Introduction to Scientific Programming using GPGPU and CUDA

Day 2

Sergio Orlandini
s.orlandini@cineca.it

Mario Tacconi
m.tacconi@cineca.it

- 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:

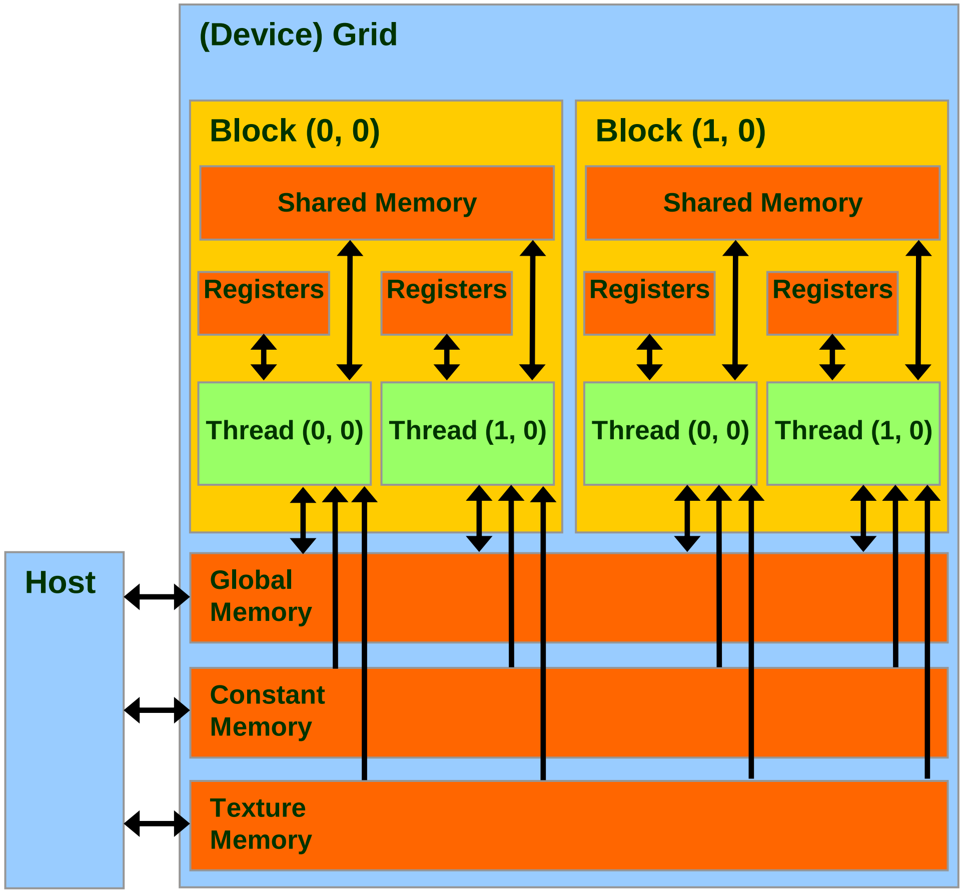
- resources of the SM assigned to its block:
  - **Registers**
  - **Shared Memory**

NB: thread belonging to different blocks cannot share these resources

- all memory type available on GPU:
  - **Global Memory**
  - **Costant 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

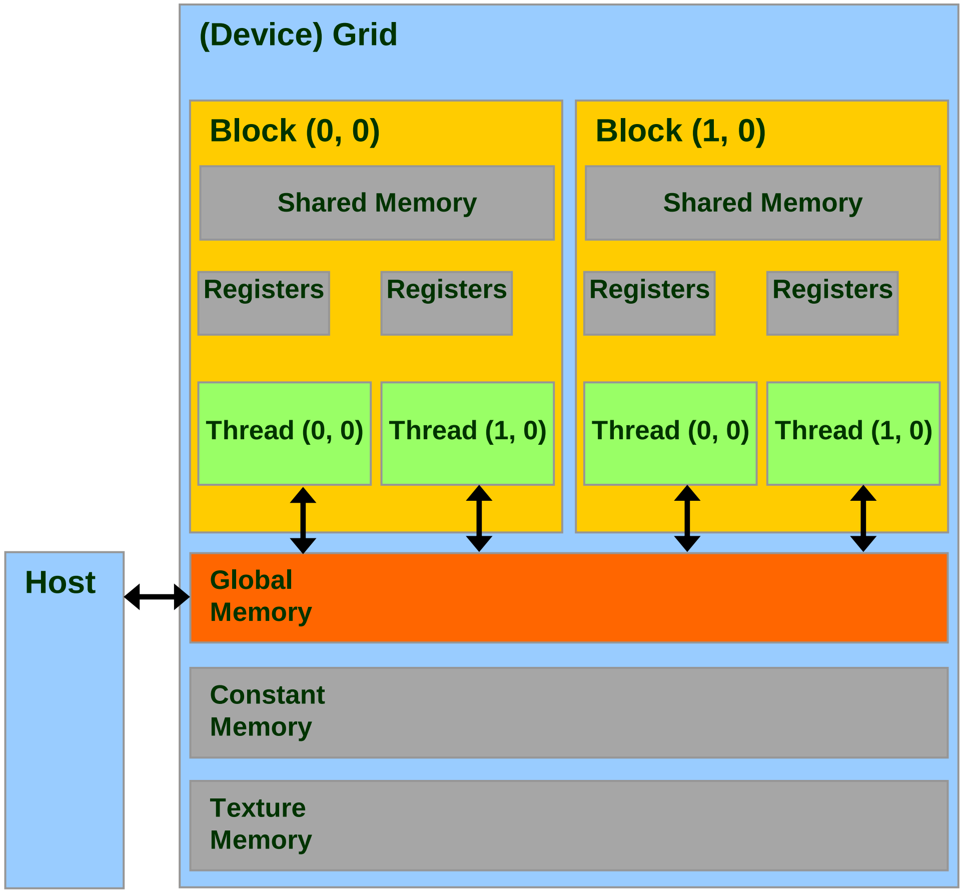


Global Memory

- ***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 use in read/write access from the CPU
  - **Very high bandwidth**

    Throughput up to 144-177 GB/s
  - **Very high latency**

    about 400-800 clock cycles



Declare Variable in *Global Memory*

- How to allocate a variable in Global Memory:

```
__device__  type  variable_name; // static

or dynamic allocation

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

```
type, device :: variable_name

or dynamic allocation

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

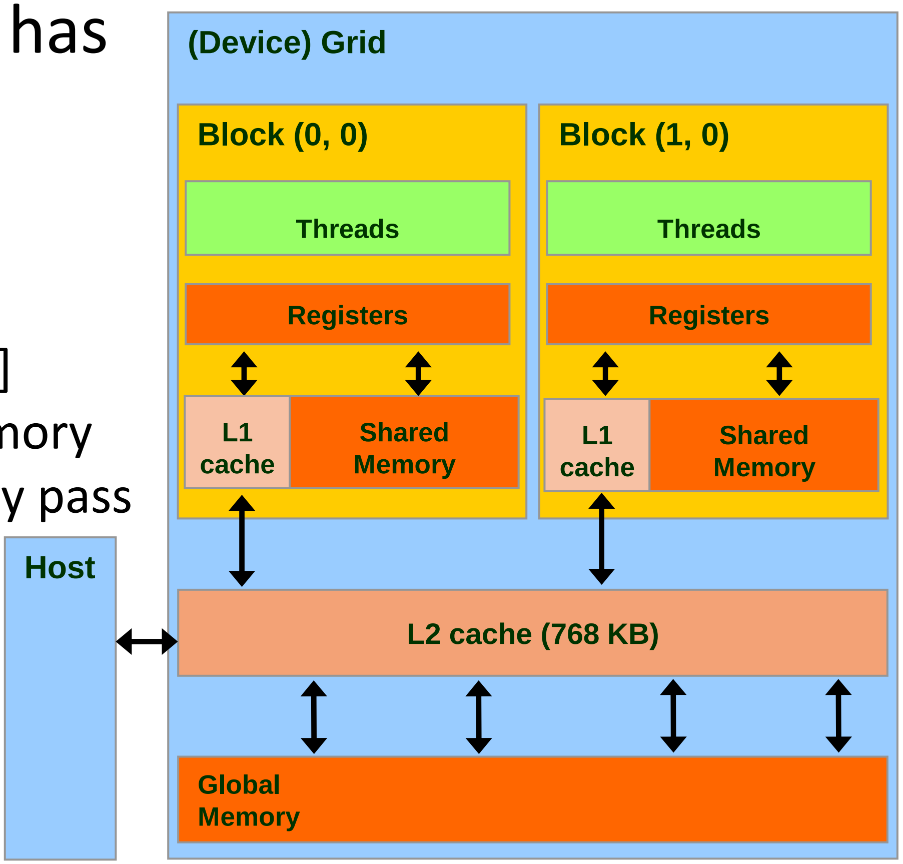
- Lifetime of the application
- Accessible by all threads of a CUDA grid and by the host

Cache Hierarchy for *Global Memory* Accesses

- Starting with the Fermi architecture, a cache hierarchy has been introduced

- 2 Levels of cache:
  - **L2** : share among all SM
    - **Fermi** [768 KB] / **Kepler** [1536 KB]
    - 25% less latency than Global Memory
    - NB : all accesses to global memory pass through L2 cache,

      also H2D/D2H memory transfers
  - **L1** : private to each SM
    - [**16/48 KB**] configurable
    - L1 + Shared Memory = 64 KB
    - **Kepler**: configurable at **32 KB**

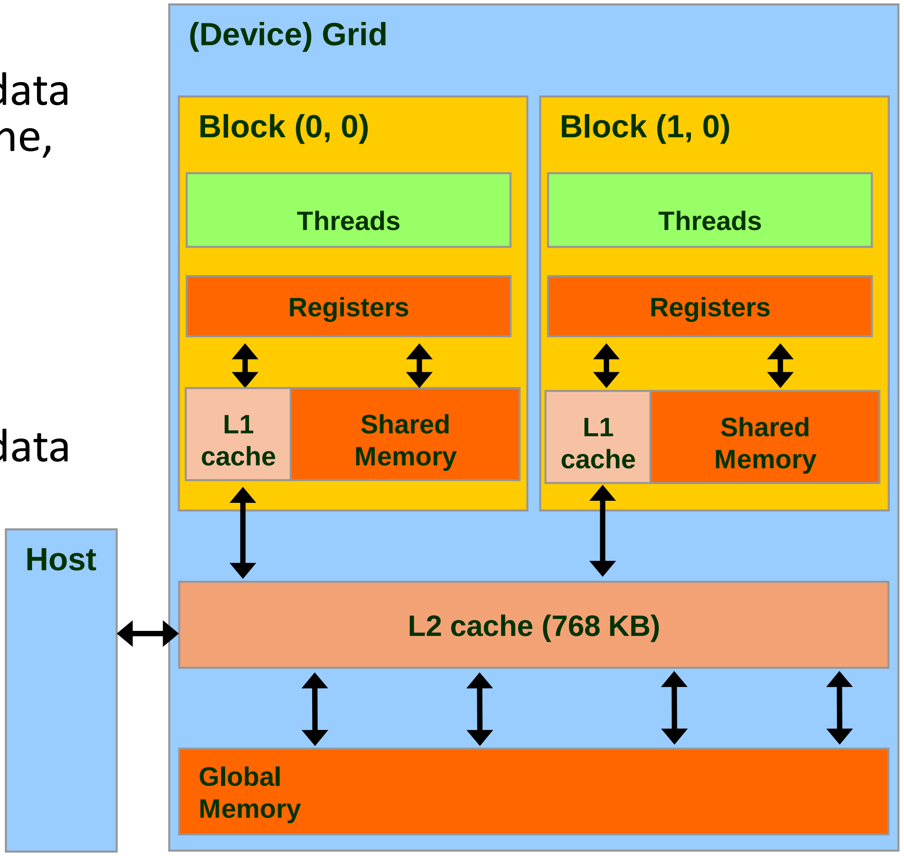


```
cudaFuncSetCacheConfig(kernel1, cudaFuncCachePreferL1);     // 48KB L1 / 16KB ShMem
cudaFuncSetCacheConfig(kernel2, cudaFuncCachePreferShared); // 16KB L1 / 48KB ShMem
```

Cache Hierarchy for *Global Memory* Accesses

Two different types of ***load*** operations:

- **Caching (default mode)**
  - when data is requested by some threads, data is first searched in L1 cache, then in L2 cache, then in global memory
  - cache line length is **128-byte**
- **Non-caching**
  - L1 cache is disabled
  - when data is requested by some threads, data is first searched in L2 cache, then in global memory
  - cache line length is **32-bytes**
  - Activated at *compile time* with option: `-Xptxas -dlcm=cg`



Just one type of ***store*** operation:

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

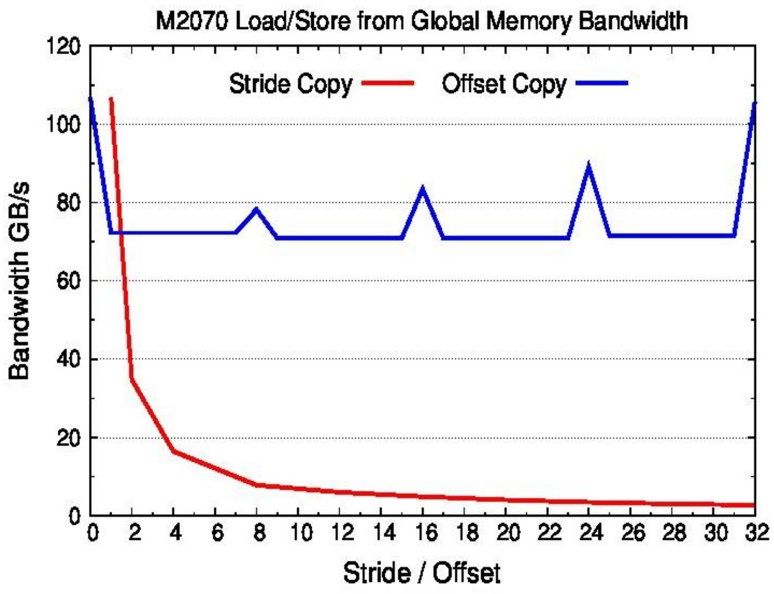
Global Memory Load/Store

```
// 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 copy	
Stride	**Bandwidth GB/s**
1	**106.6**
2	**34.8**
8	**7.9**
16	**4.9**
32	**2.7**

Offset copy	
Offset	**Bandwidth GB/s**
0	**106.6**
1	**72.2**
8	**78.2**
16	**83.4**
32	**105.7**

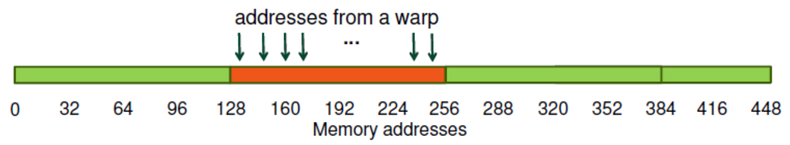


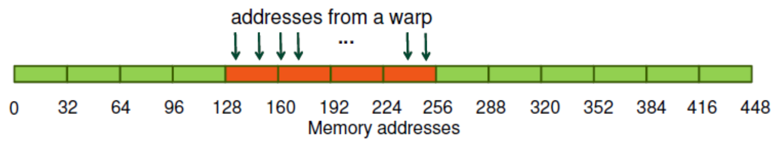
Measured on a M2070; Total elements = 16776960; Num. Blocks = 65535; Block length = 256

Loads from *Global Memory*

- All load/store request 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 calculate in which memory segments data resides
  3. *load/store* units start up requests for segment to transfer

Warp requires 32 consecutive 4-byte word aligned to segment (total 128 bytes)	
Caching Load	**Non-caching Load**
addresses fall whitin 1 cache line	addresses fall whitin 4 cache segments
128 bytes are moved across the bus	128 bytes are moved across the bus
bus utilization: **100%**	bus utilization: **100%**

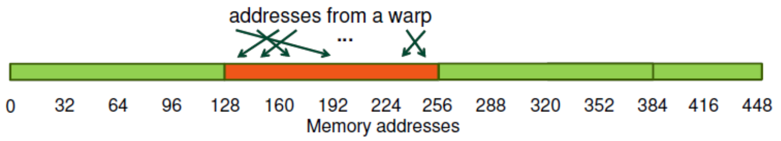


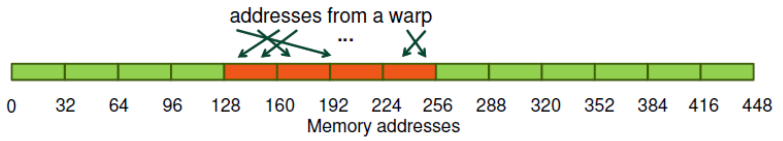


Loads from *Global Memory*

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

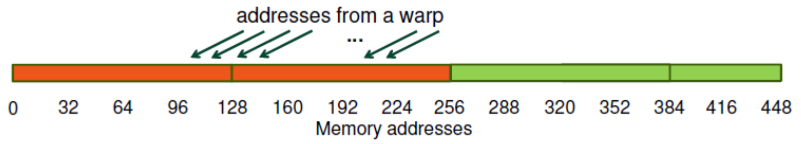
Caching Load	Non-caching Load
addresses fall within 1 cache line	addresses fall within 4 cache segments
128 bytes are moved across the bus	128 bytes are moved across the bus
bus utilization: **100%**	bus utilization: **100%**

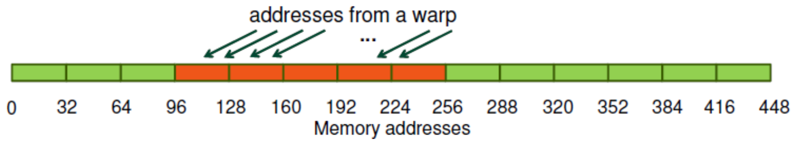




Warp requests 32 consecutive 4-bytes words not aligned to a segment (total 128 bytes)

Caching Load	Non-caching Load
addresses fall within 2 cache lines	addresses fall within at most 5 segments
256 bytes are moved across the bug	256 bytes are moved across the bus
bus utilization: **50%**	bus utilization: at least **80%**

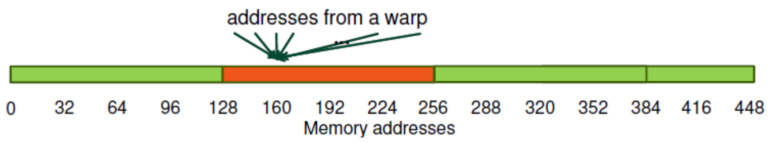


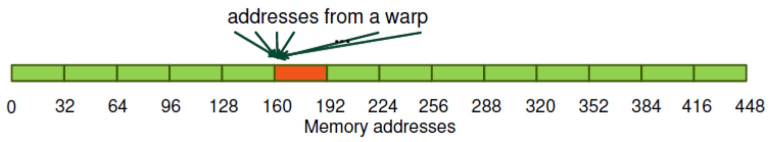


Loads from *Global Memory*

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

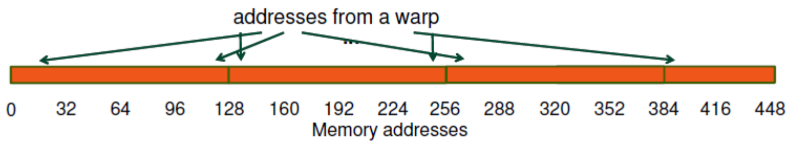
Caching Load	Non-caching Load
addresses fall within a single cache line	addresses fall within a single segment
128 bytes are moved across the bus	32 bytes are moved over the bus
bus utilization: **3.125%**	bus utilization: **12.5%**

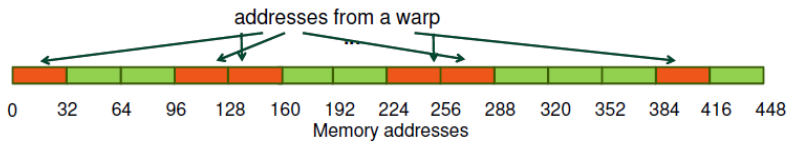




Warp requests 32 not contiguous 4-bytes words (total 128 bytes)

Caching Load	Non-caching Load
addresses fall within N different cache lines	addresses fall within N different segments
N*128 bytes are moved across the bus	N*32 bytes are moved across the bus
bus utilization: **128 / (N*128)**	bus utilization: **128 / (N*32)**





CINECA SCAI SuperComputing Applications and Innovation

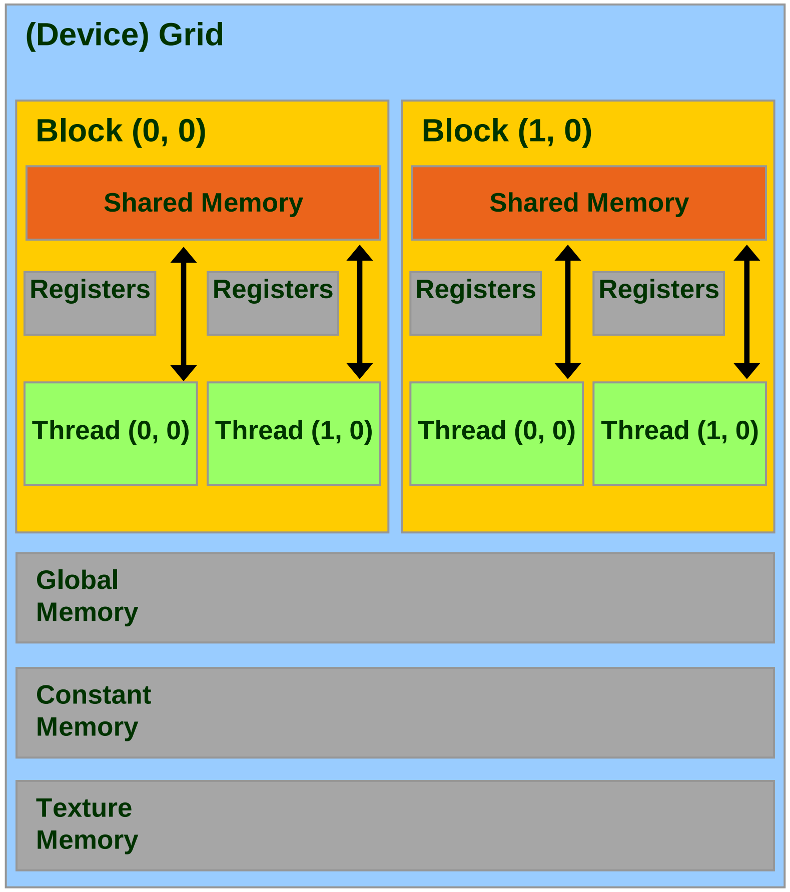
Data alignment in *Global Memory*

- 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 bytes 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, heigth = 64;
float *devPtr;
int pitch;
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
  - Accessible in read/write mode for only threads of a block
  - Alike a cache memory under the direct control of the programmer
  - Its status is not mantained 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 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

or dynamic allocation

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

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

- Lifetime of CUDA block of threads

  (NOT persistent along kernel launch!)
- Accessible only by threads of the same block

Thread Block Synchronization

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

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

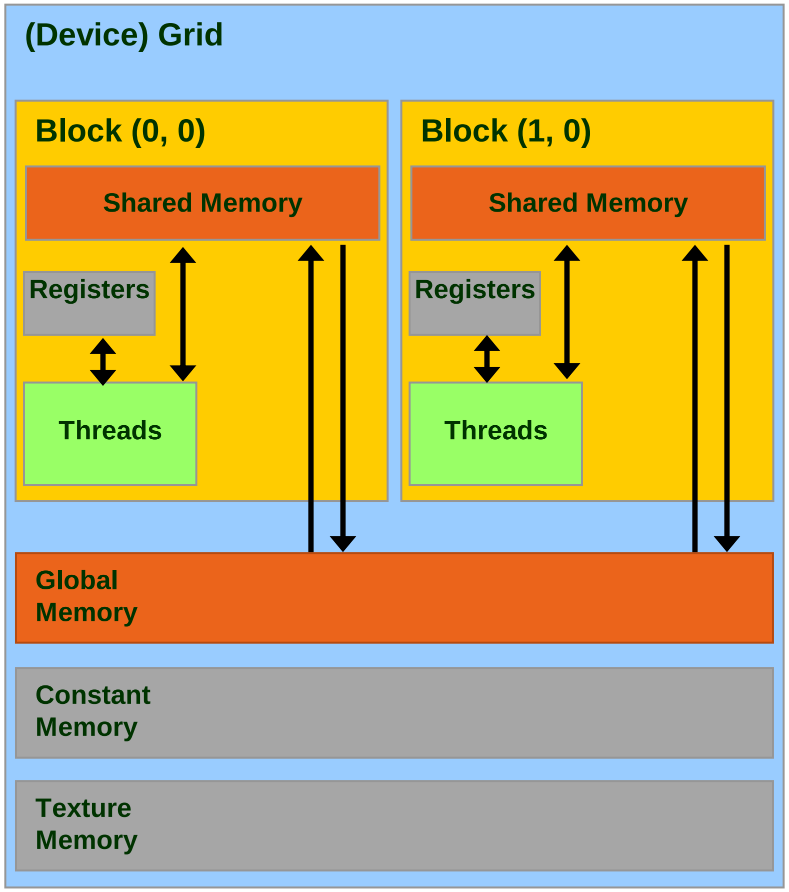
which blocks execution until all other threads reach the same call location

- NB: 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"

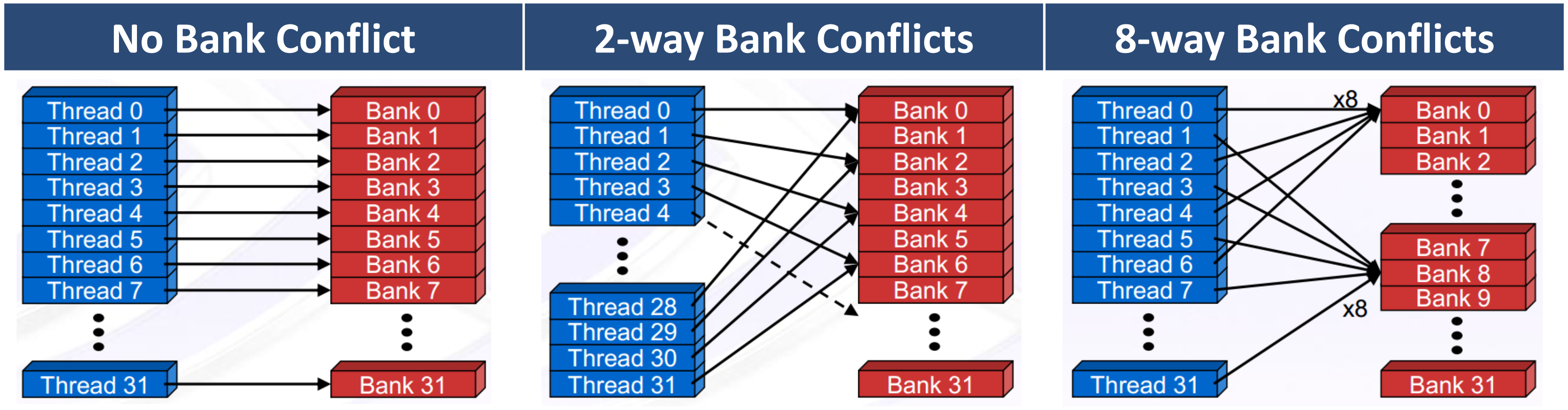
Using *Shared Memory* for Thread Cooperation

- Threads belonging to the same block can cooperate togheter using the shared memory to share data
  - if a thread needs some data which has been already retrived by another thread in the same block, this data can be shared using the shared memory
- Typical Shared Memory usage:
  1. declare a buffer residing on shared memory (this buffer is per block)
  2. load data into shared memory buffer
  3. synchronize threads so to make sure all needed data is present in the buffer
  4. performe operation on data
  5. synchronize threads so all operations have been performed
  6. write back results to global memory



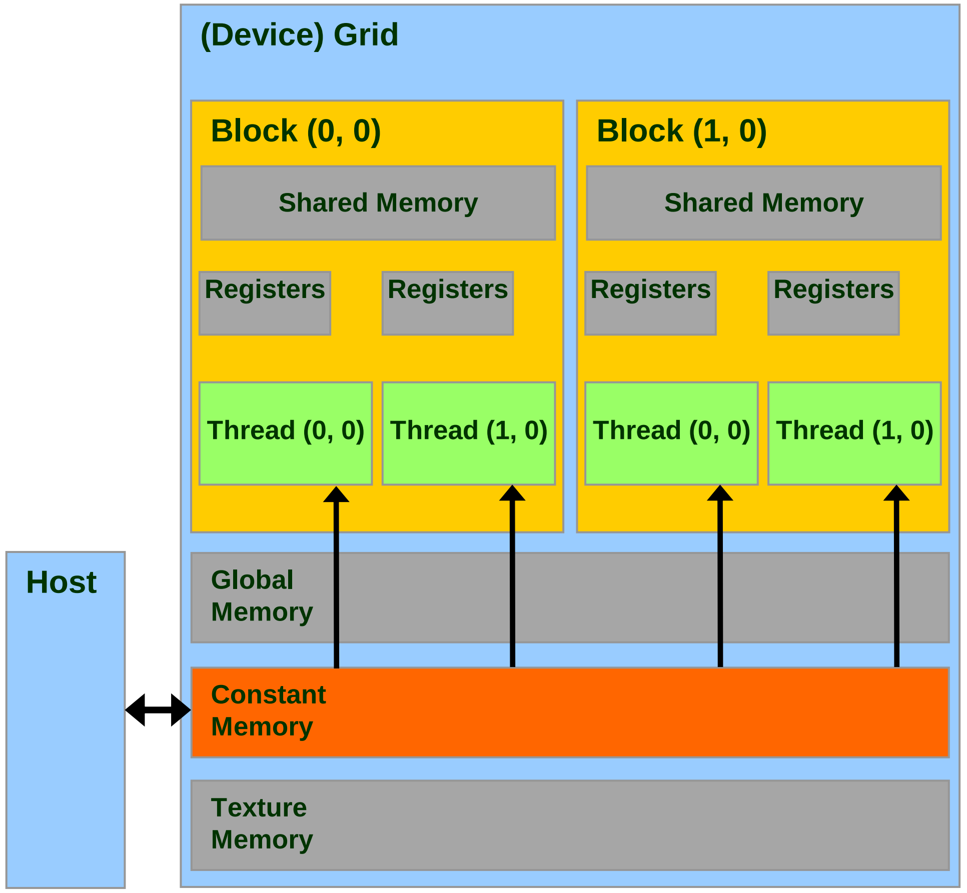
Shared Memory and Bank Accesses

- Shared memory has 32 banks organized such that 32-bit words map a 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, access is executed with only one transaction
  - **Bank Conflict**: if two or more threads requests different data belonging to the same bank, each access is serialized



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 efficient when all *thread* of a *warp* request the same memory address
  - Constant memory is initialized from host code using a special CUDA API
- Specifications:
  - Dimension : **64 KB**
  - Throughput: 32 bits per warp every 2 clock cycles



Constant Memory Allocation

```
__constant__  type  variable_name; // static

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

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

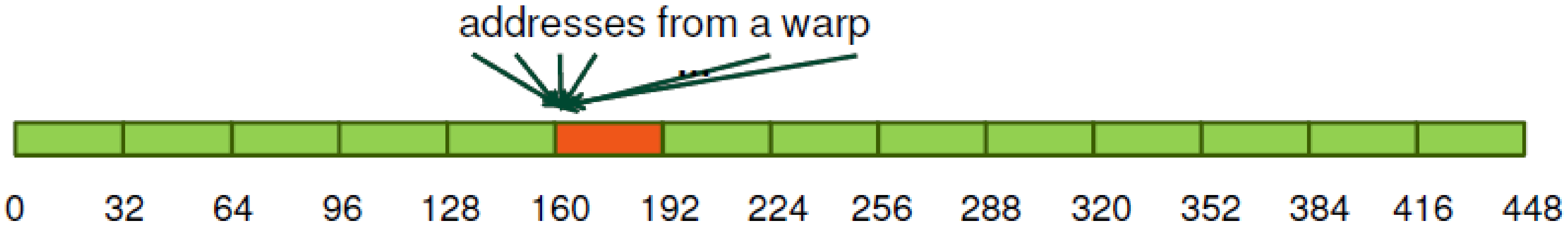
```
type, constant :: variable_name

! 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 kernel

Accessing *Constant Memory*

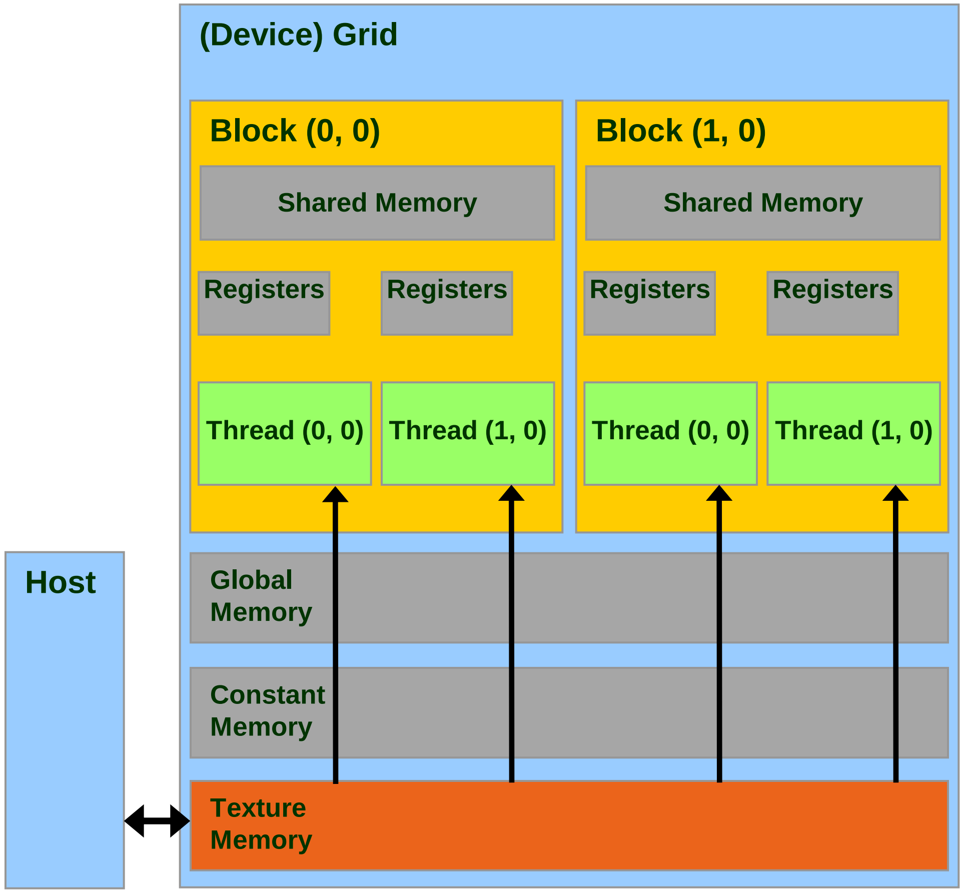
Suppose a kernel is launched using 320 warps per SM and all threads requests the same data

- 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



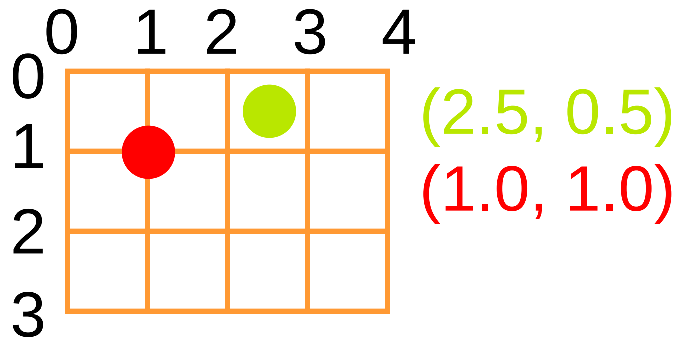
Texture Memory

- **Texture Memory** is a basic graphic rendering functionality
- as for constant memory, data actually reside in global memory, but is fetched across a 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



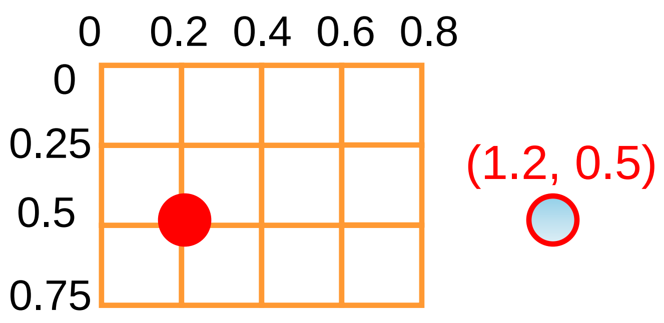
Texture Memory Addressing Features

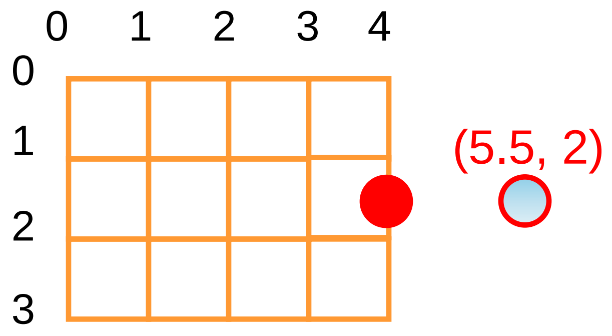
- integer 1D: [0,N-1]
- normalized 1D: [0,1-1/N]
- available interpolations:
  - floor, linear, bilinear
  - weights are 9 bit



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:

  `texture<datatype, dim> d_a_texRef;`

  `datatype` cannot be a double; `dim` can be 1, 2 or 3

- 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):

  `cudaUnbindTexture(d_a_texRef);`

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:

CINECA SCAI SuperComputing Applications and Innovation

Texture Usage Example

```
__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>(); // CREO DESC
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***

- The Kepler architecture (cc 3.5) enables global memory read through the *texture cache* :
  - without using a 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];
}
```

CINECA SCAI SuperComputing Applications and Innovation

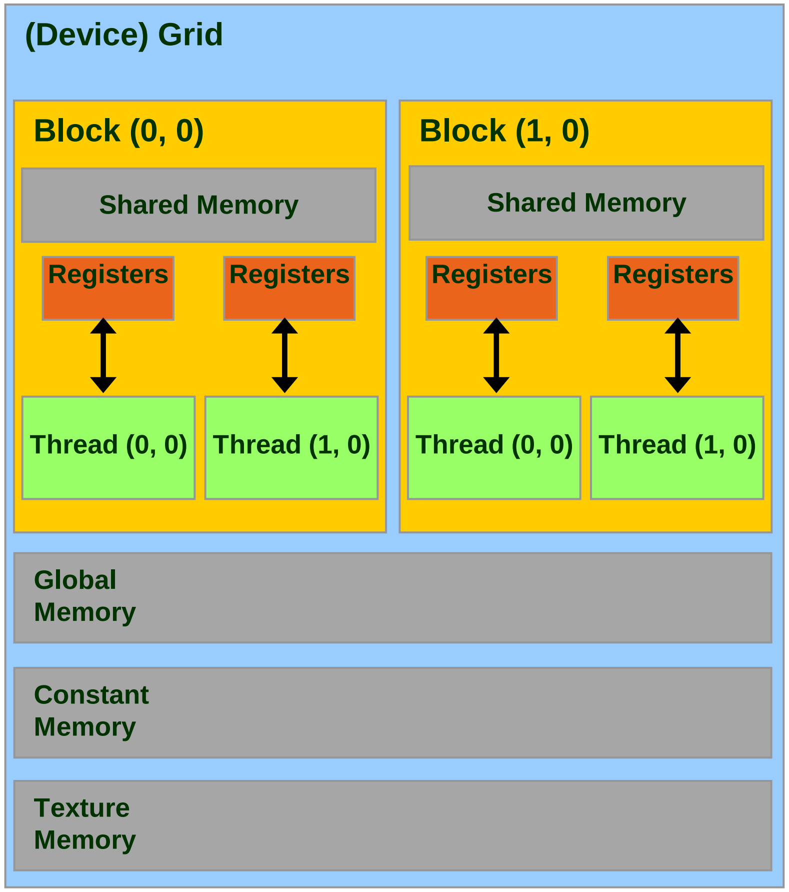
Registers

- **Registers** are used to store scalars or small array variables with frequent access by each thread
- **Fermi** : 63 registers per thread / 32 KB
- **Kepler** : 255 registers per thread / 64 KB
- WARNING:
  - Less registers a kernel needs, more blocks can be assigned to a SM
  - Attention to *Register Pressure*: can be a limiting factor
  - Number of registers per kernel can be limited during *compile time:*

    --maxregcount max_registers
  - 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 place 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), can move variables into local memory
- Local memory is often mapped to global memory
  - using same *Caching* hierachies (L1 for read-only variables)
  - facing 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]
...
```

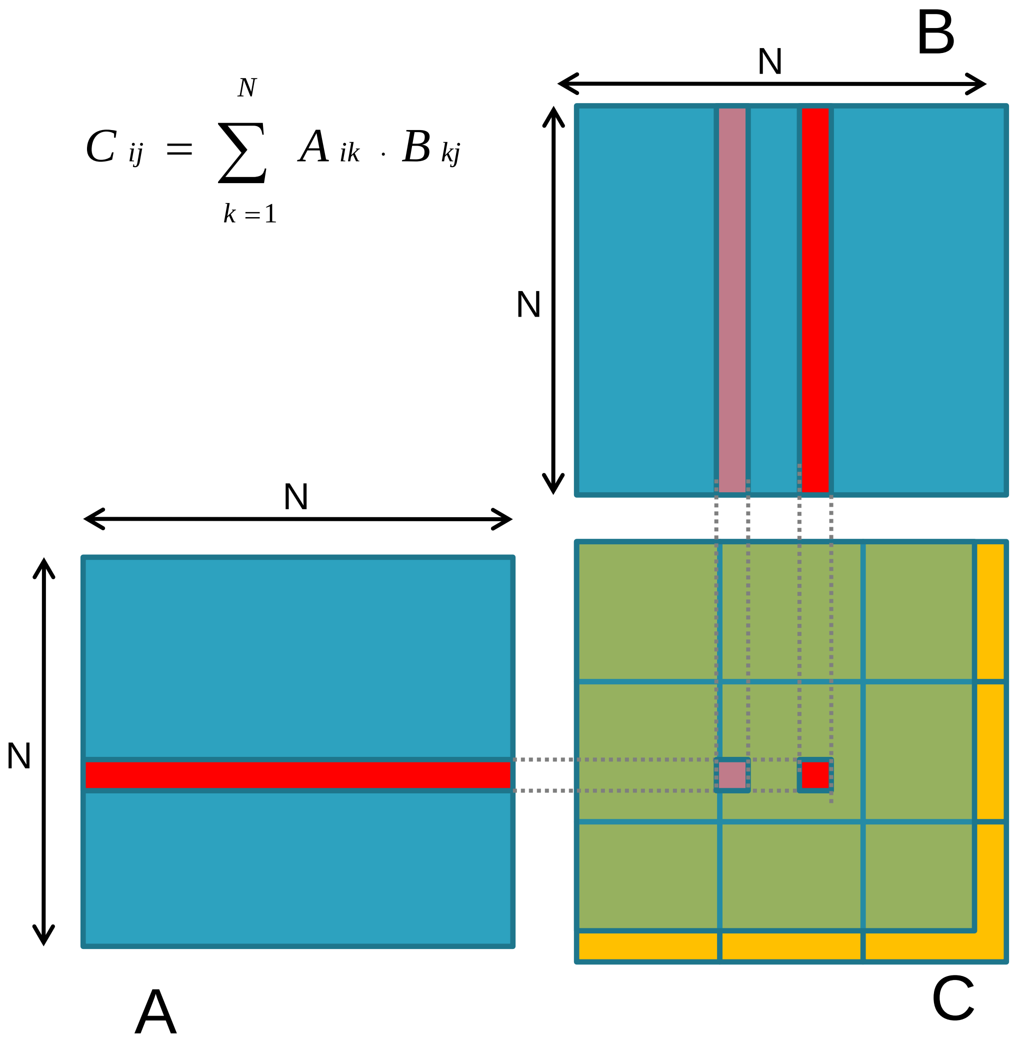
- Matrix-Matrix Product
  - limits of global memory implementation
  - using shared memory
  - implementation guidelines

SuperComputing Applications and Innovation

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)
- NB: every element of C shares same row or colum retrived 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 retrive just one data element data in parallel and store it into shared memory
  - when all threads 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
- Unfortunatly shared memory size is small
  - 16/48 KB depending on the compute capability

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$



Matrix-matrix using Shared Memory

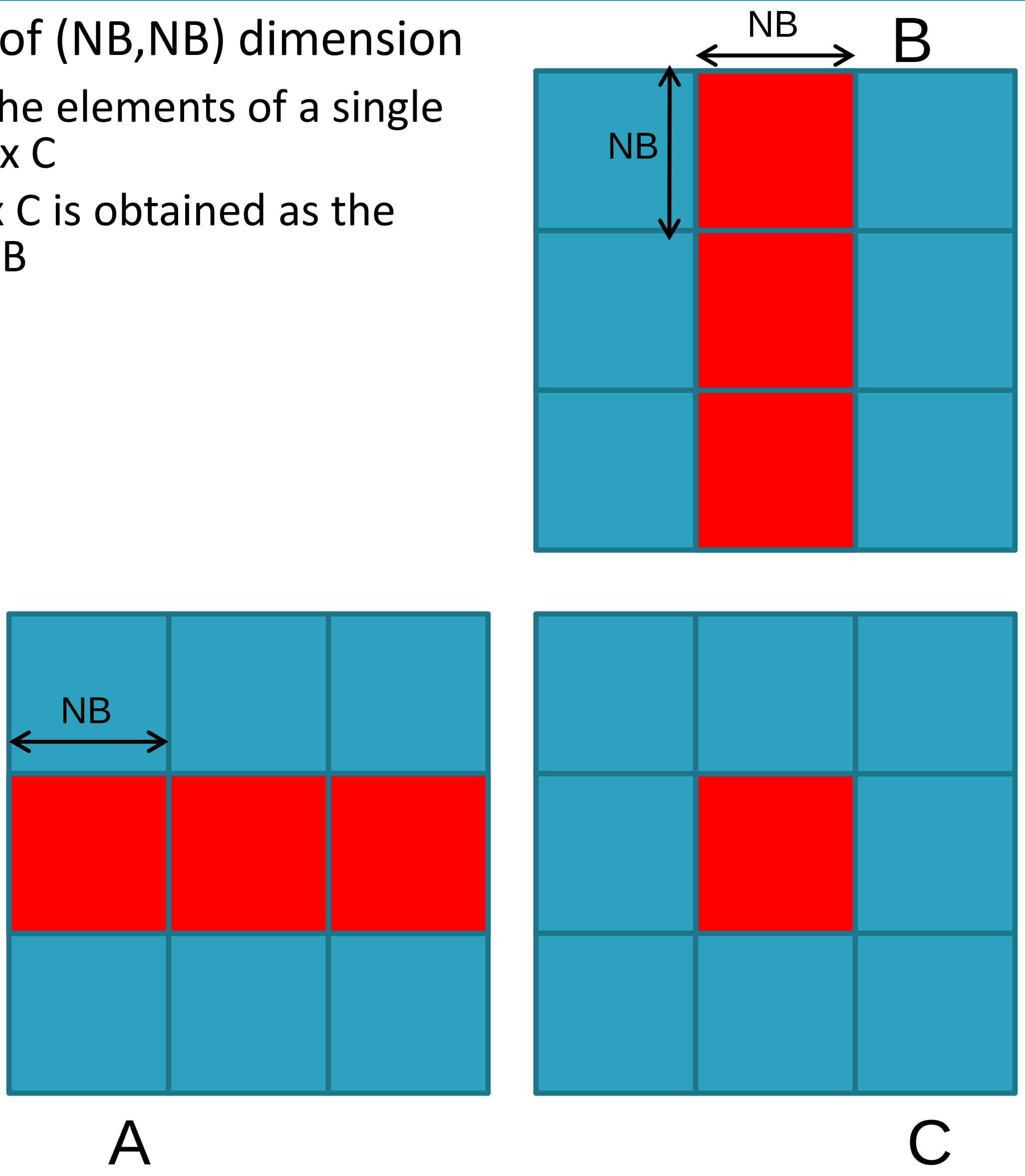
- Let’s solve the problem using blocks of (NB,NB) dimension
  - each CUDA thread block computes the elements of a single matrix block of size (NB·NB) of matrix C
  - each resulting matrix block of matrix C is obtained as the product of all sub-matrices of A and B

$$C_{ij} = \sum_{S=1}^{N/NB} \sum_{k=1}^{NB} As_{ik} \cdot Bs_{kj}$$

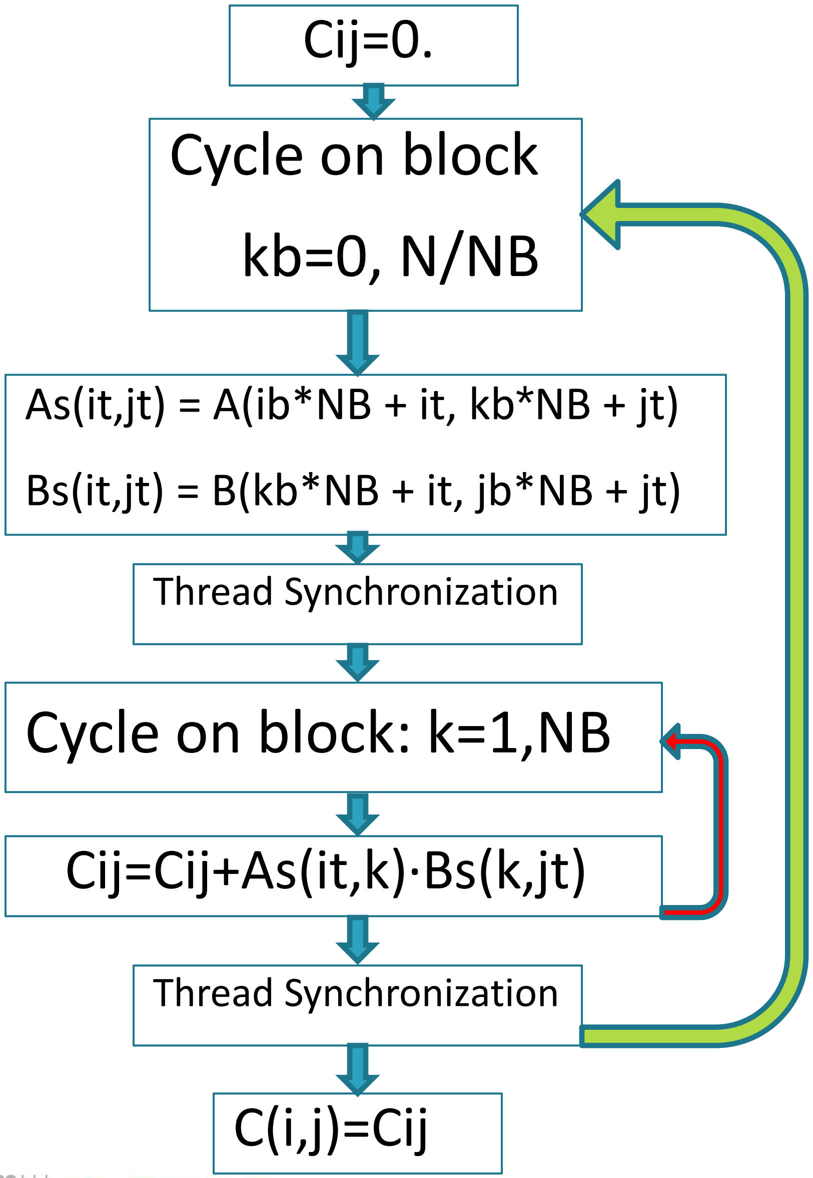
The kernel is divided in two phases:

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

- Elements of each sub-block C are accumulated using local variables in registers, then stored in global memory
- Threads synchronizations are required
  - after the load of sub-block of matrix A and B, in order to grant all data is available for sub-block matrix product
  - after the partial sub-block matrix product, in order to grant that next load of other sub-block will not overwrites elements not yet used in current block evaluation



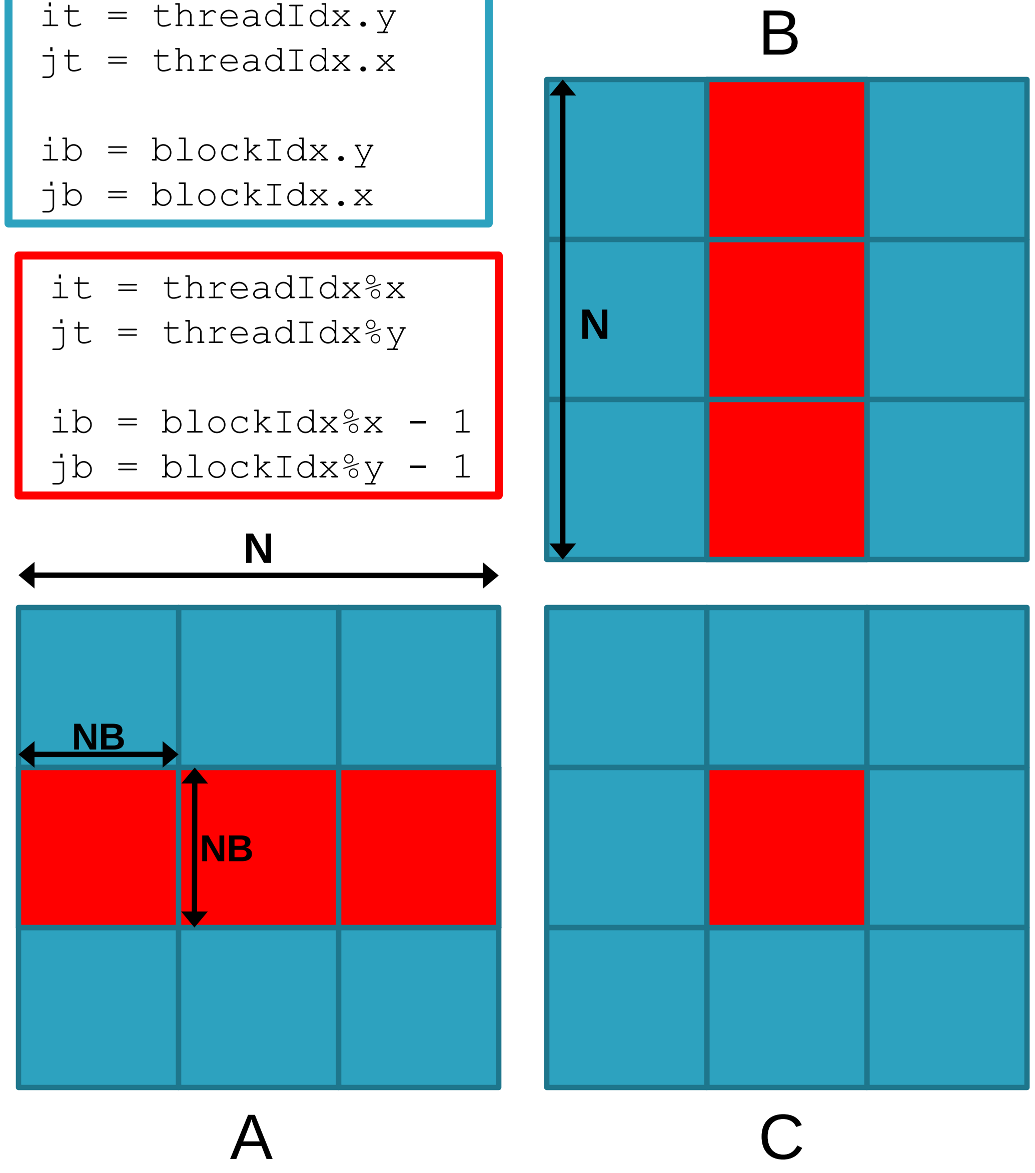
Matrix-matrix using Shared Memory: Flow



```
Cij=0.
Cycle on block
    kb=0, N/NB
As(it,jt) = A(ib*NB + it, kb*NB + jt)
Bs(it,jt) = B(kb*NB + it, jb*NB + jt)
Thread Synchronization
Cycle on block: k=1,NB
Cij=Cij+As(it,k)·Bs(k,jt)
Thread Synchronization
C(i,j)=Cij
```

```
it = threadIdx.y
jt = threadIdx.x

ib = blockIdx.y
jb = blockIdx.x
```

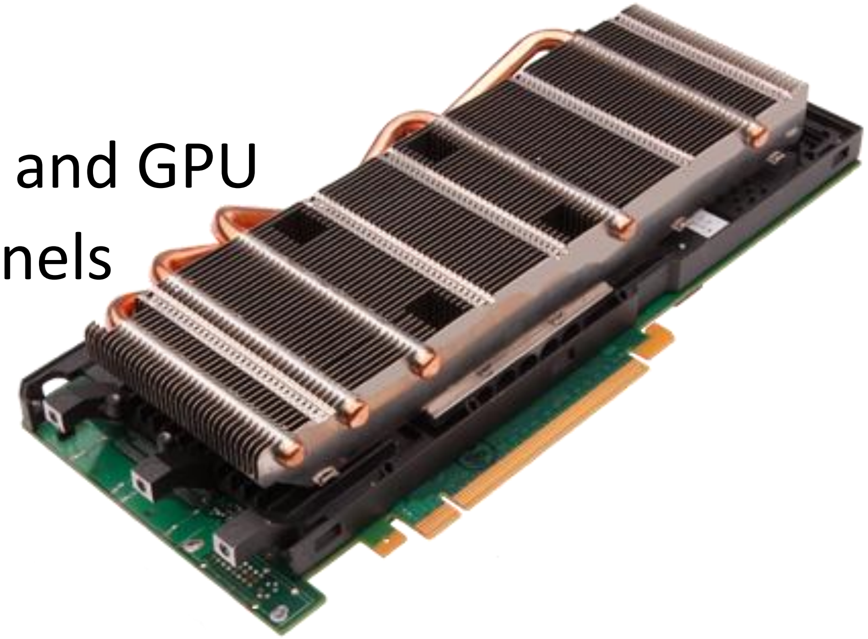
```
it = threadIdx%x
jt = threadIdx%y

ib = blockIdx%x - 1
jb = blockIdx%y - 1
```



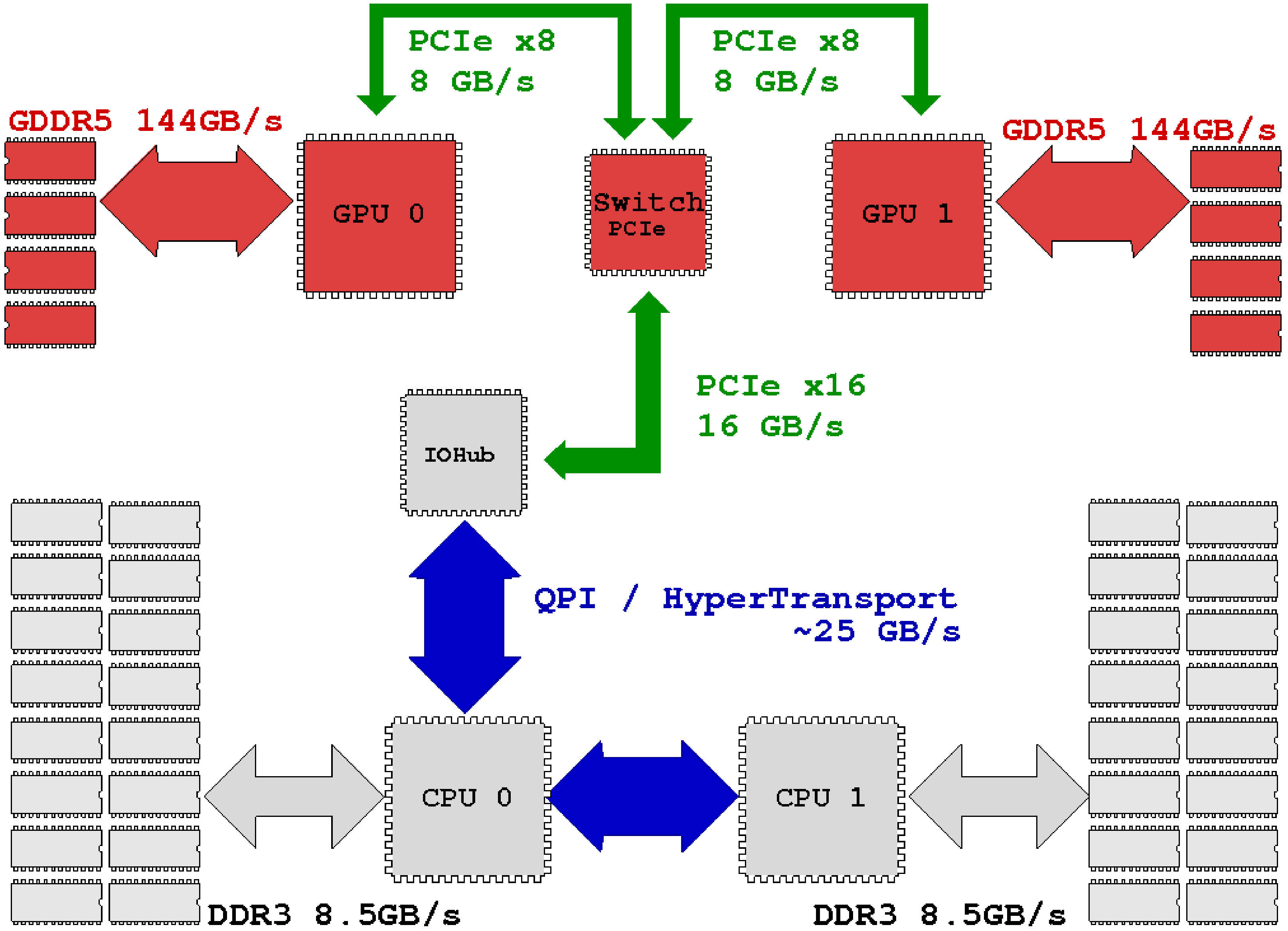
Matrix-matrix using Shared Memory: Kernel

```
// 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;

}
```

- Synchronous and Asynchronous API
- Concurrent Execution
- CPU and GPU interaction
  - concurrent execution on CPU and GPU
  - overlapping transfers and kernels
- Multi-device management
- GPU/GPU interactions



Connection Scheme of *host/device*



SuperComputing Applications and Innovation

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*

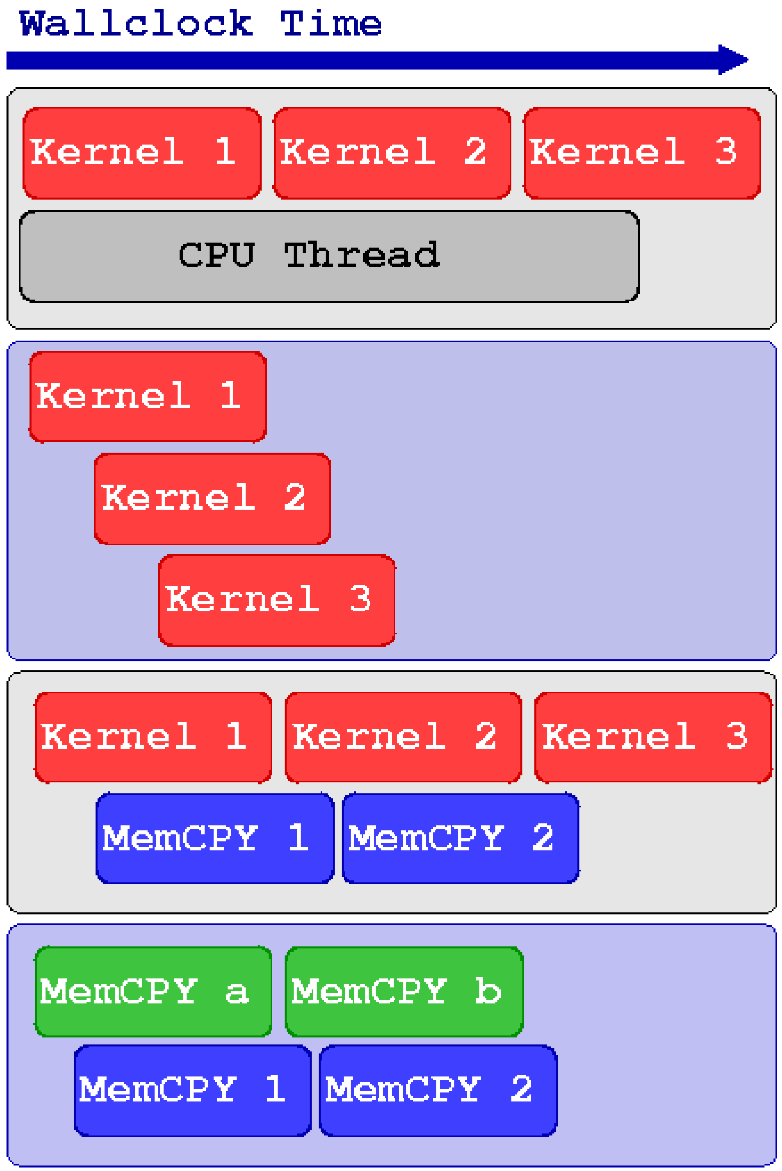
- **Non-blocking** (asynchronous):
  return control to *host* immediatelly, while its execution proceeds on *device*
  - kernel launches
  - memory transfers < 64KB
  - memory initialization on *device* (cudaMemset)
  - memory copies from *device* to *device*
  - explicit asynchronous memory transfers

- CUDA API provides asynchronous versions of their counterpart synchronous functions
- Asynchronous functions allows to set up concurrent execution of many operations on *host* and *device*

Concurrent and Asynchronous Execution

Asynchronous functions allows to expose concurrent executions:

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



Example of Concurrent Execution

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

// work on CPU while GPU is working
CPU_Function()

// Stop CPU until GPU has finished to compute
cudaDeviceSynchronize()

// Use device results on host
CPU_uses_the_GPU_kernel_results()
```

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

Before using results from CUDA kernel, synchronization between *host* and *device* is required.

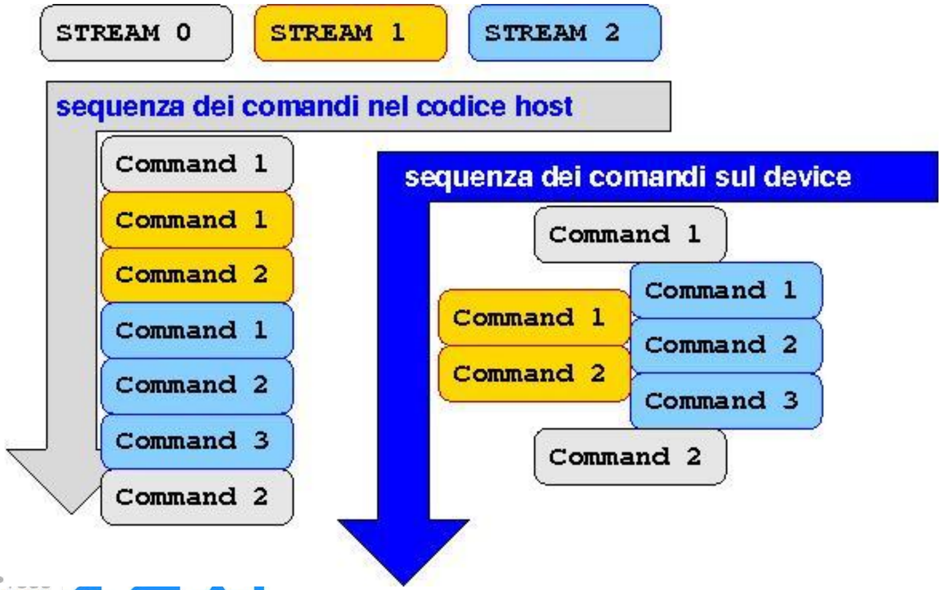
CINECA SCAI SuperComputing Applications and Innovation

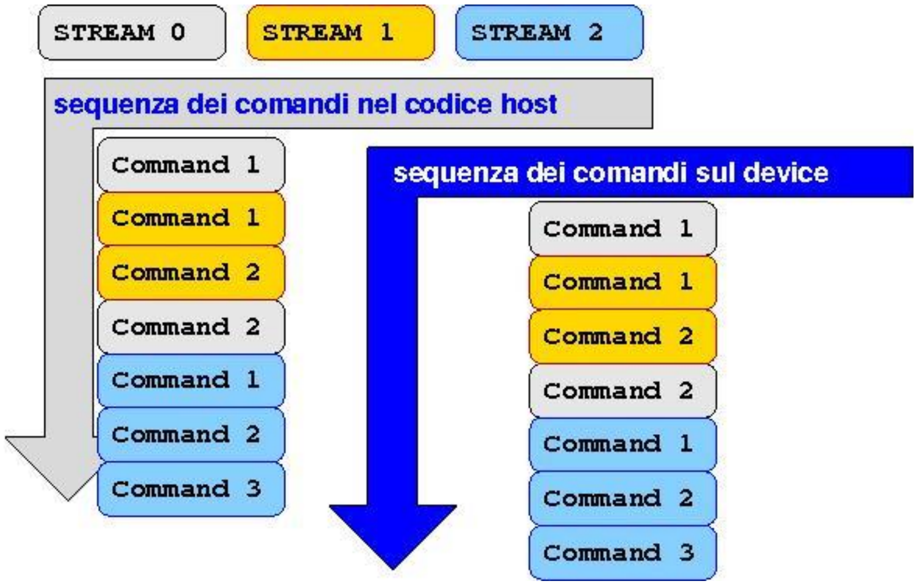
CUDA Streams

- GPU operations are implementated in CUDA using execution queues, called **streams**
- Each 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 each other
- CUDA runtime provides a **default stream** (aka stream 0) which will be the default queue of all operation if otherwise is not explicitly declared

CUDA Streams

- All operations assigned to the default stream will be executed only after all preceeding operations assigned to other streams are completed
- Any further operation assigned to stream different from 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

- **<u>Explicit Synchronizations</u>** :
  - `cudaDeviceSynchronize()`
    - Blocks host code until all operations on 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
- **<u>Implicit Synchronizations</u>** :
  - All operations assigned to the default stream
  - Page-locked memory allocations
  - Memory allocations on device
  - Settings operations on device
  - …

CUDA Streams Management

- Stream management:
  - Constructor: `cudaStreamCreate()`
  - Synchronization: `cudaStreamSynchronize()`
  - Destructor: `cudaStreamDestroy()`
- Stream allows various 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)

// concurrent execution of the same kernel
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1)
Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2)

// concurrent execution of different kernels
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

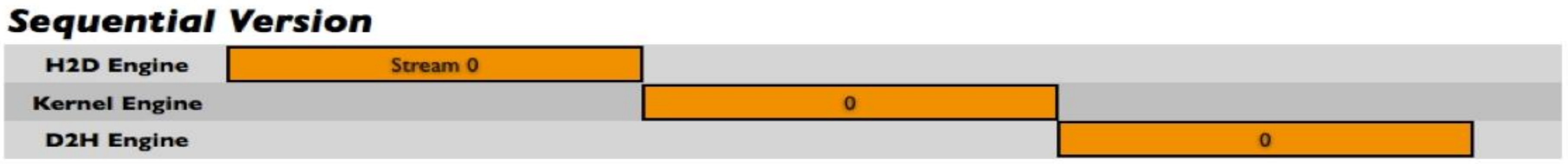
- In order to performe asynchronous data transfers between host and device the *host* memory must be of page-locked type (a.k.a pinned)
- CUDA runtime provides the following functions to handle page-locked memory:
  - `cudaMallocHost()` allocate page-locked memory on *host*
  - `cudaFreeHost()` free page-locked allocated memory on *host*
  - `cudaHostRegister()` turn *host* allocated memory into page-locked
  - `cudaHostUnregister()` turn page-locked memory into ordinary memory
- `cudaMemcpyAsync()` function explicitly performes asynchronous data transfers between *host* and *device* memory
- Data transfer operations must 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

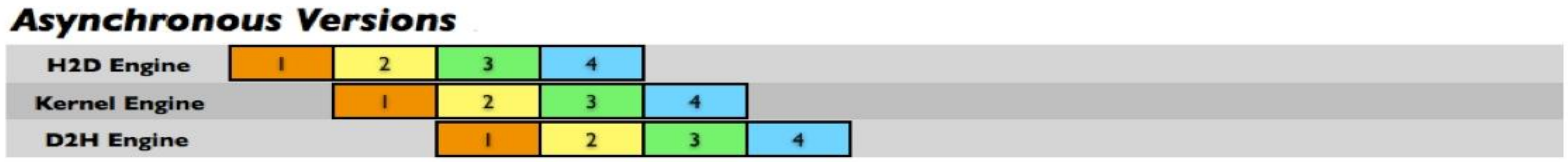
Asynchronous Data Transfers

```
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)

// concurrent and asynchronous dat atransfer H2D and 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]);
```

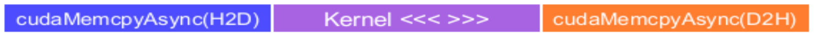




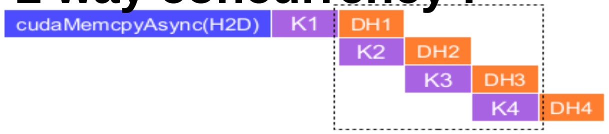
Concurrency

- 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

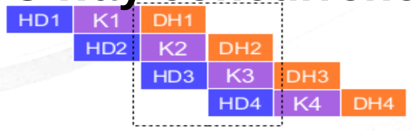
Serial :



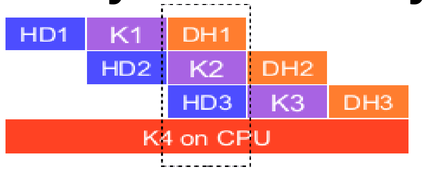
2 way concurrency :



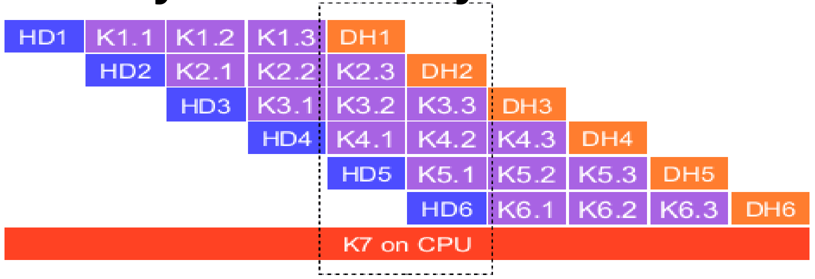
3 way concurrency :



4 way concurrency :



4/+ way concurrency :



Device Management

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

- CUDA 3.2 and previuos 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

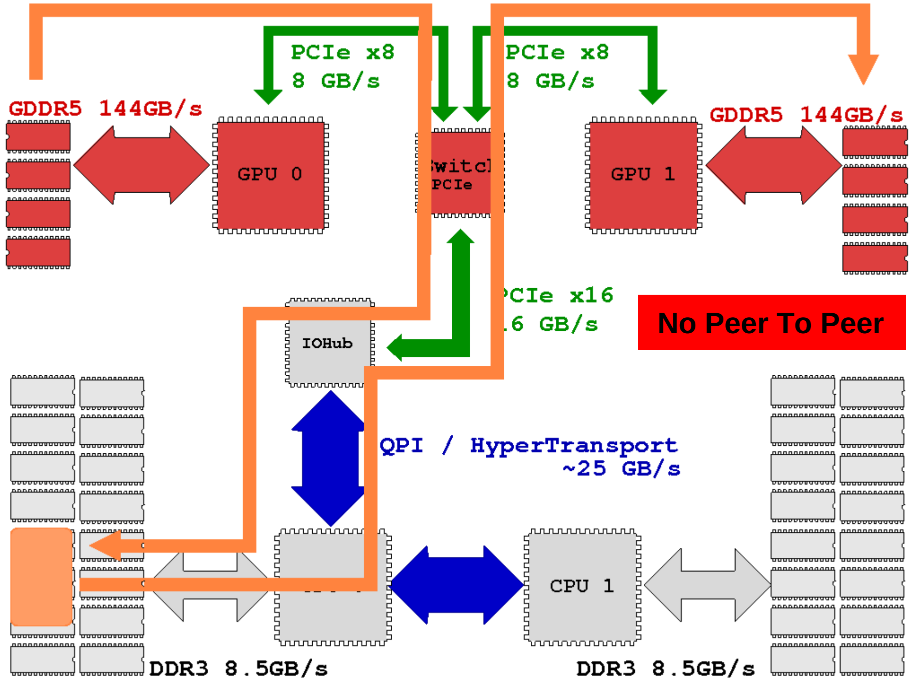
```
cudaDeviceCount(number_gpu)
cudaGetDeviceProperties(gpu_property, gpu_ID)

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

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

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

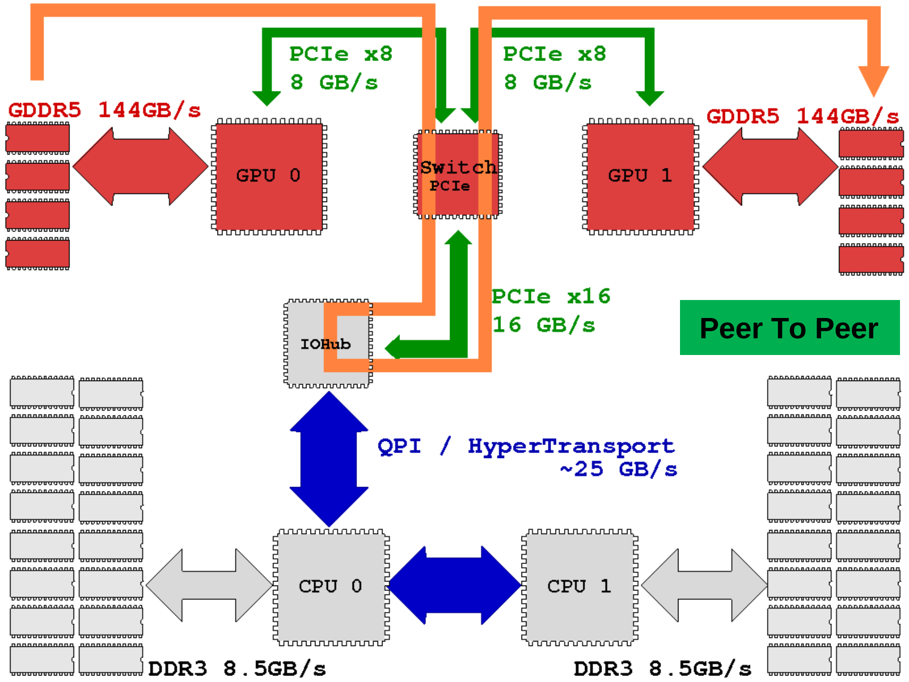
CUDA runtime allows to:

- get information on available CUDA enabled devices
- get properties of each CUDA 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)
 // gpuA performs copy from gpuA to gpuB
 cudaMemcpyPeer(buffer_B, gpuB, buffer_A, gpuA, buffer_size)
 // gpuA performs copy from gpuB to 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)
 // gpuA invokes a kernel that accesses to gpuB memory
 kernel<<<threads, blocks>>>(buffer_A, buffer_B)
```