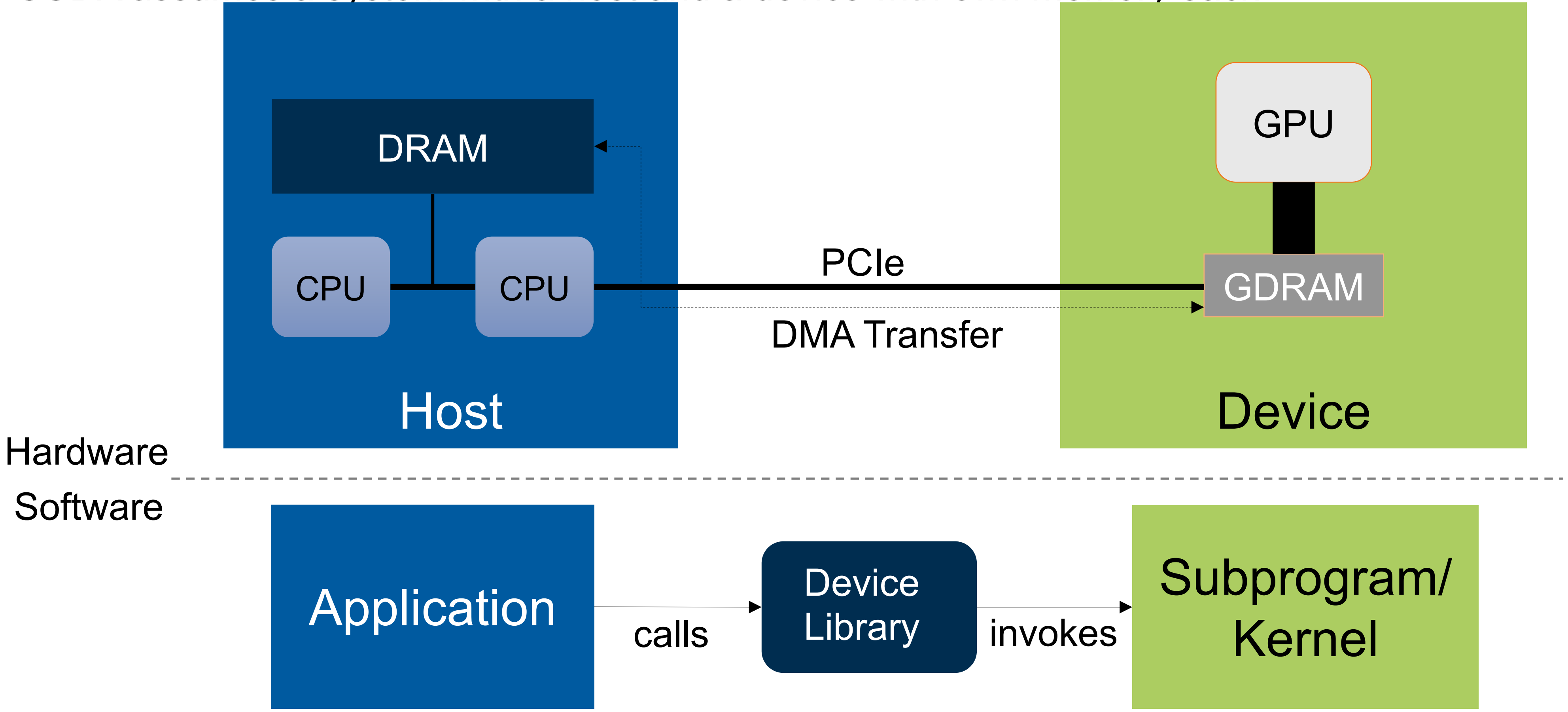
Highly Parallel Programming of GPUs

CUDA Kernels, Threads, Blocks and Grids

Wolfgang E. Nagel | Guido Juckeland | Robert Schöne

Recap: GPU System Setup

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



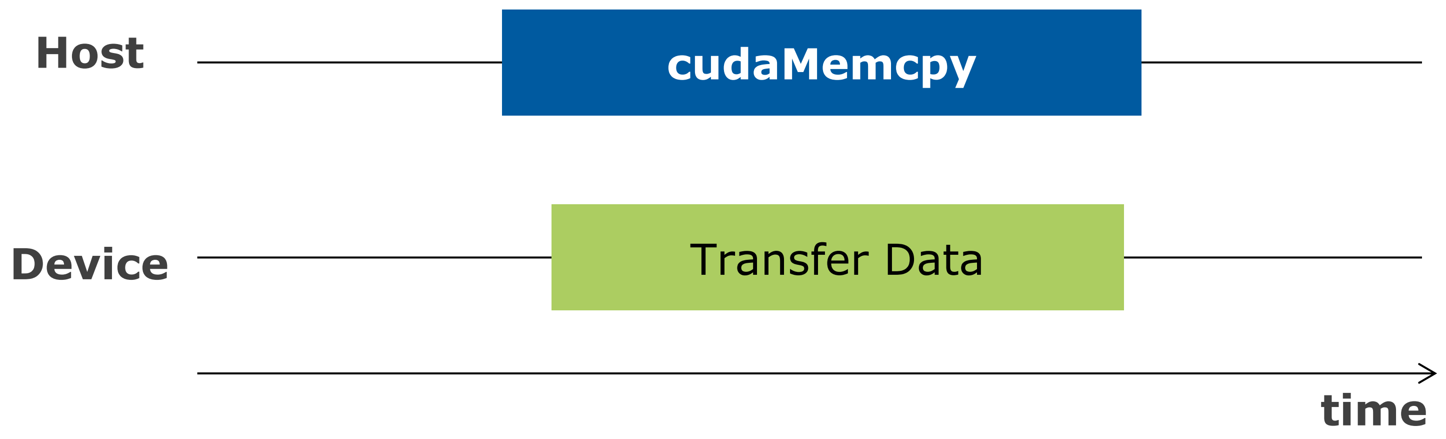
Kernels

- 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;
}
```

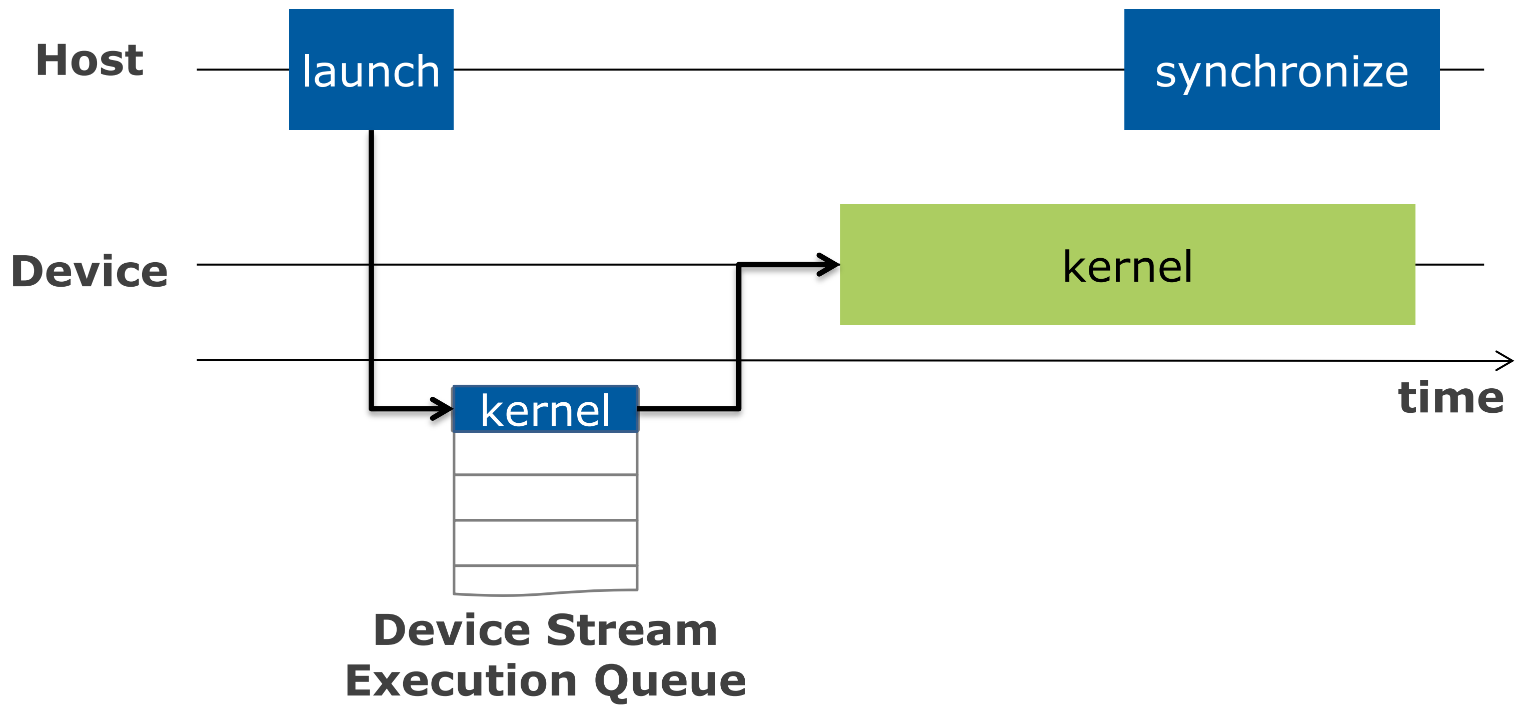
(Host-)Synchronous Execution

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() { … }
__device__ __host__ float help_do_nothing2() { … } //works on host and device
```

- 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 **I**nstruction, **M**ultiple **D**ata — **S**imultaneous **M**ulti-**T**hreading

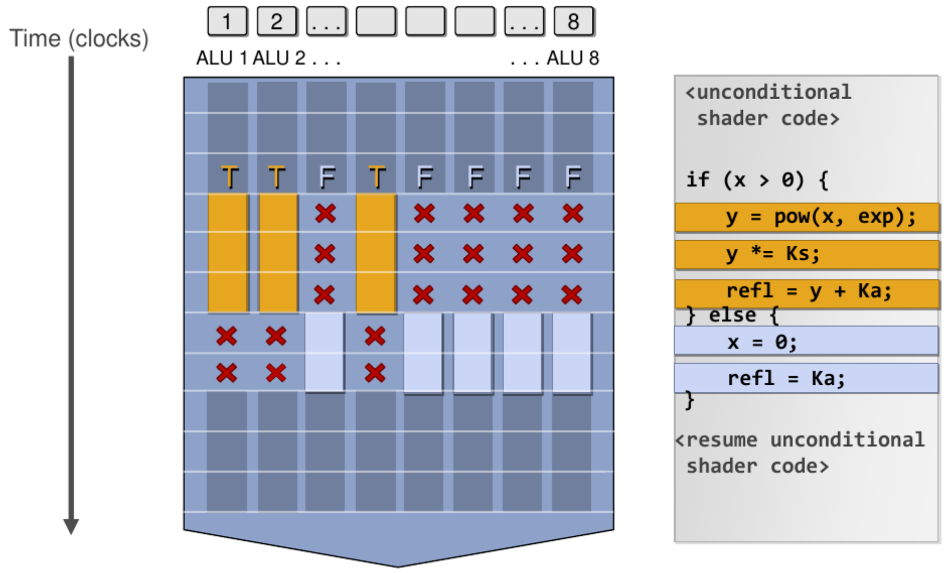
- SIMT allows multiple register sets, addresses and flow paths
- SIMT uses scalar spelling, ie.:

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

source

Single Instruction Multiple Threads (SIMT)

- SIMT: something in between SMT and SIMD
- SIMD processing can be realized by
  - Explicit vector instructions (x86 SSE, AVX, …)
  - Scalar instructions with implicit hardware vectorization (SIMT)
    - Easier to handle branches
    - Nvidia: warps, AMD: wavefronts



```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

pictogram source

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

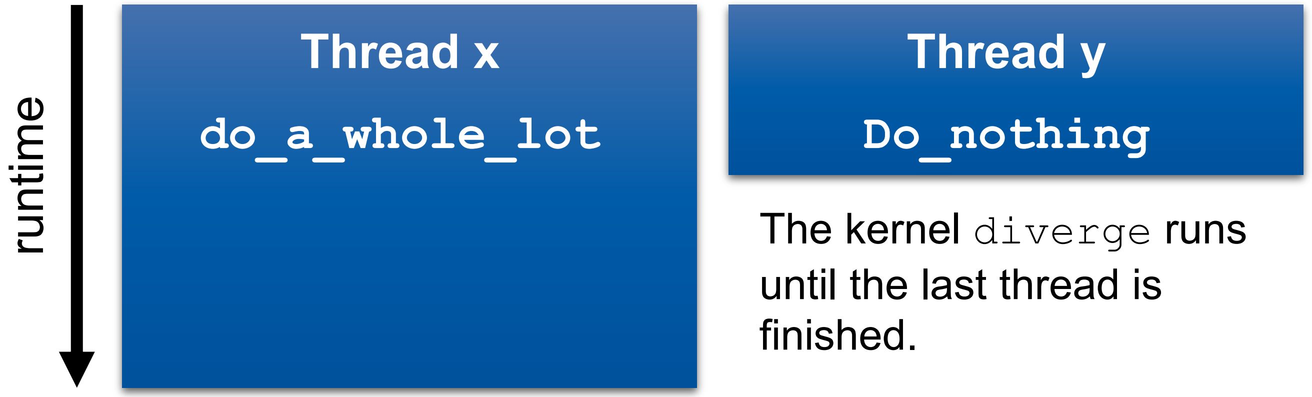
a[i]=b[i]+c[i];

Thread Divergence

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

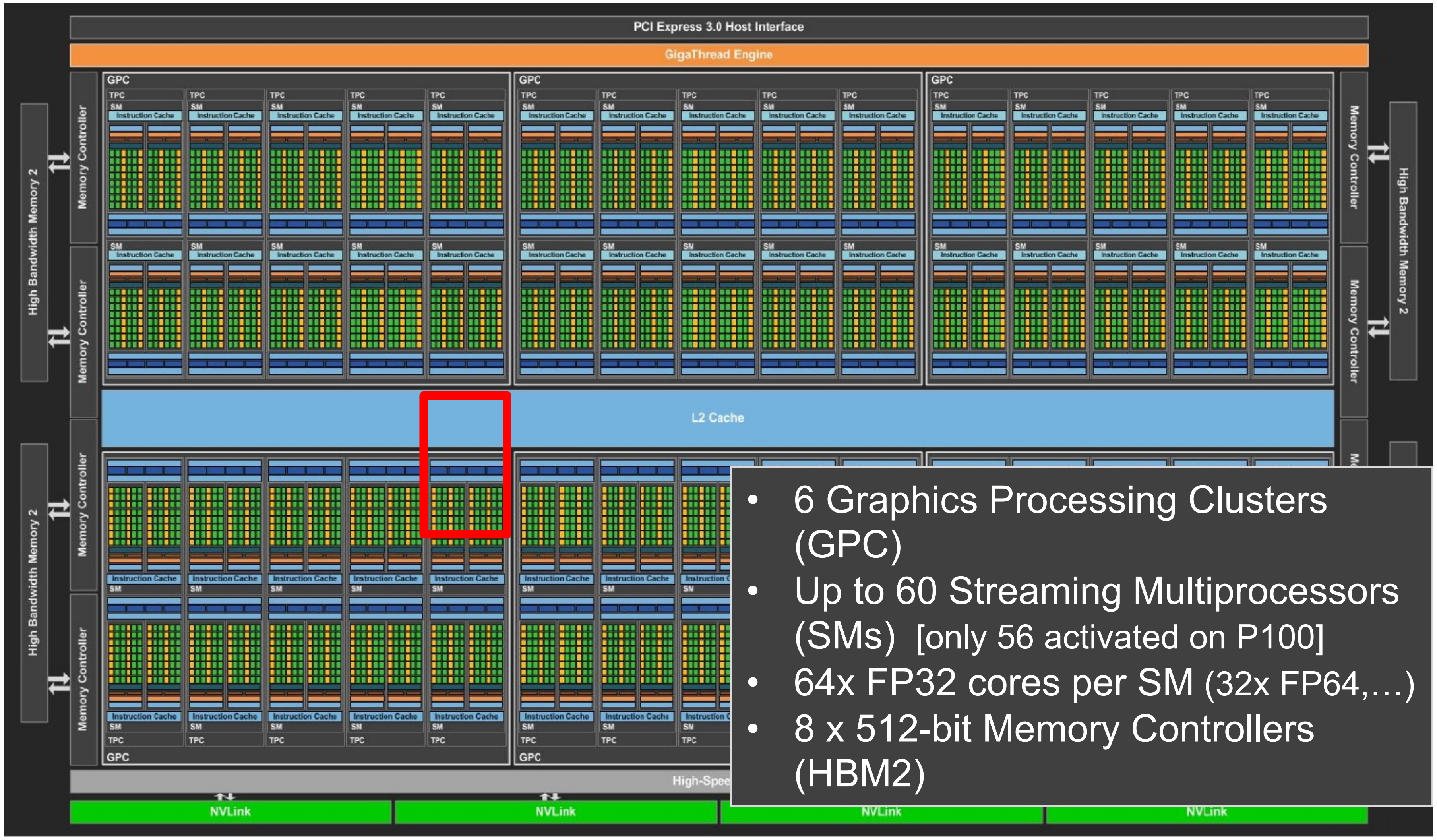
```
__global__ void diverge( void *data ) {
  if ( data[mythread] > random_number )
    do_a_whole_lot();
  else
    do_nothing();
}
```

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



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

NVIDIA Pascal GP100 Architecture



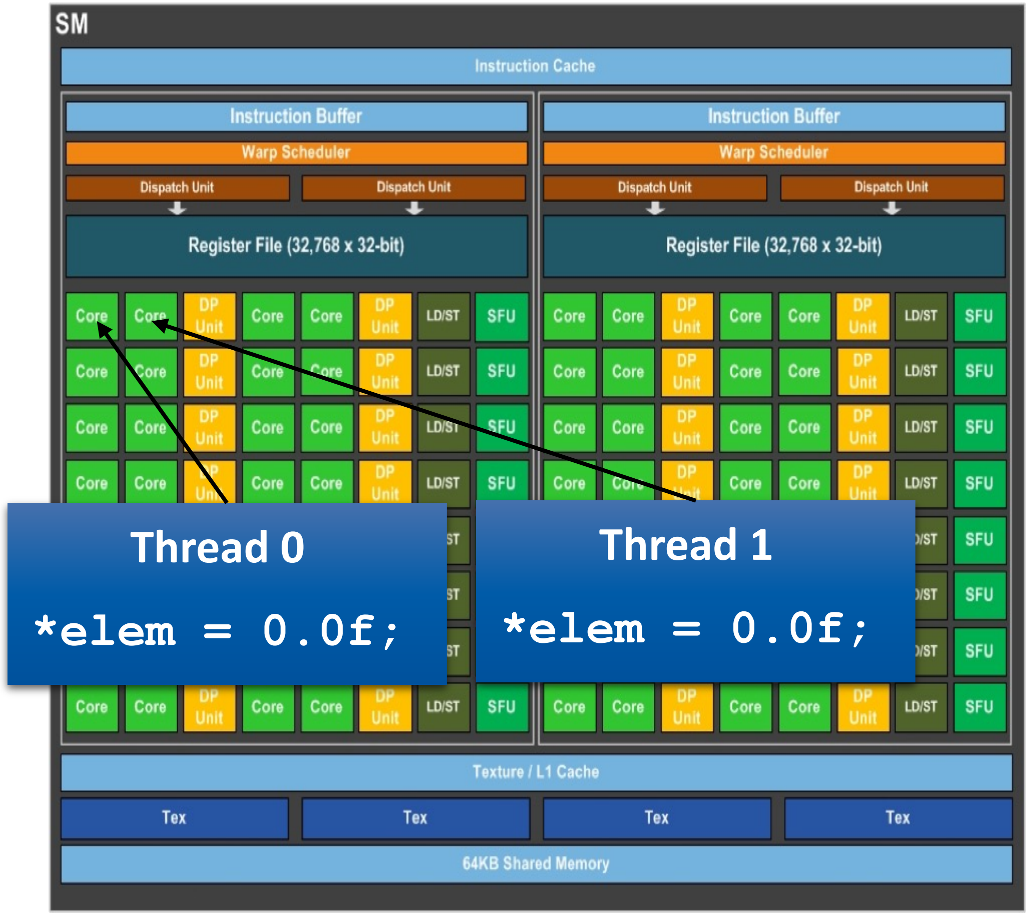
- 6 Graphics Processing Clusters (GPC)
- Up to 60 Streaming Multiprocessors (SMs) [only 56 activated on P100]
- 64x FP32 cores per SM (32x FP64,…)
- 8 x 512-bit Memory Controllers (HBM2)

Streaming Multiprocessor (Pascal)

By 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.

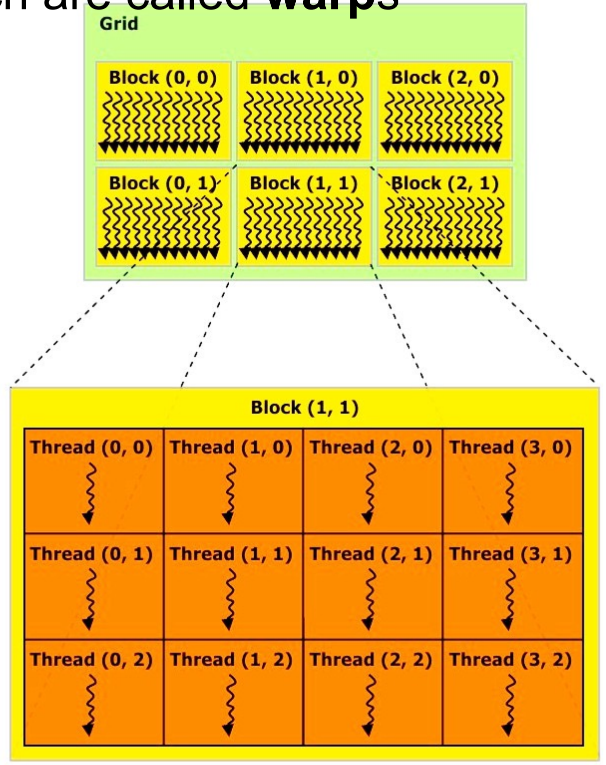


https://devblogs.nvidia.com/parallelforall/inside-pascal/

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>>>(…);
```



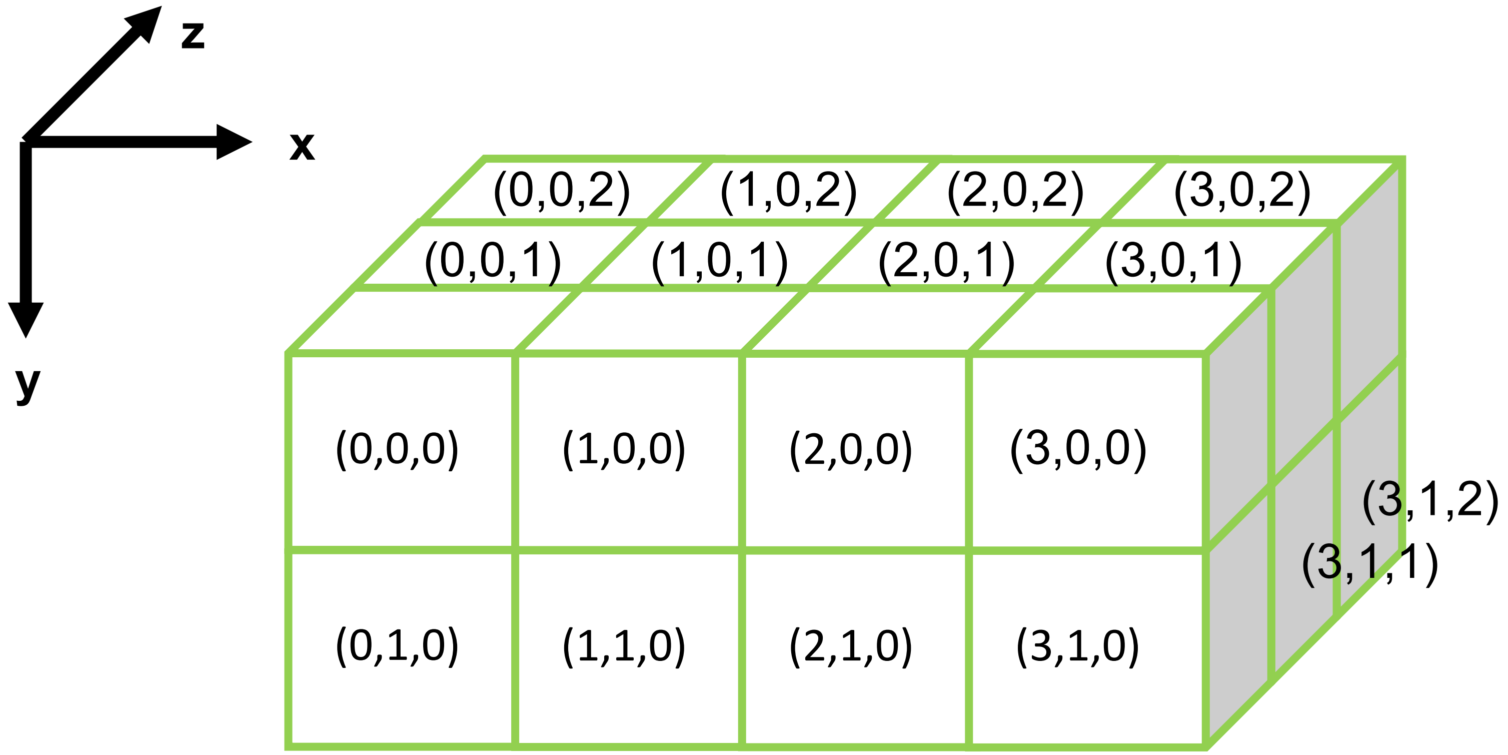
The dim3 Data Structure

```
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*2*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
Micro Architecture	*Fermi**	*Kepler⁺*	*Maxwell*	*Pascal*	*Volta*	*Ampere*
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

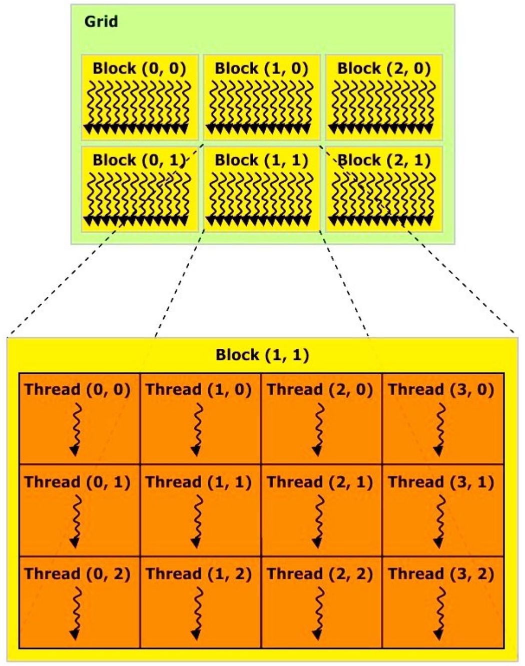
* Fermi is deprecated as of CUDA8 and without compiler support as of CUDA9

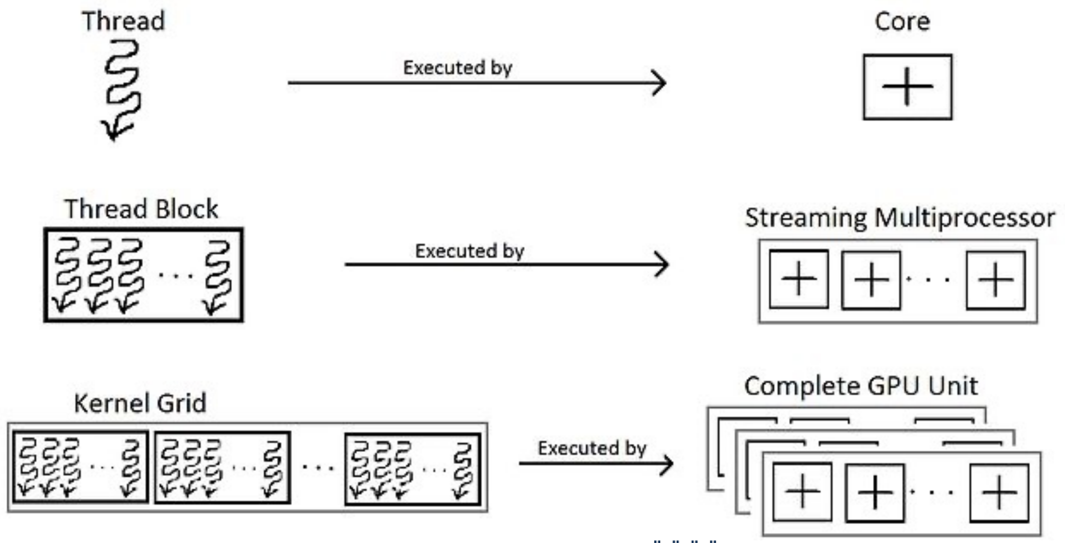
⁺ Kepler is deprecated as of CUDA10 and without compiler support as of CUDA11

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

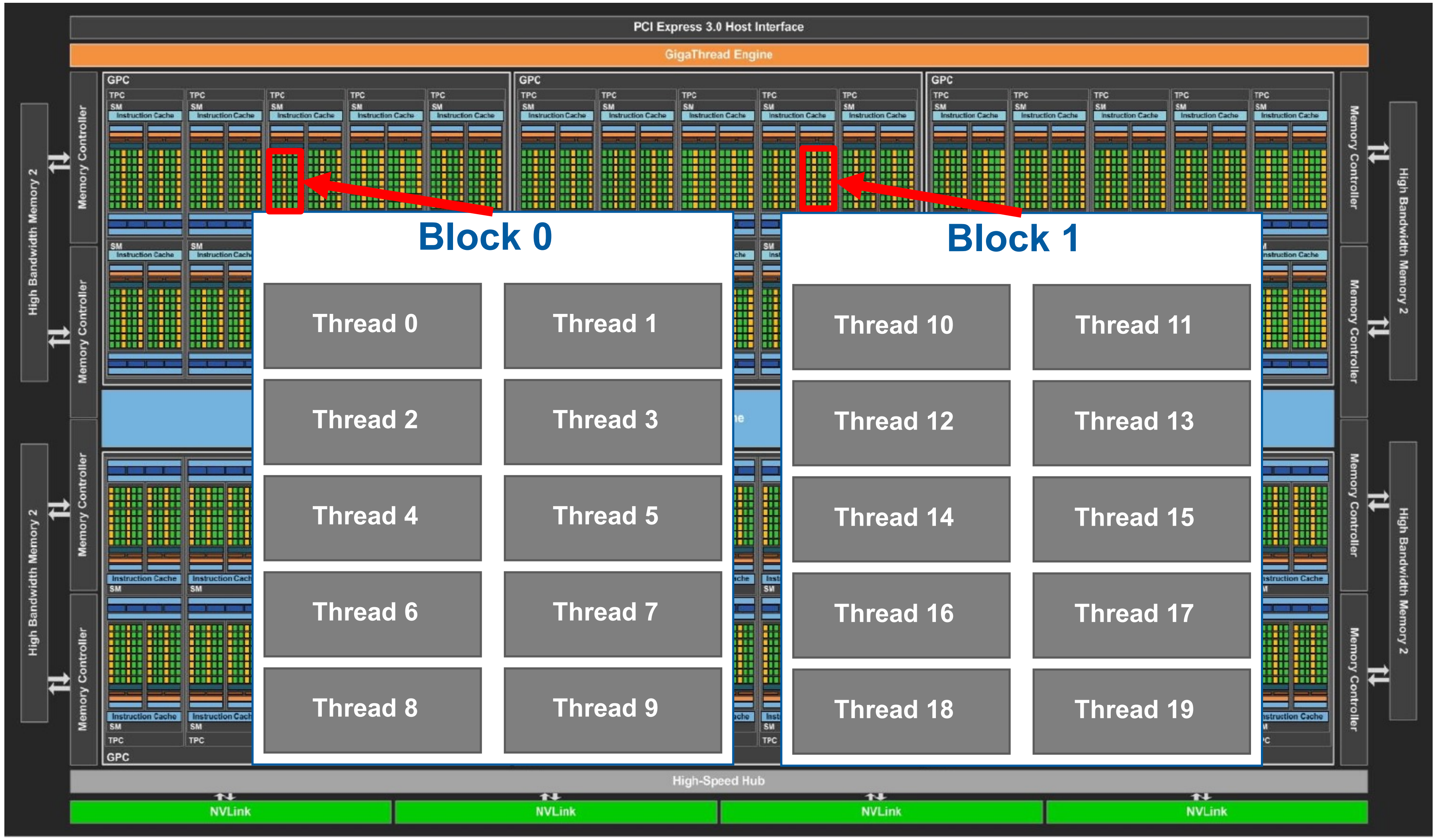
```
//integers mx, my, mz describe the grid in 3D
dim3 grid(mx, my, mz);
dim3 block(512);
//creates mx*my*mz blocks

kernel<<<grid,block>>>(…);
// grid size can also be a number

kernel<<<1024,512>>>(…);
```





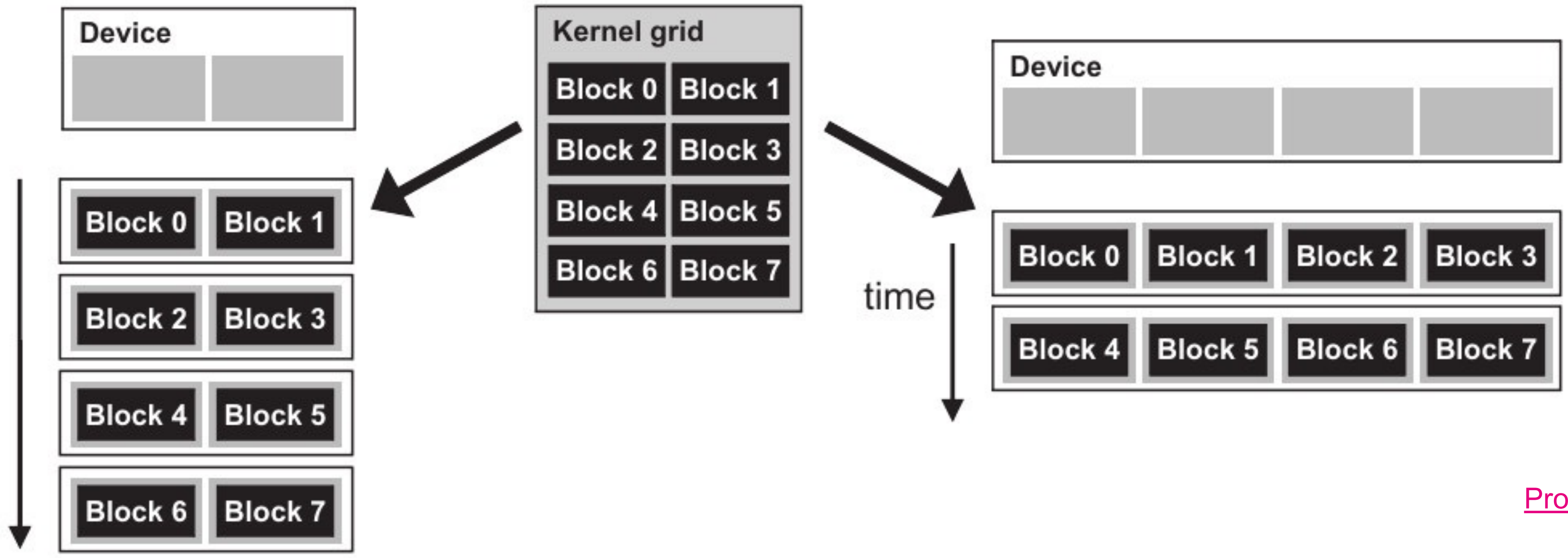
CC 4.0 by Atshardul

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
Micro Architecture	*Fermi**	*Kepler*	*Maxwell*	*Pascal*	*Volta*	*Ampere*
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:

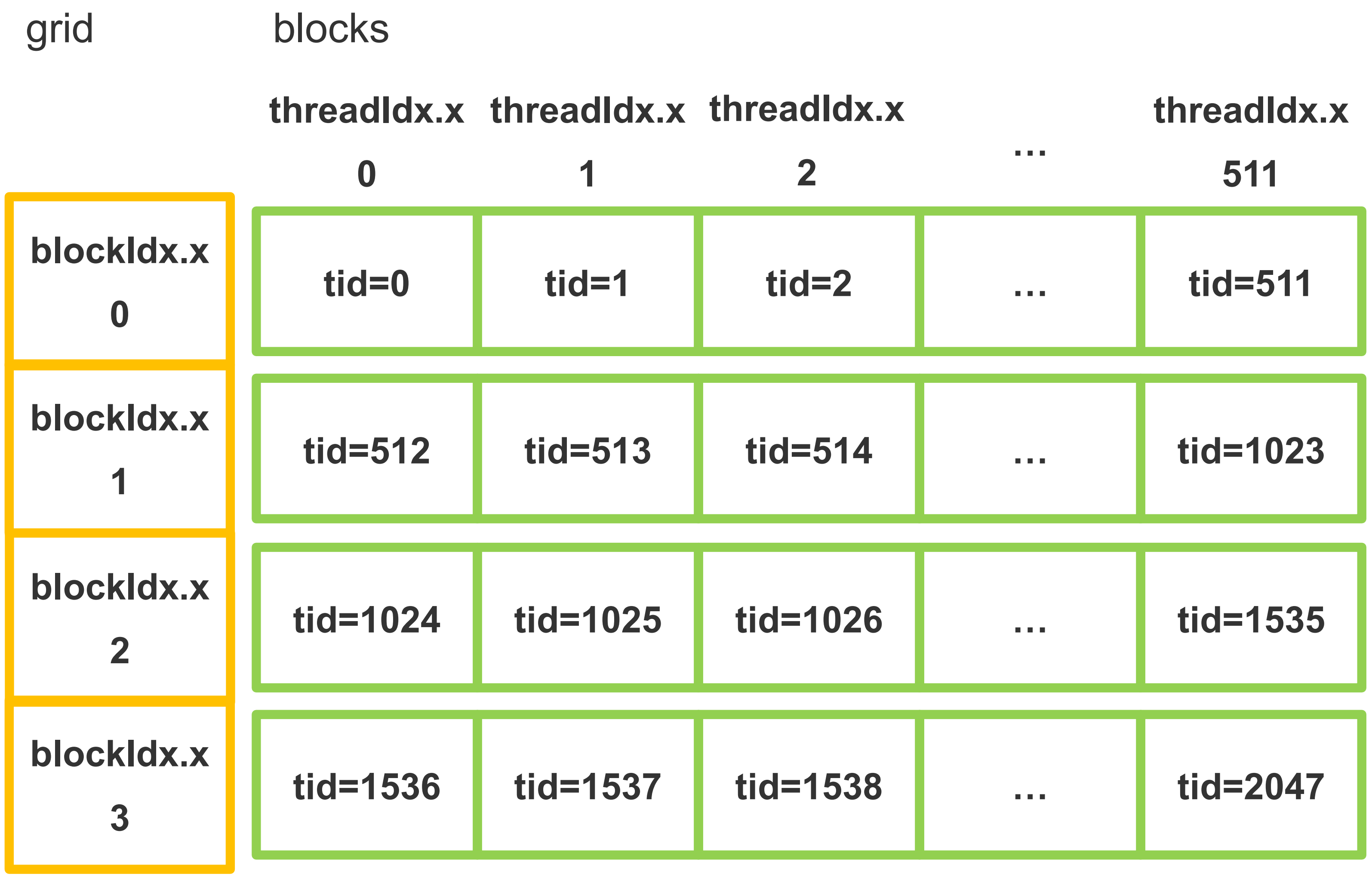
```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

1D grid of 2D block:

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

CUDA Thread Indexing Cheatsheet

Global Thread ID (Example 1D Block)



```
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>>>(1 << 20, 2.0, x_d, y_d);
```

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/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

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`
- Grid-striding loops are preferred over monolithic kernels