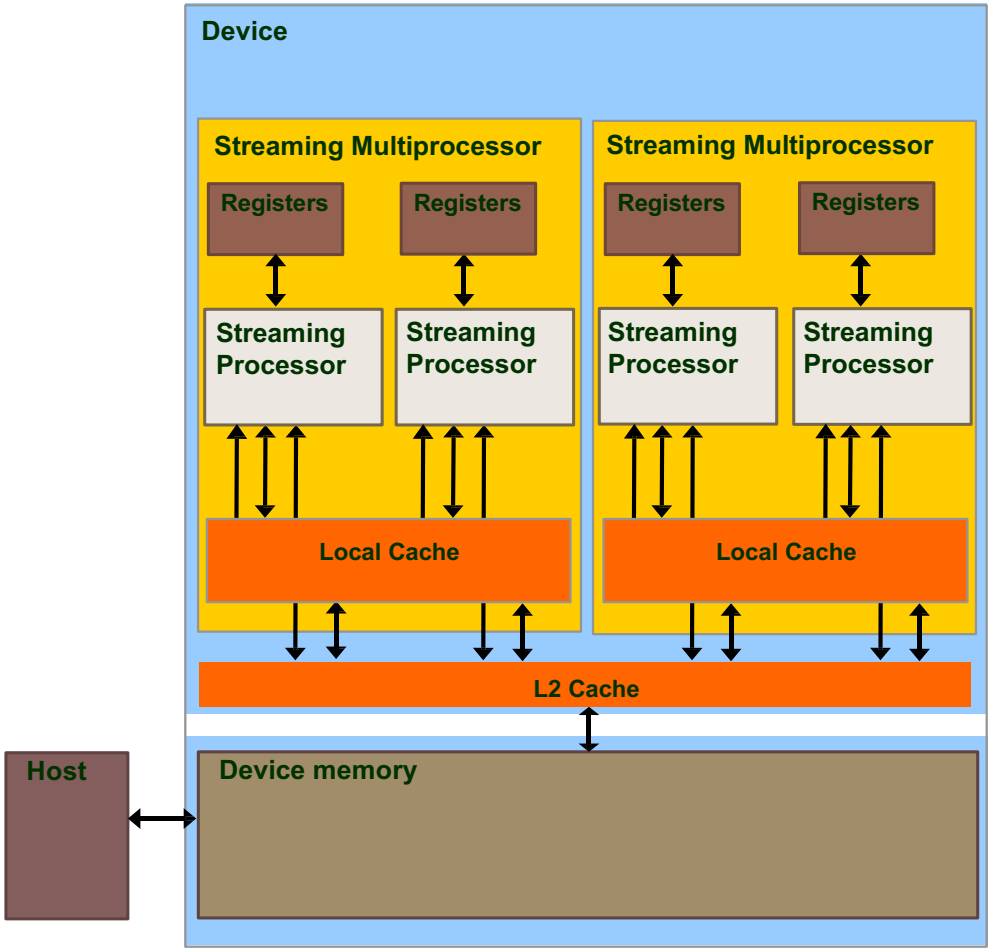
GPUs and Heterogeneous Systems
(programming models and architectures)

CUDA memory model

GPU memory hierarchy

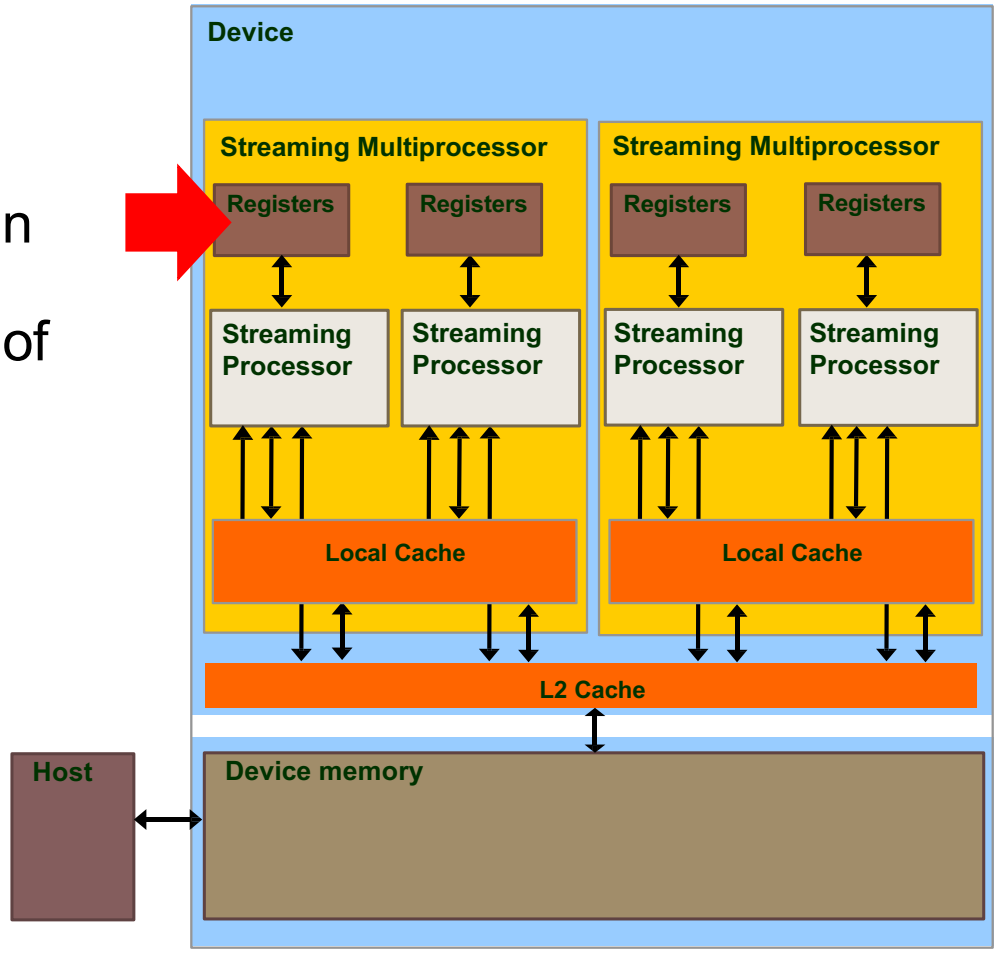
- Overall memory hierarchy:
  - Registers
  - Caches
  - Device memory



GPU memory hierarchy

- Registers
  - Private register file within each SM
  - Partitioned among threads running in the SM
  - Not enough to store all private data of each thread

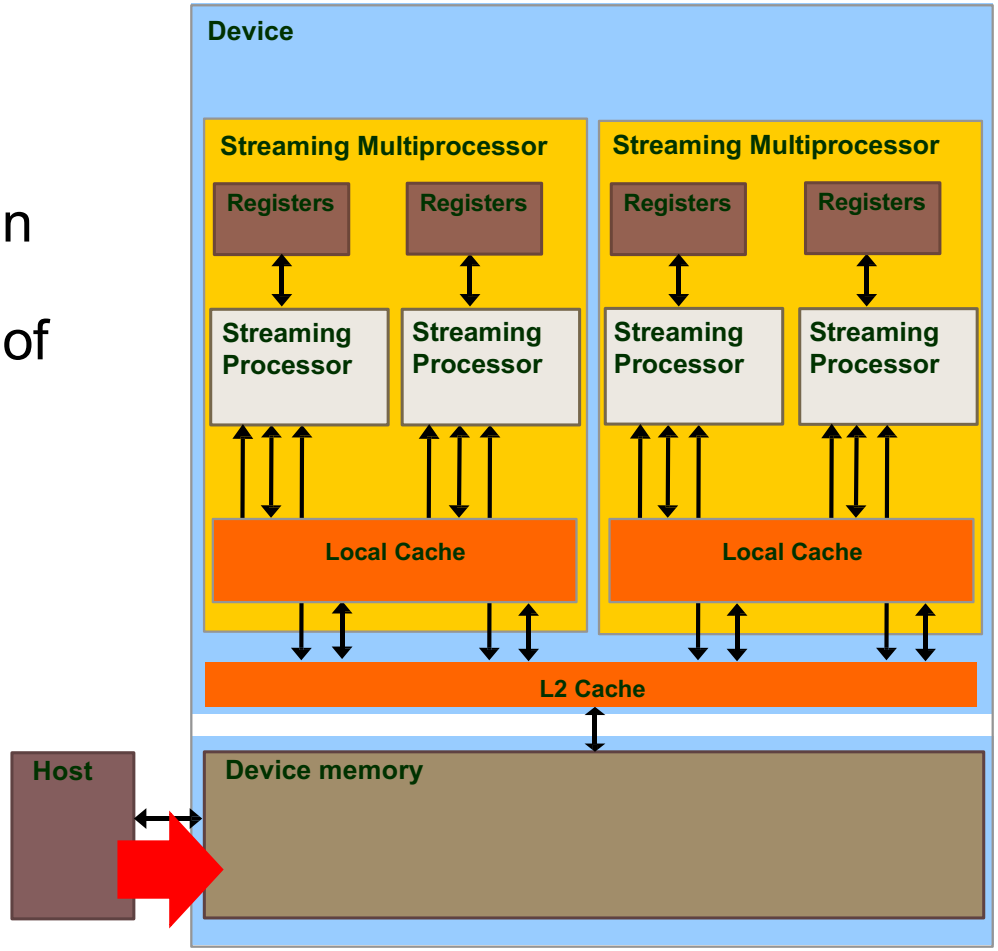
- Device memory (aka global or video memory)
  - Off-chip memory
  - Accessible by
    - All threads on SMs
    - The host



GPU memory hierarchy

- Registers
  - Private register file within each SM
  - Partitioned among threads running in the SM
  - Not enough to store all private data of each thread

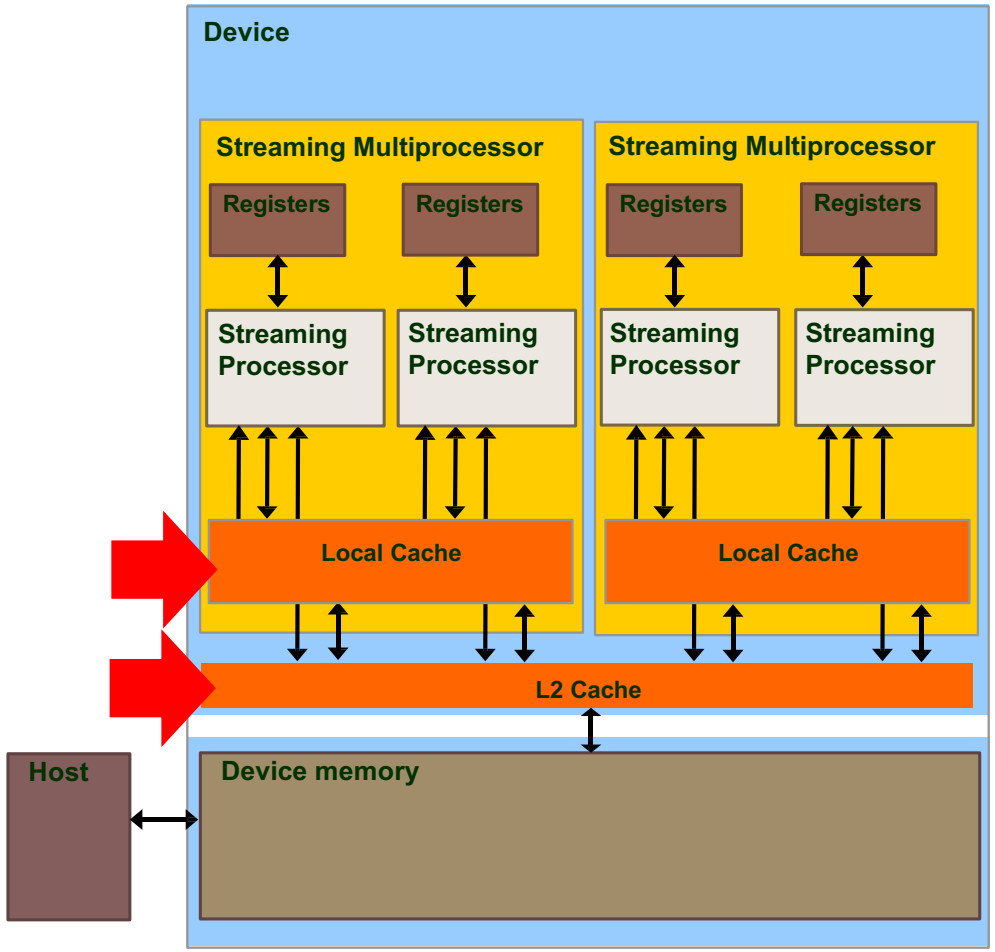
- Device memory (aka global or video memory)
  - Off-chip memory
  - Accessible by
    - All threads on SMs
    - The host



GPU memory hierarchy

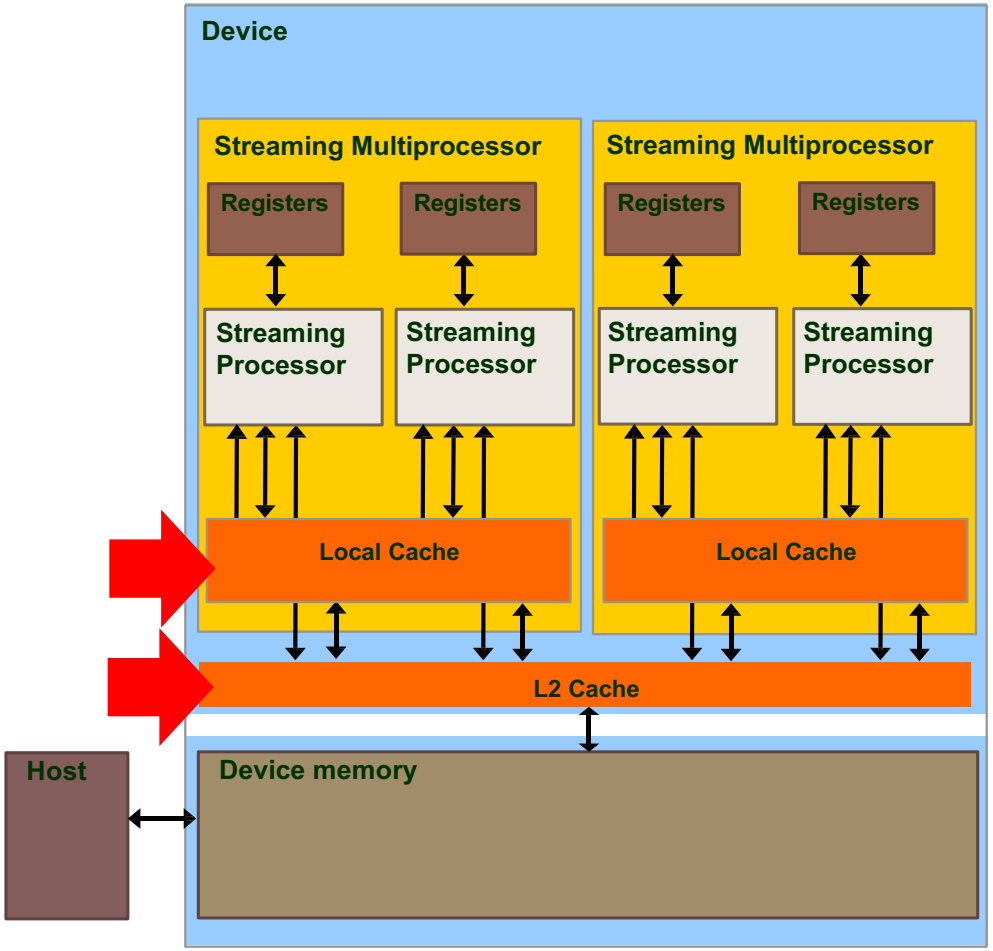
- Caches:
  - Global L2 cache
  - Per-SM L1 cache
  - Per-SM constant cache
  - Per-SM read-only/texture cache

    +
  - Per-SM shared memory



GPU memory hierarchy

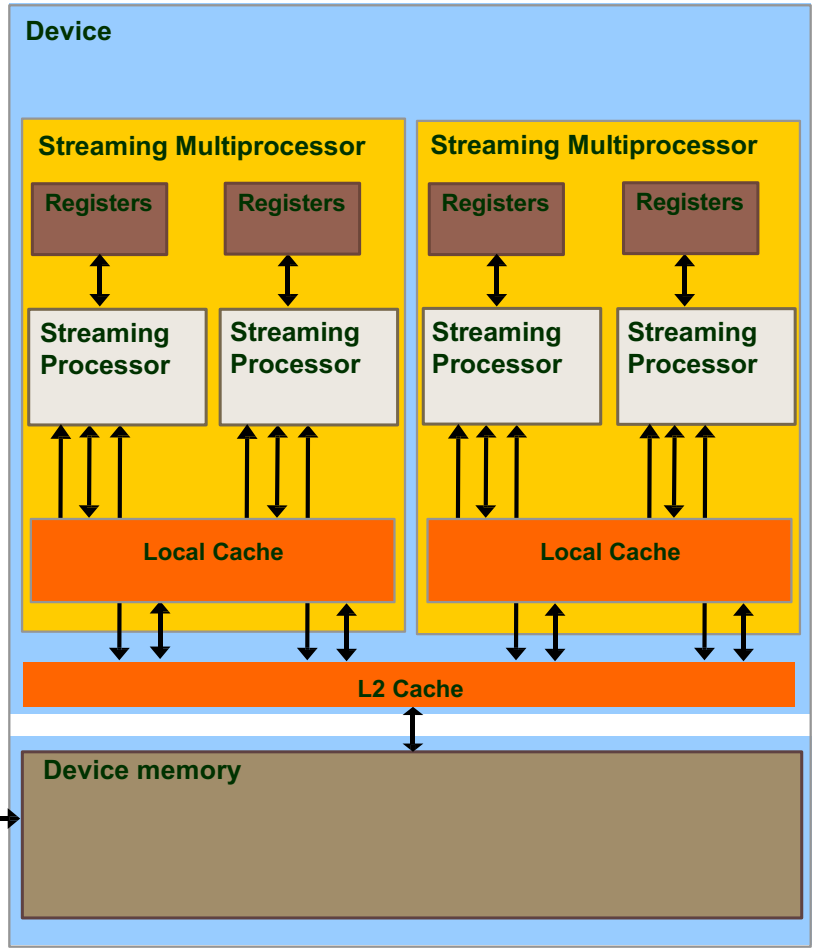
- Cache organization varies in each GPU generation
  - Kepler has unified configurable L1/shared cache
  - Maxwell/Pascal have unified L1/texture cache
  - Volta has unified L1/shared/texture cache
  - …



GPU memory hierarchy

- L1 and L2 caches are non-programmable
- Device memory is programmable
- Shared memory is a programmable scratchpad memory
- Necessity to command the usage of constant and texture caches. How?

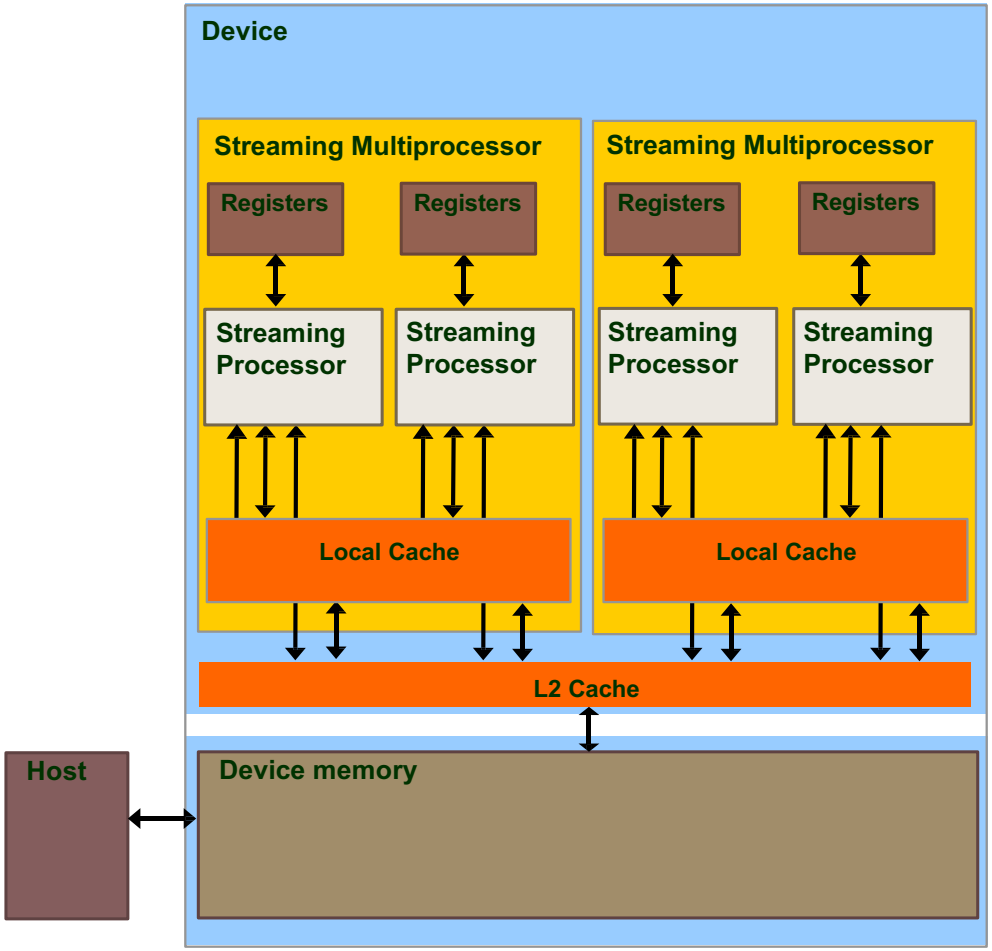
- Programmable: the programmer can explicitly control what data is placed in the memory
- Non-programmable: the programmer has no control over data placement; automatic techniques are used to achieve good performance



GPU memory hierarchy

- Memory latencies:

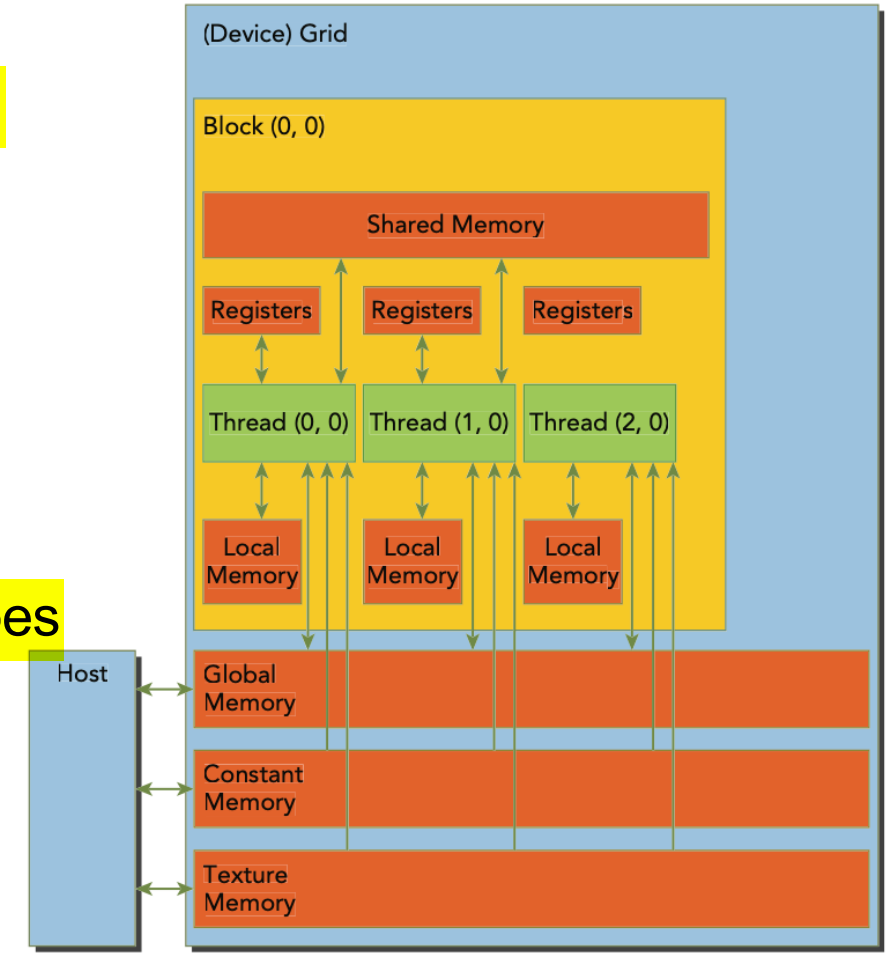
	Cost (cycles)
Registers	1
Device memory	200-800
Shared memory	~1
L1	1
Constant cache	~1
Read-only	1



CUDA memory model overview

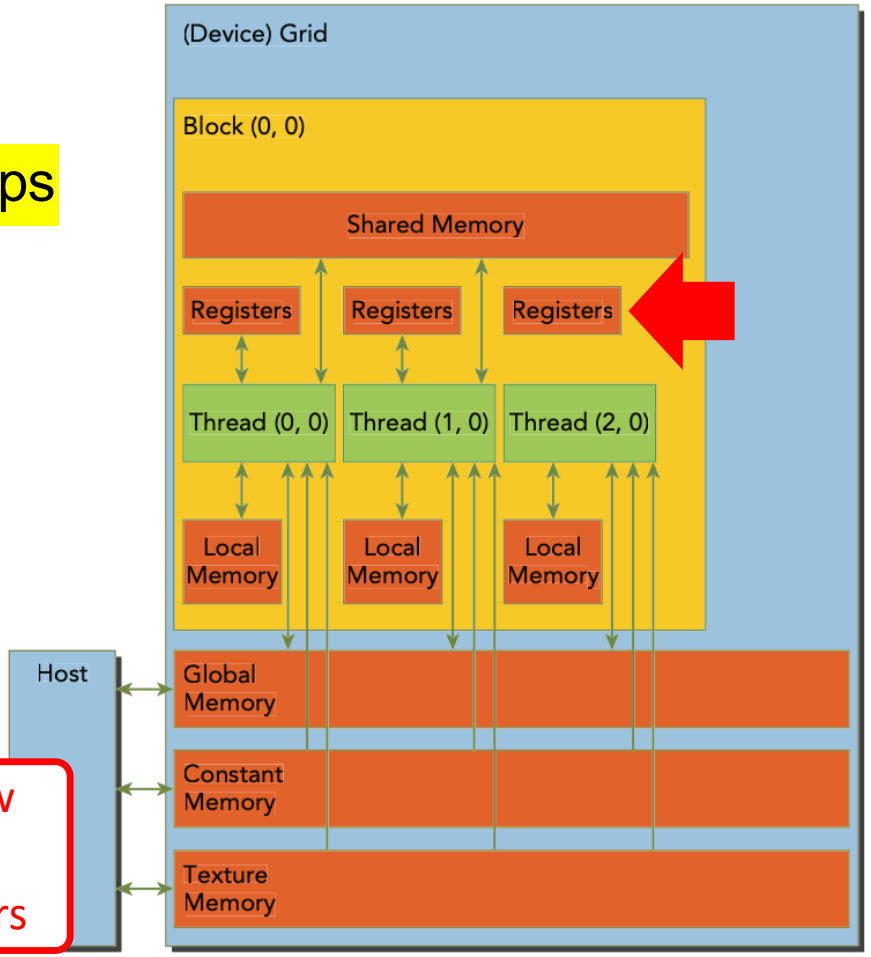
- The CUDA memory model gives a uniform and systematic abstract view on the GPU memories
- It is independent from the specific GPU generation
- It specifies how to program all memory types
  - The programmer states where to allocate each variable

Use `cudaGetDeviceProperties()` to have information on the limits on each memory type



Registers

- Registers are grouped in the register file
- Registers are partitioned among active warps in the SM
  - Registers represent a constraint on the number of blocks assignable to a SM
  - Register spilling to local memory is performed when register request is too high



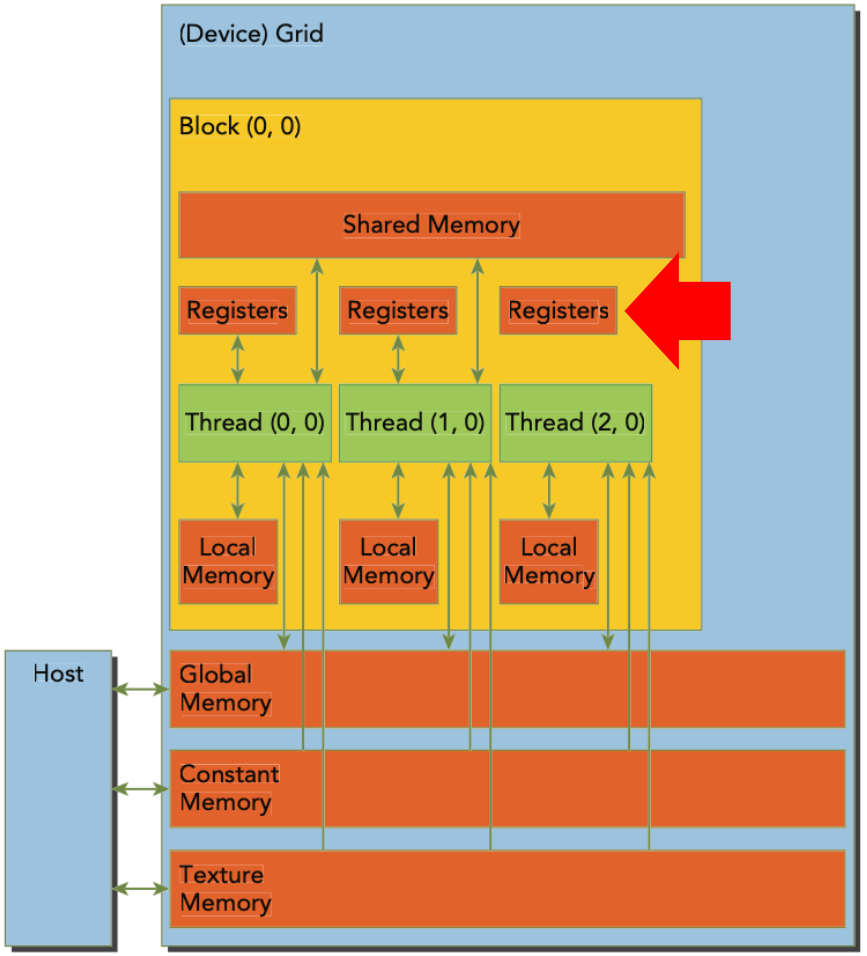
Use nvcc `--ptxas-options=-v` parameter to know how many registers are required by each thread and `-maxrregcount` to force the maximum number of registers

Registers

- Automatic variables are stored in registers

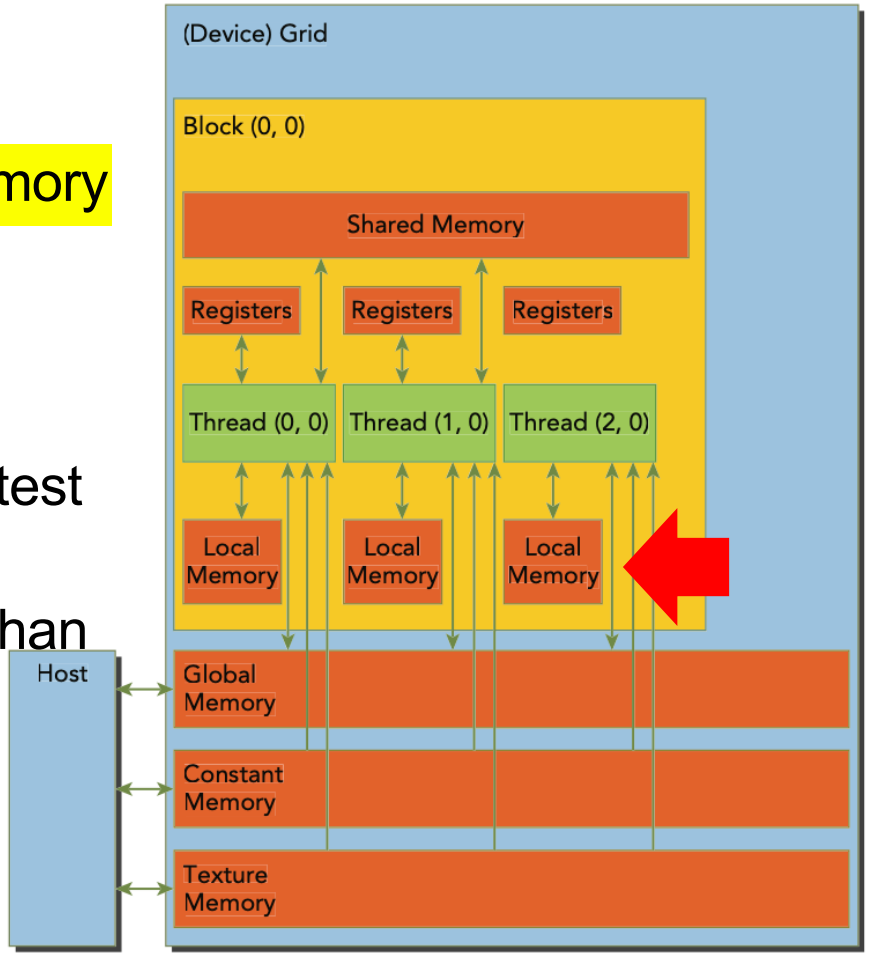
```
__global__ void foo(int* a, int* b) {
  int i;
  /*...*/
}
```

- Automatic variables are declared in the kernel function
  - Access: thread read/write
  - Scope: single thread
  - Lifetime: single thread



Local memory

- Local memory does not physically exist
  - Mapped to reserved area in device memory -> very slow!
  - There are L1/L2 caches
    - Mainly with write-through policy
      - L2 implements write-back in latest GPU families (e.g. Ampere)
    - Exploit more spatial locality rather than temporal locality

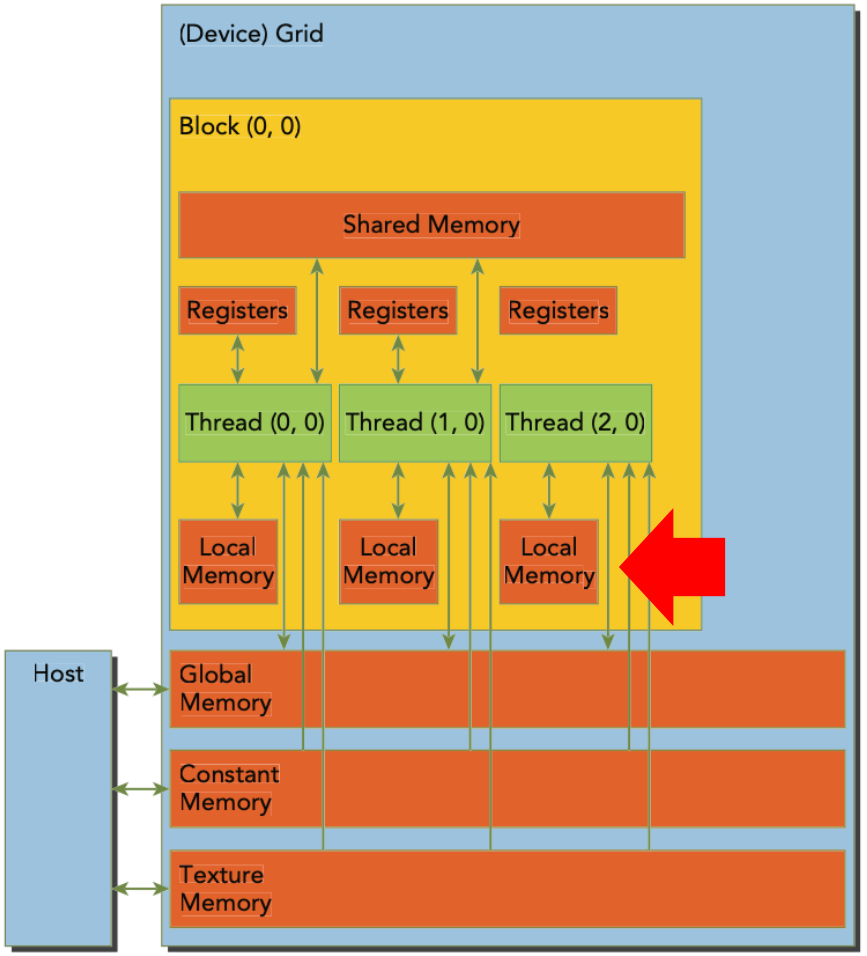


Local memory

- "Large" automatic variables are stored in local memory

```
__global__ void foo(int* a, int* b) {
  int ar[10];
  /*...*/
}
```

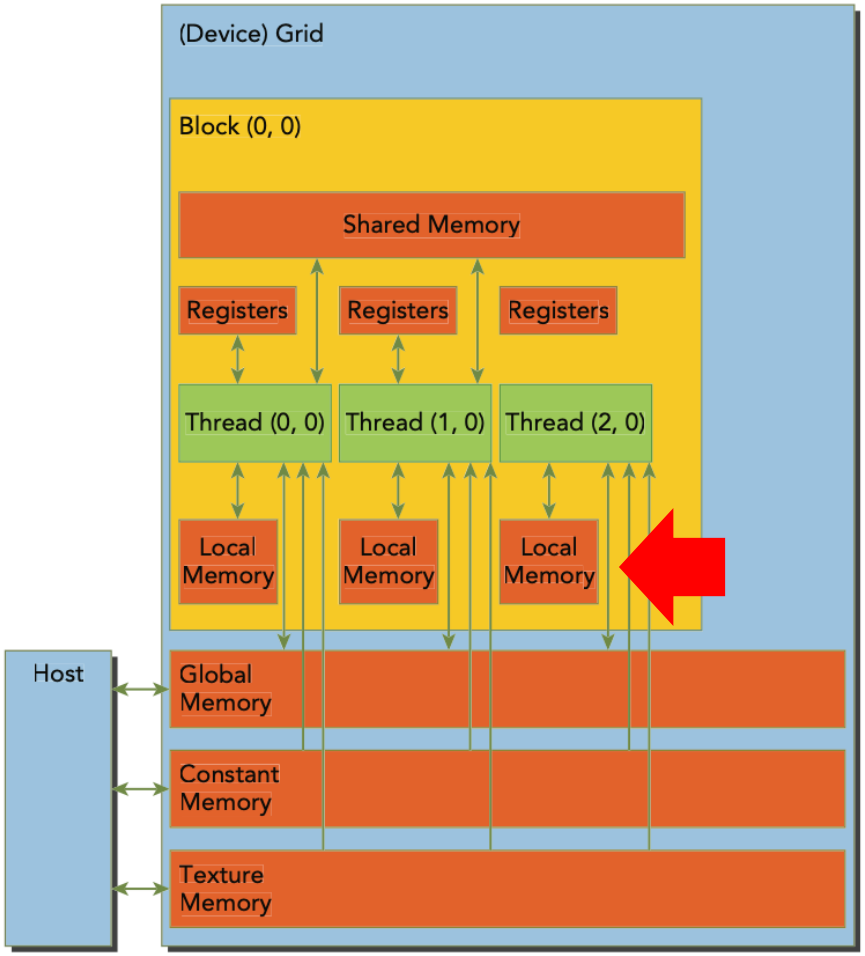
- Automatic variables are declared in the kernel function
  - Access: thread read/write
  - Scope: single thread
  - Lifetime: single thread



Local memory

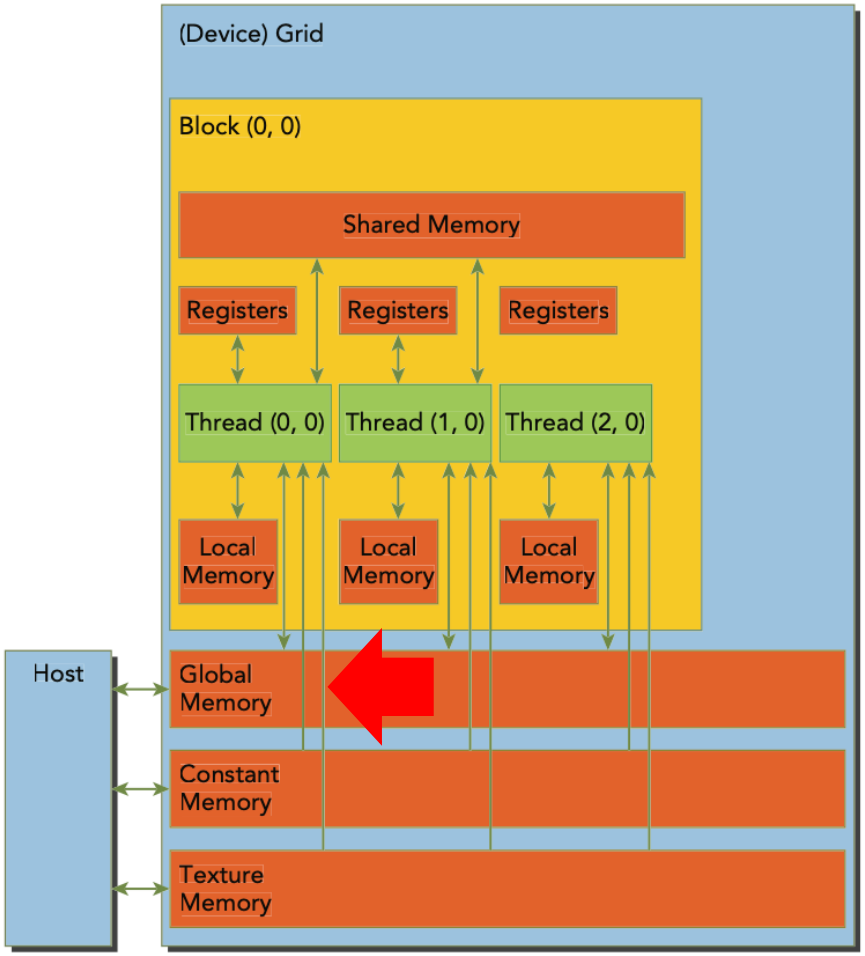
- "Large" automatic variables are:
  - Arrays
  - Large structures

- Used for variables if the reserved number of registers is exceeded
  - Register spilling



Global memory

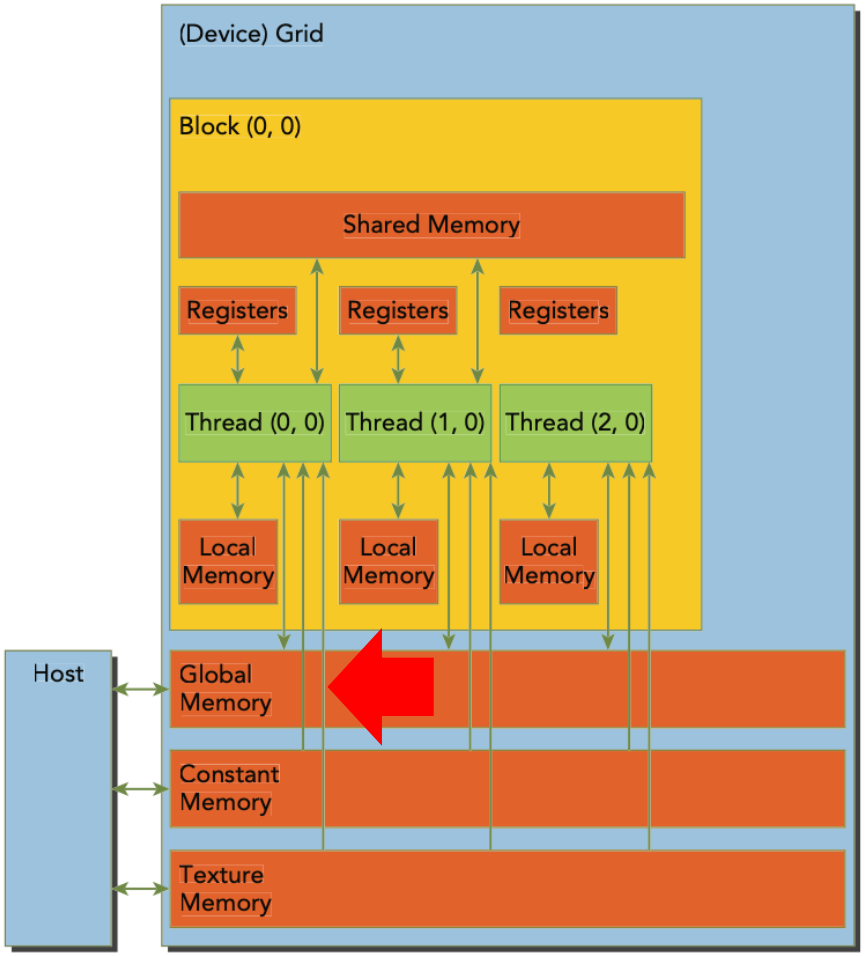
- It corresponds to the device memory
  - Largest memory with highest latency
  - There are L1/L2 caches
    - Mainly with write-through policy
      - L2 implements write-back in latest GPU families (e.g. Ampere)
    - Exploit more spatial locality rather than temporal locality



Global memory

- Device memory accesses are organized in transactions
  - Actual reads and writes are performed at specific data granularities and alignments
  - Specific memory access patters have to be followed to achieve optimal kernel execution time

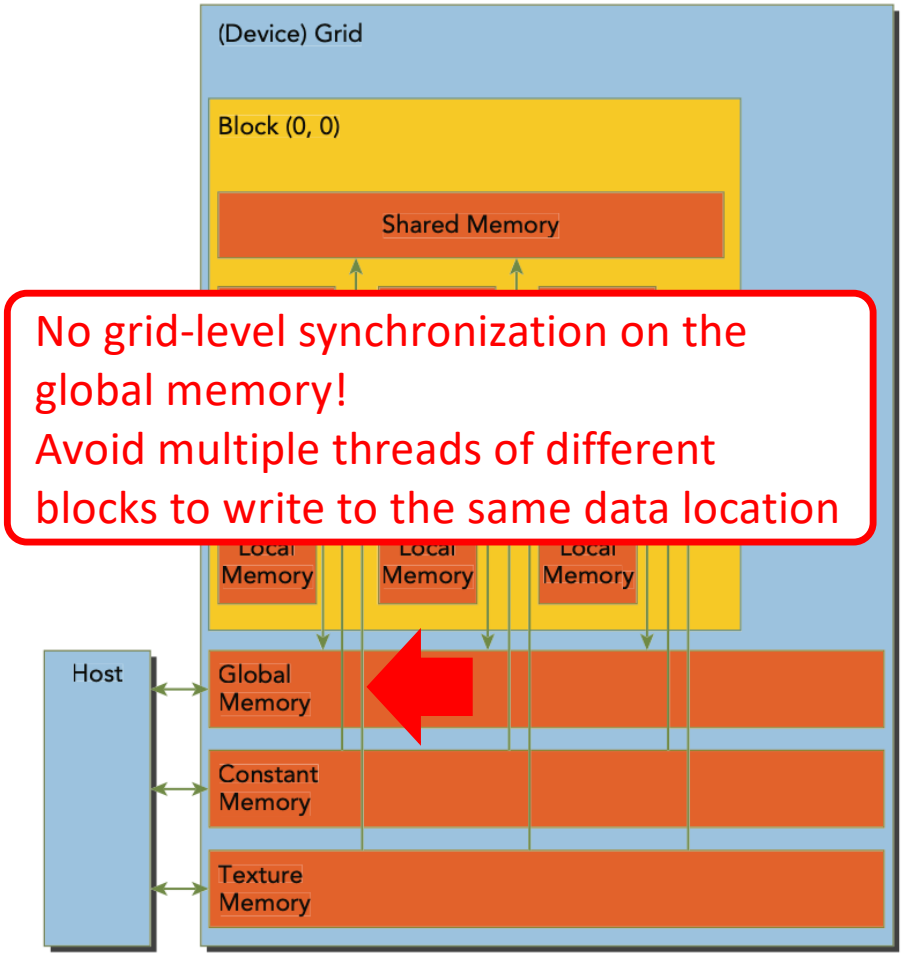
Discussed later…



Global memory

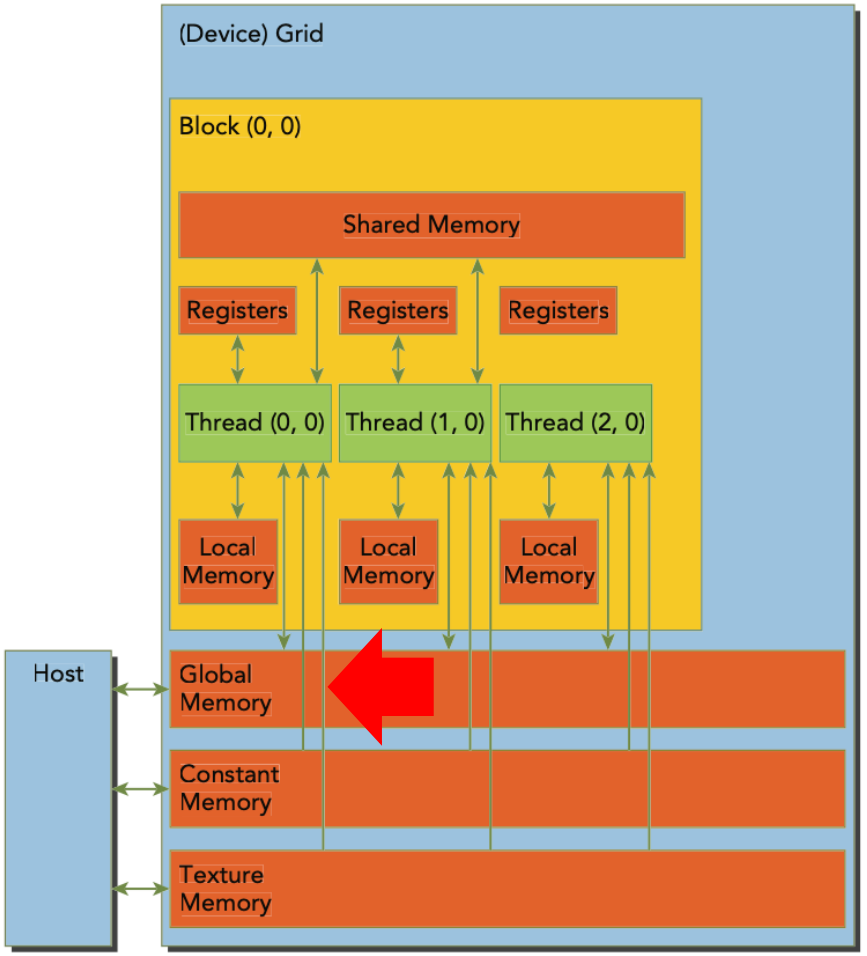
- Two possible ways to declare variables in the global memory
  - Static declaration
  - Dynamic allocation

- Global variables are declared in the host code
  - Access: read/write for all threads and host
  - Scope: application
  - Lifetime: application



Global memory

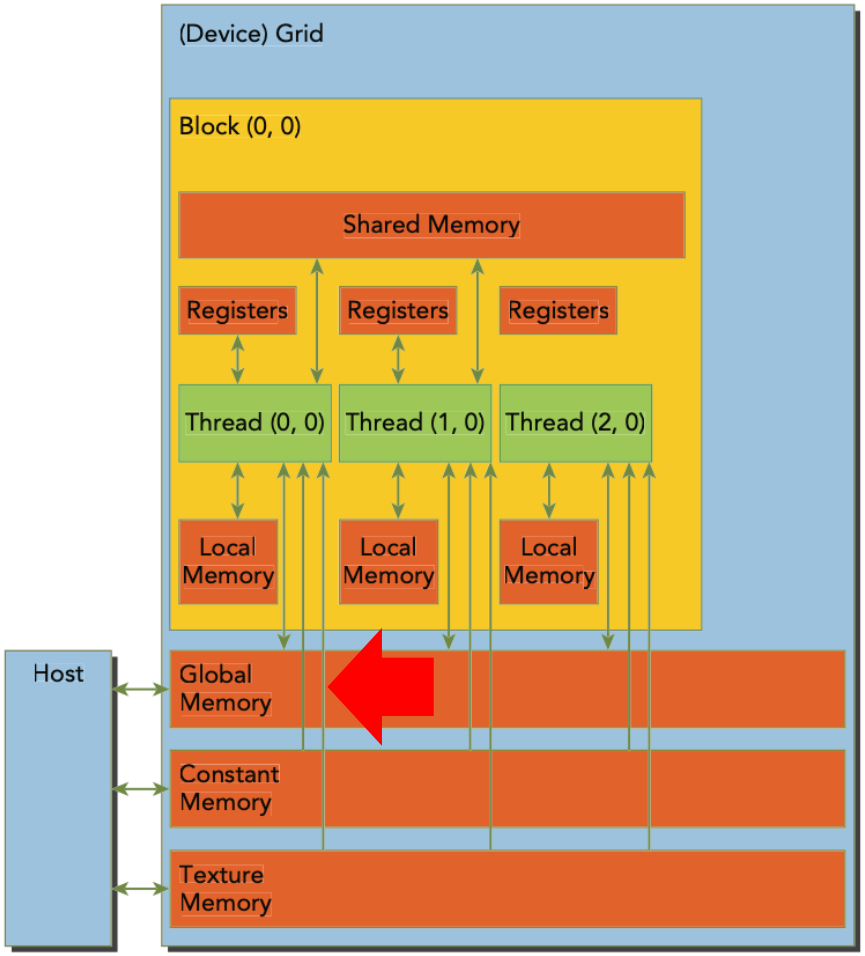
- Static declaration:
  - Declared as global variable
  - Specified `__device__` qualifier
- Access:
  - Host code accesses the variable by means of specific functions:
    - Host-> device: `cudaMemcpyToSymbol()`
    - Device->host: `cudaMemcpyFromSymbol()`
  - The kernel accesses the variable in the usual way



Global memory

- Example:

```
#define N 10

__device__ int g[N];

__global__ void foo(int* a){
    /*...*/
    g[i] = a[i];
    /*...*/
}

int main(){
  /*...*/
  int myarr[N];
  cudaMemcpyToSymbol(g, myarr, N*sizeof(int));
  /*...*/
  cudaMemcpyFromSymbol(myarr, g, N*sizeof(int));
}
```



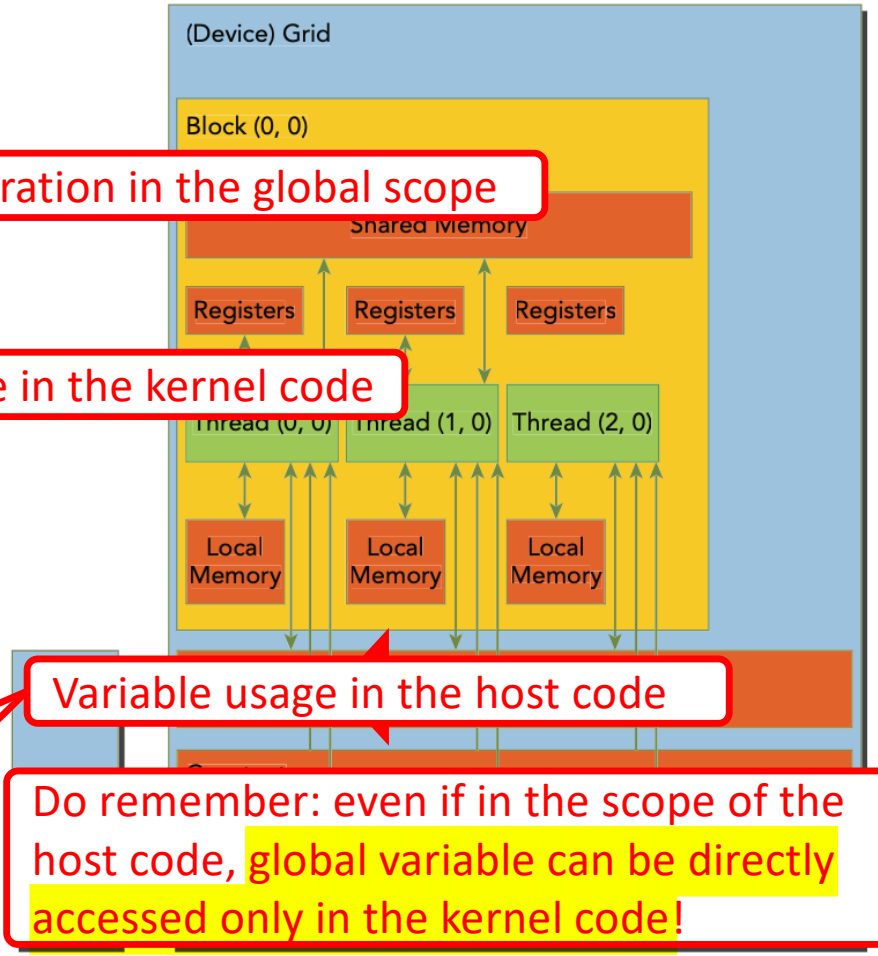
Global memory

- Example:

```
#define N 10

__device__ int g[N];

__global__ void foo(int* a){
    /*...*/
    g[i] = a[i];
    /*...*/
}

int main(){
  /*...*/
  int myarr[N];
  cudaMemcpyToSymbol(g, myarr, N*sizeof(int));
  /*...*/
  cudaMemcpyFromSymbol(myarr, g, N*sizeof(int));
}
```

Variable declaration in the global scope

Variable usage in the kernel code

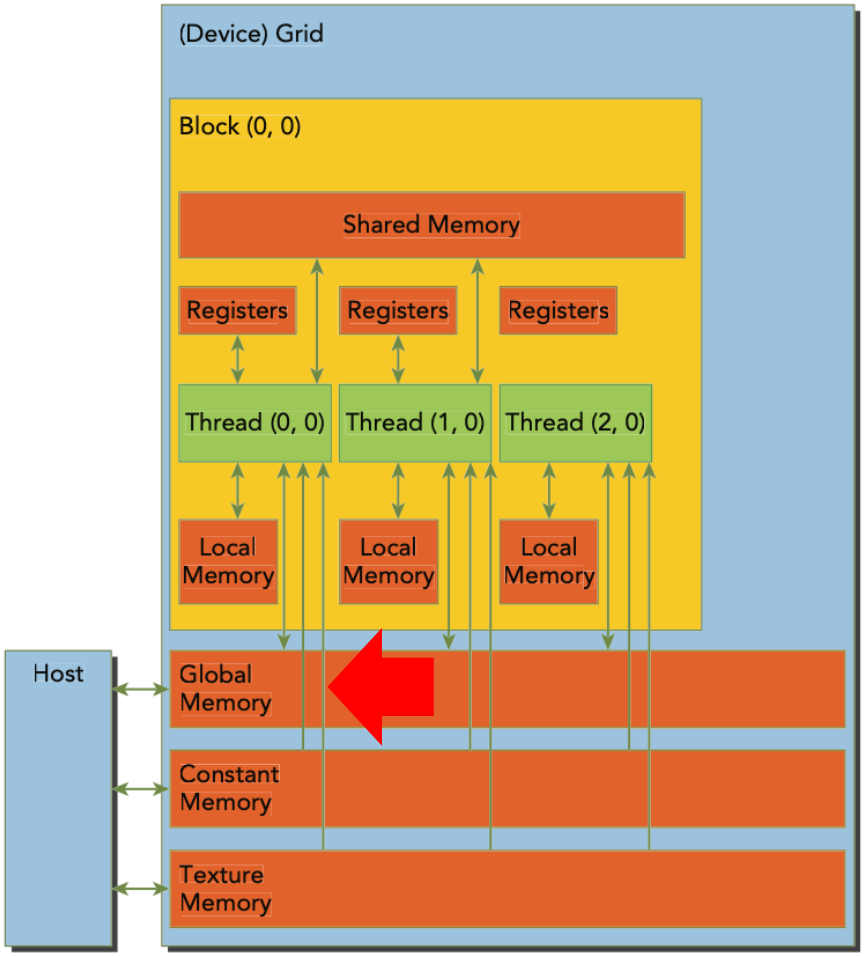


Variable usage in the host code

Do remember: even if in the scope of the host code, global variable can be directly accessed only in the kernel code!

Global memory

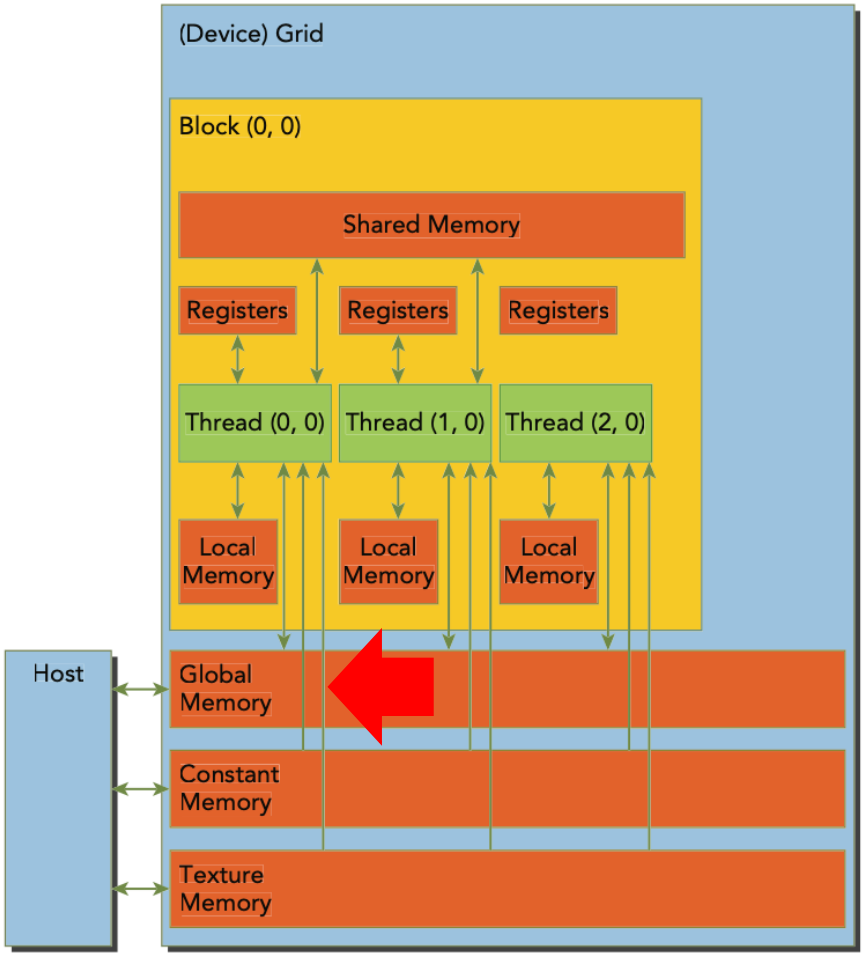
- Dynamic allocation:
  - Host declares the device pointer as a common C pointer
  - Host allocates the memory (`cudaMalloc()`) and release it (`cudaFree()`)
- Access:
  - Host code accesses the variable by means of the `cudaMemCpy()` function
  - The kernel the device pointed as function parameter and accesses it in the usual way



Global memory

- Example:

```
#define N 10

__global__ void foo(int* a){
    /*...*/
    a[i] = 10;
    /*...*/
}

int main(){
  /*...*/
  int myarr[N], *d;
  cudaMalloc(&d, N*sizeof(int));
  cudaMemcpy(d, myarr, N*sizeof(int),
                    cudaMemcpyHostToDevice);
  /*...*/
  cudaMemcpy(myarr, d, N*sizeof(int),
                    cudaMemcpyDeviceToHost);
  cudaFree(d);
}
```



Global memory

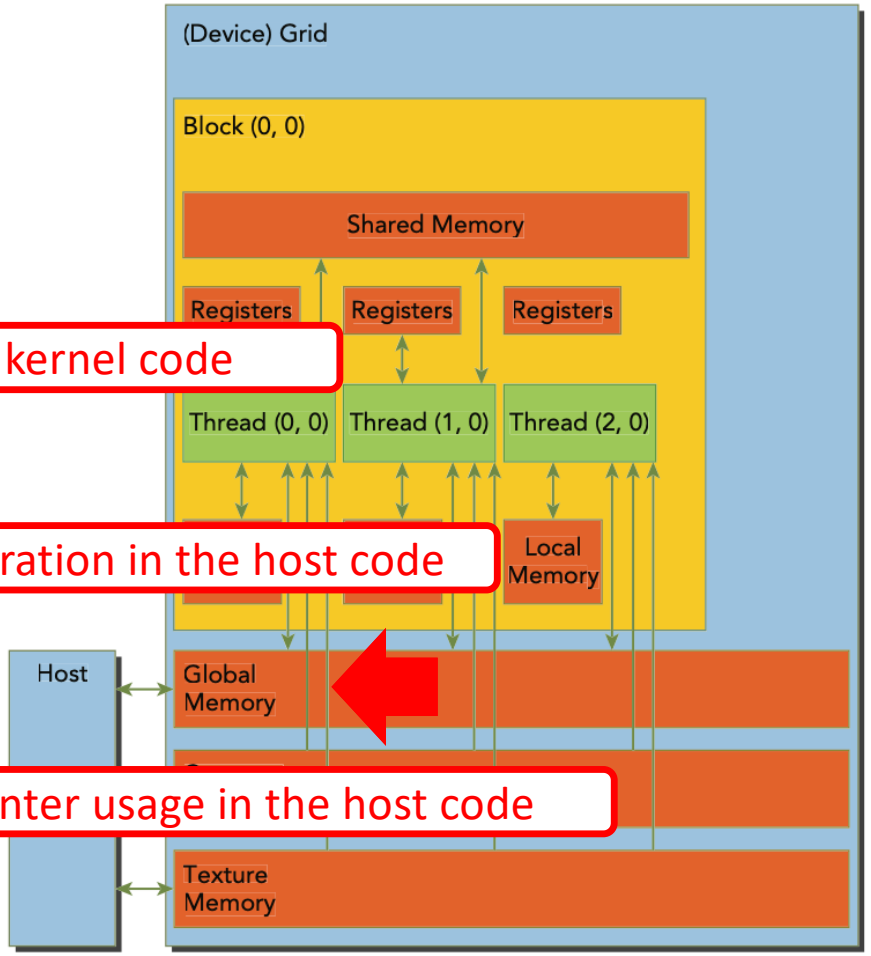
- Example:

```
#define N 10

__global__ void foo(int* a){
    /*...*/
    a[i] = 10;
    /*...*/
}

int main(){
  /*...*/
  int myarr[N], *d;
  cudaMalloc(&d, N*sizeof(int));
  cudaMemcpy(d, myarr, N*sizeof(int),
                    cudaMemcpyHostToDevice)
  /*...*/
  cudaMemcpy(myarr, d, N*sizeof(int),
                    cudaMemcpyDeviceToHost)
  cudaFree(d);
}
```

Variable usage in the kernel code

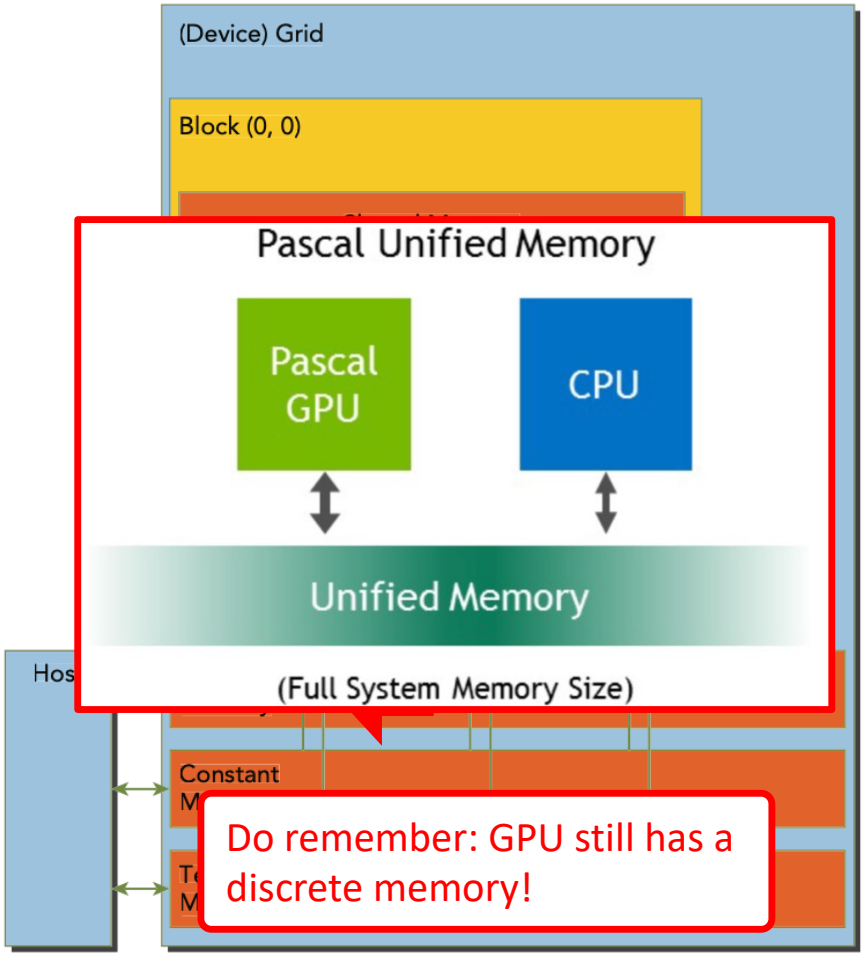
Pointer declaration in the host code

Pointer usage in the host code



Usage of the unified memory

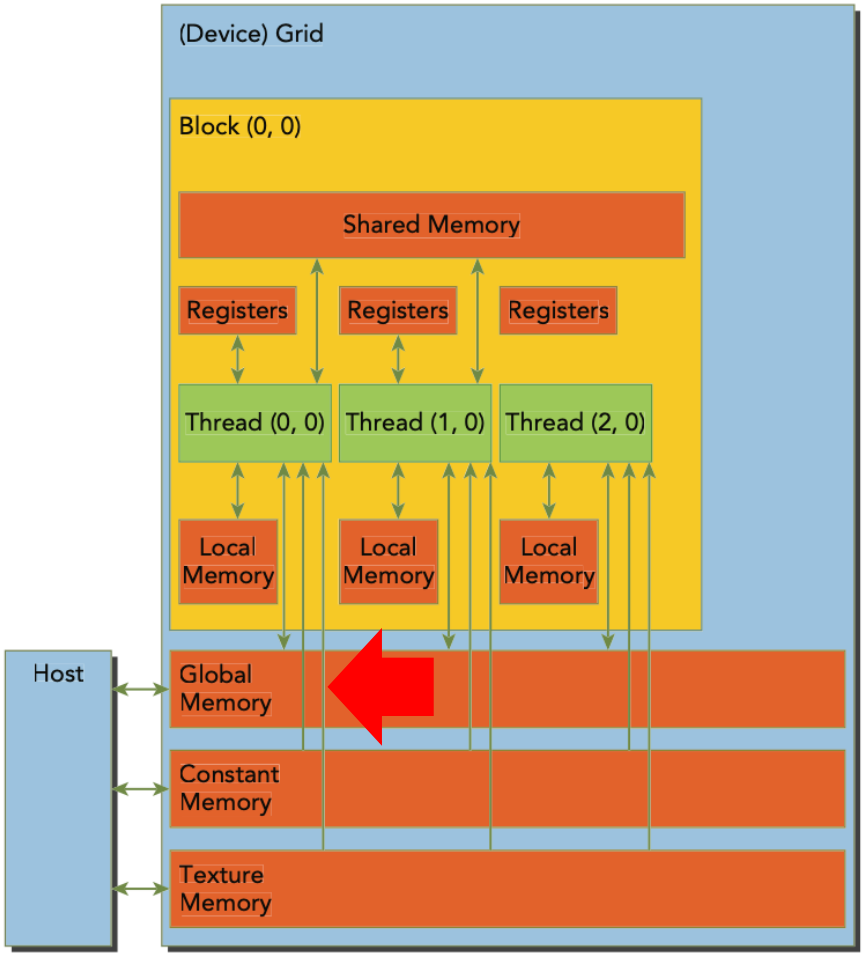
- CUDA (>=6.0 and on Kepler or later ones) supports a **unified memory** view on global memory
- It offers a unified virtual addressing between CPU and GPU
  - Memory pages are transparently transmitted between CPU and GPU memories
  - Automatic handling of page fault and global data coherence (depends on GPU generation)



Usage of the unified memory

- Dynamic memory allocation can be performed by means of `cudaMallocManaged()` function

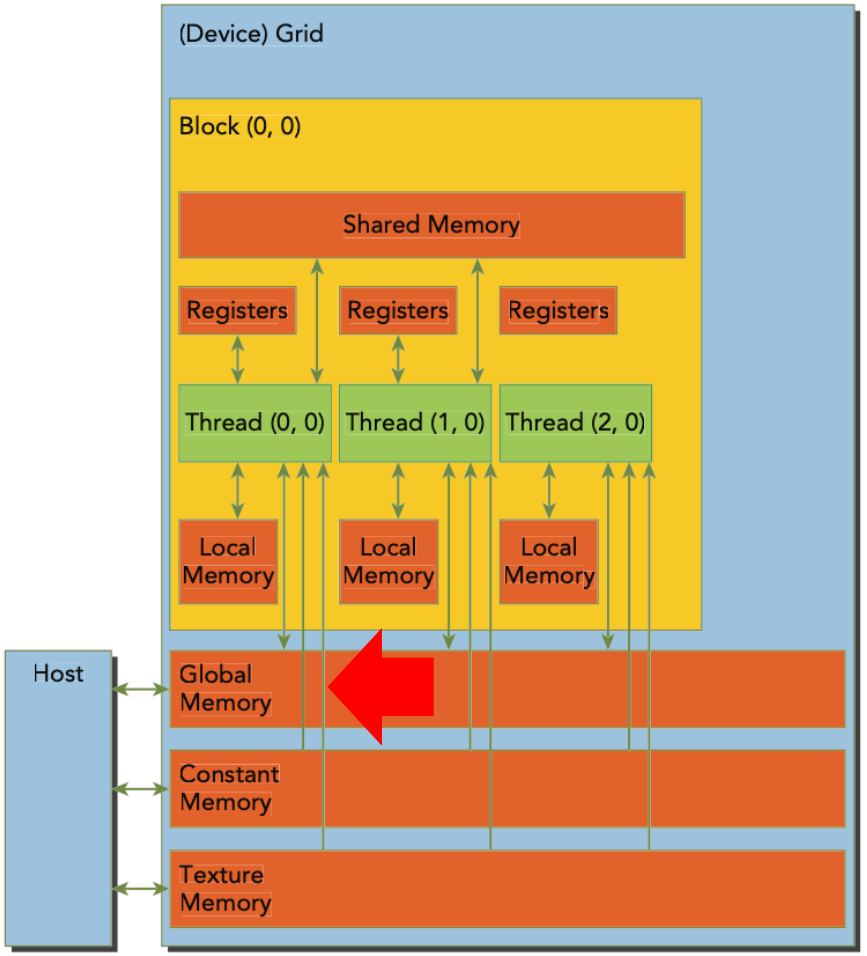
- The data transmission is transparently performed by CUDA framework
  - No need for function calls to perform memory transmission!
  - Performance depends also on the generation of the underlying GPU



Usage of the unified memory

- Example:

```
int main(){
  char *data;
  cudaMallocManaged(&data, N);
  for(int i=0; i<N; i++)
    data[i] = i; /*initialization*/
  foo<<<gdim, bdim>>>(data, N);
  cudaDeviceSynchronize();
  for(int i=0; i<N; i++)
    printf("%d", data[i]); /*use results*/
  cudaFree(data);
}
```



Usage of the unified memory

- Example:

```
int main(){
  char *data;
  cudaMallocManaged(&data, N);
  for(int i=0; i<N; i++)
    data[i] = i; /*initialization*/
  foo<<<gdim, bdim>>>(data, N);
  cudaDeviceSynchronize();
  for(int i=0; i<N; i++)
    printf("%d", data[i]); /*use results*/
  cudaFree(data);
}
```

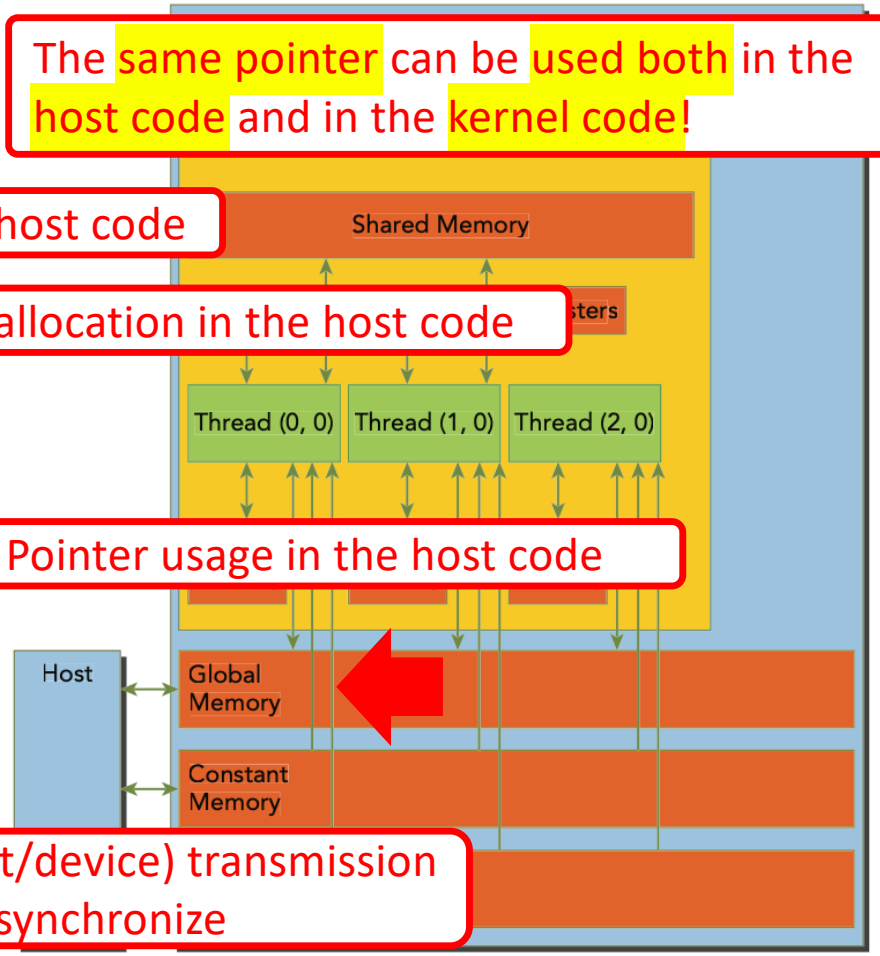
The same pointer can be used both in the host code and in the kernel code!

Pointer declaration in the host code

Memory allocation in the host code

Pointer usage in the host code

No device/host (or host/device) transmission of data -> necessity to synchronize

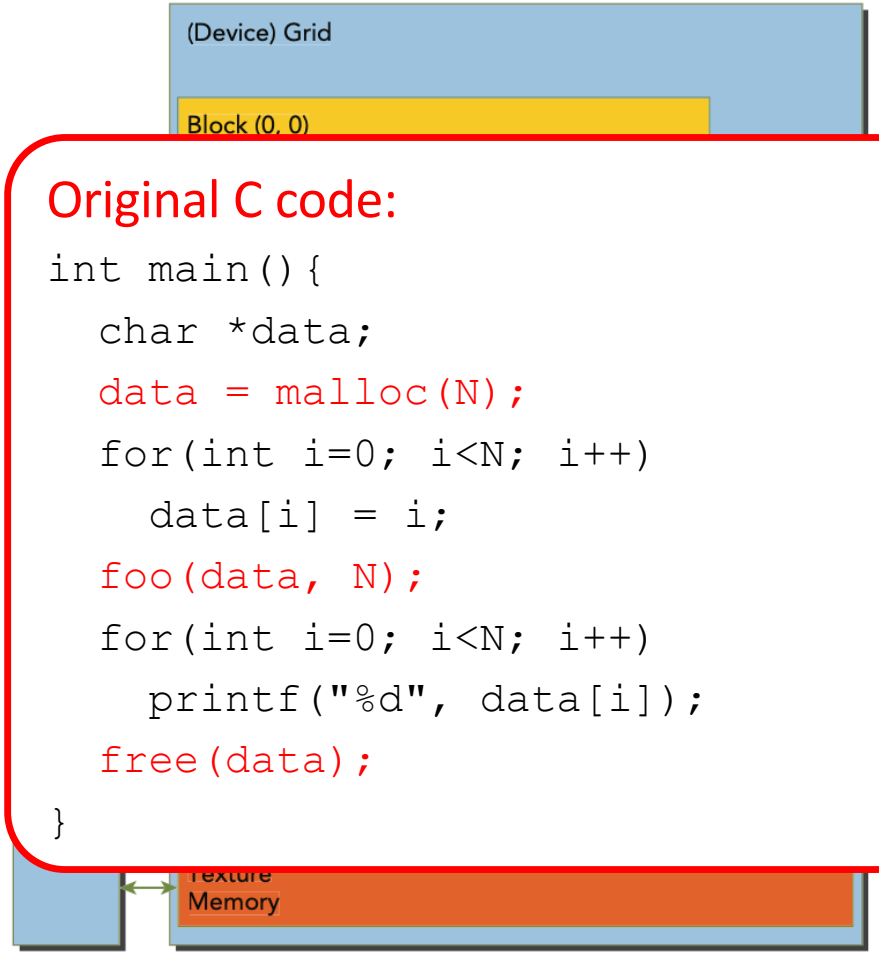


Usage of the unified memory

- Example:

```
int main(){
  char *data;
  cudaMallocManaged(&data, N);
  for(int i=0; i<N; i++)
    data[i] = i; /*initialization*/
  foo<<<gdim, bdim>>>(data, N);
  cudaDeviceSynchronize();
  for(int i=0; i<N; i++)
    printf("%d", data[i]); /*use results*/
  cudaFree(data);
}
```

Very few changes in the code!



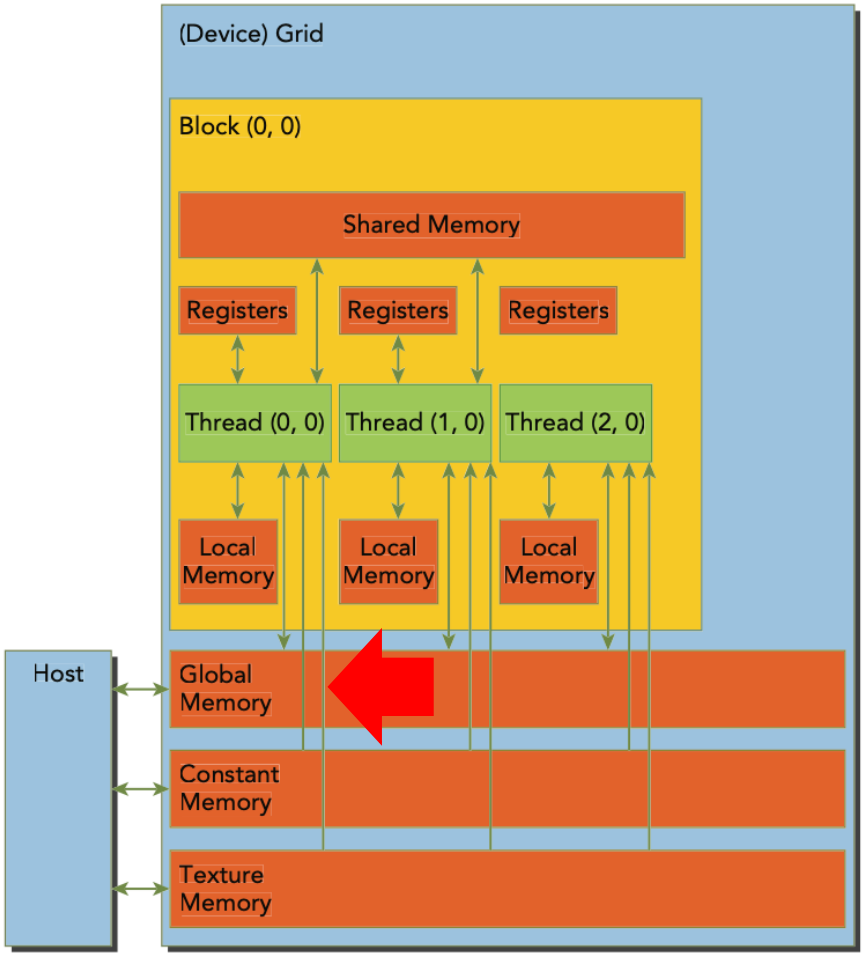
Original C code:

```
int main(){
  char *data;
  data = malloc(N);
  for(int i=0; i<N; i++)
    data[i] = i;
  foo(data, N);
  for(int i=0; i<N; i++)
    printf("%d", data[i]);
  free(data);
}
```

Usage of the unified memory

- Advantages:
  - Simpler porting of the code
  - Easier data management

- Disadvantages:
  - Complex (non-optimal) memory page management
  - Lower performance
  - Not supported by all devices (e.g., Tegra)

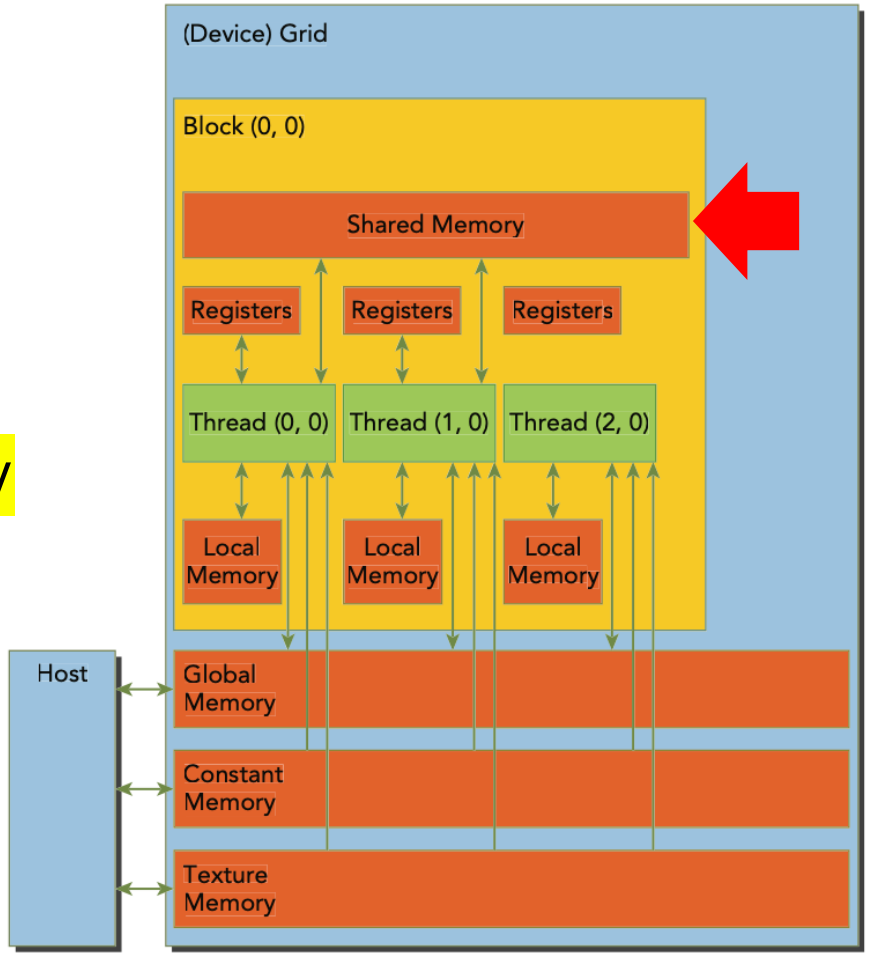


Shared memory

- The shared memory is a very fast scratchpad memory integrated in the SM
- Purposes:
  - Intra-block data sharing and thread cooperation
  - Reduce accesses to the global memory

- Together with registers, shared memory represents a constraint on the number of blocks assignable to a SM

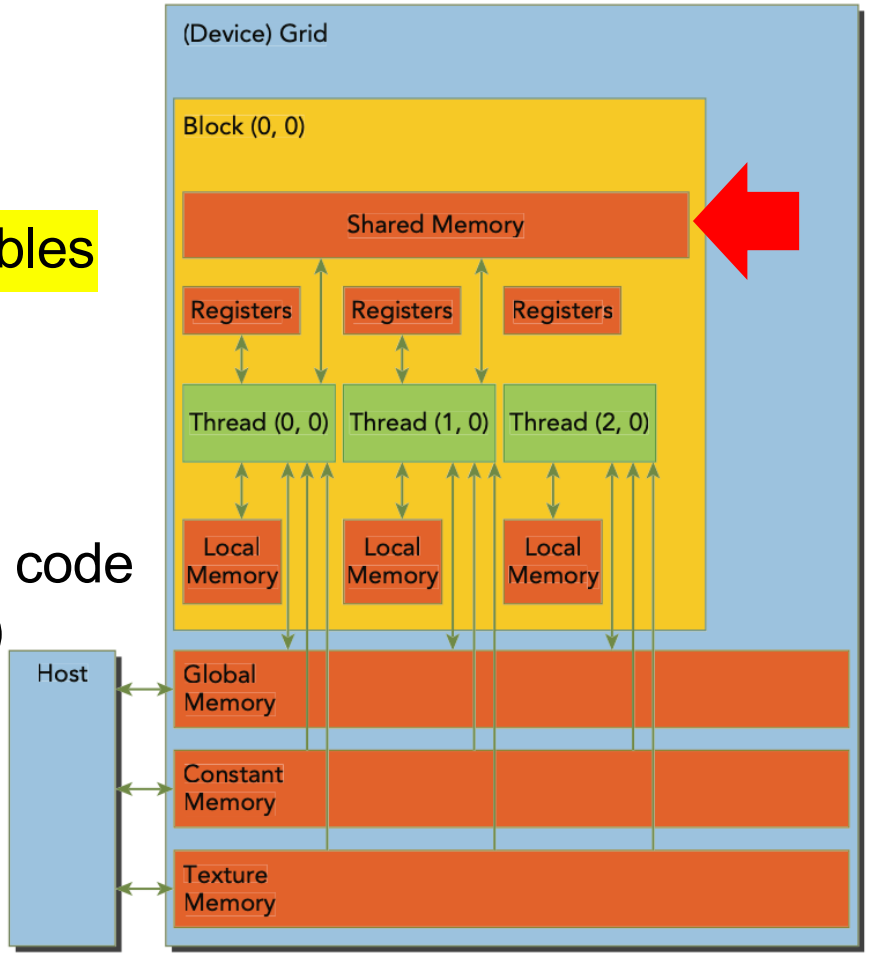
To compute the shared memory required by a block: sum of the result of `sizeof()` operator applied on each variable stored in such a memory



Shared memory

- The shared variable is indicated by the `__shared__` qualifier
- Two possible ways to allocate shared variables
  - Static allocation
  - Dynamic allocation

- Shared variables are declared in the kernel code or globally (i.e., declared for all the kernels)
  - Access: read/write for all block threads
  - Scope: block
  - Lifetime: block

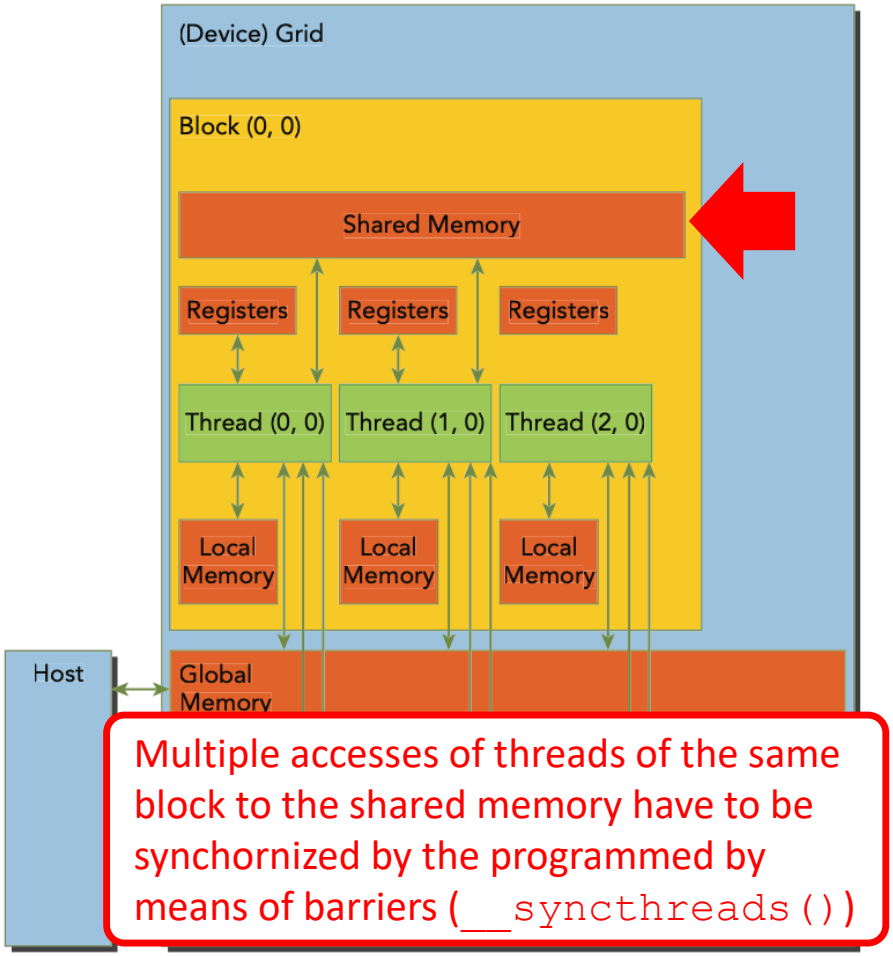


Shared memory

- Static allocation:

```
__global__ void foo(int* a, int* b) {
  __shared__ int sv[N];
  int i = threadIdx.x;
  sv[i] = a[i];
  __syncthreads();
  /*...elaborate...*/
  __syncthreads();
  b[i] = sv[(i+1)%N];
}
```

- CUDA supports declaration of up to 3D array variables

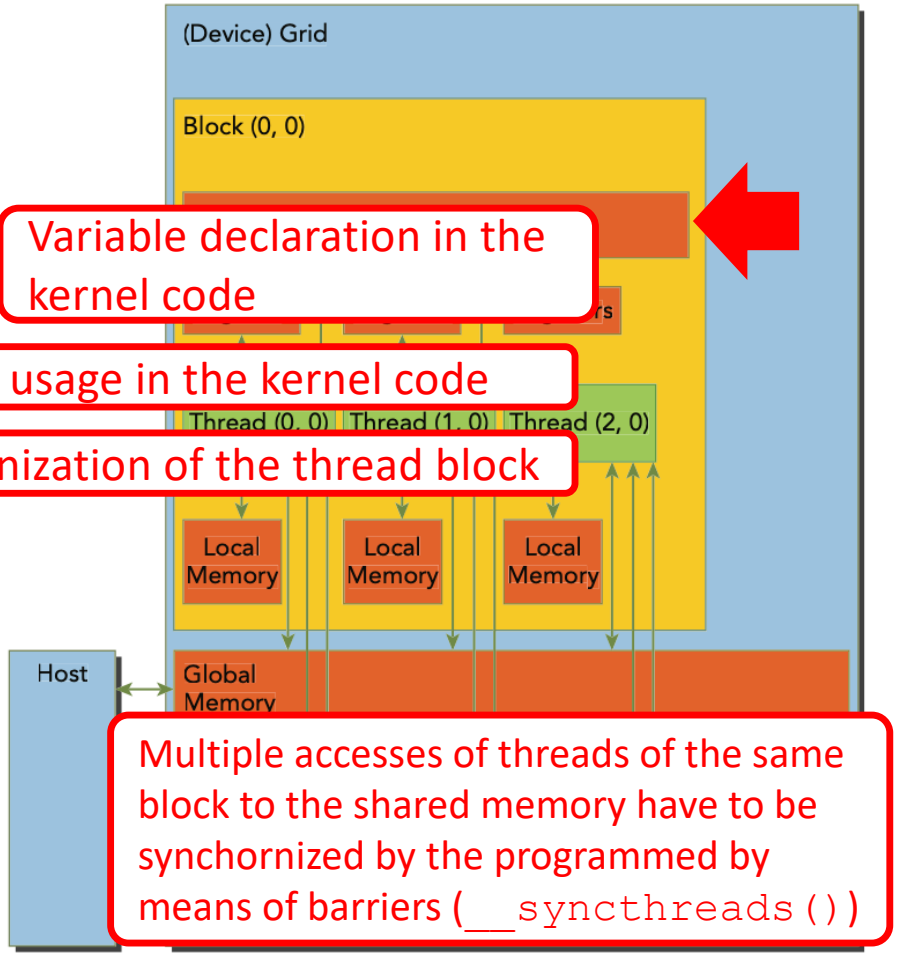


Shared memory

- Static allocation:

```
__global__ void foo(int* a, int* b) {
  __shared__ int sv[N];
  int i = threadIdx.x;
  sv[i] = a[i];
  __syncthreads();
  /*...elaborate...*/
  __syncthreads();
  b[i] = sv[(i+1)%N];
}
```

- CUDA supports declaration of up to 3D array variables

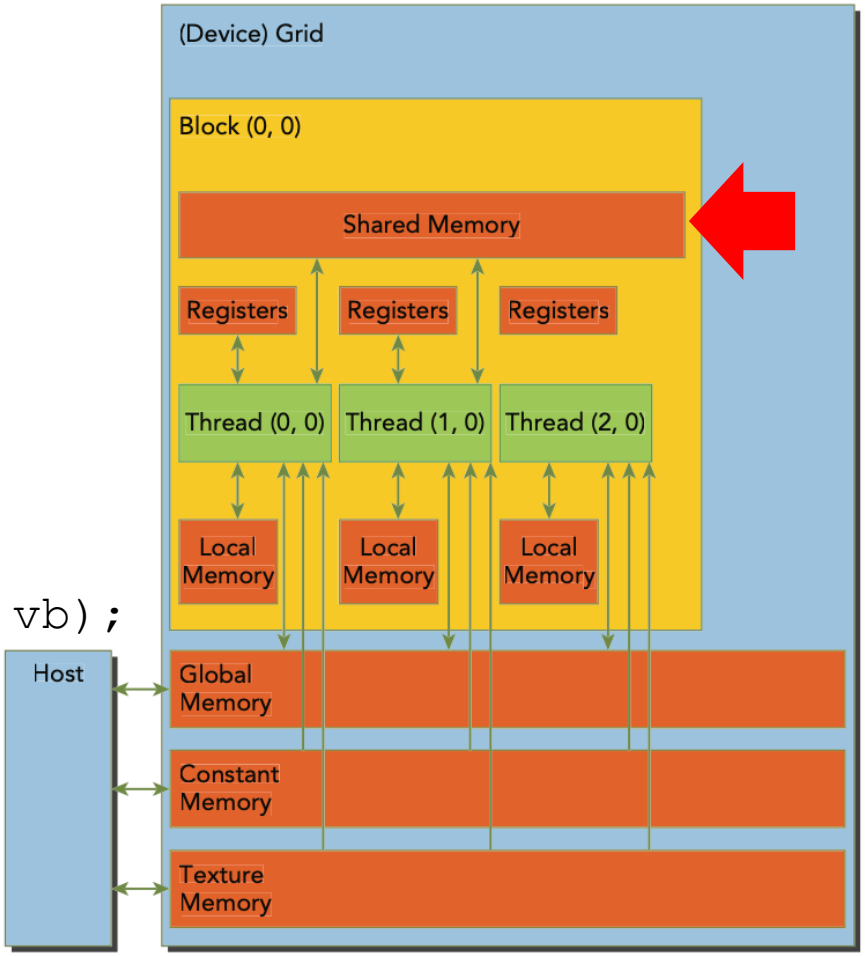


Shared memory

- Dynamic allocation:

```
__global__ void foo(int* a, int* b) {
  extern __shared__ int sv[];
  /*...*/
}

int main(){
  /*...*/
  foo<<gdim, bdim, sizeof(int)*N>>(va, vb);
}
```

- CUDA supports dynamic allocation of only 1D arrays
  - Allocation of multiple variables is tricky

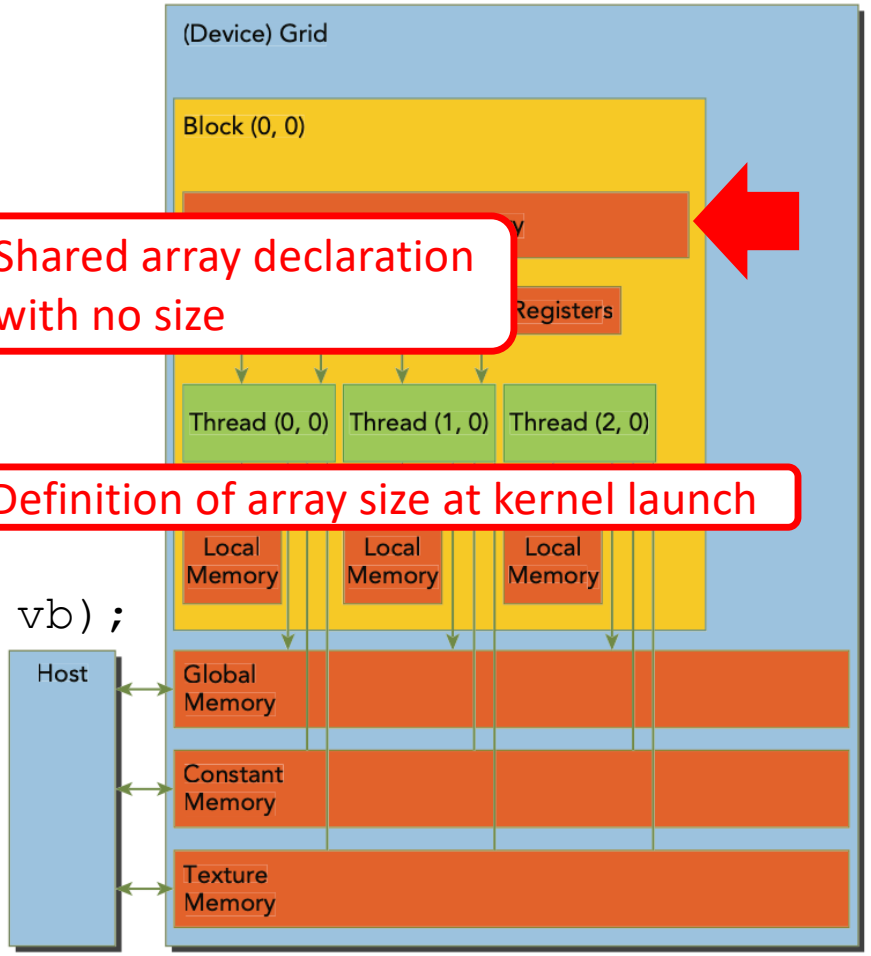


Shared memory

- Dynamic allocation:

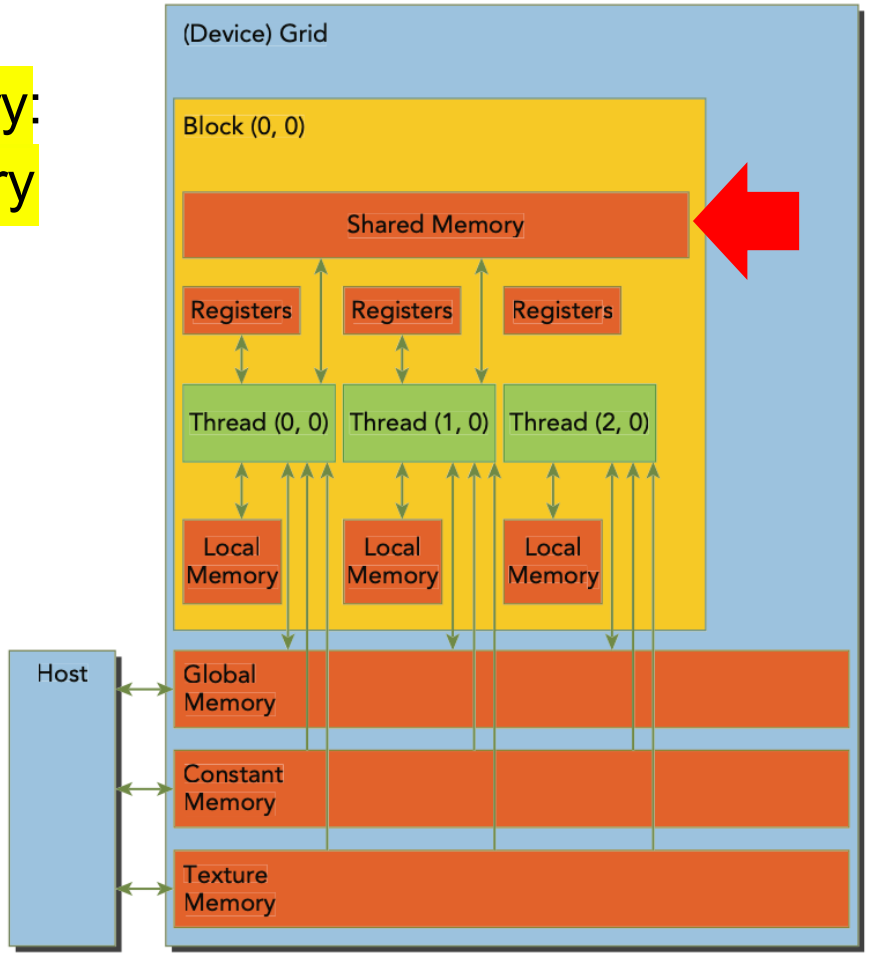
```
__global__ void foo(int* a, int* b) {
  extern __shared__ int sv[];
  /*...*/
}

int main(){
  /*...*/
  foo<<gdim, bdim, sizeof(int)*N>>(va, vb);
}
```

- CUDA supports dynamic allocation of only 1D arrays
  - Allocation of multiple variables is tricky



Shared memory

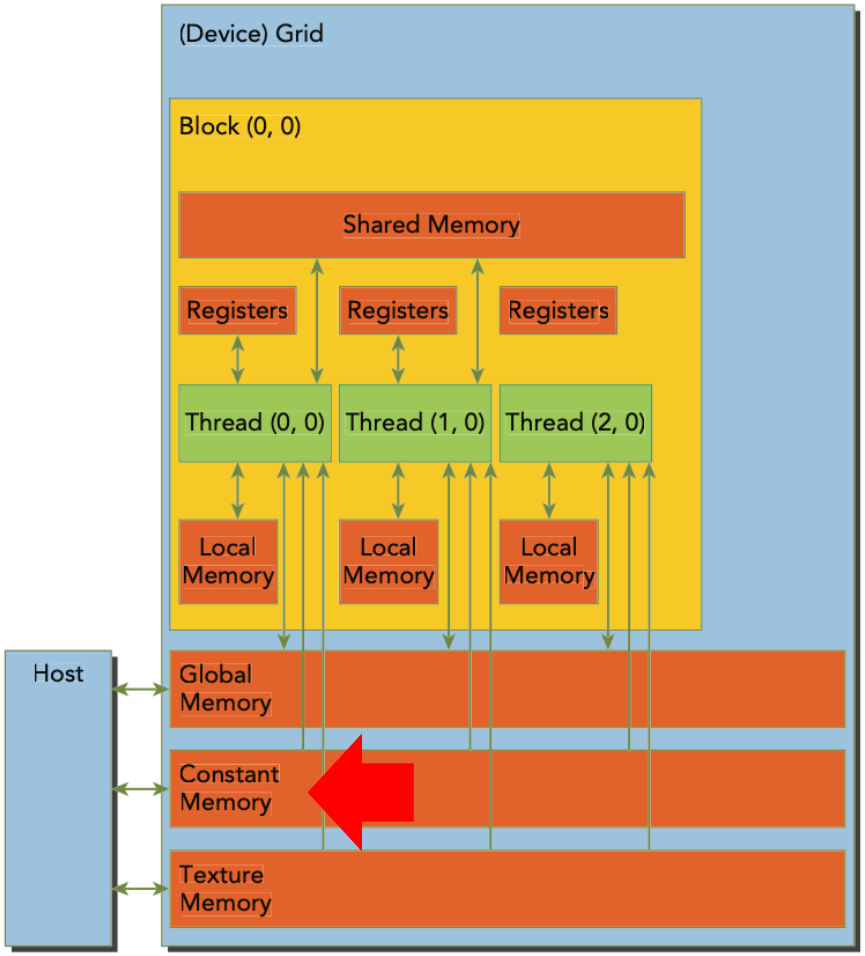
- Typical kernel workflow with shared memory:
  - Parallel load of data from global memory to shared memory
  - Synchronization of block threads
  - Elaboration of block threads on shared memory data
  - Synchronization of block threads
  - Parallel store of data from shared memory to global memory



Constant memory

- Constant memory resides in the device memory
- It has a reserved cache in the SM
- It is read-only from the device

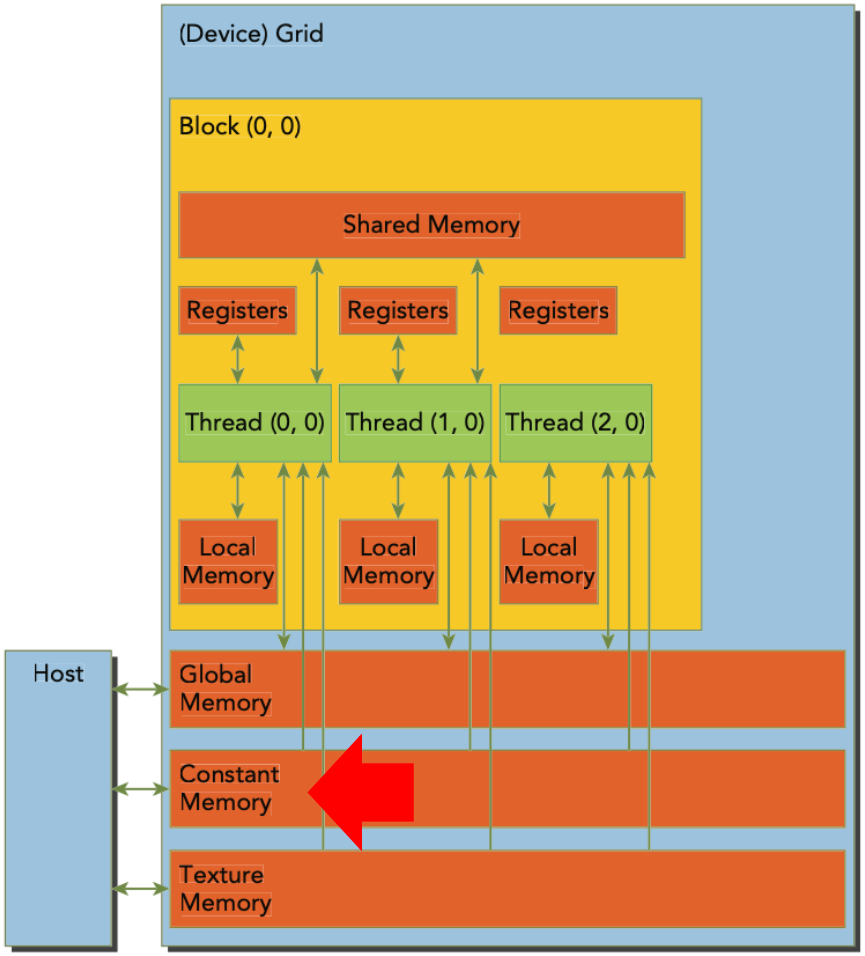
- This memory achieves best performance when all threads in a warp read from the same memory address concurrently
  - Each single read is broadcasted to the warp!
- The amount of data should be small!



Constant memory

- Variables in the constant memory are declared in the host code
  - Access:
    - Read/write for host
    - Read for all the threads
  - Scope: application
  - Lifetime: application

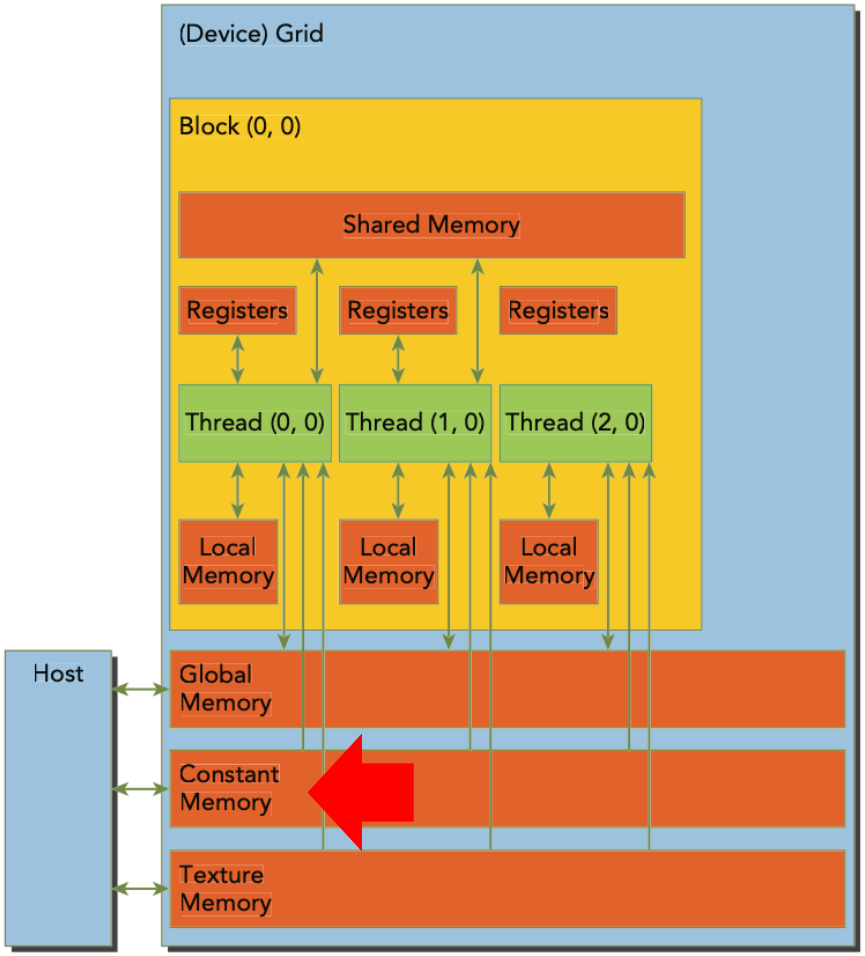
Do note confuse with `#define` macros!



Constant memory

- Declaration:
  - Declared as global variable
  - Specified `__constant__` qualifier

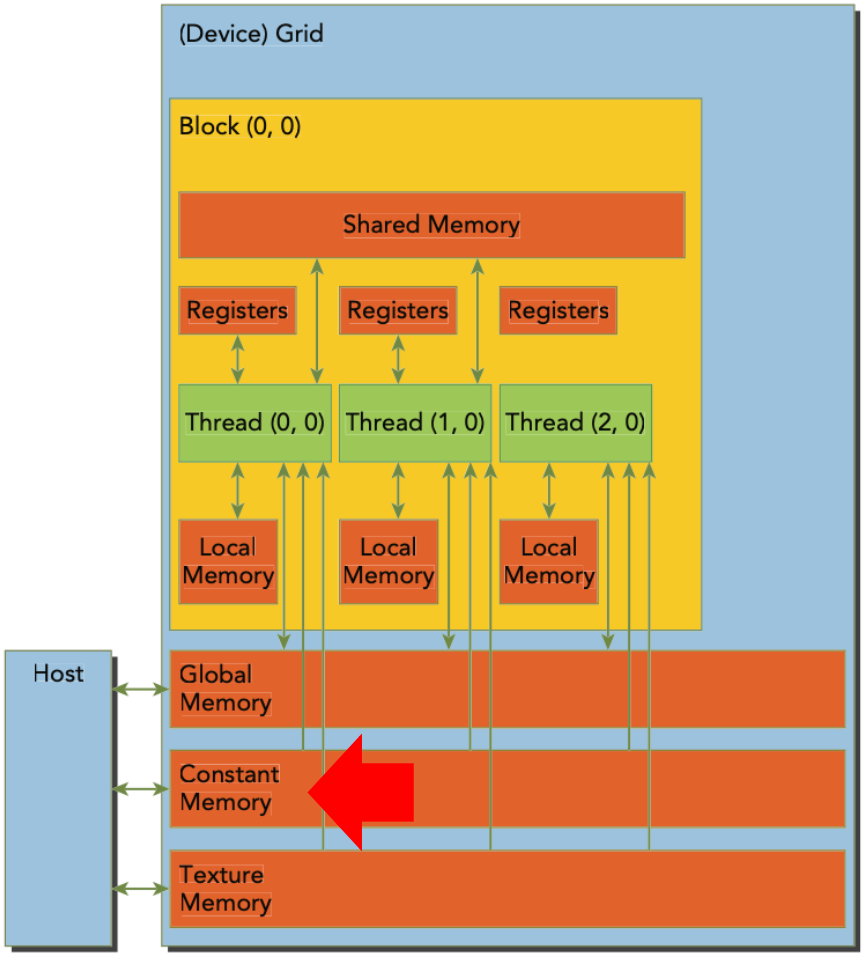
- Access:
  - Host code accesses the variable by means of specific functions:
    - Host-> device: `cudaMemcpyToSymbol()`
    - Device->host: `cudaMemcpyFromSymbol()`
  - The kernel accesses the variable in the usual way



Constant memory

- Example:

```
#define N 10

__constant__ int g[N];

__global__ void foo(int* a){
    /*...*/
    a[i] = g[0];
    /*...*/
}

int main(){
  /*...*/
  int myarr[N];
  cudaMemcpyToSymbol(g, myarr, N*sizeof(int));
  /*...*/
  cudaMemcpyFromSymbol(myarr, g, N*sizeof(int));
}
```



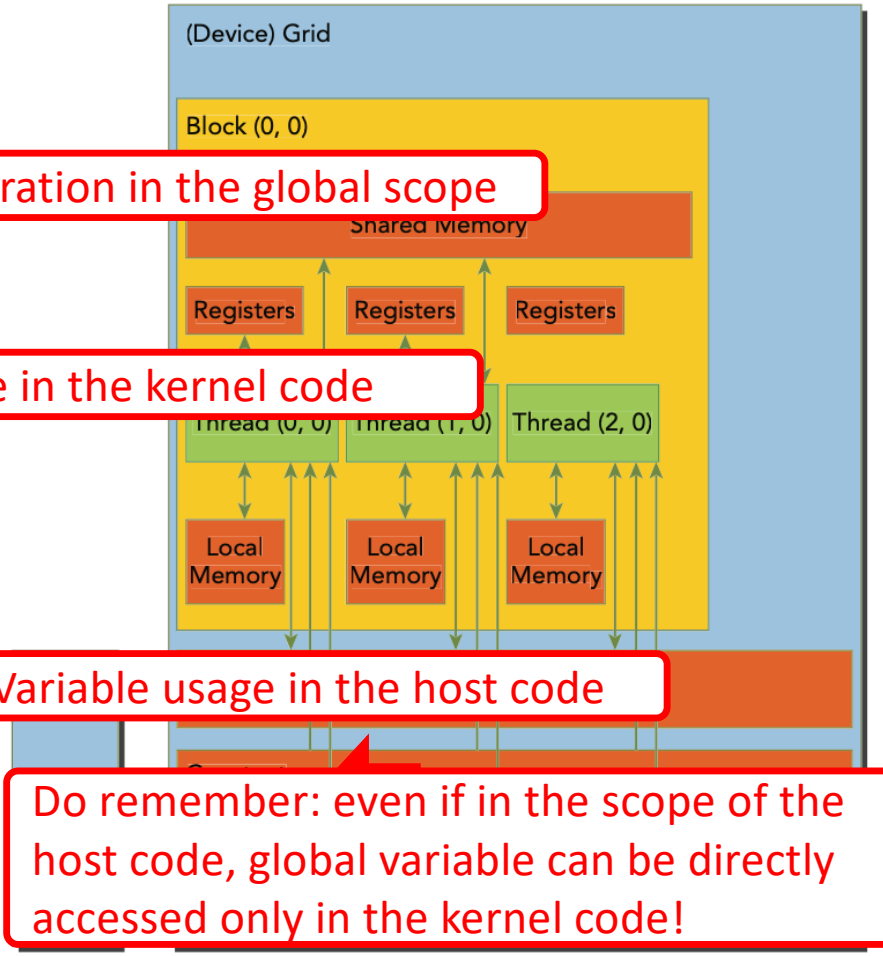
Constant memory

- Example:

```
#define N 10

__constant__ int g[N];

__global__ void foo(int* a){
    /*...*/
    a[i] = g[0];
    /*...*/
}

int main(){
  /*...*/
  int myarr[N];
  cudaMemcpyToSymbol(g, myarr, N*sizeof(int));
  /*...*/
  cudaMemcpyFromSymbol(myarr, g, N*sizeof(int));
}
```

Variable declaration in the global scope

Variable usage in the kernel code



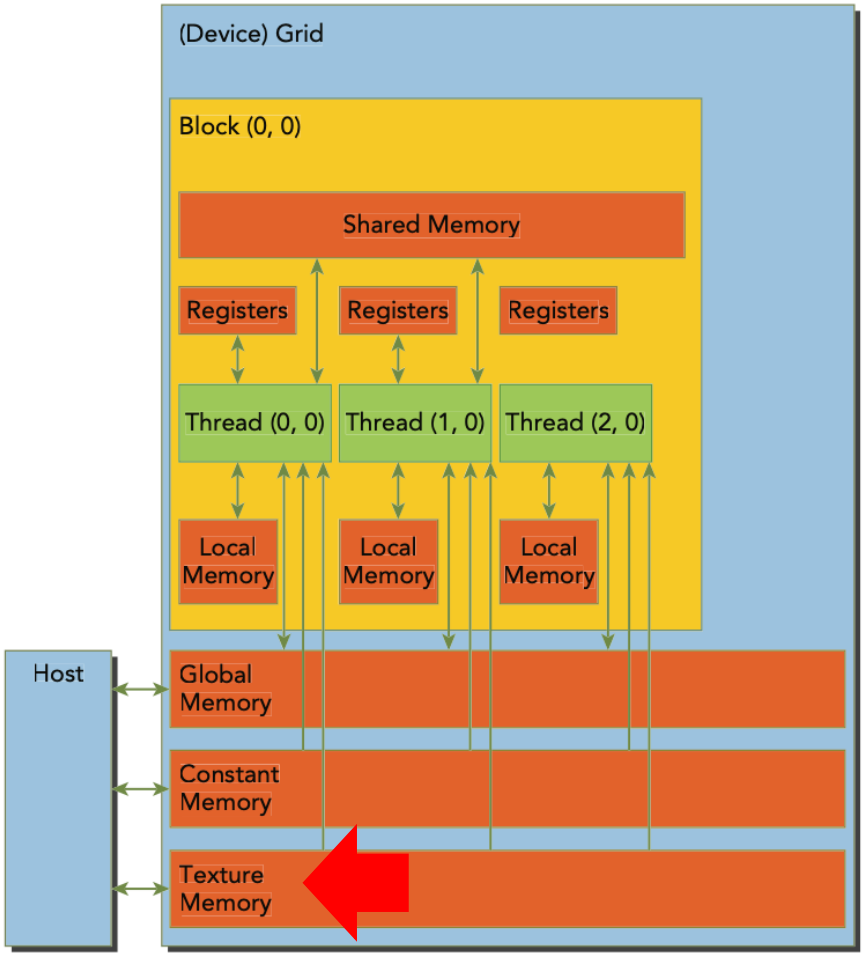
Variable usage in the host code

Do remember: even if in the scope of the host code, global variable can be directly accessed only in the kernel code!

Texture/read-only memory

- Texture/read-only memory resides in the device memory
- It has a reserved cache in the SM
- It is read-only from the device
- The cache includes support for hardware filtering (e.g., data interpolation while reading)

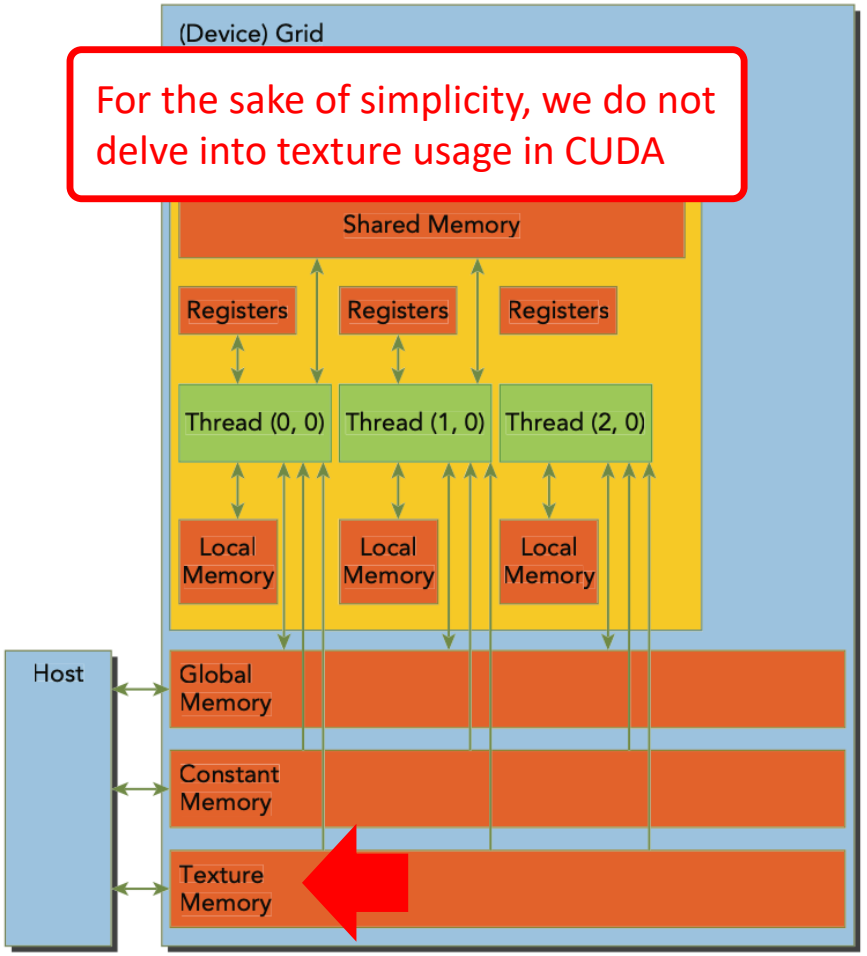
- This memory is optimized for image/texture processing
  - It achieves high performance with 2D data



Texture/read-only memory

- Read-only variables are managed in the code in the same way of global memories (dynamic application)
  - Access:
    - Read/write for host
    - Read for all the threads
  - Scope: application
  - Lifetime: application

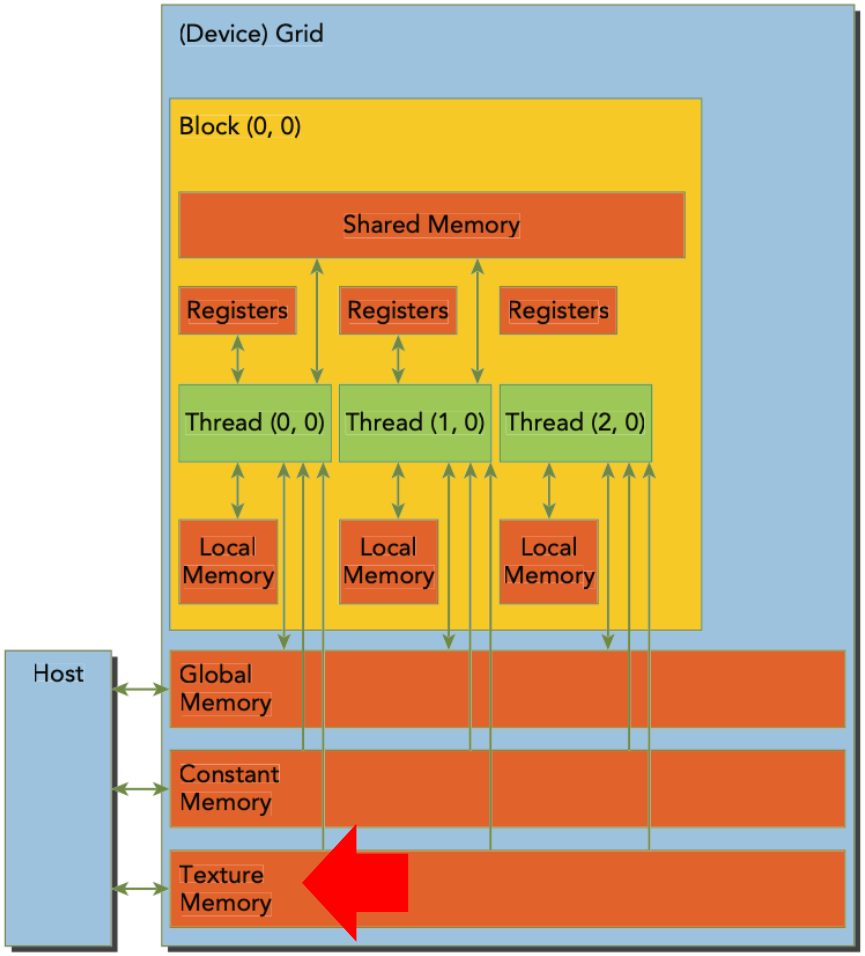
- To access a read-only variable `__ldg()` intrinsic function has to be used
- The kernel parameter is qualified as `const __restrict__`



Texture/read-only memory

- Example:

```
#define N 10

__global__ void foo(const int * __restrict__ a){
    /*...*/
    int c = __ldg(a[0]);
    /*...*/
}

int main(){
  /*...*/
  int myarr[N], *d;
  cudaMalloc(&d, N*sizeof(int));
  cudaMemcpy(d, myarr, N*sizeof(int),
                   cudaMemcpyHostToDevice);
  /*...*/
  cudaMemcpy(myarr, d, N*sizeof(int),
                   cudaMemcpyDeviceToHost);
  cudaFree(d);
}
```



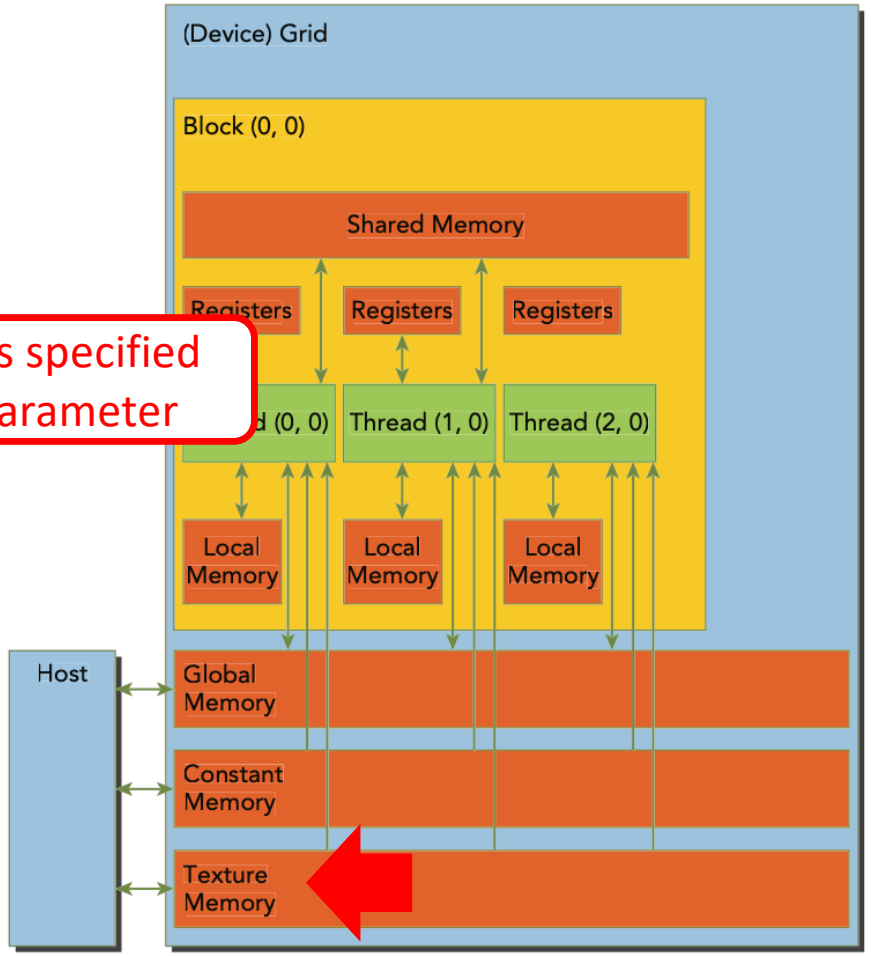
Texture/read-only memory

- Example:

```
#define N 10

__global__ void foo(const int * __restrict__ a){
    /*...*/
    int c = __ldg(a[0]);
    /*...*/
}

int main(){
  /*...*/
  int myarr[N], *d;
  cudaMalloc(&d, N*sizeof(int));
  cudaMemcpy(d, myarr, N*sizeof(int),
                    cudaMemcpyHostToDevice);
  /*...*/
  cudaMemcpy(myarr, d, N*sizeof(int),
                    cudaMemcpyDeviceToHost);
  cudaFree(d);
}
```

Forces read through read-only cache

Qualifiers specified for the parameter



CUDA variables and type qualifiers

	Variable name	Memory	Scope	Lifetime
	`int a`	Register	Thread	Thread
	`int arr[10]`	Local	Thread	Thread
`__shared__`	`int a` or `int arr[10]`	Shared	Block	Block
`__device__`	`int a` or `int arr[10]`	Global	Global	Application
`__constant__`	`int a` or `int arr[10]`	Constant	Global	Application
`__restricted__`	`constant int *p`	Texture/ Read-only	Global	Application

Summary on device memories

Memory	On/Off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in a block	Block
Global	Off	Yes	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture/ Read-only	Off	Yes	R	All threads + host	Host allocation

GPUs and Heterogeneous Systems
(programming models and architectures)

Usage of the global memory

Importance of memory access efficiency

- The kernel code is a mix arithmetic operations and memory accesses
  - Global memory accesses are considerably slower than arithmetic operations
  - If the arithmetic/memory instruction ratio is not good enough, the kernel is **memory-bounded**, i.e., performance is limited by memory accesses

- Thus, it is fundamental for the programmer to properly use the various memories
  - Where to allocate each variable
  - Follow specific memory access patterns to maximize memory efficiency
    - Specific for each memory type
    - Global memory ones are the most important for high performance

Let's focus on this aspect

L1/L2 cache organization

- All device memory accesses go though the L1/L2 caches
  - L1/L2 caches are accessed per single cache lines
  - In recent architectures lines are divided in 4 sections
- Access granularity
  - Read accesses: 32 bytes for L2
    - 128 bytes for L1 (Compute capability <5.3) Old GPUs!
    - 32 bytes for L1 (Compute capability >=5.3)
  - Write accesses: 32 bytes
- Data are aligned at 32-byte granularity in the cache
- A **transaction** is an **aligned access to a single cache line/sector**
  - Access to sectors of the same line are packed in a single transaction…

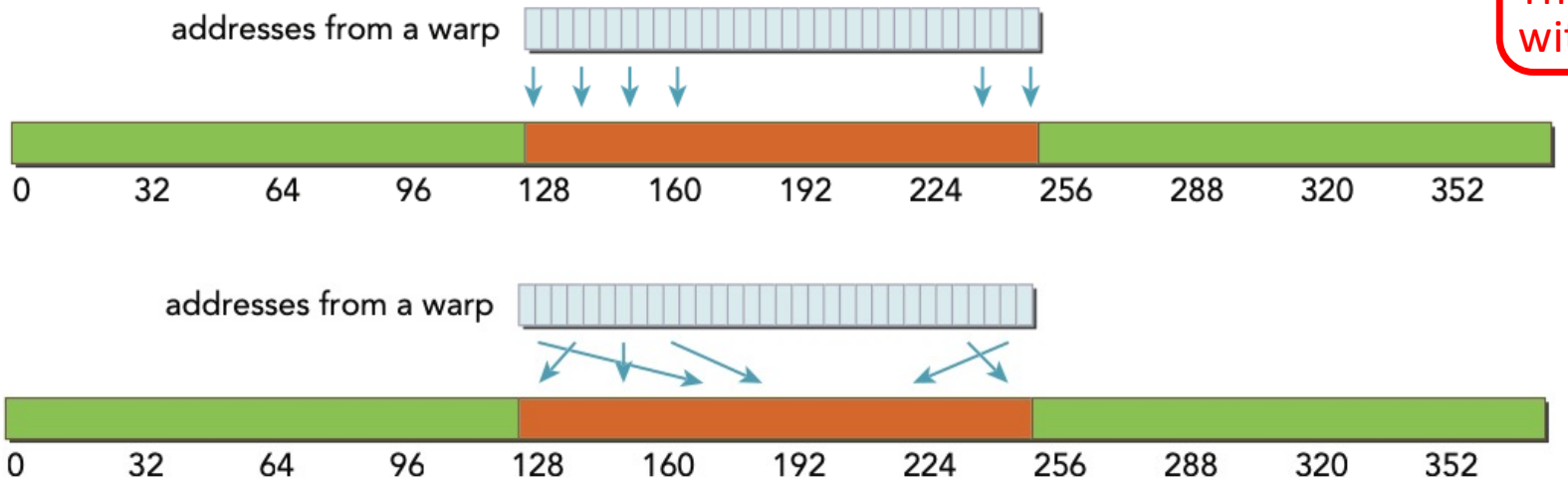
We will not delve into further details

Quest for aligned coalesced accesses

- Threads work in warps
  - They generate 32 memory access requests in a single time
- When the warp performs a memory access, the request is divided in 1 or many transactions based on the alignment and distribution of the addresses
  - **Coalesced memory accesses** occur when all 32 threads in a warp access a contiguous chunk of memory
- In order to optimize global memory usage, we need to maximize the number of **32byte coalesced aligned accesses**

Global memory access patterns

- Examples of read aligned coalescent accesses



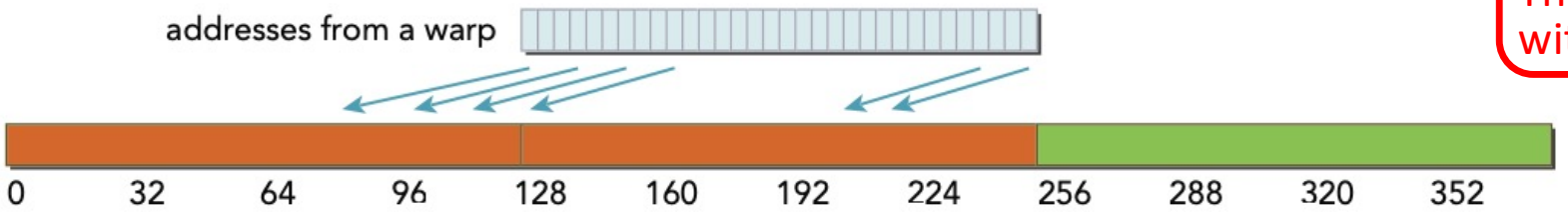
Considering compute capability <5.3
The behavior is similar with newer GPUs

- Both the two cases generate 1 transaction (100% efficiency)
- It is not relevant in the second case that addresses are nonconsecutive if they are all different from each other

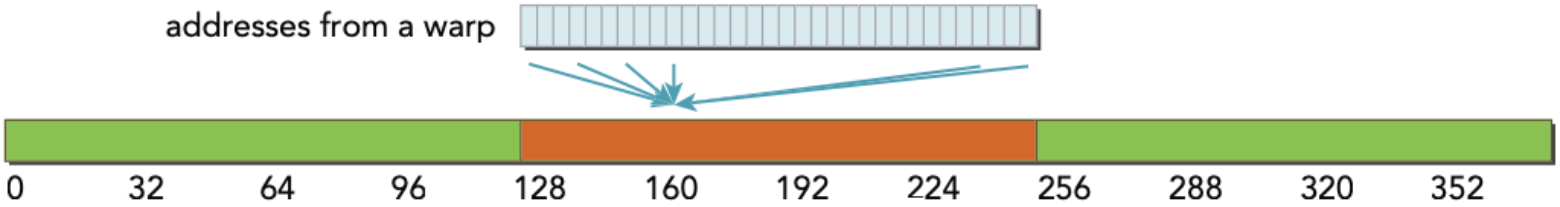
Global memory access patterns

- Examples of read misaligned non-coalescent accesses

Considering compute capability <5.3
The behavior is similar with newer GPUs



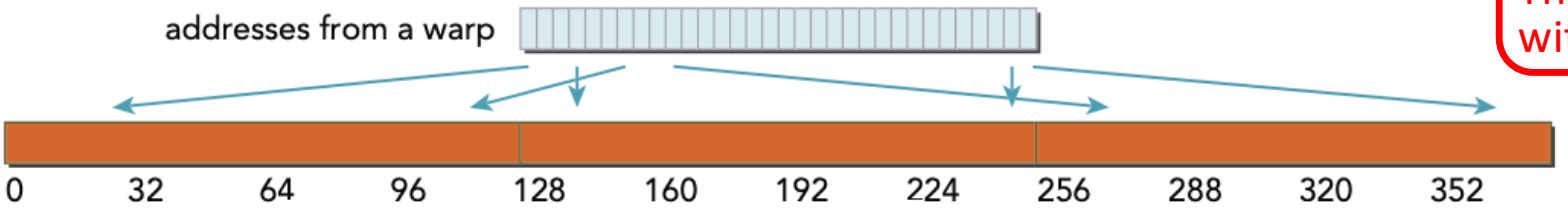
- Not aligned request causing 2 transactions (50% efficiency)



- All thread accessing the same address (3.125% efficiency)
- This is the pattern required for using constant memory!

Global memory access patterns

- Examples of read misaligned non-coalescent accesses

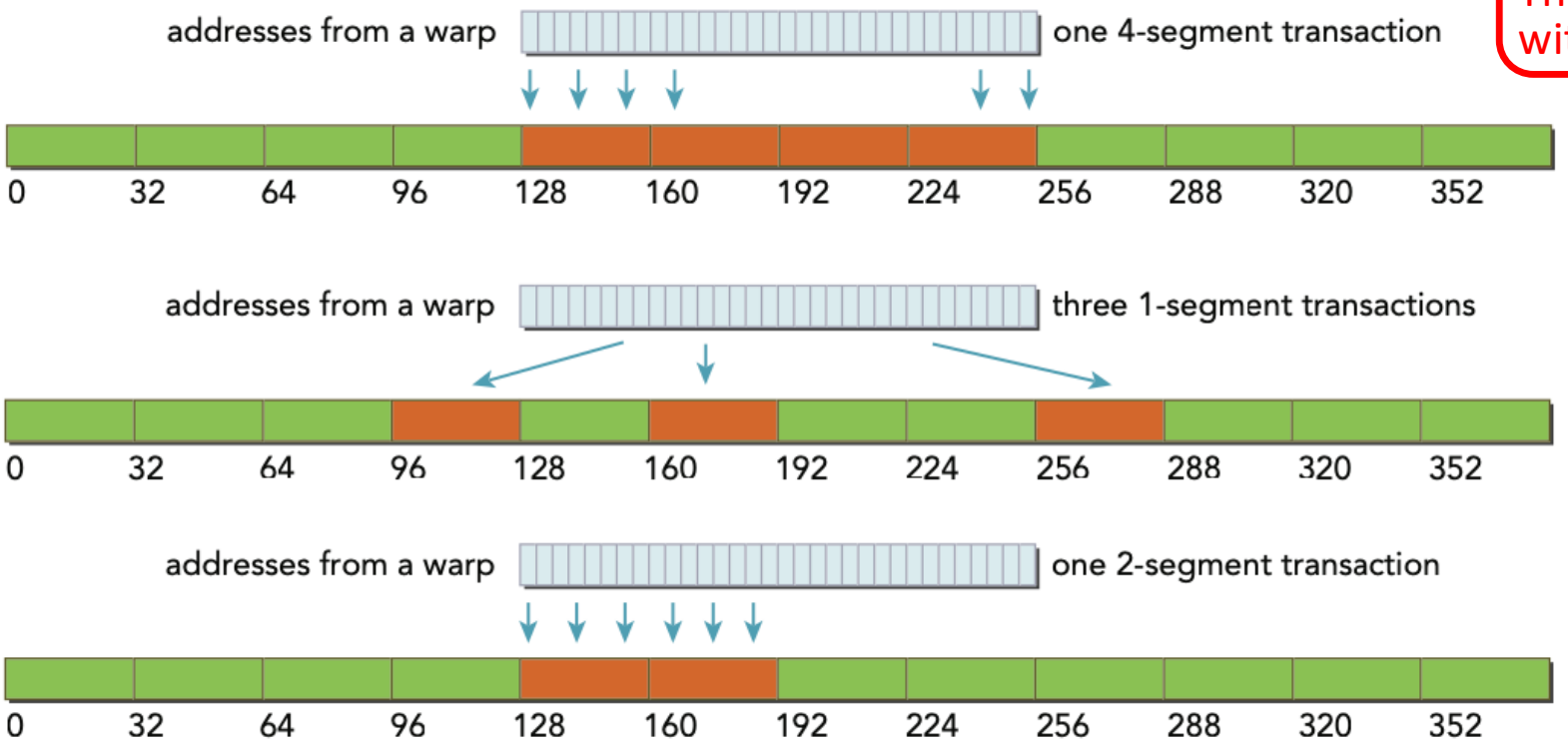


Considering compute capability <5.3
The behavior is similar with newer GPUs

- Worst situation where each thread possibly access to a different cache line requiring up to 32 transactions (3.125% efficiency)

Global memory access patterns

- Examples of write accesses



Considering compute capability <5.3
The behavior is similar with newer GPUs

Best case

Bad case

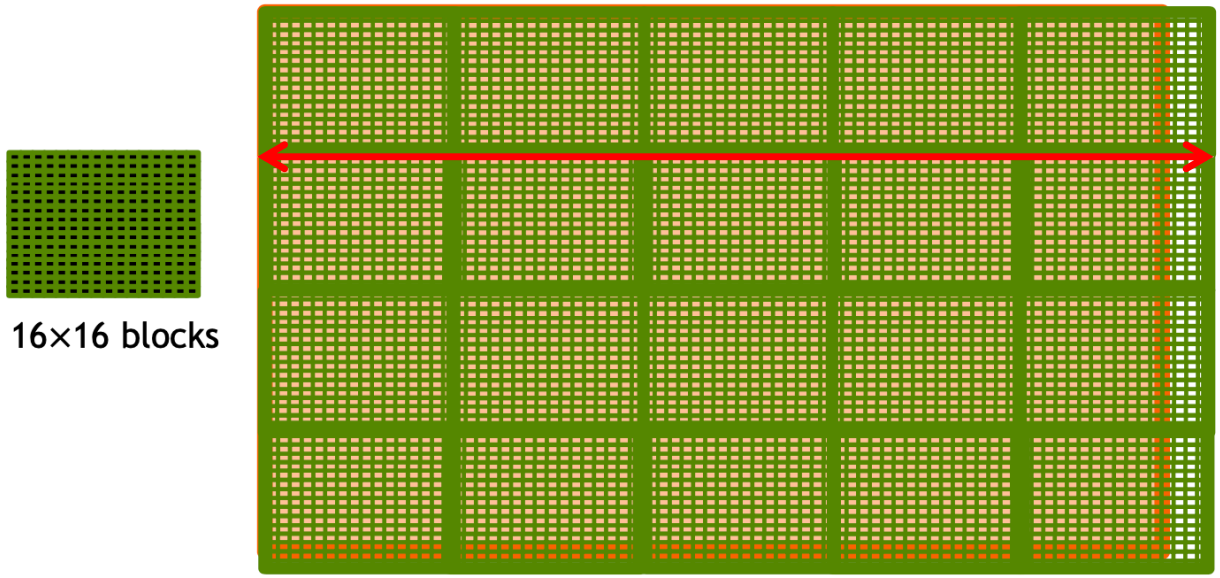
Good case

Global memory access patterns

- Regarding memory alignment
  - Built-in arrays and variables are aligned by the compiler
  - Structures require specific qualifier to force alignment
  - Linearized 2D arrays may cause misaligned accesses when the array width is not multiple of the cache line size
    - Padding can be applied to size the 2D width equal to the grid's one
    - CUDA API supports automatic management through padding

Global memory access patterns

- Padding on linearized 2D arrays:



We have discussed how both grid height and width are rounded up to multiples of the block sizes…

- To avoid misalignments, array width has to be rounded up to the first greater multiple of 32 (padding!)
- Exceeding values on the right are ignored in the computation

Global memory access patterns

- Local memory resides in device memory -> same rules of global memory hold!
  - Data of the warp are organized to be interleaved every 32-byte words
  - Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address
    - E.g., same index in an array variable, same member in a structure variable

An example of how data organization may affect memory efficiency

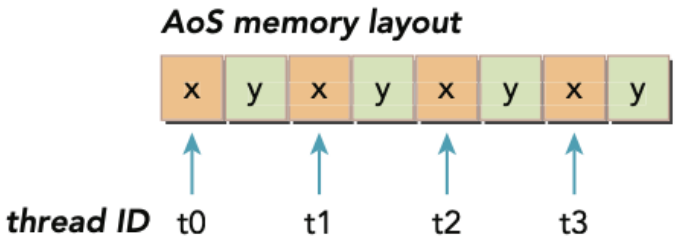
- Which is the better data structure?

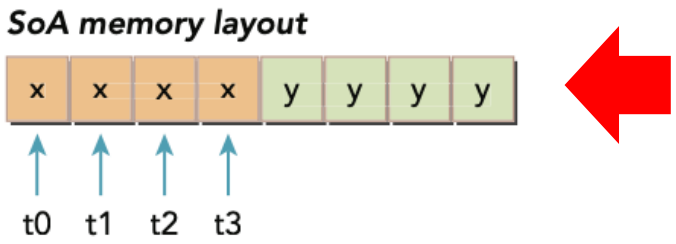
Array of structures

```
typedef struct {
  float x;
  float y;
} innerStruct_t;

innerStruct_t myAoS[N];
```

Structure of arrays

```
typedef struct {
  float x[N];
  float y[N];
} innerArray_t;

innerArray_t mySoA;
```





Analysis of the efficiency of the memory accesses

- The profiler supports the programmer in the analysis of the efficiency of the memory usage in terms of access alignment and coalescence

- Relevant metrics:
  - $\text{Global\_load\_efficiency} = \frac{\text{requested global memory load throughput}}{\text{required global memory load throughput}}$
  - $\text{Global\_store\_efficiency} = \frac{\text{requested global memory store throughput}}{\text{required global memory store throughput}}$

TA will present Nsight Compute profiler!

GPUs and Heterogeneous Systems
(programming models and architectures)

Usage of the shared memory

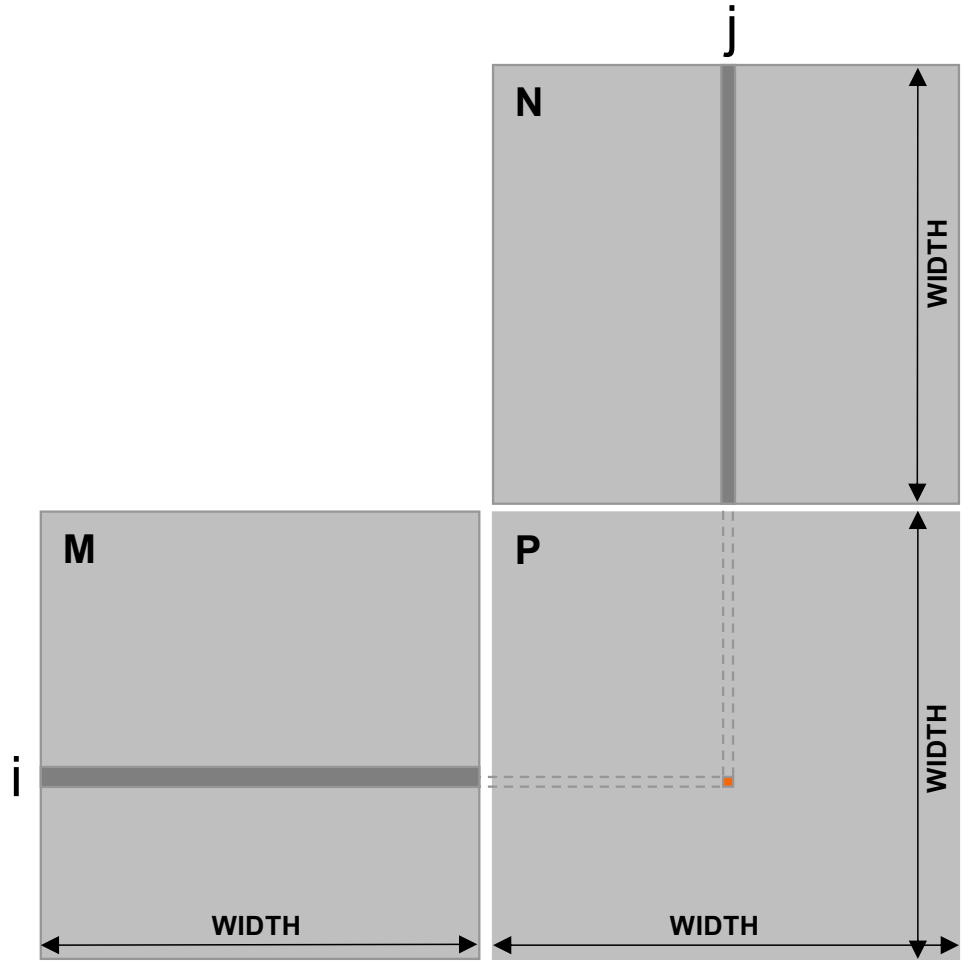
Recap on shared memory usage

- Shared memory can be used for
  - Reducing the traffic w.r.t. the global memory
  - Allowing intra-block thread collaboration

- Typical usage:
  - Parallel load of data from global memory to shared memory
  - Synchronization of block threads
  - Elaboration of block threads on shared memory data
  - Synchronization of block threads
  - Parallel store of data from shared memory to global memory

An example: matrix multiplication

- Each value P[i][j] is computed by means of the dot product of row i of N and column j of M
- Two nested loops scans all the elements in the output matrix P



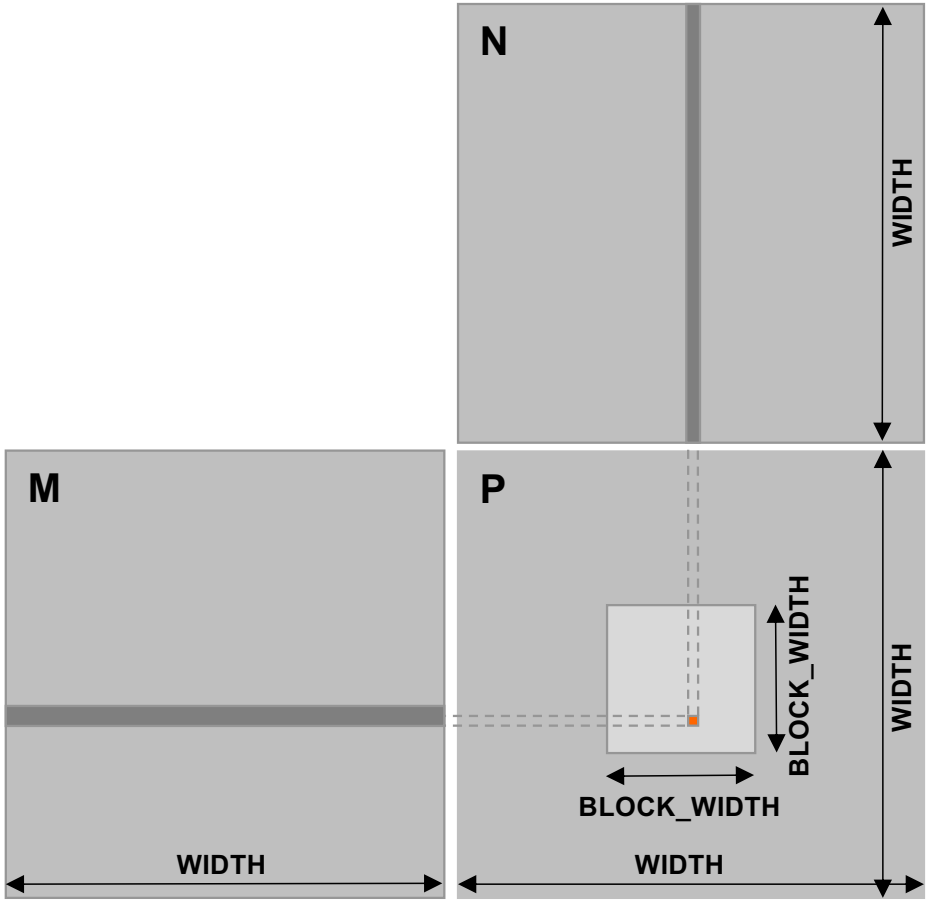
An example: matrix multiplication

```
void matrixmult(int *M, int *N, int *P,
                int numMRows, int numMColumns, int numNColumns) {
  int i, j, k;
  for(i=0; i<numMRows; i++)
    for(j=0; j<numNColumns; j++)
      for(k=0, C[i * numMColumns + j]=0; k<numMColumns; k++)
        P[i * numMColumns + j] +=
              M[i * numMColumns + k] * N[k * numNColumns + j];
}
```

- M, N, P are the pointers to the linearized 2D matrices
- It is assumed that matrices are compatible (numMColumns == numNRows)

Basic matrix multiplication kernel implementation

- Output data decomposition is applied
  - Each thread computes a value of P by means of a dot product



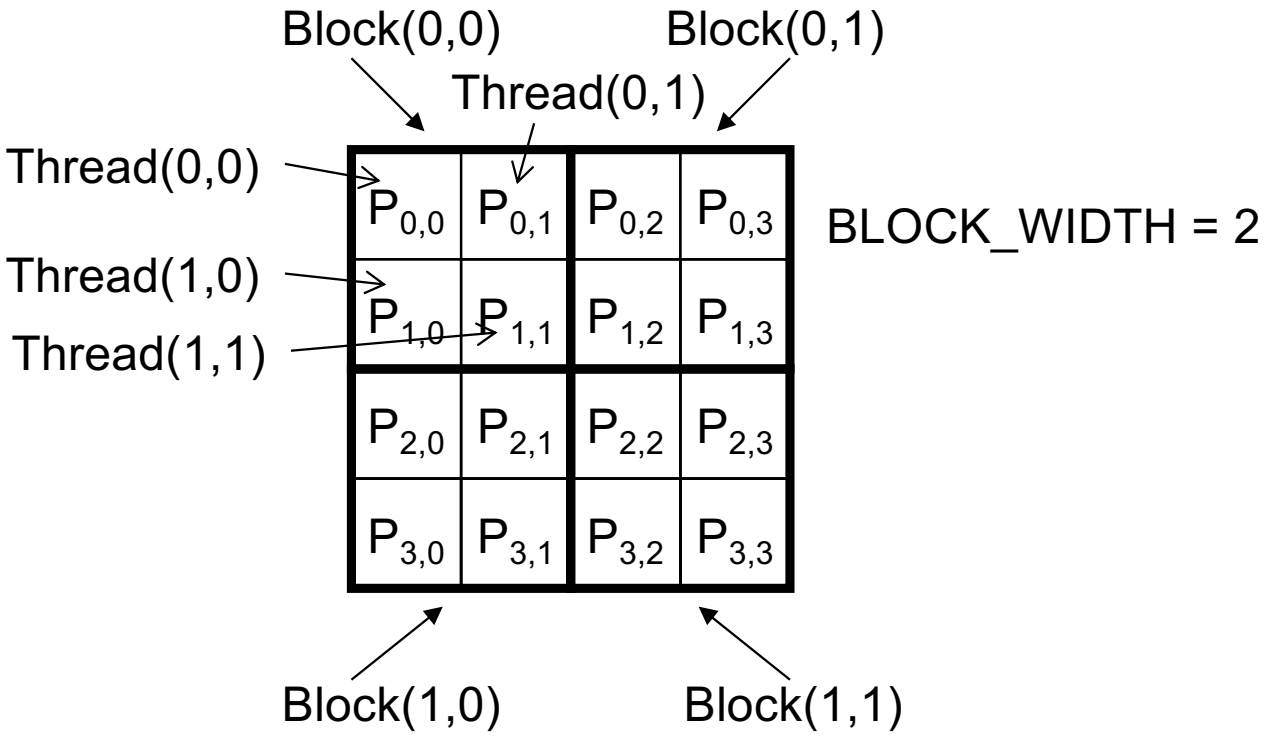
Basic matrix multiplication kernel implementation

```
__global__ void basic_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < numMRows && j < numNColumns){
    int sum = 0;
    for (int k = 0; k < numMColumns; k++)
      sum += M[i * numMColumns + k] * N[k * numNColumns + j];
    P[i * numNColumns + j] = sum;
  }
}
```

Do note: accumulating on `sum` implies a register write; instead, accumulating on P would imply a store in the global memory

Basic matrix multiplication kernel implementation

- Let's execute a toy run
  - Grid organization

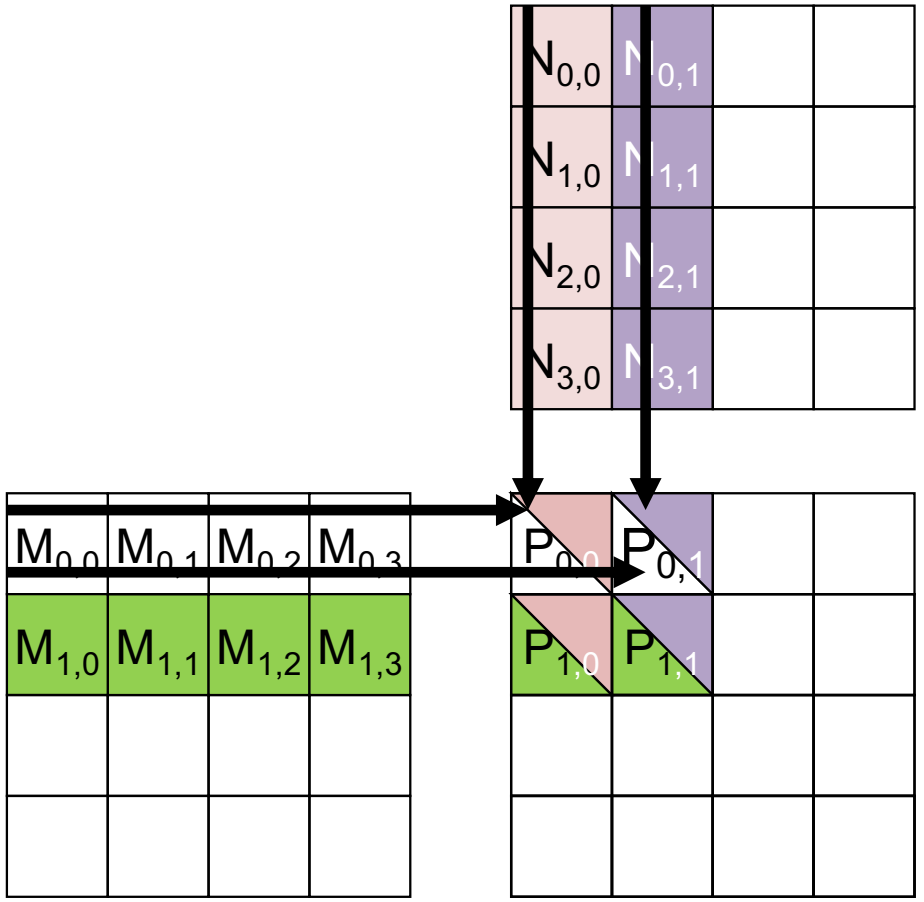


Basic matrix multiplication kernel implementation

- Let's execute a toy run:
  - Calculation of P[0][0] and P[0][1]

- Bad aspects
  - Threads of the same row of the block block concurrently access to the same position of M
  - Each column of N is read a number of times equal to the block height

High non-optimized traffic while reading data from the global memory

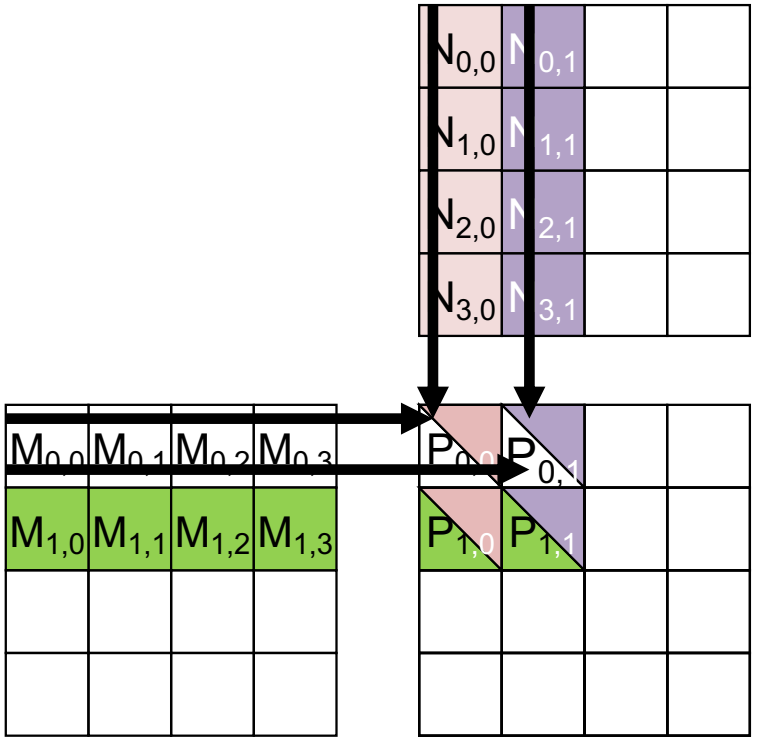


Basic matrix multiplication kernel implementation

- Trace of the memory accesses to better understand the problem:

Access order

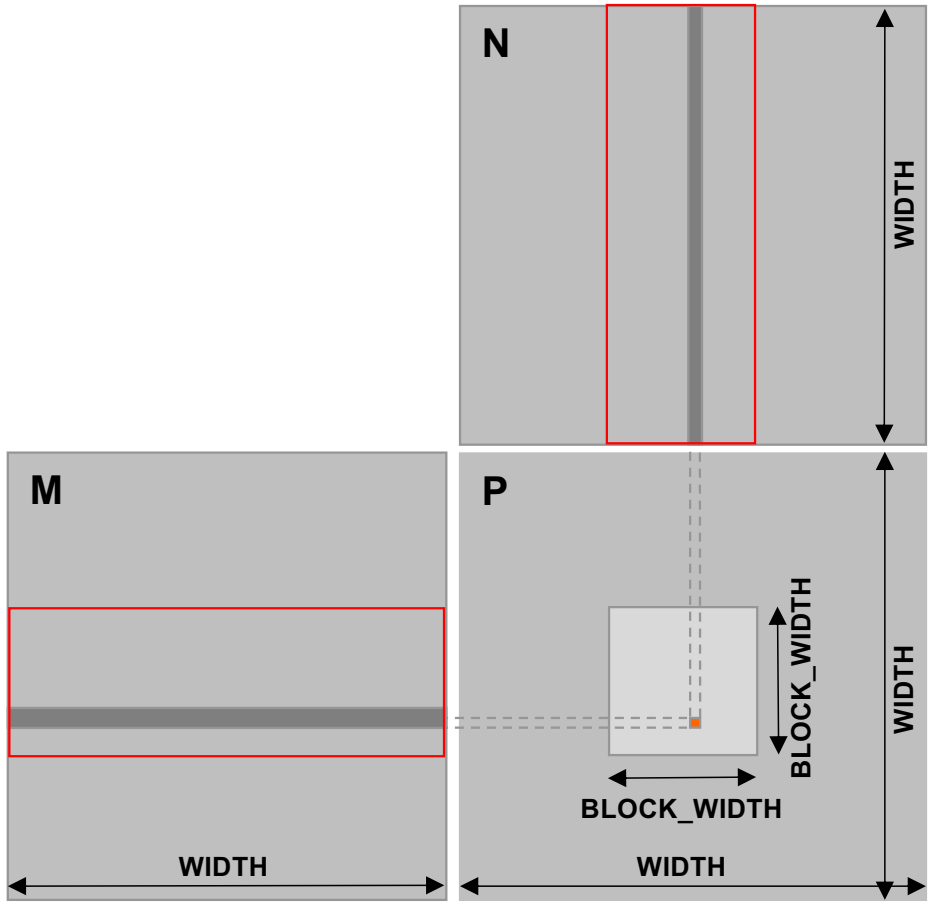
$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$



High non-optimized traffic while reading data from the global memory

Tiled matrix multiplication kernel implementation

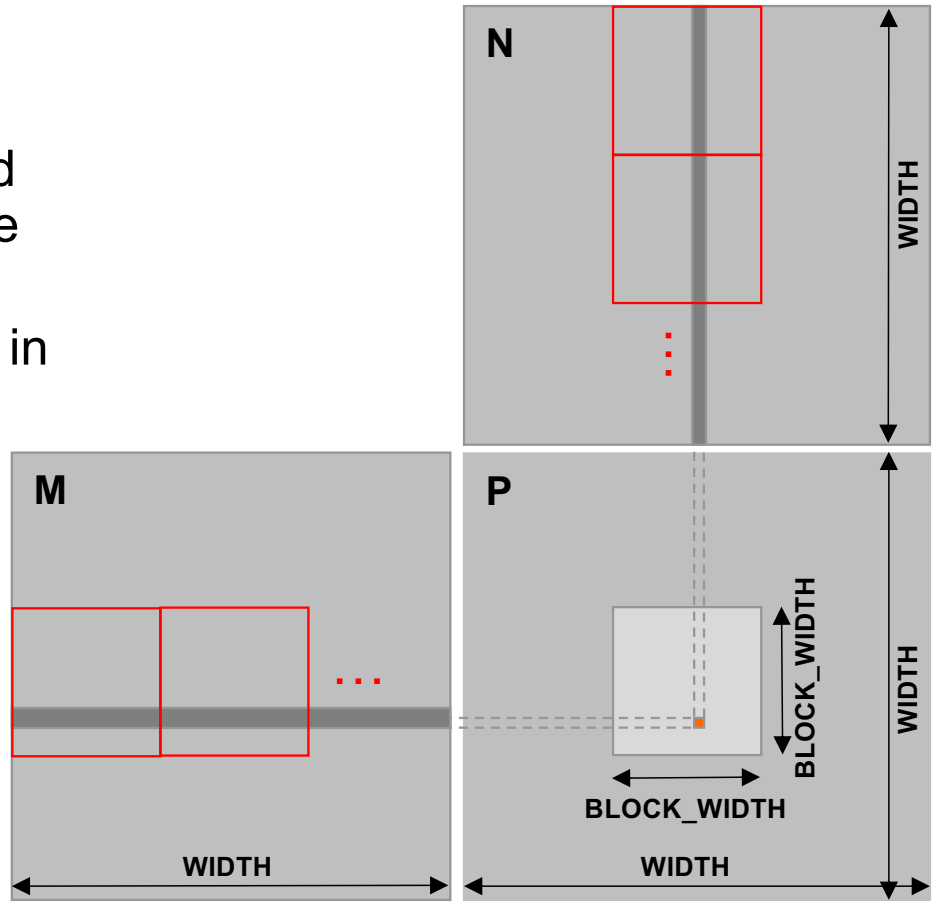
- Data access pattern:
  - Each thread accesses a row of M and a column of N (grey lines)
  - Each block of threads accesses a strip of M and a strip of N (red rectangles)



Tiled matrix multiplication kernel implementation

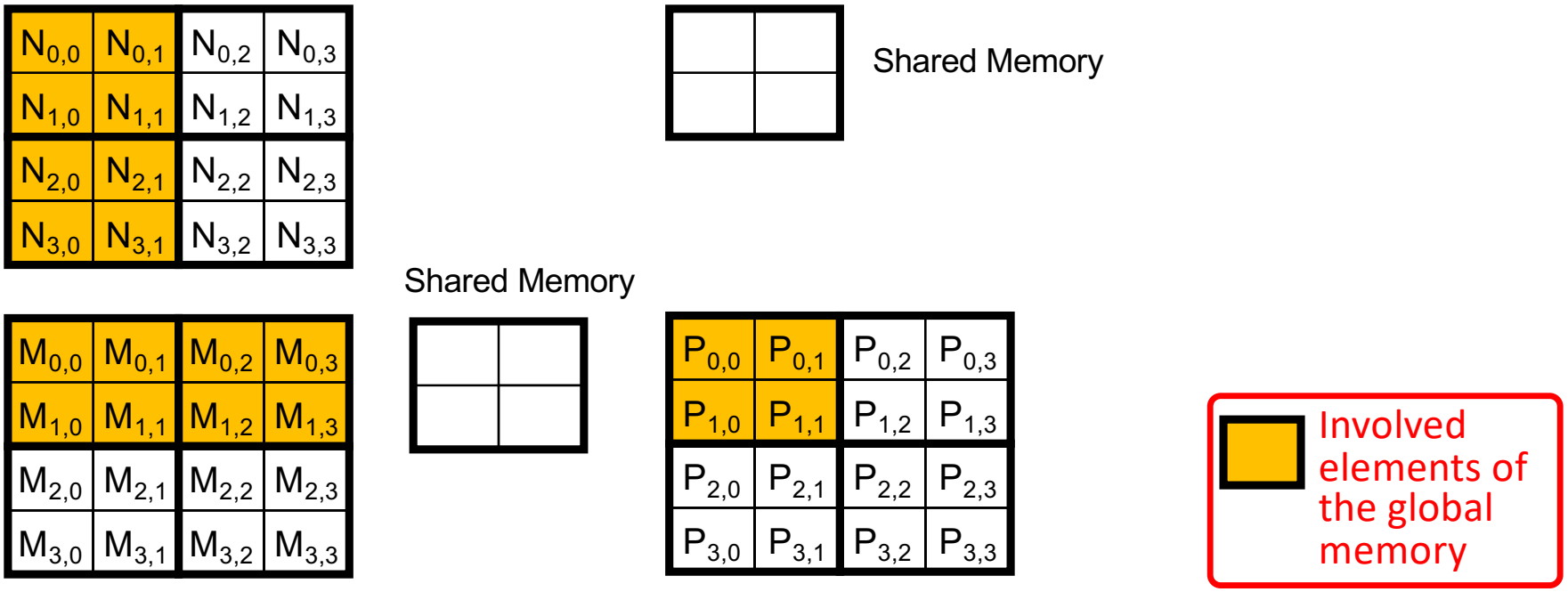
- Break up the execution of each thread into phases
  - So that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
  - The tile is of BLOCK_WIDTH elements in each dimension

- Workflow (iterated):
  - Block threads load data of a tile in the shared memory
  - Each thread computes a part of the dot product using data in the shared memory



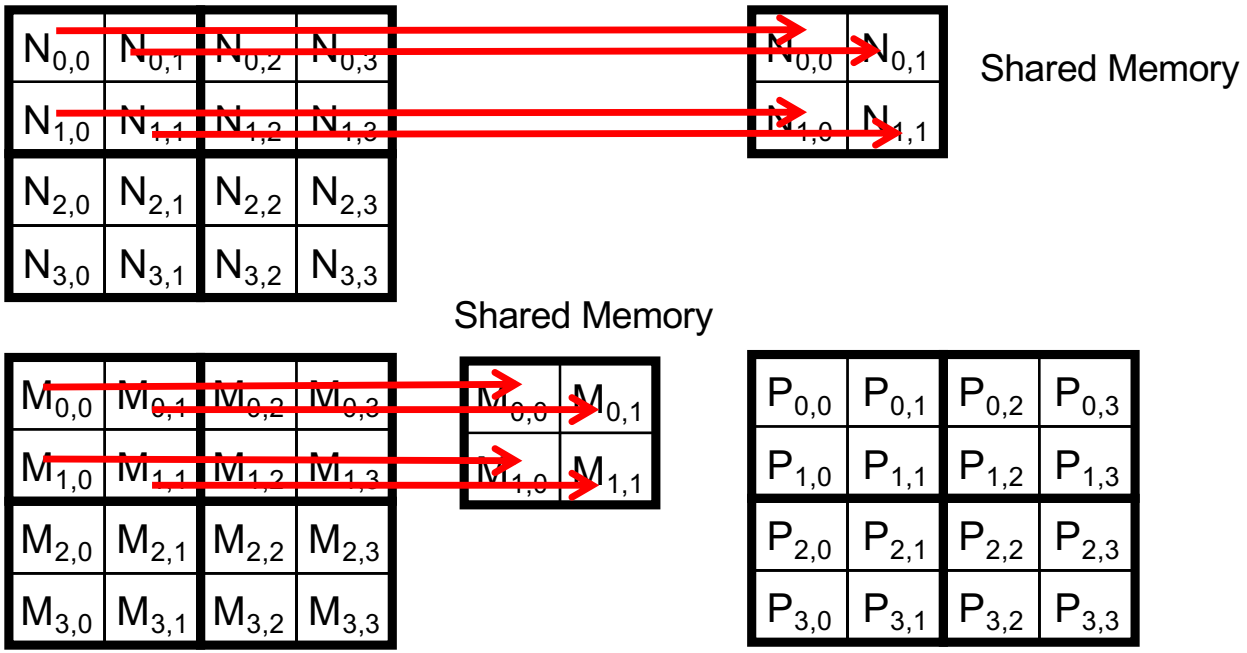
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Global and shared memory organization



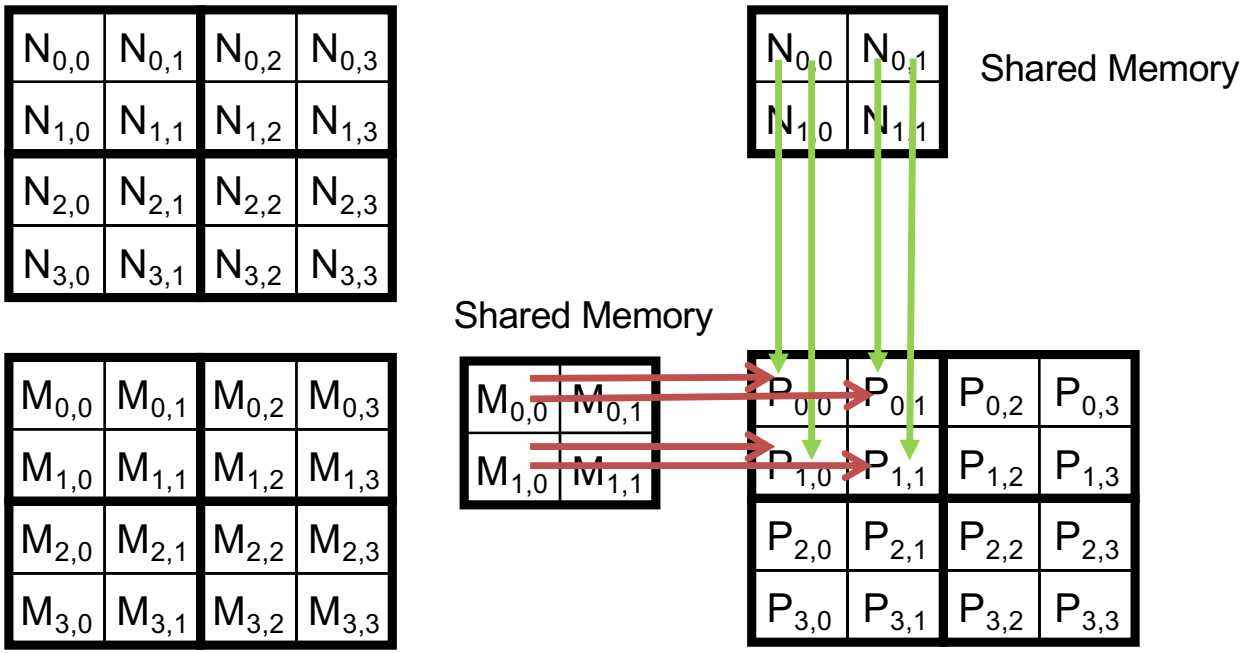
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 0: load for block (0,0)



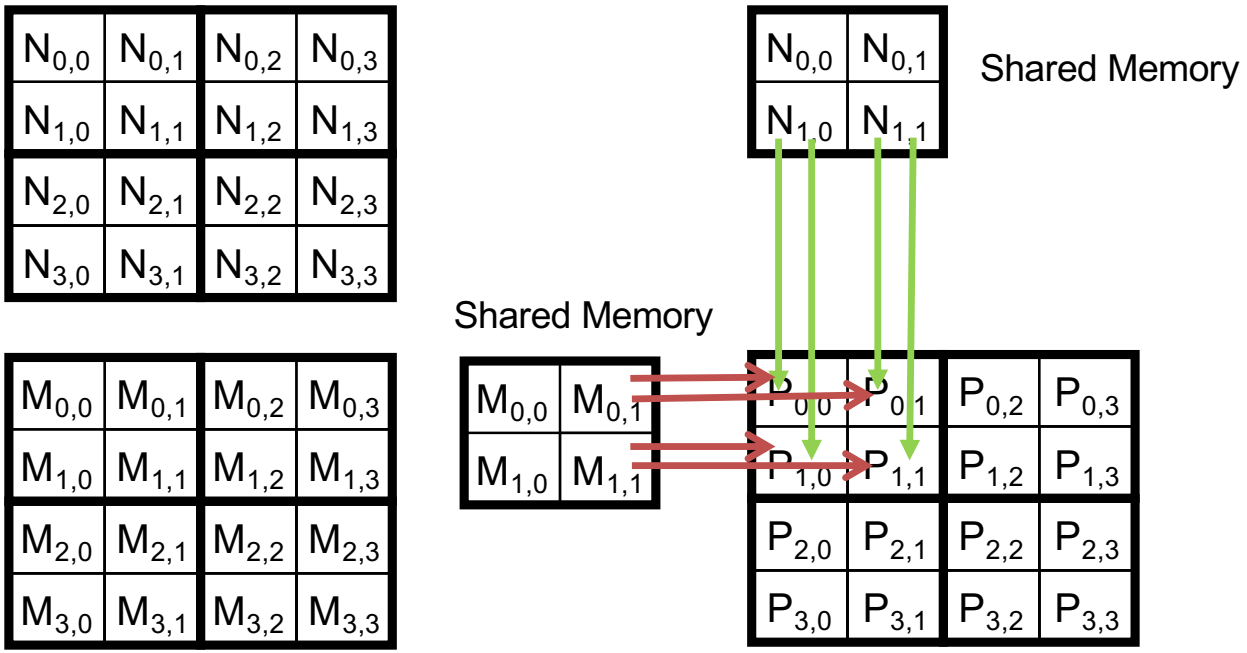
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 0: use for block (0,0) – iteration 0



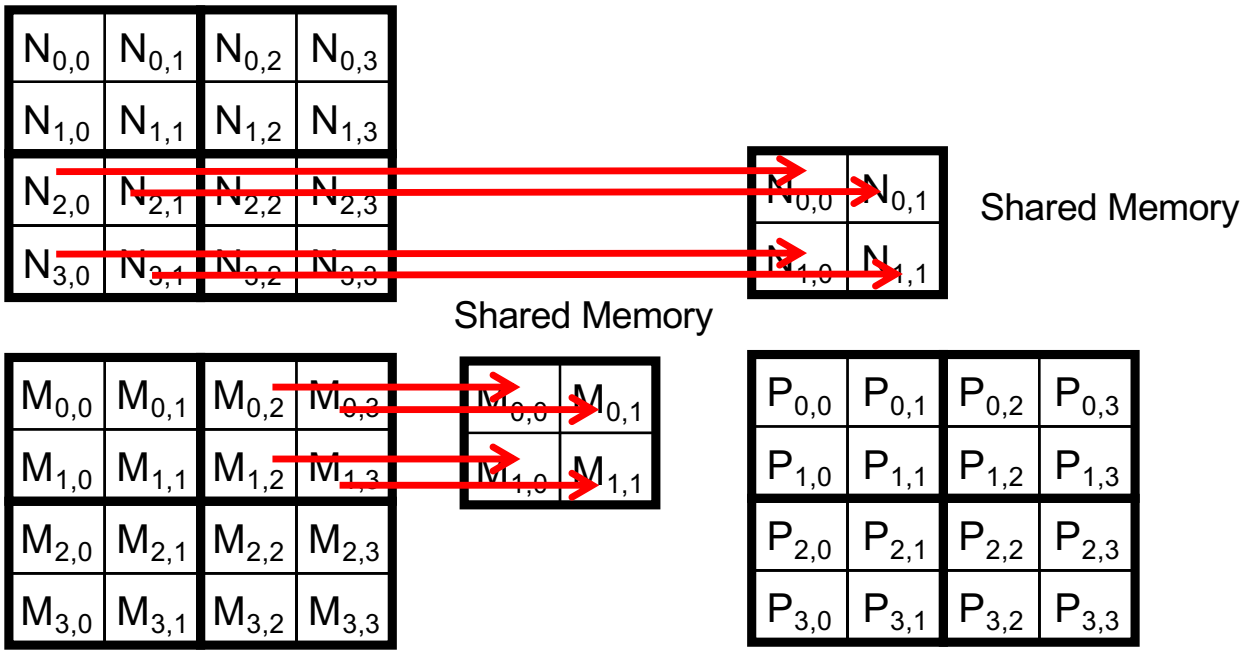
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 0: use for block (0,0) – iteration 1



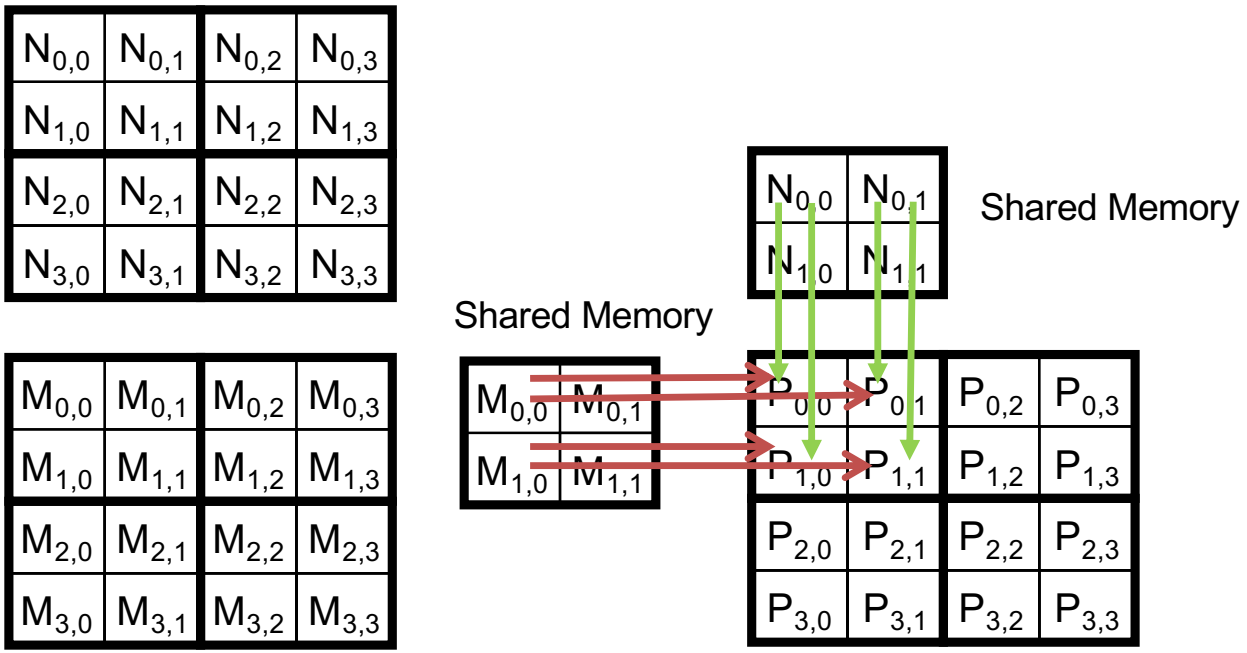
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 1: load for block (0,0)



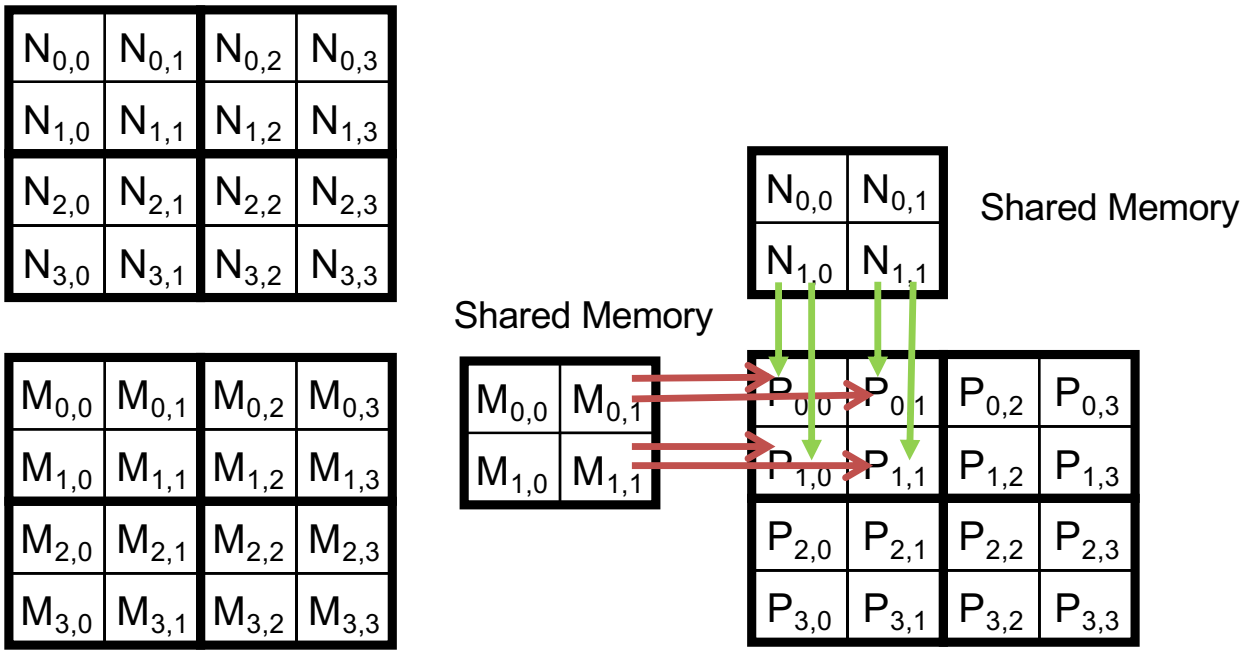
Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 1: use for block (0,0) (iteration 0)



Tiled matrix multiplication kernel implementation

- Let's execute a toy run
  - Phase 1: use for block (0,0) (iteration 1)



Tiled matrix multiplication kernel implementation

Threads with consecutive ids read from adjacent global memory addresses on both the two input matrices

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}*Nds_{0,0}$ + $Mds_{0,1}*Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}*Nds_{0,1}$ + $Mds_{0,1}*Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}*Nds_{0,0}$ + $Mds_{1,1}*Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}*Nds_{0,1}$ + $Mds_{1,1}*Nds_{1,1}$

time →

Tiled matrix multiplication kernel implementation

- The shared memory allows each value to be accessed by multiple threads concurrently
- No necessity to perform coalesced accesses

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Tiled matrix multiplication kernel implementation

- The matrixes in the shared memory are re-used multiple times to save different tiles

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * TILE_WIDTH + ty, j = bx * TILE_WIDTH + tx;
  int Pvalue = 0;

  /*...*/
}
```

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * TILE_WIDTH + ty, j = bx * TILE_WIDTH + tx;
  int Pvalue = 0;

  /*...*/
}
```

- Shared memory declaration: a matrix storing a tile per each input matrix
- The tile has the same size of the thread block

For the sake of simplicity, the height and the width of the matrices are divisible by the tile width

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * TILE_WIDTH + ty, j = bx * TILE_WIDTH + tx;
  int Pvalue = 0;

  /*...*/
}
```

bx, by: block coordinates
tx, ty: threads coordinate in the block
i, j: thread mapping with the output matrix

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();
    /*...*/
  }
  /*...*/
}
```

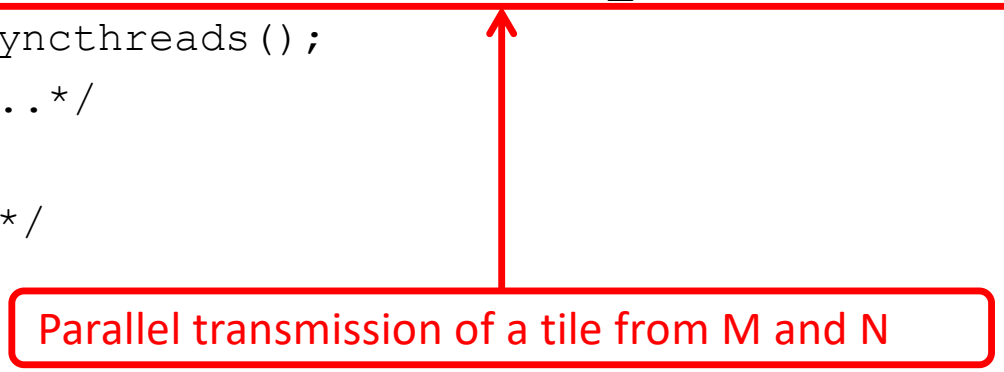
Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();
    /*...*/
  }
  /*...*/
}
```

Iteration on all the tiles

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();
    /*...*/
  }
  /*...*/
}
```



Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();
    /*...*/
  }
  /*...*/
}
```

Barrier to synchronize threads in the same block

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    /*...*/
    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numBColumns + j] = Pvalue;
}
```

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    /*...*/
    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numBColumns + j] = Pvalue;
}
```

Each thread computes its partial dot product

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    /*...*/
    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numBColumns + j] = Pvalue;
}
```

Threads are synchronized before starting the subsequent loop iteration (i.e., read new data from the global memory

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  /*...*/
  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    /*...*/
    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numBColumns + j] = Pvalue;
}
```

After the loop, each thread write its own dot product result in the global memory

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, int *P,
                                 int numMRows, int numMColumns, int numNColumns) {
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numNColumns + j] = Pvalue;
}
```

The overall kernel code

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, i
                                 int numMRows, int num
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numNColumns + j] = Pvalue;
}
```

If the height and the width of the matrices are not divisible by the tile width

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, i
                                 int numMRows, int num
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_
    __syncthreads();
  }
  P[i * numNColumns + j] = Pvalue;
}
```

If the height and the width of the matrices are not divisible by the tile width

The number of iterations has to be adjusted:
`m < (numAColumns - 1) / BLOCK_WIDTH + 1`

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, i
                                 int numMRows, int num
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numNColumns +
}
```

If the height and the width of the matrices are not divisible by the tile width

Check for overflows in accesses to M:

```
if (i < numMRows && m * BLOCK_WIDTH + tx < numMColumns)
  ds_M[ty][tx] = M[i * numMColumns + m * BLOCK_WIDTH + tx];
else
  ds_M[ty][tx] = 0;
```

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, i
                                 int numMRows, int num
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numNColumns +
}
```

If the height and the width of the matrices are not divisible by the tile width

Check for overflows in accesses to N:

```
if (j < numNColumns && m * BLOCK_WIDTH + ty < numMColumns)
  ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
else
  ds_N[ty][tx] = 0;
```

Tiled matrix multiplication kernel implementation

```
__global__ void tiled_matrixmult(int *M, int *N, i
                                 int numMRows, int num
  __shared__ int ds_M[BLOCK_WIDTH][BLOCK_WIDTH];
  __shared__ int ds_N[BLOCK_WIDTH][BLOCK_WIDTH];
  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y,
      i = by * BLOCK_WIDTH + ty, j = bx * BLOCK_WIDTH + tx;
  int Pvalue = 0;

  for (int m = 0; m < numMColumns / BLOCK_WIDTH; ++m) {
    ds_M[ty][tx] = M[i * numNColumns + m * BLOCK_WIDTH + tx];
    ds_N[ty][tx] = N[(m * BLOCK_WIDTH + ty) * numNColumns + j];
    __syncthreads();

    for (int k = 0; k < BLOCK_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __syncthreads();
  }
  P[i * numNColumns + j] = Pvalue;
}
```

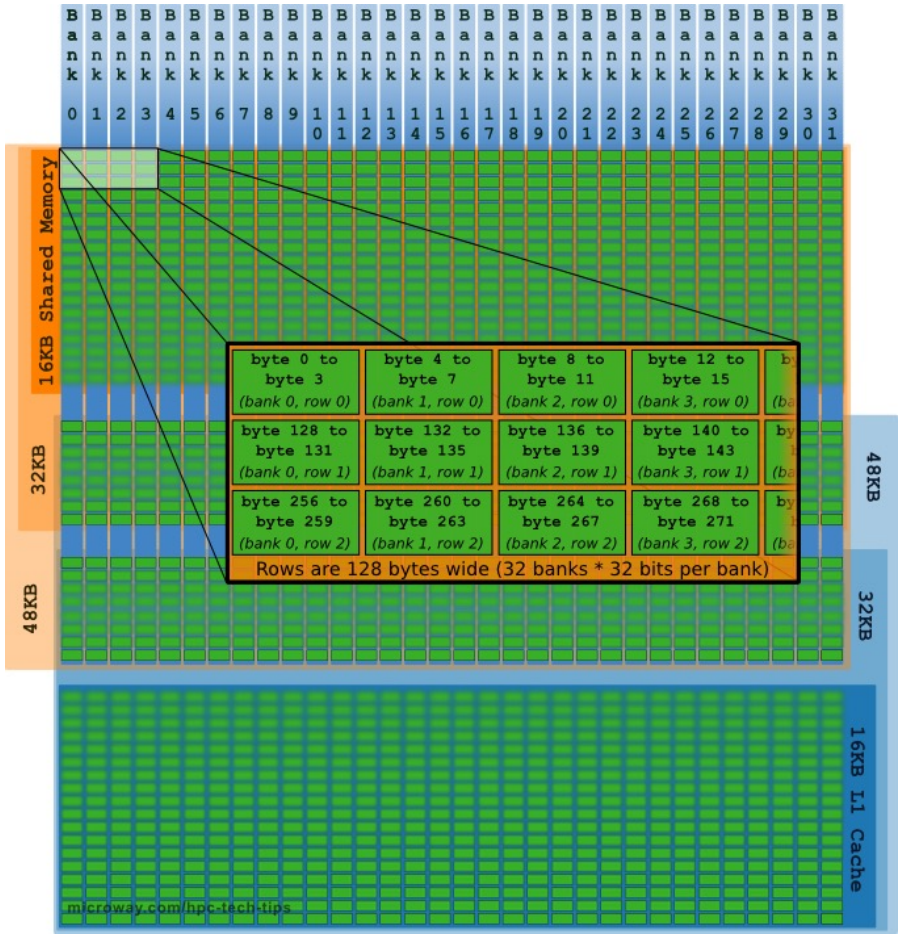
If the height and the width of the matrices are not divisible by the tile width

Check for overflows in accesses to P:

```
if (i < numMRows && j < numNColumns)
  P[i * numNColumns + j] = Pvalue;
```

Internal organization of the shared memory

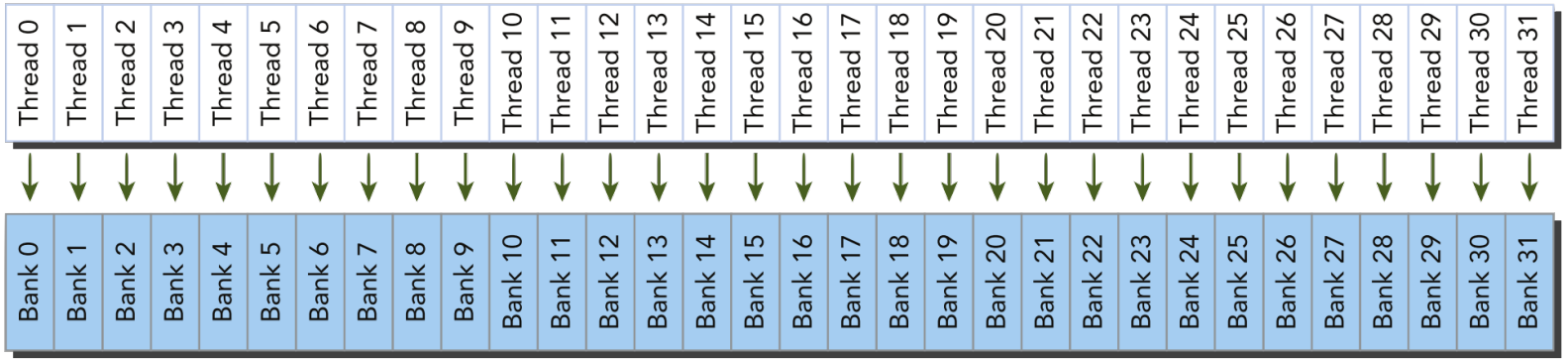
- The shared memory is organized in 32 banks (vertically aligned)
- Bank width is slightly different for compute capabilities:
  - 3.X: 64-bit words (accessible at 32bit granularity)
  - 5.X and later: 32-bit words
- Banks can be accessed concurrently at the word granularity



Access patterns to shared memory

- **Parallel access**: multiple addresses accessed across multiple banks

Request served with a single transaction!



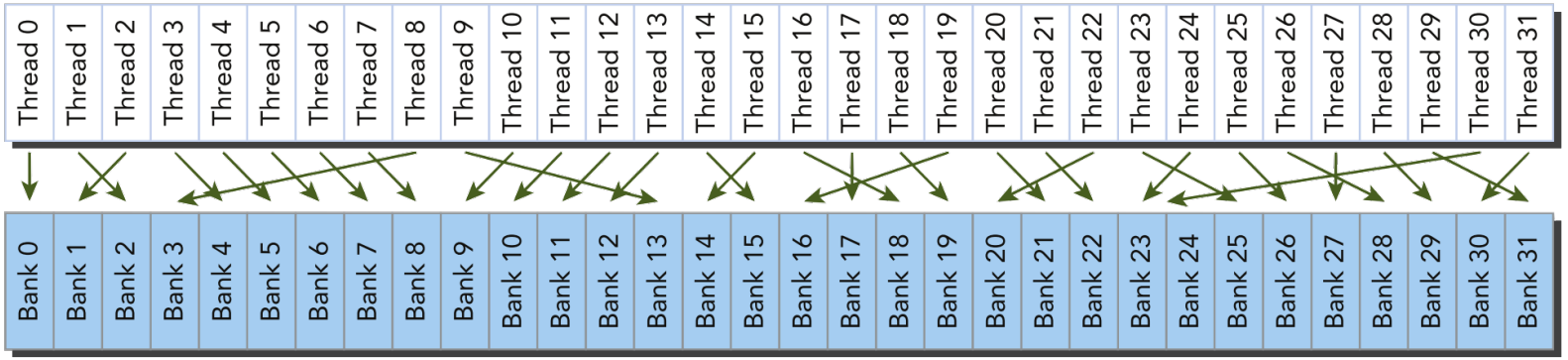
- Example of code:

```
float f = sh_array[threadIdx.x];
```

Access patterns to shared memory

- **Parallel access**: multiple addresses accessed across multiple banks

Request served with a single transaction!



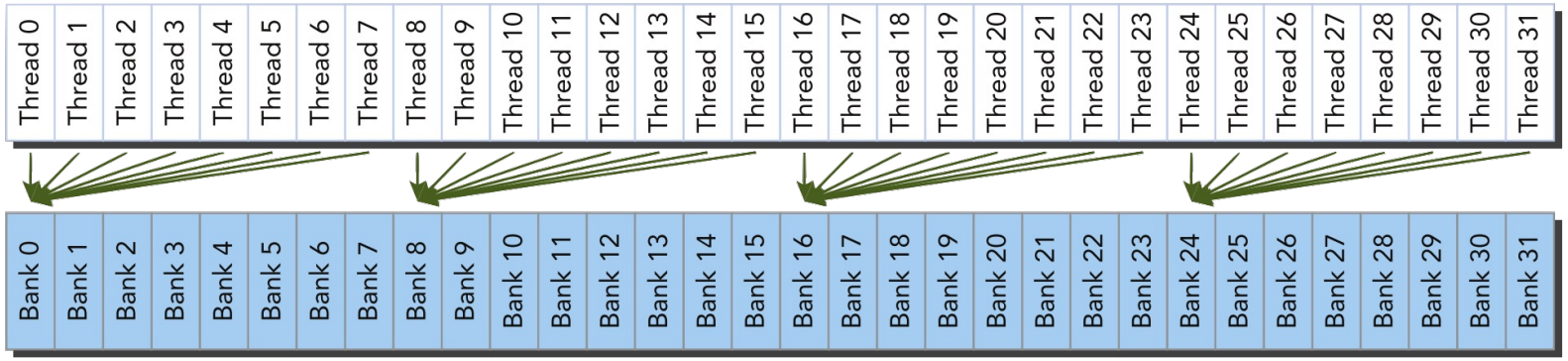
It doesn't matter if access pattern is irregular!

Access patterns to shared memory

- **Serial access**: multiple addresses accessed within the same block

Bank access conflicts!

Request served with several transactions!



- Example of code:

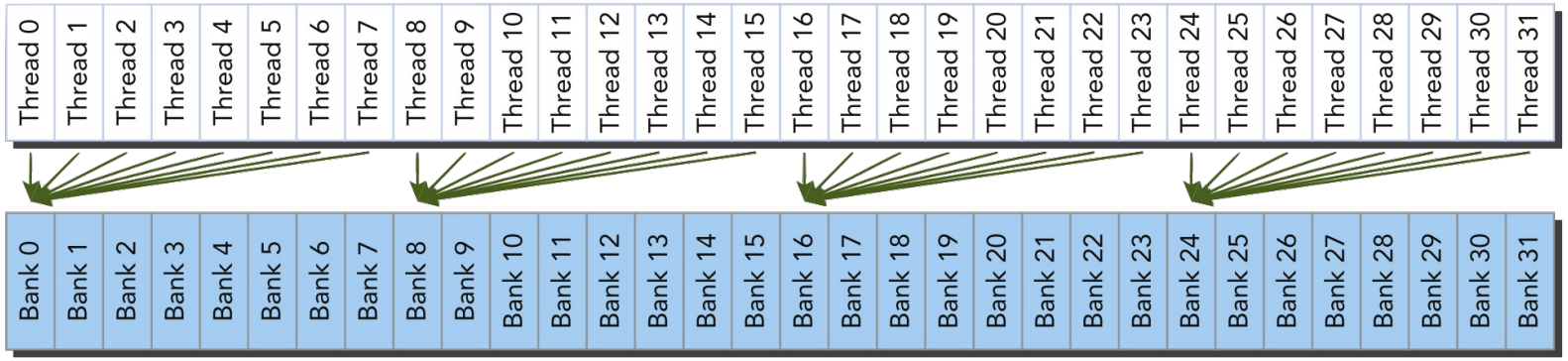
```
float f = sh_array[threadIdx.x*2];
```

An odd stride would not cause any bank conflict!

Access patterns to shared memory

- **Broadcast access**: a single address read in a single bank addresses accessed within the same block

Request served with a single transaction!

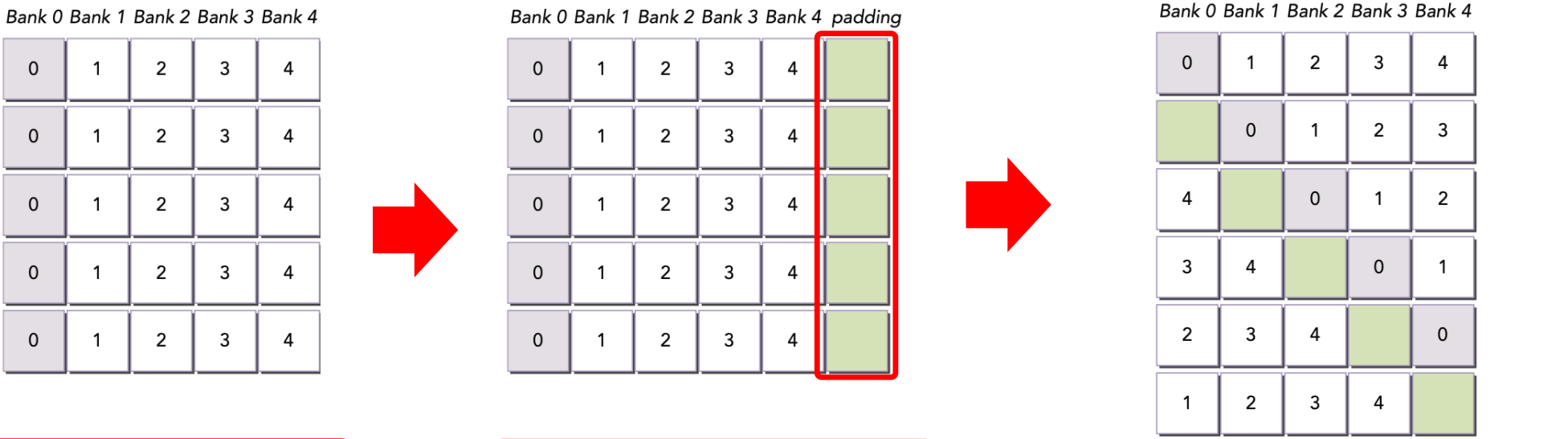


- Example of code:

```
float f = sh_array[0];
float f = sh_array[threadIdx.x*2%32];
```

Padding to solve bank conflicts

- Spare columns may be added on the right side of the matrix stored in the shared memory to avoid bank conflicts



Original matrix. All threads accesses to bank 0 causing a conflict

Modify matrix declaration by adding a padding column on the right

The new organization of the data in the shared memory does not cause any conflict

GPUs and Heterogeneous Systems
(programming models and architectures)

Atomic and warp shuffle instructions

Atomic instructions

- An **atomic instruction** performs a mathematical operation as a **single uninterruptable operation** without any interference from other threads
- Atomic instructions allow to perform write operations in the memory without any race condition
- Atomic instructions can be used on both global and shared memory

- Example:

```
__global__ void foo(int* a){
    atomicAdd(a, 1);
}
```

Atomic instructions

- An **atomic instruction** performs a mathematical operation as a **single uninterruptable operation** without any interference from other threads
- Atomic instructions allow to perform write operations in the memory without any race condition
- Atomic instructions can be used on both global and shared memory

- Example:

```
__global__ void foo(int* a){
    atomicAdd(a, 1);
}
```

Value to be added

Memory location to be atomically modified

At the end of the kernel execution, a will contain the count of the number of running threads

Atomic instructions

- Supported atomic instructions:

OPERATION	FUNCTION	SUPPORTED TYPES
Addition	`atomicAdd`	int, unsigned int, unsigned long long int, float
Subtraction	`atomicSub`	int, unsigned int
Unconditional Swap	`atomicExch`	int, unsigned int, unsigned long long int, float
Minimum	`atomicMin`	int, unsigned int, unsigned long long int
Maximum	`atomicMax`	int, unsigned int, unsigned long long int
Increment	`atomicInc`	unsigned int
Decrement	`atomicDec`	unsigned int
Compare-And-Swap	`atomicCAS`	int, unsigned int, unsigned long long int
And	`atomicAnd`	int, unsigned int, unsigned long long int
Or	`atomicOr`	int, unsigned int, unsigned long long int
Xor	`atomicXor`	int, unsigned int, unsigned long long int

Warp shuffle instructions

- A shuffle instruction allows threads in the same warp to exchange data stored in registers without using global or shared memory

- Example:

```
__global__ void test_shfl_broadcast(int *d_out, int srcThread) {
  int value = threadIdx.x;
  value = __shfl(value, srcThread, 32);
  d_out[threadIdx.x] = value;
}
```

Warp shuffle instructions

- A shuffle instruction allows threads in the same warp to exchange data stored in registers without using global or shared memory

- Example:

```
__global__ void test_shfl_broadcast(int *d_out, int srcThread) {
  int value = threadIdx.x;
  value = __shfl(value, srcThread, 32);
  d_out[threadIdx.x] = value;
}
```

The entire warp is involved in the data sharing

Variable “value” in each thread of the warp will contain the “threadIdx.x” value of thread number “srcThread”

Value exchanged by each thread

Source thread from which to take the value to be shared

References

- Slides mainly based on:
  - W.-m. W. Hwu , D. B. Kirk, I. El Hajj, **Programming Massively Parallel Processors: A Hands-on Approach**, **Chapter 5**
  - J. Chen, M. Grossman, T. McKercher, **Professional Cuda C Programming**, **Chapters 4 and 5**