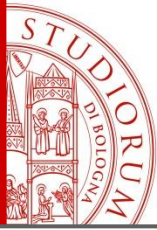


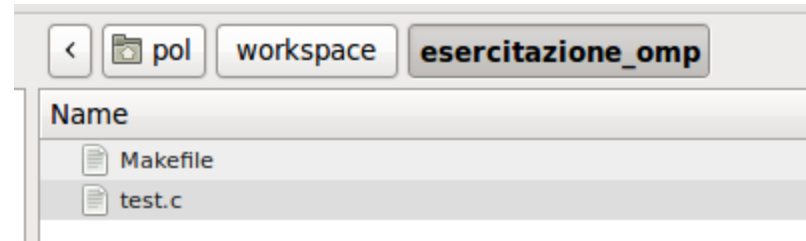
# OpenMP EXERCISE

Paolo Burgio  
paolo.burgio@unibo.it

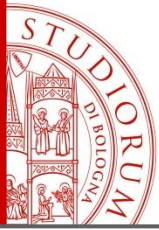


# HOWTO-run

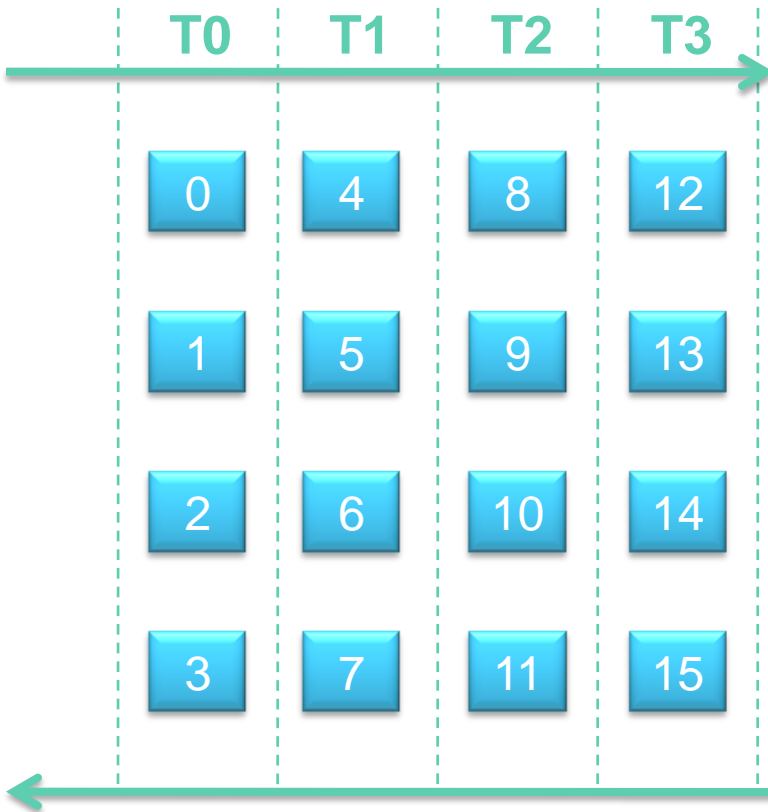
- ✓ What's on the package  
All tests are in file **test.c**



- ✓ To run, in function main(), uncomment the code for the test you want to run  
**\$make clean all run**
- ✓ **\$make clean** -> clear the workspace
- ✓ **\$make all disassemble** -> see disassembled code



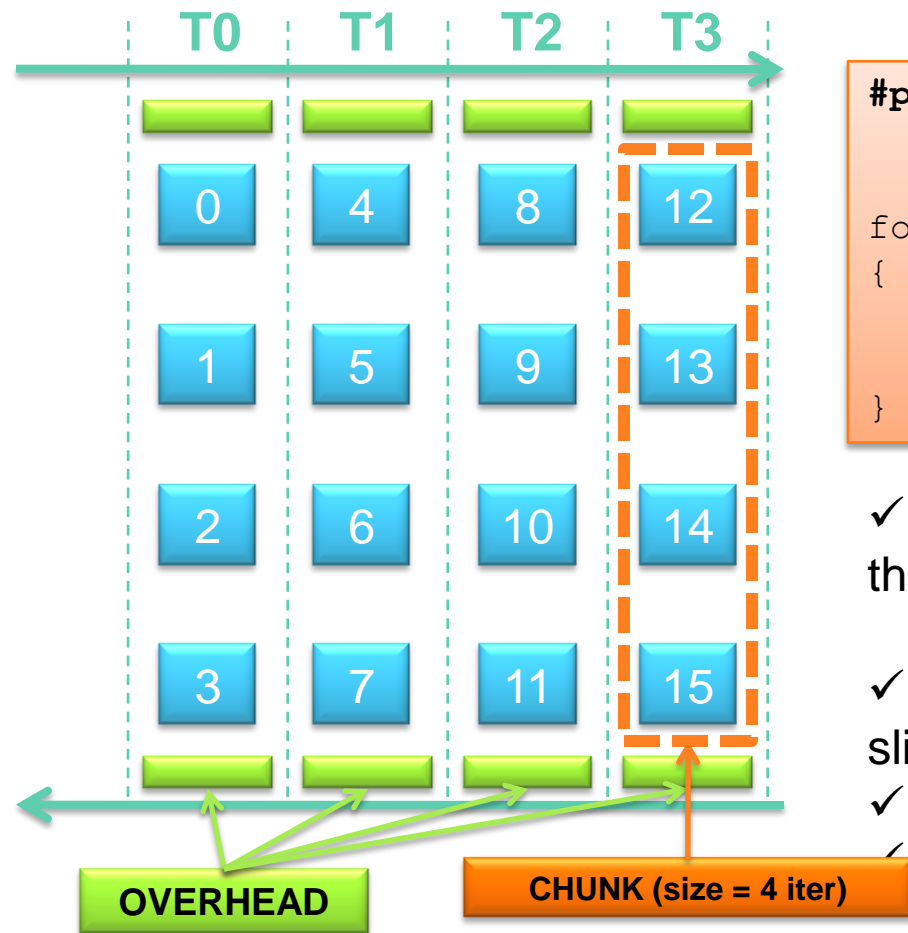
# EX 1 - Loop partitioning – Static scheduling



```
#pragma omp parallel for \  
    num_threads (4) schedule (static)  
for (uint i=0; i<16; i++)  
{  
    /* BALANCED LOOP CODE */  
}  
/* (implicit) SYNCH POINT */
```

- ✓ Iterations are statically assigned to threads
  - ✓  $16 \text{ iter} / 4 \text{ threads} = 4 \text{ iter/thread}$
- ✓ No overhead, simple adjustment of loop indexes according to thread ID
- ✓ Optimal scheduling if iworkload is balanced

# EX 1 - Loop partitioning – Dynamic scheduling

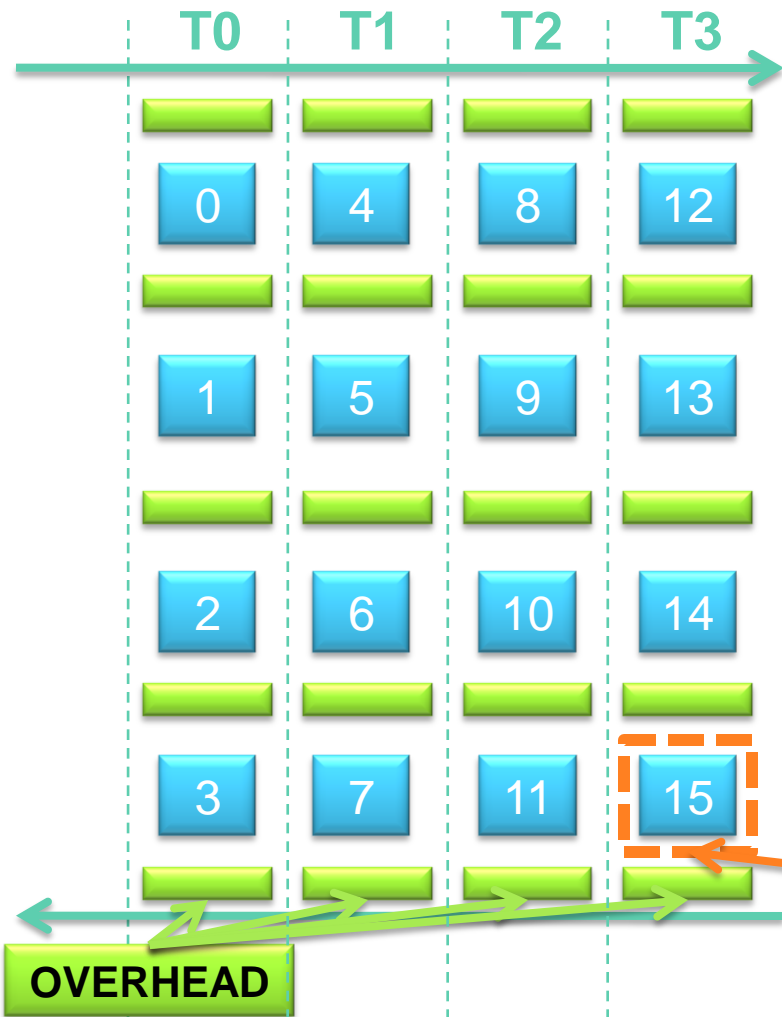


```
#pragma omp parallel for \
    num_threads (4) schedule (dynamic, 4)

for (uint i=0; i<16; i++)
{
    /* BALANCED LOOP CODE */
} /* (implicit) SYNCH POINT */
```

- ✓ Iterations are dynamically assigned to threads
  - ✓ 16 iter, 4 by 4
- ✓ Same allocation of iteration as static (prev. slide)
- ✓ Coarse granularity
- ✓ Overhead only at beginning and end

# EX 1 - Loop partitioning – Dynamic scheduling



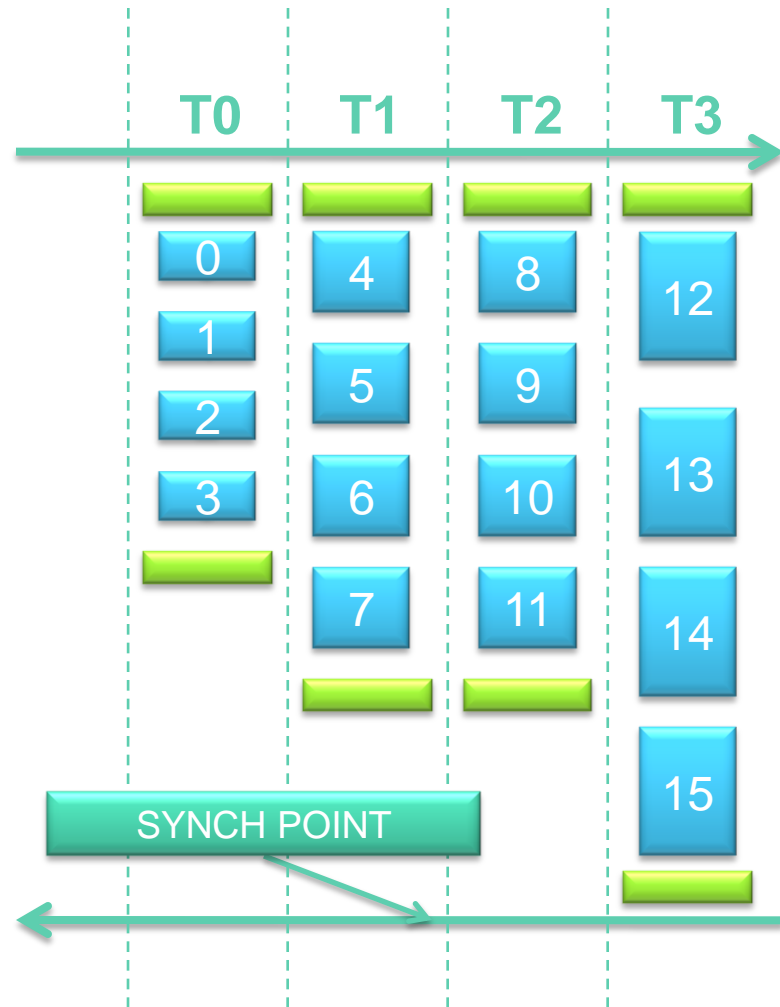
```
#pragma omp parallel for \
    num_threads (4) schedule (dynamic, 1)

for (uint i=0; i<16; i++)
{
    /* BALANCED LOOP CODE */
} /* (implicit) SYNCH POINT */
```

- ✓ Iterations are dynamically assigned to threads
  - ✓ 16 iter, 1 by 1
- ✓ Finest granularity possible
- ✓ Overhead at every iteration
- ✓ Worst performance under balanced workloads

**CHUNK (size = 1 iter)**

# EX 2 - Unbalanced loop partitioning

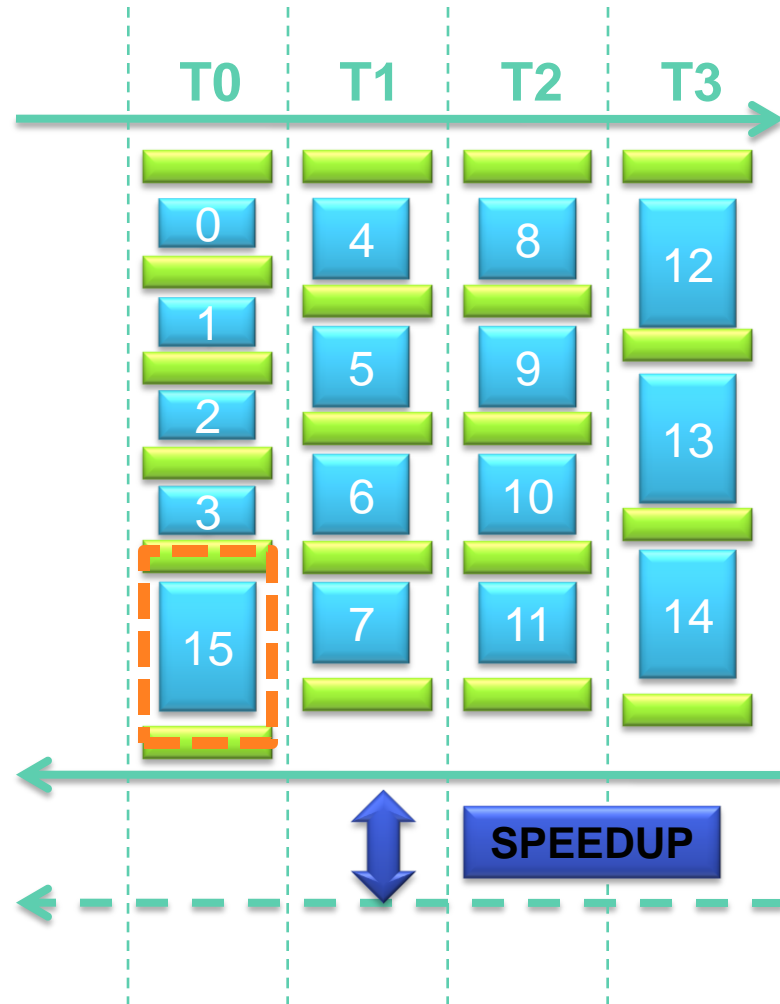


```
#pragma omp parallel for \
    num_threads (4) schedule (dynamic, 4)

for (uint i=0; i<16; i++)
{
    /* UNBALANCED LOOP CODE */
} /* (implicit) SYNCH POINT */
```

- ✓ Iterations are dynamically assigned to threads
  - ✓ 16 iter, 4 by 4
- ✓ Coarse granularity (same as static scheduling)
- ✓ Due to barrier at the end of parreg, all threads have to wait for the slowest one

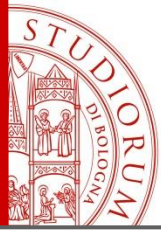
# EX 2 - Unbalanced loop partitioning



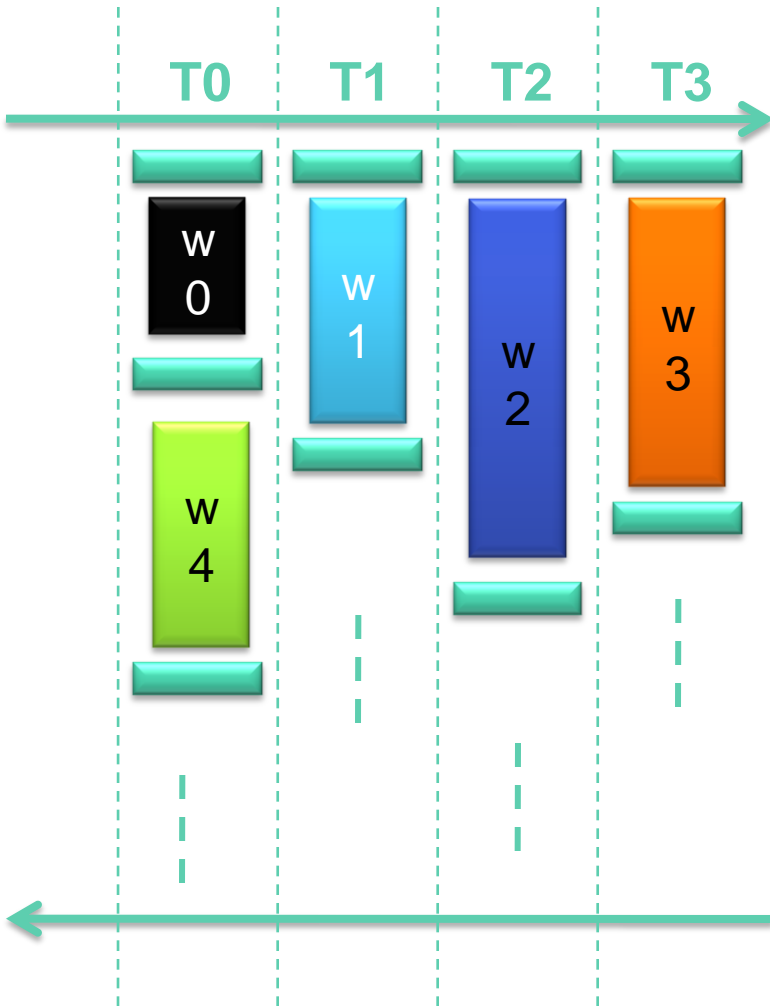
```
#pragma omp parallel for \
    num_threads (4) schedule (dynamic, 1)

for (uint i=0; i<16; i++)
{
    /* UNBALANCED LOOP CODE */
} /* (implicit) SYNCH POINT */
```

- ✓ Iterations are dynamically assigned to threads
  - ✓ 16 iter, 1 by 1
- ✓ Finest granularity balances the workload among threads
- ✓ In this case, it is worth to pay

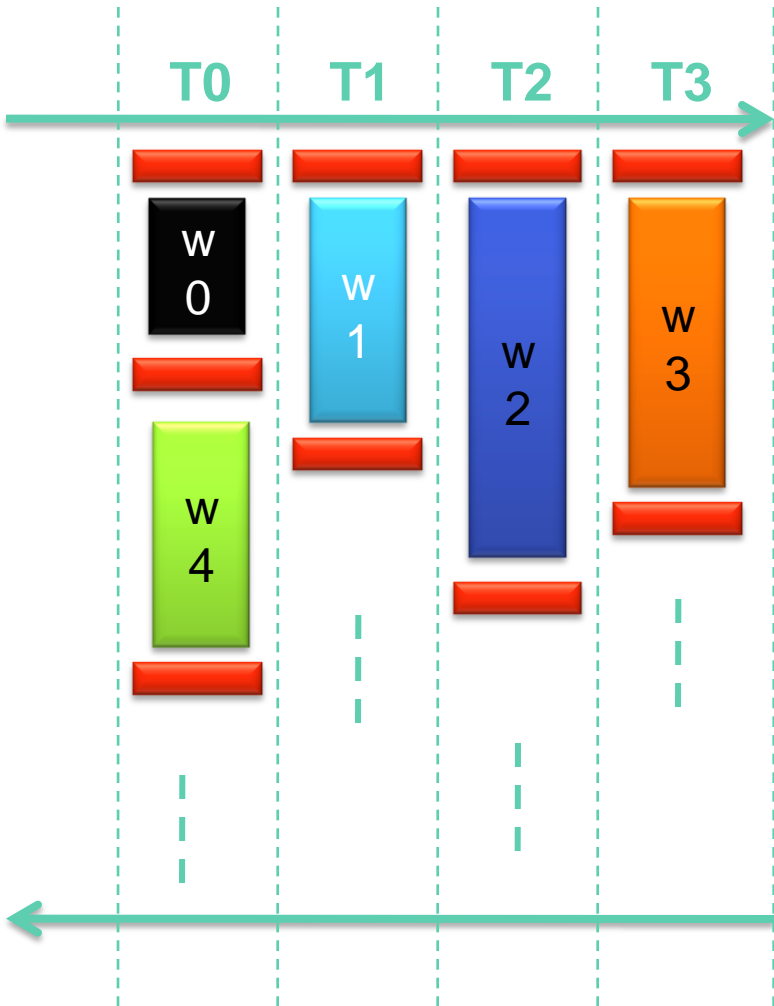


# EX 3 – Task parallelism w/sections



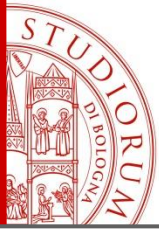
```
#pragma omp parallel sections \  
  num_threads (4)  
{  
  #pragma omp section  
  work_0 ();  
  
  #pragma omp section  
  work_1 ();  
  
  #pragma omp section  
  work_2 ();  
  
  #pragma omp section  
  work_3 ();  
  
  #pragma omp section  
  work_4 ();  
  
  // [...]  
}
```

# EX 3 – Task parallelism w/tasks



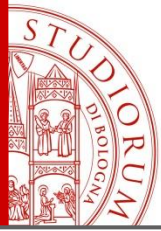
```
void (* work)(void) [5]; // FUNCTION PTRs

#pragma omp parallel num_threads (4)
{
    #pragma omp single
    {
        uint i;
        for (i=0; i<5; i++)
        {
            #pragma omp task
            work[i] ();
        }
    }
} /* (implicit) SYNCH POINT */
```



# CUDA EXERCISE

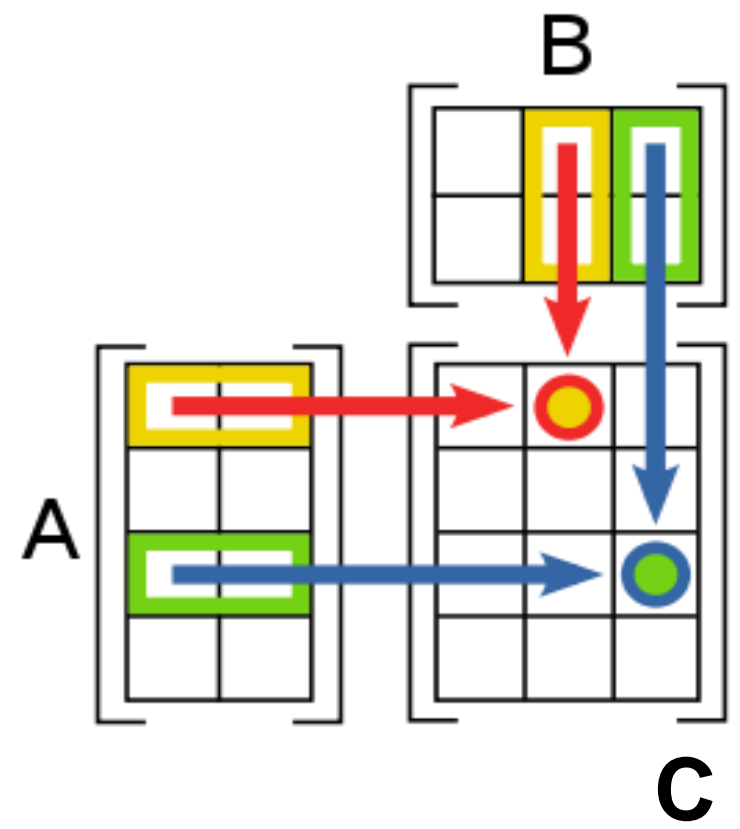
Christian Pinto  
christian.pinto@unibo.it

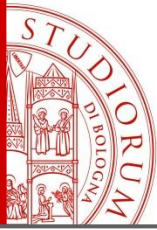


# CUDA exercise: Matrix Multiplication

$$A (m \times n) \times B (n \times p) = C (m \times p)$$

$$C_{1,2} = \sum_1^n a_{1,i} b_{i,2}$$





# CUDA exercise: APIs

## Host side

**cudaMalloc((void\*\*) pointer, int N);**

Allocate **N bytes** in device global memory.

**cudaMemcpy (void \* dst, void \* src, int transfer\_size, direction);**

Move **transfer\_size bytes** between host and device, depending on direction: **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**.

**dim3 threads(BLOCK\_SIZE, BLOCK\_SIZE);**

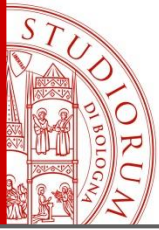
**dim3 grid(width, height);**

CUDA threads block and grid dimensions

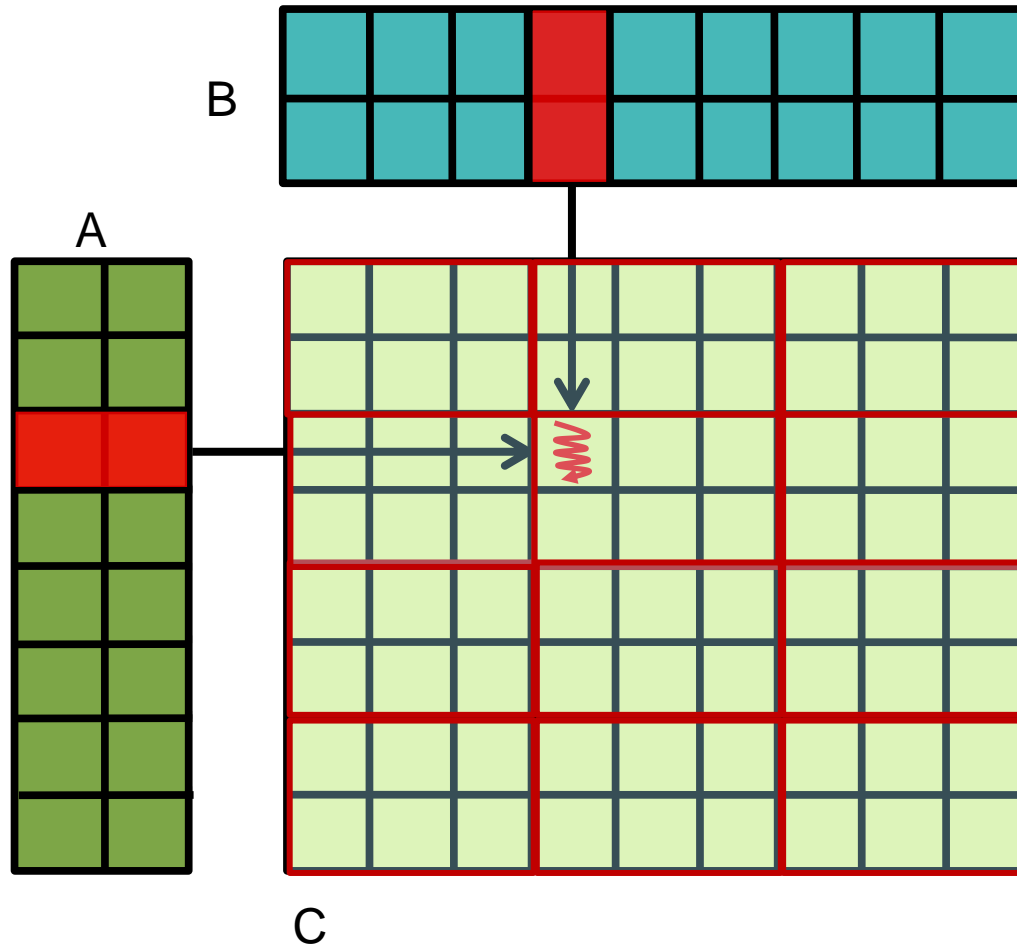
## Device side

**\_\_syncthreads()**

Intra block barrier



# CUDA exercise: naïve implementation

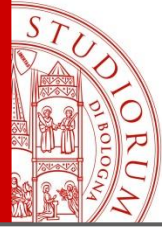


Each CUDA Thread performs the calculation of one element of the resulting matrix

i.e. cuda threads num = C  
elems num

CUDA  
32x32  
Threads Block

```
for (int i=0; i<A_width;i++)  
    C[j][k] += A[j][i] * B[i][k]
```



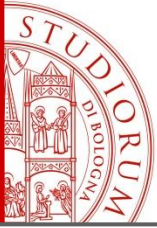
# CUDA exercise: naïve implementation TODO

---

- Host Memory Setup: allocate and initialize matrices in host memory
- Device Memory setup: allocate matrices in device global memory and fill them with previous initialized host matrices
- Implement CUDA kernel

To compile:

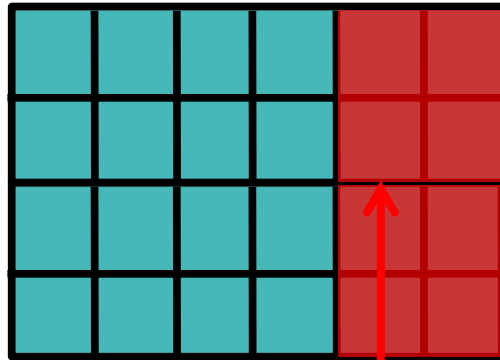
Go to src folder and run: **make clean all**



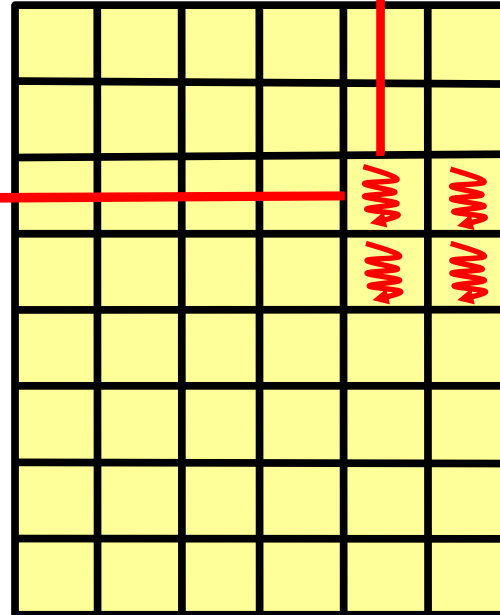
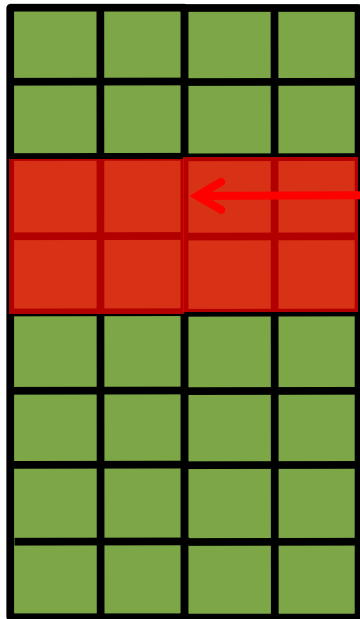
# CUDA exercise: optimized implementation

32 x 32  
Thread  
Block

B



A



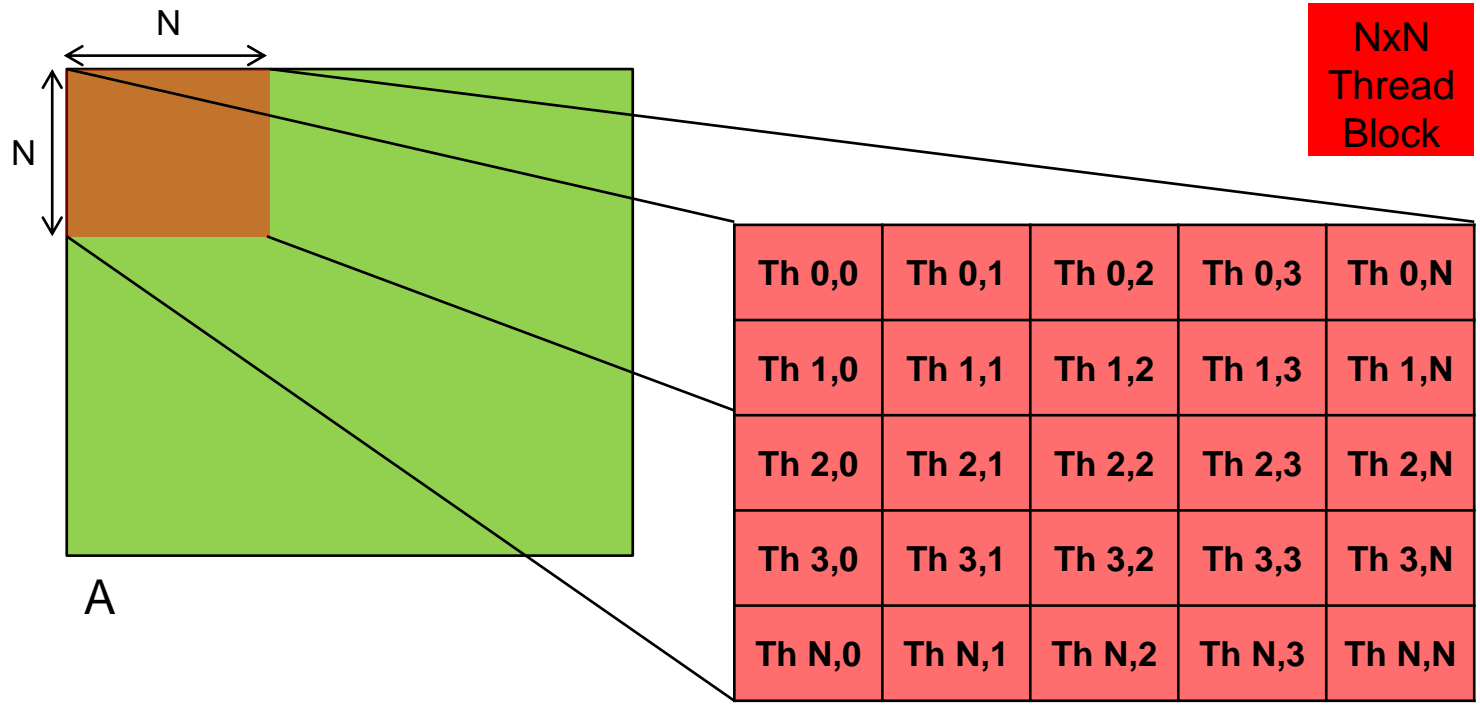
C

Each thread block copies a subset of matrices A and B from global memory to shared memory in a coalesced fashion. Each thread then applies the matrix multiplication algorithm on those subsets, getting a partial result. This operation is repeated for each subsets pairs from matrices A and B.

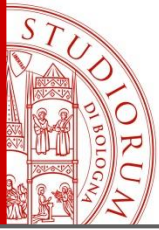
```
for(int sub=0;sub<(width_A)/BLOCK_SIZE;  
    sub++){  
    copy_current_subsets_to_shmem();  
    __syncthreads();  
    for (int i=0; i<sub_size;i++)  
        C[j][k] += subA[j][i] *  
                subB[i][k]  
}
```



# CUDA exercise: copy to shared memory



Each thread has to copy one element of the entire subset according to its ID.  
This pattern ensure coalesced accesses to global memory (if  $N = k * \text{warp\_size}$ )



# OpenCL Exercise

Giuseppe Tagliavini  
giuseppe.tagliavini@unibo.it

# Programming environment

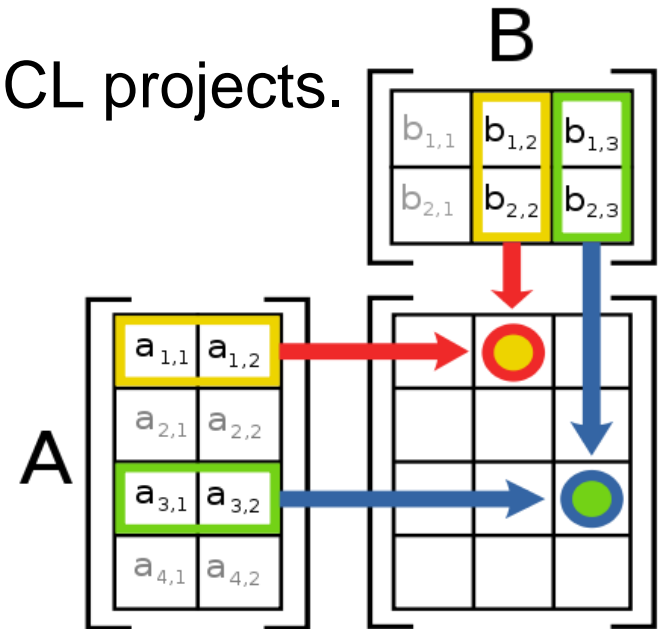
1. **opencl/common** contains some helper libraries.

2. **opencl/projects** contains the OpenCL projects.

– **opencl/projects/matmul**

3. Some useful commands:

- **make clean**      → clean all binary objects
- **make**              → compile the host code
- **make kernel**      → “compile” the kernel code





# Kernel building process

```
char *clMatrixMul = oclLoadProgSource(KERNEL_FILE, "",  
                                     &kernelLength);
```

Helper function

Kernel filename

```
oclCheckError(errcode, CL_SUCCESS);
```

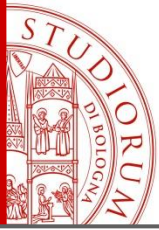
Helper function

```
clProgram = clCreateProgramWithSource(clGPUContext,  
                                     1, (const char **)&clMatrixMul,  
                                     &kernelLength, &errcode);
```

```
oclCheckError(errcode, CL_SUCCESS);
```

```
errcode = clBuildProgram(clProgram, 0,  
                        NULL, "-cl-nv-verbose", NULL, NULL);
```

Verbose kernel  
compiler output



# Part 1

---

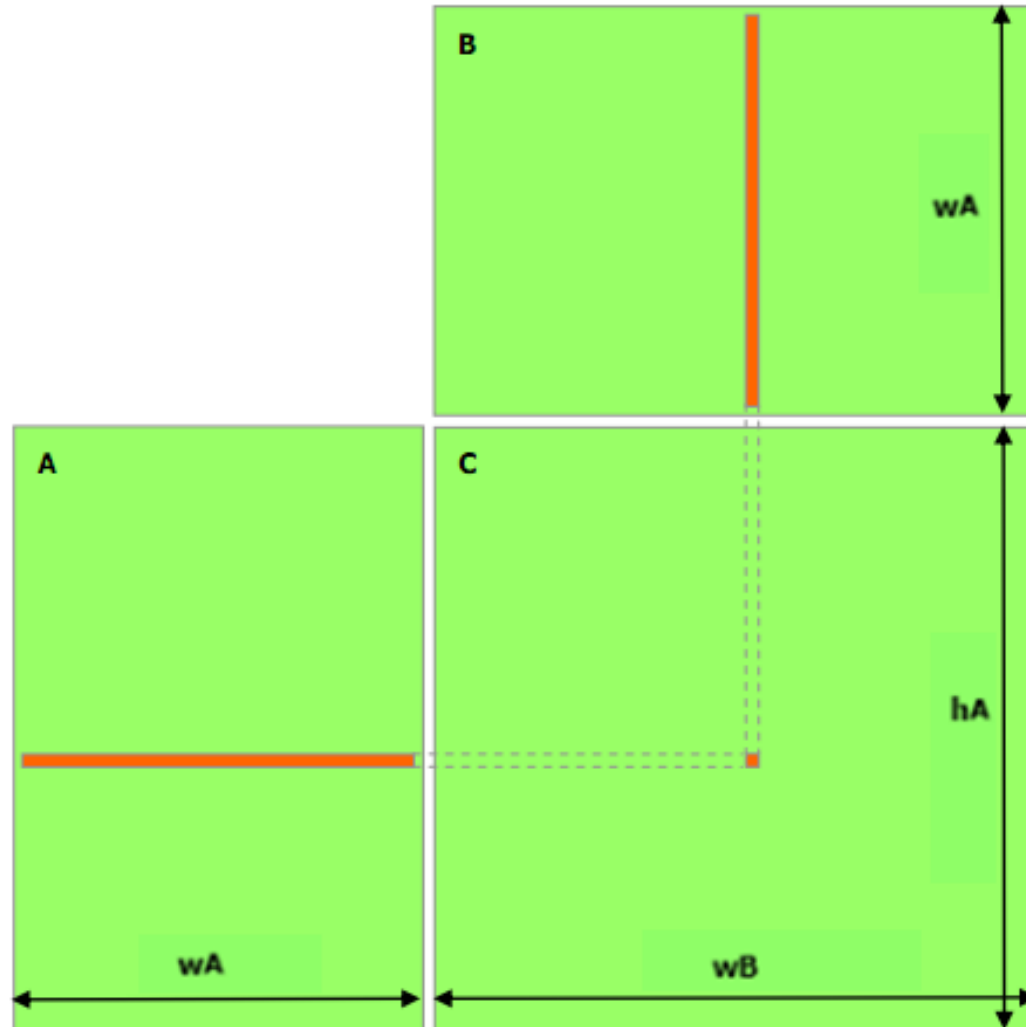
## 1. Host code

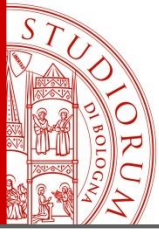
- Write the code related to device memory setup
- Write the code to launch OpenCL kernel
- Write the code to retrieve the result from device memory

## 2. Kernel code

- Write a simple kernel that directly access global device memory.

# Part 1 - Diagram





# Part 2

---

## 1. Kernel code

- Realize a kernel that uses local memory to save subparts of both input matrixes

## 2. Host code

- Adjust host code accordingly

# Part 2 - Diagram

