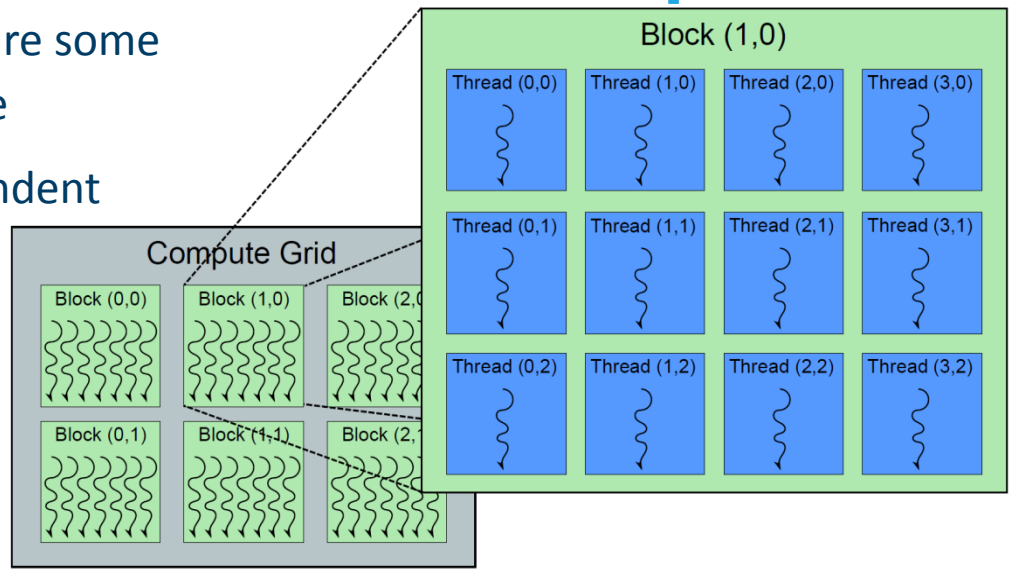
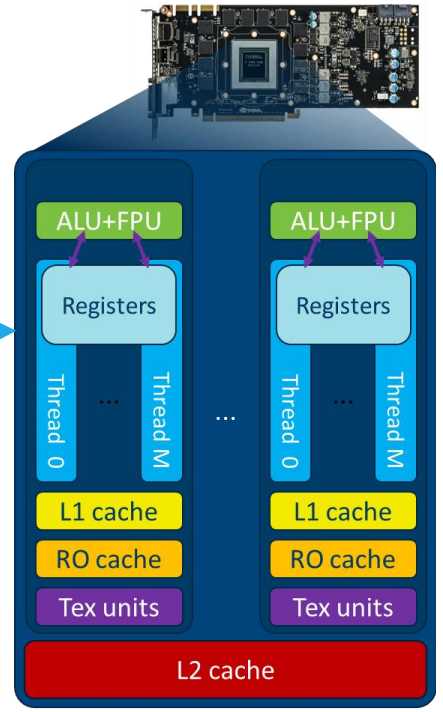
CONSERVATION LAWS ON GPUS: ADVANCED GPU

André R. Brodtkorb
Associate Professor, OsloMet – Oslo Metropolitan University
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital

Blocks and grids in CUDA

- Cuda uses blocks for parallelism
- A block runs on one of the cores on the GPU
- You have no control over which blocks run when
- Cuda schedules multiple blocks to a single core at the same time
  This gives memory parallelism to hide latencies!
- Threads within the same block can share some memory (__shared__) and synchronize
- Threads in different blocks are independent (no shared memory / synchronization)
- A block is divided into warps of 32 elements (SIMD execution)



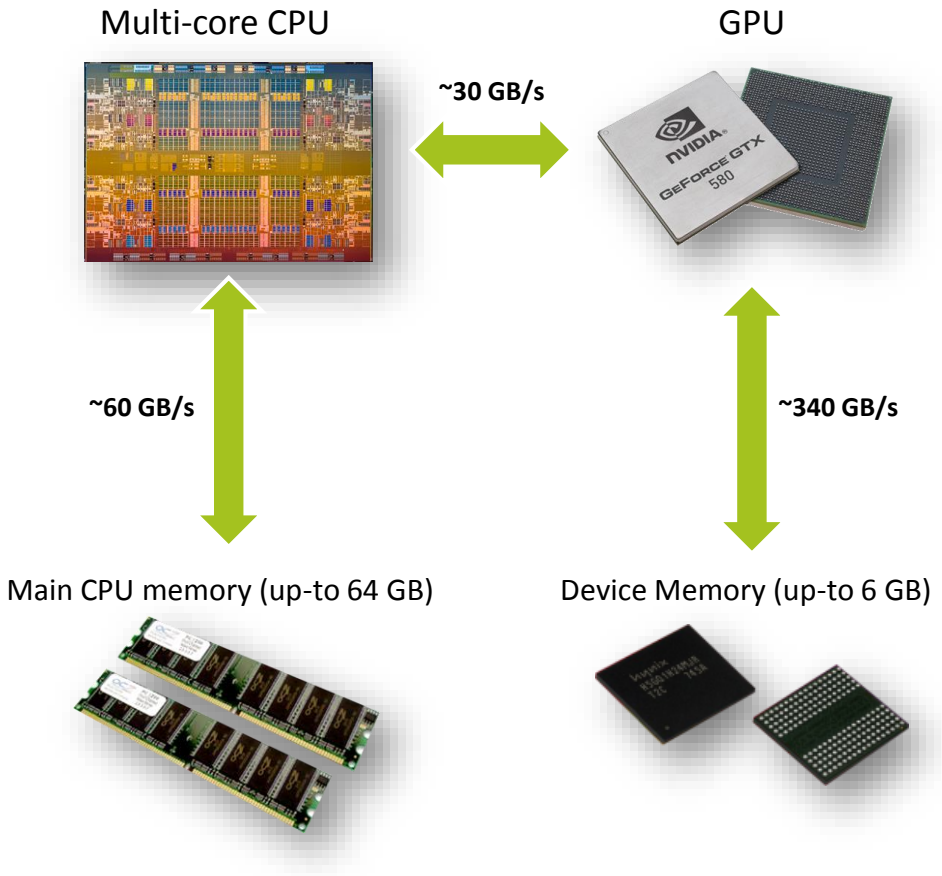


Simplified schematic of GPU design

SINTEF

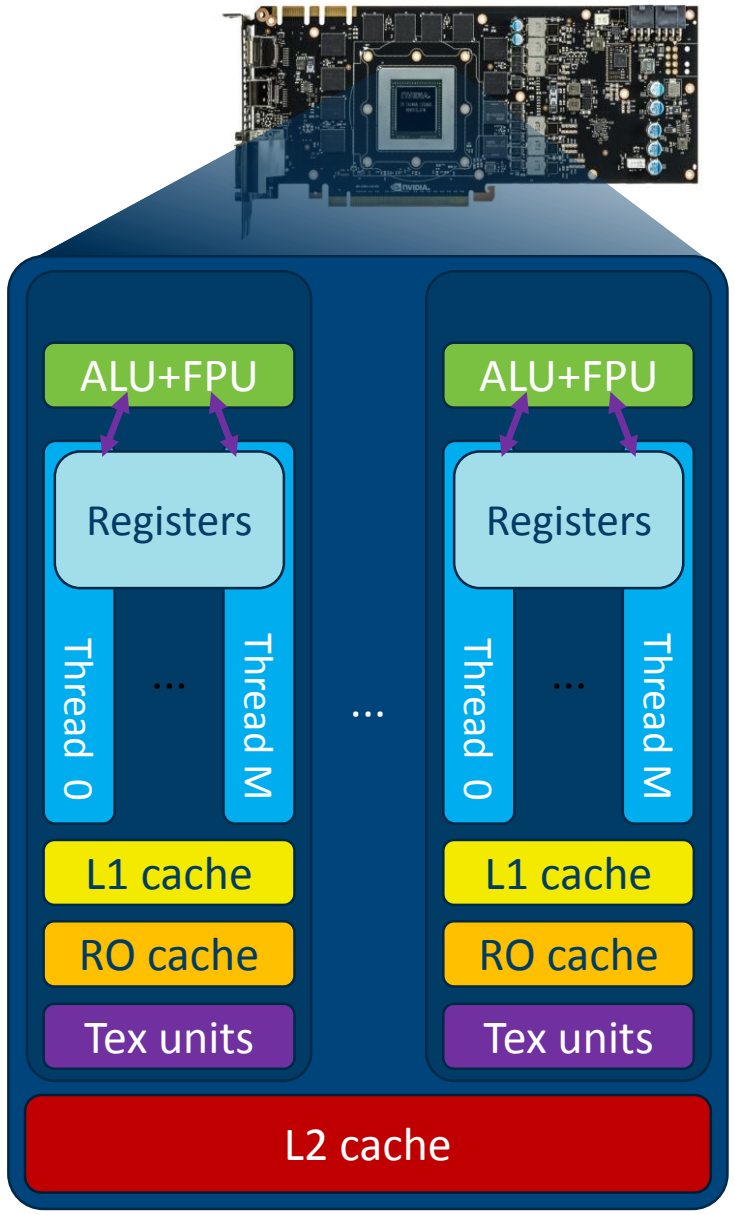
Heterogeneous Architectures

- Discrete GPUs are connected to the CPU via the PCI-express bus
  - Slow: 15.75 GB/s each direction
  - On-chip GPUs use main memory as graphics memory
- Device memory is limited but fast
  - Typically up-to 6 GB
  - Up-to 340 GB/s!
  - Fixed size, and cannot be expanded with new dimm's (like CPUs)



Caches and texture memory

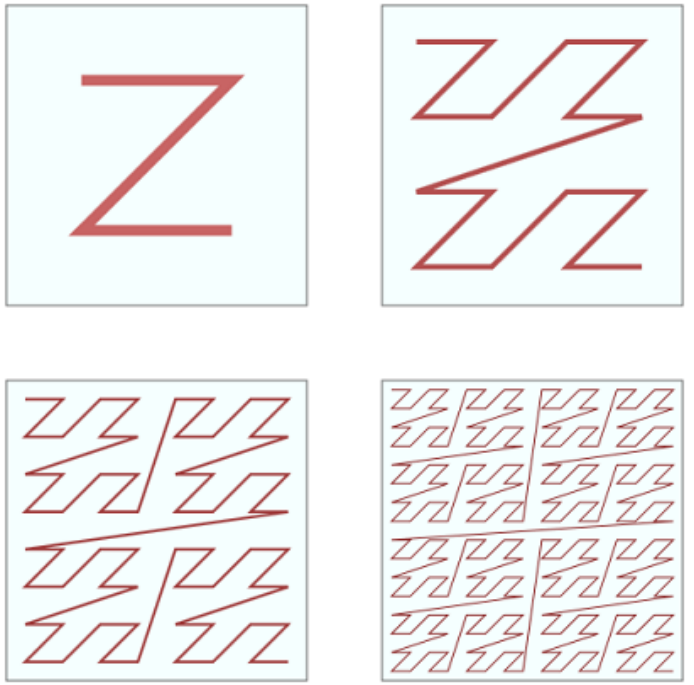
- The GPU has multiple memory caches
  - L1 and L2
  - These cache instructions and data from global memory
- Spilling registers to L1
  - When there is not enough physical registers available, some registers are moved to cache
  - Can increase performance some times
  - Controlled with –maxrregcount when compiling CUDA
- Texture caches can also be used to speed up memory reads from global memory
  - Originally a cache specialized for games
  - 2D layout and automatic coordinate transforms



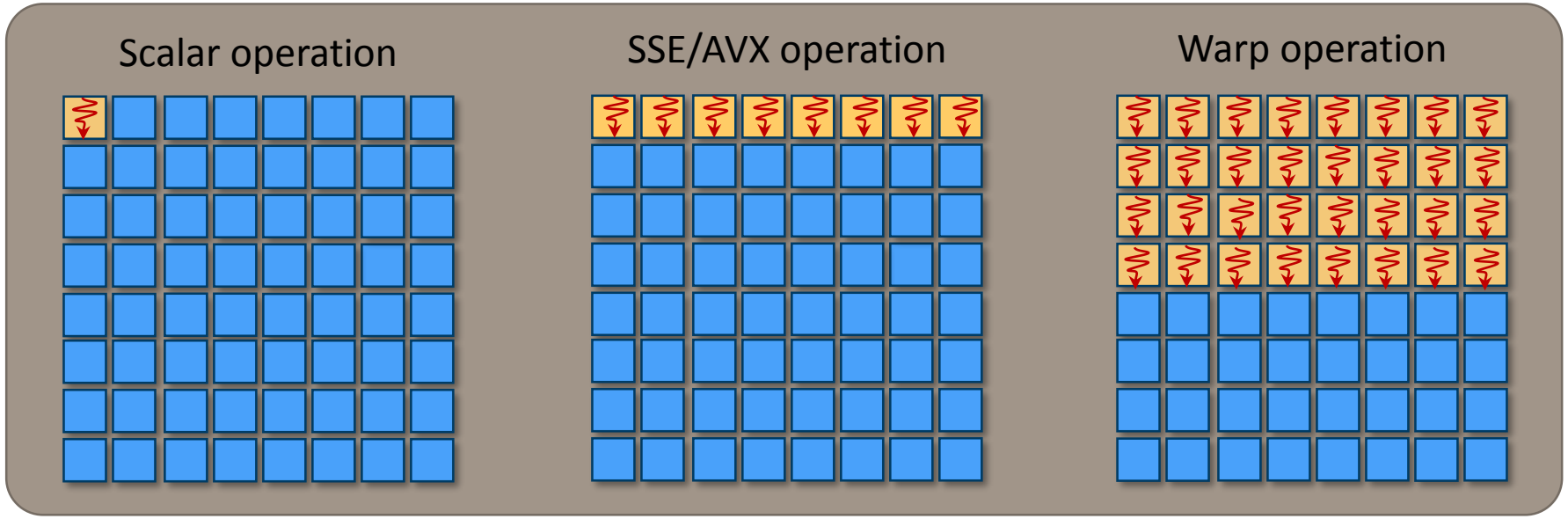
Simplified schematic of GPU design

Texture memory in PyCUDA

- Texture memory in CUDA is optimized for spatial locality
- Uses a space filling curve technology to store a 2D area in cache
- Really efficient for 2D memory access
- An extra cache that should be used if possible
- Caveat: The cache is read-only!

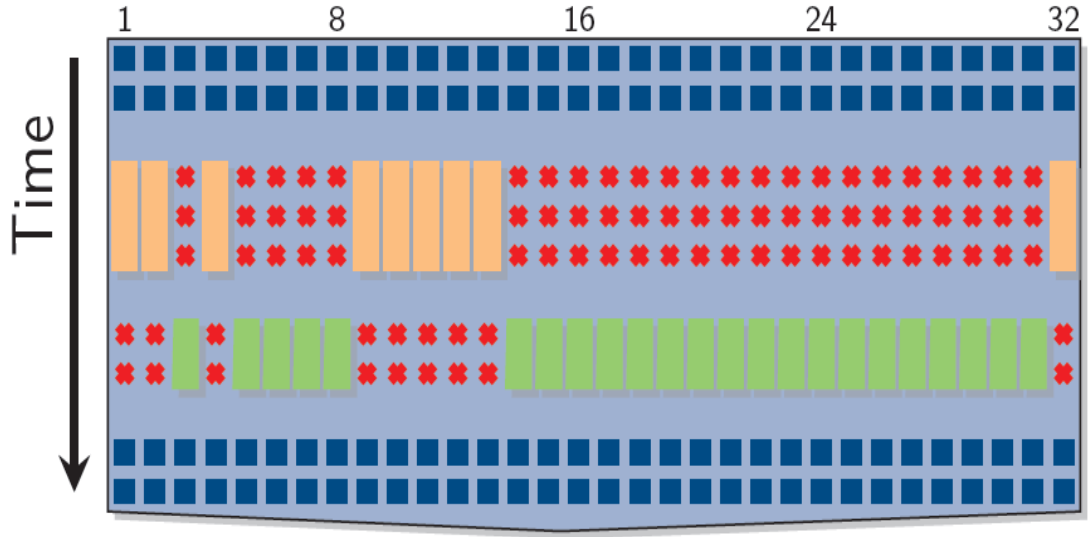


GPU Vector Execution Model



- **CPU scalar:** 1 thread, 1 operand on 1 data element
- **CPU SSE/AVX:** 1 thread, 1 operand on 2-8 data elements
- **GPU Warp:** 32 threads, 32 operands on 32 data elements
  - Exposed as **individual threads**
  - Actually runs the **same instruction**
  - Divergence implies **serialization and masking**

Serialization and masking




```
// Non-divergent code
if( x > 0 ) {
    y = pow(x, exp);
    y *= Ks;
    z = y + Ka;
} else {
    x = 0;
    z = Ka;
}
// Non-divergent code
```

Hardware automatically serializes and masks divergent code flow:

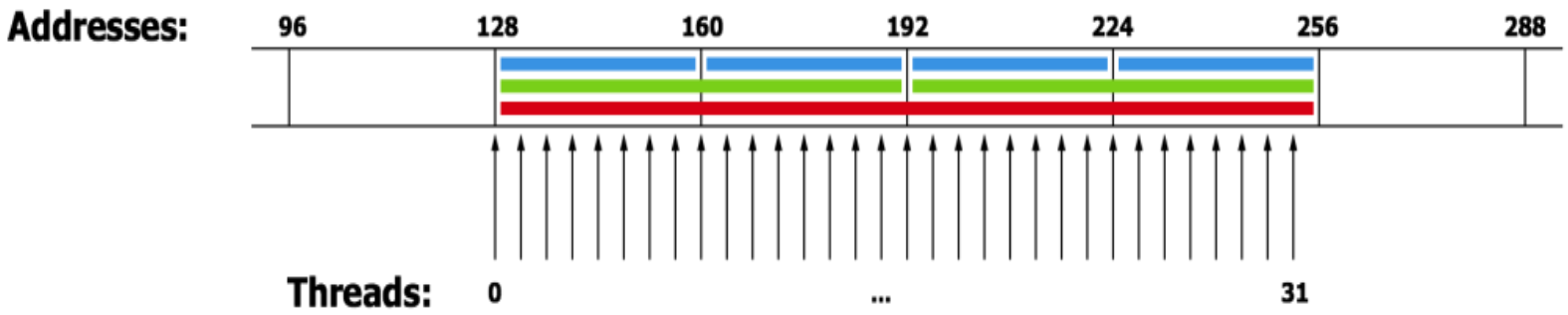
- Execution time is the sum of all branches taken
- Programmer is relieved of fiddling with element masks (which is necessary for SSE/AVX)
- Worst case 1/32 performance
- Important to **minimize divergent code flow within warps**!
  - Move conditionals into data, use min, max, conditional moves.

SINTEF

Memory access 1/2

- Accessing a single memory address triggers transfer of a full *cache line* (128 bytes)
  - The smallest unit transferrable over the memory bus
  - Identical to how CPUs transfer data

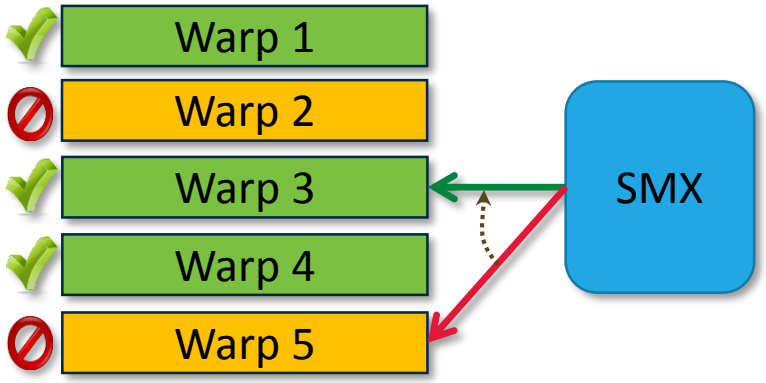
- For peak performance, 32 threads should use 32 consecutive integers/floats
  - This is referred to as coalesced reads



- On modern GPUs: Possible to transfer 32 byte segments: Better fit for random access!
- Slightly more complex in reality: see CUDA Programming Guide for full set of rules

Memory access 2/2

- GPUs have high bandwidth, and high latency
  - Latencies are on the order of hundreds to thousands of clock cycles



- Massive multithreading hides latencies
  - When one warp stalls on memory request, another warp steps in and uses execution units

- Effect: Latencies are completely hidden as long as you have enough memory parallelism:
  - More than 100 simultaneous requests for full cache lines per SM
  - Far more for random access!

For more details, see Paulius Micikevicius, GPU Performance Analysis and Optimization, 2013

Example: Parallel reduction

- Reduction is the operation of finding a single number from a series of numbers
  - Frequently used parallel building block in parallel computing
  - We've already used it to compute π

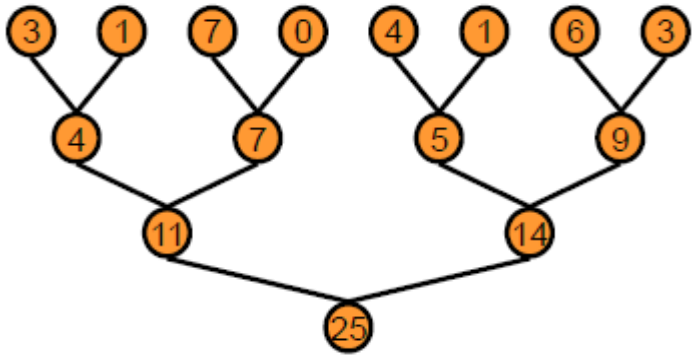
- Examples:
  - Find minimum, maximum, average, sum
  - In general: Perform a binary operation on a set data

- CPU example:

```
//Initialize to first element
T result = data[0];

//Loop through the rest of the elements
for (int i=1; i<data.size(); ++i) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

Parallel considerations

- This is a completely memory bound application
  - O(1) operation per element read and written.
  - Need to optimize for memory access!

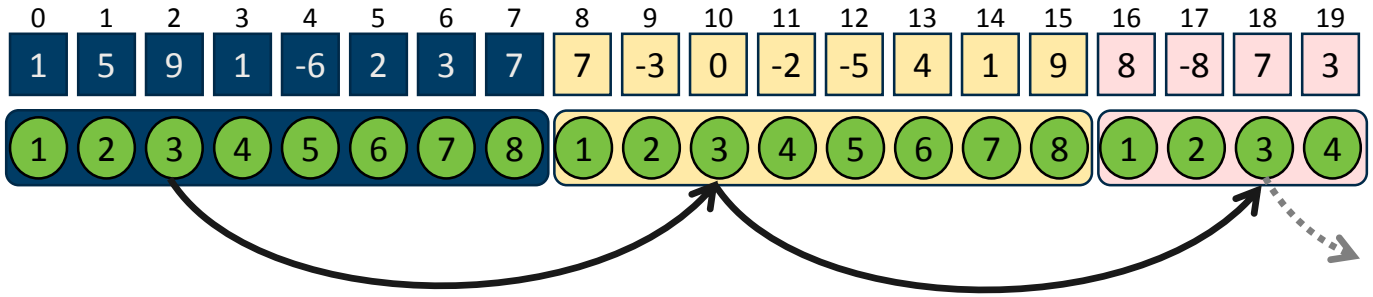


- Classical approach: represent as a binary tree
  - log2(n) levels required to reduce n elements
  - Example: 10 levels to find maximum of 1024 elements

- General idea on GPUs:
  - Use few blocks with maximum number of threads (i.e., 512 in this example)
  - *Stride* through memory until all items are read
  - Perform shared memory reduction to find single largest

Example based on Mark Harris, Optimizing parallel reduction in CUDA

Striding through data




```
for (int i=threadIdx.x; i<size; i += blockDim.x) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

- Striding ensures perfect coalesced memory reads
- Thread 2 operates on elements 2, 10, 18, etc. for a block size of 8
- We have block size of 512: Thread 2 operates on elements 2, 514, 1026, …
- Perform "two-in-one" or "three-in-one" strides for more parallel memory requests

SINTEF

Shared memory reduction 1/2

- By striding through data, we efficiently reduce N/num_blocks elements to 512.

- Now the problem becomes reducing 512 elements to 1:
  lets continue the striding, but now in shared memory

- Start by reducing from 512 to 64 (notice use of __syncthreads()):

```
__syncthreads(); // Ensure all threads have reached this point

// Reduce from 512 to 256
if(tid < 256) { sdata[tid] = sdata[tid] + sdata[tid + 256]; }
__syncthreads();

// Reduce from 256 to 128
if(tid < 128) { sdata[tid] = sdata[tid] + sdata[tid + 128]; }
__syncthreads();

// Reduce from 128 to 64
if(tid < 64) { sdata[tid] = sdata[tid] + sdata[tid +  64]; }
__syncthreads();
```

Shared memory reduction 2/2

- When we have 64 elements, we can use 32 threads to perform the final reductions
- Remember that 32 threads is one warp, and execute instructions in SIMD fashion
- This means we do not need the syncthreads:

```
if (tid < 32) {
        volatile T *smem = sdata;
        smem[tid] = smem[tid] + smem[tid + 32];
        smem[tid] = smem[tid] + smem[tid + 16];
        smem[tid] = smem[tid] + smem[tid + 8];
        smem[tid] = smem[tid] + smem[tid + 4];
        smem[tid] = smem[tid] + smem[tid + 2];
        smem[tid] = smem[tid] + smem[tid + 1];
}

if (tid == 0) {
        global_data[blockIdx.x] = sdata[0];
}
```

- Volatile basically tells the optimizer "off-limits!"
- Enables us to safely skip __syncthreads()

SINTEF

Preparing kernel calls for async execution

- Pycuda has a significant overhead for launching a kernel with many arguments
  - Function calls in Python are slow, can be optimized using socalled prepared calls

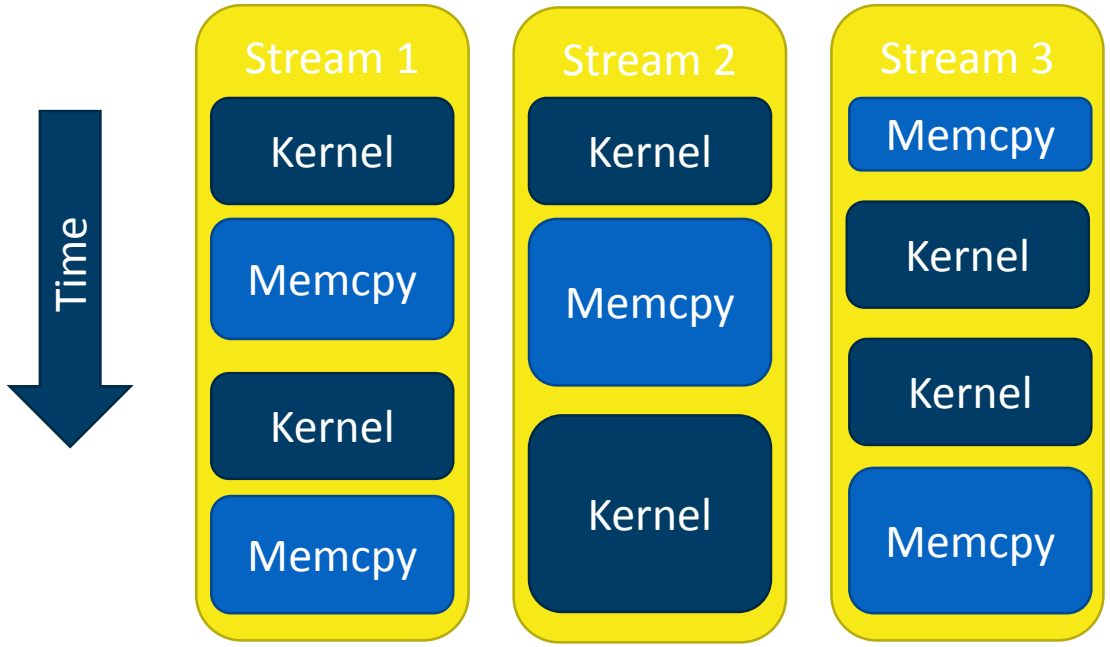
```
#Prepare function call
module = cuda_compiler.SourceModule(cuda_kernel)
kernel = module.get_function("addKernel");
kernel.prepare("PPP")
stream = cuda_driver.Stream()
…

#Execute program on device
grid = (1, 1, 1)
block = (n, 1, 1)
kernel.prepared_async_call(grid, block, stream, c_g.gpudata, a_g.gpudata, b_g.gpudata)

#Copy data from device to host
c_g.get_async(stream=stream, ary=c)

context.synchronize()
```

SINTEF

Asynchronous memory and kernel execution

- Enabling asynchronous memory transfers enables the use of concurrent streams

- The GPU can launch several kernels simultaneously and perform uploads and downloads



CUDA (NVCC) Compilation flags

- Compilation flags are important tuning parameters

- Make sure that you always test results after changing compilation flags: things might break!

- --maxrregcount – specify maximum number of registers per block
- --use_fast_math – Use fast (but less accurate) math library
- --gpu-architecture=compute_50 --gpu-code=sm_50,sm_52 – Specify which GPUs to compile/optimize for

- Specified to pycuda.compiler

class `pycuda.compiler.`**`SourceModule`**(*source*, *nvcc="nvcc"*, *options=None*, *keep=False*, *no_extern_c=False*, *arch=None*, *code=None*, *cache_dir=None*, *include_dirs=[]*)

- Full overview: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

OpenCL compile flags

- Compilation flags in OpenCL also:

- -cl-single-precision-constant -- Treat double precision floating-point constant as single precision constant.
- *-cl-denorms-are-zero – Treat very small numbers as zero*
- *-cl-mad-enable – Enable multiply-add instruction*
- *-cl-unsafe-math-optimizations – Enable unsafe/relaxed math*
- *-cl-finite-math-only – Assume no NaN or inf in floating point*
- *-cl-fast-relaxed-math – Both of the above*

- Full overview: https://www.khronos.org/registry/OpenCL/sdk/2.1/docs/man/xhtml/clBuildProgram.html