sia eseguito su un multiprocessore asimmetrico (*Symmetric Multiprocessor - SMP*) e che ad ogni thread in esecuzione venga associato un singolo thread. Il vantaggio degli SMP e' che ogni thread puo' accedere ad una sezione di memoria condivisa: nuovamente lo studio di MCTS risulta efficace perche' cio' avviene anche in CUDA, dato che ogni thread puo' accedere alla memoria condivisa di blocco e a quella globale (v. figura 2). Nel caso di MTCS vi sono tre tipi diversi di parallelizzazione, a seconda di quale sia la fase in cui avviene la parallelizzazione stessa: alle foglie, alla radice e all'albero.

Nella parallelizzazione alle **foglie**, la fase 1 e 2 sono portate avanti da un solo thread che attraversera' l'albero fino a quando non raggiunge una foglia. Arrivato alla foglia, entra in atto la fase 3, per cui il gioco viene simulato da tutti i possibili thread. Quando tutti i risultati sono stati ottenuti, vengono retropropagati da un thread verso la radice (fase 4). L'implementazione di questa versione risulta efficace e sufficientemente semplice, non richiedendo l'implementazione di mutex o semafori per gli accessi alle sezioni critiche.

Nella parallelizzazione alla **radice**, si costruiscono svariati MCTS in parallelo, con un thread su ogni albero e, come nel caso precedente, i thread non condividono informazioni. Quando il tempo a disposizione termina, tutti i figli della radice dei vari alberi MCTS vengono fusi con i loro cloni corrispondenti: per ogni gruppo di cloni, tutti i punteggi vengono sommati: la mossa migliore viene selezionata sul totale appena calcolato. Nuovamente, la parallelizzazione di questo algoritmo risulta facile e richiede una comunicazione minima fra i vari thread.

L'ultimo tipo di parallelizzazione studiata nel caso di MCTS e' quella **all'albero**. Viene utilizzato un unico albero condiviso su cui i numerosi thread possono simulare la loro partita. Poiche' ogni thread ha accesso alla struttura dati, e' necessario l'utilizzo di *mutex* per bloccare l'accesso ad alcune parti dell'albero cosi da prevenire la corruzione dei dati. Si puo' quindi decidere di bloccare l'intero albero con un *mutex globale*, o di bloccare singole parti interessate con dei *mutex locali*. Nel primo caso ogni partita viene simulata a partire dalla stessa foglia. Il maggiore svantaggio e' il limite dovuto al tempo che ogni thread deve spendere all'interno dell'albero bloccato. Per quanto la parallelizzazione risulti efficace, i thread rimarranno in attesa per lungo tempo. Il secondo metodo utilizza dei *mutex* ai nodi: ogni thread che visita un nodo blocchera' il nodo in questione, sbloccandolo quando la visita e' terminata. Se vari thread iniziano la ricerca dalla radice allo stesso tempo e' possibile che attraverseranno l'albero per larga parte nello stesso modo. Le partite che vengono simulate partono da foglie che potrebbero essere fra loro vicine, ma puo' anche accadere che vengano simulate partite a partire dalla stessa foglia. Per evitare che vegano effettuate simulazioni uguali, si decide di assegnare una "perdita virtuale" quando un nodo viene visitato da un thread, ovvero il valore di questo nodo viene decrementato; in questo modo il prossimo thread selezionera' lo stesso percorso del precedente soltanto se il valore al nodo rimane il migliore rispetto a tutti i suoi fratelli. In questo modo, i nodi che sono decisamente migliori di altri verranno esplorati comunque da tutti i thread, mentre thread con valori incerti verranno analizzati da un numero piu'

ristretto di thread: in questo modo si mantiene un buon equilibrio tra l'esplorazione e lo sfruttamento della parallelizzazione in MCTS.

Questo studio ha portato all'elaborazione di un nuovo metodo di parallelizzazione, anche per via dell'architettura del progetto del Tetris stesso. CUDA ci permette di assegnare il numero di threads che devono essere lanciati ogniqualvolta un *kernel* e' chiamato e, nella sua piu' recente versione - CUDA 5, con l'introduzione del parallelismo dinamico, abbiamo visto come sia possibile creare kernel ricorsivi con griglie innestate. La cosa piu' logica, quindi, era implementare una ricerca in parallelo del risultato migliore per ogni pezzo, su ogni singola colonna e per ogni sua possibile rotazione. Lo *speedup* sicuramente sara' notevole: invece di effettuare $O(n^{2}*m^{2})$ operazioni annestate, saranno effettuate tutte in parallelo, ovvero si avranno O(k) chiamate ricorsive, una per ognuno dei prossimi k pezzi in arrivo sulla griglia di gioco.

Il risultato e' davvero notevole. Si e' sfruttata l'idea della parallelizzazione **alle foglie** vista nel caso del MCTS, in cui pero' e' associato un thread ad ogni foglia dell'albero, andando a generare l'intero albero delle computazioni fin dalla radice, rappresentata dal pezzo correntemente in gioco. L'altezza dell'albero sara' k.

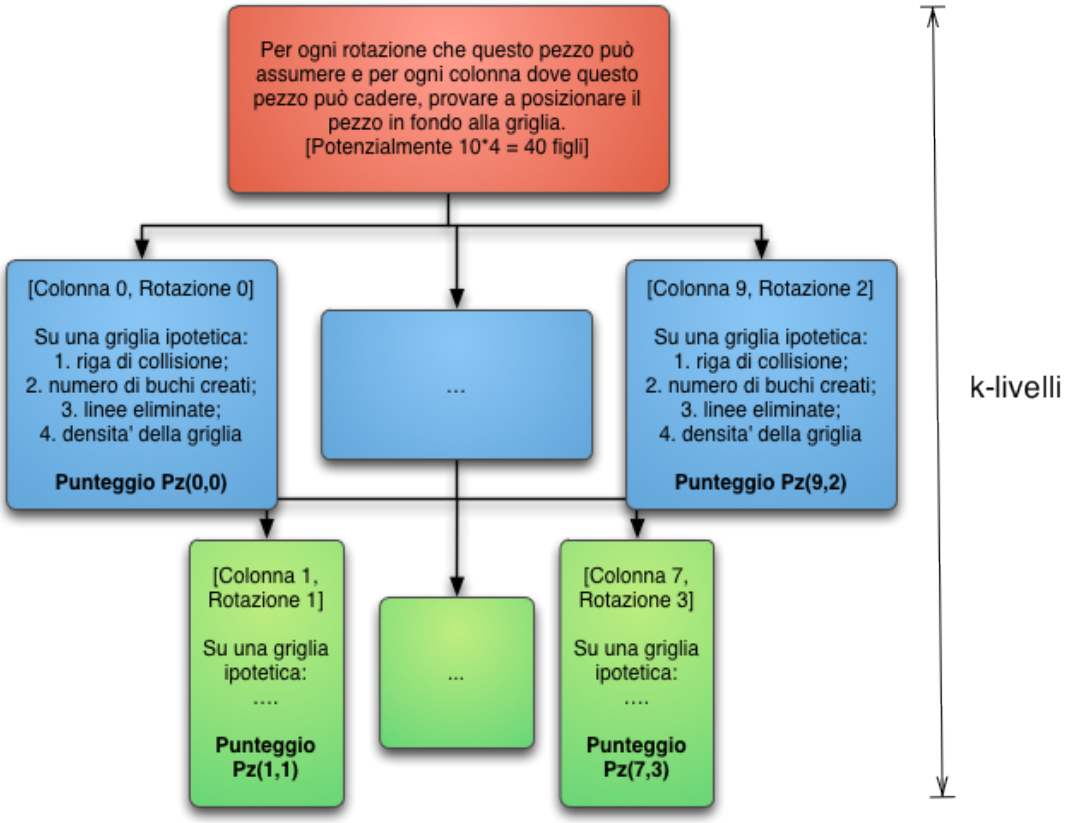

Figura 8. Albero di computazione ricorsivo a k livelli

Tutto questo lavoro di parallelizzazione ha richiesto uno sforzo sotto svariati aspetti. Il primo e il piu' importante e' stato l'incapsulamente dei dati nel modo corretto: nell'analisi dell'architettura creata da NVIDIA il collo di bottiglia e' l'accesso alla memoria centrale. Se i dati vengono organizzati nel modo opportuno e resi disponibili al *device* sulla sua stessa memoria, ovviamente il programma potra' usufruire di questo vantaggio e sfruttare al massimo la sua potenza di calcolo. Quindi prima di tutto le strutture dati del Tetris quali la griglia, i pezzi, i vettori, sono state allocate tramite tramite la `cudaMalloc()` sul device:

```
char *deviceGrid;
cudaMalloc(&deviceGrid, gridRows*gridColumns);
cudaMemcpy(deviceGrid,
(void*)grid,gridRows*gridColumns,
cudaMemcpyHostToDevice);
```

successivamente si e' utilizzata la chiamata al kernel che ha permesso l'inizio della ricerca in parallelo:

```
parallelBest <<<1, numThreads>>> (deviceGrid, gridRows, gridColumns, devicePiece, kernel_Array, 0, deviceResult);
```

si e' deciso di utilizzare una griglia composta soltanto da 1 blocco, poiche' ogni blocco, sulla scheda video utilizzata, e' in grado di gestire fino a 1024 threads, cosa che ovviamente non sara' mai verificata, visto che per ogni blocco sara' possibile avere al massimo 40 diverse combinazioni di posizione e rotazione. Il codice all'interno del kernel ovviamente varia la logica rispetto a quello gestito dalla CPU, perche' in questo caso invece di gestire i diversi casi, il kernel si occupera' di gestirne uno solo e delegare la scelta del miglior risultato, occupandosi soltanto del calcolo di quello corrente. Quindi il programmatore si trova a dover ragionare in modo diverso: invece di gestire le diverse casistiche, bastera' svolgere in modo corretto l'unica che viene considerata.

Un altro degli accorgimenti necessari per la gestione del kernel e' stato quello di creare una `struct KernelArray`, che potesse sostuire il vettore utilizzato nella versione della CPU:

```
struct KernelArray
{
    T* array;
    int size;
};
```

questo perche' CUDA non e' in grado di gestire le classi della *C++ Standard Library*, e ovviamente cio' richiede generalmente un nuovo sforzo da parte del programmatore nell'adattare quelle strutture dati che solitamente sono a sua disposizione.

Un altro problema legato alla gestione di CUDA all'interno di un progetto molto piu' ampio e' quello di utilizzare le chiamate necessarie a CUDA per gestire all'interno del kernel, le chiamate a funzione esterne. La struttura impone che tutte quelle funzioni che devono essere utilizzate all'interno di un qualasiasi kernel, debbano essere definite con il qualificatore `__device__`. Ovviamente g++ non riconosce tale qualificatore, e di conseguenza se una funzione vuole essere utilizzata sia in un file .cu, che in un un file .cpp non e' possibile senza incorrere in qualche tipo di errore, da una parte e dall'altra. Qui le macro di C sono venute in aiuto, perche' e' stato sufficiente definire:

```
#ifdef __CUDACC__
#define CUDA_CALLABLE_MEMBER __host__ __device__
#else
#define CUDA_CALLABLE_MEMBER
#endif
```

__CUDACC__ e' la macro inclusa da NVIDIA per definire nvcc, il suo compilatore. In questo modo il linking del codice con i qualificatori corretti e' avvenuto solo quando era necessario, e quindi il codice e' cosi funzionale e facilmente portabile.

Infine per garantire una corretta gestione del risultato finale migliore calcolato dal *kernel* e' stata utilizzata la direttiva di cuda per la sincronizzazione dei threads all'interno di un blocco.

Conclusioni

Affrontare CUDA ha significato rivedere il modo di incapsulare le informazioni per renderle piu' rapidamente accessibili al *device*, comprendere una nuova struttura hardware e il modo con cui essa deve lavorare in congiunzione con CPU e memoria, nel solito tentativo di evitare le richieste in IO, che da sempre rappresentano un collo di bottiglia prestazionale, iniziare a ragionare non seguendo piu' un unico filo di istruzioni, ma comprendendo che tali istruzioni, accessibili ed eseguibili contemporaneamente, richiedono una gestione accurata degli accessi a zone di memoria condivisa.

Arrivati al termine di questo lungo e interessante progetto, possiamo trarre le nostre conclusioni: la legge di Moore sostiene che la densita' dei transistor raddoppia all'incirca ogni anno e mezzo e comunemente la si associa al raddoppiamento della velocita' di clock. Questo raddoppiamento ha raggiunto un limite fisico importante: scavalcandolo in modo elegante Intel e AMD hanno deciso di incrementare la potenza di calcolo tramite l'inserimento di processori multipli su un unico chip. Questo trend di mercato rispecchia anche una necessita' da parte di chi crea il software di riuscire a sfruttare le nuove architetture al massimo. Il presupposto di questa intera tesi di laurea triennale e' proprio il bisogno da parte di un ingegnere che sta ancora formandosi di rapportarsi a queste nuove architetture avvalendosi di nuovi strumenti. La logica utilizzata da sempre nella programmazione risulta un importantissima base da cui partire, ma e' ovvio che non sara' piu' sufficiente a soddisfare le esigenze dei ricercatori, dei programmatori ma anche del piu' semplice consumatore. Il mondo

accademico quindi, deve anticipare il cambiamento, e saper cogliere questi primi segnali da parte dei produttori di hardware e software, per preparare i propri studenti ad un mondo e una mentalita' diversi da quelli che abbiamo di fronte oggi.

Bibliografia:

- Parallel Monte-Carlo Tree Search on - http://www.personeel.unimaas.nl/m-winands/documents/multithreadedMCTS2.pdf
- Coding a Tetris AI using a Genetic Algorithm on - http://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/
- Tetris AI - http://www.colinfahey.com/tetris/tetris.html
- Understanding the CUDA Data Parallel Threading Model on - http://www.pgroup.com/lit/articles/insider/v2n1a5.htm
- CUDA Tutorials on http://llpanorama.wordpress.com/cuda-tutorial/
- Parallel Programming Tutorials on - http://www.drdobbs.com/parallel/
- NVIDIA SDK: CUDA_C_Programming_Guide.pdf - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- NVIDIA SDK: CUDA_Dynamic_Parallelism_Programming_Guide.pdf - http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf
- NVIDIA SDK: CUDA_Compiler_Driver_NVCC.pdf
- CUDA on - http://en.wikipedia.org/wiki/CUDA
- SFML on - www.sfml-dev.org/