CUDA: memoria condivisa

Argomenti

Argomenti

Prodotto matrice matrice (1)

Supponiamo di avere $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times M}$, vogliamo calcolare la matrice $C \in \mathbb{R}^{M \times M}$, $C = A * B$.

Prima implementazione:

- Ogni thread calcola un elemento di C,
- Uso un blocco di threads con configurazione 2D

Prodotto matrice matrice (1): kernel

kernel

```
__global__ void my_sgemm_0kernel (int M, int K, float *A,
        int lda, float *B, int ldb, float *C, int ldc)
{

int idx_x = threadIdx.x;
int idx_y = threadIdx.y;
int kk; float sum = 0;

  if (idx_x < M && idx_y < M){

    for(kk = 0; kk < K; kk++){
      sum += A[idx_y*lda+kk] * B[kk*ldb + idx_x];
    }

    C[idx_y*ldc + idx_x] += alpha*sum;
  }

}
```

Prodotto matrice matrice (1): configurazione

configurazione di chiamata

```
TILEWIDTH = 16;
dim3 dB(TILEWIDTH,TILEWIDTH,1);
dim3 dG(1, 1, 1);

my_sgemm_0kernel <<< dG, dB >>> ( 16, K, A, K,
                                  B, M, C, M);
```

Limite:

- posso gestire matrici fino a 16×16 elementi,

Soluzione:

- sfrutto più di un blocco modificando la griglia: cambio il kernel e la configurazione di chiamata.

Prodotto matrice matrice (2): kernel

kernel

```
__global__ void my_sgemm_1kernel (int M, int K, float *A,
        int lda, float *B, int ldb, float *C, int ldc)
{

int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
int idx_y = blockIdx.y * blockDim.y + threadIdx.y;
int kk; float sum = 0;

  if (idx_x < M && idx_y < M){

    for(kk = 0; kk < K; kk++){
      sum += A[idx_y*lda+kk] * B[kk*ldb + idx_x];
    }

    C[idx_y*ldc + idx_x] += alpha*sum;
  }

}
```

Prodotto matrice matrice (2): configurazione

configurazione di chiamata

```
TILEWIDTH = 16;
dim3 dB(TILEWIDTH,TILEWIDTH,1);
dim3 dG( M/TILEWIDTH, M/TILEWIDTH, 1);

my_sgemm_0kernel <<< dG, dB >>> ( M, K, A, K,
                                  B, M, C, M);
```

Limite:

- alcuni threads utilizzano le stesse componenti della matrice, leggendoli in modo non collaborativo dalla memoria globale

Soluzione:

- sfrutto la memoria condivisa.

Argomenti

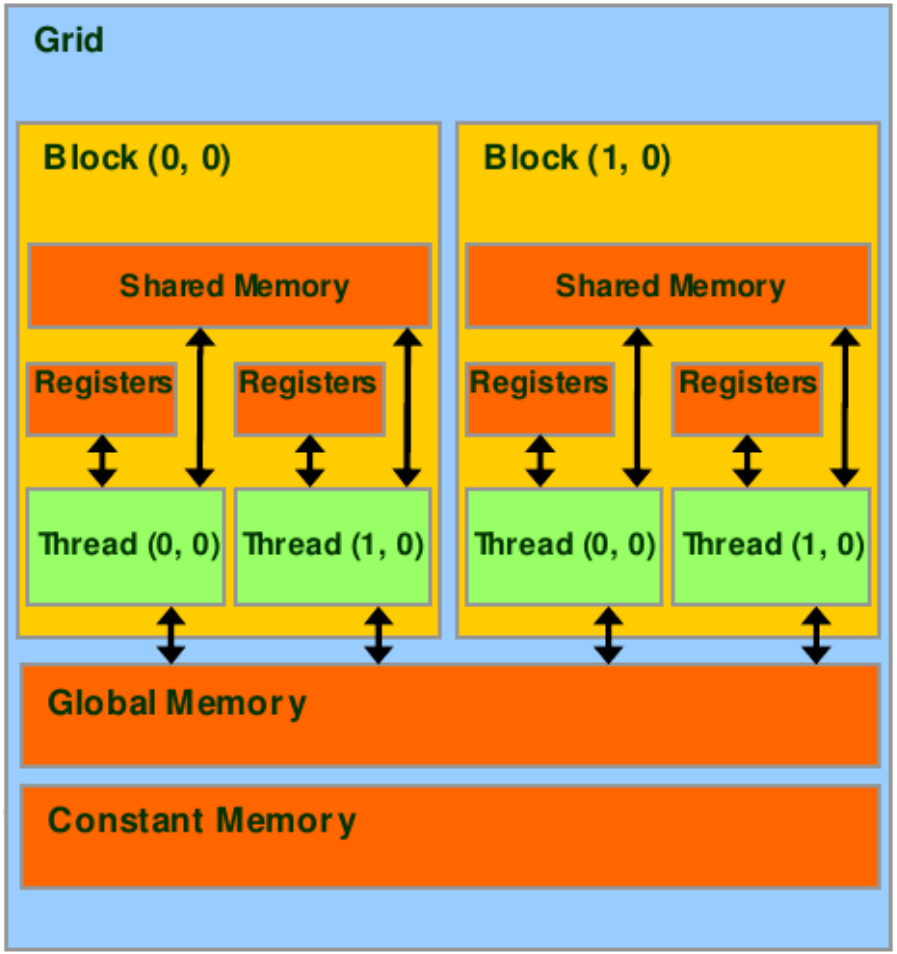
Memoria condivisa

In CUDA, la memoria globale

- può essere utilizzata da tutti i threads,
- viene allocata da `cudaMalloc`,
- può essere inizializzata dalla CPU con `cudaMemcpy`.

La memoria condivisa

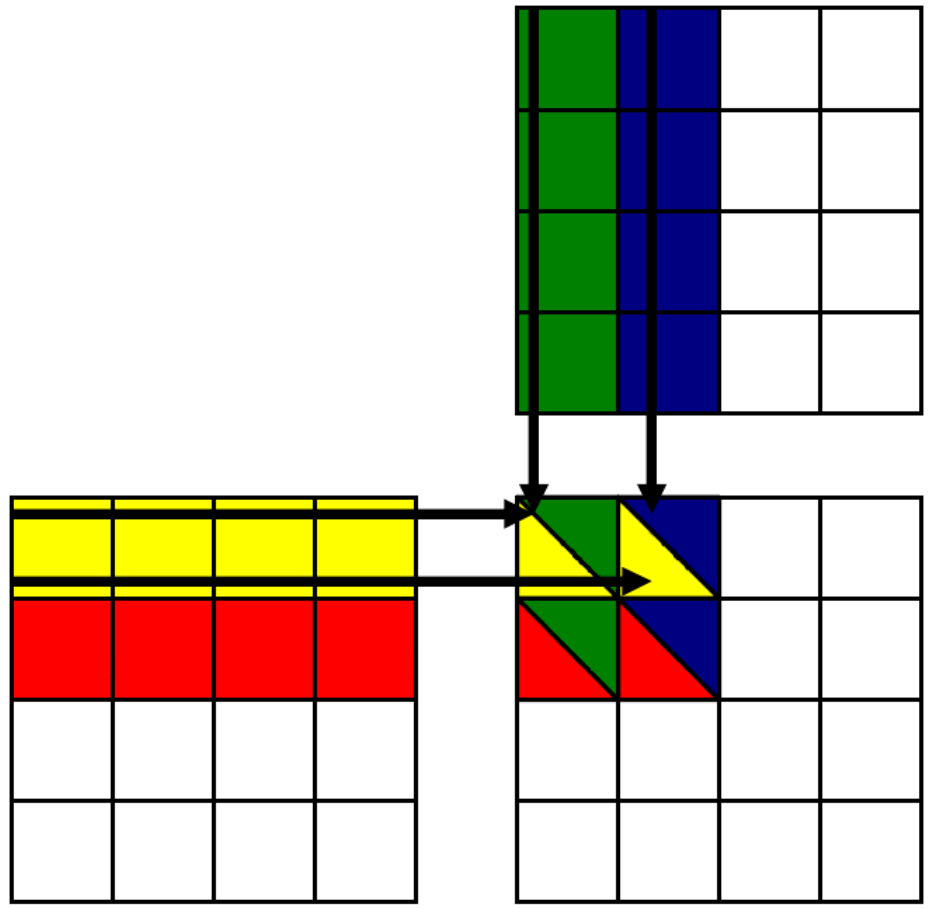
- è riservata ai threads che compongono il blocco,
- di velocità pari ai registri, on-chip,
- di piccole dimensioni (qualche KB).



Tipi di memorie

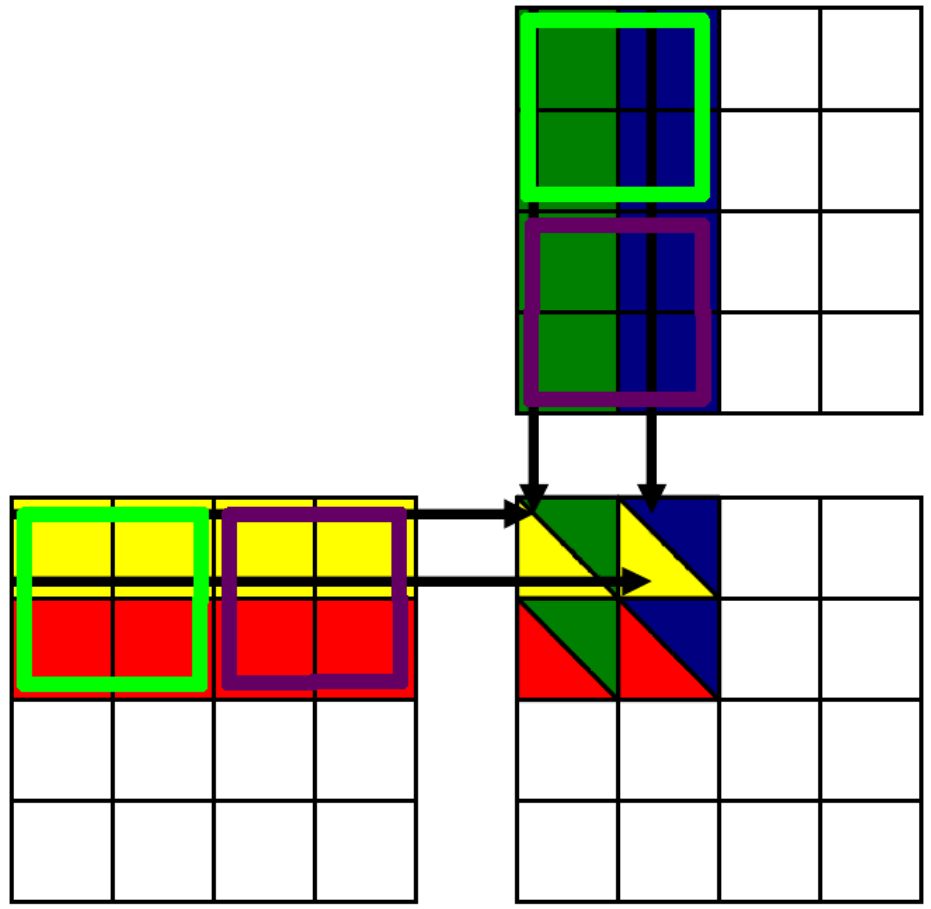
Keyword	Posizione fisica	Visibilità	Tempo di vita
variabili scalari automatiche	registri	thread	kernel
array automatici	m. globale	thread	kernel
`__device__ __shared__ int SharedVar;`	m. condivisa	blocco	kernel
`__device__ int GlobalVar;`	m. globale	griglia	applicazione
`__device__ __constant__ int ConstVar;`	m. costante	griglia	applicazione

Prodotto matrice matrice (3): memoria condivisa (I)



TILEWIDTH = 2

Prodotto matrice matrice (3): memoria condivisa (II)



TILEWIDTH = 2

Funzioni di Sincronizzazione

- La sincronizzazione è possibile solo tra threads appartenenti allo stesso blocco.
- Tutti gli appartenenti al blocco devono utilizzare la direttiva di sincronizzazione
- `void __syncthreads()`
  è usata per coordinare la comunicazione tra threads dello stesso thread block.

Quando alcuni thread nel thread block compiono accessi allo stesso indirizzo nella memoria globale o condivisa, ci sono potenziali rischi di read-after-write, write-after-read, or write-after-write. Queste potenziali inconsistenze di dati possono essere evitate sincronizzando i threads tra gli accessi.

Prodotto matrice matrice (3): kernel

kernel

```
__global__ void my_sgemm_2kernel (int M, int K, float *A, int lda,
                                  float *B, int ldb, float *C, int ldc) {
__shared__ float As[TILEWIDTH][TILEWIDTH];
__shared__ float Bs[TILEWIDTH][TILEWIDTH];

int idx_x = blockIdx.x * TILEWIDTH + threadIdx.x;
int idx_y = blockIdx.y * TILEWIDTH + threadIdx.y;
int jj,kk; float sum = 0;

    for(kk = 0; kk < K/TILEWIDTH; kk++){
        As[threadIdx.y][threadIdx.x] =
            A[idx_y*lda + (kk*TILEWIDTH + threadIdx.x)];
        Bs[threadIdx.y][threadIdx.x] =
            B[(kk*TILEWIDTH + threadIdx.y) * ldb + idx_x];

        __syncthreads();

        for(jj = 0; jj < TILEWIDTH; jj++)
            sum += As[threadIdx.y][jj] * Bs[jj][threadIdx.x];

        __syncthreads();
    }

        C[idx_y*ldc + idx_x] += alpha*sum;
}
```