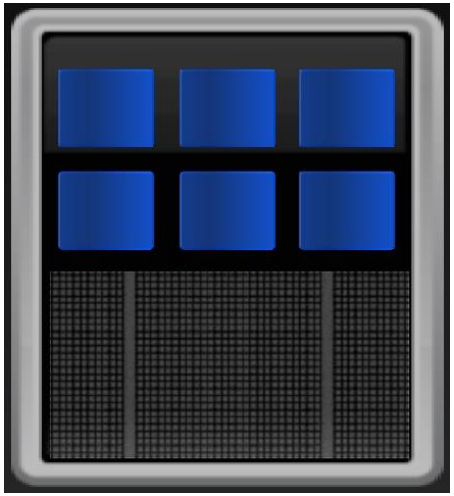
- GPU programming paradigms
- CUDA programming model
  - Heterogeneous execution
  - Writing a CUDA Kernels
  - Thread Hierarchy
- Getting started with CUDA programming:
  - the Vector-Vector Add
  - handling data transfers from CPU to GPU memory and back
  - write and launch the CUDA program

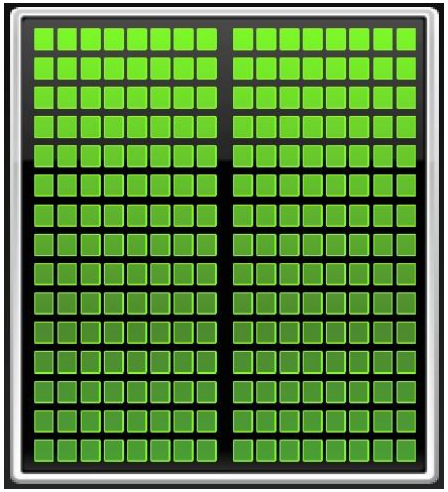
GPGPU Programming Model

- General Purpose GPU Programming relates to use of GPU computational power to solve problems other than graphics
- CPU and GPU are **separate devices** with **separate memory** space addresses
- GPU is seen as an auxiliary coprocessor equipped with thousands of cores and a high bandwidth memory
- They should work together for best benefit and performances

CPU

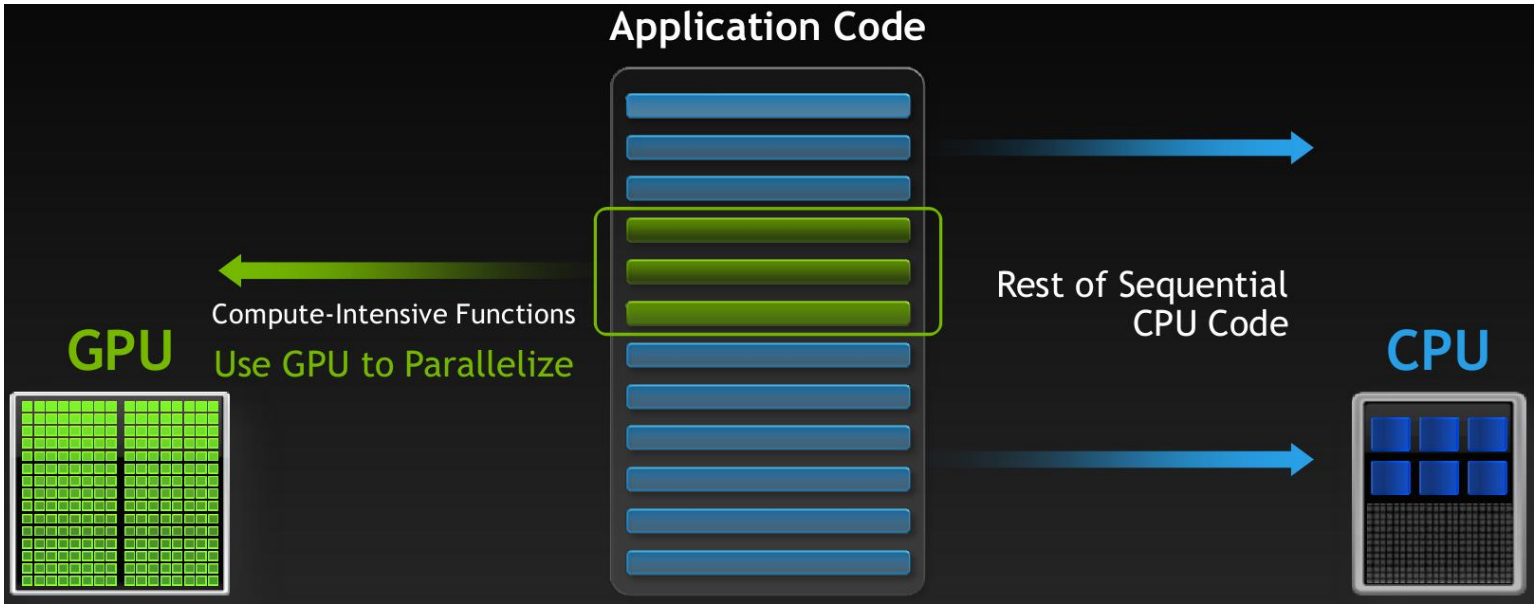


GPU



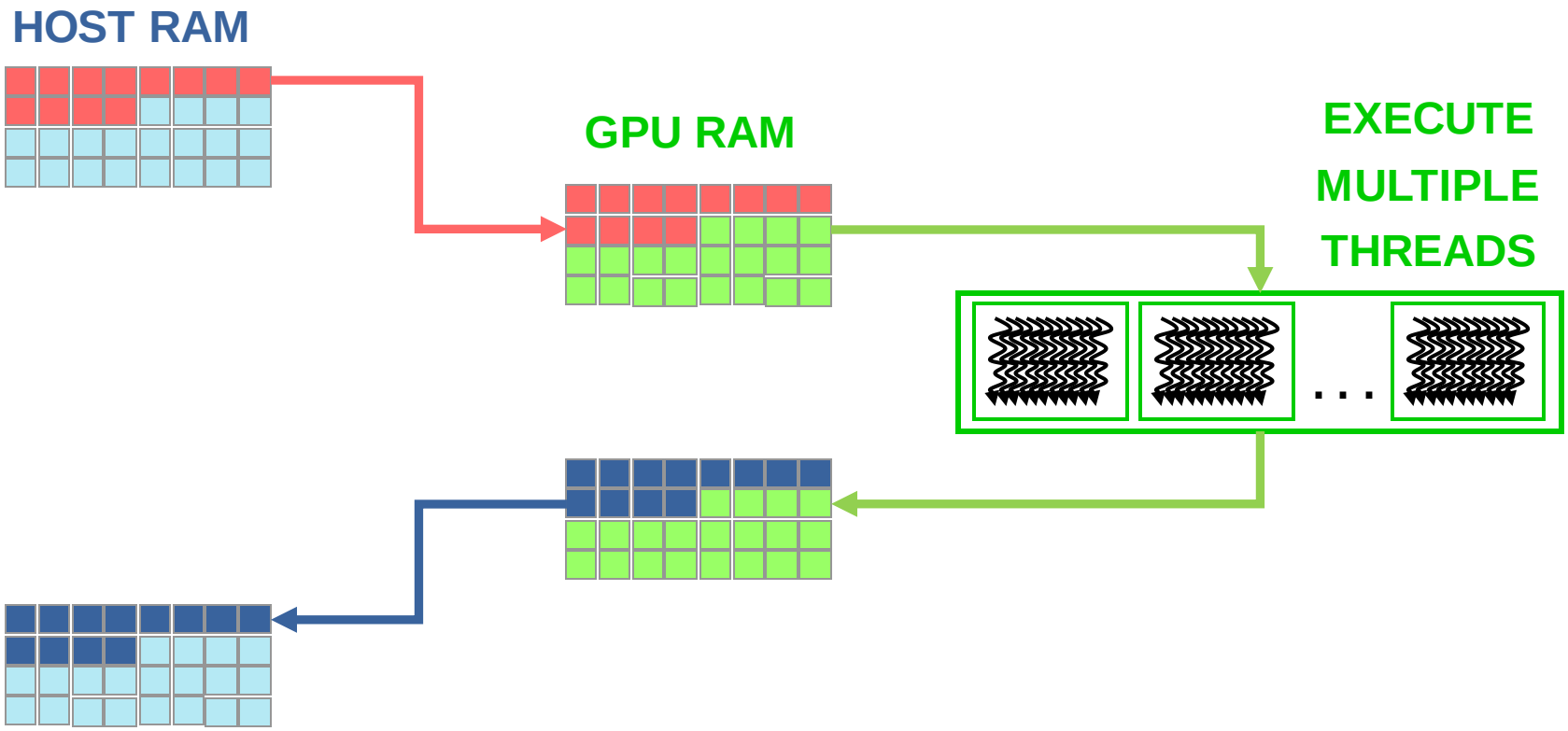
GPGPU Programming Model

- **serial parts** of a program, or those with low level of parallelism, keep running **on the CPU** (host)
- computational-intensive **data-parallel** regions are executed **on the GPU** (device)
- required data is moved on GPU memory and back to HOST memory



Data movement

- data must be moved from HOST to DEVICE memory in order to be processed on the GPU
- when data is processed, and no more needed on the GPU, it is transferred back to HOST



GPGPU Programming Model

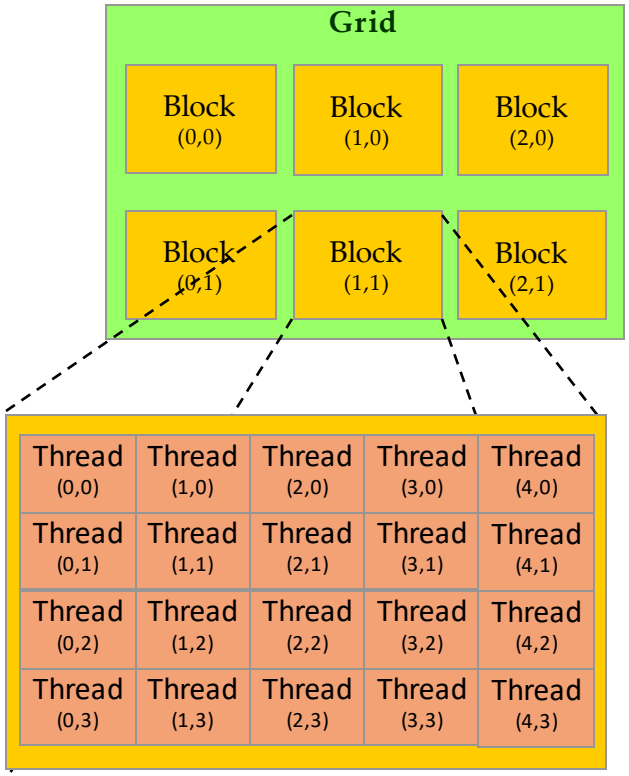
- A function which runs on a GPU is called a "**kernel**"
  - when a kernel is launched on a GPU, thousands of threads will execute its code
  - programmer chooses the number of threads to launch
  - each thread acts on a different data element independently
  - the GPU parallelism is very close to the SPMD paradigm

```
void vecAddCPU (int N, const float *A,
                const float *B, float *C)
{
   for ( int i = 0; i < N; i++ )
      c[i] = a[i] + b[i];
}
...
// call vecAddCPU on N elements
vecAddCPU ( N, a, b, c );
```

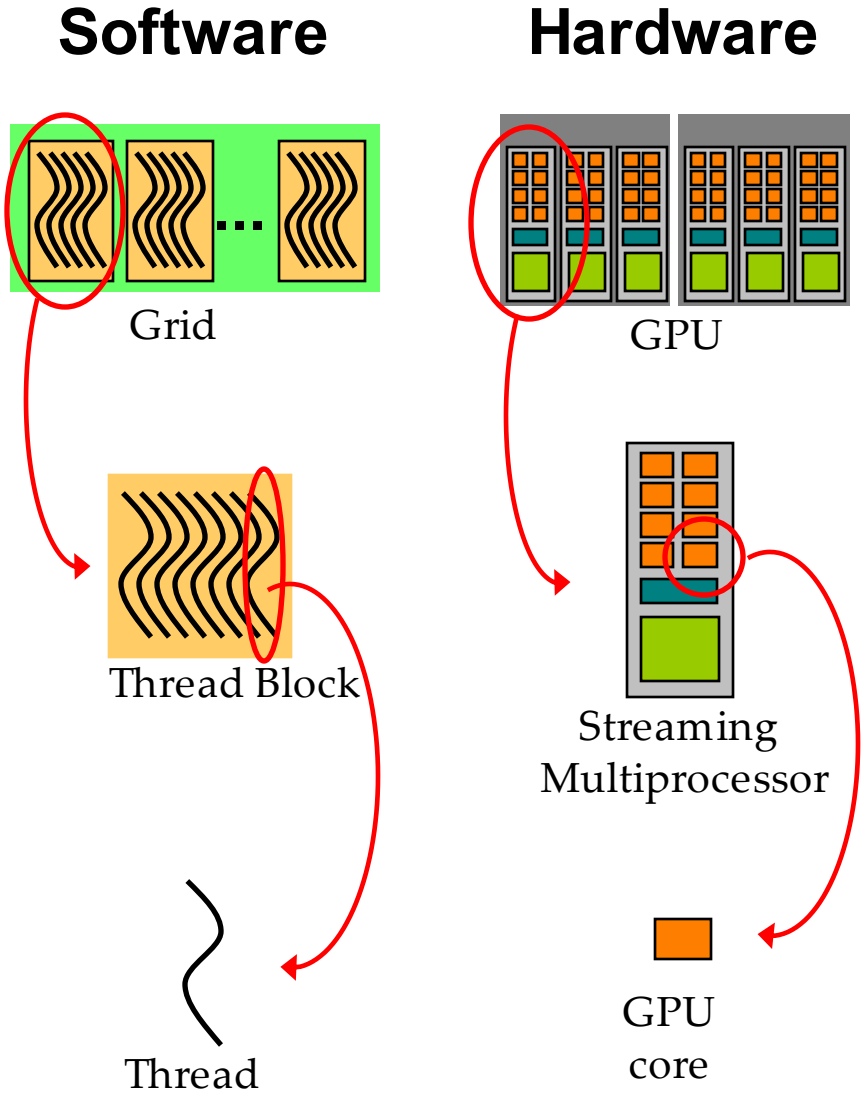
```
void vecAddGPU (int N, const float *A,
                const float *B, float *C)
{
   int i = .... // use unique thread index
    c[i] = a[i] + b[i];
}
...
// launch kernel with a grid of  thread blocks
vecAddGPU<<<N/1024,  1024>>>( N, a, b, c );
```

GPU Thread Hierarchy

- In order to compute N elements on the GPU in parallel, at least N concurrent threads must be created on the device
- GPU threads are grouped togheter in *teams* or *blocks* of threads
- Threads belonging to the same block or team can **cooperate** togheter exchanging data through a shared memory cache area
- each block of threads will be executed **independently**
- no assumption is made on the blocks **execution order**



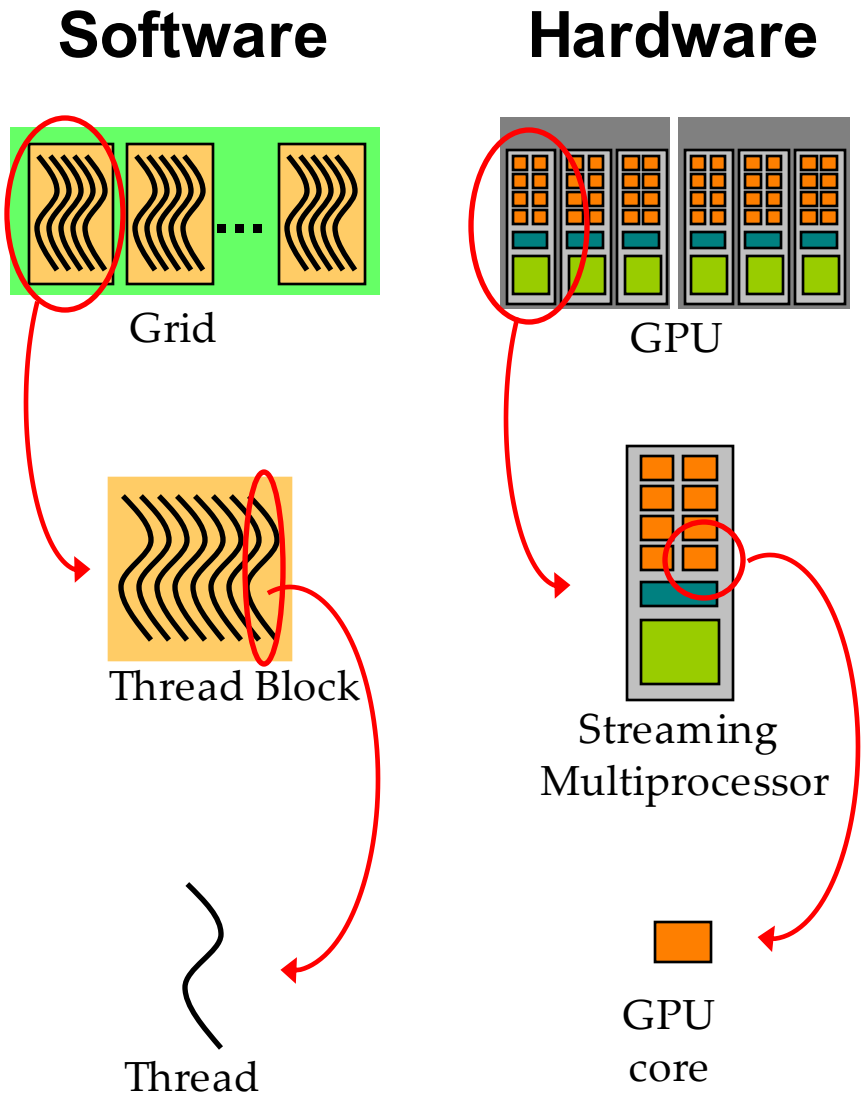
more on the GPU Execution Model



when a GPU kernel is invoked:

- each thread **block** is assigned to a **SM** in a round-robin mode
- amaximum number of blocks can be assigned to each SM:
  - upper bound depending on hardware capabilities
  - upper bound on how many resources a kernel requires (registers, shared memory, etc)
- the runtime system maintains a list of active blocks and assigns new blocks to SMs as they complete
- once a block is assigned to a SM, it remains on that SM until the work for all threads in the block is completed
- each block execution is **independent** from the other
  (no synchronization is possible among them)

more on the GPU Execution Model

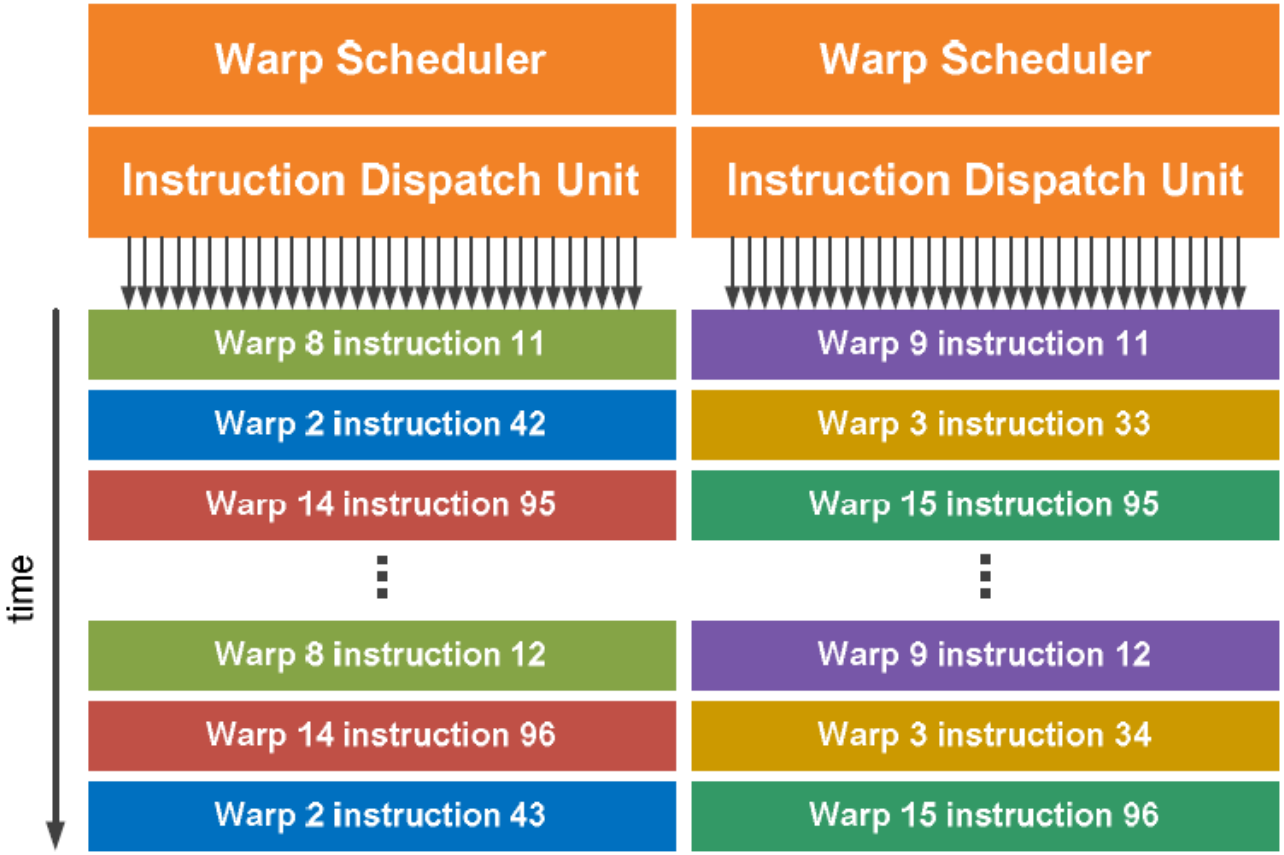


Threads of each block are partitioned into **warps** of consecutive *threads*

- A warp execute one common set of instruction at a time, mapping onto the multiple core ALU of the SM units
  - each GPU core take care of one thread in the warp
  - fully efficiency when all threads agree on their execution path
- The scheduler select for execution an eligible warp from one of the residing blocks in each SM
- A warp is eligible for execution when:
  - Next instruction was fetched and all its arguments are ready
  - Resources are ready: fp32 cores/ fp64 cores/ ldstunit /special function units

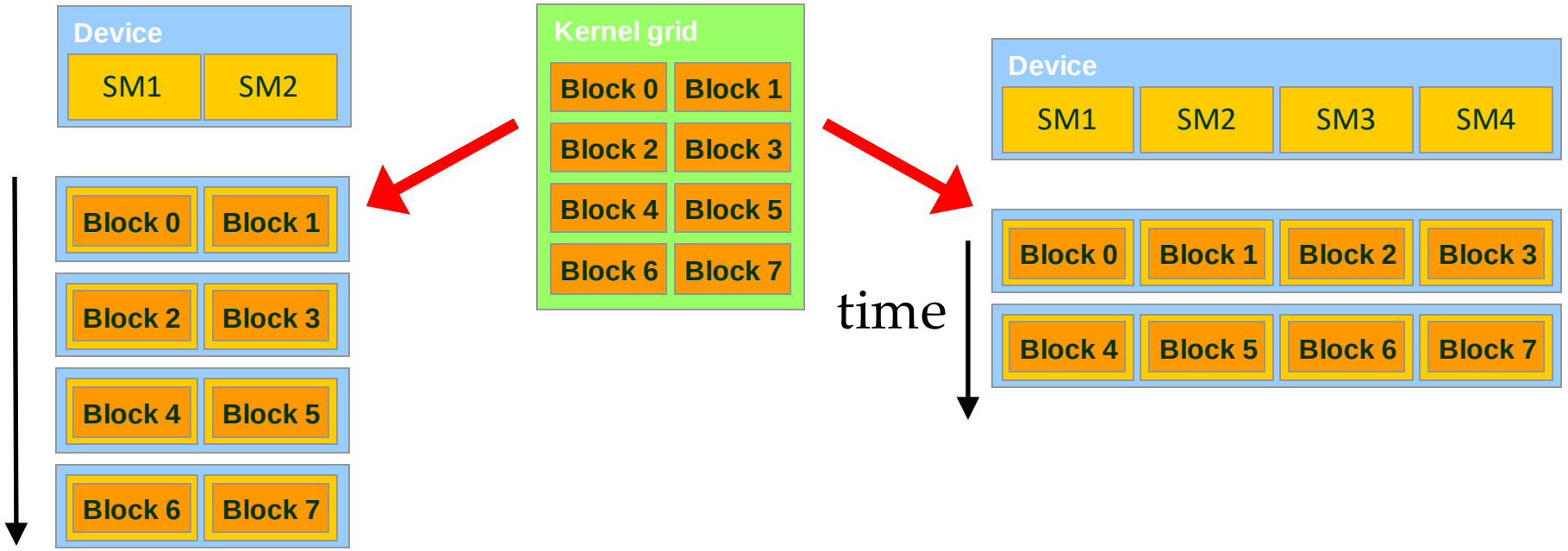
The SM warp scheduler

- The NVIDIA SM schedules threads in groups of 32 threads, called *warps*
- Using 2 warp schedulers per SM allows two warps to be issued and executed concurrently if hardware resources are available



Trasparent Scalability

- The GPU runtime system can execute thread blocks in any order relative to each other
- This flexibility enables to execute the same application code on hardware with different numbers of SM

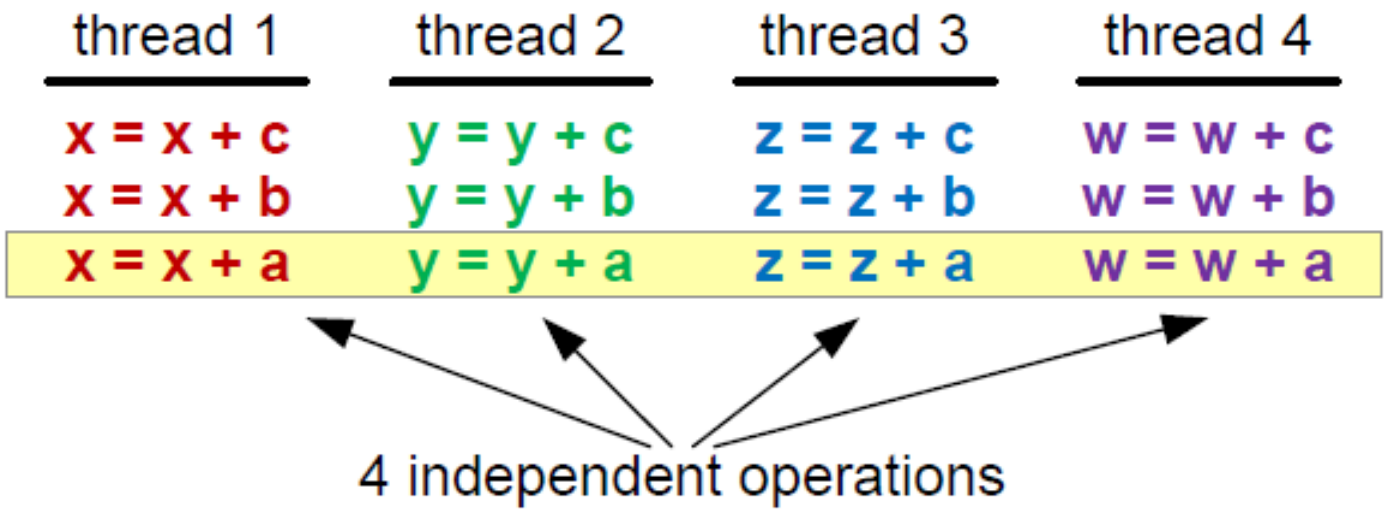


Hiding Latencies

- What is latency?
  - the number of clock cycles needed to complete an istruction
  - ... that is, the number of cycles I need to wait for before another **dependent operation** can start
    - arithmetic latency (~ 18-24 cycles)
    - memory access latency (~ 400-800 cycles)
- We cannot discard latencies (it's an hardware design effect), but we can lesser their effect and hide them.
  - saturating computational pipelines in computational bound problems
  - saturating bandwidth in memory bound problems
- We can organize our code so to provide the scheduler a sufficient number of **independent operations,** so that the more the warp are available, the more content-switch can hide latencies and proceed with other useful operations
- There are two possible ways and paradigms to use (can be combined too!)
  - Thread-Level Parallelism (TLP)
  - Instruction-Level Parallelism (ILP)

Thread-Level Parallelism (TLP)

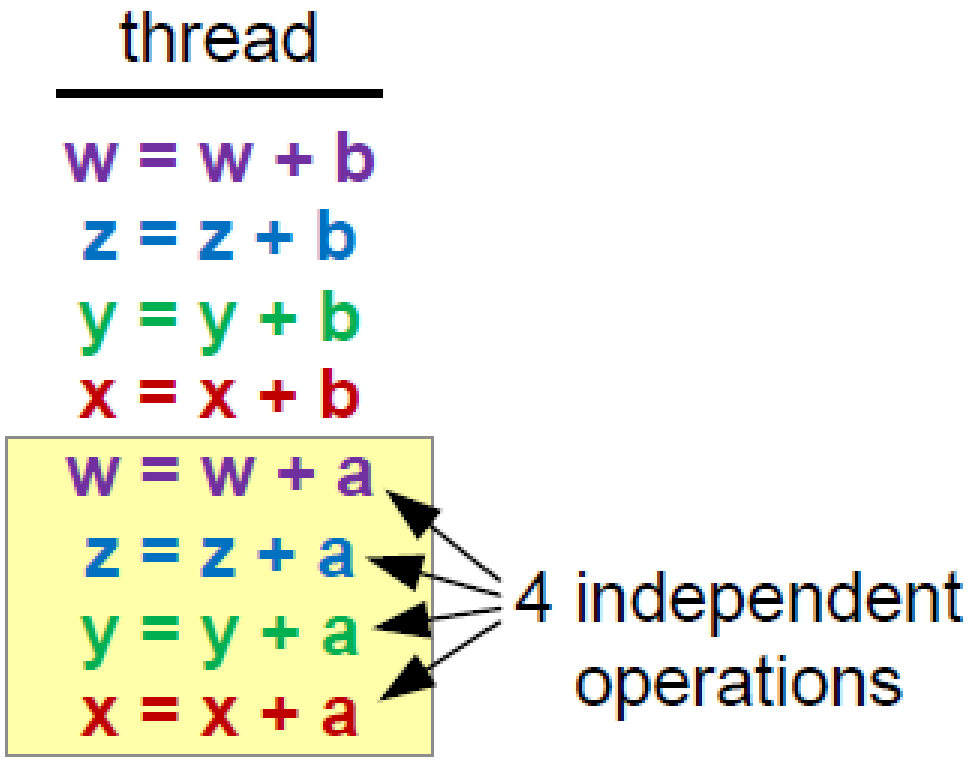
- Strive for high SM **occupancy**: that is try to provide as much threads per SM as possible, so to easy the scheduler find a warp ready to execute, while the others are still busy
- This kind of approach is effective when there is a low level of independet operations per CUDA kernels



Instruction-Level Parallelism (ILP)

- Strive for multiple independent operations inside you CUDA kernel: that is, let your kernel act on more than one data
- this will grant the scheduler to stay on the same warp and fully load each hardware pipeline

- note: the scheduler will not select a new warp untill there are eligible instructions ready to execute on the current warp



Port an Application to GPU

Three steps for a CUDA porting

1. identify data-parallel, computational intensive portions
   1. isolate them into functions (CUDA kernels candidates)
   2. identify involved data to be moved between CPU and GPU

2. translate identified CUDA kernel candidates into real CUDA kernels functions
   1. choose the appropriate thread index map to access data
   2. change code so that each thead acts on its own data

3. modify code in order to manage memory and kernel calls
   1. allocate memory on the device
   2. transfer needed data from host to device memory
   3. insert calls to CUDA kernel with execution configuration syntax
   4. transfer resulting data from device to host memory

Vector Sum

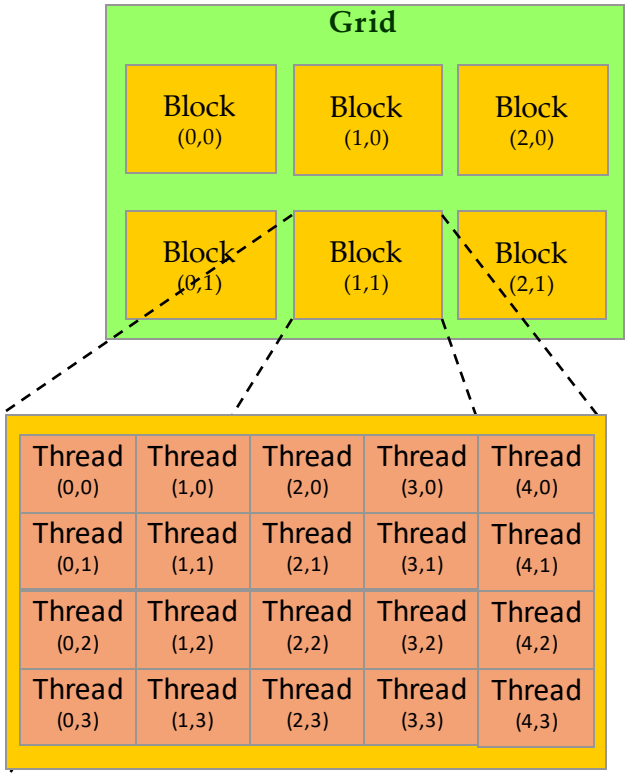
1. identify data-parallel computational intensive part

```
int main(int argc, char *argv[]) {
  int i;
  const int N = 1000000;
  double u[N], v[N], z[N];

  initVector (u, N, 1.0);
  initVector (v, N, 2.0);
  initVector (z, N, 0.0);

  printVector (u, N);
  printVector (v, N);

  // z = u + v
  for (i=0; i<N; i++)
      z[i] = u[i] + v[i];

  printVector (z, N);

  return 0;
}
```

```
program vectoradd
integer :: i
integer, parameter :: N=1000000
real(kind(0.0d0)),dimension(N):: u, v, z

call initVector (u, N, 1.0)
call initVector (v, N, 2.0)
call initVector (z, N, 0.0)

call  printVector (u, N)
call  printVector (v, N)

! z = u + v
do i = 1,N
    z(i) = u(i) + v(i)
end do

call printVector (z, N)

end program
```

GPU Thread Hierarchy

- Threads are organized into blocks of threads
  - blocks can be 1D, 2D, 3D sized in threads
- Blocks are organized in a grid of blocks
  - blocks can be organized into a 1D, 2D, 3D grid of blocks
- each block or thread has a unique ID
  - use .x, .y, .z to access its components



threadIdx:
thread coordinates inside the block

blockIdx:
block coordinates inside the grid

blockDim:
block dimensions in thread units

gridDim:
grid dimensions in block units

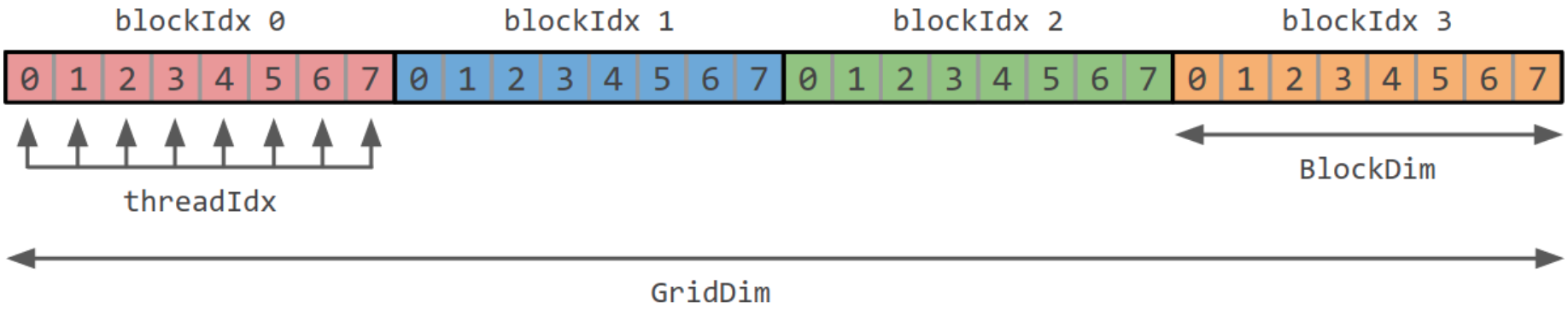
Vector Sum

2. translate the data-parallel portions into CUDA kernels

- each *thread* execute the same kernel, but acts on different data:
  - turn the loop into a CUDA kernel function
  - map each CUDA *thread* onto a unique index to access data
  - let each *thread* retrieve, compute and store its own data using the unique address
  - prevent out of border access to data if data is not a multiple of thread block size

```
// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```

```
cpuVecAdd(int N, double *u, ...) {
  for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
}
```



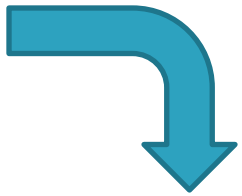
Vector Sum

2. translate the data-parallel portions into CUDA kernels

- each *thread* execute the same kernel, but acts on different data:
  - turn the loop into a CUDA kernel function
  - map each CUDA *thread* onto a unique index to access data
  - let each *thread* retrieve, compute and store its own data using the unique address
  - prevent out of border access to data if data is not a multiple of thread block size

```
// z = u + v
for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
```

```
cpuVecAdd(int N, double *u, ...) {
  for (i=0; i<N; i++)
    z[i] = u[i] + v[i];
}
```

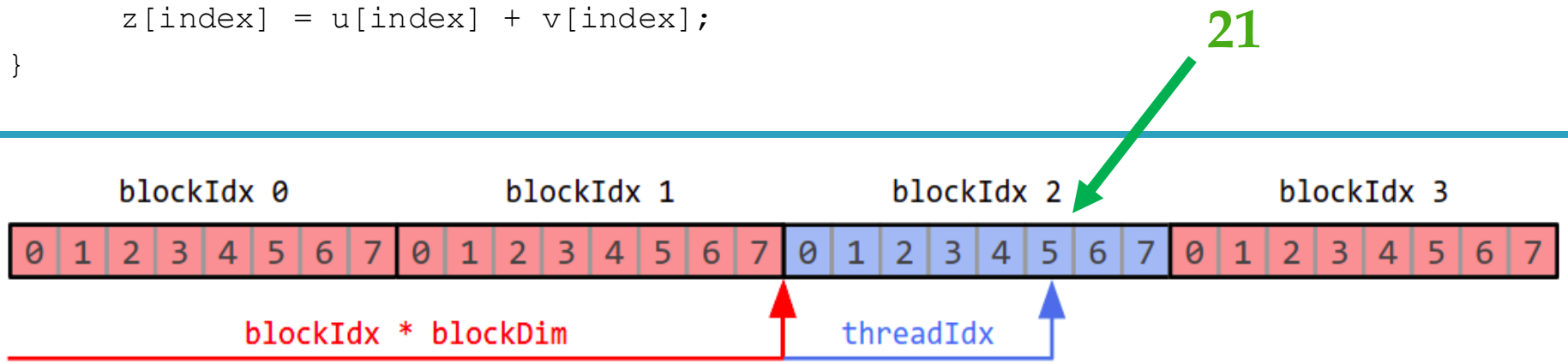


```
__global__  void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index =  blockIdx.x * blockDim.x + threadIdx.x  ;
    if (index < N)
      z[index] = u[index] + v[index];
}
```

Vector Sum

2. translate the data-parallel portions into CUDA kernels

```
__global__  void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index =  blockIdx.x * blockDim.x + threadIdx.x  ;
    if (index < N)
      z[index] = u[index] + v[index];
}
```



In the case of block of 32 threads:

for **blockIdx.x** = 0
    i = 0 * 32 + **threadIdx.x** = { 0, 1, 2, ... , 31 }

for **blockIdx.x** = 1
    i = 1 * 32 + **threadIdx.x** = { 32, 33, 34, ... , 63 }

for **blockIdx.x** = 2
    i = 2 * 32 + **threadIdx.x** = { 64, 65, 66, ... , 95 }

Vector Sum

2. translate the data-parallel portions into CUDA kernels

```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index =  blockIdx.x * blockDim.x + threadIdx.x  ;
    if (index < N)
      z[index] = u[index] + v[index];
}
```

The **__global__** qualifier declares this function to be a CUDA kernel

CUDA kernels are special C functions:

- can be called from host only
- must be called using the *execution configuration* syntax
- the return type must be *void*
- they are **asynchronous**: control is returned immediately to the host code
- an explicit synchronization is needed in order to be sure that a CUDA kernel has completed the execution

Vector Sum

2. Translate the identified parallel portions into CUDA kernels

```
module vector_algebra_cuda
use cudafor

contains
attributes(global) subroutine gpuVectAdd (N, u, v, z)
  implicit none
  integer, intent(in), value :: N
  real, intent(in) :: u(N), v(N)
  real, intent(inout) :: z(N)
  integer :: i

  i = ( blockIdx%x - 1 ) * blockDim%x + threadIdx%x

  if (i .gt. N) return

   z(i) = u(i) + v(i)
end subroutine

end module vector_algebra_cuda
```

Vector Sum

2. translate identified parallel parts into CUDA kernels

```
attributes(global) subroutine gpuVectAdd (N, u, v, z)
  ...
end subroutine

program vectorAdd
use cudafor
implicit none
interface
  attributes(global) subroutine gpuVectAdd (N, u, v, z)
    integer, intent(in), value :: N
    real, intent(in) :: u(N), v(N)
    real, intent(inout) :: z(N)
    integer :: i
  end subroutine
end interface
  ...

end program vectorAdd
```

If kernels are not defined within a module, then an explicit interface must be provided for each kernel.

Vector Sum

Insert calls to CUDA kernels using the execution configuration syntax:

`kernelCUDA<<<numBlocks,numThreads>>>(...)`

specifing the thread/block hierarchy you want to apply:

- **`numBlocks:`** specify grid size in terms of thread blocks along each dimension
- **`numThreads:`** specify the block size in terms of threads along each dimension

```
dim3 numThreads(32);
dim3 numBlocks( ( N + numThreads.x - 1 ) / numThreads.x );
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
```

```
type(dim3) :: numBlocks, numThreads
numThreads = dim3( 32, 1, 1 )
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )
call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )
```

Vector Sum

3. manage memory transfers and kernel calls

- CUDA C API: `cudaMalloc(void **p, size_t size)`
  - allocates size bytes of GPU global memory
  - p is a valid device memory address (i.e. SEGV if you dereference on the host)
  - Use **cudaFree**(void * p) to release memory

```
double *u_dev, *v_dev, *z_dev;

cudaMalloc( (void *) &u_dev, N * sizeof(double));
cudaMalloc( (void *) &v_dev, N * sizeof(double));
cudaMalloc( (void *) &z_dev, N * sizeof(double));
```

- in CUDA Fortran the attribute **device** needs to be used when declaring a GPU array. The array can be allocated by using the Fortran statement **allocate.**
- Use **deallocate** to release memory

```
real(kind(0.0d0)), allocatable, dimension(:) :: u_dev, v_dev, z_dev
attributes(device) :: u_dev, v_dev, z_dev

allocate( u_dev(N), v_dev(N), z_dev(N) )
```

Vector Sum

3. manage memory transfers and kernel calls

- CUDA C API:

```
cudaMemcpy(void *dst, void *src, size_t size, direction)
```

  - copy size bytes from the src to dst buffer

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

- CUDA Fortran:
  - rely on operator overload and use the array syntax to slice elements of the array
  - you can also use the explicit cudaMemcpy function

```
u_dev = u ; v_dev = v
     or equivalent
cudaMemcpy(u_dev, u, size(u))
```

Vector Sum: the complete CUDA code

```
double *u_dev, *v_dev, *z_dev;
cudaMalloc((void *) &u_dev, N * sizeof(double));
cudaMalloc((void *) &v_dev, N * sizeof(double));
cudaMalloc((void *) &z_dev, N * sizeof(double));

cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);

dim3 numThreads(256); // 128-512 are good choices
dim3 numBlocks( (N - 1) / numThreads.x + 1);

gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );

cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(u_dev); cudaFree(v_dev); cudaFree(z_dev);
```

```
real(kind(0.0d0)), device, allocatable, dimension(:) :: u_dev, v_dev, z_dev
type(dim3) :: numBlocks, numThreads

allocate( u_dev(N), v_dev(N), z_dev(N) )
u_dev = u; v_dev = v

numThreads = dim3( 256, 1, 1 ) ! 128-512 are good choices
numBlocks = dim3( (N + numThreads%x – 1) / numThreads%x, 1, 1 )

call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )

z = z_dev
deallocate(u_dev, v_dev, z_dev)
```

- 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:

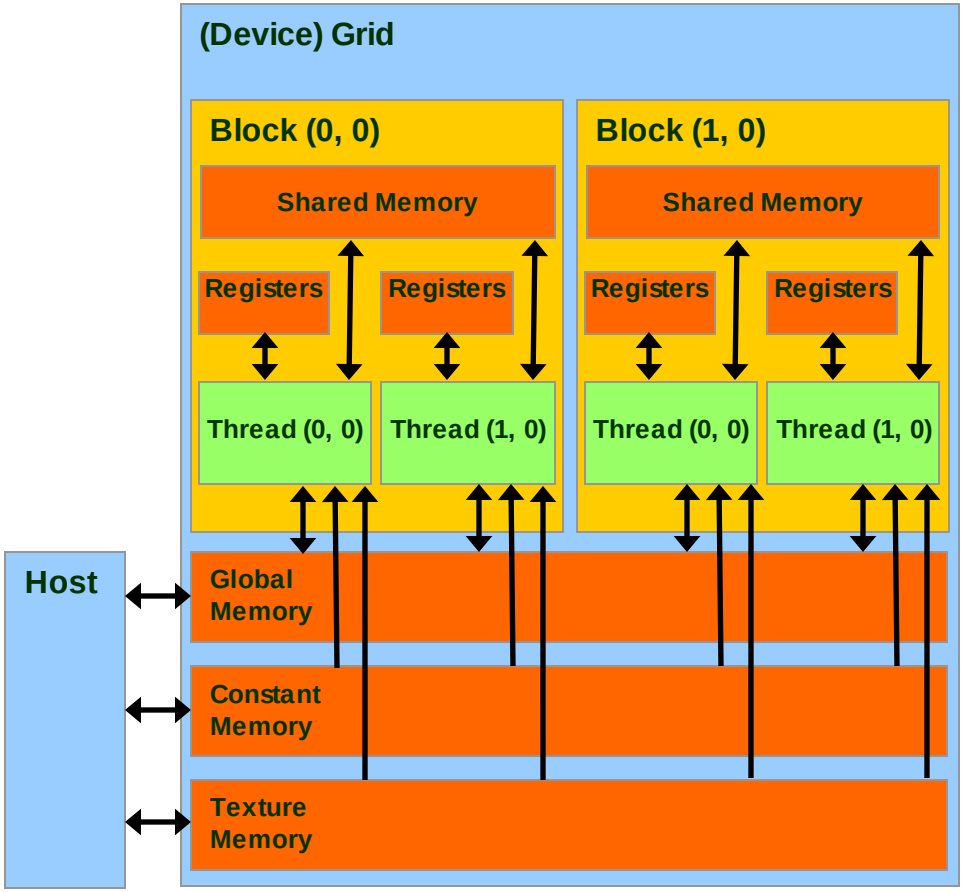
- resources of the SM assigned to its block:
  - **registers**
  - **shared memory**

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

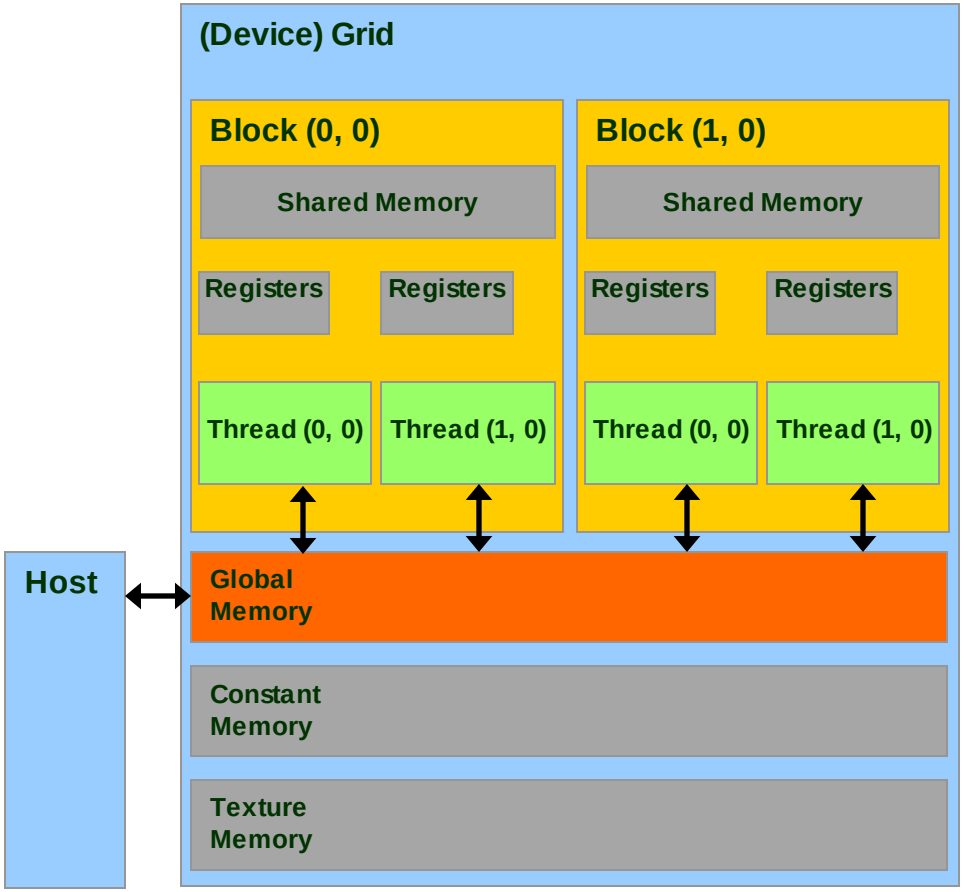


Global Memory

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

    Throughput > 900 GB/s
  - **Very high latency**

    about 400-800 clock cycles

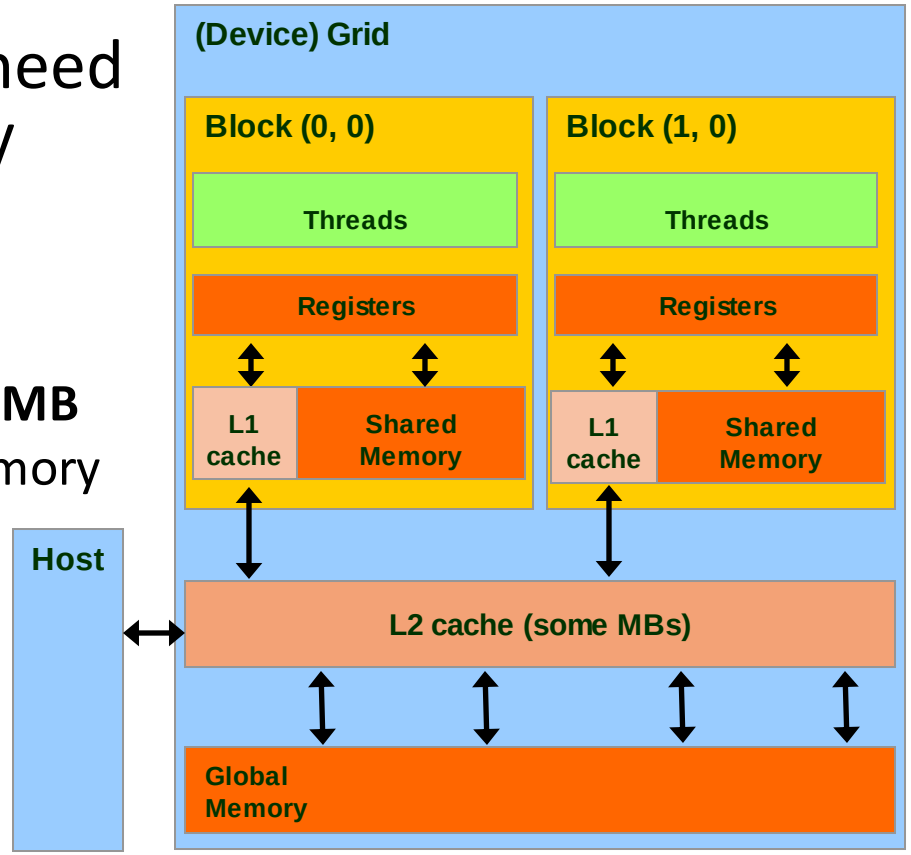


Cache Hierarchy for Global Memory Accesses

- GPU designs include cache hierarchy in order to easy the need for space and time data locality
- 2 Levels of cache:
  - **L2** : shared among all SM
    - **Kepler 1MB, Pascal 4MB, Volta 6MB**
    - 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
```

Warp execution

- Threads within a block are executes in groups of 32 called a *warp*
- if the block is 2D or 3D, the threads are ordered by first dimension, then second, then third – then split into warps of 32
- if the block size is not divisible by 32, some of the threads in the last warp don't do anything
- all threads in a warp performe the same instruction at the same time
- different warps are executed independently by the GPU scheduler

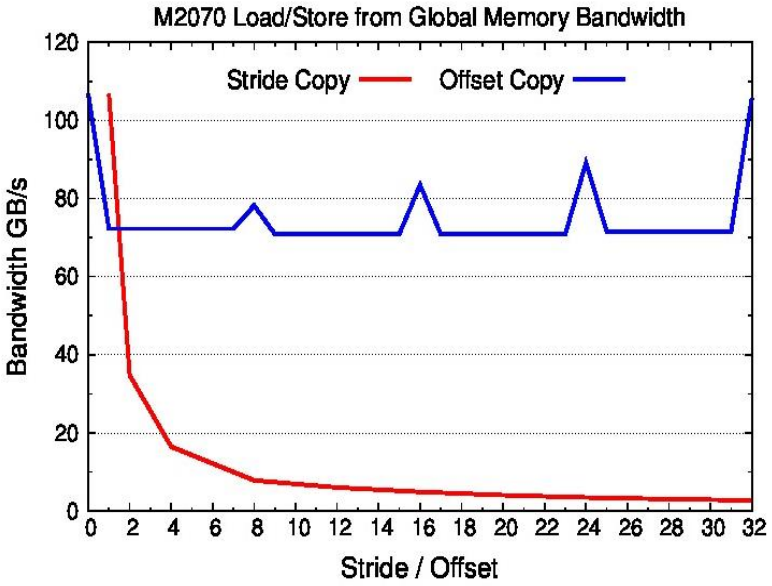
Global Memory Load/Store

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

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

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

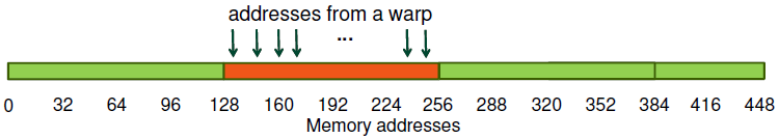


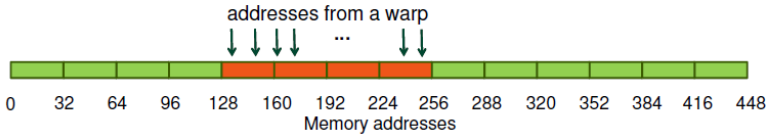
Measured on a M2070; Total elements = 16776960; Used Blocks = 65535; Block lenght = 256

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	**Non-caching Load**
all addresses belong to 1 line cache segment	all addresses belong to 4 line cache segments
128 bytes are moved over the bus	128 bytes are moved over the bus
bus utilization: **100%**	bus utilization: **100%**

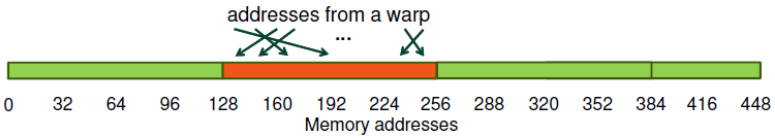


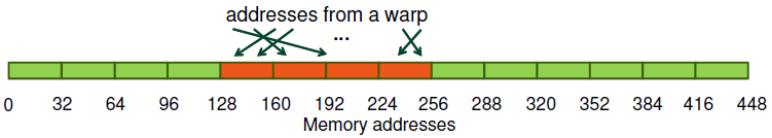


Load Operations from Global Memory

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

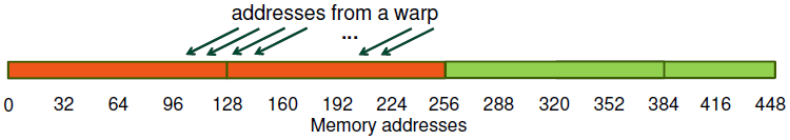
Caching Load	Non-caching Load
addresses belong to 1 line cache segments	addresses belong to 4 line cache segments
128 bytes are moved over the bus	128 bytes are moved over the bus
bus utilization: **100%**	bus utilization: **100%**

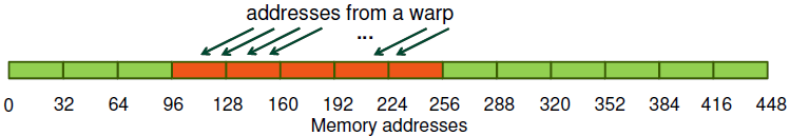




Warp requires 32 consecutive 4-bytes words not alined to segment (total 128 bytes)

Caching Load	Non-caching Load
addresses belong to 2 line cache segments	addresses belong to 5 line cache segments
256 bytes are moved over the bus	160 are moved over the bus
bus utilization: **50%**	bus utilization: **80%**

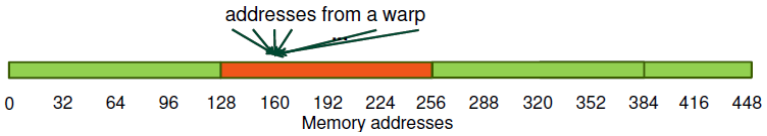


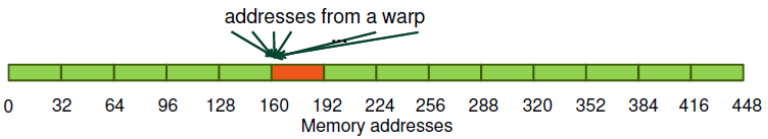


Load Operations from Global Memory

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

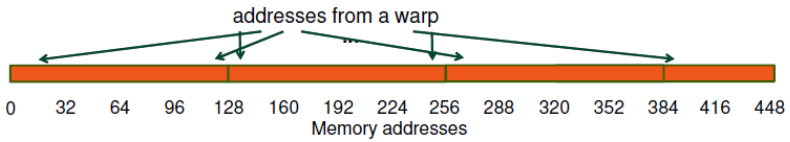
Caching Load	Non-caching Load
address belongs to only one cache line segment	address belongs to only one cache line segment
128 bytes are moved over the bus	32 bytes are moved over the bus
bus utilization: **3.125%**	bus utilization: **12.5%**

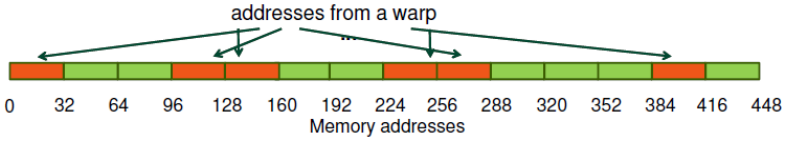




Warp request 32 not contiguous 4-bytes words (total 128 bytes)

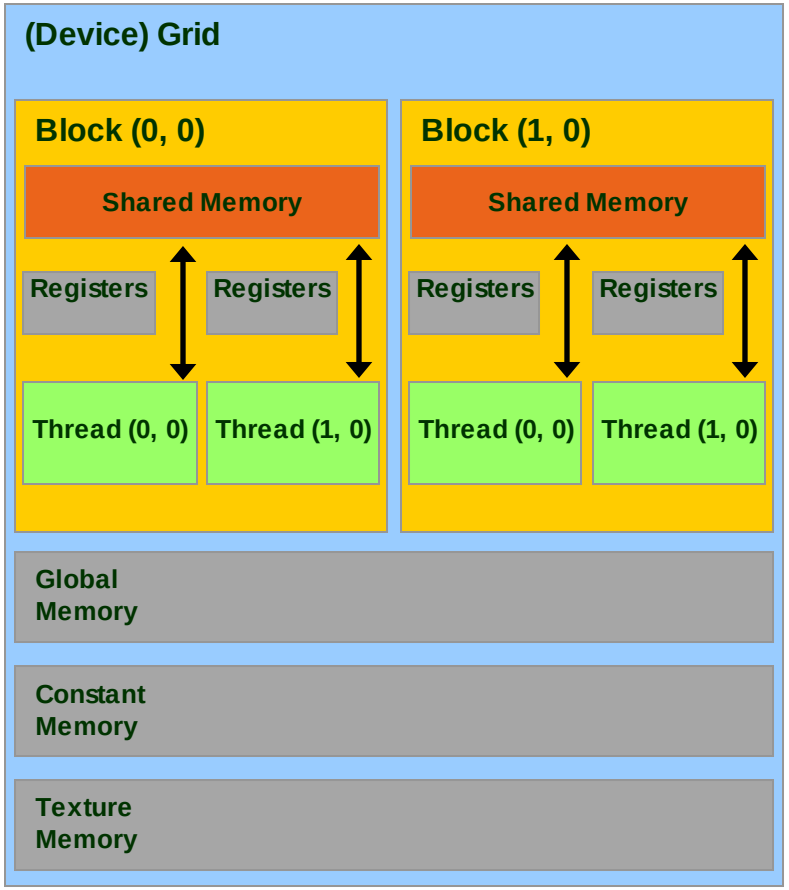
Caching Load	Non-caching Load
addresses belong to N different line cache	addresses belong to N different line cache
N*128 bytes are moved over the bus	N*32 bytes are moved over the bus
bus utilization: **128 / (N*128) = 1/N**	bus utilization: **128 / (N*32) = 4/N**





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 mantained among different kernel calls
- Specifications:
  - **Very low latency**: 2 clock cycles
  - Throughput: 32 bit every 2 cycles
  - Dimension : **48 KB [default]**
    (Configurable : 16/32/48 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()
```

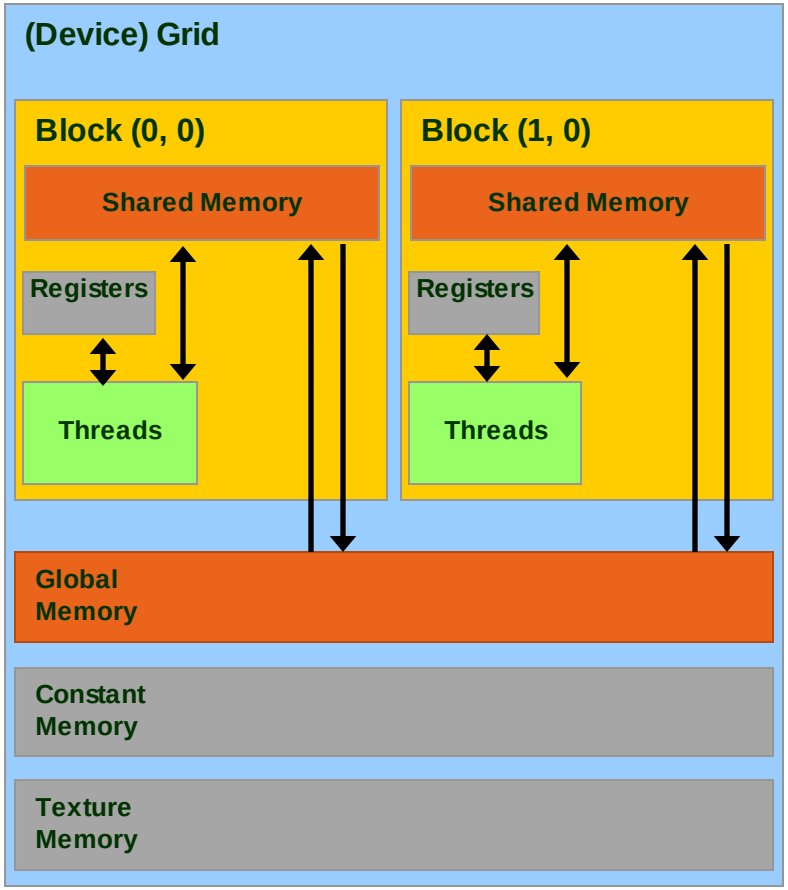
which blocks execution until all other threads reach the same call location

- 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"

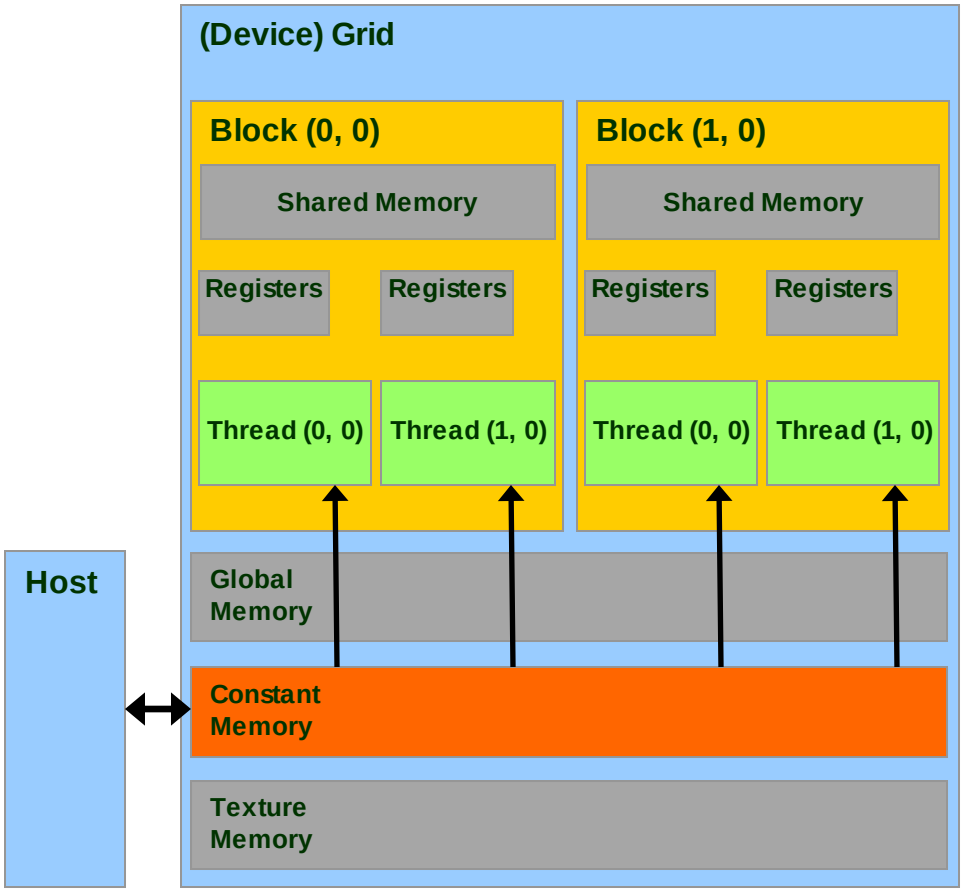
Using Shared Memory for Thread Cooperation

- Threads belonging to the same block can cooperate togheter using the shared memory to share data
  - if a thread is in need of 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 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
  - performe operation on data
  - synchronize threads so all operations have been performed
  - write back results to global memory



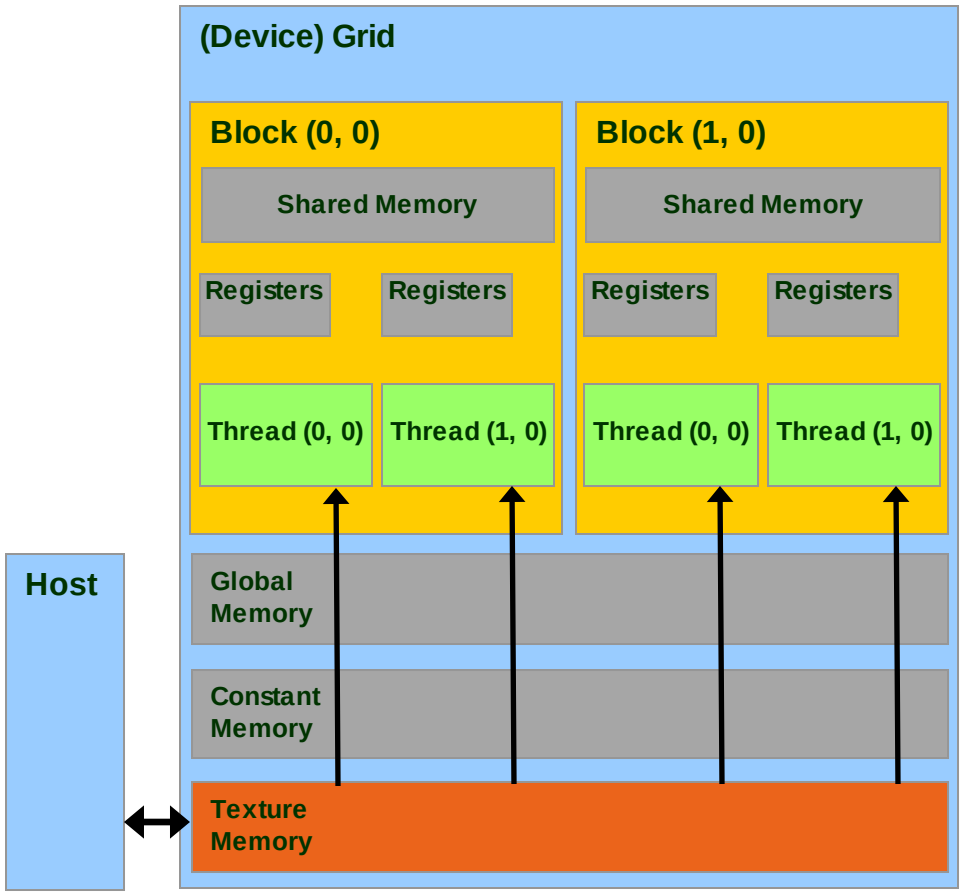
Constant Memory

- ***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



Texture Memory

- **Texture Memory** is afterall 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 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];
}
```

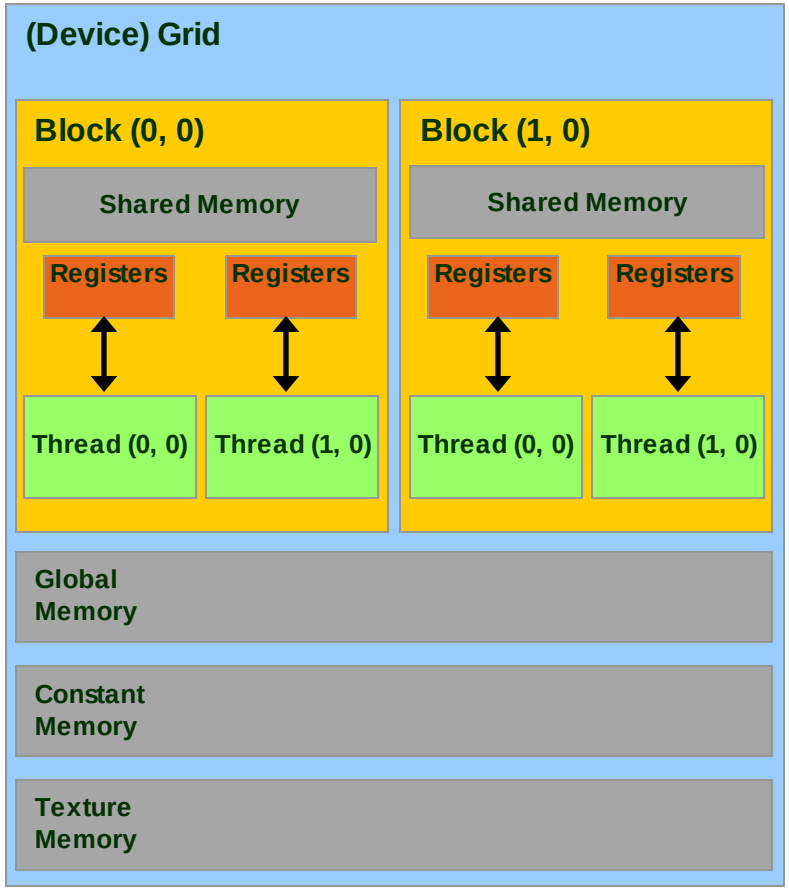
Registers

- **registers** are used to store scalar or small array variables with frequent access by each thread
  - **Kepler, Pascal, Volta** : 255 registers per thread
- 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( … ) { … }
```



Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini