Riassunto

HARDWARE PERFORMANCE

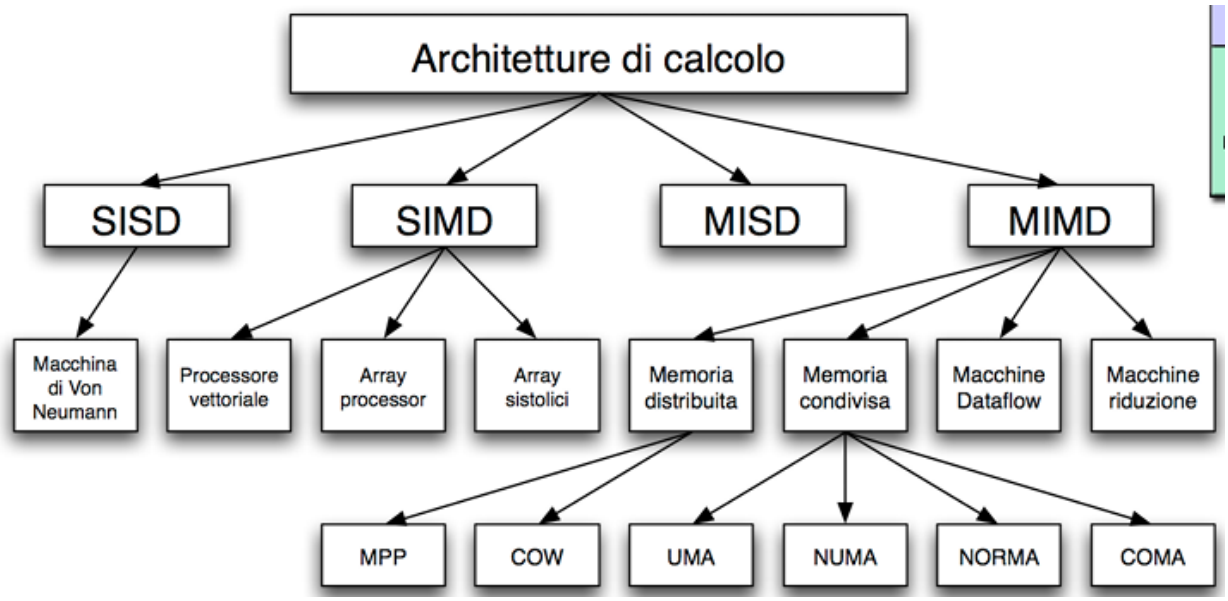
- ARCHITETTURE PARALLELE
  - distribuzione del carico computazionale su una architettura hardware in grado di sfruttare le performance dei diversi livelli di parallelismo su:
    - diversi thread o processi su un singolo nodo (memoria condivisa)
    - diversi processi su più nodi interconessi (memoria distribuita)
    - kernel di calcolo eseguiti su acceleratori (GPU)
- Parallelizzazione:
  - S**peedup** : misura dell'accelerazione del tempo di calcolo rispetto al programma non parallelizzato (**Speedup = Tseriale / Tparallelo)**
  - **Scalabilità**: capacità del programma parallelo a mantenere una crescita proporzionale allo Speed up, all'aumentare delle unità di processamento.
  - **Overhead**: limitazioni della scalabilità introdotte dalla parallelizzazione, come la comunicazione e il sincronismo tra i task.
- Programmazione a memorie:
  - memoria condivisa: realizzata con i sistemi SMP(symmetric multi processor) pool di processori indipendenti che condividono la stessa memoria Ram.
  - memoria **distribuita**: **cluster** di computer interconnessi da una rete di comunicazione.
  - GPU: creazione di un kernel di calcolo che viene avviato insieme ai dati dal risultato dell'host.
- Performance
  - CPU: operazioni svolte in una unità di tempo in virgola mobile come MFLOPS o GFLOPS
- Istruzioni vettoriali SMD e FMA: applicazioni con elevato carico computazionale sono **Data Parallel**, richiedono l'esecuzione di **SIMD** (single instruction multple

data). **FMA** sono chiamate dedicate che in un solo ciclo di clock eseguono 2 operazioni (somma e moltiplicazione)

- **ROOFLINE ANALYSIS:**
  - Il modello Roofline consente di stimare le performance di un kernel computazionale mostrando graficamente le limitazioni inerenti CPU/GPU e memoria.
- **Benchmarks**:
  - insieme di test software volti a fornire una misura delle prestazioni reali (sustained performance) di un computer per quanto riguarda diverse operazioni.
- Tempi di esecuzione:
  - time: ritorna i tempi di esecuzione del programma
  - gprofe : determina quali parti del programma consumano più tempo
  - clock_gettime() : funzione che determina i tempi di esecuzione all'interno del programma

SISTEMI PER IL CALCOLO AD ALTE PRESTAZIONI

- Tassonomia di Flynn
  - SISD: sistemi **seriali**, Un solo flusso di istruzioni eseguito dalla CPU, Un solo flusso di dati
  - SIMD: sistemi seriali con singolo flusso di istruzioni su più flussi di dati
  - MISD
  - MIMD: Ogni processore può eseguire un differente flusso di istruzioni. Ogni flusso di istruzioni lavora su un differente flusso di dati.



- Sistemi a memoria condivisa
  - UMA : tutti i processori accedono alla memoria come spazio di indirizzamento globale e le modifiche alla memoria effettuate da un processore sono visibili da tutti gli altri,
  - NUMA: ogni processore ha una propria memoria locale, l'insieme delle memorie locali forma uno spazio di indirizzi globale, accessibile da tutti i processori. I tempi di accesso alla memoria non sono uniformi (ritardi dovuti alla rete interconnessa)
- Sistemi a memoria distribuita
  - ogni processore possiede una propria memoria locale, che non fa parte dello spazio di indirizzamento degli altri processori. Ogni sistema/cpu è detto nodo e agisce in modo indipendente.
- Sistemi Ibridi
  - sistemi composti da più nodi a memoria condivisa interconnessi da una rete. I nodi possono disporre di GPU che a loro volta hanno un proprio spazio di memoria interconnesso.

PROGETTAZIONE DI PROGRAMMI PARALLELI

- **Parallel Computing**
  - la computazione parallela è una tecnica di programmazione che permette di dividere il carico di lavoro in Task diminuendo il carico complessivo.
    - **Decomposizione di dominio** (data parallelism): divisione di un dataset in piccoli sottodomini ognuno elaborato da un task.
    - **Decomposizione funzionale** (task parallelism): divisione del carico in base al lavoro che deve essere svolto, ogni task avrà un carico di lavoro indipendente e diverso.
- **Comunicazione tra i task:**
  - possono essere **punto-punto** o **collettive**.
  - Nel caso un problema non necessità di comunicazione → “embarassigly parallel”
  - La comunicazione tra task ha un **costo**, ogni messaggio inviato **influisce sulle prestazioni.**
  - i task possono interagire tra loro anche per **dipendenze** tra sezioni di codice o dati, ed eseguire il loro compito solo quando avvengono determinate azioni.
  - Per permette la **sincronizzazione tra i Task** si utilizzano 2 strumenti:
    - **barrier**: implicano il coinvolgimento di tutti i Task
    - **lock e semafori:** possono coinvolgere qualsiasi numero di task
  - **Se il sincronismo non viene gestito adeguatamente, può portare al degrado delle performance.**
- **Bilanciamento del carico:**
  - il **load balancing** è una tecnica di progetto con l’**obiettivo di distribuire il lavoro** in modo da **minimizzare i tempi di idle** (inattività) dei processi.
  - Il modello Master Slave è una tecnica comune di load balancing dove il master suddivide il lavoro in piccoli task e li assegna a un pool di slaves.
- **Granularità**

    - la decomposizione del problema può avvenire con diverse granuralità:
        - Parallelismo a grana fine: utile per bilanciare il carico e diminuire l'overhead di sincronismo. Aumento dell'overhead dovuto alle comunicazioni.
        - Parallelismo a grana grossa: migliora il rapporto tra calcolo e comunicazioni, difficile bilanciare il carico.
- **Scalabilità**
    - capacità del programma di mantenere una crescita proporzionale allo speedup al crescere delle unità di processamento.
    - Strong scalling: misura l'efficienza al crescere le unità di processamento mantenendo fissa la dimensione della complessità del problema.
    - Weak scalling: misura l'efficienza al crescere le unità di processamento aumentando la complessità del problema.
    - le limitazioni della scalabilità sono dovute a overhead introdotti dalla parallelizzazione.
- **Parallelismo automatico, guidato e manuale**
    - **Automatico** : il compilatore analizza il sorgente ed individua possibili parallelizzazioni.
    - **Guidato dal programmatore** : attraverso **direttive** il programmatore può dare indicazioni sulle parti di codice da parallelizzare e come farlo (esempio **openMP, direttive #pragma**).
    - **Esplicita** : occorre individuare manualmente i task e **programmare** esplicitamente l'interazione tra i task (esempio **MPI**).
- Programmazione delle istruzioni e parallela:
    - delle istruzioni SIMD:
        - Vettorizzazione: tecnica che consente di effettuare in parallelo la stessa operazione su tutti gli elementi di un vettore.
    - parallela MIMD:
        - i processi di un calcolatore parallelo comunicano tra loro secondo 2 schemi:

        - **Shared memory**, accedendo a variabili condivise, sincronizzazione tramite semafori (thread, processi) o **direttive OpenMP**.
        - **Message passing**: comunicando scambiandosi messaggi, grazie alla libreria **MPI** (Message Passing Interface)
- SPMD (Single program Multiple Data)
    - modello di programma in cui tutti i task eseguono la stessa copia del programma simultaneamente, elaborando dati diversi
    - Modello a memoria condivisa, unico programma con diversi thread che lavorano su dati diversi.
    - Esempio MPI: mpirun esegue n istanze del programma: `mpirun -np 4 a.out` → process number 4
- **Master Slave**
    - Un task Master che controlla e gestisce il lavoro svolto dagli altri tasks (Slaves), il modello più adatto per creare un programma Master-Slave è l'SPMD. Obiettivo è il load balancing.
    - Per creare un programma Master-Slave sono necessarie 2 operazioni fondamenali:
        - Fork: eseguita dal task Master, genera uno o più nuovi tasks che eseguono flussi paralleli al master.
        - Join: viene eseguita da tutti (o parte) dei task concorrenti, ed è di 3 tipi:
            - Join all: Il master attende l'ultimo task mentre gli altri terminano. **OpenMP** si basa sul modello Join All.
            - Join any: il primo task che arriva sblocca il master, gli altri terminano man mano che arrivano.
            - Join none: il master dopo la fork continua l'esecuzione.
        - Esempio openMP:
            - la regione parallela si attiva (fork) in openMP con la direttiva #omp parallel

- la direttiva omp for distribuisce su diversi thread le iterazioni di un ciclo for.
- le iterazioni su suddivise in chunck di dimensionne limitata che vengono distribuite dallo scheduler in modo statico o dinamico.
- al termine del ciclo for si chiude la legione parallela (join)

```
#pragma omp parallel
#pragma omp for
for (i = 1; i <= n; i++)
{
}
```

- Programmazione GPU con CUDA
  - Nvidia ha realizzato modelli GPU che possono essere utilizzzate come acceleratore del calcolo per applicazioni data parallel, Un esempio è CUDA, il cui relativo compilatore e nvcc.
  - il codice seriale o poco parallelo viene eseguito sulla cpu, il codice massicciamente parallelo di tipo data parralel viene scritto con CUDA in un kernel e trasferito su GPU.
- Sistemi ibridi
  - la programmazione ibrida combina il modello message passing MPI con il modello a thread OpenMP a cui si aggiunge la programmazione CUDA.

CUDA GPU

- Flusso dei processi
  - i dati in input vengono copiati dalla memoria della CPU alla memoria GPU
  - il codice viene caricato ed eseguito all’interno della GPU, dopo di che viene fatto il caching dei dati per ottenere delle migliori performance
  - i risultati vengono copiati dalla memoria GPU alla memoria CPU.
  - nvcc compila i programmi che sfruttano la GPU e si occupa di separare il codice per host e device, esempio: main() verrà processata da host e

mykernel() dal compilatore NVIDIA.

- Gestione della memoria
  - i device e gli host sono 2 entità separate:
    - i puntatori dei device puntano alla memoria GPU
    - i puntatori dei host puntano alla memoria CPU
  - per gestire la memoria dei device, cuda offre le API cudaMalloc(), cudaFree(), cudaMemcpy().
    - generalmente si mette la d_ davanti al nome per diferenziarle da quelle locali della cpu.

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

// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

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

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

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

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

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

Questo codice è un esempio di utilizzo di CUDA, una piattaforma di calcolo parallelo per GPU sviluppata da NVIDIA. Il codice dimostra come allocare memoria sul dispositivo,

copiare dati dal dispositivo all'host e viceversa, e lanciare un kernel CUDA per eseguire il calcolo sulla GPU.

Ecco una spiegazione passo per passo del codice:

1. Vengono dichiarate le variabili `a`, `b`, `c` come copie locali sull'host. Queste variabili conterranno i dati di input e output del kernel CUDA.
2. Vengono dichiarati i puntatori `d_a`, `d_b`, `d_c` come copie sul dispositivo. Questi puntatori saranno utilizzati per allocare memoria per i dati sul dispositivo.
3. Viene calcolata la dimensione di un intero utilizzando `sizeof(int)`.
4. Viene allocata la memoria sul dispositivo per le copie `d_a`, `d_b`, `d_c` utilizzando `cudaMalloc`. La funzione `cudaMalloc` alloca la memoria specificata dalla dimensione `size` sul dispositivo e restituisce un puntatore al blocco di memoria allocato.
5. Vengono impostati i valori di input `a` e `b` ai valori desiderati.
6. I dati di input `a` e `b` vengono copiati dalla memoria dell'host alla memoria del dispositivo utilizzando `cudaMemcpy`. La funzione `cudaMemcpy` copia i dati dalla sorgente all'oggetto di destinazione specificato. Nel caso specifico, i dati vengono copiati dalla memoria dell'host alla memoria del dispositivo.
7. Viene lanciato il kernel CUDA `add` utilizzando la sintassi `<<<1,1>>>`. Il kernel `add` somma i valori di `d_a` e `d_b` e memorizza il risultato in `d_c`. La sintassi `<<<1,1>>>` specifica che il kernel viene eseguito una sola volta su un singolo blocco di thread.
8. Il risultato calcolato dal kernel viene copiato dalla memoria del dispositivo alla memoria dell'host utilizzando `cudaMemcpy`. In questo caso, il valore calcolato viene copiato da `d_c` a `c`.
9. Viene liberata la memoria precedentemente allocata sul dispositivo utilizzando `cudaFree`.
10. Il programma termina restituendo 0 come valore di uscita.

In sintesi, il codice utilizza CUDA per eseguire il calcolo della somma tra due numeri interi sulla GPU, trasferendo i dati tra l'host e il dispositivo e utilizzando la funzione `add` come kernel CUDA per eseguire il calcolo parallelo.

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

    - fare il setup delle variabili.
    - copiare quelle necessario sulla memoria della GPU.
    - eseguire le funzioni kernel necessarie.
    - ricopiare i valori interessati sulla memoria della CPU.
- Sistema scalabile a blocchi
    - per la computazione parallela, quando uso la funzione add(), dovrò impostare i parametri nel seguente modo: **add<<< N, 1 >>>()**; per far si che la funzione venga eseguita N volte in parallelo.
    - Con un sistema scalabile è possibile utilizzare lo stesso programma con diverse tipologie di processori, quindi il programma iviene diviso in diversi blocchi eseguibili da differenti processori.
- Thread
    - Un blocco può essere diviso in più thread paralleli, questo può essere fatto modificando la funzione add() come segue:

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

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

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

/*Con queste due istruzioni ci sono problemi di data race perché le istruzioni
vengono eseguite con i WARP (pool di thread), che vengono schedulati, quello che
può succedere è che un WARP si trovi più avanti di un altro e che esegua istruzioni
successive che potrebbero dipendere da dati non ancora pronti o non ancora gestiti. */
```

    La soluzione per risolvere il problema è una barriera di sincronizzazione data dalla funzione `void __syncthreads();` (specifica per i casi in cui è possibile avvenga

ReadAfterWrite, WriteAfterRead e WriteAfterWrite) in cui nessun thread può andare avanti se tutti gli altri non hanno finito.

- Coordinazione tra host e device
  -

RIASSUNTO LAB E TEORIA COLLEGATA:

- **LAB PERFORMANCE**
  - benchmark
- **LAB Progettazione di programmi paralleli**
  - **CPI**
    - calcolo del Pi Greco
    - ottimizzazione automatica da compilazione -O2
    - **clock_gettime()** per determinare i tempi di calcolo
  - **Heat**
    - scomposizione del dominio, dividendo il rettangolo in sotto regioni, una per thread
    - legge di Amdhal alto, codice seriale
    - Overhead della parallelizzazione può includere la comunicazione tra thread
  - **Factorize**
    - scomposizione del dominio (numeri da testare tra i thread)
    - overhead introdotti dalla parallelizzazione basso, dato che maggior parte del codice è seriale
    - impatto legge di Amdhal alto, dato che è per di più seriale
- **LAB OpenMP base**
  - omp_overhead

        - calcolo dell’overhead dovuto all’avvio e chiusura di un pool di N thread.
    - omp_balancing
        - simulazione di un carico di lavoro parallelizzabile ma sbilanciato
        - il bilanciamento del carico e ottimizzazione dei tempi di calcolo di sum tramite direttive #pragma omp parallel for
        - bilanciamento del carico tramite maste-slave e master_slave con lock.
    - omp_cpi
        - strong scalling su codice per calcolo pi greco
        - lo strong scalling puo essere migliorato:
            - aumentando la dimensione del problema, qui n è settato a 20
            - gestione delle stampe, puo creare collo di bottiglia tra i thread
            - allocazione del lavoro, qui è usato #prgam omp parralel for, potremmo aumentare le performance con una (scheduling chunk)
- **Lab Applicazione OpenMP**
    - stampa tempi di calcolo (omp_get_wtime())
    - stampa rank dei thread (omp_get_thread_num())
    - accesso esclusivo stampa con #pragma omp critical
        - Heat
            - compilazione con diversi compilatori (gnu4, gnu8, intel)
            - opizioni di ottimizzazione -
                - o2 (minor tempo di compilazione e maggior rempo di esecuzione)
                - o3 (minor tempo di esecuzione e maggior tempo di compilazione)
        - Factorize
            - scaling per determinare core-hours per un numero da 128 bit
            - check point per modello master slave
- **LAB mpirun e mpicc**

- modelli a memoria distribuita con mpi
- **mpirun** realizza il modello **SPMD** (Single Program Multiple Data):
  - lo stesso programma viene eseguito su istanze multiple dette **task**.
    Ogni task ha la **propria** memoria e i propri dati. Ogni task è **identificato** da un numero intero denominato **rank** nel range [0, N-1].

- **LAB MPI base**
  - mpirun
    - caricamento moduli dell'ambiente necessari (module load intel impi)
    - esecuzione del programma su più host in parallelo (mpirun -np N) (npirun -machine machine.txt)
    - modello MPMD con `mpirun -np 4 hostname : -np 2 date`
  - mpicc
    - compilatore programma MPI
    - `#include "mpi.h"`: Questa direttiva include il file di intestazione MPI
    - `MPI_Init(&argc, &argv)`: Questa chiamata inizializza l'ambiente MPI
    - `MPI_Comm_size(MPI_COMM_WORLD, &numtasks)`: Questa chiamata restituisce il numero totale di processi (task) nel comunicatore MPI_COMM_WORLD e memorizza il valore nella variabile `numtasks`.
    - `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`: Questa chiamata restituisce il numero di rango del processo chiamante nel comunicatore MPI_COMM_WORLD e memorizza il valore nella variabile `rank`. Il rango è un identificatore unico assegnato a ciascun processo.
    - `MPI_Send(...)` e `MPI_Recv(...)`: Queste chiamate consentono la comunicazione tra processi MPI. `MPI_Send` invia un messaggio da un processo mittente a un processo destinatario, mentre `MPI_Recv` riceve un messaggio da un processo mittente.
    - `MPI_Wtime()`: Questa chiamata restituisce il tempo corrente in secondi.
    - `MPI_Barrier(MPI_COMM_WORLD)`: Questa chiamata sincronizza tutti i processi nel comunicatore MPI_COMM_WORLD, in modo che nessun processo

prosegua fino a quando tutti gli altri processi non hanno raggiunto il punto di barriera.

        - `MPI_Finalize()`: Questa chiamata termina l'ambiente MPI e deve essere chiamata da ogni processo MPI alla fine del programma.

- **LAB Applicazioni MPI**
    - da aggiungere:
        - intestazione #include <mpi.h>
        - l'inizializzazione dell'ambiente MPI: MPI_Init(&argc, &argv);
        - scomposizione di dominio
        - comunicazioni MPI e ricomposizione del dominio avvengono in MPI_GATHER nel rank 0
        - la chiusura dell'ambiente MPI: MPI_Finalize();
        - stampa tempo di calcolo t1=MPI_Wtime();
    - heat
        - **Overhead di comunicazione:** riduzione attraverso la sovrapposizione comunicazione /computazione
        - **Checkpointing.** Ogni simulazione può disporre di un tempo di calcolo limitato. Può essere utile la possibilità di salvare sul sistema di storage lo stato di avanzamento della simulazione in modo da poterlo riprendere in tempi successivi.
    - factorize
        - completare con e primitive MPI per il parallelismo
        - realizzare un modello di comunicazione tra task
- **LAB Programmazione ibrida MPI+OpenMP**
    - Nella **programmazione ibrida** (MPI + openMP) viene generalmente attivato un **solo task MPI per nodo** il quale si occupa della comunicazione con gli altri task, mentre il **calcolo all'interno del nodo viene parallelizzato con openMP**. Questa architettura consente di sfruttare la scalabilità multimodo di MPI ma riducendo al minimo l'overhead di comunicazione.

    - heat
        - creazione della versione omp+mpi aggiungendo direttive omp al codice mpi_heat
        - aggiunta stampa del master
    - factorize
        - suddivisione del dominio in base ai n tasks MPI, quindi i cicli di ogni task vengono distribuiti tra i thread openMP
        - implementazione architettura comunicazione asincrono che avvisa i thread quando viene trovato un fattore primo per interrompere l'attività
        - implementazione modello master-slave
- **LAB CUDA GPUmpiheat**
    - gpu + mpi
        - 2 gpu con comunicazione mpi per scambio dati di bordo
        - trasferimento gpu → ram (rank 0) → mpi → ram (rank 1) → gpu
        - output: ogni rank scrive su un file
    - compiliamo con nvcc specificando l'indirizzo della libreria mpi
        1. `cuda_runtime.h`: Libreria che fornisce definizioni e funzioni necessarie per la programmazione CUDA.
        2. `__global__`: Qualificatore utilizzato per dichiarare una funzione kernel CUDA che viene eseguita sul device (la GPU).
        3. `__host__ __device__`: Qualificatore utilizzato per dichiarare una funzione che può essere eseguita sia sulla CPU che sulla GPU.
        4. `cudaMemcpy`: Funzione utilizzata per copiare dati tra host e device o tra device e device.
        5. `cudaMalloc`: Funzione utilizzata per allocare memoria sul device.
        6. `cudaFree`: Funzione utilizzata per liberare la memoria allocata sul device.
- **LAB CUDA Matrix+cpi**

- **thread**, unità elementare di calcolo, puo essere lanciato in parallelo ad altri thread, il suo comportamento è programmato da un kernel
- **Blocco**, i thread (max 1024) sono raggrupati in blocchi che esegue i thread in parallelo , tutti che eseguono le stesse istruzioni (SIMD).
- I blocchi sono organizzati su una **griglia** 1D, 2D ,3D. non ce garanzia sull'ordine di esecuzione dei blocchi
  - **threadIdx**: identifica il thread all'interno del blocco (.x, .y, .z)
  - **blockIdx**: identifica il blocco all'interno della griglia (.x, .y, .z)
  - **blockDim**: descrive quanti thread ci sono in un blocco (.x, .y, .z)
  - **gridDim**: descrive quanti blocchi ci sono nella griglia (.x, .y, .z)
- cpi2
  - scritto cpi2.cu per calcolare cpi2
- matrixmul
  - estratto performance cpu, naive gpu, coalescing gpu, tiling gpu

- **LAB CUDA Heat**
  - gpu_heat
    - aggiunto timer per caloclo delle iterazioni kernel
    - modificato intestazione initialize()
  - gpu_heat_shmem
    - aggiunta condivisione delle memoria t_old
  - gpu_heat_blksz(4,8,16,32)
    - modificati i blocchi dei thread
    - i blocchi più grandi hanno più shared memory e possono performare meglio nel parallelismo