

Programmazione su Architetture Parallelle

Appunti

Andrea Mansi UniUD

II° Semestre 2020/2021

Contents

1 Introduzione	4
1.1 Tipologie di architetture parallele	5
1.2 Shared vs Distributed Memory	6
1.3 Parallel Computing: problem perspective	7
1.4 Performance, Portabilità, Produttività	7
1.5 Recap dell'architettura di Von Neumann	8
1.6 Ciclo Fetch-Decode-Execute	8
1.7 Alcune considerazioni sulle performance	9
1.8 Latenza e Throughput	10
1.9 Tassonomia di Flynn	11
1.10 SISD	11
1.11 Parallelismo implicito	11
1.12 Pipeline	13
1.13 Throughput vs Latenza	14
1.14 SIMD	16
1.15 Considerazioni sulle performance	20
1.16 MIMD	20
1.17 Alcuni paragoni	23
1.18 MISD	23
2 Graphics Processing Unit	24
2.1 Thread-Level Parallelism	24
2.2 GPU vs CPU MIMD	24
2.3 Architettura Nvidia Volta	25
2.4 Alcune definizioni	28
2.5 CUDA in a nutshell	29
2.6 SIMT vs SIMD	32
2.7 Thread Scheduling	32
2.8 Multithreading Hardware	33
3 CUDA Introduction	34
3.1 CUDA Programming Model	34
3.2 Compilation workflow	37
3.3 Esempio codice CUDA in C	38
3.4 CUDA runtime	39
3.5 Esecuzione concorrente ed asincrona	50
4 Concorrenza e Parallelismo	59
4.1 Concorrenza e Parallelismo in CUDA	60
4.2 Parallelismo dinamico in CUDA	71

Informazione sugli appunti

I questi appunti **non ufficiali** del corso di Programmazione su Architetture Parallelle (Andrea Formisano - DMIF - Università degli studi di Udine - 2020/2021) vengono riportati i principali concetti relativi al corso.

Requisiti:

- Programmazione C;
- Algoritmi e strutture dati;
- Nozioni base sulle architetture degli elaboratori;
- Nozioni base sui sistemi operativi.

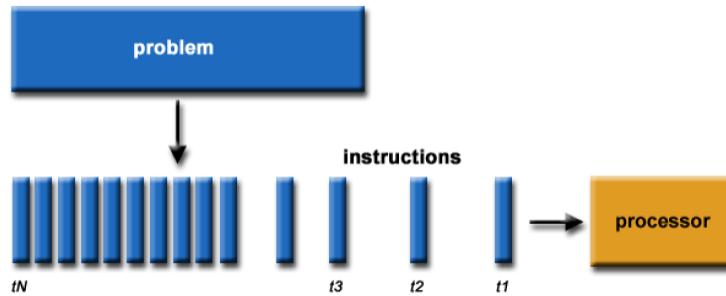
Contenuti:

- Introduzione alle architetture parallele;
- Introduzione alla programmazione parallela su:
 - su GPU CUDA compatibile (Nvidia);
 - sistemi paralleli con memoria condivisa (OpenMP);
 - sistemi paralleli con scambio di messaggi (MPI).
- Progettazione di algoritmi paralleli;
- Soluzioni comparate di problemi con algoritmi paralleli.

1 Introduzione

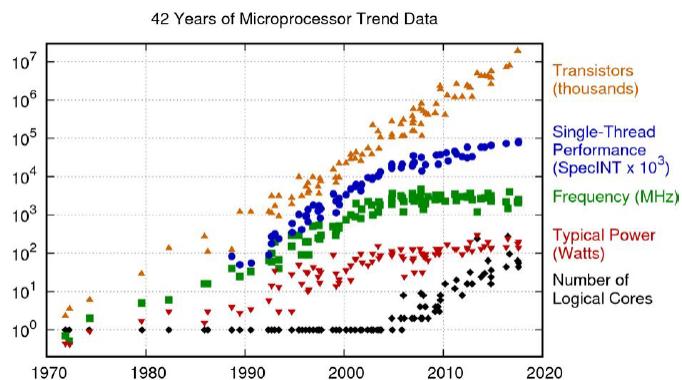
Prima di parlare di **computazione parallela** è consigliato rivedere il concetto di **computazione seriale**.

Si dice di essere in una situazione di computazione seriale se il programma in esecuzione è costituito-da/concepito-come una **sequenza** di istruzioni. Queste istruzioni vengono eseguite sequenzialmente, una alla volta. L'esecuzione è compiuta da un singolo processore e in ogni istante c'è solo una istruzione che può essere eseguita (non che viene, ma che può! è diverso).



Dato un determinato problema, se volessimo velocizzare l'esecuzione dell'algoritmo che ne determina la soluzione potremmo aumentare la potenza di calcolo (CPU con clock più elevato e/o architettura più moderna ed efficiente) ma saremmo limitati dalla legge di Moore (sostanzialmente se non siamo soddisfatti della velocità di esecuzione con la CPU più potente a nostra disposizione, dobbiamo aspettare che la tecnologia faccia progressi), oppure aumentando la velocità delle memorie (incomberemmo nello stesso problema della CPU).

Con il progresso tecnologico però, i processori oltre ad essere diventati più efficienti, i transistor sono diventati anche sempre più piccoli, di conseguenza si è reso possibile aumentare il numero di core nello stesso die.



A questo punto sorge spontaneo chiedesi: è meglio progettare una architettura con tanti core o pochi core veloci?

Single-core CPU:

- Complesso HW di controllo;
- Pro: flessibilità e performance

- Contro: costo e consumo energetico elevati

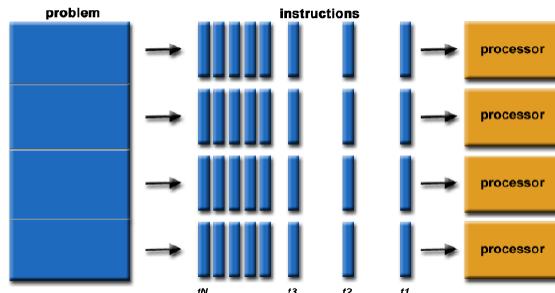
Multi-core CPU:

- HW di controllo semplice;
- Pro: in generale minor costo e minori consumi (ops/Watt)
- Contro: programming modelli più vincolati e complessi (meno flessibilità)

Quindi, chiari questi aspetti possiamo definire il **parallel computing** come:

“ L’uso simultaneo di più risorse computazionali per risolvere un problema. ”

Il programma viene ”diviso” in parti che possono essere completate concorrentemente. Ciascuna di queste parti è composta da una successione di istruzioni. Le istruzioni di una parte vengono però eseguite sequenzialmente, una alla volta, da un processore. In ogni istante quindi più istruzioni sono eseguite simultaneamente, da diversi processori. Nasce l’esigenza di un meccanismo di controllo e coordinamento globale.



Ovviamente, per potere seguire un approccio parallelo, il programma/problema in questione deve poter essere decomposto in parti che possano essere risolte simultaneamente, e permettere così l’esecuzione di più istruzioni in contemporanea (non sempre questo è possibile, se ogni istruzione dipende dall’esito di quella precedente l’esecuzione deve essere per forza seriale).

L’obiettivo è quindi cercare di risolvere il problema utilizzando più risorse computazionali in minor tempo rispetto all’approccio interamente seriale.

1.1 Tipologie di architetture parallele

Esistono più architetture parallele, ad esempio le architetture multi-processori/multi-core con bus di interconnessione (processori Intel, AMD, ARM, etc.), sistemi multi-processori/multi-core connessi in rete (vedi super calcolatori come l’IBM Summit). Esistono anche acceleratori e architetture domain-specific (GPU - Schede grafiche, TPU - Tensor Processing Unit), System on a chip, etc.

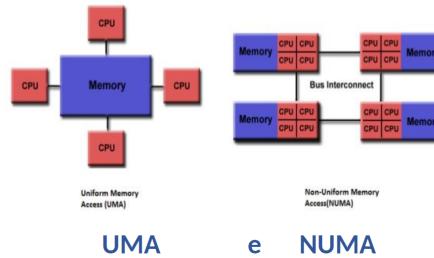
1.2 Shared vs Distributed Memory

Una prima differenza tra architetture può essere la tipologia di memoria tra i diversi processori: **condivisa** o **distribuita**. La caratterizzazione avviene in base alla organizzazione della memoria e al modo in cui i processori accedono ad essa. Un programming model fornisce tipicamente una astrazione rispetto alla architettura HW della memoria, ma questa architettura comporta alcune limitazioni e/o vantaggi.

Nel caso della **Shared-Memory** i processori di un computer parallelo possono accedere indipendentemente alla memoria che è condivisa (shared).

- Pro: accesso veloce, modifiche visibili a tutti i processori, bassi costi di sincronizzazione;
- Contro: gestione della concorrenza negli accessi alla stessa locazione di shared memory, gestione della coerenza delle copie cached (false-sharing), etc.

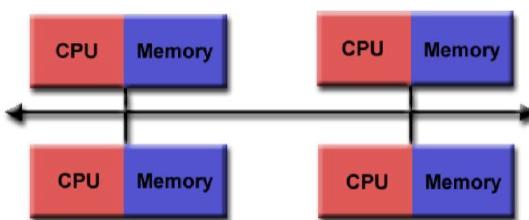
Implementazione: Open Multiprocessing (OpenMP).



Nel caso della **Distributed-Memory** ogni processore ha la propria memoria locale e vi può accedere indipendentemente. I computer sono connessi tramite una rete.

- Pro: elevata capacità di memoria, scala con il numero di nodi, ogni nodo ha la sua memoria;
- Contro: la topologia e la tecnologia della rete influenzano la performance; necessità di considerare/massimizzare la località dei dati (data locality); alti costi di sincronizzazione.

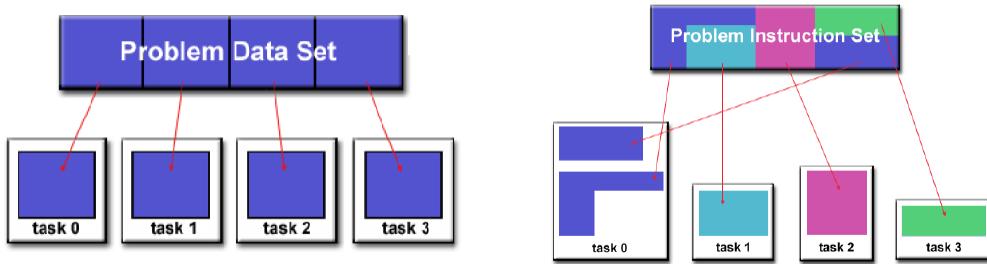
Implementazione: Message Passing Interface (MPI).



1.3 Parallel Computing: problem perspective

Vi sono due modi principali per partizionare il lavoro in “task” paralleli:

- **Data (domain) decomposition:** i dati sono suddivisi in “pezzi” più piccoli. Ogni task parallelo processa una porzione dei dati;
- **Task decomposition:** il problema viene decomposto in sotto problemi suddividendo il lavoro da compiere. Ogni task risolve un dei sotto-problemi (es: dataflow programming).



Un approccio ottimale alla programmazione parallela considera hardware e software (co-design approach). Tradizionalmente il parallelismo è stato considerato prevalentemente da una prospettiva architetturale (HW). Tuttavia, attualmente risulta vantaggioso adottare una prospettiva più ampia, che tenga conto di tutte le componenti che possono influenzare le performance (progettazione di algoritmi, decomposizione dei dati, librerie e primitive, modelli di programmazione e implementazioni, run-time e compilatori, architetture e interconnessioni di rete).

1.4 Performance, Portabilità, Produttività

La **performance** è la misura del completamento di un task secondo una specifica metrica. Le metriche possibili sono varie:

- tempo, energia consumata, risorse utilizzate, etc.
- operazioni per unità di tempo (istruzioni, floating-point ops, accessi alla memoria), efficienza, scalabilità (weak/strong), etc.
- misure domain-specific (es: inferenze compiute per unità di tempo, archiprocessati per unità di tempo, etc.).

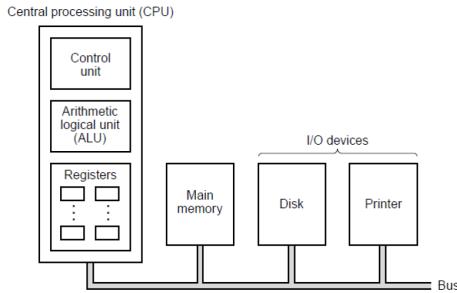
La **portabilità** è la capacità di un software di essere eseguito su differenti architetture o sistemi (ad esempio, OpenCL favorisce la portabilità fornendo un programming model che astrae le differenti architetture).

Portabilità della performance: è la capacità di un software di ottenere performance “coerenti” su diverse architetture.

Produttività: può essere considerata come una misura del rapporto tra i risultati ottenuti (ad es, la performance) rispetto allo sforzo profuso, come la difficoltà di progettare codice parallelo, di debugging, di tuning, di manutenzione, etc.

1.5 Recap dell'architettura di Von Neumann

I moderni computer sono basati sul modello di architettura detto “alla Von Neumann” (o, Princeton architecture). Anche definiti come ”stored-program computer”. Sia le istruzioni del programma che i dati sono collocati nella stessa memoria. Il modello di Von Neumann è quindi semplice: un unico “address space” (per dati e codice), un bus, una unità di computazione.



I componenti base sono quindi:

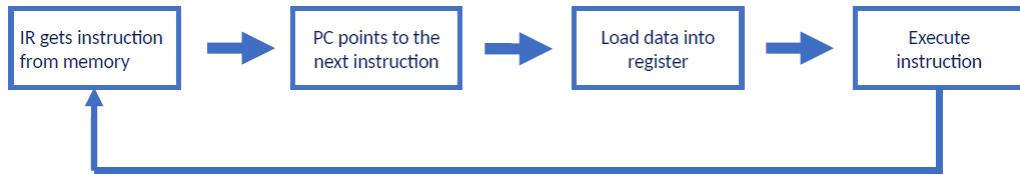
- Memoria;
- Control Unit;
- Aritmetic Logic Unit (ALU)

Read/write random access memory usata per memorizzare istruzioni e dati. Le istruzioni del programma sono codificate come i dati e specificano al computer cosa fare. I dati sono informazioni elementari utilizzate dal programma.

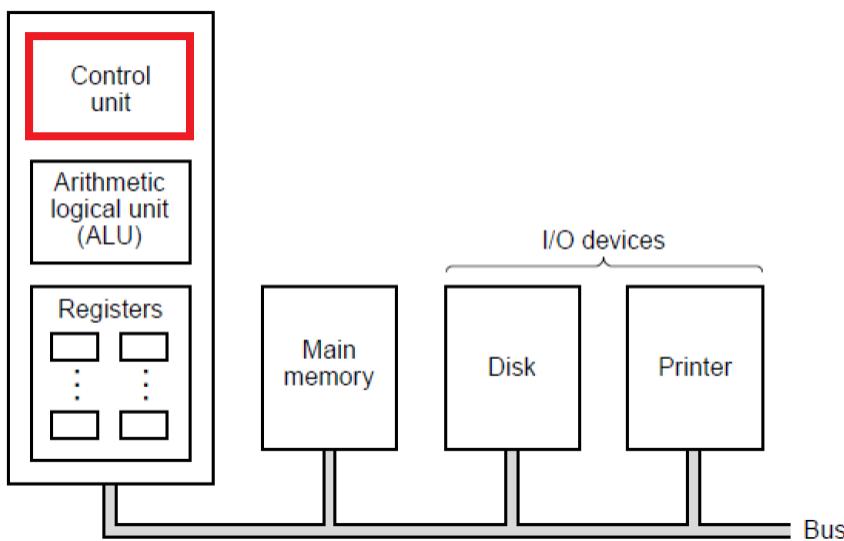
La control unit preleva (fetch) istruzioni/dati dalla memoria, decodifica le istruzioni e coordina sequenzialmente le operazioni da eseguire per completare il task programmato. L' ALU esegue le operazioni aritmetiche di base. L' Input/Output costituisce l'interfaccia con l'esterno (l'utente).

1.6 Ciclo Fetch-Decode-Execute

1. L'Instruction Register (IR) contiene la prossima istruzione (prelevata/fetch dalla memoria);
2. Il Program Counter (PC) individua la prossima istruzione;
3. Decode: determina il tipo di istruzione in IR;
4. Se l'istruzione necessita di word di memoria le si indirizza;
5. Si prelevano tali word e si caricano nei registri della CPU;
6. Si esegue l'istruzione.



Central processing unit (CPU)



L'ALU esegue diverse operazioni aritmetiche e bitwise su numeri binari interi. La FPU opera su numeri in floating-point. Le operazioni possono essere di due tipi: register-memory o register-register. Il processo degli operandi nella ALU e il salvataggio del risultato è detto **data-patch cycle**.

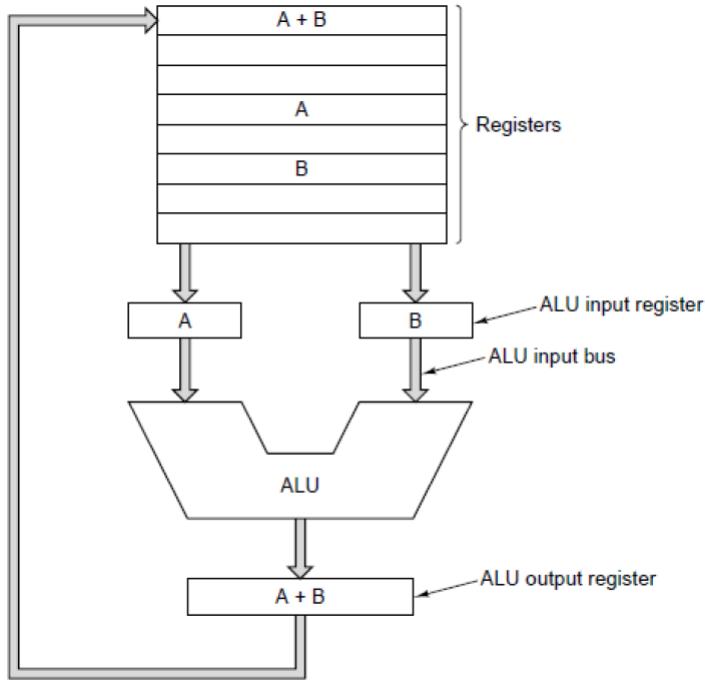
1.7 Alcune considerazioni sulle performance

Alcuni considerazioni per punti:

- Il processore esegue a velocità costante (frequenza del clock);
- La performance è influenzata dal numero di cicli F-D-E eseguiti al secondo: MIPS;
- Differenti istruzioni possono impiegare diverso tempo (ad es. le architetture CISC o ibride contemplano istruzioni di diversa lunghezza/durata);
- Una metrica spesso utilizzata è il numero di FLOPS (Floating-Point Operation per Second).

La performance dipende ovviamente anche dal task ed è influenzata da diversi fattori:

- Clock cycle time: tecnologia HW e organizzazione architetturale;
- Clock cycle per instruction (CPI: numero di cicli per completare l'istruzione): architettura e ISA;



- Instruction count (IC: numero di esecuzioni di istruzioni in un programma): ISA, struttura/tecnologia del compilatore (il programma usato per il benchmark).

Segue quindi la definizione di **tempo di CPU**:

$$\text{Tempo di CPU} = \text{IC} * \text{CPI} * \text{Clock-cycle time}$$

1.8 Latenza e Throughput

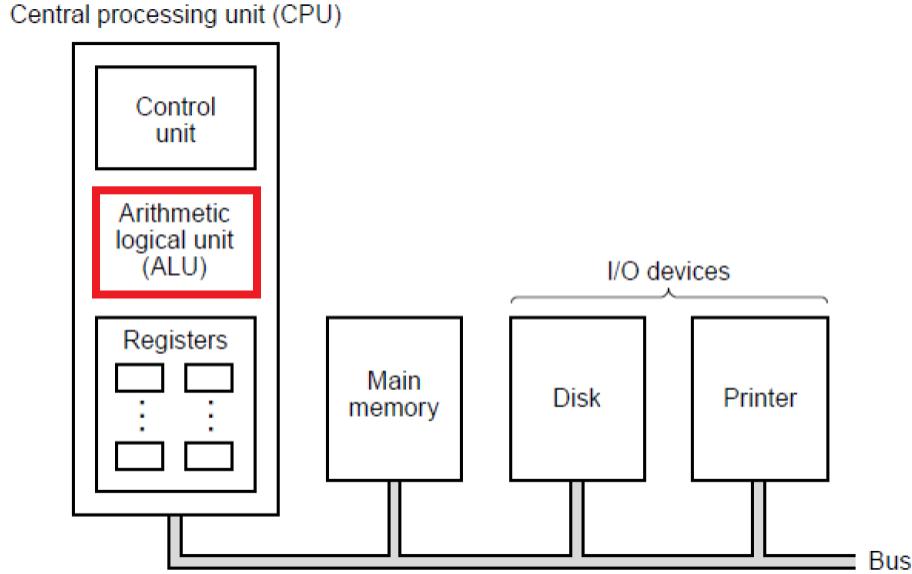
Due nozioni importanti sono quelle di:

- **Latenza:** il numero di cicli di clock necessari affinchè una istruzione ottenga i dati necessari da una altra istruzione (minimizzare);
- **Throughput:** la quantità di lavoro svolto nell'unità di tempo (massimizzare).

Si distingue tra architetture

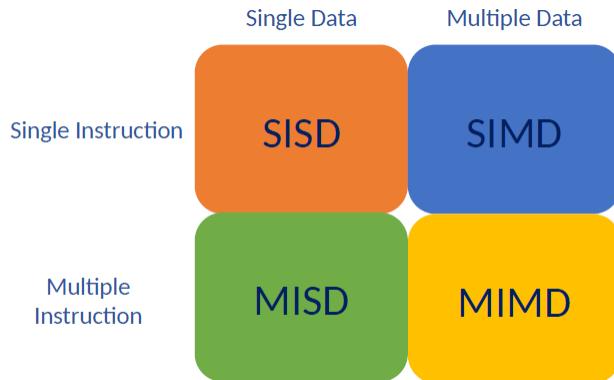
- **Latency-oriented:** mirano a minimizzare la latenza ottimizzando il tempo di esecuzione della singola istruzione seriale (tradizionali microprocessori scalari);
- **Throughput-oriented:** massimizzare il lavoro compiuto rispetto al tempo (il focus è su workload/task paralleli, anche a scapito della latenza del singolo task. Esempio: GPU). I concetti si applicano sia al processore che alla memoria.

I concetti si applicano sia al processore che alla memoria.



1.9 Tassonomia di Flynn

Diverse architetture seguono diverse organizzazioni, queste possono essere raggruppate seguendo la tassonomia di Flynn:

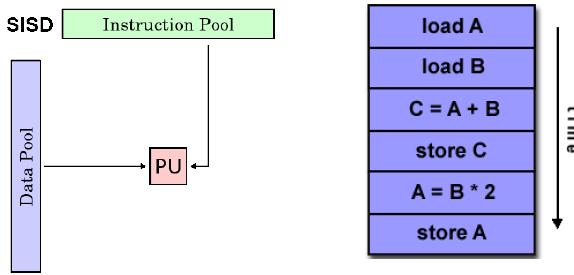


1.10 SISD

Nelle architetture seriali, durante un ciclo di clock è presente un solo instruction stream in esecuzione sulla cpu e un unico data stream utilizzato come input. L'esecuzione avviene in-order: le istruzioni sono eseguite e completato nell'ordine generato dal compilatore. Questo avviene quando una applicazione viene ad esempio eseguita su un singolo core.

1.11 Parallelismo隐式

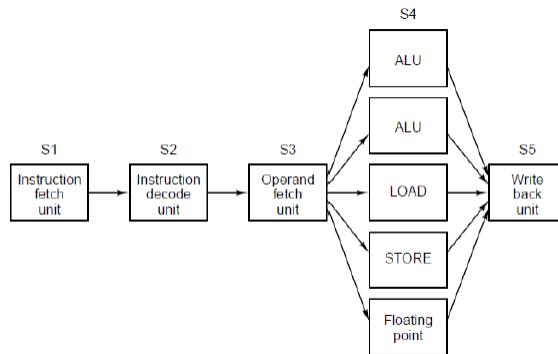
Negli ultimi decadi, al fine di migliorare i tempi di esecuzioni, il trend principale è stato aumentare il clock rate, però i sempre più livelli di integrazione portano alla disponibilità di



un maggior numero di transistor (a parità di spazio - più core). Nasce quindi il problema di come sfruttare tale disponibilità della "risorsa transistor".

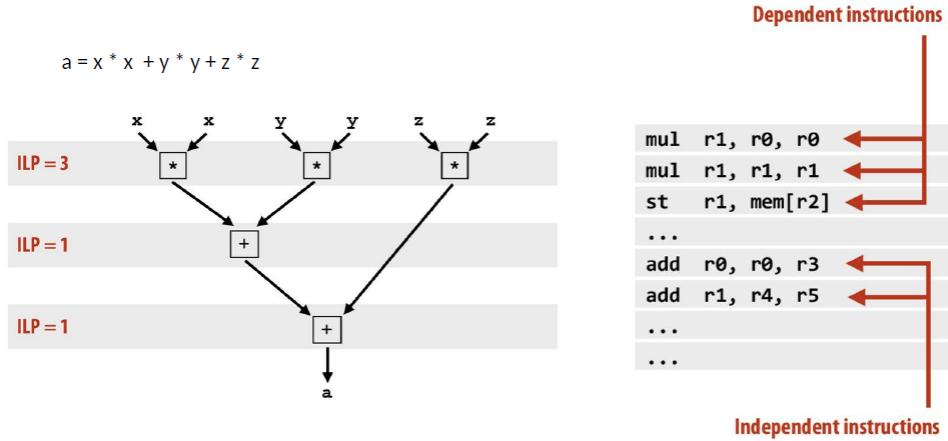
I processori attuali impiegano la "risorsa transistor" per realizzare molteplici unità funzionali in modo da poter eseguire più istruzioni nello stesso ciclo di clock (si parla di: instruction level parallelism e architetture super-scalari). Lo specifico modo in cui tali istruzioni sono selezionate ed eseguite differenzia l'architettura, ad esempio: out-of-order execution, pipelining. In questo caso si parla di **parallelismo a livello hardware**.

Nel caso di **parallelismo a livello di istruzione (ILP)**, si introducono delle execution unit (EU - anche dette functional unit - FU), ciascuna delle quali esegue specifiche operazioni (load, store, op. aritmetiche, etc.) dettate dal programma. Una EU può includere dei registri ed un controllo (internal control sequence unit - distinta dalla control unit della CPU). Esempi: ALU, AGU (address generation unit), FPU (floating-point unit), LSU (load-store unit), BEU (branch execution unit). Le architetture superscalari posseggono molteplici EU. Il meccanismo di fetch delle istruzioni deve opportunamente recuperare le istruzioni e assegnarle alle unit. L'instruction stream è tuttavia ancora sequenziale ma una CPU superscalare può eseguire più di una istruzione in un solo ciclo di clock. Ciò avviene distribuendo simultaneamente diverse istruzioni a differenti EU. Il risultato è un aumento del throughput della CPU. Si noti che: ogni execution unit non è un processore separato (o un core di un processore multi-core), ma una risorsa HW che esegue internamente alla CPU (come ad es. la ALU). Un processore superscalare è una CPU che implementa una forma di parallelismo detto instruction-level parallelism (Ribadiamo: in un singolo processore). Impone di determinare le instruction dependencies.



Se si pensa al seguente programma:

$$1. \quad e = a + b$$



$$2. f = c + d$$

$$3. m = e * f$$

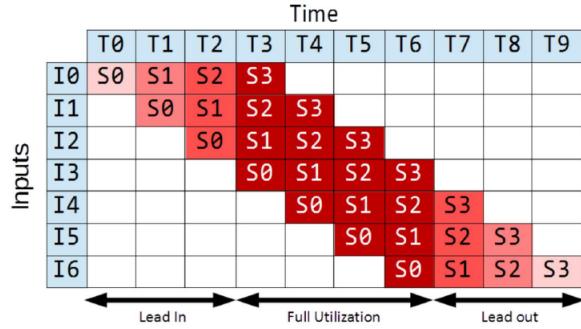
1 e 2 sono indipendenti e possono essere eseguite in parallelo, mentre 3 dipende dai risultati delle altre. Se si pensa a una CPU superscalare con almeno 2 EU in grado di eseguire le somme, saranno richiesti solo 2 cicli di clock (il primo per 1 e 2, il secondo per 3).

Nella tassonomia di Flynn un processore superscalare single-core viene classificato SISD. Si possono anche adottare altre tecniche per incrementare la performance di un superscalare:

- Speculative execution: esegue istruzioni senza la certezza che sia necessario farlo: predire i risultati (per es. nei branch condizionali) ed eseguire;
- Out-of-Order (OoO) execution: l'ordine delle istruzioni può essere modificato (a runtime);
- Pipeline;
- Vectorization (SIMD).

1.12 Pipeline

L'approccio pipeline mira a ridurre la latenza e massimizzare l'uso delle risorse hardware. Il parallelismo pipeline è quando più passi dipendono l'uno dall'altro, ma l'esecuzione può sovrapporsi e l'output di un passo è trasmesso come input al passo successivo. Il parallelismo pipeline si estende sul semplice parallelismo dei compiti, rompendo il compito in una sequenza di fasi di elaborazione. Ogni fase prende il risultato della fase precedente come input, e i risultati vengono passati a valle immediatamente. Una buona analogia è una catena di montaggio dell'automobile. Ogni stazione esegue un'azione autonoma, per esempio saldare il telaio o installare il parabrezza, ma dipende da qualche altro compito che è stato fatto prima. Ogni stazione lavora simultaneamente su una (diversa) auto parzialmente assemblata, e quando tutte le stazioni hanno finito, le auto vengono inviate alla stazione successiva. Dopo l'ultima stazione, è stata prodotta un'auto completamente assemblata. Allo stesso modo, in

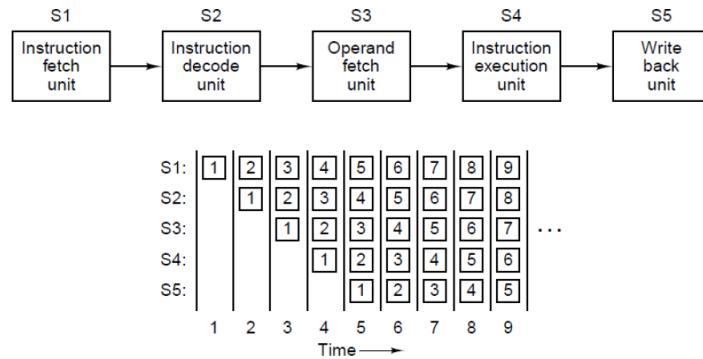


una pipeline, ogni pezzo di dati si muove da uno stadio all'altro, producendo alla fine un risultato finale.

Si parla di **pipeline bilanciata** quando tutti gli step/stage impiegano lo stesso tempo (situazione teorica). Possiamo definire due valori:

$$\text{Throughput bound} = 1/\max(\text{computation_time(stages)})$$

$$\text{Latency bound} = \#\text{stages} * \max(\text{computation_time(stages)})$$



1.13 Throughput vs Latenza

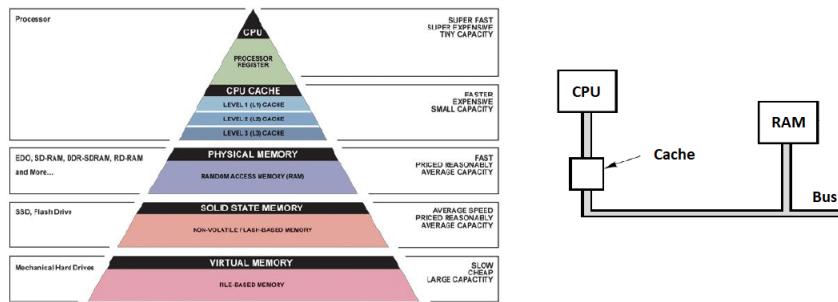
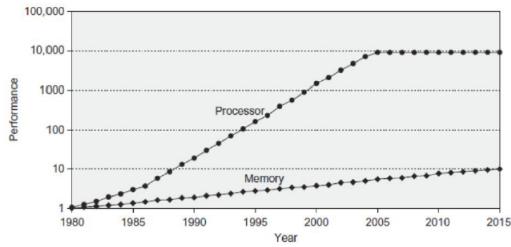
Spesso ottimizzare il throughput porta ad aumenti della latenza (potrebbe anche accadere che nello spezzare uno stage in sub-stage la somma delle durate di questi sia superiore). Tipicamente il pipelining introduce un overhead costante tra gli stage (dovuto a comunicazione, sincronizzazione, etc.), quindi:

- Soluzioni con troppi stage non sono praticabili;
- Il tempo speso da un task per attraversare la pipeline può essere superiore a quello della sua esecuzione seriale.

1.13.1 Cache

Come accennato precedentemente, l'accesso alla memoria tende a essere più lento dell'accesso ai registri della CPU (aumenta la latenza). Un metodo per ridurre la latenza è utilizzare

della cache. L'idea alla base della cache è la seguente: utilizzare una memoria piccola e veloce tra CPU e RAM per memorizzare le word (dati) più usate frequentemente.



Nel caso che venga utilizzata della cache, la cpu accede prima ad essa prima di caricare dati dalla memoria centrale. Il **principio di località** asserisce che i programmi accedono una piccola porzione dello spazio degli indirizzi in ogni istante di tempo:

- **Località temporale:** se un item è stato riferito, tenderà ad esserlo di nuovo a breve;
- **Località spaziale:** se un item è stato riferito, gli item ad esso vicini tenderanno ad essere riferiti a breve.

Si ha che: $\text{mean_access_time} = c + (1 - h) * m$

dove c è il *cache_access_time*; h è l'*hit ratio*; m è l'*access time* alla memoria principale.

La cache è organizzata in **linee di cache**. Si dice che si ha un **cache miss** quando la word desiderata non è presente in cache:

- fetch della word dalla memoria di livello subito inferiore nella gerarchia (che comporterà una latenza maggiore).
- al livello inferiore potrà esserci la memoria principale oppure un'altra cache.

Si parla di **miss rate** riferendosi alla frazione di accessi alla cache che sono cache miss. Le principali cause dei miss sono:

- **Compulsory:** il primo accesso a un blocco.
- **Capacity:** se la cache non può contenere tutti i blocchi necessari, si ha miss perché i blocchi vengono scaricati e ricaricati successivamente.

- **Conflict:** quando il programma effettua ripetuti accessi a diversi indirizzi di diversi blocchi che però sono mappati sulla stessa locazione in cache.

Si parla invece di **Cache hit** quando il dato è presente in cache e viene caricato nei registri della cpu. Altre metriche importanti sono:

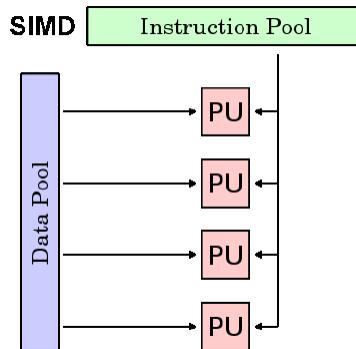
$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Missrate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Le dimensioni della cache sono un aspetto fondamentale da considerare. Maggiore dimensione del blocco di cache meglio è (riduce il miss rate) ma aumenta il costo dell'architettura, consumo e tempi di accesso.

1.14 SIMD

Ritornando alla tassonomia di Flynn, le architetture SIMD sfruttano il data-level parallelism. Le architetture SIMD sono solitamente energeticamente più efficienti di altre architetture più complesse (come MIMD): necessitano di un solo fetch per ogni operazione (parallela); ciò rende SIMD attraente anche per i device mobili. SIMD tuttavia permette al programmatore di continuare a pensare il codice in modo sequenziale (vectorization).



Con SIMD ci si riferisce al caso di una singola istruzione per ciclo di clock (SI) e in cui i processing unit eseguono la stessa istruzione, inoltre operando su un dato diverso (MD). Si ha comunque una esecuzione sincrona e deterministica (lockstep). Esistono diversi approcci SIMD:

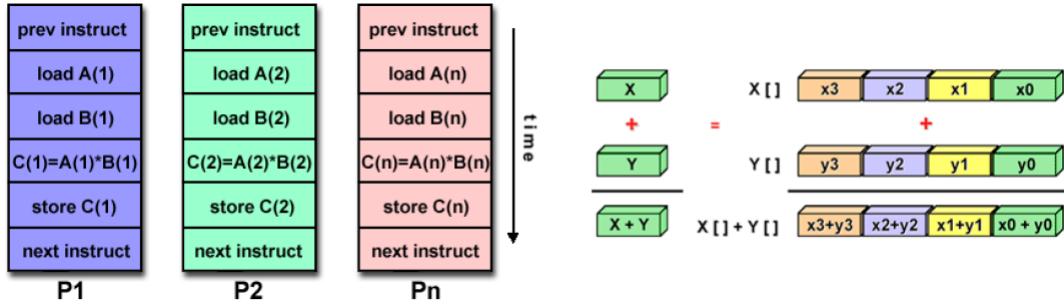
- Architetture vettoriali;
- SIMD extensions: AVX, etc.;
- Graphics Processor Units (GPUs).

È importante ricordarsi che SIMD differisce da SIMT: single instruction multiple threads.

Vediamo ora una intuizione generale del SIMD. Si pensi al seguente esempio: $z = x + y$ (vettori).

1. load:(mem->registri)
2. operazione
3. store:(registri->mem)

Con un approccio SISD si utilizzerebbe un solo registro per rifornire una FU alla volta. Nell'approccio SIMD/Vettorizzato si utilizzerebbero n registri alla volta, come da esempio nell'immagine:



1.14.1 Architetture vettoriali Analizziamo ora il caso delle architetture vettoriali. L'idea di base è leggere insiemi di dati caricandoli in **vector registers**, compiere le operazioni su tali registri e riportare i risultati in memoria. L'impiego dei registri è controllato dal compilatore in modo da compensare la latenza di memoria (pipelining nelle operazioni di load/store) e sfruttare al meglio il memory bandwidth.

Nelle architetture vettoriali si devono manipolare vettori di dati, quindi ci saranno:

- vettore di registri (ad esempio: 8-32 vector registers ciascuno contenente 64-128 elementi da 64bit);
- vettore di FU (pipelined e operano in sincronia sugli elementi dei vector registers);
- vettore di unità load-store (pipelined);
- registri per dati scalari.

Si consideri il seguente esempio con le operazioni di somma di due vettori, somma scalare a vettore e load/store di un vettore: ADDVV.D, ADDVS.D, LV/SV.

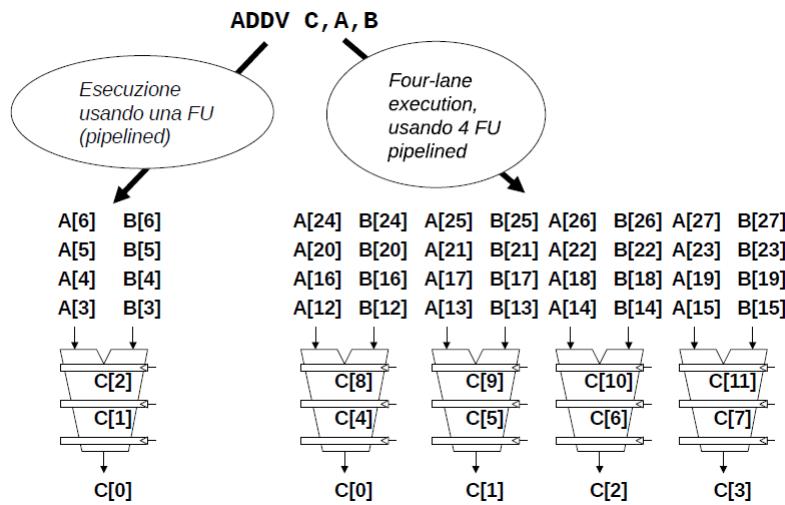
Le istruzioni SIMD operano quindi su più elementi in contemporanea e la stessa FU effettua più operazioni su "porzioni" dei vector registers grazie all'indipendenza tra gli elementi del vettore di dati (evitando dependency check).

Le architetture vettoriali sono caratterizzate da elevata flessibilità e permettono elevata compattezza delle istruzioni: una istruzione codifica n operazioni sulla stessa FU, consentendo così di ciclare su porzioni di dati più grandi. Sono scalabili in quanto si può eseguire lo stesso codice in FU pipelined e multiple (più lanes) come illustrato in figura (FU pipelined e lane multiple).

È importante considerare il funzionamento delle architetture vettoriali relativamente all'accesso alla memoria. Si pensi al seguente esempio di operazioni con matrici.

Per "vettorizzare" la moltiplicazione di una riga B per una colonna di D bisogna:

# C code	# Scalar Code	# Vector Code
<pre>for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre>LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>



- accedere a elementi adiacenti di B ed elementi adiacenti di D;
- Gli elementi di B devono essere acceduti per righe e quelli di D per colonne (NB: C memorizza per righe, Fortran per colonne).

Sono quindi necessari diversi **pattern d'accesso** alla memoria. Con **access pattern** ci si riferisce allo schema utilizzato per il load (o lo store) degli elementi di una struttura dati, rispetto ai loro indirizzi in memoria.

Con **Stride** ci si riferisce alla distanza che separa gli elementi da caricare (in successione) in un registro. Con **Allineamento** ci si riferisce alla proprietà di un dato di iniziare (in memoria) ad un indirizzo che è multiplo della dimensione indirizzabile (data size). Ad esempio un indirizzo è allineato a 4-byte se è un multiplo di 4 byte. Con **Padding** ci si riferisce alla tecnica usata per memorizzare i dati in modo allineato (si aggiungono elementi "inutili" alla struttura dati per renderne i componenti allineati).

Esistono principalmente due tipologie di stride:

- Unit-Stride (stride=1);
- Non-Unit-Stride (stride > 1, ma costante). Viene utilizzato per accedere a una successione di dati memorizzati in locazioni non consecutive.

Ad esempio:

```

for (i = 0; i < 100; i=i+1) {
    for (j = 0; j < 100; j=j+1){
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1){
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
    }
}

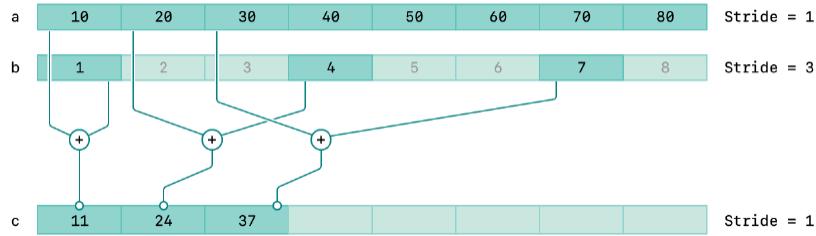
```

Row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$



Ritornando all'esempio delle moltiplicazioni tra matrici, potremmo usare unit stride per B e non-unit stride per D. Si verifica un problema supplementare detto **bank conflict**, che avviene quando (usando non-unit stride) più accessi riguardano lo stesso bank di memoria: se lo stesso memory bank viene acceduto più volte in un intervallo di tempo, più breve del suo bank busy time, allora si origina una sequenzializzazione degli accessi.

1.14.2 SIMD extensions Le SIMD extensions sono estensioni dell'instruction set di un processo e nascono dalla osservazione che solitamente le applicazioni multimediali operano su più dati più "piccoli" della dimensione dei registri macchina, ad esempio dati da 8 bit per rappresentare il colore o audio campionato usando 8 o 16 bit. Si possono usare le FU che manipolano dati da 256bit per compiere simultaneamente più operazioni su insiemi di "dati piccoli", per esempio 16 operazioni su dati da 16 bit o 32 operazioni su dati da 8 bit.

Le SIMD extensions sono caratterizzate dal fatto che il numero di operandi è codificato nel codice della operazione e che non sono utilizzabili indirizzamenti complessi (come non-unit stride).

Le SIMD extensions, rispetto alle architetture vettoriali sono più orientate al programmatore che alla generazione del codice parallelo da parte del compilatore; presentano un costo relativamente basso (semplice modifica delle FU) e sono più semplici da implementare. Necessitano memory bandwidth più bassa delle architetture con vector registers e di meno registri più piccoli (costo del context switch minore). I dati sono allineati e di dimensione fissa (quindi non possono stare a cavallo di pagine di memoria diverse: nessun page fault durante LD/ST). Con le SIMD extensions si hanno meno problemi di caching rispetto alle architetture vettoriali, inoltre è un set di istruzioni estendibile facilmente a nuove operazioni o nuovi standard multimediali (i dati hanno dimensione fissa).

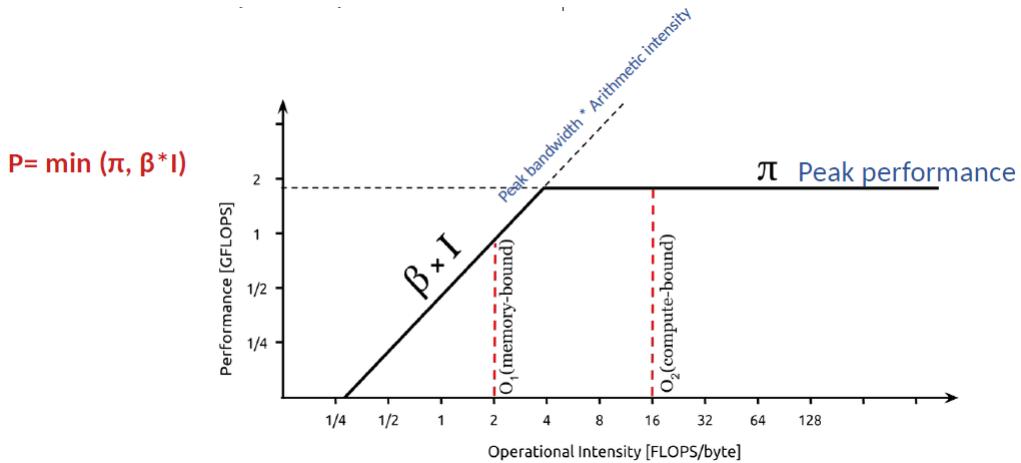
Tuttavia, non tutti gli algoritmi possono essere "vettorizzati" facilmente. Spesso ci sono irregolarità nei dati. Ciò comporta che sia il raccogliere (gathering) i dati nei registri SIMD, sia la dispersione (scattering) degli stessi in memoria non sia banale. Se possibile si opera una regolarizzazione dei dati. Inoltre le architetture SIMD possono imporre vincoli

sull'allineamento dei dati in memoria, come dover utilizzare accesso strided che può introdurre degrado nella performance dell'accesso alla memoria e/o cache; bank conflict. Dato che architetture diverse utilizzano differenti dimensioni dei registri, servono differenti tecniche/strategie di ottimizzazione per ottenere performance portability. La presenza di molti registri di grande dimensione comporta l'aumento di consumo energetico e richiede la fabbricazione di chip di maggiori dimensioni/costo.

1.15 Considerazioni sulle performance

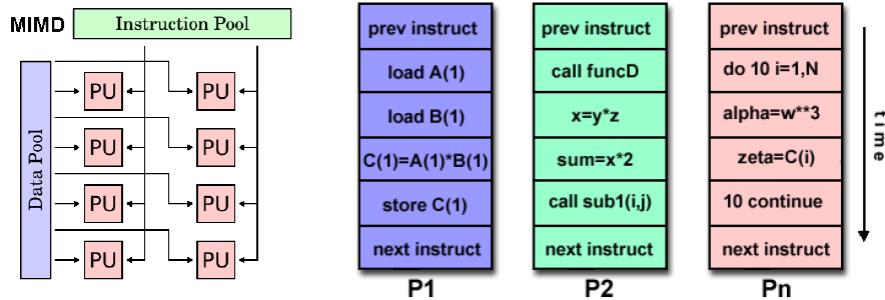
Si può combinare la **floating-point performance** e la **performance della memoria** (bandwidth) relativamente ad uno specifico hardware per valutare la performance. Si può considerare il **peak floating-point throughput** come funzione dell'intensità del carico computazionale (**arithmetic intensity**, ovvero quantità di operazioni floating-point per ogni byte letto.)

1.15.1 Modello Roofline Il modello Roofline visualizza una “stima” della performance ottenibile da un hardware: combina limite dato dalla bandwidth di memoria e il picco di performance raggiungibile dal/i processore/i. Fornisce un modo per comparare architetture.



1.16 MIMD

Le architetture MIMD consentono di sfruttare sia il data-level che il task-level parallelism. Esempi: scientific computing che preveda computazioni con matrici di grandi dimensioni oppure applicazioni di Artificial Intelligence (ad es. training di Neural Networks, etc.). MIMD prevede delle fetch units multiple (più istruzioni per operazione). Nella definizione di MIMD, con MI ci si riferisce a Multiple Instructions: tutte le processing unit eseguono istruzioni diverse, anche nello stesso ciclo di clock. Con MD ci si riferisca a Multiple Data: ogni processing unit può operare su un diverso dato. Si ha sia esecuzione sincrona che deterministica (lockstep) sia esecuzioni asincrone, di conseguenza, nel caso di esecuzioni asincrone spesso risulta necessario un uso esplicito di barriers.

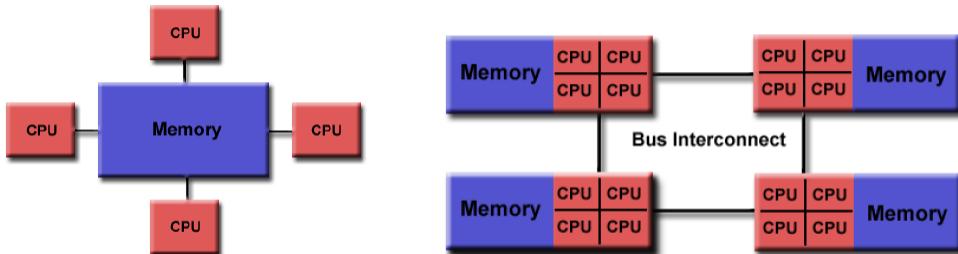


Le principali varianti di architetture MIMD sono i processori many/multi-core (shared-memory) e i sistemi distribuiti (distributed-memory). Le MIMD utilizzano una rete per realizzare le connessioni processori-memoria e memoria-memoria. Quindi è essenziale minimizzare le comunicazioni (in sostanza il numero di accessi alla memoria). Spesso le MIMD includono delle sotto-componenti SIMD.

Due aspetti chiave sono il **thread-level parallelism** per i sistemi con shared-memory e il **message passing** per i sistemi con distributed-memory.

1.16.1 Architetture shared-memory Le caratteristiche generali di una architettura shared-memory sono:

- Più processori operano indipendentemente ma condividono la memoria;
- Modifiche in una locazione di memoria sono visibili a tutti gli altri processori;
- Classificate come UMA e NUMA, con diverso tempo di accesso.



Si parla di Shared-memory uniforme se si hanno processori (pochi) identici che condividono una unica memoria, con latenza uniforme, come spazio di indirizzamento globale e con cache coherent UMA.

Con **Cache Coherency** si intende che se un processore modifica una locazione in shared memory, allora le cache sono gestite in modo che tutti i processori vedano l'update (eventualmente invalidando la cache line obsoleta e ri-accedendo in memoria shared). Viene garantita a livello hardware.

Si parla di distributed-shared memory (DSM) quando più SMP vengono collegati fisicamente. Si ha Non-Uniform memory access (NUMA) quindi la latenza non è uniforme. I

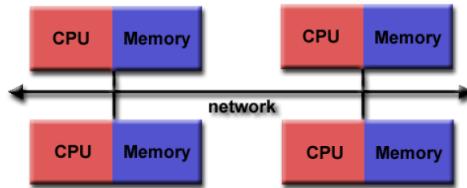
processori sono connessi via una direct (switched) o non-direct (multi-hop) interconenction network. Non tutti i processori hanno uguale tempo di accesso a tutto lo spazio di memoria. Gli accessi che attraversano la rete sono più lenti (si necessita di usare tecniche di computazione "near-core"). Nel caso sia garantita la cache coherency (non sempre vero) si parla di Cache Coherent NUMA.

In generale i pro delle architetture shared memory sono il fatto che la presenza di un global address space consente una visione della memoria più "user-friendly" e che il data-sharing tra tasks offre velocità e uniformità grazie alla prossimità dei processori.

I contro sono la scarsa scalabilità, infatti l'aggiunta di processori può causare una crescita (geometrica) del traffico tra memoria e CPU, inoltre, per sistemi che garantiscono cache coherency accade anche per il traffico associato alla gestione di cache/memoria. Il programmatore ha la responsabilità di utilizzare opportuni costrutti di sincronizzazione per assicurare la correttezza negli accessi alla memoria globale.

1.16.2 Architetture con distributed memory Le caratteristiche di base dei sistemi con distributed-memory sono:

- la necessità di una rete di comunicazione che connetta le porzioni di memoria "inter-processor" (come per i sistemi con shared memory);
- i singoli processori hanno una propria memoria locale;
- non esiste il concetto di global address space comune ai processori, ma gli indirizzi di memoria propri di un procesore non sono mappati sugli altri.



Ogni processore ha la propria memoria locale e opera indipendentemente dagli altri. Modifiche a dati in tale memoria non hanno effetto sulla memoria degli altri processori. Quindi, la nozione di cache coherency normalmente non è presente. Se un processore necessita di accedere a dati nella memoria di un altro processore è delegata solitamente al programmatore la gestione esplicita della comunicazione necessaria (come, quando, in che forma, etc.). Similmente è compito del programmatore implementare la sincronizzazione tra i task. La tecnologia impiegata per realizzare la rete di interconnessione può variare a seconda dei casi.

I principali pro delle architetture con distributed memory sono la scalabilità: la memoria scala proporzionalmente con il numero dei processori (aumentando i processori aumenta la dimensione della memoria del sistema), inoltre ogni processore ha accesso rapido alla sua porzione di memoria senza interferenze o overhead dovuto al mantenimento della cache coherency. È la soluzione effettiva dal punto di vista dei costi: si possono utilizzare componenti comuni, processori e componenti di rete normalmente disponibili sul mercato. Tuttavia ci sono dei contro: il programmatore ha grossa responsabilità per gestire molti aspetti critici

legati alla comunicazione/sincronizzazione tra processori. Può non essere facile “mappare” in questo tipo di architettura le strutture dati comunemente sviluppate/utilizzate in sistemi con uno spazio di memoria globale. Nel progetto di programmi si deve tenere presente che accedere ai dati che risiedono su altri nodi comporta tempi di accesso più alti rispetto agli accessi locali.

1.17 Alcuni paragoni

Considerando il meccanismo di comunicazione “nativo” supportato dal sistema, abbiamo:

- Shared-memory: accesso diretto alle locazioni di memoria, si ha comunicazione implicita;
- Distributed-Memory: è necessario spedire (send) messaggi attraverso la rete per accedere/scambiare dati, si ha comunicazione esplicita.

Si noti che usualmente i sistemi con shared memory utilizzano comunque della comunicazione basata su scambio messaggi al fine di gestire la sincronizzazione delle cache. Si noti anche che è possibile introdurre opportune astrazioni e implementare:

- sistema con shared memory su un sistema con distributed memory
- sistema con distributed memory su un sistema con shared memory

SIMD vs MIMD:

- Le architetture SIMD tendono a richiedere meno HW di paragonabili MIMD: necessitano di meno hardware di controllo (control unit);
- Solitamente i processori SIMD tendono ad aver più lunga fase di progetto e maggiori costi realizzativi;
- Non tutte le applicazioni sono facilmente adattabili a SIMD;
- al contrario, le piattaforme che supportano il paradigma MIMD possono essere realizzate a relativamente basso costo usando componenti comuni e con basso costo/tempo di progetto/realizzazione.

1.18 MISD

Le architetture MISD non sono diffuse, seppur rappresentando una tipologia di architettura parallela prevista dalla Tassonomia di Flynn.

- Multiple Instruction: ogni processing unit opera su dati indipendentemente dalle altre unit e eseguendo uno stream di istruzioni proprio;
- Single Data: è presente un unico data stream processato da tutte le processing unit.

2 Graphics Processing Unit

Una Graphics Processing Unit (GPU) anche detta Video Processing Unit (VPU) è un device in grado di accelerare la creazione/processamento di un'immagine da visualizzare su un display. Il termine GPU è stato introdotto da Nvidia nel 1999 quando propose la GeForce 256 definendola “The world’s first GPU”. Al giorno d’oggi si incontrano frequentemente questi termini:

- GPU computing o GPGPU: è l’uso di GPU per svolgere general purpose computing, ovvero computazioni scientifiche, o ingegneristiche, etc. non necessariamente connesse al graphic processing.
- Heterogeneous Computing: è l’utilizzo, in modo trasparente, di diversi device computazionali per compiere general purpose computing.

Ma perchè usare le GPU? Perchè risultano più economiche ed efficienti di altre architetture parallele. Consentono di utilizzare thread-level parallelism per risolvere disparati problemi.

2.1 Thread-Level Parallelism

Le architetture MIMD adottano thread-level parallelism oppure Message Passing, focalizziamoci sul primo: In generale, il thread-level parallelism:

- comporta la presenza di program counter multipli;
- orientato alle architetture cosiddette “tightly-coupled/shared-memory multiprocessors”;
- per sfruttare gli N processori si devono eseguire (almeno) N thread.

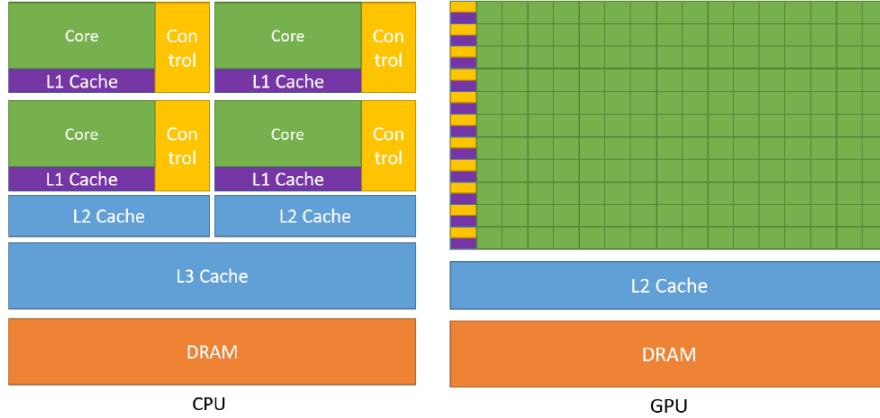
Con **grain size** ci si riferisce alla mole di lavoro/computazione assegnata ad ogni thread (si parla di fine-grained o coarse-grained parallelism).

Soltanente si ha un set di thread tightly-coupled che risolvono lo stesso task computando in parallelo. I thread possono anche essere sfruttati per realizzare data-level parallelism, ma è necessario prestare attenzione all’eventuale overhead rispetto ad una soluzione SIMD.

2.2 GPU vs CPU MIMD

Le CPU hanno tipicamente una alta frequenza di clock, cache di grande dimensione, controllo complesso, ALU potenti e complesse. Le GPU, contrariamente tendono ad avere un clock inferiore, cache più piccola, controllo semplificato, molte ALU energeticamente efficienti. Si dice che la CPU ha un design orientato alla latenza, mentre le GPU al throughput.

Un approccio generale potrebbe essere usare le CPU per le parti sequenziali, ove la latenza è rilevante: le CPU possono esibire efficienza 10x rispetto alle GPU se il codice è sequenziale e usare le GPU per le parti parallele, ove il throughput è rilevante: sul codice parallelo le GPU sono 10x (o più) veloci delle CPU.



Entrambi sono thread-level parallelism, ma in modo differente:

- CPU: progettata per eseguire sequenze di operazioni, cioè thread, il più velocemente possibile; può eseguire qualche decina di thread in parallelo;
- GPU: progettata per eseguire migliaia di thread in parallelo; ogni thread esibisce singolarmente una “bassa” performance, ma ciò è ammortizzato dal fatto che il loro numero elevato permette alto throughput. L’architettura delle GPU è detta Single Instruction Multiple Thread.

2.3 Architettura Nvidia Volta

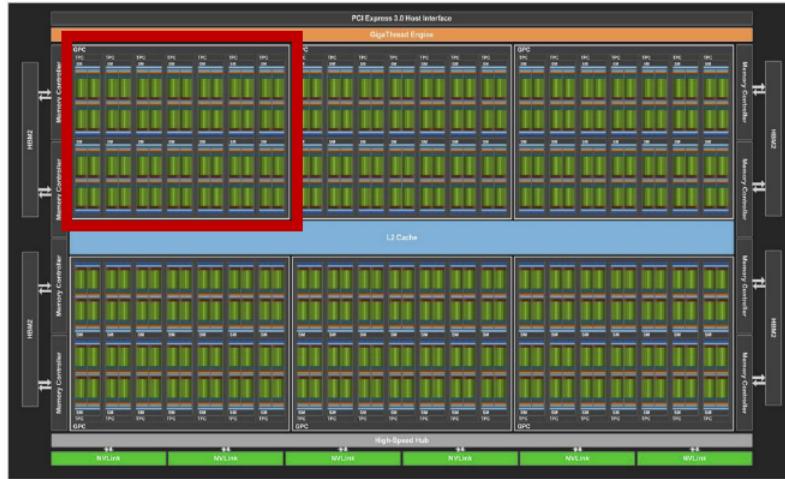
Volta è il nome in codice di una microarchitettura GPU sviluppata da Nvidia, successiva a Pascal. È stata annunciata per la prima volta su una tabella di marcia nel marzo 2013, sebbene il primo prodotto non sia stato annunciato fino a maggio 2017. L’architettura prende il nome dal chimico e fisico italiano del XVIII-XIX secolo Alessandro Volta. È stato il primo chip di Nvidia a presentare Tensor Core, core appositamente progettati che hanno prestazioni di deep learning superiori rispetto ai normali core CUDA. L’architettura è prodotta con il processo FinFET a 12 nm di TSMC. La microarchitettura Ampere è il successore di Volta.

2.3.1 GV100

Il chip GV100 è orientato ad applicazioni computazionalmente intense, come applicazioni di deep learning in IA. Similmente a molte altre GPU, una GV100 è composta da serie di GPU Processing Clusters (GPC), di Texture Processing Clusters (TPC), di Streaming Multiprocessors (SM), e da memory controllers. L’architettura GV100 prevede:

- 6 GPC, ognuno dei quali possiede:
 - 7 TPC (ciascuno con 2 SM);
 - 14 SM
- 8 memory controller a 512-bit (4096 bit totali);
- cache L2 tra GPC.

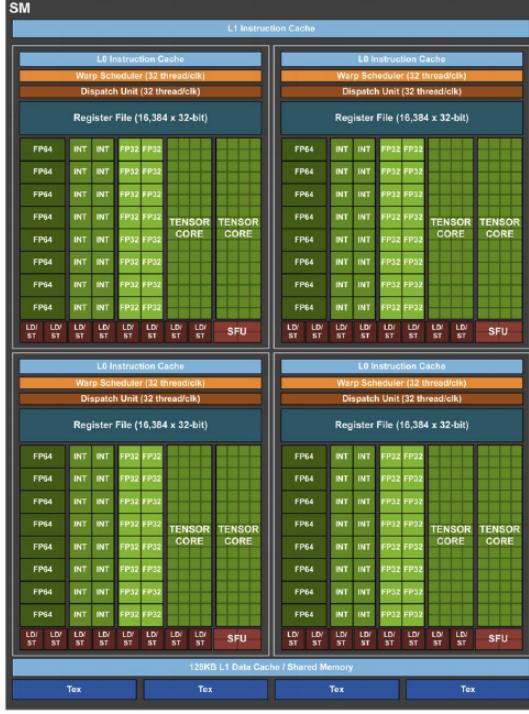
In totale: 5376 FP32 core, 5376 INT32 core, 2688 FP64 core, 672 Tensore Core, 336 Texture unit, 6144 KB di cache L2. Ogni HBM2 DRAM ha due controller. La Tesla V100 è un esempio di scheda che utilizza 80SM del chip GV100.



Struttura di uno Stream Multiprocessor (SM)

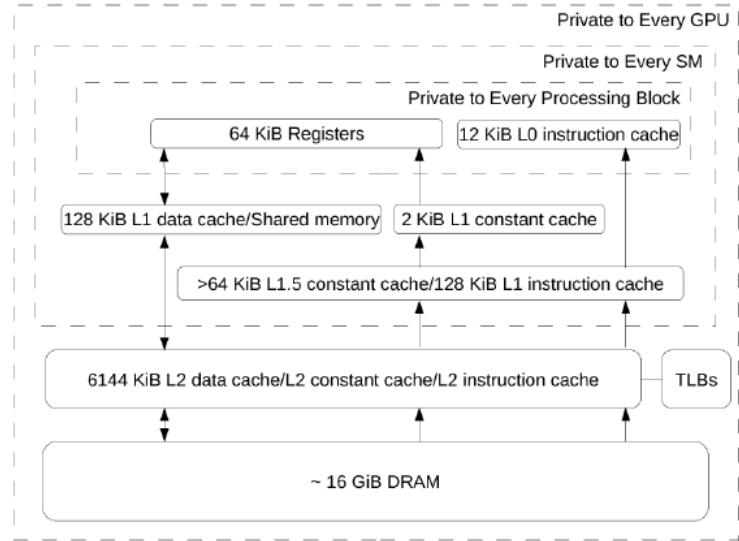
Gli stream multiprocessor sono la componente centrale di una GPU Nvidia, nel caso della GV100, ciascun SM comprende:

- 64 FP32 core
- 64 INT32 core
- 32 FP64 core
- 8 Tensore core
- 4 Texture unit
- 8 registri LD/ST
- 1 SFU (Special Function Unit)
- 16384 (x4) registri a 32-bit
- L1 Data Cache/Shared Memory



2.3.2 Gerarchia di memoria della architettura Volta

La main memory (VRAM) è tipicamente da 32 o 16 GB, con interfaccia 4096-bit HBM2, memory data-rate di 877.5 Mhz DDR e Bandwidth di 900GB/s. Inoltre in totale è presente memoria cache L2 per un totale di 6144KB, 96KB di Shared Memory per SM (L1 cache), registri per SM per un totale di 256 KB, e L0 istruzione cache.

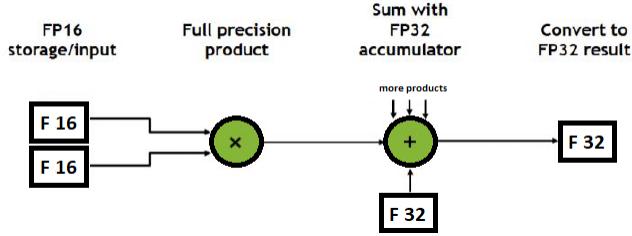


2.3.3 Tensor Core

Ogni tensor core opera su matrici floating-point ed è in grado di eseguire l'operazione $D = A * B + C$ dove A, B, C, D sono matrici 4x4. L'operazione Matrix-Matrix (GeMM) è una operazione fondamentale in molte applicazioni/algoritmi aritmeticamente intensi (ad esempio reti neurali, passi di inferenza, etc.).

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

I Tensor Cores operano su input FP16 e producono FP32. Il prodotto di due matrici di input FP16 produce un risultato “full precision” FP32, che è possibile sommare a una matrice FP32:



Quindi combinando più operazioni con i Tensor Core si possono implementare operazioni matriciali su matrici più ampie o in più dimensioni.

2.4 Alcune definizioni

Elenchiamo alcune definizioni utili nell'ambito delle GPU:

- **Platform model:** è il modello che definisce (astrae) le unità di calcolo: una o più GPU connesse e gestite da un host (la CPU);
- **Execution model:** definisce il set di istruzioni che devono essere eseguite da (i kernel della) GPU e il set di istruzioni per inizializzare e controllare l'esecuzione (host program) dei kernel;
- **Memory model:** definisce le tipologie di memoria e i modi in cui GPU e host accedono ad esse;
- **Programming model:** definisce la logica della computazione parallela, delle unità di computazione eseguite, dati, task;
- **CUDA (Compute Unified Device Architecture):** descrive Platform Model e Programming Model per programmare le GPU Nvidia con un approccio “general-purpose parallel computing”. L'obiettivo è quello di supportare lo sviluppo di applicazioni che utilizzino sia multi-core CPU che many-core GPU in modo da assicurarne la scalabilità, in modo trasparente rispetto al crescere del numero di core.

2.5 CUDA in a nutshell

CUDA si basa su tre astrazioni fondamentali, ovvero:

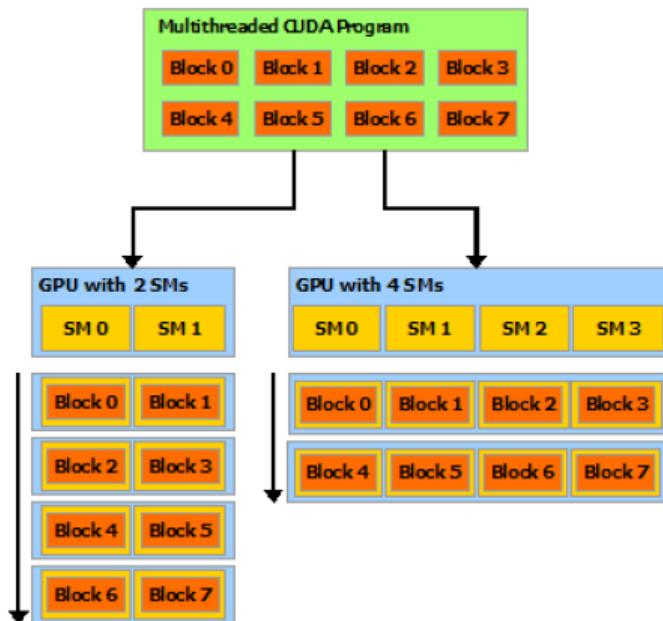
- gerarchia tra i (gruppi di) thread;
- memorie condivise;
- barrier synchronization.

Sono immediatamente accessibili al programmatore nella forma di estensioni del linguaggio di programmazione (general purpose, C/C++, Fortran, etc.). Sfruttando tali astrazioni si ha accesso, combinato, sia a data-parallelism di tipo fine-grained ed a thread-parallelism, sia a data-parallelism di tipo coarse-grained ed a task parallelism: supportano il programmatore nel partizionare il problema e identificarne i sotto-problemi che possono essere risolti indipendentemente tramite blocchi di thread, e successivamente le sottoparti di ogni sotto-problema che possono essere affrontate in modo cooperativo dai thread di un blocco.

I thread cooperano nella soluzione di un problema e, al contempo, garantiscono la scalabilità. Ogni thread esegue una istanza di un kernel su una porzione di dati.

- I thread sono organizzati in blocchi di thread (blocks). I blocchi sono numerati e, internamente ad ogni blocco, i thread sono a loro volta numerati;
- I blocchi possono essere schedulati (ed eseguiti) su uno qualsiasi degli SM liberi, in qualsiasi ordine, concorrentemente o sequenzialmente;

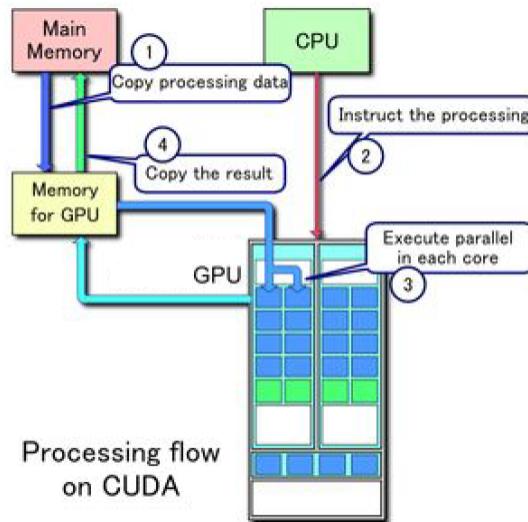
Quindi un programma CUDA può eseguire su un qualsiasi numero di SM (solamente il runtime system ha percezione degli effettivi SM fisici disponibili sulla GPU).



2.5.1 CPU/GPU CUDA Workflow

Il tipico workflow di una applicazione CUDA prevede questi passi fondamentali:

1. la CPU (host) copia i dati di input dalla memoria centrale alla memoria della GPU;
2. la CPU inizializza/lancia i kernel per la GPU;
3. i core della GPU (device) eseguono i kernel in parallelo;
4. la CPU copia i risultati dalla memoria della GPU alla memoria centrale.



Come detto, le GPU sono costituite da un set (scalabile) di multithreaded Streaming Multiprocessors (SM). Quando in un programma CUDA l'host invoca una grid di (blocchi di) kernel, i blocchi della grid vengono numerati e distribuiti tra gli SM. I thread di un blocco eseguono tutti concorrentemente su un multiprocessor. Più thread blocks possono eseguire concorrentemente sullo stesso multiprocessor. Quando dei thread blocks terminano, nuovi blocchi di thread vengono lanciati sui multiprocessors che si sono liberati.

Ogni multiprocessor è progettato per eseguire centinaia di thread, concorrentemente. Per gestire tale elevato numero di thread si impiega l'approccio SIMT: Le istruzioni vengono pipelined, sfruttando così l'instruction-level parallelism interno ad ogni singolo thread. Al contempo si sfrutta il multithreading di livello HW per il thread-level parallelism. Diversamente da ciò che accade con i core delle CPU, l'esecuzione è in-order e non si adottano tecniche quali branch prediction o speculative execution.

CUDA: Architettura SIMT

Ogni SM crea, gestisce, schedula e esegue threads in gruppi di **warp-size** (=32) thread paralleli, detti **warp**. Ogni thread dello stesso warp inizia dallo stesso indirizzo nel programma (stessa istruzione). Tuttavia, ogni thread ha il proprio program counter e register state. Quindi ogni thread è libero di divergere dagli altri thread, cioè effettuare branch ed eseguire indipendentemente.

Si parla di half-warp riferendosi alla prima o la seconda metà del warp (16 thread); un quarter-warp è invece una delle quattro metà. I thread di un warp che stanno partecipando alla istruzione corrente sono active thread, i restanti (del warp) sono inactive (disabled). I thread possono essere inactive per vari motivi. Ad esempio:

- hanno terminato l'esecuzione prima degli altri;
- hanno intrappreso un diverso branch path diverso da quello eseguito dai thread active del warp;
- sono gli ultimi thread di un block composto da un numero di thread che non è multiplo del warp size.

Quando ad un SM vengono assegnati uno o più blocks da eseguire, lo SM li partiziona in warp e ogni warp viene schedulato (warp scheduler) per essere eseguito. Il partizionamento di un blocco avviene sempre nello stesso modo: ogni warp contiene threads con ID consecutivi e in ordine crescente; il primo warp contiene il thread con ID=0. La gerarchia dei thread descrive come gli ID dei thread siano associati agli indici dei thread nei blocks.

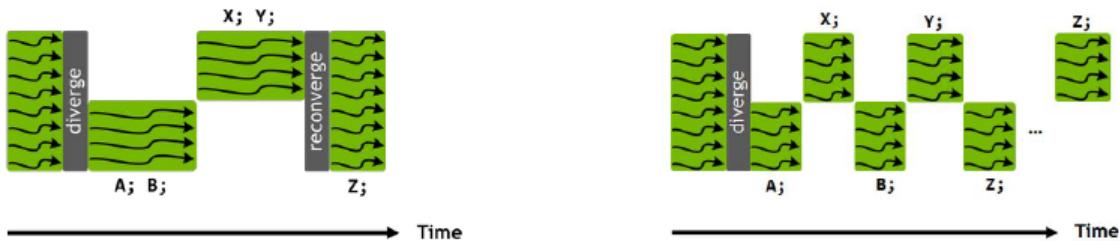
Un warp (i thread di un warp) esegue una istruzione alla volta, quindi la massima efficienza si raggiunge quando tutti i 32 thread del warp concordano sull'execution path. Se i thread di un warp divergono a causa di data-dependent conditional branch, allora il warp esegue ogni branch intrapreso da parte dei thread, disabilitando di volta in volta i thread che non hanno preso quel branch.

Branch divergence

Il **branch divergence** occorre solo internamente ai warp: warp diversi eseguono indipendentemente senza vincoli dipendenti dal path che seguono.

La causa più frequente di thread divergence sono i branch introdotti dai costrutti condizionali, come if-then-else. Se alcuni threads in un warp valutano 'true' la condizione di branch mentre altri la valutano 'false', allora i 'true' e i 'false' thread eseguiranno istruzioni diverse.

Durante l'esecuzione di un if-then-else, la GPU schedula prima i warp che eseguono la parte 'then' e poi quelli che eseguono la parte 'else'. Quindi mentre si esegue la parte 'then' tutti i thread della parte 'else' vengono disattivati, e viceversa. In architetture pre-Volta (ad es Pascal) le parti 'then' e 'else' sono eseguite in serie, non in parallelo. Nella Volta le due parti possono essere interleaved



Un esempio di codice che origina branch divergence:

```

__global__ void odd_even(int n, int* x) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( i % 2 == 0 ) {
        x[i] = x[i] + 1;
    } else {
        x[i] = x[i] + 2;
    }
}

```

2.6 SIMT vs SIMD

Le architetture SIMT somigliano alle SIMD, nel senso che la stessa istruzione controlla più processing unit. Ma ci sono delle differenze:

1. SIMD espone le caratteristiche SIMD a livello del software, mentre in SIMT le istruzioni specificano l'esecuzione e il comportamento di un singolo thread;
2. SIMT permette al programmatore di scrivere codice thread-level parallelo per thread scalari indipendenti e al contempo permette di scrivere codice data parallel per set di thread coordinati;
3. In SIMT, la cache line size può essere ignorata nella fase di progetto (ma deve comunque essere considerata quando si punta alla massima performance). Nelle architetture vettoriali, invece, i load nei vettori devono essere coalescenti e la divergenza deve essere gestita manualmente.

2.7 Thread Scheduling

La politica di scheduling dei thread chiaramente impatta sulla efficienza del Thread-Level Parallelism. Volta introduce uno scheduling più efficace rispetto alle GPU precedenti: Si ricordi che i thread di un warp eseguono contemporaneamente la stessa istruzione, eventualmente i thread inattivi del warp vengono mascherati (active mask). I thread dello stesso warp che divergono (eseguono in regioni di codice diverse) non possono comunicare scambiandosi dati e un eventuale coordinamento che utilizzi lock o mutex potrebbe portare a deadlock.

2.7.1 Independent Thread Scheduling (Volta)

In riferimento all'impossibilità di comunicare, l'architettura Volta migliora questa situazione tramite l'Independent Thread Scheduling (ITS) che permette una completa concorrenza tra tutti i thread, indipendentemente dal warp di appartenenza. Per ottenere ciò, la GPU mantiene l'execution state per ogni thread, includendo il program counter e il call stack del thread. Ciò consente una granularità di esecuzione per-thread, massimizzando l'uso delle risorse disponibili e permettendo, ad esempio, ad un singolo thread di attendere dati prodotti da un altro thread. Un ottimizzatore di schedule determina opportunamente come raggruppare i thread attivi (dello stesso warp). I thread possono quindi divergere e convergere anche con granularità sub-warp.

2.8 Multithreading Hardware

L'execution context (program counter, registri, etc.) di ogni warp processato da uno SM è mantenuto on-chip durante l'intero lifetime del warp (anche pre-Volta, come detto nella Volta inoltre, il contesto è gestito per-thread). Quindi il thread execution context switch ha costo nullo! La sua gestione è HW, usando i registri che vengono partizionati tra i warp. La cache e la shared memory sono invece partizionate tra i blocks. Lo scheduler seleziona un warp che abbia thread ready, per eseguire la prossima istruzione, e avvia l'esecuzione della istruzione in tali thread. L'overhead di gestione dei thread è quindi ridotto e ciò incrementa lo throughput globale. Il numero di block e warp che possono essere residenti e processati insieme da uno SM, per un dato kernel, quindi dipende dalla quantità di registri e di shared memory usate dal kernel rispetto a quelle disponibili sullo SM. Esiste un limite massimo al numero di blocks/warp residenti in un multiprocessor. Il limite dipende dalla Compute Capability della GPU. Se non vi sono abbastanza registri/shared-mem disponibili in almeno un SM, in modo che si possa gestire almeno un block, il launch dell'intera grid di kernel fallisce.

3 CUDA Introduction

3.1 CUDA Programming Model

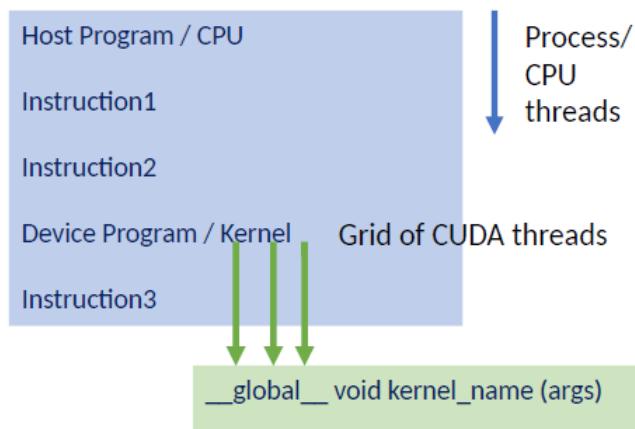
Il concetto di fondo alla base del modello di programmazione CUDA si basa sul fatto che:

”Le porzioni parallele di una applicazione possono essere delegate alla GPU”

Vengono distinte due porzioni di codice:

- **host program**: parte di codice eseguito sulla CPU;
- **device program**: parte di codice eseguito sulla GPU;

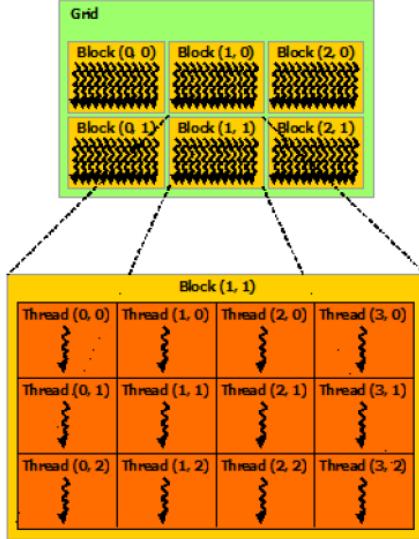
Ogni istanza del device program viene eseguita da una **grid di thread**. Questa istanza si chiama CUDA Kernel. I thread eseguono sequenzialmente le istruzioni del Kernel fornito.



Ogni thread che esegue il kernel ha un ID unico ed è accessibile dal codice del kernel tramite variabili predefinite (built-in variables). I thread sono identificati tramite i valori di un vettore a 3 componenti: **threadIdx**.

Quindi i thread si possono individuare tramite un indice 1D, 2D, oppure 3D, e formano un **block** di thread, 1D, 2D oppure 3D.

Thread dello stesso block condividono le stesse risorse (SM). I blocks sono a loro volta organizzati in una griglia (grid) 1D, 2D, oppure 3D. Il numero di thread blocks in una grid viene di solito stabilito considerando il numero di dati da processare.



Esempio di grid 2D, con blocchi 2D

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

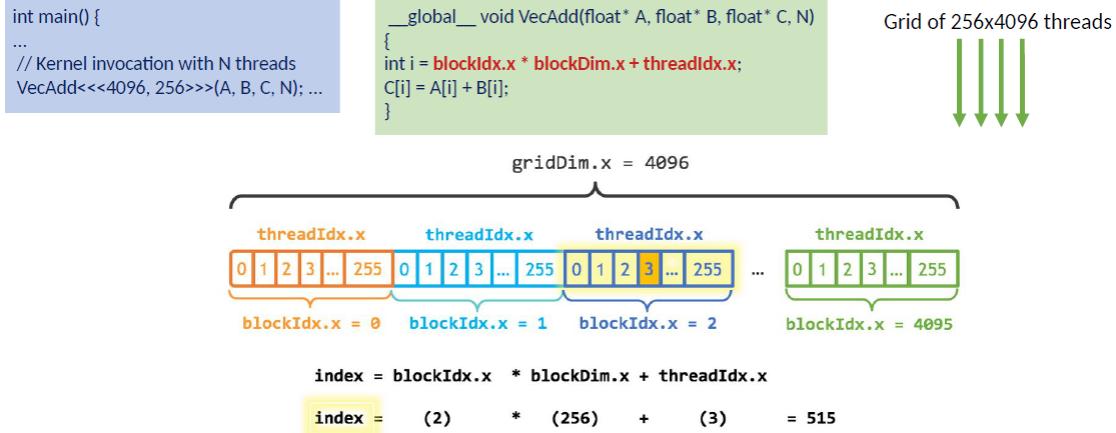
Il numero di thread-per-block e il numero di blocks nella grid viene specificato al momento del lancio tramite una sintassi che estende quella di C (usando la sintassi `<<< ... >>>` e indicando dei valori di tipo int o dim3).

Ogni block è identificato, internamente al codice del kernel, da un indice 1D, 2D o 3D accessibile tramite la variabile predefinita `blockIdx`. Inoltre, la variabile `blockDim` indica la dimensione del blocco. Esempio: in una dimensione, x, si hanno:

`blockIdx.x`: identifica il blocco nella grid 1D

`blockDim.x`: è la dimensione x del blocco

`threadIdx.x`: è l'ID del thread internamente al blocco



La sequenza lineare di thread in un blocco viene poi suddivisa in warp, blocco per blocco. I thread possono cooperare/comunicare condividendo dati attraverso le varie tipologie di memoria disponibili sulla GPU e devono quindi sincronizzarsi per coordinare le loro operazioni.

Gerarchia di memorie:

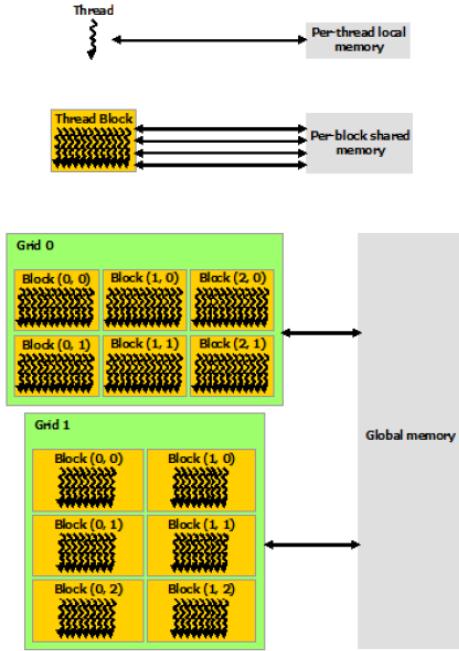
- Ogni thread ha una memoria locale (SM registers e local memory);
- Ogni block ha una porzione di shared memory visibile a tutti i suoi thread (L1/shared memory);
- Tutti i thread hanno accesso alla stessa global memory (Video RAM);
- Esistono anche due memorie read-only: constant memory e texture memory.

Sincronizzazione e barriers:

- I thread di un block possono sincronizzarsi esplicitamente (`_syncthreads()`);
- Cooperative Groups programming model: descrive come utilizzare pattern di sincronizzazione con granularità diversa dal blocco;
- A livello di grid/kernel si può imporre punti di sincronizzazione esplicita tra host e device (`cudaDeviceSynchronize()`, ...).

Ricordando che il CUDA programming model consente la programmazione etereogenea combinando host e device, possiamo dire che:

- I thread CUDA eseguono su un device fisicamente separato dal device su cui opera il processore, che sta eseguendo il codice host (C/C++);
- Host e device mantengono i propri spazi di indirizzamento in DRAM: detti host memory e device memory, rispettivamente;



- Il programma host gestisce le memorie global, constant e texture, visibili anche ai kernel, tramite opportune call al CUDA runtime support (per allocazione, trasferimenti di data, deallocazione);

Le GPU più recenti supportano anche la nozione di Unified Memory. Permette di realizzare una corrispondenza tra gli spazi di memoria host e device.

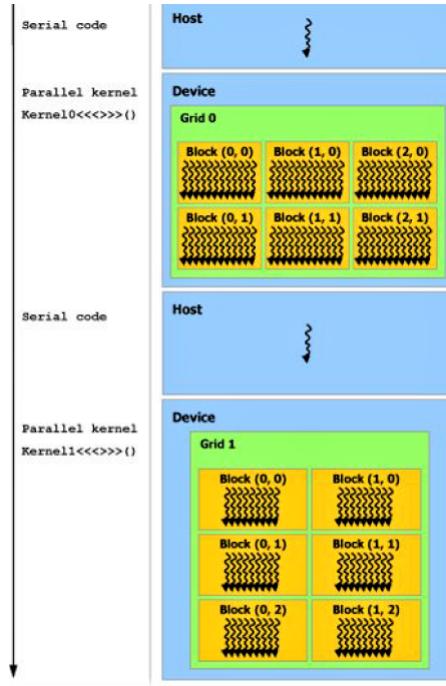
3.2 Compilation workflow

Il compilatore del codice host e codice device di cuda è **nvcc**:

- compila il codice device in una forma di assembler (PTX code) oppure in una forma binaria (cubin - CUDA binary);
- modifica il codice host in corrispondenza delle call <<< ... >>> introducendo la necessaria call alla CUDA runtime function che causa il caricamento e il lancio del kernel compilato, poi richiama il compilatore C/C++ per compilare il codice host.

PTX: Just-in-time Compilation

- il codice PTX caricato dalla applicazione al runtime viene compilato in codice binario dal device driver;
- si può specificare il binary code per la specifica architettura.



3.3 Esempio codice CUDA in C

Alcune note sul codice di esempio:

- `__global__` istruisce il compilatore che la funzione miokernel è un kernel che verrà eseguito dal device, quindi si richiede la generazione del codice di device. Una funzione `__global__` può essere invocata dall'host ed esegue sul device.
- Similmente `__host__` indica il codice di host (è implicito se omesso).
- Le funzioni `__device__` (non usato in questo esempio) sono funzioni invocate dal device che eseguono sul device.
- Possibile combinare `__host__` e `__device__`.

```
// file esempio tantiprint.cu
#include <stdio.h>

__global__ void miokernel( void ) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    printf( "Sono il thread %d!\n", tid );
}

int main( void ) {
    miokernel<<<2,32>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

3.4 CUDA runtime

Il runtime di CUDA è implementato tramite la cudart library e gestisce memoria, esecuzione dei kernel, errori etc. Sono possibili diverse gestioni della memoria: shared memory, page-locked host memory (permette un overlap tra esecuzione dei kernel e trasferimenti dati host-device). Diverse esecuzioni dei kernel: sincrona (in-order) ed esecuzione asincrona concorrente (task-parallelism).

Elenchiamo i concetti base del CUDA runtime:

- Il CUDA programming model assume il sistema come composto da un host e un device;
- Host e device hanno ognuno la propria memoria (disgiunte);
- Il CUDA runtime fornisce funzioni per allocare, deallocare, copiare la device memory, e funzioni per trasferire dati tra memoria host e memoria device;
- La device memory può essere allocata come linear memory o come CUDA array.

La linear memory viene allocata con chiamate a `cudaMalloc()` e disallocata con `cudaFree()`. I trasferimenti tra host e device (e tra device e device) sono eseguiti tramite opportune chiamate a `cudaMemcpy()`.

3.4.1 Tipologie di memoria del device: visibilità e lifetime

- Global Memory (bandwidth di circa 900GB/sec)
- L2 Cache
- L1 DataCache/Shared Memory (per SM)
- L1 Instruction cache
- L0 Instruction cache (per warp)
- Registri

Le varie tipologie differiscono per gli aspetti legati a: allocazione, inizializzazione, lifetime e scope. Similmente alle dichiarazioni sulle funzioni, le dichiarazioni sui dati sono: `_shared_`, `_device_`, `_constant_`.

Global Memory

La linear memory viene allocata con chiamate a `cudaMalloc()` e disallocata con `cudaFree()`. I trasferimenti tra host e device sono eseguiti tramite `cudaMemcpy()`. Esistono delle alternative. Per esempio per i casi in cui si vuole introdurre un particolare padding al fine di garantire l'allineamento richiesto per gli array 2D o 3D. In tal caso per l'allocazione si usa `cudaMallocPitch()` e `cudaMalloc3D()`.

La memoria allocata con `cudaMalloc()`, `cudaMallocPitch()`, etc. è visibile a tutti i thread di tutte le grid.

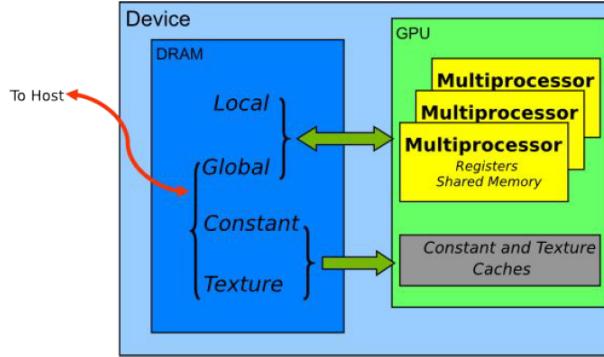


Table 1. Salient Features of Device Memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

Due modalità di allocazione della global memory lineare: statica o dinamica:

- **Static allocation:** implica che la allocazione venga fatta dal codice host (gli esempi precedenti);
- **Dynamic allocation:** implica che l'allocazione della global memory venga fatta dal kernel.

Nel caso di allocazione e disallocazione dinamica di un blocco di memoria, queste operazioni vengono fatte nello heap in global memory, da parte del kernel.

```
__host__ __device__ void* malloc(size_t size);
__device__ void * __nv_aligned_device_malloc(size_t size, size_t align);
__host__ __device__ void free(void* ptr);
```

- malloc() eseguito in kernel alloca size byte in global memory (restituisce un puntatore nello heap del thread allineato ai 16byte, o NULL in caso di errore).
- __nv_aligned_device_malloc() alloca ad un indirizzo multiplo di align
- free() libera la memoria allocata dinamicamente

La memoria allocata dinamicamente non si può gestire con le funzioni usate per gestire quella statica (ad esempio la dinamica non si può deallocare con cudaFree() e viceversa).

Lo heap ha dimensione prefissata ma modificabile tramite
cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)

È possibile copiare i dati dalla memoria globale (dinamica) indirizzata da un puntatore a una zona della memoria globale indirizzata da un altro puntatore:

```
__host__ __device__ void* memcpy(void* dest, const void* src, size_t size);
```

Similmente a quanto si può fare in C, si può assegnare un valore a tutti i byte di un blocco di memoria dinamica (si ricordi che il byte value è interpretato come unsigned char):

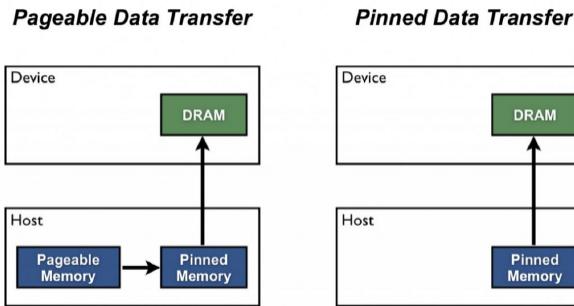
```
__host__ __device__ void* memset(void* ptr, int value, size_t size);
```

3.4.2 Tipologie di trasferimento nella global memory

Esistono quattro tipologie di trasferimento per la global memory:

- Pageable memory
- Page-Locked (o pinned)
- Write-combined memory
- Mapped memory

Partiamo analizzando le prime due.



La memoria host è pageable (paginata) e il Sistema Operativo può effettuare lo swapping delle pagine. Le allocazioni di memoria host sono pageable di default (malloc()) quindi il S.O. può spostarle su disco (swap-out). Il trasferimento DMA da/a GPU non può accedere direttamente ai dati in pageable host memory. Il trasferimento dei dati da/a GPU richiede che prima le pagine siano (temporaneamente) allocate in modo page-locked (o, pinned). Si può richiedere esplicitamente che la allocation sia permanentemente di tipo pinned memory allocation (Ma questa opzione è da usare con cautela: impatta sulle performance dell'host).

CUDA fornisce funzioni per utilizzare memoria page-locked:

- cudaHostAlloc() e cudaFreeHost() allocano e disallocano page-locked host memory
- cudaHostRegister() permette il page-lock di un blocco di memoria allocata con malloc()

I benefici sono:

- Le copie tra memoria host page-locked e memoria device possono essere eseguite contemporaneamente all'esecuzione dei kernel;
- page-locked host memory può essere mappata sullo spazio di indirizzamento del device, eliminando la necessità di effettuare copie da host a device o viceversa;
- La bandwidth migliora se la memoria è page-locked.

Analizziamo ora il caso della modalità **Write-combined memory**. Si osservi che di default la memoria host page-locked viene allocata come cachable. È però possibile allocarla come write-combined (o, write-combining memory) usando il flag `cudaHostAllocWriteCombined` alla funzione `cudaHostAlloc()`.

La memoria write-combined libera (non usa) le cache host L1 e L2, rendendo così disponibile più memoria cache al resto delle applicazioni. Inoltre, sulla write-combining memory non si effettua snooping (controllo di coerenza) e si verifica che il trasferimento attraverso in PCIe bus può migliorare anche del 40%.

Quando usarla: ovviamente la lettura di write-combining memory da parte dell'host è più lenta; in generale, la write-combining memory dovrebbe essere usata in situazioni in cui l'host effettua solo (o, prevalentemente) scritture.

L'ultimo caso è quello della **Mapped memory (zero copy)**. Partiamo da un'osservazione: un blocco di memoria host page-locked può essere anche mapped su una porzione dello spazio di indirizzamento del device. Ciò permette di evitare di gestire tale blocco dal lato device.

Un tale blocco di memoria ha quindi, in generale, due indirizzi: uno in host memory, che viene restituito da `cudaHostAlloc()` o `malloc()`, e uno su device che viene prima recuperato usando `cudaHostGetDevicePointer()` e poi utilizzato per accedere al blocco dall'interno dei kernel.

Vantaggi:

- non c'è bisogno di allocare un blocco in memoria host e un blocco in memoria device e poi effettuare le copie tra un blocco e l'altro;
- i trasferimenti di dati sono implicitamente eseguiti quando necessari;
- non c'è bisogno di usare gli stream per sovrapporre trasferimenti di dati e kernel execution;
- i dati generati dal kernel sono trasferiti automaticamente in contemporanea alla esecuzione del kernel.

Esempio:

```
int *a, *b, *c;           // host pointers
int *dev_a, *dev_b, *dev_c; // device pointers to host memory

. . .

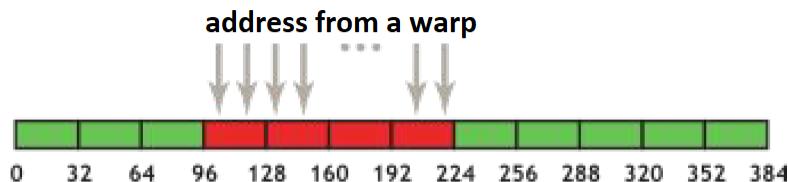
// determine device-side pointers to zero-copy data
cudaHostGetDevicePointer(&dev_a, a, 0);
cudaHostGetDevicePointer(&dev_b, b, 0);
cudaHostGetDevicePointer(&dev_c, c, 0);
```

Modifiche ad a, b, c, copiano implicitamente I dati su dev_a, dev_b, dev_c e viceversa. Si noti che: operazioni su mapped page-locked memory non sono atomiche dal punto di vista dell'host o di altri device.

Ottimizzare l'accesso del device alla global memory

I load e store sulla global memory da parte di thread dello stesso warp vengono “coalesced” (coalizzato, o coalescente) dal device eseguendo il minor numero di transazioni. Gli accessi concorrenti da parte dei thread dello stesso warp vengono coalizzati in un numero di transazioni uguale al numero di transazioni da 32-byte necessarie per servire tutti i thread del warp (per compute capability almeno 6.0). Se L1-caching è abilitato, il numero di transazioni richieste è uguale al numero di segmenti di dati richiesti allineati a 128-byte (cioè 32 integer).

Caso in cui il k-esimo thread accede la k-esima word in un array allineato a 32-byte. Per esempio, i thread di un warp accedono word da 4 byte, word adiacenti. Quattro transazioni coalizzate da 32-byte serviranno tutte le richieste di accesso:

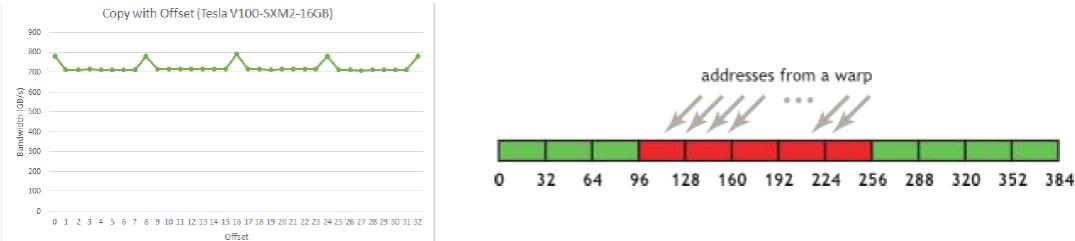


Una buona regola è utilizzare blocchi con dimensione multipla si warp-size (32).

Si parla di accessi miss-aligned quando i thread accedono a dati non allineati a 32-byte: vengono trasferiti 5 segmenti (invece di 4). Supponiamo che questo kernel sia eseguito da un host con offset che varia da 0 a 32.

A copy kernel that illustrates misaligned accesses

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```



Nel caso di accessi strided alla global memory (con non-unit stride): consideriamo questo kernel lanciato da host con stride=2.

A kernel to illustrate non-unit stride data copy

```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```



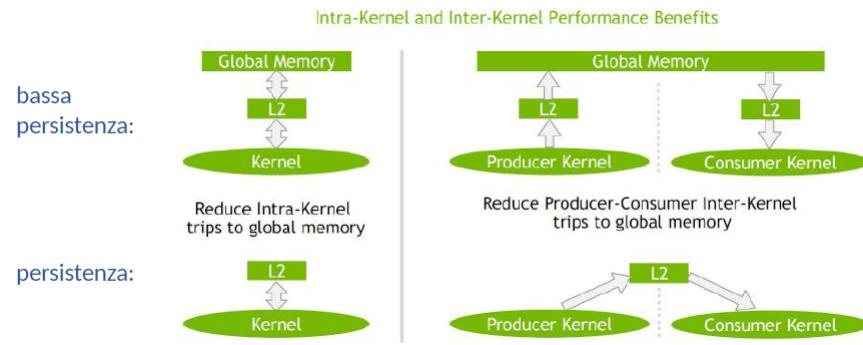
[Youtube Video: Coalesced Memory Access](#)

3.4.3 Gestione cache L2

A seconda del tipo di accesso/uso che si fa dei dati memorizzati in una regione di global memory si può caratterizzare la regione (e i dati stessi) come:

- Persisting memory: è una regione di memoria i cui dati sono acceduti ripetutamente dal/dai kernel;
- Streaming memory: è una regione i cui dati vengono acceduti una volta sola.

La cache L2 ha lo scopo di permettere una latenza inferiore negli accessi alla global memory. Ottimale sarebbe che i dati della persisting memory avessero maggiore persistenza nella cache L2. Due scenari in cui sarebbe utile una maggiore persistenza in L2: intra-kernel e inter-kernel.



Le GPU più recenti supportano la gestione via software della politica di caching in L2. È possibile identificare le porzioni di persisting memory in global memory, associare ogni porzione ad una frazione della cache L2, e cambiare la normale politica di caching per tale frazione:



L2 Cache Residency Control:

- permette di specificare range di indirizzi, cioè array di persisting memory, in global memory (`base_ptr + num_bytes`);
- riservare una percentuale della cache L2 da usare per caching degli accessi a tale array;
- assegnare a tale porzione la policy `cudaAccessPropertyPersisting`;
- questa operazione si fa nel contesto di uno stream: influenzerà tutti gli accessi di tutti i kernel lanciati da ora in poi in quello stream (fino a che non verrà ripristinata la policy normale (`cudaAccessPropertyNormal`)).

Politiche possibili:

- `cudaAccessPropertyPersisting`: i dati acceduti con questa persisting property avranno maggior persistenza in cache L2 e quindi maggior probabilità di essere trattenuti in L2 (nella set-aside portion);
- `cudaAccessPropertyStreaming`: i dati acceduti avranno minor persistenza in cache L2. Per questi dati viene utilizzata la parte di L2 non riservata ai persisting data (ma possono utilizzare la set-aside portion se è libera);
- `cudaAccessPropertyNormal`: è la politica normale.

È possibile anche configurare l'accesso ad un array come streaming, assegnando `cudaAccessPropertyStreaming` invece di `cudaAccessPropertyPersisting`. Ciò è utile quando si sa che gli accessi a tali indirizzi saranno eseguiti una volta sola. La set-aside portion viene riservata agli accessi persisting, ma l'uso non è esclusivo: se non è piena allora può essere usata da dati gestiti con politica normal o streaming. I dati con persisting property tendono a restare in L2 anche dopo che il kernel che ha configurato L2 termina. È opportuno quindi che si ripristini la politica normale appena non è più necessario gestire accessi a dati persisting, assegnando, da device, `cudaAccessPropertyNormal`:

```
cudaStreamAttributeValue attr ; /* si opera in uno specifico stream */
attr.accessPolicyWindow.base_ptr = /* indirizzo di inizio dell'array */ ;
attr.accessPolicyWindow.num_bytes = /* numero di byte nell'array */ ;
attr.accessPolicyWindow.hitProp = cudaAccessPropertyNormal;
cudaStreamSetAttribute(stream,cudaStreamAttributeAccessPolicyWindow,&attr);
```

Ottobre, da host, per tutta la L2 eseguendo: `cudaCtxResetPersistingL2Cache()`.

Esempio dettagliato:

```
cudaStream_t stream;
cudaStreamCreate(&stream); // Create CUDA stream

cudaDeviceProp prop;
cudaGetDeviceProperties( &prop, device_id);
size_t size = min( int(prop.l2CacheSize * 0.75) , prop.persistingL2CacheMaxSize );
cudaDeviceSetLimit( cudaLimitPersistingL2CacheSize, size); // set-aside 3/4 of L2 cache for persisting accesses or the max allowed

size_t window_size = min(prop.accessPolicyMaxWindowSize, num_bytes); // Select minimum of user defined num_bytes and max window size.

cudaStreamAttributeValue stream_attribute;
stream_attribute.accessPolicyWindow.base_ptr = reinterpret_cast<void*>(data1);
stream_attribute.accessPolicyWindow.num_bytes = window_size; // Stream level attributes data structure
stream_attribute.accessPolicyWindow.hitRatio = 0.6; // Global Memory data pointer
stream_attribute.accessPolicyWindow.hitProp = cudaAccessPropertyPersisting; // Number of bytes for persistence access
stream_attribute.accessPolicyWindow.missProp = cudaAccessPropertyStreaming; // Hint for cache hit ratio // Persistence Property // Type of access property on cache miss

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_attribute); // Set the attributes to a CUDA Stream

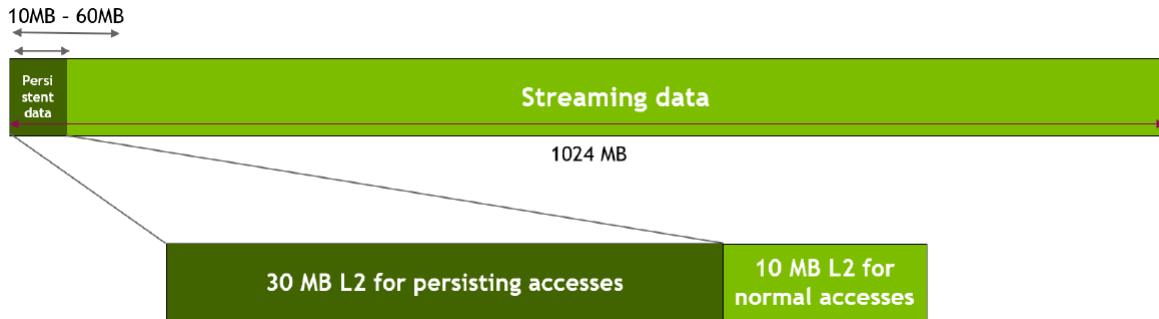
for(int i = 0; i < 10; i++) {
    cuda_kernelA<<<grid_size,block_size,0,stream>>>(data1); // This data1 is used by a kernel multiple times
} // [data1 + num_bytes) benefits from L2 persistence
cuda_kernelB<<<grid_size,block_size,0,stream>>>(data1); // A different kernel in the same stream can also benefit // from the persistence of data1

stream_attribute.accessPolicyWindow.num_bytes = 0; // Setting the window size to 0 disable it
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_attribute); // Overwrite the access policy attribute to a CUDA Stream
cudaCtxResetPersistingL2Cache(); // Remove any persistent lines in L2

cuda_kernelC<<<grid_size,block_size,0,stream>>>(data2); // data2 can now benefit from full L2 in normal mode
```

Configurazione Hit-Ratio

CUDA permette di configurare l'hit-ratio.



3.4.4 Shared Memory

La shared memory è accessibile a tutti i thread dello stesso blocco. Dovrebbe fungere da “memoria di lavoro” o essere usata come una “cache gestita via software” al fine di minimizzare gli accessi alla global memory da parte dei thread del blocco. È notevolmente più veloce della memoria locale e di quella globale: la sua latenza è anche 100 volte inferiore di quella della uncached global memory.

Quando si usa la memoria shared è fondamentale evitare le race condition che si possono verificare nell'accesso alla shared memory (da parte di thread dello stesso blocco) perché tali thread eseguono logicamente in parallelo, ma fisicamente potrebbero non eseguire nello stesso istante.

Esempio: 2 thread A e B in due warp diversi eseguono load di un dato dalla global memory e lo salvano in shared memory, poi A legge l'elemento di B dalla shared e viceversa. Se B non ha terminato la scrittura prima che A legga, si ha una race condition che porta ad un risultato indeterminato. Per assicurare la correttezza della computazione/cooperazione tra thread è necessario sincronizzarli. CUDA fornisce un costrutto di barrier: `_syncthreads()`.

La shared memory è allocata tramite la keyword: `_shared_`, sia staticamente che dinamicamente.

staticamente	dinamicamente
<pre>// run version with static shared memory cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice); staticReverse<<<1,n>>>(d_d, n);</pre>	<pre>// run dynamic shared memory version cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice); dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);</pre>
<pre>_global_ void staticReverse(int *d, int n) { _shared_ int s[64]; ← int t = threadIdx.x; int tr = n-t-1; s[t] = d[t]; _syncthreads(); d[t] = s[tr]; }</pre>	<pre>_global_ void dynamicReverse(int *d, int n) { extern _shared_ int s[]; ← int t = threadIdx.x; int tr = n-t-1; s[t] = d[t]; _syncthreads(); d[t] = s[tr]; }</pre>

Osservazione: Si noti l'uso di `_syncthreads()` per evitare le race condition.

Allocazione dinamica: La dimensione è determinata tramite il terzo parametro nella sintassi <<< ... >>> del lancio del kernel:

```
// run dynamic shared memory version
cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);

__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

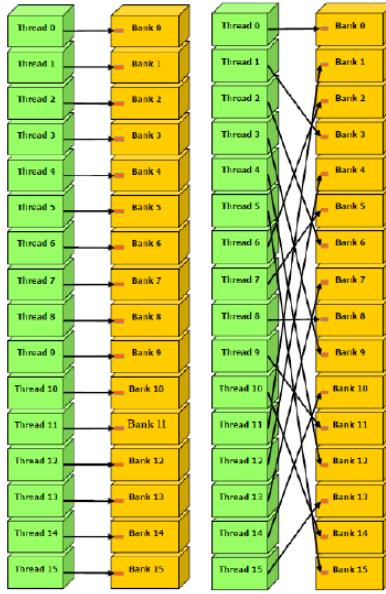
Shared Memory e Bank Conflict

- La shared memory è suddivisa in moduli di uguale dimensione detti bank (dalla cc 2.0 ci sono 32 bank);
- Word consecutive da 32-bit corrispondono a bank successivi e la bandwidth è di 32 bits per bank per clock cycle;
- Diversi bank possono essere acceduti in simultanea;
- Ogni bank può servire un solo indirizzo per clock cycle;
- La memoria può quindi servire contemporaneamente al massimo tanti accessi (a bank diversi) quanti sono i bank.

Conseguenza: N load/store di indirizzi che riferiscono B diversi bank possono essere serviti contemporaneamente ad un bandwidth che è B volte quello del singolo bank. Tuttavia: se più thread accedono indirizzi dello stesso bank (bank conflict) tali accessi vengono serializzati (tranne nel caso in cui l'accesso sia allo stesso indirizzo (accessi multicast)).

Le richieste di accesso che comportano accessi conflittuali vengono partionate dall'hardware in più accessi non conflittuali eseguiti serialmente. Quindi si ha un degrado della bandwidth di un fattore pari al numero di accessi conflittuali.

Per eliminare i bank conflict: È necessario scegliere bene l'access pattern degli accessi alla shared memory. Nella figura: access pattern con stride = 1 e stride = 3, no conflict. Configurare la dimensione dei bank (bank size) a 4 byte o a 8 byte, tramite cudaDeviceSetSharedMemConfig()(possibile solo per compute capability \geq 3.x, 4 è il default). Ad esempio, fissare bank size uguale a 8 byte può ridurre i bank conflict nell'accesso a dati in double precision. Allocare diversamente i dati in shared memory.



3.4.5 Linee guida generali (performance e memoria globale)

- Minimizzare i trasferimenti di dati tra host e device, anche se ciò comporta eseguire dei kernel che svolgono poco lavoro (e che sarebbe più efficientemente svolto dall'host) per evitare di “rimbalzare” dati tra host e device
- Preferire l'uso di page-locked global memory: offre maggiore bandwidth
- Combinare molti piccoli trasferimenti di dati in un unico trasferimento
- Se possibile eseguire i trasferimenti di dati host-device concorrentemente all'esecuzione dei kernel (o ad altri trasferimenti)
- Se possibile lavorare prevalentemente in shared memory piuttosto che in global memory (ridurre gli accessi alla global: portare i dati in shared, effettuare serie di aggiornamenti in shared, scrivere solo il risultato finale in global)

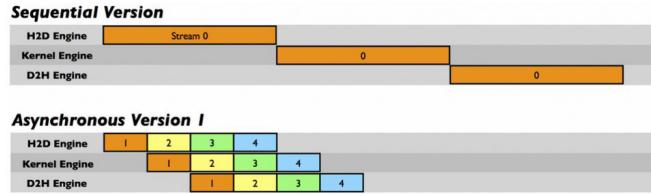
Per valutare il tempo impiegato dai trasferimenti di dati si può usare il profiler nvprof:

```
$ nvprof ./a.out
=====
NVPROF is profiling a.out...
=====
Command: a.out
=====
Profiling result:
Time(%)      Time    Calls      Avg      Min      Max  Name
 50.08  718.11us     1  718.11us  718.11us  718.11us [CUDA memcpy DtoH]
 49.92  715.94us     1  715.94us  715.94us  715.94us [CUDA memcpy HtoD]
```

3.5 Esecuzione concorrente ed asincrona

CUDA permette l'esecuzione di operazioni come task indipendenti che possono operare contemporaneamente gli uni agli altri.

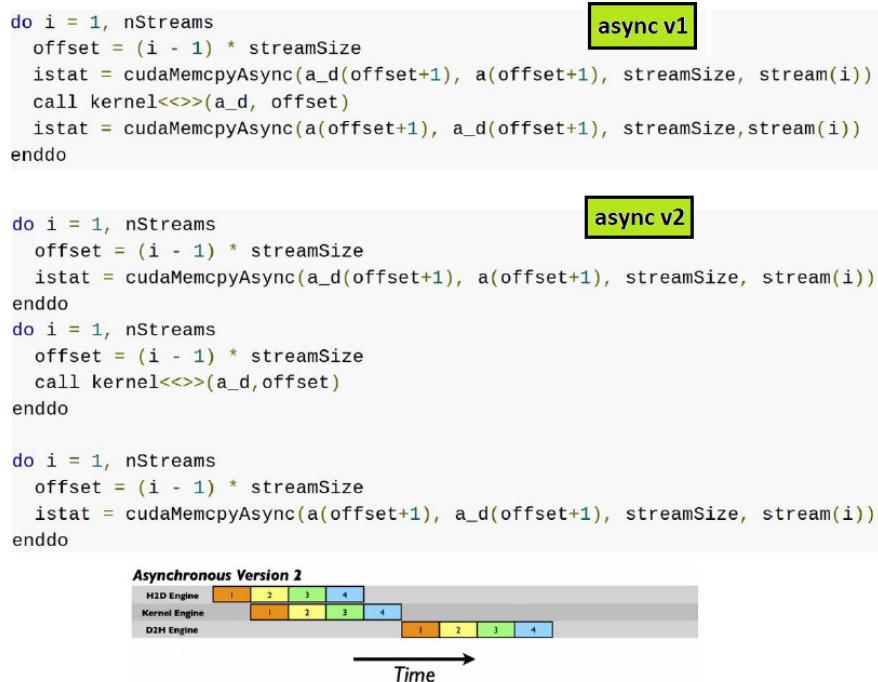
L'esecuzione asincrona (asynchronous execution) permette alle operazioni NON-bloccanti di restituire immediatamente il controllo al programma chiamante (prima del completamento dell'operazione, che nel frattempo prosegue la sua esecuzione):



Esempio: La Tesla C2050, oltre al “kernel engine” ha due “copy engine”, uno per trasferimenti host-to-device e l’altro per quelli device-to-host. In tal modo è possibile l’overlap dei trasferimenti a patto che le operazioni vengano svolte in stream diversi (nella figura: quattro stream 1, 2, 3, e 4).

Nota: in generale (in base alla logica del programma), le operazioni inviate su stream diversi necessitano di essere sincronizzate (tramite barrier, sfruttando gli eventi, ...). Le tipologie di concorrenza messe a disposizione al programmatore dipendono dalla compute capability della GPU (per es. le CC più vecchie prevedono una sola copy engine, ...).

Il risultato dipende anche da come l'host invia i comandi. Ecco possibili overlap su C2050 (esempio completo, in attesa di conoscere maggiori dettagli sugli stream): nella seconda versione del codice la scheda attende la terminazione delle attività su kernel engine prima di attivare D2H engine.



3.5.1 Operazioni concorrenti

Dicevamo che CUDA permette l'esecuzione di operazioni come task indipendenti che possono operare concorrentemente gli uni agli altri. In particolare, riguardo a:

- Computazione su host;
- Computazione su device;
- Trasferimenti dati tra host e device e tra diversi device;
- Trasferimenti di memoria internamente alla memoria globale dello stesso device.

L'esecuzione concorrente tra host e device può riguardare:

- Kernel
- Copie di memoria internamente alla memoria del device;
- Copie da host a device di un blocco di memoria di al più 64 KB;
- Copie di dati eseguite tramite le funzioni che hanno suffissi “Async”;
- Funzioni che effettuano set della memoria (memset); (in pratica la chiamata ritorna subito il controllo al chiamante; e non è detto che l'operazione sia conclusa (ad esempio, in un trasferimento di dati non è detto che al return i dati siano già stati trasferiti...)).

Osservazione: Si può disabilitare globalmente la asincronia tra i kernel, per tutte le applicazioni in esecuzione nel sistema assegnando 1 alla variabile d'ambiente CUDA_LAUNCH_BLOCKING.

3.5.2 CUDA stream

Un CUDA stream è una sequenza di comandi inviati (anche da diversi thread dell'host) che sono schedulati in ordine.

Esiste un default stream utilizzato implicitamente: tutti i comandi non esplicitamente inviati ad uno specifico stream vengono inviati al default stream (i comandi sono schedulati in ordine, ma come abbiamo visto, possono essere eseguiti da diverse unit: compute engine, copy engine (H2D, D2H)).

CUDA permette l'utilizzo contemporaneo di più stream, ciò significa che i comandi inviati su diversi stream possono eseguire concorrentemente e in modo out-of-order gli uni rispetto agli altri. Conseguentemente, eventuali dipendenze (data-dependencies) tra i comandi devono essere gestite dal programmatore introducendo punti di sincronizzazione.

Ci sono diverse tipologie di programmazione asincrona in CUDA:

1. Asynchronous programming models (CUDA Streams);
2. Task-based (CUDA graphs);
3. Event-based (CUDA event).

Creazione di uno stream CUDA

Uno stream si definisce creando uno stream object che poi si utilizza specificandolo come argomento nelle operazioni di copia della memoria o nei lanci dei kernel.

Requisito sulla gestione della memoria (per poter sfruttare l'overlap di op su memoria): la memoria host deve essere page-locked.

Esempio di creazione di due stream:

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Esecuzione di uno stream CUDA

Ognuno dei nostri due stream potrebbe comprendere una sequenza di operazioni. Ad esempio:

1. una copia (asincrona) dalla memoria host alla memoria device
2. un lancio di un kernel
3. una copia (asincrona) dalla memoria device all'host

come in:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
    (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                   size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Ove, in ogni stream i:

- si copia una porzione di array (hostPtr+i) in device memory (inputDevPtr+i)
- si processa tali dati sul device lanciando MyKernel()
- si copia i risultati (da outputDevPtr+i) dal device all'host (sovrascrivendo da hostPtr+i)

Distruzione di uno stream CUDA

Dopo l'uso, gli stream vengono eliminati tramite `cudaStreamDestroy()`.

Se il device sta ancora elaborando quando avviene la call a `cudaStreamDestroy()`, allora la funzione ritorna immediatamente ma le risorse associate allo stream vengono rilasciate appena il device ha esaurito le operazioni inviate nello stream.

Nel nostro esempio eliminiamo i due stream tramite:

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Sincronizzazione esplicita di stream CUDA

Vi sono diversi tipi di sincronizzazione esplicita che può avvenire tra gli stream:

Sincronizzare l'host con “tutto”:

- cudaDeviceSynchronize() attende fino a quando tutti i comandi già inviati (ad ogni stream) da qualsiasi host-thread sono stati completati
- blocca l'host fino a che tutte le call a cuda eseguite in precedenza sono state completate

Sincronizzazione rispetto ad uno specifico stream:

- cudaStreamSynchronize(streamID) richiede uno stream come argomento e attende fino a quando tutti i comandi già inviati a quello stream sono completati
- Si può usare per bloccare l'host fino a che tutte le call eseguite in streamID sono completate

Sincronizzazione usando gli eventi:

- Permette di implementare un modello event-based: si definisce un 'event' internamente ad uno stream al fine di usarlo per una sincronizzazione
- cudaStreamWaitEvent(stream, event): dato uno stream ed un evento introduce una attesa dell'evento in quello stream (riguarda tutti i successivi comandi inviati in stream)

Sincronizzazione implicita di stream CUDA

Vi sono operazioni che quando eseguite comportano automaticamente una sincronizzazione implicita. Per esempio, due comandi A e B, anche se in due stream diversi, non possono essere eseguiti concorrentemente se una delle successive operazioni viene invocata dall'host tra A e B (elenco non esaustivo):

- Qualsiasi comando nel default stream
- Allocazione di memoria page-locked: cudaMallocHost; cudaHostAlloc
- Allocazione di memoria sul device: cudaMalloc
- Operazioni non-Async sulla memoria: le funzioni del tipo “cudaMemcpy” senza il suffisso Async; le funzioni del tipo “cudaMemset” senza il suffisso Async
- Cambiamenti nella configurazione della coppia L1/shared-memory: cudaDeviceSetCacheConfig

3.5.3 Streams e phtreads

Un esempio di utilizzo di CUDA streams combinato con l'uso dei posix threads (pthreads) su host:

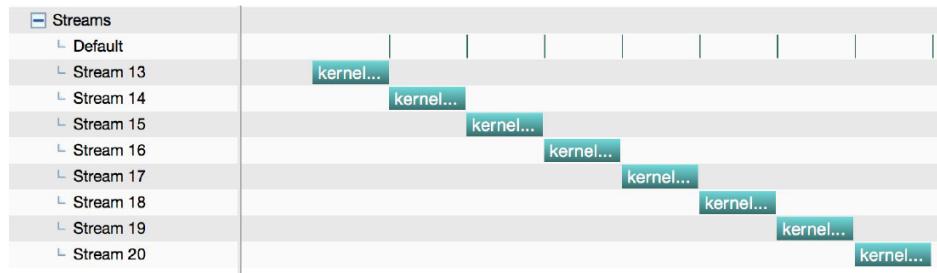
```
const int N = 1 << 20;

__global__ void kernel(float *x, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
        x[i] = sqrt(pow(3.14159,i));
    }
}

int main(){
    const int num_streams = 8;
    cudaStream_t streams[num_streams];
    float *data[num_streams];

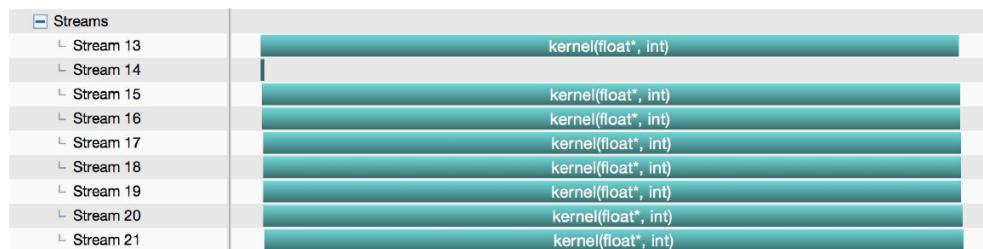
    for (int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]); // crea uno degli 8 stream
        cudaMalloc(&data[i], N * sizeof(float));
        kernel<<<1, 64, 0, streams[i]>>>(data[i], N); // lancia un kernel per stream
        kernel<<<1, 1>>>(0, 0); // lancia un "dummy kernel" nel default stream
    }
    cudaDeviceReset();
    return 0;
}
```

Compilazione con: **nvcc stream_test.cu -o stream_uno** e profilazione con nvvp:



CUDA consente una migliore interazione tra stream e pthreads in modo da “controllare” uno stream da un singolo pthread. È necessario compilare specificando che si vuole associare un pthread ad ogni stream:

Compilazione con: **nvcc --default-stream per-thread stream_test.cu -o stream_due** e profilazione con nvvp:



Un altro esempio in cui l'uso dei pthread è esplicito nel codice host: combinazione di parallelismo multi-core host-side con parallelismo SIMT di CUDA:

```
#include <pthread.h>
#include <stdio.h>
const int N = 1 << 20;

__global__ void kernel(float *x, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    for (int i = tid; i < n; i += blockDim.x * gridDim.x) {
        x[i] = sqrt(pow(3.14159,i));
    }
}

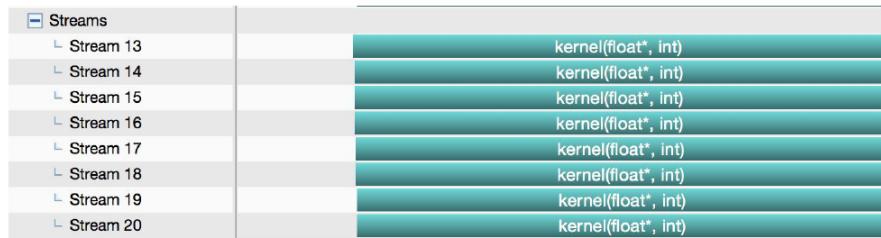
void *launch_kernel(void *dummy) {
    float *data;
    cudaMalloc(&data, N * sizeof(float));
    kernel<<<1, 64>>>(data, N);
    cudaStreamSynchronize(0);
    return NULL;
}

int main() {
    const int num_threads = 8;
    pthread_t threads[num_threads];
    for (int i = 0; i < num_threads; i++) {
        if (pthread_create(&threads[i], NULL, launch_kernel, 0)) {
            fprintf(stderr, "Error creating threadn");
            return 1;
        }
    }
    for (int i = 0; i < num_threads; i++) {
        if(pthread_join(threads[i], NULL)) {
            fprintf(stderr, "Error joining threadn");
            return 2;
        }
    }
    cudaDeviceReset();
    return 0;
}
```

Compilazione “tradizionale”: **nvcc treadstream.cu -o threadstream_1** e profilazione con nvvp:



Se usiamo la compilazione che associa threads a streams: **nvcc --default-stream per-thread threadstream.cu -o threadstream_2**



3.5.4 Esecuzione concorrente: CUDA graph

Motivazione: Molte applicazioni hanno una struttura iterativa, che procede ripetendo lo stesso workflow.

Gli stream richiedono che il lavoro venga ri-sottomesso ad ogni iterazione (rilanciare i kernel, ecc), ma così si aumenta il consumo di risorse.

Un CUDA graph (DAG) consiste di una serie di operazioni (copie di memoria e lanci di kernel) collegate da dipendenze e specificate indipendentemente dalla esecuzione.

I CUDA Graph consentono un flusso di esecuzione del tipo define-once-run-repeatedly. Per i kernel che hanno tempi di esecuzione brevi, l'overhead per effettuare il lancio può essere una porzione significativa dell'execution time totale.

Con i graph si disaccoppia la definizione di graph+dipendenze dalla esecuzione, riducendo il costo che la CPU deve pagare per lanciare ripetutamente le operazioni.

Definizione esplicita ed esecuzione di un CUDA graph

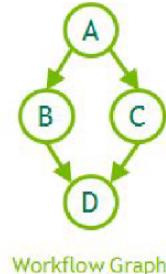
Un CUDA graph è composto da:

- Nodi = kernel (ma anche op in memoria, attesa eventi, call in host,...)
- Archi = dipendenze tra i kernel

Definizione

```
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);           // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);           // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);           // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);           // C->D
```



Esecuzione

```
// Instantiate graph and apply optimizations

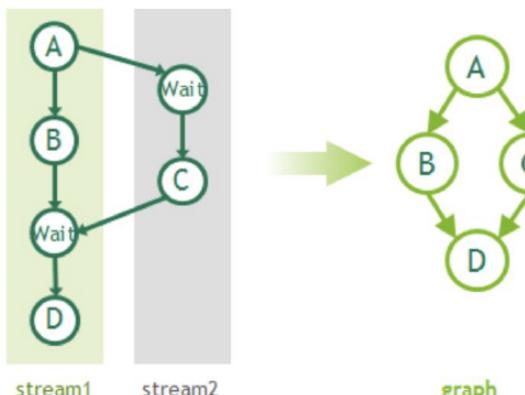
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times

for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
```

Conversione

Un graph può essere “catturato” a partire da una esecuzione che utilizza gli stream. Le dipendenze sono dedotte sfruttando gli eventi che il programmatore ha inserito tra “i nodi”:



```
// Start by initiating stream capture

cudaStreamBeginCapture(stream1);

// Build stream work as usual

A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph

cudaStreamEndCapture(stream1, &graph);
```

3.5.5 Esecuzione concorrente: eventi

Gli eventi possono essere usati per misurare il tempo. L'host può registrare un evento in modo asincrono in qualsiasi punto del programma e successivamente può interrogare per sapere se un evento è completato.

Creazione e distruzione di due eventi, nell'host program:

```
cudaEvent_t start, stop; cudaEventDestroy(start);
cudaEventCreate(&start); cudaEventDestroy(stop);
cudaEventCreate(&stop);
```

Esecuzioni basate su eventi possono essere realizzate assegnando eventi agli stream:

1. Creazione di un evento event (o array di eventi)
2. Assegnazione dell'evento event ad uno streamID: cudaEventRecord(event, streamID)
3. Attesa del completamento di un evento: cudaEventSynchronize(event) (Osservazione: differisce da cudaStreamWaitEvent(stream, event) che invece fa sì che la computazione dello stream attenda l'evento: lo stream resta bloccato fino a che l'evento non viene registrato)
4. Interrogazione sullo stato dell'evento: cudaEventQuery(event)

Un uso semplice degli eventi:

```
cudaEventRecord ( startEvent, stream );
my_kernel<<<grid,block,0,stream>>>(...);
cudaEventRecord ( endEvent, stream );

//Host can do other work

//Get runtime of my_kernel in ms
float runtime = 0.0f;
cudaEventSynchronize ( endEvent );
cudaEventElapsedTime ( &runtime, startEvent, endEvent );
```

3.5.6 Alcune considerazioni sulle performance

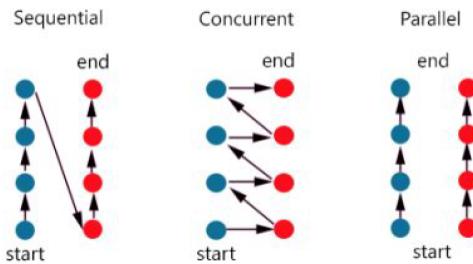
- Se si utilizzano gli stream allora non utilizzare il default stream (sync impliciti,...)
- Ottimizzare l'uso della memoria (shared vs global) con i corretti access pattern
- Massimizzare la concorrenza (esecuzione asincrona, overlap,...)
- Ottimizzare l'uso delle istruzioni per migliorarne il throughput:

- Limitare l'uso di istruzioni aritmetiche con basso throughput (per esempio preferire shift alla divisione o al modulo)
- Usare signed integer piuttosto che unsigned integers come loop counter
- Evitare istruzioni di controllo (if, switch, do, for, while) che creino divergenza intra-warp
- Usare il qualificatore `_launch_bounds_`(`maxThreadsPerBlock`, `minBlocksPerMultiprocessor`) (dopo `_global_` nella def del kernel): il compilatore migliora l'uso dei registri degli SMs
- Utilizzare `#pragma unroll n` : posto subito prima dei for-loop ne causa l'unroll
- Se possibile, utilizzare read-only data-cache, tramite istruzioni come: `_ldg(const T* address)`
- ...

4 Concorrenza e Parallelismo

Possiamo descrivere così questi due concetti:

- **Concorrenza:** una applicazione procede facendo progredire più task contemporaneamente;
- **Parallelismo:** una applicazione suddivide i suoi task in porzioni che possono essere processate in parallelo indipendentemente.



La somma elemento-per-elemento di array presenta parallelismo, relativamente al numero di elementi (la lunghezza) dell'array. Calcolare la riduzione degli elementi di un array (per es., calcolare la loro somma) richiede di considerare la concorrenza tra i thread (nel calcolare il risultato finale). Si tenga presente che delle race condition possono verificarsi quando thread concorrenti accedono alla stessa locazione di memoria: accade quando più thread tentano di modificare lo stesso dato in contemporanea, in una situazione in cui il programming model o lo scheduler degli accessi alla memoria non garantiscono un ordine delle operazioni.

Nozione di **sezione critica**: una sezione di codice che racchiude accessi a risorse condivise, acceduta contemporaneamente da più thread che effettuano modifiche.

4.0.1 Gestione delle race condition (in breve): Un **lock** o **mutex** è un meccanismo di sincronizzazione che limita l'accesso ad una risorsa (solitamente una porzione di memoria) in presenza di più flussi di esecuzione.

Solamente ad un thread/processo P viene permesso di entrare nella sua sezione critica: tale ingresso blocca l'entrata nella (loro) sezione critica ad altri thread/processi, fino alla uscita di P dalla sua sezione critica.

Un semaphore è un tipo di dato astratto utilizzabile per controllare l'accesso ad una risorsa condivisa, per implementare protocolli di mutua esclusione e di sincronizzazione (vedi corso di Sistemi Operativi).

L'implementazione di protocolli di mutua esclusione possono sfruttare operazioni atomiche per imporre serializzazione di istruzioni/accessi.

Esempio: algoritmo di Peterson

Due processi (o thread) P0 e P1.

Variabili condivise:

- flag[n], con valore booleano, denota se il processo n vuole accedere alla sezione critica
- turn, valore 0 o 1, indica quale processo ha diritto di entrare nella sua sezione critica

```
P0:    flag[0] = true;
P0_gate: turn = 1;
        while (flag[1] && turn == 1)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
flag[0] = false;
```

```
P1:    flag[1] = true;
P1_gate: turn = 0;
        while (flag[0] && turn == 0)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
flag[1] = false;
```

Esistono generalizzazioni al caso di più di 2 processi (ad esempio l'alg. del fornaio (Lamport), ...).

L'algoritmo di Peterson funziona se gli accessi alle variabili turn, flag[0], flag[1] sono fatti con istruzioni atomiche. Facilmente si verifica che, in tal caso, P0 e P1 non possono essere contemporaneamente nelle loro sezioni critiche (turn non può essere contemporaneamente 0 e 1).

- Se P0 è nella sezione critica allora flag[0] è true, mentre flag[1] può essere:
 - false: P1 ha lasciato la sezione critica
 - true: P1 non è ancora arrivato al while oppure sta aspettando perché turn==0

La soluzione garantisce mutua esclusione, progresso e attesa limitata

4.1 Concorrenza e Parallelismo in CUDA

CUDA fornisce diversi meccanismi per supportare la mutua esclusione e la cooperazione tra thread, consentendo il progetto/implementazione di algoritmi lock-free

1. Operazioni atomiche
2. Cooperative groups
3. Primitive di sincronizzazione a livello di warp (warp-level synchronization)

4.1.1 Operazioni atomiche

Le operazioni atomiche di CUDA sono implementate via hardware.

Una operazione atomica realizza una operazione read-modify-write su una locazione di memoria, con una singola istruzione macchina.

Il thread che esegue l'istruzione:

- legge il valore memorizzato nella locazione
- calcola un nuovo valore (eventualmente condizionato dall'esito di un test)
- e infine scrive in nuovo valore nella stessa locazione.

Durante tutto ciò, è garantito che nessun altro thread possa interferire con l'operazione (a nessun altro thread è permesso accedere a quella locazione), che risulta quindi essere atomica.

Le operazioni atomiche si possono usare solo nel codice device.

L'hardware assicura che nessun altro thread possa effettuare un'altra operazione read modify write sulla stessa locazione fintanto - - che la prima operazione non è terminata:

- Tutti i thread che tentano di eseguire una operazione atomica sulla stessa locazione vengono messi in attesa in una coda
- Tutti questi thread eseguiranno l'operazione in modo seriale
- Non è garantito alcun ordine particolare o alcuna sincronizzazione tra questi thread

Operazioni atomiche **system-wide**: L'operazione è atomica rispetto a tutti i thread del programma, includendo anche i thread che eseguono sulla CPU o su altre GPU Sono identificate dal suffisso _system, come ad esempio in: atomicAdd_system()

Operazioni atomiche **device-wide**: L'operazione è atomica rispetto a tutti i CUDA thread del programma che eseguono sullo stesso device Sono identificate dall'assenza di uno specifico suffisso, come in: atomicAdd()

Operazioni atomiche **block-wide**: L'operazione è atomica rispetto a tutti i CUDA threads che eseguono nello stesso block Sono identificate dal suffisso _block, come in: atomicAdd_block()

Esempi di operazioni atomiche:

int **atomicAdd**(int* address, int val) legge la word a 32-bit old presente all'indirizzo address in global o in shared memory, computa l'espressione (old + val), memorizza nella stessa locazione il valore ottenuto, restituisce old. Il tutto viene eseguito atomicamente. Disponibile anche per altri tipi di dato (interi e floating-point a 16, 32 o 64 bit)

unsigned int **atomicInc**(unsigned int* address, unsigned int val) legge la word a 32-bit old all'indirizzo address in global o shared memory, computa l'espressione (old >= val ? 0 : (old+1)), ne memorizza il risultato nella stessa locazione, restituisce old. Il tutto viene eseguito atomicamente

int **atomicCAS**(int* address, int compare, int val); legge la word old all’indirizzo address in global o shared memory, computa l’espressione (old == compare ? val : old), ne memorizza il risultato nella stessa locazione, restituisce old. Il tutto viene eseguito atomicamente. Disponibile per interi a 16, 32 o 64 bit, signed o unsigned

int **atomicAnd**(int* address, int val); legge la word old all’indirizzo address in global o shared memory, computa l’espressione (old & val), ne memorizza il risultato nella stessa locazione, restituisce old. Disponibile per interi a 32 o 64 bit

Memory Fence

Osservazione: le operazioni atomiche non fungono da memory barrier, non impongono alcuna sincronizzazione tra i thread o un particolare ordine negli accessi alla memoria.

Delle memory fence possono tuttavia essere imposte tramite opportune istruzioni, nel codice del device.

Una premessa: Nel CUDA programming model si assume che il device operi attuando un **weakly-ordered memory model**; ovvero: l’ordine in cui un thread scrive dei dati in shared memory, in global memory, in page-locked host memory, o nella memoria di un altro device, non è necessariamente l’ordine che potrebbe seguire un altro thread (sia esso un thread CUDA o host).

Conseguenza: se due thread compiono in contemporanea una lettura e una scrittura sulla stessa locazione, senza impiegare meccanismi di sincronizzazione, il risultato è indefinito.

Esempio:

```
__device__ int X = 1, Y = 2;

__device__ void writeXY() {
    X = 10;
    Y = 20;
}

__device__ void readXY() {
    int B = Y;
    int A = X;
}
```

Se un thread T1 esegue writeXY() mentre un altro thread T2 esegue readXY(), allora i valori finali di A e B sono indefiniti (perché ci possono essere lettura e scrittura contemporanee allo stesso indirizzo e ciò ha un esito non definito).

Opportune funzioni si possono usare per introdurre delle memory fence al fine di imporre un ordinamento degli accessi alla memoria e garantire l’osservabilità di aggiornamenti alla memoria da parte dei thread

Esempi: __threadfence_block():

- tutte le scritture (in qualsiasi memoria) effettuate da un thread T prima di chiamare __threadfence_block() sono osservate dagli altri thread del suo blocco come accadute prima di tutte le scritture che il thread T effettua dopo la chiamata a __threadfence_block()

- tutte le letture effettuate dal thread T prima di chiamare `__threadfence_block()` sono ordinate precedentemente a tutte le letture che il thread T effettua dopo la chiamata a `__threadfence_block()`

`__threadfence()`: come il precedente, ma considerando tutti i thread del device. Si noti che in tal caso l'osservabilità del dato si riferisce alla sua copia in memoria, non alla eventuale copia in cache a cui potrebbero accedere altri thread. Per forzare un thread ad “osservare” la versione in memoria di una variabile (e non la copia nella cache), questa variabile deve essere dichiarata usando il qualificatore volatile

Qualificatore volatile: In generale il compilatore CUDA è libero di ottimizzare gli accessi alla memoria, anche sfruttando i registri e la cache L1 (rispettando la semantica delle eventuali memory fence). Tali ottimizzazioni possono essere inibite usando volatile: in tal caso il compilatore assume che il valore di una variabile dichiarata volatile possa essere modificato in ogni istante da un altro thread e quindi ogni accesso a tale variabile va fatto direttamente in memoria (no caching)

`__threadfence_system()` : come il precedente, ma considerando tutti i thread del device, tutti i thread dell'host e tutti i thread in ogni altro device del sistema.

Riprendiamo l'esempio e modifichiamolo inserendo delle memory fence:

```
__device__ int X = 1, Y = 2;

__device__ void writeXY() {
    X = 10;
    __threadfence();
    Y = 20;
}

__device__ void readXY() {
    int B = Y;
    __threadfence();
    int A = X;
}
```

In questo caso le memory fence assicurano che per il thread T2 (che esegue `readXY()`) valga sempre che: “se B ha valore 20 allora sicuramente A ha valore 10”

Esempio: somma degli elementi di un array. Sincronizzazione tra thread e memory fence

```

__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N, volatile float* result) {
    // Ogni blocco somma una porzione dell'input:
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {
        // Il thread 0 di ogni block memorizza la somma parziale in global memory
        // Il compilatore userà un accesso diretto in memoria (no cache L1)
        // perché result è dichiarata volatile. Cosicché i thread dell'ultimo blocco
        // leggeranno i valori corretti computati dagli altri blocchi:
        result[blockIdx.x] = partialSum;

        // Il thread 0 effettua l'incremento di count, ma assicurandosi che sia
        // visibile solo dopo che l'aggiornamento di result sia visibile:
        __threadfence();
        unsigned int value = atomicInc(&count, gridDim.x);

        // Il thread 0 determina se appartiene all'ultimo blocco che ha
        // eseguito la atomicInc()
        isLastBlockDone = (value == (gridDim.x - 1));
    }

    // Sincronizzazione affinché ogni thread legga il valore
    // corretto di isLastBlockDone.
    __syncthreads();

    if (isLastBlockDone) {
        // L'ultimo blocco che esegue, somma le somme parziali
        // memorizzate in result[0 .. gridDim.x-1]
        float totalSum = calculateTotalSum(result);

        if (threadIdx.x == 0) {
            // Il thread 0 dell'ultimo block salva il risultato finale in global mem
            // e azzerza count
            result[0] = totalSum;
            count = 0;
        }
    }
}

```

Sincronizzazione intra-block tra thread La funzione intrinseca `__syncthreads()` ha finora rappresentato il meccanismo principale per la sincronizzazione tra i thread dello stesso block.

Esistono altre funzioni simili:

`int __syncthreads_count(int predicate)` come `__syncthreads()`, ma in più ogni thread del block valuta predicate (una espressione intera), il valore di ritorno è il numero di thread che hanno ottenuto valore non nullo valutando predicate

`int __syncthreads_and(int predicate)` come `__syncthreads()`, ma in più ogni thread del block valuta predicate, il valore di ritorno è nullo se e solo se almeno un thread ha ottenuto un valore nullo per predicate

`int __syncthreads_or(int predicate)` come `__syncthreads()`, ma in più ogni thread del block valuta predicate, il valore di ritorno è non nullo se e solo se almeno un thread ha ottenuto un valore non nullo per predicate

La funzione `__syncwarp()` permette invece di imporre una barrier tra i thread dello stesso warp:

`void __syncwarp(unsigned mask=0xffffffff) // mask di default: tutti i 32 thread del warp`

- il thread attende che tutti i thread del warp indicati da mask abbiano eseguito `__syncwarp()` (con stessa mask) e poi riprende l'esecuzione.
- tutti i thread (non già terminati) indicati nella mask devono eseguire una corrispondente `__syncwarp()` con la stessa mask, altrimenti il risultato non è definito.
- Inoltre garantisce il memory ordering per i thread coinvolti nella barrier.

Per esempio: se i thread di un warp devono aggiornare dei dati e poi leggere le modifiche apportate da altri thread dello stesso warp, possono scrivere in memoria e poi usare

`__syncwarp()` per essere sicuri di leggere i dati correttamente aggiornati dagli altri thread del warp.

4.1.2 Cooperative Groups

Iniziamo con una premessa: Il meccanismo di sincronizzazione tra thread di un block offerto da CUDA, tramite la funzione intrinseca `__syncthreads()` e le sue varianti, può risultare in alcune situazioni/applicazioni, troppo rigido.

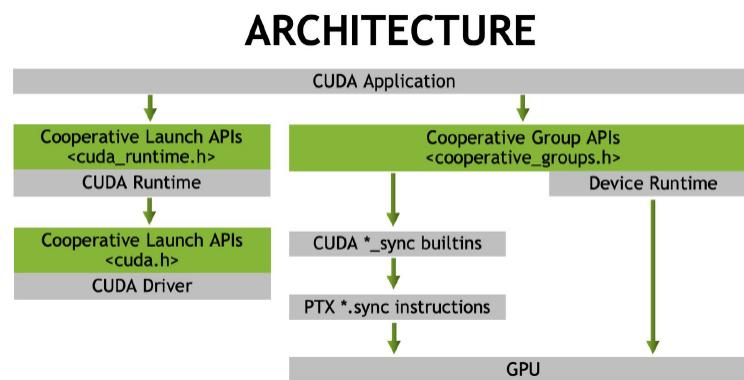
Nel progettare algoritmi paralleli in cui gruppi di thread devono collaborare, si sente a volte la necessità di costrutti più duttili che permettano di esprimere esplicitamente vincoli di sincronizzazione a granularità diverse da quella dell'intero block (o warp).

Il Cooperative Groups programming model descrive diversi pattern di sincronizzazione sia intra- che inter-blocks.

I **Cooperative Groups** offrono delle direttive che possono essere impiegate per definire, partizionare, sincronizzare gruppi di thread.

Il Cooperative Groups programming model specifica:

- tipi di dato per rappresentare gruppi di thread cooperanti e loro proprietà
- i gruppi intrinsecamente definiti dal meccanismo di lancio dei kernel di CUDA (i thread blocks)
- operazioni per partizionare i gruppi in altri gruppi
- barrier per sincronizzare i thread dei gruppi
- primitive di comunicazione nei gruppi



Concetti base: Per abilitare l'uso dei Cooperative Groups è necessario:

- includere lo header: `# include <cooperative_groups.h>`
- usare il namespace: `using namespace cooperative_groups`

- a volte si usa anche l'alias (librerie): namespace cg=cooperative_groups

La nozione principale è quella del tipo di dato `thread_group` e del tipo derivato `thread_block` (che in pratica corrisponde alla nozione di CUDA block vista finora).

I `thread_group` permettono di eseguire operazioni collettive da parte di tutti i thread di un gruppo.

Funzioni di utilità generale; per i thread in un `thread_group`:

- `unsigned size()` : fornisce il numero di thread del gruppo
- `unsigned thread_rank()` : fornisce l'indice del thread chiamante, nel gruppo

Per i thread in un `thread_block`:

- `thread_block g = this_thread_block()` : definisce `g` e lo inizializza all'insieme di thread che compongono il blocco corrente
- `dim3 group_index()` : fornisce l'indice del block nella grid (equivalente a `blockIdx`)
- `dim3 thread_index()` : fornisce l'indice del thread nel block (equivalente a `threadIdx`)

Collective operations

Le collective operation sono operazioni che coportano sincronizzazione/comunicazione in uno specifico insieme di thread.

Il caso più semplice è la barrier (che non prevede scambio di dati):

`void sync()` : sincronizza i thread del gruppo del chiamante (tutti i thread del gruppo dovranno raggiungere la barrier).

Per esempio, se `g` è definito come prima, allora `g.sync()` sincronizza tutti i thread del block; quindi (per come è stato definito `g`) è equivalente ad eseguire `_syncThreads()`.

Altre istruzioni equivalenti:`cg::synchronize(g); this_thread_block().sync();cg::synchronize(this_thread_bla`

Abbiamo detto che tramite il tipo `thread_block` si può esplicitare nel programma la nozione di CUDA block (implicita nel meccanismo di lancio dei kernel). Ecco un esempio che compie una reduction: i thread del gruppo `g` eseguono la somma di tutti i valori `val` che ogni thread ottiene come argomento:

```
...
using namespace cooperative_groups;
...
__device__ int reduce_sum(thread_group g, int *temp, int val) {
    int lane = g.thread_rank();

    // ad ogni iterazione si dimezza il numero di thread attivi
    // ogni thread somma parziale temp[i] a temp[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2) {
        temp[lane] = val;
        g.sync(); // attende che tutti i thread scrivano
        if (lane < i) val += temp[lane + i];
        g.sync(); // attende che tutti i thread leggano
    }
    return val; // nota: solo il thread 0 restituisce la somma completa
}
...
```

Partizionamento dei blocchi

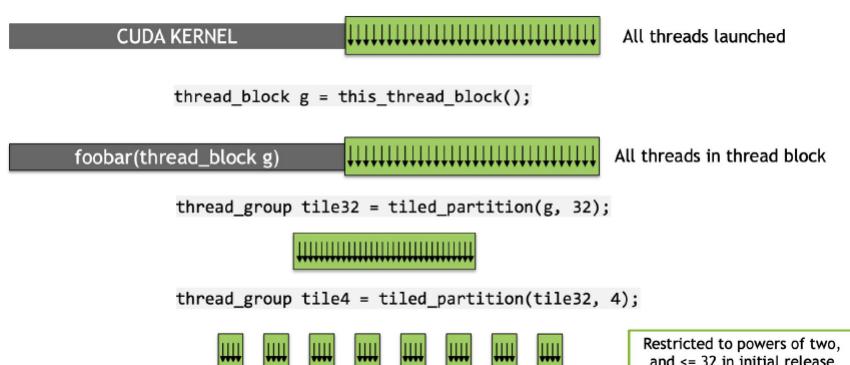
Un blocco può essere partizionato in gruppi più piccoli di thread cooperanti. Esempio:

```
...
thread_block ilBlocco = this_thread_block() // questo block
thread_group tile32 = tiled_partition(ilBlocco, 32) // partiziona in gruppi da 32 thread
thread_group tile4 = tiled_partition(tile32, 4) // partiziona i gruppi in sotto-gruppi da 4
thread
...
// se ora il codice contenesse:

if (tile4.thread_rank() == 0) printf("Hello from tile4 rank 0");
```

il messaggio sarebbe scritto solo dal thread di indice (rank) 0 di ognuno dei gruppetti da 4 thread, cioè dai thread 0,4,8,12,16,20,24 e 28 del blocco ilBlocco iniziale.

Nelle correnti versioni di CUDA, le size ammesse (secondo argomento di tiled_partition()) perdefinire partizioni sono le potenze di 2 non superiori a 32.



Riprendiamo la funzione reduce_sum(). Ecco un esempio di un suo impiego:

```
__device__ int thread_sum(int *input, int n) {
    int sum = 0;
    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n / 4;
        i += blockDim.x * gridDim.x) { // accesso strided
        int4 in = ((int4*)input)[i]; // usa vector load (più efficiente)
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n) {
    int my_sum = thread_sum(input, n);
    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);
    if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}
```

// frammenti del codice host:

...
 int n = 1<<24; // lunghezza dell'array: 16M
 int blockSize = 256;
 int nBlocks = (n + blockSize - 1) / blockSize;
 int sharedBytes = blockSize * sizeof(int);
 ...
 int *sum, *data;
 ...alloca int per sum ... su host e device (per es., d_sum)...
 ...alloca n int per data ... su host e device (per es., d_data)...
 ... inizializza data e copia input su device...

// call della reduction
sum_kernel_block<<<nBlocks, blockSize, sharedBytes>>>(d_sum, d_data, n);
 ... recupera il risultato ...

Somma degli elementi di un array data: prima calcola somme parziali tramite thread_sum(), poi usa thread_block gruppi per altrettante “somme cooperative” e atomicAdd() per sommare i risultati calcolati da ogni gruppo.

Coalesced groups In situazioni in cui vi è divergenza tra thread dello stesso warp, come sappiamo, i gruppi di thread vengono serializzati. I thread che sono sullo stesso execution path sono detti “coalesced”.

È possibile creare dinamicamente un thread_group composto dai thread attivi/coalesced, utilizzando la funzione coalesced_threads():

```
coalesced_group attivi = coalesced_threads();
```

se per esempio solo i thread 2, 4 e 8 sono attivi nel warp (perché seguono lo stesso execution path, mentre tutti gli altri thread del warp seguono un diverso path) quando si esegue il precedente statement, allora viene creato un gruppo attivi composto da questi 3 thread (a cui vengono assegnati i rank 0,1,2 nel gruppo). Si noti che ogni warp avrà il suo gruppo attivi.

Un esempio di uso dei coalesced groups. Supponiamo che in una certa situazione tutti i thread di posizione dispari in un warp divergano dagli altri. Possiamo creare un gruppo di tali thread e sincronizzare solamente loro:

```
auto block = this_thread_block();
if (block.thread_rank() % 2) {
    coalesced_group attivi = coalesced_threads();
    ...
    attivi.sync();
}
```

Grid group

Vediamo dei cenni su grid group: un meccanismo che permette di imporre sincronizzazione tra i thread della grid.

La sincronizzazione nella grid prevede di utilizzare un particolare tipo di gruppo: grid_group
grid = this_grid(); e poi di eseguire: grid.sync();

Vi sono alcuni requisiti relativi al modo di compilare il codice e al modo di lanciare i kernel (si veda la CUDA Programming Guide per i dettagli). In particolare, per lanciare il kernel invece della sintassi <<<...>>> è necessario utilizzare la funzione

```
cudaLaunchCooperativeKernel(...)
```

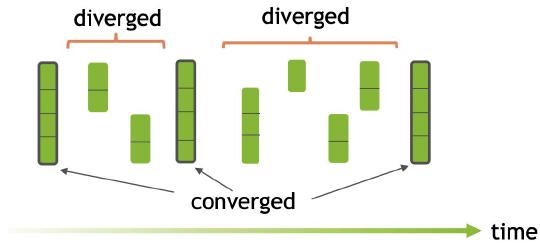
4.1.3 Warp-level synchronization

Breve reminder: in una GPU CUDA l'esecuzione procede creando, gestendo schedulando ed eseguendo i thread in gruppi di 32 thread, i warp. I thread in un warp possono divergere e ri-convergere durante l'esecuzione. La massima efficienza si ha quando tutti i thread del warp convergono:

Warp programming model

Il warp synchronous programming è una tecnica di programmazione CUDA che sfrutta il modo di esecuzione/gestione dei warp per realizzare efficientemente la comunicazione tra thread.

Ad esempio nell'eseguire reduction, scan, aggregated atomic operation, ecc. CUDA supporta warp synchronous programming fornendo warp synchronous built-in functions e cooperative group collective operations.



La comunicazione tra thread deve rispettare un vincolo: un thread T1 può accedere a dati locali di un altro thread T2 (dello stesso warp) solo se quest'ultimo è attivo e partecipa con T1 alla operazione di shuffle (shuffle command o shuffle function).

Se T2 è inattivo allora il risultato dell'operazione non è definito I thread in un warp sono identificati dalla lane e posseggono un ID (un intero tra 0 e warpSize-1).

Funzioni built-in

Activemask query:

- `unsigned __activemask()` : restituisce un intero i cui bit identificano gli attuali thread attivi del warp

Synchronized data exchange: I thread di un warp comunicano in modo sincrono:

- `__all_sync(unsigned mask, predicate)`
- `__any_sync(unsigned mask, predicate)` : valuta il predicato e restituisce un valore non nullo se tutti/qualche thread ottengono valore non nullo
- `__ballot_sync(unsigned mask, predicate)`: valuta il predicato e restituisce un intero i cui bit a 1 indicano i thread che sono attivi ed hanno ottenuto valore non nullo

L'argomento mask indica quali thread del warp devono partecipare alla operazione che viene eseguita via hardware. Tali thread devono essere convergenti nel momento in cui eseguono l'operazione e la devono eseguire con la stessa mask. Altrimenti il risultato non è definito

Synchronized data exchange: I thread di un warp comunicano in modo sincrono:

- `unsigned int __match_any_sync(unsigned mask, T value)`
- `unsigned int __match_all_sync(unsigned mask, T value, int *pred)` : effettuano una sincronizzazione e una operazione broadcast-and-compare tra i thread della mask. Compara se qualcuno/tutti i thread hanno lo stesso valore per value. Restituisce la bit map del risultato. (pred sarà vero se tutti hanno lo stesso valore)

Thread synchronization. sincronizzazione con imposizione di memory fence:

- `__syncwarp()`

Synchronized data exchange: I thread di un warp comunicano in modo sincrono:

- `T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize)`
- `T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize)`
- `T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize)`
- `T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize) : (dove T può essere int, long, unsigned int, float, etc.)`

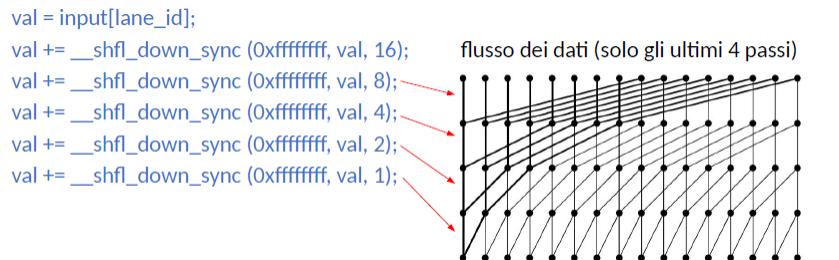
Permettono lo scambio dei (valori di) una variabile locale var (a 4 o 8 byte) tra thread dello stesso warp senza ricorrere alla shared memory. Lo scambio avviene simultaneamente tra tutti i thread attivi indicati in mask che eseguono la funzione

- `__shfl_sync` : copia diretta del valore di var posseduto dal thread della lane srcLane
- `__shfl_up_sync` : copia la var posseduta dal thread della lane precedente (al thread che esegue)
- `__shfl_down_sync` : copia la var posseduta dal thread della lane successiva
- `__shfl_xor_sync` : copia la var del thread identificato dallo xor tra lane del chiamante e laneMask width e delta possono essere usati per selezionare porzioni del warp o per accedere a thread posizionati delta lanes in avanti o indietro (invece del successivo o precedente).

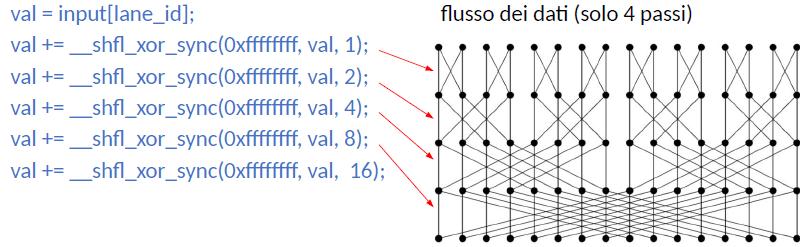
Esempio:

Problema: sommare i 32 elementi di un array `input[]`.

Versone A: con `__shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize)`; Tutti i 32 thread del warp eseguono:



Versione B: con `__shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize)`; Tutti i 32 thread del warp eseguono:



4.2 Parallelismo dinamico in CUDA

Abbiamo visto che risulta facile trasformare un loop di un codice seriale in un kernel (ad es. saxpy).

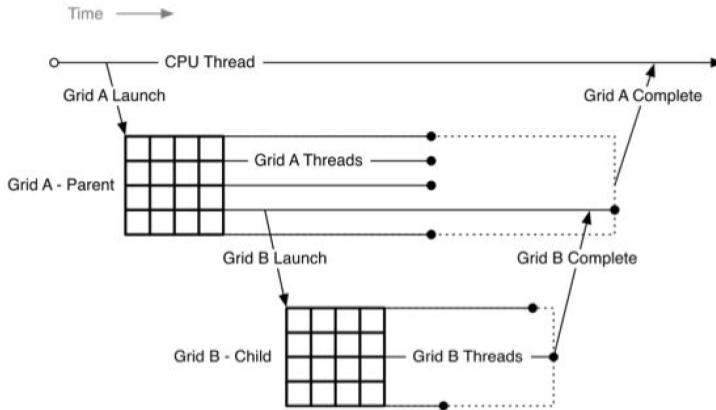
Ci sono casi però in cui la “trasformazione” non è semplice. Per esempio in casi in cui le parti di codice parallelizzabili sono nidificate. In generale vi sono molte situazioni in cui risulta difficile esporre il parallelismo “nascosto” nella soluzione seriale:

- algoritmi che usano strutture dati gerarchiche
- algoritmi risorsivi in cui ogni chiamata ricorsiva esegue del codice che presenta potenziale parallelismo
- algoritmi in cui si individuano facilmente porzioni parallelizzabili che però non sfruttano pienamente la GPU

Il dynamic parallelism di CUDA può rappresentare uno strumento utile da impiegare in questi casi.

Ricordiamo che un kernel lancia una grid composta da blocks e ogni block è composto da un insieme di thread. Chiameremo questa grid parent grid.

L’idea del dynamic parallelism è di consentire ad un thread di lanciare a sua volta un kernel, ovvero di lanciare una child grid.



- In generale, ogni thread della parent grid genera una child grid.
- Vi è una implicita sincronizzazione tra parent e child grid: la parent grid viene considerata terminata solo se tutte le child grid sono completate

- Il lancio è asincrono: il thread invoca la child grid e poi prosegue, la grid viene schedulata nel suo stream (e sarà eseguita quando ci saranno le risorse necessarie)

4.2.1 Esempio:

Scriviamo un kernel che:

- si richiama ricorsivamente
- ad ogni chiamata stampa un messaggio per visualizzare il livello di nesting
- poi si richiama con la stessa configurazione

Si noti che la chiamata ricorsiva sarà effettuata da tutti i thread del kernel chiamante, quindi per limitare il numero di kernel in esecuzione limiteremo:

- la dimensione della grid (grid 1D di due blocchi con blocchi 1D di due thread)
- il numero di chiamate ricorsive nidificate

Codice del kernel ricorsivo:

```
_global__ void kernelRicorsivo(int max_depth, int depth, int thread, int parent_uid) {
    // Ogni block genera un ID unico: il thread 0 incrementa in mutua esclusione una variabile globale in global memory
    // e condivide l'ID ottenuto con gli altri thread del blocco usando una variabile in shared memory
    __shared__ int s_uid;
    if (threadIdx.x == 0) { s_uid = atomicAdd(&g_uids, 1); }
    __syncthreads(); // barrier: tutti i thread aspettano per conoscere l'ID del loro blocco

    // Stampa info sul blocco e il chiamante
    print_info(depth, thread, s_uid, parent_uid);

    // Se abbiamo raggiunto il limite di chiamate nidificate i thread del block terminano
    if (++depth >= max_depth) { return; }
    // altrimenti ogni thread del blocco effettua una call ricorsiva
    // (con la stessa configurazione della grid e depth incrementato):
    kernelRicorsivo<<<gridDim.x, blockDim.x>>>(max_depth, depth, threadIdx.x, s_uid);
}
```

Funzione che stampa le informazioni:

```
__device__ void print_info(int depth, int thread, int uid, int parent_uid) {
    if (threadIdx.x == 0) { // solo un thread stampa
        if (depth == 0) printf("BLOCK %d lanciato da host\n", uid);
        else printf("BLOCK %d lanciato dal thread %d del block %d\n", uid, thread, parent_uid);
    }
    __syncthreads();
}
```

Codice Host:

```
__device__ int g_uids = 0; // dichiarazione della variabile globale in global memory

int main() {
    printf("Inizio main\n");
    int max_depth = 3; // fisso limite al nesting della ricorsione
    // Impostiamo max_depth
    cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, max_depth);

    // Lancia del kernel dalla CPU
    printf("Lancia il kernel kernelRicorsivo() che usa CUDA Dynamic Parallelism:\n\n");
    kernelRicorsivo<<<2, 2>>>(max_depth, 0, 0, -1);
    cudaDeviceSynchronize(); // attendo fine del kernel prima di terminare
    exit(EXIT_SUCCESS);
}
```