matfyz

Computer Graphics Charles University

GPGPU and CUDA

© 2012-2018 Josef Pelikán, Jan Horáček
CGG MFF UK Praha

pepca@cgg.mff.cuni.cz
http://cgg.mff.cuni.cz/~pepca/

Content

- advances in hardware
  - multi-core vs. many-core
  - general computing on GPU
- **CUDA** basics
  - architecture: streaming processor, streaming multiprocessor, thread warp, memory
  - C for CUDA
  - thread hierarchy, storage types, ..
- **OpenCL** vs. CUDA

Hardware advances

- **VLSI technology** evolves very quickly
  - around 2003 CPU frequencies froze (heat dissipation)
  - no more than ~3.4GHz clock
- current trend – **multi-core approach**
  - parallelizable tasks
  - changes in programming techniques (traditionally all the algorithms used to be sequential)
  - "**high-performance computing**" (HPC) community practices are becoming common in consumer segment

GPU advances

- **graphic hardware evolution**
  1. single-purpose
  2. configurable
  3. programmable
- currently **almost arbitrary algorithm** can run on a GPU
  - code size and memory are the only limits
- but **GPU is efficient** for special class of algorithms only

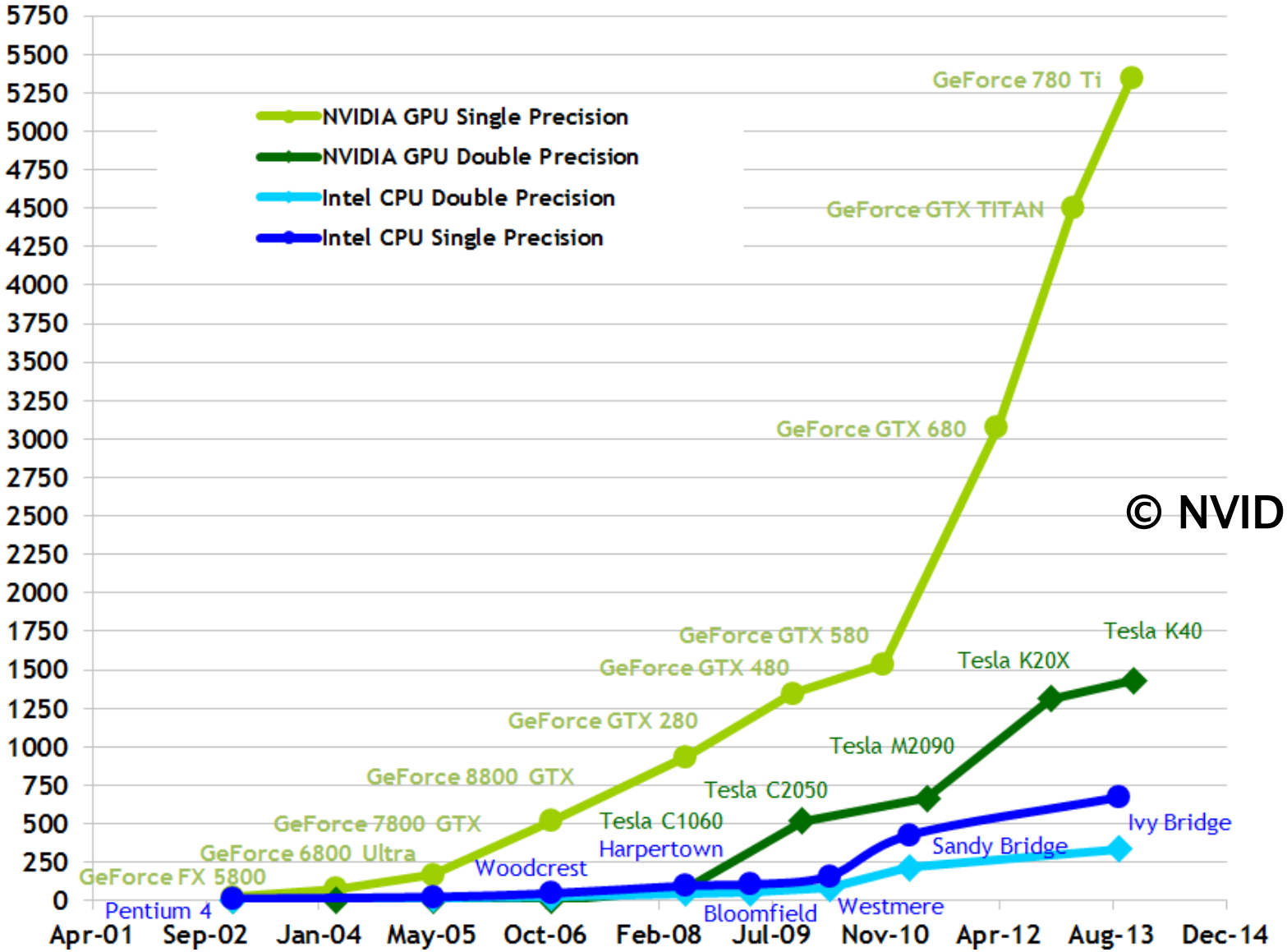
Multi–core computing

- running many **sequential algorithms**
- two- to many-core CPUs
  - ~eight to eighteen (tenths of cores)
- full **x86-64 instruction set**, **SSE**, **AVX** extensions
- example: **Intel Core i9** (Sky Lake X, Sep 2017)
  - 14 nm, 24MB L3 cache, TDP: 165W
  - *18 cores* + *hyperthreading* = 36 computing threads
  - *out-of-order* execution

Many-core computing

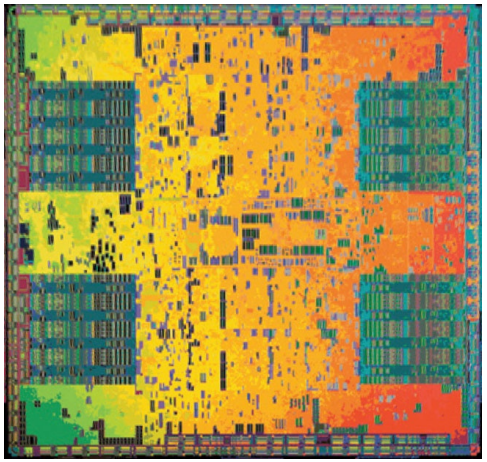
- emphasis on **parallel algorithms**
- graphic accelerators → **programmable GPUs**
- more simple instruction sets
- very high brute **computing power**
- example: **NVIDIA Quadro GP100** (Pascal, 2017)
  - 16 nm, 3584 cores, 1476MHz, TDP: 235W
  - 16 GB HBM2 RAM (4096-bit bus)
  - each core: *heavily multi-threaded*, *in-order*, *single-instruction issue processor*

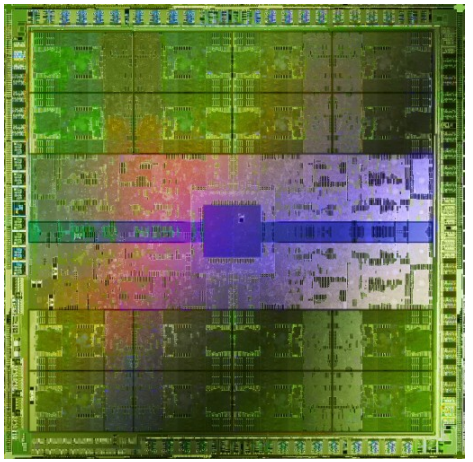
Theoretical GFLOP/s



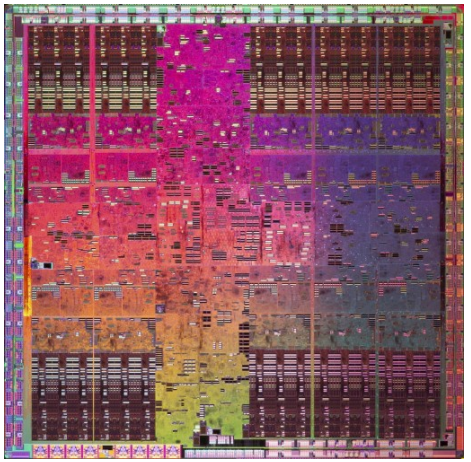
© NVIDIA Cuda

GPU vs. CPU, chip comparison I

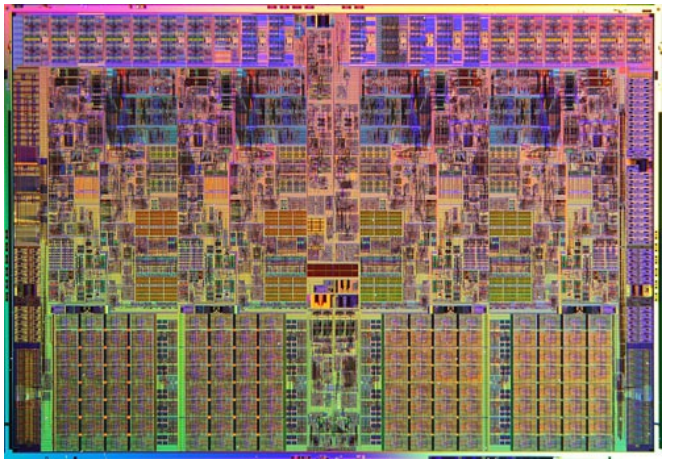




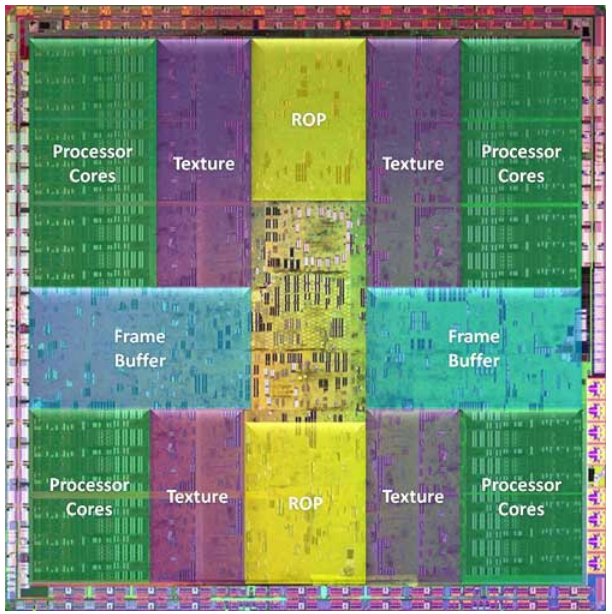
NVIDIA GT200



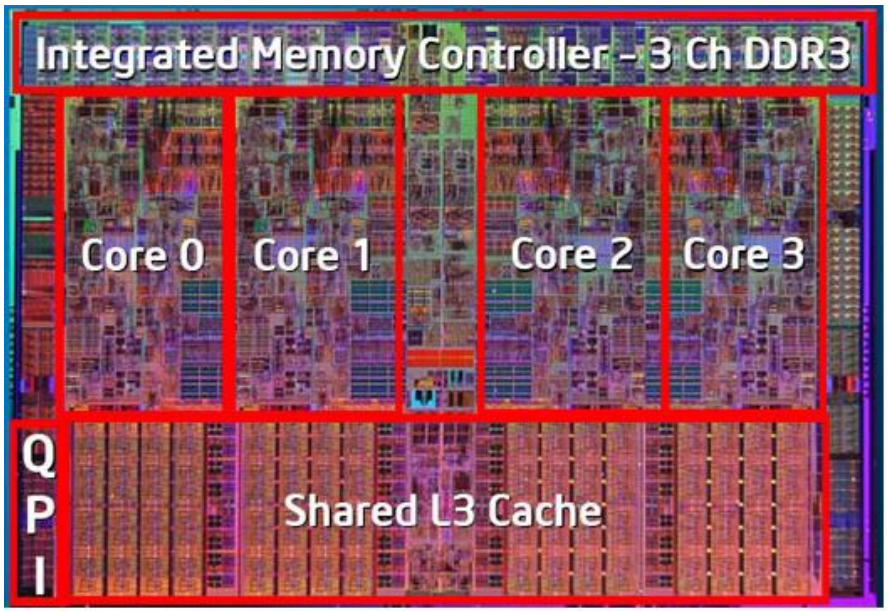
Intel i7



GPU vs. CPU, chip comparison II

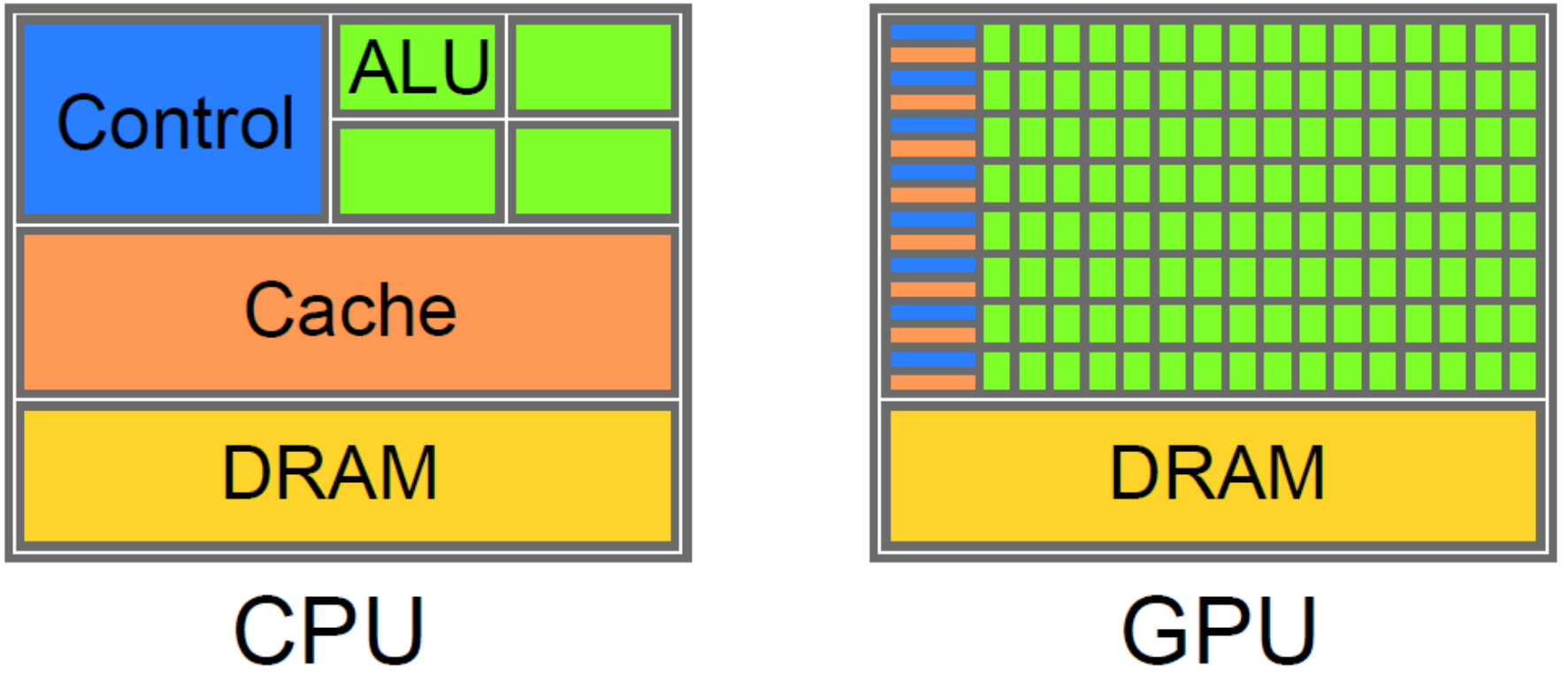


NVIDIA GT200



Intel i7

GPU vs. CPU, scheme



GPU vs. CPU

- **CPU**
  - large chip area used for **caches**
  - **low latency** is preferred
- **GPU**
  - graphic algorithms usually coherent → **hidden latency**
  - **FPU operations** are crucial for pixel fill-rate → big area dedicated to FPU units
  - **memory transfer rate** is preferred
  - does not need very much cache

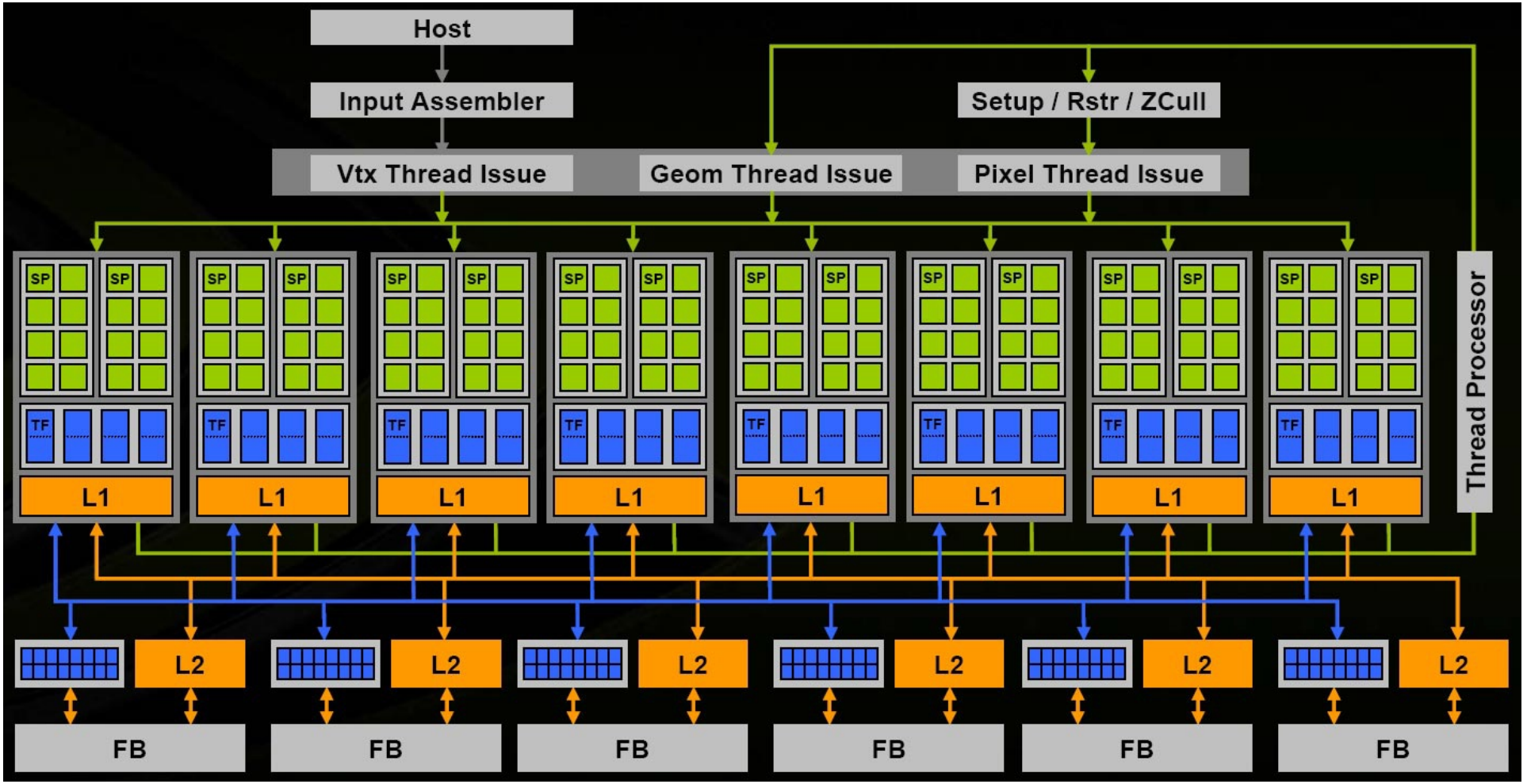
CUDA

- **GPGPU** (history) – computing in shaders, data in textures
  - more elegant access to GPU resources was needed
- 2007: **NVIDIA Tesla** architecture (G80, GeForce 8800)
  - general model for parallel programming
  - computing thread hierarchy
  - barriers for synchronization
  - atomic operations
- **CUDA:** "Compute Unified Device Architecture" (Barracuda)
  - **C for CUDA:** programming language

CUDA architecture

- CUDA GPU: group of **highly threaded streaming multiprocessors (SM)**
- set of SM form a **block**
  - number of **SM per block** depends on GPU generation
- each SM has a set of **streaming processors (SP)**
  - G80: 8 SP per SM
  - GF100: 32 SP per SM
- SP within one SM **share control circuits**, **instruction decoder** and **instruction cache !**
- **SIMT** (Single Instruction Multiple Threads)

G80 architecture

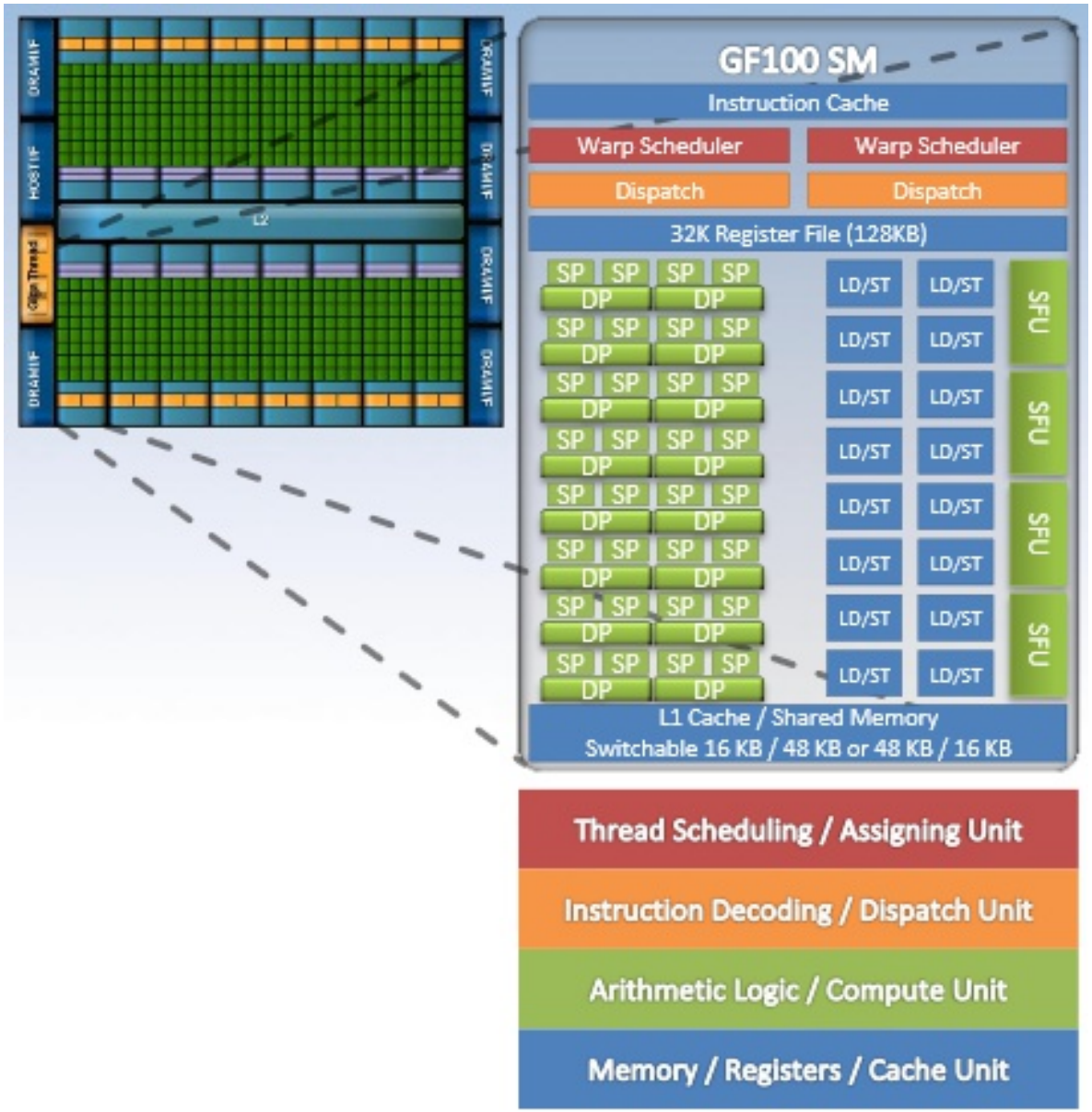


GeForce 8800

GF100 architecture (Fermi)

GeForce 480

GF100 streaming multiprocessor



Compute capability

Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21	sm_30	sm_35
Threads / Warp	32	32	32	32	32	32	32	32
Warps / Multiprocessor	24	24	32	32	48	48	64	64
Threads / Multiprocessor	768	768	1024	1024	1536	1536	2048	2048
Thread Blocks / Multiprocessor	8	8	8	8	8	8	16	16
Max Shared Memory / Multiprocessor (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
Register File Size	8192	8192	16384	16384	32768	32768	65536	65536
Register Allocation Unit Size	256	256	512	512	64	64	256	256
Allocation Granularity	block	block	block	block	warp	warp	warp	warp
Max Registers / Thread	124	124	124	124	63	63	63	255
Shared Memory Allocation Unit Size	512	512	512	512	128	128	256	256
Warp allocation granularity	2	2	2	2	2	2	4	4
Max Thread Block Size	512	512	512	512	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
[note: default at top of list]					16384	16384	16384	16384
							32768	32768
Warp register allocation granularities					64	64	256	256
[note: default at top of list]					128	128		

Data parallelism

- real world applications … **huge data**
  - FEM methods (big systems of equations)
  - particle simulations (mutual interactions)
  - Monte-Carlo simulations (massive sampling)
- **data parallelism** = many operations can be computed **simultaneously** on the same data set
- academic example: matrix multiplication

CUDA program structure I

- one or more phases (one phase = one “kernel”)
- deployment on CPU or GPU
  - CUDA source code can contain both parts
- **nvcc** (NVIDIA C compiler) separates these parts during compilation
- language: from **ANSI C** to ~complete **C++ support**
- GPU code = “**kernel**”
  - deployed on **thousands** of threads
  - GPU threads are much more **light-weighted** (thread creation, ctxsw ~ a couple of machine cycles)

CUDA program structure II

- keywords: `__global__`, `__host__`, `__device__`
  - where the code can run
  - from where it can be called

Keyword	Runs on	Called from
`__global__` float KernelFunc()	GPU	host
`__device__` float DeviceFunc()	GPU	GPU
`__host__` float HostFunc()	host	host

- `__host__` and `__device__` generate two versions (CPU & GPU)

CUDA parallelism I

- CPU: run **one** thread

```
// CPU grid computing
for ( int y = 0; y < 32; y++ )
  for ( int x = 0; x < 64; x++ )
    doSomething( x, y );
```

CUDA parallelism II

- CPU: run **one** thread
- GPU: start **32×64** threads

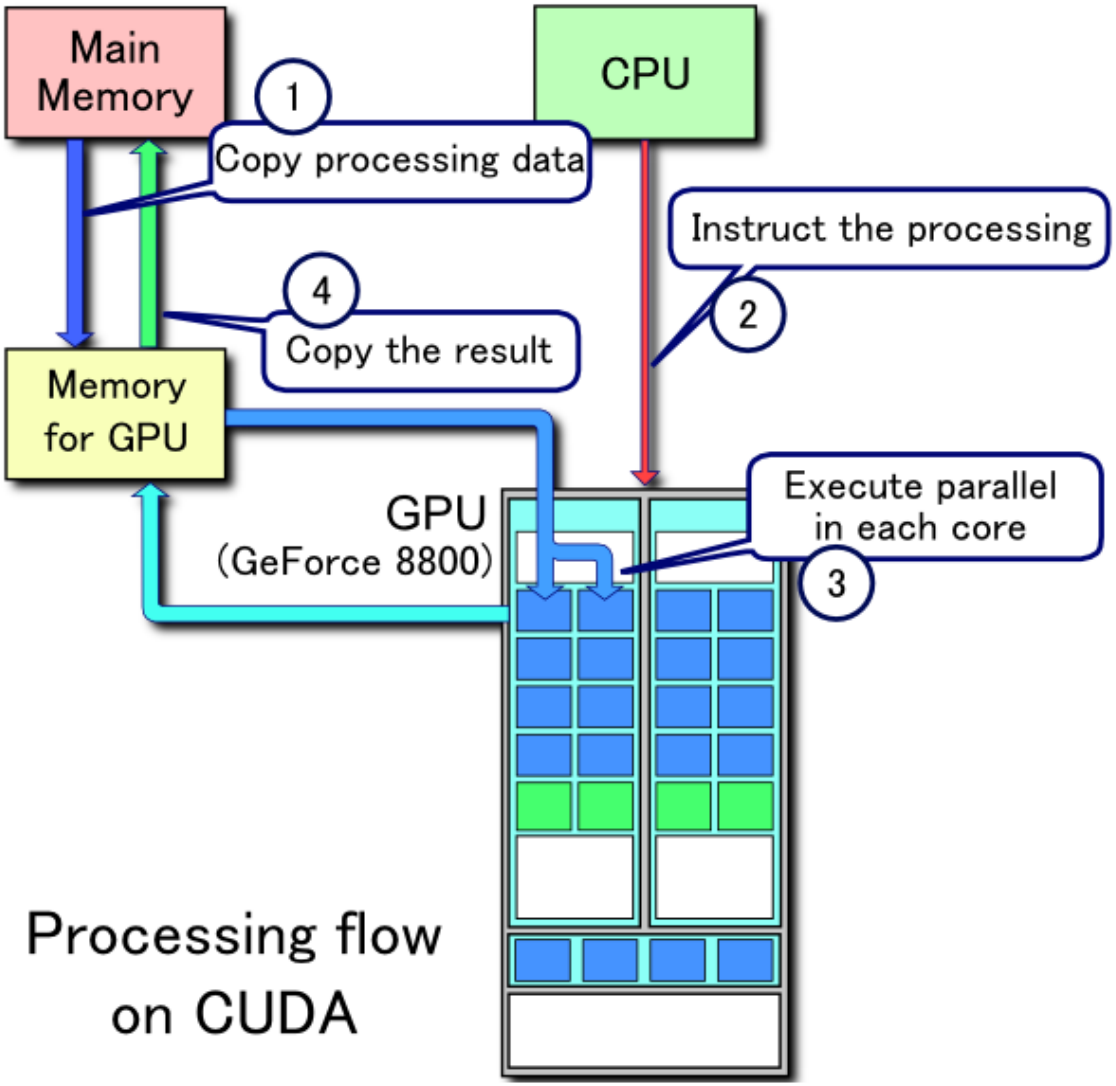
```
// CPU grid computing
for ( int y = 0; y < 32; y++ )
  for ( int x = 0; x < 64; x++ )
    doSomething( x, y );
```

```
// GPU grid computing
int x = threadIdx.x;
int y = threadIdx.y;
doSomething( x, y );
```

kernel start

```
void main ()
{
 ...
 dim3 dimBlock( 32, 64 );
 something<<<1, dimBlock>>>();
 ...
}
```

Program execution



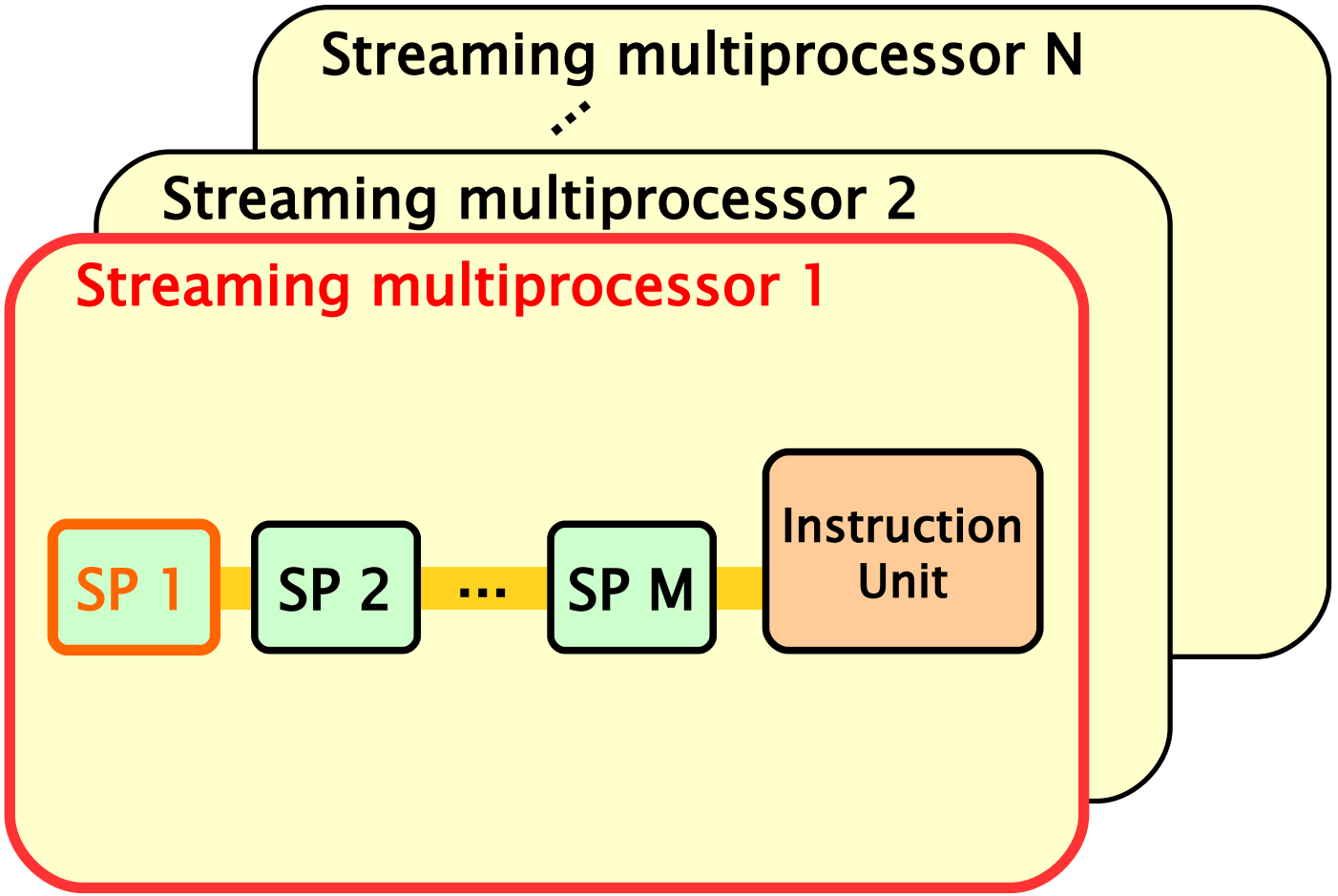
© Wikipedia

CUDA project example I

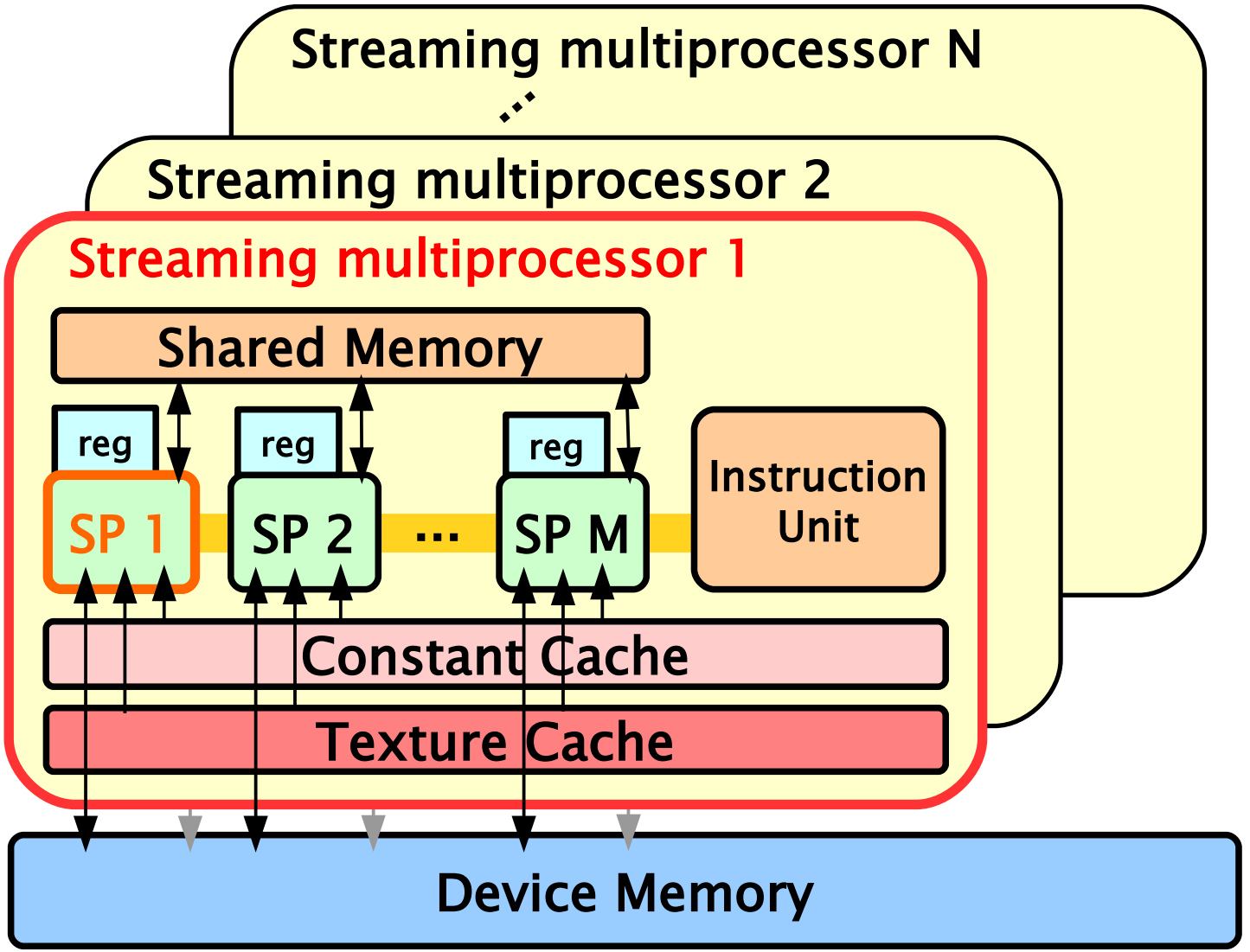
```
// kernel definition
__global__ void VecAdd ( float *a, float *b, float *c )
{
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}

void main ()
{
 float *A, *B, *C;
 ...
 // kernel execution:
 VecAdd<<<1, N>>>( A, B, C );
 ...
}
```

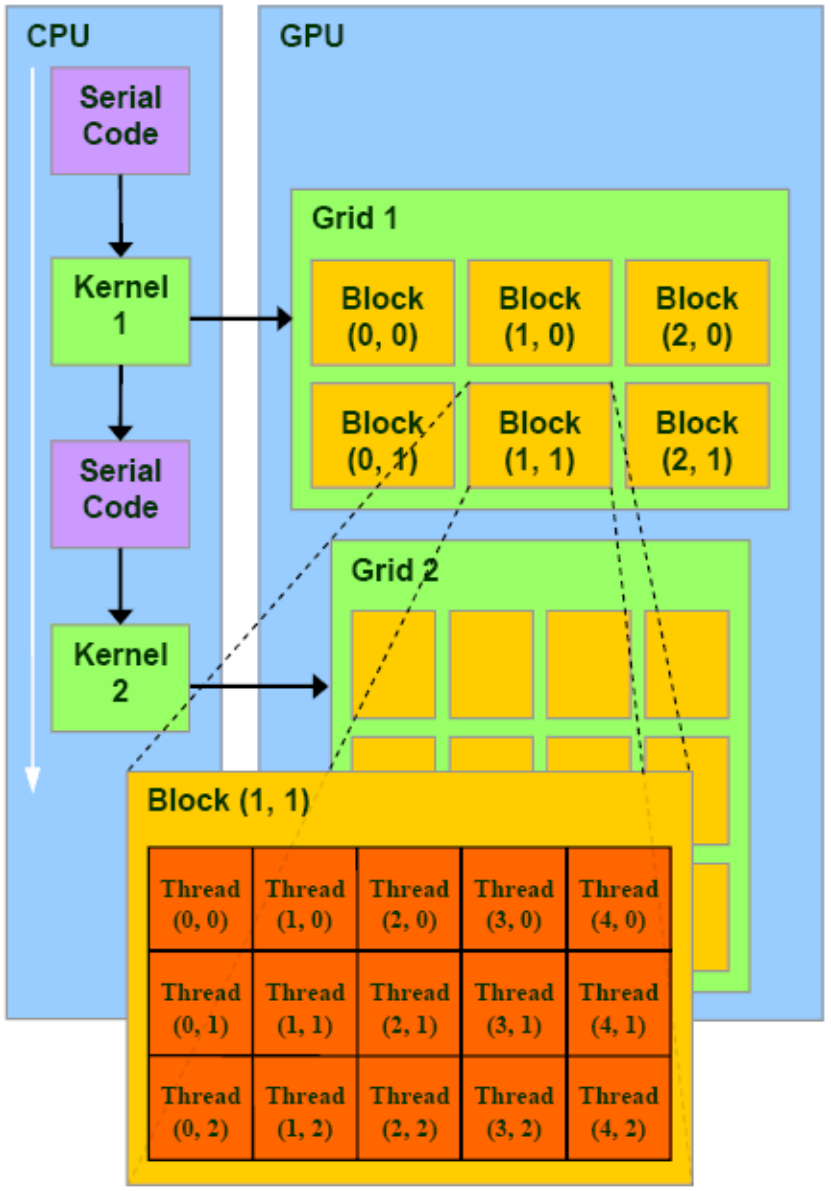
SM scheme revisited



Device memory architecture



Thread hierarchy



Kernel = grid of blocks (1–2D)

random order of block execution

Block = matrix of threads (1–3D)

shared memory

the same instructions

Warp = group of threads running in SM simultaneously

warp size is HW-dependent

Thread identifiers

Builtin variables (C[++] language extension):

- `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
  - thread indices within a block
- `blockIdx.x`, `blockIdx.y`
  - block indices within a grid (kernel)
- `blockDim.x`, `blockDim.y`, `blockDim.z`
  - current block size
- `gridDim.x`, `gridDim.y`
  - current grid size
- `warpSize`

Synchronization

- inter-thread synchronization using **barriers**
  - `__syncthreads()` function guarantees that all threads in the block execute to this point
- **if-then-else** → every thread in a block must execute the same branch!
- inter-block communication is **not possible!**
  - execution order of the blocks is arbitrary
  - optimized scheduling in the GPU
  - if you need block-based synchronization, split the job into more kernels

Thread scheduling

Need-to-know for **optimal efficiency**:

- each **SM** can have assigned up to **8 (16) blocks**
  - depends on resources
- each **block** is divided into **warps** (32 threads)
  - units actually running simultaneously on a SM
  - warp = 32 threads with **consecutive `threadIdx`-s**
- **limits**:
  - max 24 (48, **64**) warps on one SM
  - max 768 (1024, 1536, **2048**) threads on one SM

Device memory types

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached only on devices of compute capability 2.x.					

"memory coalescing" (access … hundreds of cycles)

Variable qualifiers

Declaration	Memory	Scope	Lifetime
automatic except of arrays	register	thread	kernel
automatic array	local*	thread	kernel
`__shared__`	shared	block	kernel
`__device__`	global*	grid	applic.
`__constant__`	constant	grid	applic.

- max shared memory: 16KB (**48KB**) per SM
- max constant memory: **64KB**
- max number of 32-bit registers per SM: 8K (16-**64K**)

CUDA programming

- typical CUDA **program lifetime**
  1. each thread reads part of the data from global to shared memory
  2. `__syncthreads()`
  3. **main computation**
  4. save result data
  5. read a new data and go to **2.** ?

- **if-then-else**
  - no problems if every thread in a warp executes the same branch

CUDA–OpenGL interop

- **CUDA kernel** can write data directly into pre-allocated **VBO buffer** for rendering

```
// Initialize OpenGL
cudaGLSetGLDevice( 0 );

// Create buffer object and register it with CUDA
glGenBuffers( 1, positionsVBO );
glBindBuffer( GL_ARRAY_BUFFER, &vbo );

unsigned int size = width * height * 4 * sizeof(float);
glBufferData( GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW );
glBindBuffer( GL_ARRAY_BUFFER, 0 );

cudaGraphicsGLRegisterBuffer( &positionsVBO_CUDA, positionsVBO,
    cudaGraphicsMapFlagsWriteDiscard );
```

CUDA–OpenGL interop II

```
// Map buffer object for writing from CUDA
float4* positions;
cudaGraphicsMapResources( 1, &positionsVBO_CUDA, 0 );
size_t num_bytes;
cudaGraphicsResourceGetMappedPointer( (void**)&positions,
    &num_bytes, positionsVBO_CUDA );

// Execute kernel
dim3 dimBlock( 16, 16, 1 );
dim3 dimGrid( width / dimBlock.x, height / dimBlock.y, 1 );
createVertices<<<dimGrid, dimBlock>>>( positions, time, width,
    height );

// Unmap buffer object
cudaGraphicsUnmapResources( 1, &positionsVBO_CUDA, 0 );

// Do the OpenGL rendering from the positionsVBO
...
```

CUDA project example II

```
// kernel version II
__global__ void VecAdd ( float *a, float *b, float *c, int N )
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if ( i < N )
   c[i] = a[i] + b[i];
}

void main ()
{
 const int N = ..;
 size_t size = N * sizeof(float);
 float *A, *B, *C;

 // host memory allocation:
 float* hA = (float*)malloc( size );
 float* hB = (float*)malloc( size );

 // initialize input vectors:

 ...
```

CUDA project example III

```
// allocate vectors in device memory:
float *dA, *dB, *dC;
cudaMalloc( &dA, size );
cudaMalloc( &dB, size );
cudaMalloc( &dC, size );

// copy vectors from host memory to device memory:
cudaMemcpy( dA, hA, size, cudaMemcpyHostToDevice );
cudaMemcpy( dB, hB, size, cudaMemcpyHostToDevice );

// invoke kernel:
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blockPerGrid, threadsPerBlock>>>( dA, dB, dC, N );

// copy result from device memory to host memory:
cudaMemcpy( hC, dC, size, cudaMemcpyDeviceToHost );

// free device memory:
cudaFree( dA ); cudaFree( dB ); cudaFree( dC );

...
}
```

OpenCL vs. CUDA I

- **OpenCL** does not need NVIDIA GPU
  - can even run on Intel/AMD multi-core CPUs

- **differences** (OpenCL – CUDA)
  - kernel – kernel
  - host program – host program
  - NDRange (index space) – grid
  - work item – thread
  - work group – block
  - …

OpenCL vs. CUDA II

- `get_global_id(0)` – `blockIdx.x * blockDim.x + threadIdx.x`
- `get_local_id(0)` – `threadIdx.x`
- `get_global_size(0)` – `gridDim.x * blockDim.x`
- `get_local_size(0)` – `blockDim.x`

- global memory – global memory
- constant memory – constant memory
- **local memory** – shared memory
- private memory – **local memory**

OpenCL kernel example

```
// kernel definition
__kernel void vecAdd ( __global const float *a,
                       __global const float *b,
                       __global float *c )
{
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

Sources

- David B. Kirk, Wen-mei W. Hwu: ***Programming massive parallel processors***, Morgan Kaufman, 2010, ISBN: 978-0-12-381472-2
- Jason Sanders, Edward Kandrot: ***CUDA by Example: An Introduction to General-Purpose GPU Programming***, Addison-Wesley, 2010, ISBN: 978-0131387683
- NVIDIA Corporation: ***(CUDA C/OpenCL) Programming Guide***
- NVIDIA Corporation: ***(CUDA C/OpenCL) Best Practices***