

# Introduction to Scientific Programming using GPGPU and CUDA



Day 2

***Sergio Orlandini***  
s.orlandini@cineca.it

***Luca Ferraro***  
l.ferraro@cineca.it

# Rights & Credits

These slides are CINECA 2014 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini

# ■ Memory Hierarchy on CUDA

- *Global Memory*
  - *caches*
  - *type of global memory accesses*
- *Shared Memory*
  - Matrix-Matrix Product using *Shared Memory*
- *Constant Memory*
- *Texture Memory*
- *Registers and Local Memory*



# Memory Hierarchy

All CUDA threads in a block have access to:

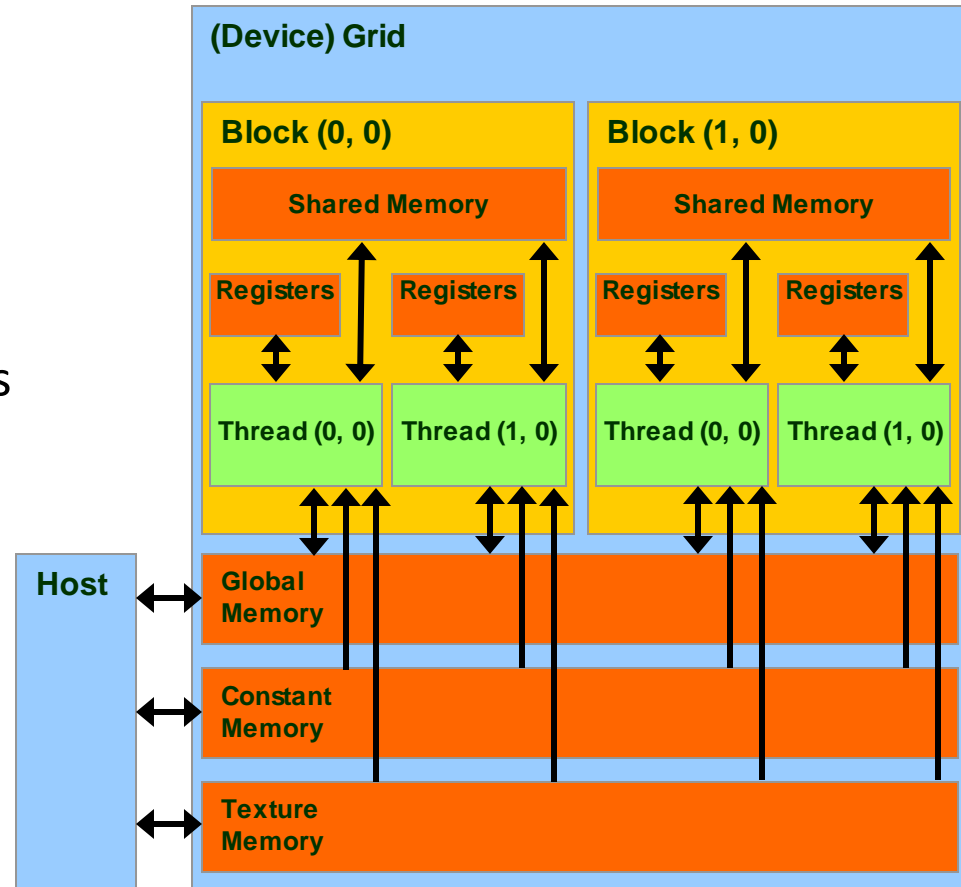
- resources of the SM assigned to its block:
  - registers**
  - shared memory**

NB: thread belonging to different blocks cannot share these resources

- all memory type available on GPU:
  - Global memory**
  - Constant Memory** (read only)
  - Texture Memory** (read only)

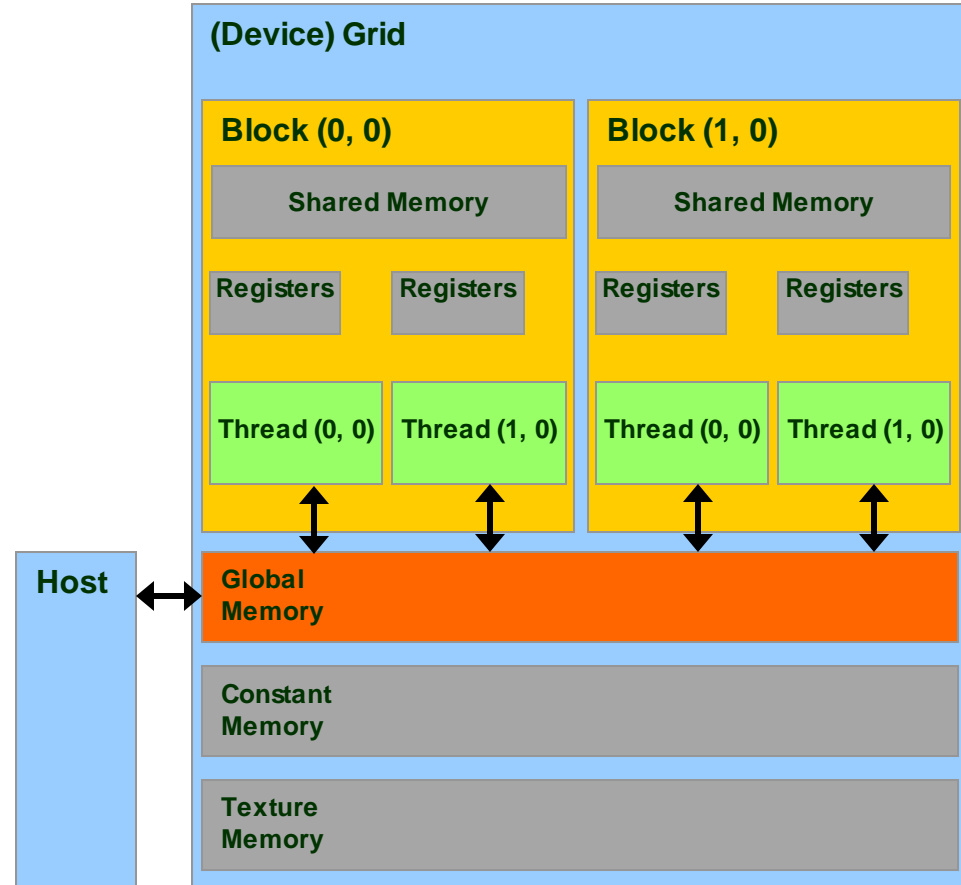
NB: CPU can access and initialize both constant and texture memory

NB: global, constant and texture memory have persistent storage duration



# Global Memory

- **Global Memory** is the larger memory available on a *device*
  - Comparable to a RAM for CPU
  - Its status is maintained among different kernel launches
  - Can be access both read/write from all threads of the kernel grid
  - Unique memory that can be used in read/write access from the CPU
  - **Very high bandwidth**  
Throughput 240-760 GB/s
  - **Very high latency**  
about 400-800 clock cycles



# Declare Variable in Global Memory

```
__device__ type variable_name; // statica
```

*or through dynamic allocation*

```
type *pointer_to_variable; // dinamica  
cudaMalloc((void **) &pointer_to_variable, size);  
cudaFree(pointer_to_variable);
```

```
type, device :: variable_name
```

*or through dynamic allocation*

```
type, device, allocatable :: variable_name  
allocate(variable_name, size)  
deallocate(variable_name)
```

# Cache Hierarchy for Global Memory Accesses

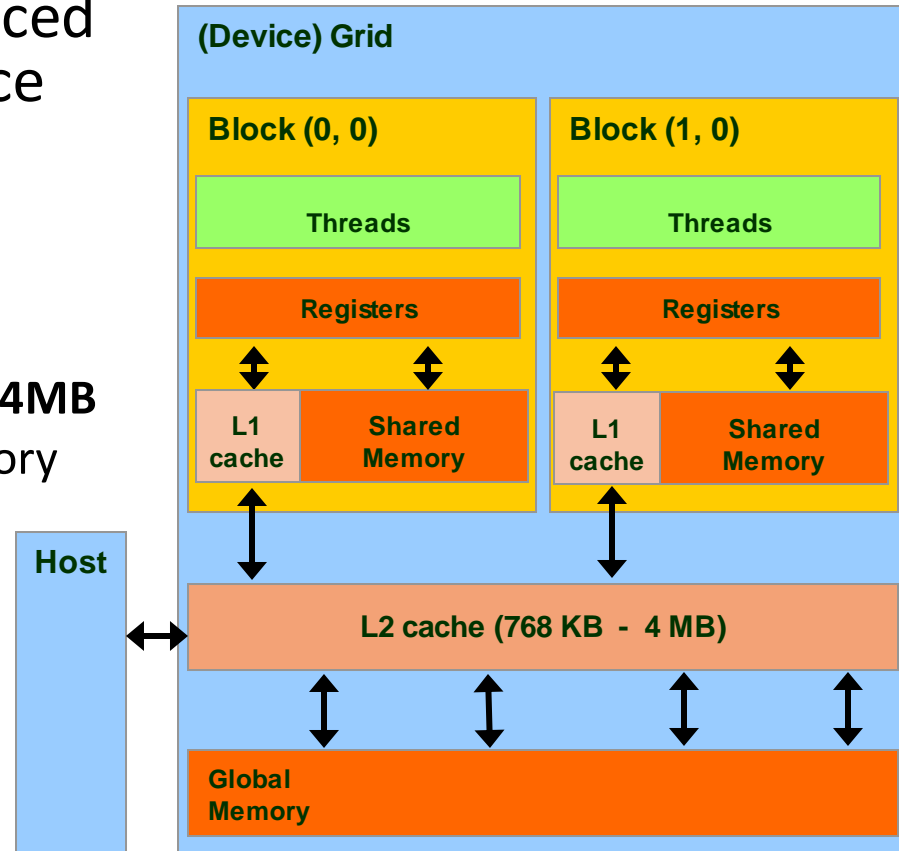
- Starting with the Fermi architecture, a cache hierarchy has been introduced in order to ease the need for space and time data locality

- 2 Levels of cache:

- L2** : shared among all SM
  - Fermi 768 KB, Kepler 1MB, Pascal 4MB
  - 25% less latency than Global Memory

NB : all accesses to the global memory pass through the L2 cache, also for H2D and D2H memory transfers

- L1** : private to each SM
  - [16/48 KB] configurable
  - L1 + Shared Memory = 64 KB
  - Kepler : configurable also as 32 KB



```
cudaFuncSetCacheConfig(kernel1, cudaFuncCachePreferL1); // 48KB L1 / 16KB ShMem  
cudaFuncSetCacheConfig(kernel2, cudaFuncCachePreferShared); // 16KB L1 / 48KB ShMem
```

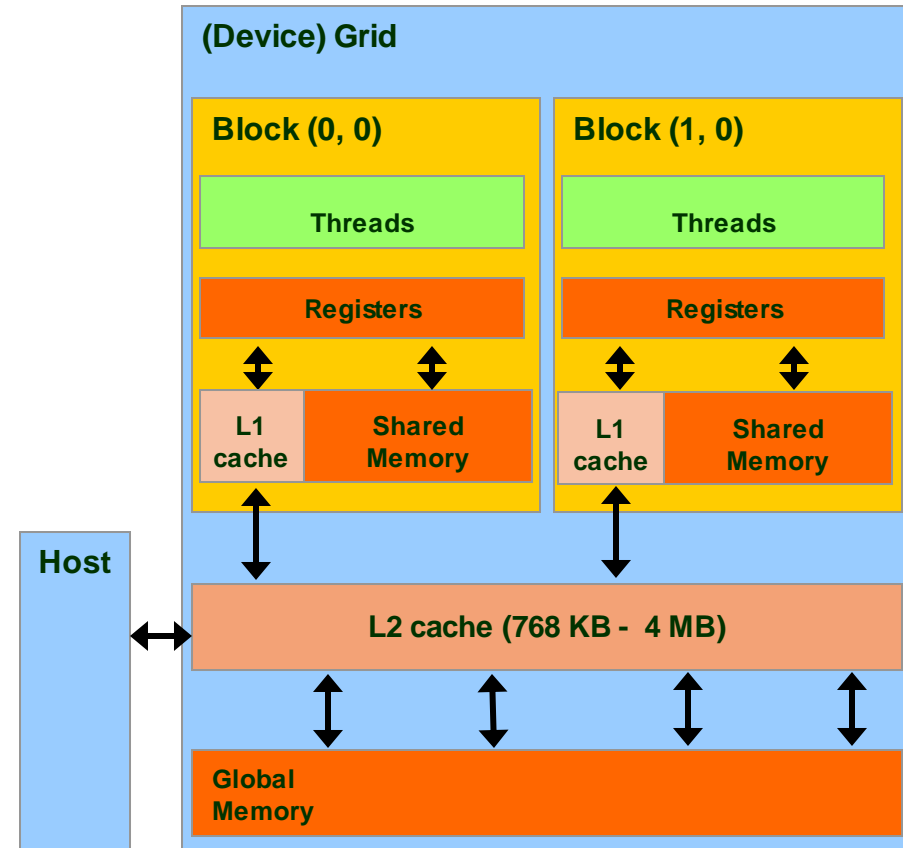
# Cache Hierarchy for Global Memory Accesses

Just one type of **store** operation:

- when data should be updated in global memory, its L1 copy is invalidated and updated the L2 cache value

Two different type of **load** operations:

- Caching (default mode)**
  - when data is requested by some thread, data is first searched in L1 cache, then in L2 cache, last in global memory
  - cache line length is **128-byte**
- Non-caching (compile time selected)**
  - the L1 cache is disabled
  - when data is requested by some thread, data is first searched in L2 cache, then in global memory
  - cache line length is **32-bytes**
  - this mode is activated at *compile time* using the compiler option:  
-Xptxas -dlcm=cg



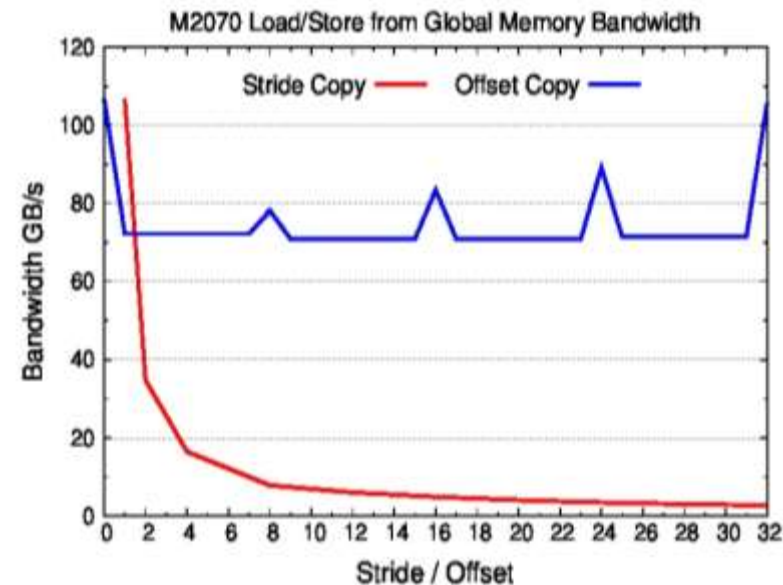


# Global Memory Load/Store

```
// strided data copy
__global__ void strideCopy (float *odata, float* idata, int stride) {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}
```

```
// offset data copy
__global__ void offsetCopy(float *odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

Strided based copy		Offset based copy	
Stride	Bandwidth GB/s	Offset	Bandwidth GB/s
1	106.6	0	106.6
2	34.8	1	72.2
8	7.9	8	78.2
16	4.9	16	83.4
32	2.7	32	105.7



Measured on a M2070; Total elements = 16776960; Used Blocks = 65535; Block length = 256

# Load Operations from Global Memory

- All load/store requests in global memory are issued per *warp* (as all other instructions)
  1. each *thread* in a *warp* compute the address to access
  2. *load/store* units select segments where data resides
  3. *load/store* start transfer of needed segments

Warp requires 32 consecutive 4-byte word aligned to segment (total 128 bytes)

## Caching Load

all addresses belong to 1 line cache segment

128 bytes are moved over the bus

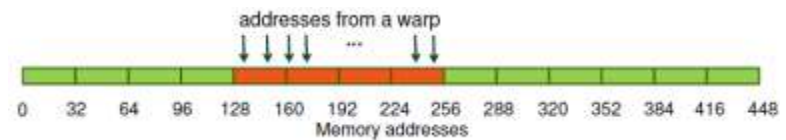
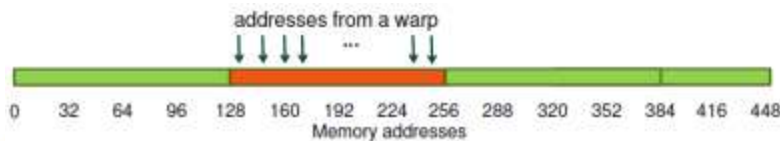
bus utilization: **100%**

## Non-caching Load

all addresses belong to 4 line cache segments

128 bytes are moved over the bus

bus utilization: **100%**



# Load Operations from Global Memory

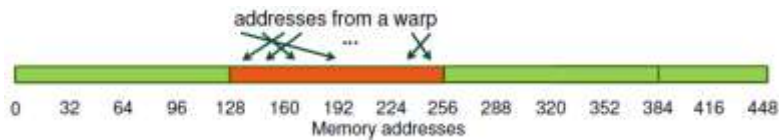
Warp requests 32 permuted 4-byte words alined to segment (total 128 bytes)

## Caching Load

addresses belong to 1 line cache segments

128 bytes are moved over the bus

bus utilization: **100%**

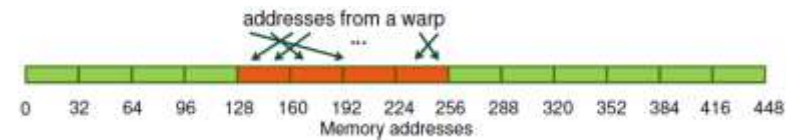


## Non-caching Load

addresses belong to 4 line cache segments

128 bytes are moved over the bus

bus utilization: **100%**



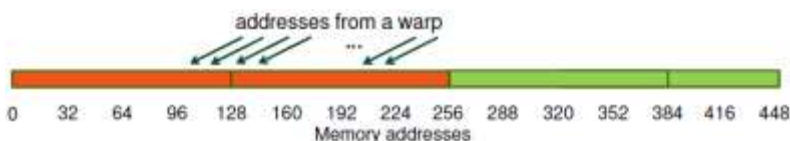
Warp requires 32 consecutive 4-bytes words not alined to segment (total 128 bytes)

## Caching Load

addresses belong to 2 line cache segments

256 bytes are moved over the bus

bus utilization: **50%**

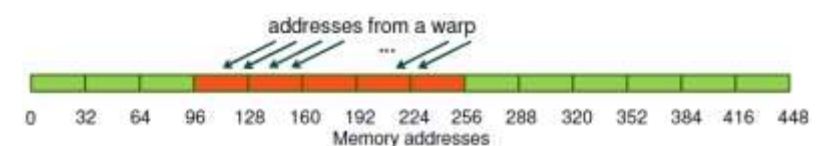


## Non-caching Load

addresses belong to 5 line cache segments

160 are moved over the bus

bus utilization: **80%**



# Load Operations from Global Memory

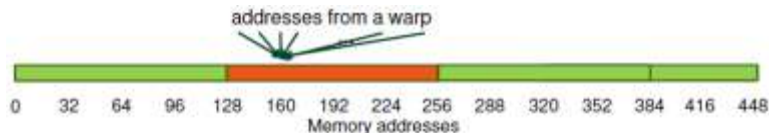
All threads in a warp request the same 4-byte word (total 4 bytes)

## Caching Load

address belongs to only one cache line segment

128 bytes are moved over the bus

bus utilization: **3.125%**



## Non-caching Load

address belongs to only one cache line segment

32 bytes are moved over the bus

bus utilization: **12.5%**



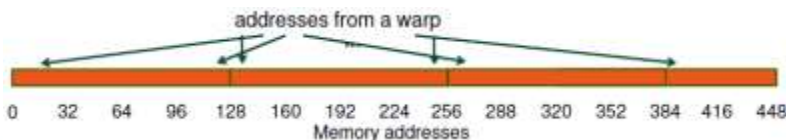
Warp request 32 not contiguous 4-bytes words (total 128 bytes)

## Caching Load

addresses belong to N different line cache

$N \times 128$  bytes are moved over the bus

bus utilization:  **$128 / (N \times 128)$**

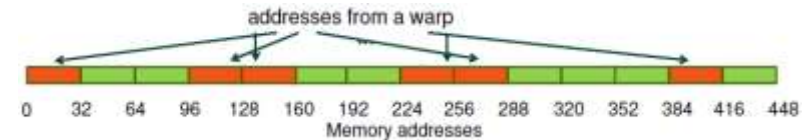


## Non-caching Load

addresses belong to N different line cache

$N \times 32$  bytes are moved over the bus

bus utilization:  **$128 / (N \times 32)$**



# Data alignment in *Global Memory*

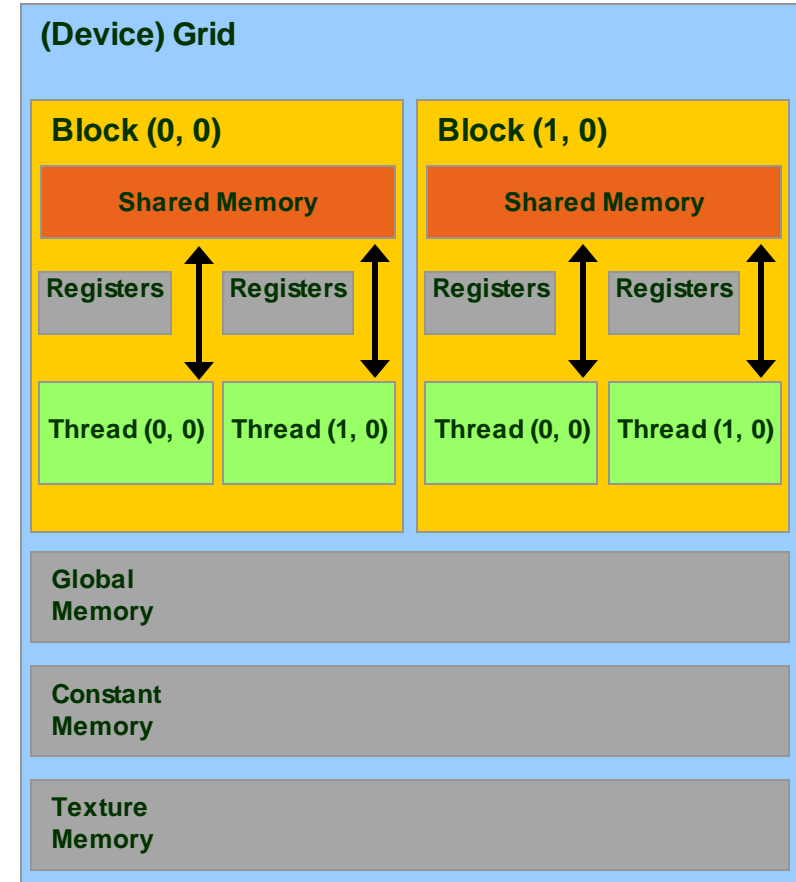
- It is very important to align data in memory so to have aligned accesses (*coalesced*) during load/store operation in global memory, reducing the number of segments moved across the bus
  - **cudaMalloc()** grants the alignment of first element in global memory, useful for one dimensional arrays
  - **cudaMallocPitch()** must be used to allocate 2d buffers
    - elements are padded so each row is aligned for coalescing accesses
    - returns an integer (*pitch*) which can be used as a stride to access row elements

```
// host code
int width = 64, height = 64; int pitch; float *devPtr;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);

// device code
__global__ myKernel(float *devPtr, int pitch, int width, int height)
{
    for (int r = 0; r < height; r++) {
        float *row = devPtr + r * pitch;
        for (int c = 0; c < width; c++)
            float element = row[c];
    }
    ...
}
```

# Shared Memory

- The **Shared Memory** is a small, but quite fast memory mounted on each SM
  - read/write access for threads of blocks residing on the same SM
  - a cache memory under the direct control of the programmer
  - its status is not maintained among different kernel calls
- Specifications:
  - **Very low latency**: 2 clock cycles
  - Throughput: 32 bit every 2 cycles
  - Dimension : **48 KB [default]**  
(Configurable : 16/48 KB)  
**Kepler** : also 32 KB



# Shared Memory Allocation

```
// statically inside the kernel
__global__ myKernelOnGPU (...) {
    ...
    __shared__ type shmem[MEMSZ];
    ...
}

or using dynamic allocation

// dynamically sized
extern __shared__ type *dynshmem;

__global__ myKernelOnGPU (...) {
    ...
    dynshmem[i] = ... ;
    ...
}

void myHostFunction() {
    ...
    myKernelOnGPU<<<gs, bs, MEMSZ>>>();
}
```

```
! statically inside the kernel
attribute(global)
subroutine myKernel(...)
    ...
    type, shared:: variable_name
    ...
end subroutine

oppure

! dynamically sized
type, shared:: dynshmem(*)

attribute(global)
subroutine myKernel(...)
    ...
    dynshmem(i) = ...
    ...
end subroutine
```

- variables allocated in shared memory has storage duration of the kernel launch (not persistent!)
- only accessible by threads of the same block

# Thread Block Synchronization

- All threads in the same block can be synchronized using the CUDA runtime API call:

`__syncthreads ()` | **call `syncthreads ()`**

which blocks execution until all other threads reach the same call location

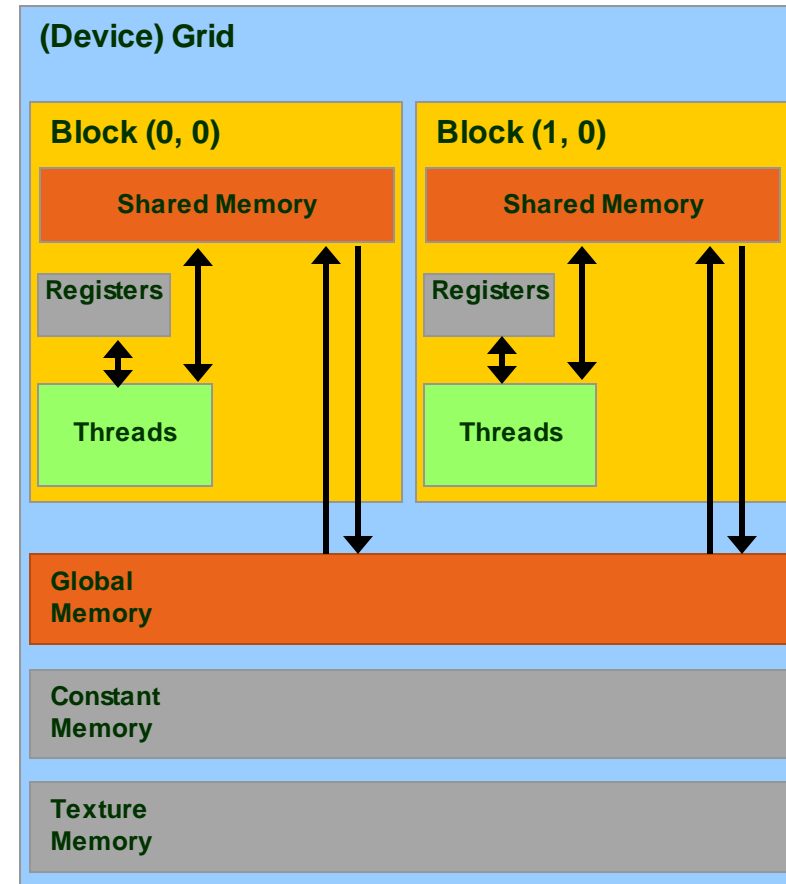
- can be used in conditional too, but only if all thread in the block reach the same synchronization call

*“... otherwise the code execution is likely to hang or produce unintended side effects”*



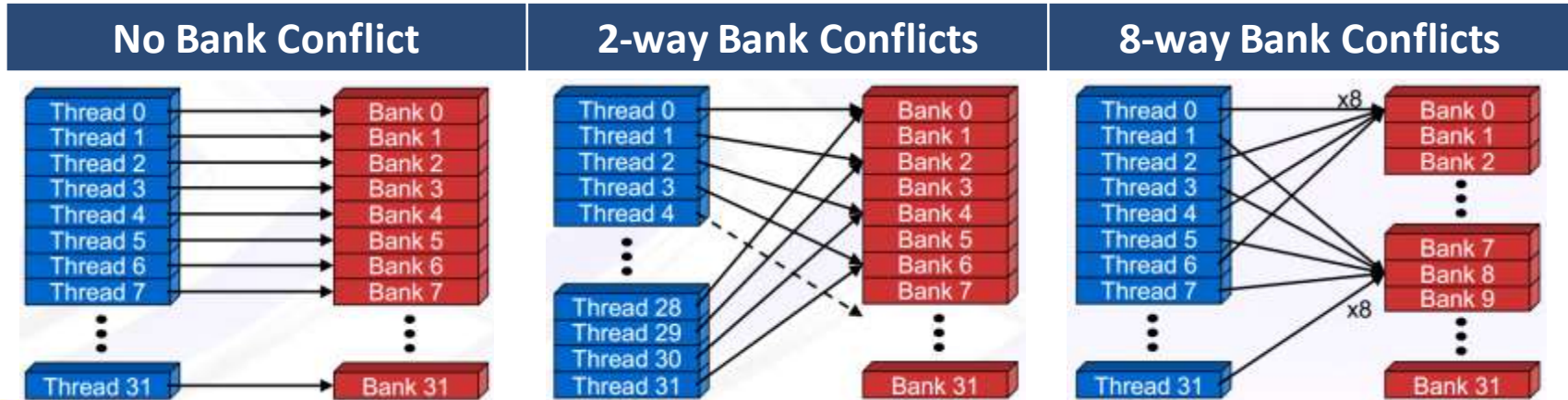
# Using Shared Memory for Thread Cooperation

- Threads belonging to the same block can cooperate together using the shared memory to share data
  - if a thread is in need of some data which has been already retrieved by another thread in the same block, this data can be shared using the shared memory
- typical Shared Memory usage pattern:
  - declare a buffer residing on shared memory (this buffer is per block)
  - load data into shared memory buffer
  - synchronize threads so to make sure all needed data is present in the buffer
  - perform operation on data
  - synchronize threads so all operations have been performed
  - write back results to global memory



# Shared Memory and Bank Accesses

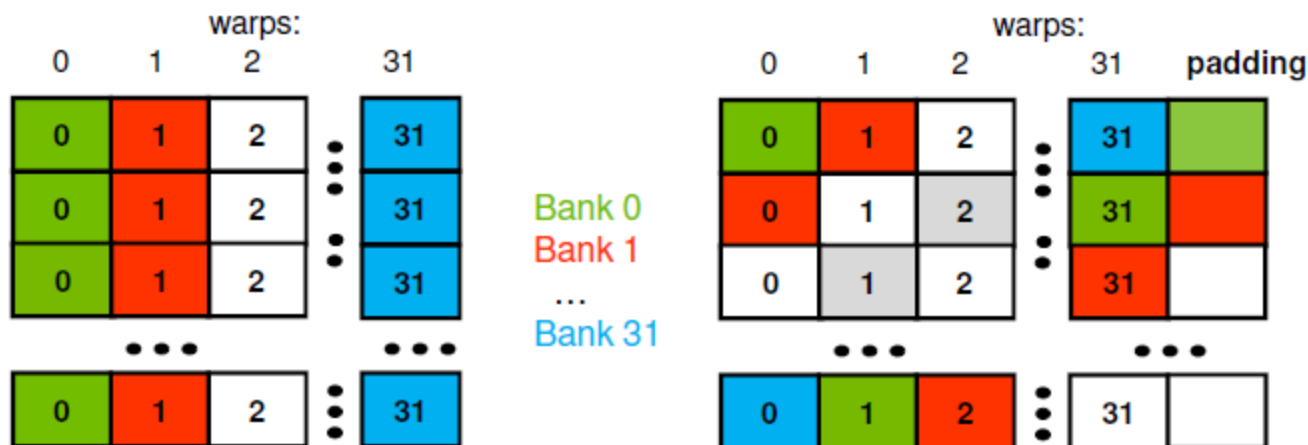
- Shared memory has 32 banks organized such that successive 32-bit words map to successive banks
  - data are distributed every 4-bytes cycling over successive banks
  - Shared memory accesses are per warp
  - **Multicast** : if N threads of the same warp request the same element, access is executed with only one transaction
  - **Broadcast** : if ALL threads of the same warp request the same element, accesso is executed with only one transaction
  - **Bank Conflict**: if two or more threads requests different data belonging to the same bank, each access is served separately (serialized)



# Avoid Bank Conflict

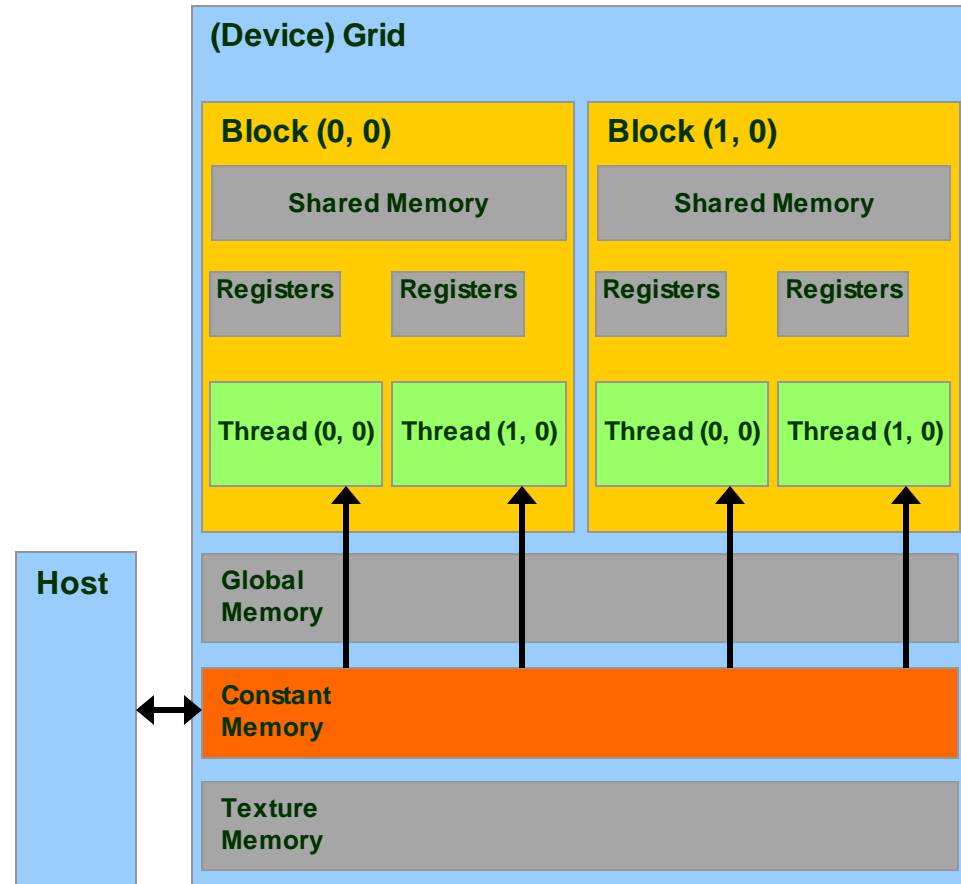
- A naive implementation of CUDA kernels using shared memory would use a tile of size 32x32 floats
  - each element resides on a single bank (4-byte)
  - data are on the same bank every 32 floats
  - so read/write by columns will turn into the worst type of bank conflict
- Use a common trick: let's size the tile using 33 elements
  - now all elements belonging to the same column reside on different banks

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



# Constant Memory

- **Constant Memory** is the ideal place to store constant data in **read-only** access from all threads
  - constant memory data actually reside in the global memory, but fetched data is moved into a dedicated *constant-cache*
  - very effective when all *thread* of a *warp* request the same memory address
  - it's values are initialized from host code using a special CUDA API
- Specifications:
  - Dimension : **64 KB**
  - Throughput: 32 bits per warp every 2 clock cycles

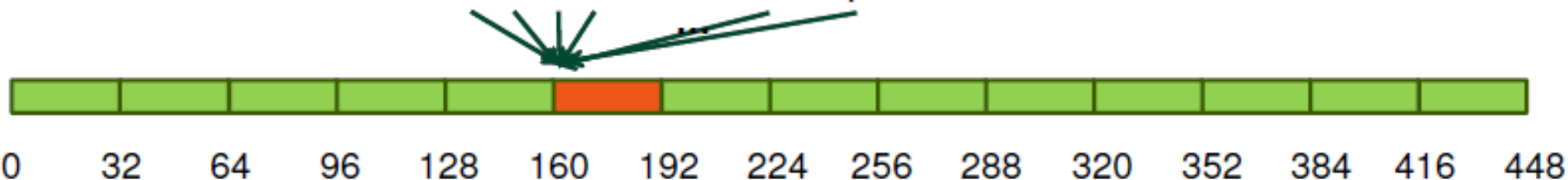


# Accessing Constant Memory

Suppose a kernel is launched using 320 warps per SM and all threads requests the same data

- if data is on global memory:
  - all *warp* will request the same segment from global memory
  - the first time segment is copied into L2 cache
  - if other data pass through L2, there are good chances it will be lost
  - there are good chances that data should be requested 320 times
- if data is in constant memory:
  - during first *warp* request, data is copied in *constant-cache*
  - since there is less traffic in *constant-cache*, there are good chances all other *warp* will find the data already in cache, so no more traffic on the BUS

addresses from a warp



# Constant Memory Allocation

```
__constant__   type  variable_name; // static

cudaMemcpyToSymbol(const_mem, &host_src, sizeof(type), cudaMemcpyHostToDevice);

// warning
// cannot be dynamically allocated
```

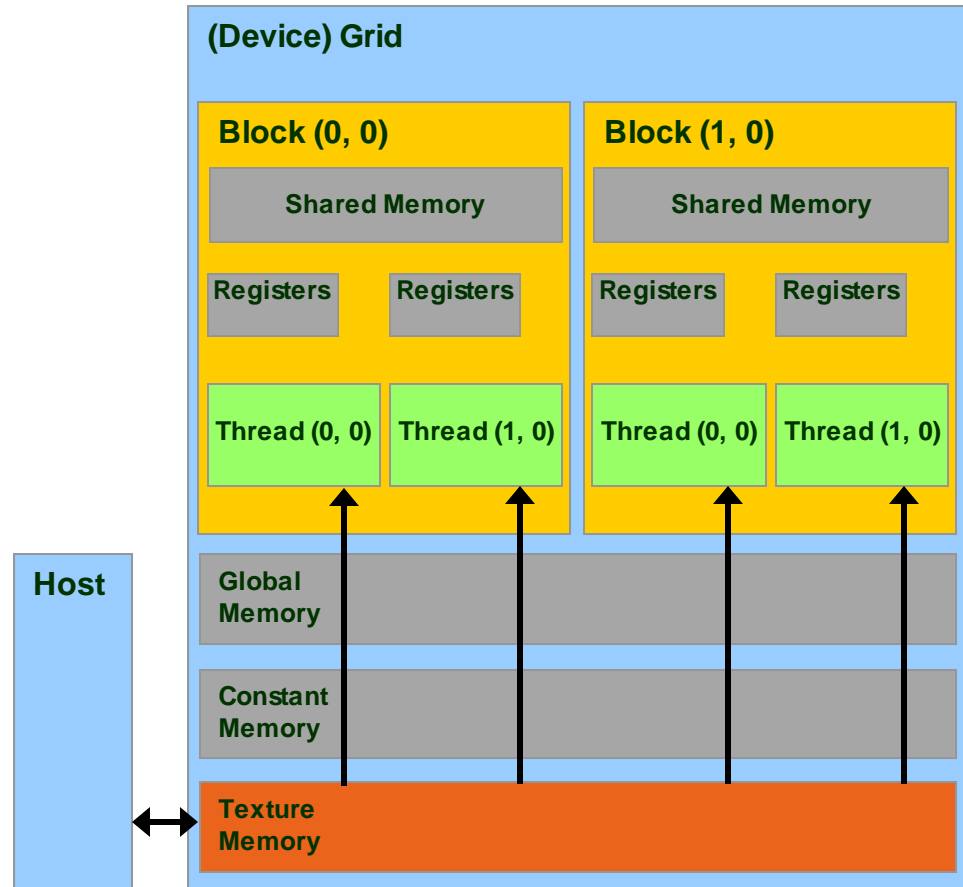
```
type, constant :: variable_name

! warning
! cannot be dynamically allocated
```

- data will reside in the constant memory address space
- has static storage duration (persists until the application ends)
- readable from all threads of a running kernel

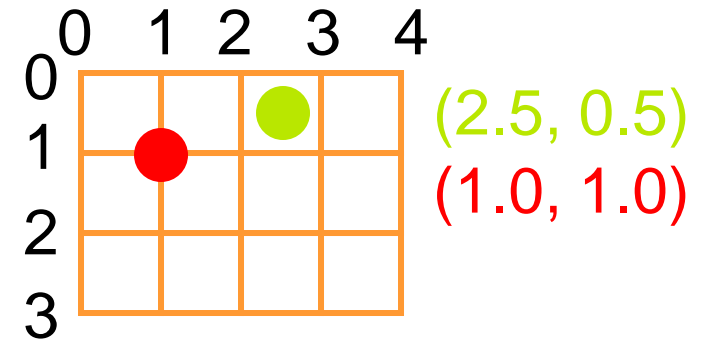
# Texture Memory

- **Texture Memory** is after all a remain of basic graphic rendering functionality needs
- as for constant memory, data actually reside in the global memory, fetched across dedicated texture-cache
- data is accessed in **read-only** using special CUDA API function, called **texture fetch**
- Specifications:
  - address resolution is more efficient since it is performed on dedicated hardware
- specialized hardware for:
  - out-of-bound address resolution
  - floating-point interpolation
  - type conversion or bit operations

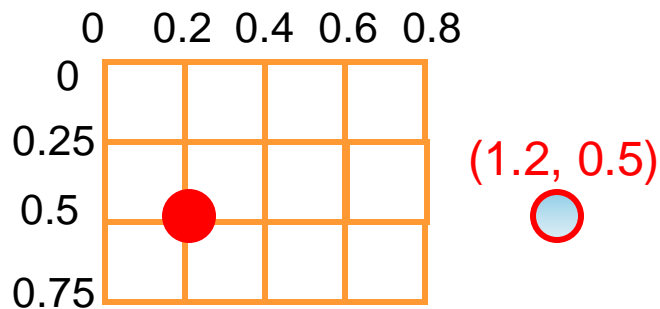


# Texture Memory Addressing Features

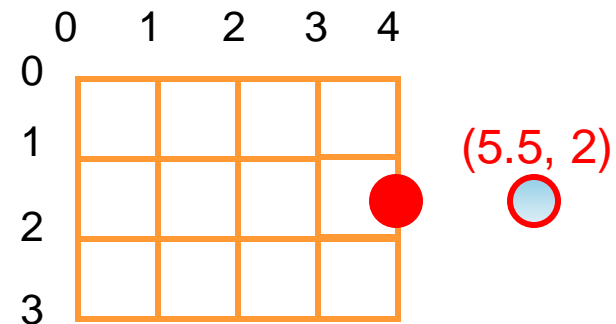
- integer 1D:  $[0, N-1]$
- normalized 1D:  $[0, 1-1/N]$
- available interpolation:
  - floor, linear, bilinear
  - weights are 9 bit



**Wrap:** out-of-border coordinates are replaced in the box using modulus (available only for normalized indexing)



**Clamp:** out-of-border coordinates are clamped to nearest box bound





# Steps for Accessing Texture Memory

## CPU

- Allocate global memory on the device (standard, pitched or as cudaArray)  
`cudaMalloc(&d_a, memsize);`
- Create a “texture reference” object at file scope (static global variable):  
`texture<DataType, Type, ReadMode> d_a_texRef;`
  - `DataType` cannot be a double
  - `Type` [optional] can be `cudaTextureType{1,2}DLayered`
  - `ReadMode` [optional]
- Create a “channel descriptor” object to describe the return type of texture memory load:  
`cudaChannelFormatDesc d_a_desc = cudaCreateChannelDesc<datatype>();`
- Bind the texture reference to memory  
`cudaBindTexture(0, d_a_texRef, d_a, d_a_desc);`
- when finished: unbind the texture reference (there is a maximum number of usable textures):

## GPU

- access data from CUDA kernels through “texture reference”:
  - `tex1Dfetch(d_a_texRef, indirizzo)` - for linear memory
  - `tex1D()`, `tex2D()`, `tex3D()` - for pitched linear texture and cudaArray:

# Texture Usage Example (old way)

```
__global__ void shiftCopy(int N, int shift, float *odata, float *idata)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = idata[xid+shift];
}

texture<float, 1> texRef; // TEXTURE creation

__global__ void textureShiftCopy(int N, int shift, float *odata)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = tex1Dfetch(texRef, xid+shift); // TEXTURE FETCHING
}

...

ShiftCopy<<<nBlocks, NUM_THREADS>>>>(N, shift, d_out, d_inp);

cudaChannelFormatDesc d_a_desc = cudaCreateChannelDesc<float>();
cudaBindTexture(0, texRef, d_a, d_a_desc); // BIND TEXTURE MEMORY
textureShiftCopy<<<nBlocks, NUM_THREADS>>>>(N, shift, d_out);
```

# Texture Memory in Kepler: aka *Read-only Cache*

- Starting from Kepler GPU architecture (cc 3.5) constant memory loads from global memory can pass through the *texture cache*:
  - without using an explicit texture *binding*
  - without limits on the maximum allowed number of texture

```
__global__ void kernel_copy (float *odata, float *idata) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    odata[index] = __ldg(idata[index]);  
}
```

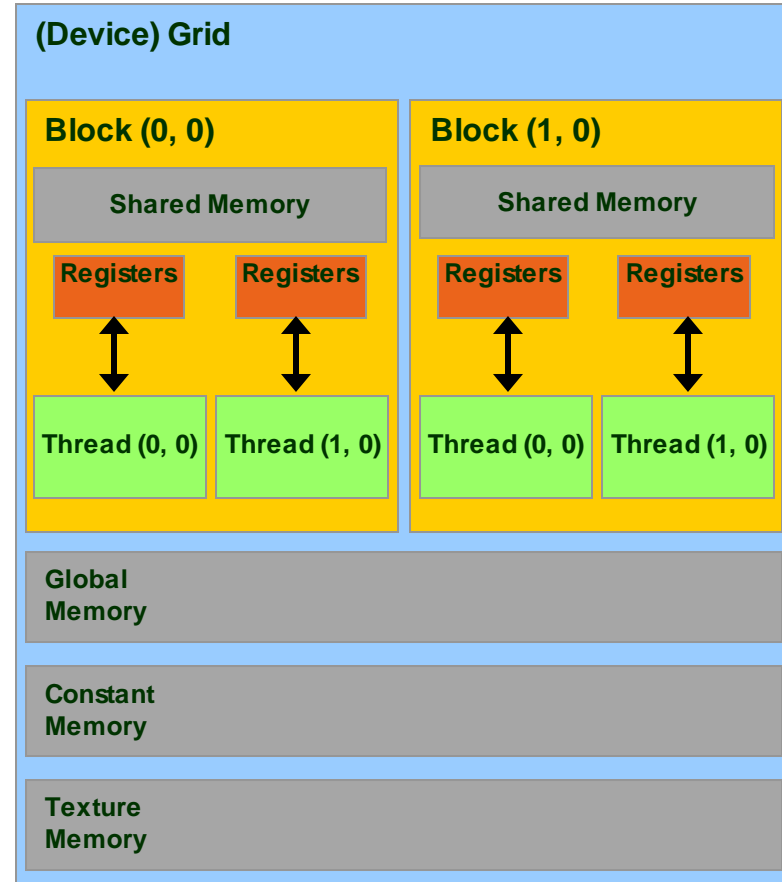
```
__global__ void kernel_copy (float *odata, const __restrict__  
float *idata) {  
    int index = blockIdx.x * blockDim.x + threadIdx.x;  
    odata[index] = idata[index];  
}
```

# Registers

- **registers** are used to store scalar or small array variables with frequent access by each thread
  - **Fermi** : 63 registers per thread / 32 KB
  - **Kepler** : 255 registers per thread / 64 KB
  - **Pascal** : same as Kepler
- **WARNING:**
  - the less registers a kernel needs, the more blocks can be assigned to a SM
  - pay attention to *Register Pressure*: can be a limiting factor for performances
  - the number of register per kernel can be limited during *compile time*:
    - `--maxregcount max_registers`
  - the number of active blocks per kernel can be forced using the CUDA special qualifier

`__launch_bounds__`

```
__global__ void  
__launch_bounds__(maxThreadsPerBlock,  
                  minBlocksPerMultiprocessor)  
my_kernel ( ... ) { ... }
```



# Local Memory

- **Local Memory** does not correspond to a real physical memory place
- Automatic variables are often placed in local memory by the compiler:
  - large structures or arrays that would consume too much register space
- If a kernel uses more registers than available (register spilling), the compiler shall move variables into local memory
- Local memory is often mapped to global memory
  - using the same *Caching* hierarchies (L1 for read-only variables)
  - facing the same latency and bandwidth limitation of global memory
- In order to obtain information on how much local, constant, shared memory and registers are required for each kernel, you can provide the following compiler options

**--ptxas-options=-v**

```
$ nvcc -arch=sm_20 -ptxas-options=-v my_kernel.cu
...
ptxas info : Used 34 registers, 60+56 bytes lmem, 44+40 bytes
smem, 20 bytes cmem[1], 12 bytes cmem[14]
...
```

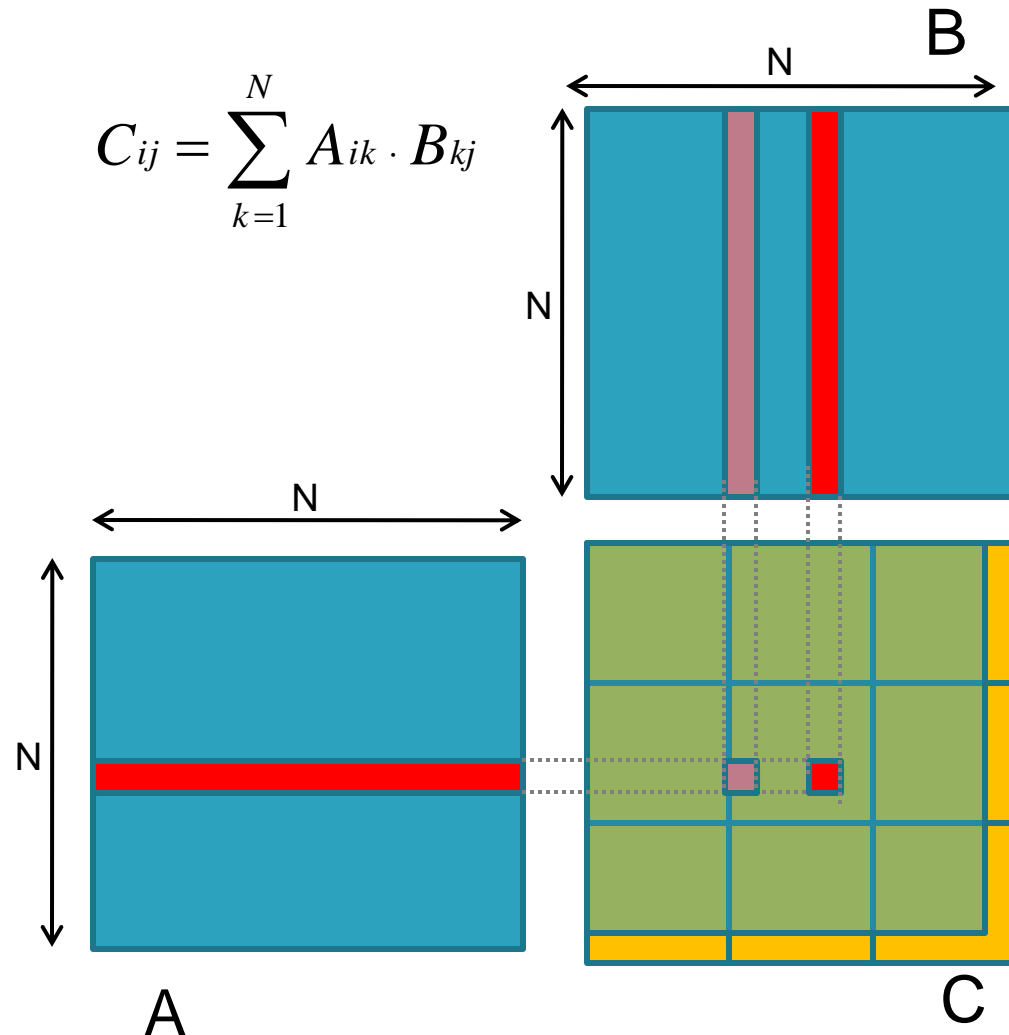
## ■ Matrix-Matrix Product

- limits of the global memory implementation
- using shared memory
- implementation guidelines



# Matrix-matrix Product using Global Memory

- Each thread compute one element of C, using 2N elements (N from A, N from B) and performing 2N floating-point operations (N add , N mul)
- Yet every element of C sharing the same row or column retrieve N times the same elements from A or B
- This implementation results in  $2N^3$  loads !!!
- We can avoid requesting the same elements many times, sharing them through the shared memory
  - each thread can retrieve just one data element data in parallel and store it into shared memory
  - when all thread have loaded needed data, they can access all the elements by the threads belonging to the same block, for example sharing a full row or column
- Unfortunately shared memory size is small
  - 16/48 KB depending on the compute capability



# Matrix-matrix using Shared Memory

- Let's solve the problem using blocks of (NB,NB) dimension

- each CUDA thread block compute the elements of a single matrix block of size (NB·NB) of matrix C
- each resulting matrix block of matrix C is obtained as the product of all sub-matrices of A and B

$$C_{ij} = \sum_{S=1}^{N/NB} \sum_{k=1}^{NB} A_{Sik} \cdot B_{S kj}$$

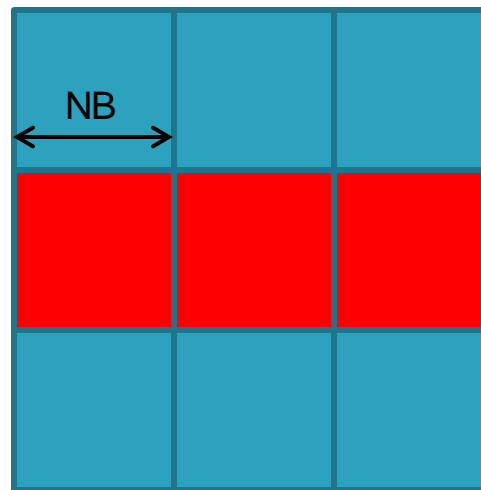
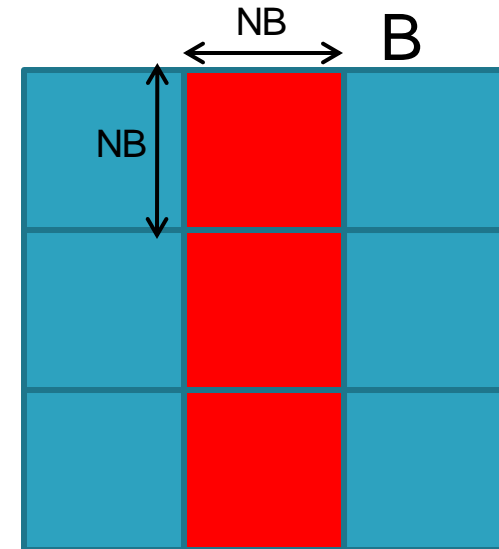
The kernel is divided in two phases:

1. thread load a block of A and B from global memory to shared memory
2. thread compute the element of subblock C reading from shared memory

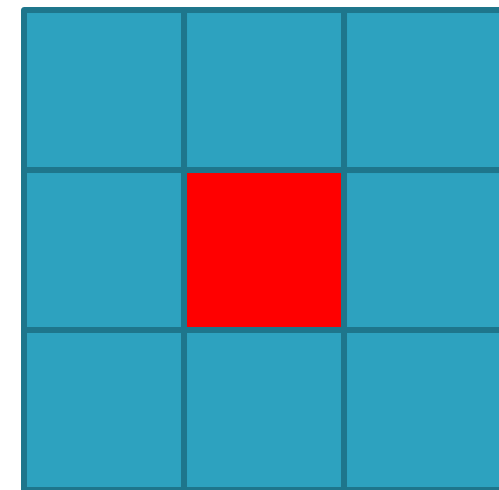
- Elements of each subblock C are accumulated using local variables in registers, then stored in global memory

- Two thread synchronization are required

- after the load of subblock of matrix A and B, so to grant all data is available for subblock matrix product
- after the partial subblock matrix product, so to grant that next load of other subblock will not overwrite elements not yet used in current block evaluation



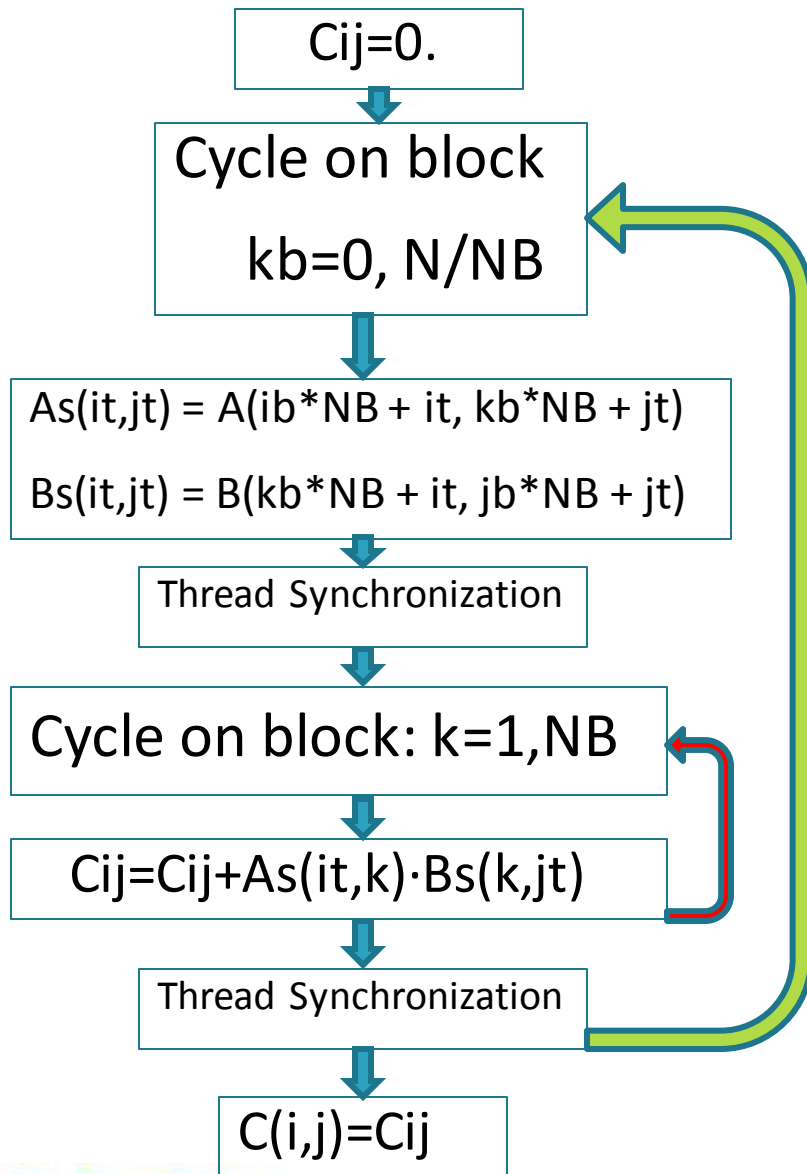
A



C

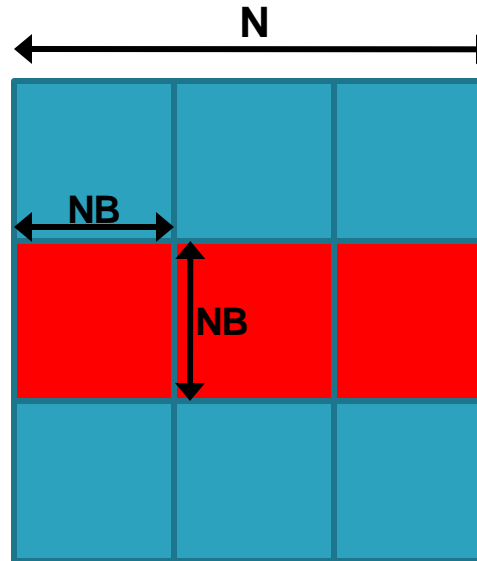


# Matrix-matrix using Shared Memory: Flow

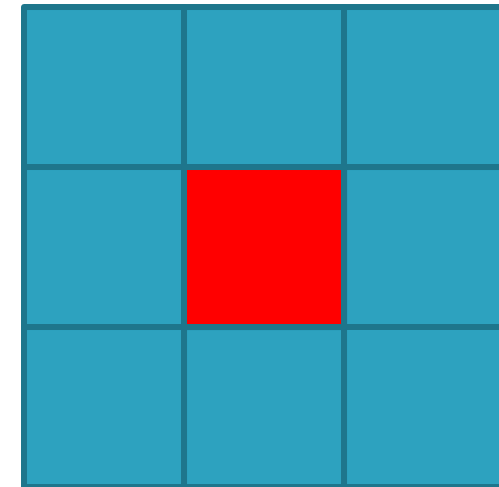
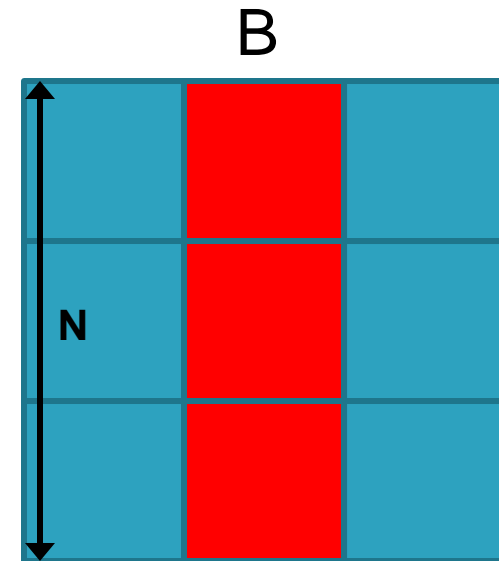


```
it = threadIdx.y  
jt = threadIdx.x  
  
ib = blockIdx.y  
jb = blockIdx.x
```

```
it = threadIdx%x  
jt = threadIdx%y  
  
ib = blockIdx%x - 1  
jb = blockIdx%y - 1
```



A



C

# Matrix-matrix using Shared Memory: Kernel

```
// Matrix multiplication kernel called by MatMul_gpu()
__global__ void MatMul_kernel (float *A, float *B, float *C, int N)
{
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float Asub[NB][NB];
    __shared__ float Bsub[NB][NB];

    // Block row and column
    int ib = blockIdx.y;
    int jb = blockIdx.x;

    // Thread row and column within Csub
    int it = threadIdx.y;
    int jt = threadIdx.x;

    int a_offset , b_offset, c_offset;

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
```

```
    for (int kb = 0; kb < (A.width / NB); ++kb) {

        // Get the starting address of Asub and Bsub
        a_offset = get_offset (ib, kb, N);
        b_offset = get_offset (kb, jb, N);

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        Asub[it][jt] = A[a_offset + it*N + jt];
        Bsub[it][jt] = B[b_offset + it*N + jt];

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int k = 0; k < NB; ++k) {
            Cvalue += Asub[it][k] * Bsub[k][jt];
        }
        // Synchronize to make sure that the preceding
        // computation is done
        __syncthreads();
    }

    // Get the starting address (c_offset) of Csub
    c_offset = get_offset (ib, jb, N);
    // Each thread block computes one sub-matrix Csub of C
    C[c_offset + it*N + jt] = Cvalue;
}
```

## ■ Matrix Transpose

- evaluating performance
- using shared memory for coalesced-accesses
- avoiding bank-conflicts



# Matrix Transpose

- Let's implement matrix transpose with the following simple design:
  - out-of-place buffers
  - square matrices with size modulo 32 elements
- This is a memory-bounded kernel
  - no computation on elements
  - just load and stores
- We will use effective bandwidth (GB/s) as a metric to measure the performance of such kernels

# Simple Copy Kernel

```
__global__ void copy (float *idata, float *odata, int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in  = width * yIndex + xIndex;
    int index_out = index_in;

    odata[index_out] = idata[index_in];
}
```

- We can use a simple copy kernel as a reference
    - TILE\_DIM is the size of the square sub-matrix block
    - we map CUDA blocks to sub-matrix blocks
- matrix sub-block size == CUDA thread block size

# Naive Transpose

```
__global__ void transposeNaive(float *idata, float *odata, int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in  = width * yIndex + xIndex;
    int index_out = height * xIndex + yIndex;

    odata[index_out] = idata[index_in];
}
```

```
attributes(global) subroutine transposeNaive (idata, odata, width, height)
    integer, intent(in), value :: width, height
    real, intent(in) :: idata(width,height)
    real, intent(out) :: odata(height,width)

    i = ( blockIdx%x - 1 ) * TILE_DIM + threadIdx%x
    j = ( blockIdx%y - 1 ) * TILE_DIM + threadIdx%y

    odata(j,i) = idata(i,j)

end subroutine
```

# Measuring Performance

```
// take measurements for loop over kernel launches
cudaEventRecord(start);
for (int i=0; i < NUM_REPS; i++)
    kernel<<<grid, threads>>>(
        d_idata, d_odata, width, height);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float outerTime;
cudaEventElapsedTime(&outerTime, start, stop);

// take measurements for loop inside kernel
cudaEventRecord(start);
kernel<<<grid, threads>>>(
    d_idata, d_odata, width, height, NUM_REPS);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float innerTime;
cudaEventElapsedTime(&innerTime, start, stop);
```

## Effective Bandwidth (GB/s) on 2048x2048 S2050

<i>kernel</i>	<i>Performance [GB/s]</i>
<b>Copy</b>	60.9
<b>Naive</b>	22.4

... what is happening?

# Accessi non coalesced

- All loads from input matrix are coalesced:
  - each *warp* reads a line of contiguous elements
    - 32 float belonging to the same cache line
- yet all stores into transposed matrix are not coalesced:
  - the copy kernel store by lines
  - the naive transpose kernel store by columns
    - threads in a warp write an elements into different segments
    - the matrix stride rules how distant those segments are
- the naive transpose kernel performs 32 different stores per row



# Coalesced Transpose



- To avoid non-coalesced store we should store by row:
  - let's fill a tile in shared memory with data to be transposed
  - we don't get any penalty writing elements by columns into shared-memory
  - the transpose operation is now performed in shared-memory
  - once the tile is filled, we write back into global memory by rows

# Coalesced Transpose

```
__global__ void transposeCoalesced(float *idata, float *odata, int width, int
height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = width * yIndex + xIndex;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = height * yIndex + xIndex;

    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

# Coalesced Matrix Transpose

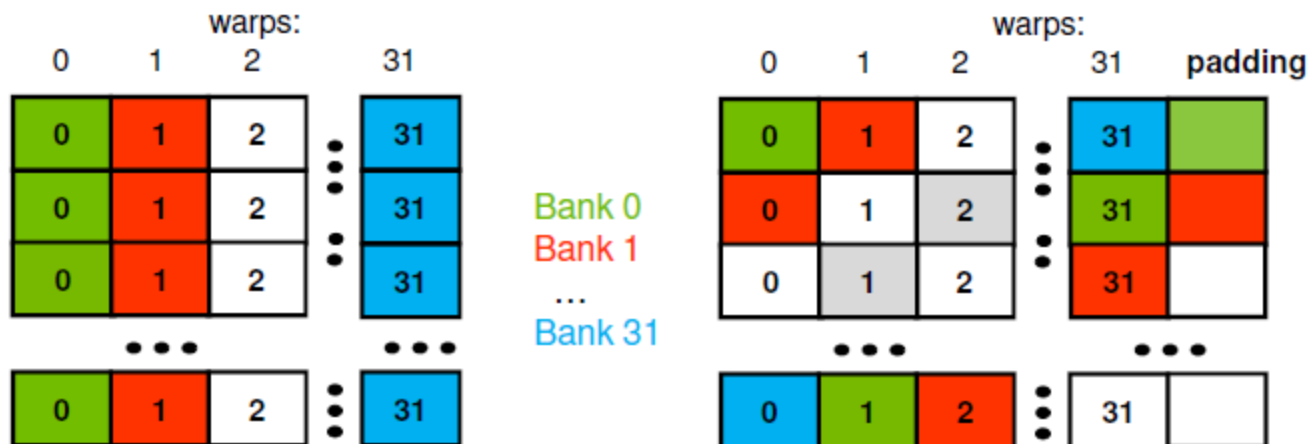
Effective Bandwidth (GB/s) on 2048x2048 S2050	
<i>kernel</i>	<i>Performance [GB/s]</i>
<b>Copy</b>	60.9
<b>Naive</b>	22.4
<b>Coalesced</b>	24.1

... mmm, we are still missing something

# Avoiding Bank Conflict

- our new coalesced transpose use a shared memory tile of 32x32 float
  - each element resides on successive bank (4-byte)
  - accessing elements with a 32 size stride will fetch them from the same bank
  - any read/write access to this tile by column will get a 32 bank conflict
- use the trick! a new tile of 32x33 elements
  - element of the same tile column will reside on different banks
  - no more bank conflicts at all

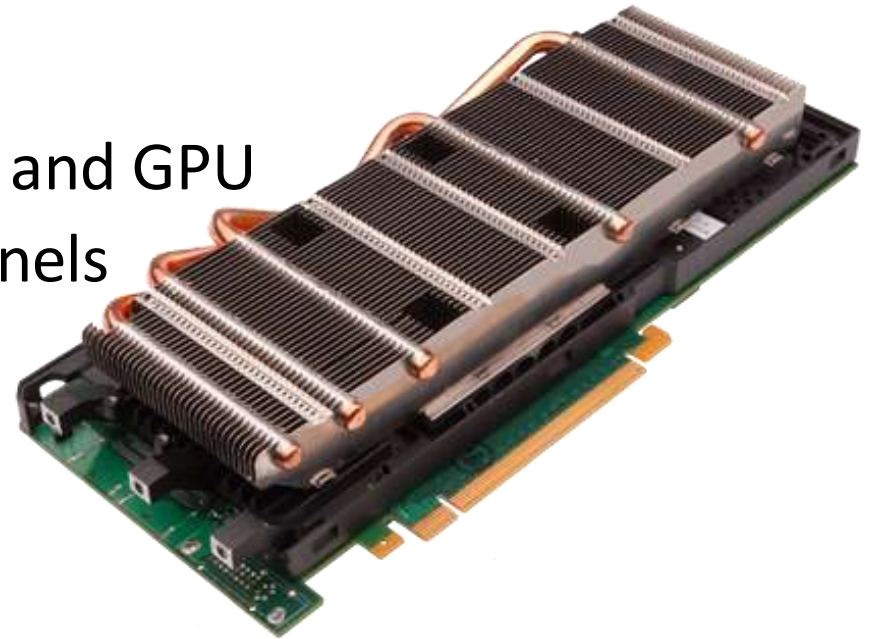
```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



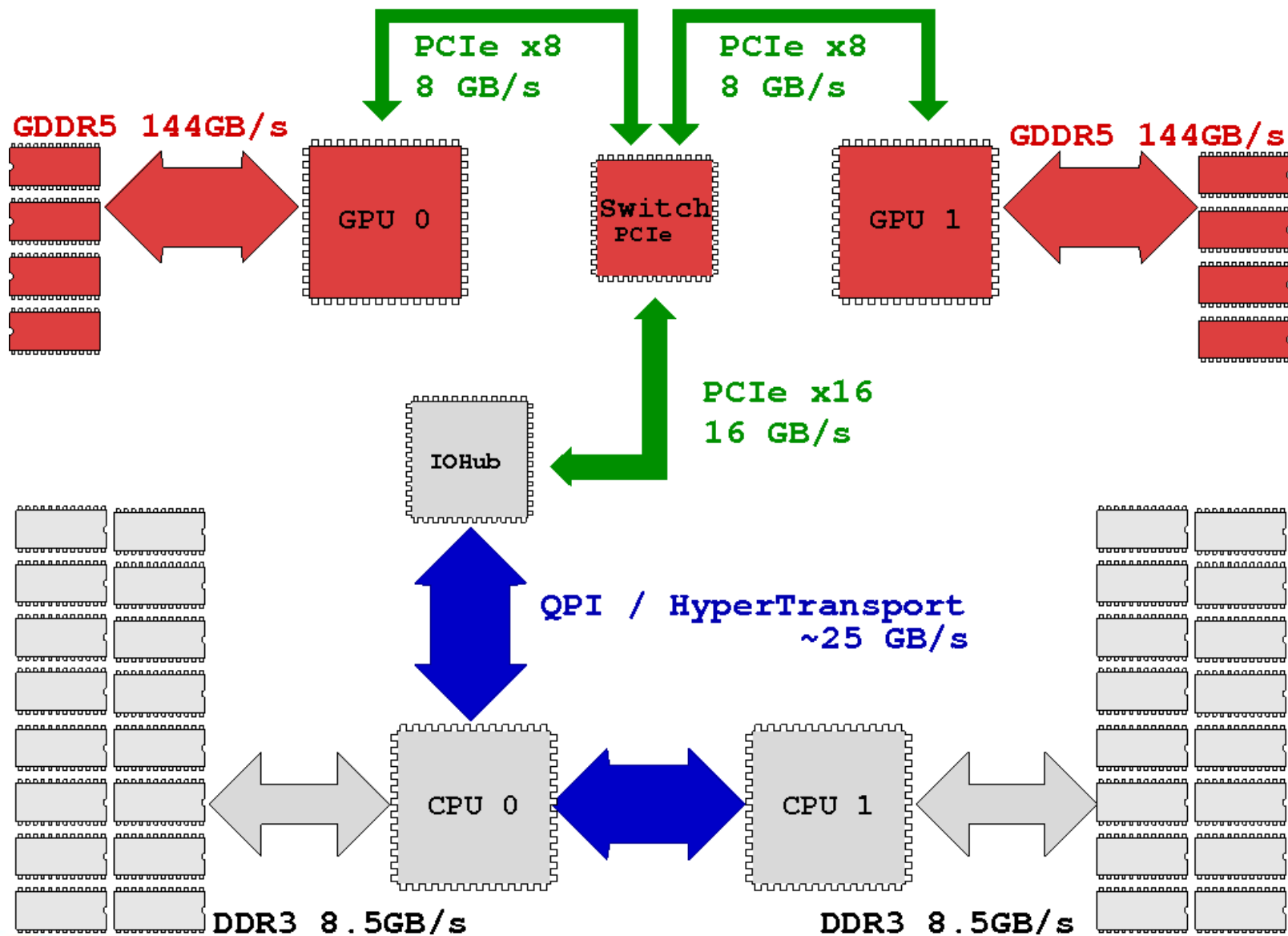
# Avoiding Bank Conflict

Effective Bandwidth (GB/s) on 2048x2048 S2050	
<i>kernel</i>	<i>Performance [GB/s]</i>
<b>Copy</b>	60.9
<b>Naive</b>	22.4
<b>Coalesced</b>	24.1
<b>no Bank Conflicts</b>	46.6

- Synchronous and Asynchronous
- Concurrent Execution
- CPU and GPU interaction
  - concurrent execution on CPU and GPU
  - overlapping transfers and kernels
- Managing multi-device
- GPU/GPU interactions



# Connection Scheme of *host/device*



# Blocking and Non-blocking Functions

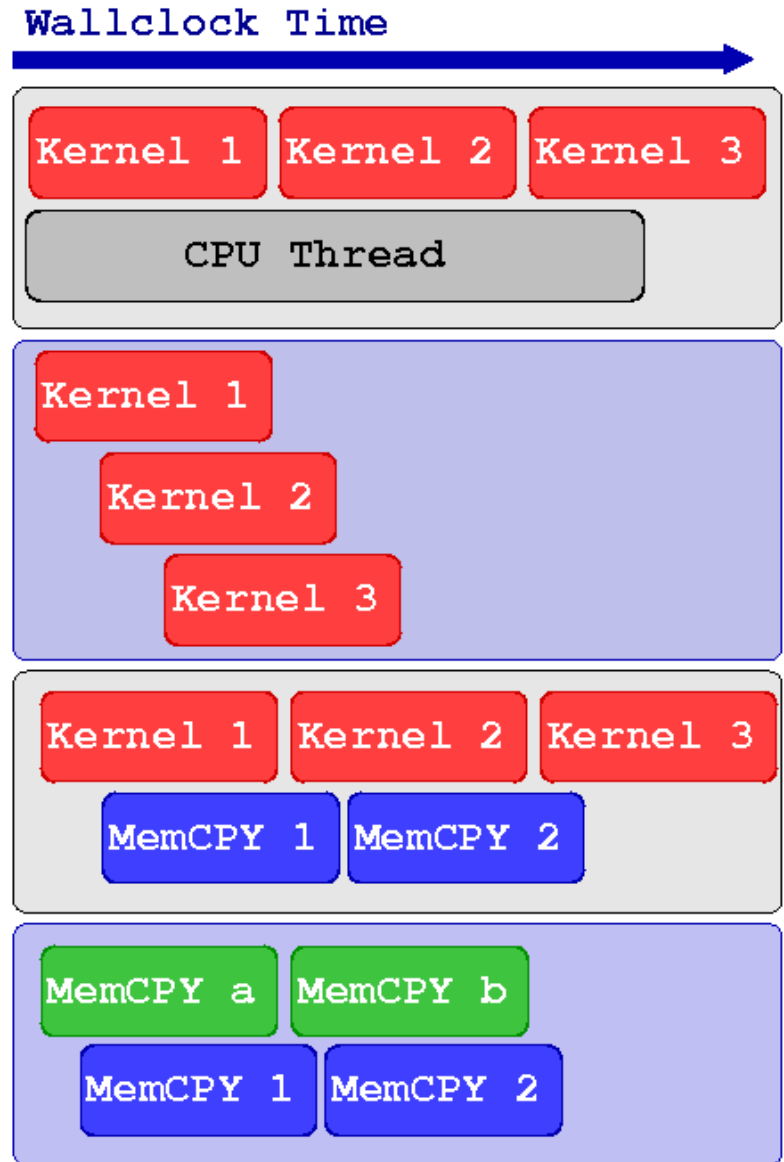
- every CUDA action is submitted to an execution queue on the *device*
- CUDA runtime functions can be divided in two categories:
  - **blocking** (synchronous):  
return control to *host* thread after execution is completed on *device*
    - all memory transfer > 64KB
    - all memory allocation on *device*
    - allocation of page locked memory on *host*
  - **Non-blocking** (asynchronous):  
return control to *host* immediately, while its execution proceeds on *device*
    - all kernel launch are asynchronous
    - all memory transfers < 64KB
    - memory initialization on *device* (cudaMemset)
    - memory copies from *device* to *device*
    - explicit asynchronous memory transfers
- CUDA API provides asynchronous versions of their counterpart synchronous functions
- Asynchronous functions allow to set up concurrent execution of many operations on *host* and *device*



# Concurrent and Asynchronous Execution

Asynchronous functions allows to expose concurrent execution:

1. Overlap computation on host and on device
2. execution of more than on kernel on the same *device*
3. data transfers between *host* and *device* while executing a kernel
4. data transfers from *host* to *device*, while transferring data from *device* to *host*



# Example of Concurrent Execution

```
cudaSetDevice(0)
kernel <<<threads, Blocks>>> (a, b, c)

// execute some work on CPU while GPU keeps on computing
CPU_Function()

// blocks CPU until GPU has finished its work
cudaDeviceSynchronize()

// CPU can use data resulting from the GPU computation
CPU_uses_the_GPU_kernel_results()
```

Since CUDA kernel invocation is an asynchronous operation, CPU can proceed and evaluate the `CPU_Function()` while the GPU is involved in kernel execution (*concurrent execution*).

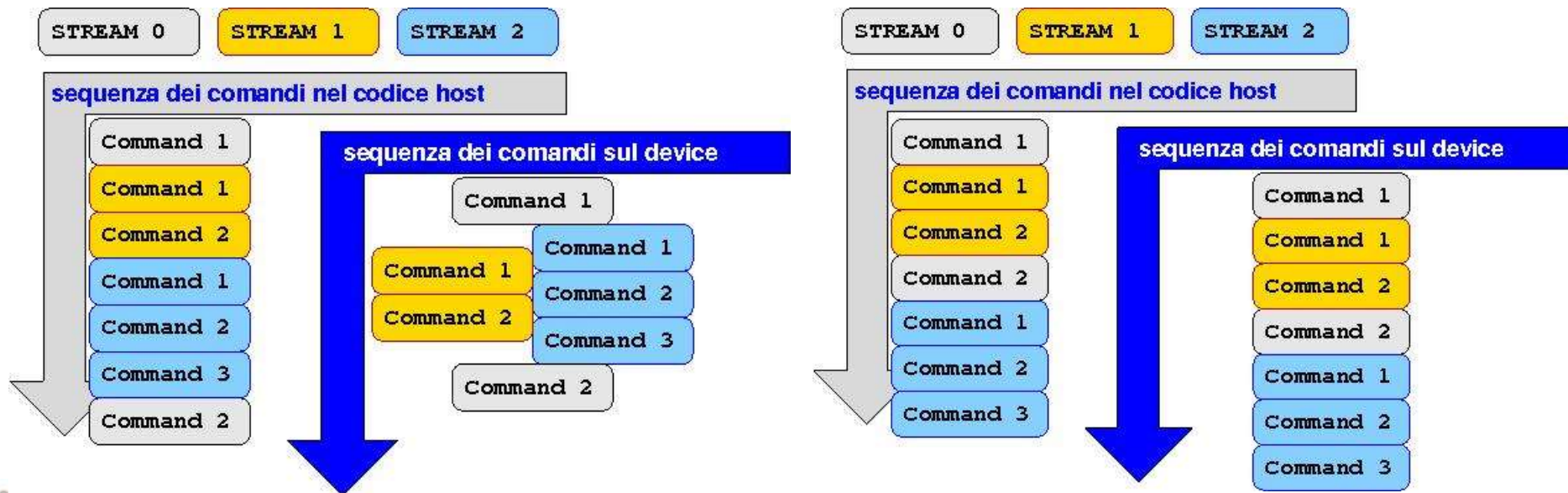
Before using the results from your CUDA kernel, some form of synchronization among *host* and *device* is required.

# CUDA Streams

- GPU operations are implemented in CUDA using execution queues, called **streams**
- any operation pushed in a stream will be executed only after all other operations in the same stream are completed (FIFO queue behaviour)
- operations assigned to different streams can be executed in any order with respect to each other
- The CUDA runtime provides a **default stream** (stream 0) which will be the default queue of all operation if not explicitly declared otherwise

# CUDA Streams

- All operations assigned to the default stream will be executed only after all preceeding operations already assigned to other streams are completed
- Any further operation assigned to other stream different from the default will begin only after all operations on the default stream are completed
- operations assigned to the default stream act as implicit synchronization barriers among other streams



# Synchronization

## ■ Explicit Synchronizations :

- `cudaDeviceSynchronize()`
  - Blocks host code until all operations on the device are completed
- `cudaStreamSynchronize(stream)`
  - Blocks host code until all operations on a stream are completed
- `cudaStreamWaitEvent(stream, event)`
  - Blocks all operations assigned to a stream until event is reached

## ■ Implicit Synchronizations :

- All operations assigned to the default stream
- All page-locked memory allocations
- All memory allocations on device
- All settings operation on device
- ...

# Managing CUDA Streams

## ■ Stream management:

- Constructor: `cudaStreamCreate()`
- Synchronization: `cudaStreamSynchronize()`
- Destructor: `cudaStreamDestroy()`

## ■ Stream allows for several execution modes, depending on the compute capability:

- concurrent execution of more than one kernel per GPU
- concurrent asynchronous data transfers in both H2D and D2H directions
- concurrent execution on device/host and data transfers from host and device

# Kernel Concurrent Execution

```
cudaSetDevice(0)
```

```
cudaStreamCreate(stream1)
```

```
cudaStreamCreate(stream2)
```

```
// lancio concorrente dello stesso kernel
```

```
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1)
```

```
Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2)
```

```
// lancio concorrente di kernel diversi
```

```
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp, out_1)
```

```
Kernel_2<<<blocks, threads, SharedMem, stream2>>>(inp, out_2)
```

```
cudaStreamDestroy(stream1)
```

```
cudaStreamDestroy(stream2)
```

# Asynchronous Data Transfers

- *host* memory must be of page-locked type (a.k.a pinned) in order to perform asynchronous data transfers between host and device
- CUDA runtime provides the following functions to handle page-locked memory:
  - `cudaMallocHost()` allocate page-locked memory on *host*
  - `cudaFreeHost()` free page-locked allocated memory
  - `cudaHostRegister()` turn *host* allocated memory into page-locked
  - `cudaHostUnregister()` turn page-locked memory into ordinary memory
- the `cudaMemcpyAsync()` function explicitly performs asynchronous data transfers between *host* and *device* memory
- data transfer operations should be queued into a stream different from the default one in order to be asynchronous
- Using page-locked memory allows data transfers between *host* and *device* memory with higher bandwidth performances



# Asynchronous Data Transfers

```
cudaStreamCreate(stream_a)
cudaStreamCreate(stream_b)

cudaMallocHost(h_buffer_a, buffer_a_size)
cudaMallocHost(h_buffer_b, buffer_b_size)

cudaMalloc(d_buffer_a, buffer_a_size)
cudaMalloc(d_buffer_b, buffer_b_size)

// trasferimento asincrono e concorrente H2D e D2H
cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size,
cudaMemcpyHostToDevice, stream_a)
cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size,
cudaMemcpyDeviceToHost, stream_b)

cudaStreamDestroy(stream_a)
cudaStreamDestroy(stream_b)

cudaFreeHost(h_buffer_a)
cudaFreeHost(h_buffer_b)
```

# Asynchronous Data Transfers

```
cudaStream_t stream[4];
for (int i=0; i<4; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 4 * size);

for (int i=0; i<4; ++i) {
    cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                  size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

    cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                  size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();

for (int i=0; i<4; ++i) cudaStreamDestroy(&stream[i]);
```

## Sequential Version



## Asynchronous Versions



# Concurrency

- Concurrency: when two or more CUDA operations proceed at the same time
  - **Fermi** : up to 16 kernel CUDA / **Kepler** : up to 32 kernel CUDA
  - 2 data transfers host/device (duplex)
  - concurrency with host operations
- Requirements for concurrency:
  - operations must be assigned to streams different from the default stream
  - host/device data transfers should be asynchronous and host memory must be page-locked
  - only if there are enough hw resources left to use (SharedMem, Registers, Blocks, PCIe bus, ...)
    - No kernel concurrency if all SM on the device are in use
    - data transfers won't take place if other transfers are still going on

## Serial :



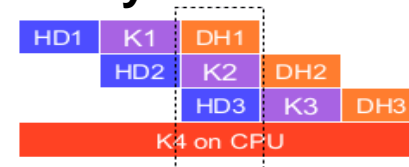
## 2 way concurrency :



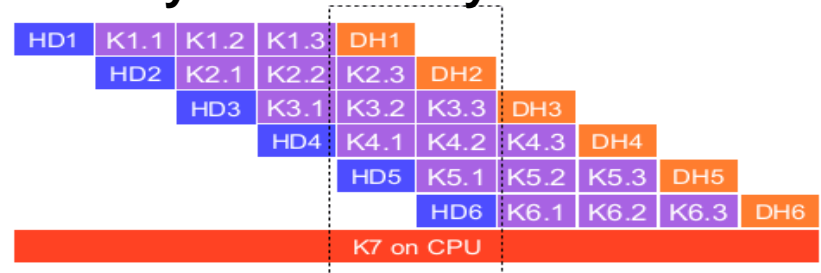
## 3 way concurrency :



## 4 way concurrency :



## 4/+ way concurrency :



# Stream Priorities

- Relative priorities of streams can be specified at creation
- If not specified, all streams get the same priority
- runtime will choose which operation start first among equivalent priority streams

```
// get the range of stream priorities for this device
int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low, &priority_high);

// create streams with highest and lowest priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking,
priority_high);
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking,
priority_low);
```

# Device Management

CUDA runtime allows to control more than one GPU device available on a computing node (multi-GPU programming):

- CUDA 3.2 and previous versions
  - a multi-thread or multi-process parallel paradigm was required to access and use more than one device
- CUDA 4.0 and later versions
  - new runtime API to select and to control all available devices from a serial program (single host core)
  - you can still use a parallel programming approach (multi-thread or multi-process):
    - each process or thread will be always able to access all devices
    - you can select which devices a thread/process can control

# Device Management

```
cudaDeviceCount(number_gpu)
cudaGetDeviceProperties(gpu_property, gpu_ID)
```

```
cudaSetDevice(0)
kernel_0 <<<threads, Blocks>>> (a, b, c)
```

```
cudaSetDevice(1)
kernel_1 <<<threads, Blocks>>> (d, e, f)
```

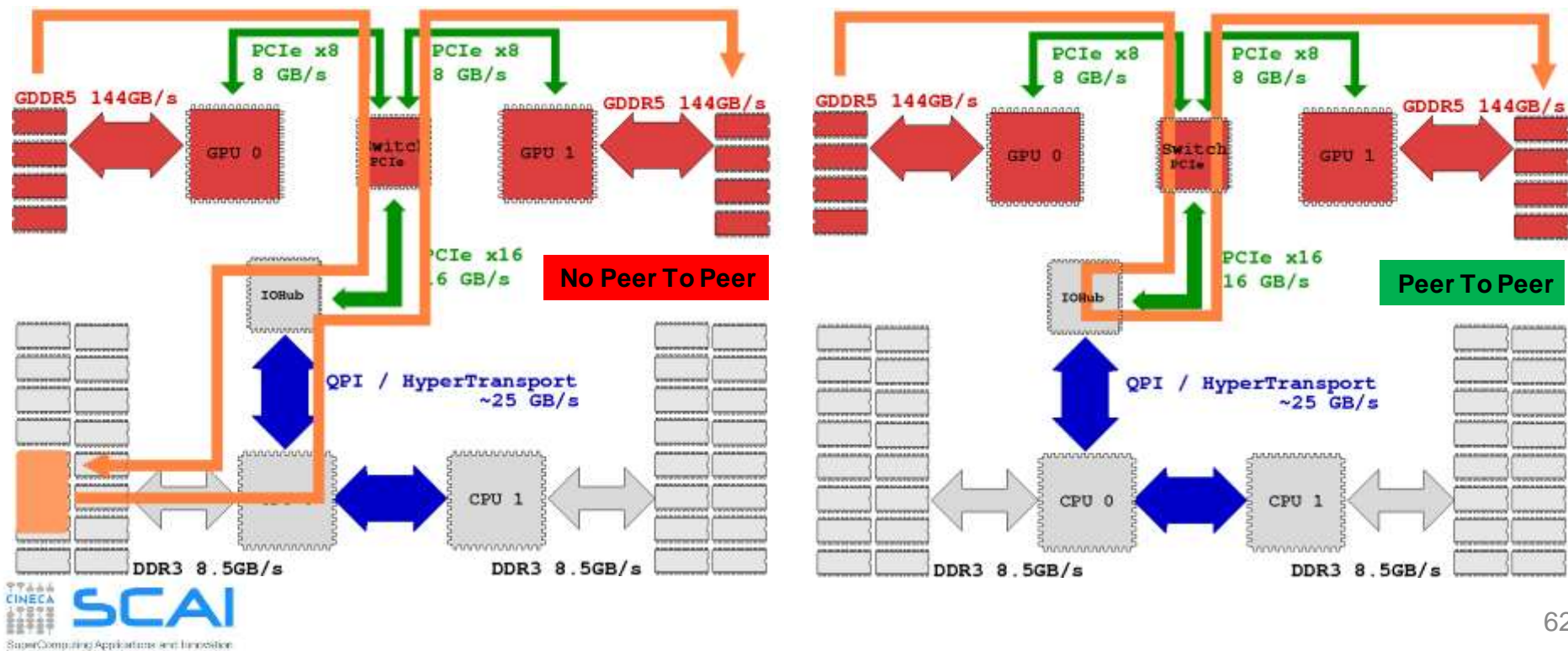
```
For each device:
    cudaSetDevice(device)
    cudaDeviceSynchronize()
```

CUDA runtime has API to :

- get information on available CUDA enabled devices
- get specifics of each CUDA enable device (cc, memory sizes, clock, etc)
- select a device and queue CUDA operations on that device
- manage synchronization among available devices

# Peer to Peer Transfers

- A *device* can directly transfer or access data to/from another *device*
- This kind of direct transfer is called Peer to Peer (P2P)
- P2P transfers are more efficient and do not require a *host* buffer
  - direct access avoid host memory copy



# Peer to Peer Transfer Pseudocode

```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

cudaSetDevice(gpuB)
cudaMalloc(buffer_B, buffer_size)

cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

If answer is true:
    cudaDeviceEnablePeerAccess(gpuB, 0)
    // la gpuA esegue la copia da gpuA a gpuB
    cudaMemcpyPeer(buffer_B, gpuB, buffer_A, gpuA, buffer_size)
    // la gpuA esegue la copia da gpuB a gpuA
    cudaMemcpyPeer(buffer_A, gpuA, buffer_B, gpuB, buffer_size)
```



# Peer to Peer Direct Access Pseudocode

```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

cudaSetDevice(gpuB)
cudaMalloc(buffer_B, buffer_size)

cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

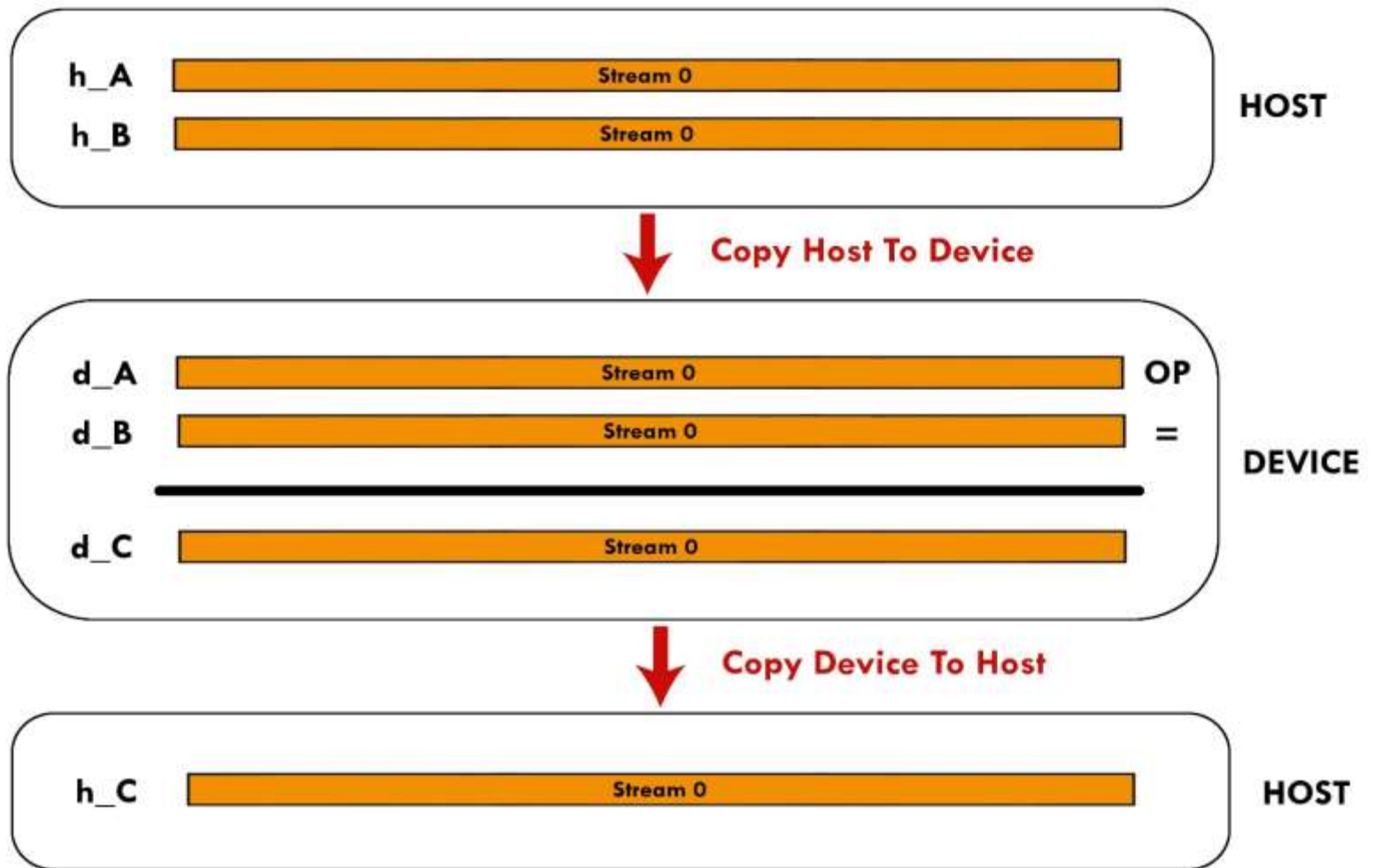
If answer is true:
    cudaDeviceEnablePeerAccess(gpuB, 0)
    // la gpuA esegue il kernel che accede sia alla sua memoria
    // che direttamente alla memoria di gpuB
    kernel<<<threads, blocks>>>(buffer_A, buffer_B)
```

# Hands-on Streams: naive version

Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Allocate `d_A`, `d_B`, `d_C` single precision arrays on the *device*
- Transfer data from `h_A` and `h_B` arrays on the `d_A` and `d_B` arrays
- Launch the `arrayFunc()` kernel which combine data from `d_A` and `d_B` and write results onto array `d_C`
- Copy back `d_C` array from *device* in `h_C` array on *host*
- Measure the total elapsed time to perform both kernel and memory transfers using `cudaEvents`
- Execute the `funcArrayCPU()` function which replicates the same CUDA kernel on host for result comparison
- Measure the elapsed time of the `funcArrayCPU()` function
- Compute the Speed Up of GPU implementation as CPU time / GPU time

# Hands-on Streams: naive version

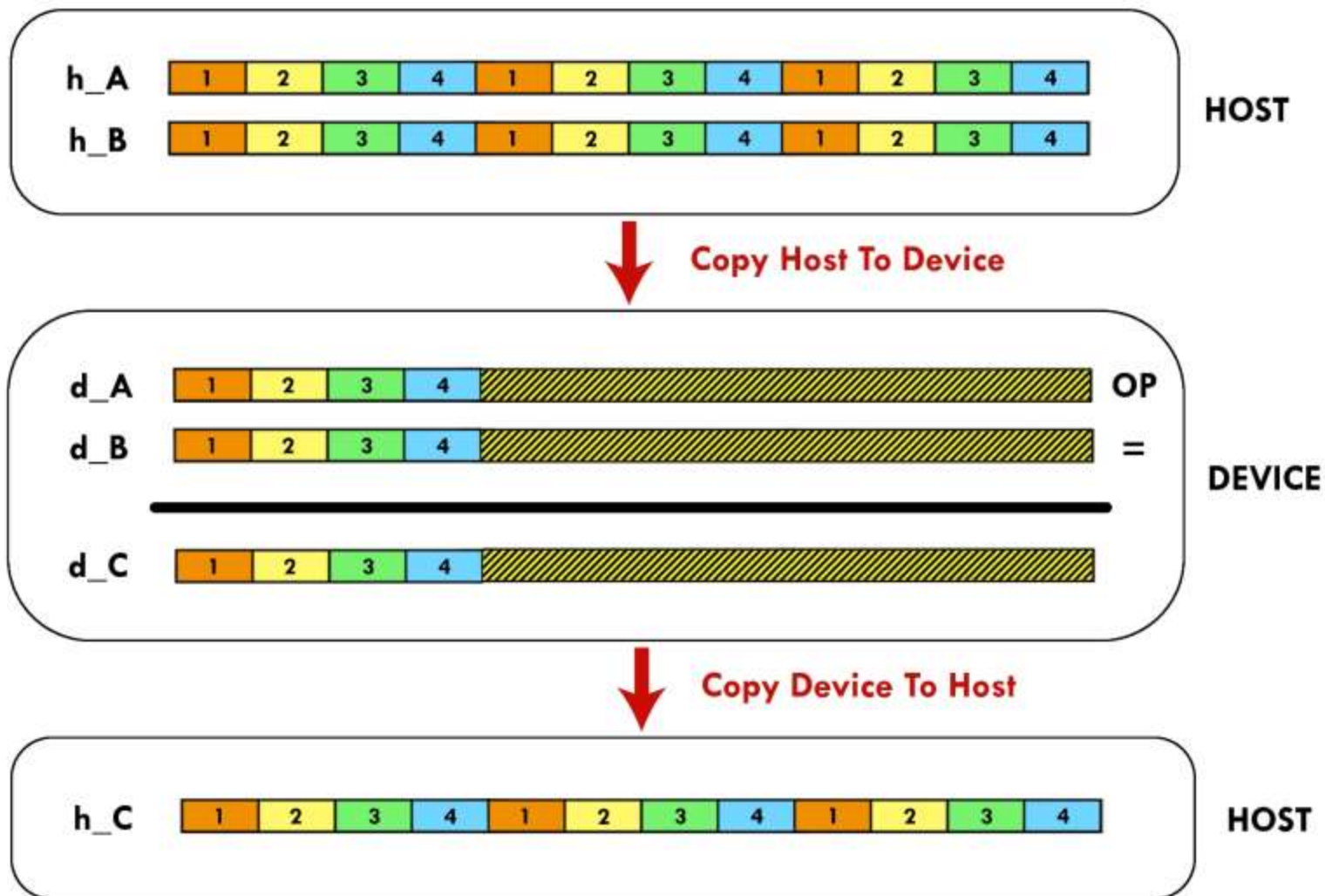


# Hands-on Streams: using cudaStreams

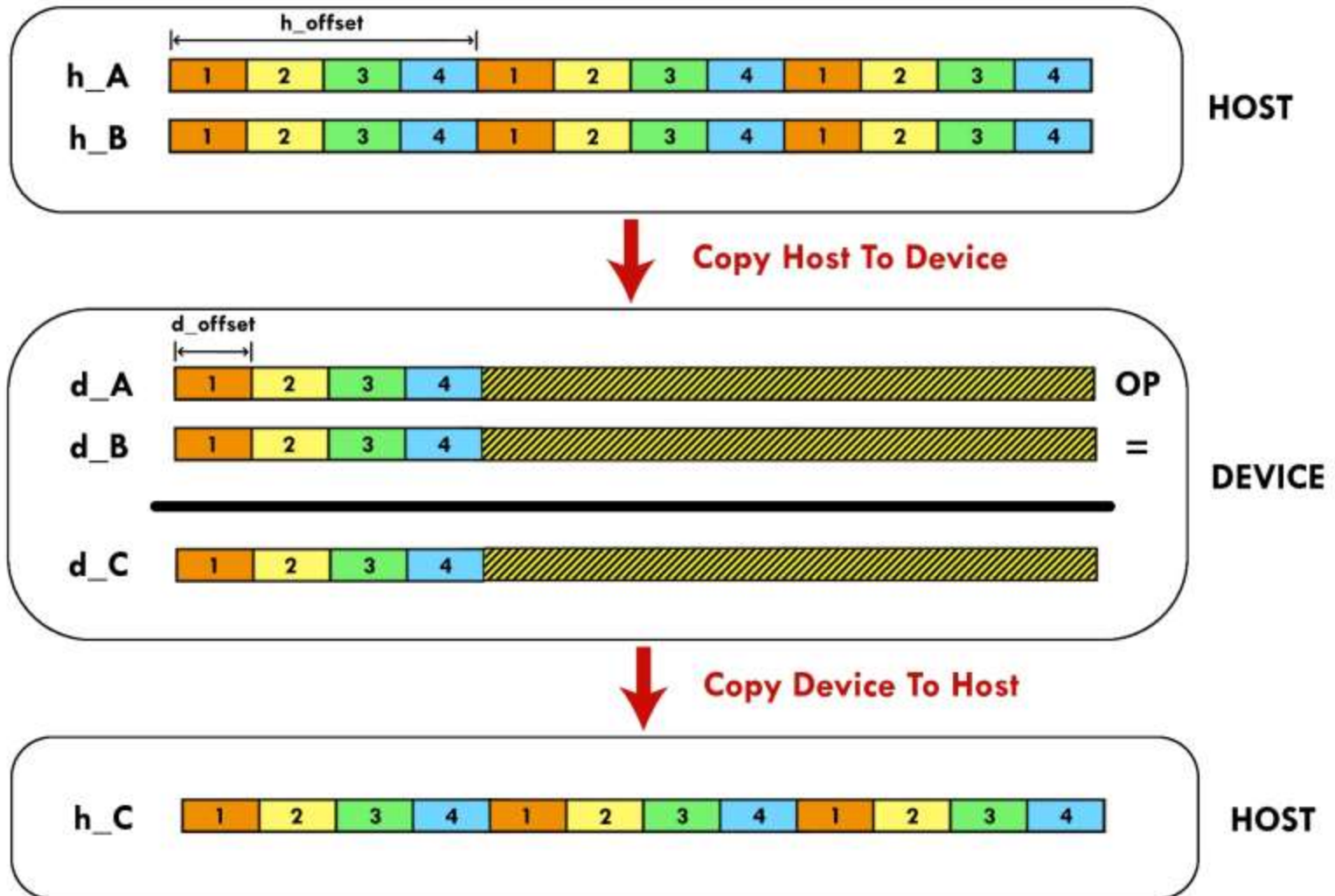
Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Split the elaboration of `h_A`, `h_B` arrays into chunks of `chunk_size` size elements
- Create `streams_number` of `cudaStream`
- Allocate `d_A`, `d_B`, `d_C` of `chunk_size * streams_number` size on the *device*
- Assign to each `cudaStream` the elaboration of each chunk. Each stream will:
  - copy a chunk of data from `h_A` and `h_B` on `d_A` and `d_B` buffers
  - Launch the kernel `arrayFunc`
  - Copy back to *host* the results from `d_C` into `h_C`
- Measure execution time and compare the speedup with respect *naïve* implementation
  - Try to change the number of active streams, the chunk size, etc...

# Hands-on Streams: using cudaStreams



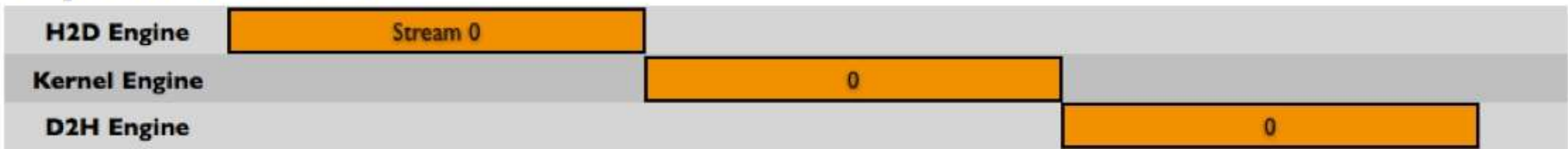
# Hands-on Streams: using cudaStreams



# Hands-on Streams: using cudaStreams

## Execution Time Lines

### Sequential Version



### Asynchronous Versions



Time →

# Hands-on Streams: using cudaStreams

## CUDA Runtime functions to implement the code (C for CUDA):

- `cudaError_t cudaStreamCreate(cudaStream_t *stream)`
- `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- `cudaError_t cudaDeviceSynchronize(void)`
- `cudaError_t cudaMemcpyAsync(void* dst, void* src, size_t nbyte, enum cudaMemcpyKind kind, cudaStream_t stream)`

## CUDA Runtime functions to implement the code (CUDA FORTRAN):

- `integer function cudaStreamCreate(stream)`  
    `integer :: stream`
- `integer function cudaStreamDestroy(stream)`  
    `integer :: stream`
- `integer function cudaDeviceSynchronize()`
- `integer function cudaMemcpyAsync(dst, src, nelements, kind, stream)`



# Hands-on Streams: cudaStreams and Multi-GPU

Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Split the elaboration of `h_A`, `h_B` arrays into chunks of `chunk_size` size elements
- Assign to each available GPU device a balanced number of chunks to process
- Create `streams_number` of `cudaStream`
- Allocate `d_A`, `d_B`, `d_C` of `chunk_size * streams_number` size on the *device*
- Assign to each `cudaStream` the elaboration of each chunk. Each stream will:
  - copy a chunk of data from `h_A` and `h_B` on `d_A` and `d_B` buffers
  - Launch the kernel `arrayFunc`
  - Copy back to *host* the results from `d_C` into `h_C`
- Measure execution time and compare the speedup with respect single GPU implementation