

An introductory course on General Purpose Computing on GPUs

Francesco Lettich

Insight Lab
Universidade Federal do Ceará, Fortaleza, Brasil



Outline

- ✦ **Historical background**
- ✦ **GPU architecture overview**
- ✦ **GPU programming overview**
- ✦ **Advanced topics**
- ✦ **Final questions**

Historical background – '70s

- **Arcade system boards** started using **specialized graphics chips** since the '70s to accelerate the execution of some simple 2D graphic operations.



Historical background – '80s

- Hardware producers kept adding features to graphics chips, mainly in terms of **supported 2D operations** and **memory capabilities**.



- Production of the first, proper **video cards** towards the end of the '80s.

Historical background – '90s

- Producers kept adding more and more 2D functionalities: emergence of the first **2D APIs** (e.g., WinG graphics library, DirectDraw).
- By mid-'90s, **real-time 3D graphics** became common – **3DFX GPUs?**
- Emergence of **3D graphics APIs** to program the GPUs: **DirectX, OpenGL, Glide.**

Historical background – Present / 1

- From the beginning of the '00s, **GPUs** and the **related APIs** started to become **increasingly complex** and **powerful**, due to the **needs** of the **video-game** industry.
- Developers and academics started to implement **non-graphics operations** on GPU – for instance, **matrix multiplication**.
- Trick: **image-based** representations for **input/output data structures**, and implement **vertex** and **pixel shaders (functions)** that do “**generic**” **computation** on them.

Historical background – Present / 2

- Unfortunately, GPGPU was still quite hard to do: limited **programming model** and **earlier GPUs** had many **architectural limitations**, especially related to **memory accesses**.
- **First release** of the NVIDIA **CUDA** framework in **2007**: provides the first programming model to perform **general purpose computation** on GPUs. **GPUs** also started to **adapt** to **GPGPU** needs.
- **Current trend**: modern GPU architectures are **integrating hardware features** typical of multi-core CPUs.

Why using the GPUs? / 1

- ✦ Provide **massive amounts of (parallel) computational resources**. In some cases, GPU-based algorithms achieve **speedups > 100x** w.r.t. the **sequential CPU** counterpart.
- ✦ In case of big speedups, GPUs can be **very convenient** in terms of **economic costs** and **power consumption** w.r.t. CPUs.
- ✦ Nowadays, GPUs are “**general**” enough to implement **complex, general, parallel** algorithms...
- ✦ Fourier transform, Monte-Carlo methods, operations on matrices, cryptography, bioinformatics, astronomy, etc.

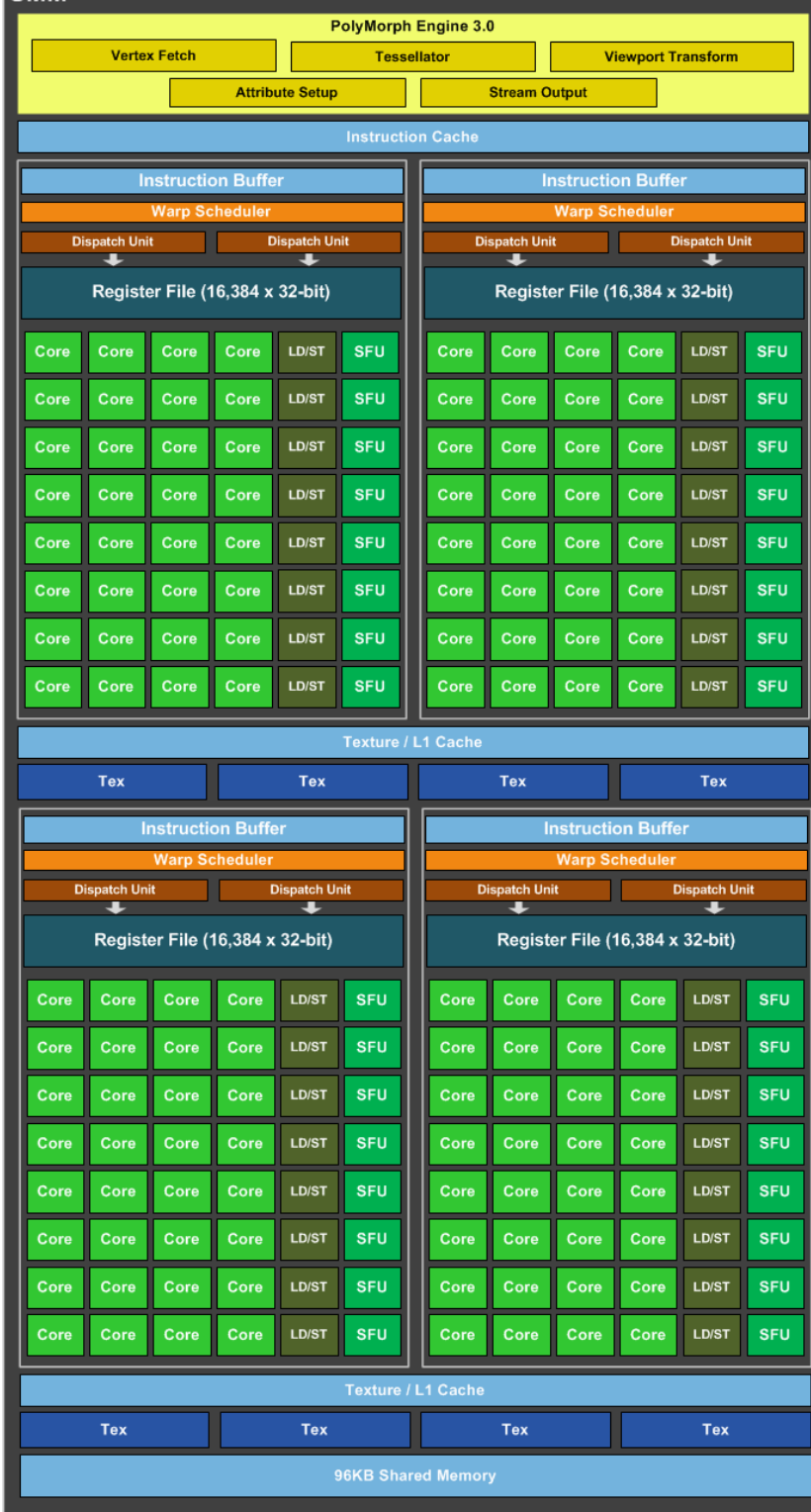
Why using the GPUs? / 2

- The use of GPUs **looks promising** if we have **problems** that allow to **devise solutions** that...
- are **massively data-parallel**, with **none** or **few dependencies** among **data-parallel** tasks.
- the related **data structures** (properly designed) **fit well** the **peculiarities** of the **hierarchy of memories** typical of **GPUs**.

Running outline

- Historical background
- GPU architecture overview
- GPU programming overview
- Advanced topics
- Final questions






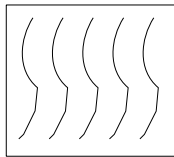
Architecture – SM / 1

- A streaming multiprocessor (SM) is a highly parallel multiprocessor.
- Each SM possesses **Z** physical cores (green squares).
- Threads are scheduled on the cores in groups of **W** ($Z \bmod W = 0$) threads. **W = 32** on current GPUs.
- Each group of threads is called warp. A warp is essentially a SIMD thread.
- Threads of a warp execute in lockstep.
- An SM can execute up to Z / W^{12} warps per cycle.

Architecture – SM / 2

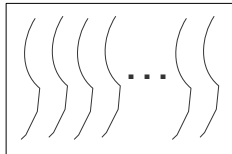

Thread


Core

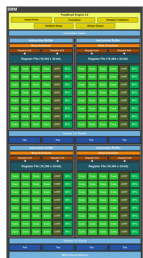


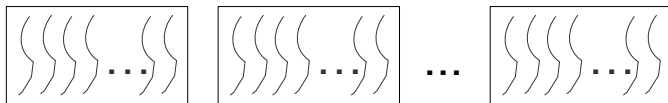
Warp (SIMD thread)
(32 threads executed in lockstep)


Group of 32 cores

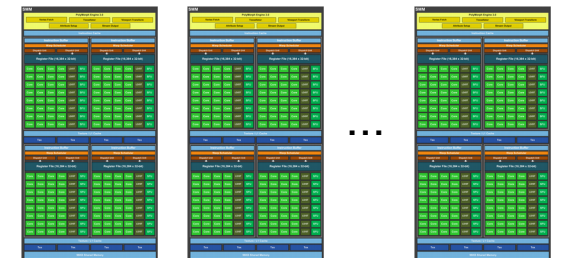


Thread-block
(set of warps)

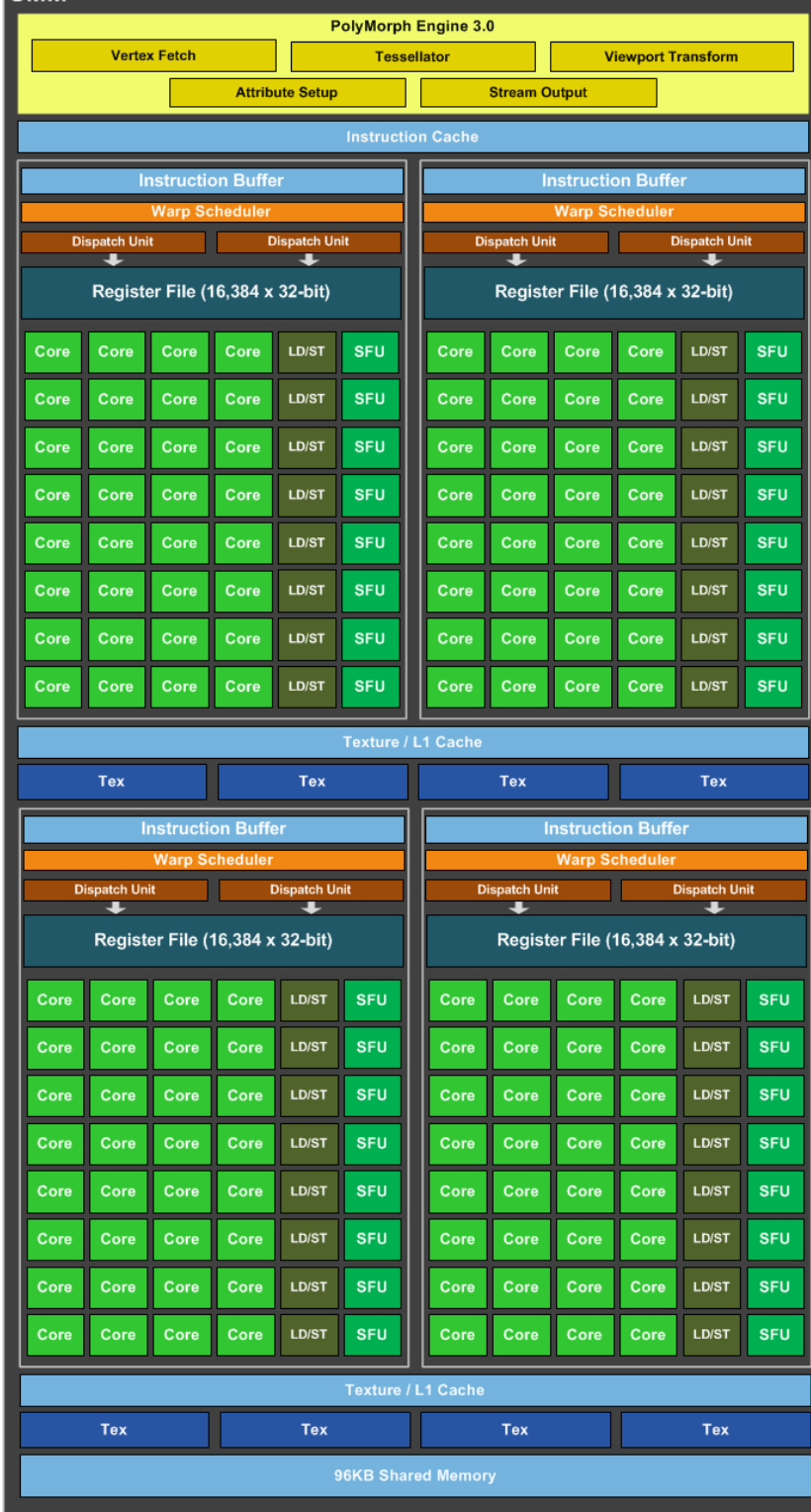

SM



Set of thread-blocks



Set of SMs



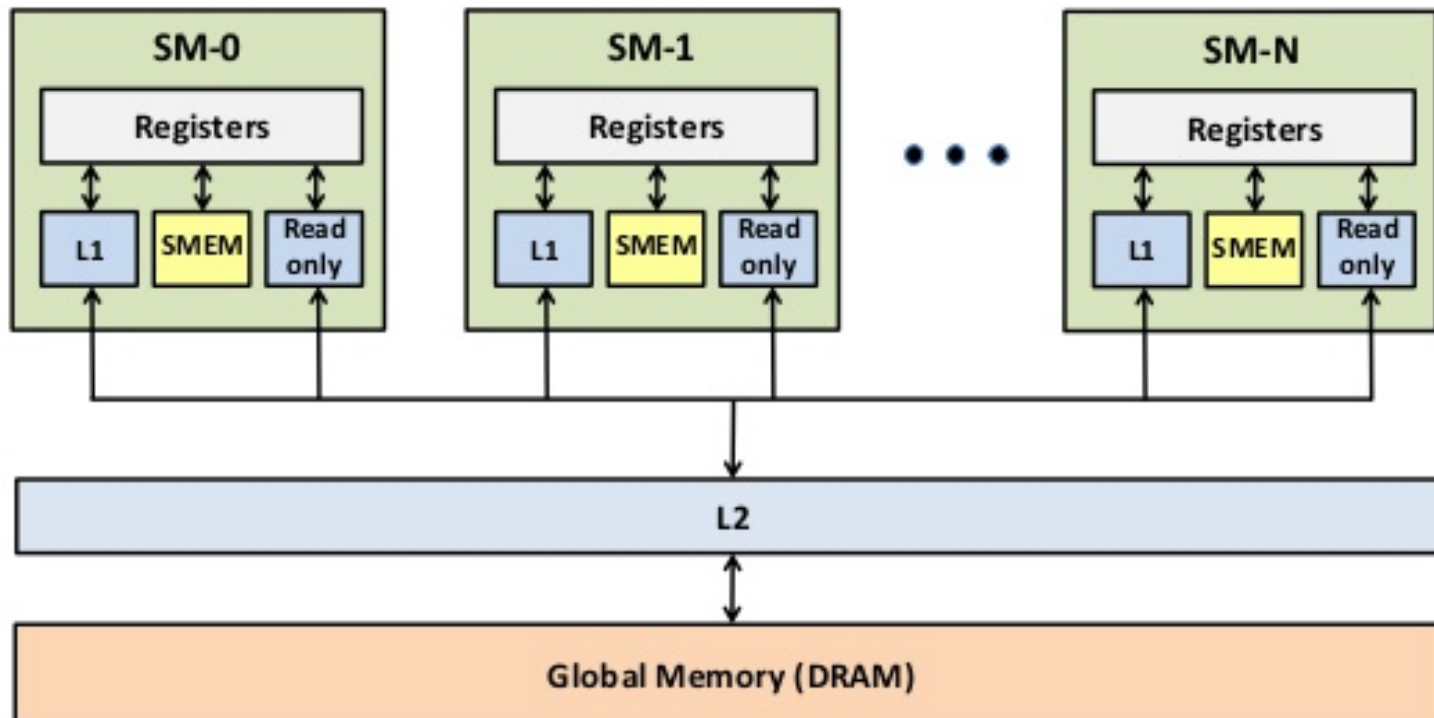
Architecture – SM / 3

- Each SM possesses also a set of **special function units (SFU)**: operations on **doubles**.
- Each SM possesses a set of **load/store units**.
- Finally, each SM possesses **its own memory resources**, shared across its cores:
 - **Shared memory**, **thread registers**, **L1 cache**, **constant cache**.

Architecture – Hierarchy of memories

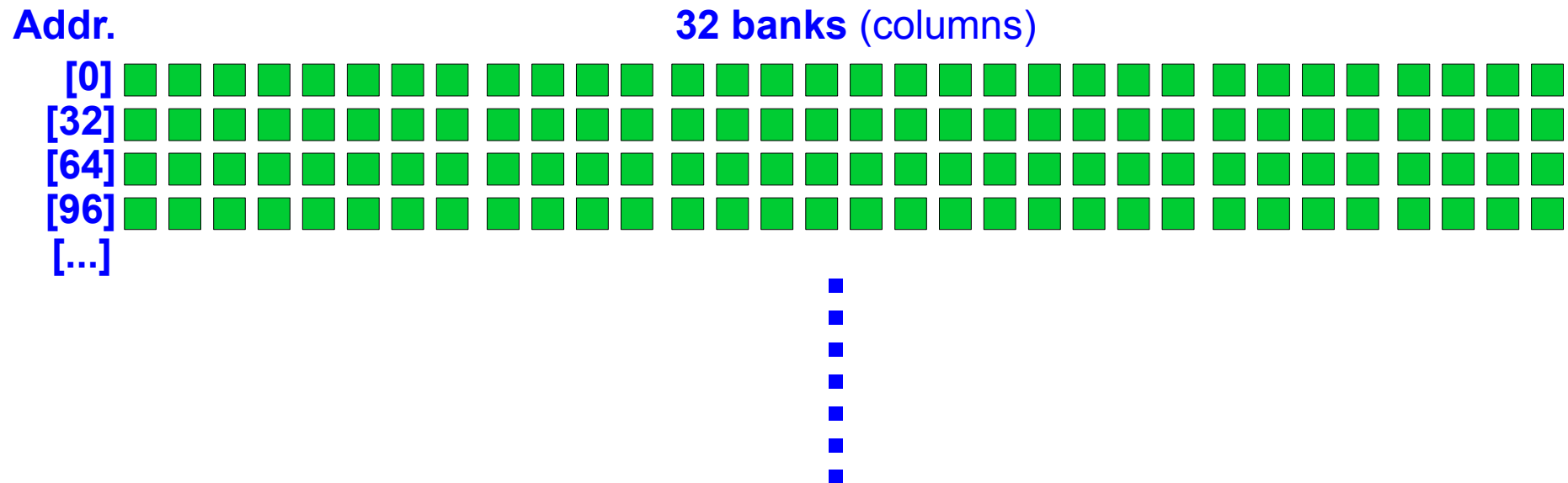
- GPUs have a **complex hierarchy** of memories.

Kepler Memory Hierarchy



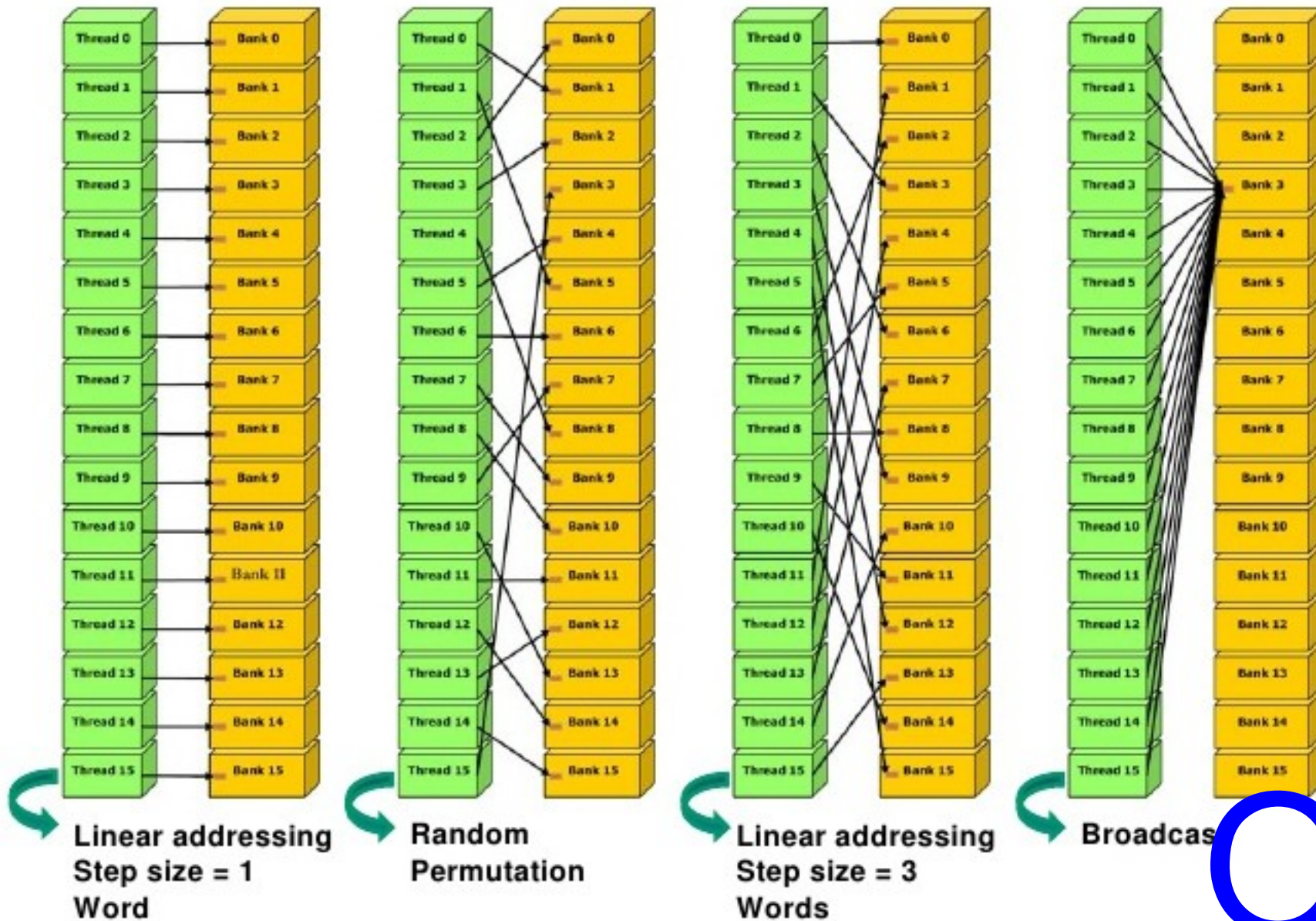
Architecture – Shared memory / 1

- Each SM has its **own shared memory unit**.
- Typically **small** in size (e.g., **96KB** on the GTX 1080).
- Not a cache, as it is **directly** managed by the **programmer**.
- At least **one order of magnitude faster** than the GPU main memory (320 GB/s vs > 2 TB/s, GTX 1080)...**VERY** fast.
- Ideal for managing **small chunks** of data characterized by **spatial/temporal locality**.
- **Structured** to take into **account** the **SIMD execution** model used with **warps**.

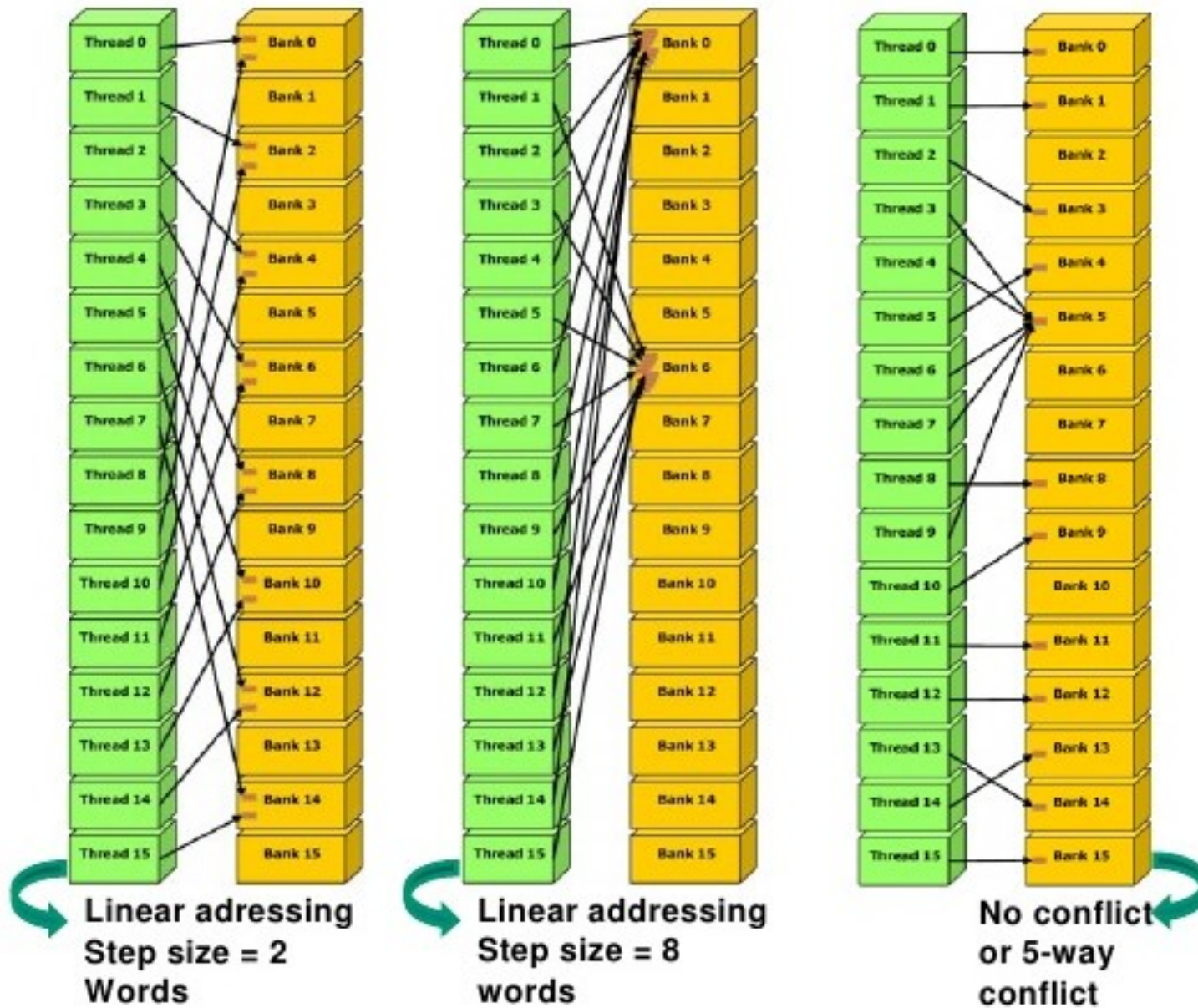


- ➡ Inside a **warp**, at a given cycle each **thread** should access a bank **not accessed** by **any other thread** in the warp. 17

Architecture – Shared memory / 3



Architecture – Shared memory / 4



KO!

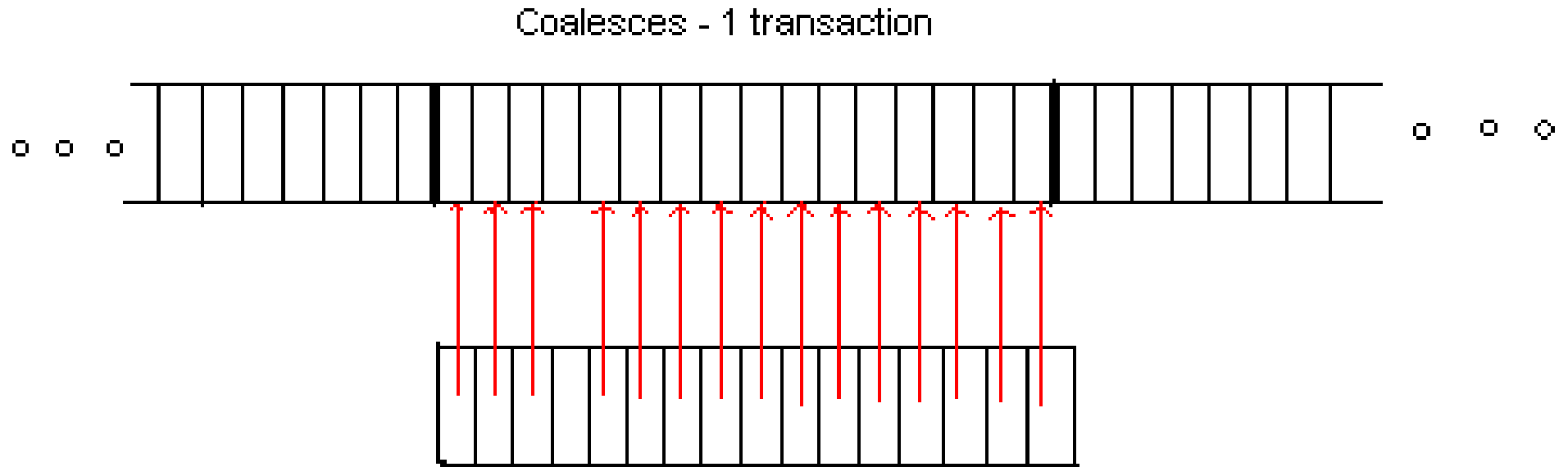
Architecture – Global Memory / 1

- ✦ **Global memory** is the “**big**” memory in the GPUs.
- ✦ Typical size is **4-8 GB** on current GPUs.
- ✦ Lots **faster** than the typical CPU RAM: **320 GB/s** (GTX 1080) vs **25 GB/s** (DDR4 3200 MT/s).
- ✦ **Structured** to take into **account** the **SIMD** execution model used with **warps**.
- ✦ Ideal to store data that **exibith strong spatial/temporal locality OR** that can be moved in **small chunks** to **shared**.

Architecture – Global Memory / 2

- ✦ Threads of a warp need to access data in global memory **in parallel**...we need proper **cooperative access patterns**!
- ✦ We want to **minimize** the **number of transfers** needed to read/write data...
- ✦ Cache line of the L1/L2 caches is typically **128 bytes**...
- ✦ Consequently, the global memory is optimally accessed when accessing memory **segments aligned at 128 bytes**.
- ✦ If threads access the elements of a segment with a 1-1 relationship, we achieve full throughput thanks to **coalescing**.

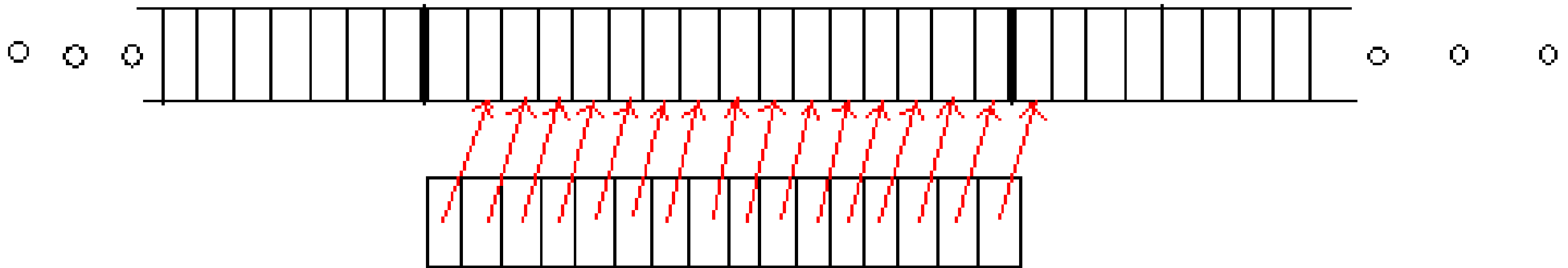
Architecture – Global Memory / 3



- ✚ **OK:** the threads of the warp are accessing the elements in global memory **aligned** within the **same 128-byte segment**. This yields a **single transaction**.
- ✚ **NOTE:** it's also okay to have **permuted accesses**, as long as these fall inside the **same segment**.

Architecture – Global Memory / 4

Misaligned - 16 transactions



- **KO:** the threads of the warp access the elements **misaligned** with respect to the **128 bytes boundaries**.
- This yields 2 transactions...**waste of bandwidth!**

Architecture – Global Memory / 5

- ✦ In general, it is **not always possible** to perform **perfect coalesced accesses**...
- ✦ This **depends** on the **underlying data structures**.
- ✦ If possible, **exploit spatial/temporal locality** – this favours cached accesses and compensates the performance penalty due to suboptimal memory access patterns.

Architecture – L1 and L2 caches / 1

- ✦ **Each SM** has its own **L1 cache** – typically **48KB** on current GPUs. It **caches** the **accesses** to the **global memory**.
- ✦ All the SMs access a **shared L2 cache**, which role is to cache the accesses to the **global memory**. Typical size is **1.5-2MB** on current GPUs.
- ✦ Also, on current models the **L1 cache line** is **128 bytes**, while the **L2 cache line** is **32 or 128 bytes**.

Architecture – L1 and L2 caches / 2

- Performance is maximized when it is possible to use data structures stored to **global memory** whose **accesses** are **characterized** by:
 - strong **spatial locality**
 - strong **temporal locality**
 - a **mix** of the above.
- Indeed, the thousands of parallel cores are **computationally “hungry”**
- They need to be continuously feeded with data to process: **important** to carefully **design** the **data structures**!

Architecture – Other memories

- Each SM has a **set of registers** – typically $64K * 4$ bytes (**256KB**). Extremely **fast**.
- Used to **implement** the **stack of threads** (local variables).
- Each SM has a read-only cache, i.e., **constant memory**. Useful if threads executed in an SM have to **repeatedly access** a set of **constants**.
- **Small** in size (8KB).

GPU architecture – Final remarks

- ✦ To sum up, GPUs turn out to be convenient when we have a problem whose solution can be **massively parallelized...**
- ✦ with data-parallel tasks having **none or few dependencies**.
- ✦ **warps** are guaranteed to be executed in **lockstep**: this must be **exploited** to **maximize parallelism**.
- ✦ **lots of care** when **accessing** the various **memories...**
GPU algorithms are typically **memory-bounded** due to (sometimes **inevitable**) **sub-optimal** access patterns.

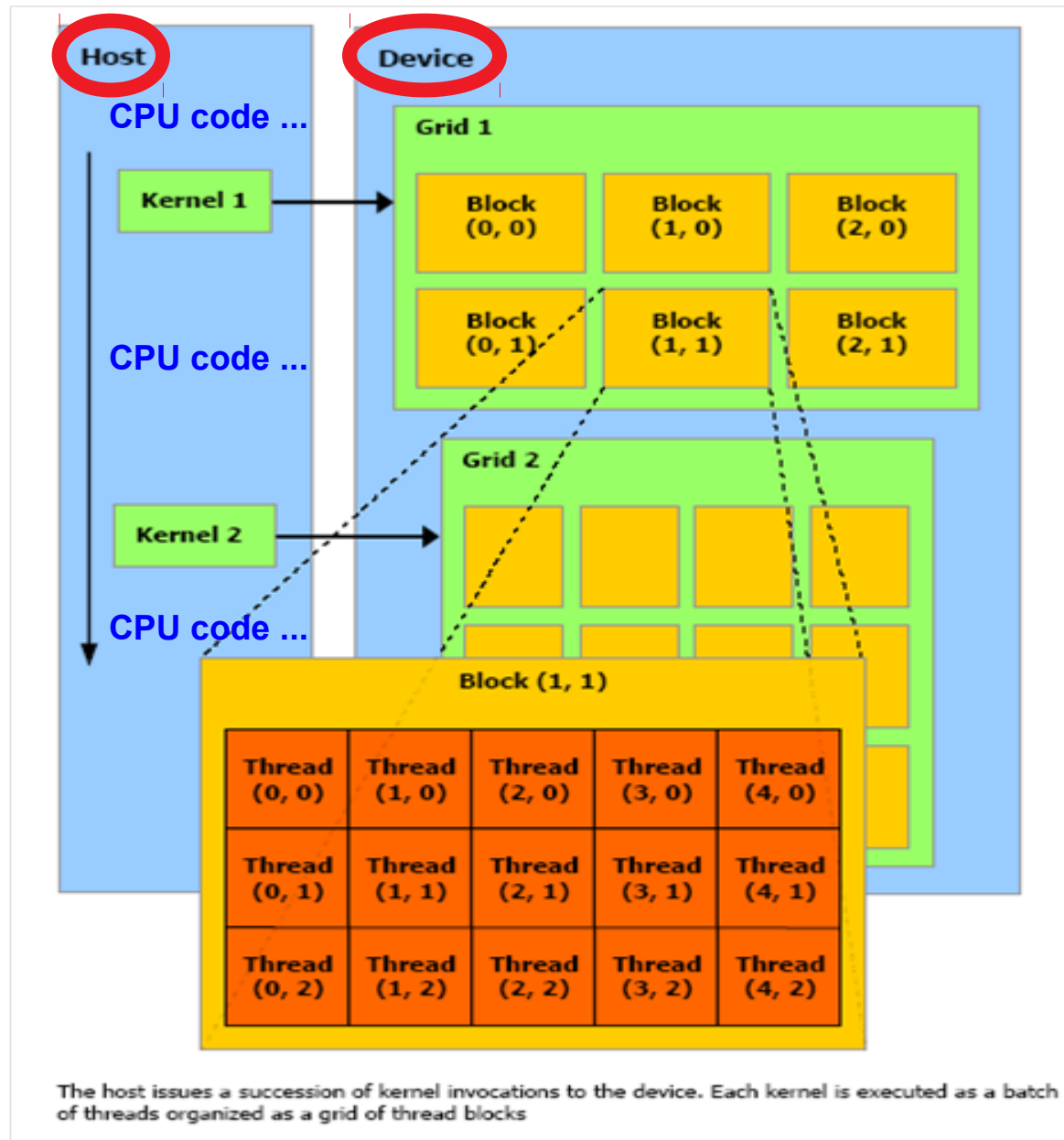
Outline

- Historical background
- GPU architecture overview
- **GPU programming overview**
- Advanced topics
- Final questions

GPU programming – Intro / 1

- The **CUDA framework** provides a **compiler (nvcc)** and a set of **tools** that allow to **program/execute/debug** GPU-based applications.
- **CUDA** logically **separates** each application in **two distinct parts**:
 - **Host program**: the **components** of the application that are executed **on CPU**.
 - **Device functions**: the **components** of the application that are executed **on GPU**: **set of functions** compiled for the GPU. Also known as **kernels**.

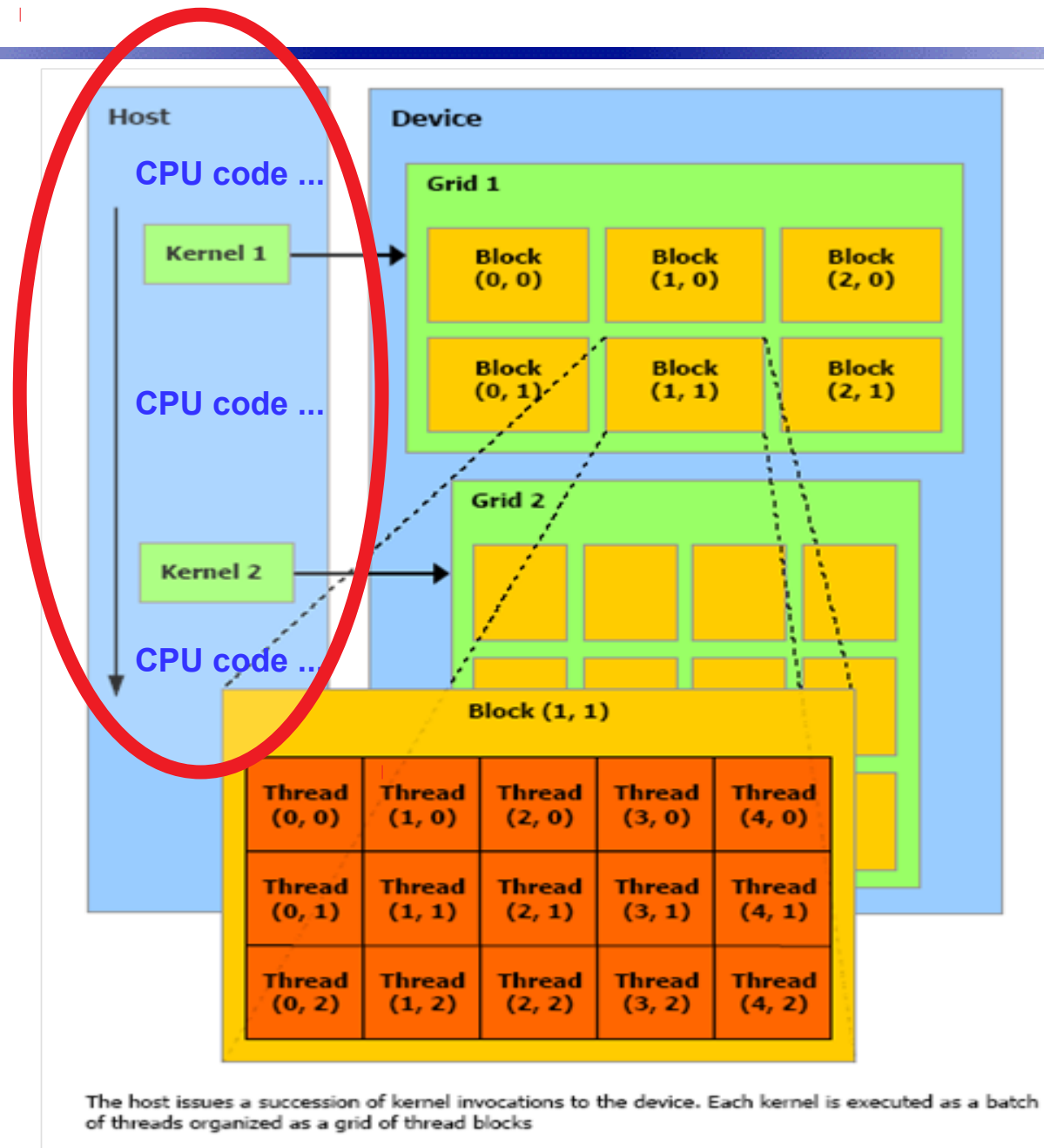
GPU programming – Intro / 2



GPU programming – Host program / 1

- The **host (CPU)** part of the program is in charge of:
 - **allocating and initializing the data structures** used by **the kernels**.
 - **executing sequential/multi-core CPU code**.
 - **managing the activities of the GPU(s)**, mainly **moving data** from/to the GPU(s) and **managing the execution of kernels**.

GPU programming – Host program / 2



GPU programming – Host program / 3

➡ “Hello World” example (C):

```
// Device program
__global__ void kernelTest() { }
```

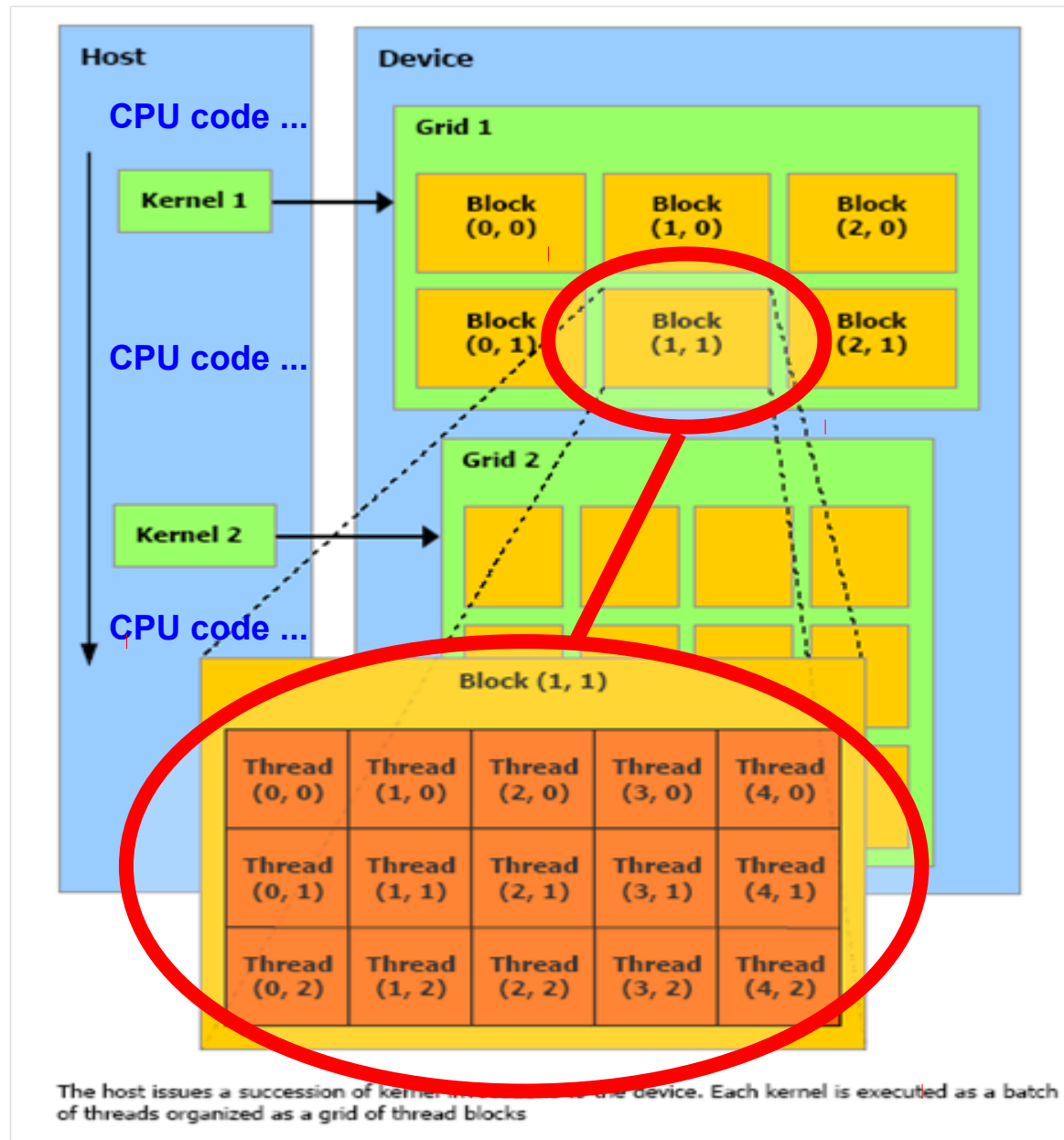


```
// Host program
int main()
{
    dim3 dimGrid(2,3);           // A grid of 2X3 blocks.
    dim3 dimBlock(3,5);          // A grid of 3x5 threads/block.
    kernelTest<<<dimGrid, dimBlock>>>>(); // Execute the kernel on GPU.
    printf( "Hello, World!\n" );
    return 0;
}
```

➡ The **__global__** qualifier specifies that `kernelTest` has to be executed on the **GPU**.

➡ **kernel<<<B, TB>>>** specifies that the function *kernel* executes on **TB** * **B** threads, where **B** is the # of **blocks** and **TB** is the # of **threads/block**.

GPU programming – Host program / 4



GPU programming – Host program / 4

- String transformation example (**allocation** and **initialization** of data structures):

```
int main(int argc, char** argv)
{
    // Initialize the string
    char str[] = "Hello World!";

    // Mangle the content in str.
    // NOTE - the null character is left intact
    for(int i = 0; i < 12; i++)
        str[i] -= i;
```

....

GPU programming – Host program / 5

+...

```
// allocate memory on the GPU
char *d_str;
int size = sizeof(str);
cudaMalloc((void**)&d_str, size);

// copy the string to the device
cudaMemcpy(d_str, str, size, cudaMemcpyHostToDevice);

// invoke the kernel
// NOTE: 3 blocks, each having 4 threads.
transformString<<<3,4>>>(d_str);
```

GPU programming – Host program / 6

...

```
// retrieve the results
cudaMemcpy(str, d_str, size, cudaMemcpyDeviceToHost);

// free allocated memory
cudaFree(d_str);

printf("%s\n", str);
}
```

GPU programming – Kernels / 1

- A **device** (GPU) function, aka **kernel**, represents a function to be **executed** on **GPU**.
- The **workload** gets **partitioned** across **thread-blocks**. Each **thread-block** is in charge of computing a **data-parallel task**.
- Each **thread-block** is assigned to a specific **streaming multiprocessor**.
- Each thread-block is **executed independently** with respect to other thread-blocks.
- **Threads within** a thread-block are **scheduled in warps**, i.e., **groups** of 32 threads. Each thread **executes the function**.
- Thus, a data-parallel task can be **further decomposed in**³⁹ **sub-tasks** within a thread-block.

GPU Programming – Kernels / 2



GPU programming – Kernels / 3

- **Threads of a thread-block share some common resources**
 - those of the SM to which the thread-block is assigned:
 - **Shared memory** (max 48KB out of 96KB)
 - **L1 cache** (48KB)
 - **Registers** of the SM (256KB)
 - **Constant memory** (8KB)
- This reflects the **architectural design choices behind the GPUs** mentioned at the beginning of these slides.
- Depending on the resources needed by individual thread-blocks, each SM may **execute concurrently multiple blocks**.
- Having multiple thread-blocks per SM is a **GOOD THING** (keeps SMs **busy** with computation!).

GPU programming – Kernels / 4

- String transformation example (cont'd)...

```
__global__ void helloWorld(char* str)
{
    // determine the index of this thread.
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // unmangle output
    str[idx] += idx;
}
```

- The **__global__** qualifier specifies that the function executes on GPU and can be called from the CPU.
- Each thread determines its own “**piece**” of data to process according to its **index**.

GPU programming – Kernels / 5

- ✦ **Main types** of memories that can be used within a kernel...

```
__global__ void memories(char* str)
{
    // Local variable (each thread has one).
    int idx = threadIdx.x;

    // Shared memory variable.
    // NOTE: shared among the threads of the block.
    __shared__ int str[10];
}
```

Outline

- + Historical background
- + GPU architecture overview
- + GPU programming overview
- + **Advanced topics**
- + Final questions

Advanced topics

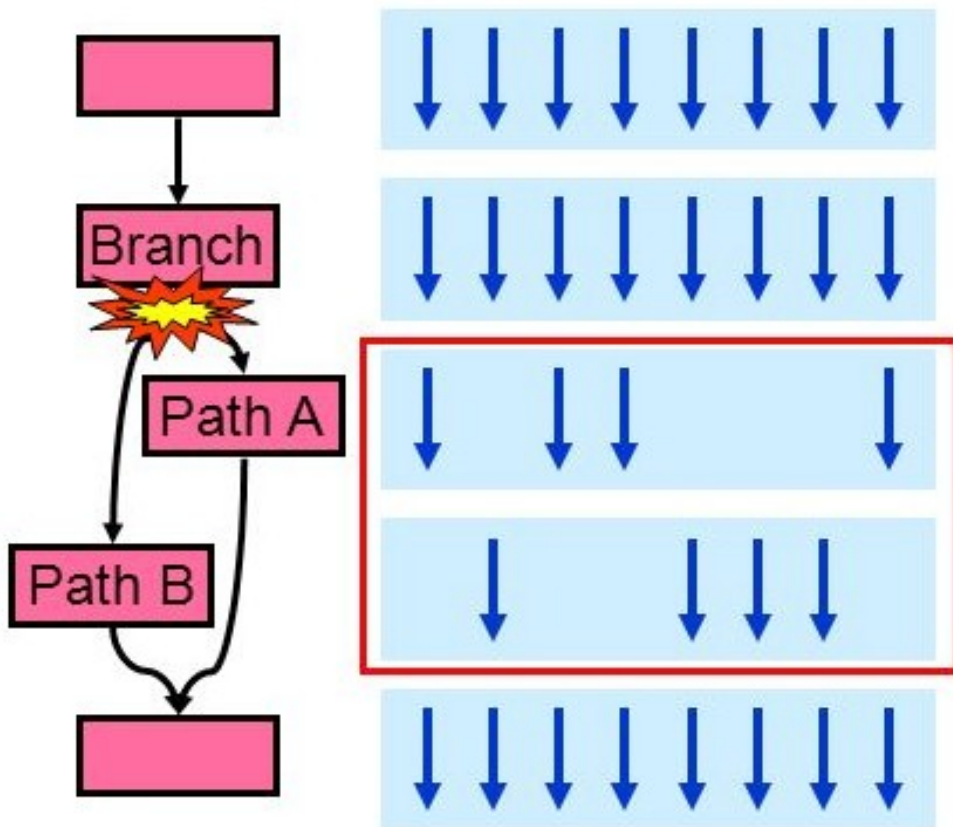
- Some important **technical details** to consider during development of GPU-based algorithms:
 - Warp-level parallelism, aka how to **splitting** a **data-parallel task** in further **sub-tasks**.
 - Branch divergence within **warps**.
 - **Block-level barriers** to manage **data dependencies** between **warps**.
 - Maximize SM occupancy, i.e., how to choose a **proper number of threads per thread-block**.

Advanced topics – Branch Divergence / 1

- ✦ Remember: a warp is a SIMD thread, i.e., a set of 32 threads executing the **same instruction** during some **cycle**.
- ✦ **Problem:** what happens if we have an “if – else” construct?

```
__global__ void divergence()  
{  
    // Index of this thread.  
    int idx = blockIdx.x * blockDim.x +  
             threadIdx.x;  
  
    // Example of branch divergence...  
    if(idx % 2 == 0)  
    {...} // Path A  
    else  
    {...} // Path B  
}
```

Advanced topics – Branch Divergence / 2



- ➔ First, **path A** is executed.
- ➔ **Half** of the **threads** in a warp remain **inactive**.
- ➔ Then, **path B** is executed.
- ➔ **Again**, **half** of the **threads** in a warp remain **inactive**.
- ➔ Sequentialization **wastes resources**.

Advanced topics – Warp-level parallelism / 1

- ✦ Each **warp** is executed **independently** from the other warps of its thread-block...
- ✦ They just **share** the **resources** of their **thread-block** (SM) and **live** as long as its thread-block is **alive**.
- ✦ We can use this property to further split a data-parallel task into **data-parallel sub-tasks**.
- ✦ **Example**: let us suppose that we have a set of vectors A and a set of vectors B: we want to add the i -th vector of A with the i -th vector of B...

Advanced topics – Warp-level parallelism / 2

```
__global__ void warpPar(int** A, int** B, int** out,
int sizeVector)
{
    // Global index of the thread.
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Global index of the warp.
    int warpID = idx / 32;

    // Index of thread within its warp.
    int threadID = idx % 32;

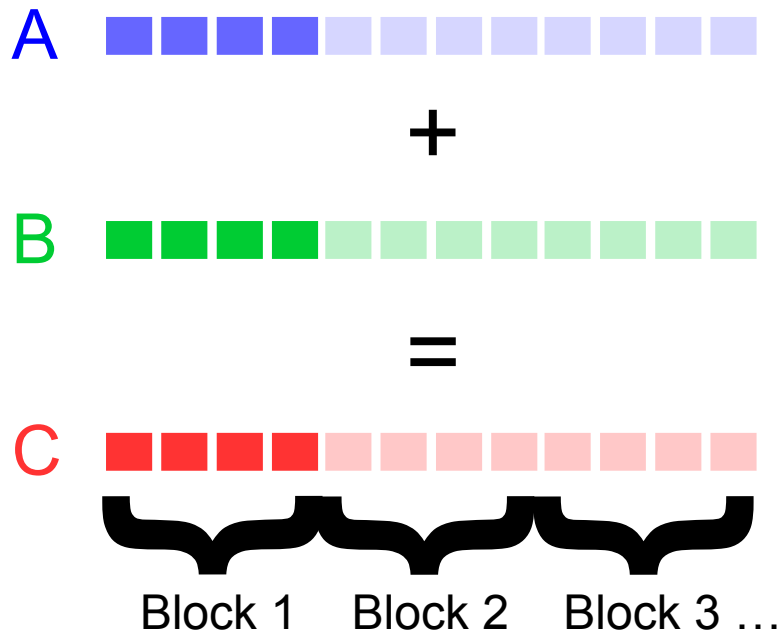
    // Initialize the pointers.
    int* vecA = A[warpID];
    int* vecB = B[warpID];
    int* out = out[warpID];
```

...

Advanced topics – Warp-level parallelism / 3

...

```
for(int i = threadIdx; i < sizeVector, i = i + 32)
    vecOut[i] = vecA[i] + vecB[i];
}
```



✦ Each warp performs **independently** an **add** on a **pair** of **vectors** from A and B.

✦ Each warp performs the add on a **block** of **32 elements** at a time.

✦ Exploits **coalescing**.

Advanced topics – Barriers / 1

- ✦ Sometimes we have **data-dependencies** between warps...
- ✦ Example: let us suppose that the threads in a thread-block have to **initialize** a **region** of **shared** memory...

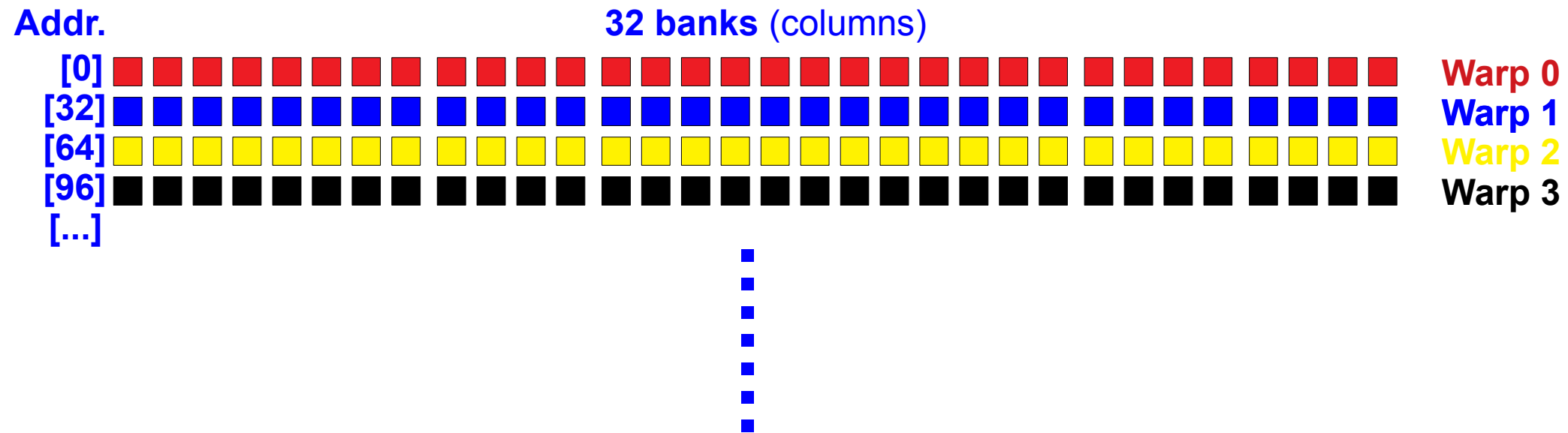
```
__global__ void dataDep(...)
{
    const int SIZEBUFF = 4096;
    __shared__ int buffShared[SIZEBUFF];

    // Init the shared memory buffer.
    for(uint32_t i = threadIdx.x; i < SIZEBUFF; i += blockDim.x)
        buffShared[i] = ~0;

    __syncthreads(); // Thread-block level barrier.

    // Remaining computation...
```

Advanced topics – Barriers / 2



➤ As warps proceed **independently**, we need to wait that all the **warps** in a thread-block **finish** to **initialize** the **segments** of shared memory they have in charge.

➤ If we do not take care of this, the **area** of the **shared memory** that has to be initialized will be **inconsistent**!

Advanced topics – Kernel Configuration / 1

- ✦ How do we **choose** a **proper number** of threads per thread-block?
- ✦ Depends on **several factors...**
 - ✦ **how many resources** each **thread-block** needs to process its “piece” of input?
 - ✦ **Memory latencies** caused by accesses to the global memory have important effects?
Having **lots** of **warps** per **SM** helps to **mask latencies!**

Advanced topics – Kernel Configuration / 2

- ✦ Keeping in mind that each SM can run concurrently multiple thread-blocks (warps), we want to maximize **SM occupancy**.
- ✦ **Occupancy** is the **average number of warps** that the SM is actively executing per cycle.
- ✦ We can follow these **best-practices**:
 - ✦ When the amount of **resources needed** per single **thread-blocks** is high, use a **high number of threads** (max per thread-block: 1024).
 - ✦ Conversely, just **maximize occupancy** according to the GPU specifications.

Some useful pointers...

- **CUDA programming guide:**

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

- **CUDA best practices guide:**

<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>

- **CUDA Thrust library:**

<http://docs.nvidia.com/cuda/thrust/index.html>

- **Miscellaneous CUDA documentation** (compiler, tuning, debugging, etc.):

<http://docs.nvidia.com/cuda/>

Questions?