

# HPC

Gianluca

18-06-2021

## Contents

<b>1</b>	<b>Introduzione</b>	<b>5</b>
1.1	Definizione . . . . .	5
1.2	HPC vs Supercomputing . . . . .	5
1.3	Embedded Systems . . . . .	5
<b>2</b>	<b>The Multicore Revolution</b>	<b>7</b>
2.1	Power wall . . . . .	7
2.2	ILP wall . . . . .	9
2.3	Memory wall . . . . .	11
2.4	Transition to multicore . . . . .	11
2.5	Utilization wall . . . . .	13
2.6	The four horsemen . . . . .	13
2.6.1	The shrinking horseman . . . . .	13
2.6.2	The dim horseman . . . . .	13
2.6.3	The specialized horseman . . . . .	14
2.6.4	Deus ex machina horseman . . . . .	14
<b>3</b>	<b>Introduction to Parallel Systems</b>	<b>14</b>
3.1	Tassonomia delle architetture parallele (Flynn) . . . . .	15
3.1.1	SISD . . . . .	15
3.1.2	MISD . . . . .	15
3.1.3	SIMD . . . . .	16
3.1.4	MIMD . . . . .	16
3.2	Architetture SIMD . . . . .	17
3.3	Architetture MIMD . . . . .	19
3.3.1	Shared memory . . . . .	20
3.3.2	Implementazione fisica . . . . .	21
3.3.3	Distributed memory . . . . .	21

3.3.4	Implementazione fisica . . . . .	22
<b>4</b>	<b>Novel Issues in Shared-Memory Multicore Systems: Coherence</b>	<b>23</b>
4.1	Protocollo MSI . . . . .	24
4.2	Protocollo MESI . . . . .	25
4.3	Performace nei SMMP . . . . .	25
4.4	Implementazione protocollo di coerenza . . . . .	26
4.4.1	Snoopy broadcast . . . . .	26
4.4.2	Directory . . . . .	26
<b>5</b>	<b>Novel Issues in Shared-Memory Multicore Systems: Synchronization</b>	<b>28</b>
5.1	Lock . . . . .	28
5.2	Criteri di performance . . . . .	29
5.2.1	Lock basato sulla test&set . . . . .	29
<b>6</b>	<b>Novel Issues in Shared-Memory Multicore Systems: Consistency</b>	<b>30</b>
6.1	Sequential consistency model . . . . .	31
6.2	Relaxed consistency model . . . . .	31
6.3	Weak consistency model . . . . .	31
<b>7</b>	<b>Hardware Architectures for HPC on-chip: CPU</b>	<b>31</b>
7.1	L'evoluzione dei multicore . . . . .	32
7.2	Il modello big.LITTLE . . . . .	32
7.2.1	Cluster migration model . . . . .	33
<b>8</b>	<b>Hardware Architectures for HPC on-chip: GPU</b>	<b>33</b>
8.1	Slimmed design . . . . .	34
8.2	SIMD processing . . . . .	35
8.3	Execution context interleaving . . . . .	36
8.4	Gerarchia di memoria GPU . . . . .	36
<b>9</b>	<b>Hardware Architectures for HPC on-chip: FPGA</b>	<b>37</b>
9.1	Componenti delle FPGA . . . . .	38
<b>10</b>	<b>Understanding Performance</b>	<b>39</b>
10.1	Coverage . . . . .	40
10.1.1	Scaling . . . . .	40
10.2	Granularity . . . . .	40

10.2.1	Load balancing statico . . . . .	41
10.2.2	Load balancing dinamico . . . . .	41
10.2.3	Metriche di performance . . . . .	41
10.3	Locality . . . . .	42
10.3.1	Distributed Shared Memory . . . . .	42
<b>11</b>	<b>Parallel Programming in Practice</b>	<b>45</b>
11.1	Decomposizione . . . . .	46
11.1.1	Analisi delle dipendenze . . . . .	47
11.2	Assegnamento . . . . .	47
11.2.1	Assegnamento statico . . . . .	47
11.2.2	Assegnamento semi-statico . . . . .	47
11.2.3	Assegnamento dinamico . . . . .	48
11.3	Orchestration . . . . .	48
11.4	Mapping . . . . .	48
11.4.1	Performance analysis . . . . .	49
<b>12</b>	<b>OpenMP</b>	<b>50</b>
12.1	Componenti OpenMP . . . . .	51
12.2	Direttiva "parallel" . . . . .	51
12.3	Direttiva "for" . . . . .	52
12.4	Clausola "schedule" . . . . .	53
12.5	Direttiva "barrier" . . . . .	54
12.6	Direttiva "critical" . . . . .	54
12.7	Direttiva "master" e "single" . . . . .	55
12.8	Task . . . . .	55
<b>13</b>	<b>OpenMP Accelerator Model</b>	<b>57</b>
13.1	Terminologia . . . . .	58
13.2	Data mapping . . . . .	59
<b>14</b>	<b>CUDA Basics</b>	<b>61</b>
14.1	Execution model . . . . .	61
14.2	CUDA programming . . . . .	63
14.3	CUDA driver vs Runtime API . . . . .	66
<b>15</b>	<b>CUDA Memory Model</b>	<b>66</b>
15.1	Global Memory . . . . .	67
15.2	Shared Memory . . . . .	68
15.2.1	Thread synchronization . . . . .	69

15.3	Constant Memory . . . . .	70
15.4	Registri . . . . .	71
15.4.1	Local memory . . . . .	71
15.5	Grid Synchronization . . . . .	71
<b>16</b>	<b>CUDA Advanced Features</b>	<b>71</b>
16.1	Pageable and Pinned Memory . . . . .	72
16.2	Unified Virtual Memory . . . . .	73
16.3	Asynchronous Data Transfer . . . . .	73
16.3.1	Sincronizzazione Esplicita . . . . .	74
16.3.2	Sincronizzazione Implicita . . . . .	74
16.3.3	Gestione CUDA Streams . . . . .	74
16.4	Dynamic Parallelism . . . . .	75
16.5	Summary of Key Performance Issues . . . . .	76
16.5.1	Shared Memory . . . . .	76
16.5.2	Control Flow . . . . .	76
16.5.3	Memory Coalescence . . . . .	77
16.5.4	SoA vs AoS . . . . .	78
16.5.5	Latency Hiding . . . . .	79
16.5.6	Occupancy . . . . .	79
<b>17</b>	<b>Reconfigurable FPGA-based System Design</b>	<b>79</b>
17.1	Technology Mapping . . . . .	81
17.2	FPGA Bitstream . . . . .	83
<b>18</b>	<b>High-Level Synthesis (HLS)</b>	<b>83</b>
<b>19</b>	<b>Optimizing Compilers for Heterogeneous Systems</b>	<b>86</b>
19.1	Scanner . . . . .	88
19.2	Parser . . . . .	88
19.3	Semantic Routines . . . . .	88
19.4	Code Generator . . . . .	88
19.5	Tipi di IR . . . . .	88
19.6	Ottimizzazione . . . . .	89
19.6.1	Constant Folding . . . . .	90
19.6.2	Constant Propagation . . . . .	90
19.6.3	Copy Propagation . . . . .	90
19.6.4	Strength Reduction . . . . .	90
19.6.5	Dead Code Elimination . . . . .	90
19.7	SSA . . . . .	90

# 1 Introduzione

## 1.1 Definizione

Cos'è l'*high performance computing*? Definizione formale: pratica di aggregare potenza di calcolo in maniera tale da riuscire a fornire una performance che è molto più alta di quella di un normale computer/workstation. La sua finalità è quella di risolvere problemi complessi: in termini computazionali o in termini di dimensioni dei data set. Fa riferimento ad una serie di tecnologie utilizzate da computer cluster per creare sistemi di elaborazione in grado di fornire prestazioni elevate, tipicamente ricorrendo al calcolo parallelo.

## 1.2 HPC vs Supercomputing

Storicamente c'è un parallelo tra queste due pratiche. Il discorso di aggregare potenza di calcolo (o mettere insieme più nodi di calcolo) è legato da un punto di vista applicativo a quei domini nei quali la potenza di calcolo elevata è indispensabile per essere eseguiti in tempi ragionevoli o per migliorarne l'accuratezza (es: modellare fenomeni complessi). I supercomputer hanno problematiche differenti, sfruttano delle tecniche di interconnessione che hanno latenze per le quali anche i paradigmi di programmazione (come partizionano il problema tra i vari nodi e come mi preoccupo della latenza di comunicazione) sono molto particolari e diversi da quelli **SoC** (system-on-chip), su cui ci concentriamo. Ci sono dei punti in comune nello sviluppo del singolo nodo: si utilizzano i paradigmi di **parallelismo** ed **eterogeneità**. I due domini si differenziano anche sul tipo applicazioni che ci girano sopra.

## 1.3 Embedded Systems

Programmazione parallela argomento di nicchia fino al 2005, nonchè costosa. Oggi i processori paralleli sono ovunque, nei cosiddetti sistemi embedded. Un **embedded system** è un dispositivo che include un computer programmabile, ma non è un computer general purpose. Inizialmente sistemi sviluppati per svolgere un compito in particolare (lavatrice, treno...) quindi venivamo progettati sfruttando le caratteristiche dell'applicazione in modo da ottimizzarne il design. Nei general purpose computer abbiamo la gerarchia di memoria, l'IS, il sistema di memoria virtuale, etc. In un sistema embedded rimuoviamo le parti che non sono indispensabili e riduciamo così i consumi (spesso vanno a batteria) e i costi, rendendoli sistemi estremamente *predicibili*. IoT: sistemi equipaggiati per computazione e soprattutto con abilità di connessione. Embedded artificial intelligence: auto a guida

autonoma, droni, robot... Cosa rende possibile questa rivoluzione? Sfruttiamo la definizione di Gordon Bell: *"ogni 10 anni abbiamo a disposizione una nuova classe di computer dal un prezzo inferiore, basata su una nuova piattaforma di programmazione, un nuovo sistema di rete ed interfaccia tali da rivoluzionare il mercato"*. Ci sono 4 elementi che rendono vero questa "legge":

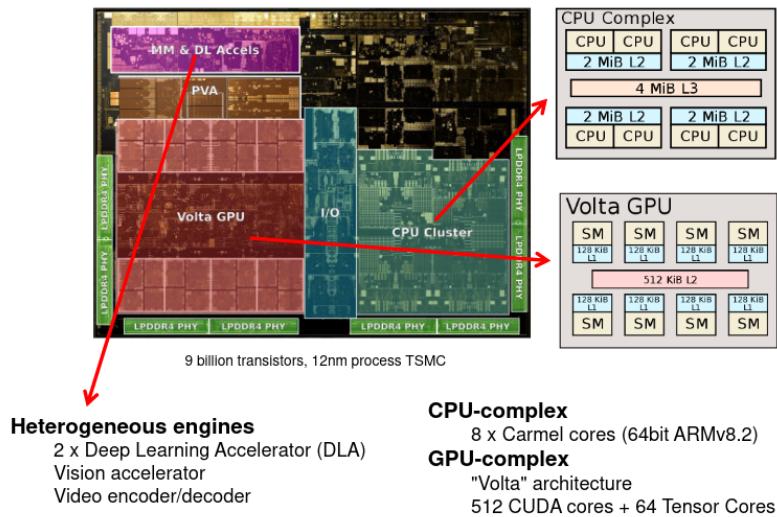
1. la tecnologia che utilizziamo, in particolare il **transistor count**: il numero di transistor che, di generazione in generazione, riusciamo ad integrare su uno stesso chip/circuito integrato. **Legge di Moore**: ogni ~18 mesi il numero di transistor che riusciamo ad integrare su un unico pezzo di silicio raddoppia. A parità di area avrò funzionalità più sofisticate.
2. **Miniaturizzazione**: riusciamo ad integrare il doppio dei transistor perchè la fisica che sta dietro alla stampa dei circuiti integrati è tale per cui rendono sempre più piccolo il transistor a parità di funzionalità. Circuiti più piccoli, interconnessione più piccole, integrazione del sistema più piccola, etc. Su un unico chip non ho solo la CPU ho un intero sistema!
3. Costo della tecnologia: diminuisce, in accordo con Bell.
4. Sviluppo delle tecnologie di comunicazione: range (capacità di raggiungere distanze) e banda maggiore.

I paradigmi che hanno reso possibile questa rivoluzione sono:

- l'elevato parallelismo
- l'eterogeneità: ci sono diversi attori che fanno calcolo
- core organizzati in piccoli cluster (on chip)

Si parla di omogeneità o eterogeneità quando vogliamo precisare il fatto che l'organizzazione del sistema di calcolo sfrutta risorse che sono tra di loro uguali o meno. Spesso localmente un sottosistema è omogeneo, ma globalmente è eterogeneo (e.g. NVIDIA Xavier): a livello di sistema, oltre al complesso CPU, abbiamo quello GPU, batterie di acceleratori dedicati (accelerazione video, machine/deep learning).

## NVIDIA Xavier (sept, 2016)



## 2 The Multicore Revolution

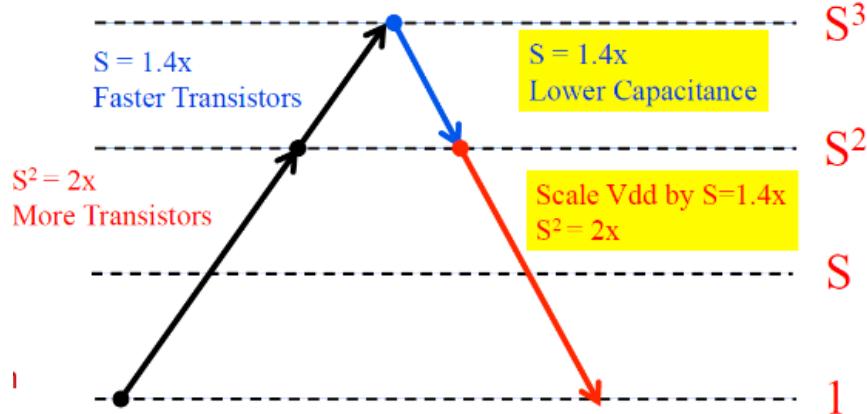
### 2.1 Power wall

Dal 1986 al 2002 c'è una crescita costante di performance che si attesta intorno al 50%. Dopo il 2002 questo trend inizia a rallentare fino ad arrivare ad un aumento del 6%. Ci si rende conto che iniziano ad esserci dei benefici sempre minori dalla maniera in cui si progettano i processori (all'epoca basati su un singolo core). Le metodologie di progetto sono tutte improntate a cercare di fare andare sempre più veloce il processore e cercando di sfruttare il parallelismo interno ai programmi, il cosiddetto **instruction level parallelism**. Dal 2006 in poi si passa al multicore! Il tutto nasce dalla già citata legge che governa l'evoluzione della tecnologia di integrazione su silicio, la **legge di Moore**: *"Di generazione tecnologica in generazione il numero di transistor raddoppia ogni 18 mesi"*. Cos'è che ha fermato l'incremento di performance legato all'incremento dei transistor? E' dipeso da un fenomeno fisico: con la miniaturizzazione della tecnologia dei dispositivi tutta l'attività di switching dei transistor veniva convertita in calore, calore che si faceva sempre più faticosa a dissipare. Introduciamo un'altra legge, quella di Dennard (1974): *"La dimensione dei transistor si può ridurre dello 0.7x ogni generazione tecnologica"*. C'è quindi una riduzione del delay di 0.7x e dunque un incremento

della frequenza operativa del 1.4x. La formula che governa il consumo di potenza di un trasistor che fa *switching activity* è:

$$P = C \cdot V^2 \cdot f \quad (1)$$

dove  $P$  (la potenza) è direttamente proporzionale a  $C$  (il carico capacitivo) che è costante per un certo nodo tecnologico e a  $V$  (il voltaggio) e ad  $f$  (la frequenza). Aumentando dunque la frequenza ci sarà sicuramente una diretta conseguenza sul consumo di potenza dei transistor. Dennard dice che in realtà c'è un bilancio energetico per il quale possiamo garantire che l'aumento incontrollato della potenza non avverrà. Riducendo il carico capacitivo di 1.4x e riducendo il voltaggio di 1.4x ( $V$  nella formula è al quadrato ottengo 2x) ottengo un bilanciamento perfetto. Ricapitolando il 2.8x di aumento di switching activity è bilanciato dal decremento di 2.8x del consumo di potenza per transistor.



In realtà, quello che si osserva all'inizio degli anni 2000 è qualcosa che fino a quel momento non era stato particolarmente significativo. La potenza è data da due componenti: dinamica e statica.

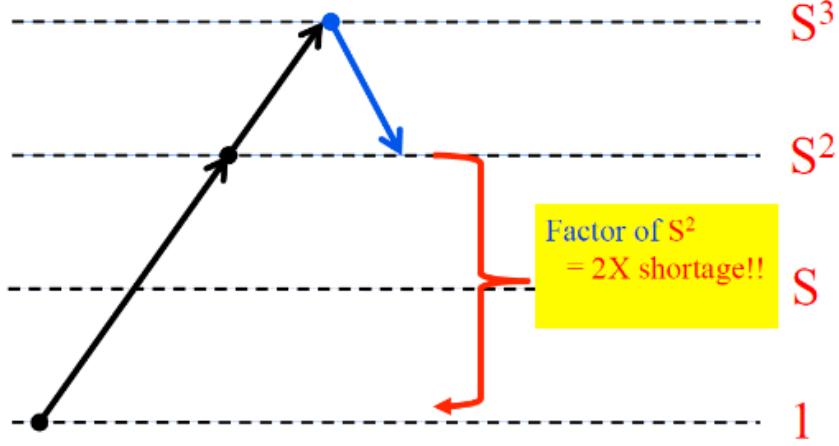
$$P_{tot} = P_{dyn} + P_{static} \quad (2)$$

$$P_{static} = k_d \cdot V \cdot Current_{leak} \quad (3)$$

$$P_{dyn} = C \cdot V^2 \cdot f \quad (4)$$

La **corrente di leakage** è sempre presente anche quando i transistor non stanno facendo switching, basta che il circuito sia alimentato. Prima il contributo della corrente statica era trascurabile, ma man mano che si

miniaturizza questo contributo diventa sempre più significativo al punto in cui non sono più in grado di scalare il voltaggio secondo le previsioni di Dennard e l'equazione non è più bilanciata. Mi rimane un fattore 2x che non riesco più a dissipare. Vado incontro al cosiddetto **power wall**.



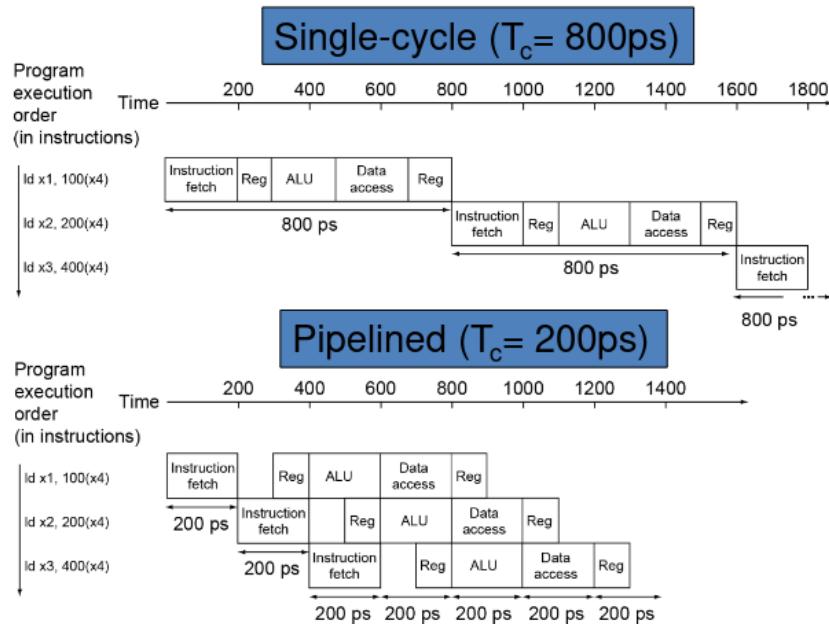
## 2.2 ILP wall

I single core sfruttano l'instruction level parallelism per incrementare la performance. Una qualunque CPU è organizzata come una rete logica che svolge un certo numero di operazioni, il minimo sindacale è:

1. *instruction fetch*
2. fase di *decode*, dove si accede al register file
3. la fase di *execute* vera e propria, dove usiamo la ALU per calcolare il risultato
4. una fase di *data access*, nella quale accediamo se necessario alla memoria in lettura o scrittura
5. una fase di *writeback* dove riscriviamo il risultato sul *register file*

Se la CPU non sfrutta un design pipelined io ho un'operazione per ciclo di clock e questo comporta che per eseguire un'istruzione in un ciclo sono vincolato a mettere una dietro l'altra tutto le fasi che compongono l'esecuzione di questa istruzione; quindi il mio tempo di clock deve essere grande abbastanza da accomodare tutte queste fasi. Da un punto di vista fisico il segnale elettrico si deve propagare lungo una lunga rete combinatoria

e quindi il mio clock cycle deve essere largo abbastanza. Un design pipelined ci aiuta a ridurre il tempo di clock: metto dei registri tra una fase e l'altra dell'esecuzione della mia istruzione, dunque il tempo di clock non è più quello che serve perché il segnale elettrico si propaghi attraverso tutte queste fasi, ma perchè si propaghi attraverso la più lunga di queste fasi. La latenza per eseguire la singola istruzione aumenta, ma è migliorato il **throughput** (numero di istruzioni che eseguo per singolo ciclo). Quando la pipeline sarà completamente piena potrò avere tante istruzioni che eseguono quanti sono gli stadi.



Più profonda è la pipeline (i.e. più stadi ha), minore diventa il periodo di clock, dunque maggiore sarà la frequenza operativa. Inoltre c'è anche il paradigma di design detto **multiple issue**: il mio processore è in grado di eseguire più di una istruzione nello stesso ciclo perchè c'è parallelismo, ovvero, se ho due istruzioni che non dipendono l'una dall'altra le eseguo in parallelo, replicando la pipeline. Questo fa sì che io possa cominciare ad eseguire più di una istruzione per clock cycle, quindi il CPI (clock per instruction, senza pipeline = 1) diventa <1. Nella realtà le dipendenze impediscono che le pipeline siano sfruttate al massimo. La multiple issue si può gestire in maniera statica o dinamica:

- statica: il compilatore (o programmatore paziente) scrive il programma specificando lo schedule delle istruzioni, risolve tutte le dipendenze e

dice alla CPU in che ordine eseguire (processori detti *very long instruction word*).

- dinamica: si usa quando il valore dei dati viene stabilito a runtime. Vengono introdotti i **superscalar processors**, i transistor li uso per creare un blocco logico che capisce a runtime quante istruzioni posso mandare in esecuzione, evitando le dipendenze. Le ultime evoluzioni di single core riguardavano questi blocchi di scheduling della pipeline, che si chiamano **out-of-order** scheduler. Questo hw è molto sofisticato, utilizza un alto numero di transistor che consumano potenza dinamica per riuscire ad eseguire quante più istruzioni contemporaneamente possibili. Si osserva che l'out-of-order scheduling consuma molta potenza.

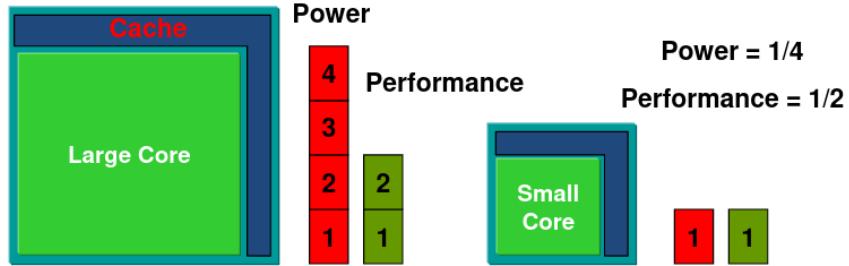
Non ho più instruction level parallelism di quello che il mio processore riesce a sfruttare: ILP wall.

### 2.3 Memory wall

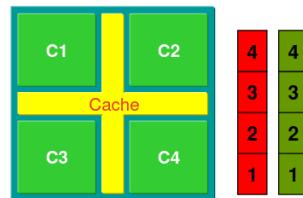
Se guardiamo ai primi microprocessori (1980) questi non hanno una cache. Nel 1995 si arriva ad avere 2 livelli di cache. Questo è sintomatico di un fenomeno per il quale la velocità con cui le CPU (l'integrazione di transistor su singolo chip) cresce con una certa velocità in termini di performance mentre la DRAM cresce molto più lentamente. Il gap tra le due cresce del 50% all'anno. Se la CPU ci mette 1 anno e mezzo per diventare 2 volte più potente la DRAM ce ne mette 10. Ci si rende conto che c'è bisogno di dedicare più transistor a migliorare il funzionamento del sottosistema di memoria. L'approccio multicore consiste nel tener conto di tutti i muri tecnologici che abbiamo visto e cambiare la maniera in cui i transistor vengono utilizzati.

### 2.4 Transition to multicore

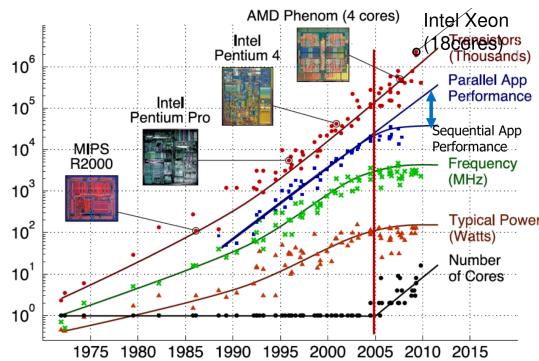
L'idea chiave è: processore più semplice (meno transistor), più lento e meno power-demanding. Si studiano l'incremento di perfomance comparato all'incremento di consumo di potenza attraverso underclock e overclock di un processore. Si osserva che se overclocko il di 20% ho 13% di incremento in performance e 73% di consumi in più. Se underclock si ha una riduzione della performance di 13% ma una riduzione di potenza del 50%.



Idea: piuttosto che lavorare con la frequenza posso impilare due core underclockati e ottengo lo stesso incremento in performance dell'overclock e un incremento in power comsumption che è trascurabile rispetto al single core. Il multicore è più *power-efficient* e c'è una migliore possibilità di gestire la problematica della potenza e quella termica (spegnere core che non lavorano).



La transizione al multicore porta ad un plateau della curva di consumo di potenza e della frequenza, mentre i transistor continuano a crescere. La curva della performance cresce o va verso un plateau a seconda del fatto che io stia sfruttando il parallelismo.



## 2.5 Utilization wall

Il multicore ha risolto tutti i problemi? La promessa era quella di avere un incremento nel numero di core di un fattore 2x ad ogni generazione con crescita di frequenza più o meno invariata. In realtà nel salto tra il nodo da 65nm a quello di 32nm (due generazioni) il numero di core passa da 4 a 8 (anzichè 16) perchè il resto dell'area del mio chip rimane *dark* o *dim*. L'incremento del numero di core in verità aumenta del 1.4x ogni generazione. Il problema è sempre quello della dissipazione di potenza (scaling di Dennard). Il fattore di perdita del recupero di un 2x in power non è sparito coi multicore, il multicore mi ha aiutato ad usare più core meno potenti e a non aumentare la frequenza per aumentare la performance. Si arriva a scontrarsi contro l'**utilization wall**: ho sempre il doppio dei transistor, ma non posso usarli perchè non riuscirei a gestire l'incremento di generazione di calore. C'è sempre più *dark silicon* (silicio scuro) sui chip.

## 2.6 The four horsemen

Si pensa a quattro possibili soluzioni per l'utilizzo del dark silicon.

### 2.6.1 The shrinking horseman

L'area è costosa: costruiamo dei chip più piccoli anzichè includere dark silicon nei nostri progetti. Il dark silicon però non è necessariamente inutile, significa che non lo uso tutto il tempo o che è underclockato. L'importante è che non vengano usati tutti tutto il tempo. Anche all'epoca le prime GPU integrate su singolo chip avevano zone dark, che venivano utilizzate all'occorrenza. Un altro esempio è la cache di terzo livello: normalmente se il sistema è ben progettato, il working set tipico dei programmi che girano sui core è tale da poter essere ospitato all'interno della cache di secondo livello. Quella di terzo è usata sporadicamente, ad esempio all'inizio quando abbiamo le call di cache miss, ma poi la maggior parte del traffico di memoria si gestisce tra il primo e secondo livello.

### 2.6.2 The dim horseman

Introduciamo il concetto di *dimming*: mettiamo sul chip tanti core che alla massima frequenza eccederebbero il power budget, ma poi li underclockiamo (**spatial dimming**) o li usiamo a burst (**temporal dimming**). Esempi di spacial dimming sono i processori multicore di prima generazione, seguivano

la regola "aumentiamo il numero di core". L'esempio classico del temporal dimming ce l'abbiamo nell'organizzazione BIG.little delle CPU general purpose che ci sono su questi SoC.

### 2.6.3 The specialized horseman

Mi rendo conto che in un chip moderno le cose che devo fare che sono tante: uso il silicio per mettere nel chip tante unità di calcolo, non solo general purpose processors. Queste risorse di calcolo devono essere specializzate nel fare una cosa e avranno una maggiore efficienza. Questo è il concetto che sta alla base dell'eterogeneità architetturale: i nostri SoC hanno GPU, decoder/encoder video, acceleratori per reti neurali, etc.

### 2.6.4 Deus ex machina horseman

Concetto che risale al teatro greco: la trama dell'opera è così complicata che per risolverla arriva l'intervento divino. Tutte queste problematiche sono dovute alla tecnologia CMOS, ad un certo punto arriveremo alle dimensioni atomiche e la legge di Moore come la conosciamo non varrà più. Si inizia a ragionare su soluzioni che trascendono le limitazioni inerenti alla tecnologia

## 3 Introduction to Parallel Systems

Dall'avvento dei multicore si nota da una parte un plateau di frequenza e consumo di potenza dei processori, dall'altra la curva della performance si biforca: programmazione sequenziale vs programmazione parallela. Riepilogo pro e contro dell'utilizzo di sistemi multicore:

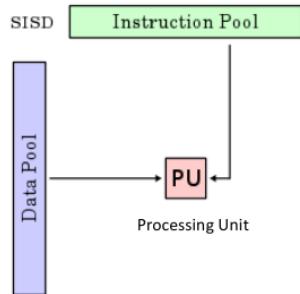
Vantaggi	Svantaggi
riuso dello stesso core più volte (scalabilità)	limitazioni della parallelizzazione (introduce overhead)
uso effettivo di miliardi di transistor	algoritmi e hardware limitano la performance
core meno potenti e meno costosi	necessario un nuovo modo di programmare: condividere/coordinare task su più processori

### 3.1 Tassonomia delle architetture parallele (Flynn)

		Data streams	
		Single	Multiple
Instruction Streams	Single	<b>SISD</b>	<b>SIMD</b>
	Multiple	<b>MISD</b>	<b>MIMD</b>

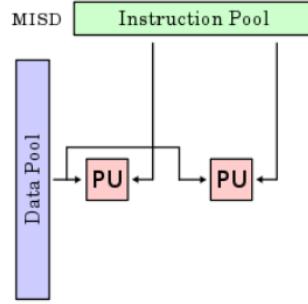
#### 3.1.1 SISD

Computer sequenziale che non sfrutta il parallelismo (e.g. single core fino al Pentium 4).



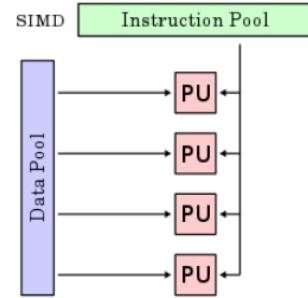
#### 3.1.2 MISD

Si sviluppa più di un percorso dalla instruction memory, possiamo alimentare in parallelo più di una PU (processing unit), ma il data pool resta unico. Queste PU eseguono porzioni diverse di una computazione su uno stream di dati unico. Non è completamente parallela, ma una sorta di pipelining: le PU eseguono in sequenza varie fasi di trasformazione di uno stream di dati (e.g. array processors).



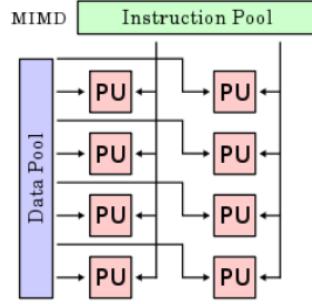
### 3.1.3 SIMD

C'è un'unica unità di instruction fetch & decode. Replico una parte del data-path: fase di *fetch&decode* comune mentre *execute* ed accesso alla memoria è replicata tante volte. Ogni parte replicata ha un accesso dedicato ai dati (ho quindi **data parallelism**) ed eseguo la stessa operazioni su dati differenti (e.g. GPU, CPU con estensioni vettoriali).



### 3.1.4 MIMD

Sfrutto il parallelismo sia nell'asse data sia in quello instruction: avrò streaming di istruzioni disgiunti (programma A e programma B in parallelo). Sfrutto processori autonomi che possono eseguire istruzioni diverse su dati diversi (e.g. computer general purpose attuali). All'interno di ciascun thread di esecuzione ho data parallelism.



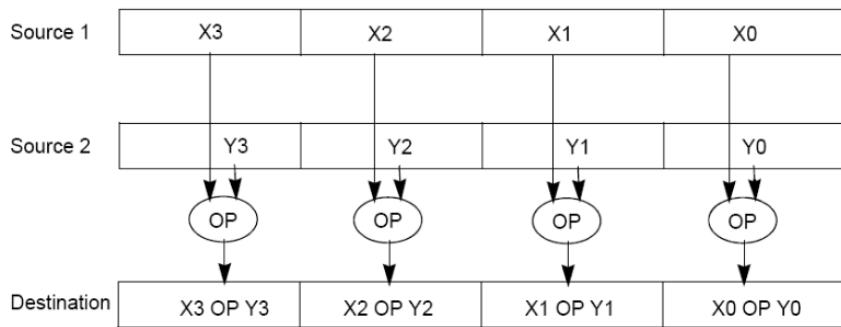
Dal 2011 SIMD e MIMD sono i modelli più utilizzati per i computer paralleli. Lo stile di programmazione più comune è **SPMD** (Single Program Multiple Data): un singolo programma che esegue su tutti i processori di un'architettura MIMD.

### 3.2 Architetture SIMD

SIMD come abbiamo detto utilizza un'unica istruzione (fetch&decode comune) e la distribuisce su più di un'unità (dataopath/ALU) che gestisce pezzi diversi di un dato in parallelo. Esempio: il filtering. Ho un vettore di dati che rappresentano i pixel di un'immagine e devo aggiungere ad ogni elemento uno scalare.

$$x[i] = x[i] + s, \quad 0 \leq i < 1000 \quad (5)$$

Tradizionalmente eseguirei 1000 iterazioni, una dopo l'altra. SIMD prende un'istruzione e la replica su un certo numero di dati.



Mi serve dell'hardware dedicato, progettato per avere dei registri sorgente con ampiezza pari a più di un dato, in base all'ampiezza dell'unità SIMD.

Per implementare il paradigma SIMD devo modificare il mio datapath perchè replichi parte del register file o abbia un register file dedicato che contenga registri più lunghi di quelli tradizionali. Inoltre devo creare una ALU che riesca a gestire in ingresso questi registri più larghi.

- SISD

```
for each i in array
{
    load x[i] to a register
    add scalar coefficient s
    write the result from the register to memory
}
```

- SIMD

```
for each 4 members in array
{
    load 4 members of x to the SIMD register
    calculate 4 additions in one operation
    write the result from the register to memory
}
```

Perchè funzioni ho bisogno di alcune accortezze:

- l'operazione dev'essere specificata su valori adiacenti in memoria
- devo srotolare il loop (**loop unrolling**) di almeno tante iterazioni quante sono quelle che ci servono per tenere occupata la nostra SIMD.

```
for(i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

diventa

```
for(i=1000; i>0; i=i-4)
{
    x[ i ] = x[ i ] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
    x[i-3] = x[i-3] + s;
}
```

Stiamo parlando di eseguire una frazione delle iterazioni originali del loop perchè ho:

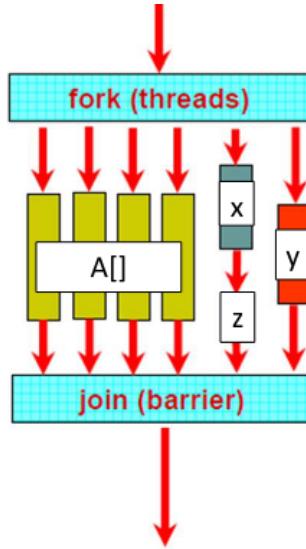
- una parte dei register file che ha registri larghi 4 (nell'esempio)
- una parte di ALU che riesce a fare la somma di 4 operazioni in un unico ciclo
- una parte di load store unit che riesce a riscrivermi 4 valori in un colpo solo nella memoria

In generale descriviamo un *loop unrolling* in questo modo: dato un loop di  $n$  iterazioni, un'unità SIMD di ampiezza  $k$ , srotolo il loop di un fattore  $k$  e genero  $k$  copie del loop body. Eseguiamo il loop body  $\lfloor \frac{n}{k} \rfloor$  volte.

### 3.3 Architetture MIMD

Queste architetture prevedono almeno due processori interconnessi da una rete di comunicazione. Il concetto di MIMD si focalizza sul **thread-level parallelism**: ho un certo numero di thread, ovvero dei contesti di esecuzioni di un programma disgiunti (ogni unità di esecuzione ha un program counter) che lavora in parallelo. Per  $n$  processori necessito di  $n$  thread. Bisogna fare attenzione e proporzionare adeguatamente la granularità di assegnazione di lavoro (quantità di computazione) dei thread, perchè gli overhead legati al parallelismo potrebbero superare i benefit. Le architetture MIMD supportano due tipi di parallelismo:

- **data parallelism**: esegue la stessa operazione, ma su dati differenti
- **task parallelism**: esegue funzioni differenti

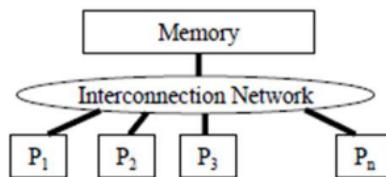


Per quanto riguarda la memoria esistono due principali paradigmi architetturali per i sistemi multicore:

- shared memory
- distributed memory

### 3.3.1 Shared memory

Fisicamente esiste solo una copia dei dati in memoria che è condivisa da più core. L'atomicità, il locking e la sincronizzazione sono essenziali per la correttezza e sono da gestire esplicitamente. Questa modalità di gestione è la più efficiente, ma genera problemi di **scalabilità**: il rischio che l'accesso alla memoria diventi un collo di bottiglia aumenta con il numero di processori che la condividono.



In questo caso la comunicazione avviene tramite variabili condivise: c'è un solo posto dove cercare una variabile - la memoria globale.

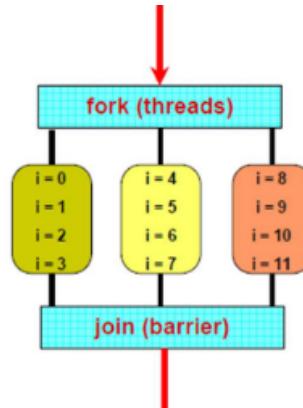
Esempio:

```

for (i=0; i<12; i++)
    C[i] = A[i] + B[i];

```

Un singolo processo può biforcarsi in più thread concorrenti; ogni thread ha risorse private (e.g. *i*) e risorse condivise (e.g. *A,B,C*).



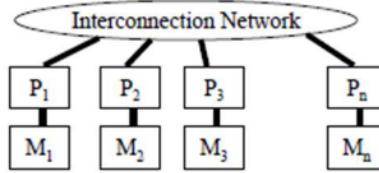
### 3.3.2 Implementazione fisica

La shared memory è una proprietà logica, dal punto di vista fisico posso implementarla in modi diversi:

- **physically shared** (SMP - Symmetric Multi-Processors): è il caso in cui la memoria è fisicamente condivisa e i core hanno lo stesso tempo di accesso alla memoria (**UMA** - Uniform Memory Access)
- **physically distributed** (DSM - Distributed Shared Memory): fisicamente la memoria è distribuita ma è implementata un'astrazione che consente di vedere la memoria come condivisa. Questo comporta che non tutti gli accessi in memoria avvengano con lo stesso tempo (**NUMA** - Non Uniform Memory Access).

### 3.3.3 Distributed memory

Ogni core ha una memoria locale alla quale accede più frequentemente ed in modo efficiente. La comunicazione con dati remoti (su memorie locali di altri processori) avviene in maniera esplicita.



La distribuzione dei dati e l'orchestrazione della comunicazione è essenziale per la performance. Questo design offre una maggiore scalabilità, ma comporta una perdita di efficienza. Si cerca di massimizzare la comunicazione tra i processori e la memoria locale, perché quella con le altre locazioni è lenta, siccome avviene con uno scambio di messaggi esplicito.

### 3.3.4 Implementazione fisica

Lo standard di comunicazione è il **Message Passing Interface** (MPI) e le primitive sono la `send()` e la `receive()`. Spesso sono bloccanti, ma possono usare la DMA per evitare di dare troppo workload alla CPU. Esempio: calcola la distanza da ogni punto in  $A[1..4]$  ad ogni punto in  $B[1..4]$  e memorizza il risultato in  $C[1..4][1..4]$ .

```

for(i=1; i<4; i++)
    for(j=1; j<4; j++)
        C[i][j] = distance(A[i], B[j])

```

Figure 1: programma sequenziale

```

A[n] = {};
B[n] = {};
Send(A[n/2 +1:n], B[1:n]);
for(i=1; i<n/2; i++){
    for(j=1; j<n; j++){
        C[i][j] = distance(A[i], B[j]);
    }
}
Receive(C[n/2 +1:n] [1:n]);

```

Figure 2: processore 1

```

A[n] = {};
B[n] = {};

```

```

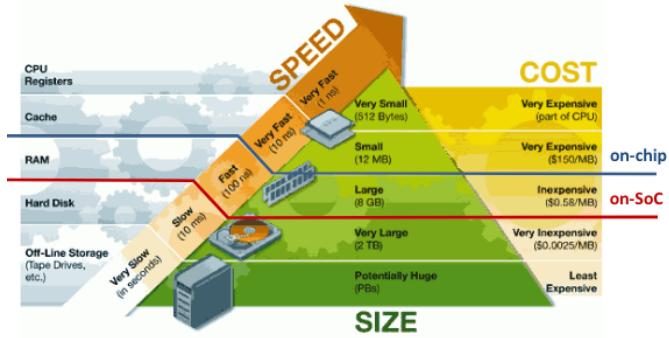
Receive(A[n/2 +1:n], B[1:n]);
for(i=n/2 +1; i<n; i++){
    for(j=1; j<n; j++){
        C[i][j] = distance(A[i], B[j]);
    }
}
Send(C[n/2 +1:n] [1:n]);

```

Figure 3: processore 2

## 4 Novel Issues in Shared-Memory Multicore Systems: Coherence

Tutto ha origine dal memory wall, ovvero il gap tra la crescita delle DRAM (7% ogni anno) e quello delle CPU (50% ogni anno). Questa problematica riguarda anche i sistemi multicore e porta alla necessità di trovare un compromesso tra velocità, costo e dimensione.



Oltre a questo tradeoff per i sistemi multicore si aggiungono altre tre problematiche:

- **coerenza** - vedo la copia più recente del dato?
- **sincronizzazione** - come proteggo gli accessi in memoria a dati condivisi?
- **consistenza** - quando risulta visibile un valore scritto da altri processori?

Partiamo della coerenza (i.e. *La coerenza prima di tutto*). Nei sistemi single core il problema sorge quando c'è un I/O che legge un dato che ha valori diversi tra cache e memoria. Nel contesto multicore il problema è quello di avere copie diverse in cache di processori diversi. Una definizione informale di coerenza potrebbe essere: "*una read deve restituire la write più recente*" È troppo stringente e difficile da implementare. Una definizione migliore sarebbe: "*ogni write deve essere prima o poi vista da una read*" Ovvero, una write seguita da una read dello stesso processore P senza write in mezzo ritorna il valore scritto, inoltre, ogni write sarà - con una certa tempistica - visibile da tutti. Ci servono alcune regole per assicurare la coerenza:

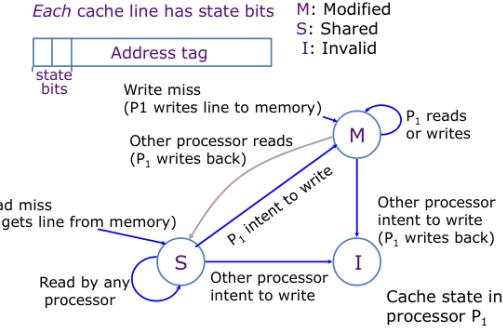
- se P1 scrive  $x$  e P2 lo legge, la write e la read devono essere sufficientemente distanti
- le write su una singola locazione sono serializzate e si vedrà l'ultima scritta.

Una prima idea è quella di dare al compito al sistema di interconnessione di invalidare le copie nelle cache in base ai tentativi di lettura e scrittura dei processori. Un meccanismo per implementare quest'idea è lo **snoopy cache coherence**: ho un BUS condiviso che vede tutte le transazioni che tutti i sistemi che possono leggere/scrivere in memoria effettuano. In caso di **write miss** l'indirizzo viene invalidato su tutti gli altri processori, mentre per le **read miss** se viene trovata una dirty copy in qualche cache deve essere fatto un writeback verso la DRAM prima che il processore possa leggere il dato.

## 4.1 Protocollo MSI

Un'implementazione del meccanismo di snoopy cache coherence è il protocollo MSI. Aggiungo ad ogni linea di cache due bit di stato per memorizzare tre stati:

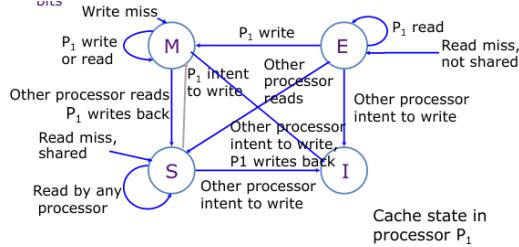
- M (modified): il processore ha nella cache il dato più recente, ma è l'unico ad averlo. È necessario fare un writeback nel caso altri processori provino a leggere/scrivere
- S (shared): il processore ha nella cache il dato più recente ed è lo stesso che è visto anche da gli altri processori
- I (invalid): il processore non ha nella cache il dato più recente



Problema: se è presente una cache nello stato M allora nessun'altra cache può avere una copia di quel dato. Ma è necessario che tutte le letture siano shared? Potrei avere anche dei dati privati.

## 4.2 Protocollo MESI

Si possono ottenere performance migliori per dati privati aggiungendo uno stato E (exclusive but unmodified).



## 4.3 Performance nei SMMP

La performance di un sistema SMMP (Shared Memory Multi Processor) è una combinazione di pattern di traffico di tipo cache tradizionale di un sistema a singolo core più il traffico aggiuntivo specifico del protocollo di coerenza. Alle miss di un sistema classico (*compulsory, capacity, conflict*) si aggiunge la *coherence* miss. Il traffico generato da coherence/ miss può essere causato da:

- **true sharing misses:** il dato è effettivamente invalido
- **false sharing misses:** il dato è teoricamente ancora valido, ma a causa della granularità delle linee di cache non è possibile distinguerlo da un altro dato che deve essere invalidato sulla stessa linea di cache (la dimensione delle linee > 1 word).

All'aumentare della dimensione della cache diminuiscono le compulsory, capacity, conflict miss, ma non le coherence miss. Quest'ultime aumentano invece con il numero di core.

#### 4.4 Implementazione protocollo di coerenza

Per implementare un sistema di cache coerenti devo trovare le risposte alle seguenti domande:

1. quando invoco il protocollo?
2. come trovo lo stato di un indirizzo di cache nelle altre cache?
3. come localizzo le altre copie del dato?
4. come comunico e agisco con le altre copie?

Per quanto riguarda la prima domanda il protocollo è invocato sempre allo stesso modo, ovvero quando il processore tenta di accedere ad una linea di cache (avviene un fault su quell'indirizzo). Le risposte alle domande 2, 3, 4 dipendono dalla specifica implementazione del protocollo di coerenza.

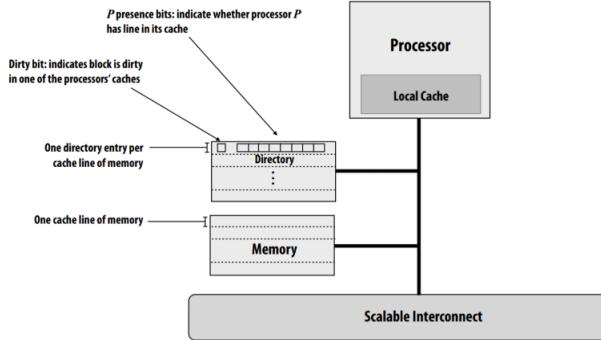
##### 4.4.1 Snoopy broadcast

Nei protocolli snoopy vengono fatti dei **broadcast**: il processore che ha l'esperienza di questo fault manda una richiesta di ricerca, dopodiché gli altri processori rispondono con le misure necessarie. Il broadcast è un'operazione costosa, sebbene semplice. Il BUS non scala e le informazioni devono propagarsi serializzate, con l'eventualità di un collo di bottiglia. Se invece ho un interconnection network scalabile dovrà generare più transazioni e quindi mi andrà ad aumentare la complessità. Il protocollo di coerenza deve evolvere per mapparsi sull'architettura della memoria (cc-NUMA vs DSM, Distributed Shared Memory system). Un'altra opzione è quella di creare delle gerarchie di snoopy, creando cluster di processori e propagare la richiesta di snoop su più livelli; rimane comunque il problema del collo di bottiglia sul nodo radice

##### 4.4.2 Directory

Un'idea alternativa per rendere scalabile il protocollo potrebbe essere un meccanismo punto a punto tra cache che coinvolge solo i nodi interessati. Ad ogni linea di cache è associata una **directory** (blocco di memoria) che mantiene l'informazione sullo stato dei blocchi delle altre cache localmente

a ciascun processore. Quando avviene una miss viene cercata la entry nella directory e si comunica solo coi nodi che detengono la copia.



La directory contiene una linea per ogni linea della memoria e dentro di essa è presente un **dirty bit** che specifica se la linea è modificata di una cache di un processore e dei **presence bit** che indicano se il processore  $P_i$  possiede la linea nella propria cache. Per ogni blocco di memoria ho un **home node** e chiamo **requesting node** il nodo contenente il processore che sta chiedendo accessso a quel blocco. I vantaggi dell'uso di directory sono i seguenti:

- per le read la directory dice al requesting node esattamente dove trovare il blocco di memoria con il dato richiesto (sia home node sia un altro node se il blocco è dirty)
- per le write la directory dice al requesting node quali nodi invalidare. I vantaggi dipendono dal numero di directory che condividono il dato, nel caso peggiore è come fare un broadcast

In generale le directory sono utili a limitare il traffico. Una possibile ottimizzazione alle directory può essere quella di diminuire lo storage utilizzato in quanto gli overhead sono proporzionali alla dimensione della directory ( $P \cdot M$ ). Si può agire:

1. diminuendo  $M$  (numero di blocchi in memoria)
  - aumentando la dimensione della cache line. Non funziona bene se il pattern di accesso è sparso.
  - utilizzando **sparse directory**: si riempie dinamicamente la directory con i puntatori alla memoria, non mi serve avere tutti i riferimenti alla memoria
2. riducendo  $P$  (numero di nodi)

- ponendo più processori sullo stesso nodo. È poi da gestire la coerenza all'interno del nodo.
- utilizzando il **limited pointer scheme**: si basa sull'idea che non mi serve sapere cosa fanno tutti gli altri processori ma mi basta sapere solo quelli che hanno i dati, usa un approccio dinamico invece che una dimensione statica.

## 5 Novel Issues in Shared-Memory Multicore Systems: Synchronization

Il problema da risolvere in questo caso è la protezione dei dati condivisi. Quando più processori accedono agli stessi dati, questi accessi devono essere sincronizzati e gestiti per prevenire effetti indesiderati. Un programma può essere visto come una collezione di thread di controllo, abbiamo un program counter che realizza un control flow. Ogni thread ha un insieme di variabili che sono private (e.g. lo stack) allocate in un'area dedicata - solo logicamente privata - e variabili condivise. In un modello shared memory la comunicazione è implicita: tutti i thread chiamano le variabili con lo stesso nome, mentre nei sistemi a memoria distribuita uso message passing in maniera esplicita. La sincronizzazione è esplicita e serve ad evitare **race condition**; per fare questo utilizza delle **operazioni atomiche**. Un'operazione atomica esegue per intero in modo indivisibile: se esegue, esegue dall'inizio alla fine. I tipi di sincronizzazione che posso avere sono: **mutual exclusion** (lock) o **sincronizzazione ad eventi** (punto a punto, barriera).

### 5.1 Lock

Un lock è un'astrazione software/hardware che impedisce a qualcuno di fare qualcosa. Si esegue un lock prima di entrare nella *critical section* e un unlock all'uscita, si sta in *wait* nel caso la risorsa che mi serve sia occupata. È importante utilizzare il lock con una granularità appropriata: dato che le CS vengono eseguite sequenzialmente potrei perdere parallelismo. Per implementare un lock sono necessarie delle primitive hardware per garantire l'atomicità:

```
test&set (&address) { /* most architectures */
    result = M[address];
    M[address] = 1;
    return result;
}
```

```

swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
}

compare&swap(&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    } else {
        return failure;
    }
}

```

## 5.2 Criteri di performance

- latenza (time per op): quanto tempo per eseguire le opeazioni di lock?
- bandwidth (ops per sec): il locking coinvolge il busy waiting e crea transazioni che vanno ad impattare sulla banda
- traffico: quanti eventi si verificano sulle risorse condivise (core, BUS, memoria)?
- fairness: ci sono problemi di starvation? chi garantisce che la gestione del lock sia equa?
- storage: quanda memoria richiede?

### 5.2.1 Lock basato sulla test&set

```

int locked = 0;

acquire() {
    while (test&set(locked))
        nop;
}

release() {
    locked = 0;
}

```

```

ts R0, mem[addr] // atomically load mem[addr] into R0
// and set mem[addr] to 1

```

Con questa implementazione, M[addr] viene impostato in un busy loop, generando così continuo traffico di coerenza. Invece che fare la t&s potrei fare prima un test (poco costoso, non genera) e poi la t&s.

```

void Lock(volatile int* lock) {
    while(1) {
        if (test&set(*lock) == 0) return;
    }
}

void Unlock(volatile int* lock) {
    *lock = 0;
}

```

Non garantisce che quando esco dal while ed entro della t&s il lock sia libero. Ha una latenza più alta (eseguo qualche istruzione in più), ma genera molto meno traffico sul BUS. È un'implementazione più scalabile, il costo dello storage è lo stesso della t&s e ancora non si pone problemi di fairness. Si può implementare una test&test&set con backoff nei casi di contesta sistematica: si aggiunge un'attesa prima di prendere il lock che incrementa ad ogni fallimento dell'acquisizione. Per evitare che i processori tentino di accedere al lock nello stesso momento si può implementare una **ticket lock**: i processori aspettano il loro turno su una variabile. Nell'**array-based lock** invece ogni processore fa busy waiting su una locazione di memoria differente. Richiede più memoria degli approcci precedenti.

## 6 Novel Issues in Shared-Memory Multicore Systems: Consistency

Questa problematica è relativa alla propagazione dei valori e la domanda che mi pongo è: con che tempistica processori diversi vedono la stessa write? Non si tratta solo del fatto che i valori vengano correttamente propagati (coerenza), ma anche della tempistica con cui questo avviene (consistenza). Utilizziamo quindi dei **memory consistency models** che definiscono dei worst case/boundaries al tempo con cui riteniamo lecito che questo ritardo di propagazione avvenga. La condizione di correttezza varia a seconda del memory model. In un processore sequenziale il risultato dell'esecuzione è lo

stesso che si avrebbe se le operazioni fossero eseguite nell'ordine specificato dal programma: una read deve ritornare l'ultimo valore scritto alla stessa locazione, dove ultimo assume il significato del program order. Il program order si preoccupa soltanto di quello che accade in un singolo processore, io devo considerare anche il **visibility order**, cioè le operazioni ch avvengono complessivamente su tutti i processori.

### 6.1 Sequential consistency model

Definizione 1979: un multiprocessore è sequenzialmente consistente se il risultato di un'esecuzione è lo stesso del caso in cui le operazioni di tutti i processori fossero eseguite in qualche ordine sequenziale. Implementare questo modello implica ritardare tutti gli accessi alla memoria finché tutte le invalidazioni causate dall'accesso precedente non sono state completate. Questo è inefficiente e spesso è un overkill perchè i programmi sono per loro natura sincronizzati e quindi l'accesso nell'ordine corretto è garantito dai costrutti.

### 6.2 Relaxed consistency model

In questo modello vengono messe a disposizione read e write out-of-order se eseguiti su address diversi. Contente il rilassamento write-after-read, write-after-write, read-after-write e read-after-read.

### 6.3 Weak consistency model

Viene lasciata al programmatore la responsabilità di garantire la memory consistency attraverso delle istruzioni di **fence**, costruiti simili alle barriere. Non sincronizzo tutti il thread ma impongo un'attesa sulla transazione di memoria. Tutte le operazioni sui dati che avvengono prima dell'istruzione di fence devono essere completate prima di andare oltre.

## 7 Hardware Architectures for HPC on-chip: CPU

Recall: nei sistemi multicore inevitabilmente avrò del dark silicon, porzione che non posso far "switchare" a massima frequenza perchè eccederei il power budget. La soluzione che abbiamo esaminato è quella del cosiddetto *dim horseman*: attraverso l'*underclocking* dei processori (**spatial dimming**) o l'utilizzo di questi in *burst* (**temporal dimming**) risolve la questione dei silicio scuro.

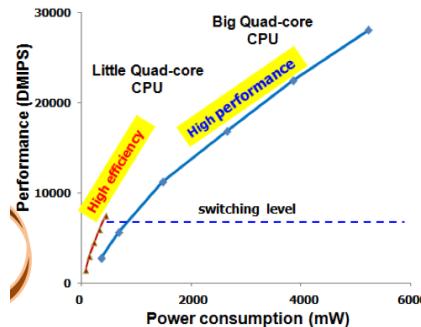
## 7.1 L'evoluzione dei multicore

Dall'avvento dei sistemi multicore il numero dei core crebbe fino 16/32 core, in seguito l'aumento dei core rese le architetture incapaci di fornire sufficiente banda di memoria e velocità di comunicazione tra i core. Distinguiamo quindi tra i processori con numero di core  $< 32$ , i multicore, e quelli con numero di core  $> 32$ , che chiamiamo **manycore**. Questi ultimi utilizzano una gerarchia di memoria gestita esplicitamente; diventa impossibile, infatti, gestire la gerarchia di memoria di un sistema che ha centinaia di core con una memoria basata su cache (e cache corerenti). Chi scrive il programma deve preoccuparsi di allocare manualmente memoria nei vari livelli della gerarchia. Possiamo classificare i processori multicore anche in base al layout delle loro CPU:

- multicore con core CPU **omogenee** (multicore tradizionali e many-core)
- multicore con core CPU **eterogenee** (processori big.LITTLE)

## 7.2 Il modello big.LITTLE

Questo modello con CPU eterogenee si utilizza quando si deve gestire un carico di lavoro che è variabile nel tempo. I core offrono una serie di **punti operativi** (coppia tensione di alimentazione, frequenza max) per risolvere i problemi di consumo energetico. In base alla richiesta di performance si utilizza il cluster big o quello LITTLE.



Il cluster LITTLE è configurabile da estremamente basso consumo e può salire rapidamente (high efficiency). Il cluster big invece serve per task più intensi che richiedono performance elevate con un conseguente aumento del consumo di potenza. Il paradigma big.LITTLE può essere implementato in maniera esclusiva (solo uno dei due cluster funziona in un determinato

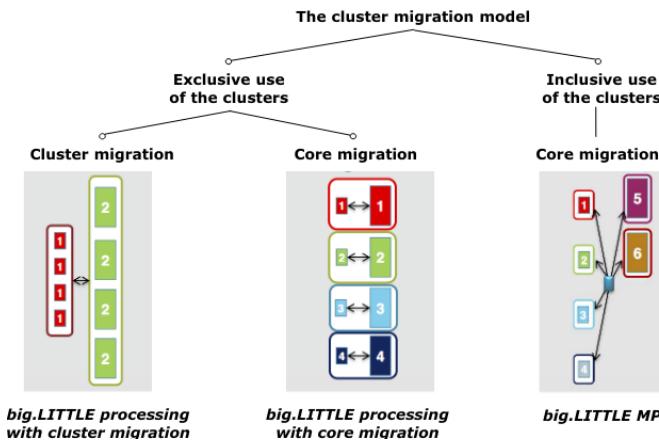
momento) o inclusiva (potenzialmente i due cluster potrebbero essere usati nello stesso momento).

### 7.2.1 Cluster migration model

Per quanto riguarda l'uso esclusivo dei cluster posso avere:

- **cluster migration**: se il workload diventa più elevato delle capacità del core LITTLE, ci sarà una migrazione verso il core big (e viceversa)
- **core migration**: ci sono più core cluster raggruppati in coppie big.LITTLE. Ogni core LITTLE può switchare verso la sua controparte big nel caso di carico di lavoro più alto e viceversa. Ogni core switch è indipendente dagli altri

Per il modello inclusivo ho solo core migration.

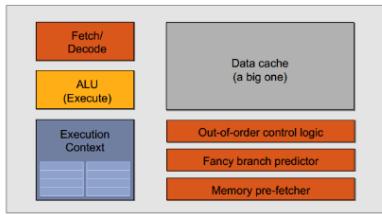


Gli switch vengono gestiti dal sistema operativo seguendo l'algoritmo **DVFS** (Dynamic Voltage and Frequency Scaling).

## 8 Hardware Architectures for HPC on-chip: GPU

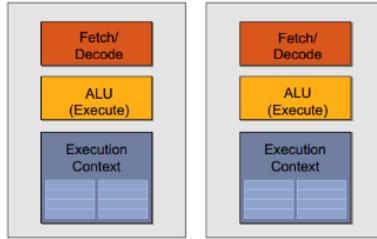
Recall: la soluzione cosiddetta dello specialized horseman proponeva di usare tutta l'area di dark silicon del chip riempiendola di core eterogenei specializzati nel compiere singoli task in maniera più efficiente, da usare su necessità. Più l'hardware si specializza più diventa efficiente e meno diventa flessibile. Inoltre anche il costo di design di alza in base alla specializzazione (ASIC prototipazione + realizzazione molto costose). I workload grafici che le GPU elaborano si dicono **embarrassingly parallel**, cioè sono operazioni

che godono di alto parallelismo: non ci sono (o quasi) dipendenze tra dati, c'è bisogno di poca sincronizzazione e in generale risultano essere bassi gli overhead dovuti alla parallelizzazione. Le GPU moderne vanno oltre la grafica, ci sono vari workload di applicazioni real-life che hanno abbondante parallelismo (machine vision, calcolo scientifico, moltiplicazione matriciale, etc.), per questo oggi si definiscono GPGPU (General Purpose GPU). La GPU è un chip eterogeneo multi-processore. Il suo design deriva da quello della CPU: si è partiti da un core tradizionale CPU-style.

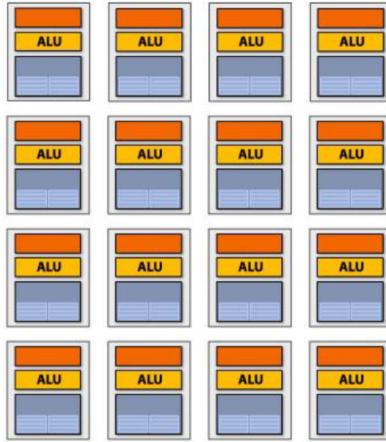


## 8.1 Slimmed design

Idea #1: rimuovere tutti quei componenti che servono ad ottimizzare l'esecuzione di un singolo thread (*out-of-order logic, memory pre-fetcher, branch predictor*) e al loro posto replico un core più semplice (solo fetch/decode, ALU, execution context)

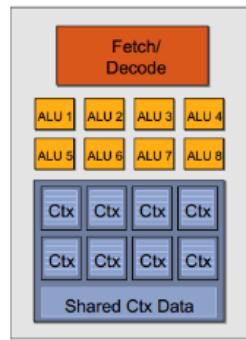


In questo modo posso eseguire, in base a quanti core replica, frammenti di codice contemporaneamente, nel caso il tipo di programma sia embarrassingly parallel. A questo punto si replicano i core (più semplici), così da poter eseguire il codice in parallelo. Andando avanti nella replicazione, arriviamo ad un numero molto alto, con 16 core ho 16 *instruction streams* simultanei.

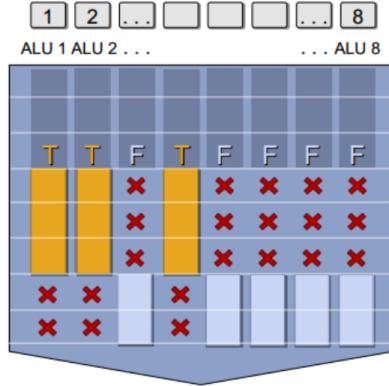


## 8.2 SIMD processing

Idea #2: i vari frammenti che metto in esecuzione sono istanze dello stesso frammento di codice declinate su dati diversi (concetto di SIMD). Posso dunque ammortizzare la complessità della logica che uso per gestire lo stream di istruzioni condividendo la logica di fetch/decode tra più ALU; esse faranno i calcoli su dati diversi, ma avranno un *program counter* comune.



Un problema che riguarda l'utilizzo di una fetch/decode condivisa è quello dei **branch**: non ho capacità di differenziare il control flow tra un processore e l'altro. Se ho una condizione da verificare normalmente in base al caso in cui mi trovo i program counter puntano per alcuni nel caso *if* per altri nel caso *else*.



Avendo un solo program counter si è costretti a sequenzializzare l'esecuzione dei due flussi e così facendo la ALU che si trovano nel branch false stallano e poi stallano le altre. Inoltre si possono verificare stalli quando ci sono delle dipendenze tra dati e se devo fare un accesso in memoria lo pago caro (a livello di latenza), non avendo più nè cache nè logica di pre-fetching.

### 8.3 Execution context interleaving

Idea #3: cerco di garantire che l'hardware abbia sempre a disposizione codice da eseguire, cosicchè lo scheduler possa, in caso di stalli, assegnare una nuova porzione di lavoro. Creo una coda di lavori da cui attingere quando ho degli stalli e per fare questo mi serve avere dei meccanismo di context switch molto efficienti. Più blocchi di lavoro ho, meglio nascondo gli stalli e la latenza. Per implementare **memory latency hiding** c'è il costo di implementare per ognuno di questi thread (semplificati) un contesto (registri, memoria): al crescere del numero dei thread aumenta la quantità di memoria che mi serve per salvare questi contesti. Ho un tradeoff tra latency hiding e memoria per il **pool di execution context**.

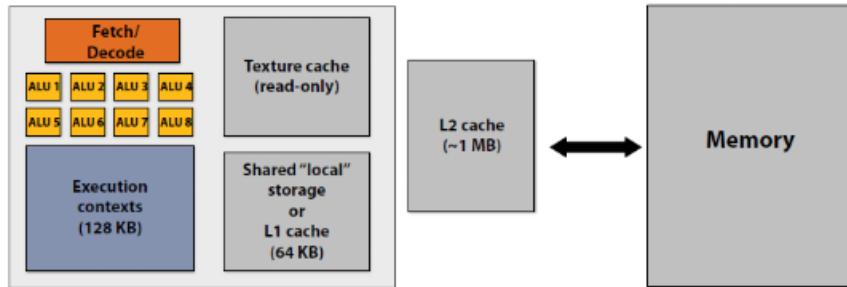
### 8.4 Gerarchia di memoria GPU

L'architettura delle GPU è progettata per garantire un throughput molto più elevato di una CPU: la GPU è un *data-cruncher*, devo continuamente approvvigionare i core con dati e per fare questo mi serve una banda larga (150 GB/s vs 25 GB/s). Il bottleneck delle GPU rimane quindi l'accesso in memoria; per ridurre l'uso della banda posso:

- formulare i problemi in modo da averre intensità aritmetica alta (memory-bound kernel vs CPU-bound)

- prediligere lo sharing di risorse senza passare dalla main memory (on-chip storage)

Nelle GPU moderne in realtà ho una gerarchia di memoria in un certo senso ibrida: ho un parte di scratchpad (da gestire manualmente) e una parte di cachehe (L2 condivisa da più cluster ed L1 all'interno della singola unità).

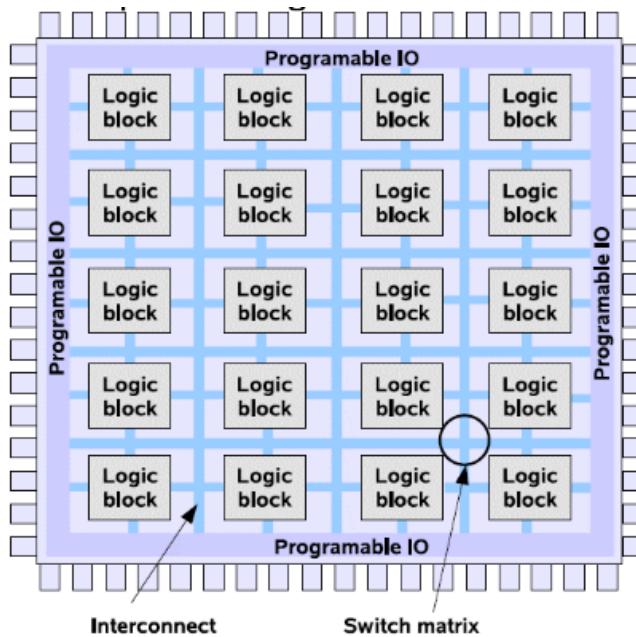


## 9 Hardware Architectures for HPC on-chip: FPGA

La progettazione di sistemi specializzati/eterogenei - che seguono la filosofia dello specialized horseman - va sempre incontro al tradeoff tra efficienza e flessibilità. Un sistema più specializzato è più efficiente (energeticamente), ma è meno flessibile a problemi general purpose. L'FPGA (Field-Programmable Gate Arrays) è un'architettura che combina l'alta flessibilità del software con le performance dell'hardware usando connessioni ad alta velocità programmabili. A differenza della CPU e GPU, l'FPGA è svincolata dall'architettura di Von Neumann e consente di apportare cambiamenti sostanziali al datapath e al control flow. La differenza dagli ASIC (Application Specific Integrated Circuit), invece, è quella di essere **riprogrammabile**: possiamo adattare l'hardware dinamicamente caricando un nuovo circuito. L'hardware riconfigurabile nasce per riempire il gap tra hardware puro e software, raggiungendo potenzialmente una performance molto maggiore del software pur mantenendo un più alto livello di flessibilità rispetto all'hardware. Una scheda FPGA è una matrice che interconnette - tramite fili - dei **Configurable Logic Block**. Ogni blocco mi permette di realizzare una rete logica combinatoria o sequenziale.

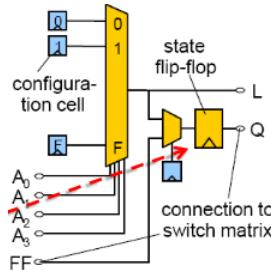
- **Rete logica combinatoria:** applicato un valore ai suoi ingressi, instantaneamente mi viene prodotto un valore sulle uscite che dipende soltanto dagli ingressi che sto applicando (è detta anche rete senza memoria).

- **Rete logica sequenziale:** ha memoria, produce in uscita un valore che dipende sia dai valori degli ingressi, ma anche da altri applicati in precedenza. Sulle FPGA sono presenti degli I/O pads che mi permettono di mandare segnali verso l'interno e l'esterno della scheda. Sono detti anche IOB (I/O blocks) e sono organizzati in banchi che cambiano di dimensione in base al costruttore.

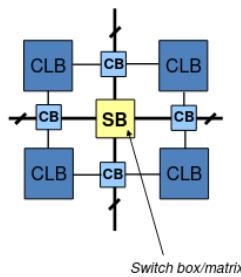


## 9.1 Componenti delle FPGA

Una CLB è formata da **slices**, a loro volta composte da **look-up tables**, e poi da **flip-flop** (il più elementare componente della memoria). Una LUT è fondamentalmente un multiplexer che valuta una **tabella di verità** memorizzata nella SRAM. Per gestire la logica sequenziale si aggiunge un flip-flop all'output della LUT.



Le CLB sono connesse tramite CB (Connection Box) ai fili per il routing, che consentono lo spostamento del segnale solamente lungo uno specifico cavo. Per interconnettere più fili si usano delle SB (Switch Box), che connettono i canali del routing verticali e orizzontali. Una SB contiene 6 transistor programmabili che ne definiscono la **routability** nelle varie direzioni.



Per quanto riguarda la memoria posso definire nel mio progetto FPGA della SRAM creata interconnettendo opportunamente tanti flip-flop (**distributed RAM**). L'altra opzione è quella di usare i **block RAM**, cioè piccoli blocchi di RAM dedicata. Distinguiamo infine tra architettura a **componenti discreti**, l'FPGA si connette al processore tramite link esterni, e a **componenti integrati**, all'interno dello stesso SOC integro CPU ed FPGA e le varie interfacce. Nell'ultimo caso la comunicazione sarà più veloce in quanto avviene all'interno dello stesso blocco di silicio.

## 10 Understanding Performance

Ci sono tre fattori che influenzano un programma parallelo:

- **coverage**: quantità e distribuzione di parallelismo in un algoritmo
- **granularity**: rapporto tra lavoro utile e overhead dovuti al parallelismo
- **locality** della computazione e comunicazione

## 10.1 Coverage

Non tutti i programmi sono embarrassingly parallel: hanno parti sequenziali e parti parallele. Voglio riuscire a misurare il miglioramento (speedup) dopo aver applicato della parallelizzazione.

**Legge di Amdahl:** *il miglioramento di performance che posso ottenere nello sfruttare una modalità di esecuzione più veloce (e.g. parallelizzazione) è limitato dalla frazione di tempo in cui posso usare questo modo di esecuzione più veloce.*

Sia  $p$  la parte di lavoro parallelizzabile,  $n$  il numero dei processori,  $s$  lo speedup, allora:  $s = \frac{1}{(1-p)+\frac{p}{n}}$ . Lo speedup tende a  $\frac{1}{(1-p)}$  quando il numero di processori ( $n$ ) tende ad infinito, cioè la parte non parallela limita la performance. Ha senso parallelizzare quando il programma ha una grande parte di lavoro parallelo. Un altro ostacolo sono gli overhead della parallelizzazione: costi di lanciare i thread, comunicazione tra dati condivisi, sincronizzazione, computazione extra. Bisogna dunque cercare un tradeoff tra granularità dei task e overhead: un algoritmo ha bisogno di unità di lavoro sufficientemente grandi per mantenere gli overhead bassi, ma non troppo grandi da non avere abbastanza lavoro parallelo.

### 10.1.1 Scaling

Lo scaling è la capacità di un programma parallelo di mantenere costante la crescita di speedup all'aumentare del numero di processori. Parliamo di **strong scaling** quando lo speedup è raggiunto senza far crescere la dimensione del problema e di **weak scaling** quando invece sono costretta ad aumentare la dimensione del problema per avere abbastanza lavoro da distribuire in parallelo.

## 10.2 Granularity

È una misura qualitativa del rapporto che esiste tra lavoro utile e lavoro aggiuntivo dovuto al parallelismo. Distinguono tra parallelismo:

- **fine-grain**, bassa granularità, overhead alti, carico di lavoro bilanciato
- **coarse-grain**, alta granularità, overhead ammortizzati, carico di lavoro più difficilmente bilanciabile

Il bilanciamento del carico di lavoro (**load balancing**) dipende da come distribuisco il lavoro tra i processori. Esso dipende soprattutto dalla granularità

con la quale parallelizzo: se il lavoro è sbilanciato si generano situazioni di stallo e si abbassa l'utilizzazione.

### 10.2.1 Load balancing statico

È il programmatore ad assegnare a priori il quantitativo di lavoro ai vari core. Questo tipo di bilanciamento ha un costo basso e funziona bene se ho lavoro e core omogenei.

### 10.2.2 Load balancing dinamico

Ho una coda di unità di lavoro che distribuisco tra i vari core che viene eseguito man mano che i processori finiscono quello che stanno facendo. Questo approccio ha costi più alti in quanto devo eseguire uno scheduler che gestisce la coda ma si adatta meglio nei casi in cui ho lavoro non variabile o core eterogenei. In uno schema ibrido è possibile utilizzare tecniche di **stealing**.

### 10.2.3 Metriche di performance

Nel caso di parallel programming che prevede comunicazione remota sono fatti chiave:

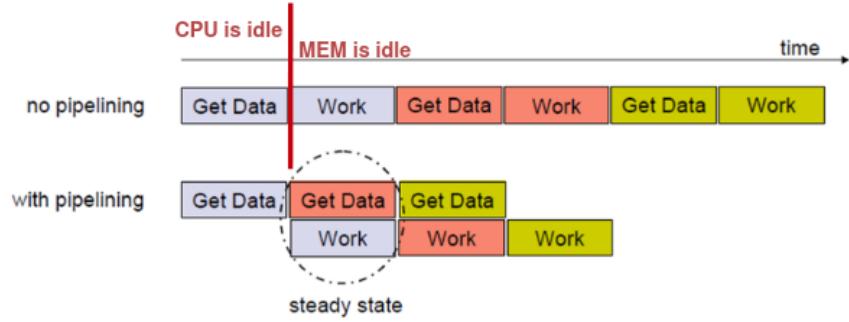
- la banda della rete (*network bandwidth*): misura di throughput (quantità di dati per unità di tempo)
- la latenza di comunicazione (*communication latency*): quanto tempo serve perchè un messaggio attraversi questo network - shared BUS *vs* network on-chip

Per nascondere la latenza sfruttiamo dei meccanismi di overlap, data prefetch, switch in caso di stalli e pipelining. Complessivamente il costo della computazione è espresso dal seguente modello:

$$C = f \cdot (o + l + \frac{\frac{n}{m}}{B} + t - overlap)$$

Dove  $f$  è la frequenza dei messaggi,  $o$  è l'overhead per messaggio,  $l$  è la latenza per messaggio,  $n$  i dati inviati,  $m$  numero di messaggi,  $t$  il costo della contesa per messaggio. Per sfruttare l'overlap ho bisogno di unità preposte ed esclusivamente dedicate per il trasferimento dei dati (come la DMA) - non posso avere la CPU che fa computazione e trasferimento dati. Anche la DMA però ha un costo e se il messaggio è breve potrebbe essere più conveniente usare la CPU. In caso di messaggi di dato usiamo la DMA che

ha una maggiore capacità di **burst transfer**. Per sovrapporre l'esecuzione di comunicazione con l'esecuzione di lavoro sulla CPU si fa usando degli schemi di pipelining come il **double buffering**.



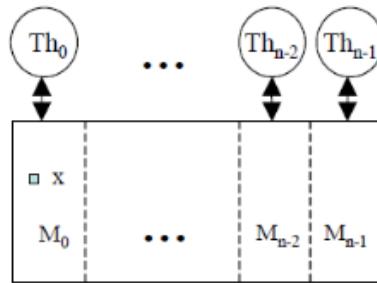
Ho due buffer: copio i dati sul primo e ci inizio a fare i calcoli, in contemporanea programmo il DMA perchè mi prefetchi un secondo buffer. A questo punto switcho tra un buffer e l'altro: la fase di prefetch dei dati per la prossima unità di lavoro si sovrappone a quella di lavoro sul primo buffer.

## 10.3 Locality

Differenziamo tra macchine con UMA (memoria centrale equidistante da tutti i processori) e macchine con NUMA (memoria distribuita con address space comune a distanza variabile dai processori).

### 10.3.1 Distributed Shared Memory

A.k.a. Partitioned Global Address Space (PGAS): ogni processore ha un nodo di memoria locale che è visibile globalmente a tutti gli altri processori presenti nel sistema. Gli accessi locali sono veloci, quelli remoti sono lenti (NUMA).



Le possibili ottimizzazioni per la gerarchia di memoria sono:

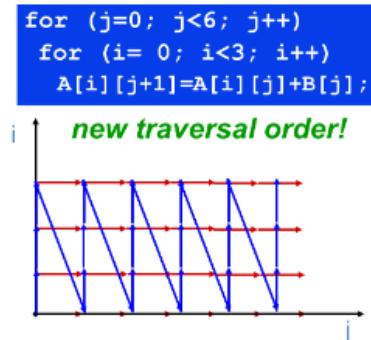
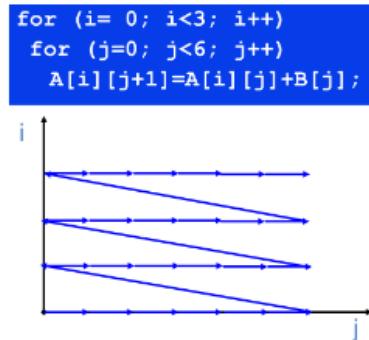
- riduzione della latenza: tempo che intercorre tra quando effettuo una richiesta e quando ricevo una risposta
- massimizzazione della banda: quantità di dati che riesco a recuperare per unità di tempo
- gestire gli overhead di comunicazione (e.g. il costo delle copie)

Per raggiungere questi obiettivi sfrutto i principi di riuso (temporale) e località (spaziale): spesso i programmi riusano lo stesso dato o dati che sono adiacenti nelle locazioni di memoria.

Esempio:

```
for (i=1; i<N; i++){
    for (j=1; j<N; j++){
        A[j] = A[j+1]+A[j-1]
    }
}
```

In questo caso  $A[j]$  ha riuso spaziale nell'loop  $j$ . Una tecnica per migliorare la località ed il riuso è il **reordering**. Le accortezze che devo sempre avere quando sfrutto il riordino sono la **safety** e la **profitability**. Un esempio di questa trasformazione è il **loop permutation**, che ottimizza la località spaziale. Si riarrangiano i loop annidati per spostare il parallelismo ad un livello di granularità appropriato: più spostiamo il punto dove parallelizziamo verso loop interni, più il parallelismo diventa fine-grained (bene se vogliamo applicare uno schema SIMD, altrimenti costo molto alto); altrimenti se voglio avere parallelismo a livello di thread (approccio SPMD) mi sposto verso livelli più esterni del loop.



Un'altra trasformazione di riordino è il **tiling** (o blocking): spezzo il loop originale in blocchi più piccoli con l'obiettivo di mantenere validi ed usare dati che sono già in cache (voglio evitare *capacity miss*). Questa tecnica è fondamentale per gestire la problematica di storage limitato (vale per la cache, registri, scratchpad memory). Si può applicare gerarchicamente per far sì che ai vari livelli abbia sempre un access pattern che minimizza il numero di miss.

Esempio:

```
for (j=1; j<M; j++)
    for (i=1; i<N; i++)
        D[i] = D[i] +B[j,i]
```

*Strip mine* (decomposizione del loop in sottoblocchi):

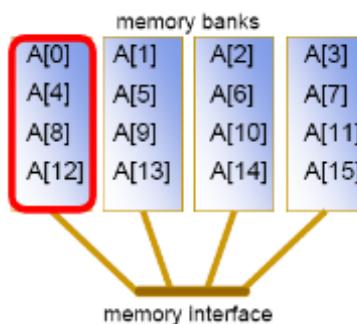
```
for (j=1; j<M; j++)
    for (i=1; i<N; i+=s)
        for (ii=i; ii<min(i+s-1,N); ii++)
            D[ii] = D[ii] +B[j,ii]
```

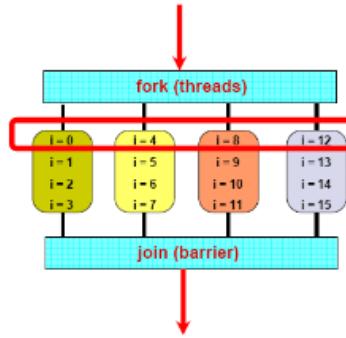
e poi *permute*:

```
for (i=1; i<N; i+=s)
    for (j=1; j<M; j++)
        for (ii=i; ii<min(i+s-1,N); ii++)
            D[ii] = D[ii] +B[j,ii]
```

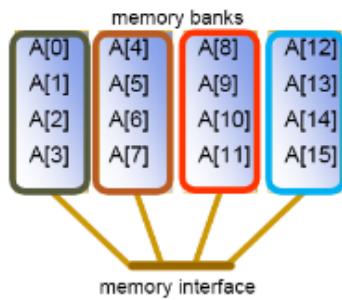
Per quanto riguarda la *safety* devo garantire che il riordino non rompa delle dipendenze tra dati. Mentre a livello di *profitability* devo analizzare e trovare lo sweet spot tra riuso e cache miss. Un'ultima ottimizzazione per quanto riguarda la località è il **memory banking**, ovvero l'organizzazione della memoria in banchi. Questa tecnica permette di avere più accessi gestiti in parallelo.

Esempio: quattro processori si dividono il loop ma cercano di accedere tutti allo stesso blocco in contemporanea.

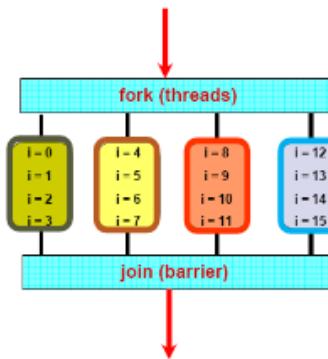




Il programma deve generare accessi alla memoria spaziati da un fattore che è friendly per il banking factor.



In questo modo ogni thread accede al suo blocco di memoria.

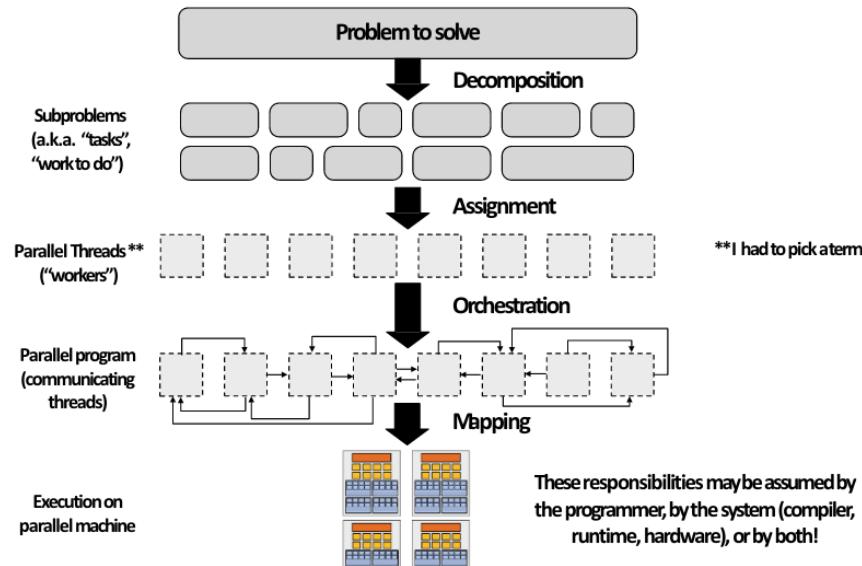


## 11 Parallel Programming in Practice

Per creare un processo parallelo generalmente si seguono i seguenti step:

1. identificare il lavoro parallelizzabile
2. partizionare il lavoro ed i dati
3. gestire il partizionamento tra i thread

Un obiettivo generale è quello di massimizzare lo speedup  $S_p = \frac{t_1}{t_p}$ . La creazione di un programma parallelo può essere vista come un vero e proprio workflow/algoritmo.



### 11.1 Decomposizione

Obiettivo: creare almeno tanti task (che possono essere eseguiti in parallelo) quanti i nostri processori. Per sfruttare la concorrenza la sfida principale è quella di identificare le dipendenze: esse infatti determinano la parte sequenziale del nostro programma che - secondo la legge di Amdahl - limita lo speed-up.  $S \leq \frac{1}{s}$  La decomposizione, ad oggi, è ancora nelle mani del programmatore: non ci sono tool automatici in grado di farlo. Decomposizioni comuni sono: function call, loop. Per identificare il parallelismo si ricorre a **task decomposition** e **data decomposition**. Il primo riguarda suddivisione della computazione indipendente inherente all'algoritmo. La seconda invece punta il focus sul trovare dove la stessa computazione è applicata a parti diverse di una stessa struttura dati. In generale è più facile partire con un alto numero di task - ed eventualmente metterli insieme in un secondo momento - che il contrario. Le guidelines per la decomposizione sono:

- *flessibilità* in numero di task e dimensione e numero di data chunk
- *efficienza* a livello di lavoro, i task devono ammortizzare il costo della loro creazione e i data chunk dovrebbero generale quantità di lavoro equilibrata
- *semplicità* il codice deve rimanere leggibile, facile da capire e debuggare

In generale si parte dalla task decomposition per poi arrivare a quella sui dati. Nel caso in cui il programma lavori con grandi strutture dati si preferisce partire dalla data decomposition.

### 11.1.1 Analisi delle dipendenze

Come posso capire se due task possono essere eseguiti in parallelo? Mi viene in aiuto la **condizione di Bernstein**: due task  $T_1$  e  $T_2$  sono paralleli se

- $R_1 \cap W_2 = \emptyset$
- $R_2 \cap W_1 = \emptyset$
- $W_1 \cap W_2 = \emptyset$

Dove con  $R_i$  indico il read set del task  $T_i$  e con  $W_j$  il write set del task  $T_k$ .

## 11.2 Assegnamento

Pensiamo ai task come cose da fare e i thread come workers. L'obiettivo è assegnare i task (che ho identificato nella fase di decomposizione) ai thread (i workers) in modo da avere un buon bilanciamento del carico. Idealmente tutti i processori lavorano per tutto il tempo e finiscono nello stesso istante.

### 11.2.1 Assegnamento statico

È molto semplice e ha pochissimo overhead. Si usa quando la quantità di lavoro e il numero di task è predicibile. Il tempo di durata di ogni task deve essere noto a priori.

### 11.2.2 Assegnamento semi-statico

Si basa sull'idea che quello che ho appena osservato (nel passato recente) è un buon indicatore di quello che succederà nel futuro. Le applicazioni profilano il tempo di esecuzione e riaggiustano l'assegnamento (dinamicamente). Tra i vari aggiustamenti l'assegnamento è statico.

### 11.2.3 Assegnamento dinamico

Usato nel caso non si sa a priori il numero e la durata dei task. Il programma determina l'assegnamento a runtime per assicurare una distribuzione equa del carico. Si usa un sistema **pull**: è il thread che fa richiesta del nuovo task da eseguire. Bisogna fare attenzione alla dimensione del task, anche in questo caso siamo soggetti al tradeoff tra granularità e overhead di parallelismo: se i task sono troppo piccoli avrà un buon bilanciamento del carico, ma alti costi di sincronizzazione. Potrebbe comunque capitare che ci sia un task di grandi dimensioni lasciato per ultimo e questo porterebbe ad uno sbilanciamento del carico. Per prevenire che ciò accada si può ricorrere allo **smarter task scheduling**: i task di dimensioni maggiori vengono assegnati all'inizio della computazione - per sfruttare questa tecnica è necessario avere parziale conoscenza del workload a priori. Un altro modo per diminuire l'overhead di sincronizzazione è quello di usare un insieme di code: eviteremo che i thread si debbano sincronizzare su una singola coda. Se un thread finisce la sua coda può "rubare" lavoro da un'altra coda (tecnica di **stealing**).

## 11.3 Orchestration

A questo punto il nostro obiettivo è definire come interagiscono i task tra di loro in termini di:

- usare le giuste primitive di sincronizzazione per prevenire race condition
- usare strutture dati appropriate per conservare la località
- strutturare la comunicazione

L'orchestration è strettamente collegata all'hardware sottostante.

## 11.4 Mapping

Il mapping specifica la mappatura delle astrazioni usate per esecuzione, comunicazione e sincronizzazione sull'hardware effettivo. Esempi:

- mappare sui core usando il sistema operativo (e.g. pthread assegnati ad un execution context su un core CPU)
- mappare usando il compilatore (e.g. mappare costrutti di alto livello di OpenMP ai pthread sottostanti)
- mappare con l'hardware (e.g. blocchi di CUDA thread sui core CPU)

Le decisioni da prendere per effettuare il mapping non sono banali: meglio mappare thread correlati (*maximise locality and sharing & minimize communication and sync*) o thread non correlati (memory bound e CPU bound per massimizzare uso risorse)?

#### 11.4.1 Performance analysis

Normalmente si parte da un profiling dell'applicazione, si prova la soluzione più semplice, si misura la performance e, se necessario, si aggiorna la soluzione. Inanzitutto bisogna identificare gli **hot spot**:

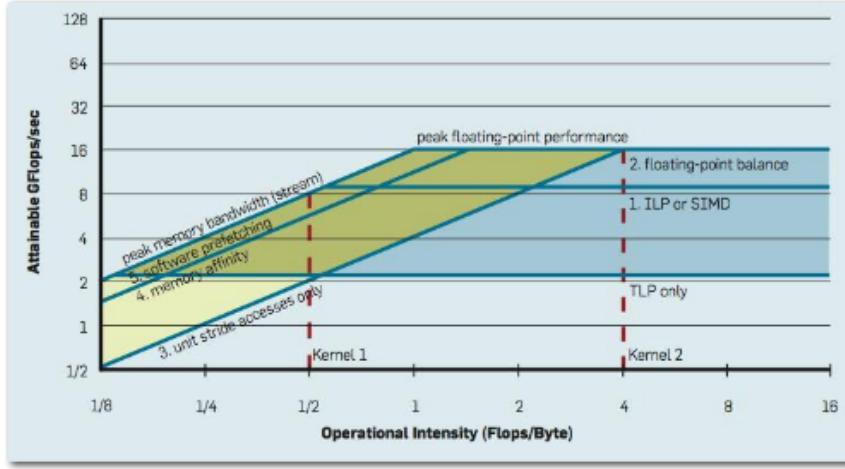
- dove si spende le maggior parte del programma?
- qual è il fattore limitante dello speedup? Computazione, memoria o sincronizzazione?
- quali margini di miglioramento ho? Quali **high watermark** posso stabilire?

Per misurare la performance inizio misurando il tempo d'esecuzione. Un primo possibile benchmark potrebbe essere fatto utilizzando la primitiva UNIX `time`: `$ gcc sample.c $ time a.out -03`

In questo modo però non riesco a correlare l'incremento di performance alle parti del codice. Una soluzione più fine sarebbe mettere dei **timestamp** in punti specifici del programma: i processori moderni mettono a disposizione dei **performance counter**, registri hardware che contengono informazioni relative all'esecuzione. Con questa modalità possiamo misurare esattamente quello che vogliamo, ma richiede l'intervento manuale e si potrebbe incorrere nell'effetto Heisenberg. Altri tipi di profiling che non richiedono modifiche al programma sono:

- **event-based profiling**, mi registro su un evento e misuro ciò che mi serve
- **time-base profiling**, misuro con una certa frequenza di campionamento

Posso misurare: clock cycle, pipeline stalls, cache hit/miss, numero di load/store, numero di floating point operation da cui si può derivare l'utilizzazione del procesore, le istruzioni per ciclo, bandwidth, etc. Il **roofline model** offre informazioni sulle performance di picco di una specifica macchina in funzione dell'intensità aritmetica.



In diagonale sulla sinistra ho il limite di Gflops/s che posso ottenere in relazione alle tecniche di accesso alla memoria che uso. Le linee orizzontali sulla destra rappresentano il limite di Flops/byte che posso ottenere in relazione alle tecniche di parallelismo che uso. Le tecniche per stabilire gli high watermark sono:

- aggiungere istruzioni matematiche per capire se il programma è CPU-bound
- togliere istruzioni matematiche e aggiungere operazioni di memoria per capire se è memory-bound
- leggere lo stesso dato per testare la località della cache
- togliere le operazioni atomiche/lock per studiare la sincronizzazione

## 12 OpenMP

È un programming model ormai standard de facto per sistemi shared memory. Comprende una collezione di direttive per il compilatore, routine di libreria, variabili d'ambiente e clausole. OpenMP segue un approccio incrementale: partendo dal programma sequenziale applico via via parallelismo dove possibile. Ha bisogno di supporto speciale a livello di compilatore, che traduce le direttive in chiamate a funzione della libreria di supporto a runtime che a sua volta si appoggia ad una libreria di più basso livello. Nella sua versione più di base utilizza il parallelismo fork/join: il programma nasce da un singolo thread (**master**) e si dirama (**fork**) creando altri thread nella fase

parallel, infinite si conclude in una barriera (**join**). Per specificare questo tipo di esecuzione uso le **pragma** (*pragmatic information*) in C, C++ che sono un modo che il programmatore ha per comunicare con il compilatore.

```
#pragma omp <rest of pragma>
```

## 12.1 Componenti OpenMP

Le direttive sono specializzate per funzionalità:

- regioni parallele: `#pragma omp parallel`
- work sharing: `#pragma omp for`, `#pragma omp sections`
- sincronizzazione: `#pragma omp barrier`, `#pragma omp critical`, `#pragma omp atomic`

Le clausole complementano le direttive:

- data scoping: `private`, `shared`, `reduction`
- loop scheduling: `static`, `dynamic`

Le funzioni della runtime library implementano le funzionalità delle direttive:

- manipolazione di thread: `omp_get_thread_num()`, `omp_get_num_threads()`
- sincronizzazione: `omp_set_lock()`, `omp_unset_lock()`

## 12.2 Direttiva "parallel"

Costrutto fondamentale per fare l'**outline** della computazione parallela all'interno di un programma sequenziale. Il codice dentro lo scope è replicato in più istanze ognuna delle quali sarà assegnata ad un thread. Fare outline significa creare una funzione dedicata che racchiude il codice nello scope della direttiva.

```
int main ()  
{  
#pragma omp parallel  
{  
    printf ("Hello world");  
}  
}
```

```

int main ()
{
    omp_parallel_start(&parfun, ...);
    parfun();
    omp_parallel_end();
}

int parfun()
{
    printf("Hello world!");
}

```

Il costrutto pragma viene sostituito con chiamate alla libreria di runtime. Per specificare come sono viste le variabili tra thread uso le clausole **shared**, **private**.

```

int main()
{
    int id;
    int a = 5;
#pragma omp parallel shared(a) private(id)
{
    id = omp_get_thread_num();
    if (id == 0)
        printf("Master: a = %d.", a*2);
    else
        printf("Slave: a = %d.", a)
}
}

```

Nel caso **shared** la variabile viene passata per puntatore: **a** è allocata sullo stack del thread master. I thread POSIX condividono lo spazio di memoria dello stesso processo. Con la clausola **private** **id** viene dichiarata nuovamente per ogni thread. Altre clausole rilevanti sono la **firstprivate** (copy in, private storage) e **lastprivate** (copy out, private storage).

### 12.3 Direttiva "for"

Mentre la pragma **parallel** replica sui vari thread il codice, quella **for** (all'interno di una **parallel**) dice al compilatore di non replicare l'intero loop ma specificare che le iterazioni del loop assumono valori diversi per i vari thread.

```

int main ()
{
#pragma omp parallel for
{
    for (i=0; i<10; i++)
        a[i] = i;
}
}

```

Il codice per il ciclo for viene spostato nella funzione outlined e ogni thread avrà differenti coppie LB, UB.

```

int main ()
{
    omp_parallel_start(&parfun, ...);
    parfun();
    omp_parallel_end();
}

int parfun ()
{
    int LB = ...;
    int UB = ...;
    for (i=LB; i<UB; i++)
        a[i] = i;
}

```

## 12.4 Clausola "schedule"

Può essere **static**: a tempo di compilazione genero tutto il codice che mi serve per schedulare il loop (overhead basso). Viene usata quando il numero di iterazioni è noto a priori e la loro durata è equivalente. Se la durata delle iterazioni è variabile si rischia di avere un workload sbilanciato.

```

#pragma omp for schedule(static)
{
    for (i=0; i<12; i++)
        a[i] = i
}

```

Nello scheduling statico il *datachunk* corrisponde a  $C = \lceil \frac{N}{N_{thr}} \rceil$

thread ID (TID)	0	1	2	3
LB = C · TID	0	3	6	9
UB = min {C · (TID + 1), N}	3	6	9	12

In alternativa lo scheduling può essere **dynamic**: di default i data chunk sono il più piccolo possibile e sono raccolti in una coda alla quale i thread accedono in modo sincronizzato.

```
int parfun()
{
    int LB, UB;
    GOMP_loop_dynamic_next(&LB, &UB);
    for (i=LB; i<UB; i++) {...}
}
```

GOMP sta per GNU OMP. Se la granularità del lavoro è fine l'overhead è significativo. Possiamo modificare una granularità più coars in questo modo: `#pragma omp for schedule(dynamic,n)` dove `n` è la dimensione del chunk.

## 12.5 Direttiva "barrier"

`#pragma omp barrier` è implicita alla fine dei costrutti `parallel`, `for`, `sections` a meno chè non sia specificata la clausola `nowait`.

## 12.6 Direttiva "critical"

`#pragma omp critical` funziona come una lock/unlock. All'interno di un blocco `critical` l'esecuzione diventa di fatto sequenziale, dunque si cerca di mantenere questi blocchi il più piccoli possibile.

```
double area, pi, x;
int i, n;
#pragma omp parallel for private(x) shared(area)
{
    for(i=0; i<n; i++) {
        x = (i +0.5)/n;
        #pragma omp critical
            area += 4.0/(1.0 + x*x);
    }
}
pi = area/n;
```

Anzichè accedere tutti ad una variabile condivisa, OpenMP usa il costrutto **reduction** per memorizzare risultati parziali delle operazioni di una riduzione in variabili private, combinandole alla fine del loop.

```
double area, pi, x;
int i, n;
#pragma omp parallel for private(x) shared(area) reduction(+:area)
{
    for(i=0; i<n; i++) {
        x = (i +0.5)/n;
        area += 4.0/(1.0 + x*x);
    }
}
pi = area/n;
```

## 12.7 Direttiva "master" e "single"

La **master** denota un blocco che esegue solamente il *master thread*, mentre **single** il primo thread che arriva. Master non ha sincronizzazione implicita alla fine, se necessario devo esplicitare una barriera; nella single invece la barriera è implicita. Le direttive parallel sono molto costose quindi se devo fare lavori in parallelo e poi alcuni solo su un thread è utile usare la master o la single.

## 12.8 Task

OpenMP permette il paradigma MPMD (task parallelism) mediante la direttiva **sections**.

```
int main()
{
#pragma omp parallel sections
{
#pragma omp section
    v = alpha();
#pragma omp section
    w = beta ();
}
#pragma omp parallel sections
{
#pragma omp section
```

```

    v = delta();
#pragma omp section
    w = gamma (v, w);
}
z = epsilon (x, y);
printf("%f\n", z);
}

```

La sections ha dei limiti: tutti i task devono essere dichiarati staticamente nel codice e non si possono usare nei cicli while (a meno di unrolling). Nella specifica 3.0 è stato aggiunto il supporto ai task che permette di gestire: i loop *unbonded*, algoritmi ricorsivi, schemi *producer/consumer*, etc. Un **task** è un'unità di lavoro che può essere eseguita in un momento arbitrario (subito o *deferred*) rispettando le dipendenze. È composto da: codice da eseguire, data environment (piccolo stack che crea il contesto ed eventualmente piccolo heap) e delle ICV (interval control variables). La sintassi è: `#pragma omp task [clauses]`. Ogni thread che incontra questa direttiva crea un task: fa outlining del codice con una funzione a parte e poi prepara il data environment. È evidente che sia un costrutto altamente componibile e può essere innestato in regione parallele, altri task, costrutti di worksharing. Le clausole di scoping possono essere: `shared`, `private`, `firstprivate`, `default(none)`; se non è specificato di default le variabili globali sono shared altrimenti firstprivate. Per quanto riguarda la sincronizzazione sappiamo che le barriere riguardano i thread, non i task; con la `#pragma omp taskwait` sospendiamo il task che la incontra finché i suoi figli diretti non hanno terminato l'esecuzione. I task sono eseguiti dal thread del team che li ha generati. Dalla v3.0 le regioni parallele creano task: viene generato un task implicito per ogni thread.

```

void traverseList(List l)
{
    Element e;
    for(e=e->first; e; e=e->next)
#pragma omp task
    process(e);
#pragma omp taskwait
}

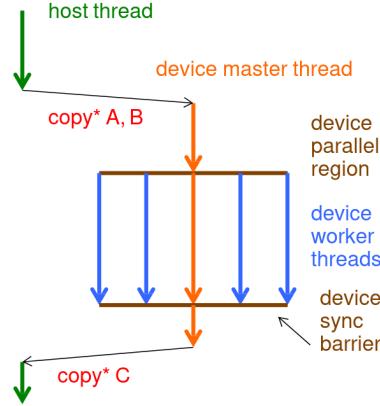
```

## 13 OpenMP Accelerator Model

OpenMP v4+ decide di abbracciare il supporto per sistemi eterogeni: viene definito il **device model**: un host (CPU) e uno o più device (GPU). Il device è composto da una o più **compute unit** (cluster) e dentro ognuna di queste c'è uno o più **processing element**. La memoria è divisa in host memory e device memory. Questo modello assomiglia al fork/join ma comprende più attori: in questo caso il forking prende il nome di **off-loading**, cioè scaricare il caico del sistema host su uno dedicato.

1. il programma parte e lancia il thread iniziale che esegue su host
2. ci sono regioni parallele in tutto il programma
3. il task iniziale parte con l'esecuzione ed incontra la direttiva **target**
4. il task genera un **target task**
5. il target task lancia una **target region** sul device
6. sul device si crea un implicito singolo thread - la **target** non crea di per sè parallelismo, ma contesto per l'esecuzione sul device.
7. il thread esegue sul device
8. quando il lavoro sul device è finito si restituisce il controllo all'host

```
double A[n,n] , B[n,n] , C[n,n];
#pragma omp target map(to: A, B) map(from:C)
{
    int n =64;
    #pragma omp parallel for
    for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++){
            for(k=0; k<n; k++){
                C[i,j] = A[i,k] * B[k,j];
            }
        }
    }
}
```



La clausola `map` viene utilizzata per specificare la necessità di copiare dei dati: mappa i *data environment*. I costrutti per l'esecuzione su un device sono la `omp target` e la `omp declare target` che si usa attorno ad una funzione che deve eseguire sul device. I costrutti per il worksharing (dopo aver trasferito il controllo da host thread a device thread) sono la `omp teams`, per blocchi strutturati e la `omp distributes`, per spezzare i for loops tra più cluster. La target è sincrona ma può essere resa asincrona con la `nowait`.

### 13.1 Terminologia

Una **league** è un insieme di **team** generati dal costrutto `teams`. `teams` fornisce un mapping 1:1 dei thread sui cluster (compute unit) della mia architettura. Un **contention group** è l'insieme dei thread di un team di una league ed i loro thread discendenti. Il costrutto `teams` crea una league di team di thread:

- il master thread di ogni team esegue la `teams` region
- il numero di thread viene specificato dalla clausola `num_teams`
- ogni team esegue con un numero di thread minore di `thread_limit`
- i thread di team diversi non possono sincronizzarsi tra loro

La `teams` deve essere perfettamente innestata in un costrutto `target`; dentro `teams` posso usare solamente: `distribute`, `distribute simd` (utilizza la vettorizzazione) regioni `parallel`, `omp_get_num_teams()`, `omp_get_teams_num()`. Il costrutto `distribute` distribuisce le iterazioni del loop associato tra i master thread di ogni team che appartiene alla regione. Non c'è una barriera

implicita. `dist_schedule(kind[,chunk_size])` se è specificato lo scheduling deve essere statico.

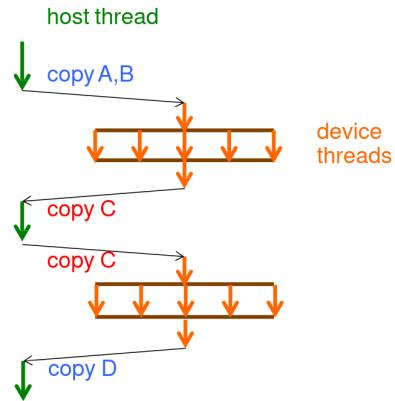
### 13.2 Data mapping

Una **mapped variable** è la variabile corrispondente in un device data environment alla variabile originale nell'host data environment. Un **mappable type** è un tipo di variabile/struttura dati che può essere mappata. La clausola `map` determina come una variabile originale di un data environment viene mappata ad una corrispondente nel device data environment. La sintassi è: `#pragma omp target map([[map-type-modifier[,]] map-type:] list)` dove `map-type` è:

- `alloc`, alloca memoria per la variabile corrispondente
- `to`, alloca e copia host → device
- `from`, alloca e copia device → host
- `tofrom`, sia `to` che `from`

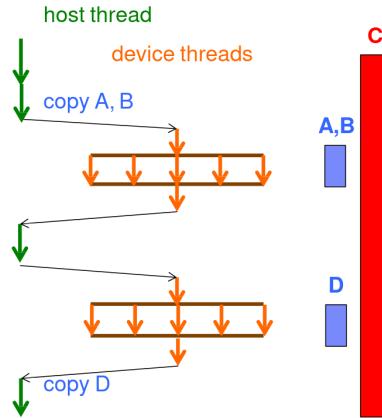
Esempio:

```
double A[n,n], B[n,n], C[n,n];  
#pragma omp target  
map(to: A, B) map(from: C)  
{  
    // define C in terms of A, B  
}  
#pragma omp target  
map(to: C) map(from: D)  
{  
    // define D in terms of C  
}
```



Il data scope è limitato dai costrutti **target**: nessun data scope per la variabile **C** tra i due costrutti risulta in copie non necessarie di **C**. È possibile definire una regione per la quale una variabile è mappata sul device, eventualmente più grande di un blocco target con la **#pragma omp target data**

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target data
map(alloc: C)
{
#pragma omp target
    map(to: A, B)
    {
        // define C in terms of A, B
    }
#pragma omp target
    map(from: D)
    {
        // define D in terms of C
    }
}
```



È possibile aggiornare una variabile offloadata con `target data` in modo completamente non strutturato con la `#pragma omp target update`, la clausole possono essere `device`, `to`, `from`, `if`, `nowait`, `depend`. La direttiva `declare target` dichiara in maniera statica il mapping di certe variabili di un programma su un device per tutta la sua durata. La `target enter data` e la sua duale `target exit data` permettono di mappare variabili in maniera non strutturata: può estendere il suo scope oltre quello di una singola funzione. La `map` non comporta necessariamente una copia: nel caso in cui host e device condividono la memoria e anche il meccanismo di traduzione degli indirizzi non c'è una copia, ma condivisione di puntatori.

## 14 CUDA Basics

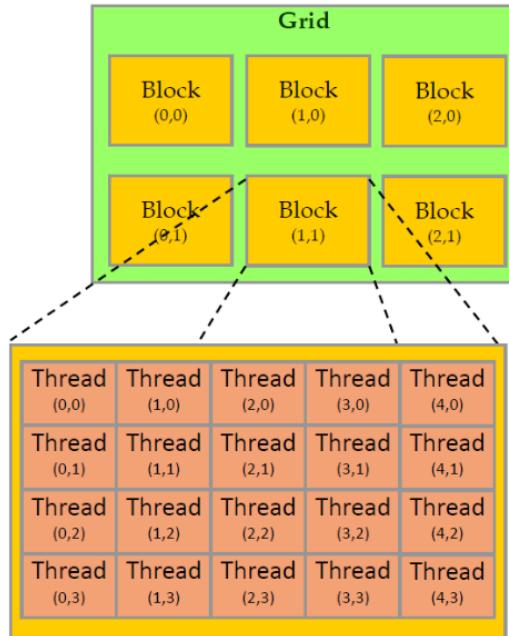
CUDA (Compute Unified Device Architecture) è un software layer che dà accesso all'instruction set virtuale della GPU. Il virtual instruction set è il **PTX** (Parallel Thread eXecution) e segue un paradigma di programmazione multi-threaded: la GPU è vista come una virtual machine many core. CUDA fornisce un'estensione alla tradizionale programmazione C/C++, Fortran per astrarre l'uso del PTX.

### 14.1 Execution model

Nel modello CUDA la CPU principale vede la GPU principale come un coprocessore esterno con propria capacità di storage (dal punto di vista logico le memorie sono separate). Le porzioni di codice altamente data-parallel possono essere *outlined*: codice CPU trasferito all'interno di una funzione

kernel che esegue con il modello **SIMT** (Single Instruction Multiple Thread). Un thread GPU è più leggero - si porta dietro dei contesti che sono gestiti con tanto supporto hardware, dato che l'obiettivo è quello di avere context switch a costo zero. Si nasconde la latenza tramite lo scheduling di un numero di thread molto più alto di quello che è il numero delle ALU fisiche - quando eseguiamo un kernel dobbiamo specificare il numero di thread e far sì che sia abbastanza alto per consentire uno scheduling ottimale. I thread sono raggruppati in **blocks** che a loro volta formano le **grid** - tutti e tre questi elementi possono essere definiti su 3 dimensioni. Per identificare le coordinate del thread o blocco corrente i CUDA kernel usano le variabili:

- **threadIdx**, coordinate del thread nel blocco
- **blockIdx**, coordinate del blocco nella griglia
- **blockDim**, dimensioni del blocco in unità thread
- **gridDim**, dimensioni della griglia in unità blocco



Questa suddivisione è una astrazione del funzionamento di una GPU: il thread si mappa concettualmente su un CUDA core (ALU), un blocco

su uno **streaming multi-processor** e una griglia su più streaming multi-processor. Il pattern di accesso alla memoria è il principale responsabile del buon funzionamento del mio codice: la memoria è un array 1-dimensionale ovvero una serie molto lunga di celle individuate da un indirizzo crescente. Si cerca sempre di scrivere i kernel

## 14.2 CUDA programming

Il procedimento per scrivere un programma CUDA è il seguente:

1. identificare porzioni di codice altamente data parallel ed isolarele (**outline**)
2. capire quali sono le strutture dati da trasferire sul device
3. implementare il CUDA kernel - sarà codice thread-centrico, ogni thread esegue lo stesso kernel ma su dati diversi
4. modificare il codice host perchè ci siano questi step: allocare memoria sul device, trasferire i dati da host a device, lanciare l'esecuzione sull'acceleratore, eventualmente sincronizzare, copyback sull'host

In CUDA è sempre esplicito l'utilizzo di qualunque risorsa (e.g. blocco di computazione, memoria) dunque mi servono keyword che specificano dove compilare parti del codice. **\_\_global\_\_** è la keyword per i kernel GPU; solo host può chiamare questa funzione e deve ritornare void obbligatoriamente. L'esecuzione può essere asincrona.

```
__global__ void gpuVectAdd (const double *u, const double *v,
                           double *z, int N)
{
    // index is a unique identifier of each GPU thread
    int index = blockIdx.x * blockDim.x + threadIdx.x ;
    if (index < N)
        z[index] = u[index] + v[index];
```

Per quanto riguarda la copia dei dati prima bisogna allocare dei buffer (esplícitamente)

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void**)&u_dev, N * sizeof(double));
cudaMalloc((void**)&v_dev, N * sizeof(double));
cudaMalloc((void**)&z_dev, N * sizeof(double));
```

L’indirizzo del buffer allocato (sulla memoria del device) viene passato chiaramente all’host, perchè la `cudaMalloc` viene chiamata sull’host prima di fare offloading. `cudaMemcpy(void*dst, void*src, size_t size, direction)` copia `size` byte dall’indirizzo `src` (puntatore a source) all’indirizzo `dst` (puntatore a host) specificando in che direzione effettuare la copia.

```
cudaMemcpy(u_dev, u, N*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, N*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(z_dev, z, N*sizeof(double), cudaMemcpyDeviceToHost);
```

Dalla versione 4.0 CUDA supporta l’UVA (Unified Virtual Addressing) tra la memoria GPU e quella host: il programmatore tratta le strutture dati (la loro allocazione) come se lo spazio di memoria fosse unico. Può essere fatto su memoria shared ma anche su memoria distribuita e logicamente shared attraverso l’uso della `cudaMemcpyDefault`. Per invocare la kernel function da lato host:

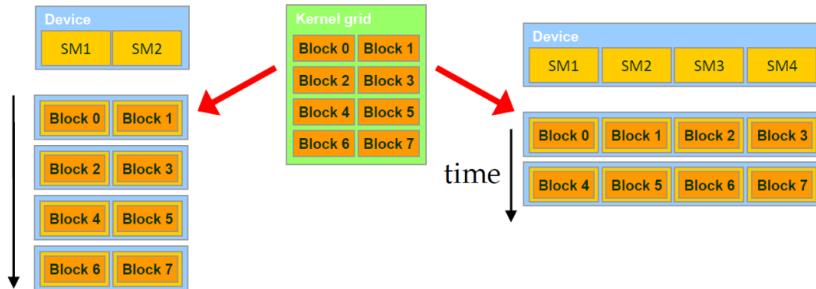
```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void**)&u_dev, N * sizeof(double));
cudaMalloc((void**)&v_dev, N * sizeof(double));
cudaMalloc((void**)&z_dev, N * sizeof(double));
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
dim3 numThreads(256); // 128-512 are good choices
dim3 numBlocks((N + numThreads.x -1) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>(u_dev, v_dev, z_dev, N);
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

Esistono tre qualificatori per i diversitipi di funzione:

- `__device__` eseguibile e chiamabile solo sul device
- `__host__` eseguibile e chiamabile solo sull’host
- `__global__` eseguibile sul device e chiamabile solo sull’host

Esiste quindi un mapping diretto tra le astrazioni offerte a livello CUDA (thread, block, grid) e i componenti architetturali sottostanti (ALU/processore, streaming multi-processor, device). Lo streaming multi-processor organizza i suoi processori in gruppi (tipicamente da 32) che possono essere visti come un’unità SIMD di profondità 32 - l’unità di fetch/decode è comune al gruppo. In CUDA parliamo di **warp** quando intendiamo un gruppo di thread di numero tale da tenere occupati tutti i processori (ALU) all’interno dell’unità

SIMD. Il numero di thread può essere organizzato in maniera tale che ci siano molti più gruppi delle unità fisiche - lo scheduler selezionerà di volta in volta il warp per l'esecuzione e se un blocco va in stallo (e.g. accede alla memoria) allora può deschedulare un warp per un altro. Il programma deve essere embarrassingly parallel: non c'è bisogno di sincronizzare l'esecuzione perché non c'è un ordine di esecuzione - i blocchi sono indipendenti. Le astrazioni aiutano la scalabilità: la struttura Nvidia è sempre organizzata allo stesso modo e il *programming model* riflette questo tipo di struttura, dunque si può scalare verso un numero di multi-processor arbitrario. Non c'è un mapping vincolante 1-1 tra streaming multi-processor e block. Quando un programma CUDA lancia un kernel su un device, i blocchi nella griglia sono enumerati e distribuiti in base alla capacità di esecuzione disponibile.



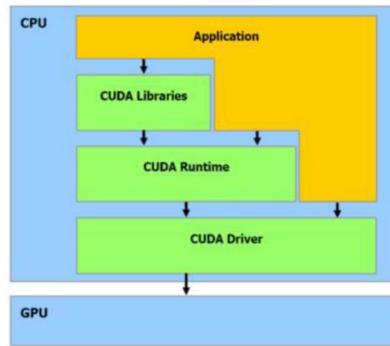
I thread in un blocco sono eseguiti concorrentemente su uno streaming multi-processor e i thread appartenenti a multipli blocchi possono eseguire concorrentemente su streaming multi-processor. Se un blocco termina o stalla, altri blocchi vengono mandati sugli streaming multi-processor idle. I registri sono assegnati dinamicamente ai blocchi di uno streaming multi-processor. I thread di un blocco hanno accesso solamente ai registri che gli sono assegnati. Il context switch a costo zero è possibile perché la ri-schedulazione dei warp non sovravascrive le informazioni contenute nei registri dei blocchi. Esistono dei limiti sul numero di risorse hardware che posso richiedere: il context switch è a costo zero ma il contesto usa delle risorse di tipo registro e shared memory. Esempio:

- architettura da 32768 registri
- blocchi da 32x8 thread
- kernel necessita di 30 registri Ogni blocco ha bisogno di  $30 \times 32 \times 8 = 7680$  registri.  $32768 \div 7680 = 4,27$ , cioè più eseguire 4 blocchi. Se

l'uso dei registri aumenta del 10%?  $33 \times 32 \times 8 = 8448$ ,  $32768 \div 8448 = 3,88$  può eseguire solo 3 blocchi alla volta! Ho una riduzione del parallelismo.

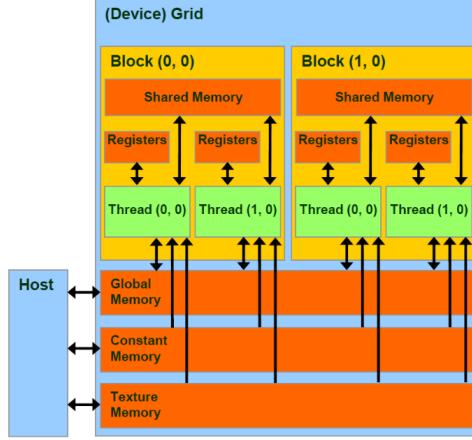
### 14.3 CUDA driver vs Runtime API

I programmi CUDA supportano due interfacce: C for CUDA (CUDART) e CUDA driver API. La driver API è più di basso livello e mi dà più controllo sul software sottostante, soprattutto di come viene caricato un kernel, mentre la CUDA runtime è un livello di astrazione più alto.



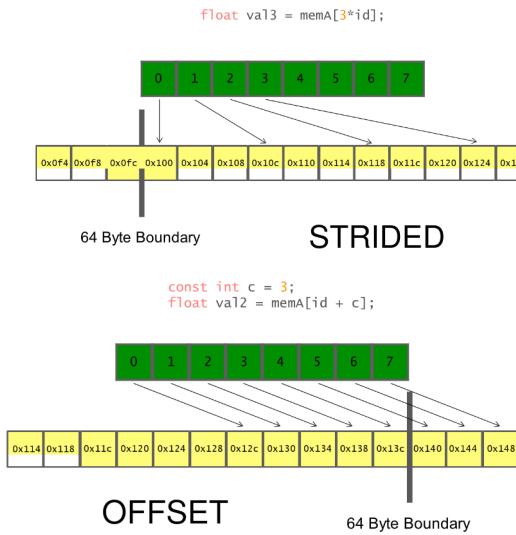
## 15 CUDA Memory Model

Il fatto di avere context switching a costo zero è reso possibile dal fatto che replichiamo i registri in un numero sufficiente da dare la possibilità a più di un blocco di esistere contemporaneamente sull' architettura. I thread di un blocco posso accedere ai registri e alla shared memori. Inoltre, tutti i thread possono accedere alla global memory, alla costant memory e alla texture memory. La CPU può scrivere sulla constant e texture memory.



### 15.1 Global Memory

La global memory è la memoria più grande del sistema: ci risiede il programma ed il grosso dei dati. È persistente tra l'invocazione di un kernel e l'altra; la GPU, la CPU e tutti i thread possono accedervi. Dato il suo mapping con la DRAM ha una latenza d'accesso molto alta. Ha anche una bandwidth larga. Per l'allocazione (statica) esplicita uso la keyword `--device--`. Si ha **memory coalescence** quando gli accessi dei vari thread in un warp sono tutti distanziati esattamente di una word e sono allineati ai boundaries della dimensione della linea. Tipicamente però ci sono degli accessi **strided** o con **offset**.



Tra la global memory e gli streaming multi-processor ci sono tipicamente due livelli di cache: L2 (2MB), condivisa dagli streaming multi-processor ma ulteriormente clusterizzabile ed L1 (16-48KB), privata ai singoli streaming multi-processor. Gli accessi alla global memory passano per la L2. In origine la shared memory era interamente una scratchpad, successivamente si è aggiunta la possibilità di configurarla in parte come cache e in parte come scratchpad. Nel datapath del processore CUDA c'è un'unità dedicata: la **load-store unit**, che gestisce l'accesso alla gerarchia di memoria. I thread di un warp scrivono l'indirizzo al quale vogliono accedere sulle load-store unit; esse hanno la responsabilità di calcolare il routing, l'indirizzo che voglio leggere/scrivere e si coordinano tra di loro per far partire queste richieste. Il sistema è sensibile al discorso dell'allineamento: se i warp accedono in maniera contigua, ma non allineata alla word in memoria, potremmo essere soggetti ad un tipo di access pattern con offset e questo abbatterebbe le performance. Quando utilizziamo la memoria allocata dinamicamente la `cudaMalloc` ci garantisce che il primo elemento del buffer di memoria sia allineato alla struttura della global memory. Funziona bene con array 1-dimensionali. Inoltre, abbiamo a disposizione una primitiva `cudaMallocPitch` che ragione in termini di 2 dimensioni: garantisce che il primo elemento di ogni riga sia allineato alla global memory attraverso il **padding** - mi rimarrà però un pezzetto di riga non utilizzato.

## 15.2 Shared Memory

La CUDA shared memory è privata al blocco: solo i thread del blocco possono fare read/write su essa. È non persistente, cioè non è garantito che tra una chiamata ad un kernel e l'altra le variabili permangano. Ha bassa latenza di accesso (cache di 1° livello, 1-2 cicli) e la dimensione di default è 48KB, configurabile a 16KB o 32KB. A 16KB lascia più spazio alla L1 cache che è automatica (buona scelta se inesperti). A 48KB invece si dà maggior spazio alla scratchpad, il cui uso è a carico del programmatore. Il fatto di condividere la memoria tra tanti thread è soggetto a contesa nel caso ci sia un solo ingresso per fare scrittura o lettura: questo potrebbe portare ad una latenza fino a 32 cicli. Posso però organizzare la memoria in **banchi** (32 nello specifico) ognuno con una porta dedicata. Se i thread rispettano lo stesso comportamento *coalescent* visto per la global memory allora posso davvero ho la latenza di 1 ciclo nel caso in cui 32 thread accedono ad indirizzi contigui. L'organizzazione dell'indirizzamento deve essere tale che indirizzi contigui vadano su banchi contigui (i.e. indirizzo 0 sul banco 0, indirizzo 2 sul banco 1, indirizzo 8 sul banco 2, etc.). Questa suddivisione viene chia-

mata **address interleaving** e grazie ad essa posso soddisfare 32 richieste in un solo ciclo. Se  $n$  thread dello stesso warp cercano di accedere allo stesso elemento, l'accesso viene eseguito in 1 transaction, ovvero un **multicast**. Nel caso in cui siano tutti i thread a voler accedere allo stesso elemento si parla di **broadcast**. Rimangono però conflitti (ed accessi serializzati) nei casi di tentativi di accesso da parte di più thread allo stesso banco verso elementi diversi. Per utilizzare la shared memory ho due opzioni: allocazione statica (`__shared__ type shmem[MEMSZ];`) o allocazione dinamica (`extern __shared__ type *dynshmem;`). Nel secondo caso non è nota a priori la dimensione del buffer che va sul device; si determinerà a runtime e sarà un parametro di offloading:

```
extern __shared__ type *dynshmem;
__global__ myKernelOnGPU (...) {
    ...
    dynshmem[i] = ... ;
    ...
}
void myHostFunction() {
    ...
    myKernelOnGPU<<<gs,bs,MEMSZ>>>();
}
```

Se servono più array di dimensione non nota questo meccanismo non si può estendere arbitrariamente: bisogna impacchettare e spacchettare a mano. Dichiaro un array `extern` di dimensione sconosciuta ed uso dei puntatori per dividerlo in array multipli.

```
extern __shared__ int s[];
int *integerData = s; // nI ints
float *floatData = (float*)&integerData[nI]; // nF floats
char *charData = (char*)&floatData[nF]; // nC chars
```

### 15.2.1 Thread synchronization

Quando si usa un approccio di tipo *load* nella shared memory (da parte di un thread) è necessario che tutti i thread si accertino che la load sia stata effettuata. CUDA API mette a disposizione la `__syncthreads()` come principale primitiva di sincronizzazione. Tutti i thread di un blocco si fermano finché tutti non hanno raggiunto quel punto (come se fosse una barriera per il blocco). Bisogna fare attenzione ai condizionali: la `__syncthreads()` deve

comparire in tutti i branch del control flow. In generale è più efficiente progettare algoritmi che evitano la sincronizzazione sulla GPU quando possibile, soprattutto sulla memoria globale.

```
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}
// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

In questo esempio dobbiamo assicurare che T1 legga la locazione dopo che T0 ha effettuato la write.

```
// Cuda code
__global__ void update(int* x, int* y) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    __syncthreads();
    if (i == 1) *y = *x;
}
// Host code
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

### 15.3 Constant Memory

È utile per casi in cui si usino dati statici (e.g. coefficienti, pesi) non noti a *compile time* che vengono copiati una volta per tutte. I dati sono memorizzati sulla global memory, ma è presente una cache dedicata, la **constant-cache**, che evita accessi multipli da parte dei thread. Il primo warp paga la latenza per popolare la cache, ma successivamente in maniera deterministica i dati saranno disponibili per gli altri warp nella cache. La dimensione di questa memoria è 64KB, lo scope è l'intero file (variabile globale), è visibile da tutti i thread della griglia e il suo tempo di vita è quello dell'applicazione. Per allocare una variabile costante sulla global memory uso la keyword **`__constant__`**.

```

__constant__ type variable_name; // static
cudaMemcpyToSymbol(variable_name, &host_src,
    sizeof(type), cudaMemcpyHostToDevice);
// NO dynamic allocation

```

## 15.4 Registri

Le variabili locali ai thread sono memorizzati sui registri (stack, memoria automatica). È necessario monitorare il numero di registri usati perché più alto è il numero di registri utilizzati, minore è il numero di thread/blocchi di thread che possono eseguire su uno streaming multi-processor. È possibile dare dei bound specifici (numero di thread per blocco, numero di blocchi per griglia) possiamo usare il qualificatore

```

__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)

```

### 15.4.1 Local memory

Non è una memoria fisica ma un concetto astratto di memoria, utilizzato per ospitare la memoria automatica dei thread (e.g. stack). Il compilatore tenterà di usare dei registri per lo stack, ma ad un certo punto farà **spilling** sulla memoria globale.

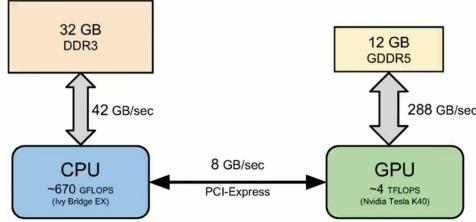
## 15.5 Grid Synchronization

CUDA dispone di funzioni atomiche per rinforzare l'accesso atomico alle variabili condivise tra thread dello stesso blocco, ma anche thread di blocchi diversi della stessa griglia (e.g. addizione, minimo/massimo, incremento/decremento, scambio atomico di un valore, etc.). Non è una vera e propria sincronizzazione, ma almeno è qualcosa. È da evitare quando possibile in quanto abbassa la performance.

# 16 CUDA Advanced Features

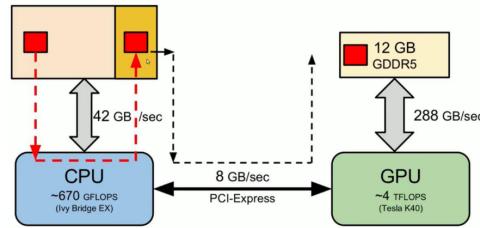
L'impatto del trasferimento dati consuma molto tempo. Non c'è solo un discorso di trasferimento all'atto dell'esecuzione del kernel, ma ci sono una serie di azioni implicite che sotto comportano il movimento di dati: quando facciamo un offloading il driver sottostante implicitamente fa il loading del binary code del kernel e poi, sempre il driver, si occupa anche di caricare gli argomenti del kernel. La comunicazione tra CPU-GPU (nel caso di un

dispositivo discreto) è il punto dolente e CUDA dedica molto effort ad ottimizzarlo.

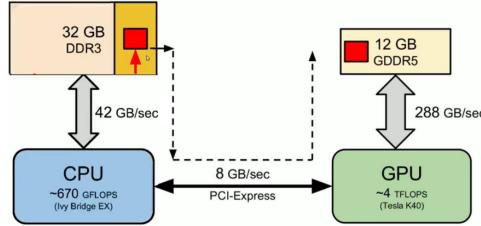


## 16.1 Pageable and Pinned Memory

La pageable memory è la porzione di memoria che risiede in user space e può essere swappata su disco.

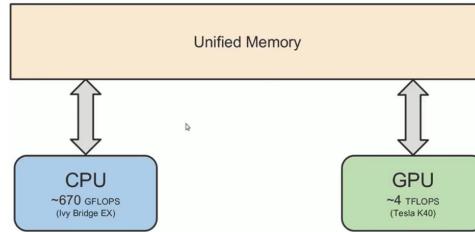


Il paging comporta dei problemi nel momento in cui ci troviamo a dividere memoria tra CPU e GPU; prima di poter essere usata dalla GPU (i.e. traducendo indirizzo virtuale in fisico) deve essere copiata in kernel memory. Si riserva una fetta della DRAM della CPU e dice al SO di non trattarla con il paging, ma usarla come buffer di memoria (si fa una memup). Per fare questo c'è bisogno di una copia di primo livello del dato - successivamente una di secondo livello, dalla regione non-paged CPU alla GPU. Questo sottopone tutto l'execution model ad un ulteriore inefficienza: alla porzione non paged non è permesso di entrare in cache, dunque la CPU deve fare avanti indietro. Un altro modo per consentire questo scambio è la pinned memory (o **page-locked**): una regione di memoria direttamente accessibile da programmatore che risiede nel kernel space, non soggetta a swap (banda alta) e che non beneficia del meccanismo di caching lato host. Devo fare attenzione alla dimensione di questa regione perché ho spazio limitato.



## 16.2 Unified Virtual Memory

Astrazione per semplificare l'accesso alla memoria a livello logico, ma non necessariamente fisico. Logicamente shared, fisicamente distribuita.



Dal punto di vista del programmatore lavorare con una UVM

- semplifica la programmazione: si possono condividere i puntatori, evita memcpy, etc.
- essendo gestito internamente non è detto che sia fatto nella maniera ottimale (come avrebbe fatto il programmatore)
- elimina la necessità delle *deep copy* semplifica la copia delle struct e delle liste. Si osserva che, seppur la UVM sia comoda dal punto di vista del programmatore, in realtà nasconde copie che si pagano dal punto di vista della performance. Con pageable o pinned memory è il programmatore a trasferire in maniera monolitica tutti i dati necessari, una sola volta. Con UVM è il runtime a dover fetchare i dati on-demand e, non sapendo quanti di questi siano necessari le performance ne risentiranno (+ copie -> + overhead). Inoltre UVM implica l'uso di critical section e dunque un ulteriore rallentamento.

## 16.3 Asynchronous Data Transfer

`cudaMemcpy()` è bloccante, ritorna quando ha terminato. `cudaMemcpyAsync()`, invece, ritorna immediatamente il controllo alla CPU: per poter usare la

memoria asincrona c'è bisogno di memoria pinned o degli **streams CUDA**, che permettono di realizzare il latency hiding (anche a livello dell'intero kernel). Li immagino come una coda FIFO di comandi GPU: copie di memoria, esecuzioni di pezzi di kernel ed eventi. CUDA dalla v6 funziona sempre come uno stream, di default è lo 0, ma posso definire altri stream che saranno slegati tra loro. Do agli scheduler hardware della GPU maggior libertà di buttar dentro lavoro: prima potevo farlo solo tramite la descrizione della griglia. Stream diversi possono eseguire in maniera disordinata tra di loro, ma ordinata rispetto al singolo stream.

#### 16.3.1 Sincronizzazione Esplicita

- `cudaDeviceSynchronize()`: la più coarse-grained, stalla l'esecuzione sull'host finchè tutti i comandi su tutti gli stream non hanno completato.
- `cudaStreamSynchronize(stream)`: stalla l'esecuzione sull'host finchè tutti i comandi dello stream specificato non hanno completato
- `cudaStreamWaitEvent(streamX, event)`: i comandi sullo stream X aspetta che l'evento `event` venga eseguito

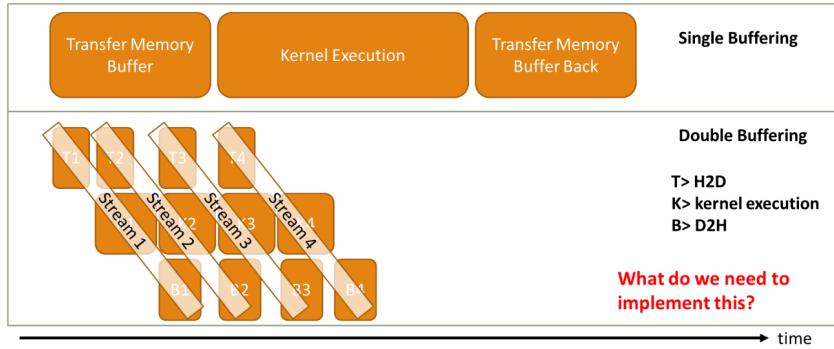
#### 16.3.2 Sincronizzazione Implicita

- tutti i comandi dello stream di default
- allocazioni di memoria *pagelock*: `cudaMallocHost`, `cudaHostAlloc`
- allocationi di memoria sul device: `cudaMalloc`
- versioni bloccanti delle primitive di copia: `cudaMemcpy`, `cudaMemset`

#### 16.3.3 Gestione CUDA Streams

- creare uno stream: `cudaStreamCreate(streamPointer)`
- sincronizzarlo con l'host: `cudaStreamSynchronize(streamPointer)`
- distruggere uno stream: `cudaStreamDestroy(streamPointer)`

Gli streams e i trasferimenti asincroni sono utili per fare overlap tra computazione e trasferimenti di memoria all'interno del device, sfruttando la tecnica del **double buffering**, che sfrutta l'esecuzione su stream multipli per migliorare la concorrenza.



Implementazione:

```

cudaStream_t stream[4];
for (int i=0; i<4; i++)
    cudaStreamCreate(&stream[i]);

float* hPtr, * d_inp, *d_out;
cudaMallocHost((void**)&hPtr, 4 * size); //Allocates memory pinned memory on host
cudaMalloc((void**)&d_inp, 4 * size);    //Allocates memory memory on device
cudaMalloc((void**)&d_out, 4 * size);    //Allocates memory memory on device

for (int i=0; i<4; i++) {
    cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);
    cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
    size, cudaMemcpyDeviceToHost, stream[i]);
}

cudaDeviceSynchronize();
for (int i=0; i<4; i++)
    cudaStreamDestroy(&stream[i]);

```

Il double buffering viene usato in algoritmi locali (e.g. kernel filtering) e per algoritmi globali se si hanno stream di dati dove non ci siamo dipendenze tra i frame.

#### 16.4 Dynamic Parallelism

Un punto critico di CUDA è l'offloading: se riuscissi ad organizzare la mia computazione la maggior parte del tempo dentro la GPU avrei meno prob-

leme di andare avanti indietro. Dalla v3.5 è possibile lanciare un kernel a partire da un altro kernel, senza dover tornare sull'host.

```
// Host Code
ParentKernel<<<256, 64>>(data);

// Device Code
__global__ ParentKernel(void *data){
    ChildKernel<<<16, 1>>>(data);
}
__global__ ChildKernel(void* data){
    ...
}
```

È utile nei casi in cui il partizionamento non sia ottimale - finora abbiamo visto partizionamento statico, cioè decido a priori che dimensioni ha la griglia dei thread. La tecnologia che permette il parallelismo dinamico chiama **Hyper-Q** e permette, tra le altre cose, di riorganizzare la griglia in maniera dinamica. Rende più semplice la programmazione e permette di riusare dati già presenti sulla GPU senza dover rifare trasferimenti di dati.

## 16.5 Summary of Key Performance Issues

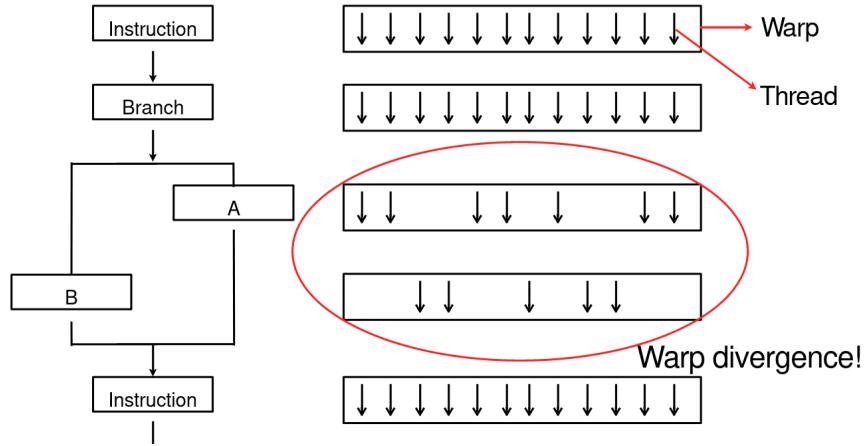
### 16.5.1 Shared Memory

Per la shared memory abbiamo visto:

- il **tiling**, ovvero spezzare loop di grandi dimensioni in altri più piccoli che sfruttano la shared memory per evitare di accedere alla global memory
- il **banking**, cioè organizzare la memoria in banchi per permettere l'accesso a dati diversi (se su banchi diversi) a più thread in contemporanea. Ci genera serializzazione nel caso ci sia un conflitto di accesso di più thread sullo stesso blocco.

### 16.5.2 Control Flow

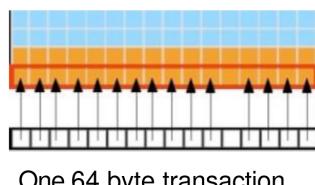
I thread sono eseguiti in *warp* e all'interno di uno stesso warp l'hardware non è capace di eseguire i branch condizionali (if/else) contemporaneamente.



Bisogna provare a far sì che ogni thread nello stesso warp faccia la stessa cosa, dunque evitare branch o fare branch a multipli della warp size (e.g. `if (threadId < 32) .... else ...`); in quel caso non c'è **warp divergence** o **control divergence**, ma dobbiamo stare attenti perché non possiamo fare affidamento sull'ordine di esecuzione dei vari wrap.

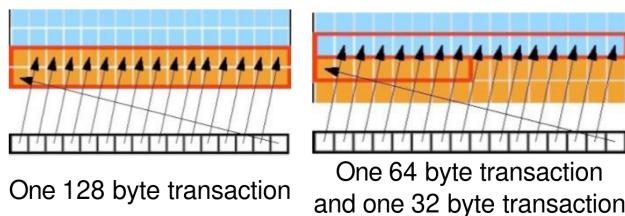
### 16.5.3 Memory Coalescence

L'accesso alla global memory avviene in chunk di 32, 64 o 128 byte allineati. La dimensione della transaction dipende dal fatto che ci siano o meno altri thread che sfrutterebbero i dati fetchati; se non ci sono, la transazione sarà da 32 (il minimo). Il caso peggiore è quando ogni thread fetcha 32byte e ne usa soltanto 4 (accesso strided). Patter d'accesso semplice:

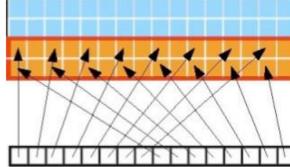


One 64 byte transaction

Accesso sequenziale, ma disallineato:



Accesso *strided*:



One 128 byte transaction, but half of bandwidth is wasted

#### 16.5.4 SoA vs AoS

Un array di struct si presenta così:

```
struct record
{
    int key;
    int value;
    int flag;
};

record *d_record;
cudaMalloc((void**) &d_records, ...);
```

Mentre una struct di array in questo modo:

```
struct SoA
{
    int *key;
    int *value;
    int *flag;
};

SoA *d_AoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.value, ...);
cudaMalloc((void**) &d_SoA_data.flag, ...);
```

La SoA è più efficiente in quanto la `cudaMalloc` garantisce memoria allineata.

```
global void bar (record *AoS_data, SoA SoA_data)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    // AoS wastes bandwidth
```

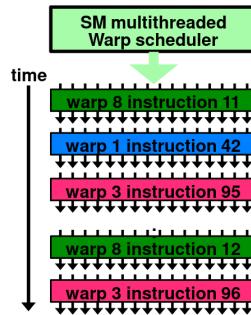
```

int key = AoS_data[i].key;
// SoA efficient use of bandwidth
int key_better = SoA_data.keys[i];
}

```

### 16.5.5 Latency Hiding

Un warp non viene schedulato finché tutti i thread non hanno finito l'istruzione precedente; queste istruzioni possono avere alta latenza (e.g. accesso alla global memory). Idealmente ci vorrebbero abbastanza warp da tenere sempre occupata la GPU durante l'attesa.



### 16.5.6 Occupancy

È il rapporto tra i warp ed il massimo numero di warp e dipende da:

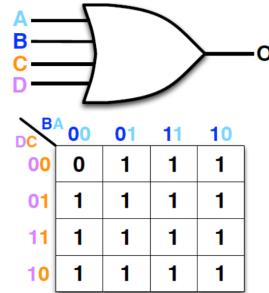
- massimo numero di blocchi per SM
- massimo numero di thread
- shared memory
- registri

Una bassa occupancy può portare ad avere problemi nel fare latency hiding.

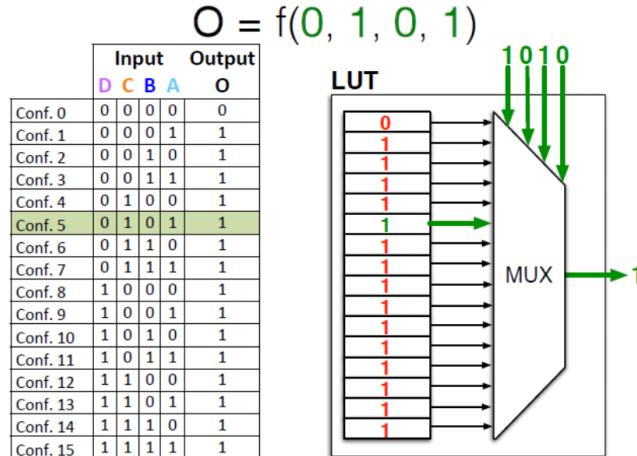
## 17 Reconfigurable FPGA-based System Design

Una FPGA è una griglia composta da **CLB**, fili e **pad I/O** che permette di realizzare una funzione logica - sequenziale o combinatoria. All'interno dei CLB troviamo la **LUT** (look-up table), multiplexer che valuta una tabella

di verità memorizzata in SRAM, e i **flip-flop**, che permettono la memorizzazione di bit e l'implementazione la logica sequenziale. Un esempio è l'OR a 4 input:



Se vogliamo calcolare  $O = f(0, 1, 0, 1)$  allora:



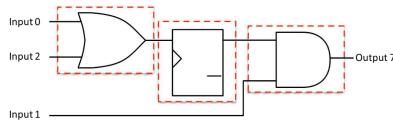
All'interno delle FPGAabbiamo inoltre i **CB**, che interconnettono i CLB e fanno muovere il segnale lungo un filo e una direzione e le **SB** (o **SM**), che invece sono utilizzate per connettere orizzontalmente e verticalmente le CB e fare il routing del segnale. La **routability** è la misura di quanti circuiti possono essere implementati: una maggiore flessibilità comporta una miglior routability. La memoria è implementata in due possibili modi:

- embedded (**block RAM**), cioè componenti discreti considerabili come blocchi che vanno ad inserirsi nella matrice interconnessa tramite gli switch box

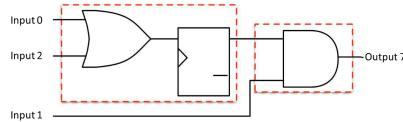
- **distributed RAM**, cioè realizzata combinando i flip-flop per realizzare dei registri, array di celle e quindi RAM

## 17.1 Technology Mapping

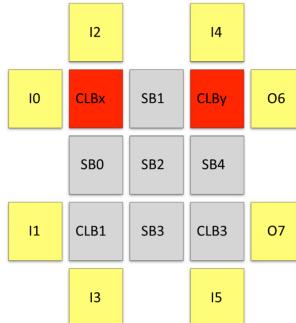
Per mappare questo circuito 2x2 semplificato abbiamo due modi: usare 3 CLB o 2 CLB



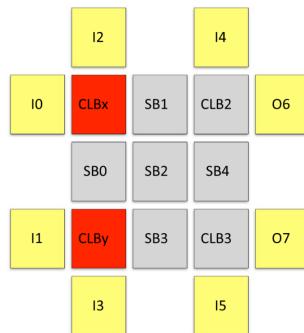
o 2 CLB



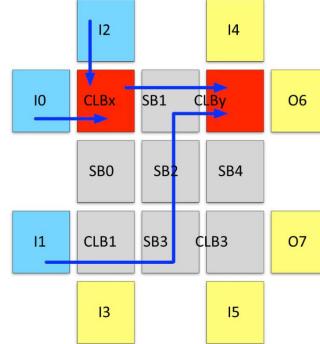
Scegliamo di utilizzare 2 CLB, ora procediamo con il placement.



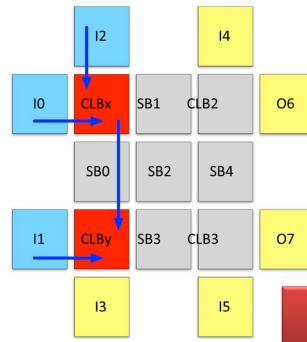
Oppure:



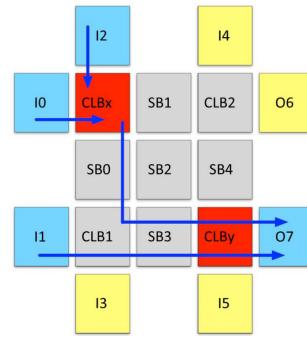
Dopodichè dobbiamo decidere il routing. Soluzione con 3 SB:



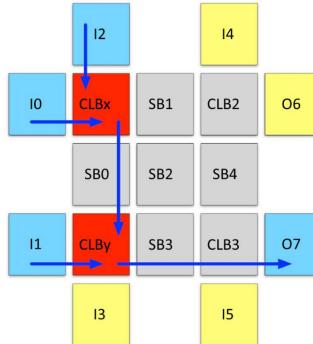
E una soluzione con 1 SB:



Ora dobbiamo decidere l'output pin. In questa soluzione abbiamo spostato CLBy e utilizziamo 2 SB e 3 wires:



Riusciamo però ad utilizzare solo 2 wires, posizionando gli elementi della FPGA in questo modo:

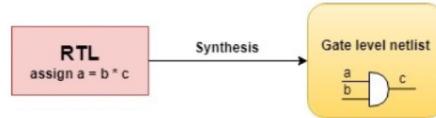


## 17.2 FPGA Bitstream

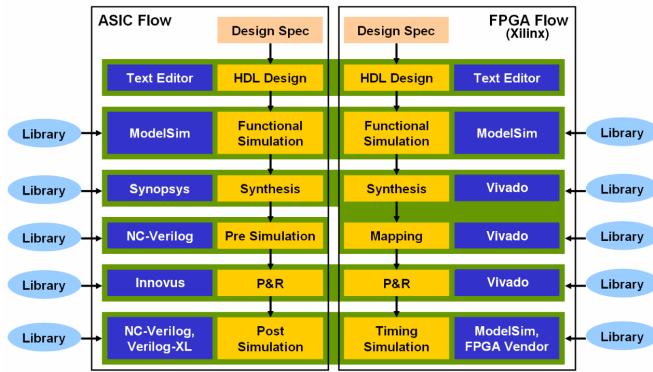
Realizzare una rete logica/circuito/design hardware su FPGA comporta andare a configurare individualmente i blocchetti architetturali. Per svolgere questo compito si introduce il concetto di **bitstream**: collegandosi ad un sistema host (e.g. porta seriale) inviamo un flusso di bit che va a configuarare i comportamenti delle celle della FPGA. È dunque un file di configurazione che risulta dalla sintesi di **placement** e **routing**. Solitamente la FPGA è configurata in modo completo, ma è possibile anche riconfigurarla parzialmente in maniera statica (stop altre parti per caricare bitstream) o dinamica (mantengo in esecuzione e carico bitstream). Inoltre, la riconfigurazione può avvenire in maniera esterna o interna/embedded (la parte logica della FPGA funge da manager)

## 18 High-Level Synthesis (HLS)

La sintesi è la generazione di un bitstream a partire da un design. Come si genera il bitstream di configurazione per le FPGA? Una possibilità è l'Hardware Description Language (HDL). Ho a disposizione dei costrutti sintattici per descrivere il comportamento di una rete logica. Il livello di astrazione del HDL è quello del RTL (Register Transfer Level). Il linguaggio mette a disposizione dei **building block**: operatori che sono gli equivalenti delle porte logiche, fili, adders, multiplier, flip-flop, etc. Ho bisogno di keyword per gestire il clock e keyword per capire come rendere sensibili al clock gli elementi di logica combinatoria. La sintesi è il processo con il quale, a partire da una descrizione astratta dell'hardware arrivo a quella che si chiama **netlist gate-level** (gate è un componente del transistor). Nel FPGA il netlist è un bitstream; non arrivo a progettare transistor, ma i bit di configurazione dei componenti discreti (CLB, SB, etc.).

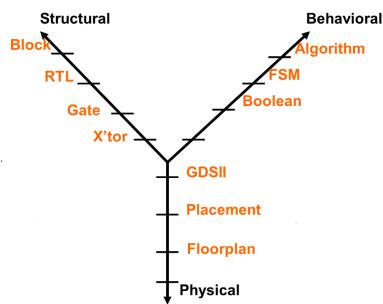


Dalla netlist possono derivare due processi: ASIC o FPGA.



- **functional simulation:** la uso prima di sintetizzare per studiare la logica e metterla a punto (non uso ancora la FPGA, ma il PC).
- **mapping:** che blocchi utilizzare? Come mappare le porte sui CLB? E i flip-flop?
- **place and route (P&R):** nel caso FPGA è decidere dove mettere i CLB e fare il routing. Per l'ASIC è la mappa fisica dei transistor. È una fase più lunga rispetto ad FPGA, è un processo iterativo per arrivare sul silicio.

Lo sviluppo dell'hardware è percepito su tre domini: *comportamentale, strutturale, fisico*. Essi vengono rappresentati nel grafico Y-chart nel quale man mano mi avvicino al centro scendo verso livelli di astrazione più bassi.



- **Layout synthesis:** a partire da una rappresentazione a livello di gate netlist vado verso il placement. *structural* → *physical*
  - **Logic synthesis:** a partire dalla descrizione della rete logica (boolean) mi porta ad un equivalente RTL. *behavioural* → *structural*
  - **High level synthesis:** posso astrarre ad alto livello (C/C++) parto dall'algoritmo e arrivo ai blocchi. *behavioural* → *structural*
- HLS è una metodologia di design hardware automatizzata che trasforma un linguaggio di alto livello in un equivalente RTL adatto per l'implementazione hardware. I benefit più notevoli sono:

1. *produttività*: costa meno in termini di mesi/uomo
2. *portabilità*: da una singola versione del codice è facile derivare multiple implementazioni
3. *permutabilità*: consente una rapida **design space exploration** (esplorare le possibili varianti)

HLS è un sottoinsieme del C/C++ con però delle estensioni che permettono di descrivere delle caratteristiche tipiche dell'hardware. Per estrarre l'hardware dal codice andiamo a scomporre il codice in **control flow** e **data flow**. Il compilatore HLS deriva il control flow graph e il data graph a partire dai quali può generare delle strutture di controllo (FSM, macchine a stati finiti) e il data path (operazioni di calcolo vero e proprio). La fasi di scheduling e binding sono molto importanti:

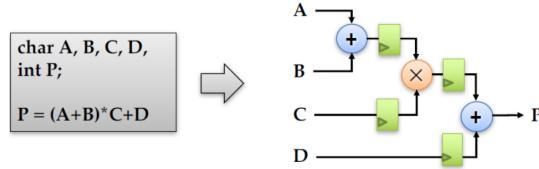
- scheduling: in quale ciclo farò un'operazione? Quanti cicli impiegherò per fare le varie operazioni? Devo tenere in considerazioni i limiti della tecnologia e le direttive dell'utente.
- binding: quali operatori dovrò utilizzare per fare l'operazione? Devo mappare gli operatori ai core dalle librerie hardware

I costrutti C sono mappati sulle componenti hardware in questo modo:

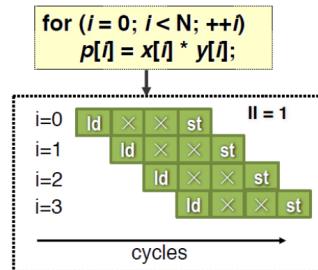
- funzioni → moduli
- argomenti → porte I/O
- operatori → unità funzionali
- scalari → fili o registri

- array → memorie
- control flow → logica di controllo (FSM)

HLS genera circuiti datapath per la maggior parte a partire da espressioni:

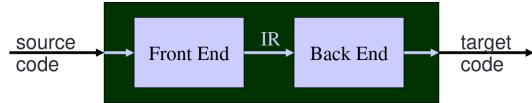


Di default i loop vengono eseguiti una interazione per volta, in ordine; per avere un parallelismo maggiore usiamo il loop unrolling e il loop pipelining. Il loop unrolling (`#pragma HLS unroll`) serve per esporre maggiormente il parallelismo ed ottenere una latenza inferiore. D'altro canto però aumenta il numero di operazioni da eseguire, dando effetti negativi su area, potenza e tempo di compilazione (che aumenta esponenzialmente). Anche il loop pipelining (`#pragma HLS pipeline`) è un'ottimizzazione importante per l'HLS; permette che una nuova iterazione inizi ad eseguire prima che la precedente abbia completato.

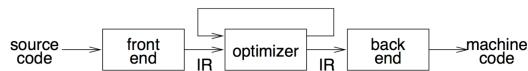


## 19 Optimizing Compilers for Heterogeneous Systems

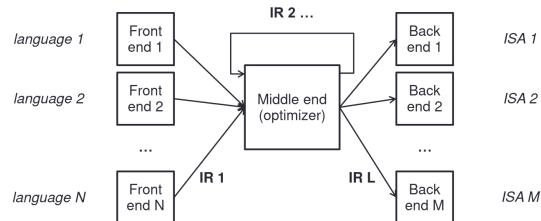
Un compilatore moderno è tipicamente strutturato con un componente di **front end** e uno di **back end** che interagiscono scambiandosi informazioni tramite un formato intermedio (IR). Il compito di un compilatore è quello di generare codice per una macchina target a partire da un linguaggio; questo è il code generation/code synthesis. Un'altra cosa che fa è il processo di analisi. L'analisi è un componente fondamentale anche per l'ottimizzazione.



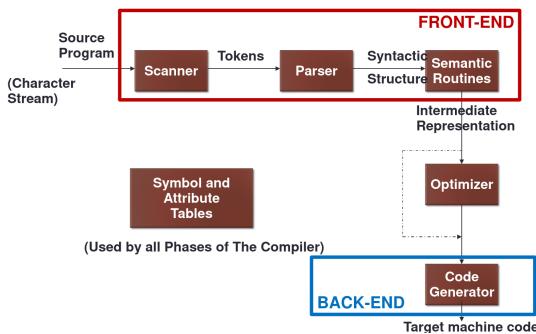
A partire dalla IR derivata dal sorgente vado verso un blocco centrale, il cuore del compilatore moderno, chiamato ottimizzatore (o middle-end). Questo blocco è una pipeline che lavora con la IR, trasformandola con l'obiettivo di ricavarne una versione ottimizzata. Una volta ottenuta questa versione passiamo al blocco di back end che trasforma il codice ottimizzato in codice macchina.



Le IR vengono utilizzate perché offrono modularità e permettono di svincolarsi dalle specificità del linguaggio e dell'architettura della macchina (**machine-independent optimization**). Un compilatore ha tanti front-end, che supportano linguaggi differenti e tanti back-end, che contengono la **machine description** delle diverse ISA. Normalmente una pipeline di ottimizzazione ha centinaia di passi e il risultato cambia in base all'ordine di applicazione. Per questo spesso i passi si ripetono più volte dopo certe ottimizzazioni.



La struttura di un compilatore, più nello specifico è la seguente:

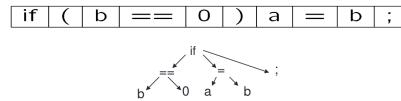


## 19.1 Scanner

Diamo in ingresso il programma come stringa di caratteri. Ha la responsabilità di creare dei **token**, attraverso l'analisi lessicale. Possiamo dare un primo feedback nel caso ci siano delle espressioni non riconoscibili.

## 19.2 Parser

Lavora con le grammatiche, fa analisi sintattica. A partire dai token costruisce l'**abstract syntax tree** che è una rappresentazione intermedia, ma di alto livello (i.e. specifica per il linguaggio).



## 19.3 Semantic Routines

Ultimo stadio del front-end. Fa analisi semantica di ogni costrutto e fa effettivamente la traduzione.

## 19.4 Code Generator

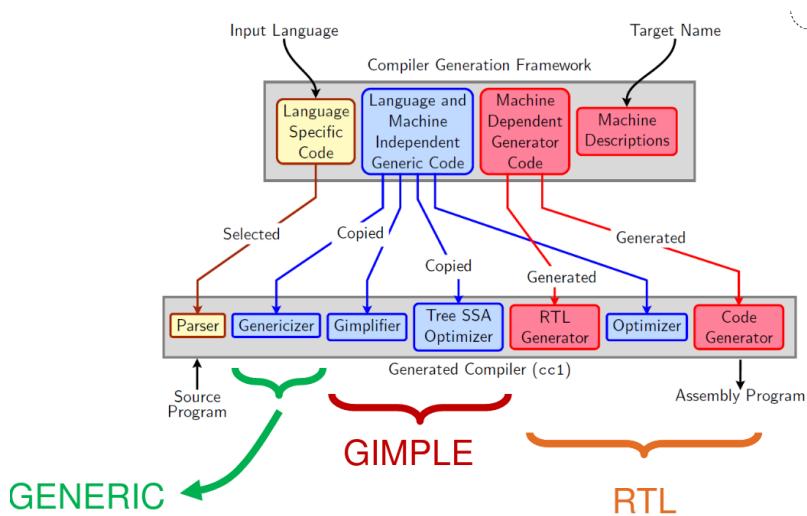
Fa parte del back-end. Genera codice macchina e al suo interno sono incluse tutte le trasformazioni e le ottimizzazioni che sono specifiche della macchina. Esempio: la vettorizzazione. Possiamo progettare il nostro hardware affinchè abbia delle unità SIMD programmabili in modo che una singola iterazione del loop srotolato processi  $n$  elementi in parallelo (con  $n$  larghezza del vettorizzatore). La vettorizzazione non si può implementare tutta dentro l'ottimizzatore! Infatti non conosco i dettagli dell'hardware, banalmente non so quando è profonda la SIMD. Solitamente questo processo avviene in due fasi: dentro l'ottimizzatore trasformo i miei loop perché diventino paralleli (e.g. analisi delle dipendenze, cambio ordine loop, etc.) e a questo livello marchio il loop dicendo "è parallelizzabile con questa granularità". Il backend considera queste informazioni ed genera le istruzioni SIMD vere e proprie.

## 19.5 Tipi di IR

- Abstract Syntax Tree (AST)
- Directed Acyclic Graph (DAG)

- Control Flow Graph (CFG)
- Program Dependence Graph (PDG)
- Static Single Assignment form (SSA)
- 3-Address Code (3AC)
- Hybrid combinations

Il compilatore `gcc` ha tre IR: generic (Abstract Syntax Tree), GIMPLE (3AC) poi trasformata in SSA) e RTL.



All'inizio era presente solo l'RTL (Register Transfer Language), IR di basso livello, funziona bene per ottimizzazioni vicine alla macchina. GIMPLE invece rende semplice e modulare la maniera in cui rappresentiamo il programma. Il control flow deve essere il più elementare possibile: il programma viene trasformato in una sequenza di statement (espressioni in 3AC) più delle informazioni di salto.

## 19.6 Ottimizzazione

L'idea è quella di trasformare più e più volte il programma con finalità di *performance* o di *dimensione* o di tradeoff tra le due.

### 19.6.1 Constant Folding

Un esempio di ottimizzazione è la **constant folding**: cerca le espressioni che possono essere determinate a compile time e le sostituisce con il valore numerico/logico.

### 19.6.2 Constant Propagation

Un'altra ottimizzazione diffusa. Propaga i valori costanti attraverso tutto il programma. Se assegno a  $b$  il valore 3 posso sostituire agli usi di  $b$  su tutto il codice.

### 19.6.3 Copy Propagation

Se ho un assegnamento,  $x := y$ , posso eliminare gli usi successivi di  $x$  (solo se  $x$  e  $y$  non vengono modificati successivamente). Tipicamente ho tante definizioni di variabile, quindi serve capire il tempo di vita di una variable, per non fare propagazioni inutili (o sbagliate). Questo è il motivo per cui si introduce la forma SSA.

### 19.6.4 Strength Reduction

Dove possibile rimpiazziamo le moltiplicazioni con degli shift o delle somme. Esempio:  $x * 2$  diventa  $x + x$

### 19.6.5 Dead Code Elimination

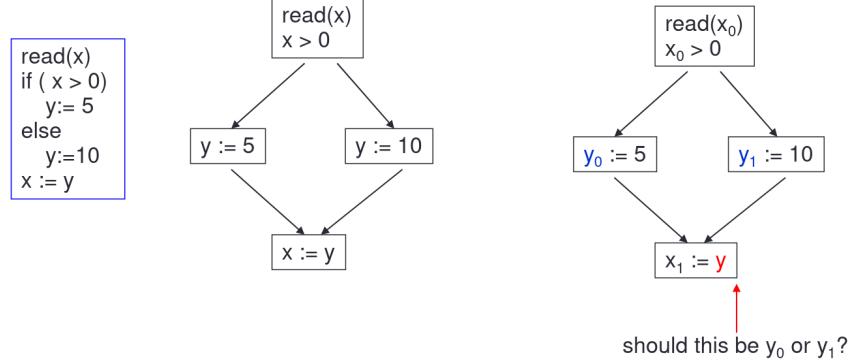
Codice non utilizzato da nessuno va rimosso: propagazioni di valori `false` o variabili non più utilizzate.

## 19.7 SSA

Static Single Assignment form. Il problema delle definizioni e tempi di vita delle variabili in passato veniva risolto con delle complesse analisi di catene di usi-definizioni e studio del tempo di vita. Gli algoritmi utilizzati che consumavano molto tempo e memoria. Si è poi arrivati all'idea delle SSA, che è piuttosto semplice: do un numero (univoco) ad ogni versione di una variabile. Questo semplifica di molto la data flow analysis.



Un primo problema sorge in una situazione del genere:



Questo dipende dal control flow: la forma SSA aggiunge il concetto di nodo  $\varphi$  o funzione  $\varphi$ , statement che metto all'inizio del **basic block** per capire da dove viene la variabile. Il risultato di  $\varphi(y_0, y_1)$  dipenderà dal blocco di provenienza di  $y$ .

