PARALLEL HIGH PERFORMANCE COMPUTING
CdS magistrale in informatica

9 – prodotto di matrici su GPU

Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

wpage.unina.it/lapegna

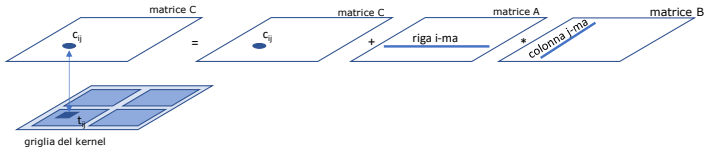
il prodotto di matrici

Date tre matrici A (di N righe e M colonne), B (di M righe e P colonne) e C (di N righe e P colonne) si vuole sviluppare un kernel CUDA che calcoli

$$C = C + A*B$$

Tale problema è un tipico esempio dove è possibile individuare un **parallelismo sui dati di tipo SIMD, ed è pertanto è naturalmente predisposto al GPU computing.**

ogni componente $c_{i,j}$ della matrice soluzione C dipende da due indici, per cui **l'idea di base è quella di organizzare i blocchi e i thread del kernel secondo la stessa struttura 2-dimensionale** e far calcolare ad ogni thread una componente di C, concorrentemente agli altri. Più precisamente, ogni thread $t_{i,j}$ calcola il prodotto scalare tra la riga ima di A e la colonna j-ma di B.

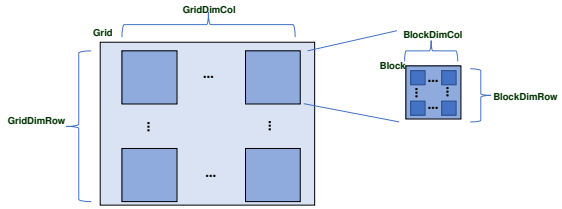


organizzazione della griglia di esecuzione

Siano allora BlockDimRow e BlockDimCol il numero di thread che definiscono la **struttura 2-dimensionale dei thread in ogni blocco** (ad esempio BlockDimRow=32 e BlockDimCol=32 così che il numero complessivo di thread in ogni blocco sia BlockDimRow x BlockDimCol = 1024, il massimo consentito da CUDA).

Considerando per semplicità il caso in cui N e P siano multipli di BlockDimRow e BlockDimCol, si ha che la **griglia di esecuzione del kernel avrà un numero di blocchi** rispettivamente uguale a

- GridDimRow = N / BlockDimRow lungo le righe
- GridDimCol = P / BlockDimCol lungo le colonne



algoritmo del kernel

Si osservi, inoltre, che BlockDimRow e BlockDimCol corrispondono alle variabili di ambiente blockDim.x e blockDim.y, per cui ciascun thread avrà i consueti identificativi globali

idglob_x = blockDim.x*blockIdx.x + threadIdx.x;

idglob_y = blockDim.y*blockIdx.y + threadIdx.y;

Come conseguenza, ciascun thread del kernel deve calcolare il prodotto scalare tra

- la riga di A con indice idglob_x
- la colonna di B con indice idglob_y

cioè

$$c_{iglob\_x\ iglob\_y} = \sum_{k=0}^{M-1} a_{iglob\_x\ ,k}\ \ b_{k,iglob\_y}$$

Si osservi come, in questo algoritmo, ciascun thread all'interno del kernel deve **effettuare 2M accessi alla memoria globale per eseguire 2M operazioni**

function matmatgpu1

Nella slide seguente è riportata la function matmatgpu1

```
void matmatgpu1(int lda, int ldb, int ldc, double *A, double *B, double *C,
                int N, int M, int P)
```

Poiché i dati A, B e C risiedono inizialmente nella memoria dell'host, prima dell'esecuzione del kernel è necessario il loro trasferimento in altre aree di memoria Ad, Bd e Cd allocati nella memoria device.

Si osservi che se le leading dimension degli array non coincidono con il numero di colonne delle matrici, allora le relative righe non sono contigue in memoria. Per effettuare un solo trasferimento può essere utile utilizzare un area di memoria (buffer) in cui compattare gli elementi prima del trasferimento.

Dopo la configurazione della griglia e l'esecuzione del kernel, si procede a trasferire i risultati nella memoria host.

La function matmatgpu1 considera il caso in cui N e P siano multipli di BlockDimRow e BlockDimCol, ma una eventuale generalizzazione non ne altera in maniera significativa la struttura.

function matmatgpu1

```
void matmatgpu1(int lda, int ldb, int ldc, double *A, double *B, double *C,
        int N, int M, int P){

__global__ void kernel1(double *, double *, double *, int, int, int);// prototipo kernel
double *Adev, *Bdev, *Cdev, *buffer;
int i, j, mas, BlockDimRow, BlockDimCol;

if ( N > M ) mas = N; else mas = M;  if ( P > mas ) mas = P;                  // buffer per il trasferimento
buffer = (double *)malloc(sizeof(double)*mas*mas);

cudaMalloc( (void**)&Adev, sizeof(double)*N*M );                              // allocazione memoria device
cudaMalloc( (void**)&Bdev, sizeof(double)*M*P );
cudaMalloc( (void**)&Cdev, sizeof(double)*N*P );

for(i=0; i<N; i++){                                                           // trasferimento matrice A
for(j=0; j<M; j++){
*(buffer + i*M + j)  =  *(A + i*lda + j);  }}
cudaMemcpy(Adev, buffer, sizeof(double)*N*M, cudaMemcpyHostToDevice);

for(i = 0; i < M; i++){                                                        // trasferimento matrice B
for(j = 0; j < P; j++){
*(buffer + i*P + j)  =  *(B + i*ldb + j);  }}
cudaMemcpy(Bdev, buffer, sizeof(double)*M*P, cudaMemcpyHostToDevice);

for(i = 0; i < N; i++){                                                       // trasferimento matrice C
for(j = 0; j < P; j++){
*(buffer + i*P + j)  =  *(C + i*ldc + j);  }}
cudaMemcpy(Cdev, buffer, sizeof(double)*N*P, cudaMemcpyHostToDevice);
```

continua

function matmatgpu1

```
BlockDimRow = 32; BlockDimCol = 32;                                           // configurazione griglia
dim3 DimBlock(BlockDimRow, BlockDimCol);
dim3 DimGrid( N/BlockDimRow, P/BlockDimCol );
kernel1 <<< DimGrid , DimBlock >>> (Adev, Bdev, Cdev, N, M, P);               // esecuzione kernel
cudaDeviceSynchronize();

cudaMemcpy(buffer, Cdev, sizeof(double)*N*P, cudaMemcpyDeviceToHost);// trasferimento risultato
for(i=0; i<N; i++){
for(j = 0; j < P; j++){
*(C + i*ldc + j) =  *(buffer + i*P + j);  }}

cudaFree(Adev); cudaFree(Bdev); cudaFree(Cdev);

}                                                                             // fine function
```

kernel kernel1

Il kernel richiamato dalla function matmatgpu1, definisce gli identificativi globali e calcola il prodotto scalare tra una riga di A e una colonna di B

```
__global__ void kernel1(double *Adev, double *Bdev, double *Cdev, int N, int M, int P){
int k, idglob_x, idglob_y;
double sum;

idglob_x = blockDim.x*blockIdx.x + threadIdx.x;              // identificativi globali del thread
idglob_y = blockDim.y*blockIdx.y + threadIdx.y;

        sum = Cdev[idglob_x *P + idglob_y];
         for(k=0; k<M; k++){                                 // calcolo prodotto scalare
               sum = sum + Adev[idglob_x*M + k ] * Bdev[idglob_y + k*P];
        }
        Cdev[idglob_x *P + idglob_y] = sum;

}                                                            // fine kernel1
```

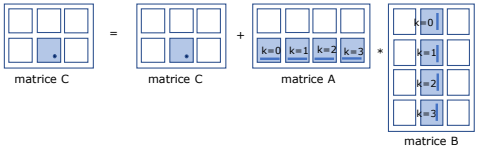

memoria shared

Una caratteristica importante del modello di programmazione CUDA è la presenza di una **memoria veloce (chiamata shared memory) condivisa da tutti i thread all'interno di un singolo blocco** della griglia di esecuzione del kernel. Tale memoria ha una latenza circa 100 volte inferiore rispetto a quella della global memory, per cui un suo utilizzo consente di migliorare notevolmente le prestazioni del codice.

A tal fine si consideri il caso di una griglia di esecuzione in cui **i blocchi dei thread sono quadrati** (ad esempio BlockDimRow = BlockDimCol = 32). Il prodotto C=C+A*B può essere decomposto come prodotto a blocchi del tipo

$$C(i,j) = \sum_{k=0}^{M/32-1} A(i,k)B(k,j)$$

dove ogni blocco di C è assegnato ad un blocco della griglia di esecuzione del kernel e i **blocchi di A e di B hanno le stesse dimensioni di quelli di C.**



algoritmo del kernel

Quest'ultima caratteristica fa si che, per un **fissato valore del passo k,** ciascun thread all'interno di un blocco può copiare **un elemento di A e un elemento di B dalla global memory alla shared memory concorrentemente agli altri thread** del blocco.

Poiché la shared memory è condivisa tra tutti i thread del blocco, **ciascun thread può successivamente calcolare un contributo** al prodotto scalare utilizzando i **dati presenti nella shared memory prelevati dagli altri thread**.

Più precisamente, ciascun thread calcola il prodotto scalare, in **M/32 passi**, utilizzando **32 componenti alla volta** di A e B, nel seguente modo

$$c_{iglob\_x\ iglob\_y} = \sum_{k=0}^{31} a_{iglob\_x,\,k}\ \ b_{k,iglob\_y} + \sum_{k=32}^{63} a_{iglob\_x,\,k}\ \ b_{k,iglob\_y} + \ \ldots\ldots$$

Si è considerato il caso in cui le dimensioni delle matrici siano multiplo di BlockDimRow = BlockDimCol =32. La generalizzazione deve tenere conto di eventuali thread non attivi.

E' importante osservare come, in questo algoritmo, ciascun thread all'interno del kernel **effettua 1 solo accesso alla memoria globale per eseguire un numero maggiore di operazioni (32 nell'esempio).**

kernel kernel2

Di seguito è riportato il kernel kernel2 che implementa l'algoritmo descritto che fa uso della shared memory. Esso può essere richiamato direttamente dalla function matmatgpu1 al posto di kernel1. Le istruzioni di sincronizzazione garantiscono che tutti i thread abbiano finito di copiare i propri elementi di A e di B nella shared memory prima di procedere con il calcolo del prodotto scalare.

```
__global__ void kernel2 (double *Adev, double *Bdev, double *Cdev, int N, int M, int P){
int k , kk, end, idglob_x, idglob_y;
__shared__ double Ashared[32][32], Bshared[32][32];                  // dichiarazione shared mem.
double sum;

idglob_x = blockDim.x*blockIdx.x + threadIdx.x;                      // identificativi globali
idglob_y = blockDim.y*blockIdx.y + threadIdx.y;


        sum = Cdev[idglob_x *P + idglob_y];
        for(k = 0; k < M/32 ; k++){                                  // ciclo sui blocchi

               // copia di un elemento di A e di B nella shared memory
                Ashared[ threadIdx.x ][ threadIdx.y ] = Adev[ idglob_x*M + threadIdx.y + 32*k];
                Bshared[ threadIdx.x ][ threadIdx.y ] = Bdev[ idglob_y + (32*k+threadIdx.x)*P];
                __syncthreads ( ) ;

                for(kk=0; kk<32; kk++){                              // contributo al prod. scal.
                    sum = sum + Ashared[ threadIdx.x ][ kk ] * Bshared[ kk ][ threadIdx.y ];
                }
                __syncthreads ( ) ;
        }
        Cdev[idglob_x *P + idglob_y] = sum;

}                                                                    // fine kernel 2
```