

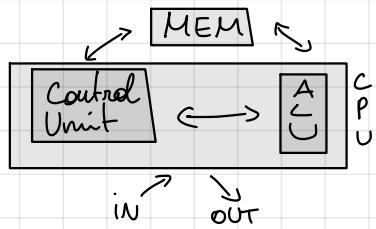
# Lezione 1 - 28/09/20

- hardware.pdf  
L1 e L2:  
• 1-38 intro HPC  
• 84-109 mem e cluster  
• 110-140 cenni soff model

## • Von Neumann Architecture:

Main components:

- memory
- control unit
- ALU
- input / output



## Elements of parallel computing:

- hardware  $\rightarrow$  parallelismo
- system software  $\rightarrow$  os parallelo
- application software  $\rightarrow$  alg paralleli

$\rightarrow$  goals: speedup  $T_p = T_s/p$

NB: concetto di parallelismo  $\rightarrow$  concorrenza di operazioni per ottenere prestazioni migliore, e i loro problemi.  
Le differenze fra gli algoritmi paralleli e sequenziali.

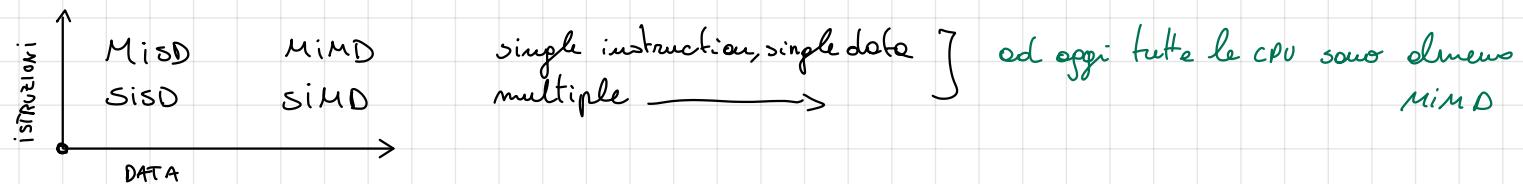
Lo scopo ultimo del parallelismo è lo speedup  $\rightarrow$  + veloce, di quante volte (rispetto a es al sequenziale)  
 $\hookrightarrow$  risolvere problemi che richiedono tanta memoria.

## Divisione organizzazione calcolo parallelo

Dividiamo la visione logica del livello fisico  
quello che vede l'utente hardware, architettura.

Potremo anche programmare a livello più alto, ma le traduzioni potrebbero avere dei problemi:  
conoscere l'architettura del nostro hw per scrivere meglio i programmi.

## Flynn's Taxonomy (1966)



\* immagine Flynn sp ⑦

## Data parallel model → GPUs

La maggior parte del lavoro parallelo si svolge su dei dati set, tipicamente array.

Le task lavorano collettivamente sulle stesse dati, ma in punti differenti.

Le memorie possono essere shared → memoria globale, tutte le task hanno accesso  
distributed → dati spartiti come "chunks" nelle memorie locali  
per ogni task

## Data level parallelism:

Molti struttura dati identiche, però la SIMD ha problemi con lo switch dove ogni unità deve eseguire una operazione diversa sui suoi dati.

**Modello ibrido:** combina più di un tipo dei modelli classici. Un esempio può essere la combinazione fra i CPU attraverso la rete o sulla stessa scheda madre.

Averno 4 cores si può lavorare su più threads che però possono anche dialogare fra loro.

OpenMP e MPI sono due strumenti che useremo. C'è uno scambio di informazioni (P14)!

Sfruttiamo più macchine multicore su un cluster.

SISD →

SIMD →

MISD → Con lo stesso dato fanno operazioni diverse. Non è molto usato e non presta particolarmente bene.

MIMD → Ogni thread può svolgere un'operazione diversa parallelamente, anche su dati diversi. Libertà flusso.

Non è detto che tutti gli accessi alla memoria siano paralleli, possono essere sequenziali, dipende dalle CPU.

Le macchine MIMD raccolgono all'interno esecuzioni SIMD.

## Stili di programmazione

### Single program multiple data (SPMD)

Ci sono diverse strategie per programmare. SPMD è una di quelle → l'algoritmo lavora su un certo sottoinsieme di dati. Ogni programma lavora su un insieme di dati diverso, le varie istanze si devono riconoscere e comunicare.

### Multiple program multiple data (MPMD)

Programma che viene diviso in diverse parti tutte diverse fra loro. Lavorano indipendentemente. Devono comunque essere coordinati. Posso fare decomposizioni.

CPU vs GPU + hardware.pdf

# LEZIONE 2 - 29/03/20

[...]

**Gerarchia di memoria:** SRAM → DRAM → magnetic drive

O.S. 2.5 SO-70  $5 \times 10^6 - 20 \times 10^6$  Acc T.

**Latency and bandwidth:** caratteristiche fondamentali per osservare i programmi hpc.

Quando usciamo dal singolo elaboratore, allora cambiano le prestazioni.

Si può anche distribuire su nodi over the internet → dipende da cose vogliamo fare.

Bisogna sempre (in HPC) controllare la dislocazione dei dati.

**La cache:** Si memorizzano informazioni usate molto frequentemente, risponde più velocemente delle altre memorie (tipo la ram). Si conta hit or miss.

Caratteristiche sono hit rate - hit time - miss penalty

**Parallelismo e gerarchie di memoria: multi-core**

La mem è condivisa ma i due core possono avere ognuno i propri cache → problema: se le "foto" cache sono in momenti diversi, il processore 2 non sa dei cambiamenti fatti da p1.

Possano elaborare dati condivisi che non rispecchiano l'ultima modifica.

↳ cache coherence: bisogna fare update dei dati (non c'è una race condition perché un p2 ha una informazione vecchia).

**El-Dorado of computer design:** avere un super computer semplicemente attaccando insieme tanti appetti più piccoli → non c'è unico come sembra. Si arriva al cluster.

**Multi-core e multi-processore:** i processori interni al chip vengono chiamati cores.

Caratteristiche:

**Job/process level parallelism:** alte velocità per jobs indipendenti e paralleli. (livello+comune parallelismo) può causare bottleneck se x esempio manca abbastanza RAM

**Parallel processing program:** un singolo programma che può opporsi a più risorse/processori simultaneamente.

**Terminologia:**

- **Nodo:** computer che può essere usato singolarmente ed è autosufficiente. Unità minima hardware.

I nodi poi sono connessi alla rete e vanno a formare una macchina di Von Neuman.

- **CPU/socket / processor / core:** la CPU è una, formata da diverse core (chiamati a volte socket)

- **Task:** una sezione logica discreta di un lavoro da fare. Un pezzo di lavoro.

- **Shared memory:** se i task girano su nodi diversi, non ho una mem locale. Un programma.

- **Symmetric Multi-processor (SMP):** è una architettura hardware, diversi processori condividono una memoria unica.

- **Distributed memory:** quando ho un cluster, i miei task vedranno i dati che ho in locale, serve la comunicazione per vedere i dati presenti in rete. Quelli in RAM saranno più economici → si crea una simmetria.

↳ comunicazione: rete che si collega oppure bus che interconnettono.

- **Synchronization:** coordinazione dei task paralleli in tempo reale.

- **Granularità:** misura qualitativa delle attività.

- **Parallel overhead:** tempo richiesto per coordinare tasks paralleli.

**NB:** troppi lavoratori non sono per forza cosa buona.

- **Scalabilità**: mostra la capacità del sistema di dimostrare un aumento proporzionale nelle velocità parallele con l'aggiunta di più processori.

## LEZIONE 3 - 05/10/20 (Laboratorio)

Si prendono in considerazione alcuni semplici programmi seriali  $\rightarrow$  li analizziamo in prospettiva di una eventuale parallelizzazione. Bisogna individuare le parti parallelizzabili e non.

$T_{np}$  = tempo non parallelizzabile       $T_p$  = tempo parallelo       $P$  = # processori

1) **Legge di Amdahl (Speed Up)** =  $T_{serial} / T_{parallel} = (T_{np} + T_p) / (T_{np} + T_p / P)$

2) **Comunicazione**: determinare l'impatto della comunicazione  $\rightarrow$  tempo richiesto per la comunicazione =  $T_{comm}$   $\rightarrow$  somma dei tempi dei messaggi scambiati.

Per ogni messaggio il tempo è  $T_m = \text{Latency} + \text{Message Length} / \text{Bandwidth}$   
 $\hookrightarrow$  è rilevante per il parallelismo (multiprocesso) ma non multithread (memoria condivisa)

3) Altro overhead: tempo di startup di processi/thread, sincronismi, ecc.

Speed Up reale atteso:  $S_{real} = (T_p + T_{np}) / (T_{np} + T_p / P + T_{comm} + T_{startup} + T_{sync})$

ES 1-2-3 didattico - linux

## LEZIONE 4 - 06/10/20

## LEZIONE 5 - 12/10/10 - Risoluzione esercitazione

Perliano dell'esercitazione. La legge di Amdahl ci permette di calcolare lo speedup. Cambia se la memoria è condivisa oppure no.

Sul secondo esercizio il problema di parallelizzazione può avvenire sullo startup e sul completamento.

In che modo si distribuisce il calcolo? = Decomposizione di dominio.

La funzione parallelizzabile è quella di Jacobi.

Correzione del laboratorio con spiegazione.

Punto principale: differenza fra memoria  $\rightarrow$  condivisa  
 $\downarrow$  distribuita

## LEZIONE 6 - 13/10/20

### Le leggi di Amdahl:

Avere un programma parallelo efficiente è difficile per diversi motivi:

- trovare le parti effettivamente parallelizzabili (legge Amdahl)
- Granularità: quanto grande ogni task parallelo deve essere
- Località: dove andiamo a mettere i dati e quanto costa spostarli. Spost  $\gg$  calcolo
- Bilanciamento del carico: non vogliamo un mega processore che aspetta uno lento.
- Coordinazione e sincronizzazione: condividere dati in modo sicuro.
- Performance modeling/debugging/tuning

#### • Multicore needs

- $\hookrightarrow$  reduce communication/sync
- $\hookrightarrow$  load balance
- $\hookrightarrow$  cache coherence

#### • Single core needs

- $\hookrightarrow$  memory hierarchy
- $\hookrightarrow$  instruction synchronization
- $\hookrightarrow$  I/O: redundant arrays of inexpensive disks

Slide: scheduling, load balancing, time for synchronisation, overhead for comms between ports.

La legge di Amdahl: exec time after improvement = 
$$\frac{\text{exec time affected by improvement}}{\text{Amount of improvement}} + \frac{\text{exec time unaffected}}{\text{Amount of improvement}}$$

speed up = 
$$\frac{\text{exec time before}}{(\text{Et}_b - \text{exec time affected}) + \frac{\text{exec time affected}}{100 \times \# processors}}$$

**Limi<sup>t</sup>i e costi:** bisogna capire quale può essere il costo in proporzione allo speed up ottenibile.

Bisogna cercare sempre di raggiungere lo speedup teorico massimo prima di farlo.

Se buona parte del codice non è parallelizzabile allora bisogna lavorare su quello piuttosto che altro.

Per esempio, lo speed up max teorico non tiene conto di quello che può essere il ritardo di rete.

NB: su grandi cluster possiamo avere migliaia di processori, ma per avere speed up di migliaia di volte bisogna avere un codice che lo permette → profiling!

## Performance scaling:

**Strong scaling:** speed-up on multiprocessors without increase on problem size.

**Weak scaling:** speed-up on multiprocessors, while increasing the size of the problem proportionally to the increase in the number of processors.

||

Come il programma si comporta quando risolvo il problema aumentando i processori.

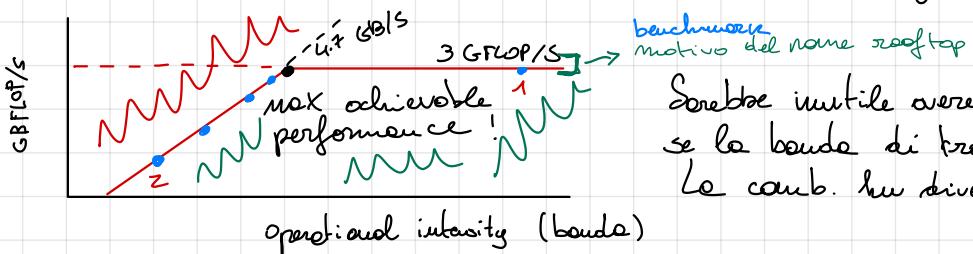
Di norma c'è un max teorico. Parliamo di durabilità.

Il weak scaling è ottimo per trarre eventuali problemi di comunicazione.

P75-76 MG

## The Roofline Model:

Questo modello mette in relazione le potenze di calcolo e le capacità di trasferire i dati. Da qui possiamo capire le intensità operazionale [flop/byte] → cose uso di più fra CPU e trasferimento dati? Trovo così i limiti teorici di alg e bw e bottleneck.



Sarebbe inutile avere più potenza di calcolo se la banda di trasferimento non basta.

La comb. bw diventa limitante

1 Se tutti i miei benchmarks vanno tutti a destra, ho raggiunto il max che la CPU mi dà.

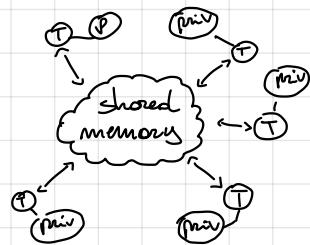
2 Se i benchmarks vanno tutti (molto) a sx, allora la banda è il bottleneck.

# LEZIONE 7 - 19/10/20

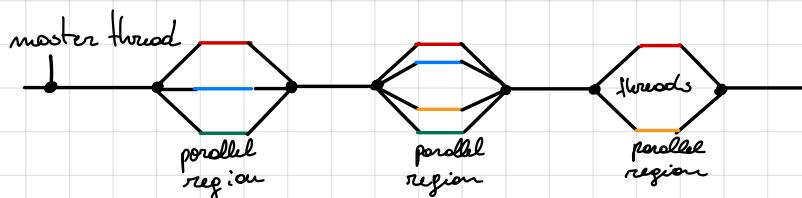
- Cyberchallenge.com → bisogna vedere se l'uni si candida.
- Open MP per parallelizzazione → cluster linux → cluster vero.
- Controllo e "correzione" per esecuzione 10
- Informazioni + spiegazioni per esercitazione 16.

\* L'inizializzazione e la finalizzazione di un problema normalmente è parallelizzabile, ma noi ci concentriamo sulla parte di calcolo e non su quelle due parti.  
 \* Un ciclo for (con diversi for) indipendenti l'uno dall'altro sono sempre parallelizzabili.

- Open MP: librerie che ci consentono di implementare la memoria condivisa in Fortran, C, C++.



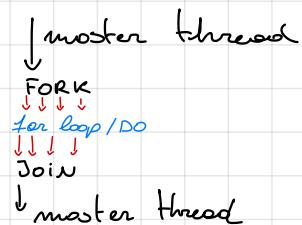
Fork-Join Model:



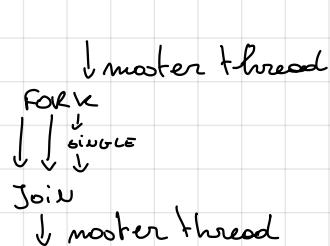
N.B. In OpenMP es. c una variabile  $j$  privata non è protetta → ci sarà race condition.  
 Proteggere le variabili che possono essere modificate durante una sezione critica è importante.

- For directive:

- requires a for loop
- makes the loop index private to each other
- runs a subset of operation in each thread



- Single and master directives: the single directive specifies that the enclosed code is to be executed by only one threads in the team.



Il programma overhead vuole creare un overhead per il programma dove il programma attende un tempo pari al tempo di sleep. (Tempo reale - tempo sleep) / n. iter × 1M → tempo iterazione

## LEZIONE 8 - 20/10/20

Slide di riposo di quello che è stato fatto: matini → parallelizzazione + supercomputer.

### Gerarchie di memoria:

Le cache sono piccole pool di memoria che contengono le info più usate dal computer.

Le cache sono:  
1) delle più piccole e veloci  
2) delle più grandi e lente

### Memoria condivisa:

Nei computer con memoria condivisa tutti i processori vedono lo stesso indirizzo globale.

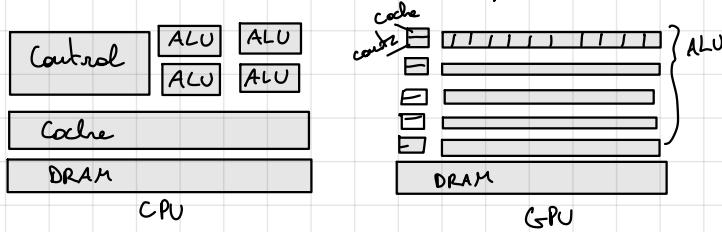
Nascono problemi di coerenza → serve sincronizzazione + protocolli di accesso alle memorie, ma si riesce a leggere subito tutti i dati senza comunicazioni.

→ Non è scalabile: + processori + conflitti in accesso = bottleneck in memoria.

Di norma la mem. condivisa viene usata con un max. di processori.

- SMP - symmetric multi processor // multi core processor //

+ GPU:



### Memoria distribuita:

I dati sono sullo stesso network ma non sulle stesse macchine. Servirà scambiare messaggi in rete e ormai OpenMPI è diventato uno standard. Si riesce a crescere per ogni CPU aggiunta, senza overhead e bottleneck in memoria e il livello economico è molto conveniente.

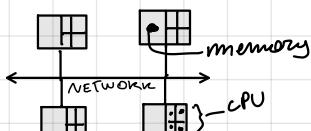
Pero il programmatore è responsabile per la comunicazione fra le CPU e di tutti i loro problemi.

Può essere difficile creare un sistema uniformemente distribuito → i tempi di accesso alle memorie potrebbero non essere uniformi, tutti gli accessi esterni costano più di quelli locali.

Potrei però usare hardware "classico".

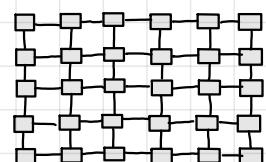
### Approccio misto:

Nei cluster ad alte prestazioni, quello che funziona comodamente è un insieme di nodi in rete che però sono creati da nodi locali che usano memoria condivisa.

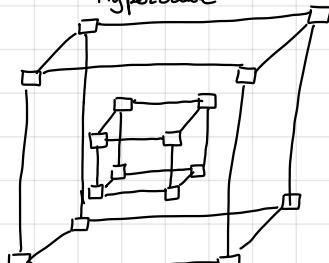


NB: la topologia della rete ha forte impatto sulla rete stessa e livello di banda e latenza.

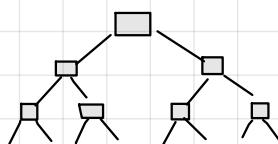
### 2-dimensional - network



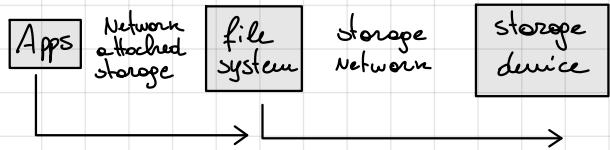
### Hypercube



### Fat tree



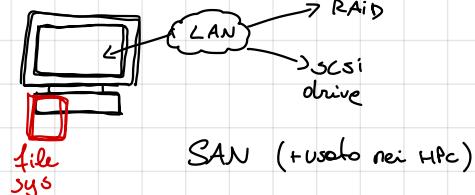
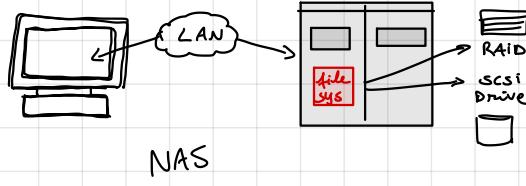
## Storage Area Network



Possibilità di memorizzare dati in modo permanente nel network del nostro cluster.  
Due metodi sono possibili: **NAS** Network attached storage  
**SAN** Storage area Network

Entrambi possono rendere le info sulla rete disponibili all'utente, ma le configurazioni sono diverse.  
A seconda di dove si trova la rete fra i 3 oggetti, varie

- 1) Rete fra utente e file system = **NAS** → utente chiede tramite protocolli di rete un file
- 2) Rete fra file system e storage device = **SAN** → utente separa file sys e mem

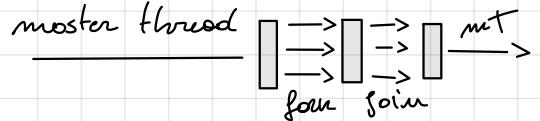


## LEZIONE 9 - 26/10/20

**RSA:** per il cluster hpc → usiamo le coppie di chiavi per eseguire l'accesso senza psw.  
Ci permette di passare fra vari nodi senza autenticarsi tutte le volte.

Accesso al cluster → login ecc.

**OpenMP:** usando "section" generiamo le sezioni parallele del nostro programma e poi viene definito cosa vogliamo che la parte del programma faccia.  
\* Single and master directives =>

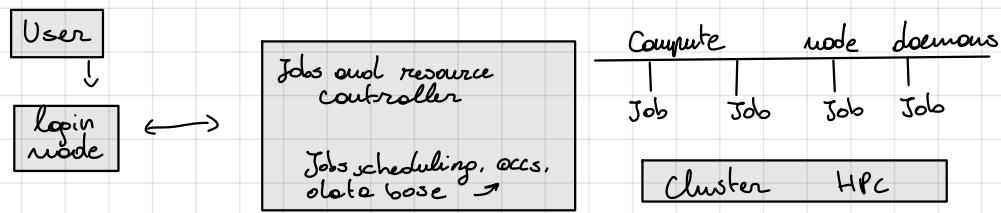


// T serial - speedup// concetti

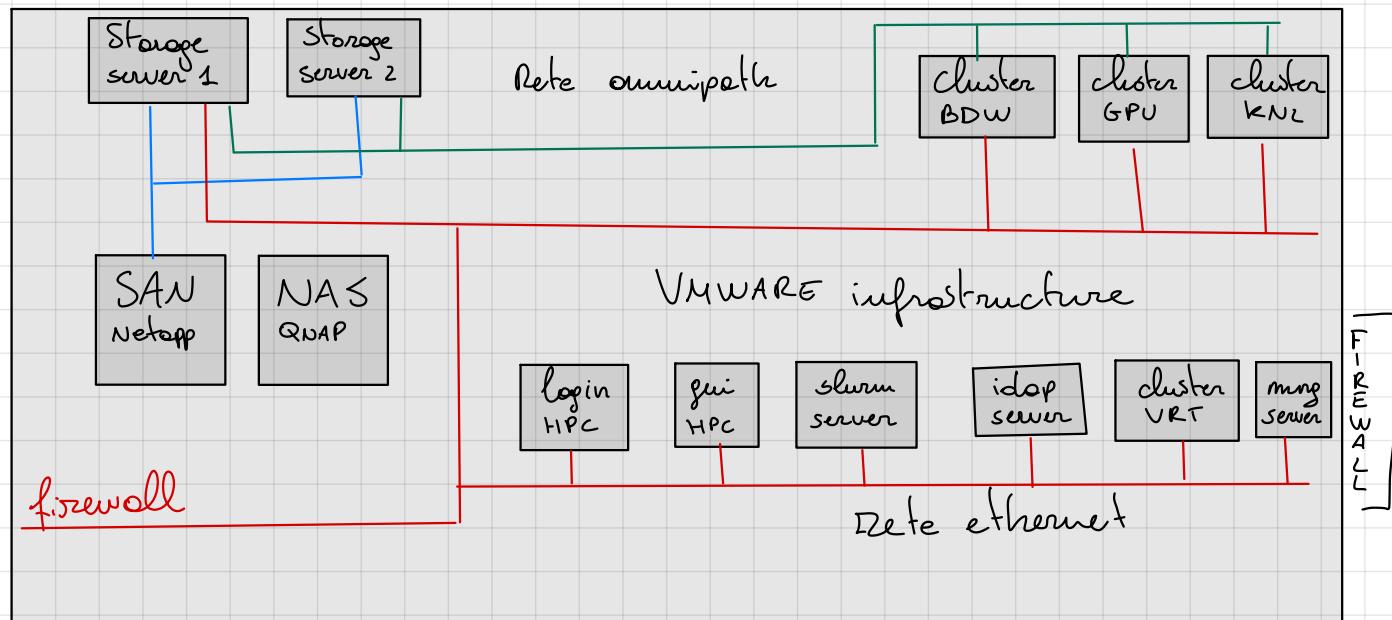
Operazioni sul cluster + esercitazioni:

# LEZIONE 10 - 27/10/20

**Slurm and moab:** Si parla di workload manager  $\rightarrow$  jobs e script che descrivono al manager cosa fare. Slurm è il workload manager, che è quello che usiamo noi.



Struttura cluster HPC:



Operazioni sul cluster + esercitazioni

22/10/20 Breve riassunto di cose importanti viste finora :

Legge di Amdahl: Serve per calcolare lo speedup teorico di un sistema quando viene migliorato nel sistema. Con  $F$  frazione di calcolo parallelizzabile,  $(1-F)$  non parallelizz. ed  $N$  il numero di CPU

$$\text{speedup} = \frac{1}{(1-F) + F/N}$$

Flynn's taxonomy:

	Single Instruction	Multiple Instruction
Single Data	SISD	MiSD
Mult Data	SiMD	MiMD

Overhead:

E' la quantita' di tempo necessarie a coordinare le sessioni di lavoro parallele, tiene conto di:

- tempo di startup/avvio sessione
- Sincronizzazioni
- Scambio di dati
- Overhead del software
- tempo di chiusura di una sessione

Strong scaling e Weak scaling:

# LEZIONE 12 - 9/11/20

## • OPEN MPI ::

### Distributed memory:

It requires a communication network for the information exchange

Each processor has its own local memory  $\rightarrow$  each memory has a separate independent address space

R/W operations are local  $\rightarrow$  no cache coherence problems

Model is called message passing  $\rightarrow$  each task accesses only its local memory and communicates with remote tasks to access remote data

### What's MPI?

$\Rightarrow$  Message passing library specification for parallel computers, clusters, heterogeneous networks

### OPEN MPI

Compared with MPICH, it's more efficient and portable

### Hostfiles:

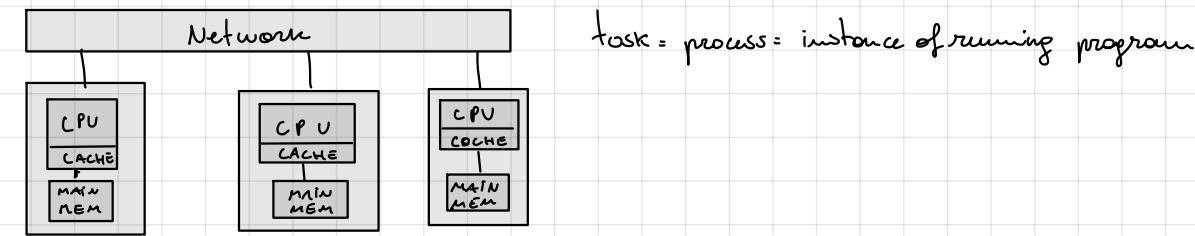
they're simply text files with hosts specified, one per line. Each host can also specify a default max number of slots to be used on that host.

example: # ...

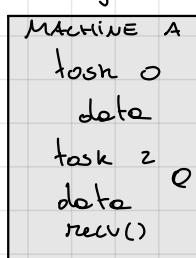
# ...  
host.example.com

### Message passing with MPI:

Here we exchange informations explicitly since we don't have shared memory



Cooperation among processes is based on explicit communications. A sender process and a receiver process exchange messages



Each process is an instance of running sub-program  
Usually, the same sub program is executed over different data sets.  $\Rightarrow$  SPMD (single prog., mult data)  
Each process is identified by a rank from 0-n-1  
MPI-D can be emulated with SPMD

## Messages

two main parts

Envelope: source: sender rank dest: recv rank  
tag: ID of message (0 to MPI\_TAG\_UB) communicator: context of communication

Body: type: MPI datatype length: number of elems  
buffer: array of elems

## Point-to-point communication:

Most simple type of communication, involves only 2 processes.

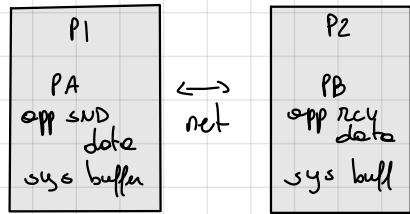
It can be → synchronous or asynchronous

↳ waits for feedback

↳ free to do other stuff in the meanwhile

## Buffering:

Receiver would want to save incoming data in a buffer (MPI own) and then move them into RAM.  
We can't do asynchronous without a buffer for example.



the sys buffer:

- may exists on both sides: SND / RCV
- is usually limited
- unpredictable
- cannot be controlled by programmer

MPI: also offers for a user managed transfer buffer

## Operation modes:

Some operations may cause the blocking of the caller - all of them can be

↳ nonblocking ⇒ process continues after the call

nonblock + wait = blocks

↳ may test or wait for remote process completion

- synchronous send → doesn't need a buffer
- asynchronous send → buffered send  
↳ standard send
- ready send    - recv    - sendrecv

} blocking

↳ non blocking

- Some primitives, sender never blocks → necessary check on buffer reusability on completed communication.

## Collective operations:

Usually involves more than 2 processes:

- Barrier (for synchronisation) => Data movement (collective communication) => Reduction (collective computation)  
Processors all have to wait each other
- Broadcast - scatter - Gather
- Min/max - Sum - logical AND, OR...  
+ User defined

## Functions in MPI:

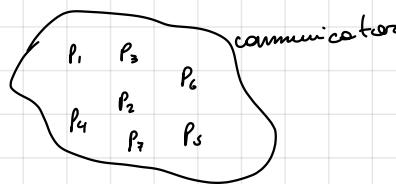
error = MPI\_xxxx (parameter, ...); MPI\_ is a reserved namespace

#include <mpi.h> → only one header

### Communicators:

It's a set of processes that communicate with each other:

- has a name
- has a size
- each process is univocally identified
- processes are equal



NB: two processes can communicate if they belong to the same communicator

↳ default is MPI\_COMM\_WORLD having rank 0...N-1

### Rank size:

to know its rank a process calls MPI\_Comm\_rank (MPI\_Comm comm, int \*rank)

to know communicator size calls MPI\_Comm\_size (MPI\_Comm comm, int \*size)

### Init and exit

int MPI\_Init (int \*argc, char \*\*argv) → int MPI\_Finalize()

p.34 → example on MPI(4).pdf → hello world

### Point-to-point communications:

int MPI\_Send (void \*buf, → pointer to msg to be sent  
int count → number of elements in the msg  
MPI\_Datatype datatype → type of elems  
int dest, → rank of the recipient  
int tag, → positive int, user purpose  
MPI\_Comm comm ) → communicator

NB: Wildcording: it's possible to leave the sender unspecified (in Recv) by means of MPI\_ANY\_SOURCE  
It's also possible to leave the tag unspecified  
No wildcard for communicator

MPI\_Recv (void \*buf, → pointer to array where msg is saved  
int count → "some"  
MPI\_Datatype datatype → "some"  
int source, → rank of sender  
int tag, → only certain tags are for reception  
MPI\_Comm comm, → "some"  
MPI\_Status status) → info about envelope of msg

NB: the envelope of a message is given by MPI\_Status struct, that can be retrieved from the status variable.

NB: to get the # of elements with a specific datatype in the received message:  
int MPI\_Get\_Count (MPI\_Status status, datatype)

NB: the order of the messages sent and received is preserved  
starvation and deadlock are still a problem to manage

### Measuring execution time:

Standard timers aren't adequate for MPI.

The execution time of a task is measured by getting the current time before and after the execution

### Completed communications:

A communication is locally completed on a process if the latter has completed its part of operations related to communication. → can execute what's after send or recv

A communication is globally completed if all involved processes have completed their operations related to communications.

must finish copying from buffer before overriding.

## Completed communications

The completion phase of SEND depends on the message size:

- buffered for small size.
- synchronous for large size.

If we do blocking sends we need to be sure the process can actually continue to the recv and not end up in a deadlock.

Doing a non blocking send is better.

## Communication completion Criteria:

When are they completed?

- Sync send

→ when the application buffer can be reused and the message reception has started

- Buffered send

→ when the message has been completely copied to the transfer buffer.

- Standard send

→ when the application buffer can be reused

- Receive

→ completed when the message has arrived

Blocking and non blocking calls: P. 55-57

LEZIONE 13 10/11/20

MPI RUN esercizi e esercitazione

LEZIONE 14 16/11/20

Primitive MPI / mpi\_bose => slide mpi helloworld

LEZIONE 15 17/11/20

- Chiomate std-MPI:

- blocking calls

- synchronous send → MPI\_Ssend (P.60)
- buffered send → MPI\_Bsend (P.61)
- standard send → MPI\_Send
- ready send → MPI\_Rsend
- receive → MPI\_Recv

- non blocking

- synchronous non blocking send
- buffered non blocking
- standard non blocking send
- ready non blocking send
- non blocking receive

} MPI\_1 [...]  
} 1 vs 5  
for non  
blocking

- Application buffer: ricordare il concetto [...]

Nelle **sync** send non bisogna definire esplicitamente un buffer ed il mittente è in sync con il ricevente.

Nelle **buff send** bisogna invece explicitare il buffer per poi farne un detach.

Nelle **std** terms il controllo dopo che il messaggio è stato mandato (e ricevuto)

Nelle **ready** terms il controllo dopo che B chiama la receive → può avere un comportamento pericoloso

**• MPI - Sendrecv** → send e recv sono fatti insieme. È bloccante, e entrambe le chiamate usano lo stesso communicator. Bisogna avere due buffer differenti. Send e recv possono avere tag diversi.

### • Nonblocking comms

3 phases: beginning of send/recv → execution of an activity that doesn't involve the comm → wait for completion ⇒ posso overlap communication phases with computing phases and reduce latency. + we avoid deadlock.  
cons: hard to program.

### • Synchronous blocking:

Issend → Wait ↗ between the two calls I can execute code  
↗ bring up code before the wait (which is blocking)  
↗ until wait is complete, buffer isn't loaded.  
↳ Issend + wait = Jsend

### • std nonblocking send:

initialized send → identify buffer → request handle

### • Completion test ⇒ wait makes a non blocking → blocking + wait returns a bool

• Deadlocks: Anche con le send e recv si possono creare dei deadlock

- MPI\_Ssend(..., right.rank,...)
- MPI\_Recv(..., left.rank,...)

↳ Ssend doesn't complete until corresponding Recv has been started, but since all processes are blocked on Ssend, no one will Recv

Avoid deadlocks ↗ 1 change calls order  
↗ 2 use non blocking  
↗ 3 use explicit buffer

### P71 - memorize + example 72/73

### Collective Communicator:

(non and  
blocking collective 100-108)

- all process must communicate, blocking and non blocking, no tags, recv buff size == tag  
3 class; ↗ all to one  
    ↗ one to all  
    ↗ all to all

• Barrier synch → It's a barrier that makes sure all processes reach the same point in time

• Broadcast → one to all: one rank sends to the other ranks. After a broadcast all ranks will have the same value.

• Scatter → break the send buffer and send pieces of it to various ranks. Every rank needs a buffer and even root will recv some.

• Gather → the opposite. In both cases buffers have to be precisely the right size.

• Reduction: used to perform computations that involve distributed data within a group of processes. ⇒ Recv a lot and put it together → similar to gather + math

# LEZIONE 16 23/11/20 - LE GPU

Ci concentriamo subito sulle GPU Nvidia che utilizzano CUDA come architettura.

CUDA arch.  $\xrightarrow{\text{expose GPU computing}}$  retain performance  $\xrightarrow{\text{GPU}}$   
Nose per rappresentare dei punti sullo schermo  
possono dei pixel  $\rightarrow 2k \rightarrow 4k$  ecc.

CUDA C/C++  $\Rightarrow$  industry standard

## Terminologia:

host  $\rightarrow$  the CPU and RAM (host memory)  
device  $\rightarrow$  the GPU and its memory (device memory)

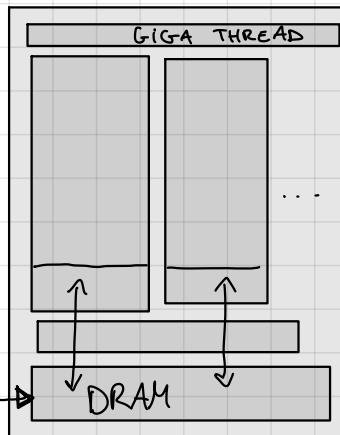
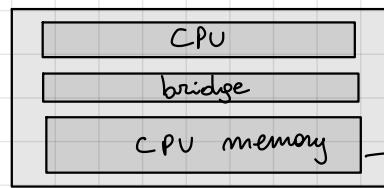
Ad oggi già appaiono ibridi che sono CPU+GPU insieme, in futuro potrebbero anche avere senso.

## Heterogeneous computing

device code { | codice | } parallel func  
host code { | codice | } serial code  
- parallel code  
- serial code

tipicamente abbiamo fasi alternate di utilizzo CPU e GPU; ma ovviamente si può anche fare utilizzo parallelo delle due.

## Simple processing flow



1. Copy input data from CPU mem to the GPU memory
2. Load GPU code and execute it, caching data on chip for performance.
3. Copy results from GPU mem to CPU mem for more elaborations

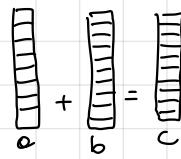
## Hello world!

```
int main(void)
    printf("hello world!\n");
    return 0;
}
-- global --
void mykernel(void) {
    int main(void) {
        mykernel<<<1,1>>>();
        printf(...);
        return 0;
    }
}
```

Standard C runs on host, NVIDIA compiler (nvcc) can be used to compile programs with no device code.

- CUDA C/C++ uses `--global--` to indicate that that function will only run on the device but will be called by the host.  $\rightarrow$  kw
- nvcc separates source code into host and device components
- `<<<1,1>>>` triple angled brackets mark a call from host code to device code. They indicate the parallelism level we want to use.

# Parallel programming in C/C++ CUDA



Example on how parallelism is useful  $\rightarrow$  sum of two vectors instead of using a for cycle we can use many threads

NB: We use pointers for the variables  $\rightarrow$  we need to allocate memory on the GPU.  
We use pointers because what's on the CPU wouldn't make sense on the GPU otherwise.

\* Host and device memories are separate entities!

Device  $\rightarrow$  pointers point to GPU mem.

$\Downarrow$  may be passed to/from host code

$\Downarrow$  may not be dereferenced in host code

host  $\rightarrow$  pointers points to CPU memory

$\Downarrow$  may be passed to/from device code

$\Downarrow$  may not be dereferenced in device code

LEZIONE 17 24/11/20

Recap  $\rightarrow$  2 memorie distinte  $\xrightarrow{\text{CPU}}$   $\xrightarrow{\text{GPU}}$

chiamate  $\rightarrow$  esplicite  $\xrightarrow{\text{cuda\_malloc}}$   $\xrightarrow{\text{cuda\_free}}$

GPU dati  $\xrightarrow{\text{CPU}}$  scambio e chiamate

Moving to parallel

Different level of parallelism are possible  $\Rightarrow$  odd  $\ll<1, 1>>$  ()  
how many times the code is run in parallel

block: a work block that can be launched many times

$\rightarrow$  blocks can differ slightly in what they do

grid: the group of blocks the GPU is working on

$\Downarrow$  on the device each block is executed in parallel  
 $\Downarrow$  each block controls a different part of the array

Threads:

A block can be split into parallel threads (double subdivision)  $\rightarrow$  many blocks have many threads  
We can even use 1 block instead of many, but many threads inside the block.

block  $\ll<1, N>>$  # of threads per block  $\rightsquigarrow$  idea: combine block and threads together.

Why many threads and double subdivision?  $\Rightarrow$  threads can do stuff (sync and cooms) that blocks cannot.

• Combine blocks and threads

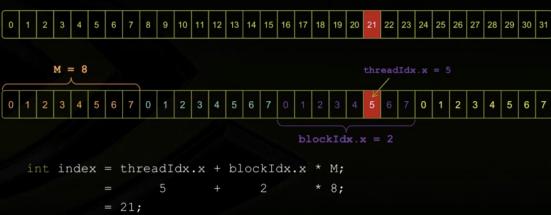
Consider indexing an array with one element per thread (8 threads/block)
threadIdx.x      threadIdx.x      threadIdx.x      threadIdx.x 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 blockIdx.x = 0    blockIdx.x = 1    blockIdx.x = 2    blockIdx.x = 3

- With M threads per block, a unique index for each thread is given by int index = threadIdx.x + blockIdx.x \* M;

M represents the block dimension in terms of thread number.

block/thread/index

- Which thread will operate on the red element?



Example: linearizzazione semplice  $\rightarrow$  ogni thread ha un indice univoco.

For threads per block  $\hookrightarrow$

$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x};$

$N/\text{threads\_per\_block} \Rightarrow$  l'array va suddiviso.

## Handling arbitrary vector size

the typical problems aren't friendly multiples of  $\text{blockDim.x}$

$\hookrightarrow$  avoid accessing beyond the end of the arrays  $\rightarrow$  update kernel launch

## Threads pros:

add a level of complexity  
 threads  $\hookrightarrow$  don't seem to be a gain } but threads  $\hookrightarrow$  communicate } efficiently  
 $\hookrightarrow$  synchronize }

## Implementing within a block

Each thread processes one output element

$\hookrightarrow$  input elements are read several times  $\Rightarrow$  waste of access time

$\hookrightarrow$  SOL: sharing data between threads

\* Within a block, threads share data via shared memory

By opposition to device memory, we have a global memory  
 (data isn't visible to threads in other block)

• Cache data into shared memory  $\rightarrow$  read  $\rightarrow$  compute  $\rightarrow$  write

$\hookrightarrow$  read from global  $\rightarrow$  cache or shared  $\rightarrow$  great advantage in access time.

= when i need to read the same data many times, it's better to cache it.

Shared memory is used for those times where we need to read the same data many times and then to make the final global write.

Synchthreads: sync all threads within a block, all threads must reach their barrier

## LEZIONE 18 - 30/11/20

### Coordinating host and devices

• Kernel launches are synchronous  $\rightarrow$  controls return to CPU immediately  $\rightarrow$  CPU can do more  
 $\hookrightarrow$  CPU needs to sync before consuming the results.

$\Downarrow$   $\text{cudaMemcpy}()$

Chiaviere blocanti  $\hookrightarrow$   $\text{cudaMemcpyAsync}()$

$\hookrightarrow$   $\text{cudaDeviceSynchronize}()$

All cuda API calls return an error  $\rightarrow$  can always see what went wrong in case of need.

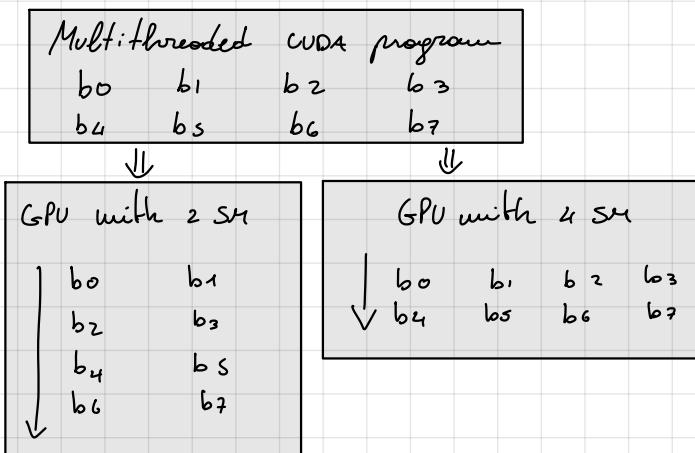
- Applications can query and select GPU
- Multiple host threads can share a device  $\rightarrow$  1 GPU can have many kernels
- A single host thread can manage multiple devices



Le GPU privilegiano la velocità di trasferimento dei blocchi di dati  $\rightarrow$  latenza più alta.

- \* Non ho bisogno di una cache grande  $\rightarrow$  mentre faccio tutti i calcoli, ho il tempo per ricevere i dati successivi e latenza alta.
- \* Quando si ragiona in dati paralleli c'è bisogno di fare un mapping fra i thread e gli elementi. È una scelta libera che influenza fortemente le prestazioni del nostro programma.
- \* Indipendentemente dalla nostra architettura, possiamo avere un programma scalabile  $\rightarrow$  aumentano i core come aumenta la scalabilità.
  - gerarchia di thread
- \* astrazioni chiave  $\hookrightarrow$  memoria condivisa  $\Rightarrow$  Ogni blocco ha una vita indipendente
  - $\hookrightarrow$  barriera di sincronizzazione

Ogni blocco di thread può essere schedulato su un processore.  
I blocchi possono essere compilati sui processori senza toccare il sorgente.



SM = streaming multiprocessor (core sim)

Tanti blocchi  $\rightarrow$  coda sempre piena  $\rightarrow$  scalabilità max  $\rightarrow$  efficienza garantita.

## Gerarchie di thread:

I Thread possono essere indicizzati su una, due o tre dimensioni di thread-index. Forma quindi blocchi di thread 1d - 2d - 3d. Questo forma ci permette di invocare naturalmente gli elementi come in vettori o matrici.

Limiti  $\rightarrow$  c'è sempre un max thread # per blocco (1024)
 

- $\hookrightarrow$  non significa che ci sono 1024 ALU e pieno parallelismo fra tutti i thread.

Blocchi  $\rightarrow$  organizzati in griglie dimensionali  $\rightarrow$  thread indicizzati nel blocco.

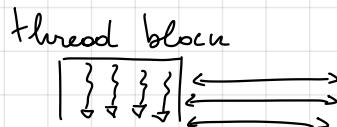
- $\hookrightarrow$  indicizzati in modo potenzialmente diverso dai thread contenuti.
- $\hookrightarrow$  non hanno vincoli fra loro  $\rightarrow$  ordine di esecuzione deve essere variabile
- $\hookrightarrow$  altrimenti si schedule  $\rightarrow$  entiere race condition.

NB: Ogni blocco deve sempre leggere e scrivere in zone di memoria diverse → assegnare a thread zone di mem distinte  
Alt rimanenti schedulare e sincronizzare

NB: Esiste una mem read-only (deriva da lettura grafica 3D)

Memoria:

{ Thread ← → per thread local mem



(Jump cap 4)

Implementazione hardware / Single (warp)

Un multi-processore esegue centinaia di threads insieme. Per farlo si usa una architettura **SIMT** → invece di lanciare ogni thread come "a sé stante" (MIMD); li lanciamo in modo diverso → il multi-processore può lanciare gruppi di 32 threads chiamati **WARP**: le cose è trasparente.

↳ iniziano insieme con lo stesso program counter.

→ 1 sole in SIMT

Ogni singolo thread può comunque lavorare con il suo program counter → branching.  
↳ ma ogni thread può fare cose diverse degli altri.

Solo gli stessi thread che eseguono le stesse istruzioni, possono lavorare in parallelo.  
Nel flusso del programma avrò sottothread che si differenziano e vengono lanciati successivamente in gruppo.

blocco → warp (32) → thread (1024)

↳ scheduler del warp

Ogni warp contiene thread consecutivi rispetto all' id del thread → se sempre quale warp contiene un thread.

Il warp è la vera unità che sta lavorando in parallelo sulla GPU.

Quando c'è una divergenza di codice lo scheduler esegue prima una parte del warp e poi la seconda su cicli di clock diversi.

████████ } l'esecuzione redoppia, a botte di 16. Un SIMD avrebbe potuto avere multiple ALU.

Quindi → una divergenza del codice → coste prestazioni  
↳ non c'è che sia geniale  
↳ facciamo fare ai warp le stesse cose.

SIMD e SIMD sono simili, ma SIMD ha lo SWAG

# LEZIONE 19 - 11/12/20

Riferimenti: l19 pdf - laboratorio hpc - progr GPU

# LEZIONE 20 - 11/12/20

R = 9.1/9.1.1/9.1.2/9.1.4/9.2.1/9.2.2

GPU → alte velocità di calcolo → solo se buon utilizzo delle memorie interne.

## Host-device data transfer

GPU → buono a trasferire dati all'interno di sé stessa → bottleneck in RAM transfer.  
→ trasferire poco se di frequente o in modo completo.

High priority: minimize data transfer between host device

\* Pinned memory: page locked sulla ram, ottimizza i trasferimenti host  $\Rightarrow$  device ma ve vuole poco.

\* Esiste la versione asincrona delle `cudaMemcpy()` non bloccante  $\rightarrow$  lavorano dopo aver schedulato la copia  $\rightarrow$  si può usare solo se usiamo la pinned memory prima  $\rightarrow$  tiene in RAM le rdss.  
- Stream ID  $\rightarrow$  possono viaggiare in parallelo.  $\hookrightarrow$  copia asincrona  $\rightarrow$  sviluppo e attivita'.

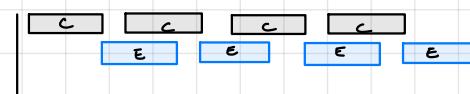
NB: se lavoro sugli stessi dati uno noce condition, devo quindi sincronizzare lo stesso stream oppure usare due stream di dati diversi.

\* Se invece definisco due stream diversi e faccio `cudaMemcpy()` su uno e kernel sull'altro allora i due stream possono proseguire in parallelo.

sequential



concurrent



] solo con sync e buffer  
adeguato

(No zero-copy)

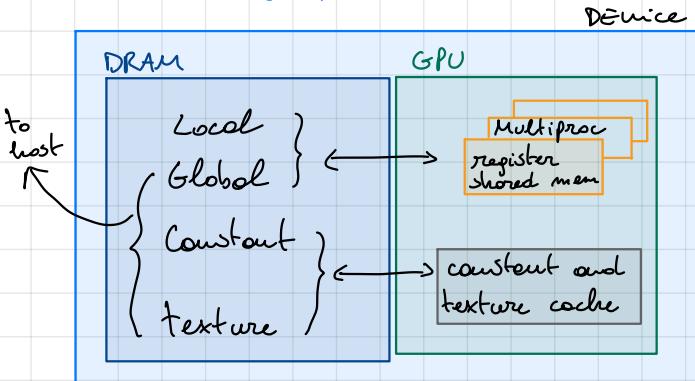
## Unified Virtual Addressing:

Possiamo usare un meccanismo che nasconde la allocazione esplicita delle memorie.

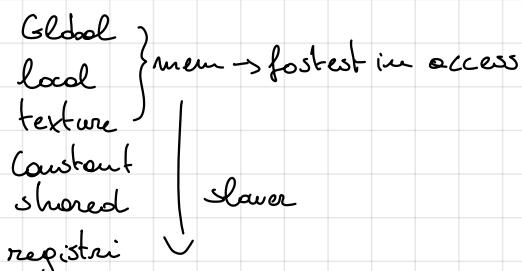
Normalmente dobbiamo `malloc()`  $\rightarrow$  utilizzo e traccia puntatore / `cudaMalloc()`  $\rightarrow$  `cudaPointargetAttributes()`.

Può essere + efficiente il controllo completo  $\rightarrow$  p2p data transf  $\rightarrow$  usiamo questo meccanismo

## Device memory spaces



Sui vari blocchi di memoria, la memoria globale è la più grande.



\* Spilling: succede quando si sovrappone la memoria che stiamo usando e i dati: vanno su una memoria, con costi di accessi più alti.

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

† Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.  
 † Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.

### texture:

Le GPU usano per creare immagini sullo schermo. Texture raccolgono i colori da disegnare (R) ai colori giusti sui nostri cubi o triangoli.

} GPU  
CPU

La presenza di queste piccole cache migliora le prestazioni in caso di spilling. (?)!!!

### Coalesced Access to global memory

È importantissima la posizione dei dati sulla memoria globale quando vogliamo accedervi.

Coalesced → le richieste sono organizzate bene e ogni warp ottiene transazioni a blocco.

Accessi di memoria organizzati bene sono fondamentali per come il programma funziona.

Se tutti gli accessi (del warp) passano dalla 21 → 128 byte possono essere trasferiti in blocco. Quindi farlo continuamente e blocchi di 128 è molto efficiente.

NB: Vanno evitati accessi sposi:

### 9.2.1.1 9.2.1.2 → esempi grafici + 9.2.1.4 Strided access

Nella progettazione dati conviene sempre progettare strutture di array invece che array di strutture

### Shared memory:

Molto veloce, risposte in tempi rapidi.

Viene di solito suddivisa in blocchi uguali su cui si può fare R/W parallelemente → dipende dalla organizzazione → 2 op su stesso blocco → perdo velocità.

Altrimenti se tutti i thread del warp leggono lo stesso blocco → broadcast → very fast

Anche nella memoria shared i dati vanno organizzati in base al numero di thread.

Cambio pdf → code by example c7 → heat eq.

### Texture and simple heating model:

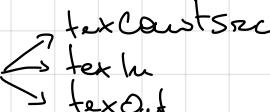
- Le texture possono essere usate anche per effettuare accessi sposi in memoria
- Colore: possiamo vedere le temp delle celle adiacenti alla heater seguendo una formula.

### Simulazione del colore:

Usiamo una matrice per simulare le celle → con 1° kernel settiamo le temp di certe celle →

→ 2° kernel per applicare stencil → 2 buffer → 1 lettura  
→ 1 scrittura

Si usano 3 texture → collegate con l'area di memoria (binding)



↳ GPU può accedere ai dati usando fetch solo × texture.

# LEZIONE 21 - 14/12/20

[LAB - heat-gpu on cluster]

# LEZIONE 22 - 15/12/20

Occupancy: consumo dello GPU rispetto al nostro codice e al nostro hardware.  
↳ calculator → Nvidia made spreadsheet calculating it with custom input.

Nelle potenze massime di calcolo teorica si arriva allo occupancy, ma possono esserci dei limiti

Utilizzo di molti GPU → maggiore scalabilità → maggiore potenza (se GPU non basta)

## Multi-GPU programming (with MPI)

Uniamo il parallelismo di calcoli paralleli su molti GPU schede video ⇒ maggiore performance

- GPU's do not share global memory. Although, we can share stuff if using some bus.  
It won't happen usually, one node is made by 1 CPU + 1 GPU. A node could be 1CPU + 7GPU (unipr)
- Application code is responsible to move data between the GPUs

GPUs → consecutive IDs starting with 0 → a host thread can maintain more than one GPU context at a time → we can choose which one to work with.

- `CudaSetDevice` allows us to choose which. → no `call = device 0`.

Multiple hosts can also establish contexts with the same GPU.

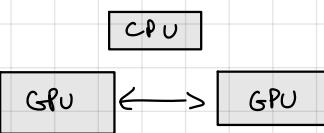
## CUDA Features:

- Control multiple GPUs with a single CPU threads.

- Streams: enable executing

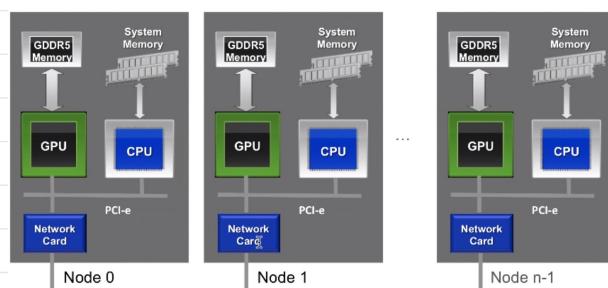
[...]

## P2P GPU communication:



## Inter communication with MPI:

### MPI+CUDA



- Con MPI possiamo trasferire dati fra 2 GPU con le send etc.
- MPI lavora però su RAM CPU → bisogna preparare gli scambi nel codice della applicazione → MPI run sono i launcher

- Idea: dividere il dominio in parti più piccole per proseguire con i calcoli.

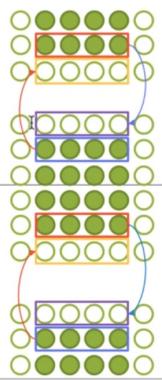
↳ parallelismo: griglia suddiviso su GP  
thread → blocco → GPU

## EXAMPLE: JACOBI TOP/BOTTOM HALO UPDATE

Plain C on CPU

```
MPI_Sendrecv(Tnew+offset first row, m-2, MPI_DOUBLE, t_nb, 0,
Tnew+offset bottom boundary, m-2, MPI_DOUBLE, b_nb, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(Tnew+offset last row, m-2, MPI_DOUBLE, b_nb, 1,
Tnew+offset top boundary, m-2, MPI_DOUBLE, t_nb, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

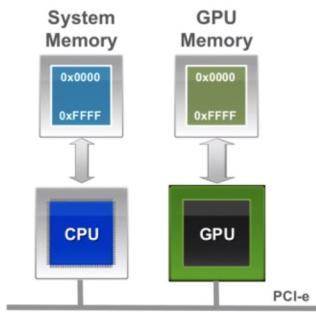


Le send() e recv() di MPI diventano fondamentali per lo scambio dei dati. I buffer andranno preparati a priori.

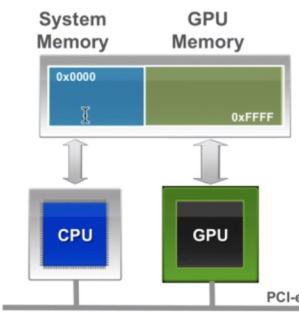
Nell'esempio bisogna spedire ai vicini i dati di cui hanno bisogno.

## UNIFIED VIRTUAL ADDRESSING

No UVA : Separate Address Spaces



UVA : Single Address Space



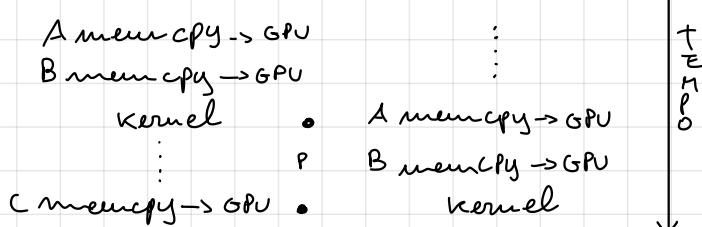
UVA:

Possiamo utilizzarla per fare una send da buffer senza passare da tutti i possibili intermedi (ci sono comunque degli scambi ma non passiamo da CPU e RAM, ma usiamo il bus. Possiamo anche passare delle reti senza passare dallo CPU/RAM).

## Streams:

Oltre a quello di default gli altri vengono creati e richieste.

↳ a meno che non ne vengano creati, le operazioni sono sequentiali su uno stream, che però sono parallelizzabili rispetto ad un altro stream.



LEZIONE 23 - 21/12/20 - Applicazioni sph e swe

## LEZIONE 24 - 22/12/20

Profilazione ed esecuzione assistita:

Memoria globale vs shared memory → i tempi possono avere risultati diversi, dipende. Usando le bande di memoria (+cache ecc), la memoria shared potrebbe non dare gli stessi vantaggi che invece dorebbe su une architetture più vecchie (per esempio).

NVIDIA Visual Profiler:

Aiuta a mostrarti le ottimizzazioni dei kernel, calcoli e tutto quello che facciamo. Possiamo vedere anche le varie memory e verso dove sono mandate.

Concurrent kernels e streams in parallelo sono anche semplici da vedere → questo ci mostra come lo scheduler si è comportato.

`nvprof`: è il programma che effettivamente esegue il profiloing.

`nvprof --usage //`

↳ gpu trace → chiamate kernel, blocchi, linea del tempo completa.  
chiamate CUDA → host

Event query → utile per debugging, info su cache L1, hit/miss corrispondenti.

NB:

| Non è vero che cercare di arrivare alla occupancy teorica massima sia la strada migliore in tutti i casi.