Numerical Simulation on the GPU

- Roadmap
  - Part 1: GPU – architecture and programming concepts
  - Part 2: An introduction to GPU programming using CUDA
  - Part 3: Numerical simulation techniques (grid and particle based approaches)
  - Part 4: Mapping numerical simulation techniques to the GPU
  - Part 5: GPU based techniques for scientific visualization

- Lecturer:
  Prof. Dr. R. Westermann
  Chair for Computer Graphics and Visualization, TUM

Credits

- Mark Harris
  - http://gpgpu.org/
  - http://www.markmark.net/
- Michael Garland
  - http://mgarland.org/papers.html
- David Kirk
  - http://research.nvidia.com/users/david-kirk
- John Owens
  - http://www.ece.ucdavis.edu/~jowens/
  - http://gpgpu.org/s2007
- Kayvon Fatahalian
  - http://graphics.stanford.edu/~kayvonf/
- NVIDIA
  - http://www.nvidia.com/object/cuda_home_new.html
  - http://developer.nvidia.com/object/cuda_training.html
- Markus Hadwiger
  - http://www.voreen.org/277-Eurographics-2009-Tutorial.html

Further ressources

- Dominik Göddeke -- GPGPU Tutorials
  - http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html
- Robert Strzodka -- GPGPU Tutorials
  - http://www.mpi-inf.mpg.de/~strzodka/
- Tutorial on Volume Visualization
  - http://www.vis.uni-stuttgart.de/vis03_tutorial/

GPUs
Architecture and Programming

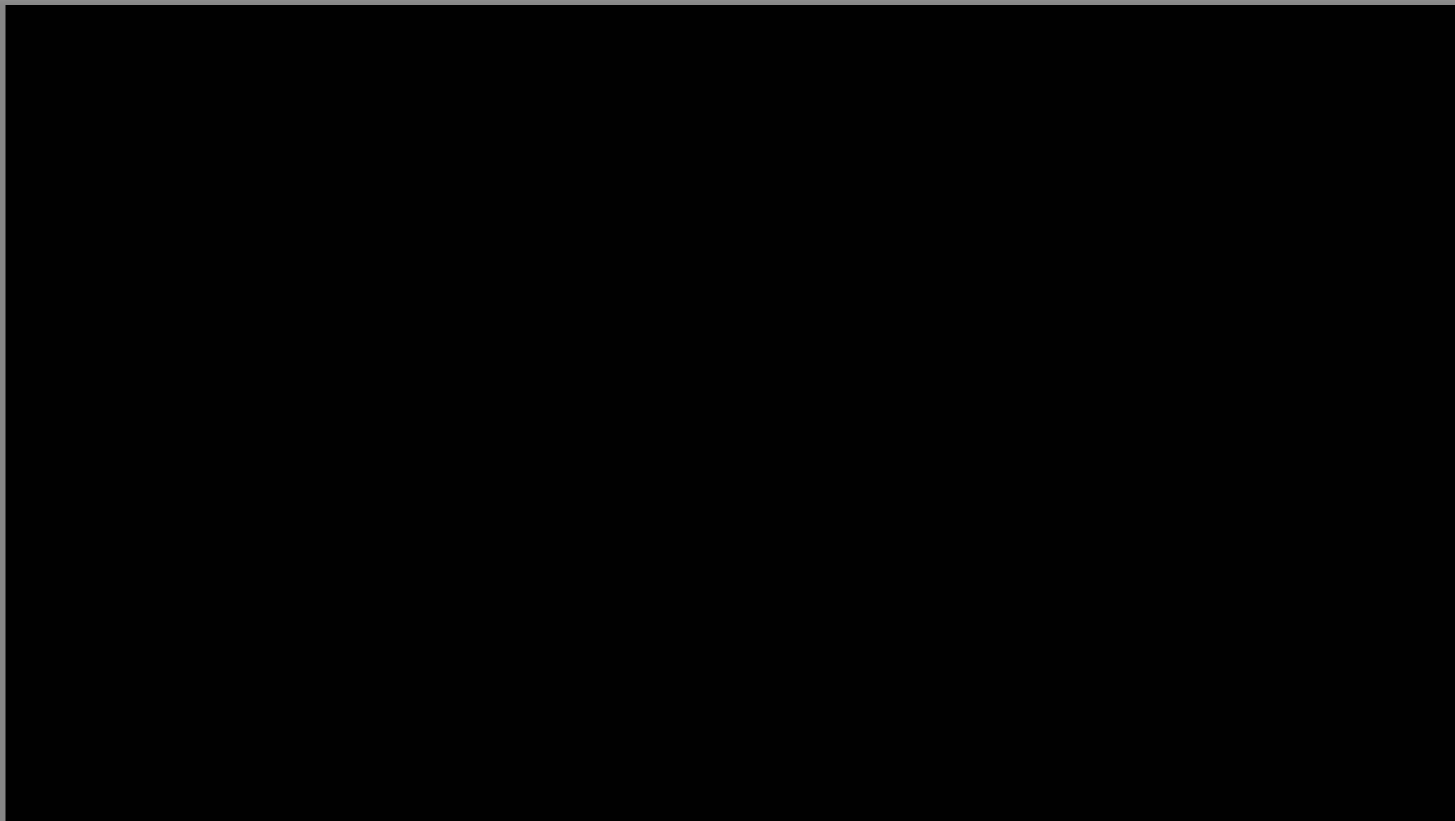
Rüdiger Westermann
Chair for Computer Graphics & Visualization

Overview

- GPUs
  - Performance analysis
  - Architectural design
  - Execution model
  - Programming concepts

GPUs (Graphics Processing Units)

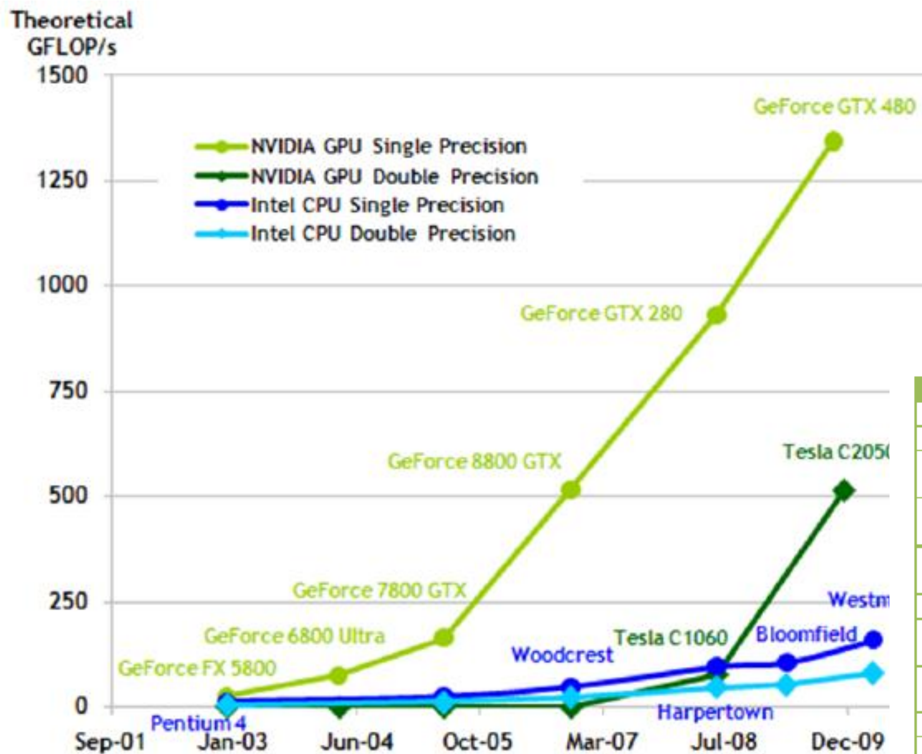
- Where does the interest in GPUs come from?



Video games sales close to 10 billion in 2009

GPUs

- Where does the interest in GPUs come from?



  – GPUs deliver unprecedented arithmetic performance

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

GPUs

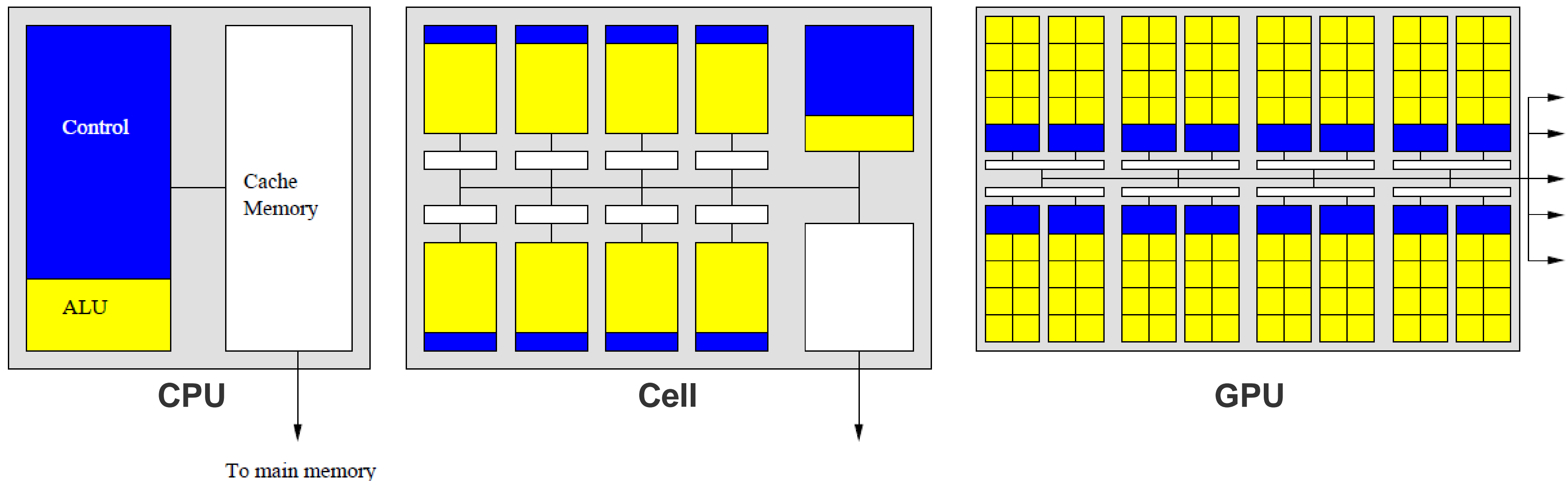
- A closer look on arithmetic and memory throughput:

	GeForce GTX 280	Radeon HD 4870	Radeon HD 5870	Fermi GTX 480	Intel Core i7 989 Core6
single-precision arithmetic (GFLOPS)	933	1200	2720	1345	130
double-precision arithmetic (GFLOPS)	78	240	544	620	130
memory bandwidth (GB/s)	141	115	153	177	120
processor clock (MHz)	602	750	850	1400	3300

  - **Inside gained**: best performance if calculation is more common than data accesses (1.4 * 512 * 3 * 4 GB/s = 8601,6 GB/s, roughly 10 ops per word)
  - **Inside gained**: There must be some parallelism in it

GPUs

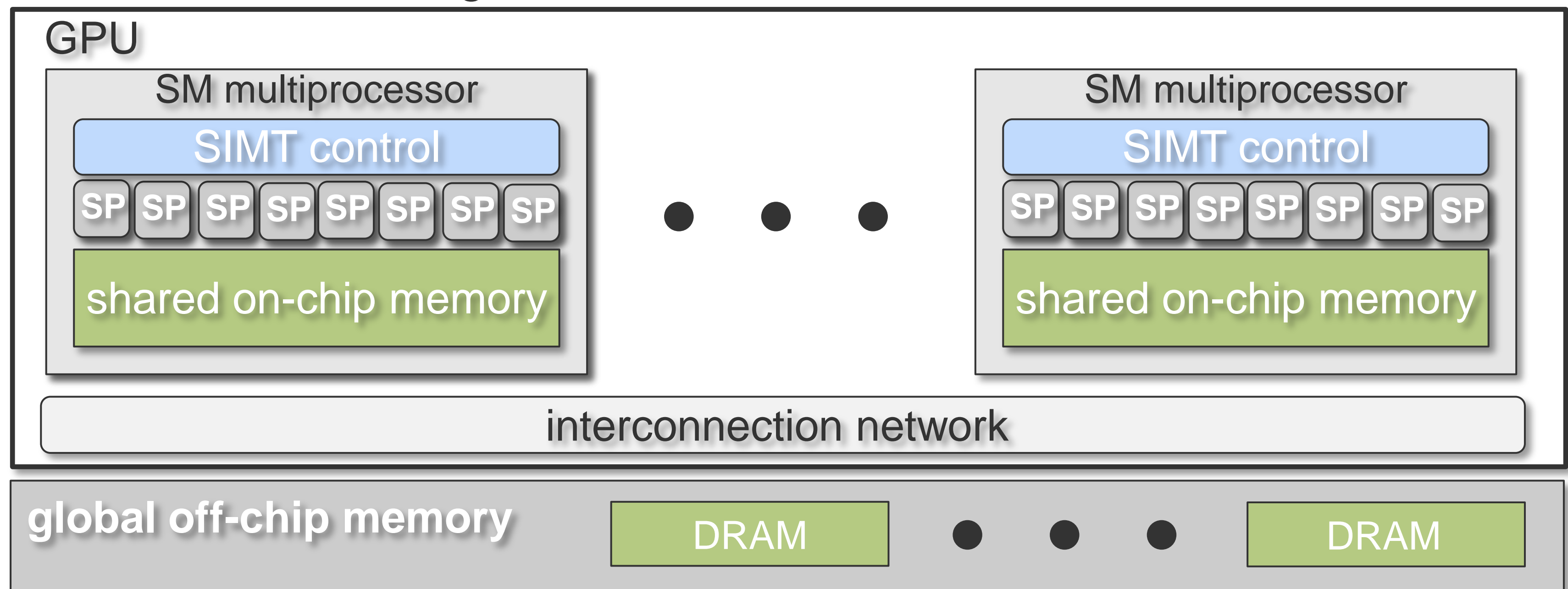
- Architectural design in the light of a CPU‘s design



- Conventional CPUs are equipped with complex control logic and large caches
  - Execution optimizations like branch prediction, speculative prefetching etc.
  - Large caches hide data access latencies due to slow main memory

GPUs

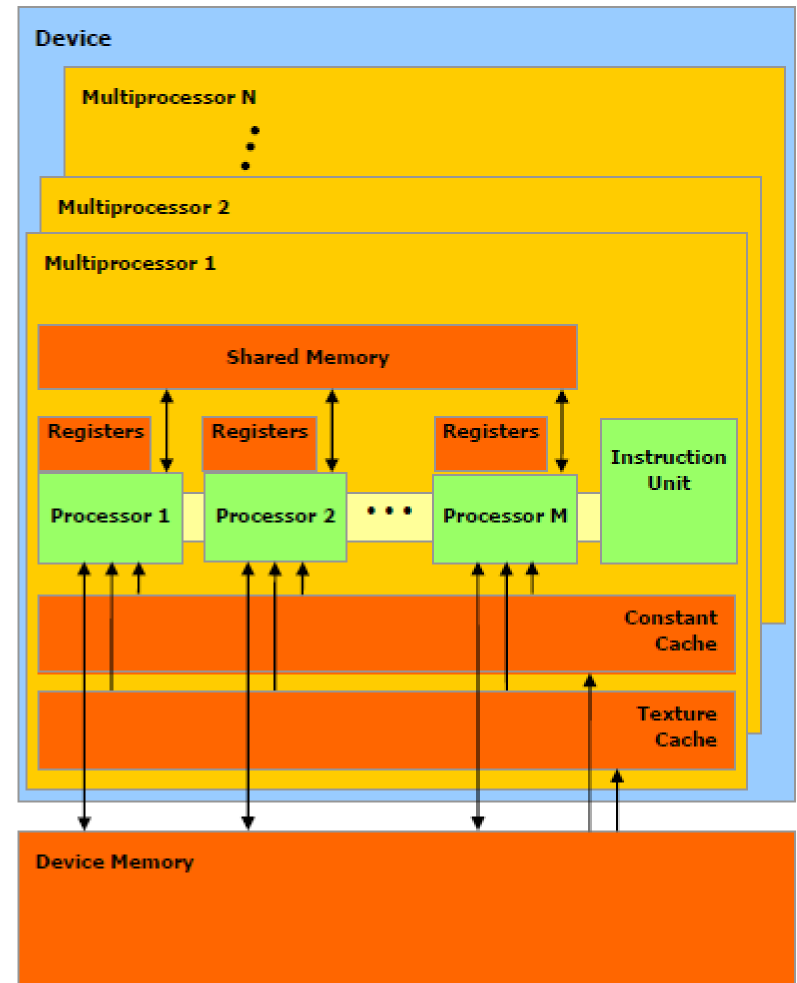
- Architectural design



  - Up to 30 **S**treaming **M**ultiprocessors (Fermi: 16 SMs)
  - **Per SM**: 8 **S**calar-**P**rocessors (Integer and Float operations), 16 KB shared-memory, “off-chip” DRAM (Fermi: 32 SPs, 64 KB RAM, L1/L2 cache hierarchy)

GPUs

- The memory hierarchy
  - Reading from device memory is about 100x slower than shared memory access
  - Read/write operations are always in chunks of 128 Bit
  - Memory accesses should be coalesced into a single contiguous aligned memory access.
  - Accesses of many threads to the same memory location are serialized
  - Constant memory is read-only



Courtesy: NVIDIA

GPUs

- The memory hierarchy

Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- Scalar variables reside in fast, on-chip registers
- Shared variables reside in fast, on-chip memory
- Thread-local arrays & global variables reside in uncached off-chip memory
- Constant variables reside in cached off-chip memory

GPUs

- The parallel execution model on the GPU
  - Instruction stream oriented vs. data stream oriented

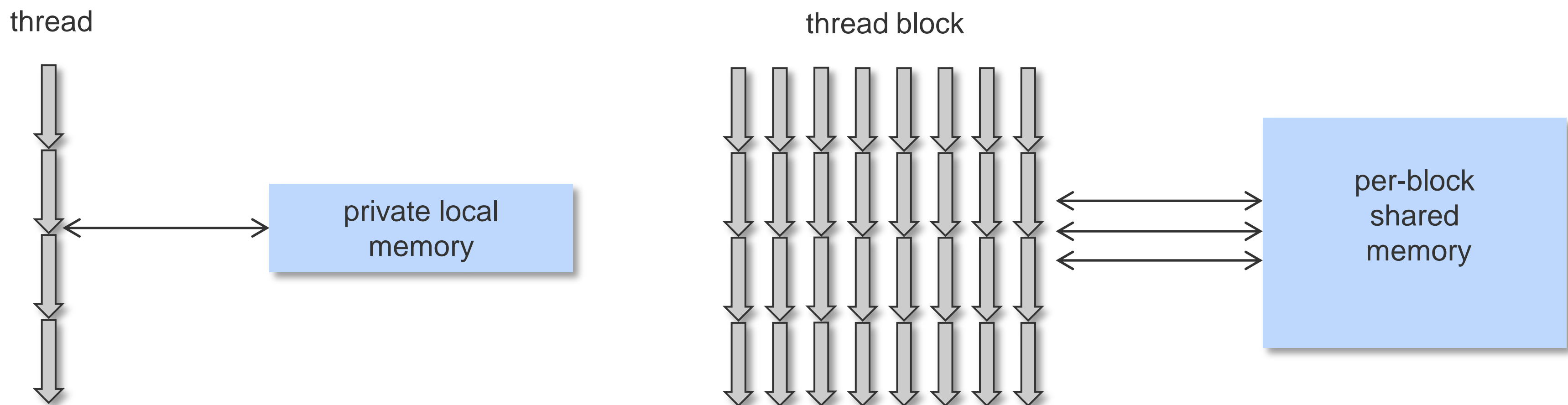
Addition of two 2D arrays: **C = A + B**

instruction stream processing data
optimized for running a single instruction stream fast
CPU
	for(y=0; y<HEIGHT; **y++**)
	for(x=0; x<WIDTH; **x++**)
	C[y][x] = **A[y][x]** + **B[y][x]**;

data streams undergoing a kernel operation optimized for running the same kernel concurrently on many simple cores **GPU**	inputStreams(**A,B);** outputStream(**C);** kernelProgram(OP_**ADD);** processStreams();

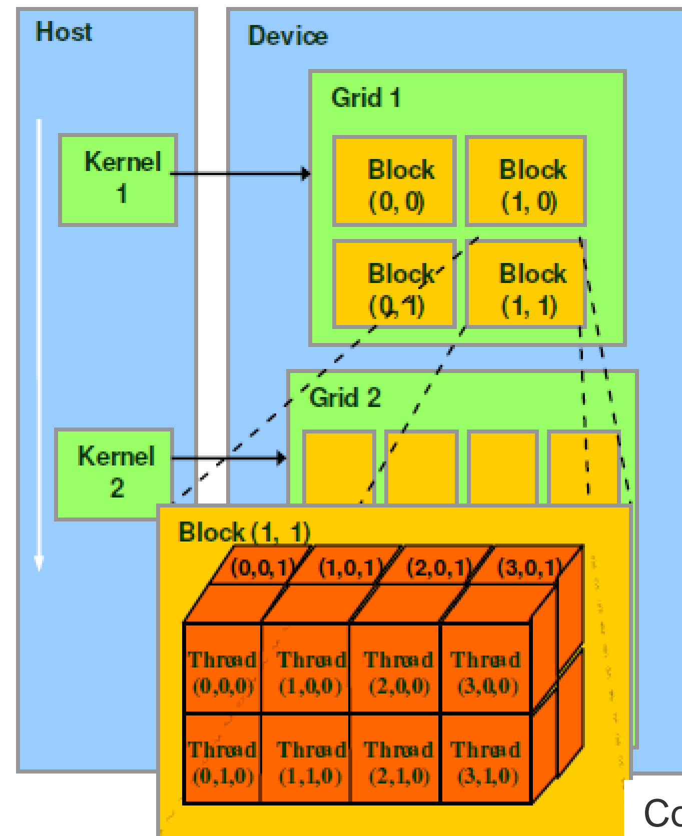
GPUs – the parallel execution model

- A GPU program calls kernels, which execute in parallel across a set of parallel threads
  - A thread has access to registers and thread private memory
- Threads are grouped into thread blocks
  - Threads in a block can communicate via shared memory



GPUs – the parallel execution model

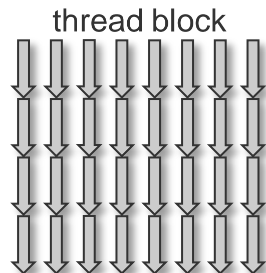
- Thread organization
  - Grids of blocks
  - One kernel per grid
  - Blocks of threads
  - Each of multiple dimensions

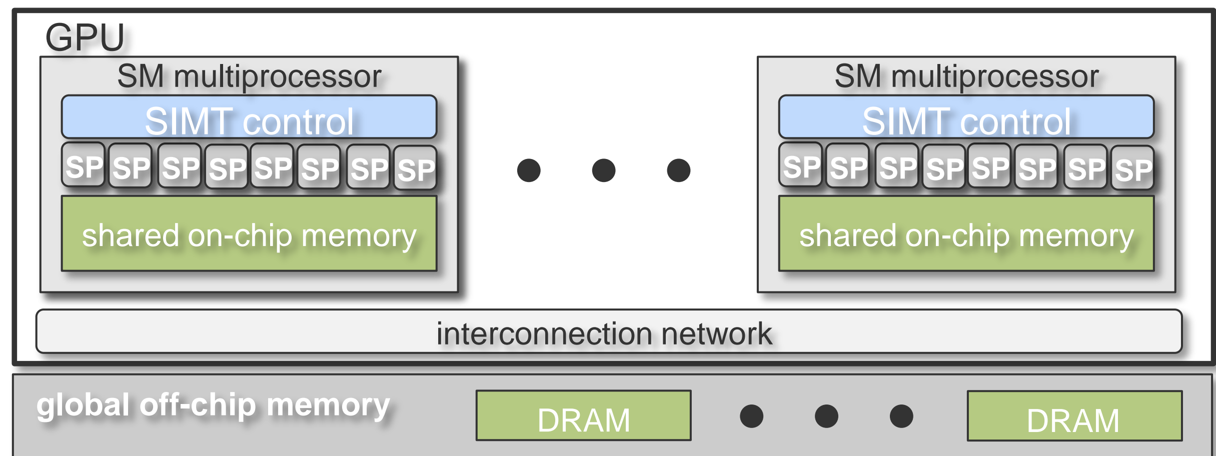


Courtesy: NVIDIA

GPUs – the parallel execution model

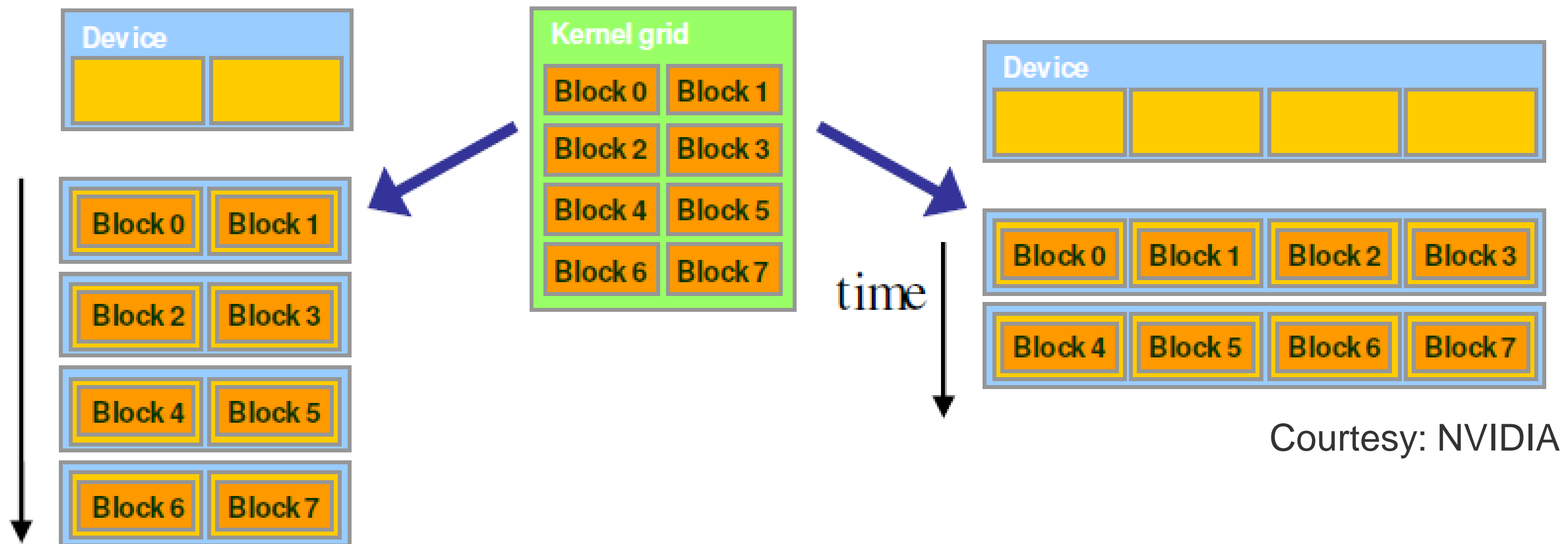
- Thread blocks are assigned to a SM
  - Threads run concurrently on a SM





GPUs – the parallel execution model

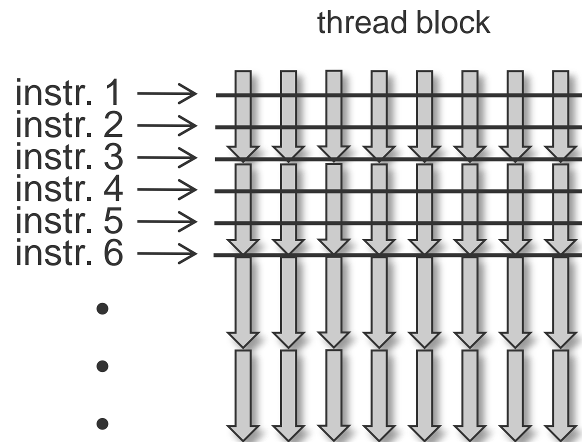
- Thread blocks are assigned to a SM
  - Threads within a block (but not across blocks) can be synchronized via barriers
  - Each block can execute in any order relative to other blocks.



Courtesy: NVIDIA

GPUs – the parallel execution model

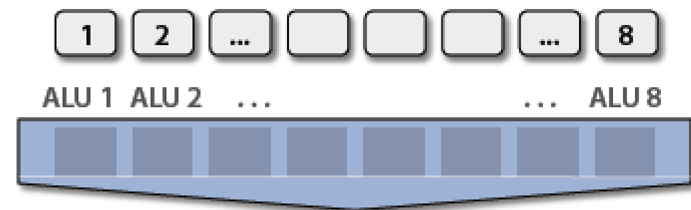
- Threads in a thread block execute in a SIMD fashion on each SM
  - Threads per group execute the same instruction stream with different data
  - Threads per group execute the instruction stream in lock-step, i.e., at an instance in time every thread's instruction pointer points to the same instruction
    - Avoids cost/complexity of managing an instruction stream across many SPs
  - Is most efficient if threads in a block execute the same code path



GPUs – the parallel execution model

- Execution divergence example

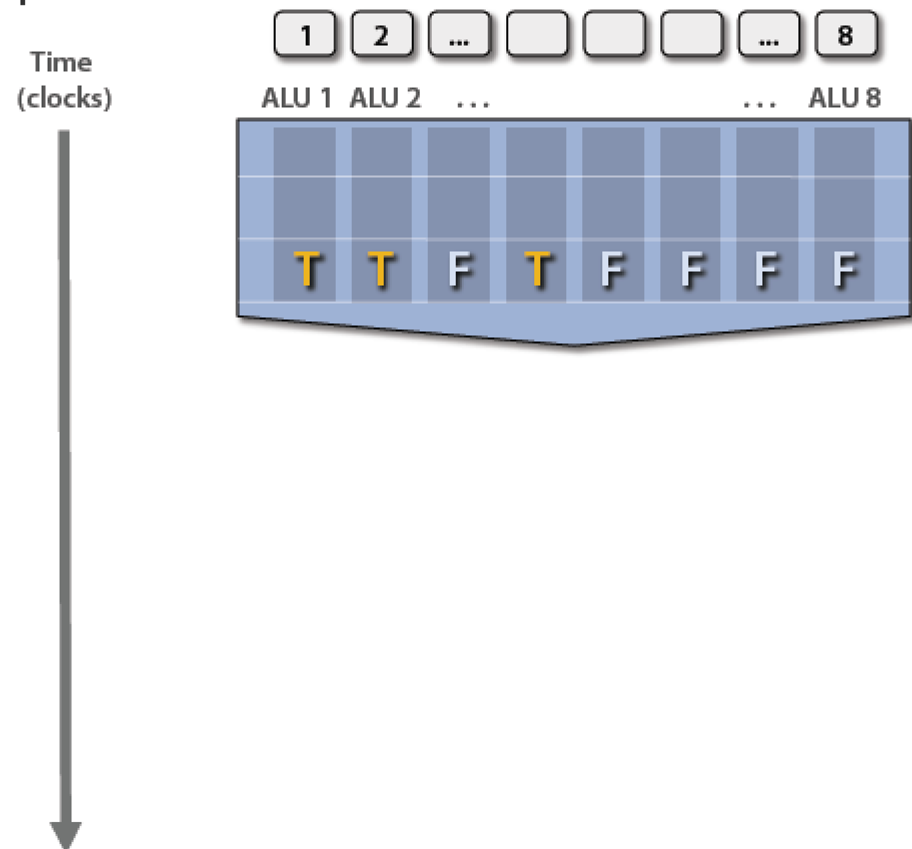
```
<thread kernel>


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

<end thread kernel>
```



GPUs – the parallel execution model

- Execution divergence example

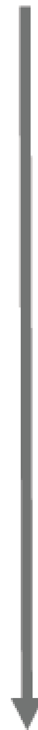
```
<thread kernel>

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

<end thread kernel>
```

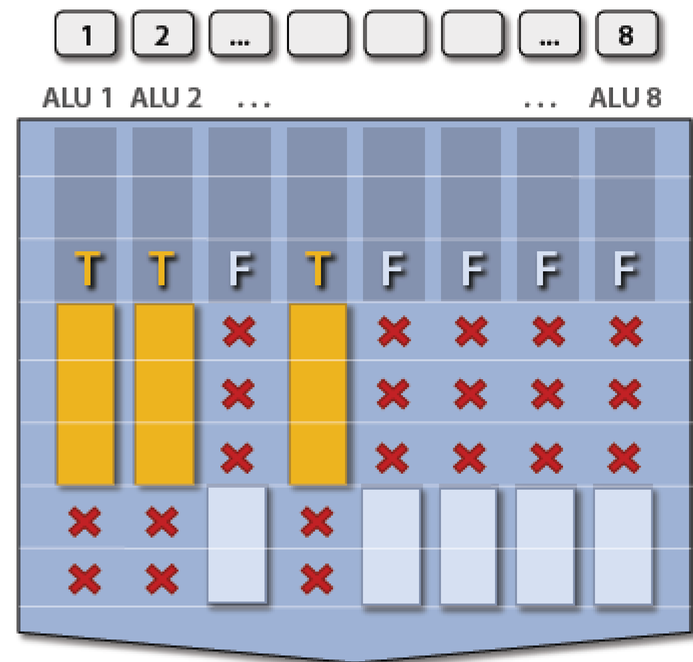


GPUs – the parallel execution model

- Execution divergence example

```
<thread kernel>

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

<end thread kernel>
```





Not all ALUs do useful work!
Worst case: 1/8 performance

GPUs – the parallel execution model

- Key to efficient GPU programming:
  - Divide the algorithm into many parallel tasks (threads) which can be executed in a SIMD-like fashion, thus avoiding execution divergence
  - Merge sort as an example:

```
merge(A, B) {

   if isEmpty(A) return B
   if isEmpty(B) return A
   if head(A) < head(B)
      return head(A) + merge(tail(A), B)
   else
      return head(B) + merge(A, tail(B))

} // end merge
```

GPUs – the parallel execution model

- Key to efficient GPU programming:
  - Divide the algorithm into many parallel tasks (threads) which can be executed in a SIMD-like fashion, thus avoiding execution divergence
  - Sorting via divide-and-conquer:

```
merge(A, B) {

   s = elementIn(A,B)

   A1 = {r∈A: r < s}
   A2 = {r∈A: r > s}
   B1 = {r∈B: r < s}
   B2 = {r∈B: r > s}

   return merge(A1,B1) + s + merge(A2,B2)

} // end merge
```

GPUs – the parallel execution model

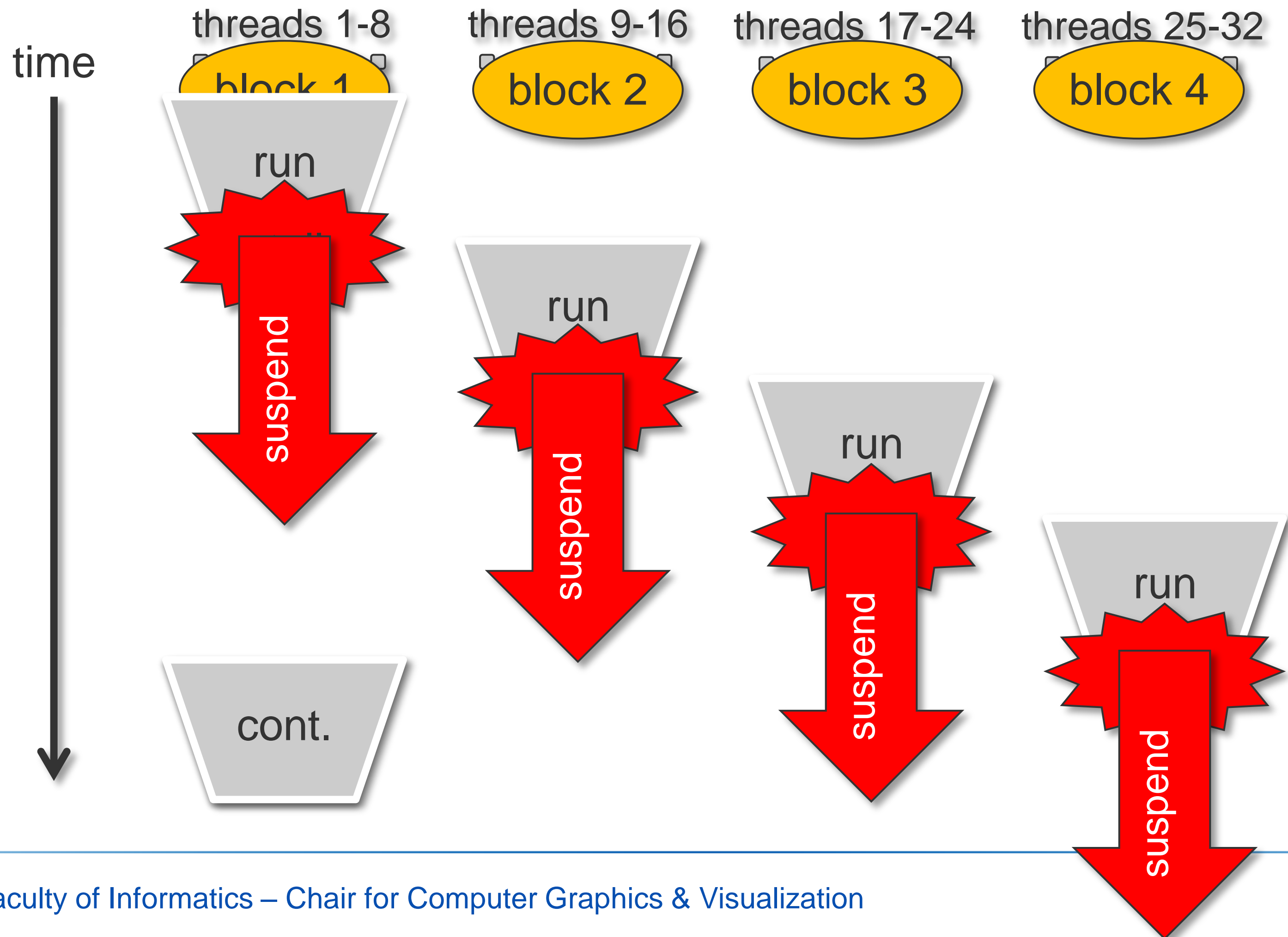
- Key to efficient GPU programming:
  - Divide the algorithm into many parallel tasks (threads) which can be executed in a SIMD-like fashion, thus avoiding execution divergence
  - Parallel sorting via divide-and-conquer :

```
merge(A, B) {

   S = {s1,. . ., sk} = sortedElementsIn(A, B)

   Ai = {r∈A: si-1 < r < si}
   Bi = {r∈B: si-1 < r < si}

   return merge(A1,B1) + s1 + . . . + sk + merge(Ak,Bk)
} // end merge
```

GPUs – the parallel execution model

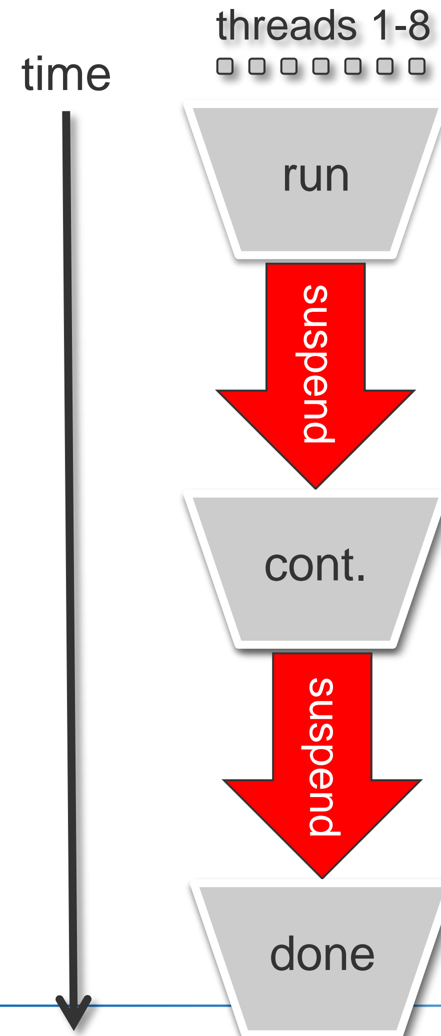
- The problem of memory access latency and stalls thereof
  - It may take many cycles to fetch an item from memory
  - Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation
  - Remember, that GPUs have removed the caches and logic that helps avoid stalls
  - **Solution**: interleave processing of many threads on a single core to avoid stalls caused by high latency operations

GPUs – the parallel execution model



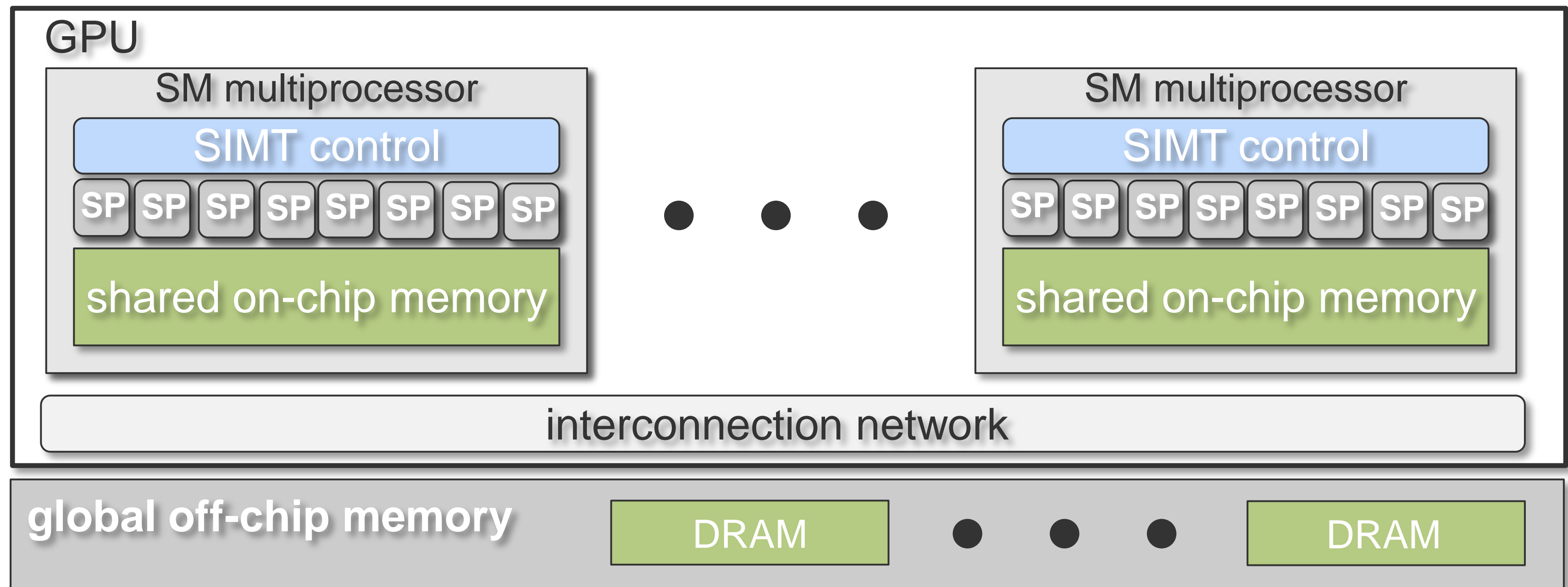
GPUs – the parallel execution model

- Increase run time of one group to maximize the throughput of many groups
- Parallel hardware multithreading swaps in/out many threads per millisecond
- But, threading eats away registers (context of swapped-out threads)



GPUs

- A parallel SIMT (Single Instruction Multiple Thread) architecture



  - Hardware multithreading (15G threads/second)
  - **Throughput oriented** – at the cost of higher latency of single processes