CUDA programming interface - CUDA C

Presentation

CUDA functions

Device settings

Memory (de)allocation within the device global memory

data transfers between host memory and device global memory

Data partitioning and kernel over this partition

Synchronization

Running several kernels concurrently

Function calls within the device

Compiling a CUDA code with nvcc

Examples - host codes and device codes

Presentation

- *CUDA C* =
  - C language
  - library of *CUDA functions*
  - *C language extensions* for kernel definition and call

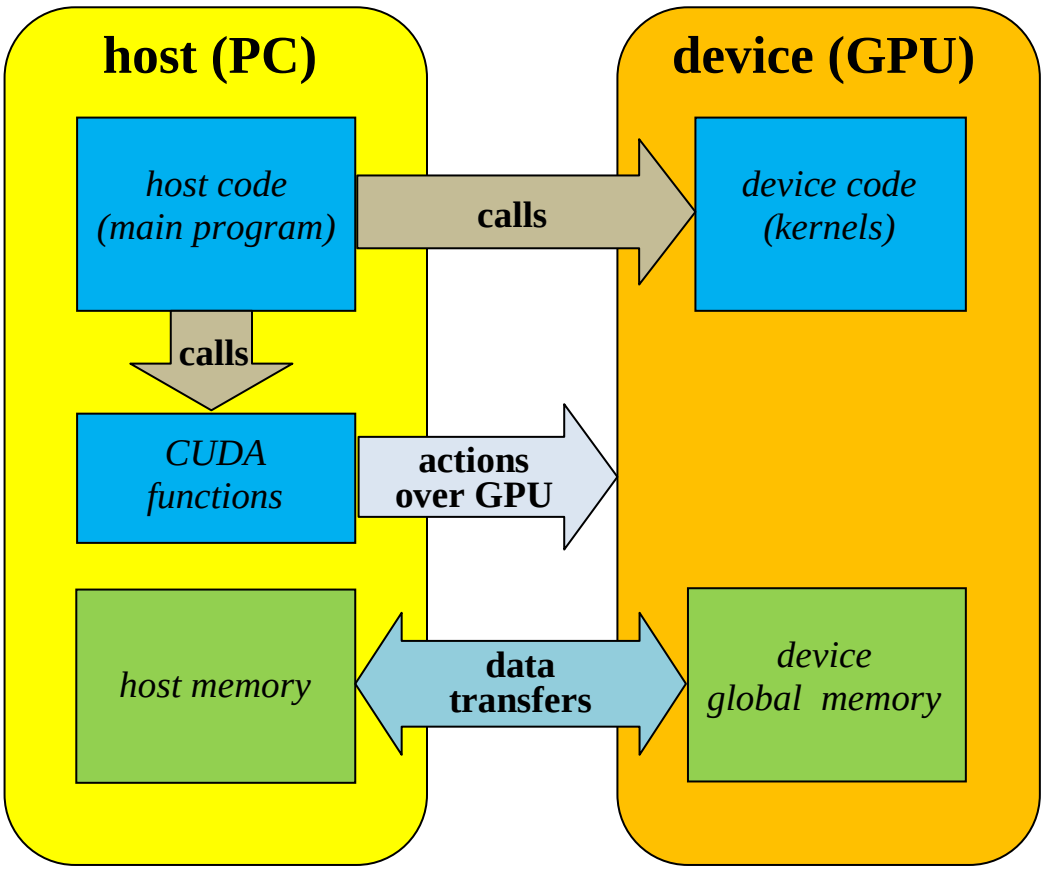
→ *extensions to C:*

__global__

__device__

dim3 blocks(12 , 2 , 4);
dim3 threads(10 , 10 , 10);
my_kernel <<< blocks , threads >>> (*arguments*);

- *the host code*
  - controls the device
  - launches the execution of the *device code*



CUDA functions:

CUDA functions are { called by the host code / executed by the host }

→ used to apply actions over the GPU:

- query how many devices are connected to the host
  → select one of them for calculations
- read and configure settings of the GPU
- memory allocations - deallocations inside device global memory
- data transfers between host memory and device global memory
- synchronization between host and device
- . . . . . .

note: the name of a CUDA function always starts with "**cuda**"

launching the execution of a kernel is NOT done with a CUDA function,
but with some specific syntax

if error during execution of a CUDA function, the host code continues to execute !
= *extremely dangerous !*

→ **ALWAYS** encapsulate a call to a CUDA function
inside a call to our little **CUDA_CALL** function (insert this definition in your host code):

```
void CUDA_CALL(char *call_name , cudaError_t error)
{   if (error != cudaSuccess)
    {  printf("\n %s for cuda call  %s" , cudaGetErrorString(error) , call_name);
       exit(0);  }  }
```

example:

CUDA_CALL("setting first GPU" , cudaSetDevice(device));

→ if the value of **device** does not correspond to a CUDA card installed on the PC,
then an error message is printed and the program terminates

Device settings

- *query how many devices are connected to the host:*

```
int deviceCount;

CUDA_CALL("how many devices?" , cudaGetDeviceCount(&deviceCount));
```

→ device index: **device** = 0 , 1 , 2 , ... , **deviceCount** - 1

a priori, the values of index **device** are attributed by CUDA to the graphic cards of the PC in an arbitrary manner

but, if only one graphic card is inside the PC, its **device** value is obviously **0**

- *select a device for calculations:*

several threads may run concurrently on the host

→ selection by the currently active host thread of the device of index **device**:

```
CUDA_CALL("set GPU" , cudaSetDevice(device));
```

(if no explicit selection, the device of index **device** = 0 is selected by default)

→ then, any kernel launched by this host thread will execute on the selected device

several host threads can use simultaneously the same device

a host thread can use only one device at a time
→ if several devices, you should define and run one host thread for each device

query the properties of a device:

```
cudaDeviceProp deviceProp;

CUDA_CALL("device prop." , cudaGetDeviceProperties(&deviceProp , device));
```

deviceProp.name	name of the graphic card, e.g. "C2050"
deviceProp.pciDeviceID	PCI Express device (slot) identifier on the motherboard
deviceProp.ECCEnabled	is ECC mode enabled? (error correction mode)
deviceProp.major	major revision number, i.e. 1 or 2
deviceProp.minor	minor revision number
deviceProp.clockRate	clock frequency (in kHz) of the cores
. . . . .	and more. . .

example: selecting the first GPU which happens to be a C2050

```
void set_GPU(int device)
{
   cudaDeviceProp deviceProp;
   CUDA_CALL("setting GPU" , cudaSetDevice(device));
   CUDA_CALL("device prop." , cudaGetDeviceProperties(&deviceProp , device));

   printf(  "\nthe device %d, GPU %s, is OK for CDM calculations. ECC is %s\n"
            , device , deviceProp.name , (deviceProp.ECCEnabled ? "ON" : "OFF"));
}

. . . . . .

int device , deviceCount;
cudaGetDeviceCount(&deviceCount);

for (device = 0 ; device < deviceCount ; device++)
{  cudaDeviceProp deviceProp;
   CUDA_CALL("device prop." , cudaGetDeviceProperties(&deviceProp , device));
   if (!strstr(deviceProp.name , "C2050"))
   {  set_GPU(device);    break;  }  }
```

Memory (de)allocation within the device global memory

- *allocate a bloc of memory of* **s** *bytes inside the device global memory:*

```
double *d_p;    size_t s;

CUDA_CALL("memory allocation" , cudaMalloc(&d_p , s));
```

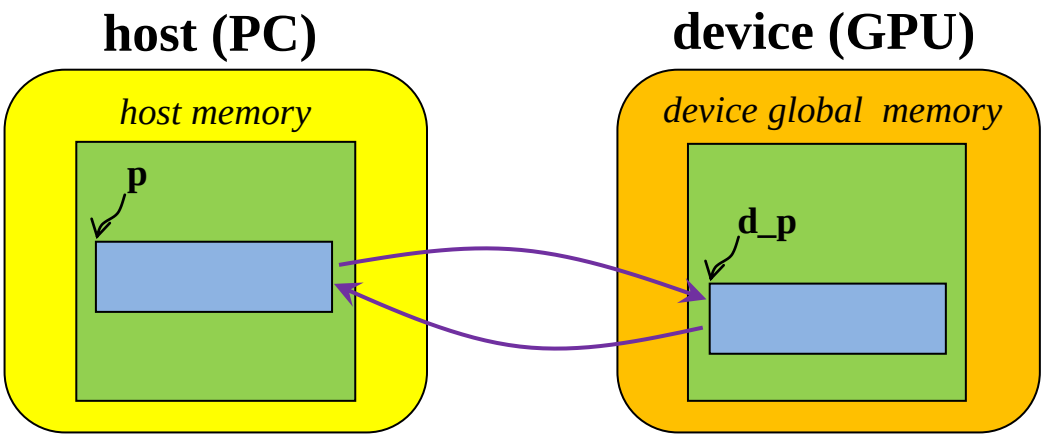
→ **d_p** now contains the start address of the new block in the global memory

cudaMalloc returns only an error code, and NOT a pointer like **malloc** does

- *deallocate a bloc of memory inside the device global memory:*

```
CUDA_CALL("deallocate" , cudaFree(d_p));
```

Data transfers between host memory and device global memory



- *transfer from host memory to device global memory:*

```
CUDA_CALL("host to device" , cudaMemcpy(d_p , p , s , cudaMemcpyHostToDevice));
```

- *transfer from device global memory to host memory:*

```
CUDA_CALL("device to host" , cudaMemcpy(p , d_p , s , cudaMemcpyDeviceToHost));
```

Data partitioning and kernel over this partition

grid of blocks of data elements:

partition the 1D, 2D or 3D set of data elements into a grid of blocks

→ over a certain dimension: { index of a block / index of an element of a block }

we will consider *absolute* indices (i.e., based on the initial indexation of the data elements) and not *relative* indices (i.e., based on the blocks themselves)

1D partition:

int n elements in all
int SB elements per block
⇒ *number of blocks*: **int nB = n / SB + (n % SB ? 1 : 0)**

if **n** divided exactly by **SB** → every block contains exactly **SB** elements
if **n** not divided exactly by **SB** → a block contains at most **SB** elements (either **SB** or **SB - 1** if **nB > SB**)

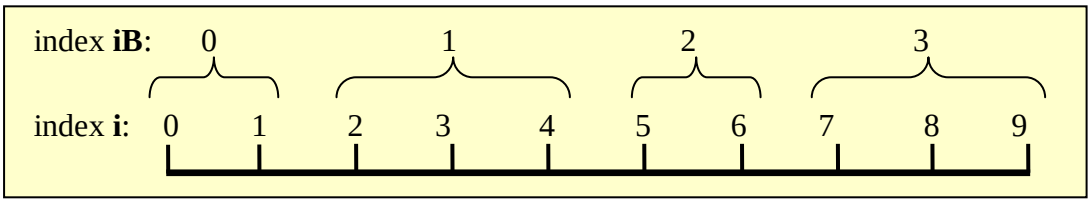
index of a block: **int iB** such that **0 ≤ iB < nB**

→ restriction to a segment of blocks: **iB0 ≤ iB < iB1** (absolute index, i.e. we do not set the index at 0 at the segment's beginning)

index of an element of block **iB**: **int i** such that **i0 ≤ i < i1** with { **int i0 = iB * n / nB**, **int i1 = (iB + 1) * n / nB** }

(absolute index, i.e. we do not set **i** at 0 at the block's beginning)

example: **n = 10 , SB = 3** ⇒ **nB = 3 + 1 = 4**



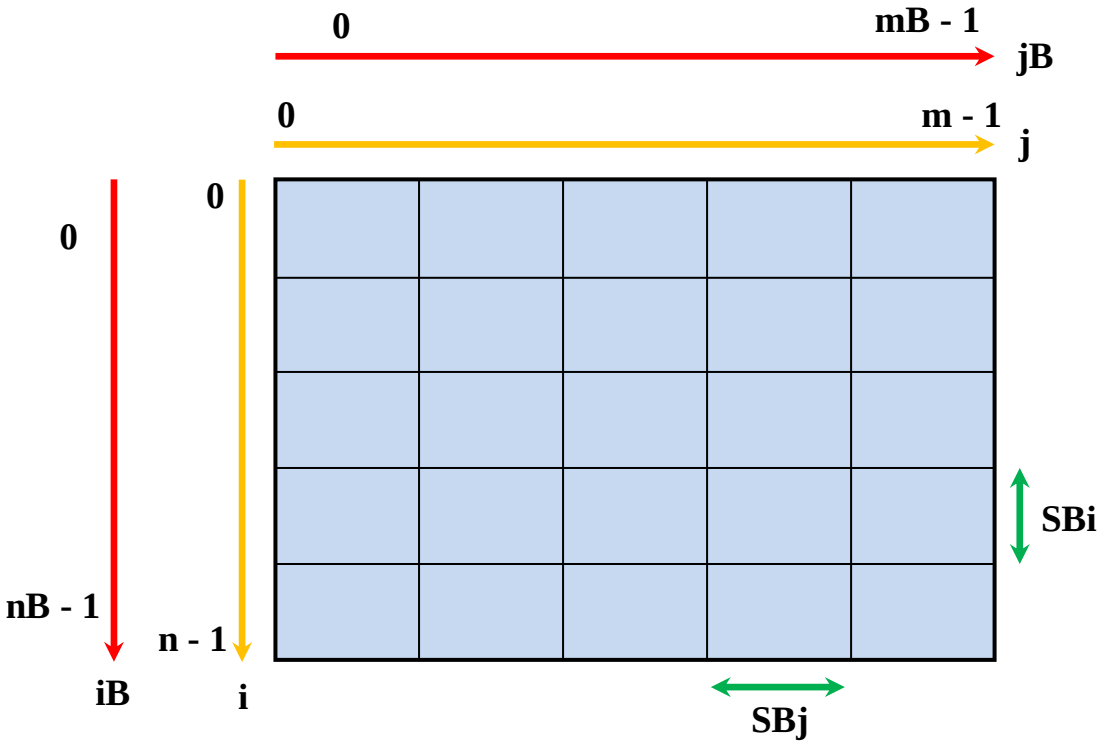
- the effect of truncature is distributed over the succession of blocks
- each block has at most **SB** elements

2D partition:

array of **n × m** data elements and blocks of size **SBi × SBj** (at most)

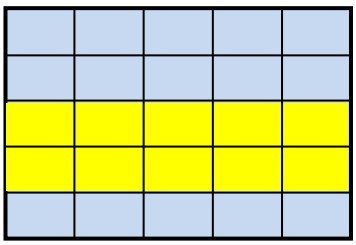
```
int nB  =  n  / SBi + (n  % SBi  ? 1 : 0)
int mB  =  m / SBj + (m % SBj ? 1 : 0)
```

indices of a block: **int iB , jB** such that **0 ≤ iB < nB** and **0 ≤ jB < mB**



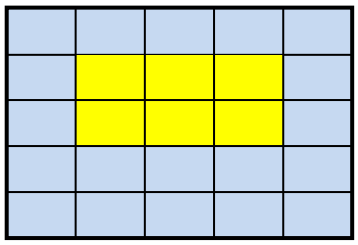
restriction to a *stripe of blocks*:

iB0 ≤ iB < iB1 and **0 ≤ jB < mB**



restriction to a *patch of blocks*:

iB0 ≤ iB < iB1 and **jB0 ≤ jB < jB1**



indices of an element of block **(iB , jB)**: **int i , j** such that **i0 ≤ i < i1** and **j0 ≤ j < j1**

with
```
int i0 = iB * n  / nB   ;  int i1  = (iB + 1) * n  / nB
int j0 = jB * m / mB  ;  int j1 = (jB + 1) * m / mB
```

- *3D partition:*

  array of **n × m × p** data elements and blocks of size **SBi × SBj × SBk**

  similar to 2D partition

- *size of a block:*

at most 1024 threads per block (for 2.0)

→ if 1D: **SB ≤ 1024**

→ if 2D: **SBi × SBj ≤ 1024** ($\mathbf{SB}^2 \leq \mathbf{1024} \Rightarrow \mathbf{SB} \leq \mathbf{32}$)

→ if 3D: **SBi × SBj × SBk ≤ 1024** ($\mathbf{SB}^3 \leq \mathbf{1024} \Rightarrow \mathbf{SB} \leq \mathbf{10}$)

- in theory, the block size should be as big as possible to "overflow" the 32 cores of the multiprocessor
- in practice, you should adjust the size empirically

  →case of matrix calculations: **SB = 27** gives fastest code, and not **32**

- *size of the grid:*

at most 65536 blocks per grid dimension

built-in variables:

variables **gridDim** , **blockDim** , **blockIdx** , **threadIdx** can be read inside a kernel

do not write into these variables! they are provided as such by CUDA

read them only inside a kernel

- *dimension-related variables:*

gridDim.x , **gridDim.y** , **gridDim.z**	→ sizes of the patch of blocks actually selected
blockDim.x , **blockDim.y** , **blockDim.z**	→ sizes of the block (equal to **SBi** , **SBj** , **SBk**)

note: use only **.x** for 1D and **.x** and **.y** for 2D

we will not use these Dim-like variables

- *index-related variables:*

blockIdx.x , **blockIdx.y** , **blockIdx.z**	→ relative indices within patch of blocks
threadIdx.x , **threadIdx.y** , **threadIdx.z**	→ relative indices within block

- *which approach to the index issue?*

the classic approach (see CUDA Programming Guide) focuses on "relative indices"
→ its problems:
  - in general, **gridDim.x** is NOT **nB**, i.e. a relative index is NOT an absolute index
    → *very confusing* when it comes to actual programming
  - truncature effects are concentrated on the last block (may then have only one thread...)

our approach offers a clean framework for general cases

note: we will not use **gridDim** and **blockDim**
but we will have to pass **n** and **nB** (and **iB0**) as arguments to the kernel

1D partition:

- *run the kernel on all the blocks of the data partition:* **0 ≤ iB < nB**

kernel call:

```
int nB  =  n / SB + (n % SB ? 1 : 0)

my_kernel <<< nB , SB >>> (n , nB ,  . . . );
```

partition

kernel def.:

```
__global__ void my_kernel(int n , int nB ,  . . . )
{
   int iB = blockIdx.x;    int i0 = iB * n / nB , i1 = (iB + 1) * n / nB;
   int i = i0 + threadIdx.x;

   if (i < i1)
   {
      . . . . . . .
   }
}
```

relative index to absolute index

because of truncature...

parallel code

- *run the kernel only on a segment of blocks of the data partition:* **iB0 ≤ iB < iB1**

kernel call:

```
int nB  =  n / SB + (n % SB ? 1 : 0)

my_kernel <<< iB1 - iB0 , SB >>> (n , nB , iB0 ,  . . . );
```

kernel def.:

```
__global__ void my_kernel(int n , int nB , int iB0 , . . .)
{
   int iB = iB0 + blockIdx.x;    int i0 = iB * n / nB , i1 = (iB + 1) * n / nB;
   int i = i0 + threadIdx.x;

   if (i < i1)
   {
      . . . . . . .
   }
}
```

relative index to absolute index

2D partition:

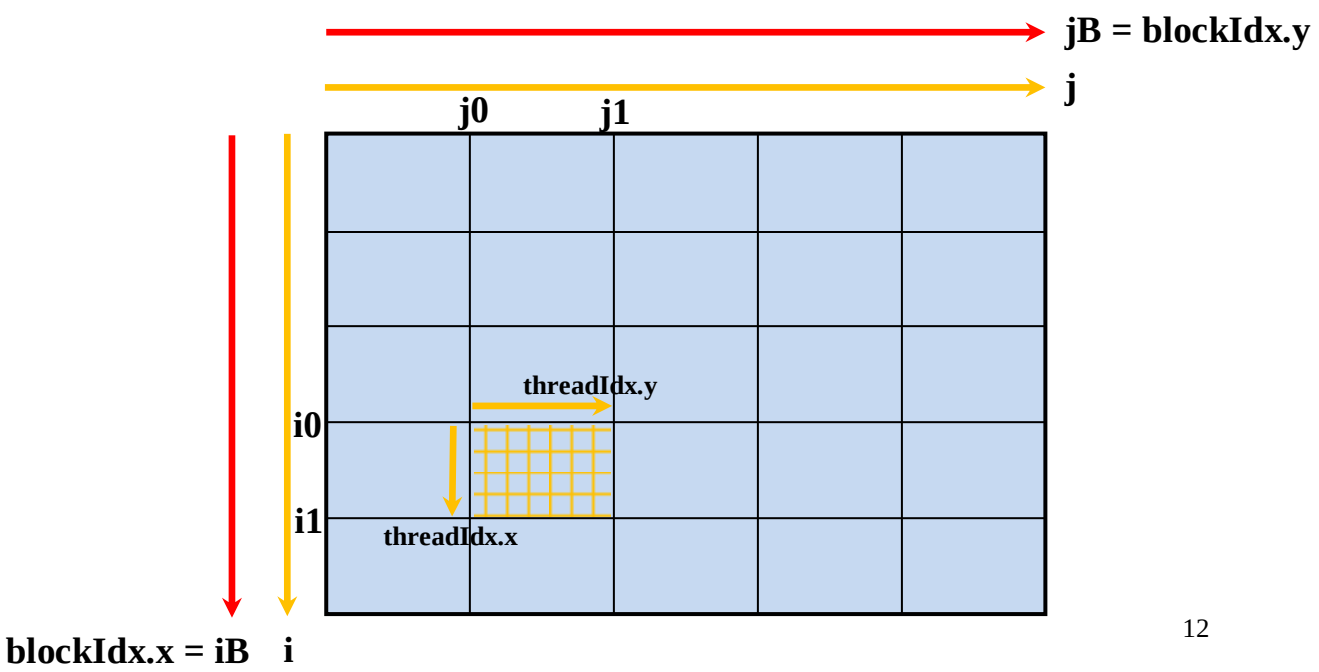
- *run the kernel on all the blocks of the data partition:* **$0 \le iB < nB$** and **$0 \le jB < mB$**

kernel call:

```
int nB  =  n / SBi + (n % SBi ? 1 : 0);
int mB  =  m / SBj + (m % SBj ? 1 : 0);

dim3 blocks(nB , mB);
dim3 threads(SBi , SBj);

my_kernel <<< blocks , threads >>> (n , m , nB , mB ,  . . . );
```

kernel def.:

```
__global__ void my_kernel(int n , int m , int nB , int mB , . . .)
{
   int iB  = blockIdx.x;     int i0 = iB  * n / nB  ,  i1 = (iB + 1) * n / nB;
   int i  = i0 + threadIdx.x;

   int jB  = blockIdx.y;     int j0 = jB  * m / mB  ,  j1 = (jB + 1) * m / mB;
   int j  = j0 + threadIdx.y;

   if (i < i1  &&  j < j1)
   {
      . . . . . . .
   }
}
```



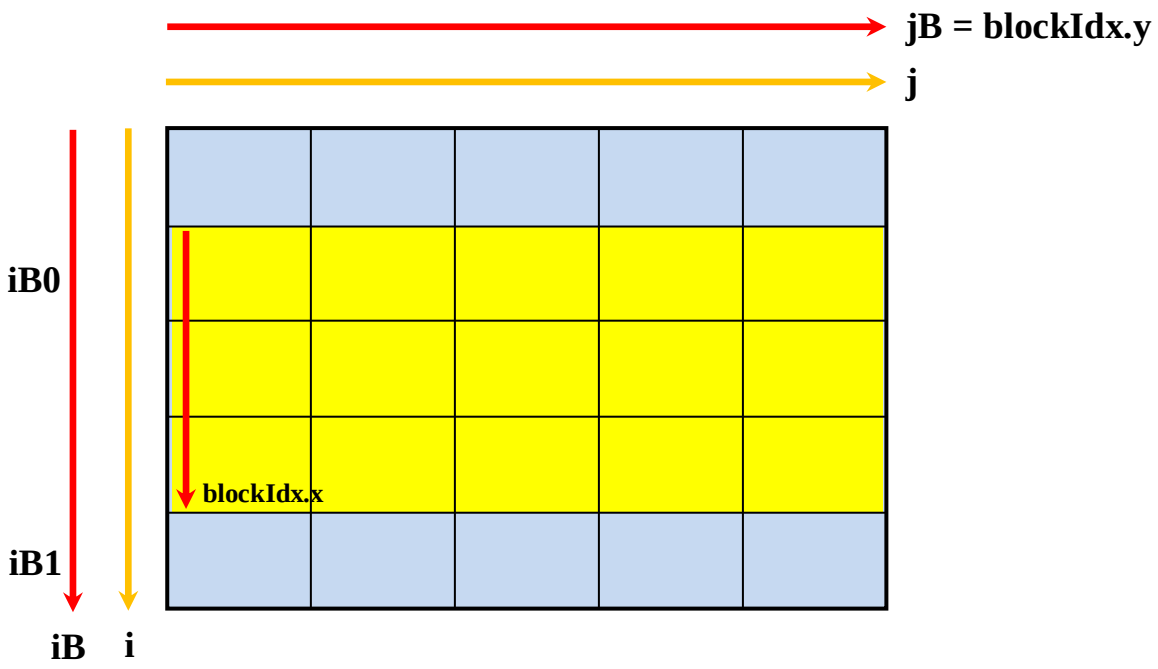
- *run the kernel on a stripe of blocks of the data partition:* **iB0 ≤ iB < iB1** and **0 ≤ jB < mB**
  (similar case with restriction over **jB**)

kernel call:

```
int nB  =  n / SBi + (n % SBi ? 1 : 0);
int mB  =  m / SBj + (m % SBj ? 1 : 0);

dim3 blocks(iB1 - iB0 , mB);
dim3 threads(SBi , SBj);

my_kernel <<< blocks , threads >>> (n , m , nB , mB , iB0 , . . .);
```

kernel def.:

```
__global__ void my_kernel(int n , int m , int nB , int mB , int iB0 , . . .)
{
   int iB  = iB0 + blockIdx.x;     int i0 = iB  * n / nB  ,  i1 = (iB + 1) * n / nB;
   int i  = i0 + threadIdx.x;

   int jB  = blockIdx.y;     int j0 = jB  * m / mB  ,  j1 = (jB + 1) * m / mB;
   int j  = j0 + threadIdx.y;

   if (i < i1  &&  j < j1)
   {
      . . . . . . .
   }
}
```



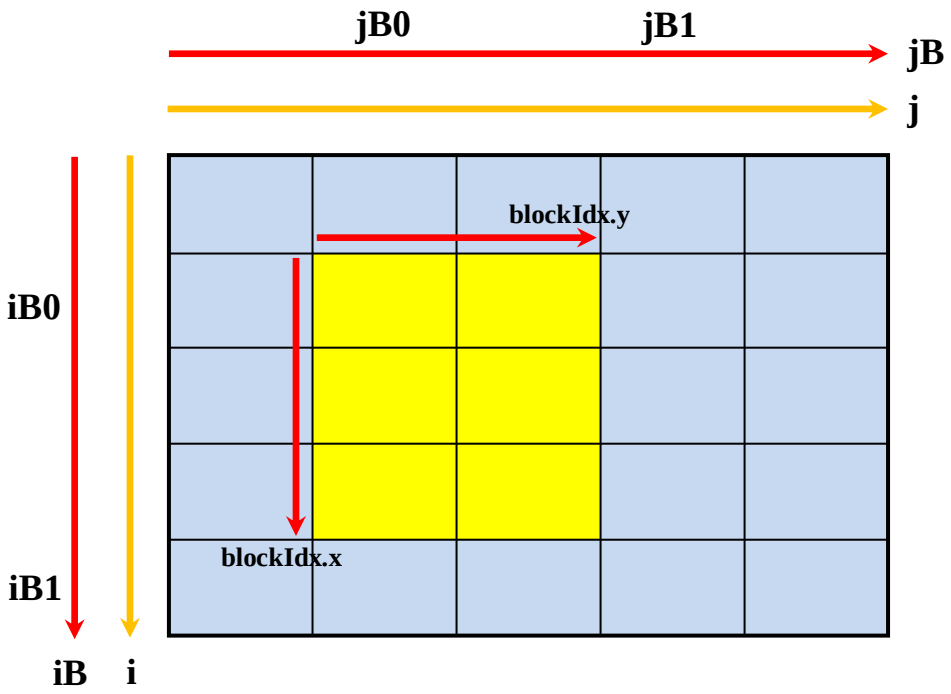
- *run the kernel on a patch of blocks of the data partition:* **iB0 ≤ iB < iB1** and **jB0 ≤ jB < jB1**

kernel call:

```
int nB  =  n / SBi + (n % SBi ? 1 : 0);
int mB  =  m / SBj + (m % SBj ? 1 : 0);

dim3 blocks(iB1 - iB0 , jB1 - jB0);
dim3 threads(SBi , SBj);

my_kernel <<< blocks , threads >>> (n , m , nB , mB , iB0 , jB0 ,  . . .);
```

kernel def.:

```
__global__ void my_kernel(int n , int m , int nB , int mB , int iB0 , int jB0 , ...)
{
   int iB  = iB0 + blockIdx.x;    int i0 = iB  * n / nB  ,  i1 = (iB + 1) * n / nB;
   int i  = i0 + threadIdx.x;

   int jB  = jB0 + blockIdx.y;    int j0 = jB  * m / mB  ,  j1 = (jB + 1) * m / mB;
   int j  = j0 + threadIdx.y;

   if (i < i1  &&  j < j1)
   {
      . . . . . . .
   }
}
```



Synchronization

synchronization between host and device:

by default, no synchronization between *host thread* and *device threads*

but, in general, we want the host thread to launch the kernel
and then wait for all the threads to terminate

- → two ways to define a *meeting point* inside the host code:

  - after kernel call: **CUDA_CALL("meeting point" , cudaThreadSynchronize());**

  or

  - after kernel call, make a call to any CUDA function

    (*note:* often, it will be a call to **cudaMemcpy** to transfer results to the host memory)

synchronization between threads of a block:

- inside the kernel code, define a meeting point between all the threads of a block:

__syncthreads();

→ will be used when we want to optimize the kernel by means of the shared memory

__syncthreads(); can be used *only* inside the kernel code and applies *only* within each block
→ it is NOT a "CUDA function"

__syncthreads(); can be used inside an **if** statement provided that
the test of this **if** has the same value for all the threads of the block !
else, strange and unpredictable behavior...

synchronization between any threads with atomic functions:

atomic function = *mutually exclusive* operation over a variable in device global memory
or in a multiprocessor shared memory

→ can be executed concurrently by any threads of any blocks
without any risk of corrupting the variable

can be used *only* inside functions executed in the core (kernels and **__device__** functions)

can be used without restrictions *only* on 2.x

- *adds* **v** *to the value of* ***p** *, then returns the old value of* ***p** *:*

```
int atomicAdd(int *p , int v);
```

```
float atomicAdd(float *p , float v);
```

- *substracts* **v** *to the value of* ***p** *, then returns the old value of* ***p** *:*

```
int atomicSub(int *p , int v);
```

- *stores* **v** *inside* ***p** *, then returns the old value of* ***p** *:*

```
int atomicExch(int *p , int v);
```

```
float atomicExch(float *p , float v);
```

- *stores inside* ***p** *the min / max of* **v** *and* ***p** *, then returns the old value of* ***p** *:*

```
int atomicMin(int *p , int v);
```

```
int atomicMax(int *p , int v);
```

Running several kernels concurrently

- *launching concurrent kernels* from a *single host thread*:

  succession of kernel launches in different *streams*

→ how to launch a kernel within a stream:

```
cudaStream_t  my_stream;

cudaStreamCreate(&my_stream);

. . . . . . .

my_kernel <<< nB , SB , 0 , my_stream >>> (n , nB);

. . . . . . .

cudaStreamDestroy(my_stream);
```

no communication possible between these concurrent kernels

only for 2.x

at most 16 concurrent kernels (with 2.x)

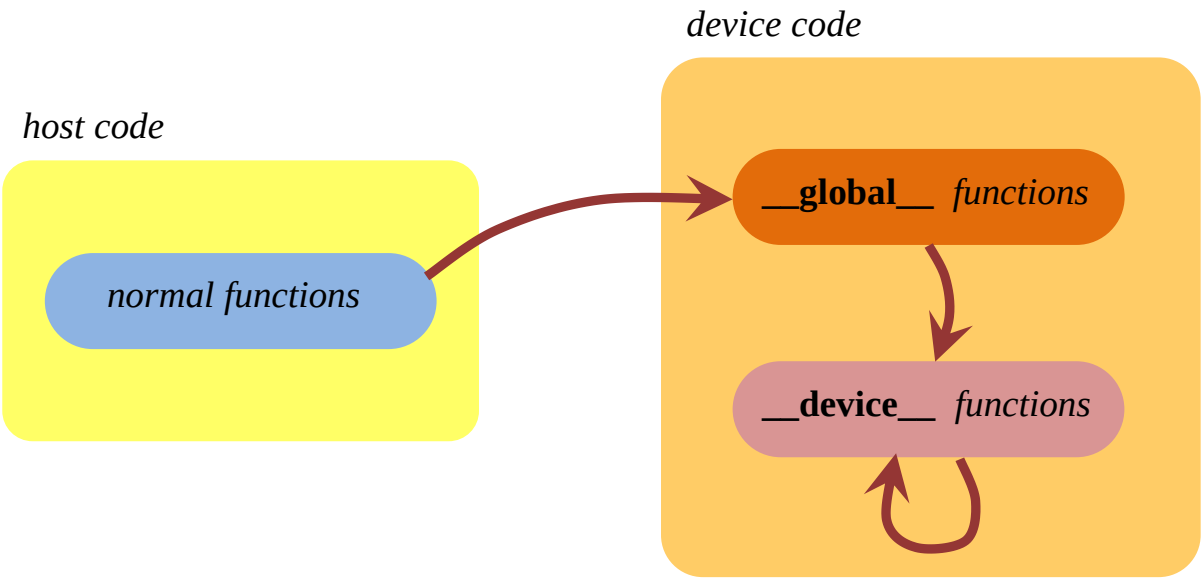
Function calls within the device

- *three sorts of function definitions in our program:*

normal function definition → executed on *host*, callable from *host* only
= *part of the host code*

function definition with **__global__** → executed on *device*, callable from *host* only
= *kernel*
= *part of the device code*

function definition with **__device__** → executed on *device*, callable from *device* only
(calls starting from a kernel)
= *part of the device code*



- *recursion within the device:*

kernels cannot be recursive (only the host can call them, i.e. not themselves)

__device__ functions can be recursive (for 2.x)

- *pointers and dynamic allocation inside the device code:*

malloc and **free** inside the codes of the kernels and of the **__device__** functions:

```
void *malloc(size_t);
```

```
void free(void *);
```

without restrictions only for 2.x

→ several possible programming approaches:

- may be performed by each of the threads

<u>or</u>

- may be performed by just one thread per block:

if (threadIdx.x == 0) → conditional **malloc** and **free**
only for the first thread of each block

<u>or</u>

- may be performed by just one thread over all the blocks:

if (blockIdx.x == 0 && threadIdx.x == 0) → conditional **malloc** and **free**
only for the first thread of the first block

→ the allocation takes places inside the *device heap* which is within the *device global memory*

the device heap is used only for on-device dynamic allocations, NOT for **cudaMalloc()**
(these two sorts of allocations both take place in the global memory but are totally separate)

device heap size = 8 MB by default

set the device heap size by calling from host (just once before first call of a kernel with **malloc**):

```
CUDA_CALL("heap size" , cudaThreadSetLimit(cudaLimitMallocHeapSize , size));
```

Compiling a CUDA code with nvcc

- *include files:*

#include <cuda.h> → *CUDA functions*

#include <cublas.h> → *CUBLAS functions*

- *presentation:*

version used: **nvcc3.2**

download and install **nvcc3.2** from www.nvidia.com → downloads

installed on **Windows 7 64 bits** → requires **Microsoft Visual Studio 9.0 64 bits**

nvcc3.2 does not work with **Visual Studio 2010** !

nvcc compiler will use **cl** compiler of **Visual C++**

the file containing the CUDA code must have extension **.cu**

- *options of nvcc command line compiler:*

-O → optimize

-c → compile only (produce **.obj** file)

-o → specify output file

-arch sm_20 → specify that the architecture has compute capability 2.0

→ allow
- code optimizations specifically for this architecture
- **printf** inside the kernels
- variables of type **double**

-Xcompiler → pass an option to **cl** compiler

→
- **-Xcompiler -O2** → **cl** must optimize for speed
- **-Xcompiler -F4096** → **cl** must set C stack size as 4096 B (for host code)
- **-Xcompiler -MT** → **cl** must allow multithreading (for host code)

*example: compiling a code stored in one file only (**P.cu**)*

```
call "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat"

"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\bin\nvcc"
                                    -O   -arch  sm_20   -Xcompiler  -O2   P.cu   -o   P.exe
```

example: compiling a code stored in two files
*(**P.cpp** normal C code with main // **C.cu** CUDA code)*

```
call "C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat"

"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\bin\nvcc"
                                -O   -arch  sm_20   -Xcompiler  -O2   C.cu   -c   -o   C.obj

"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v3.2\bin\nvcc"
                                -O   -Xcompiler  -O2   P.cpp   C.obj   -o   P.exe
```

Examples - host codes and device codes

example: applying a polynomial (coef. $\mathbf{A_{p-1}}$ *, ... ,* $\mathbf{A_0}$*) over each element of an array* **X** *(size* **n***)*

```
#define SB  1024
////------------------------------------- HOST CODE -------------------------------------
void pol_vect(double *Y , double *X , double *A , int n , int p)
{
   int nB   =  n  / SB + (n  % SB ? 1 : 0);

   double *d_A;   size_t sA = sizeof(double) * p;   CUDA_CALL("A" , cudaMalloc(&d_A , sA));
   double *d_X;   size_t sX = sizeof(double) * n;   CUDA_CALL("X" , cudaMalloc(&d_X , sX));
   double *d_Y;   size_t sY = sizeof(double) * n;   CUDA_CALL("Y" , cudaMalloc(&d_Y , sY));

   CUDA_CALL("d_A <-- A" , cudaMemcpy(d_A , A , sA , cudaMemcpyHostToDevice));
   CUDA_CALL("d_X <-- X" , cudaMemcpy(d_X , X , sX , cudaMemcpyHostToDevice));

   kernel_pol_vect <<< nB , SB >>> (d_Y , d_X , d_A , n , p , nB);

   CUDA_CALL("Y <-- d_Y" , cudaMemcpy(Y , d_Y , sY , cudaMemcpyDeviceToHost));

   CUDA_CALL("free d_A" , cudaFree(d_A));
   CUDA_CALL("free d_X" , cudaFree(d_X));
   CUDA_CALL("free d_Y" , cudaFree(d_Y));
}


////------------------------------------- DEVICE CODE -------------------------------------
__global__ void kernel_pol_vect(double *Y , double *X , double *A , int n , int p , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;     int i = i0 + threadIdx.x;
   int k;

   if (i < i1)
   {
      double y = A[p] , x = X[i];
      for (k = p - 1 ; k >= 0 ; k--)
         y = y * x + A[k];
      Y[i] = y;
   }
}
```

example: image filtering: image **X(n×n)** → *filtered image* **Y(n×n)**

```
#define SB 25
////------------------------------------- HOST CODE -------------------------------------
void filter_image(double *Y , double *X , int n)
{
   int nB   =  n  / SB + (n  % SB ? 1 : 0);

   double *d_X; size_t sX = sizeof(double) * n * n; CUDA_CALL("X",cudaMalloc(&d_X,sX));
   double *d_Y; size_t sY = sizeof(double) * n * n; CUDA_CALL("Y",cudaMalloc(&d_Y,sY));

   CUDA_CALL("d_X <-- X" , cudaMemcpy(d_X , X , sX , cudaMemcpyHostToDevice));

   dim3 blocks(nB , nB);
   dim3 threads(SB , SB);
   kernel_filter_image <<< blocks , threads >>> (d_Y , d_X , n , nB);

   CUDA_CALL("Y <-- d_Y" , cudaMemcpy(Y , d_Y , sY , cudaMemcpyDeviceToHost));

   CUDA_CALL("free d_X" , cudaFree(d_X));
   CUDA_CALL("free d_Y" , cudaFree(d_Y));
}


////------------------------------------- DEVICE CODE -------------------------------------
__global__ void kernel_filter_image(double *Y , double *X , int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n / nB ,  i1 = (iB + 1) * n / nB;    int i = i0 + threadIdx.x;
   int jB = blockIdx.y;    int j0 = jB * n / nB , j1 = (jB + 1) * n / nB;    int j = j0 + threadIdx.y;
   int ii , ii0 , ii1 , jj , jj0 , jj1;    int q;

   if (i < i1  && j < j1)
   {
      Mij(Y , n , i , j) = 0;    q = 0;

      ii0 = max(i - 1 , 0);     ii1 = min(i + 1 , n - 1);
      jj0 = max(j - 1 , 0);    jj1 = min(j + 1 , n - 1);
      for (ii = ii0 ; ii <= ii1 ; ii++)  for (jj = jj0 ; jj <= jj1 ; jj++)
      {  Mij(Y , n , i , j) += Mij(X , n , ii , jj);    q++;  }

      Mij(Y , n , i , j) /= q;
   }
}
```

example: *matrix multiplication:* **C(n×m) = A(n×p) × B(p×m)**

```
for (k = 0 ; k < p ; k++)
   for (i = 0 ; i < n ; i++)
      for (j = 0 ; j < m ; j++)
         Mij(C , m , i , j) += Mij(A , p , i , k) * Mij(B , m , k , j);
```

replace each loop with two nested loops:
the outer on the block index
and the inner on the index within the block

```
nB  =  n  / SB + (n  % SB ? 1 : 0);
mB  =  m / SB + (m % SB ? 1 : 0);
pB  =  p  / SB + (p  % SB ? 1 : 0);

for (kB = 0 ; kB < pB ; kB++)
{  k0 = kB * p / pB;    k1 = (kB + 1) * p / pB;
   for (k = k0 ; k < k1 ; k++)

   for (iB = 0 ; iB < nB ; iB++)
   {  i0 = iB * n   / nB;     i1 = (iB + 1) * n   / nB;
      for (i = i0 ; i < i1 ; i++)

      for (jB = 0 ; jB < mB ; jB++)
      {  j0 = jB * m / mB;    j1 = (jB + 1) * m / mB;
         for (j = j0 ; j < j1 ; j++)

            Mij(C , m , i , j) += Mij(A , p , i , k) * Mij(B , m , k , j);  }  }  }
```

reorganize the loops

```
for (kB = 0 ; kB < pB ; kB++)
{  k0 = kB * p / pB;    k1 = (kB + 1) * p / pB;

   for (iB = 0 ; iB < nB ; iB++)
      for (jB = 0 ; jB < mB ; jB++)
      {  i0 = iB * n   / nB;     i1 = (iB + 1) * n   / nB;
         j0 = jB * m / mB;    j1 = (jB + 1) * m / mB;

         for (i = i0 ; i < i1 ; i++)
            for (j = j0 ; j < j1 ; j++)
               for (k = k0 ; k < k1 ; k++)

                  Mij(C , m , i , j) += Mij(A , p , i , k) * Mij(B , m , k , j);  }  }
```

parallel execution: grid of blocks

parallel execution: block of threads

```
#define SB 27

#define Mij(M , srow , i , j)  (*((M) + (i) * (srow) + (j)))

////---------------------------------------- HOST CODE ----------------------------------------
void mul_mat(double *C , double *A , double *B , int n , int m , int p)
{
   for (int i = 0 ; i < n ; i++)  for (int j = 0 ; j < m ; j++)  Mij(C , m , i , j) = 0;

   int nB   =  n  / SB + (n  % SB ? 1 : 0);
   int mB  =  m / SB + (m % SB ? 1 : 0);
   int pB   =  p  / SB + (p  % SB ? 1 : 0);

   double *d_A;  size_t sA = sizeof(double) *n*p;   CUDA_CALL("A", cudaMalloc(&d_A,sA));
   double *d_B;  size_t sB = sizeof(double) *p*m;  CUDA_CALL("B", cudaMalloc(&d_B,sB));
   double *d_C;  size_t sC = sizeof(double) *n*m;  CUDA_CALL("C", cudaMalloc(&d_C,sC));

   CUDA_CALL("d_A <-- A" , cudaMemcpy(d_A , A , sA , cudaMemcpyHostToDevice));
   CUDA_CALL("d_B <-- B" , cudaMemcpy(d_B , B , sB , cudaMemcpyHostToDevice));
   CUDA_CALL("d_C <-- C" , cudaMemcpy(d_C , C , sC , cudaMemcpyHostToDevice));

   dim3 blocks(nB , mB);
   dim3 threads(SB , SB);

   for (int kB = 0 ; kB < pB ; kB++)
   {  int k0 = kB * p / pB , k1 = (kB + 1) * p / pB;
      kernel_mul_mat <<< blocks , threads >>> (d_C , d_A , d_B , n , m , p , nB , mB , k0 , k1);
      CUDA_CALL("meeting point" , cudaThreadSynchronize());  }

   CUDA_CALL("C <-- d_C" , cudaMemcpy(C , d_C , sC , cudaMemcpyDeviceToHost));

   CUDA_CALL("free A" , cudaFree(d_A));
   CUDA_CALL("free B" , cudaFree(d_B));
   CUDA_CALL("free C" , cudaFree(d_C));
}
```

```
////---------------------------------------- DEVICE CODE ----------------------------------------
__global__ void kernel_mul_mat(  double *C , double *A , double *B , int n , int m , int p
                                                    , int nB, int mB , int k0 , int k1)
{
   int iB = blockIdx.x;    int i0 = iB * n   / nB  , i1 = (iB + 1) * n   / nB;     int i = i0 + threadIdx.x;
   int jB = blockIdx.y;    int j0 = jB * m / mB , j1 = (jB + 1) * m / mB;    int j = j0 + threadIdx.y;
   int k;

   if (i < i1  &&  j < j1)
   {
      for (k = k0 ; k < k1 ; k++)
         Mij(C , m , i , j) += Mij(A , p , i , k) * Mij(B , m , k , j);
   }
}
```

example: atomic operation - synchronization between all threads of a kernel

*→ each thread increments a single variable **C** in global memory*

```
#define SB  1024
////---------------------------------------- HOST CODE ----------------------------------------
void counter_threads(int *C , int n)
{
   int nB   =  n  / SB + (n  % SB ? 1 : 0);

   int *d_C;    size_t sC = sizeof(int);    CUDA_CALL("malloc d_C" , cudaMalloc(&d_C , sC));

   CUDA_CALL("d_C <-- C" , cudaMemcpy(d_C , C , sC , cudaMemcpyHostToDevice));

   kernel_counter_threads <<< nB , SB >>> (d_C , n , nB);

   CUDA_CALL("C <-- d_C" , cudaMemcpy(C , d_C , sC , cudaMemcpyDeviceToHost));

   CUDA_CALL("free d_C" , cudaFree(d_C));
}


////---------------------------------------- DEVICE CODE ----------------------------------------
__global__ void kernel_counter_threads(int *C , int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;     int i = i0 + threadIdx.x;

   if (i < i1)
   {
      atomicAdd(C , 1);        // and NOT   *C += 1;   ! ! !
   }
}
```

example: running several threads concurrently

```
#define SB  10
////------------------------------------- HOST CODE -------------------------------------
void several_threads(int n)
{
   int nB   =  n  / SB + (n  % SB ? 1 : 0);
   cudaStream_t  stream_A , stream_B;
   cudaStreamCreate(&stream_A);    cudaStreamCreate(&stream_B);

   kernel_A <<< nB , SB , 0 , stream_A >>> (n , nB);
   kernel_B <<< nB , SB , 0 , stream_B >>> (n , nB);

   cudaStreamDestroy(stream_A);    cudaStreamDestroy(stream_B);
   CUDA_CALL("force flushing of the output buffer" , cudaThreadSynchronize());
}


////------------------------------------- DEVICE CODE -------------------------------------
__global__ void kernel_A(int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;     int i = i0 + threadIdx.x;
   if (i < i1)
      printf("A");
}

__global__ void kernel_B(int n , int nB)
{
   int iB = blockIdx.x;    int i0 = iB * n  / nB  , i1 = (iB + 1) * n  / nB;     int i = i0 + threadIdx.x;
   if (i < i1)
      printf("B");
}
```

→ *what is printed in the shell*:

```
AAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBBBBBBBBBBBBB
BBBBBBBBAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBB
BBBBBBAAAAAAAAAABBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABBBBBBBBBBAAAAAAAAAAAAAAAAAAAABBBBBB
BBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAA
ABBBBBBBBBBAAAAAAAAAABBBBBBBBBBAAAAAAAAAABBBBBBBBBA
AAAAAAAAAAAAAAAAAAABBBBBBBBBBAAAAAAAAAA
```