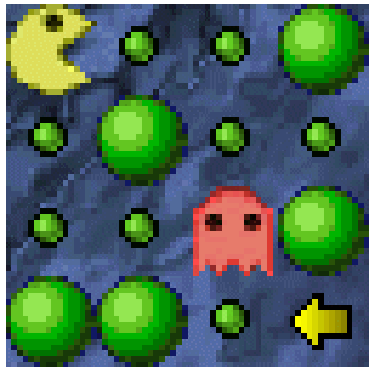
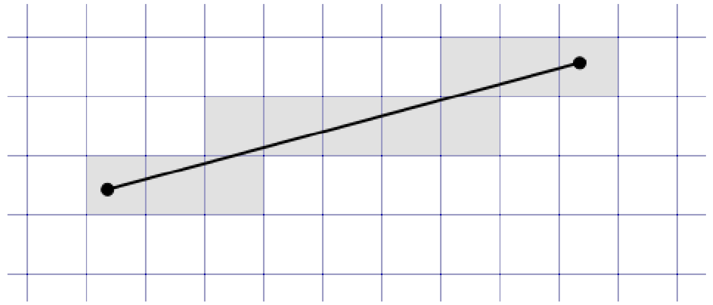
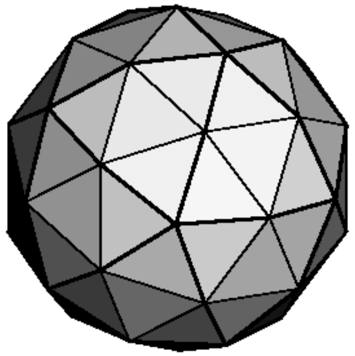
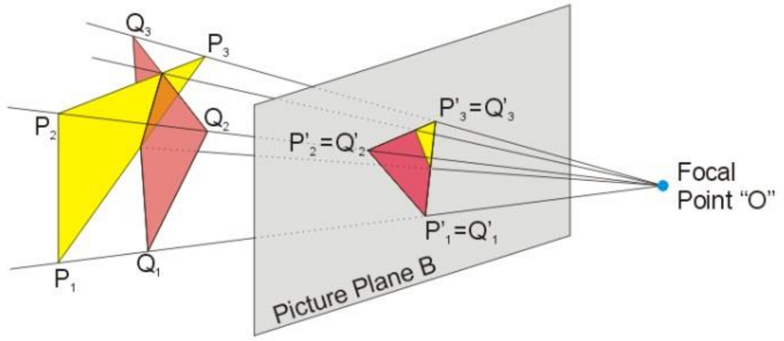
HPC: GPU

UROŠ LOTRIČ

History

Graphics accelerators

- 2D
  - bit-blit (block image transfer without blinking),
  - line graphics (Bresenham),
  - Clipping of unvisible part of image
- 3D
  - object is modelled with polygons which are projected to the screen
  - wire frame, invisible edges, painting, lightning, textures , shading
  - lots of mathematical operations: projections and rotations

History

Pixel shaders

- 2D acceleration
- Determine pixels colours

Vertex shaders

- 3D acceleration
- Perform geometric transformations to map a vertex to screen coordinates

Problem

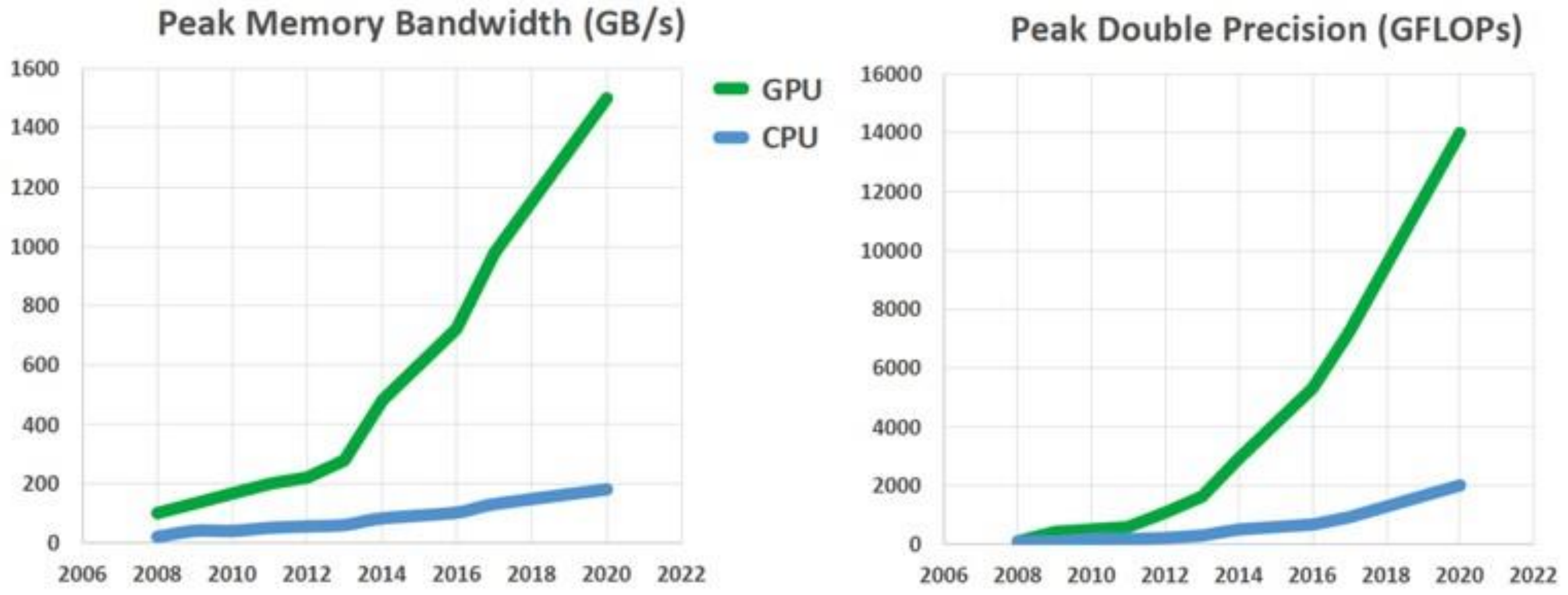
- Image editing involves pixel shaders
- CAD applications involve mainly vertex shaders

Solution

- CUDA (Compute Unified Device Architecture)
- Streaming processors: general shaders which can be used in 2D and 3D
- Added floating point operations, additional general purpose instructions

CPU vs GPU

Performance increase in last decade



CPU vs GPU

Complex control logic

Large caches

Optimized for serial operations

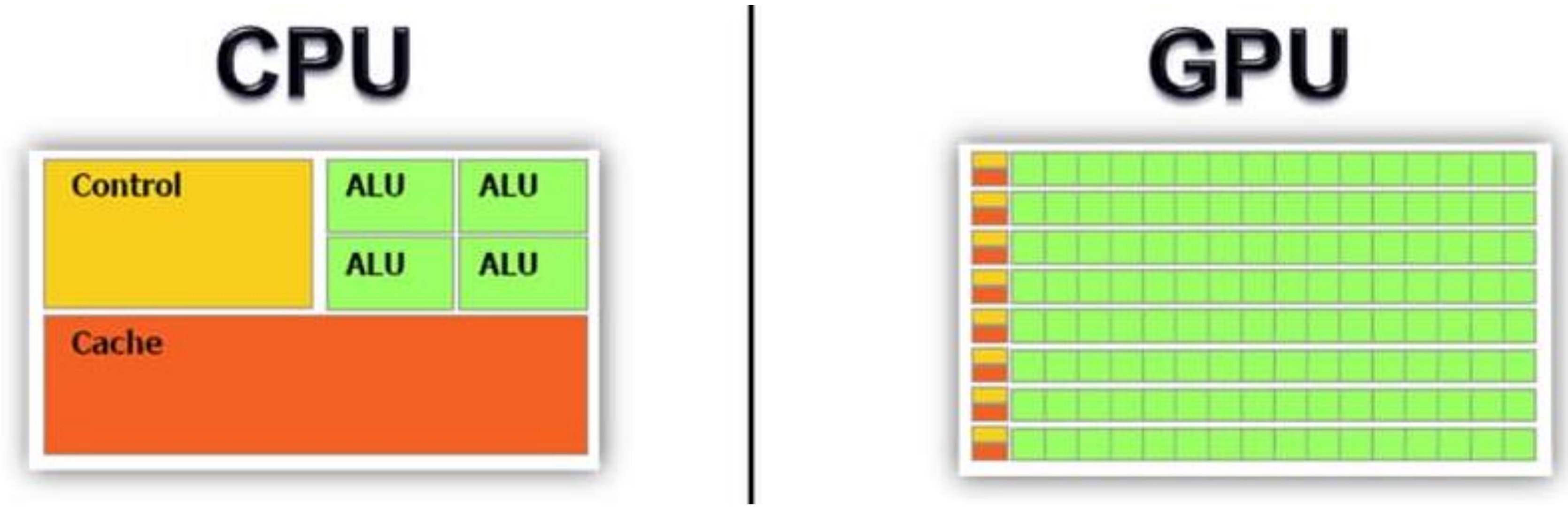
All types of applications, also tree operations, recursion

High portion of simpler processing units (ALUs), highly parallel

Build for parallel operations

High latency tolerance

Applications with high level of parallelism



GPU programming

Unusual programming model, conceptually different from CPU

Re-coding

- Some new approaches fight against it

Philosophy

- Create unlimited number of threads
- Threads are dynamically scheduled on hardware

Applications which excel on GPU

- High level of data parallelism
- Huge quantities of data
- 2D/3D structures with limited dependencies

GPU processor hierarchy

Designed to execute thousands of arithmetic operations simultaneously

To put a lot of processing power to one die

- We need a slimmer design
- All complex and large units are removed
  - Cache, branch predictor, out-of-order logic
- Control logic (fetch/decode) shared among ALUs
  - ALUs process the same instruction on different data
- Memory shared among ALUs to be able to exchange data

GPU processor hierarchy

Compute unit (multiprocessor)

- Basic computational building block
- Is equivalent to cores in CPU
- Is composed of many processing elements
- Entirely new instruction set, simpler for compiler, more constant performance
- SIMD parallelism
- Do not support branch prediction and speculative execution
- Have less cache then CPU
- Terminology: stream multiprocessor (Nvidia), SIMD engine (AMD)

GPU processor hierarchy

Processing element (core, shader processor)

- Is equivalent to ALU in CPU
- They share fetch/decode logic
- ALUs run the same instruction on different data
- Terminology: stream processor (Nvidia), ALU (AMD)

GPU processor hierarchy

Tens of compute units

Striving towards large number of PEs to efficiently hide memory-latency

- Completely different from CPUs where caches and out-of-order execution is used for latency hiding

Example

- Nvidia tesla K40m has 15 CUs with 192 PEs each

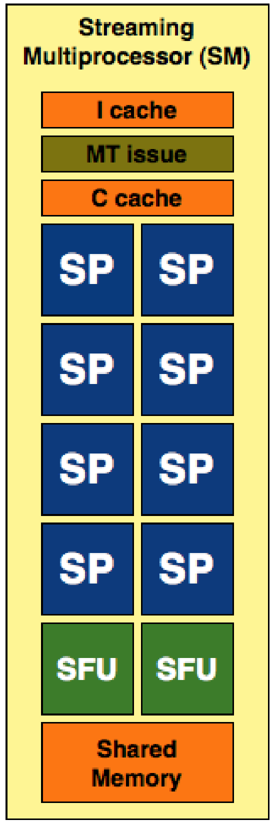
Evolution of Nvidia CU microarchitectures

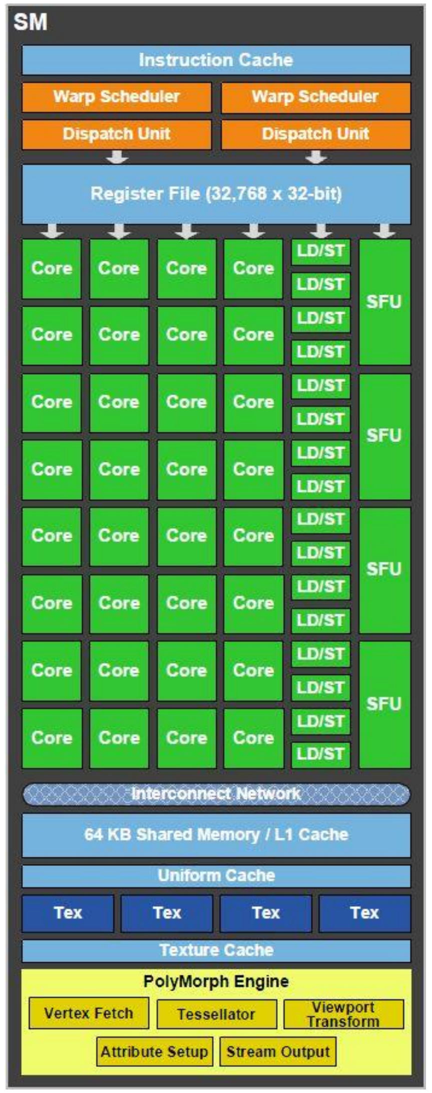
Tesla

- 8 PE (SP – Streaming processor)
- 2 SFU - Special Function Units
- 1 warp scheduler

Fermi

- 32 Pes (Cores)
- 2 warp schedulers
- 16 LD/ST units



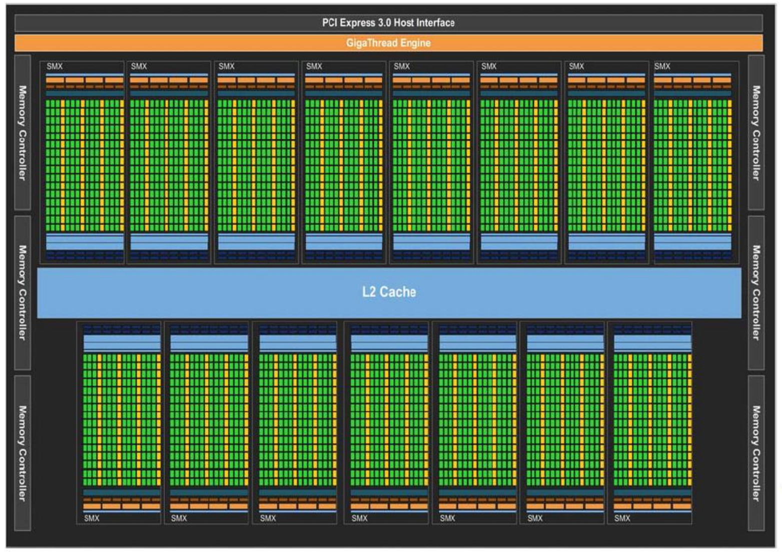


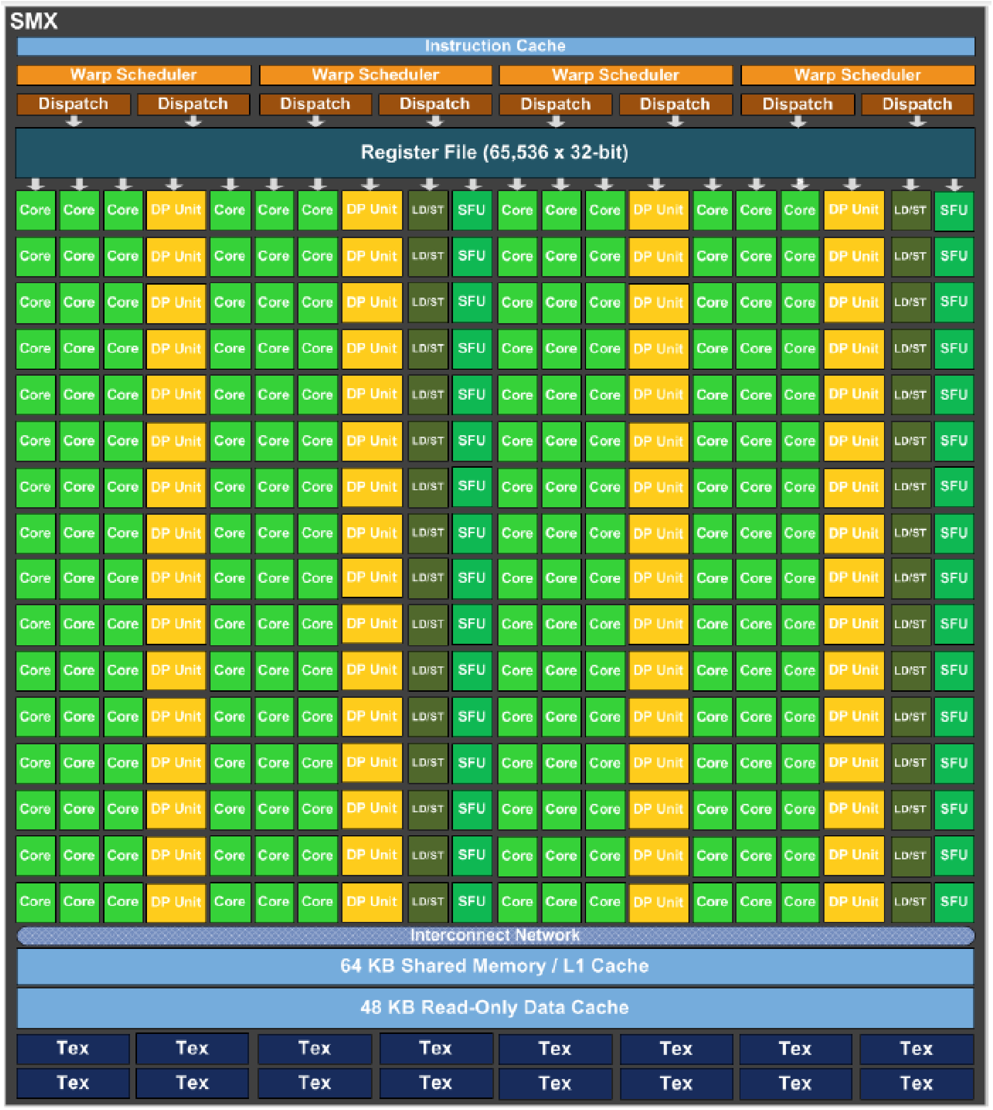
Evolution of Nvidia CU microarchitectures

Kepler

- 192 PE
- 32 SFU
- 32 LD/ST units
- 64 DP (double precsision) units
- 4 warp schedulers

- Tesla K40

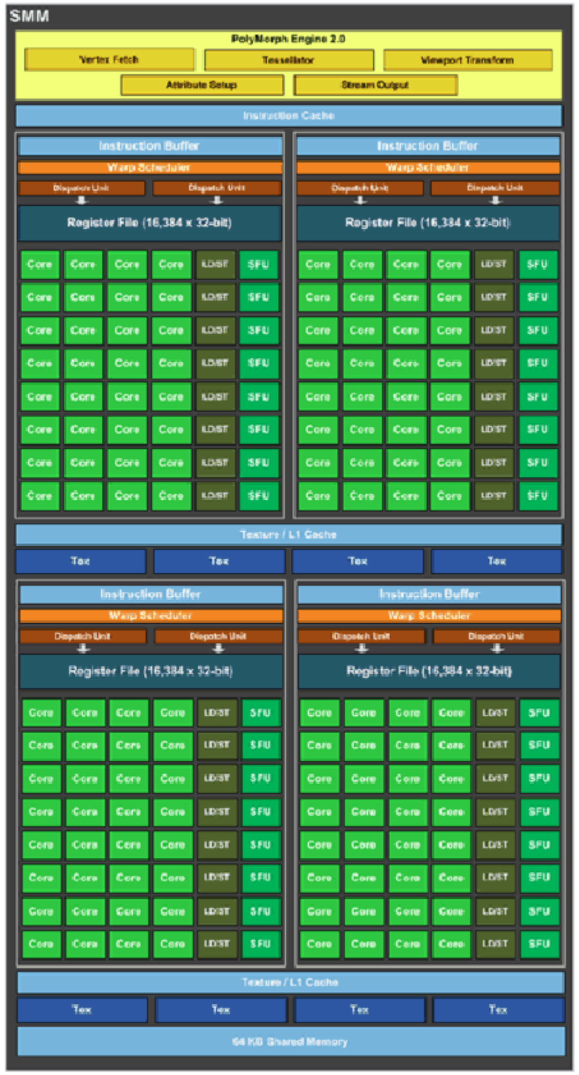


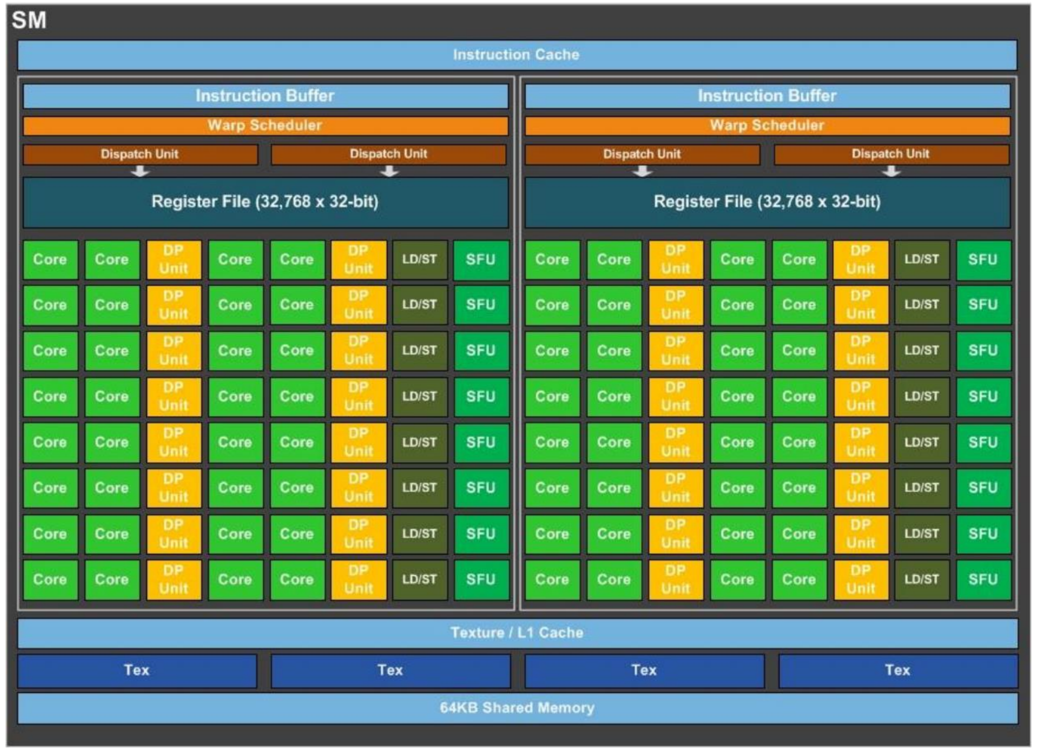

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Evolution of Nvidia CU microarchitectures

Maxwell
- 128 PE

Pascal
- 64 PE
- 32 DP units
- GPU-GPU memory transfers
- Half-precision

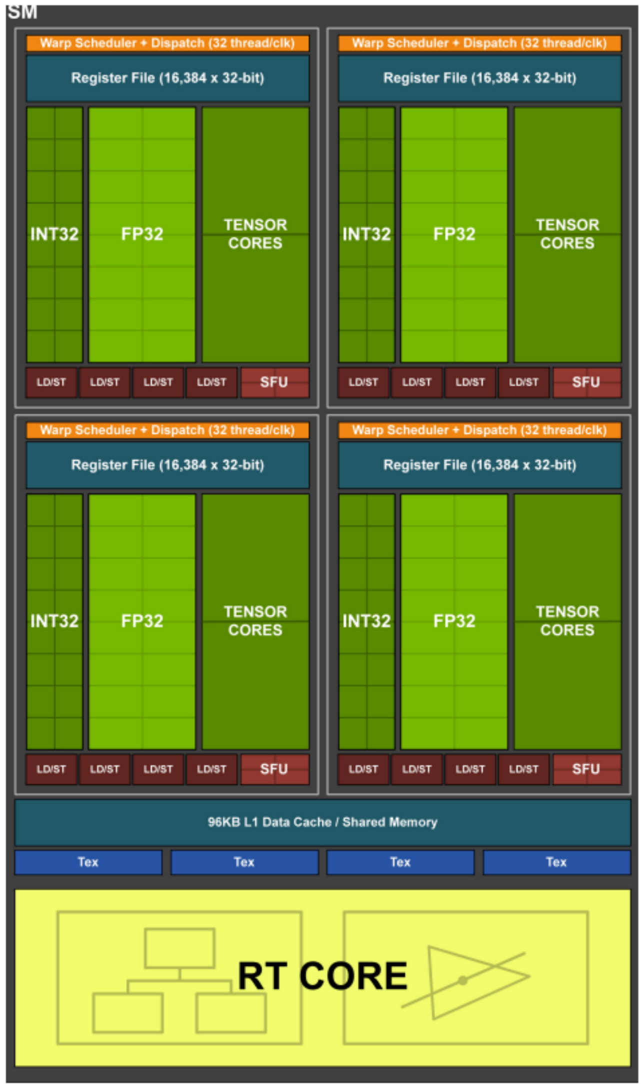




Evolution of Nvidia CU microarchitectures

Turing

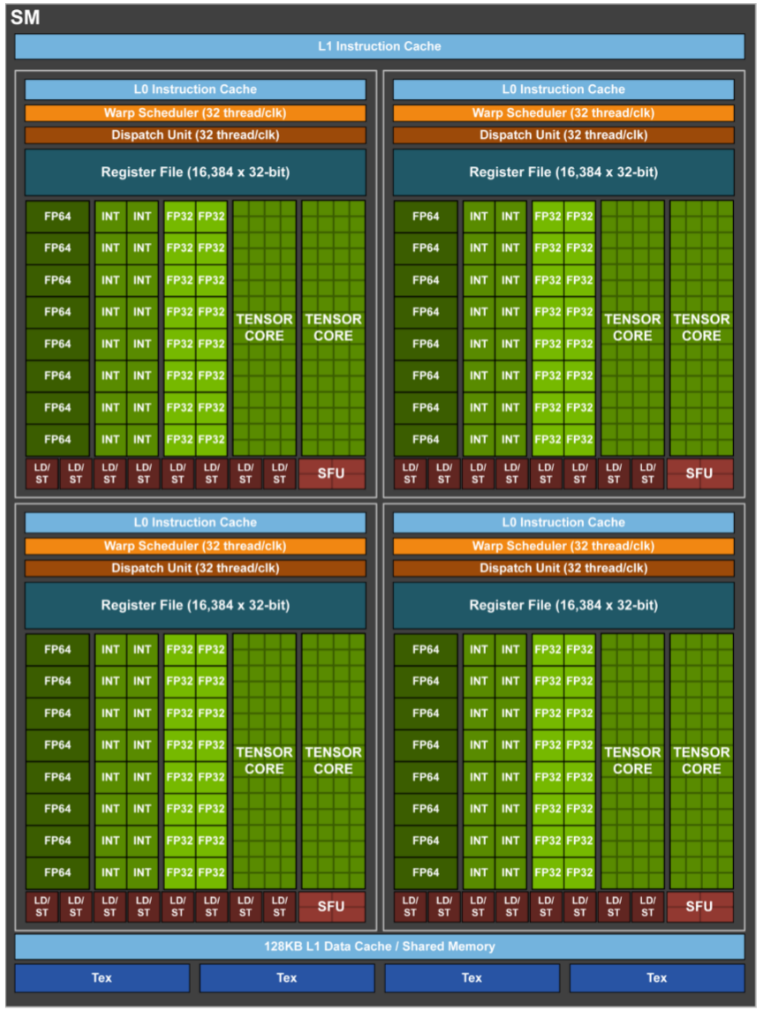
- 64 PE
  - int 32 bit, fp 32 bit
- 8 tensor cores
    - matrix operations
    - fp 16 bit
    - int 16, 8, 4 bit
  - RT core
    - Ray Tracing

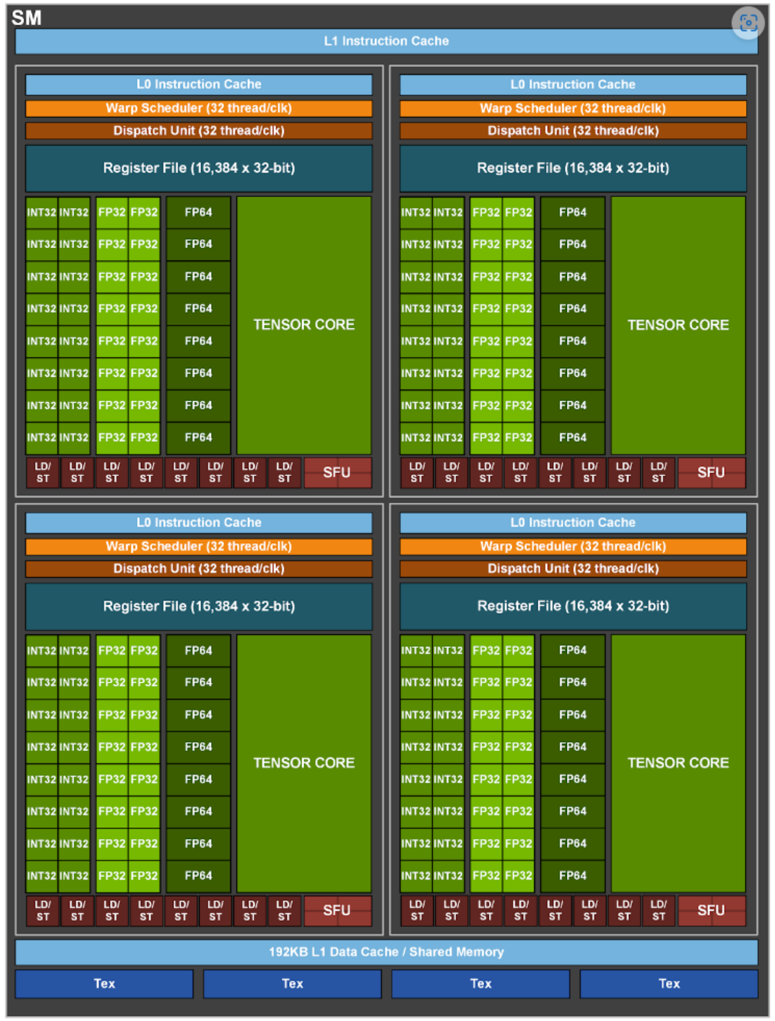

SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Evolution of Nvidia CU microarchitectures

Volta and Ampere

- More advanced tensor cores

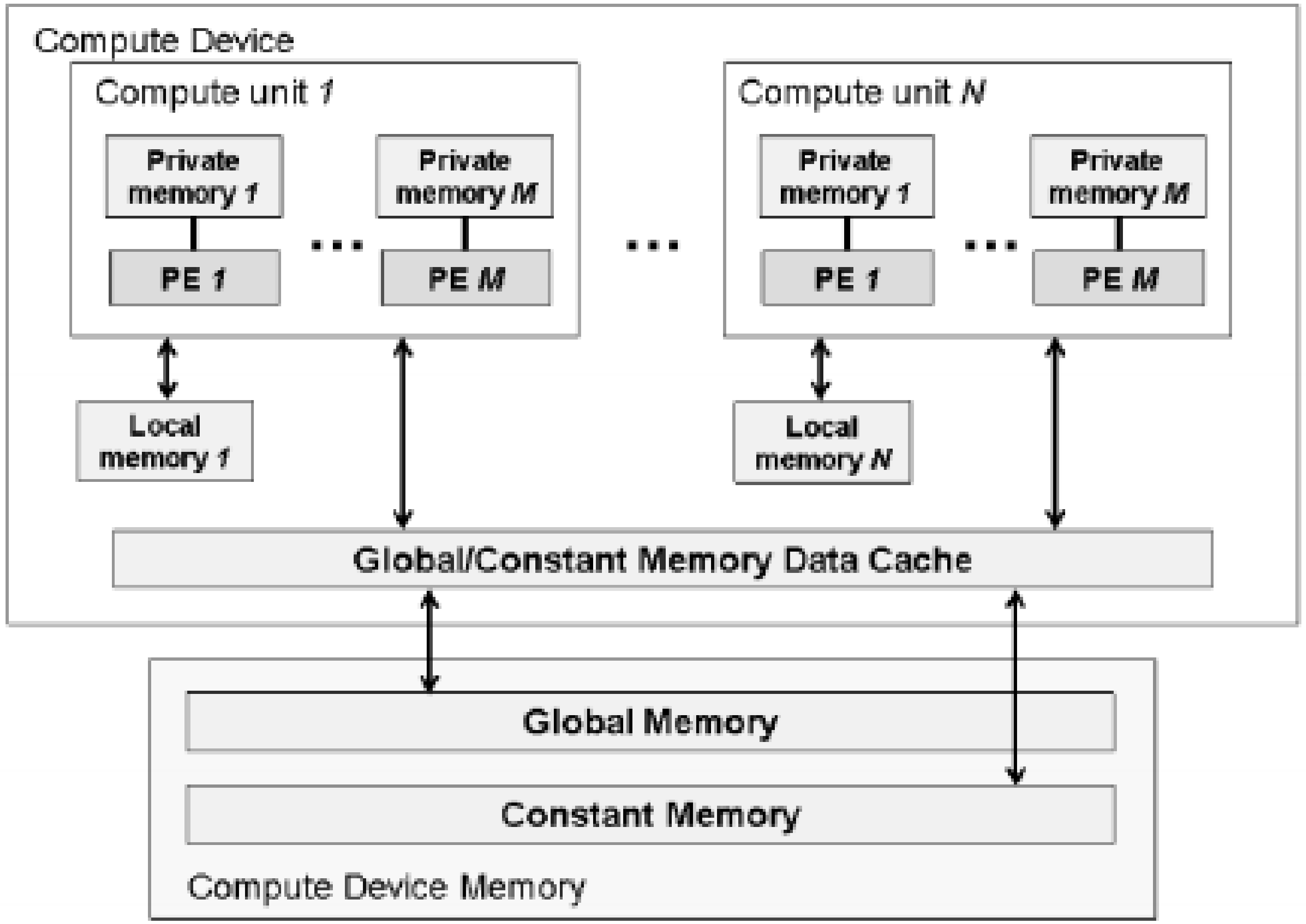

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Memory hierarchy

Compute unit

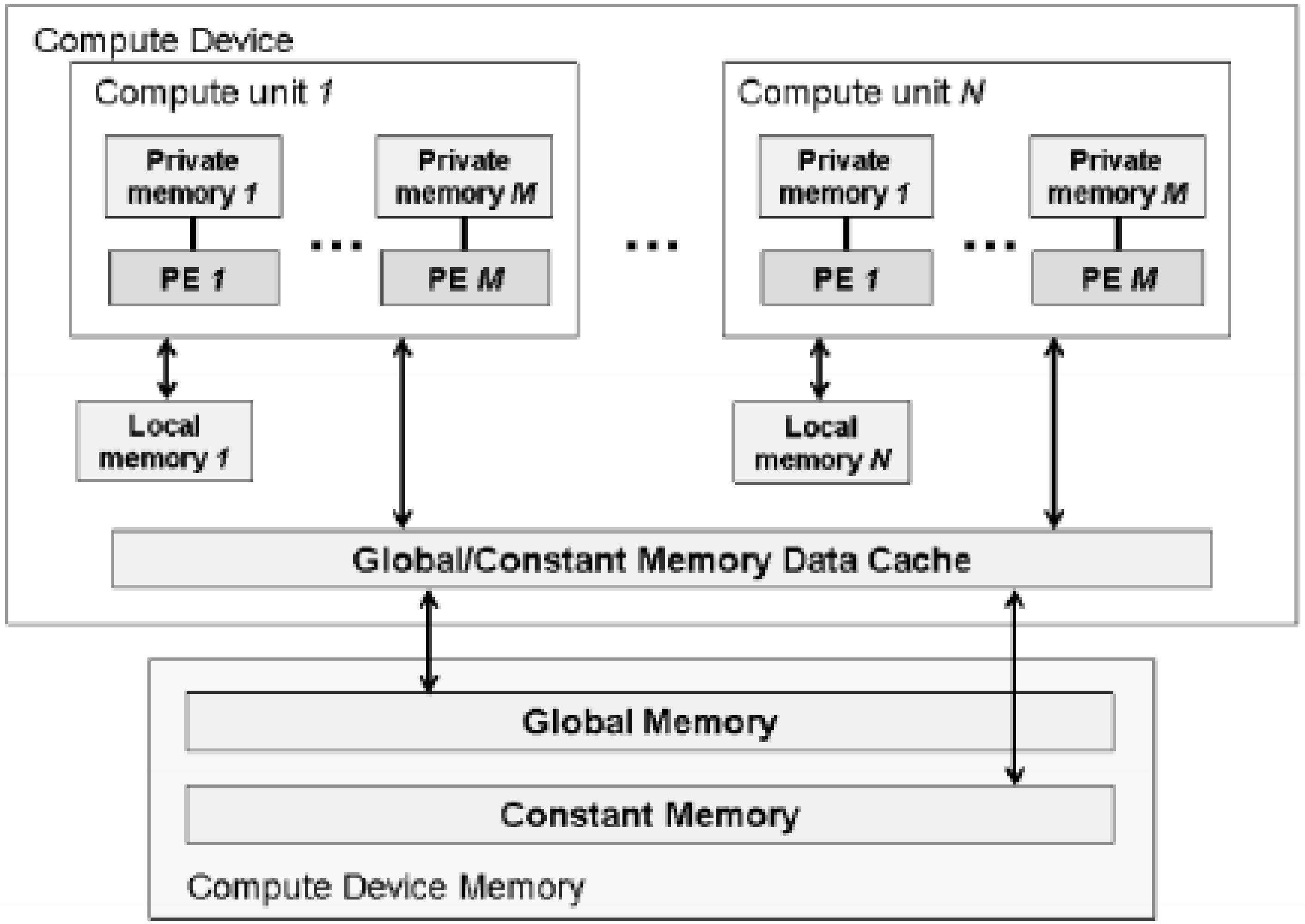
- Private memory (registers)
  - Kepler 64k x 32-bit
  - Divided among processing elements
  - Access time 1 cycle
- Local (shared) memory
  - Available to all processing elements
  - Kepler 64k
  - Access times 1 – 32 cycles



Memory hierarchy

Compute device

- Global Memory
  - Read and write
  - Nvidia K40m 12 GB GDDR5
- Constant Memory
  - Read only
- Both memories
  - Accessible from host and device
  - Cached
  - Access time ~ 500 cycles



Thread scheduling

Hierarchical organization that suits processor organization

Thread-block

- a group of threads

Thread

- Exclusive access to private memory (registers) and local memory
- Has access to the dedicated private and shared memory
- Has access to global and constant memory

Threads in a thread-block

- Execute on the same computing unit
- All access shared memory
- Can synchronize at barrier
- Threads in different thread-blocks cannot always synchronize (barrier)

Thread scheduling

Step 1: thread-block scheduling

- Thread-blocks execute independently of each other
- One or more thread-blocks can be assigned to one compute unit
- The order of thread-block execution
  - Determined by hardware scheduler
  - If there are more thread-blocks than compute units,
    some thread-blocks may not go to execution before other finish
- Number of threads per thread-block is specified by a programmer
  - Influences number of registers allocated to a thread

Thread scheduling

Step 2: thread scheduling within a thread-block

- Warps are groups of consecutive threads
  - 32 (Nvidia), 64 (AMD)
- Compute unit schedules warps to processing elements
- Threads in a warp execute the same program
  - SIMT
  - Are free to branch and execute independently, each thread has its own program counter
  - Warp executes one instruction at a time
  - In case of divergence, processing elements execute each path sequentially, masking work-items not in the path
  - For good performance we should avoid branching within a warp

Thread scheduling

Step 2: thread scheduling within a thread-block

- Latency hiding
  - Number of clock-cycles needed to issue next warp for execution
  - A warp can wait to get operands (memory) or that all work-items reach a barrier (synchronization)
  - Scheduler can execute any warp that is ready
  - Full utilization when a warp is ready in each clock-cycle (latency is completely hidden)
- Switching between warps has no cost
  - Warp execution context (PC, registers, …) is maintained on a compute unit for the entire warp lifetime

Thread scheduling and memory

Private memory (registers)

- Is equally split to all threads executing on a compute unit
- More threads per thread-block we have, less private memory belongs to each

Global memory

- Coalesced access
  - One segment (128 B) can be delivered in one transaction
  - To improve performance, threads in a warp should access contiguous elements in memory to minimize the number of transactions

Constant memory

- Supports broadcasting of a single value to all threads in a warp in one cycle

GPU programming frameworks

Nvidia CUDA

- CUDA C, only for Nvidia hardware
- Firmly tight to Nvidia hardware
- Installed in majority of HPC systems

OpenCL

- Supports GPUs of different vendors
- Supports also CPUs, FPGAs, …
- Does not have so many features as CUDA
- OpenCL C to write kernels – C-like functions executed by work-items

New approaches

- Intel One API
- OpenMP 4.5

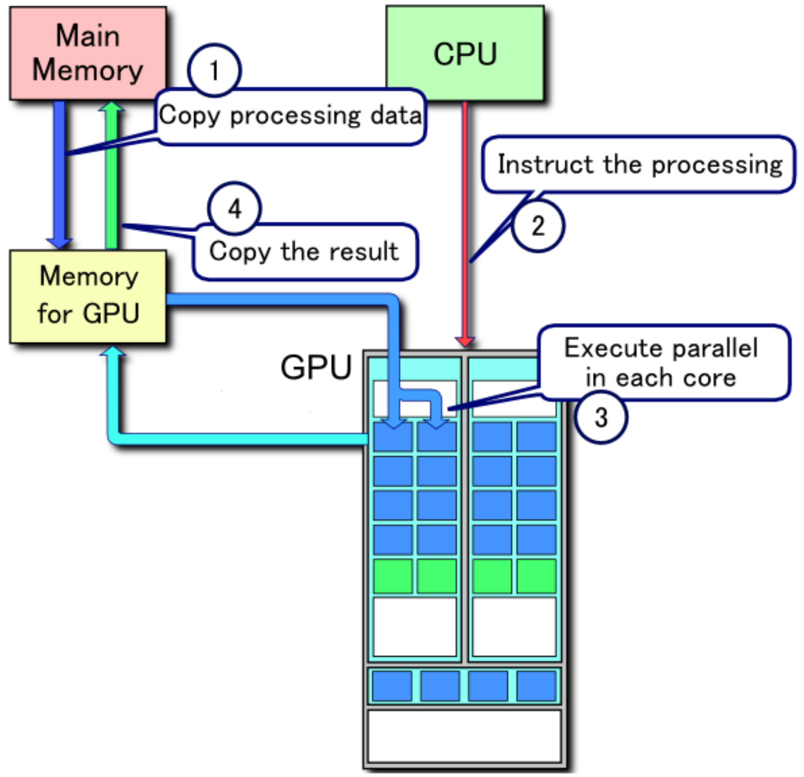
Execution model

Offload model

- Host copies data to device
- Host triggers execution on device
- Device executes program (kernel) in parallel
- Host transfers results from device

Host executes serial code

Device executes parallel code

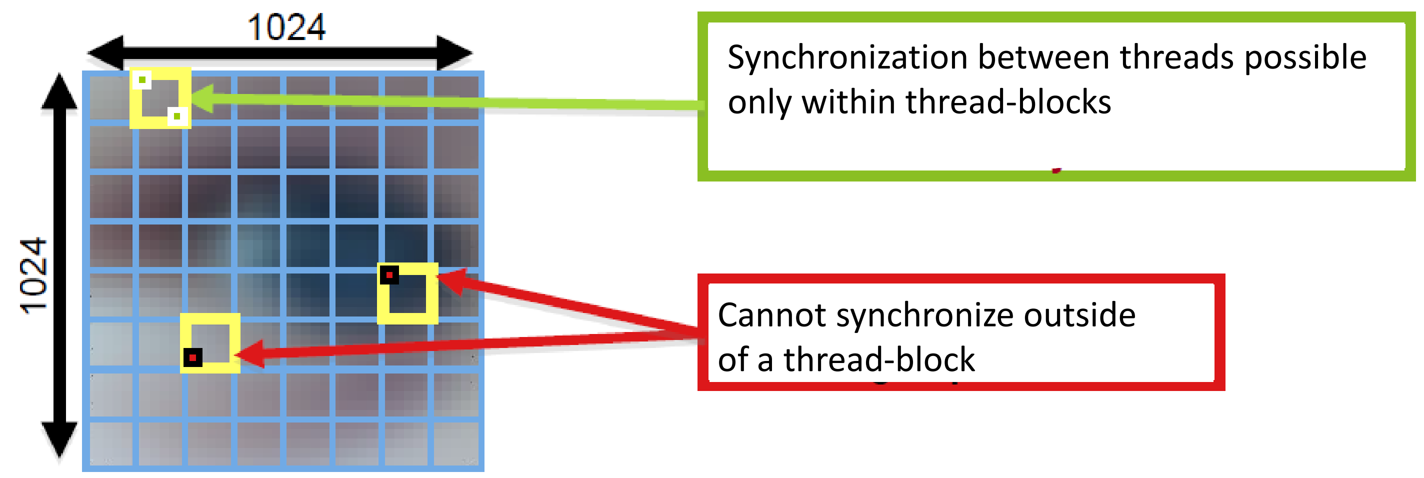


Programming models fit to the hardware hierarchy

Thread organization

Problem description

- Number of thread-blocks and number of threads within a thread-block determines problem size – how many threads will execute the kernel
- 1D, 2D, or 3D thread index space
- Each thread executes the same kernel for one point in the index space
- Synchronization is only possible among threads in a thread-block!


1024
1024
Synchronization between threads possible only within thread-blocks
Cannot synchronize outside of a thread-block

Thread organization

Kernel execution

- Kernels are functions executed on a device
- Kernel call is specified as

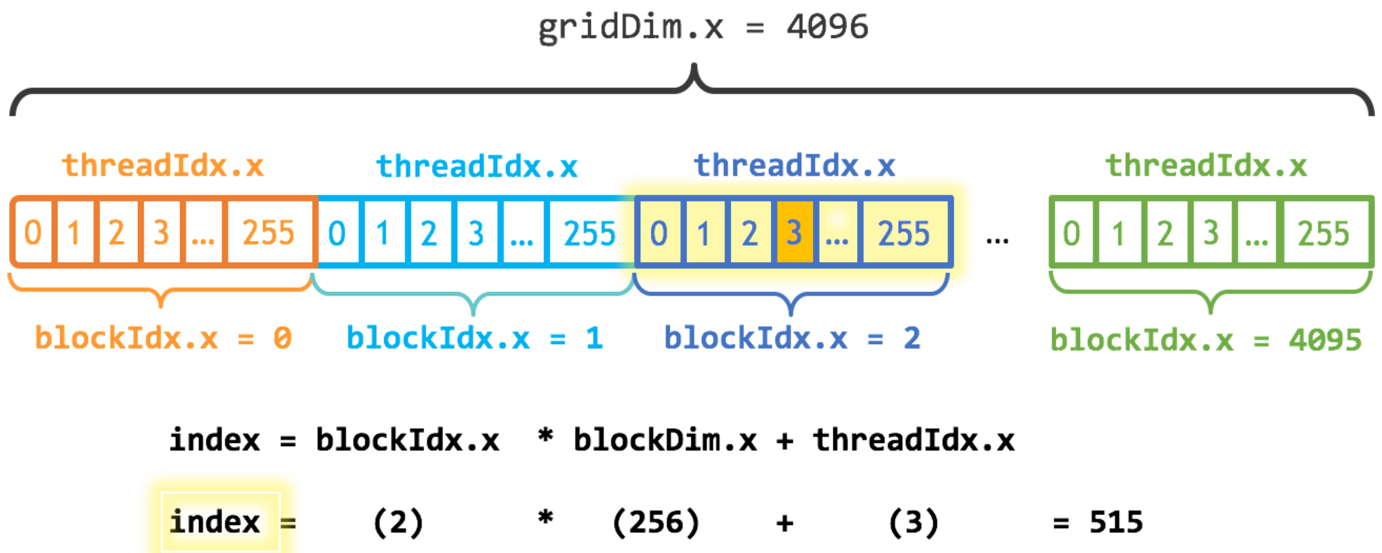
```
kernel_function<<<gridSize, blockSize>>>(arguments)
```

  - gridSize – number of thread-blocks
  - blockSize – number of threads in a thread-block
- Declaration of gridSize and blockSize
  - `dim3 gridDim(x,y,z)`
  - `dim3 blockDim(x,y,z)`

Thread organization

Thread organization

- Figure:
  Example of 1D indexing



- Variables in kernel, which describe thread organization
  - threadIdx.x, threadIdx.y, threadIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - blockIdx.x, blockIdx.y, blockIdx.z
  - gridDim.x, gridDim.y, gridDim.z
- Warps are formed by consecutive threads in x-dimension, followed by y-dimension and z-dimension

Kernel

Cuda C

- Kernel function must be preceded with qualifier __global__
  - Kernel function always returns void
  - It cannot return a value to a host (CPU) as it is executed on a different hardware (GPU)
- Functions preceded with the qualifier __device__ execute only on device
  - They can be called by kernel function
- Functions preceded with the qualifier __host__ execute only on host
- Memory qualifiers
  - __shared__ defines memory structure shared among all threads in a thread-block
  - __constant__ for usage of constant memory

Examples

deviceinfo.c

saxpy.c

- sum a*x plus y
- Computation of **y** = a***x** + **y**
- a is scalar, **x** and **y** are vectors

```
for (size_t i = 0; i < n; ++i)
  y[i] = a * x[i] + y[i];
```