

CORSO PROGRAMMAZIONE PARALLELA E HPC: GPU

Alessandro Dal Palù¹

1. Dipartimento di Scienze Matematiche, Fisiche e Informatiche,
Università di Parma

A.A 22/23

- Architettura GPU
- Modello di programmazione
- Il prodotto tra matrici (CPU)
- Esempio implementazione GPU
- Utilizzo Shared Memory su GPU
- Confronti

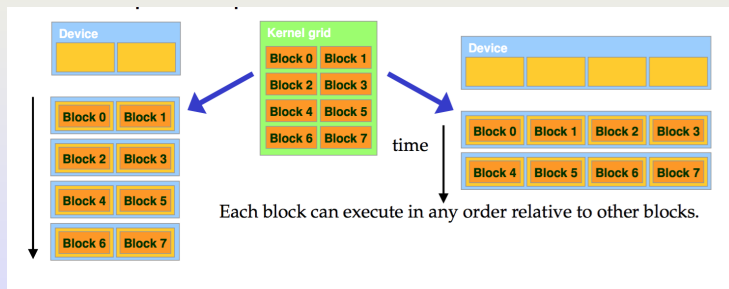
- Approfondimenti:
- <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- Ci riferiamo al modello CUDA C (definito da Nvidia)

THREAD

- Unità elementare di calcolo
- Esegue un flusso di esecuzione
- Può essere lanciato in parallelo ad altri thread
- Il suo comportamento è programmato da un *kernel*

BLOCCO

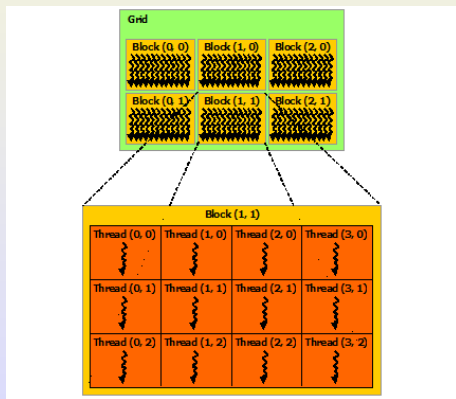
- I thread (max 1024) sono raggruppati in blocchi
- I thread di un blocco sono eseguiti in parallelo
- I thread del blocco eseguono le stesse istruzioni (idea SIMD)
- A seconda dei processori presenti, più blocchi eseguiti in parallelo



ARCHITETTURA GPU

ORGANIZZAZIONE

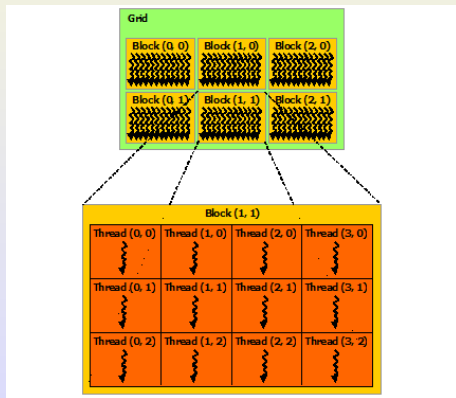
- I thread sono organizzati 1D, 2D o 3D nel blocco
- I blocchi sono organizzati su una griglia 1D, 2D o 3D
- Blocchi diversi possono essere eseguiti da processori diversi
- Non c'è garanzia sull'ordine di esecuzione dei blocchi



ARCHITETTURA GPU

ORGANIZZAZIONE

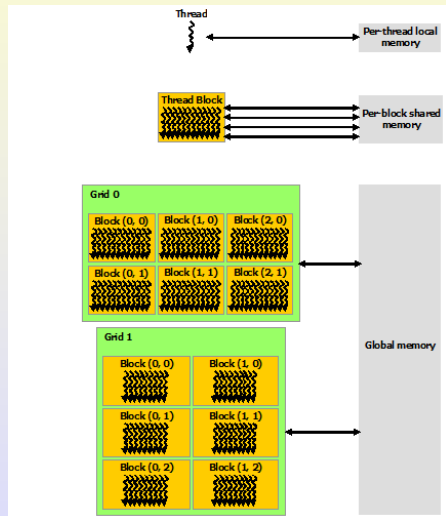
- `threadIdx`: identifica il thread all'interno del blocco (.x, .y, .z)
- `blockIdx`: identifica il blocco all'interno della griglia (.x, .y, .z)
- `blockDim`: descrive quanti thread ci sono in un blocco (.x, .y, .z)
- `gridDim`: descrive quanti blocchi ci sono nella griglia (.x, .y, .z)



- Le memorie presenti su una GPU sono di diverso tipo
- Il loro utilizzo ha un impatto notevole sui trasferimenti dati
- Necessaria una strutturazione dell'algoritmo parallelo per massimizzare prestazioni

ARCHITETTURA GPU

MEMORIE P100



- 255 registri per thread
- 64KB memoria condivisa per blocco (veloce se non ci sono conflitti di accesso)
- 12GB memoria globale (tempo accesso alto, trasferimenti a blocchi a grande larghezza di banda)

ARCHITETTURA GPU

ESEMPIO DI CALCOLO

- Il programma gira su CPU (host)
- Il programma chiede alla GPU (device) di eseguire un kernel
- La CPU può fare altro durante il lavoro della GPU
- E' possibile lanciare più kernel in parallelo (se non ci sono dipendenze)
- La CPU colleziona risultati dalla GPU e prosegue

C Program Sequential Execution

Serial code

Parallel kernel
Kernel0<<<>>>()

Serial code

Parallel kernel
Kernel1<<<>>>()

Host

Device

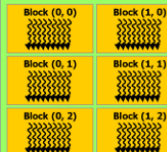
Grid 0



Host

Device

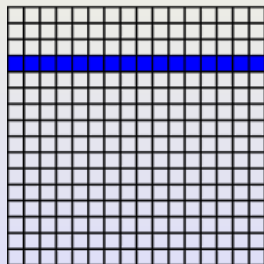
Grid 1



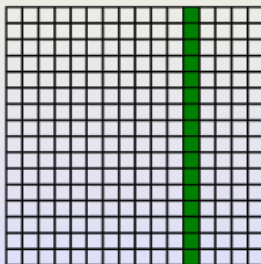
CALCOLO PRODOTTO MATRICI

- Prodotto $C = A \times B$ (A dimensione M x K, B dimensione K x N)
- Ogni cella di C richiede il prodotto scalare di due vettori
- Ogni cella di A o B viene letta K volte

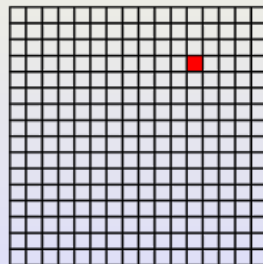
A



B



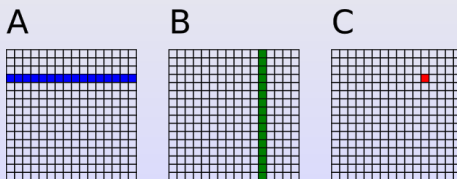
C



CALCOLO PRODOTTO MATRICI

IMPLEMENTAZIONE CPU

```
void main(){  
    for i = 0 to M do  
        for j = 0 to N do  
            /* compute element C(i,j) */  
            for k = 0 to K do  
                 $C(i,j) = C(i,j) + A(i,k) * B(k,j)$   
            end  
        end  
    end  
}
```

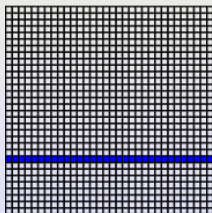


CALCOLO PRODOTTO MATRICI

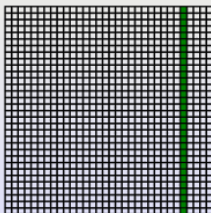
PARALLELIZZAZIONE SU GPU: IDEA NAIVE

- Ogni thread calcola una cella di $C(i, j)$
- Ogni thread carica una riga di A e una colonna di B dalla memoria globale
- Il thread calcola il prodotto e scrive il risultato nella cella in memoria globale

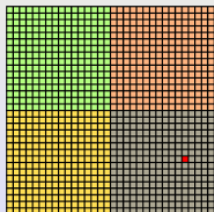
A



B



C



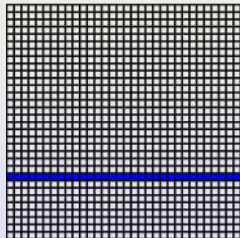
blockidx.x = 0 blockidx.x = 1
blockidx.y = 0 blockidx.y = 0
blockidx.x = 0 blockidx.x = 1
blockidx.y = 1 blockidx.y = 1

CALCOLO PRODOTTO MATRICI

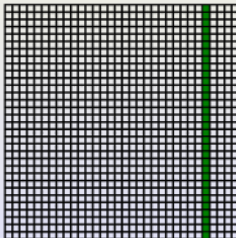
PARALLELEIZZAZIONE SU GPU: IDEA NAIVE

- Bisogna dividere il lavoro in blocchi da 1024 thread
- Viene comodo lavorare in 2D
- Ogni blocco può lavorare in modo indipendente dagli altri

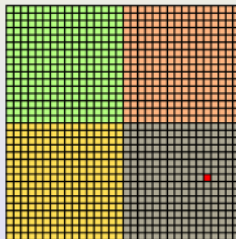
A



B



C

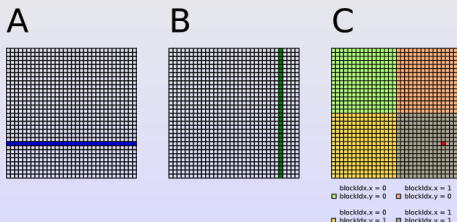


blockidx.x = 0	blockidx.x = 1
blockidx.y = 0	blockidx.y = 0
blockidx.x = 0	blockidx.x = 1
blockidx.y = 1	blockidx.y = 1

CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: IDEA NAIVE - CPU

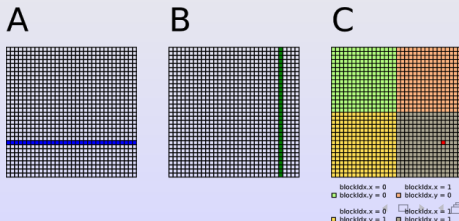
```
void main(){  
    define A_cpu, B_cpu, C_cpu in the CPU memory  
    define A_gpu, B_gpu, C_gpu in the GPU memory  
    memcpy A_cpu to A_gpu  
    memcpy B_cpu to B_gpu  
    dim3 dimBlock(32, 32)  
    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y)  
    matrixMul<<<dimGrid, dimBlock>>>(A_gpu,B_gpu,C_gpu,K)  
    memcpy C_gpu to C_cpu  
}
```



CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: IDEA NAIVE - GPU

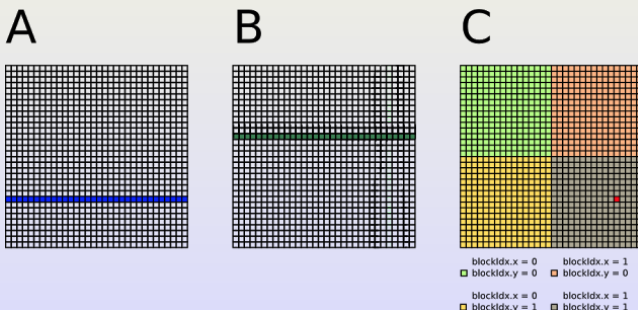
```
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K) {  
    temp = 0  
    // Row i of matrix C  
    i = blockIdx.y * blockDim.y + threadIdx.y  
    // Column j of matrix C  
    j = blockIdx.x * blockDim.x + threadIdx.x  
    for k = 0 to K-1 do  
        accu = accu + A_gpu(i,k) * B_gpu(k,j)  
    C_gpu(i,j) = accu  
}
```



CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: IDEA NAIVE

- Critiche: troppe letture da memoria globale
- Se i dati sono contigui e letti da thread contigui, buone prestazioni
- Tuttavia la matrice B non viene letta con celle contigue (inefficiente)
- Proposta: memorizzo B trasposta (memory coalescing)



CALCOLO PRODOTTO MATRICI

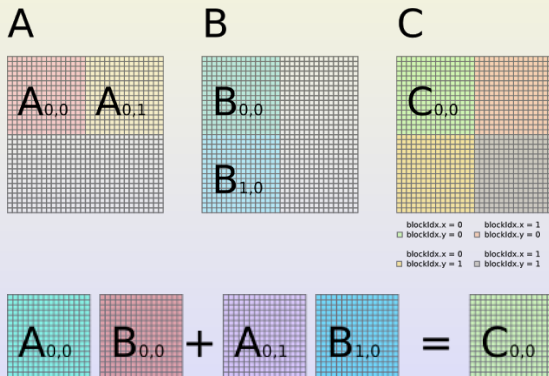
PARALLELIZZAZIONE SU GPU: IDEA NAIVE

- Critiche: troppe letture da memoria globale
- Soluzione: sfrutto la memoria condivisa del blocco
- Evito di leggere tante volte gli stessi dati
- Organizzo il calcolo parallelo in modo diverso

CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA

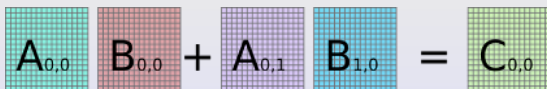
- Divido le matrici in blocchi (tiles)
- Calcolo ogni blocco di C in piu' fasi



CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA

- Ad ogni iterazione, ogni thread copia un blocco di A e uno di B dalla memoria globale in shared
- Ogni thread calcola il prodotto e aggiorna il risultato in un registro
- Al termine delle iterazioni, tutti i thread memorizzano il loro risultato (blocco di C) in memoria globale

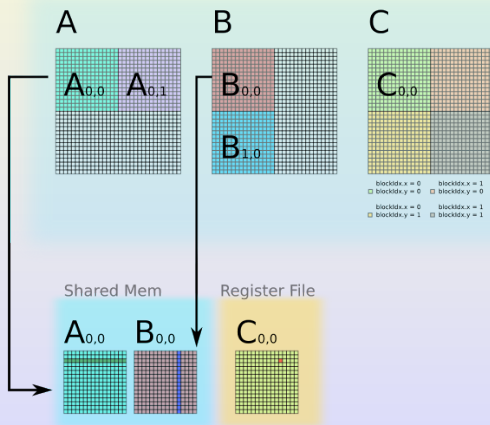

$$A_{0,0} B_{0,0} + A_{0,1} B_{1,0} = C_{0,0}$$

CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA

Prima coppia di blocchi

Global Memory

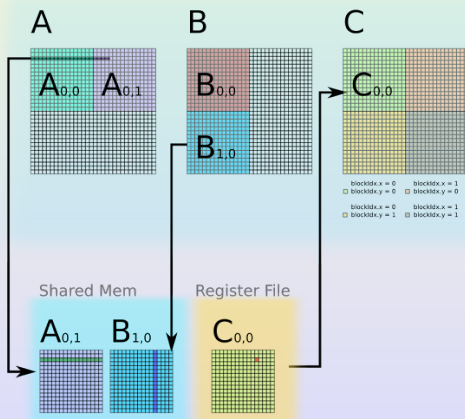


CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA

Ultima coppia di blocchi

Global Memory



CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA - GPU

```
__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K) {
    __shared__ float A_tile(blockDim.y, blockDim.x)
    __shared__ float B_tile(blockDim.x, blockDim.y)
    accu = 0
    for tileIdx = 0 to (K/blockDim.x - 1) do

//carica blocchi di A e B in shared mem
        i = blockIdx.y * blockDim.y + threadIdx.y
        j = tileIdx * blockDim.x + threadIdx.x
        A_tile(threadIdx.y, threadIdx.x) = A_gpu(i,j)
        B_tile(threadIdx.x, threadIdx.y) = B_gpu(j,i)
        __sync()
    ...
}
```

CALCOLO PRODOTTO MATRICI

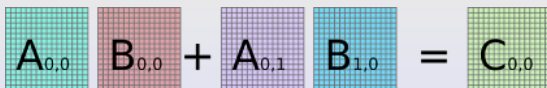
PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA - GPU

```
__global__ void matrixMul(A_gpu, B_gpu, C_gpu, K) {  
    ...  
    for tileIdx = 0 to (K/blockDim.x - 1) do  
        //carica blocchi di A e B in shared mem  
        // Prodotto scalare (accumulato)  
        for k = 0 to blockDim.x do  
            accu = accu + A_tile(thrIdx.y, k) * B_tile(k, thrIdx.x)  
        end  
        __sync()  
    end  
  
    //scrive il blocco in global  
    i = blockIdx.y * blockDim.y + threadIdx.y  
    j = blockIdx.x * blockDim.x + threadIdx.x  
    C_gpu(i, j) = accu  
}
```


CALCOLO PRODOTTO MATRICI

PARALLELIZZAZIONE SU GPU: MEMORIA CONDIVISA

- Le operazioni di calcolo (somme, moltiplicazioni) sono invariate
- Gli accessi alla memoria globale sono stati divisi per la dimensione del blocco
- Infatti un blocco di una matrice è letto una volta dalla globale e usato dalla shared più volte per calcolare il blocco C


$$A_{0,0} B_{0,0} + A_{0,1} B_{1,0} = C_{0,0}$$

SETUP

PREPARARE L'AMBIENTE

- Per poter generare gli eseguibili per una GPU, bisogna preparare l'ambiente
- Lanciare `> module load cuda`
- I files della lezione sono in
`/hpc/home/alessandro.dalpalu/gpu/matrixMul`
- Lanciare

```
cd
mkdir gpu
cd gpu
cp -R /hpc/home/alessandro.dalpalu/gpu/matrixMul .
cp -R /hpc/home/alessandro.dalpalu/gpu/cpi2 .
```

SETUP

PREPARARE L'AMBIENTE

- Usiamo un gestore della compilazione (Cmake)
- Lanciare

```
cd matrixMul/bin  
cmake ..  
make  
cd ..
```

PROVARE L'ESEMPIO

```
> cat slurm_launch_single
```

```
#!/bin/sh
#SBATCH --partition=gpu    # Richiedi un nodo con una gpu
#SBATCH --nodes=1
#SBATCH --mem=4G
#SBATCH --qos=gpu
#SBATCH --gres=gpu:p100:1
# Dichiaro che il job durera' al massimo 1 minuto
#SBATCH --time=0-00:01:00
#stampa il nome del nodo assegnato e argomenti
echo "#SLURM_JOB_NODELIST      : $SLURM_JOB_NODELIST"
echo "#CUDA_VISIBLE_DEVICES   : $CUDA_VISIBLE_DEVICES"
echo "size A= $WA X $HA,B= $WB X $HB"
module load cuda                #esegui il programma
./bin/matrixMul -wA=$WA -hA=$HA -wB=$WB -hB=$HB
```

PROVARE L'ESEMPIO

- Lo script prevede dei parametri da passare al programma
- E' possibile provare diverse dimensioni delle matrici
- `sbatch --export=WA=100,HA=100,WB=100,HB=1000
slurm_launch_single`
- Con eventuale reservation
- `sbatch --export=WA=100,HA=100,WB=100,HB=1000
--reservation=t_2022_hpcprogparg_20230525
slurm_launch_single`

PROVARE L'ESEMPIO

- Lo script `go` lancia diverse dimensioni (da confrontare)
- Attenzione: `sbatch` viene chiamato internamente a `go`
- Bisogna togliere la reservation se si lancia fuori dalla lezione!

```
> cat go
```

```
#!/bin/bash
for i in $(seq 100 100 1000); do
  sbatch --export=WA=$i,HA=$i,WB=$i,HB=$i \
    slurm_launch_single
done
```

PROVARE L'ESEMPIO

- Lanciare lo script `go`
- I risultati saranno scritti su vari file.
- Ogni esecuzione prova diverse idee (leggere sito indicato all'inizio)
- Per estrarre le performances di un esperimento (es. Tiling):

```
cat *.o* | grep -A1 "Tiling GPU" | grep time > tiling.txt
```

- Estrarre le prestazioni per CPU, Naive GPU, coalescing GPU e Tiling GPU.
- Caricare i 4 files su didattica-linux
- Adattare lo script in python (`cpi2_scaling.py`) per produrre un grafico con le 4 serie di dati
- Scrivere un programma per calcolare `cpi2` partendo dal file di esempio "`cpi2/cpi2.cu`"
- Compilare con `nvcc -arch=sm_60 cpi2.cu` e lanciare con `slurm_launch_cpi2`
- Includere il calcolo dei tempi di esecuzione