GPUs and Heterogeneous Systems
(programming models and architectures)

More on CUDA programming model and CUDA execution model

Data decomposition: mapping threads to multidimensional data

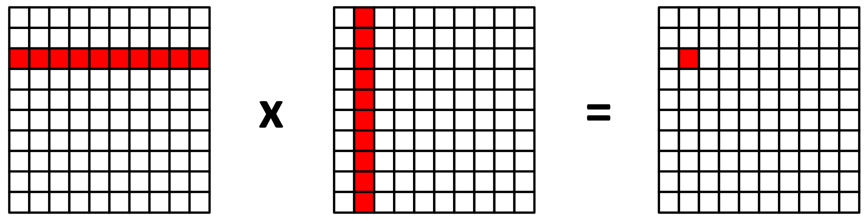
- Functions accelerated on GPU are massively data parallel
  - How to map data on threads?

- **Data decomposition** is adopted to define the kernel function:
  - The overall processed data are divided in N single data items
  - The thread function is defined to elaborate a single data item

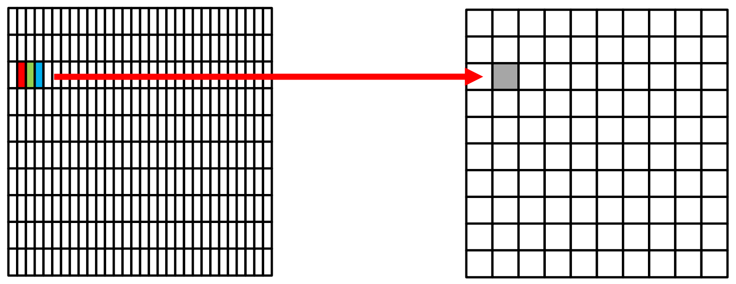
- Two main aspects need to be considered:
  - Data decomposition can be applied on inputs or outputs
  - Threads have to be organized in a grid

Output data decomposition

- **Output data decomposition**: the thread is mapped on the single output data item
  - It can be applied on any **one-to-one** or **many-to-one function**
- Examples:
  - Matrix multiplication

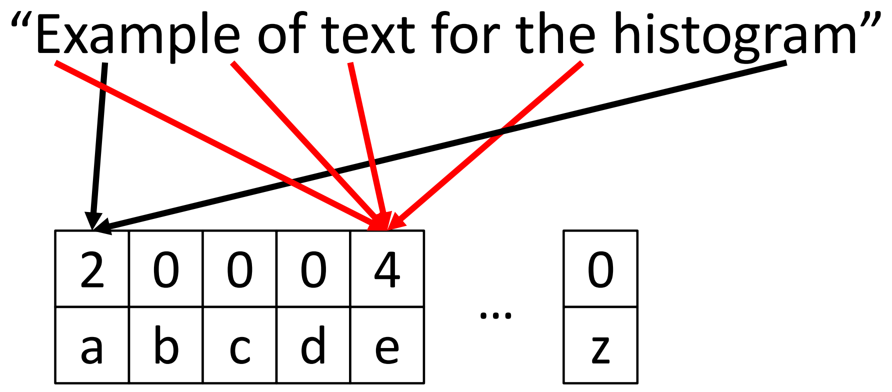


  - Image RGB to gray conversion



Input data decomposition

- **Input data decomposition**: the thread is mapped on the single input data item
  - It can be applied on any **one-to-many functions**
- Examples:
  - Frequency count of each letter in a text (histogram)



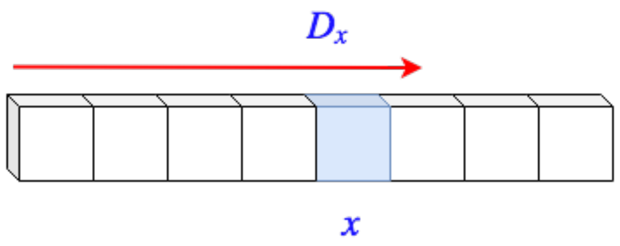
At present we have not studied yet all mechanisms necessary to solve this problem in CUDA

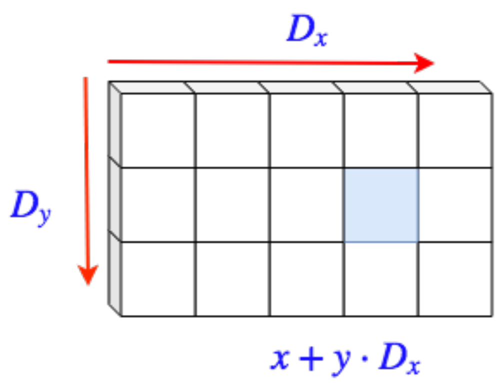
How to select the grid dimensionality

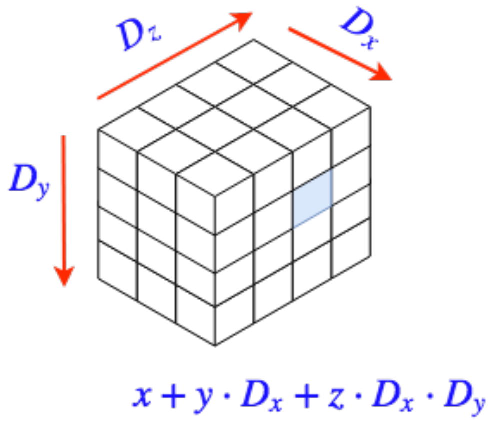
- The thread grid can be defined with up to 3 dimensions
- Grid dimensionality is selected based on the problem dimensionality
  - 1D problems
    - Elaborations on vectors
  - 2D problems
    - Elaborations on matrices, images, …
  - 3D problems
    - Elaborations on data volumes, …
  - Problems with higher dimensionality have to be simplified to a problem with maximum 3 dimensions

How to select the grid dimensionality

- Even if the grid can be defined with up to 3 dimensions, the grid is internally linearized with a row major layout







- This slide simplifies the discussion by not mentioning blocks
- Linearization is applied 1) per blocks and 2) per block threads

Data linearization

- Since C/CUDA supports only 1D dynamic memory allocation, multidimensional data are also linearized in the device memory

```
#define NROWS 100
#define NCOLS 100

__global__ void foo(int* ma, int x){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  ma[j*NCOLS+i] = ma[j*NCOLS+i] * x;
}

int main(){
  int h_ma[NROWS][NCOLS];
  int *d_ma;
  cudaMalloc(&d_ma, sizeof(int)*(NCOLS*NROWS));
  /* ... */
  foo<<<...>>>(d_ma, 2);
}
```

Data linearization

- Since C/CUDA supports only 1D dynamic memory allocation, multidimensional data are also linearized in the device memory

```
#define NROWS 100
#define NCOLS 100

__global__ void foo(int* ma, int x){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  ma[j*NCOLS+i] = ma[j*NCOLS+i] * x;
}

int main(){
  int h_ma[NROWS][NCOLS];
  int *d_ma;
  cudaMalloc(&d_ma, sizeof(int)*(NCOLS*NROWS));
  /* ... */
  foo<<<...>>>(d_ma, 2);
}
```

Linearized access

1D memory allocation

Data linearization

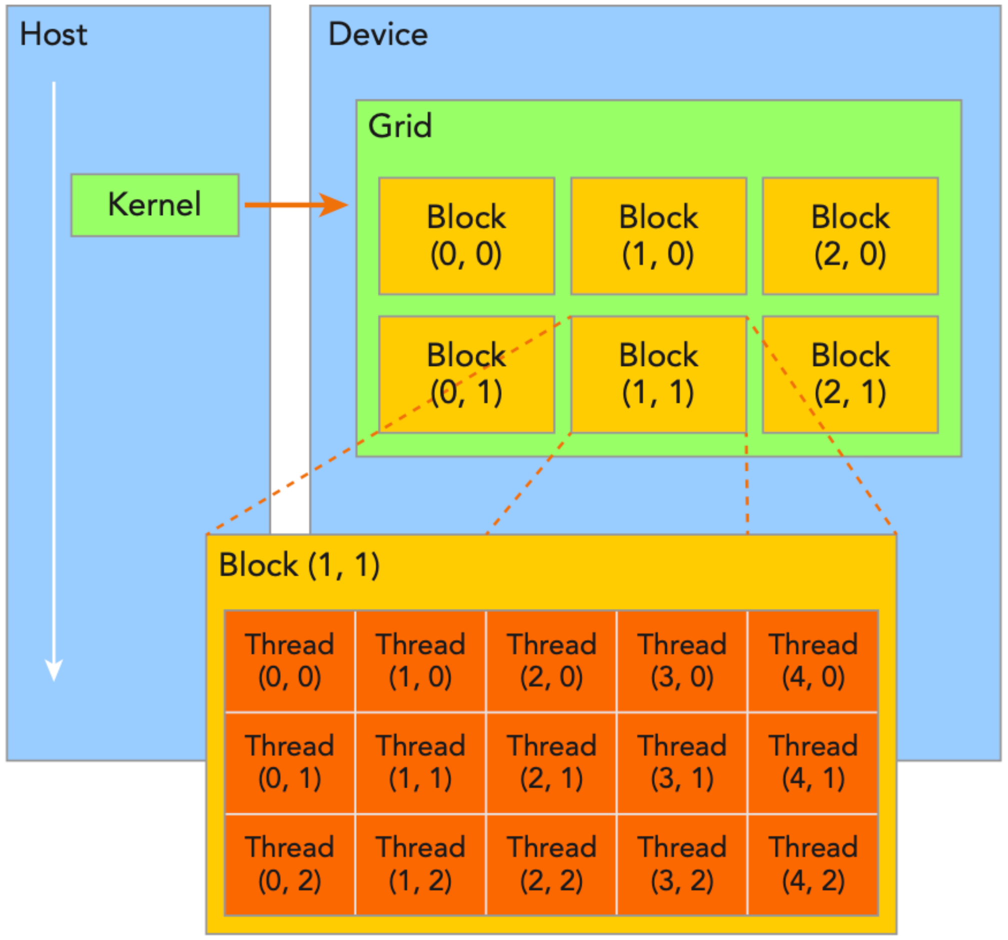
- Since C/CUDA supports only 1D dynamic memory allocation, multidimensional data are also linearized in the device memory

```
#define NROWS 100
#define NCOLS 100

__global__ void foo(int* ma, int x){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  ma[j*NCOLS+i] = ma[j*NCOLS+i] * x;
}

int main(){
  int h_ma[NROWS][NCOLS];
  int *d_ma;
  cudaMalloc(&d_ma, sizeof(int)*(NCOLS*NROWS));
  /* ... */
  foo<<<...>>>(d_ma, 2);
}
```

- Linearization is here applied by means of a row-major layout
- Sometimes a column-major layout may be used (e.g., in libraries derived by Fortran code)

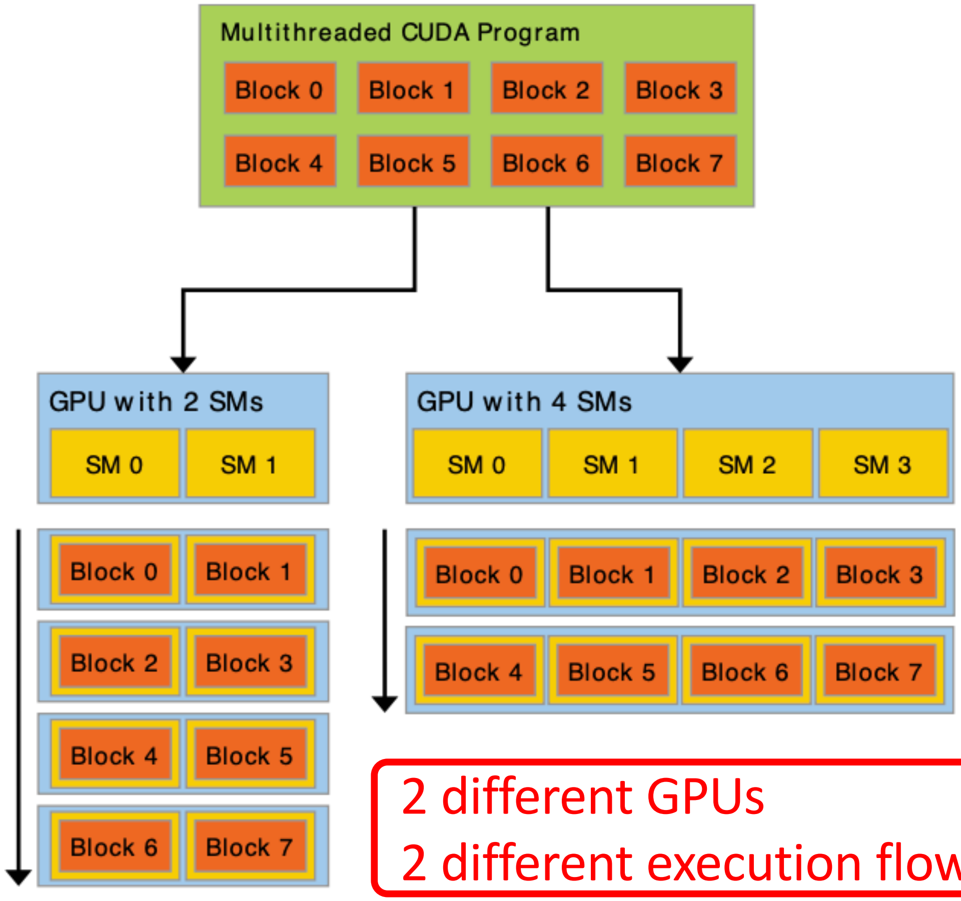
CUDA thread hierarchy

- Threads are hierarchically organized in a grid
  - Threads are grouped in **blocks**
  - Blocks are organized in a **grid**
- Both blocks and grids are N-dimensional (up to 3) arrays



Execution model – block dispatching

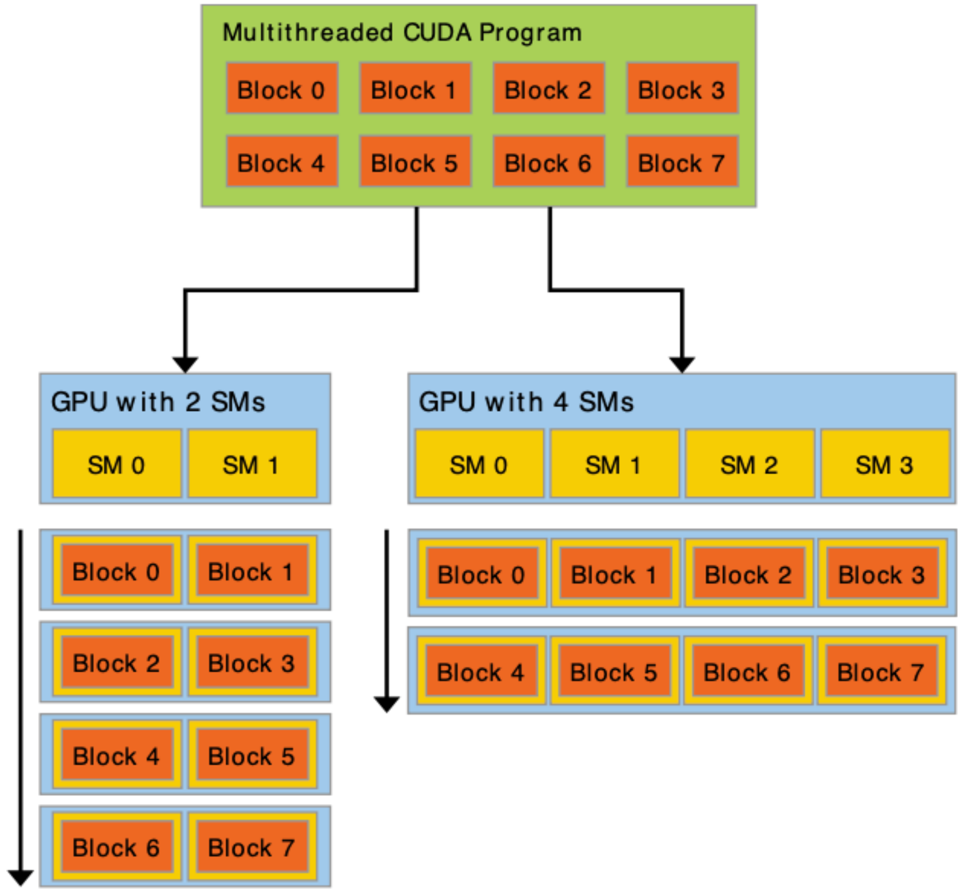
- The global scheduler dynamically dispatches blocks of each kernel grid on the various streaming multiprocessors (SM)
  - Each block is assigned to a single SM (no relocation!)
  - If assigned to an SM the block is called **active block**
  - Multiple blocks may be assigned to a single SM



2 different GPUs
2 different execution flow

Execution model – block dispatching

- Blocks are executed in any order independently from each other
- Transparent scalability!
  - Blocks are distributed on the SMs available in current GPU

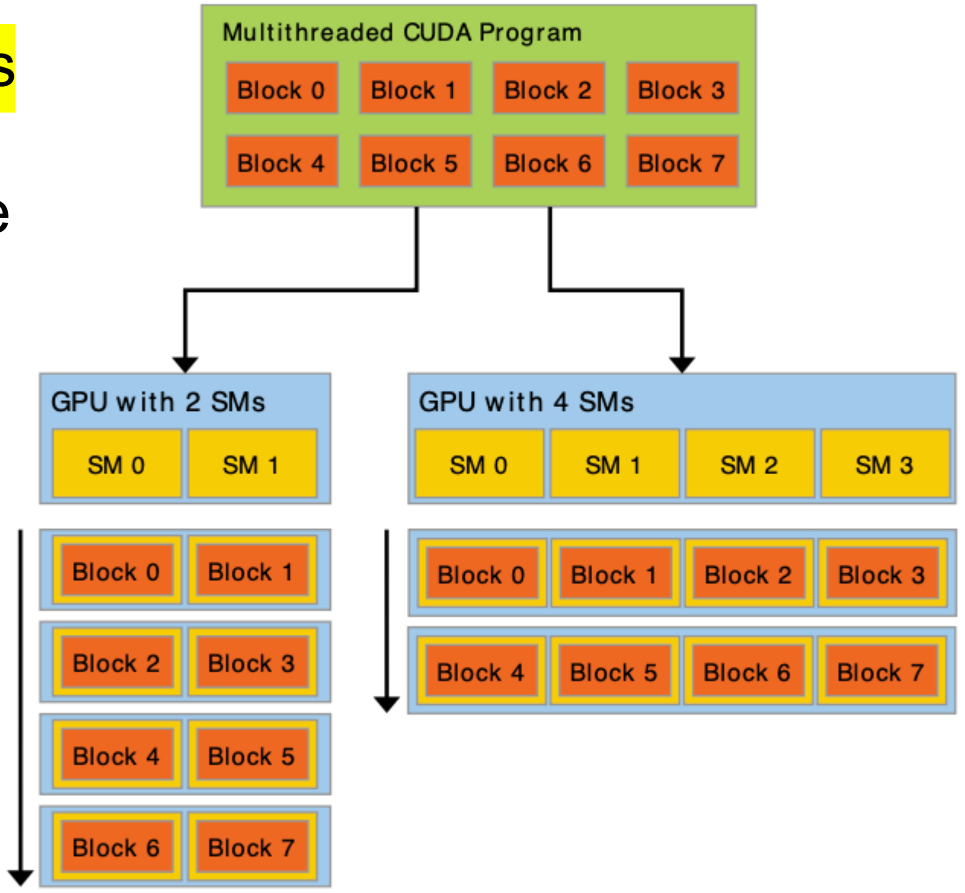


Execution model – block dispatching

- Each SM has predefined resources
  - Max #threads, max #blocks, #registers, shared memory size
- The kernel is profiled by the compiler in terms of resource requirements

⬇

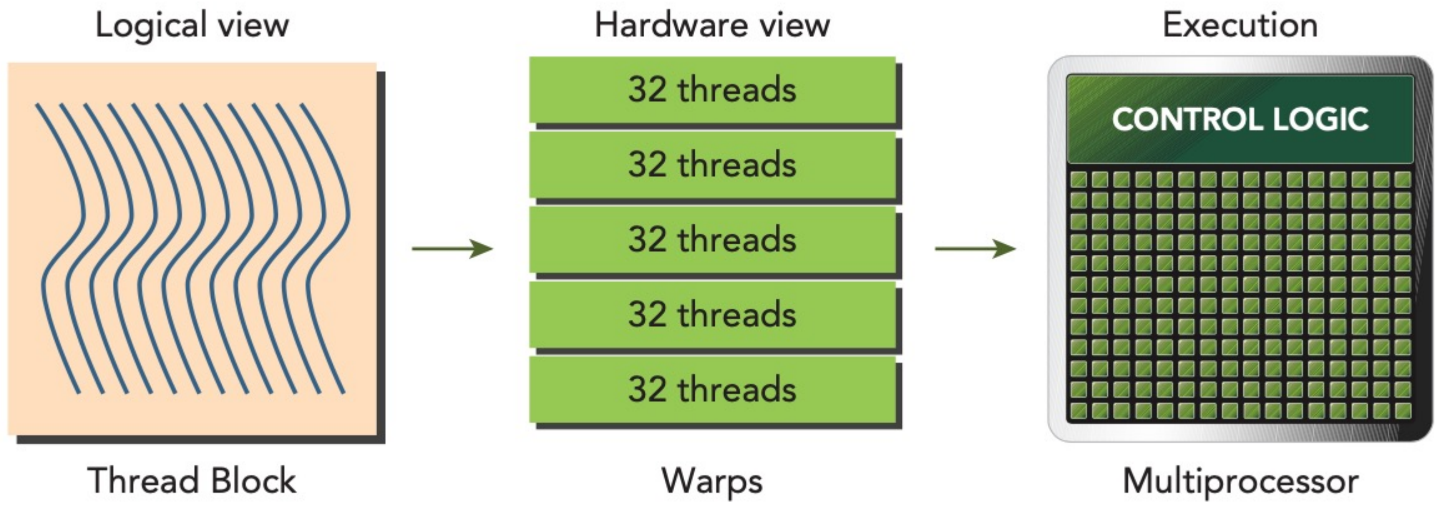
- A block is assigned to an SM only if there are enough resources
- Resources are reserved for the block for its entire execution



Execution model – warp scheduling

- The SM divides the block in **active warps** (32 threads each)
- A warp is executed with a SIMT approach
- Execution of several warps is interleaved

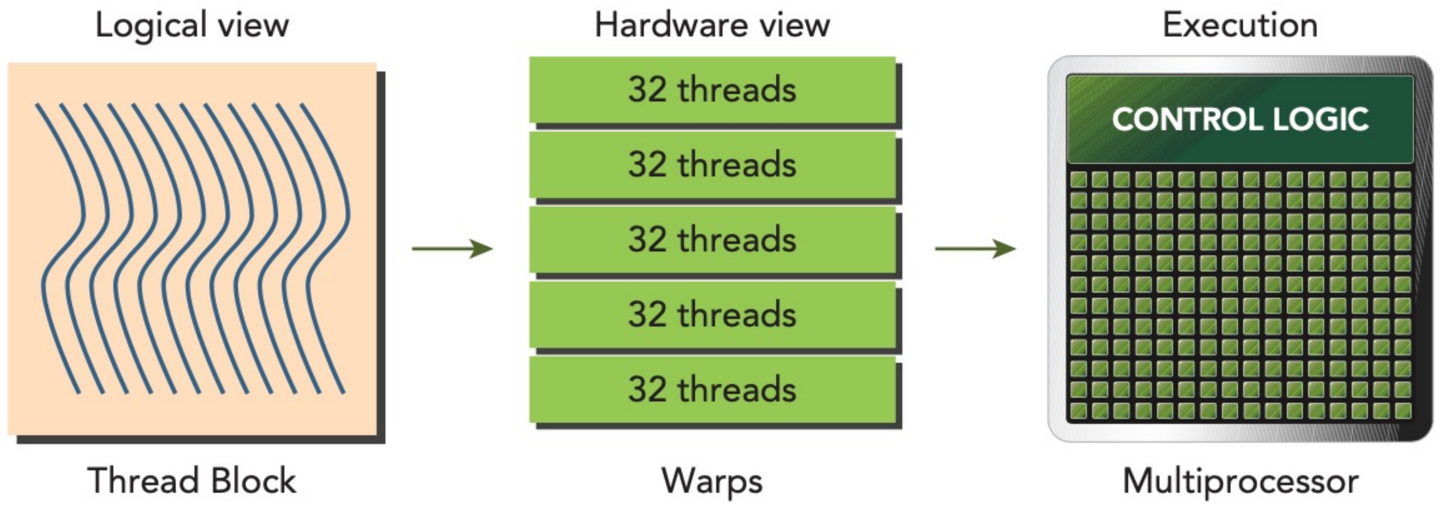
If block size is not multiple of 32, inactive threads are added to the last warp



Execution model – warp scheduling

- Each active warp may be classified as:
  - **Selected warp** – if currently executed
  - **Stalled warp** – if not ready to be executed
  - **Eligible warp** – if ready to be executed

Warp schedulers will select a warp to execute from eligible ones



Execution model – considerations

Application level

- Threads of the same grid are concurrent

BUT

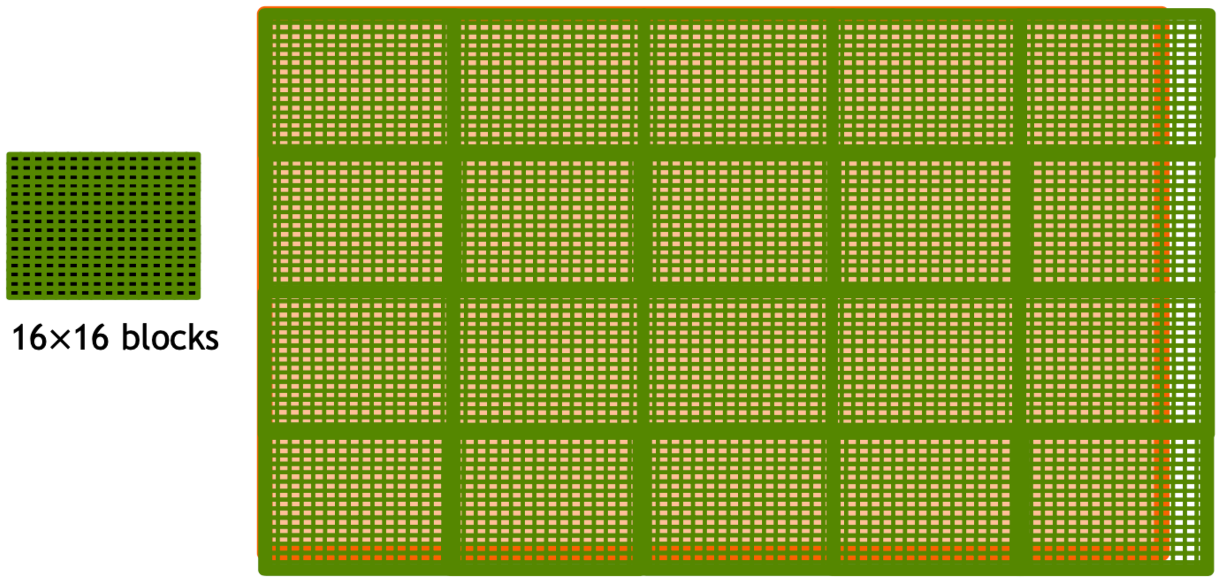
Architecture level

- Blocks may be dispatched to SMs at different instants of time
- Warps in each block may be scheduled in any order in the SMs

- **Advantages**: simpler and higher performance architecture

How to manage situations in which the data size cannot be a multiple of the number of threads in the grid?

- Let's consider the case where a 62x76 picture has to be processed with a kernel having 16x16 blocks



62×76 picture

How to manage situations in which the data size cannot be a multiple of the number of threads in the grid?

```
#define N 1000

__global__ void vsumKernel(int* a, int* b, int *c, int dim){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < dim)
    c[i] = a[i] + b[i];
}

int main(){
  /* ... */
  vsumKernel<<<ceil(N/256.0), 256>>>(d_va, d_vb, d_vc, N);
  /* ... */
}
```

How to manage situations in which the data size cannot be a multiple of the number of threads in the grid?

```
#define N 1000

__global__ void vsumKernel(int* a, int* b, int *c, int dim){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < dim)
    c[i] = a[i] + b[i];
}

int main(){
  /* ... */
  vsumKernel<<<ceil(N/256.0), 256>>>(d_va, d_vb, d_vc, N);
  /* ... */
}
```

Block size should be multiple of 32 for performance reasons

The block size is not a divider of the data size

How to manage situations in which the data size cannot be a multiple of the number of threads in the grid?

```
#define N 1000

__global__ void vsumKernel(int* a, int* b, int *c, int dim){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < dim)
    c[i] = a[i] + b[i];
}

int main(){
  /* ... */
  vsumKernel<<<ceil(N/256.0), 256>>>(d_va, d_vb, d_vc, N);
  /* ... */
}
```

Round up the number of blocks to the first greater integer value

Last threads in the last warp of each block do nothing

How to tune the block size?

- GPU is a throughput-oriented system
  - Best performance are achieved by maximizing resource utilization
  - High resource utilization is obtained by interleaving many warps in the SM
    - When a warp stalls, the SM executes another warp (**latency hiding!**)

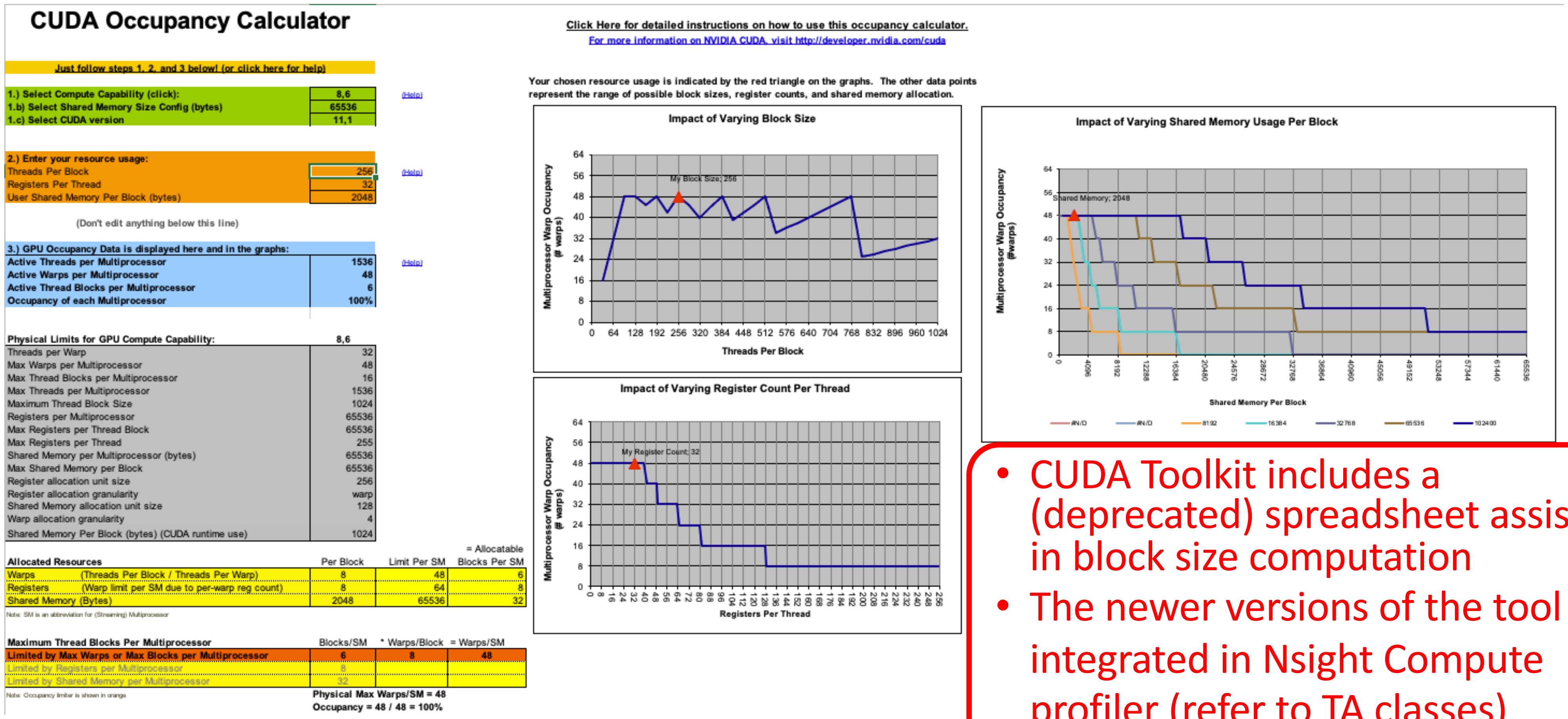
- A starting point to minimize kernel execution time is to **maximize SM occupancy**

$$\text{SM occupancy} = \frac{\text{active warps}}{\text{maximum warps}}$$

- Then, optimal performance is determined by means of a **systematic profiling** of kernel execution w.r.t. various aspects (resource utilization, memory accesses, …)

Discussed in TA classes

How to tune the block size?



CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8,6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11,1	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	8,6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	48	6
Registers	(Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)		2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48
Limited by Registers per Multiprocessor	8		
Limited by Shared Memory per Multiprocessor	32		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48
Occupancy = 48 / 48 = 100%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Impact of Varying Block Size
Multiprocessor Warp Occupancy (# warps)
My Block Size; 256
Threads Per Block
Impact of Varying Register Count Per Thread
My Register Count; 32
Registers Per Thread
Impact of Varying Shared Memory Usage Per Block
Shared Memory; 2048
Shared Memory Per Block

- CUDA Toolkit includes a (deprecated) spreadsheet assisting in block size computation
- The newer versions of the tool is integrated in Nsight Compute profiler (refer to TA classes)

How to tune the block size?

- Occupancy maximization is constrained by block resource requirements:
  - #registers
    - Compiler flag to get it: `--ptxas-options=-v`
    - Compiler flag to set it: `-maxregistercount=NUM`
  - Amount of shared memory
    - Computed from the source code

How to tune the block size?

- Guidelines for manual block size tuning:
  - Block size should be a multiple of warp size
  - Avoid small block sizes
  - Adjust block size up and down according to kernel resource requirements
  - Number of blocks should be much larger than the number of SMs
  - Conduct systematic experiments to identify the best configuration

The tuning activity becomes considerably much more complex with 2/3D blocks and grids

How to tune the block size?

- CUDA supports the computation of the occupancy metric:

```
__global__ void vsum(int* a, int* b, int *c);

int main(){
  int device;
  cudaDeviceProp prop;
  int numBlocks, activeWarps, maxWarps, blockSize;
  double occupancy;
  /* ... */
  cudaGetDevice(&device);
  cudaGetDeviceProperties(&prop, device);

  cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, vsum, blockSize, 0);
  activeWarps = numBlocks * blockSize / prop.warpSize;
  maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

  occupancy = (double)activeWarps / maxWarps;  /* ... */
}
```

How to tune the block size?

- CUDA supports the computation of the occupancy metric:

```
__global__ void vsum(int* a, int* b, int *c);

int main(){
  int device;
  cudaDeviceProp prop;
  int numBlocks, activeWarps, maxWarps, blockSize;
  double occupancy;
  /* ... */
  cudaGetDevice(&device);
  cudaGetDeviceProperties(&prop, device)

  cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, vsum, blockSize, 0);
  activeWarps = numBlocks * blockSize / prop.warpSize;
  maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

  occupancy = (double)activeWarps / maxWarps;  /* ... */
}
```

Get device properties

Compute the number of active blocks

How to tune the block size?

- CUDA supports the computation of the occupancy metric:

```
__global__ void vsum(int* a, int* b, int *c);

int main(){
  int device;
  cudaDeviceProp prop;
  int numBlocks, activeWarps, maxWarps, blockSize;
  double occupancy;
  /* ... */
  cudaGetDevice(&device);
  cudaGetDeviceProperties(&prop, device);

  cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, vsum, blockSize, 0);
  activeWarps = numBlocks * blockSize / prop.warpSize;
  maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

  occupancy = (double)activeWarps / maxWarps;  /* ... */
}
```

Parameters:
- Returned # blocks
- Analyzed kernel
- # threads in the block
- Amount of shared memory

#registers is retrieved from the kernel binary code

How to tune the block size?

- CUDA supports the computation of the occupancy metric:

```
__global__ void vsum(int* a, int* b, int *c);

int main(){
  int device;
  cudaDeviceProp prop;
  int numBlocks, activeWarps, maxWarps, blockSize;
  double occupancy;
  /* ... */
  cudaGetDevice(&device);
  cudaGetDeviceProperties(&prop, device);

  cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocks, vsum, blockSize, 0);
  activeWarps = numBlocks * blockSize / prop.warpSize;
  maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

  occupancy = (double)activeWarps / maxWarps;  /* ... */
}
```

Compute occupancy

How to tune the block size?

- CUDA supports the computation of the potential block size directly in the application source code:

```
#define N 1024

__global__ void vsumKernel(int* a, int* b, int *c){
  /*...*/
}

int main(){
  int gridS, minGridS, blockS;
  /* ... */
  cudaOccupancyMaxPotentialBlockSize(&minGridS, &blockS, (void *)vsumKernel, 0, N);
  gridS = (N + blockS - 1) / blockS;
  vsumKernel<<<gridS, blockS>>>(d_va, d_vb, d_vc);
  /* ... */
}
```

How to tune the block size?

- CUDA supports the computation of the potential block size directly in the application source code:

- Returned min grid size
- Returned # blocks
- Analyzed kernel
- Amount of shared memory
- Overall #threads

```
#define N 1024

__global__ void vsumKernel(int* a, int* b, int *c){
  /*...*/
}

int main(){
  int gridS, minGridS, blockS;
  /* ... */
  cudaOccupancyMaxPotentialBlockSize(&minGridS, &blockS, (void *)vsumKernel, 0, N);
  gridS = (N + blockS - 1) / blockS;
  vsumKernel<<<gridS, blockS>>>(d_va, d_vb, d_vc);
  /* ... */
}
```

#registers is retrieved from the kernel binary code

How to tune the block size?

- CUDA supports the computation of the potential block size directly in the application source code:

```
#define N 1024

__global__ void vsumKernel(int* a, int* b, int *c){
  /*...*/
}

int main(){
  int gridS, minGridS, blockS;
  /* ... */
  cudaOccupancyMaxPotentialBlockSize(&minGridS, &blockS, (void *)vsumKernel, 0, N);
  gridS = (N + blockS - 1) / blockS;
  vsumKernel<<<gridS, blockS>>>(d_va, d_vb, d_vc);
  /* ... */
}
```

Compute the grid size by rounding up

Warp divergence

- Warp divergence is another factor of performance degradation
  - It happens when threads in the same warp takes different directions during the execution of a branch or loop statement

```
__global__ void math_foo(int* ma){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float a=0;

  if(i%2)
    a=100;
  else
    a=200;
  ma[i]=a;
}
```

Warp divergence

- Organize and partition data so that all threads in the same warp take the same direction
  - The feasibility depends on the algorithm under design

```
__global__ void math_foo(int* ma){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float a=0;

  ~~if(i%2)~~
    a=100;
  else
    a=200;
  ma[i]=a;
}
```

```
if((i/WARPSIZE)%2==0)
```

```
#define WARPSIZE 32
```

In this example values saved in the array are sorted in a different way... Moreover, it is assumed to have an array size multiple of 64

Execution flow of the CUDA calls

- When the host program calls a CUDA function
  - The command is enqueued in a SW queue called **stream**
  - The GPU executes commands in order from the HW queue connected to the stream

- It is possible to enqueue multiple kernel launches in a row
  - They will be executed in a sequential way

We will see in next classes how to parallelize the execution of multiple kernels

Blocking and non-blocking CUDA calls

- CUDA calls may be:
  - **Blocking**
    - The host program is blocked until the function returns
    - E.g.: `cudaMemcpy`
  - **Non-blocking**
    - The host program continues asynchronously its execution
    - E.g.: kernel launch

System-level synchronization

- A barrier can be used to force the host to wait the termination of the kernel execution

Barrier: synchronization point where all threads have to stop and cannot proceed until all other threads reach that point

```
/*...*/
int main(){
   /*...*/
   /* kernel launch */
   vsumKernel<<<N/256.0, 256>>>(d_va, d_vb, d_vc);
   cudaDeviceSynchronize();
   /* ... */
}
```

The host is blocked till to the kernel termination

cudaDeviceSynchronize() can be used to force a sync for error checking in kernel execution (while testing the program)

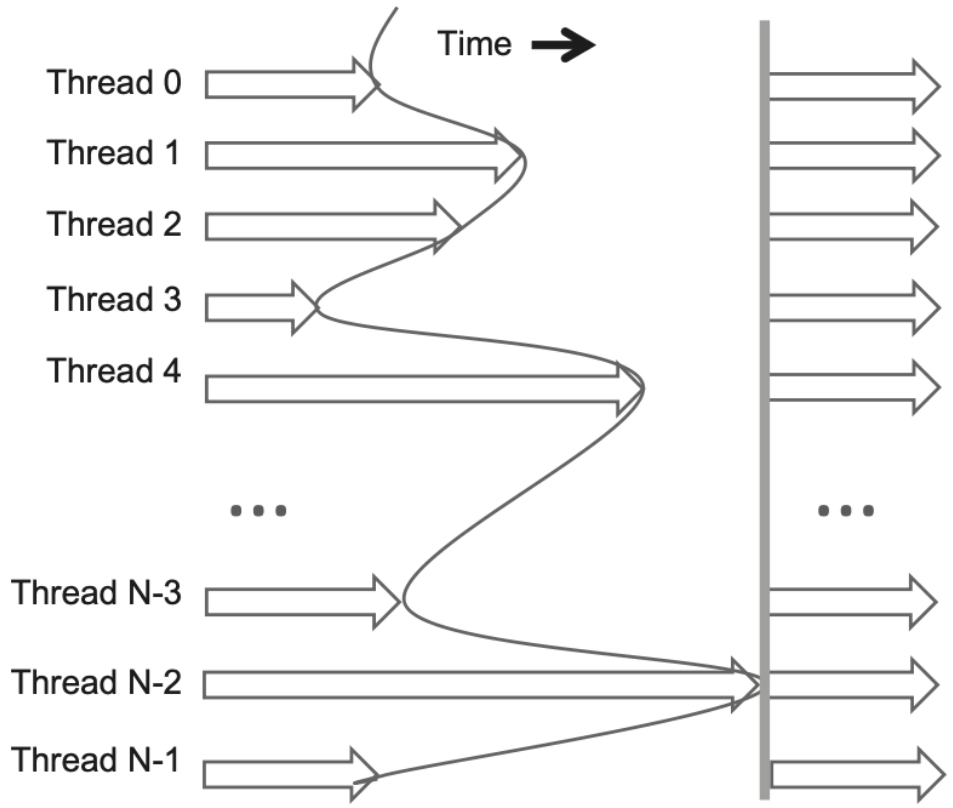
Thread synchronization

- Threads in the same grid are logically concurrent and are actually executed in any order
- Threads **CANNOT** share data without synchronization
  – Each thread has to write only its own data item
  – Threads cannot read data written by other threads

- Synchronization have to be explicitly invoked by the programmer by means of barriers
  – At block level
  – At grid level Read carefully later

Block-level synchronization

- Within the same block it is possible to synchronize threads by using a barrier
- Barrier call:
  `__syncthreads()`



We will discuss more on block synchronization in the next classes

Grid-level synchronization

- There is no barrier call applicable among threads of different blocks at grid level
  - Motivations: the GPU architecture does not feature inter-SM synchronization mechanisms
- To synchronize the entire grid, it is necessary to
  - Split the kernel code in two new ones at the desired synchronization point
  - Invoke the two kernels in sequence in the host code

Indeed, atomic instructions allow grid-level synchronization only during concurrent write operations (discussed in the next classes…)

Measure kernel execution time

- To analyze performance, it is necessary to measure kernel execution time
- Measures can be performed using either with
  - CPU timers, or
  - GPU timers
- Do remember that kernel launch is asynchronous!

Measure kernel execution time with CPU timers

```
#include <sys/time.h>
/*...*/
inline double milliseconds(){
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((float)tp.tv_sec * 1.e+3 + (float)tp.tv_usec * 1.e-3);
}

int main(){
  double start, end, exectime;
  /*...*/
  start = milliseconds();
  vsumKernel<<<N/256.0, 256>>>(d_va, d_vb, d_vc);
  cudaDeviceSynchronize();
  end = milliseconds();
  exectime = end - start;
  /*...*/
}
```

Measure kernel execution time with CPU timers

```
#include <sys/time.h>
/*...*/
inline double milliseconds(){
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((float)tp.tv_sec * 1.e+3 + (float)tp.tv_usec * 1.e-3);
}

int main(){
  double start, end, exectime;
  /*...*/
  start = milliseconds();
  vsumKernel<<<N/256.0, 256>>>(d_va, d_vb, d_vc);
  cudaDeviceSynchronize()
  end = milliseconds();
  exectime = end - start;
  /*...*/
}
```

The function returns the time in seconds elapsed since 00:00:00, January 1, 1970 (Unix Epoch)

Take time before kernel call

Synchronize and take time when the barrier unblocks

Compute elapsed time

Measure kernel execution time with GPU timers

```
/*...*/
int main(){
  /*...*/
  cudaEvent_t start, stop;
  float time;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start);
  vsumKernel<<<N/256, 256>>>(d_va, d_vb, d_vc);
  cudaEventRecord(end);
  cudaDeviceSynchronize();
  cudaEventElapsedTime(&time, start, stop);
  /*...*/
  cudaEventDestroy(start);
  cudaEventDestroy(end);
}
```

Measure kernel execution time with GPU timers

```
/*...*/
int main(){
  /*...*/
  cudaEvent_t start, stop;
  float time;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start);
  vsumKernel<<<N/256, 256>>>(d_va, d_vb, d_vc);
  cudaEventRecord(end);
  cudaDeviceSynchronize();
  cudaEventElapsedTime(&time, start, stop);
  /*...*/
  cudaEventDestroy(start);
  cudaEventDestroy(end);
}
```

- A CUDA event is a marker that can be pushed in the stream
- It can be used to mark to a specific point in the sequence of submitted commands
- The device records a timestamp for the event when it reaches that event in the stream

Measure kernel execution time with GPU timers

```
/*...*/
int main(){
  /*...*/
  cudaEvent_t start, stop;
  float time;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  cudaEventRecord(start);
  vsumKernel<<<N/256.0, 256>>>(d_va, d_vb, d_vc);
  cudaEventRecord(end);
  cudaDeviceSynchronize();
  cudaEventElapsedTime(&time, start, stop);
  /*...*/
  cudaEventDestroy(start);
  cudaEventDestroy(end);
}
```

Event variables

Create events

Push an event before kernel call

Push an event after kernel call

Wait for the end of kernel execution

Compute elapsed time

Destroy events

References

- Slides mainly based on:
  - W.-m. W. Hwu , D. B. Kirk, I. El Hajj, **Programming Massively Parallel Processors: A Hands-on Approach**, **<u>Chapters 3 and 4</u>**
  - J. Chen, M. Grossman, T. McKercher, **Professional Cuda C Programming**, **<u>Chapter 3</u>**