



Introduction to GPU Usage

HIP Winter School 2021

Computational Science Department · Dr. Guido Juckeland · g.juckeland@hzdr.de · www.hzdr.de/fwcc



Content of the Day

What can you expect?

Goal for the day:

- You are able to program parallel GPU accelerated applications yourself
- You have a basic understanding of performance considerations and common error messages

Topics covered:

- Basic concepts of GPU parallel programming
- Introduction to CUDA and related techniques for using GPUs
 - Data transfers
 - Kernel generation
 - Device data management
- Basic techniques for high performing GPU codes

Content of the Day

What do you know / expect?

Turn to the common pad at

<https://notes.desy.de/eP0TXBEdQG6UebrLIBGKGQ?both>

Answer the following two questions:

- When did you last use a GPU?
- What do you expect from the day?

Let's get started...

Parallelism or why GPUs?



Parallelism is Everywhere



Parallelism occurs whenever multiple agents work together in solving a larger problem. You use it when one agent cannot solve the problem in time.

Parallel Computing

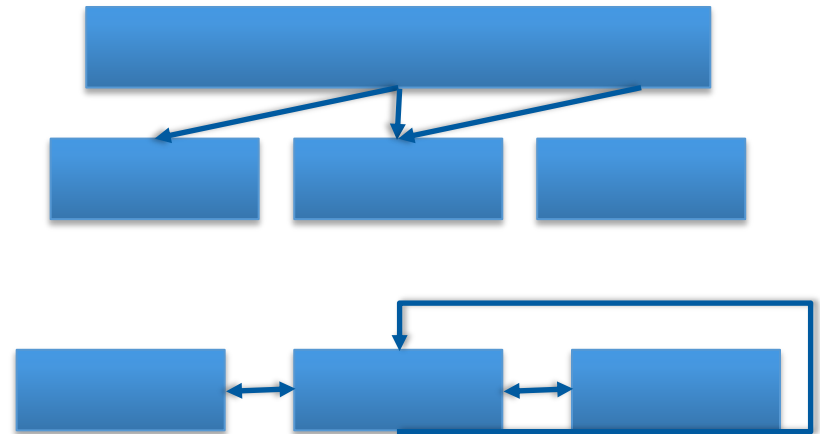
Take a large computational problem

Break it into smaller parts

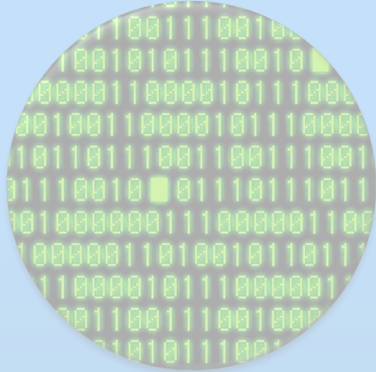
Solve the parts concurrently

If necessary:

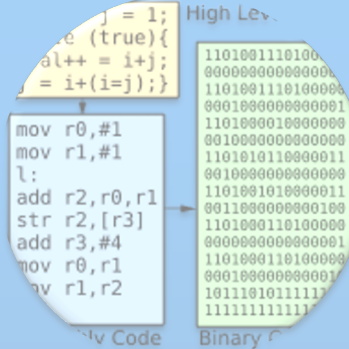
- Communicate partial results
- Repeat until solution is reached



Types of Parallelism



Bit-wise
parallelism



Instruction
level
parallelism



Data
parallelism



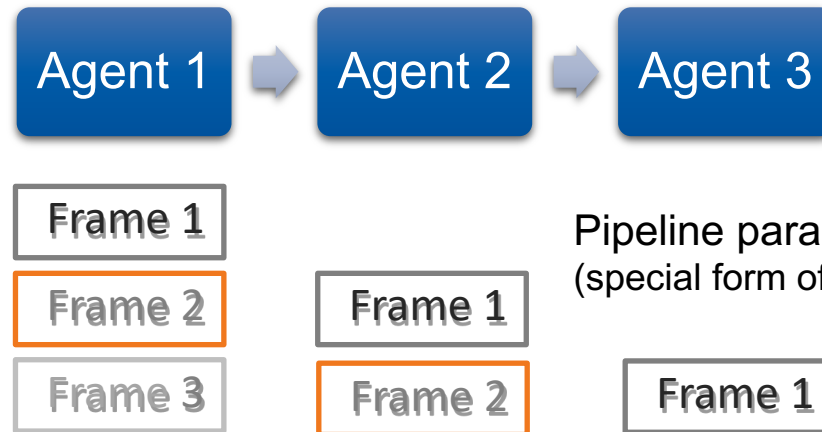
Task
parallelism



Task Parallelism

Performs (sub-)tasks on sets of data (concurrently = in parallel, if possible)

- Example application:
Video processing
- Agent 1: Load frames
- Agent 2: Remove blur
- Agent 3: Adjust colors,...
- ...



Pipeline parallelism
(special form of task parallelism)

Agent = Thread or process

Task parallelism usually does not scale with the problem size

Load-balancing and synchronization are important

Multi-core CPUs are often a good choice for task parallel applications

On a coarse-grained level (low communication), GPUs also can handle it well, but their main talent is data parallelism

Data Parallelism

Performs the same task on different data

- Example: Video processing
- Agent 1: works on top left corner
- Agent 2: works on top right corner
- Agent 3: works on bottom left corner
- ...



Agent = Thread or process

Works well on CPUs and GPUs

Requires dividable data

Parallelization

Porting or refactoring a code to run parallel on GPUs is usually no easy task

- **Find hotspots and embarrassingly¹ parallel-enabled portions**

- exploit data parallelism and SIMD programming model

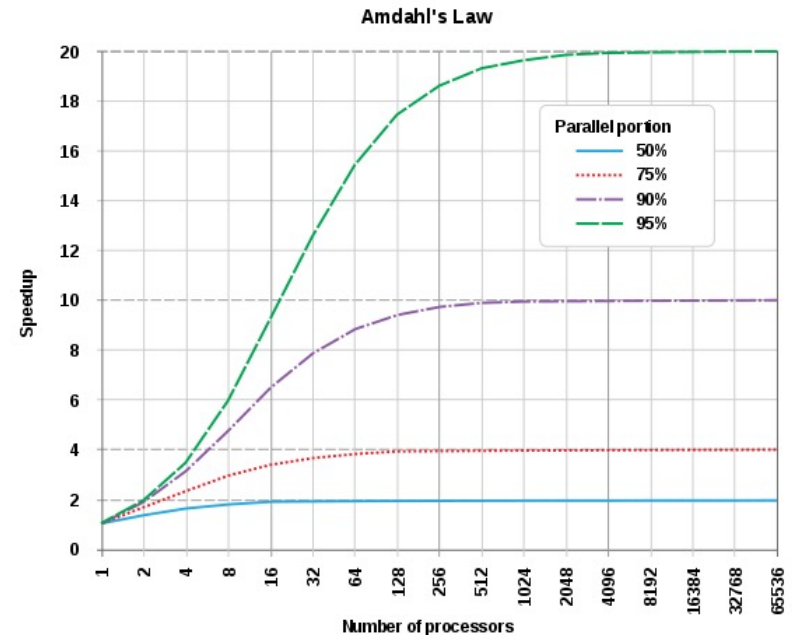
- **If serial part of the algorithm is too high, then parallelization does not help much**

- Amdahl's Law:
$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

S – theoretical speedup

N – number of processors

P – parallel portion



Daniels220 CC 3.0, [wiki](#)

Top 500 (June 2021)

# Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1 Japan	Fugaku, A64FX, Fujitsu	7,630,848	442,010.0	537,212.0	29,899
2 United States	Summit, AC922, IBM Power9 (NVIDIA GV100)	2,414,592	148,600.0	200,794.9	10,096
3 United States	Sierra, IBM Power9 (NVIDIA GV100)	1,572,480	94,640.0	125,712.0	7,438
4 China	Sunway, SW26010	10,649,600	93,014.6	125,435.9	15,371
5 United States	Permuter, AMD EPYC (NVIDIA A100)	706,304	64,590.0	89,794.5	2,528
6 United States	Selene, NVIDIA DGX SuperPOD, AMD EPYC (NVIDIA A100)	555,520	63,460.0	79,215.0	2,646
7 China	Tianhe-2A (NUDT Matrix-2000)	4,981,760	61,444.5	100,678.7	18,482
8 Germany	JUWELS Booster, AMD EPYC (NVIDIA A100)	449,280	44,120.0	70,980.0	1,764
9 Italy	HPC5, Dell + Intel Xeon, (NVIDIA V100)	669,760	35,450.0	51,720.8	2,252
10 United States	Frontera, Dell + Intel Xeon	448,448	23,516.4	38,745.9	N/A

Top500

Parallel C++ APIs (Incomplete)

	CPU	GPU (NVIDIA)	GPU (AMD)	FPGA
CUDA	-	x	-	-
<u>HCC</u>	x	-	x	-
OpenCL	x	x	x	x
HIP	x	x (via nvcc)	x (via hcc/clang)	-
SYCL	x	x (experimental)	<u>x</u> (prototype)	x
OpenGL Direct3D Vulkan	-	x	x	-
<u>OpenMP4/5</u>	x	x	x	<u>x</u>
OpenACC (GCC, NVIDIA)	x	x	x	<u>o</u> , <u>o</u>
MPI	x	GPU transfers if MPI is CUDA-aware	-	-

Parallel Hardware Examples (Incomplete)

<u>Nvidia</u>	GTX	Tesla	SoC
Kepler	680	K80	Tegra K1
Maxwell	980	M40	Tegra X1
Pascal	1080	P100	Tegra X2
Volta	(Titan)	V100	Xavier
Turing	2080	-	Orin?
Ampere	3080	A100	Orin?

<u>AMD</u>	Gaming	Pro (W+S)	MI
GCN 1st	HD 7970	FirePro W2100	
GCN 2nd	HD 8770	FirePro W4200	
GCN 3rd	R9 285	FirePro S7150	MI8
GCN 4th	RX 480	RadeonPro 580	MI6
GCN 5th	RX Vega64	RadeonPro SSG	MI25, ..
RDNA	RX 5700	?	MI100

Some Accelerator Cards



Nvidia Tesla V100 (Volta)
(2017)

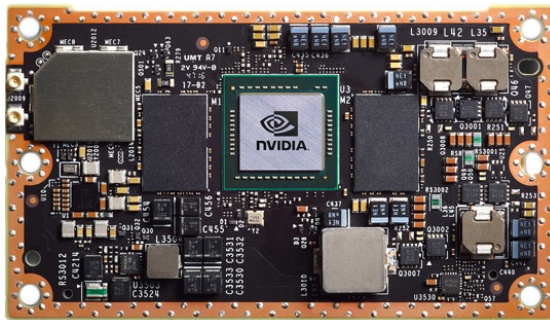
XBOX ONE X



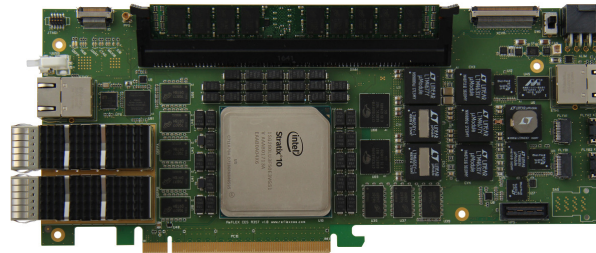
PS4 PRO

... semi-custom AMD
GPUs (2017)

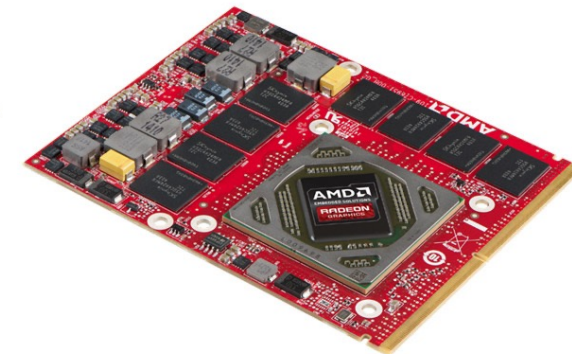
Radeon Pro SSG (Vega)
(2017)



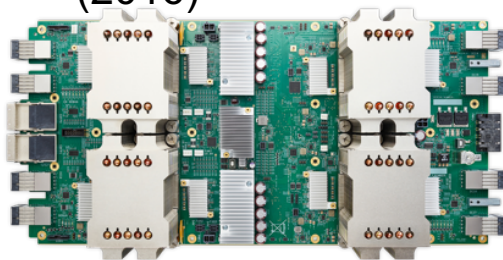
Nvidia Tegra X2 (Pascal)
(2016)



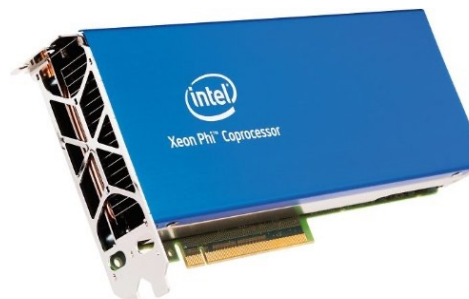
Altera Stratix 10 FPGA
(2013)



Radeon Embedded
E9550 MXM (Polaris)
(2016)



Google TPU
(2017)



Intel Xeon Phi 7120P
(2013)

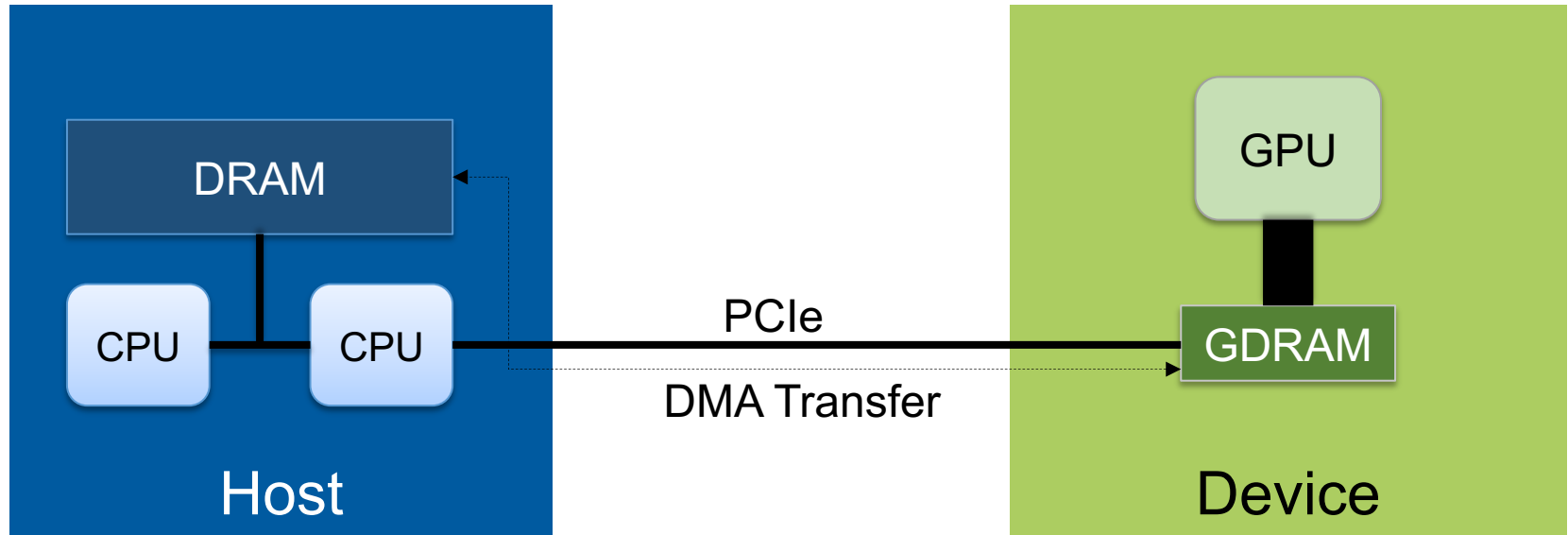
Moving in...

Getting data onto and off the GPU



GPU System Setup

- CUDA assumes a system with a host and a device with own memory each



Hardware

Software



Defining the Terms HOST and DEVICE



HOST



DEVICE

CUDA: Allocating and Freeing Memory on the Device

```
int main( void ) {
```

```
    float *device_pointer;    //Pointer to float element
```

```
    // The following line allocates memory for one float on the GPU and
```

```
    // sets device pointer to the beginning of that memory area
```

```
    cudaMalloc( &device_pointer, sizeof(float) );
```

```
    // The following line releases the allocated GPU memory for
```

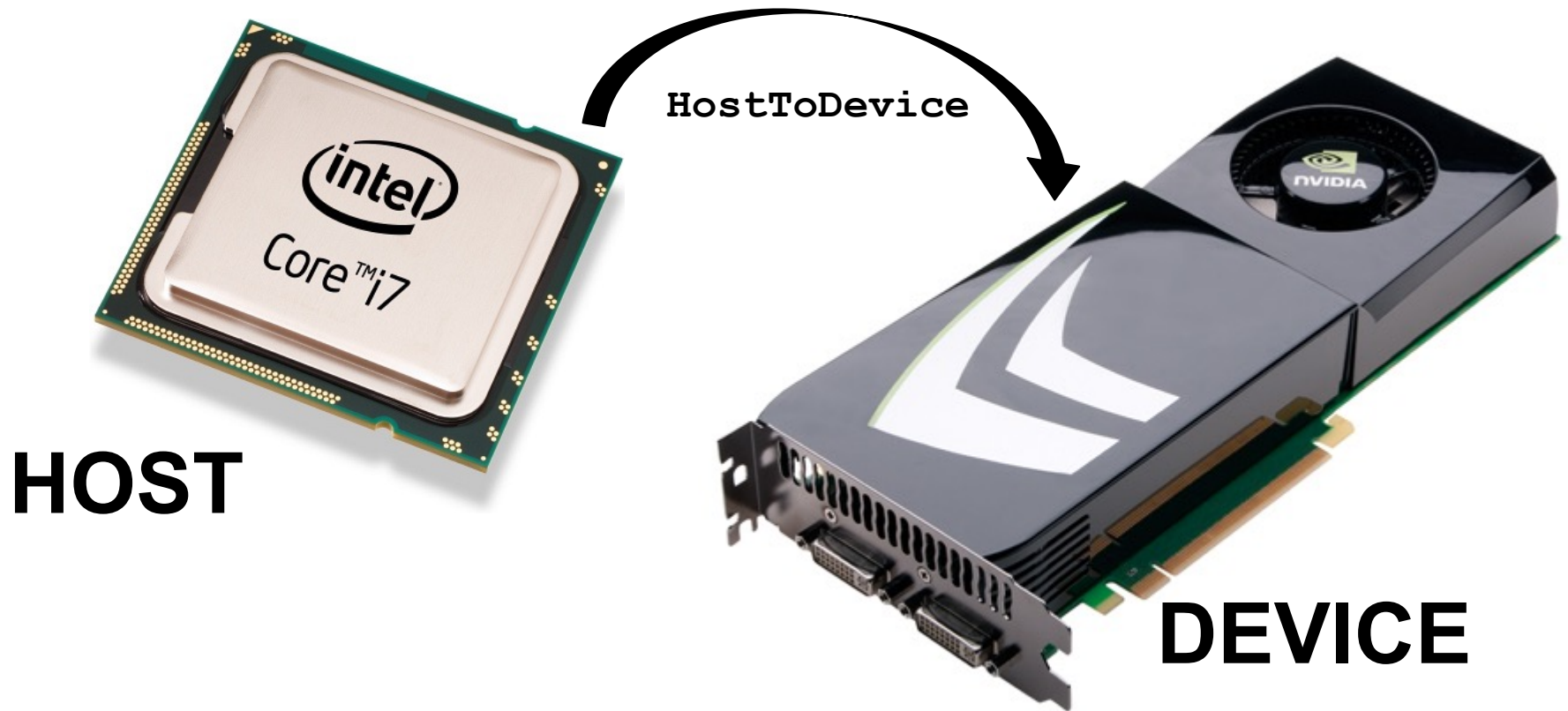
```
    // device_pointer so that it may be used again by another allocation
```

```
    cudaFree( device_pointer );
```

```
    return 0;
```

```
}
```

Data Transfers (1)

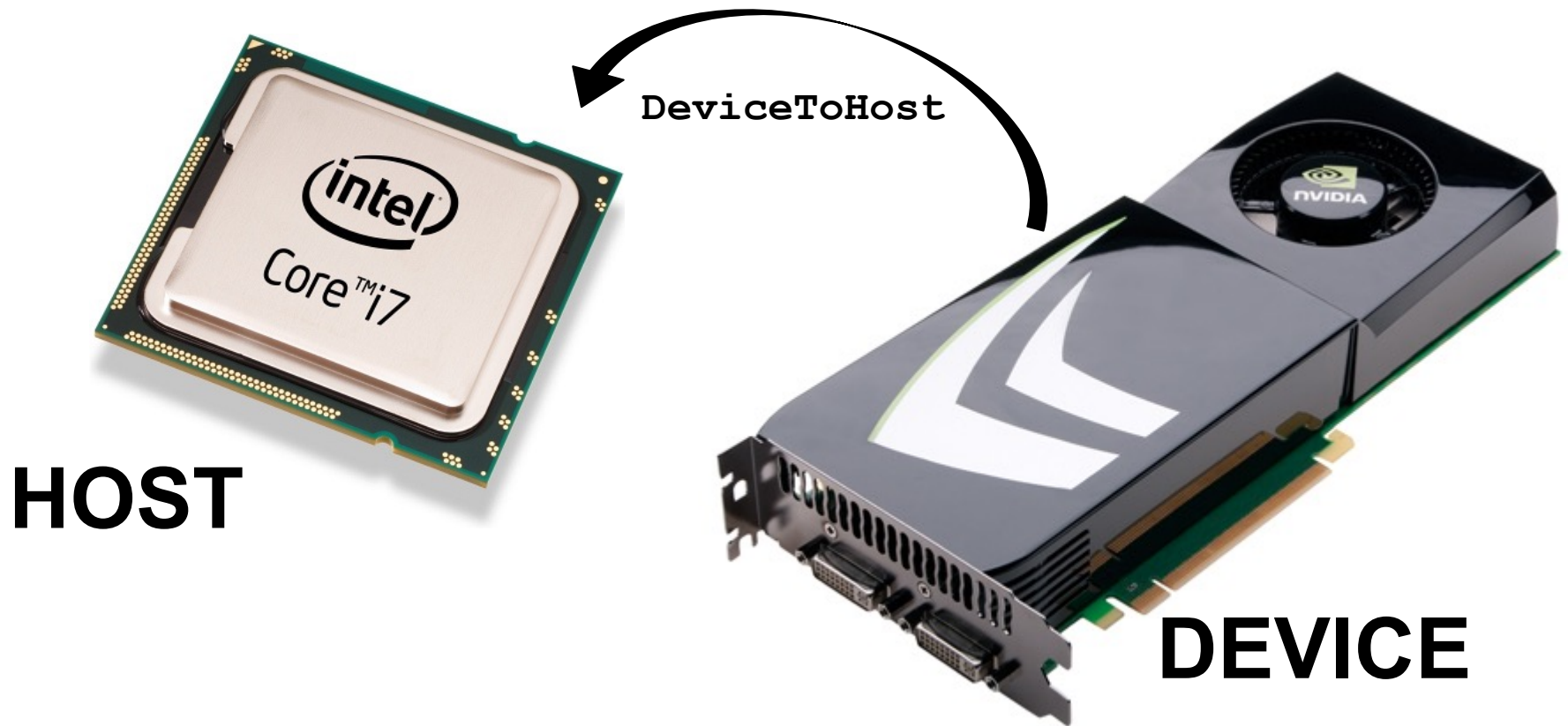


Data Transfer from HOST to DEVICE

```
#define NELEMENTS 16
```


```
int main( void ) {  
    float hostvariable[NELEMENTS];           //float-array on the HOST  
    float *device_pointer;                   //allocated pointer to the DEVICE  
    cudaMalloc( &device_pointer, NELEMENTS*sizeof(float) );  
    cudaMemcpy( device_pointer,              //Pointer to DEVICE memory (dest.)  
                hostvariable,                //Pointer to host memory (source)  
                NELEMENTS*sizeof(float),    //number of bytes to transfer  
                cudaMemcpyHostToDevice );    //direction of transfer  
    cudaFree( device_pointer );  
    return 0;  
}
```

Data Transfers (2)



Data Transfers from DEVICE to HOST

```
#define NELEMENTS 16
```

```
int main( void ) {  
    float hostvariable[NELEMENTS];           //float-array on the HOST  
    float *device_pointer;                   //allocated pointer to the DEVICE  
    cudaMalloc( &device_pointer, NELEMENTS*sizeof(float) );  
    cudaMemcpy( hostvariable,  //Pointer to HOST memory (dest.)  
    device_pointer,                          //Pointer to DEVICE memory (source)  
    NELEMENTS*sizeof(float),                //number of bytes to transfer  
    cudaMemcpyDeviceToHost );               //direction of transfer  
    cudaFree( device_pointer );  
  
    return 0;  
}
```


There is more than one copy operation available

cudaMemcpy

cudaMemcpy2D

cudaMemcpy2DArrayToArray

cudaMemcpy2DAsync

cudaMemcpy2DFromArray

cudaMemcpy2DFromArrayAsync

cudaMemcpy2DToArray

cudaMemcpy2DToArrayAsync

cudaMemcpy3D

cudaMemcpy3DAsync

cudaMemcpy3DPeer

cudaMemcpy3DPeerAsync

cudaMemcpyAsync

cudaMemcpyPeer

cudaMemcpyPeerAsync

cudaMemcpyArrayToArray

cudaMemcpyFromArray

cudaMemcpyFromArrayAsync

cudaMemcpyToArray

cudaMemcpyToArrayAsync

cudaMemcpyFromSymbol

cudaMemcpyFromSymbolAsync

cudaMemcpyToSymbol

cudaMemcpyToSymbolAsync

Possible transfer directions

cudaMemcpyHostToDevice → Transfers data from host to device

cudaMemcpyDeviceToDevice → Transfers data on the device

cudaMemcpyDeviceToHost → Transfers data from device to host

cudaMemcpyHostToHost → Transfers data on the host

cudaMemcpyDefault → Detects by memory address
requires Unified Virtual Addressing (UVA)

Catching errors with `cudaError_t`

CUDA functions return a value of type `cudaError_t`:

```
typedef enum cudaError cudaError_t
```

Values can be for instance:

<code>cudaSuccess</code>	The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see <code>cudaEventQuery()</code> and <code>cudaStreamQuery()</code>).
<code>cudaErrorMemoryAllocation</code>	The API call failed because it was unable to allocate enough memory to perform the requested operation.
<code>cudaErrorInitializationError</code>	The API call failed because the CUDA driver and runtime could not be initialized.
<code>cudaErrorLaunchFailure</code>	An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory.
...	...

Automatic Error Interpretation

In order to interpret `cudaError_t` one can use the following function

```
const char* cudaGetErrorString( cudaError_t error)
```

Example:

```
cudaError_t error = cudaGetLastError();  
printf( "CUDA error: %s\n", cudaGetErrorString( error ) );
```

Catch and Handle Errors by Default

```
#define HANDLE_ERROR( err ) \
    (handleCudaError( err, __FILE__, __LINE__ ) )

static void handleCudaError( cudaError_t err, const char *file, int line ) {

    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ), file, line );
        exit( EXIT_FAILURE );
    }
}
```

- Prepend all CUDA calls with HANDLE_ERROR, e.g.:
HANDLE_ERROR(cudaMalloc(&devicepointer, sizeof(float)));
- Always check kernel launch:
mykernel<<<1,1>>>();
HANDLE_ERROR(cudaGetLastError());

Tool time

How to get this going



CUDA Developer Tools

- IDE
 - Nsight Eclipse / Visual Studio (Linux/Mac / Windows)
 - also for Android: Nsight Tegra (Eclipse / Visual Studio)
 - comes with integrated debugging and analysis tools
- Standalone Performance Tools (CUDA >=10)
 - Nsight Systems - System-wide application algorithm tuning
 - Nsight Compute - Debug/optimize specific CUDA kernel
 - Nsight Graphics - Debug/optimize specific graphics shader
- Classic Tools Set
 - **CUDA compiler `nvcc`**
 - **Memory & Race Checker `cuda-memcheck`**
 - **Built-in profiler `nvprof` (`nsys --stats=true` Pascal and later)**
 - Visual Profiler `nvvp`
 - Debugger `cuda-gdb`

<https://developer.nvidia.com/debugging-solutions>

<https://developer.nvidia.com/gameworks-tools-overview>

<https://developer.nvidia.com/develop4tegra>

Compile and Run a CUDA Program

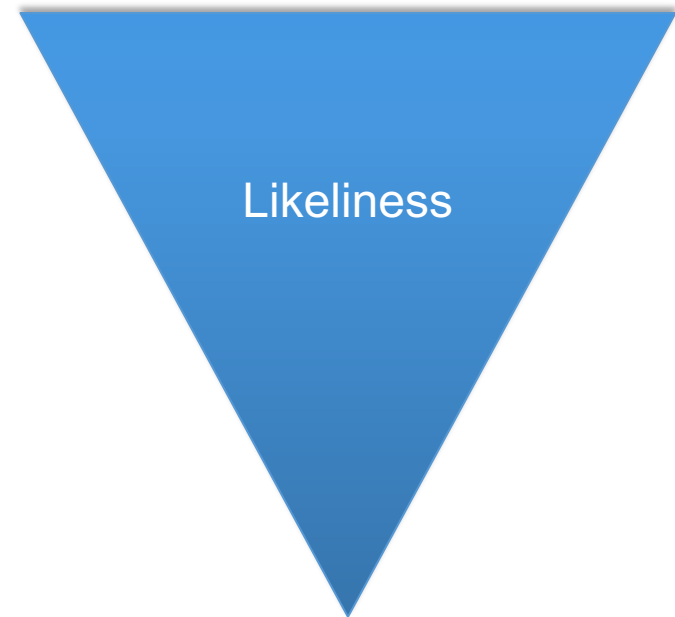
- On remote systems make sure that the CUDA environment is available (usually requires a `module load cuda` or similar)
- Name your CUDA files with the suffix `.cu`
- Compile your program using `nvcc` (e.g. `nvcc myprogram.cu`)
- Execute your program by running `./a.out`
- On remote systems the node you compile on might not feature a GPU, you will have to use a batch system to access node with GPUs
- `nvcc` supports some host compiler flags, otherwise use `-Xcompiler` to forward to host compiler, e.g. `-Xcompiler -fopenmp`
- Debugging flags
 - `-g`: include host debugging symbols
 - `-G`: include device debugging symbols (turns off optimizations!)
 - `-lineinfo`: include line information with symbols (also for profiling)

The Five Basic CUDA Functions

- `cudaMalloc` to allocate memory on the device
- `cudaMemcpy` to transfer data to and from the device
- Kernel invocations
- Handling errors
- `cudaFree` to release allocated memory on the device

The Five Results of CUDA Programming

- **Compiler error**
- **Program crashes**
- **Program produces wrong results**
- **Program runs very slow**
- **Program runs fast and correct**



(Five) Recommendations

- Use comments
- Implement error handling
- Test for correctness
- Use expressive names for functions and variables, e.g. put a “_d” and a “_h” as a suffix to now data locality
- Use building blocks where possible (libraries, function reuse, etc.)

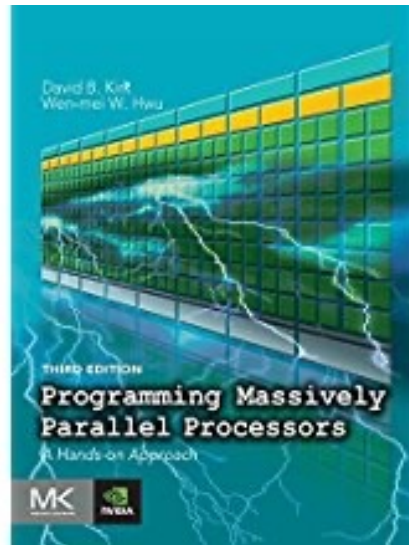
Further Reading

CUDA programming guide

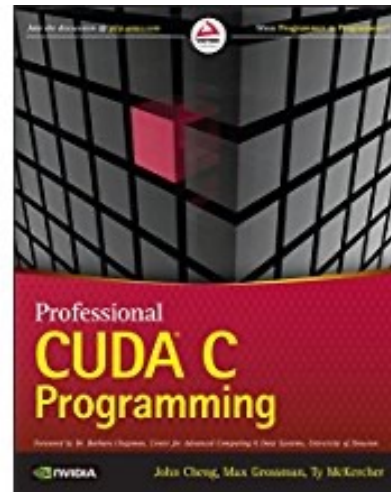
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Books:

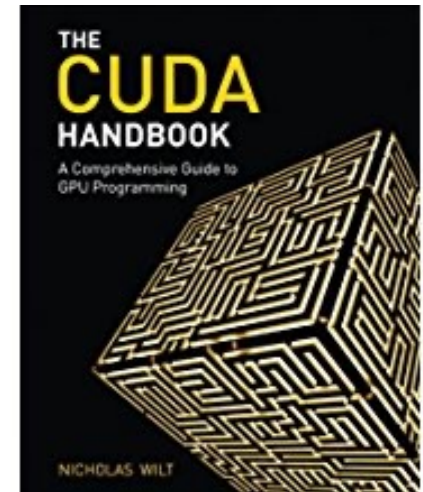
... Advanced ...



**Programming Massively
Parallel Processors
2016
([3. Auflage](#))**



**[Professional
CUDA C Programming](#)
2014**



**[The CUDA Handbook](#)
2013**

Lab 1

→ In the pad



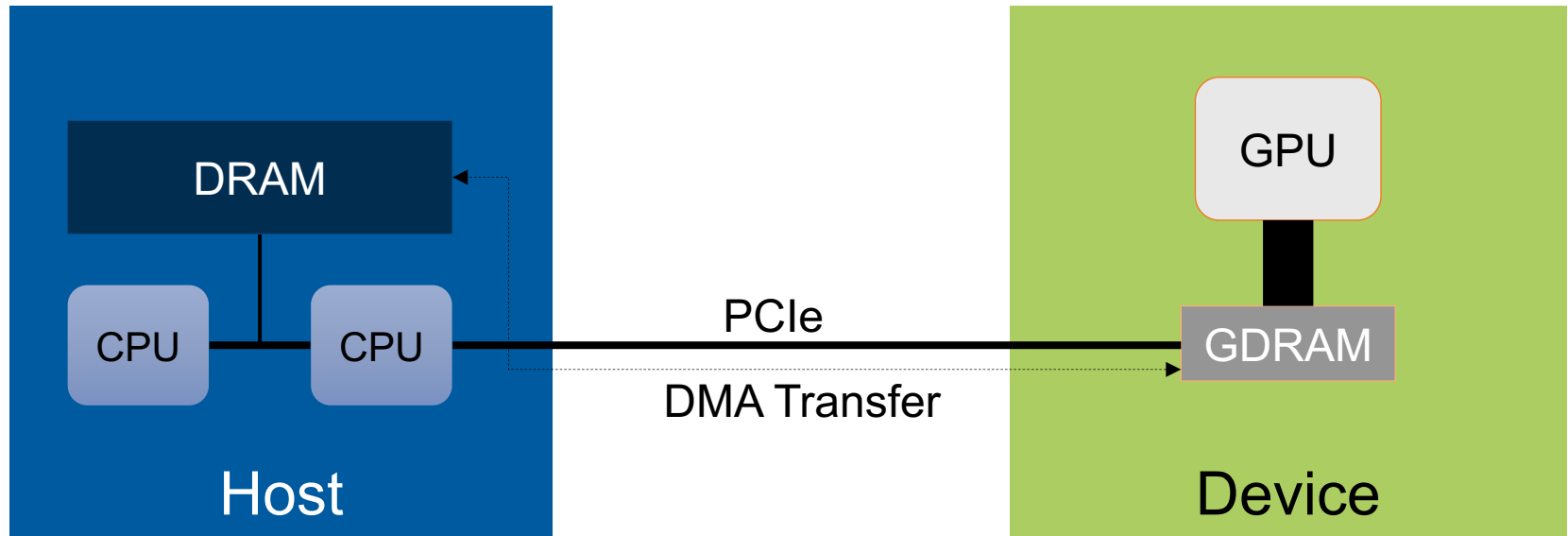
Getting into it...

Kernels, threads and blocks



Recap: GPU System Setup

- CUDA assumes a system with a host and a device with own memory each



Hardware

Software



Kernels

A (device) **kernel** is a piece of a program that will be compiled for being executed on the GPU.

Kernels are invoked by the host on the device

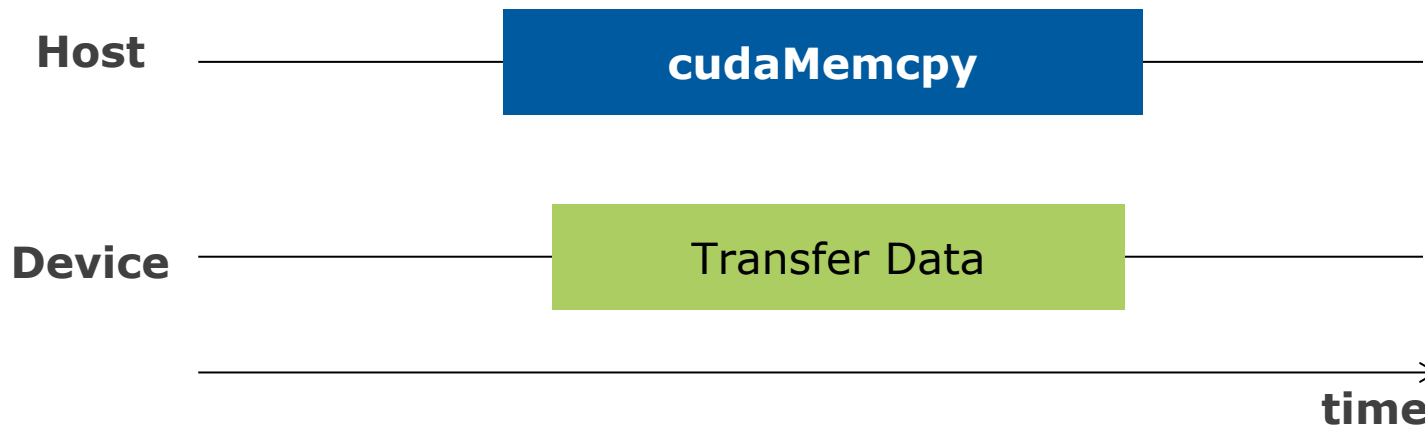
Kernel launches are asynchronous on the host (in CUDA and OpenCL)

Limited C++11/14 support in kernels, see [CUDA8](#) and [CUDA9](#) features

```
int main( void ) {  
    // ...  
        kernel1<<<...,...>>>(...);  
        kernel2<<<...,...>>>(...);  
        kernel3<<<...,...>>>(...);  
    // ...  
    return 0;  
}
```

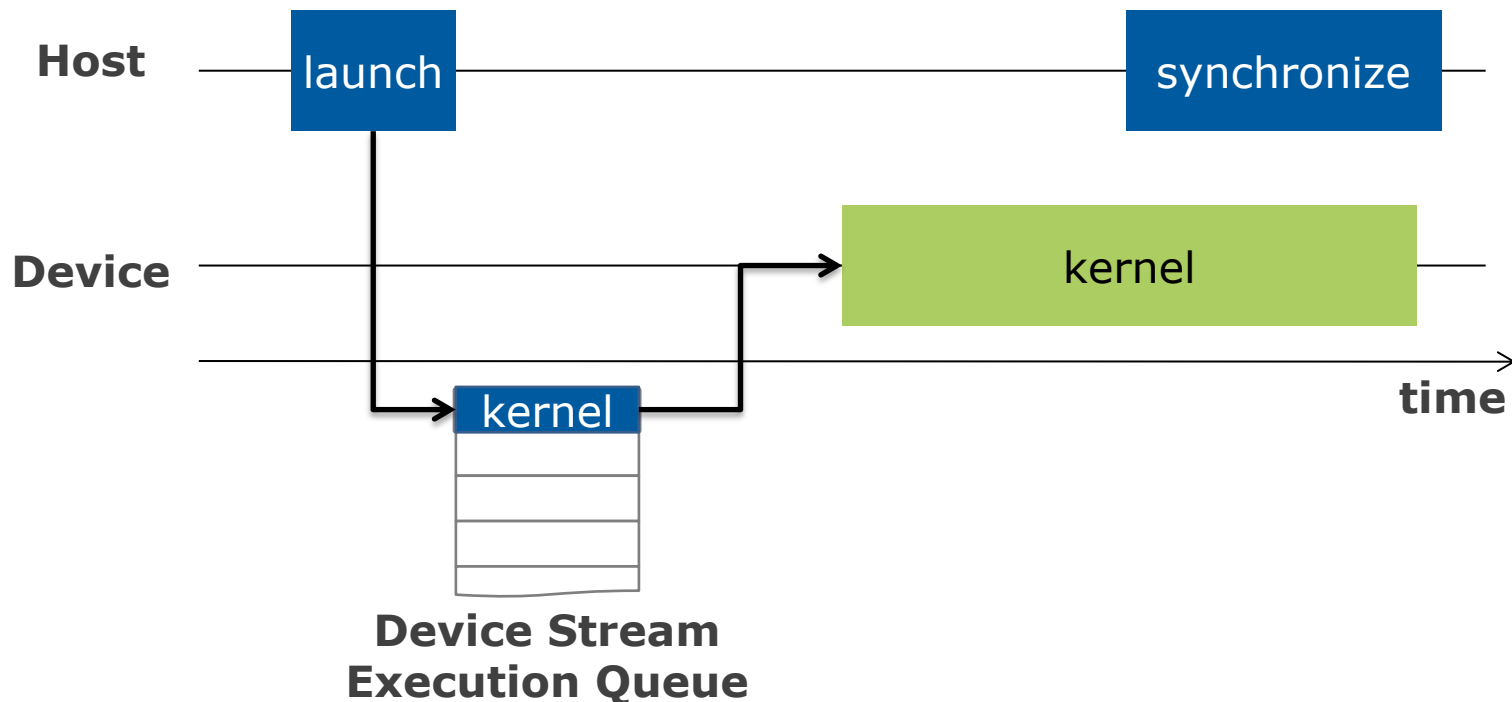
(Host-)Synchronous Execution

Synchronous operations wait until the device activity is completed



Asynchronous Execution

- Asynchronous device activities are launched by the CPU without blocking its execution
- The host needs to request the execution status of the device to explicitly synchronize with it



Kernel Declaration

- Kernels are declared like “normal” functions of return type void and prepended by the key word `__global__`

Example:

```
__global__ void do_nothing(float *data){ ... }
```

- Since kernels are launched asynchronously they cannot return a value
- Kernels can invoke device functions

```
__device__ float help_do_nothing() { ... }
```

- Kernels can run concurrently

```
kernel1<<<...,>>>(...); // generates many parallel threads
```

```
kernel2<<<...,>>>(...); // generates many parallel threads
```

```
kernel3<<<...,>>>(...); // generates many parallel threads
```

Where is the parallelism?

- A kernel function is the code to be executed on the device side
- A kernel defines the computation & data access of a single thread
- Many CUDA threads perform the same computation in parallel
- CUDA uses a relaxed, more expressive SIMD programming model:
=> SIMT (Single Instruction, Multiple Threads)
- SIMT is hybrid of SIMD and SMT

Efficiency ←→ *Flexibility*

SIMD SIMT SMT

Single Instruction, Multiple Data

Simultaneous Multi-Threading

- SIMT allows multiple register sets, addresses and flow paths
- SIMT uses scalar spelling, ie.:

```
int idx = /* compute global thread id */;  
a[idx] = b[idx]+c[idx];
```

Where is the parallelism?

```
__global__ void add(float *a, float *b, float *c) {  
    int i = /* compute global thread id */;  
    a[i]=b[i]+c[i]; //no loop!  
} ...  
add<<<...,...>>>( a_dev, b_dev, c_dev );
```

Thread 0

$a[0] = b[0] + c[0];$

Thread 1

$a[1] = b[1] + c[1];$

Thread i

$a[i] = b[i] + c[i];$

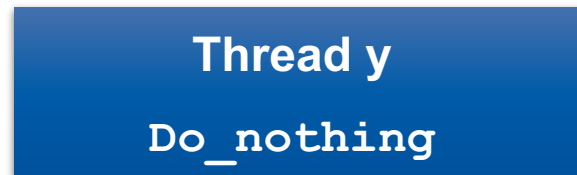
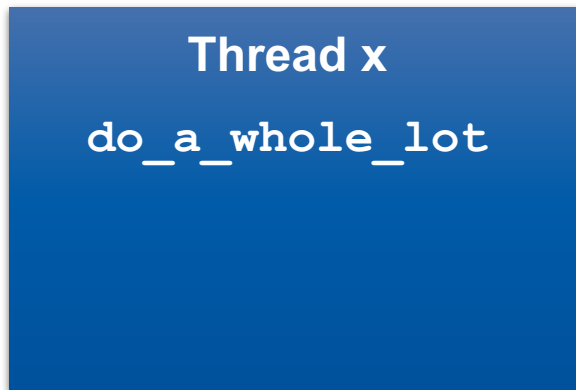
Thread Divergence

When threads do different things, the runtime of the threads can vary.

```
__global__ void diverge( void *data ) {  
    if ( data[mythread] > random_number )  
        do_a_whole_lot();  
    else  
        do_nothing();  
}
```

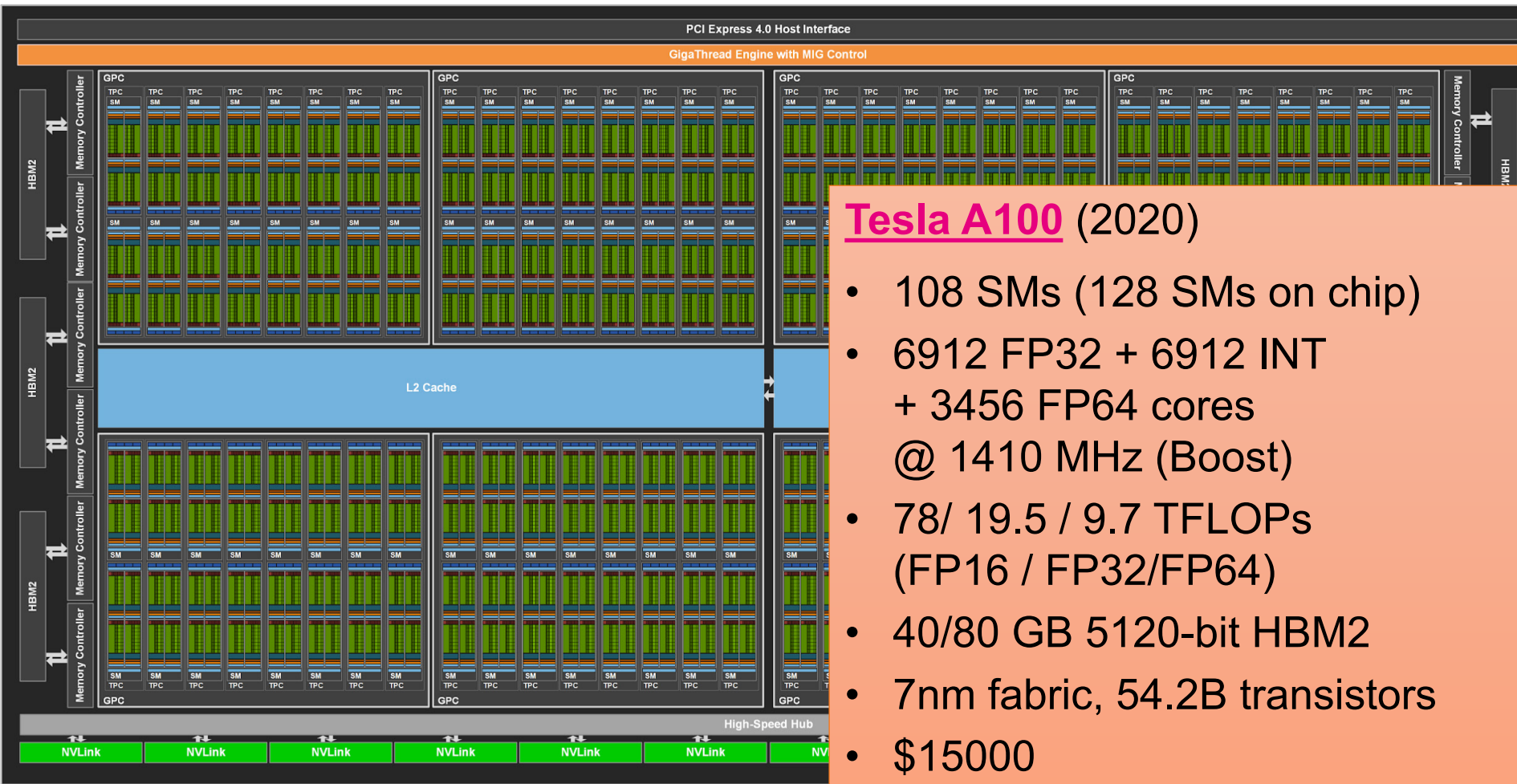
If thread x and y are in a SIMT group,
different execution paths become
serialized as well

runtime
↓



The kernel `diverge` runs
until the last thread is
finished.

Nvidia Ampere Architecture



Tesla A100 (2020)

- 108 SMs (128 SMs on chip)
- 6912 FP32 + 6912 INT + 3456 FP64 cores @ 1410 MHz (Boost)
- 78/ 19.5 / 9.7 TFLOPs (FP16 / FP32/FP64)
- 40/80 GB 5120-bit HBM2
- 7nm fabric, 54.2B transistors
- \$15000

<https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>

Streaming Multiprocessor (Pascal)

With CUDA there is a two-level thread hierarchy decomposed into blocks of threads and grids of blocks.

Threads are grouped in **blocks** which are executed on one Streaming Multiprocessor (SM).

They can cooperate using a (small) shared memory. Threads from different blocks cannot cooperate directly.



<https://devblogs.nvidia.com/parallelforall/inside-pascal/>

Thread Blocks

- Arrangement of threads is called thread block
- Threads are executed in SIMD fashion in groups of 32 threads, which are called warps
 - warp size may change in the future
- Order of warp execution is not fixed and can vary
- Synchronization by `__syncthreads()`

//integers nx, ny, nz describe the block in 3D

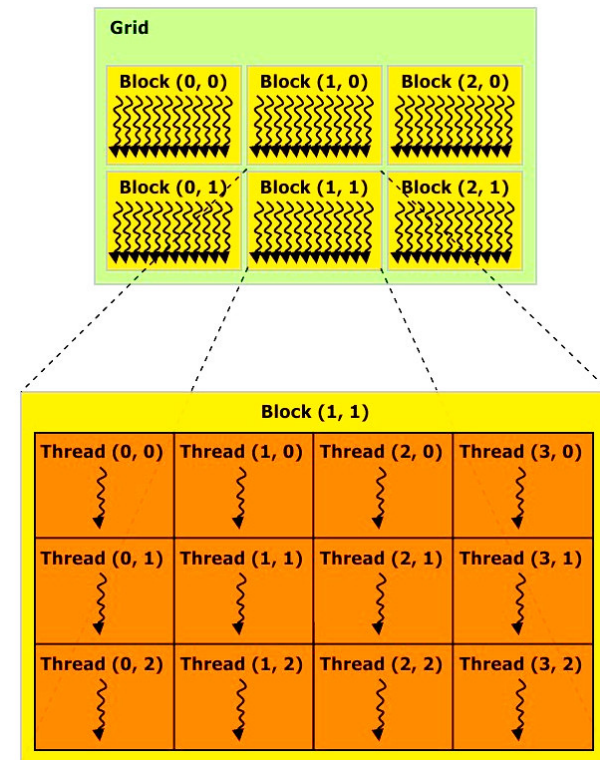
```
dim3 block(nx, ny, nz);
```

```
//creates nx*ny*nz threads in 1 block
```

```
kernel<<<1,block>>>(...);
```

```
// block size can also be a number for 1D
```

```
kernel<<<1,512>>>(...);
```



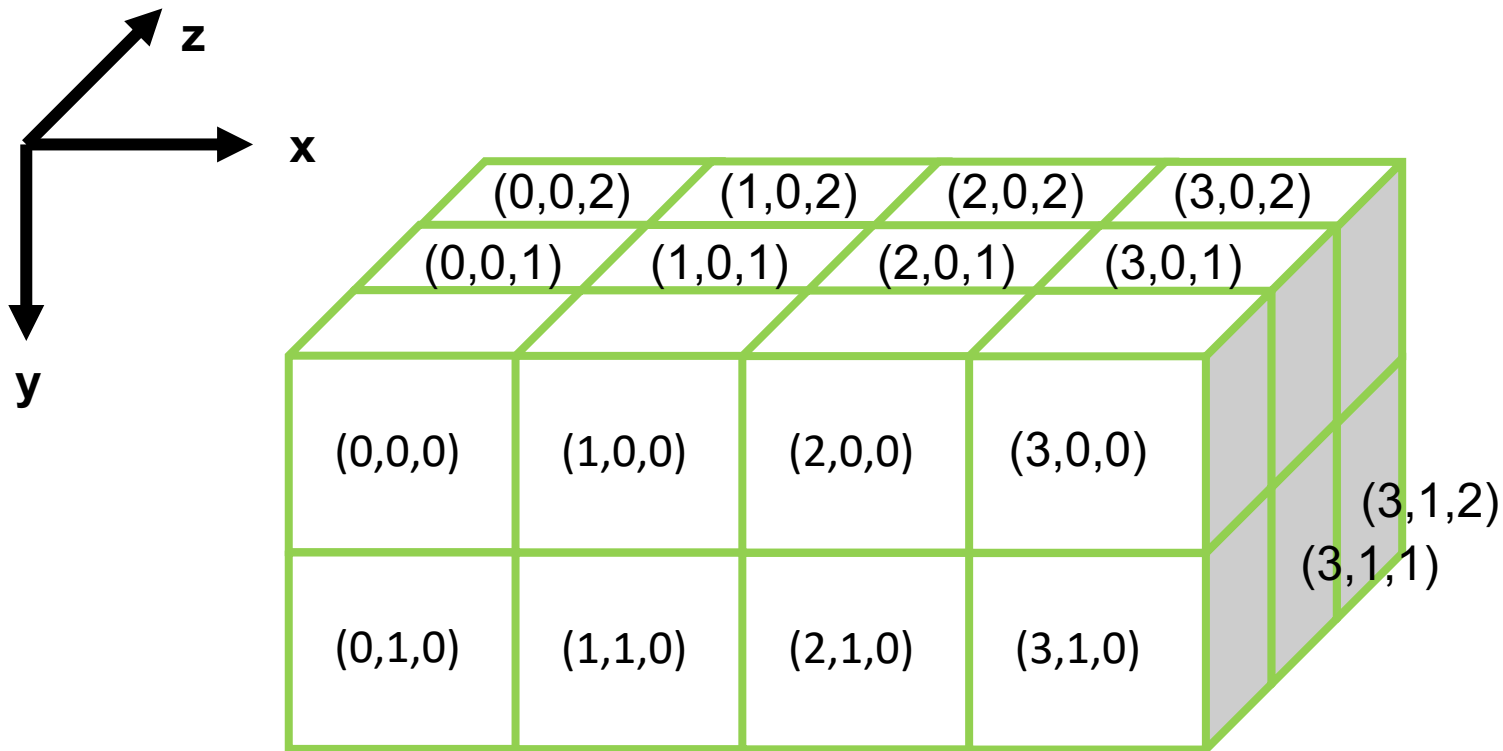
The dim3 Data Structure

```
struct dim3
{
    unsigned int x, y, z;
};
```

Create with just assigning a variable, unused dimensions are set to 1

Threads in a Block

```
dim3 block(4,2,3);  
kernel<<<1,block>>>(...);
```



New Threads on the Block

```
__global__ void kernel( void *data ) {  
    int tidx = threadIdx.x; //position of threads within block x  
    int tidy = threadIdx.y; //position of threads within block y  
    int tidz = threadIdx.z; //position of threads within block z  
}
```

```
dim3 block(4,2,3);  
kernel<<<1,block>>>(data);
```

Calls a kernel with $24 = 4 \cdot 2 \cdot 3$ threads

```
(threadIdx.x, threadIdx.y, threadIdx.z) :  
(0,0,0), (1,0,0), (2,0,0), (3,0,0), (0,1,0), (1,1,0), (2,1,0), (3,1,0),  
(0,0,1), (1,0,1), (2,0,1), (3,0,1), (0,1,1), (1,1,1), (2,1,1), (3,1,1),  
(0,0,2), (1,0,2), (2,0,2), (3,0,2), (0,1,2), (1,1,2), (2,1,2), (3,1,2)
```

Block Size Restrictions

- Total number of threads in a block is the product of the number of threads in each dimension
- Total number of threads and threads per dimension have limits

CUDA Compute Capability	2.x	3.x	5.x	6.x	7.0	8.0
<i>Micro Architecture</i>	<i>Fermi*</i>	<i>Kepler⁺</i>	<i>Maxwell</i>	<i>Pascal</i>	<i>Volta</i>	<i>Ampere</i>
Max. block size in x,y	1024					
Max. block size in z	64					
Max. threads per block	1024					

Comprehensive tables of device properties: <https://en.wikipedia.org/wiki/CUDA>

* Fermi is deprecated as of CUDA8 and without compiler support as of CUDA9

+ Kepler is deprecated as of CUDA10 and without compiler support as of CUDA11

Multiple Thread Blocks (a.k.a. Grid of Blocks)

- Arrangement of blocks is called grid
- Order of block execution is not fixed
- Multiple blocks can reside on one multiprocessor (as long as resources are available)
- No synchronization between blocks
- Blocks are distributed over all multiprocessors

//integers mx, my, mz describe the grid in 3D

```
dim3 grid(mx, my, mz);
```

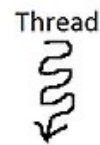
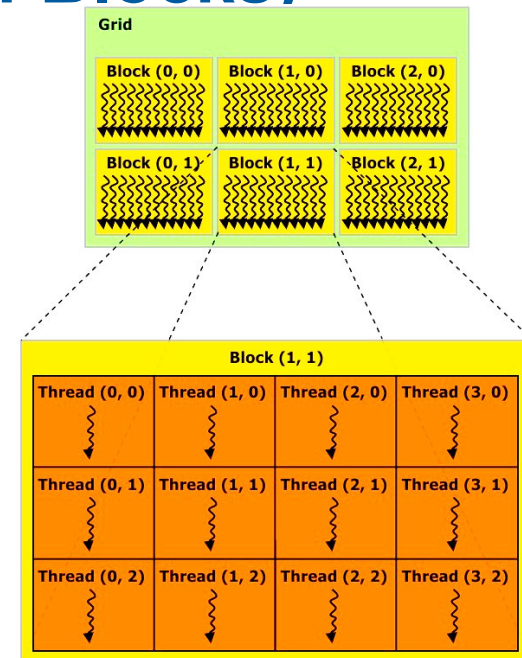
```
dim3 block(512);
```

//creates mx*my*mz blocks

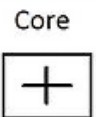
```
kernel<<<grid,block>>>(...);
```

// grid size can also be a number

```
kernel<<<1024,512>>>(...);
```



Executed by

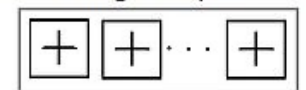


Thread Block

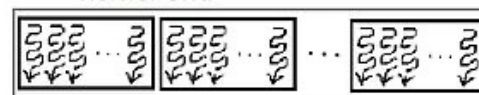


Executed by

Streaming Multiprocessor

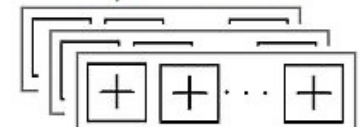


Kernel Grid



Executed by

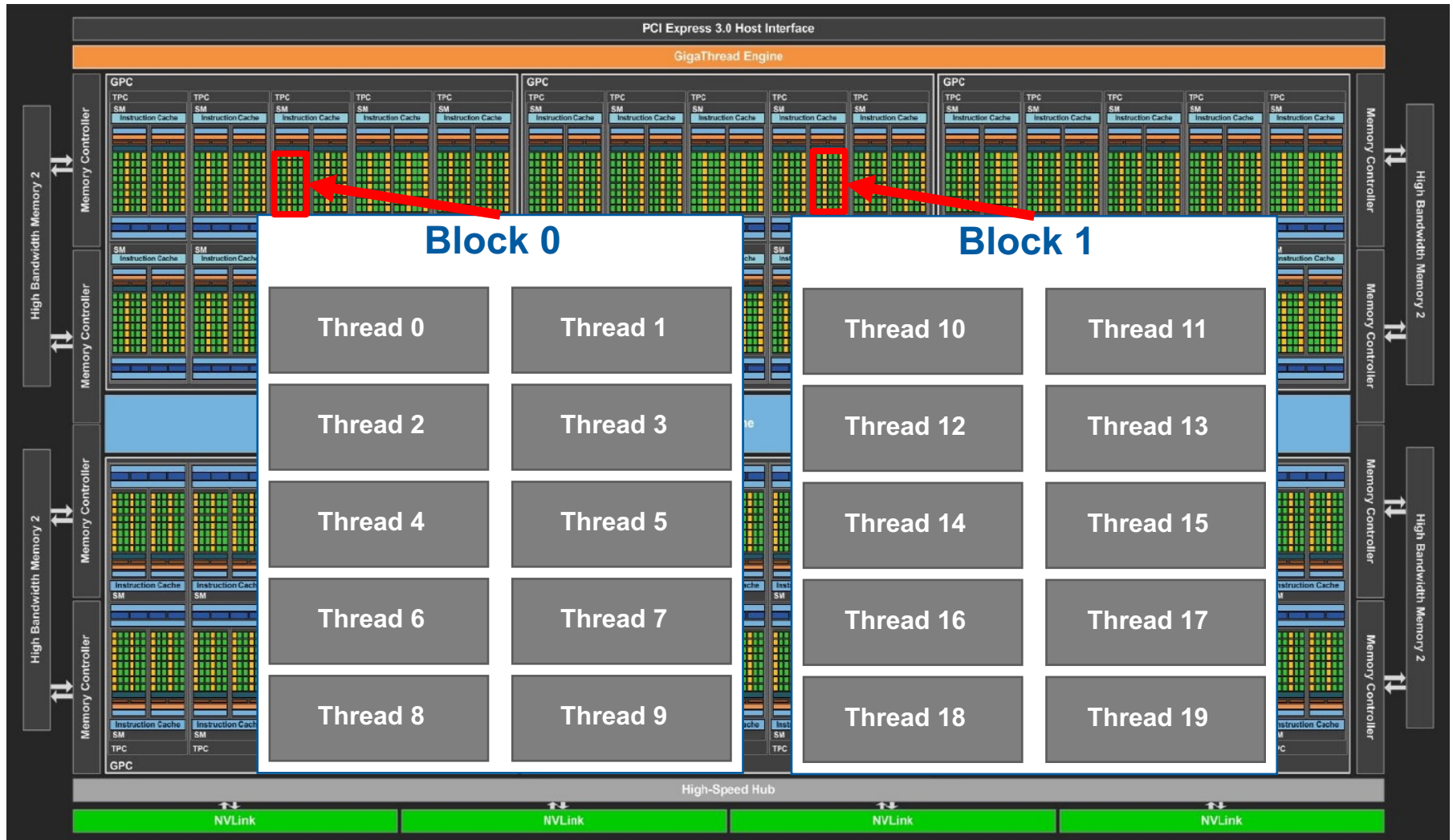
Complete GPU Unit



concept 1

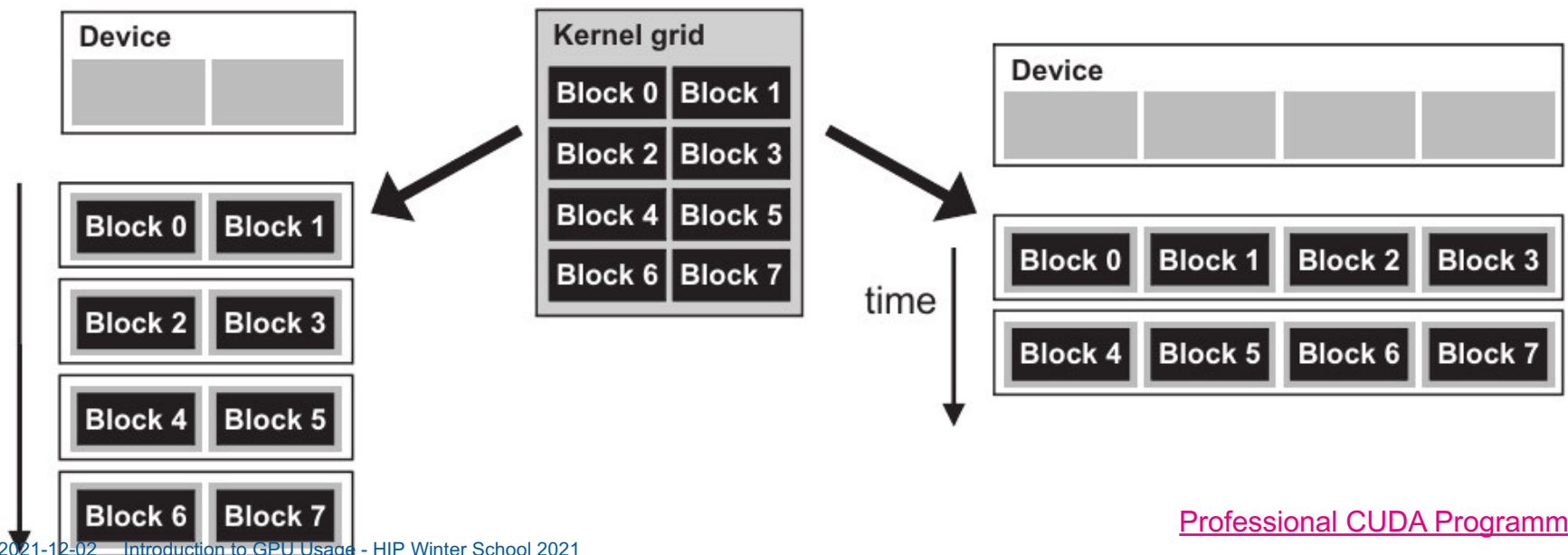
CC 4.0 by Atshardul

Thread + Block Mapping



Transparent Scalability

- Each block can execute in any order relative to others
- Threads are assigned to SMs in block granularity
 - SM maintains thread/block idx's
 - SM manages/schedules thread execution
 - SM implements zero-overhead warp scheduling
- Hardware is free to assign blocks to any processors at any time
- A kernel scales to any number of parallel processors



Grid Size Restrictions

- Total number of blocks in a grid is the product of the number of blocks in each dimension
- Total number of blocks and blocks per dimension have limits
- Gives a kernel launch error if launch configuration is invalid (check by `cudaGetLastError`)

CUDA Compute Capability	2.x	3.x	5.x	6.x	7.0	8.0
<i>Micro Architecture</i>	<i>Fermi*</i>	<i>Kepler</i>	<i>Maxwell</i>	<i>Pascal</i>	<i>Volta</i>	<i>Ampere</i>
Max. grid size in x	2 ³¹ -1					
Max. grid size in y or z	65535					
Max. resident blocks per SM	16		32			
Max. resident threads per SM	2048					

IDs with Blocks and Grids

grid

blocks

blockIdx.x 0	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 1	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 2	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
...	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 65534	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511

Global Thread ID

- Threads need to decide on which data they need to work
- Requires ID and size queries

Type	ID	Size
Thread	threadIdx	-
Block	blockIdx	blockDim
Grid	-	gridDim

All variables are available in all three dimensions.

Examples:

1D grid of 1D block:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

1D grid of 2D block:

```
int idx = blockIdx.x * blockDim.x * blockDim.y  
+ threadIdx.y * blockDim.x + threadIdx.x;
```

Global Thread ID (Example 1D Block)

grid	blocks				
	threadIdx.x	threadIdx.x	threadIdx.x	...	threadIdx.x
	0	1	2	...	511
blockIdx.x 0	tid=0	tid=1	tid=2	...	tid=511
blockIdx.x 1	tid=512	tid=513	tid=514	...	tid=1023
blockIdx.x 2	tid=1024	tid=1025	tid=1026	...	tid=1535
blockIdx.x 3	tid=1536	tid=1537	tid=1538	...	tid=2047

$$tid = threadIdx.x + blockIdx.x * blockDim.x;$$

Global Thread ID (3D Block)

```
__global__ void settozero( float *elem ) {  
    int tid = threadIdx.x          +  
              threadIdx.y * blockDim.x +  
              threadIdx.z * blockDim.x * blockDim.y;  
    elem[tid] = 0.0f;  
}  
  
int main( int argc, char *argv[] ) {  
...  
    dim3 block3d(32, 2, 2); // 32x2x2 thread block  
    dim3 grid1d(16); // 1D grid of 16 3D thread blocks  
    settozero<<<grid1d, block3d>>>(elem_d);  
...  
}
```

Side note: flat index (tid=...) may cause non-coalesced memory access
(we will come back to it later on)

Monolithic Kernels

Rule of thumb: every thread creates one output element
(assumes that there are enough threads to cover the entire array)

Example: Single Precision $A * X + Y$ (SAXPY)

```
__global__ void saxpy(int N, float a, float *x, float *y) {  
    // who am I?  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
  
    // if I am inside the vector, work on my data  
    if ( i < N ) {  
        y[i] = a * x[i] + y[i];  
    }  
}  
  
// Perform SAXPY on 1M elements  
saxpy<<<4096,256>>>(1000000, 2.0, x_d, y_d);
```


Summary

- Kernel launch configuration requires grid and block dimension (1-3D)
- Kernel launches are always asynchronous
- Kernel functions have `__global__` attribute and returns `void`
- Kernels are processed in SIMT and SPMD fashion
- Each 32 threads represent a SIMD group called warp (may change)
- Threads/Warps are separated into thread blocks
- Set of thread blocks is called a grid
- Kernel launch must be checked by `cudaGetLastError`

Lab 2

→ In the pad



Location, location, location...

It's all about memory



Device and Host Memory

Device

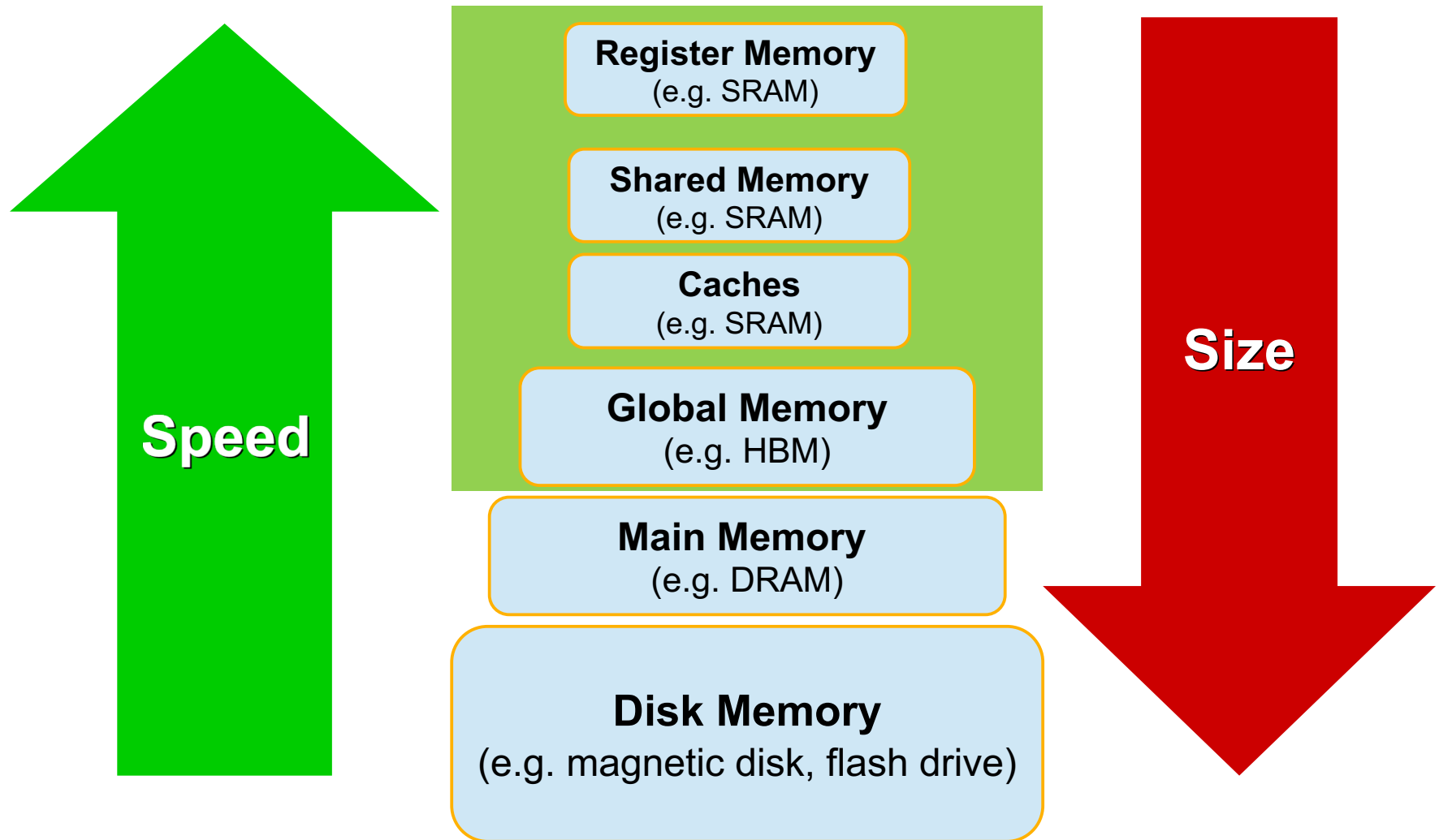


Host





Memory Hierarchy



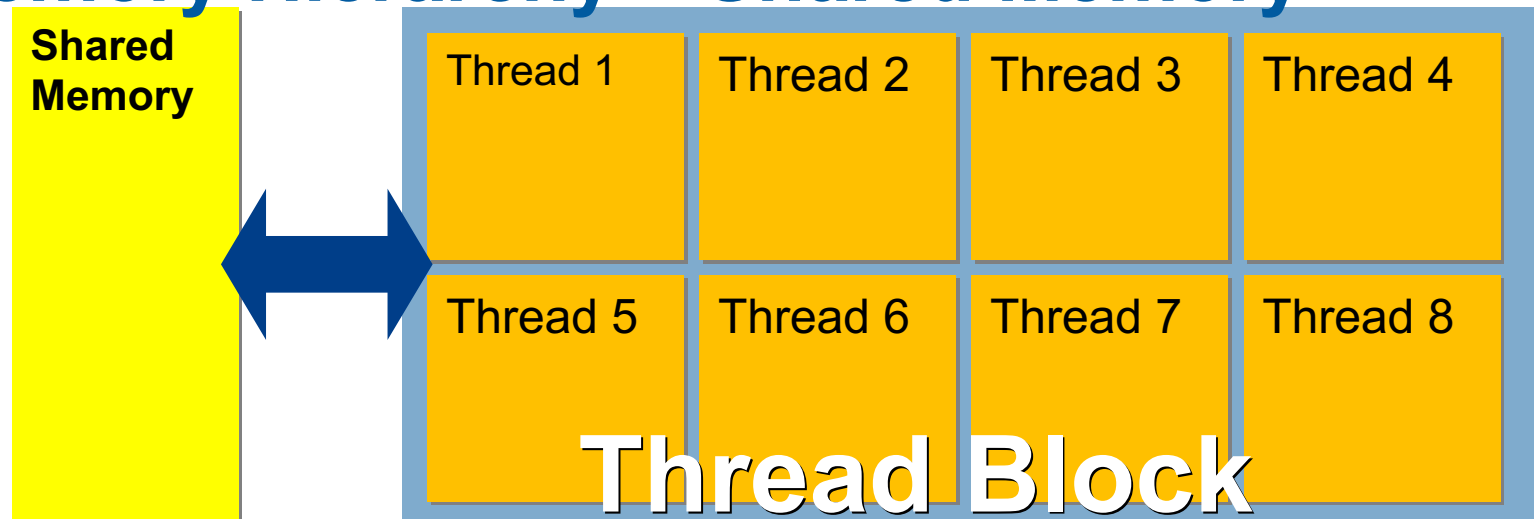
Memory Hierarchy - Register



Number of 32-bit registers per multiprocessor: 64k (except CC 3.7: 128k)

<i>What?</i>	Register is the memory for the ALUs, so it is on-chip and fast!
<i>How?</i>	Declare a variable, e.g. <code>int counter;</code>
<i>Who?</i>	A register is assigned to a single thread only
<i>Scope?</i>	Lifetime of a thread
<i>Access?</i>	Read+Write, thread-private
<i>Problems?</i>	Affects occupancy and if a thread wants too many registers, they become spilled out to local memory (slow)

Memory Hierarchy – Shared Memory



Maximum Shared Memory per Thread Block: 48 KB

Maximum Shared Memory per Multiprocessor: {48,64,96,112} KB

What?

Fast as registers, on-chip memory, shared by threads

How?

```
__shared__ float commondata[threadsPerBlock];
```

```
__shared__ float commondata*;
```

Who?

All threads within a thread block

Scope?

Lifetime of the thread block

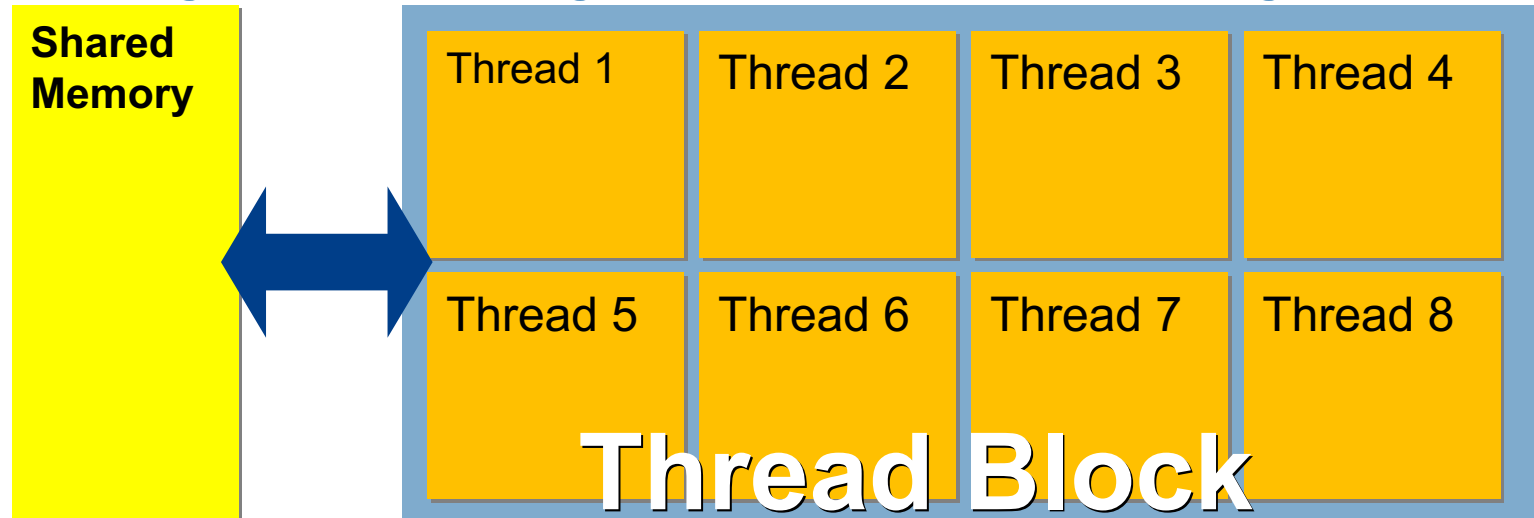
Access?

Read+Write

Problems?

Affects occupancy, bank-conflict impacts performance,
some GPUs have bank width configurations (4byte, 8byte)

Memory Hierarchy – Shared Memory



Maximum Shared Memory per Thread Block: 48 KB

Maximum Shared Memory per Multiprocessor: {48,64,96,112} KB

- Shared memory is divided into banks to achieve high bandwidth
- It services as many simultaneous accesses as it has banks
- Shared memory helps to reduce multiple loads of device data
- ... conversion of data layout
- ... communication within thread block

Memory Hierarchy – Shared Memory

Creates an Integer Array `a` with 256 elements in shared memory for each thread block

- **Static** shared memory allocation:

```
__global__ void foo() {  
    __shared__ int a[256];  
}
```

- **Dynamic** shared memory allocation:

```
}  
__global__ void foo() {  
    extern __shared__ int a[];  
    ...  
}
```

Kernel launch configures shared memory size for each thread block:

```
foo<<< NBLOCKS, NTHREADS, NTHREADS*sizeof(int)>>>();
```

Memory Hierarchy – Shared Memory

Fill the shared memory and sync the threads:

```
__global__ void foo(int *g_data){
    __shared__ int a[256];
    int idx=... //get global index
    ...
    // every thread copies one b to a
    a[threadIdx.x] = g_data[idx];

    // wait for all threads in a block
    __syncthreads();
    ...
}
```

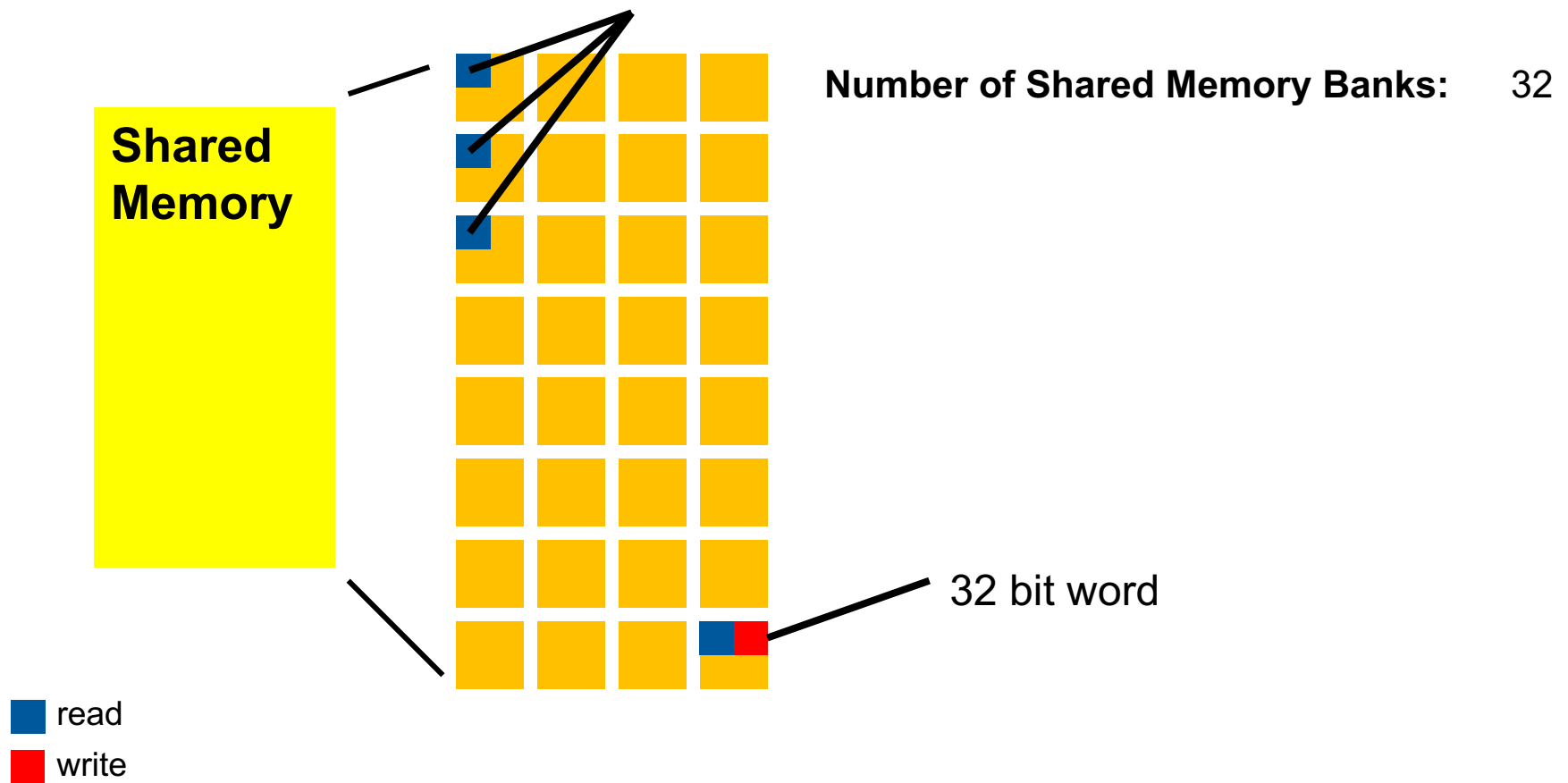
Memory Hierarchy – Shared Memory

Typical workflow:

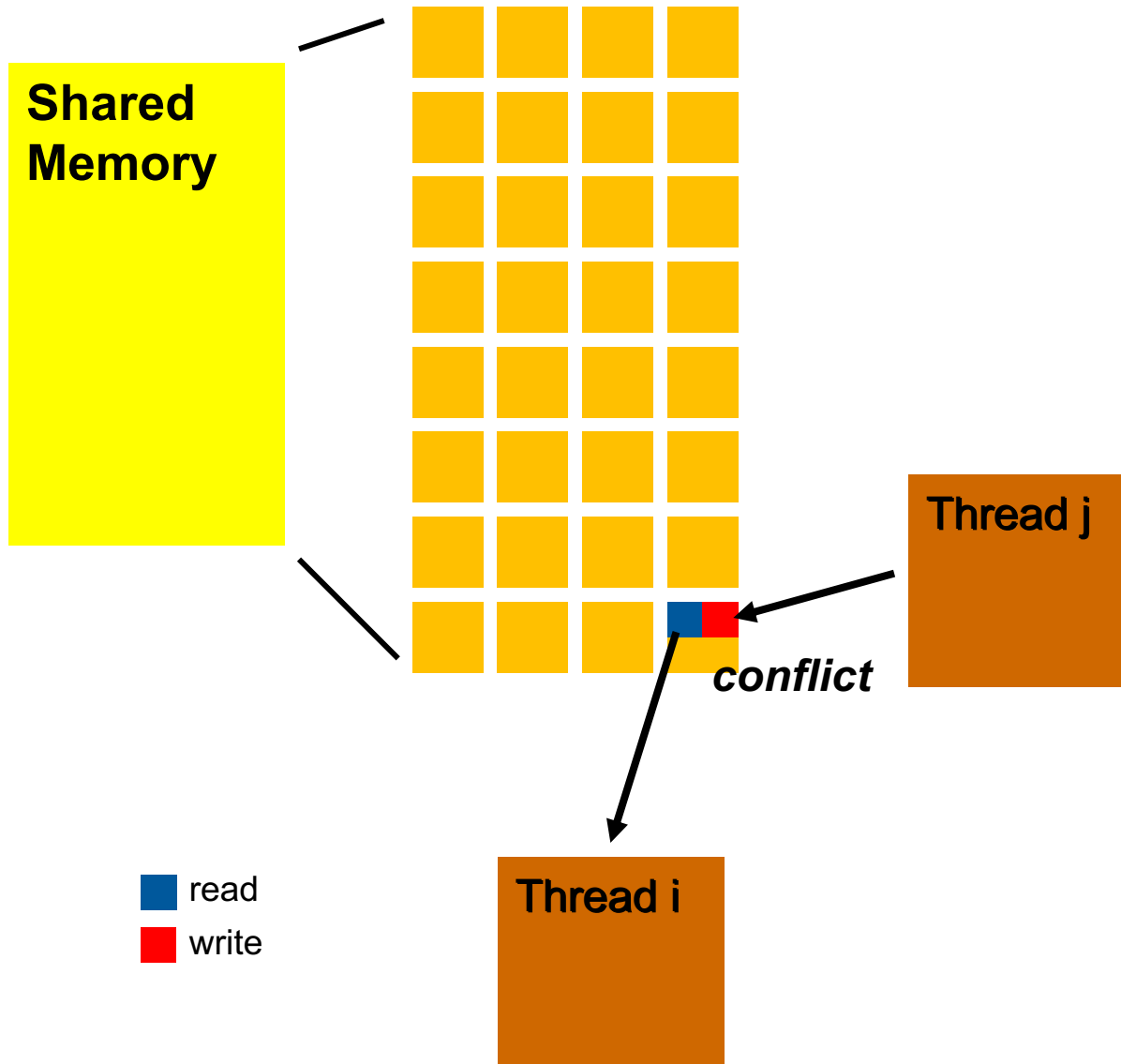
```
__global__ void foo(int *g_data){  
    __shared__ int a[256];  
    int idx=... //get global index  
    ...  
    for(...) {  
        // copy reused data to shared memory  
        // sync  
        for() { /*compute*/ }  
        // sync  
        // write back to global memory  
    }  
}
```

Shared Memory Banks

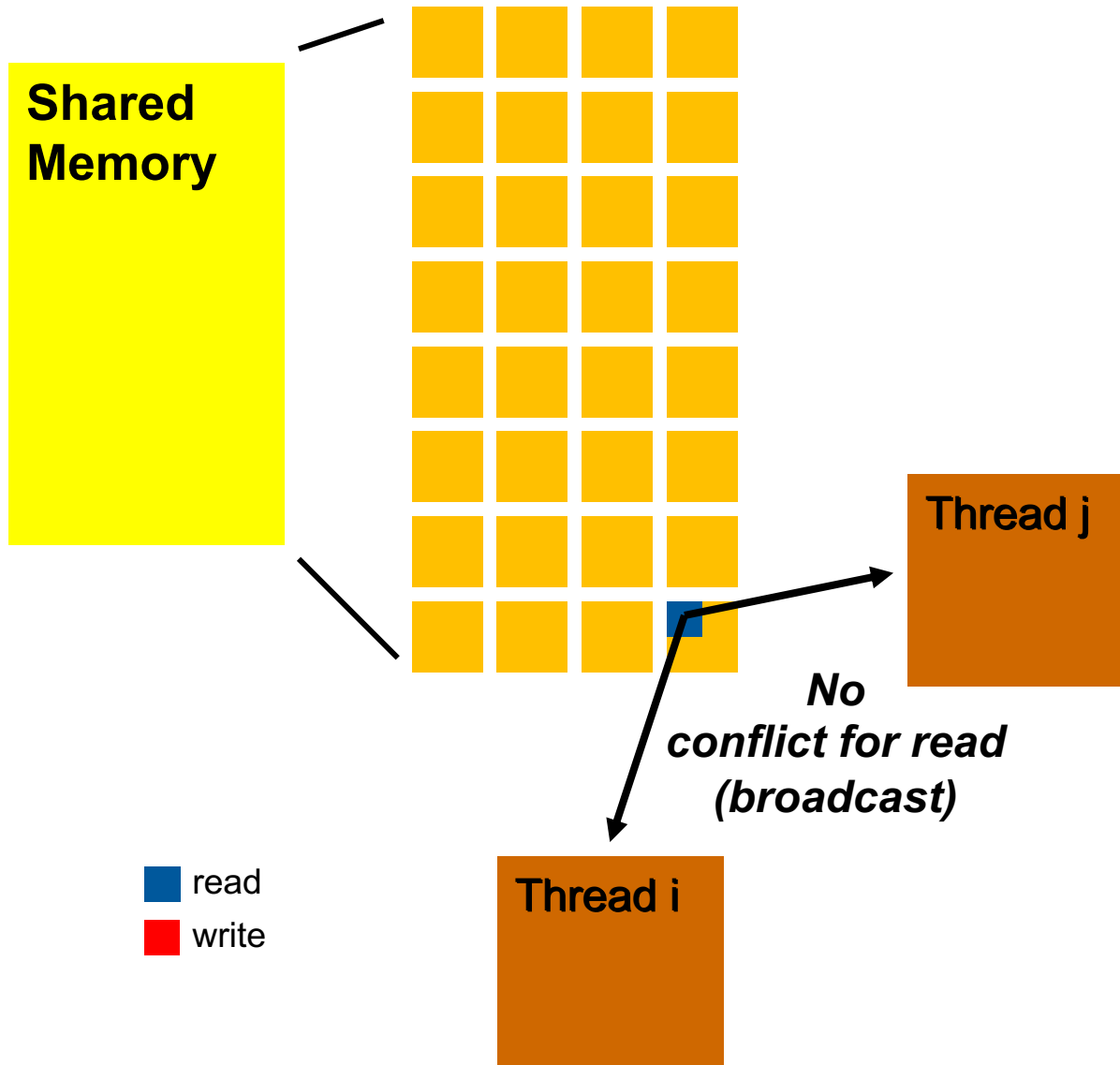
- Equally sized memory modules, which can be accessed simultaneously
- Consecutive 32 bit words become stored consecutively (*interleaving*)



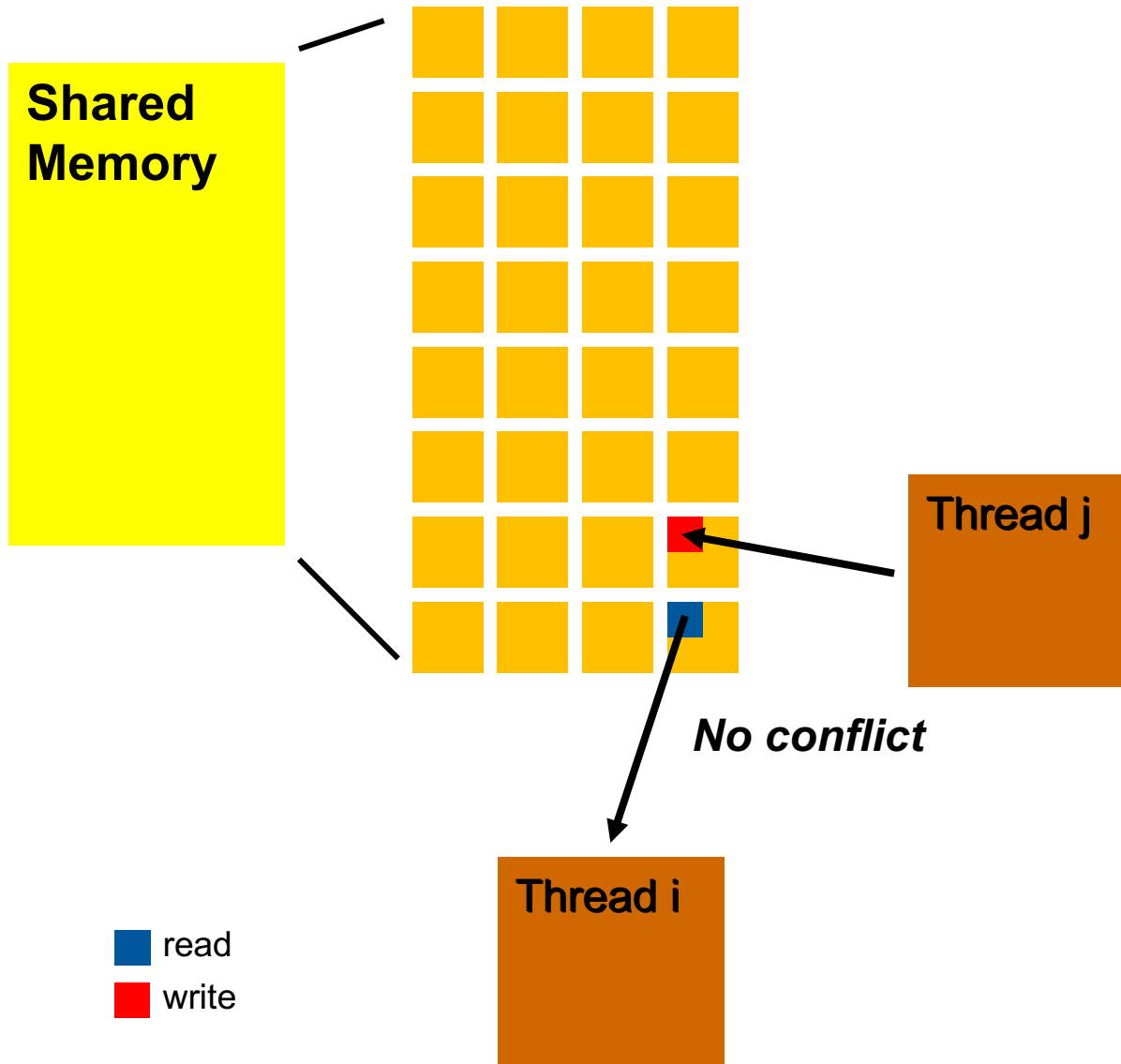
Shared Memory Bank Conflict



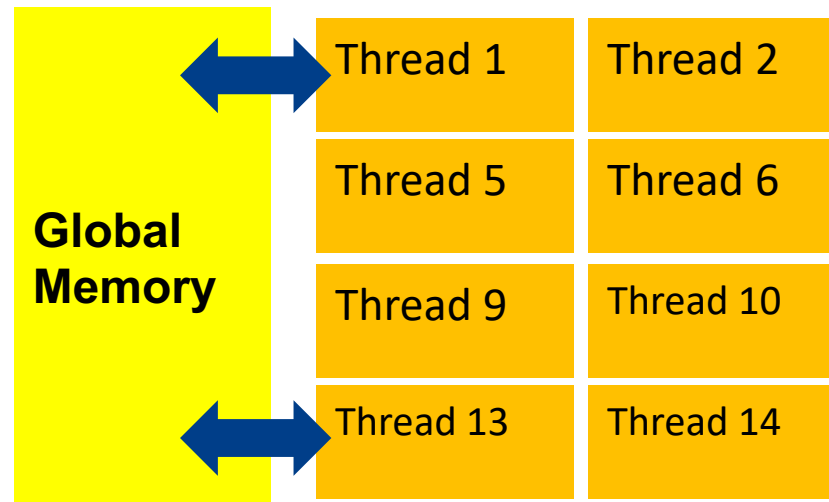
Shared Memory Bank Conflict



Shared Memory Bank Conflict



Memory Hierarchy – Global Memory



Size typically ranges from 4 up to 32 GB

What?

How?

Who?

Scope?

Access?

Problems?

memory to communicate data between multiprocessors and devices

```
__global__ float comdata[maxMem];
```

All threads on the device, host, other GPUs (UVA)

Lifetime of application

R+W

Bad alignment of data can slow down even further

Memory Hierarchy – Global Memory Alignment

Memory (4 x 128 Byte)



Requested memory (1 x 128 Byte not aligned)



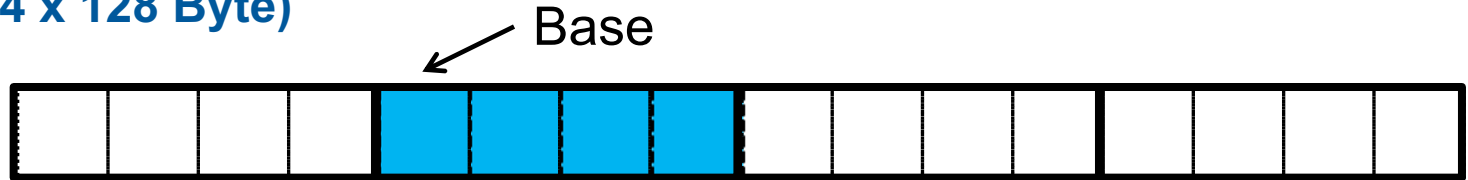
Transferred memory (2 x 128 Byte)



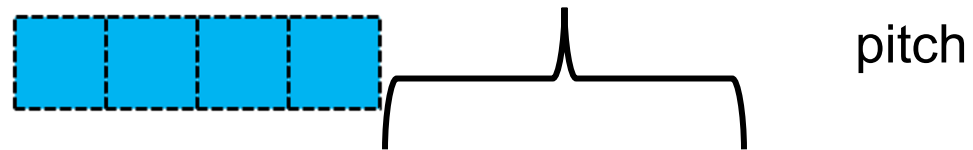
50% wasted

Memory Hierarchy – Global Memory Alignment

Memory (4 x 128 Byte)



Requested memory (1 x 128 Byte aligned)



Transferred memory (1 x 128 Byte)



2D data access

$$T^* \text{ elem} = (T^*) ((\text{char}^*) \text{Base} + \text{Row} * \text{pitch}) + \text{Column}$$

Memory Hierarchy – Global Memory Alignment

Create aligned memory by simply using `cudaMalloc*`

1D Array

- `cudaMalloc()` **start address is aligned**

2D Array

- `cudaMallocPitch()` **start address of every row is aligned (like multiple `cudaMalloc`'s)**

Alignment of structures

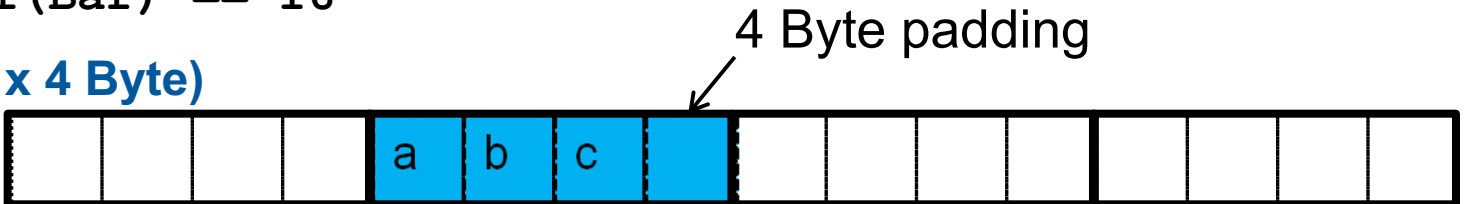
```
struct Foo {  
    float a;  
    float b;  
    float c; }  
// sizeof(Foo) == 12
```

Memory (16 x 4 Byte)

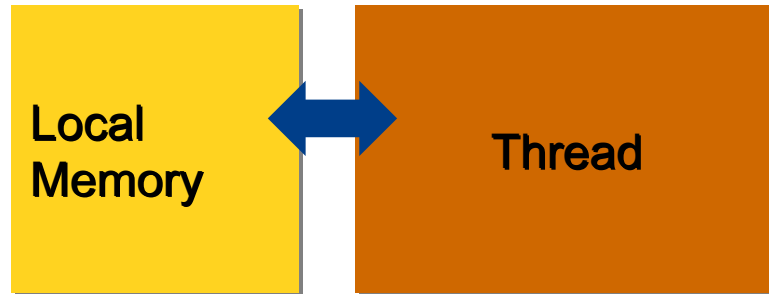


```
struct __align__(16) Bar {  
    float a;  
    float b;  
    float c; }  
// sizeof(Bar) == 16
```

Memory (16 x 4 Byte)



Local Memory: Hidden Global Memory

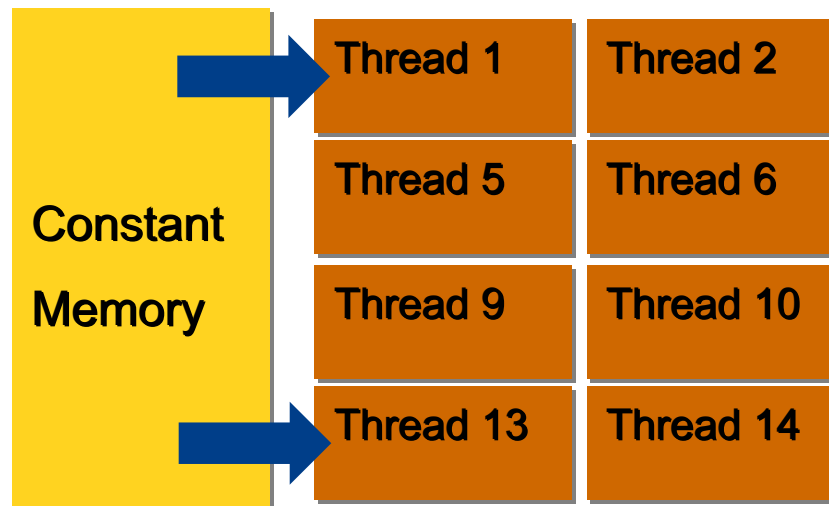


- *Register-spills*
- *non-uniform, dynamic arrays*

Size of Local Memory per thread: 512kB

What?	Thread-private Global Memory, usually cached by L1 and L2
How?	<code>__local__ int counter; int counter[n];</code>
Who?	Thread-private
Scope?	Lifetime of thread
Access?	R+W
Problems?	Arrays/Structures are probably stored inefficiently (like AoS)
nvprof shows if local memory is used, or use nvcc flags for kernel stats	
nvcc --resource-usage or nvcc -warn-lmem-usage	

Constant Memory: Read-Only Device Memory



Size of Constant Memory: 64kB

What?

How?

Who?

Scope?

Access?

Problems?

Off-chip memory, own on-chip cache (8-10kB), broadcast (saves loads)

```
__constant__ float constants_d [N];
```

```
cudaMemcpyToSymbol(constants_d, constants_h, size);
```

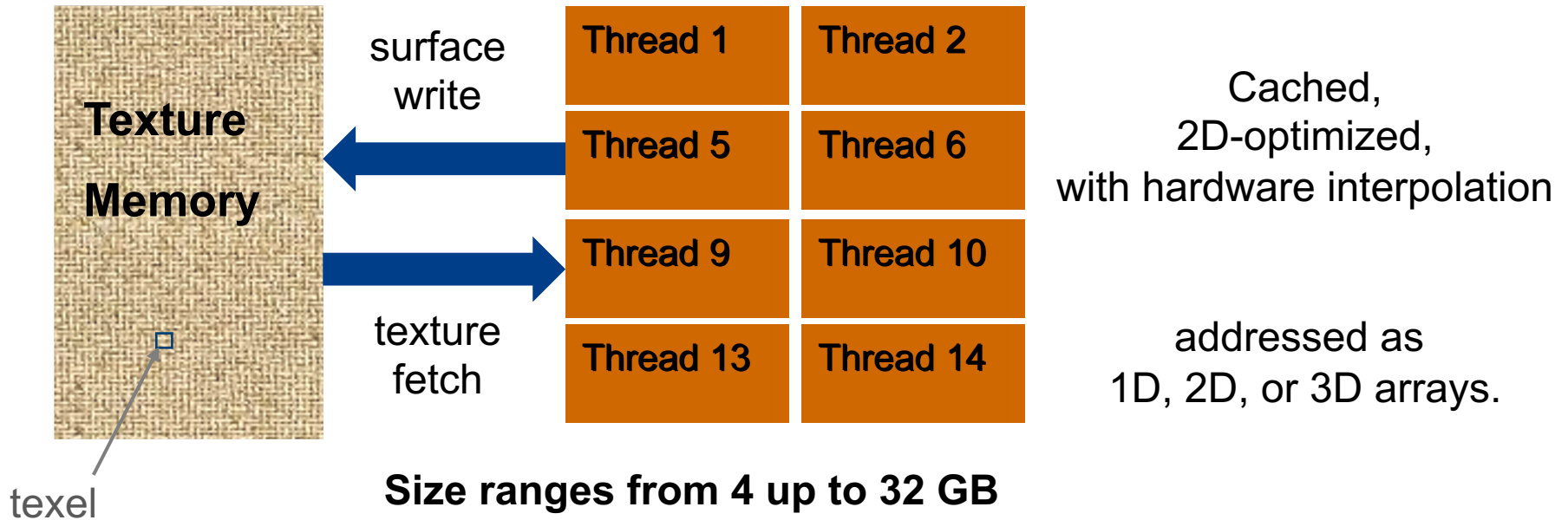
Device, host

Lifetime of application

R (Host can write)

transactions become serialized if warp requests different words

Texture Memory: For The Nostalgic?



What?

Uses global memory and L2, own on-chip cache(12-48KB), optimized for accesses with 2D spatial locality, automatic data & index interpolation, boundary handling

How?

... next page ...

Who?

Device, host

Scope?

Lifetime of application

Access?

R (Host can write) (device can write, if surfaces are used)

Problems?

... next page ...

Lab 3

→ In the pad



Engage...

Making things go even faster



Grid-Striding Loops

Rule of thumb: Use the C loop nest and change the **step width**

Example: Single Precision $A * X + Y$ (SAXPY)

```
__global__ void saxpy(int N, float a, float *x, float *y) {  
    for ( int i = blockIdx.x * blockDim.x + threadIdx.x;  
        i < N;  
        i += blockDim.x * gridDim.x ) { //stride of the loop  
        y[i] = a * x[i] + y[i];  
    }  
}  
  
int numSMs;  
cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount, devId);  
// Perform SAXPY on 1M elements  
saxpy<<<8*numSMs, 256>>>(1 << 20, 2.0, x_d, y_d);
```

device id

(0 = first visible CUDA device)

Why 8? Max. resident threads/SM = 2048 = 8*256

(rule of thumb, optimal number depends on algorithm)

<https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>

Grid-Striding Loops

- Loop over data with one grid-size at a time
- Allow to utilize multiprocessors on the device more balanced (number of blocks should be a multiple of the number of available multiprocessors)
- Improve scalability, because the problem size does not depend on the grid size that is supported by a device
- Easily enable to limit the block number to improve thread reuse (avoids thread creation and destruction costs) and tune performance
- Enable easy debugging by switching to serial execution, e.g.
`saxpy<<<1,1>>>(1<<20, 2.0, x_d, y_d);`
- Improve readability (kernel code is more similar to the CPU code)
- Improve portability (libraries such as *Hemi* allow to write portable kernels)

<https://devblogs.nvidia.com/paralleforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>

Nvidia's Compute Capability (CC)

- Defines set of GPU hardware and programming features
- CC is a version tag = `<major>.<minor>`, e.g. 3.7 for K80, where “3” marks Kepler and “7” the minor version within Kepler architecture
- Kernel codes designed for older CC still work on latest GPUs (probably runs inefficiently)
- So Kepler code runs on Volta, but Volta code may not work on Kepler (if you use Volta features like thread-independent scheduling)
- GPU generation and corresponding CUDA features must be known to write good code
 - How much shared memory per block can be used?
 - How many blocks can be resident on an SM?
 - Is half-precision hardware available? ...

[The CUDA wiki](#) has a nice lookup matrix for the CC features.

Nvidia's CUDA Compiler

NVCC compiler (is based on LLVM)

- Compiles C/C++ CUDA code parts and delegates host code to host compiler (gcc, msvc, ...)
- Allows to compile to an intermediate virtual language PTX (must be compiled at runtime by the driver) or directly to machine code (GPU must match CC)

• Different generations can be compiled into the binary

<=SM2x – Older cards such as GeForce GTX590 (removed as of CUDA9)

SM3x – **Kepler**

SM5x – **Maxwell**

SM6x – **Pascal** (requires CUDA8 to compile)

SM7.0-7.2 – **Volta** (requires CUDA9 to compile)

SM7.5 – **Turing** (requires CUDA10 to compile)

SM8.0 – **Ampere** (requires CUDA 11 to compile)

Occupancy – Keep the GPU Busy to Hide the Latencies

$$\text{Occupancy} = \frac{\text{active number of warps per SM}}{\text{max number of warps per SM}}$$

Maximum number of warps per multiprocessor: 64

Active number of warps per multiprocessor: it depends ...

- Number of registers used by the threads
 - Amount of shared memory used by the blocks
 - Number of threads per block
(should be multiple of warp size)
-
- **Occupancy is not the only performance factor**,
so lower occupancy can allow better performance in some cases.
 - Experiments usually required to find optimal configuration.
 - To determine optimal occupancy and kernel launch configuration an [excel sheet](#)
or a function [cudaOccupancyMaxActiveBlocksPerMultiprocessor\(\)](#) can be used.

Kernel resource limits

	A100
Max. threads per block	1024
Max. concurrent blocks per SM	32
Max. concurrent threads (warps) per SM	2048
Number of 32-bit registers per SM	64K
Max. 32-bit registers per block	64K
Max. 32-bit registers per thread	255
Max. shared memory per SM	164kB
Max. shared memory per block	163kB

Simultaneous processing of warps / blocks on a multiprocessor:

- more registers \Rightarrow less warps
- more shared memory \Rightarrow less blocks, . . .

Kernel resource limits

If each Block has 16x16 threads and each thread uses 40 registers, how many threads can run on each SM?

- Each Block requires $40 \times 256 = 10240$ registers
- $65536 = 6 \times 10240 + \text{remainder}$
- So, six blocks can run on an SM as far as registers are concerned

How about when each thread increases the use of registers by 4?

- Each Block now requires $44 \times 256 = 11264$ registers
- $65536 < 11264 \times 6$
- Only five blocks can run on an SM, 1/6 reduction of thread-level parallelism (TLP)!!!

Expose sufficient parallelism using TLP and ILP.

Occupancy

Execution context of a warp

- consists of: Program counters, registers and shared memory
- is maintained on-chip during warp lifetime (context switching has no cost)

Keep the device busy and hide all the latencies with enough warps!

occupancy = active warps/maximum warps

- number of threads per block: multiple of warp size (32), start with 128 or 256
- number of blocks: start with 64 warps per SM (for grid-stride looping)
- experiments required to find optimal execution and resource configuration
- guidance by CUDA occupancy calculator (Excel sheet) or nvvp
- *Do we want higher occupancy?* Maybe yes. Latency (of memory op. and algorithmic op.) can be hidden with more threads running.
- *Is occupancy THE metric of performance?* **No!** It's just one of the contributing factors.

Lab 4

→ In the pad



Ask me anything

Also provide feedback please

