CUDA: gestione threads e memoria globale

Argomenti

Addizione tra vettori

Kernel di addizione

```
__global__ void add(int *a, int *b, int *c){
    *c = *a + *b;
}
```

- `__global__` indica che la funzione verrà chiamata dall'host e verrà eseguita dal device
- i puntatori a, b e c devono fare riferimento a memoria allocata sul device.

Argomenti

Tipi di memorie

Tipo di memoria	Visibilità	Tempo di vita
Memoria privata del thread (registri)	thread	esecuzione del Kernel
Memoria condivisa	thread block	esecuzione del Kernel
Memoria globale	all	applicazione

Gestione della memoria globale

Allocazione

```
cudaError_t cudaMalloc(void ** ptr, size_t size)
```

Alloca size bytes di memoria sul device e fornisce il puntatore per accedervi nella variabile *ptr. Ritorna cudaErrorMemoryAllocation in caso di errore.

Deallocazione

```
cudaFree( void *ptr)
```

Esempio

```
int main (int argc, char *argv[]){
double *dati_d;
int N = 1024;

    cudaMalloc( (void **) &dati_d, N * sizeof(double));

    cudaFree(dati_d);
  }
```

Copia

Copia

```
cudaError_t cudaMemcpy (void *dst, const void *src,
                        size_t size,
                        enum cudaMemcpyKind kind)
```

Copia size bytes di memoria da src a dst, il tipo di copia è specificato da kind: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice.

Argomenti

Somma di vettori: main

```
int main (int argc, char *argv[]){
double *a_h, *b_h, c_h*; // dati sull'host
double *a_d, *b_d, c_d*; // dati sul device
int N = 1024;

    a_h = (double *) malloc(N * sizeof(double));
    b_h = (double *) malloc(N * sizeof(double));

    cudaMalloc( (void **) &a_d, N * sizeof(double));
    cudaMalloc( (void **) &b_d, N * sizeof(double));
    cudaMalloc( (void **) &c_d, N * sizeof(double));

    /* copia dei dati sul device */
    cudaMemcpy(a_d, a_h, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, N * sizeof(double), cudaMemcpyHostToDevice);

    /* lancio del kernel: c_d = a_d + b_d */

    c_h = (double *) malloc(N * sizeof(double));
    cudaMemcpy(c_h, c_d, N * sizeof(double), cudaMemcpyDeviceToHost);

    cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);

    free(a_h); free(b_h); free(c_h);
  }
```

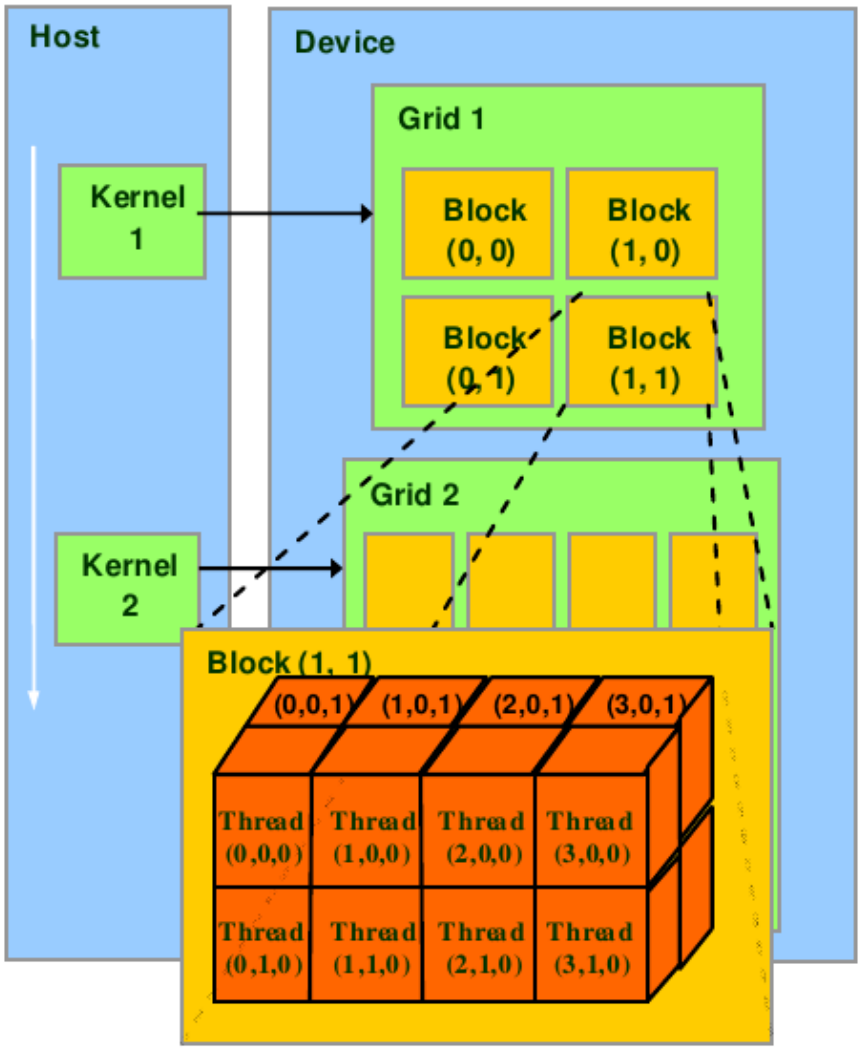
Somma di vettori: kernel

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

Utilizzando la variabile threadIdx.x, ogni thread accede a una porzione diversa dei dati.

Argomenti

Blocchi e Griglie




```
dim3 dimGrid(2,2,1);
dim3 dimBlock(4,2,2);

Kernel_1 <<< dimGrid, dimBlock >>> ();
```

Gestire la dimensione dei blocchi

Un blocco di thread:

- viene assegnato ad un unico streaming multiprocessor (SM),
- tutti i thread di un blocco condividono la stessa memoria `__shared__`,
- un blocco può contenere un massimo di 1024 threads (dipende dall'architettura),
- la configurazione del blocco si sceglie in fase di esecuzione del kernel.

I blocchi di thread sono a loro volta raggruppati in una griglia, la cui dimensione è scelta durante la chiamata al blocco:

Esecuzione di un kernel

```
add <<< dimGrid, dimBlock >>> ( c_d, a_d, b_d );
```

dove dimGrid e dimBlock sono di tipo dim3.
I threads si possono individuare in modo univoco osservando la struttura threadIdx. I blocchi di threads si possono individuare in modo univoco osservando la struttura blockIdx.

Argomenti

Somma di vettori (2): main

```
#define THREADS_PER_BLOCK 512
int main (int argc, char *argv[]){
double *a_h, *b_h, c_h*; // dati sull'host
double *a_d, *b_d, c_d*; // dati sul device
int N = 1024;

/*
  Allocazione memoria, copia dati
 */

    dim3 dimBlock ( THREADS_PER_BLOCK, 1, 1 );
    dim3 dimGrid  ( N / THREADS_PER_BLOCK, 1, 1);

    add <<< dimGrid, dimBlock >>> (c_d, a_d, b_d);

/*
  Copia, deallocazione
*/

}
```

Vengono richiesti 512 threads per blocco e un numero di blocchi sufficienti per coprire tutti i dati.

Somma di vettori (2): kernel

```
__global__ void add( double* c, double* a, double *b ){
int idx = blockIdx.x * blockDim.x + threadIdx.x;

    c[idx] = a[idx] + b[idx];

}
```

Utilizzando le variabili threadIdx.x e blockIdx.x, ogni blocco accede a una porzione diversa dei dati. Altre variabili automatiche disponibili: gridDim, blockDim.

Esempio concluso

main

```
#define THREADS_PER_BLOCK 512
int main (int argc, char *argv[]){
double *a_h, *b_h, c_h*; // dati sull'host
double *a_d, *b_d, c_d*; // dati sul device
int N = 1024;

    /* Allocazione memoria, copia dati  */
    dim3 dimBlock ( THREADS_PER_BLOCK, 1, 1 );
    dim3 dimGrid  ( (N + THREADS_PER_BLOCK -1) / THREADS_PER_BLOCK, 1, 1);

    add <<< dimGrid, dimBlock >>> (c_d, a_d, b_d, N);

    /* Copia, deallocazione */
}
```

kernel

```
__global__ void add( double *c, double *a, double *b, int N ){
int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        c[idx] = a[idx] + b[idx];
}
```