HPC

Architetture parallele

Nei sistemi di calcolo moderni l’accelerazione (speedup) si ottiene distribuendo il carico computazionale su di una architettura hardware in grado di sfruttare le performance dei diversi livelli di parallelismo su:

- Diversi thread o processi su un singolo nodo (memoria condivisa)
- Diversi processi su più nodi interconnessi (memoria distribuita)
- Kernel di calcolo eseguiti su acceleratori (GPU)

Programmi paralleli

Con Speedup si intende la misura dell’ accelerazione del tempo di calcolo rispetto al programma non parallelizzato.

Speedup = Tseriale / Tparallelo

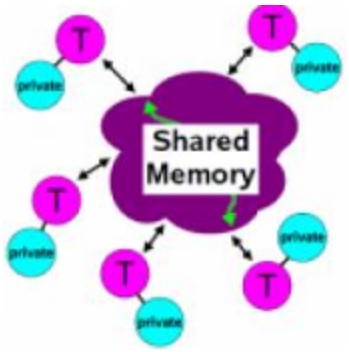
Questo numero, nel caso ideale coincide con il numero dei processori.

Con Scalabilità si intende la capacità del programma parallelo di mantenere una crescita proporzionata dallo Speedup al crescere delle unità di processamento.

Con Overhead si intendono le limitazioni della scalabilità introdotte dalla parallelizzazione, come la comunicazione e il sincronismo tra i Task.

Programmazione a memoria condivisa

Un architettura a memoria condivisa è realizzata con i sistemi SMP (Symmetric Multi Processor) in cui un pool di processori omogenei operano in modo indipendente ma condividono lo spazio di indirizzamento della memoria RAM.



I task del programma possono essere assegnati a thread in esecuzione su differenti unità di processamento.

Programmazione a memoria distribuita

Una architettura a memoria distribuita è costituita da un cluster di computer interconnessi da una rete di comunicazione.

I task (processi) sono eseguiti sulle CPU del cluster e hanno il proprio spazio di indirizzamento.

Programmazione GPU

La GPU è nata come coprocessore per il rendering di immagini su un dispositivo di visualizzazione, ma grazie alla sua programmabilità, si è iniziato a utilizzarla come coprocessore della CPU.

La programmazione consiste nella creazione di un kernel di calcolo che verrà tipicamente avviato insieme ai dati dal risultato dell'host.

Performance

La performance è misurata nel tempo impiegato per terminare il lavoro e in base al consumo di risorse computazionali e di energia.

Le risorse hardware che incidono sulle performance sono le unità di processamento, la memoria, lo storage e la comunicazione in rete; le risorse software che incidono sulle prestazioni sono gli algoritmi, l’organizzazione dai dati dell’ applicazione, l’hardware exploitation (cioè la capacità di sfruttare le risorse hardware disponibili) e le caratteristiche del software utilizzato.

- **Theoretical Peak Performance** è una stima della performance di un componente hardware
- **Sustained Performance** sono le prestazioni effettive misurate di un componente hardware, ottenute tramite l'esecuzione di specifici programmi detti Benchmark.

CPU

La performa di una CPU è data dalle operazioni svolte in un unità di tempo, prima si misuravano in MIPS (milioni di istruzioni per secondo) e adesso si valutano in operazioni in virgola mobile come gli MFLOPS o i GFLOPS.

Le prestazioni di un core di calcolo (operazioni in virgola mobile al secondo, **FLOPS**) sono determinate dal numero di cicli di **Clock** per secondo e dal numero di operazioni f.p. per ciclo di clock (**FLOPs** / **cycle**) che il core può eseguire: **FLOPS = Clock * FLOPs / cycle**

Il numero di operazioni per ciclo dipende anche dalla precisione del dato in virgola mobile che può essere:

- Singola (SP, 32bit)
- Doppia (DP, 64bit)
- Mezza (HP, 16bit)

Istruzioni vettoriali (SIMD) e FMA

Le applicazioni con elevato carico computazionale sono **Data Parallel**, quindi, richiedono l'esecuzione di **SIMD** (Single Instruction Multiple Data).

Dato che le operazioni sono spesso delle moltiplicazioni vettoriali, sono state aggiunte delle istruzioni dedicate chiamate **FMA** (Fused Multiply-Add) che in un solo ciclo di Clock riescono a eseguire 2 operazioni, ovvero la somma e la moltiplicazione.

Memoria

La gerarchia della memoria determina tempi di accesso differenti a seconda della localizzazione del dato. Nel caso in cui la memoria non riuscisse a fornire dati con il ritmo richiesto dal processore, può diventare un *bottle neck*, portando alla riduzione delle prestazioni.

RoofLine Analysis

Il modello Roofline consente in maniera intuitiva di stimare le performance di un kernel computazionale (un kernel è il nucleo del sistema operativo, gestisce funzioni di controllo fondamentali del computer) mostrando graficamente le limitazioni inerenti CPU/GPU e memoria.

FLOP/s = min (peak FLOP/s, Peak Memory bandwidth x Aritmetic Intensity)

Dove Arithmetic Intensity = Total FLOPS/Total Bytes x Aritmetic Intensity)

Storage

Nei sistemi HPC ci sono più tipi di storage (online, nearline, offline) a cui corrispondono dispositivi con caratteristiche diverse (riguardo prestazioni, capacità e costi)

NAS (Network Attached Storage)

Un NAS è un dispositivo collegato alla rete la cui funzione è quella di consentire agli utenti di accedere e condividere una memoria di massa, in pratica costituita da uno o più dichi rigidi, questi ultimi, possono essere sia interni che esterni alla rete.

Vantaggi: condivisione dati, gestione centralizzata, basso costo

Svantaggi: basse prestazioni, risorse limitate

SAN (Storage Area Network)

E’ una rete ad alta velocità di trasmissione costituita esclusivamente da dispositivi di memorizzazione di massa, che possono essere anche di tipo differente.

Scopo: fornire le risorse da lui memorizzate a qualsiasi computer a lui connesso.

Vantaggi: condivisine dati, gestione centralizzata, basso costo.

Svantaggi: alte prestazione, scalabilità, ridondanza.

Clustered File System: GPFS

GPFS è un file system condiviso in rete, costituito da un insieme di dichi su cui GPFS memorizza dati e metadati.

▼ Funzioni

- failover: in caso di server failure, il client viene servito da un altro NSD server.
- shared disk: tutti i dischi vengono utilizzati contemporaneamente da tutti i nodi.

- byte range locking: accesso simultaneo di più utenti allo stesso file.
- stripe: il singolo file viene suddiviso in blocchi che vengono collocati su tutti i dischi del file system.
- tiering: permette di definire gerarchie di storage

Benchmarks

Con il termine benchmark si intende un insieme di test software che forniscono una stima delle prestazioni reali di un computer, in base a diverse operazioni eseguibili.

Tempi di esecuzione

time: è un comando e ritorna i tempi di esecuzione di un programma

- real: tempo reale di esecuzione
- user: tempo di utilizzo della CPU nello stato User
- sys: tempo di utilizzo della CPU nello stato Kernel

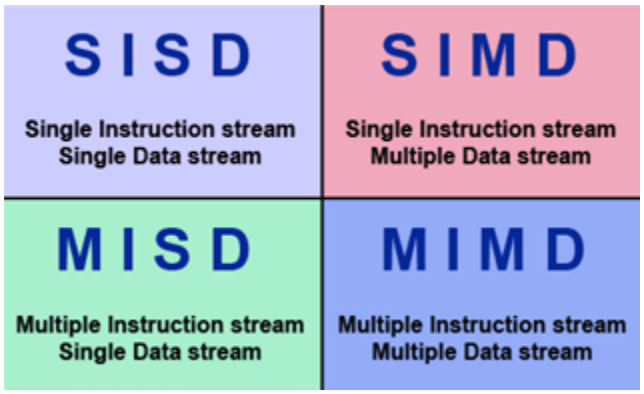
gprof: è un profiler che permette di determinare quali parti del programma consumano più tempo, per eseguirlo e ottenere un file contente queste informazioni bisogna ,compilare un programma con l'opzione `-pg`

Al momento dell' esecuzione verrà generato un file gmon.out che potrà essere analizzato con il comando gprof.

clock_gettime(): è una funzione e consente di determinare i tempi di esecuzione all' interno di un programma, come il wall clock time o il tempo di utilizzo della CPU.

Tassonomia di Flynn

La tassonomia di Flynn permette di classificare i sistemi di calcolo parallelo, in base al flusso di istruzioni e di dati che possono gestire, che possono essere Single o Multiple



▼ SISD

Sono dei sistemi seriali, che permettono di gestire un singolo flusso di istruzioni eseguito dalla CPU e può essere elaborato un singolo flusso di dati.

▼ SIMD

Sono dei sistemi dotati di una sola unità di controllo, la CPU può eseguire un singolo flusso di istruzioni e possono essere elaborati più dati contemporaneamente, quindi ogni singola istruzione opera simultaneamente su più dati.

- PROCESSORI VETTORIALI : sono degli array di elementi di elaborazione che hanno accesso (condiviso) all'unità di controllo

  Ci sono diverse Processing Unit e le istruzioni sono distribuite in parallelo a tutte le Processing Unit; per poter trasmettere i dati ed elaborarli è necessaria una rete di comunicazione per i dati.

- ISTRUZIONI VETTORIALI : sono delle istruzioni che specificano una particolare operazione che deve essere eseguita su un insieme di operandi detto vettore, quindi, la memoria è condivisa e il parallelismo è realizzato all' interno del processore.

- Le unità funzionali che eseguono istruzioni vettoriali sfruttano il pipelining per eseguire la stessa operazione su tutte le coppie di operandi.

▼ MIMD

Ogni processore può eseguire un differente flusso di istruzioni e ogni flusso di istruzioni lavora su un differente flusso di dati.

Sistemi a memoria condivisa UMA (Uniform Memory Access)

In questi sistemi tutti i processori accedono alla memoria come spazio di indirizzamento globale e le modifiche alla memoria effettuate da un processore sono visibili da tutti gli altri processori.

I tempi di accesso alla memoria sono uniformi e dato che la memoria è condivisa i processori sono chiamati tightly coupled systems per l'alto grado della condivisione delle risorse.

Sistemi a memoria condivisa NUMA

In questi sistemi, ogni processore ha una propria memoria locale, e l'insieme delle memorie locali forma uno spazio degli indirizzi globale, accessibile da tutti i processori.

I tempi di accesso alla memoria non sono uniformi, poiché l'accesso è più veloce se il processore accede alla propria memoria locale, ma quando si accede alla memoria dei processori remoti si ha un ritardo dovuto alla rete interna di interconnessione (bisogna attraversa un BUS per raggiungere l'altra memoria)

Sistemi a memoria distribuita

Ogni processore possiede una propria memoria locale, che non fa parte dello spazio di indirizzamento degli altri processori. Ogni sistema CPU/Memoria è detto nodo e agisce in modo indipendente.

Vantaggi: il numero di processori e la memoria complessiva scalano con il numero di nodi, i costi sono contenuti

Svantaggi: il programmatore deve gestire i dettagli della comunicazione tra i nodi, il tempo per l'accesso alla memoria remota dipende dal tipo di infrastruttura di rete

Sistemi ibridi

Ad oggi i sistemi sono ibridi, quindi composti da più nodi a memoria condivisa e interconnessi da una rete; inoltre, i nodi possono disporre di acceleratori basati su GPU che dispongono a loro volta di un proprio spazio di memoria e comunicano con altri nodi attraverso i BUS di sistema.

Parallel Computing

La computazione parallela è una tecnica di programmazione che permette di dividere il carico di lavoro in Task diminuendo il carico computazionale complessivo. I Task verranno poi eseguiti su diversi sistemi di parallelismo.

La divisione del carico di lavoro può essere a livello di domini o di funzioni:

- **Decomposizione di dominio** (data parallelism)

  Si utilizza quando è necessario elaborare un data set di grandi dimensioni con dati strutturati che vengono suddivisi in sottodomini di dimensioni più piccole. Ogni sottodominio viene elaborato da un task specifico.

- **Decomposizione funzionale** (task parallelism)

  Si utilizza quando si vuole dividere il carico di lavoro in base al lavoro che deve essere svolto, quindi si avranno elaborazioni indipendenti e diverse, per cui ogni Task prenderà in carico un lavoro specifico.

  Vantaggi: scalabile con un numero di elaborazioni indipendenti.

  Svantaggi: è utile solo quando vanno eseguite elaborazioni sufficientemente complesse.

Comunicazione tra i Task

Le comunicazioni tra i Task possono essere: punto-punto o collettive

I Task possono avere la necessità di comunicare in base ai diversi tipi di problema, nel caso in cui il tipo di problema non necessitasse comunicazione tra i Task allora si definirebbe “embarassingly parallel”

La comunicazione tra i Task ha un costo, ed ogni messaggi inviato richiede un tempo che influisce sulle prestazioni.

- Per le comunicazioni punto-punto, T_invio_mess = T_latenza + *numero di byte del messaggio* (M) / *larghezza della banda* (Band)
- Per le comunicazioni collettive: T_collettivo = T_invio_mess * *numero di Task* (P) - 1

Avendo *n* processi e usando la strategia divide-et-impera è possibile distribuire i dati in input ai Task in $log_2(p)$ passi, rispetto che in $p-1$

I Task possono interagire tra loro anche per dipendenze tra sezioni di codice o dati, e quindi eseguire il loro compito solo quando avvengono determinate azioni

In un programma data-parallel tutti i Task iterano la stessa funzione su dati diversi e possono procedere all' iterazione N+1 solo quando tutti hanno completato l'iterazione N.

Per permette la **sincronizzazione tra i Task** si utilizzano 2 strumenti:

- barrier: implicano il coinvolgimento di tutti i Task
- lock e semafori: possono coinvolgere qualsiasi numero di task

Se il sincronismo non viene gestito adeguatamente, può portare al degrado delle performance.

Bilanciamento del carico

Il load balancing è una tecnica di progetto con l'obiettivo di di distribuire il lavoro in modo da minimizzare i **tempi di Idle** (tempo di inattività) dei processi.

Una tecnica comune per bilanciare il carico è il modello master slave (detto a volte manager-worker), in cui un task (master) ha il compito di suddividere il lavoro in piccoli task e gestire lo scheduling dinamico dei task, assegnandoli verso un pool di Slaves.

Granularità

La decomposizione del problema può avvenire con diverse granularità:

- Parallelismo a grana fine (fine grain)
  - Utile per bilanciare il carico e diminuire l'overhead di sincronismo
  - Potrebbe portare ad un aumento delle comunicazioni e del conseguente overhead
- Parallelismo a grana grossa (coarse grain)
  - Migliora il rapporto tra calcolo e le comunicazioni
  - Può essere difficile bilanciare il carico

Performance

Con il termine SpeedUp si intende la misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato: Speedup = T_seriale / T_parallelo. L'efficienza (E) è il rapporto tra E = Speedup /P e nel caso ideale vale 1.

Scalabilità

Scalabilità : è la capacità del programma parallelo di mantenere una crescita proporzionata dello Speedup al crescere delle unità di processamento

- **Strong Scaling** : misura l'efficienza dell'applicazione al crescere del numero di processori mantenendo fissa la dimensione complessiva del problema.
- **Weak Scaling** : misura l'efficienza dell'applicazione al crescere del numero di processori mantenendo fissa la dimensione del problema assegnata ad ogni processore.
- **Overhead** : le limitazioni della scalabilità sono dovute a Overhead introdotti dalla parallelizzazione.

Programma Parallelo

E' un programma in grado di suddividere l'algoritmo in diversi Task distribuiti sulle diverse unità di processamento e coordinati tra loro per realizzare un obiettivo computazionale complessivo.

L'esecuzione di un programma parallelo richiede:

- un calcolatore parallelo
- un linguaggio di programmazione che permetta di descrivere esplicitamente algoritmi paralleli **(parallelismo esplicito)**
- un compilatore in grado di parallelizzare automaticamente parti del programma sequenziali **(parallelismo implicito)**

Parallelismo automatico, guidato e manuale

- Automatico : il compilatore analizza il sorgente ed individua possibili parallelizzazioni.
- Guidato dal programmatore : attraverso direttive il programmatore può dare indicazioni sulle parti di codice da parallelizzare e come farlo (esempio openMP).
- Esplicita : occorre individuare manualmente i task e programmare esplicitamente l'interazione tra i task (esempio MPI).

Programmazione delle istruzioni SIMD

La vettorizzazione è una tecnica che consente di effettuare in parallelo la stessa operazione su tutti gli elementi di un vettore. Viene realizzata mediante architetture SIMD (Single Instruction Multiple Data). La programmazione vettoriale può essere: automatica, guidata dal programmatore, o esplicita.

Programmazione parallela MIMD

I processori di un calcolatore parallelo comunicano tra loro secondo 2 schemi di comunicazione:

- Shared memory : I processori comunicano accedendo a variabili condivise
- Message passing : I processori comunicano scambiandosi messaggi

Questi schemi di comunicazione usano rispettivamente un paradigma:

- Paradigma a memoria condivisa : i processori interagiscono esclusivamente operando su risorse comuni
- Paradigma a memoria locale : non esistono risorse comuni, i processi gestiscono solo informazioni locali e l'unica modalità di interazione è costituita dallo scambio di messaggi

Memoria condivisa (Shared Memory)

Quando si ha una memoria condivisa i task comunicano accedendo a variabili e strutture dati condivise, inoltre, per garantire parallelismo, sono necessari strumenti che permettano la sincronizzazione delle operazioni.

- PROCESSI : è possibile creare sezioni di memoria condivisa, è possibile la sincronizzazione con i semafori.
- THREAD : è possibile la comunicazione a memoria condivisa tra più thread, è possibile la sincronizzazione con i semafori
- OpenMP: Corrisponde a una sezione di codice che si intende eseguire in parallelo e viene marcata attraverso una direttiva che crea dei thread prima della loro esecuzione in parallelo.

SPMD (Single Program Multiple Data)

E’ un modello di programmazione in cui tutti i task eseguono la stessa copia del programma simultaneamente, elaborando dati diversi, infatti esso è un modello tipico di

programma a memoria condivisa, ovvero un unico programma con diversi thread che lavorano su dati diversi.

Il modello SPMD è usato anche nella programmazione MPI: mpirun è il comando MPI che mette in esecuzione n istanze dello stesso programma.

`mpirun -np 4 a.out` → process number 4

Memoria distribuita (Message Passing)

Quando si ha una memoria distribuita i processi comunicano scambiandosi messaggi, questo avviene grazie alla libreria MPI (Message Passing Interface) che fornisce funzioni per comunicazioni punto-punto e comunicazioni collettive.

Master-Slave

Un task Master controlla il lavoro svolto dagli altri task (slaves), il modello più adatto per creare un programma Master Slave è l' SPMD.

Per creare un programma Master-Slave sono necessarie 2 operazioni fondamentali:

- Fork: viene eseguita dal task Master e genera dinamicamente uno o più nuovi task che eseguono un nuovo flusso di controllo parallelamente al master
- Join: viene eseguita da tutti (o parte) dei task concorrenti e può essere di 3 tipi:
    1. Join All: Quando il task più lento ha raggiunto la join il master continua l'esecuzione mentre gli altri terminano.
    2. Join any : Il primo task che raggiunge la join sblocca il master. Gli altri terminano dopo aver raggiunto la join.
    3. Join none: il master thread attiva la fork e continua l'esecuzione.

L'utilizzo elevato di strutture Join fork / All Join può introdurre overhead dovuti a start e stop dei thread.

Il costruttore di GPU NVIDIA ha realizzato diversi modelli di GPU che possono essere utilizzate come acceleratore del calcolo per applicazioni «data parallel» e ha sviluppato il modello di programmazione CUDA, la libreria e il relativo compilatore

Il codice seriale (non parallelo) viene eseguito sulla CPU. Il codice parallelo di tipo «data parallel» viene scritto con CUDA in un «kernel» e trasferito sulla GPU assieme ai

dati da elaborare, al termine dell'esecuzione i risultati vengono trasferiti dalla GPU alla CPU.

Sistemi Ibridi

La programmazione delle architetture ibride avviene combinando il modello passing (MPI) con il modello a thread (OpenMP message) a cui si aggiunge la programmazione CUDA se il sistema dispone di GPU.

Il flusso dei processi

1. I dati in input vengono copiati dalla memoria della CPU nella memoria della GPU.
2. Il codice viene caricato ed eseguito all'interno della GPU, dopodiché viene fatto il caching dei dati su un chip per ottenere delle migliori performance.
3. I risultati vengono copiati dalla memoria della GPU nella memoria della CPU.

Si utilizza il compilatore nvcc per compilare i programmi in cui si sfruttano le funzioni delle GPU, nvcc si occuperà anche di separare il codice sorgente distinguendolo tra gli host e i device (NVIDIA).

Ad esempio, la funzione main() verrà processata dal compilatore host (compilatore standard), e la funzione mykernel() verrà processata dal compilatore NVIDIA.

Gestione della memoria

I device e gli host sono 2 entità separate:

- I puntatori dei device puntano alla memoria della GPU.
- I puntatori degli host puntano alla memoria della CPU.

Le API di CUDA per gestire la memoria dei device sono: cudaMalloc() , cudaFree() , cudaMemcpy().
Generalmente si mette la d_ davanti al nome delle variabili del device (GPU), per differenziarle da quelle locali della CPU.

```
int main(void) {
int a, b, c; // host copies of a, b, c
int *d_a, *d_b, *d_c; // device copies of a, b, c
int size = sizeof(int);

// Allocate space for device copies of a, b, c
```

```
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Setup input values
a = 2;
b = 7;

// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

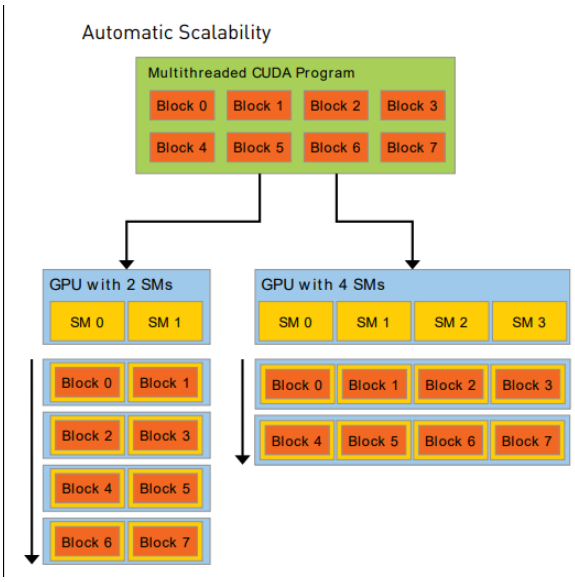
Questo è il classico pattern da adottare per la computazione parallela sulle GPU, per cui è necessario:

- fare il setup delle variabili.
- copiare quelle necessario sulla memoria della CPU.
- eseguire le funzioni kernel necessarie.
- ricopiare i valori interessati sulla memoria della CPU.

Sistema Scalabile e Blocchi

Per far si che sia possibile la computazione parallela, quando uso la funzione add(), dovrò impostare i parametri nel seguente modo: add<<< N, 1 >>>(); per far si che la funzione venga eseguita N volte in parallelo.

Utilizzando blockIdx.x per accedere al indice di quel determinato blocco, riuscirò a far si che ogni blocco gestisca elementi differenti dell’array.

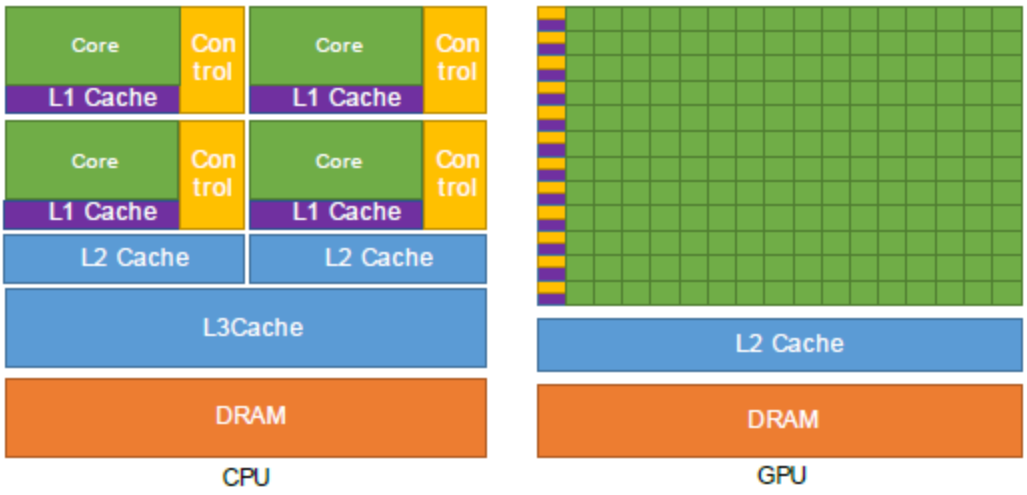


- Attraverso un sistema scalabile è possibile utilizzare lo stesso programma, adattabile a differenti tipologie di processori.
- E’ necessario pensare al programma come suddiviso in blocchi concettuali e quindi avere questi blocchi eseguibili in parallelo ognuno da un processore diverso.
  - Avere uno scheduler che assegna i blocchi ai vari processori.
  - Una scheda con più processori esegue più blocchi più velocemente.
  - I blocchi devono essere eseguiti in parallelo senza vincoli di dipendenza.

Pro e Contro delle CPU e delle GPU

- GPU
  - PRO: tanti dati trasportati contemporaneamente (banda larga).
  - CONTRO: alta latenza.

  Ogni riga verde all’interno della GPU è un SM (Streaming Multiprocessor)
- CPU
  - PRO: bassa latenza.
  - CONTRO: pochi dati trasportati contemporaneamente (banda piccola).

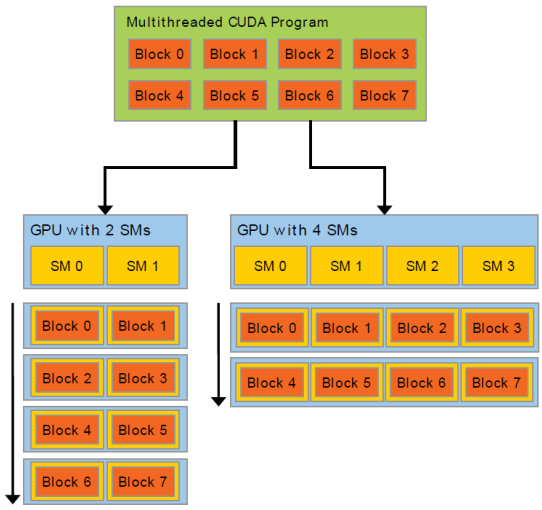


Si parla di programmazione eterogenea se il lavoro è svolto sia da CPU che da GPU.

Thread

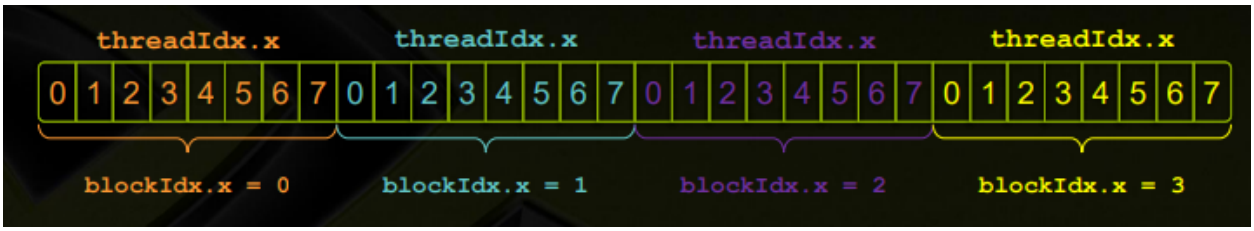
Un blocco può essere diviso in più thread paralleli, questo può essere fatto modificando la funzione add() come segue:

```
_global_ void add(int *a, int *b, int *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```



Combinazione di Blocchi e Thread

Si prende in considerazione il seguente array:




```
int index = threadIdx.x + blockIdx.x * M;
```

Questa linea calcola un indice univoco per ogni thread, quando si hanno M thread per blocco.

```
__global__ void add(int *a, int *b, int *c){
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  c[index] = a[index] + b[index];
}
```

La prima linea è la dichiarazione di una funzione chiamata ‘add’ che viene eseguita su un dispositivo GPU, il prefisso _global_ indica che la funzione verrà eseguita sulla GPU e può essere chiamata dall’ host.

La seconda linea calcola un indice univoco per ogni thread in esecuzione. In un'applicazione CUDA, l'esecuzione viene divisa in blocchi di thread, e ogni blocco può avere molti thread.

`threadIdx.x` è l'indice del thread all'interno del suo blocco, `blockIdx.x` è l'indice del blocco all'interno della griglia di blocchi e `blockDim.x` è il numero di thread per blocco.

La terza linea esegue l’addizione di vettori, ed ogni thread somma un elemento univoco dei vettori di input ‘a’ e ‘b’ e memorizza il risultato nel vettore di input ‘c’.

Data race

Supponiamo che un thread legga la shared memory prima che questa sia stata scritta,

```
shr[i] = ram[index];
a = shr[i]; //race condition
```

Con queste due istruzioni ci sono problemi di data race perché le istruzioni vengono eseguite con i WARP (pool di thread), che vengono schedulati, quello che può succedere è che un WARP si trovi più avanti di un altro e che esegua istruzioni successive che potrebbero dipendere da dati non ancora pronti o non ancora gestiti. La soluzione per risolvere il problema è una barriera di sincronizzazione data dalla funzione `void __syncthreads();` (specifica per i casi in cui è possibile avvenga ReadAfterWrite, WriteAfterRead e WriteAfterWrite) in cui nessun thread può andare avanti se tutti gli altri non hanno finito.

Esecuzioni Divergenti

```
if(threadidx.x < 10){
  //solo per i primi 10 thread
}
```

In questo caso si rende possibile l'esecuzione solo ai primi 10 thread, questo è un esempio di una barriera di sincronizzazione in cui si evita che i thread successivi ai primi 10 vadano a usare dati non ancora creati o gestiti dai primi 10.

Coordinazione tra Host e Device

Le esecuzioni da parte del kernel sono asincrone e quando terminano il controllo ritorna alla CPU immediatamente. Una volta terminate le esecuzioni del kernel la CPU ha bisogno di sincronizzarsi prima di usare i risultati ottenuti.

- cudaMemcpy()
  - Blocca la CPU fino al completamento della copia. La copia inizia quando tutte le chiamate CUDA precedenti sono state completate.
- cudaMemcpyAsync()
  - Asincrono, non blocca la CPU.
- cudaDeviceSynchronize()
  - Blocca la CPU fino al completamento di tutte le chiamate CUDA precedenti.

Gestione dei Device

- Le applicazioni possono richiedere e selezionare diverse GPU

- Più thread possono condividere un device.
- Un singolo host può gestire diversi device.

Gerarchia di memoria

Un thread ha a disposizione una memoria locale formata da registri (massimo 255.
Un blocco di thread ha a disposizione una memoria condivisa che nasce e muore con il blocco di lavoro e può essere letta solo dal blocco di lavoro.
Tutti i blocchi e tutti i thread possono accedere alla memoria globale, ma è molto costoso e richiede sincronizzazione.

Sincronizzazione esplicita

- cudaDeviceSynchronize()
  Aspetta finché tutte le operazioni di tutti i thread dell'host non sono completate.
- cudaStreamSynchronize()
  Prende uno stream come parametro e aspetta che tutte le operazioni di quello stream siano completate. Utilizzabile per la sincronizzazione di uno stream con l'host.
- cudaStreamWaitEvent()
  Prende uno stream e un evento come parametri e mette in attesa tutti le operazioni dello stream finché l'evento non è completato.
- cudaStreamQuery()
  Per sapere se tutti i comandi in uno stream sono stati completati.

Sincronizzazione implicita

Due comandi di stream diversi non possono eseguirsi concorrentemente se una di queste operazioni viene lanciata dall'host

- Una pagina di memoria viene bloccata dall'host
- Allocazione di memoria da parte del device
- Copia di memoria tra due indirizzi dello stesso device
- Qualsiasi comando CUDA sullo stream NULL

Modalità di accesso alla memoria globale

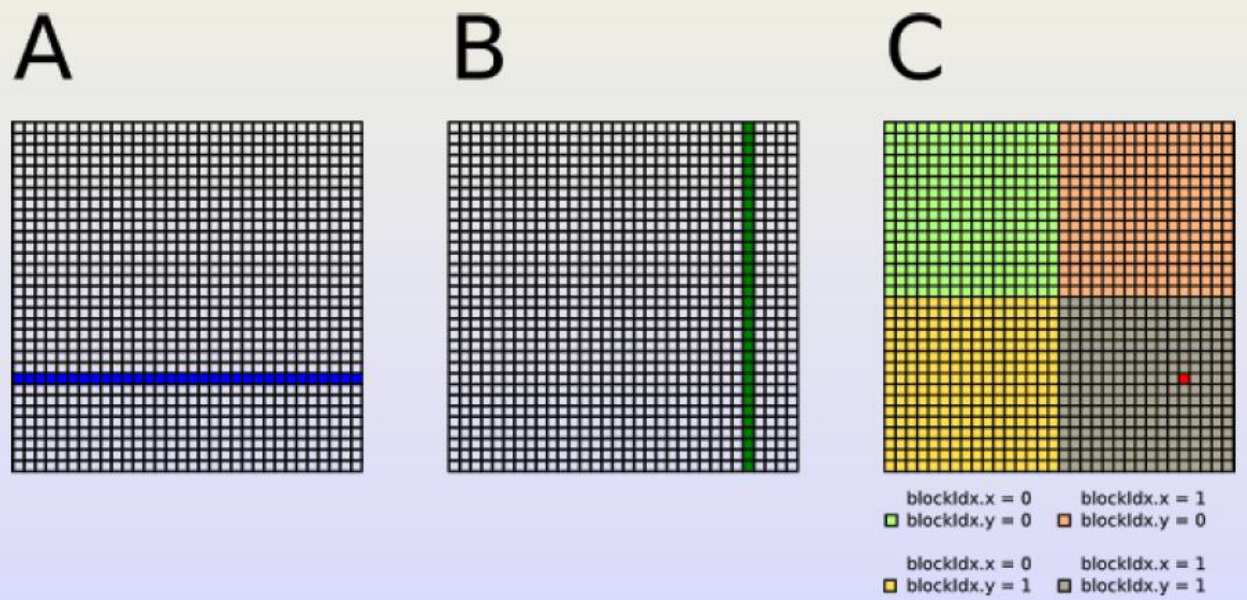
Se non si progetta bene l’accesso alla memoria globale da parte dei thread si perde molto speedup (alta latenza).
Tutte le richieste R/W dei thread di un WARP devono essere ben organizzate in modo che l’accesso sia effettuato con una transazione unica a blocchi creando dei pattern di accesso.
L’accesso concorrente dai vari thread del WARP si unisce automaticamente in un numero di transazioni equivalente al numero di transazioni a 32 byte necessarie a servire tutti i thread, quindi c’è un’organizzazione a blocchi di 32 byte.

Moltiplicazione di matrici

Il prodotto tra matrici è possibile solo se il numero di colonne della prima è uguale al numero di righe della seconda.
Ogni cella del risultato è il prodotto scalare riga colonna.
Ogni cella di A o B viene letta K volte (K = colonne prima matrice = righe seconda matrice).

```
void main(){
  for i = 0 to M do //per tutte le righe di A
    for j = 0 to N do //per tutte le colonne di B
      /* compute element C(i,j) */
        for k = 0 to K do //si scorrono gli elementi di C
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
          //valore precedente + il prodotto della coppia
        end
    end
 end
}
```

L’unico vincolo per la parallelizzazione è l’accumulo (calcolo degli elementi di C), calcolato il risultato questo viene caricato in memoria globale.

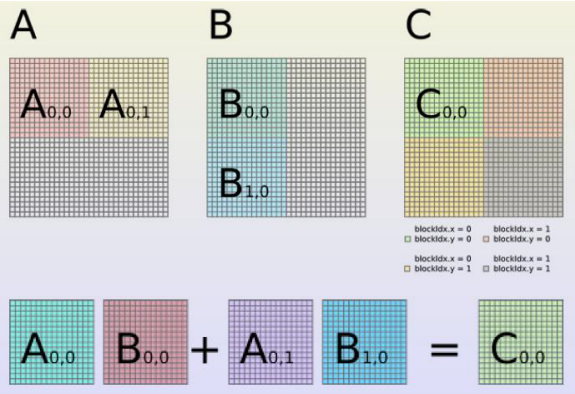


Per fare il calcolo del prodotto notiamo che i risultati di ogni blocco sono indipendenti dagli altri, quindi sono parallelizzabili senza sincronizzazione.

Utilizzo della memoria shared

Per calcolare un prodotto si devono leggere una colonna e una riga e così dovrò fare per tutti i prodotti della matrice C, quindi per calcolare il risultato che verrà memorizzato in C dovrò leggere k volte righe e colonne, tutto questo, usando un solo blocco di lavoro.

L'opzione migliore è di partizionare le matrici A e B in diverse zone di lavoro in modo da fare operazioni che riguardano diversi quadranti della matrice



Simulazione trasferimento di calore

Si ha una griglia che contiene valori di temperatura, alcuni sono fissi durante la simulazione , altri devono essere calcolati in base ai valori dei precedenti per ricostruire i flussi di calore.

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k \cdot (T_{NEIGHBOR} - T_{OLD})$$

Si calcolano le differenze tra la cella old e i vicini.
Dato che tutti i conti sono fatti su una “fotografia” bisogna scrivere il risultato su un’altra matrice, altrimenti gli altri calcoli paralleli andrebbero a leggere valori già aggiornati e quindi sbagliati.
Si hanno quindi due buffer

1. uno con la matrice new (in scrittura)
2. uno con la matrice old (in lettura)

anche se è molto costoso in termini di memoria, quindi si può modificare la formula:

$$T_{NEW} = T_{OLD} + k \cdot (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 \cdot T_{OLD})$$

- k è la velocità di diffusione del calore, si decide a priori (uguale a 1 in questo caso, non si può aumentare troppo perché si deve tenere conto dei tempi di calcolo).
- Si effettua la somma dei 4 vicini immediati meno il proprio valore moltiplicato per 4.
- Si ha un thread per ogni cella, in maniera contigua.
- Si può immaginare che tutti i thread del WARP chiedano come primo elemento quello sopra, quindi tutti i thread chiederanno la fila di celle sopra, questo sarà il pattern della prima richiesta in blocco in memoria.
- Dato che sono indirizzi contigui si ricadrà in un blocco contiguo di memoria, è una lettura efficiente.
- Come secondo elemento tutti i thread chiederanno la cella a destra, si tratta sempre di indirizzi contigui. Il terzo valore sarà quello sotto, il quarto quello a sinistra, sempre con lo stesso ragionamento.

Implementazione

Ad alto livello si ha un’iterazione che sostituisce i valori vecchi con i nuovi.
Dato che alcuni elementi non cambiano valore, anche se saranno ricalcolati dovranno essere reimpostati al loro valore costante.
Si ha bisogno, per alcuni punti, di ricopiare i valori costanti nella posizione corretta.
Conviene creare un kernel che si occupa solo della scrittura delle celle costanti.
Questo kernel ha a disposizione la matrice iniziale che descrive le celle costanti, se il valore è diverso da una costante (scelta a priori) si può copiare.
Si preparano quindi kernel separati, il primo per portare tutti i valori da scrivere prima di elaborare, il secondo per l’elaborazione.
Sono sincroni, il secondo parte solo se il primo ha finito.
Per evitare problemi di sincronizzazione e di dipendenze sui dati si può uscire dal kernel che ha terminato, sincronizzare da CPU e far partire un altro kernel.
Se si ha una lettura in memoria globale la suddivisione in blocchi non influenza il processo dato che gli accessi sono organizzati in pattern buoni, anche se si va a richiedere la lettura di un valore gestito da un altro thread di un altro blocco.
Le celle ai bordi, che lavoreranno con 3 o 4 dato al posto di 5, devono essere controllate affinché i thread non facciano richieste in memoria ad indirizzi non definiti.
I controlli in questo caso sono necessari anche se portano a divergenza del codice.
Kernel per la copia dei valori costanti

```
__global__ void copy_const_kernel( float *iptr, const float *cptr ) {
  // map from threadIdx/BlockIdx to pixel position
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;
  int offset = x + y * blockDim.x * gridDim.x; //la matrice è linearizzata
  if (cptr[offset] != 0) iptr[offset] = cptr[offset];
}
```

global void copy_const_kernel(float *iptr, const float *cptr) { // map from threadIdx/BlockIdx to pixel position int x = threadIdx.x + blockIdx.x * blockDim.x; int y = threadIdx.y + blockIdx.y * blockDim.y; int offset = x + y * blockDim.x * gridDim.x; //la matrice è linearizzata if (cptr[offset] != 0) iptr[offset] = cptr[offset]; }
Se la memoria costante ha un valore diverso da 0 si va a sovrascrivere sulla matrice old il valore che si trova in modo da resettarlo.

GPU e MPI

Usando in parallelo sia GPU che MPI aggiunge un altro livello di parallelismo, per cui si hanno n processi ciascuno con una GPU dedicata.
Si prepara un modello che cresce in maniera regolare, si ha una matrice fatta da due pezzi, si taglia a metà e ogni metà viene assegnata ad una GPU.

Si prepara un modello che cresce in maniera regolare, si ha una matrice fatta da due pezzi, si taglia a metà e ogni metà viene assegnata ad una GPU.