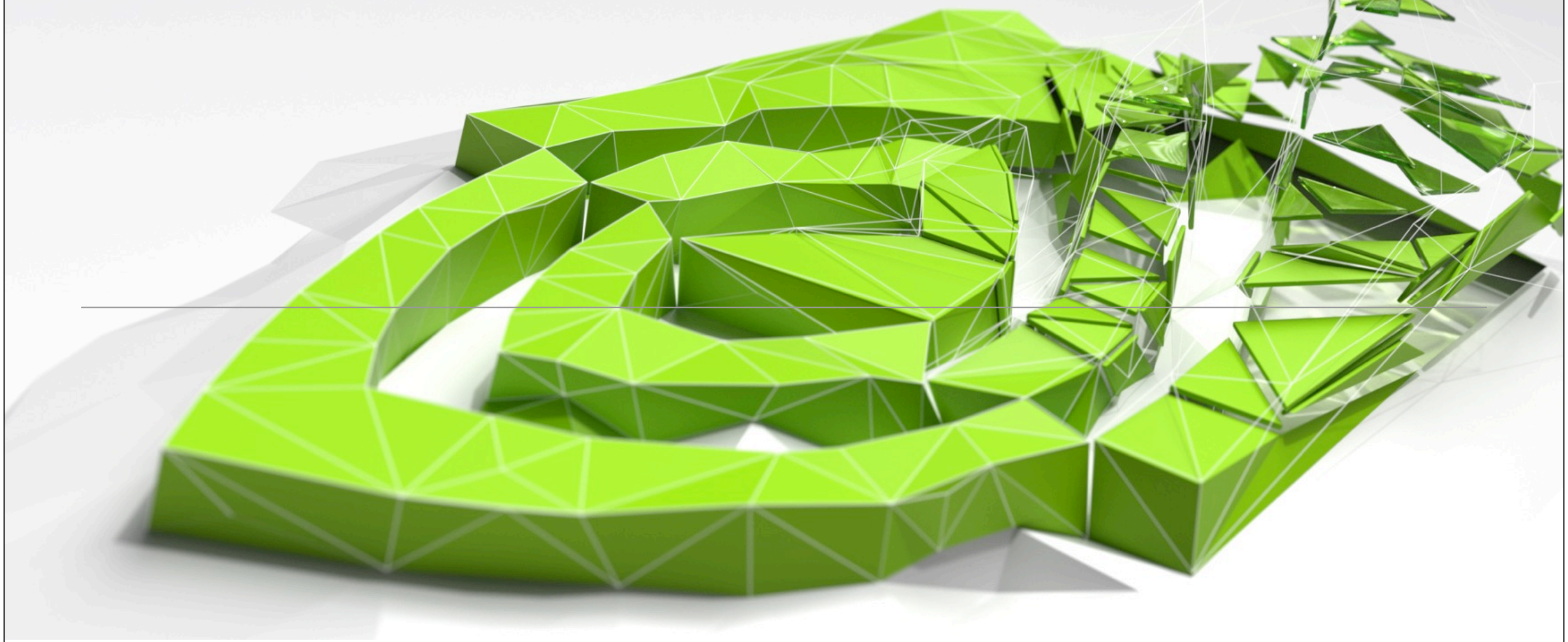
Introduction to CUDA Programming

PRACE School

Barcelona Supercomputing Center. April, 16-20th, 2018



Manuel Ujaldón

Full Professor @ University of Malaga (Spain)

Former CUDA Fellow @ NVIDIA Corporation (USA)

Agenda

Day	Time frame	Contents	Level
16/4	09:00 - 10:45	The GPU hardware: Many-core Nvidia developments	Basic
16/4	10:45 - 11:15	Break	
16/4	11:15 - 13:00	CUDA Programming: Threads, blocks, kernels, grids	Basic
16/4	13:00 - 14:00	Lunch break	
16/4	14:00 - 15:45	CUDA tools: Compiling, profiling, occupancy calculator	Basic
16/4	15:45 - 16:15	Break	
16/4	16:15 - 18:00	CUDA examples: VectorAdd, Stencils	Basic
17/4	09:00 - 10:45	CUDA examples: Matrices Multiply. Assorted optimizations	Intermediate
17/4	10:45 - 11:15	Break	
17/4	11:15 - 13:00	Inside Kepler & Maxwell: Hyper-Q, dyn. par., unified memory	Advanced
18/4	09:00 - 10:45	Inside Pascal and Volta: Stacked memory, NV-link, tensor cores	Intermediate
18/4	10:45 - 11:15	Break	
18/4	11:15 - 13:00	OpenACC and other approaches to GPGPU. Bibliography	Basic

Tutorial contents [148 slides]

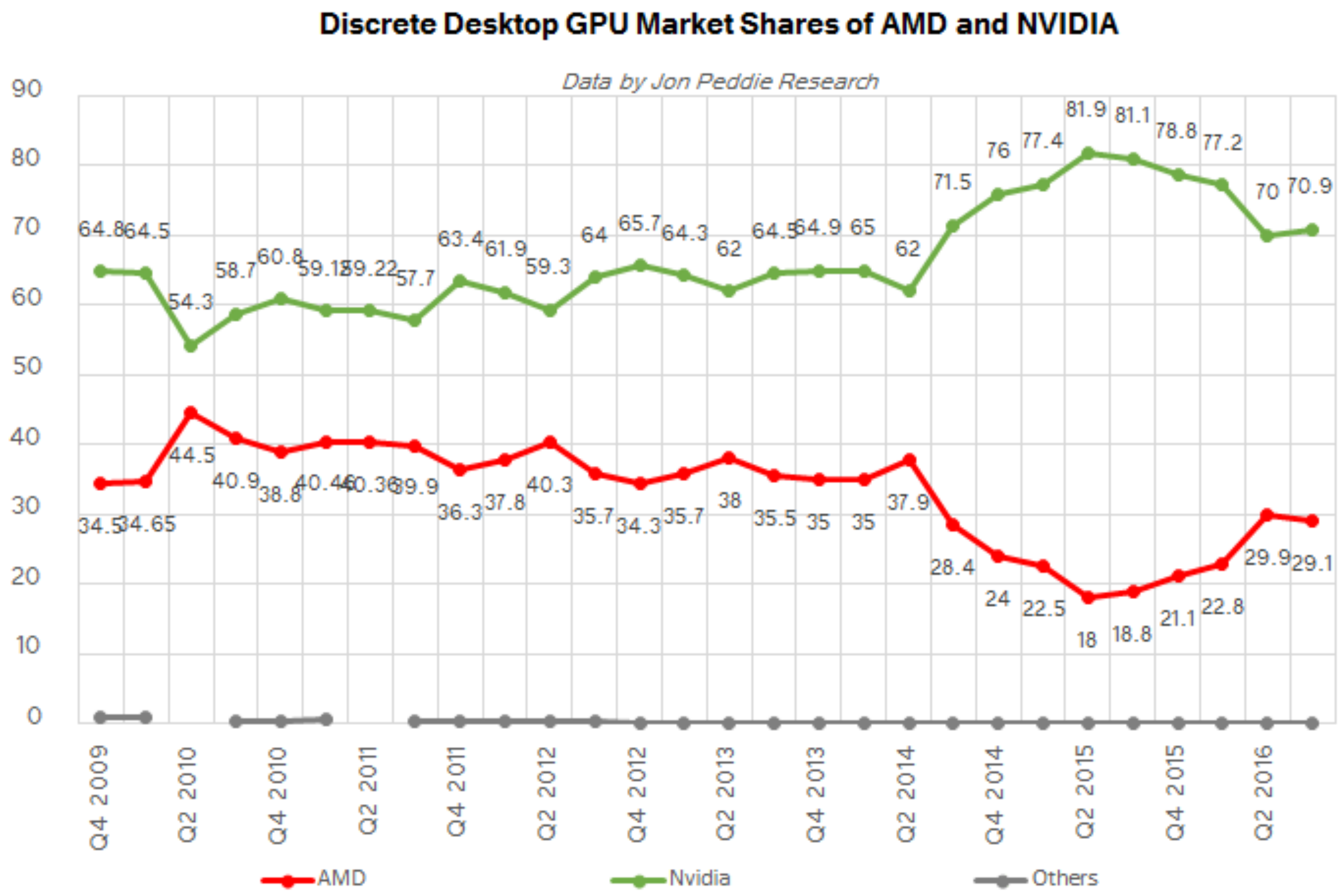
1. Introduction. [23 slides]
2. Architecture. [40]
    1. CUDA hardware model. [3]
    2. First generation: Tesla (2007-2009). [3]
    3. Second generation: Fermi (2010-2011). [4]
    4. Third generation: Kepler (2012-2015). [4]
    5. Fourth generation: Maxwell (2015-2016). [6]
    6. Fifth generation: Pascal (2016-17). [10]
    7. Sixth generation: Volta (2018-?). [8]
    8. Summary. [2]
3. Programming. [19]
4. Syntax. [15]
    1. Basic elements. [9]
    2. A couple of preliminary examples. [6]
5. Compilation and tools [12]
6. Examples: Base [2], VectorAdd [5], Stencil [8], Reverse [4], MxM [11]
7. Bibliography, resources and tools. [9]

Prerequisites for this tutorial

- You (probably) need experience with C.
- You do not need parallel programming background (but it helps if you have it).
- You do not need knowledge about the GPU architecture: We will start with the basic pillars.
- You do not need graphics experience. Those were the old times (shaders, Cg). With CUDA, it is not required any knowledge about vertices, pixels, textures, ...

I. Introduction

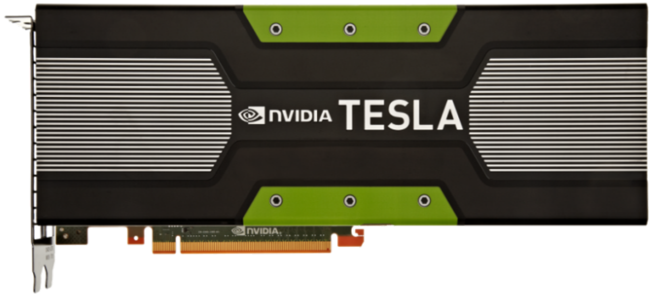
Market share: 2010-2016



Welcome to the GPU world

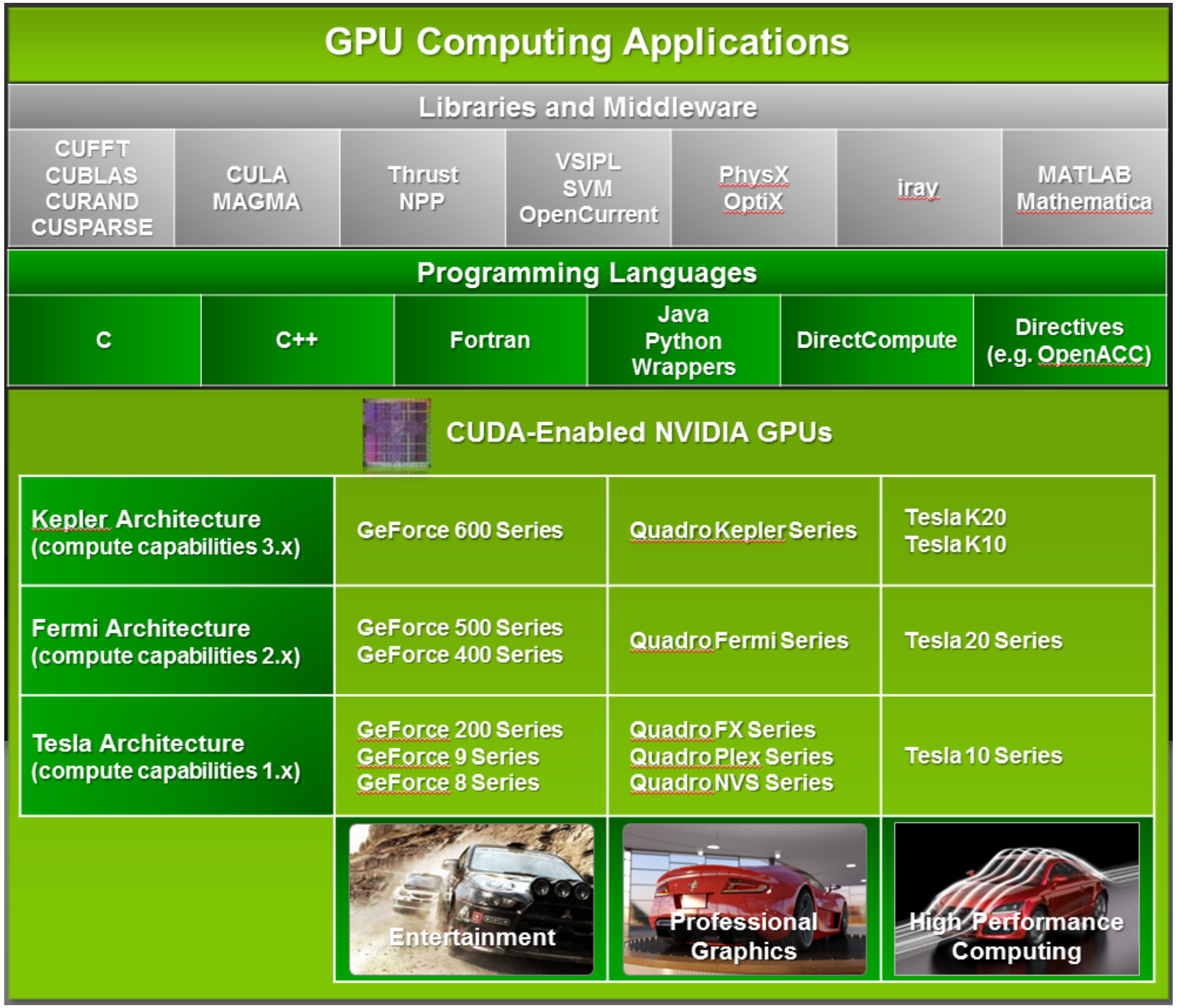
Commercial models: GeForce vs. Tesla

- Designed for gamers:
  - Price is a priority (<500€).
  - Availability and popularity.
  - Small video memory (1-2 GB.).
  - Frequency slightly ahead.
  - Perfect for developing code which can later run on a Tesla.



- Oriented to HPC:
  - Reliable (3 years warranty).
  - For cluster deployment.
  - More video memory (6-12 GB.).
  - Tested for endless run (24/7).
  - GPUDirect (RDMA) and other features for GPU clusters.

The characters of this story: The CUDA family picture



GPU Computing Applications

Libraries and Middleware

CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

Kepler Architecture (compute capabilities 3.x)	GeForce 600 Series	Quadro Kepler Series	Tesla K20 Tesla K10
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series	Quadro Fermi Series	Tesla 20 Series
Tesla Architecture (compute capabilities 1.x)	GeForce 200 Series GeForce 9 Series GeForce 8 Series	Quadro FX Series Quadro Plex Series Quadro NVS Series	Tesla 10 Series
	Entertainment	Professional Graphics	High Performance Computing

The impressive evolution of CUDA

Year 2008

100.000.000
CUDA-capable GPUs
(6.000 Teslas only)

150.000
CUDA downloads

1
supercomputer
in top500.org
(77 TFLOPS)

60
university courses

4.000
academic papers

Year 2016

600.000.000 CUDA-capable GPUs
(and 450.000 Tesla high-end GPUs)

3.000.000 CUDA downloads per year
(that is, one every 9 seconds)

63 supercomputers
in TOP500.org
Agregate: 80.000 TFLOPS
(more than 14% of the
567 PFLOPs in top500)

840 university courses

60.000
academic papers

Summary of GPU evolution

- 2001: First many-cores (vertex and pixel processors).
- 2003: Those processor become programmable (using Cg).
- 2006: Vertex and pixel processors unify.
- 2007: CUDA emerges.
- 2008: Double precision floating-point arithmetic.
- 2010: Operands are IEEE-normalized and memory is ECC.
- 2012: Wider support for irregular computing.
- 2014: The CPU-GPU memory space is unified.
- 2016: 3D memory. NV-link.
- Still pending: Reliability in clusters and connection to disk.

The 3 features which have made the GPU such a unique processor

- Simplified.
  - The control required for one thread is amortized by 31 more (**warp**).
- Scalability.
  - Makes use of the huge **data volume** handled by applications to define a sustainable parallelization model.
- Productivity.
  - Endowed with efficient mechanisms for **switching immediately** to another thread whenever the one being executed suffers from **stalls**.
- CUDA essential keywords:
  - Warp, SIMD, latency hiding, free context switch.

Three reason for feeling attracted to GPUs

- Cost
  - Low price due to a massive selling marketplace.
  - Three GPUs are sold for each CPU, and the ratio keeps growing.
- Ubiquitous
  - Everybody already has a bunch of GPUs.
  - And you can purchase one almost everywhere.
- Power
  - Ten years ago GPUs exceed 200 watts. Now, they populate the Green 500 list. Progression in floating-point computation:

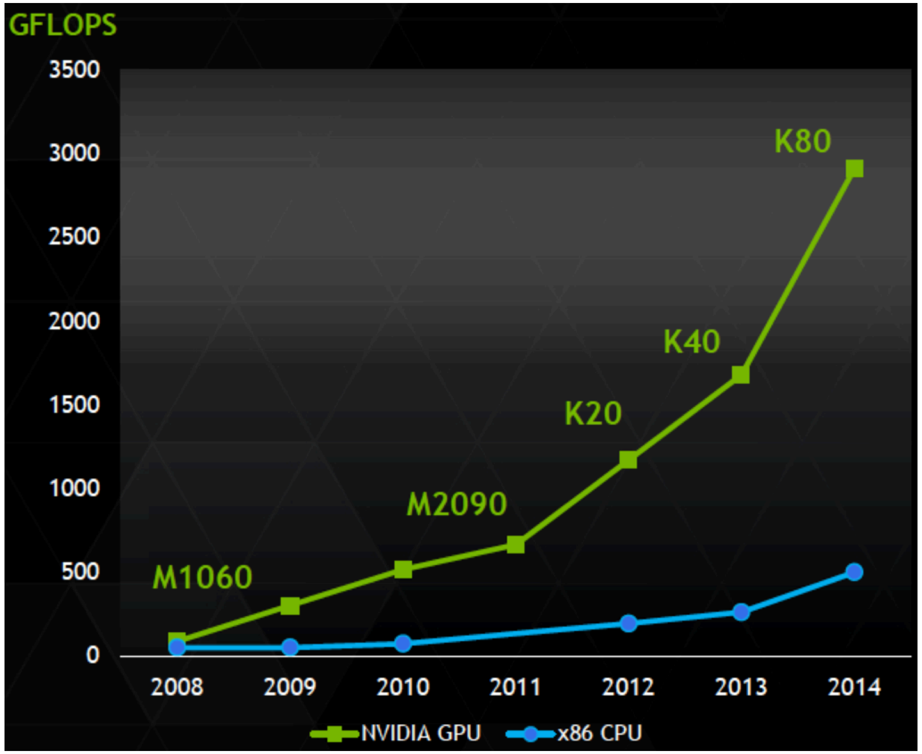
	GFLOPS/w on float (32-bit)	GFLOPS/w. on double (64-bit)
Fermi (2010)	5-6	3
Kepler (2012)	15-17	7
Maxwell (2014)	40	12

Highlights

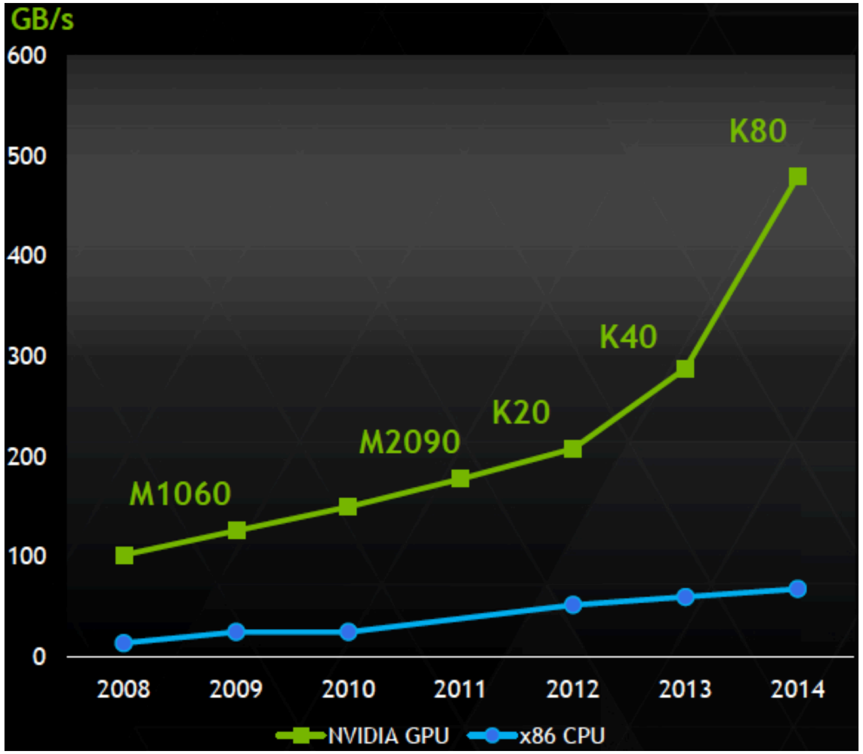
- In processing power:
  - Frequency gives up the leadership: Heat and voltage set the barrier.
  - Instruction level parallelism (ILP), task parallelism (multi-thread) and symmetric multiprocessing (SMP) saturate.
  - Solution: Exploit data parallelism on GPU, which is more scalable.
- In static memory (SRAM):
  - Alternative: Leave small caches visible to programmer.
- In dynamic memory (DRAM):
  - Increment the bandwidth (OK), but also the latency (upps).
  - Solution: **Stacked-DRAM**, or the way to solve the ***memory wall*** by contributing simultaneously with quantity (GB.) and quality (speed).

GPU peak performance vs. CPU

Peak GFLOPS (fp64)



Memory Bandwidth



- GPU 6x faster on "`double`":
  - GPU: 3000 GFLOPS
  - CPU: 500 GFLOPS

- GPU 6x more bandwidth:
  - 7 GHz x 48 bytes = 336 GB/s.
  - 2 GHz x 32 bytes = 64 GB/s.

What is CUDA?
"Compute Unified Device Architecture"

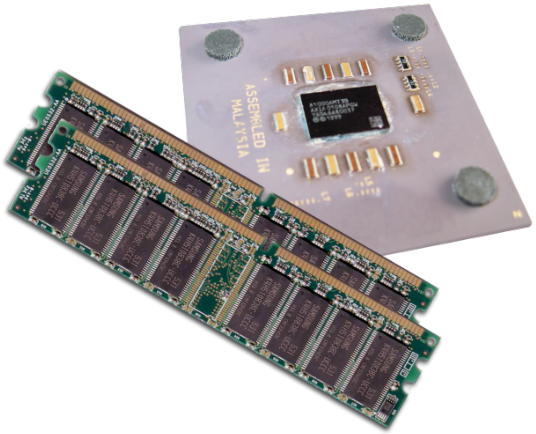
- A platform designed jointly at software and hardware levels to make use of the GPU computational power in general-purpose applications at three levels:
  - Software: It allows to program the GPU with minimal but powerful SIMD extensions to enable heterogeneous programming and attain an efficient and scalable execution.
  - Firmware: It offers a driver oriented to GPGPU programming, which is compatible with the one used for rendering. Straightforward APIs manage devices, memory, ...
  - Hardware: It exposes GPU parallelism for general-purpose computing via a number of twin multiprocessors endowed with cores and a memory hierarchy.

CUDA C at a glance

- Essentially, it is C language with minimal extensions:
  - Programmer writes the program for a single thread, and the code is automatically instanciated over hundreds of threads.
- CUDA defines:
  - An architectural model:
    - With many processing cores grouped in multiprocessors who share a SIMD control unit.
  - A programming model:
    - Based on massive data parallelism and fine-grain parallelism.
    - Scalable: The code is executed on a different number of cores without recompiling it.
  - A memory management model:
    - More explicit to the programmer, where caches are not transparent anymore.
- Goals:
  - Build a code which scales to hundreds of cores in a simple way, allowing us to declare thousands of threads.
  - Allow heterogeneous computing (between CPUs and GPUs).

Heterogeneous Computing (1/4)

- Terminology:
  - Host: The CPU and the memory on motherboard [DDR3].
  - Device: The graphics card [GPU + video memory]:
    - GPU: Nvidia GeForce/Tesla.
    - Video memory: GDDR5 or 3D memory.

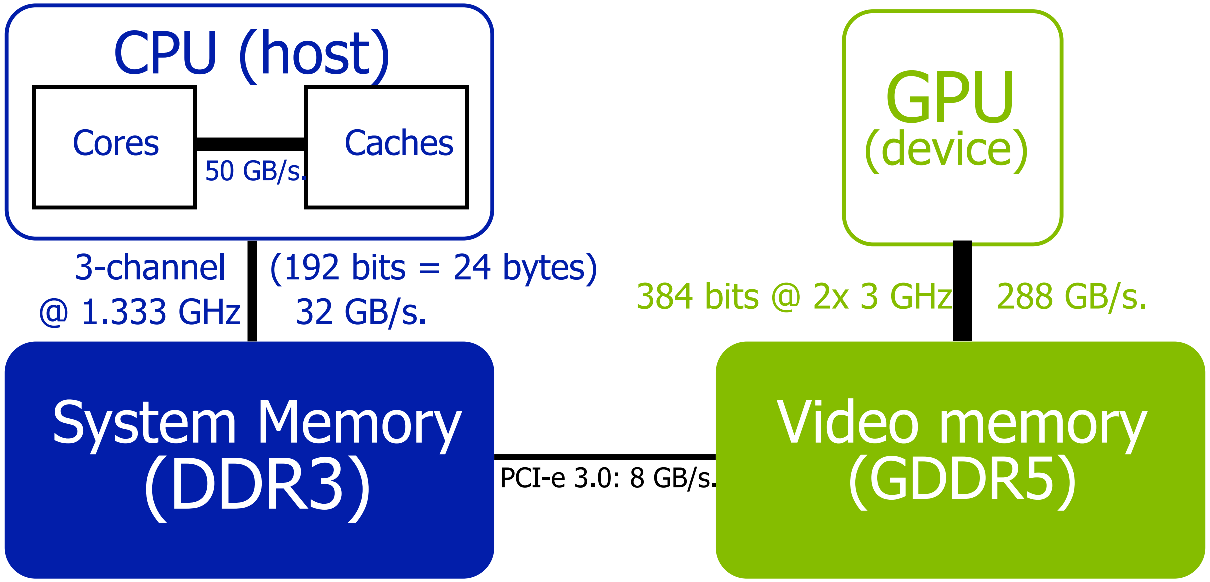


Host

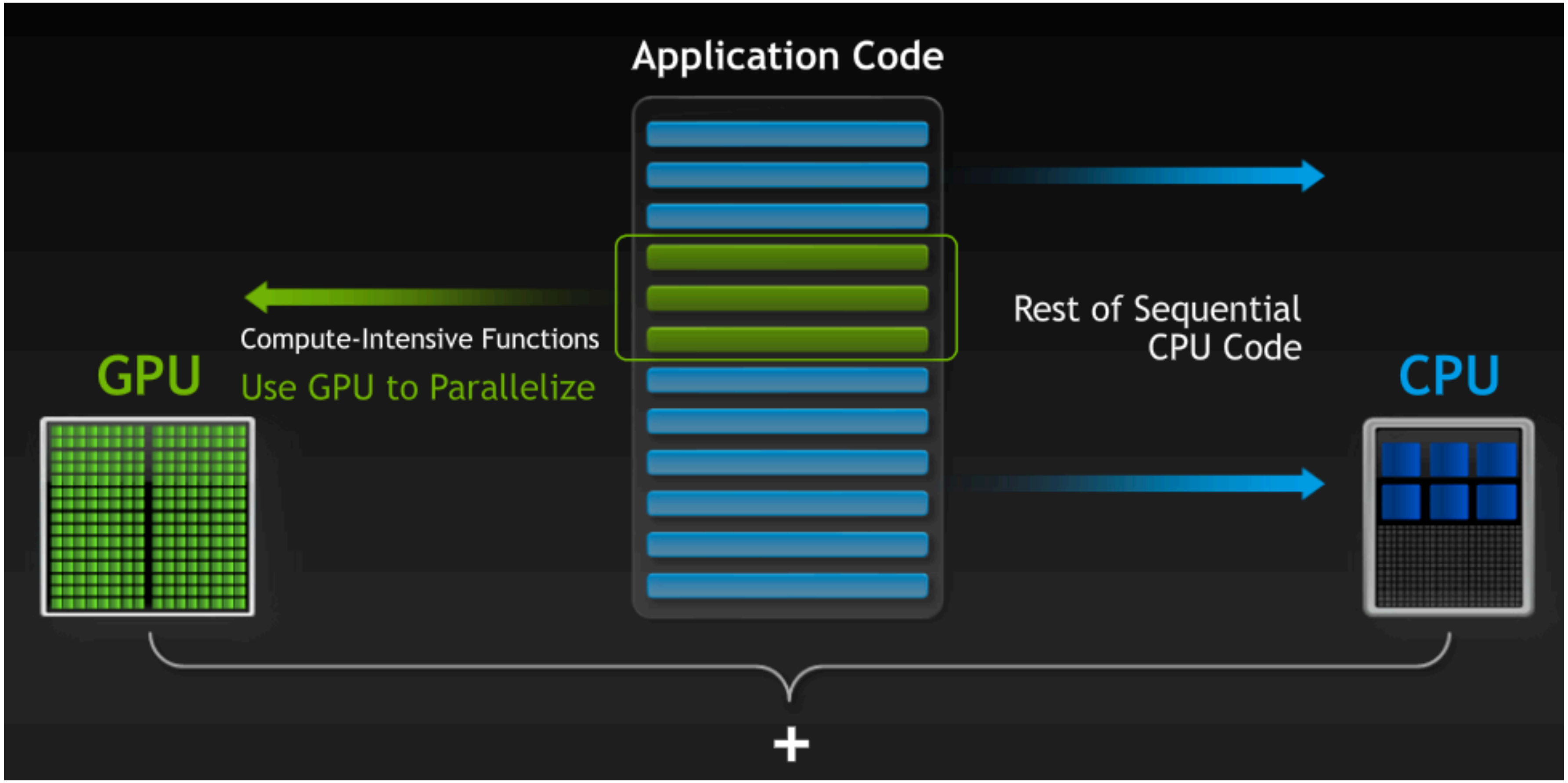
Device

Heterogeneous Computing (2/4)

- CUDA executes a program on a device (the GPU), which is seen as a co-processor for the host (the CPU).
- CUDA can be seen as a library of functions which contains 3 types of components:
  - Host: Control and access to devices.
  - Device: Specific functions for the devices.
  - All: Vector data types and a set of routines supported on both sides.



Heterogeneous Computing (3/4)



- The code to be written in CUDA can be lower than 5%, but exceed 50% of the execution time if remains on CPU.

Heterogeneous Computing (4/4)

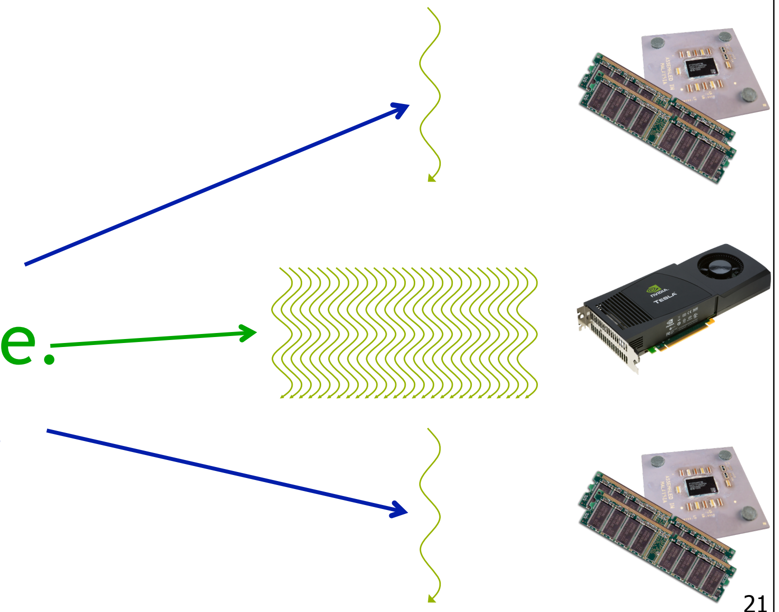
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}
```

DEVICE CODE:
Parallel function
written in CUDA.

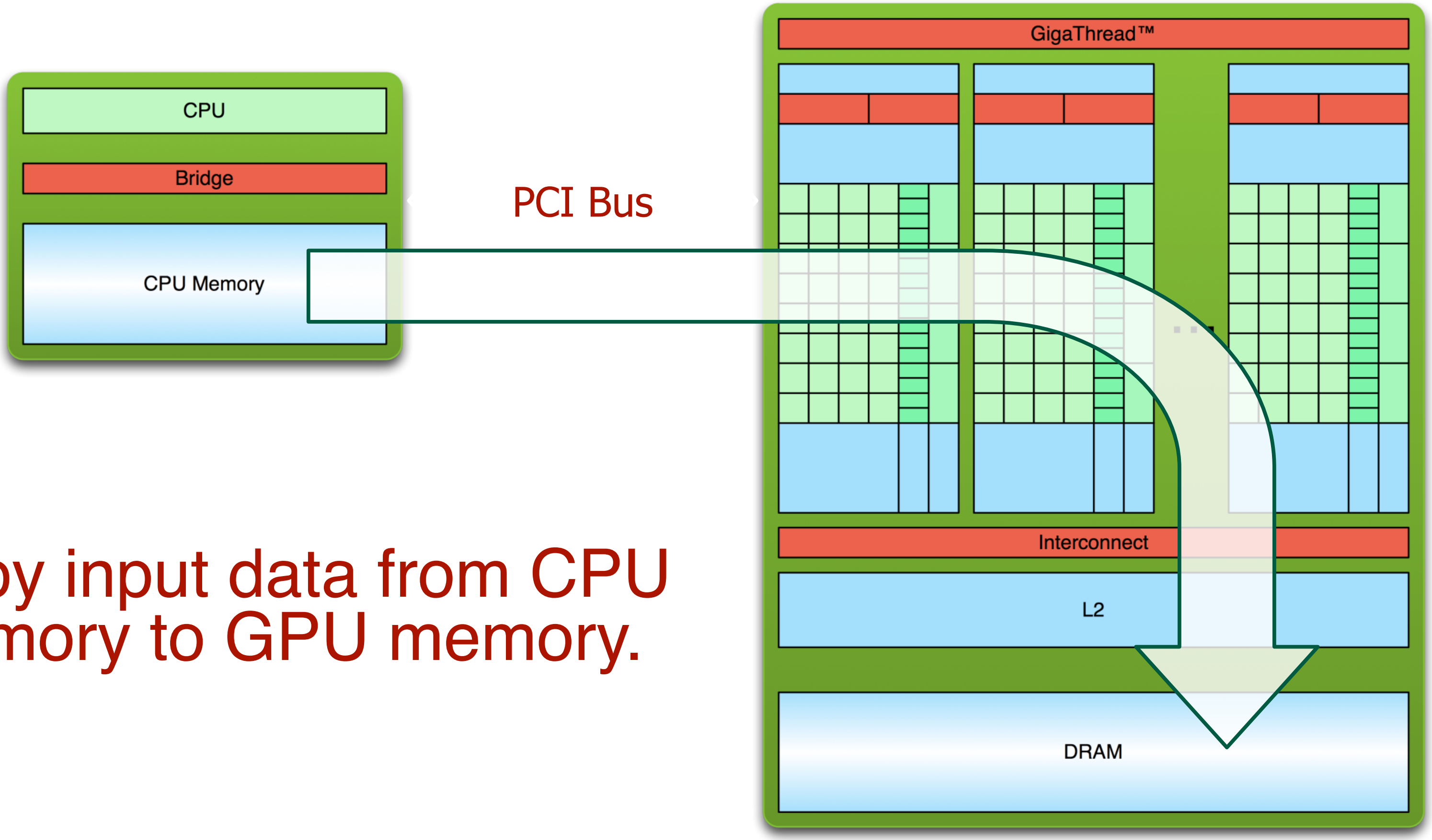
```
int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

HOST CODE:

- Serial code.
- Parallel code.
- Serial code.

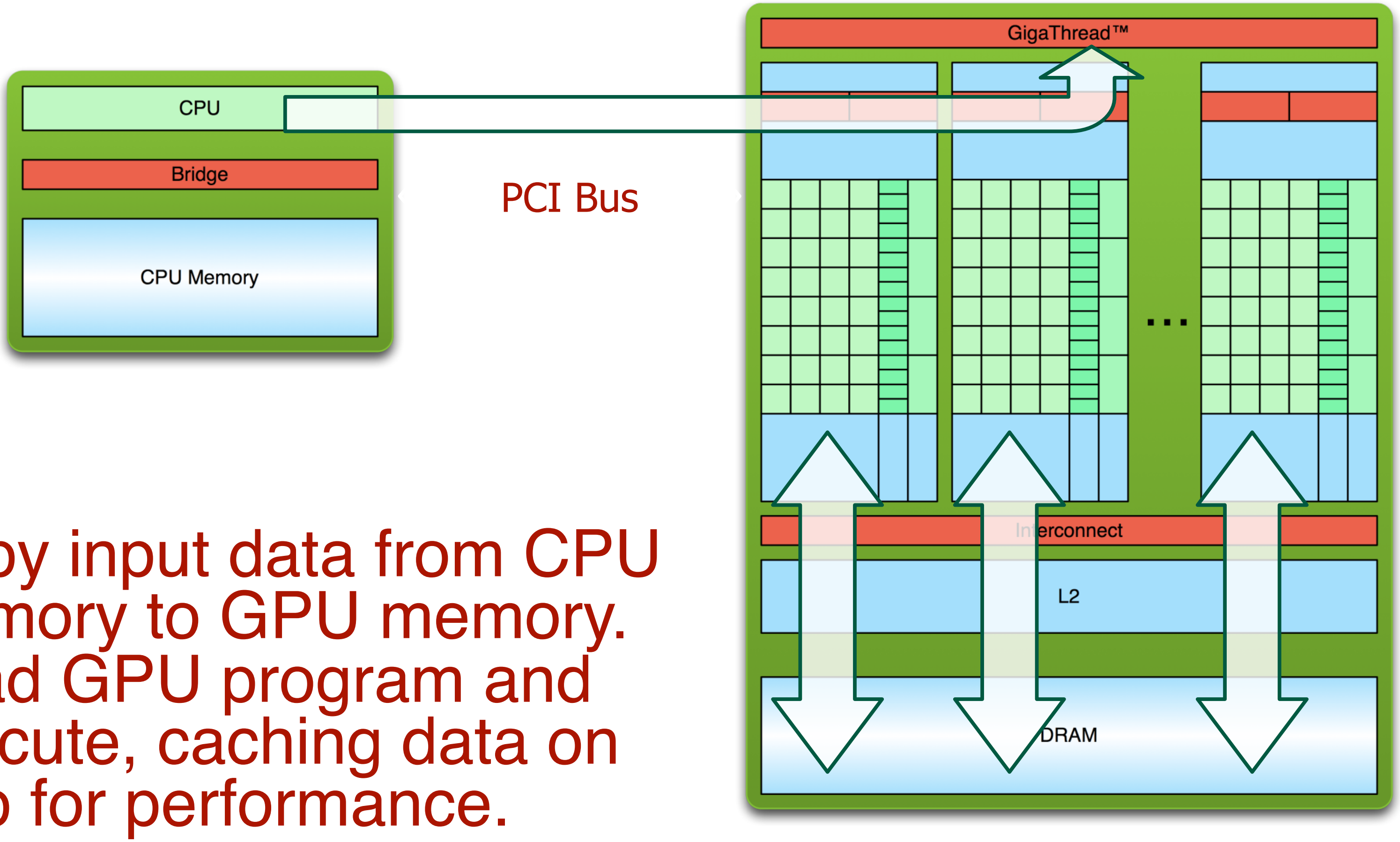


Simple Processing Flow (1/3)



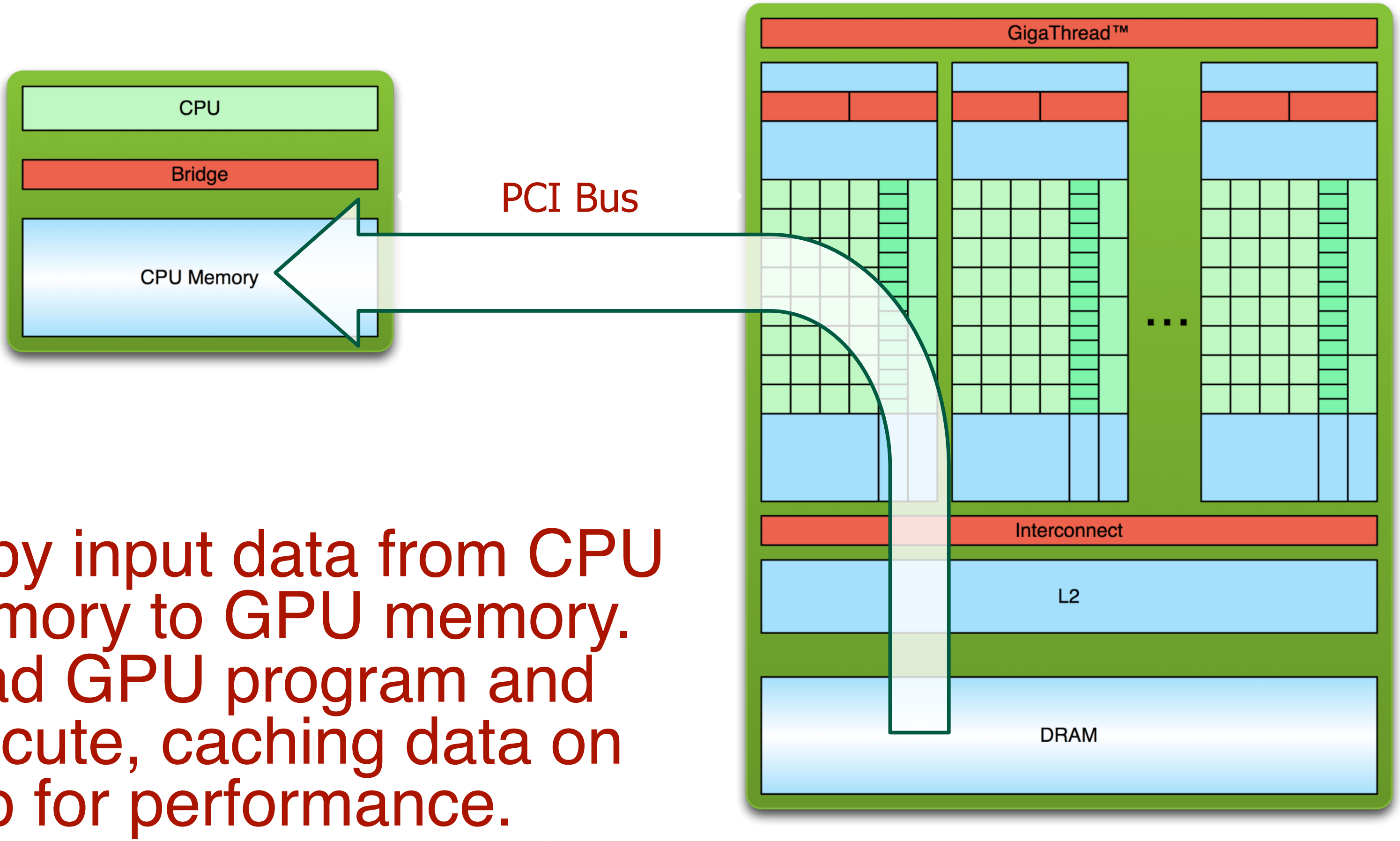
1. Copy input data from CPU memory to GPU memory.

Simple Processing Flow (2/3)



1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.

Simple Processing Flow (3/3)



1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.
3. Transfer results from GPU memory to CPU memory.

The classic example

```c
int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- Standard C that runs on the host.
- NVIDIA compiler (nvcc) can be used to compile programs with no device code.

Hello World! with device code (1/2)

```
__global__ void mykernel(void)
{
  printf("Hello World!\n");
}
int main(void)
{
  mykernel<<<1,1>>>();
  return 0;
}
```

- Two new syntactic elements:
  - The CUDA C keyword __global__ indicates a function that runs on the device and is called from host code.
  - mykernel<<<1,1>>> is a CUDA kernel launch from the host code.
- That's all that is required to execute a function on the GPU!

- `nvcc` separates source code into host and device.
- Device functions (like `mykernel()`) are procesed by NVIDIA compiler.
- Host functions (like `main()`) are processed by host compiler (`gcc` for Unix, `cl.exe` for Windows).

Hello World! with device code (2/2)

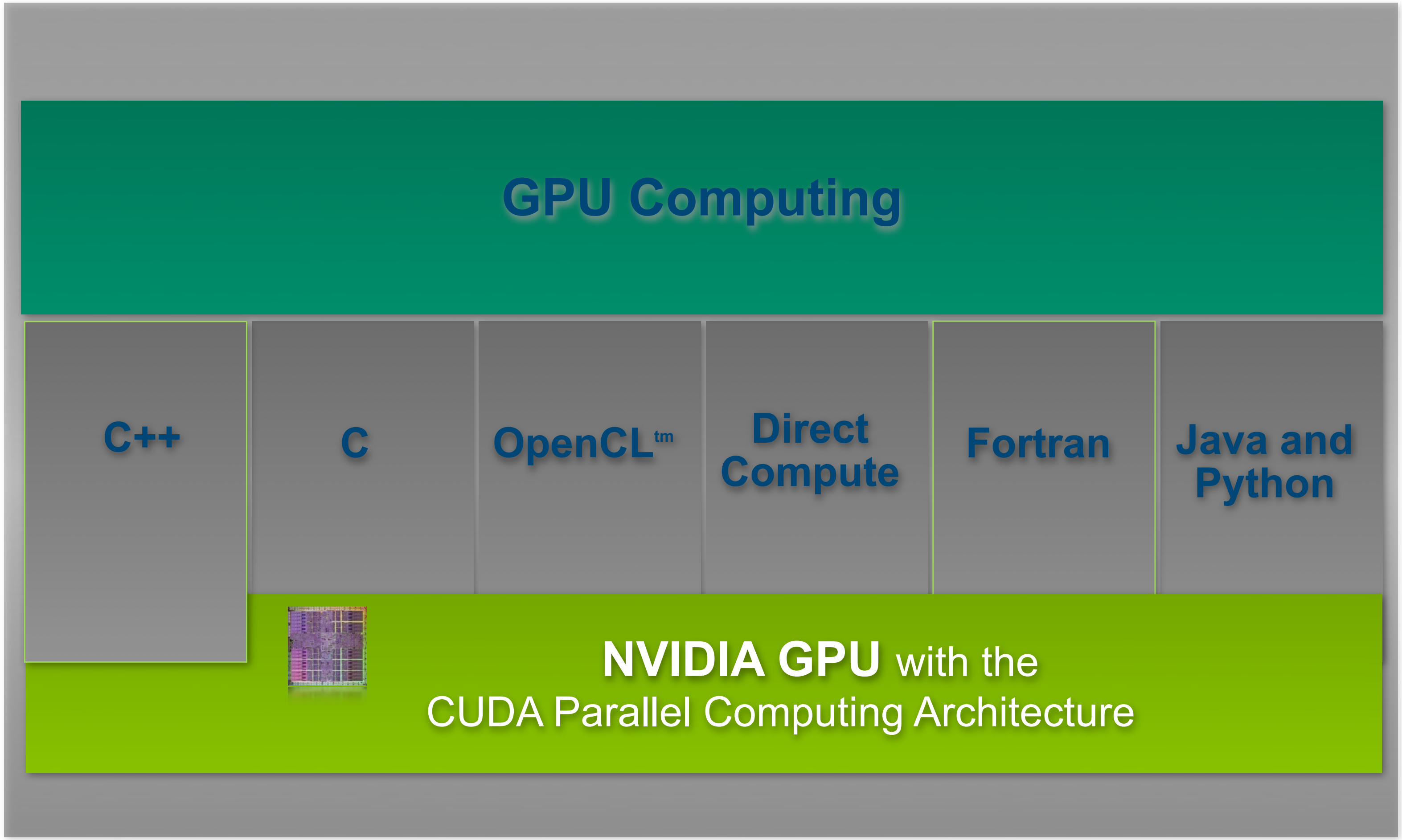
```
__global__ void mykernel(void)
{
}

int main(void) {
  mykernel<<<1,1>>>();
  printf("Hello World!\n");
  return 0;
}
```

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- mykernel() does nothing this time.
- Triple angle brackets mark a call from host code to device code.
  - Also called a “kernel launch”.
  - Parameters <<<1,1>>> describe CUDA parallelism (blocks and threads).

If we have a CUDA architecture, we can approach programming in different ways...



- ... but this tutorial focuses on CUDA C.

II. Architecture

"… and if software people wants good machines, they must learn more about hardware to influence that way hardware designers …"

David A. Patterson & John Hennessy

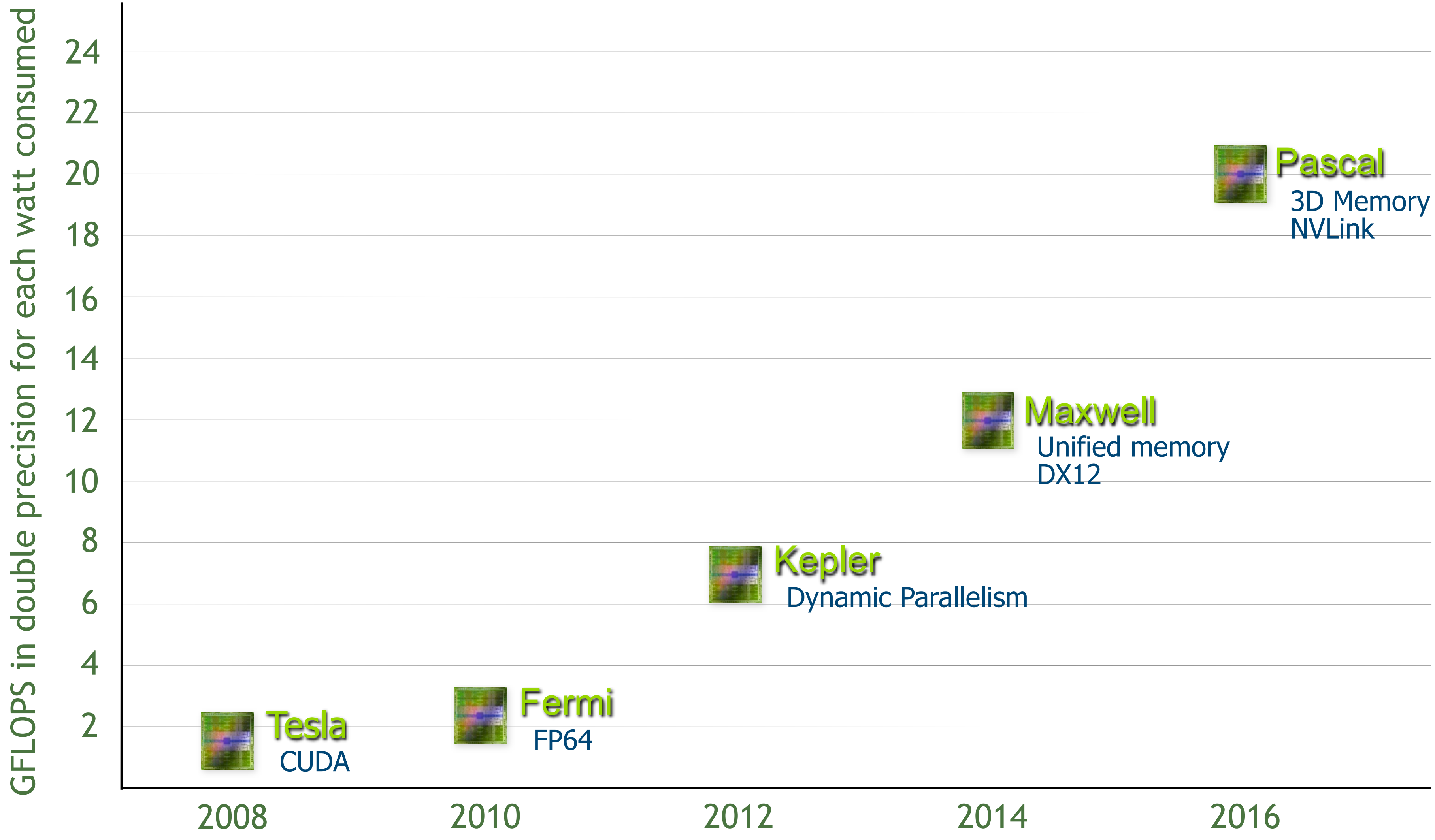
Organization and Computer Design

Mc-Graw-Hill (1995)

Chapter 9, page 569

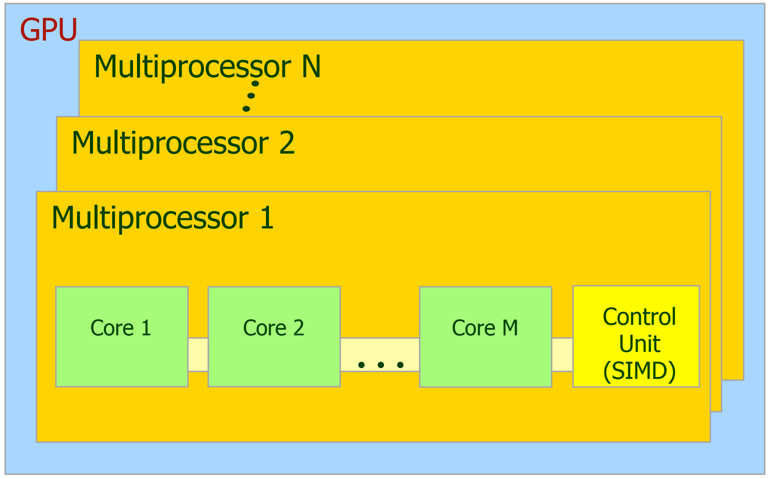
II.1. CUDA hardware model

Overview of CUDA hardware generations



The CUDA hardware model: SIMD processors structured, a tale of hardware scalability

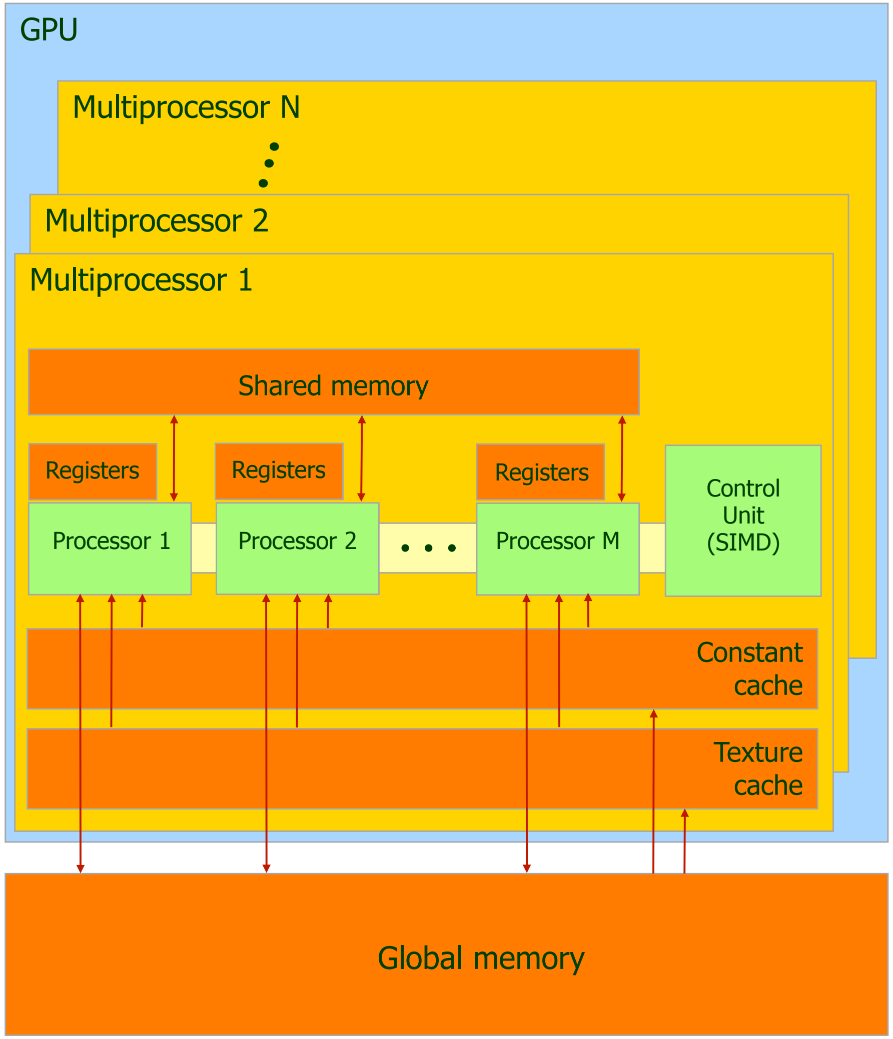
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.
    - Coarse-grain parallelism.



	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)	GK110 (Kepler)	(GM200) Maxwell
Period	2006-07	2008-09	2010-11	2012-13	2014-15
N (multip.)	16	30	14-16	13-15	4-24
M (cores/mult.)	8	8	32	192	128
# cores	128	240	448-512	2496-2880	512-3072

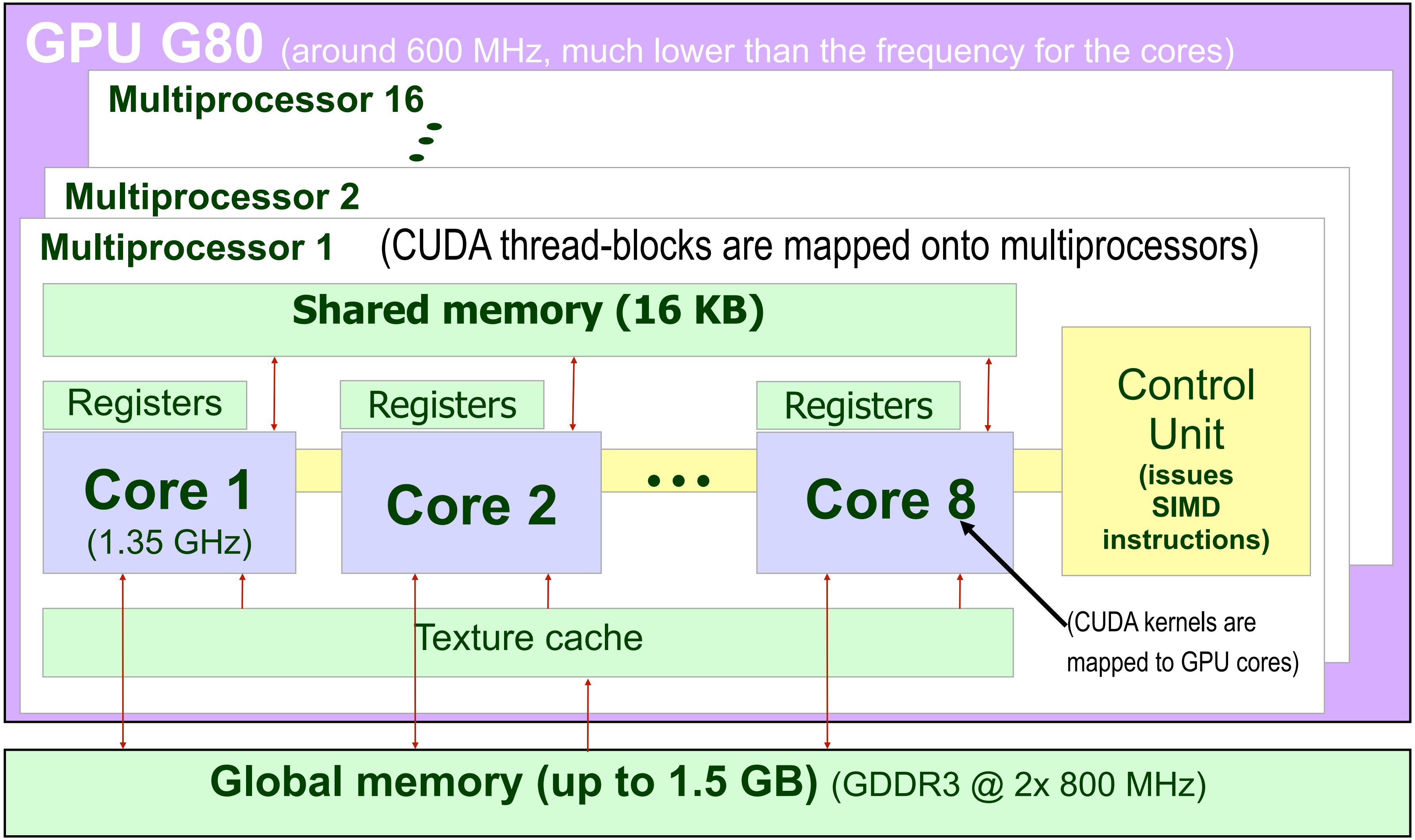
Memory hierarchy

- Each multiprocessor has:
  - A register file.
  - Shared memory.
  - A constant cache and a texture cache, both read-only.
- Global memory is the actual video memory (GDDR5):
  - Three times faster than the DDR3 used by the CPU, but...
  - ... around 500 times slower than shared memory! (DRAM versus SRAM).



II.2. First generation: Tesla (G80 and GT200)

The first generation: G80 (GeForce 8800)



GPU G80 (around 600 MHz, much lower than the frequency for the cores)

Multiprocessor 16

Multiprocessor 2

Multiprocessor 1 (CUDA thread-blocks are mapped onto multiprocessors)

Shared memory (16 KB)

Registers

Registers

Registers

Core 1 (1.35 GHz)

Core 2

• • •

Core 8

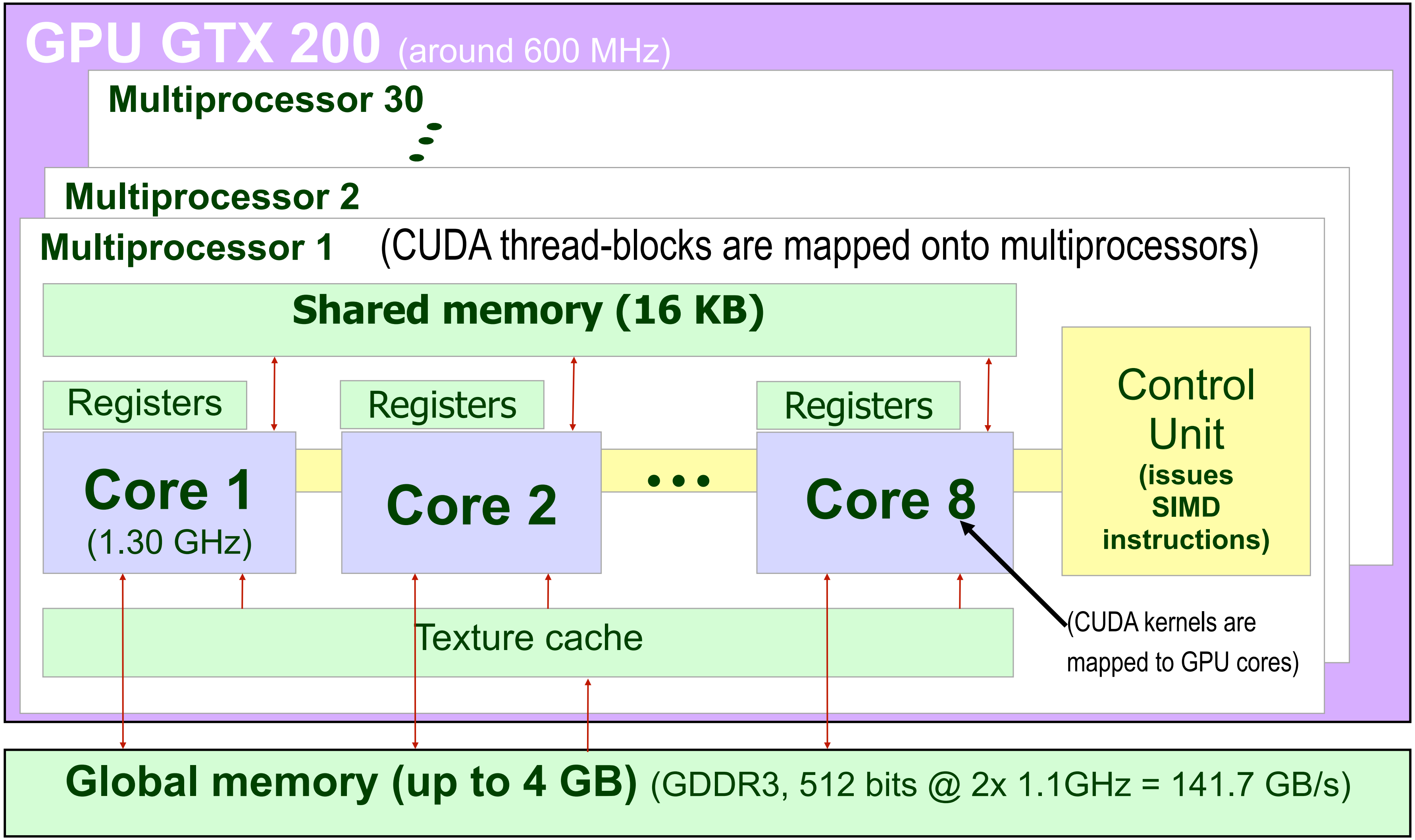
Control Unit (issues SIMD instructions)

(CUDA kernels are mapped to GPU cores)

Texture cache

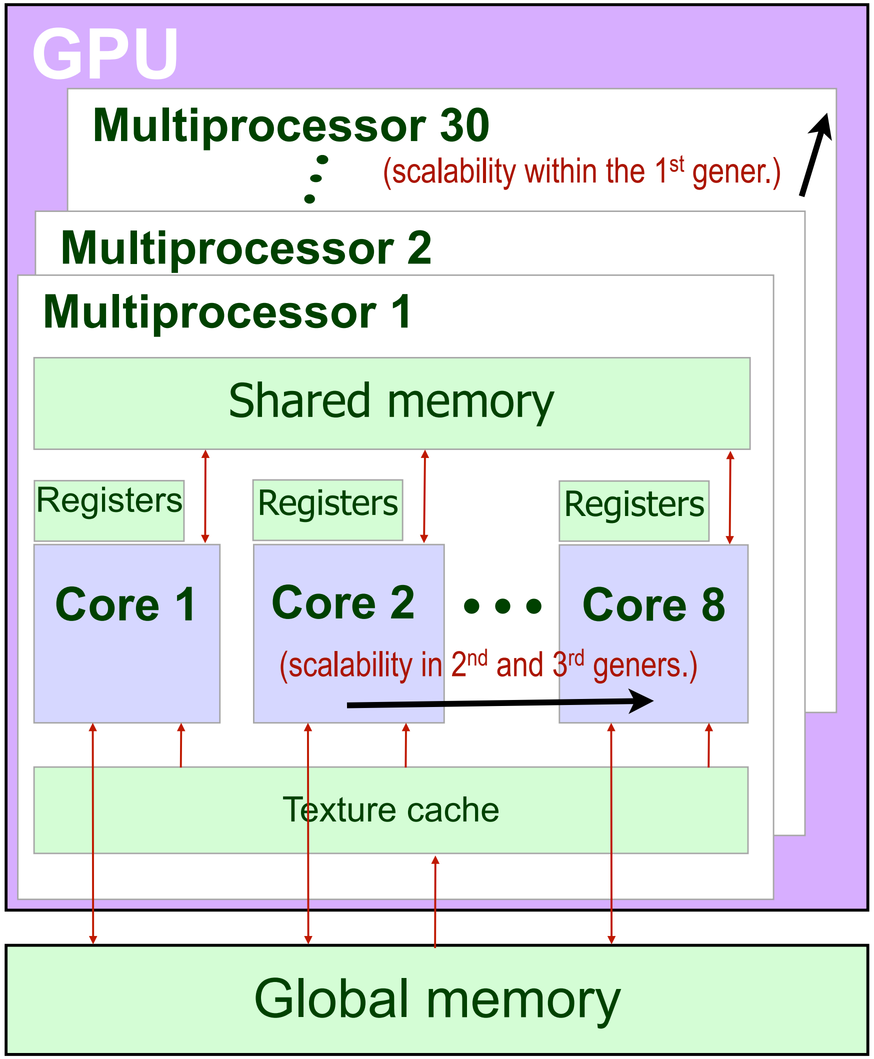
Global memory (up to 1.5 GB) (GDDR3 @ 2x 800 MHz)

The first generation: GT200 (GTX 200)


GPU GTX 200 (around 600 MHz)
Multiprocessor 30
Multiprocessor 2
Multiprocessor 1
(CUDA thread-blocks are mapped onto multiprocessors)
Shared memory (16 KB)
Registers
Registers
Registers
Core 1
(1.30 GHz)
Core 2
• • •
Core 8
Control Unit
(issues SIMD instructions)
(CUDA kernels are mapped to GPU cores)
Texture cache
Global memory (up to 4 GB) (GDDR3, 512 bits @ 2x 1.1GHz = 141.7 GB/s)

Scalability for future generations: Alternatives for increasing performance

- Raise the number of multiprocessors (basic node), that is, we grow over the Z dimension. This is the path followed by 1st gener. (16 to 30).
- Raise the number of processors within a multiprocessor, which means growing over the X dimension. That is what the 2nd and 3rd geners. have done (from 8 to 32 and from there to 192).
- Increment the size of shared memory (extending the Y dim.).



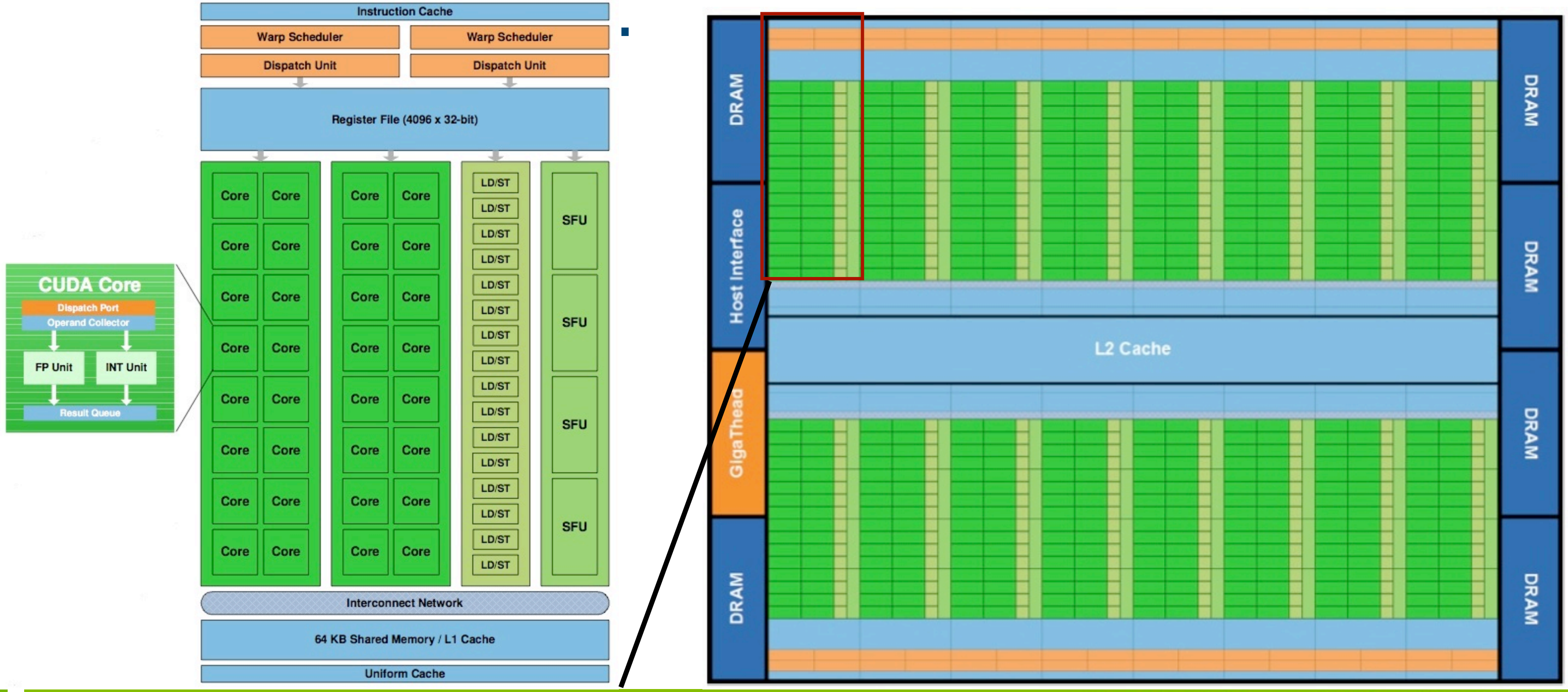
II. 3. Second generation: Fermi (GFxxx)

Fermi hardware compared to its predecessors

GPU architecture	G80	GT200	GF110 (Fermi)
Commercial sample	**GeForce 8800**	**GTX 200**	**GTX 580**
Year released	**2006**	**2008**	**2010**
Number of transistors	681 millions	1400 millions	3000 millions
Integer and fp32 cores	128	240	512
fp64 (double precision)	0	30	256
Double precision floating-point speed	None	30 madds/cycle	256 madds/cycle
Warp scheduler(s)	1	1	2
Shared memory size	16 KB	16 KB	16 KB + 48 KB (or vice versa)
L1 cache size	None	None	
L2 cache size	None	None	768 KB
DRAM error correction	No	No	Yes (elective)
Address bus (width)	32 bits	32 bits	64 bits

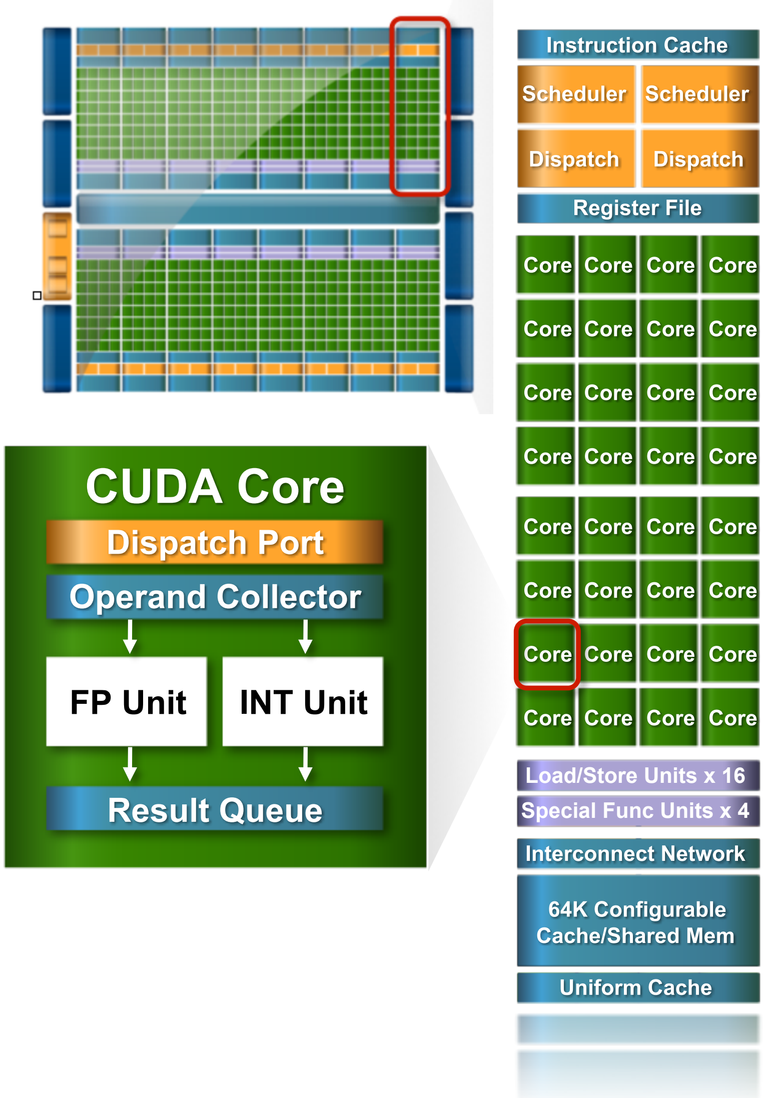
Fermi: An architectural overview

- Up to 512 cores (16 SMs, each endowed with 32 cores).
- Dual scheduler at the front-end of each SM.
- 64 KB. on each SM for shared memory and L1 cache.



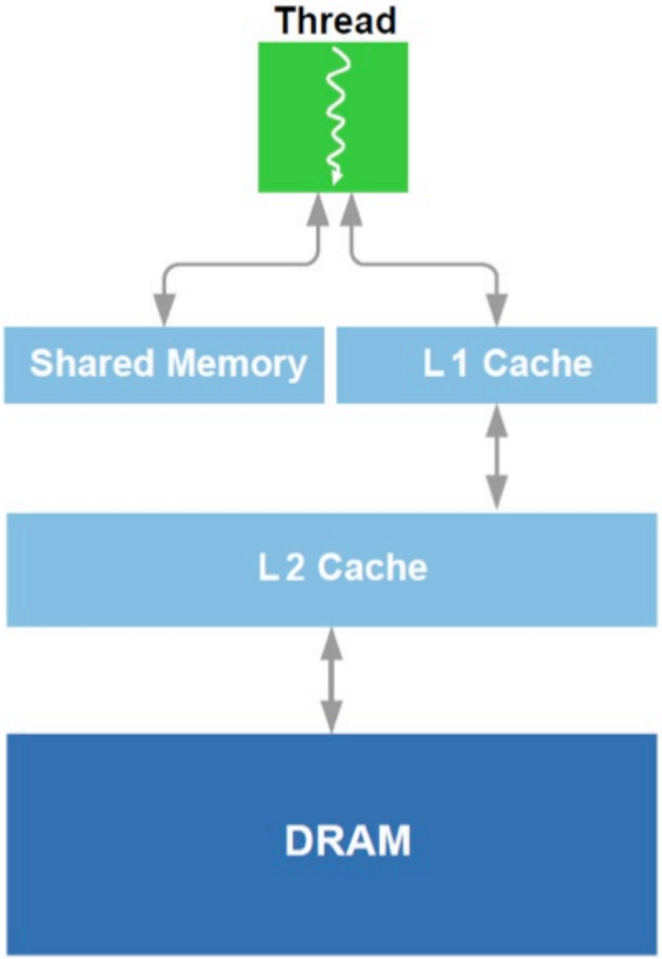
Arithmetic enhancements

- Integer (ALUs):
  - Redesigned to optimize 64-bit integer arithmetic.
  - Extended precision operations.
- Fused instructions ("madd"):
  - Available for both single and double precision data types.
- Floating-point (FPUs):
  - Implements IEEE-754 format on its recent 2008 upgrade, which was ahead of most CPUs.



The memory hierarchy

- Fermi is the first GPU with a L1 cache, combined with shared memory for a total of 64 KB for each SM (32 cores). 64 KB are split into 3:1 or 1:3 proportions (programmer's choice).
- There is also a L2 cache of 768 KB. with data coherence shared by all multiprocessors (SMs).

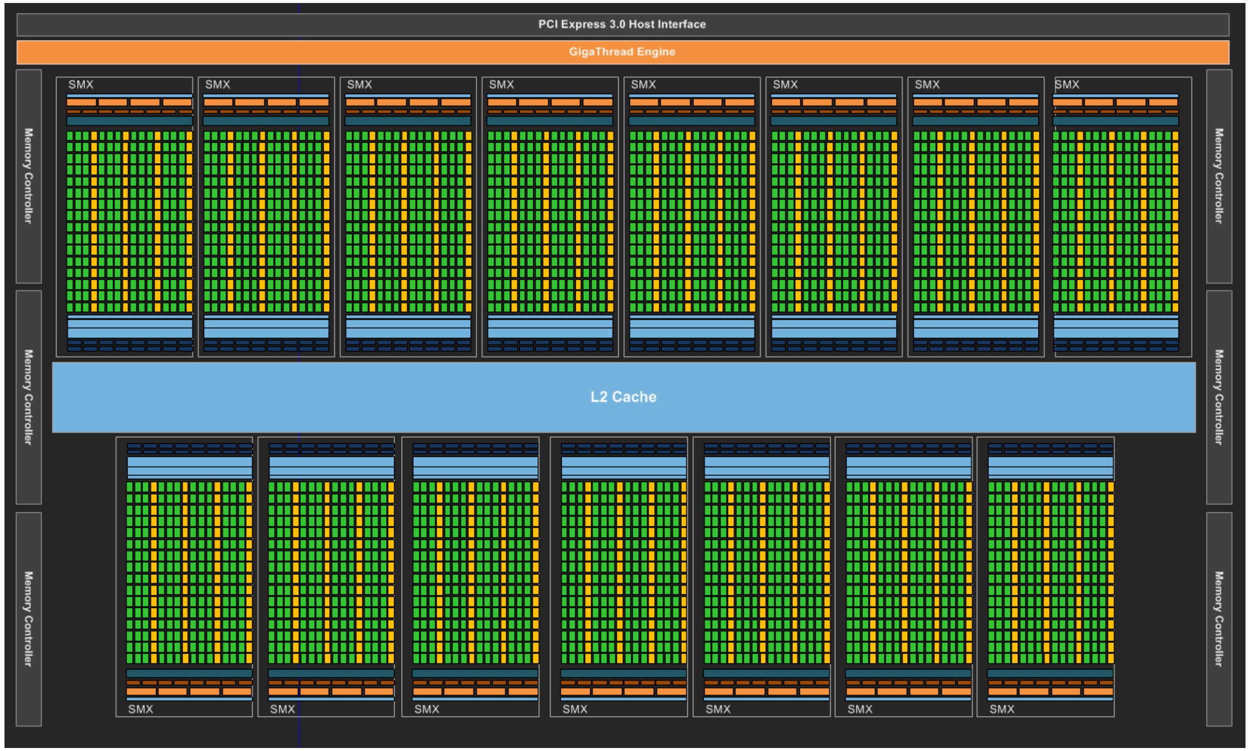


II. 4. Third generation: Kepler (GKxxx)

NVIDIA

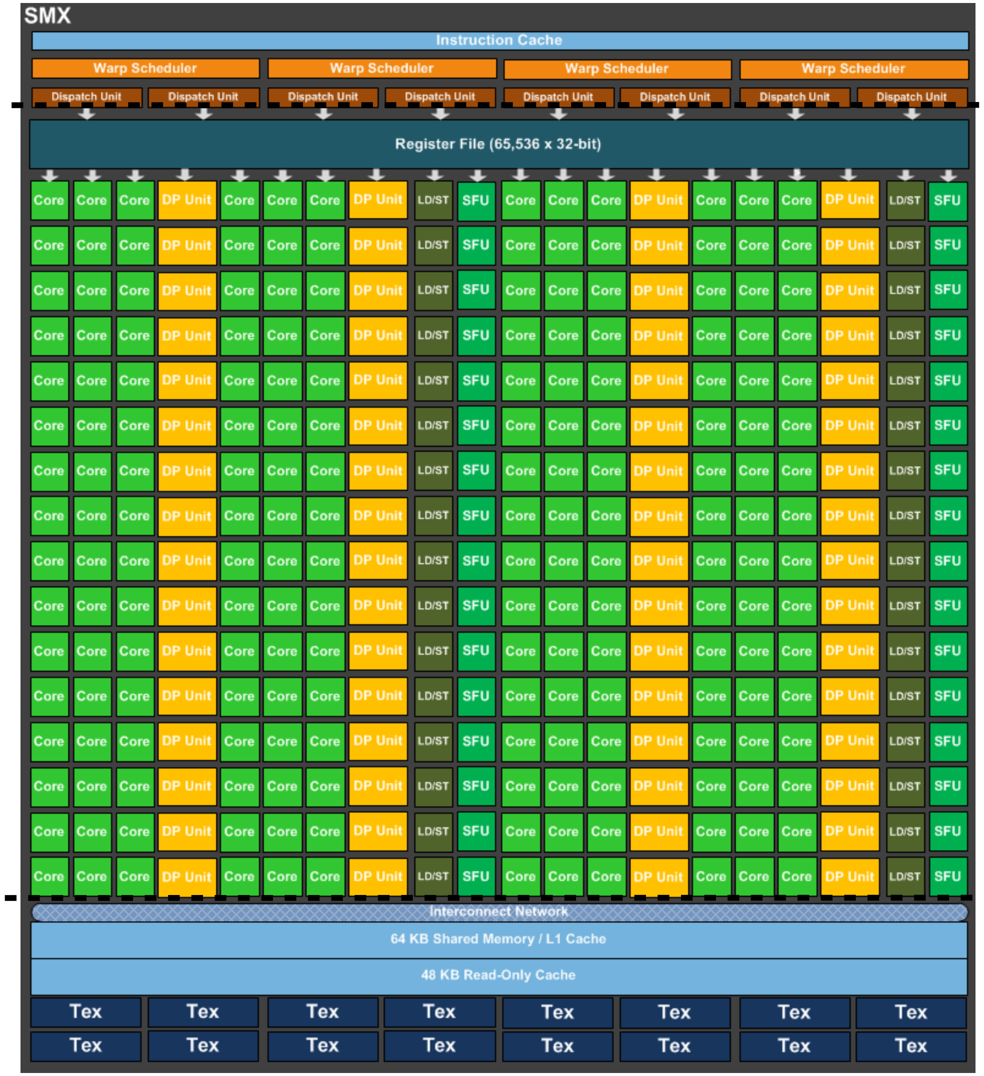
Kepler GK110 Block Diagram

- 7.1 billion transistors.
- 15 SMX multiprocs.
- > 1 TFLOP FP64.
- 1.5 MB L2 Cache.
- 384-bit GDDR5.
- PCI Express Gen3.



The SMX multiprocessor

Instruction scheduling and issuing in **warps**



Front-end

Instructions execution.
512 functional units:

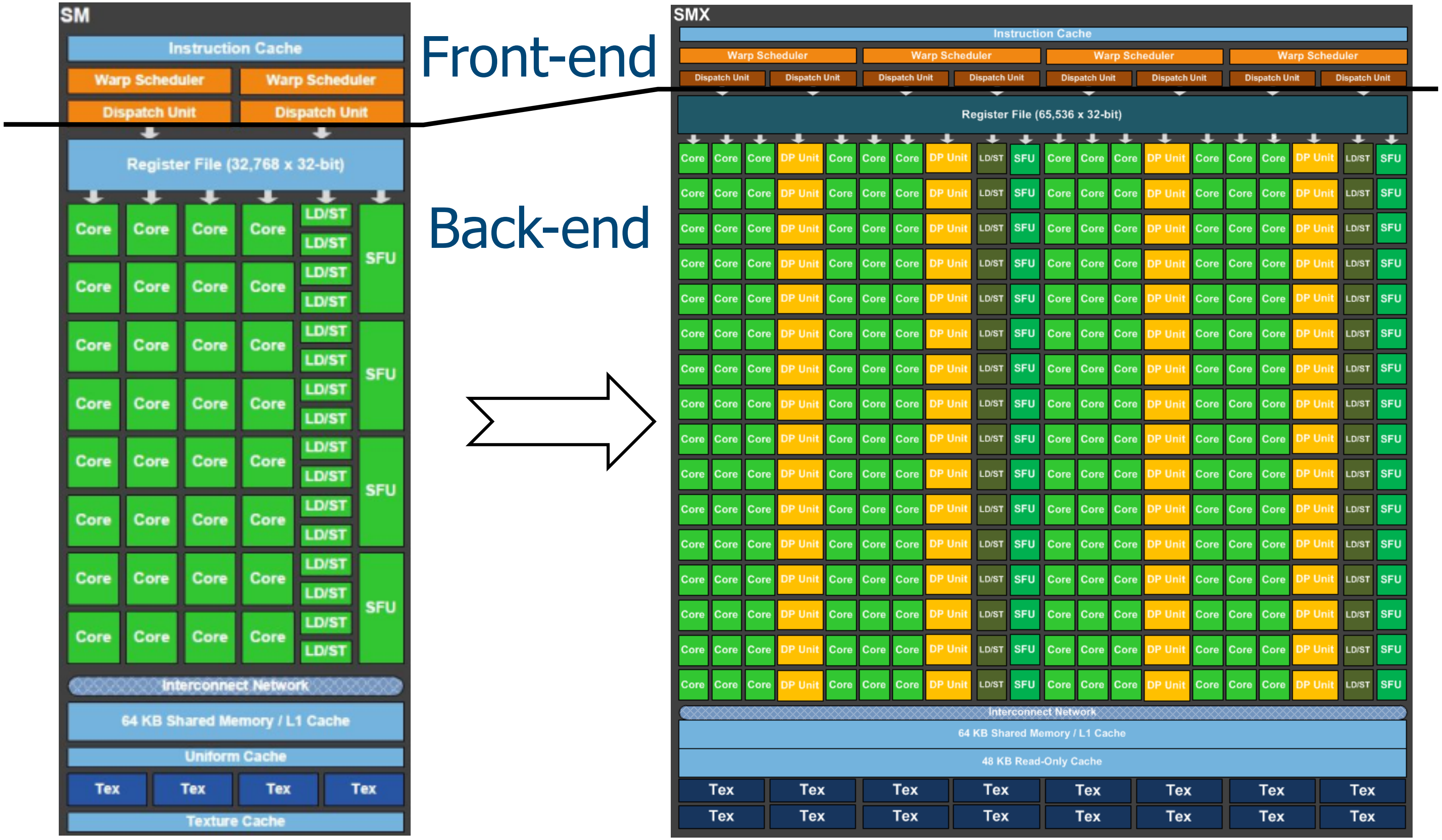
- 192 for ALUs.
- 192 for FPUs S.P.
- 64 for FPUs D.P.
- 32 for load/store.
- 32 for SFUs (log,sqrt, ...)

Back-end

Memory access

Interface

From SM multiprocessor in Fermi GF100 to SMX multiprocessor in Kepler GK110



SM

Instruction Cache

Warp Scheduler | Warp Scheduler

Dispatch Unit | Dispatch Unit

Register File (32,768 x 32-bit)

Core Core Core Core LD/ST LD/ST SFU
Core Core Core Core LD/ST LD/ST
Core Core Core Core LD/ST LD/ST SFU
Core Core Core Core LD/ST LD/ST
Core Core Core Core LD/ST LD/ST SFU
Core Core Core Core LD/ST LD/ST
Core Core Core Core LD/ST LD/ST SFU
Core Core Core Core LD/ST LD/ST

Interconnect Network

64 KB Shared Memory / L1 Cache

Uniform Cache

Tex Tex Tex Tex

Texture Cache

Front-end

Back-end

SMX

Instruction Cache

Warp Scheduler | Warp Scheduler | Warp Scheduler | Warp Scheduler

Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit | Dispatch Unit

Register File (65,536 x 32-bit)

Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU

Interconnect Network

64 KB Shared Memory / L1 Cache

48 KB Read-Only Cache

Tex Tex Tex Tex Tex Tex Tex Tex
Tex Tex Tex Tex Tex Tex Tex Tex

SMX Balance of Resources

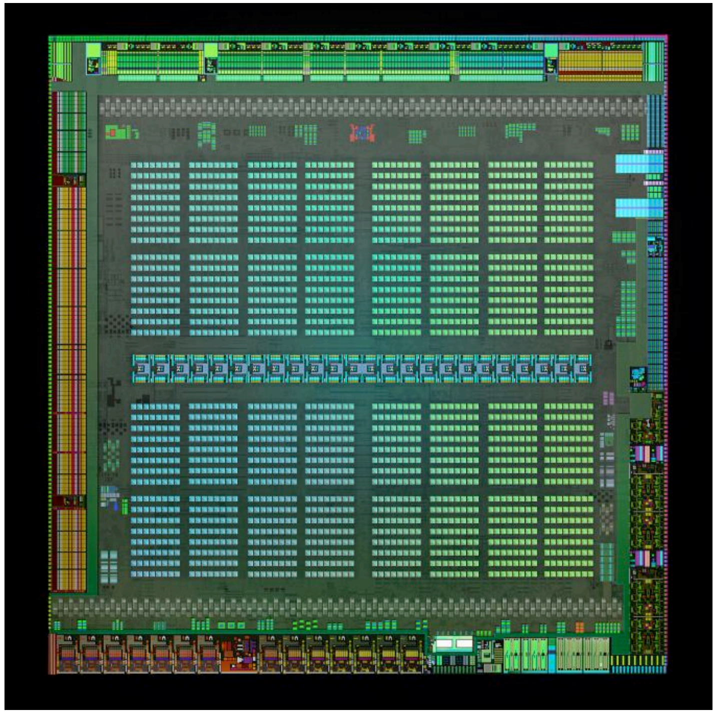
Resource	Kepler GK110 vs. Fermi GF100
Floating-point throughput	2-3x
Maximum number of blocks per SMX	2x
Maximum number of threads per SMX	1.3x
Register file bandwidth	2x
Register file capacity	2x
Shared memory bandwidth	2x
Shared memory capacity	1x
L2 bandwidth	2x
L2 cache capacity	2x

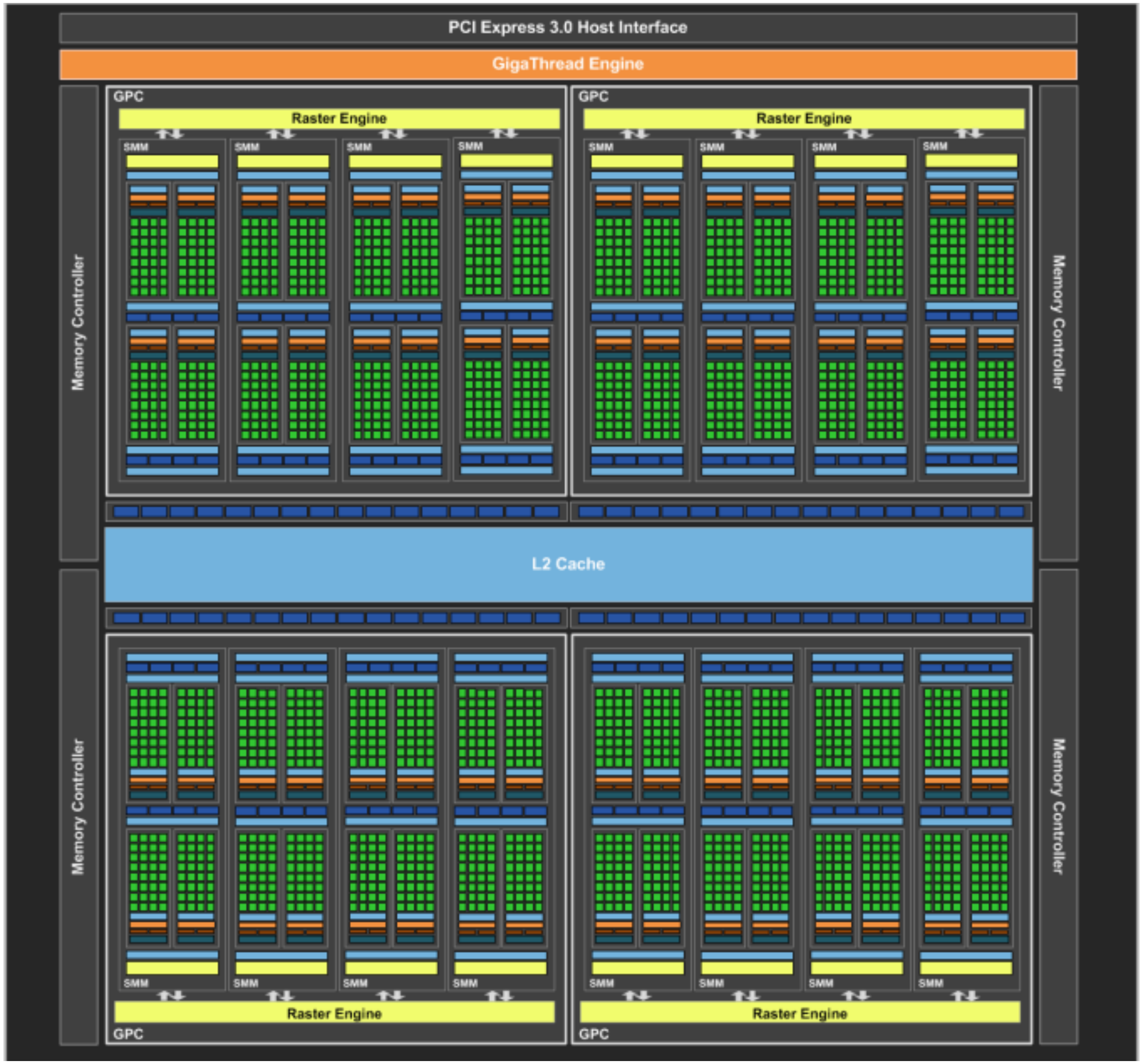
II. 5. Fourth generation: Maxwell (GMxxx)

NVIDIA

Maxwell and SMM multiprocessors (for GeForce GTX 980, 16 SMMs)

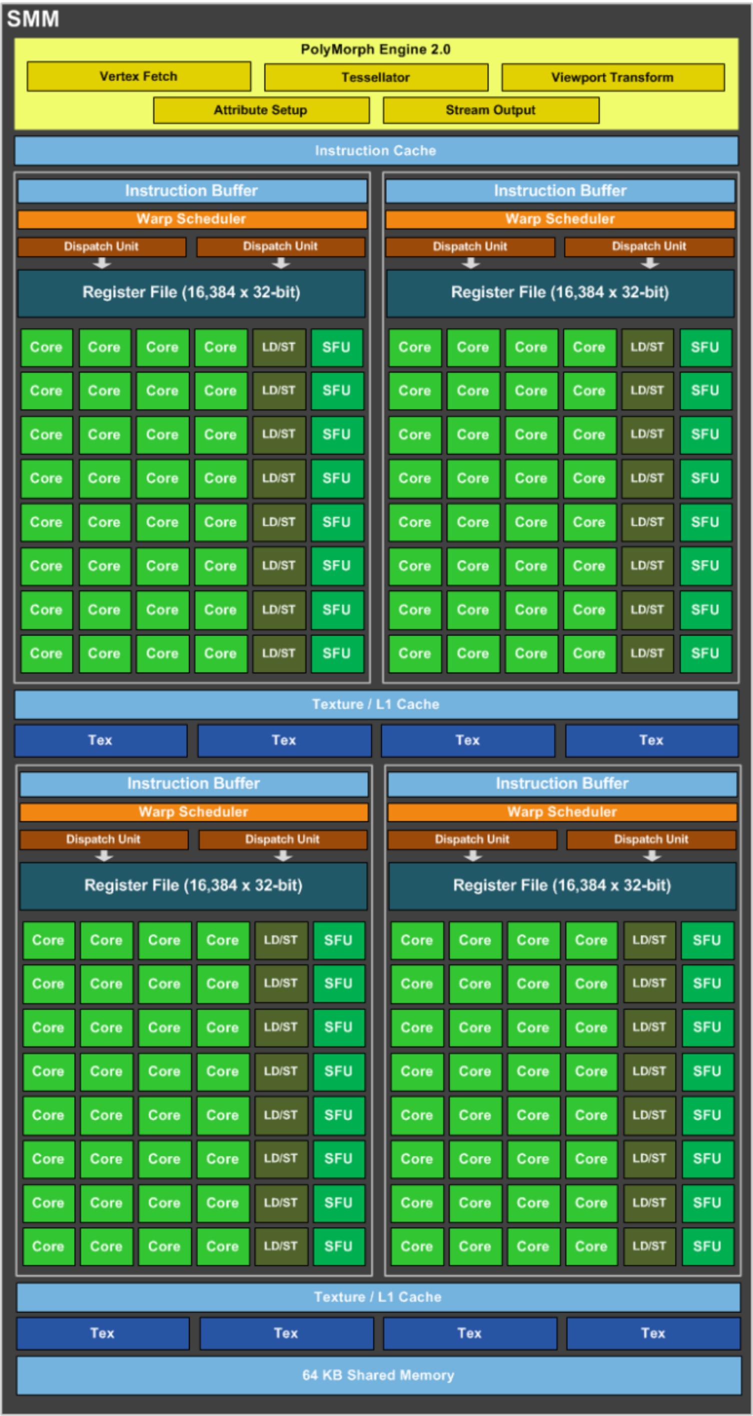
- 1870 Mt.
- 148 mm^2.



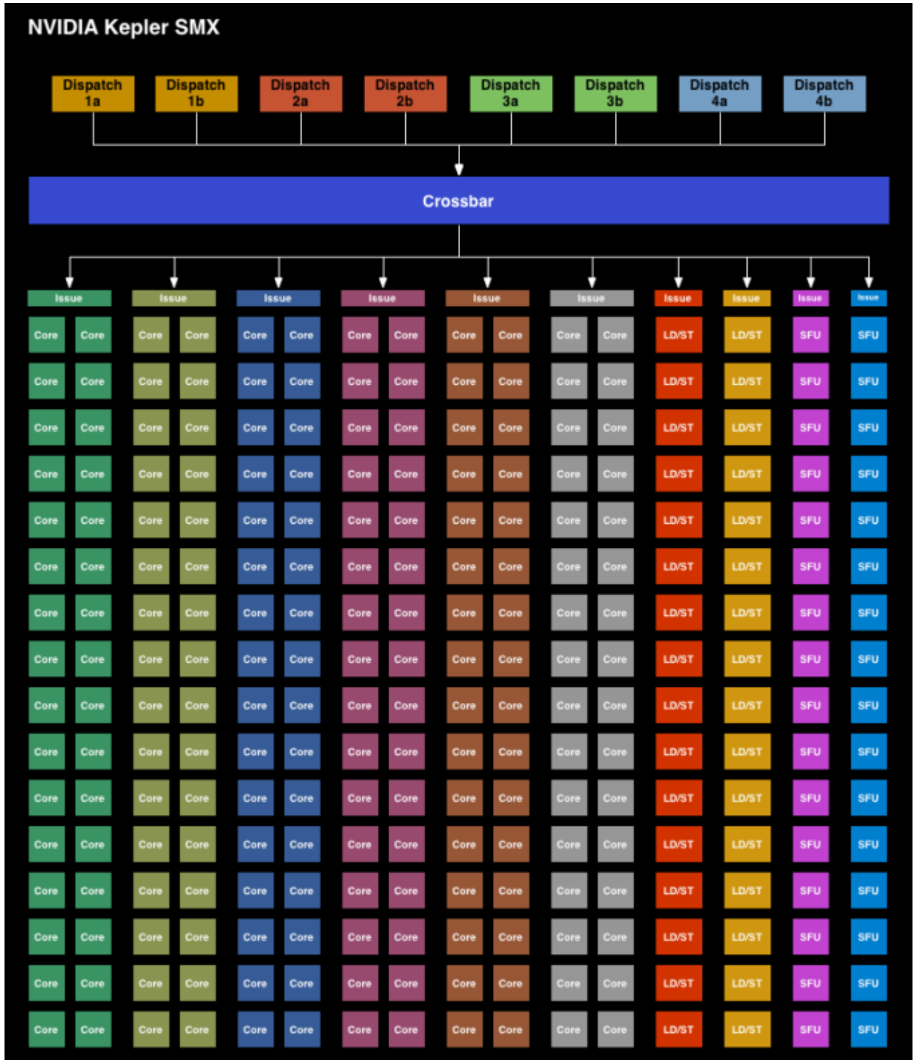


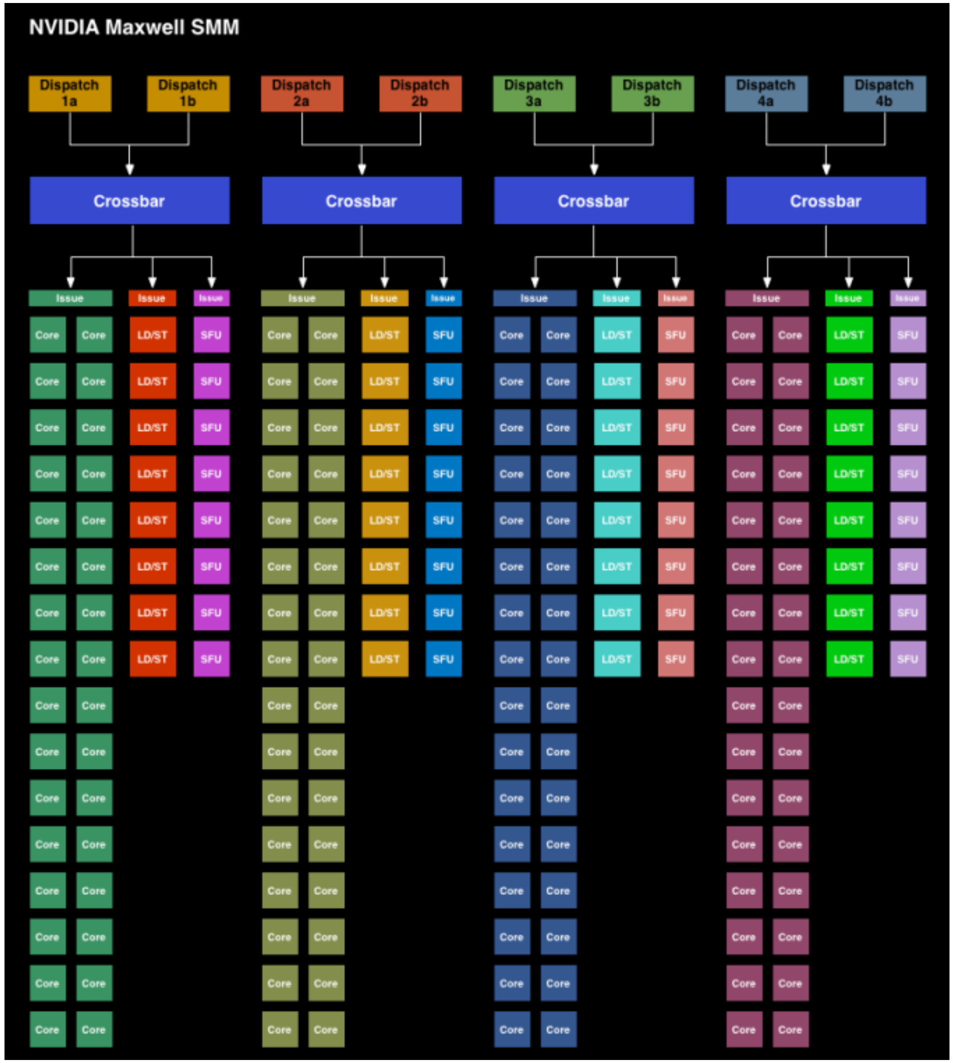
The SMMs

- Keep the same 4 warp schedulers, and the same LD/ST and SFU units.
- Reduce the number of cores for `int` and `float`: from 192 to 128 units.



A comparison versus Kepler

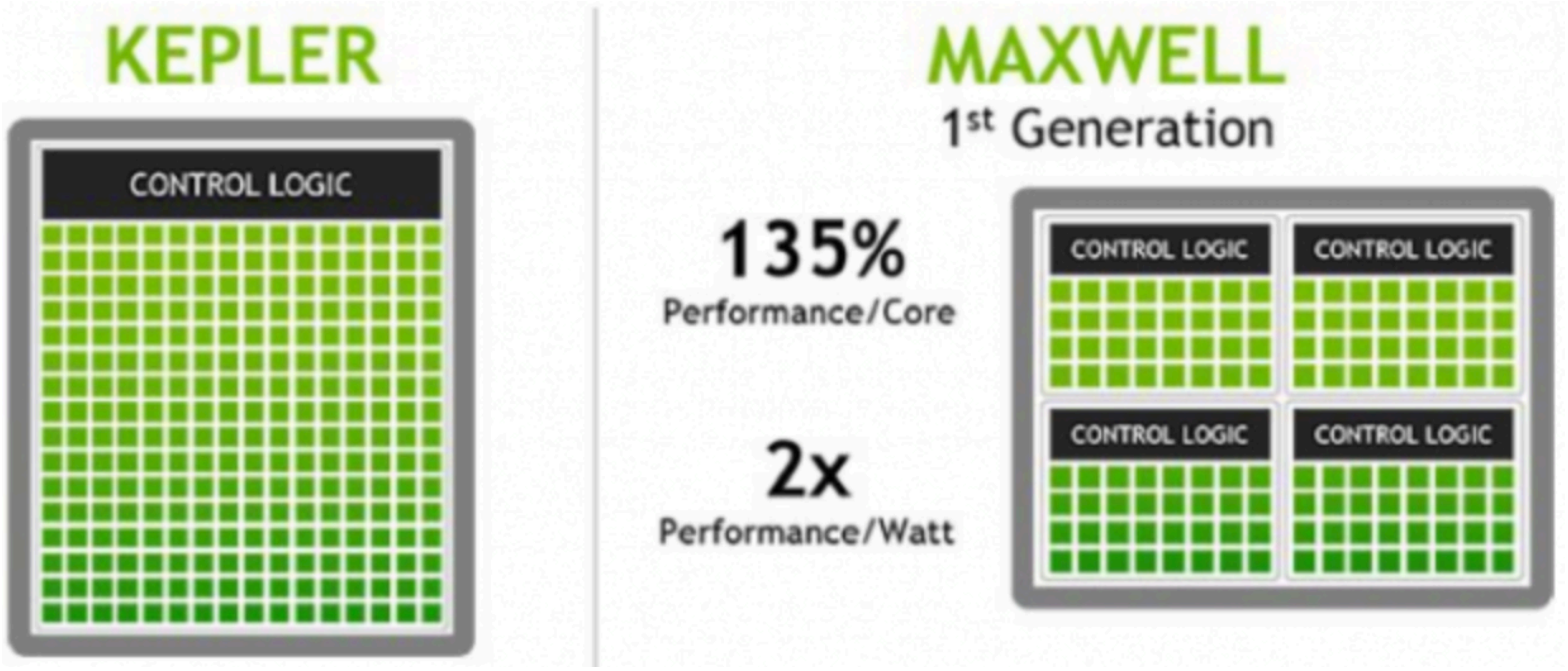




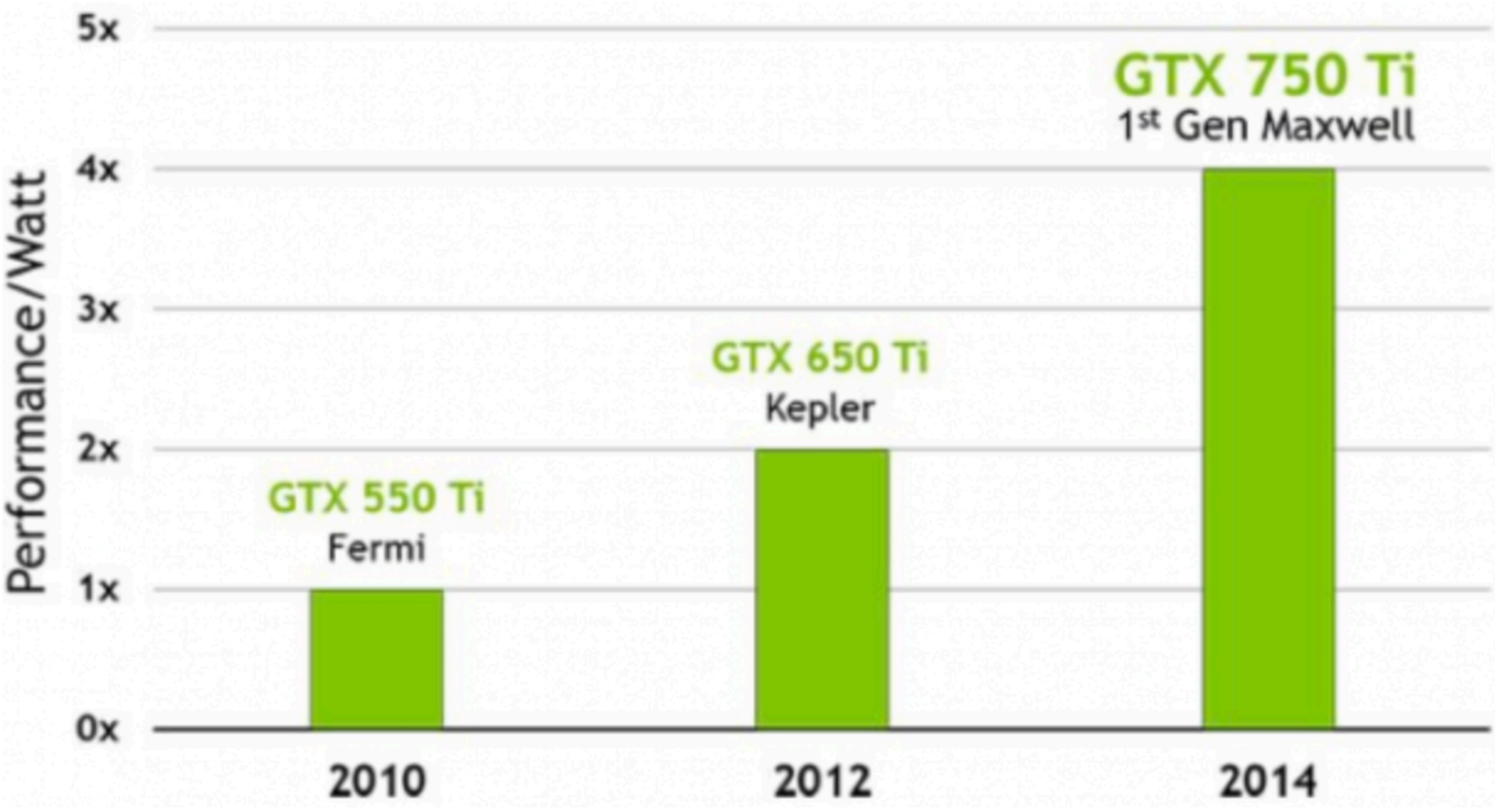
Some commercial models for CCC 5.2 (all @ 28 nm)

GeForce	GTX 950	GTX 960	GTX 970	GTX980	GTX 980 Ti	Titan X
Release date	Aug’15	Aug’15	Sep’14	Sep’14	Jun’15	Mar’15
GPU (code name)	GM206-250	GM206-300	GM204-200	GM204-400	GM200-310	GM200-400
Multiprocessors	6	8	13	16	22	24
Number of cores	768	1024	1664	2048	2816	3072
Cores frequency (MHz)	1024-1188	1127-1178	1050-1178	1126-1216	1000-1075	1000-1075
DRAM bus width	128 bits	128 bits	256 bits	256 bits	384 bits	384 bits
DRAM frequency	2x 3.3 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz	2x 3.5 GHz
DRAM bandwidth	105.6 GB/s	112 GB/s	224 GB/s	224 GB/s	336.5 GB/s	336.5 GB/s
GDDR5 memory size	2 GB	2 GB	4 GB	4 GB	6 GB	12 GB
Million of transistors	2940	2940	5200	5200	8000	8000
Die size	228 mm^2	228 mm^2	398 mm^2	398 mm^2	601 mm^2	601 mm^2
Maximum TDP	90 W	120 W	145 W	165 W	250 W	250 W
Power connectors	1 x 6 pines	1 x 6 pines	2 x 6 pines	2 x 6 pines	6 + 8 pines	6 + 8 pines
Price ($ upon release)	149	199	329	549	649	999

Major enhancements

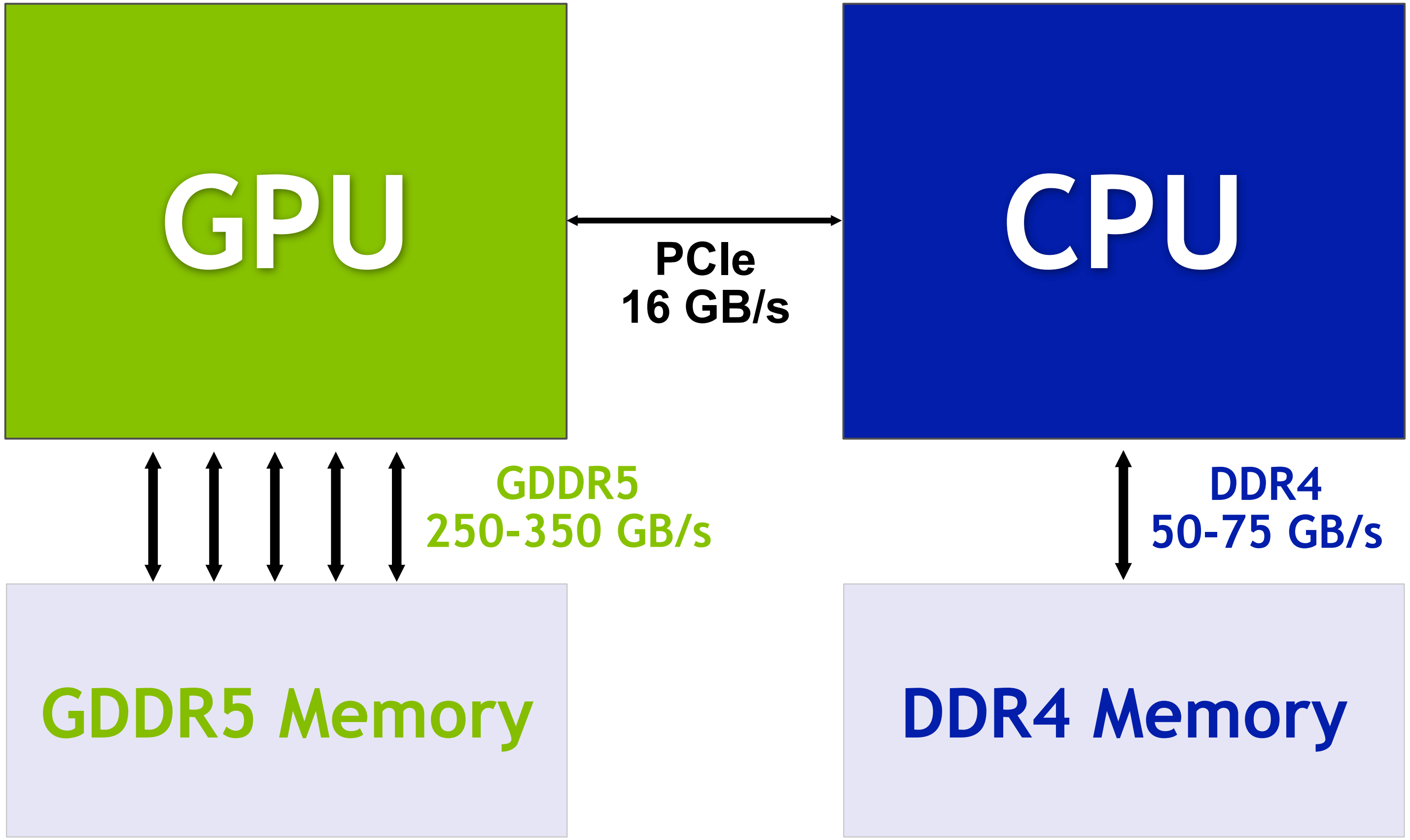


Power efficiency

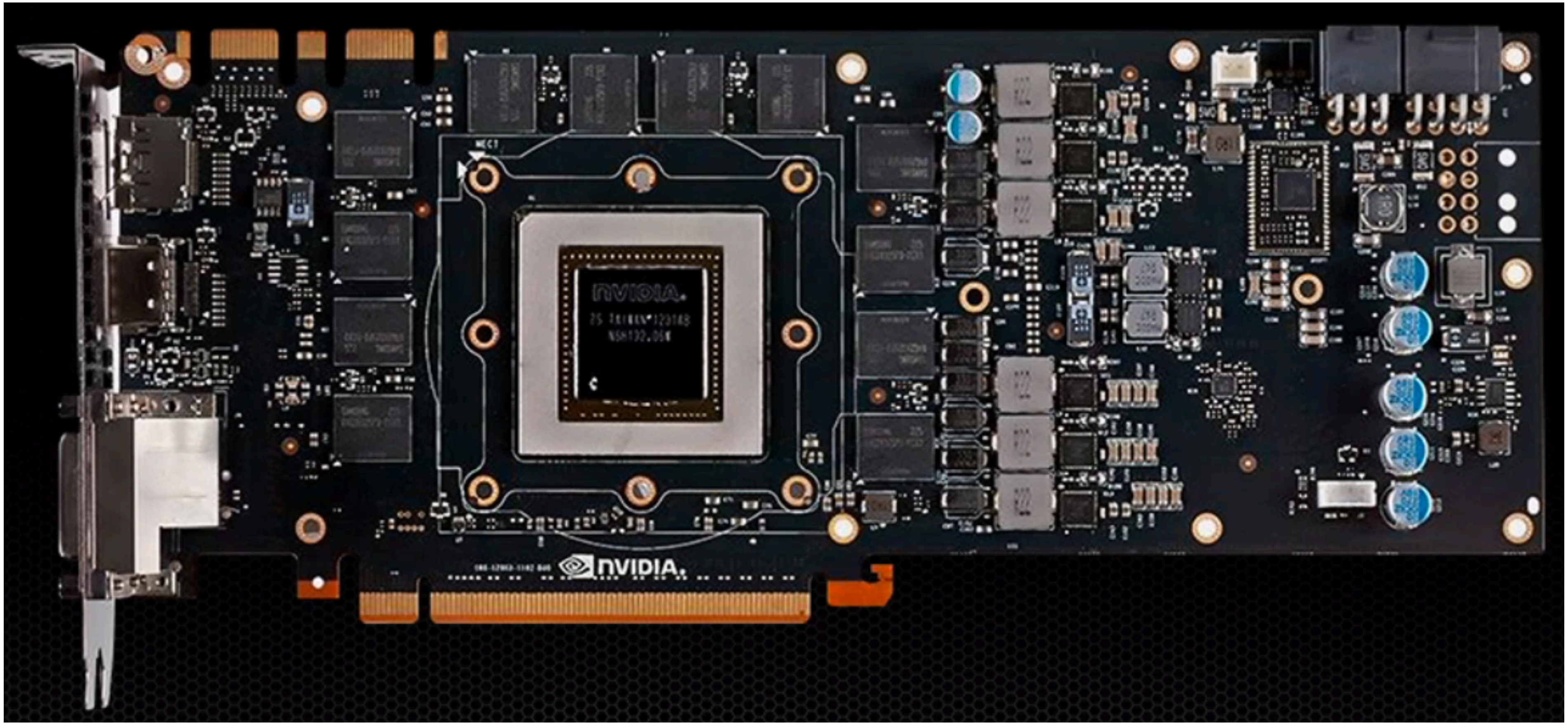


II. 6. Fifth generation: Pascal (GPxxx)

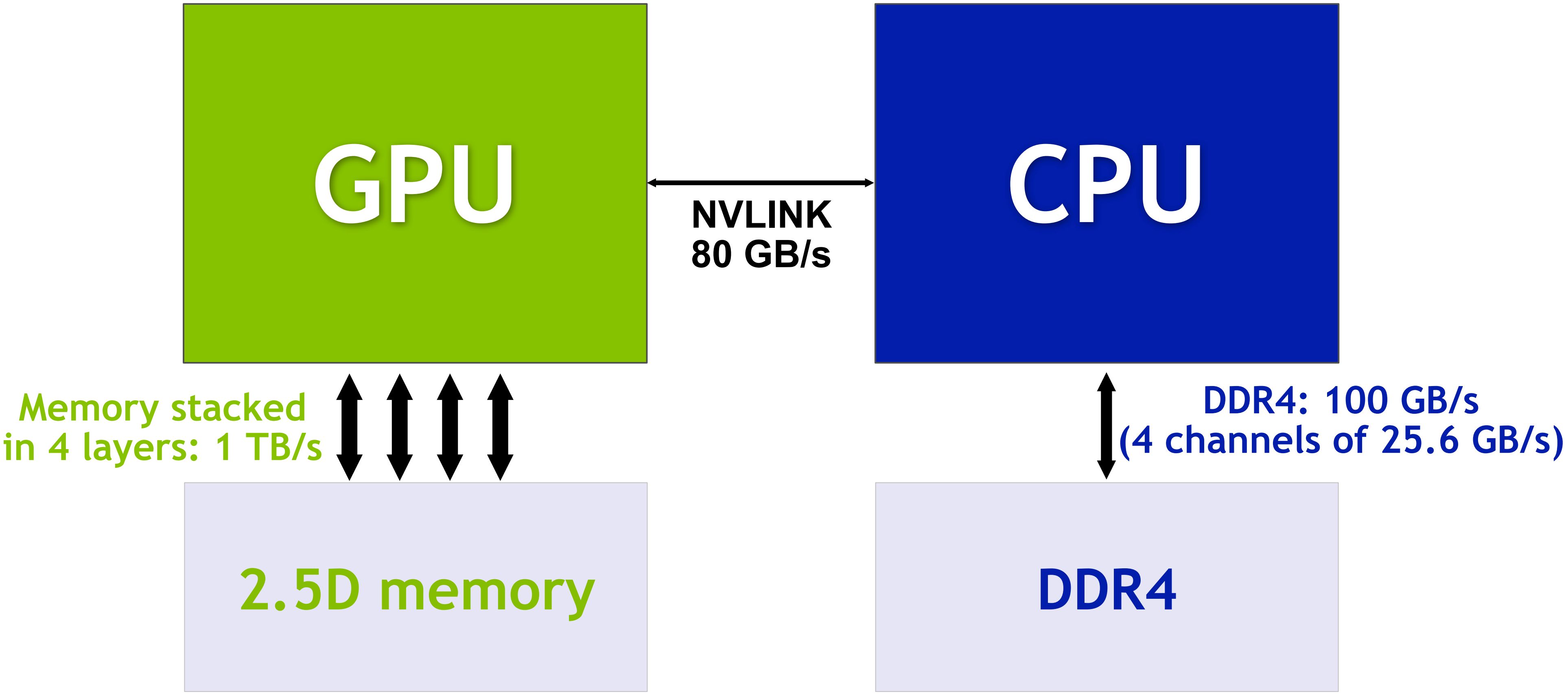
Today


GPU
CPU
PCIe
16 GB/s
GDDR5
250-350 GB/s
DDR4
50-75 GB/s
GDDR5 Memory
DDR4 Memory

A 2015 graphics card: Kepler/Maxwell GPU with GDDR5 memory



In 2017



GPU

NVLINK
80 GB/s

CPU

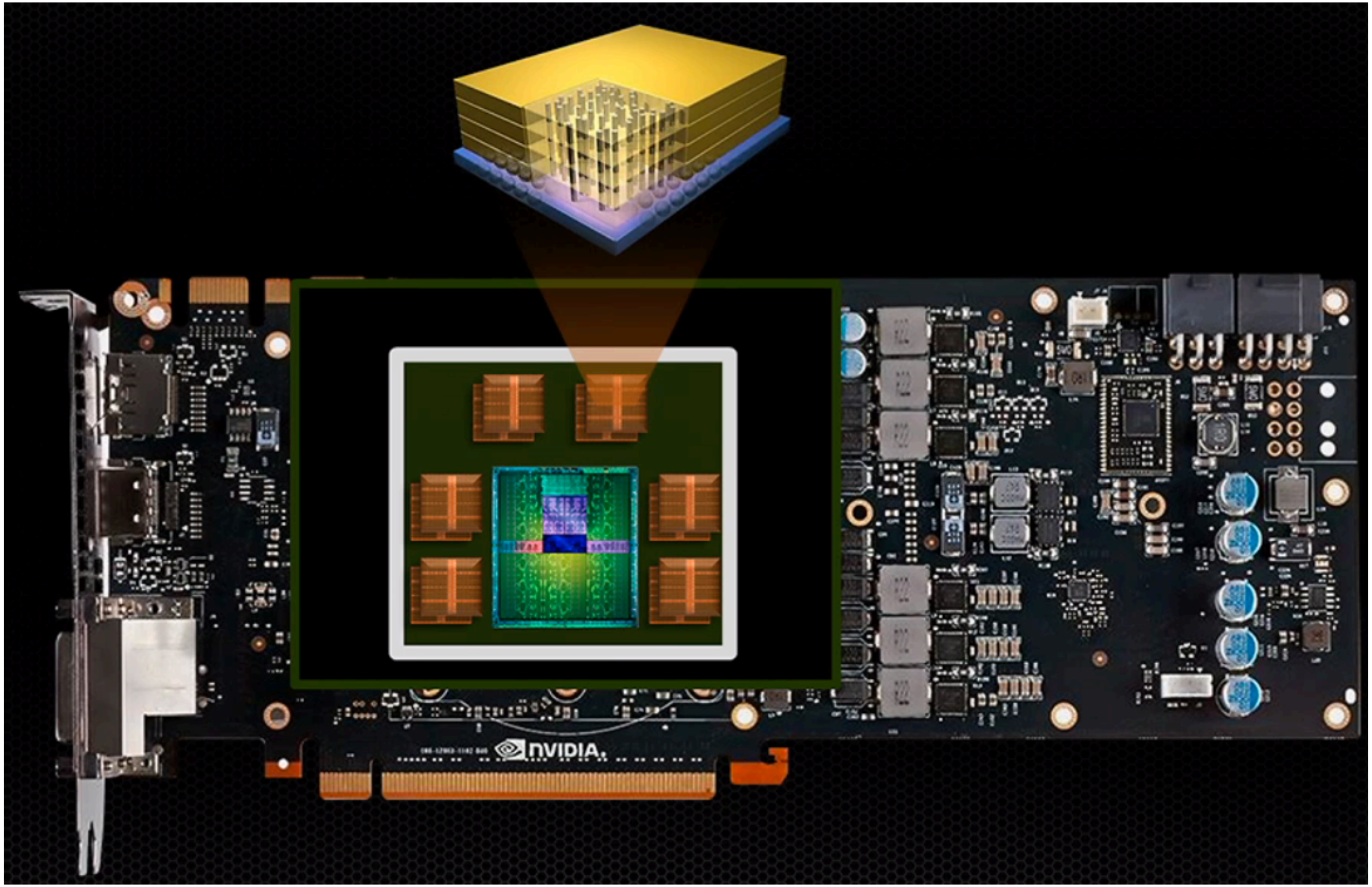
Memory stacked
in 4 layers: 1 TB/s

DDR4: 100 GB/s
(4 channels of 25.6 GB/s)

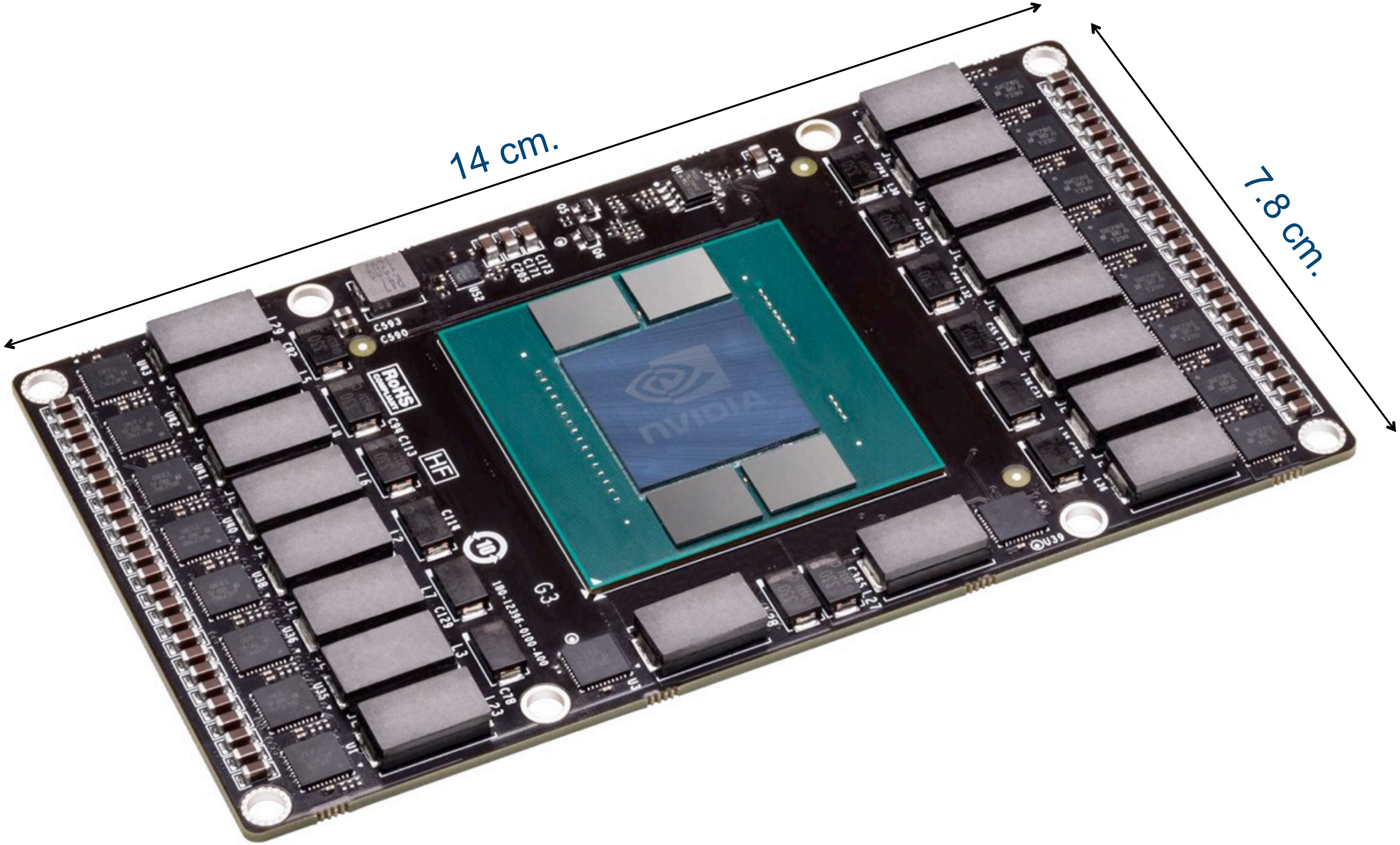
2.5D memory

DDR4

A 2017 graphics card: Pascal GPU with Stacked DRAM



A Pascal GPU prototype



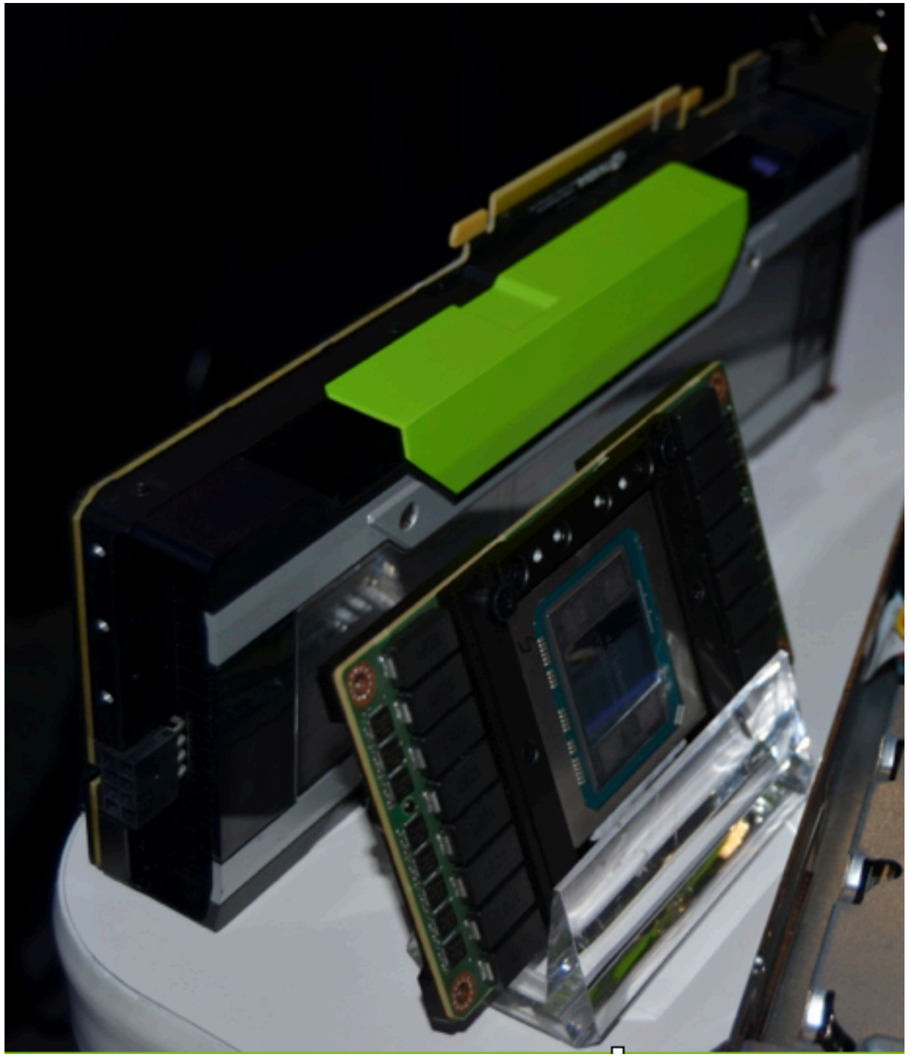
First commercial model: GeForce GTX 1080. Comparative with the previous 2 generations

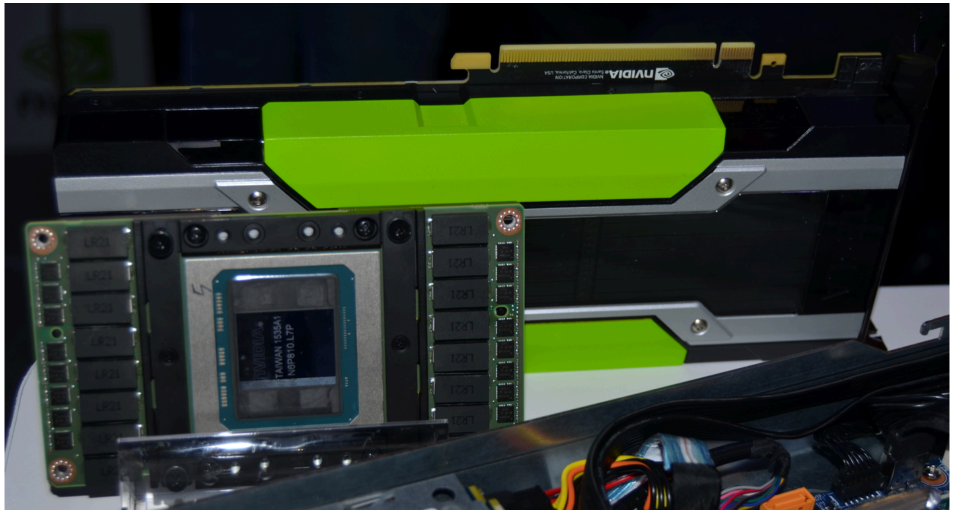
	GTX 680 (Kepler)	GTX 980 (Maxwell)	GTX 1080 (Pascal)
Year	2012	2014	2016
Transistors	3.54 B @ 28 nm.	5.2 B @ 28 nm.	7.2 B @ 16 nm.
Power consumption & die size	195 W & 294 mm^2	165 W & 398 mm^2	180 W & 314 mm^2
Multiprocessors	8	16	40
Cores / Multiproc.	192	128	64
Cores / GPU	1536	2048	2560
Clock (wo. and w. GPU Boost)	1006, 1058 MHz	1126, 1216 MHz	1607, 1733 MHz
Peak performance	3250 GFLOPS	4980 GFLOPS	8873 GFLOPS
Shared memory	16, 32, 48 KB	64 KB	
L1 cache size	48, 32, 16 KB	Integrated with texture cache	
L2 cache size (smaller than Tesla models)	512 KB	2048 KB	
DRAM memory: Interface	256-bit GDDR5	256-bit GDDR5	256-bit GDDR5X
DRAM memory: Frequency	2x 3000 MHz	2x 3500 MHz	4x 2500 MHz
DRAM memory: Bandwidth	192.2 GB/s	224 GB/s	320 GB/s

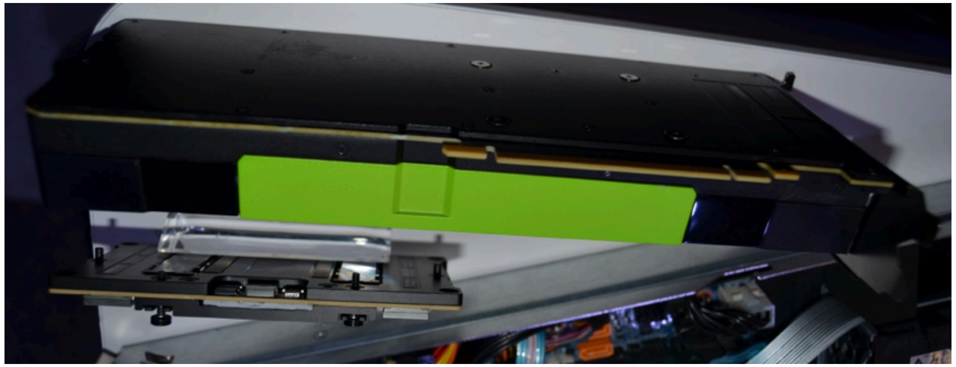
First Tesla model for Pascal: P100. Comparative with 2 previous generations

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 w. NV-link	P100 w. PCI-e
Release date	2012	November, 2015	April, 2016	
Transistors	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET (610 mm^2)	
# of multiprocessors	15	24	56	
fp32 cores / Multiproc.	192	128	64	
fp32 cores / GPU	2880	3072	**3584**	
fp64 cores / Multiproc.	64	4	32	
fp64 cores / GPU	960 (1/3 fp32)	96 (1/32 fp32)	1792 (1/2 fp32)	
Clock frequency	745,810,875 MHz	948, 1114 MHz	1328, 1480 MHz	1126, 1303 MHz
Thermal Design Power	235 W	250 W	300 W	250 W
Peak performance (DP)	1680 GFLOPS	213 GFLOPS	**5304 GFLOPS**	4670 GFLOPS
L2 cache size	1536 KB	3072 KB	4096 KB	
Memory interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	
Memory size	Up to 12 GB	Up to 24 GB	**16 GB**	
Memory bandwidth	288 GB/s	288 GB/s	**720 GB/s**	

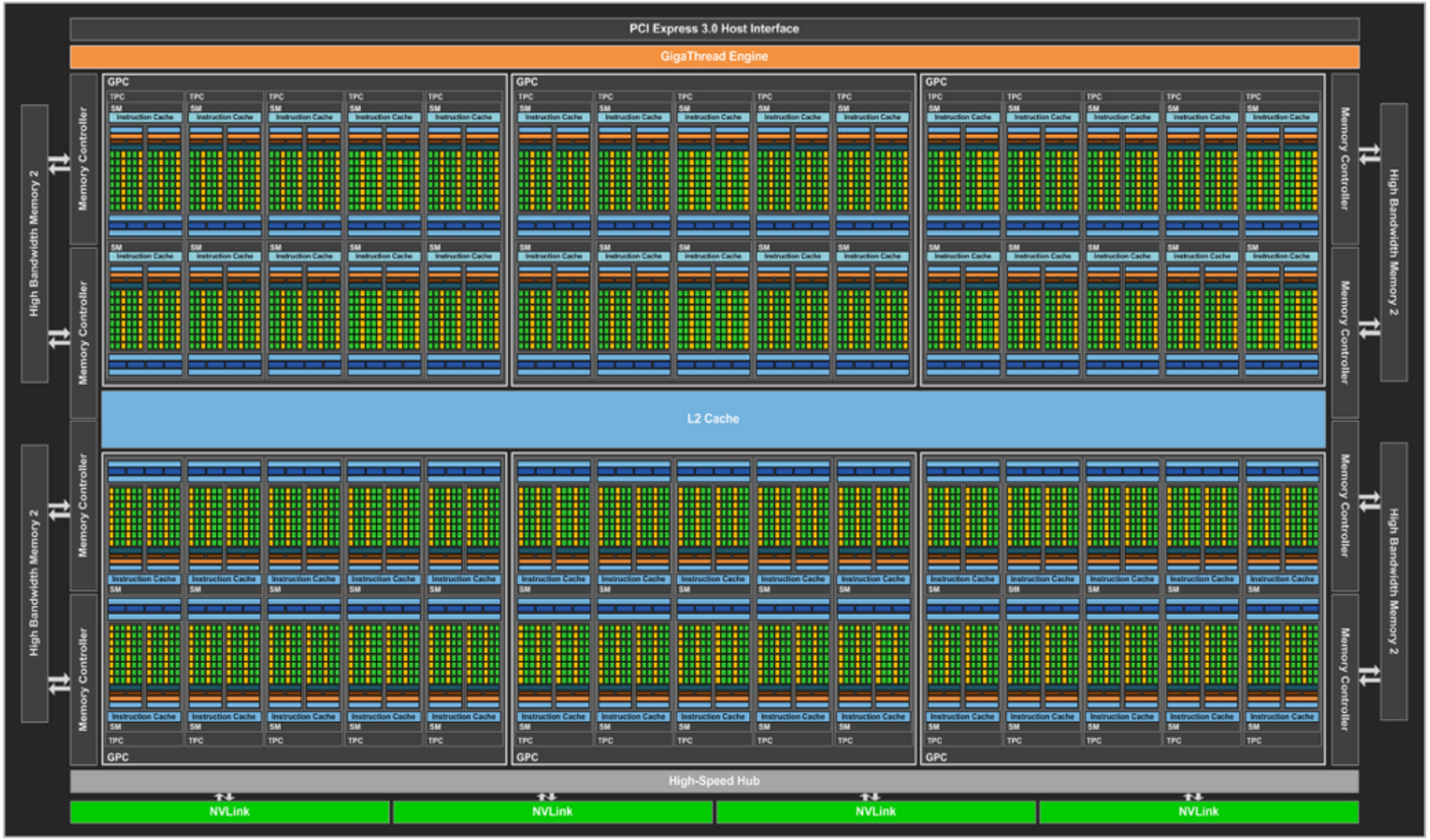
The two form factors:
PCI-e Slot vs. NVLink Socket (SXM2)



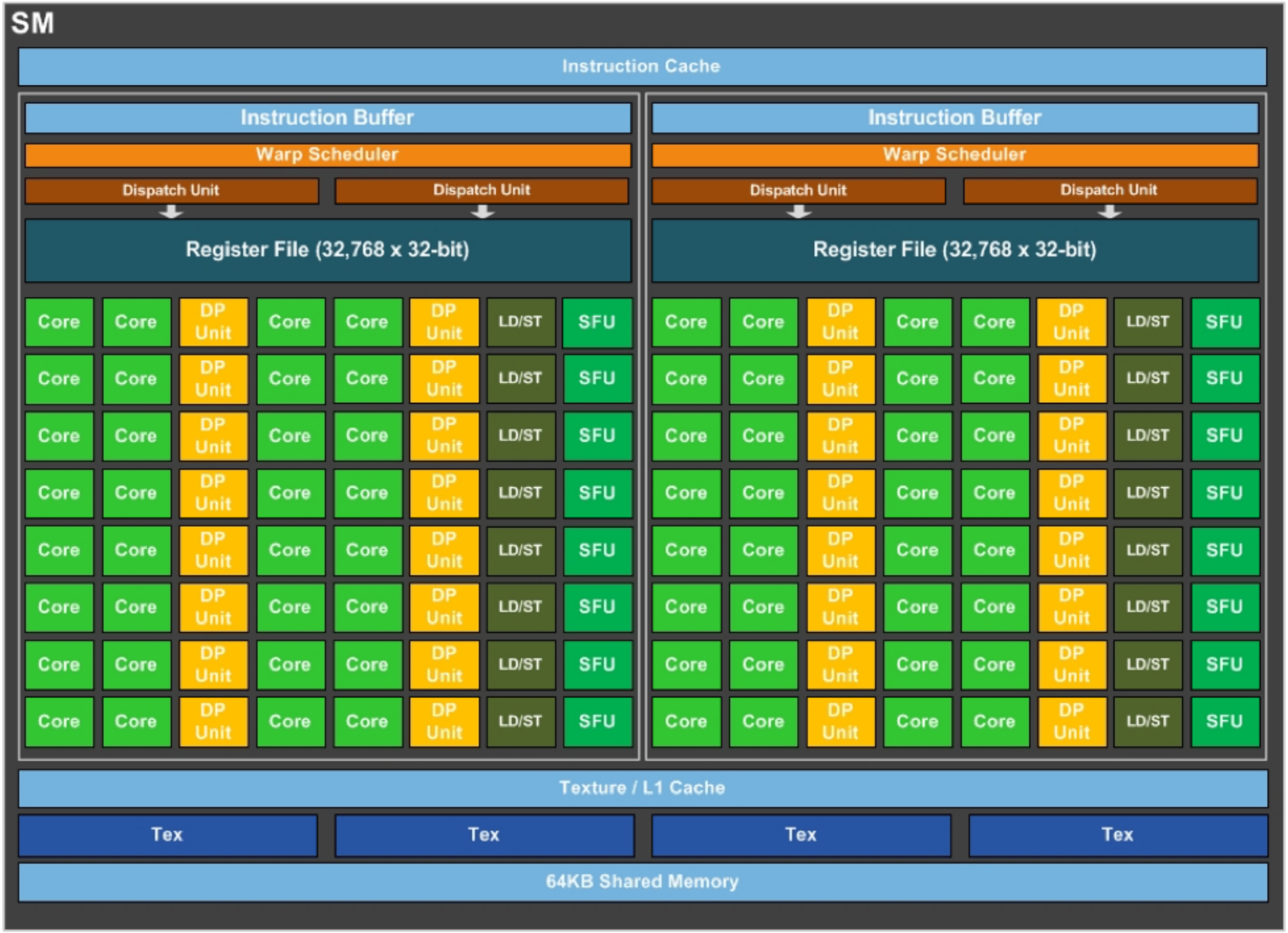




The physical layout for multiprocessors, memory controllers and buses



Pascal multiprocessor

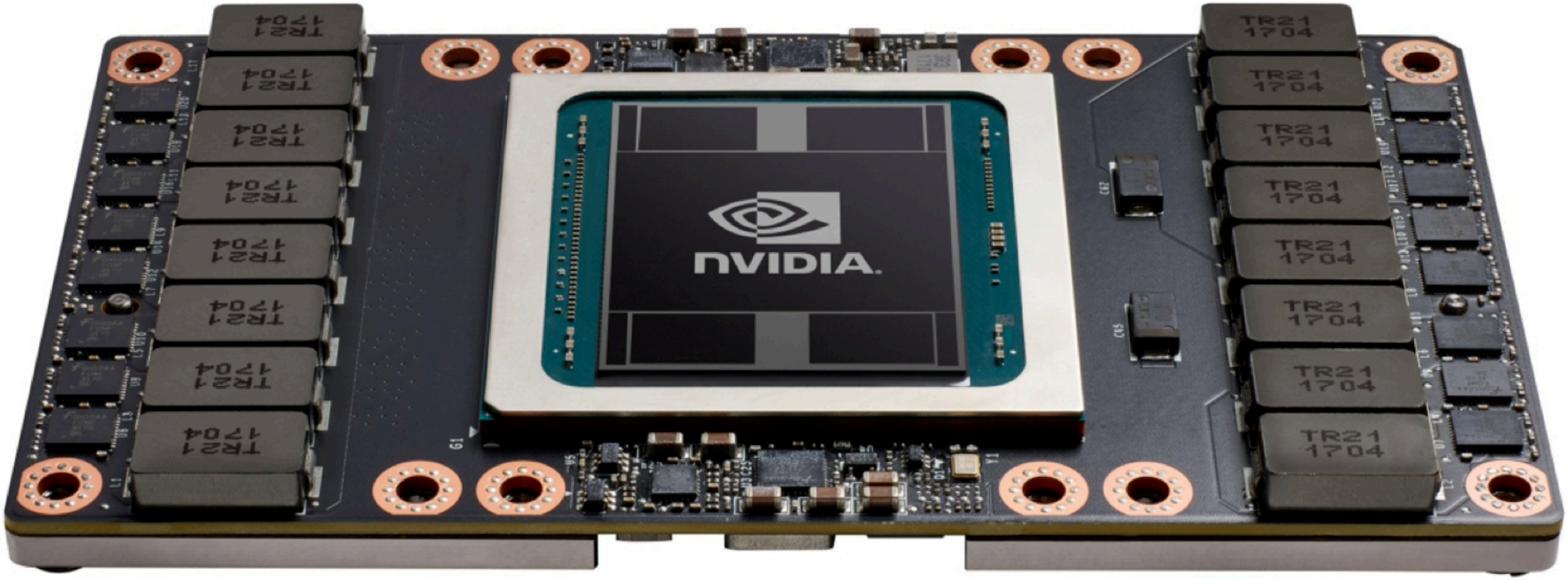


II. 7. Sixth generation: Volta (GVxxx)

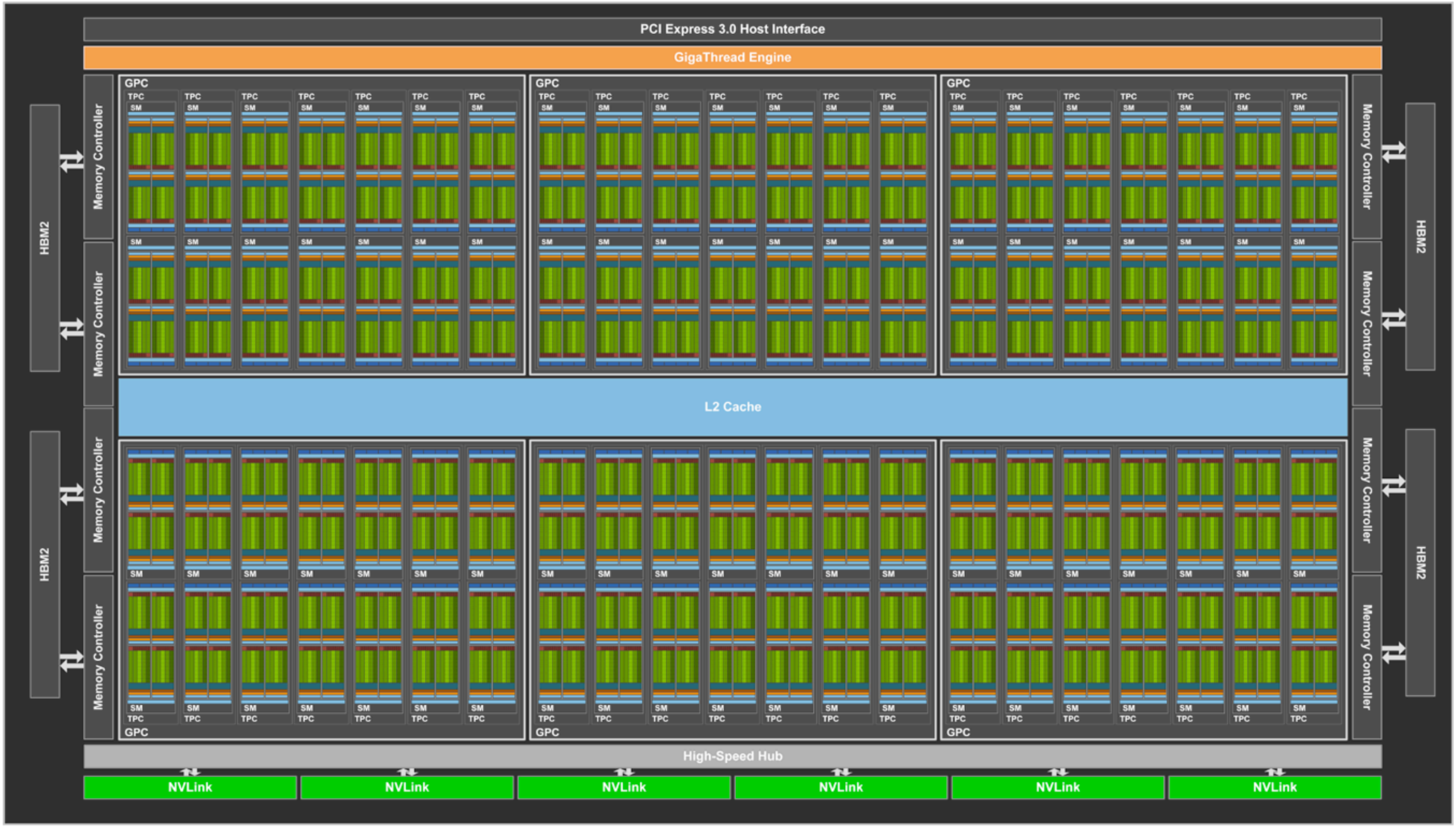
Comparison with Tesla models in previous generations

	K40 (Kepler)	M40 (Maxwell)	P100 (Pascal)	V100 (Volta)
GPU (chip)	GK110	GM200	GP100	GV100
Million of transistors	7100	8000	15300	21100
Die size	551 mm^2	601 mm^2	610 mm^2	815 mm^2
Manufacturing process	28 nm.	28 nm.	16 nm. FinFET	12 nm. FFN
Thermal Design Power	235 W.	250 W.	300 W.	300 W.
Number of fp32 cores	2880 (15 x 192)	3072 (24 x 128)	3584 (56 x 64)	5120 (80 x 64)
Number of fp64 units	960	96	1792	2560
Frequency (regular & boost)	745 & 875 MHz	948 & 1114 MHz	1328 & 1480 MHz	1370 & 1455 MHz
TFLOPS (fp16, fp32, fp64)	No, 5.04, 1.68	No, 6.8, 2.1	21.2, 10.6, 5.3	30, 15, 7.5
Memory interface	384-bit GDDR5		4096-bit HBM2	
Video memory	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 cache	1536 KB	3072 KB	4096 KB	6144 KB
Shared memory / SM	48 KB	96 KB	64 KB	Up to 96 KB
Register file / SM	65536	65536	65536	65536

This is how the commercial product looks like

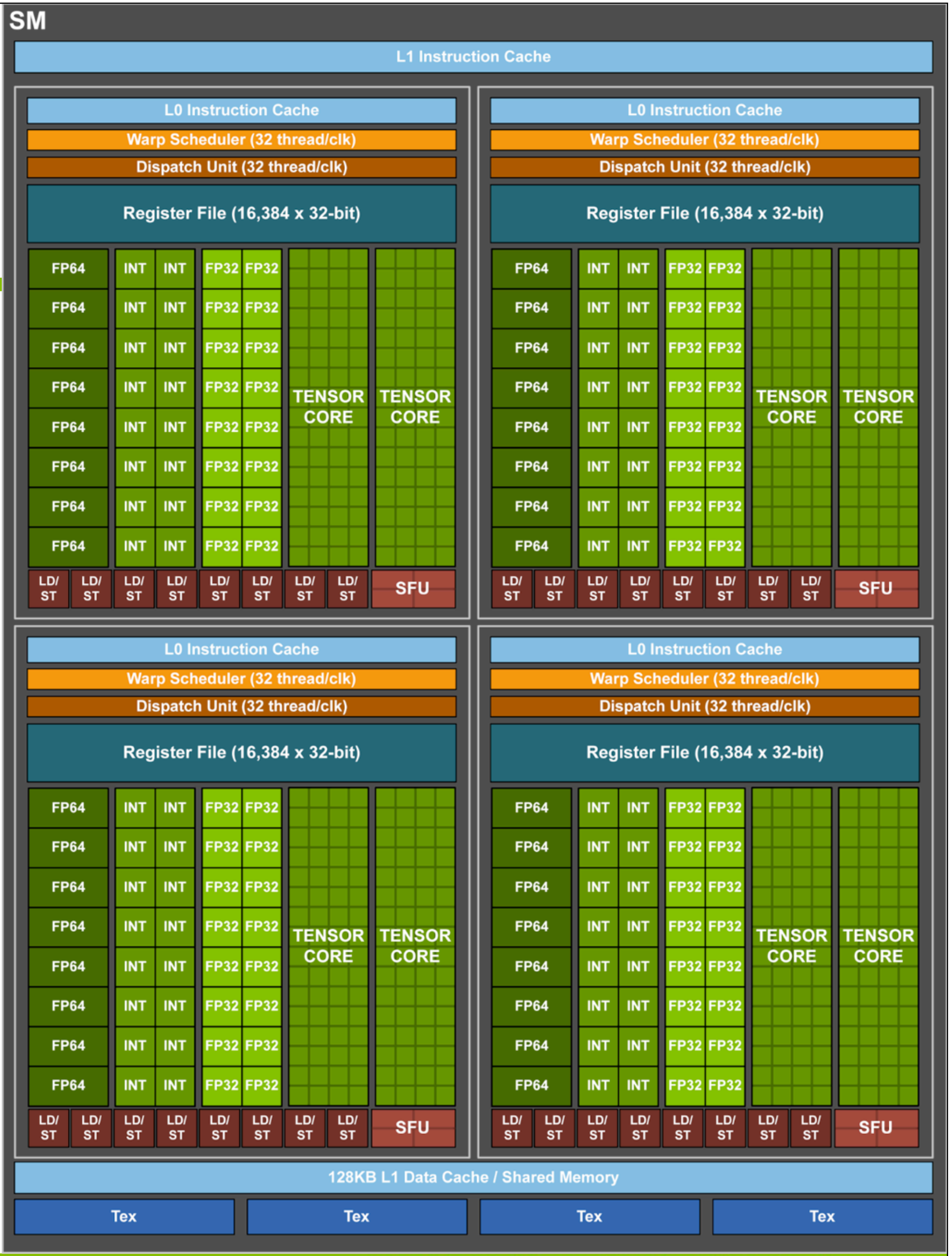


The GV100 GPU: 6 GPC, 84 SM, 42 TPC and 8 512-bit memory controllers (Tesla V100 uses only 80 SMs)



The Volta SM

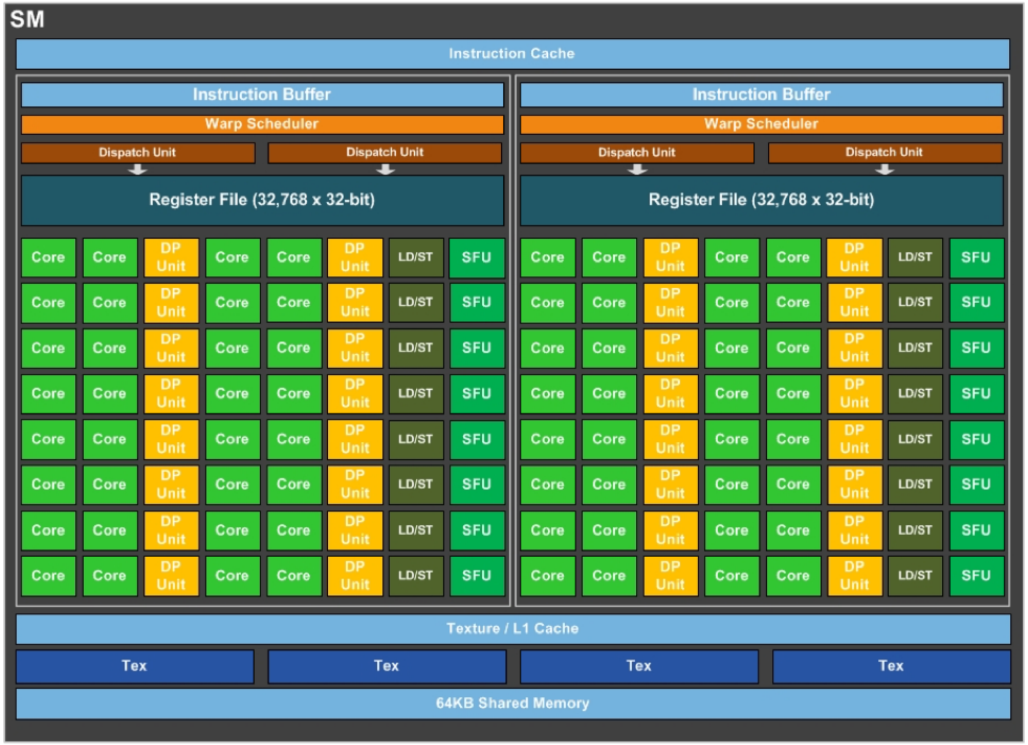
- Cores:
  - 64 int32 (“int”).
  - 64 fp32 (“float”).
  - 32 fp64 (“double”).
  - 8 tensor units.
- Units:
  - 8 load/store.
  - 4 textures.
- Memory:
  - 64K 32-bit registers.
  - L0 instruction cache (replacing instruction buffers)
  - 128 KB L1 Data/Shared.


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
Tex
Tex
Tex

The Volta SM partitioning versus Pascal SM

	GP100 SM	GV100 SM
Processing sets ("cloned templates")	2	4
int32 cores / set	32	16
fp32 cores / set	32	16
fp64 cores / set	16	8
Tensor cores / set	None	2
L0 instruction cache / set	None (instruction buffer instead)	1
Register file / set	128 K	64 K
Warp schedulers / set	1	1
Dispatch units / set	1	1

Multiprocesador evolution: From Pascal to Volta



SM

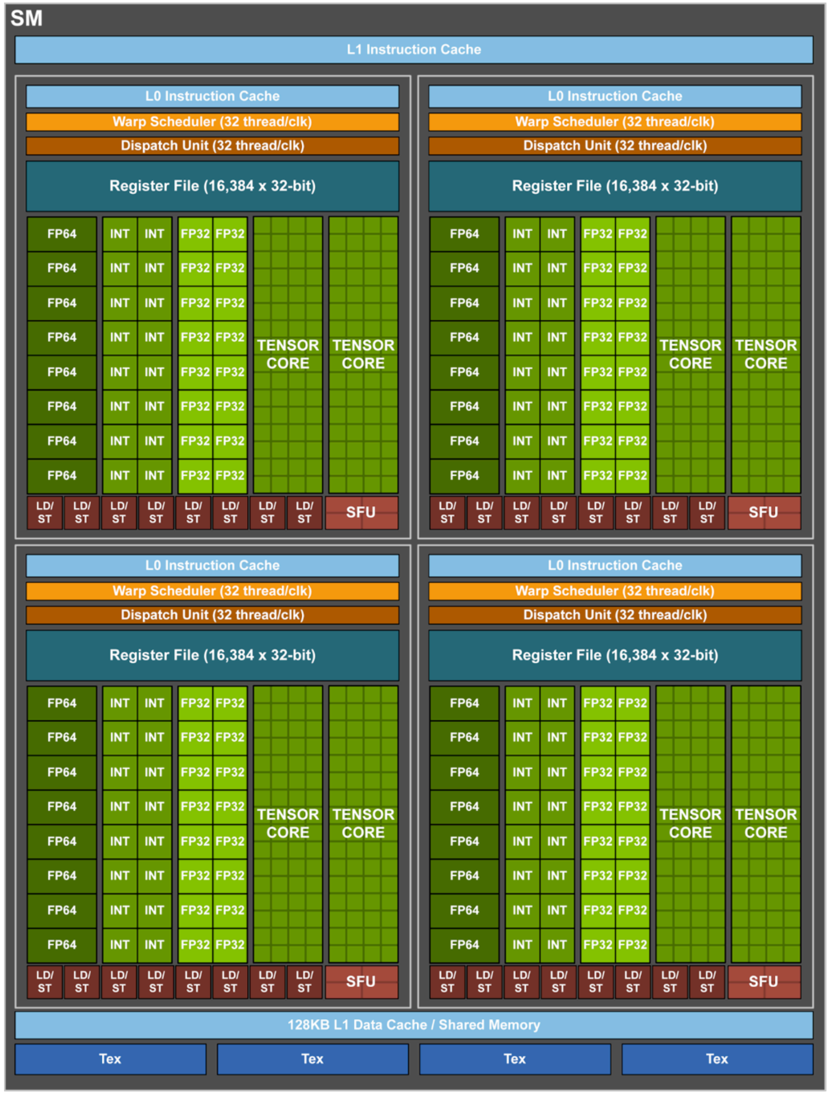
Instruction Cache

Instruction Buffer								Instruction Buffer							
Warp Scheduler								Warp Scheduler							
Dispatch Unit				Dispatch Unit				Dispatch Unit				Dispatch Unit			
Register File (32,768 x 32-bit)								Register File (32,768 x 32-bit)							
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU
Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU	Core	Core	DP Unit	Core	Core	DP Unit	LD/ST	SFU

Texture / L1 Cache

Tex | Tex | Tex | Tex

64KB Shared Memory



SM

L1 Instruction Cache

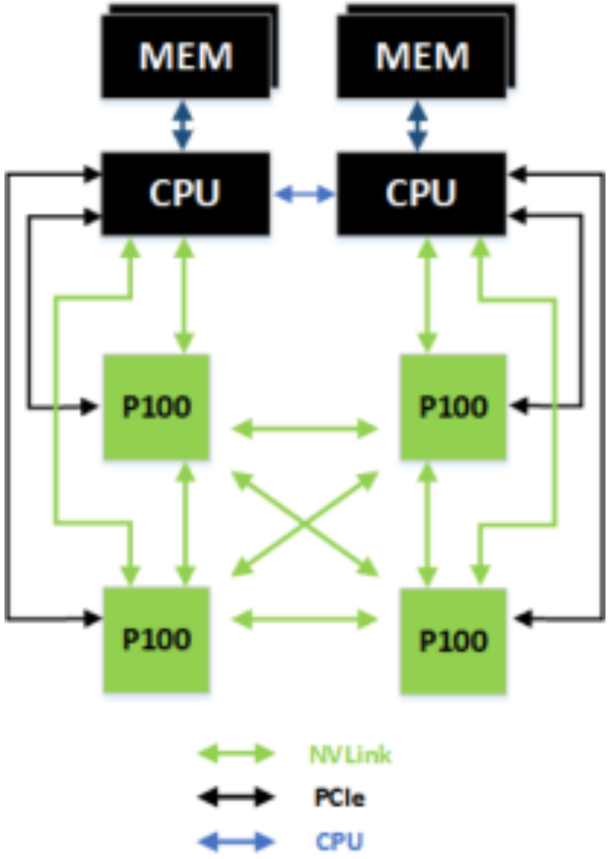
L0 Instruction Cache							L0 Instruction Cache							
Warp Scheduler (32 thread/clk)							Warp Scheduler (32 thread/clk)							
Dispatch Unit (32 thread/clk)							Dispatch Unit (32 thread/clk)							
Register File (16,384 x 32-bit)							Register File (16,384 x 32-bit)							
FP64	INT	INT	FP32	FP32	TENSOR CORE	TENSOR CORE	FP64	INT	INT	FP32	FP32	TENSOR CORE	TENSOR CORE	
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST						SFU	LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST						SFU	
L0 Instruction Cache							L0 Instruction Cache							
Warp Scheduler (32 thread/clk)							Warp Scheduler (32 thread/clk)							
Dispatch Unit (32 thread/clk)							Dispatch Unit (32 thread/clk)							
Register File (16,384 x 32-bit)							Register File (16,384 x 32-bit)							
FP64	INT	INT	FP32	FP32	TENSOR CORE	TENSOR CORE	FP64	INT	INT	FP32	FP32	TENSOR CORE	TENSOR CORE	
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
FP64	INT	INT	FP32	FP32			FP64	INT	INT	FP32	FP32			
LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST						SFU	LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST LD/ST						SFU	

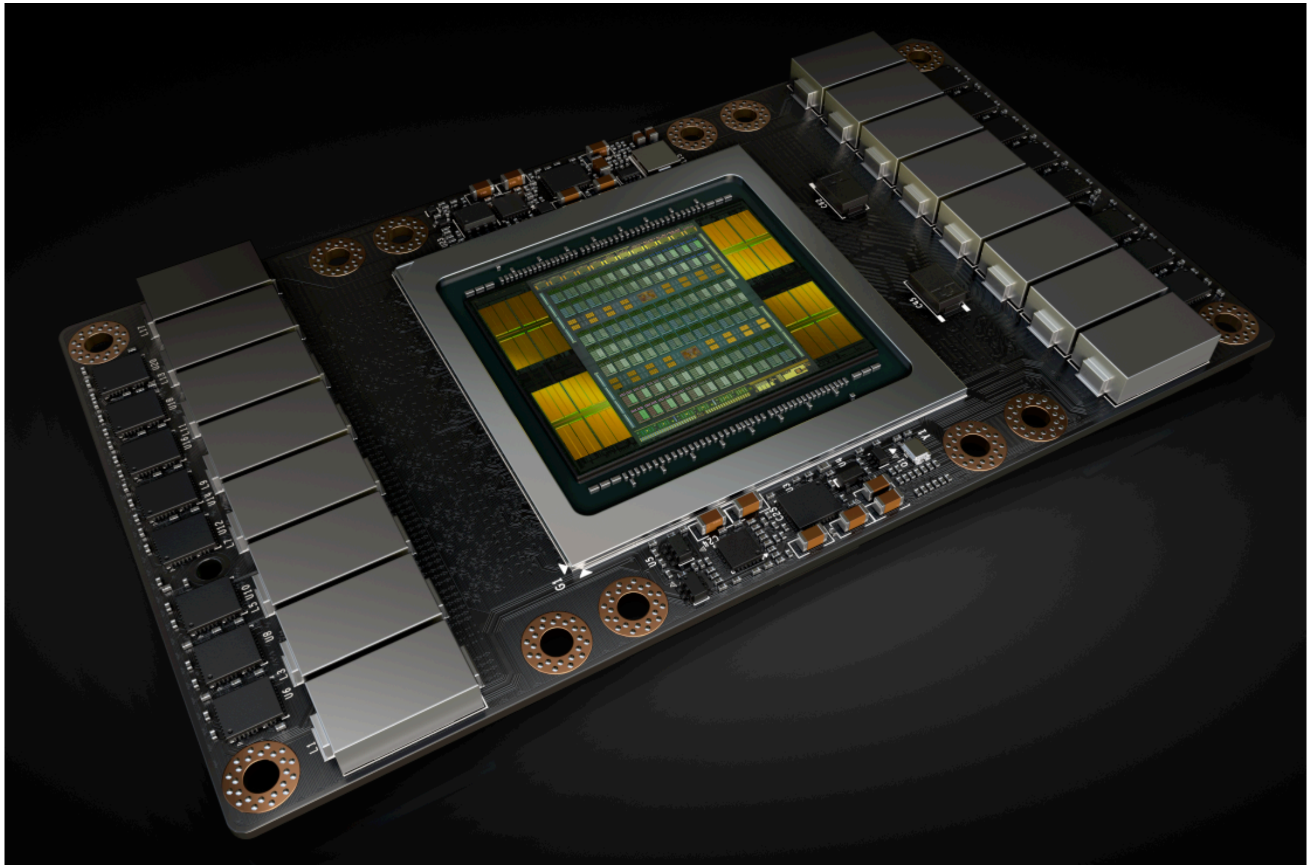
128KB L1 Data Cache / Shared Memory

Tex | Tex | Tex | Tex

Interconnect: Sockets and slots

- 2nd generation NVLink interconnect with 6 x 25 GB/s. links (vs. 4 x 20 GB/s. in Pascal).





Volta's comparison summary vs. Pascal

	GP100	GV100	Ratio
FP32 & FP64 peak performance	10 & 5 TFLOPS	15 & 7.5 TFLOPS	1.5x
DL training	10 TFLOPS	120 TFLOPS	12x
DL inferencing	21 TFLOPS	120 TFLOPS	6x
L1 caches (one per multiprocessor)	1.3 MB	10 MB	7.7x
L2 cache	4 MB	6 MB	1.5x
HBM2 bandwidth	720 GB/s	900 GB/s	1.2x
STREAM Triad performance (benchmark)	557 GB/s	855 GB/s	1.5x
NV-link bandwidth	160 GB/s	300 GB/s	1.8x

II. 8. A summary of four generations

Scalability for the architecture: A summary of four generations (2006-2015)

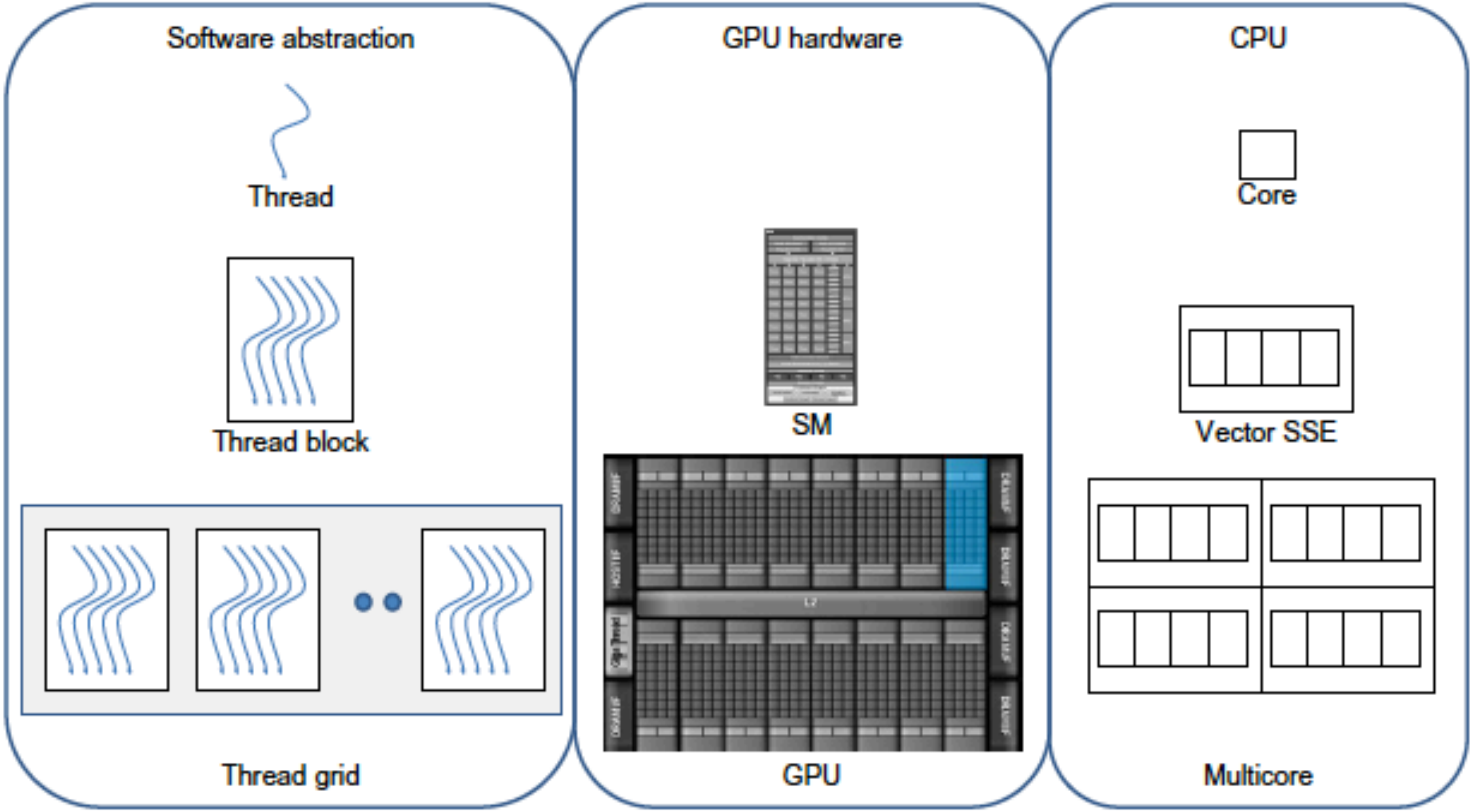
	Tesla		Fermi		Kepler				Maxwell	
Architecture	G80	GT200	GF100	GF104	GK104 (K10)	GK110 (K20X)	GK110 (K40)	GK210 (K80)	GM107 (GTX750)	GM204 (GTX980)
Time frame	2006 /07	2008 /09	2010	2011	2012	2013	2013 /14	2014	2014 /15	2014 /15
CUDA Compute Capability	1.0	1.3	2.0	2.1	3.0	3.5	3.5	3.7	5.0	5.2
N (multiprocs.)	16	30	16	7	8	14	15	30	5	16
M (cores/multip.)	8	8	32	48	192	192	192	192	128	128
Number of cores	128	240	512	336	1536	2688	2880	5760	640	2048

New models for 2016/17

	Maxwell				Pascal	
Architecture	GM107 (GTX750)	GM204 (GTX980)	GM200 (Titan X)	GM200 (Tesla M40)	GP104 (GeForce GTX 1080)	GP100 (Tesla P100)
Time Frame	2014 /15	2014 /15	2016	2016	2016	2017
CUDA Compute Capability	5.0	5.2	5.3	5.3	6.0	6.0
N (multiprocs.)	5	16	24	24	40	56
M (cores/multip.)	128	128	128	128	64	64
Number of cores	640	2048	3072	3072	2560	3584

III. Programming

Comparing the GPU and the CPU



From POSIX threads in CPU to CUDA threads in GPU

POSIX-threads in CPU

```
#define NUM_THREADS 16
void *myfun (void *threadId)
{
  int tid = (int) threadId;
  float result = sin(tid) * tan(tid);
  pthread_exit(NULL);
}

void main()
{
  int t;
  for (t=0; t<NUM_THREADS; t++)
  pthread_create(NULL,NULL,myfun,t);
  pthread_exit(NULL);
}
```

CUDA in GPU, followed by host code in CPU

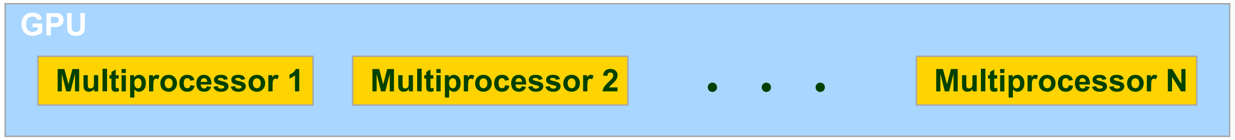
```
#define NUM_BLOCKS 1
#define BLOCKSIZE 16
__global__ void mykernel()
{
 int tid = threadIdx.x;
 float result = sin(tid) * tan(tid);
}

void main()
{
 dim3 dimGrid (NUM_BLOCKS);
 dim3 dimBlock (BLOCKSIZE);
 mykernel<<<dimGrid, dimBlock>>>();
 return EXIT_SUCCESS;
}
```

2D configuration: Grid of 2x2 blocks, 4 threads each

```
#define NUM_BLX 2
#define NUM_BLY 2
#define BLOCKSIZE 4
__global__ void mykernel()
{
  int bid=blockIdx.x*gridDim.y+blockIdx.y;
  int tid=bid*blockDim.x+ threadIdx.x;
  float result = sin(tid) * tan(tid);
}

void main()
{
  dim3 dimGrid (NUM_BLX, NUM_BLY);
  dim3 dimBlock(BLOCKSIZE);
  mykernel<<<dimGrid, dimBlock>>>();
  return EXIT_SUCCESS;
}
```

The CUDA programming model

- The GPU (device) is a highly multithreaded coprocessor to the CPU (host):
  - Has its own DRAM (device memory).
  - Executes many threads in parallel on several multiprocessor cores.



- CUDA threads are **extremely lightweight**.
  - Very little creation overhead.
  - Context switching is essentially free.
- Programmer’s goal: Declare thousands of threads to ensure the full utilization of hardware resources.

Structure of a CUDA program

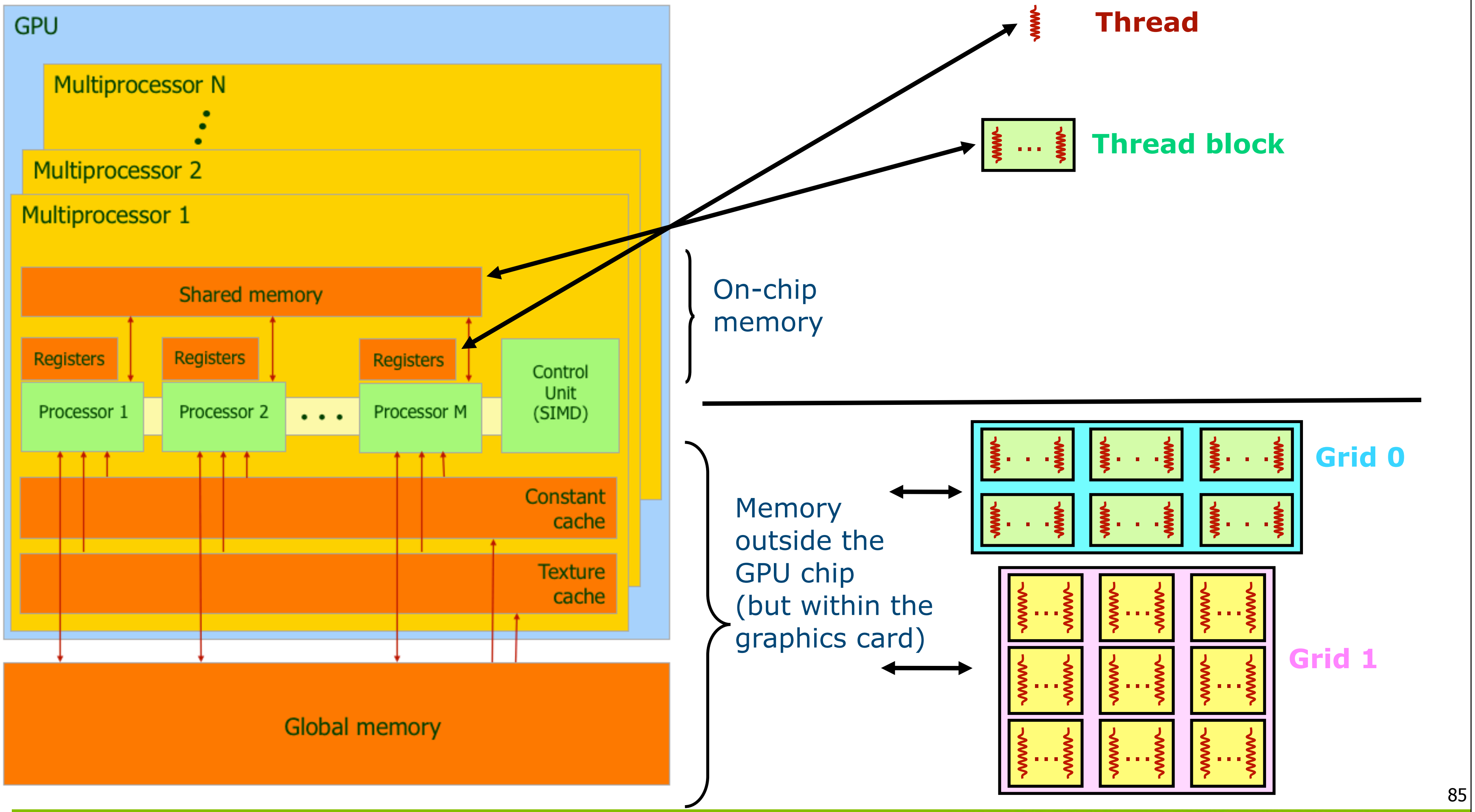
- Each multiprocessor (SM) processes batches of blocks one after another.
  - Active blocks = blocks processed by one multiprocessor in one batch.
  - Active threads = all the threads from the active blocks.
- Registers and shared memory within a multiprocessor are split among the active threads. Therefore, for any given kernel, the number of active blocks depends on:
  - The number of registers that the kernel requires.
  - How much shared memory the kernel consumes.

Preliminary definitions

Programmers face the challenge of exposing parallelism for thousands cores using the following elements:

- Device = GPU = Set of multiprocessors.
- Multiprocessor = Set of processors + shared memory.
- Kernel = Program ready to run on GPU.
- Grid = Array of thread blocks that execute a kernel.
- Thread block = Group of SIMD threads that:
  - Execute a kernel on different data based on threadID and blockID.
  - Can communicate via shared memory.
- Warp size = 32. This is the granularity of the scheduler for issuing threads to the execution units.

The relation between hardware and software from a memory access perspective



Resources and limitations depending on CUDA hardware generation (CCC)

	CUDA Compute Capability (CCC)						Limitation	Impact
	1.0, 1.1	1.2, 1.3	2.0, 2.1	3.0, 3.5, 3.7	5.0, 5.2, 5.3	6.0		
Multiprocessors / GPU	16	30	14-16	13-16	4, 5, ...	40-56	Hardware	Scalability
Cores / Multiprocessor	8	8	32	192	128	64		
Threads / Warp	32	32	32	32	32	32	Software	Throughput
Blocks / Multiprocessor	8	8	8	16	32	32		
Threads / Block	512	512	1024	1024	1024	1024	Software	Parallelism
Threads / Multiprocessor	768	1024	1536	2048	2048	2048		
32-bits regs./ Multip.	8K	16K	32K	64K	64K	64K	Hardware	Working set
Shared memory / Multip.	16K	16K	16KB 48KB	16KB, 32K, 48K	64K (5.0) 96K (5.2)	64KB.		

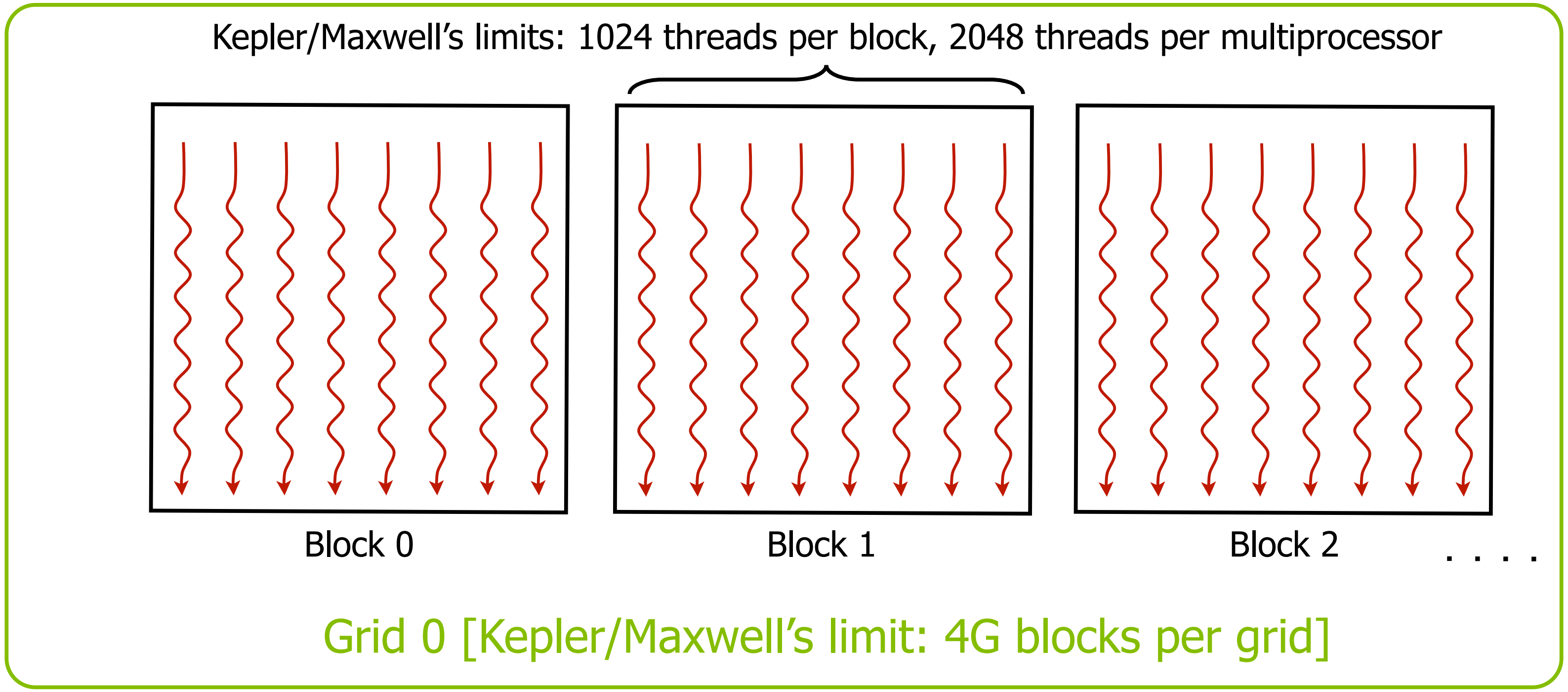
The CCC relation with the GPU marketplace

CCC	Code names	Models aimed to CUDA	Commercial series	Year range	Manufacturing process @ TSMC
1.0	G80	Many	8xxx	2006-07	90 nm.
1.1	G84,6 G92,4,6,8	Many	8xxx/9xxx	2007-09	80, 65, 55 nm.
1.2	GT215,6,8	Few	2xx	2009-10	40 nm.
1.3	GT200	Many	2xx	2008-09	65, 55 nm.
2.0	GF100, GF110	Huge	4xx/5xx	2010-11	40 nm.
2.1	GF104,6,8, GF114,6,8,9	Few	4xx/5xx/7xx	2010-13	40 nm.
3.0	GK104,6,7	Some	6xx/7xx	2012-14	28 nm.
3.5	GK110, GK208	Huge	6xx/7xx/Titan	2013-14	28 nm.
3.7	GK210 (2xGK110)	Very few	Titan	2014	28 nm.
5.0	GM107,8	Many	7xx	2014-15	28 nm.
5.2	GM200,4,6	Many	9xx/Titan	2014-15	28 nm.
6.0	GP104, GP100	Todos	10xx/P100	2016-17	16 nm. finFET

Guidelines to identify Nvidia commercial series

- 200: Developed during 3Q'08, until 4Q'09. Upgrades the G80 with 240 cores (GTX260 and GTX280).
- 400: Starts in 1Q'10. Introduces Fermi. Until 3Q'11.
- 500: Starts in 4Q'10 with GTX580 [GF110], and concludes the Fermi generation in 1Q'12.
- 600: 2012-13. Introduces Kepler, but also includes Fermis.
- 700: 2013-14. Focuses on Kepler, but brings the last Fermi models [GF108] and the first Maxwells [GM107, GM108].
- 800M: 1Q'14 and only for laptops, combining Fermi [GF117], Kepler [GK104] and Maxwell [GM107, GM108].
- 900: Starts in 4Q'14, with a Maxwell upgrade [GM20x].
- 1000: Starts in 2Q'16, with the first Pascal models [GP10x].

GPU threads and blocks



Kepler/Maxwell's limits: 1024 threads per block, 2048 threads per multiprocessor
Block 0
Block 1
Block 2
. . . .
Grid 0 [Kepler/Maxwell's limit: 4G blocks per grid]

Blocks are assigned to multiprocessors →

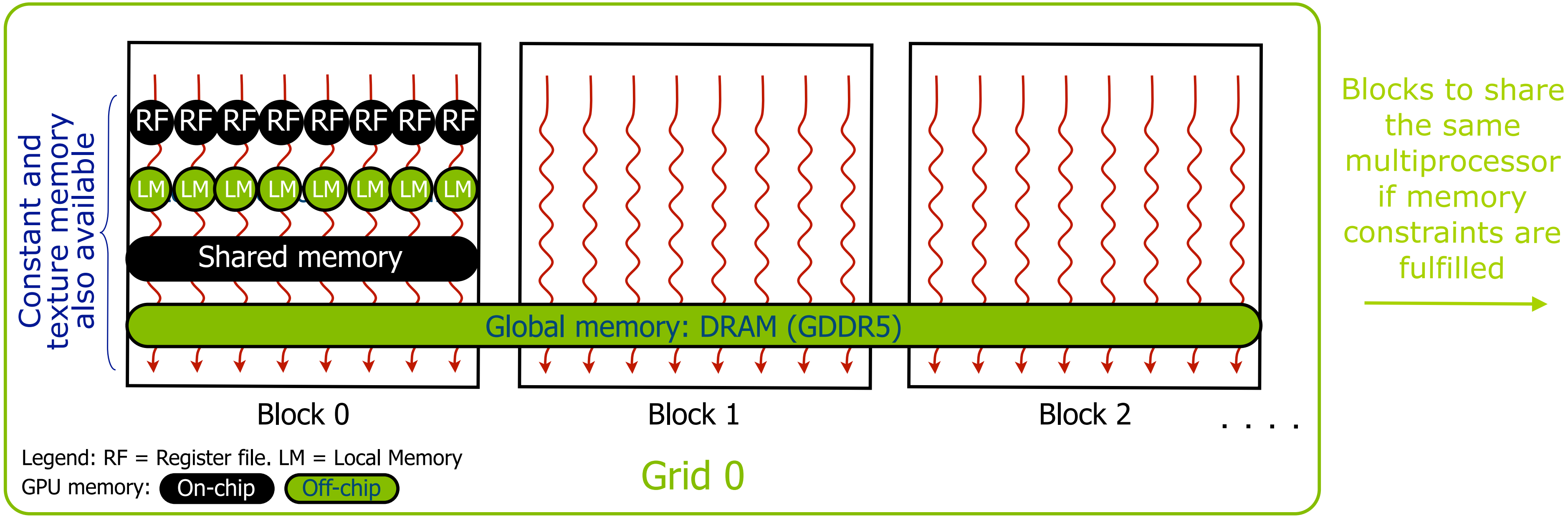
[Limit: 16-32 concurrent blocks per multiprocessor]

- Threads are assigned to multiprocessors in blocks, and to cores via warps, which is the scheduling unit (32 threads).
- Threads of a block share information via shared memory, and can synchronize via `syncthreads()` calls.

Playing with parallel constrainsts in Maxwell to maximize concurrency

- Limits within a SMM multiprocessor: [1] 32 concurrent blocks, [2] 1024 threads/block and [3] 2048 threads total.
- 1 block of 2048 threads. Forbidden by [2].
- 2 blocks of 1024 threads. Feasible on the same multiproc.
- 4 blocks of 512 threads. Feasible on the same multiproc.
- 4 blocks of 1024 threads. Forbidden by [3] on the same multiprocessor, feasible involving two multiprocessors.
- 8 blocks of 256 threads. Feasible on the same multiproc.
- 256 blocks of 8 threads. Forbidden by [1] on the same multiprocessor, feasible involving 8 multiprocessors.

GPU memory: Scope and location


Constant and texture memory also available
RF
LM
Shared memory
Global memory: DRAM (GDDR5)
Block 0
Block 1
Block 2
. . . .
Legend: RF = Register file. LM = Local Memory
GPU memory: On-chip Off-chip
Grid 0
Blocks to share the same multiprocessor if memory constraints are fulfilled

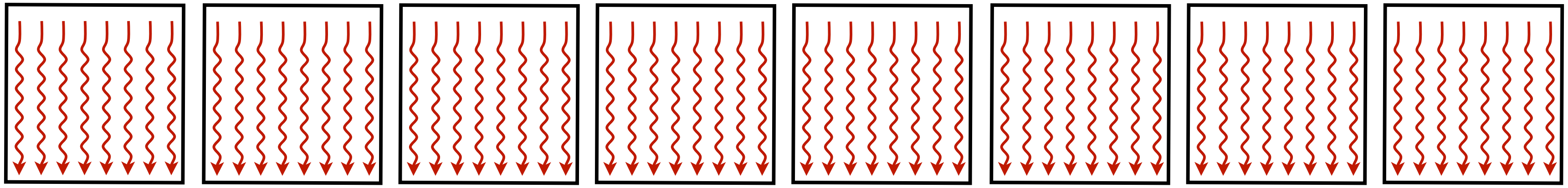
- Threads within a block can use the shared memory to perform tasks in a more cooperative and faster manner.
- Global memory is the only visible to threads, blocks and kernels.

Playing with memory constraints in Maxwell (CCC 5.2) to maximize the use of resources

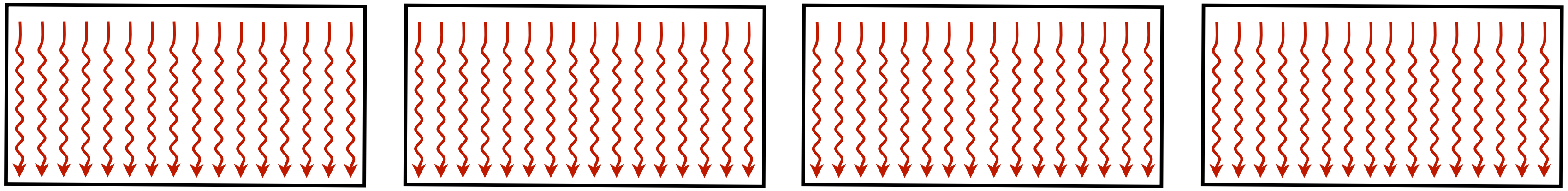
- Limits within a SMM multiprocessor:
  - 64 Kregisters.
  - 96 Kbytes of shared memory.
- That way:
  - To allow a second block to execute on the same multiprocessor, each block must use at most 32 Kregs. and 48 KB of shared memory.
  - To allow a third block to execute on the same multiprocessor, each block must use at most 21.3 Kregs. and 32 KB. of shared mem.
- ... and so on. In general, the less memory used, the more concurrency for blocks execution.
- There is a trade-off between memory and parallelism!

Think small: 1D partitioning on a 64 elements vector

- Remember: Use finest grained parallelism (assign one data to each thread). Threads and blocks deployment:
  - 8 blocks of 8 threads each. Risk on smaller blocks: Waste parallelism if the limit of 16-32 blocks per multip. is reached.



  - 4 blocks of 16 threads each. Risk on larger blocks: Squeeze the working set for each thread (remember that shared memory and register file are shared by all threads).

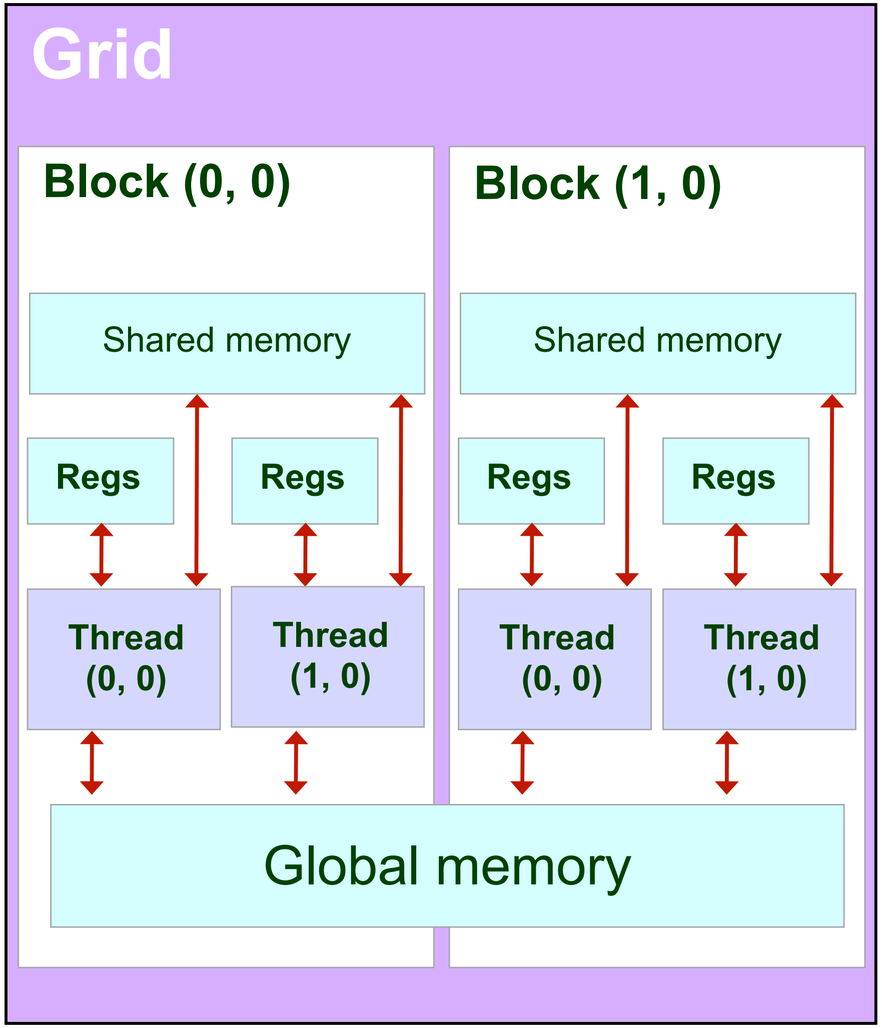


Now think big:
1D partitioning on a 64 **million** elems. array

- Maximum number of threads per block: 1024.
- Maximum number of blocks:
  - 64K on Fermi.
  - 4G on Kepler/Maxwell.
- Larger sizes for data structures can only be covered with a huge number of blocks (keeping fine-grained parallelism).
- Choices:
  - 64K blocks of 1K threads each (maximum for Fermi).
  - 128K blocks of 512 threads each (no longer available on Fermi).
  - 256K blocks of 256 threads each (no longer available on Fermi).
  - ... and so on.

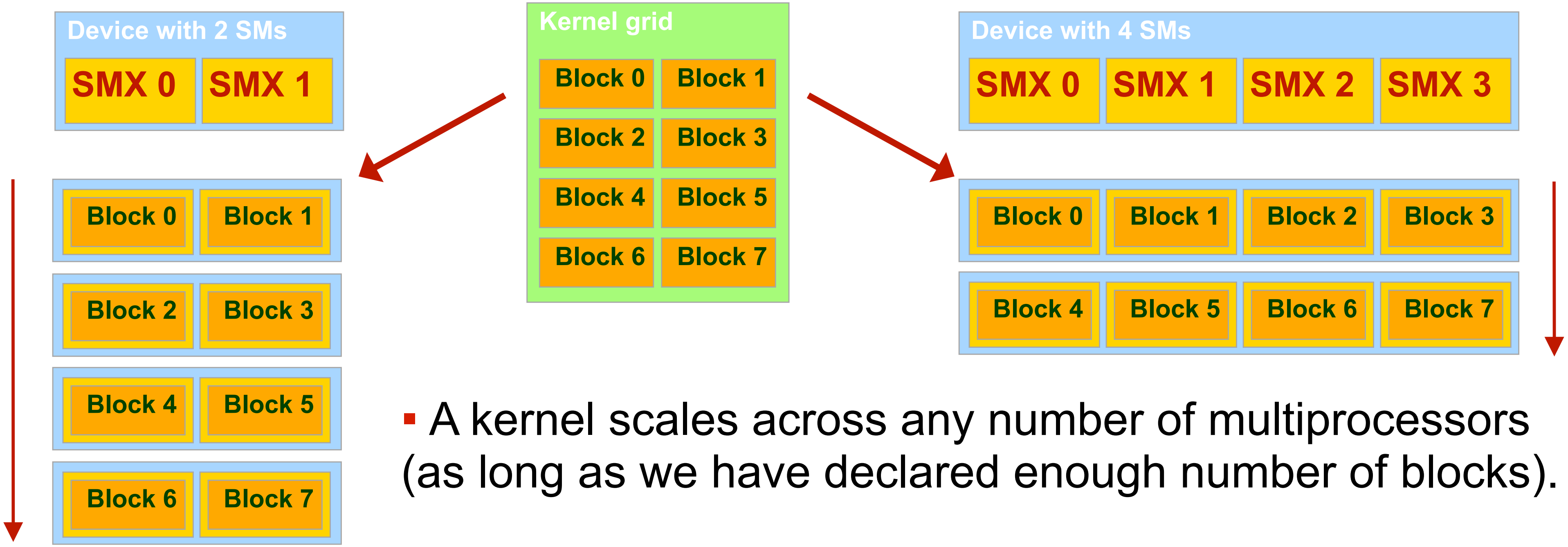
Summarizing about kernels, blocks, threads and parallelism

- Kernels are launched in grids.
- Each block executes fully on a single multiprocessor (SMX/SMM).
  - Does not migrate.
- Several blocks can reside concurrently on one SMX/SMM.
  - With control limitations. For example, in Kepler/Maxwell, we have:
    - Up to **16/32** concurrent blocks.
    - Up to **1024** threads per block.
    - Up to **2048** threads per SMX/SMM.
  - But usually, tighter limitations arise due to shared use of the register file and shared memory among all threads (as we have seen 3 slides ago).



Transparent scalability

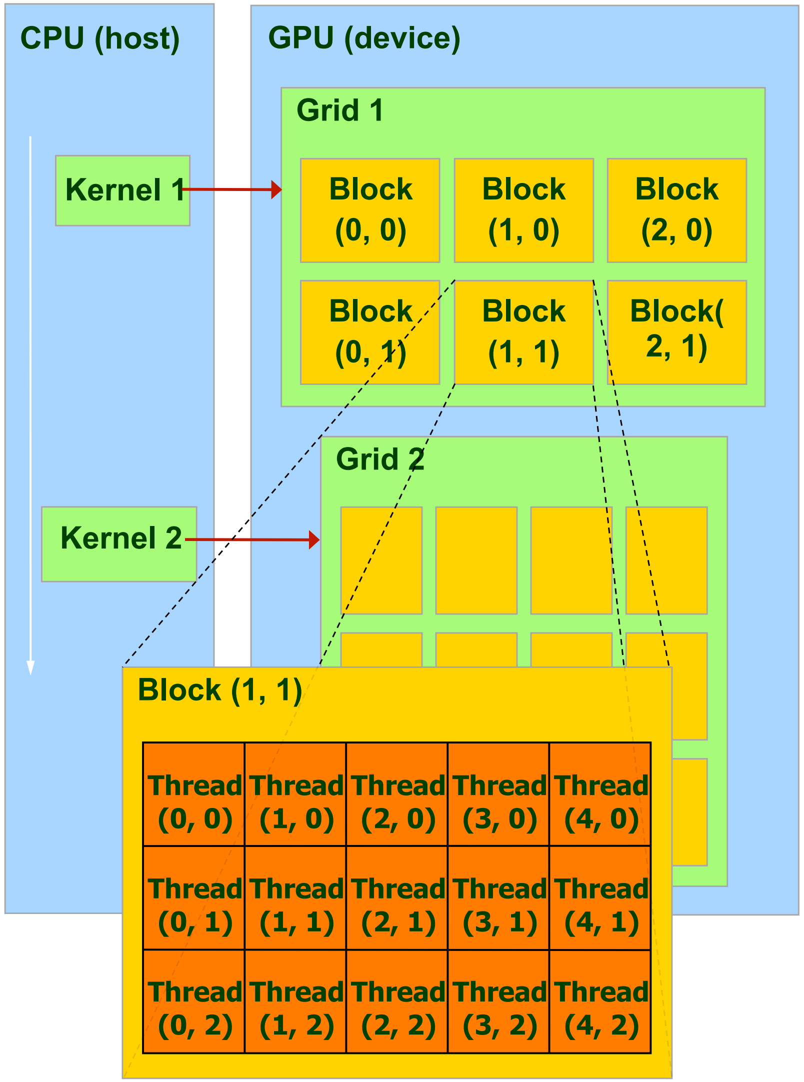
- Since blocks cannot synchronize:
  - The hardware is free to schedule the execution of a block on any multiprocessor.
  - Blocks may run sequentially or concurrently depending on resources usage.



- A kernel scales across any number of multiprocessors (as long as we have declared enough number of blocks).

Partitioning data and computations

- A block is a batch of threads which can cooperate by:
  - Sharing data via shared memory.
  - Synchronizing their execution for hazard-free memory accesses.
- A kernel is executed as a 1D or 2D grid of 1D, 2D or 3D of thread blocks.
- Multidimensional IDs are very convenient when addressing multidimensional arrays, for each thread has to bound its area/ volume of local computation.



CPU (host)

GPU (device)

Kernel 1

Grid 1

Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block(2, 1)

Kernel 2

Grid 2

Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

Memory spaces

- The CPU and the GPU have separated memory spaces:
  - To communicate them, we use the PCI express bus.
  - The GPU uses specific functions to allocate memory and copy data from CPU in a similar manner to what we are used with the C language (`malloc/free`).
- Pointers are only addresses:
  - You cannot derive from a pointer value if the address belongs to either the CPU or the GPU space.
  - You have to be very careful when handling pointers, as the program usually crashes when a CPU data attemps to be accessed from GPU and vice versa **(with the introduction of unified memory, this situation changes from CUDA 6.0 on)**.

IV. Syntax

IV. 1. Basic elements

CUDA is C with some extra keywords. A preliminar example

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke the SAXPY function sequentially
saxpy_serial(n, 2.0, x, y);
```

C code on the CPU

Equivalent CUDA code for its parallel execution on GPUs:

```
__global__ void saxpy_parallel(int n, float a, float *x,
float *y)
{   // More on parallel access patterns later in example 2
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke SAXPY in parallel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

List of extensions added to the C language

- Type qualifiers:
  - global, device, shared, local, constant.
- Keywords:
  - threadIdx, blockIdx, gridDim, blockDim.
- Intrinsics:
  - __syncthreads();
- Runtime API:
  - Memory, symbols, execution management.
- Kernel functions to launch code to the GPU from the CPU.

```
__device__ float array[N];

__global__ void med_filter(float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx.x] = image[i];


__syncthreads();
  ...
  image[j] = result;
}

// Allocate memory in the GPU
void *myimage;
cudaMalloc(&myimage, bytes);


// 100 thread blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Interaction between CPU and GPU

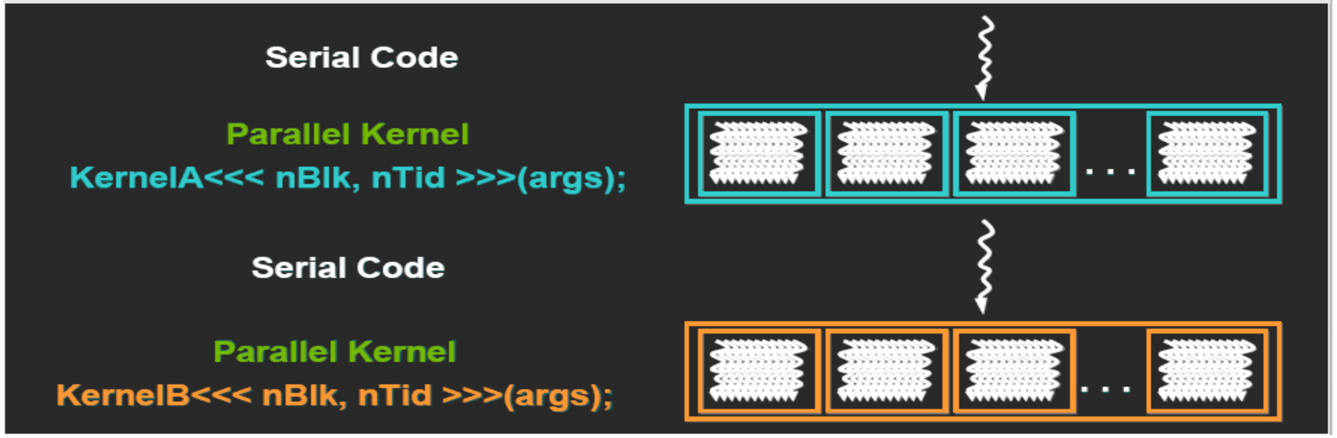
- CUDA extends the C language with a new type of function, kernel, which executes code in parallel on all active threads within GPU. Remaining code is native C executed on CPU.
- The typical `main()` of C combines the sequential execution on CPU and the parallel execution on GPU of CUDA kernels.
- A kernel is launched in an asynchronous way, that is, control always returns immediately to the CPU.
- Each GPU kernel has an implicit barrier when it ends, that is, it does not conclude until all its threads are over.
- We can exploit the CPU-GPU biprocessor by interleaving code with a similar workload on both.

Interaction between CPU and GPU (cont.)

Execution

```
__global__  kernelA(){···}
__global__  kernelB(){···}
int main()
•••
kernelA <<< dimGridA, dimBlockA >>> (params.);
•••
kernelB <<< dimGridB, dimBlockB >>> (params.);
•••
```

CPU
GPU
CPU
GPU
CPU



Serial Code
Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);
Serial Code
Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);

- A kernel does not start until all previous kernels are over.
- **Streams** allow you to run kernels in parallel.

Modifiers for the functions and launching executions on GPU

- Modifiers for the functions executed on GPU:

```
__global__ void MyKernel() { } // Invoked by the CPU
__device__ float MyFunc() { }  // Invoked by the GPU
```

- Modifiers for the variables within GPU:

```
__shared__ float MySharedArray[32];  // In shared mem.
__constant__ float MyConstantArray[32];
```

- Configuration for the execution to launch kernels:

```
dim2 gridDim(100,50);    // 5000 thread blocks
dim3 blockDim(4,8,8);    //  256 threads per blocks
MyKernel <<< gridDim,blockDim >>> (pars.); // Launch
```

  - Note: We can see an optional third parameter here to indicate as a hint the amount of shared memory allocated dynamically by the kernel during its execution.

Intrinsics

```
dim3 gridDim;  // Grid dimension: Number of blocks on each dim.
dim3 blockDim; // Block dimension: Block size on each dim.

uint3 blockIdx; // Index to the block within the mesh
uint3 threadIdx; // Index to the thread in the block

void __syncthreads(); // Explicit synchronization
```

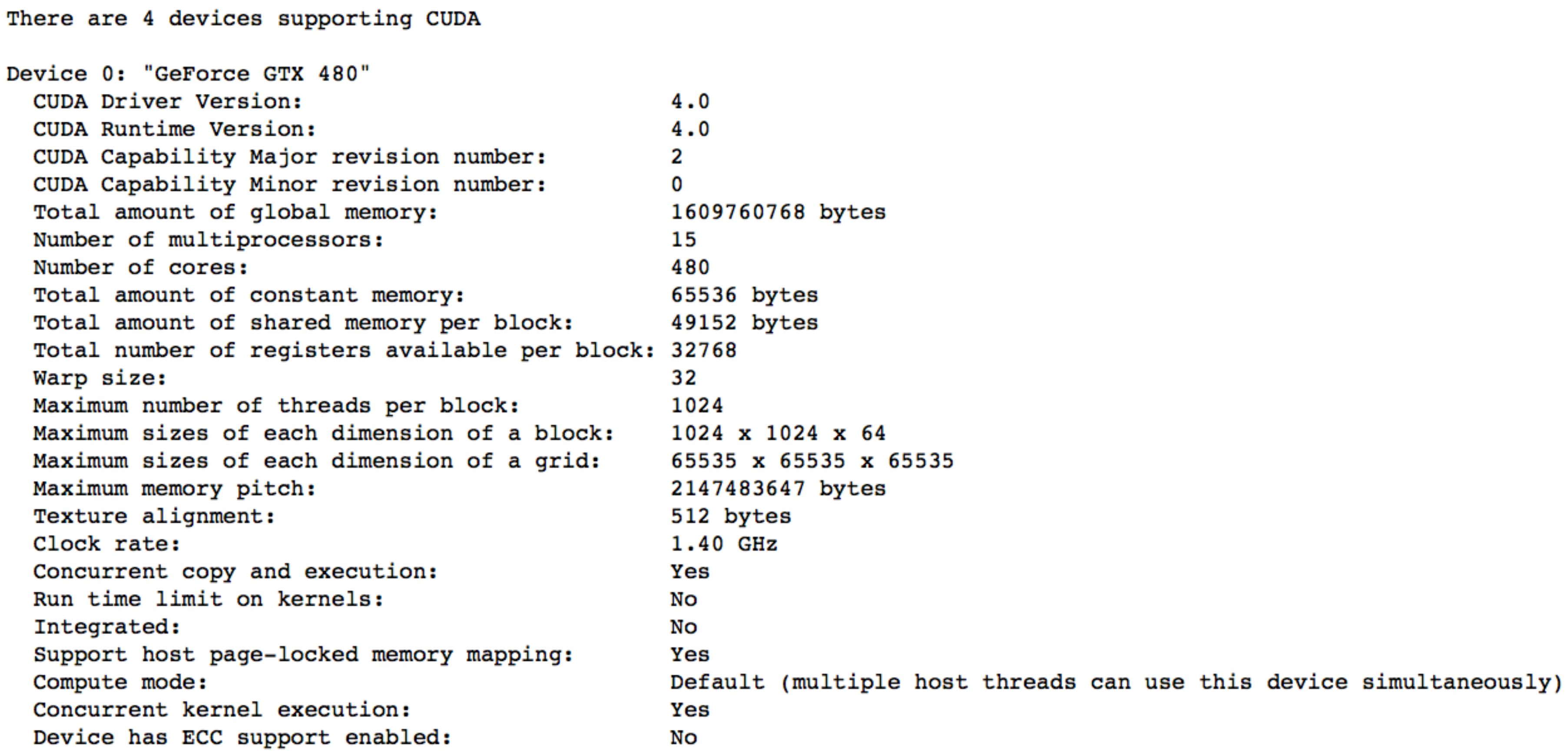
- Programmer has to choose the block size and the number of blocks to exploit the maximum amount of parallelism for the code during its execution.

Functions to query at runtime the hardware resources we count on

- Each GPU available at hardware level receives an integer tag which identifies it, starting in 0.
- To know the number of GPUs available:
  - `cudaGetDeviceCount(int* count);`
- To know the resources available on GPU dev (cache, registers, clock frequency, ...):
  - `cudaGetDeviceProperties(struct cudaDeviceProp* prop, int dev);`
- To know the GPU that better meets certain requirements:
  - `cudaChooseDevice(int* dev, const struct cudaDeviceProp* prop);`
- To select a particular GPU:
  - `cudaSetDevice(int dev);`
- To know in which GPU we are executing the code:
  - `cudaGetDevice(int* dev);`

The output of cudaGetDeviceProperties

- This is exactly the output you get from the “DeviceQuery” code in the CUDA SDK.



```
There are 4 devices supporting CUDA

Device 0: "GeForce GTX 480"
  CUDA Driver Version:                           4.0
  CUDA Runtime Version:                          4.0
  CUDA Capability Major revision number:         2
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 1609760768 bytes
  Number of multiprocessors:                     15
  Number of cores:                               480
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                No
```

Let's manage video memory

- To allocate and free GPU memory:
  - `cudaMalloc(pointer, size)`
  - `cudaFree(pointer)`
- To move memory areas between CPU and GPU:
  - On the CPU side, we declare `malloc(h_A).`
  - Also on the GPU side, we declare `cudaMalloc(d_A).`
  - And once this is done, we can:
    - Transfer data from the CPU to the GPU:
      - `cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);`
    - Transfer data from the GPU to the CPU:
      - `cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);`
  - Prefix **"h_"** useful in practice as a tag for "host memory pointer".
  - Prefix **"d_"** also useful as a tag for "device (video) memory".

IV. 2. A couple of examples

Example 1: What your code has to do

- Allocate N integers in CPU memory.
- Allocate N integers in GPU memory.
- Initialize GPU memory to zero.
- Copy values from GPU to CPU.
- Print values.

Example 1: Solution [C code in red, CUDA extensions in blue]

```
int main()
{
    int N = 16;
    int num_bytes = N*sizeof(int);
    int *d_a=0, *h_a=0;  // Pointers in device (GPU) and host (CPU)

    h_a = (int*) malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);

    if( 0==h_a || 0==d_a ) printf("I couldn’t allocate memory\n");

    cudaMemset( d_a, 0, num_bytes);
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);

    for (int i=0; i<N; i++) printf("%d ", h_a[i]);

    free(h_a);
    cudaFree(d_a);
}
```

Asynchronous memory transfers

- `cudaMemcpy()` calls are synchronous, that is:
  - They do not start until all previous CUDA calls have finalized.
  - The return to the CPU does not take place until we have performed the actual copy in memory.
- From CUDA Compute Capabilities 1.2 on, it is possible to use the `cudaMemcpyAsync()` variant, which introduces the following differences:
  - The return to the CPU is immediate.
  - We can overlap computation and communication.

Example 2: Increment a scalar value “b” to the N elements of an array

The C program.

This file is compiled with **gcc**

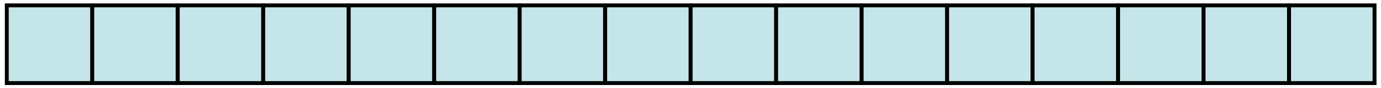
```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
                a[idx] = a[idx] + b;
}



void main()
{
    .....
        increment_cpu(a, b, N);
}
```

The CUDA kernel running on GPU followed by host code running on CPU.

This file is compiled with **nvcc**

```
__global__ void increment_gpu(float *a, float b, int N)
{
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < N)
                a[idx] = a[idx] + b;
}


void main()
{
    .....
        dim3 dimBlock (blocksize);
        dim3 dimGrid (ceil(N/(float)blocksize));
        increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Example 2: Increment a scalar "b" to the N elements of a vector



Say N=16 and blockDim=4. Then we have 4 thread blocks, and each thread computes a single element of the vector. This is what we want: fine-grained parallelism for the GPU.

Language extensions

blockIdx.x = 0
blockDim.x = 4
threadIdx.x = 0,1,2,3
idx = 0,1,2,3
	blockIdx.x = 1
	blockDim.x = 4
	threadIdx.x = 0,1,2,3
	idx = 4,5,6,7
		blockIdx.x = 2
		blockDim.x = 4
		threadIdx.x = 0,1,2,3
		idx = 8,9,10,11
			blockIdx.x = 3
			blockDim.x = 4
			threadIdx.x = 0,1,2,3
			idx = 12,13,14,15

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

It will map from local index **`threadIdx.x`** to global index

Same access pattern for all threads

Warning: blockDim.x should be >= 32 (warp size), this is just an example

More details for the CPU code of example 2 [red for C, green for variables, blue for CUDA]

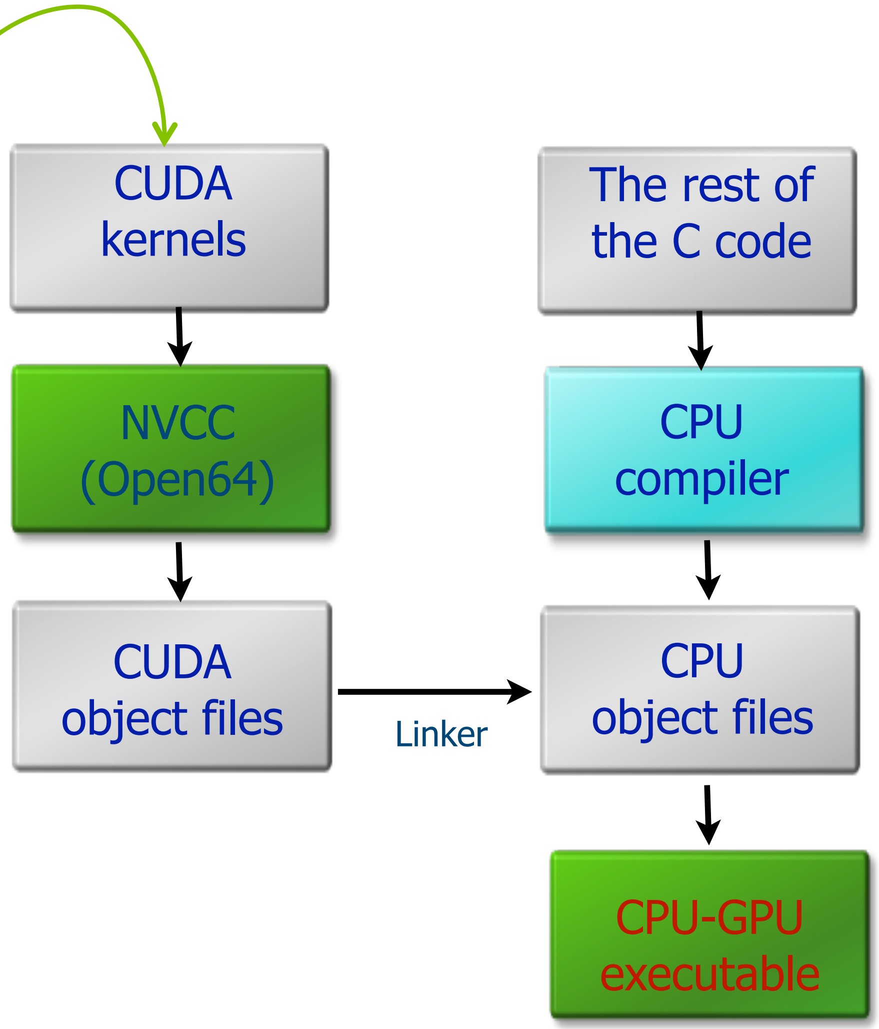
```
// Reserve memory on the CPU
unsigned int numBytes = N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// Reserve memory on the GPU
float* d_A = 0;  cudaMalloc(&d_A, numbytes);

// Copy data from CPU to GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// Execute CUDA kernel with a number of blocks and block size
increment_gpu <<< N/blockSize, blockSize >>> (d_A, b);

// Copy data back to the CPU
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// Free video memory
cudaFree(d_A);
```

V. Compilation

The global process

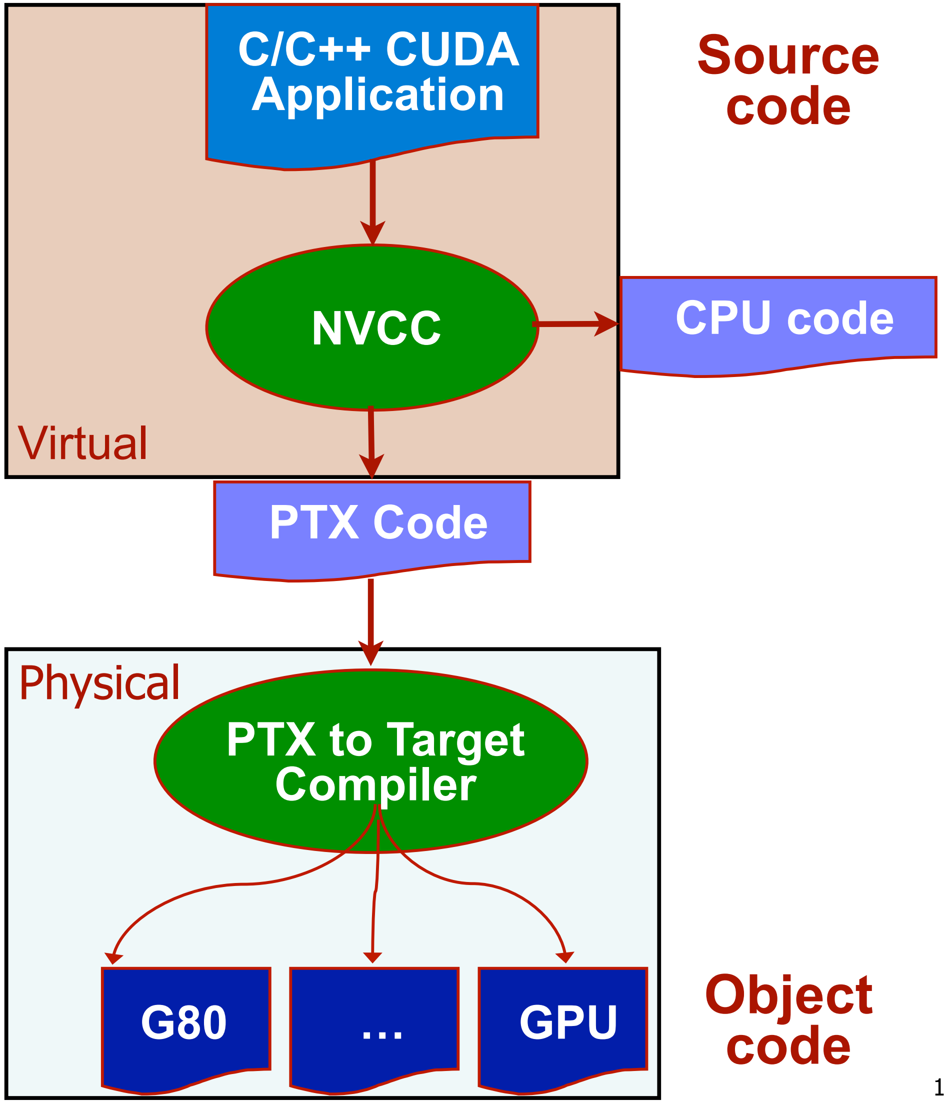
```
void function_in_CPU(… )
{
  ...
}
void other_funcs_CPU(int ...)
{
  ...
}

void saxpy_serial(float ... )
{
 for (int i = 0; i < n; ++i)
   y[i] = a*x[i] + y[i];
}

void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```

Identify CUDA kernels and rewrite them to exploit GPU parallelism

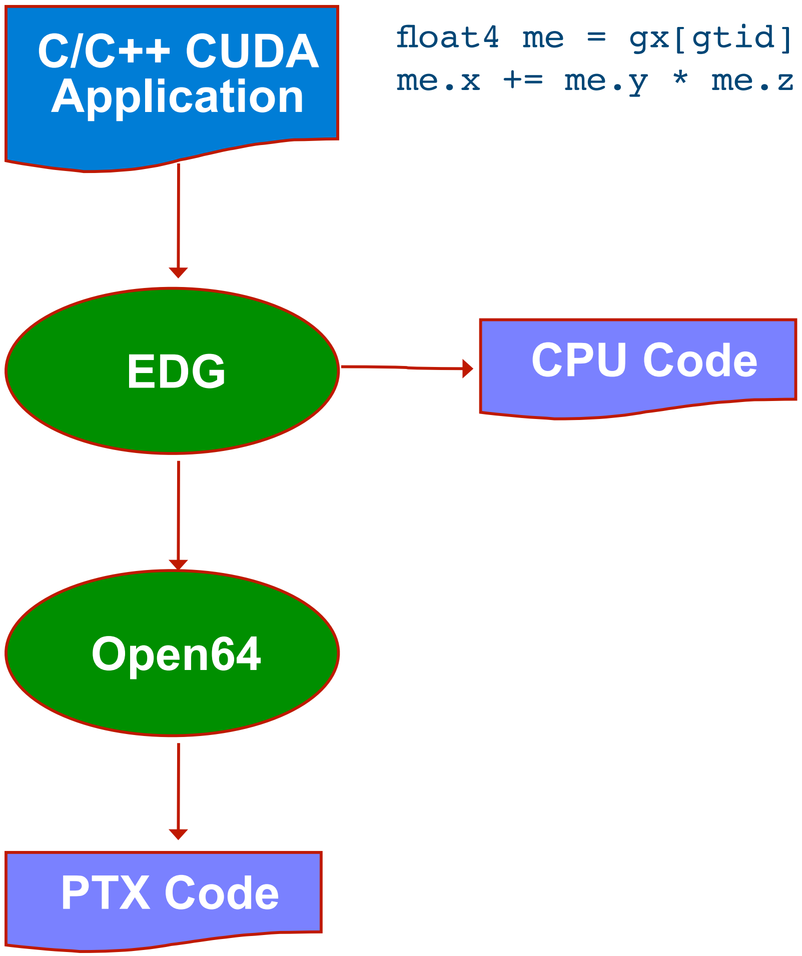


Compilation modules

- A CUDA code is compiled with the NVCC compiler.
    - NVCC separates CPU code and GPU code.
- The compilation is a two step process:
    - Virtual: Generates PTX (Parallel Thread eXecution).
    - Physical: Generates the binary for a specific GPU (or even a CPU - more on this later).



The `nvcc` compiler and PTX virtual machine



```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

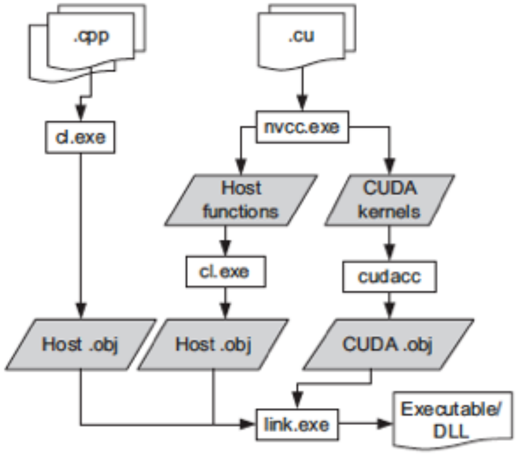
- EDG
  - Separates GPU and CPU code.
- Open64
  - Generates PTX assembler.
- Parallel Thread eXecution (PTX)
  - Virtual machine and ISA.
  - Programming model.
  - Resources and execution states.

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

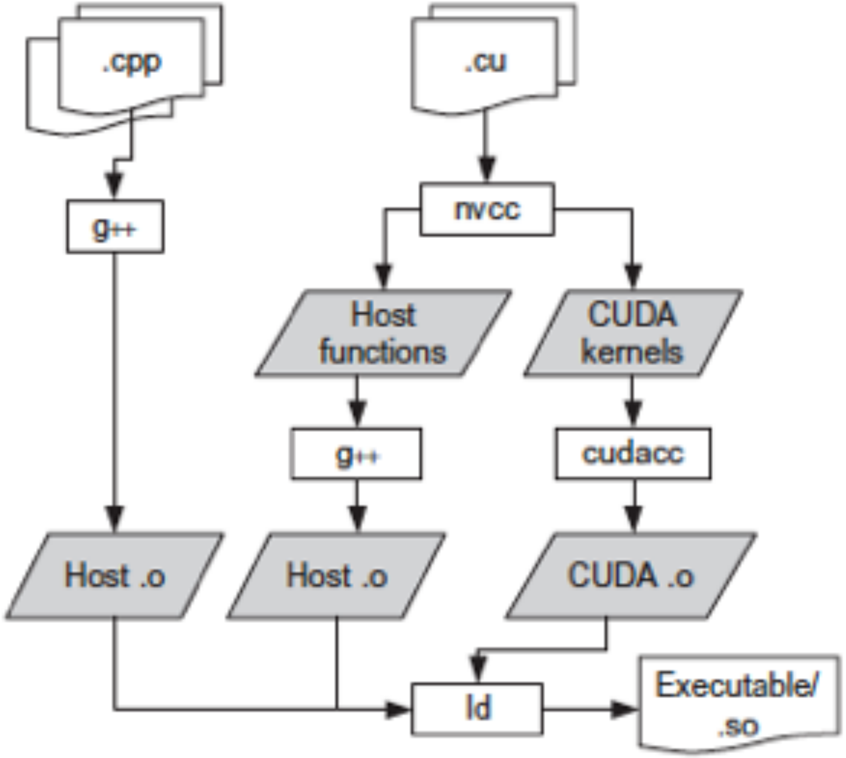
NVCC (NVidia CUDA Compiler)

- NVCC is a compiler driver.
  - Invokes all compilers and tools required, like cudacc, g++, cl, ...
- NVCC produces two outputs:
  - C code for the CPU, which must be compiled with the rest of the applic. using another compilation tool.
  - PTX object code for the GPU.

Compilation process in Windows:



Compilation process in Linux:



Determining resource usage

- Compile the kernel code with the -cubin flag to determine register usage.
  - On-line alternative: **`nvcc –ptxas-options=-v`**
- Open the `.cubin` file with a text editor and look for the “code” section:

```
architecture {sm_10}
abiversion {0}
modname {cubin}
code  {
      name = myGPUcode
      lmem = 0
      smem = 68
      reg = 20
      bar = 0
      bincode  {
            0xa0004205 0x04200780 0x40024c09 0x0020
            ...
```

Per thread:
local memory
(used by compiler to spill registers to device memory)

Per thread-block:
shared memory

Per thread:
registers

Configuration for the execution: Heuristics

- The number of threads must be a multiple of warp size.
  - To avoid wasting computation on incomplete warps.
- The number of blocks must exceed the number of SMXs (1), and, if possible, double that number (2):
  - (1) So that each multiprocessor can have at least a block to work with.
  - (2) So that there is at least an active block which guarantees occupancy of that SMX when the block being executed suffers from a stall due to a memory access, unavailability of resources, bank conflicts, global stalls of all threads on a synchronization point (`__syncthreads()`), etc.
- Resources used by a block (register file and shared memory) must be at least half of the total available.
  - Otherwise, it is better to merge blocks.

Heuristics (cont.)

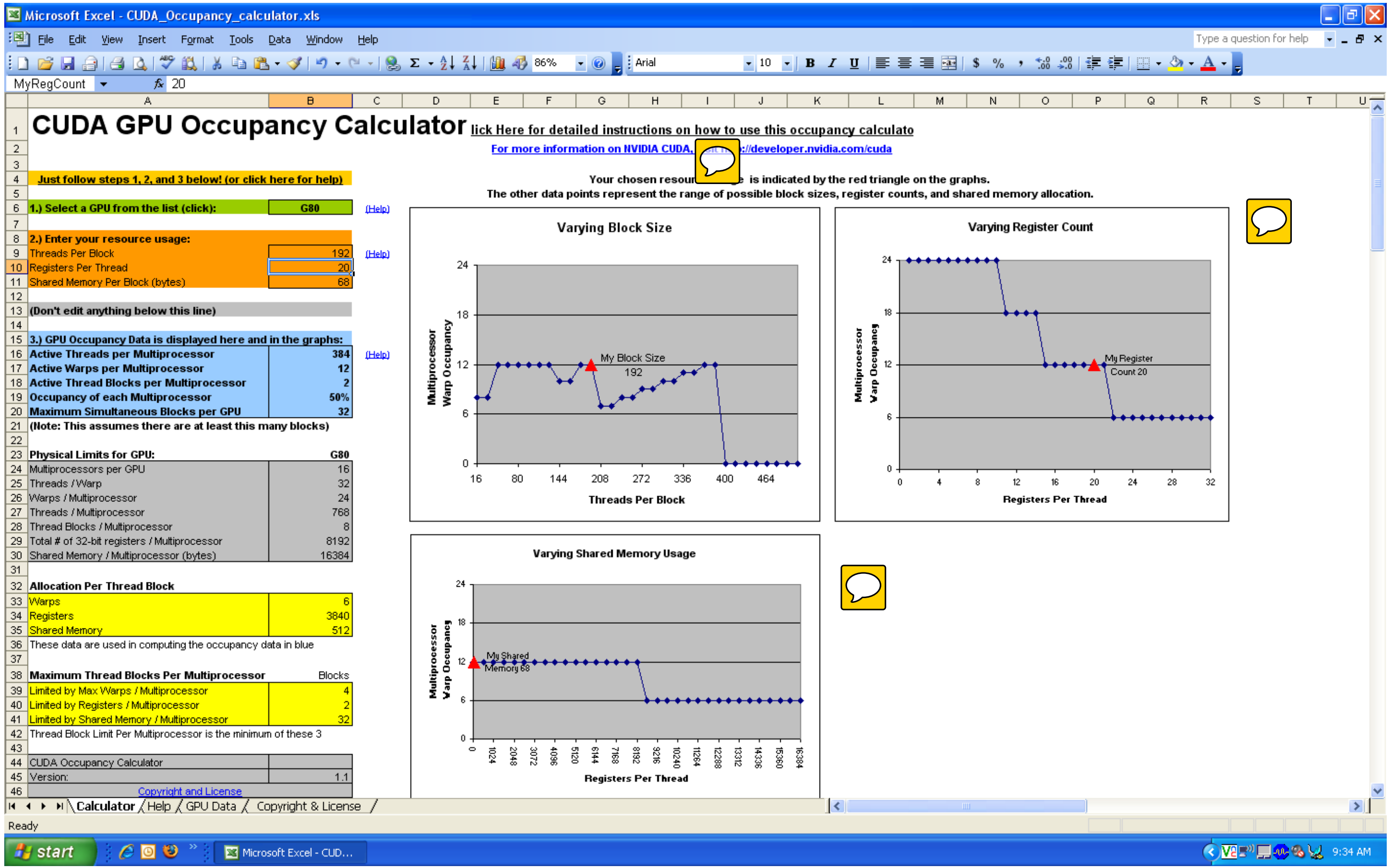
- General rules for the code to be scalable in future generations and for the blocks stream to be processed within a pipeline:
  - (1) Think big for the number of blocks.
  - (2) Think small for the size of threads.
- Tradeoff: More threads per block means better memory latency hiding, but also means fewer registers per thread.
- Hint: Use at least 64 threads per block, or even better, 128 or 256 threads (often there is still enough number of registers).
- Tradeoff: Increasing occupancy does not necessarily mean higher performance, but the low occupancy for a SMX prevents from hide latency on memory bound kernels.
- Hint: Pay attention to arithmetic intensity and parallelism.

Parametrization of an application

- Everything related to performance is application-dependent, so you have to experiment for achieving optimal results.
- GPUs may also vary in many ways depending on a particular model:
  - Number of multiprocessors (SMs) and cores per SM.
  - Memory bandwidth: From 100 GB/s to 500 GB/s.
  - Register file size per SM: 8K, 16K, 32K (Fermi), 64K (Kepler).
  - Shared memory size: 16 KB. per SM before Fermi, up to 48 KB. now.
  - Threads: Check the per-block and the global limits.
    - Per-block: 512 (G80 and GT200), 1024 (Fermi and Kepler).
    - Total: 768 (G80), 1024 (GT200), 1536 (Fermi), 2048 (Kepler).

CUDA Occupancy Calculator

- To help you select parameters for your application wisely
  - http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



Microsoft Excel - CUDA_Occupancy_calculator.xls
File Edit View Insert Format Tools Data Window Help
Type a question for help
86%
Arial
10
MyRegCount
start
Microsoft Excel - CUD...
9:34 AM
Ready

MyRegCount *fx* 20

	A	B	C
1	**CUDA GPU Occupancy Calculator**		
4	Just follow steps 1, 2, and 3 below! (or click here for help)		
6	1.) Select a GPU from the list (click):	G80	(Help)
8	2.) Enter your resource usage:		
9	Threads Per Block	192	(Help)
10	Registers Per Thread	20	
11	Shared Memory Per Block (bytes)	68	
13	(Don't edit anything below this line)		
15	3.) GPU Occupancy Data is displayed here and in the graphs:		
16	Active Threads per Multiprocessor	384	(Help)
17	Active Warps per Multiprocessor	12	
18	Active Thread Blocks per Multiprocessor	2	
19	Occupancy of each Multiprocessor	50%	
20	Maximum Simultaneous Blocks per GPU	32	
21	(Note: This assumes there are at least this many blocks)		
23	Physical Limits for GPU:	G80	
24	Multiprocessors per GPU	16	
25	Threads / Warp	32	
26	Warps / Multiprocessor	24	
27	Threads / Multiprocessor	768	
28	Thread Blocks / Multiprocessor	8	
29	Total # of 32-bit registers / Multiprocessor	8192	
30	Shared Memory / Multiprocessor (bytes)	16384	
32	Allocation Per Thread Block		
33	Warps	6	
34	Registers	3840	
35	Shared Memory	512	
36	These data are used in computing the occupancy data in blue		
38	Maximum Thread Blocks Per Multiprocessor	Blocks	
39	Limited by Max Warps / Multiprocessor	4	
40	Limited by Registers / Multiprocessor	2	
41	Limited by Shared Memory / Multiprocessor	32	
42	Thread Block Limit Per Multiprocessor is the minimum of these 3		
44	CUDA Occupancy Calculator		
45	Version:	1.1	
46	Copyright and License		

lick Here for detailed instructions on how to use this occupancy calculato

For more information on NVIDIA CUDA, ://developer.nvidia.com/cuda

Your chosen resour e is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

To reach the maximum degree of parallelism, use wisely the orange table of the tool (1)

- The first row is the number of threads per block:
  - The limit is 1024 in Fermi and Kepler generations.
  - Power of two values are usually the best choices.
  - List of potential candidates: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.
  - We'll use 256 as first estimate, development cycles will tune the optimal value here, but usually:
    - Small values [2, 4, 8, 16] do not fully exploit the warp size and shared memory banks.
    - Intermediate values [32, 64] compromise thread cooperation and scalability in Kepler, Maxwell and future GPUs.
    - Large values [512, 1024] prevent from having enough number of concurrent blocks on each multiprocessor (the limits for the threads per block and per SMX are very close to each other). Also, the amount of registers per thread is too small.

To reach the maximum degree of parallelism, use wisely the orange table of the tool (2)

- The second row is the number of registers per thread.
  - We access the .cubin file to know this.
  - The limit for each SM is 8K (G80), 16K (GT200), 32K (Fermi), 64K (Kepler), so when consuming 10 regs./thread is possible to execute:
    - On G80: 768 threads/SM, that is, 3 blocks of 256 thr [3*256*10=7680] (< 8192).
    - On Kepler: We reach the maximum of 2048 threads per SMX, but the use of registers is very low (we could have used up to 29 registers per thread):
      8 blocks * 256 threads/block * 10 registers/thread = 20480 regs. (< 65536 max.).
  - In the G80 case, using 11 registers/thread, it would have meant to stay in 2 blocks, sacrificing 1/3 of parallelism => It is worth cutting that register down working more on the CUDA code for the thread.
  - In Kepler, we may use up to 29 registers without compromising parallelism.

To reach the maximum degree of parallelism, use wisely the orange table of the tool (3)

- The third row is the shared memory spent for each block:
  - We will also get this from the .cubin file, though we can carry out a manual accounting, as everything depends on where we put the `__shared__` prefix during memory declarations in our program.
  - Limit: 16 KB (CCC 1.x), 16/48 KB (CCC 2.x), 16/32/48 KB (3.x).
  - In the previous case for the G80, we won't spend more than 5 KB of shared memory per block, so that we can reach the maximum of 3 concurrent blocks on each multiprocessor:
    - 3 blocks x 5 KB./block = 15 KB (< 16 KB.)
  - With more than 5.34 KB. of shared memory used for each block, we sacrifice 33% of parallelism, the same performance hit than previously if we were unable of cutting down to 10 registers/thread.

VI. Examples: VectorAdd, Stencil, ReverseArray, MxM

NVIDIA

Step for building the CUDA source code

1. Identify those parts with a good potential to run in parallel exploiting SIMD data parallelism.
2. Identify all data necessary for the computations.
3. Move data to the GPU.
4. Call to the computational kernel.
5. Establish the required CPU-GPU synchronization.
6. Transfer results from GPU back to CPU.
7. Integrate the GPU results into CPU variables.

Coordinated efforts in parallel are required

- Parallelism is given by blocks and threads.
- Threads within each block may require an explicit synchronization, as only within a warp it is guaranteed its joint evolution (SIMD). Example:

```
a[i] = b[i] + 7;
syncthreads();
x[i] = a[i-1]; // The warp 1 reads here the value of a[31],
               // which should have been written by warp 0 BEFORE
```

- Kernel borders place implicit barriers:
  - `Kernel1 <<<nblocks,nthreads>>> (a,b,c);`
  - `Kernel2 <<<nblocks,nthreads>>> (a,b);`
- Blocks can coordinate using atomic operations:
  - Example: Increment a counter `atomicInc();`

VI. 1. Adding two vectors

The required code for the GPU kernel and its invocation from the CPU side

```
// Add two vectors of size N: C[1..N] = A[1..N] + B[1..N]
// Each thread calculates a single component of the output vector
__global__ void vecAdd(float* A, float* B, float* C) {
       int tid = blockIdx.x * blockDim.x + threadIdx.x;
       C[tid] = A[tid] + B[tid];
}
```

GPU code

```
int main() { // Launch N/256 blocks of 256 threads each
       vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

CPU code

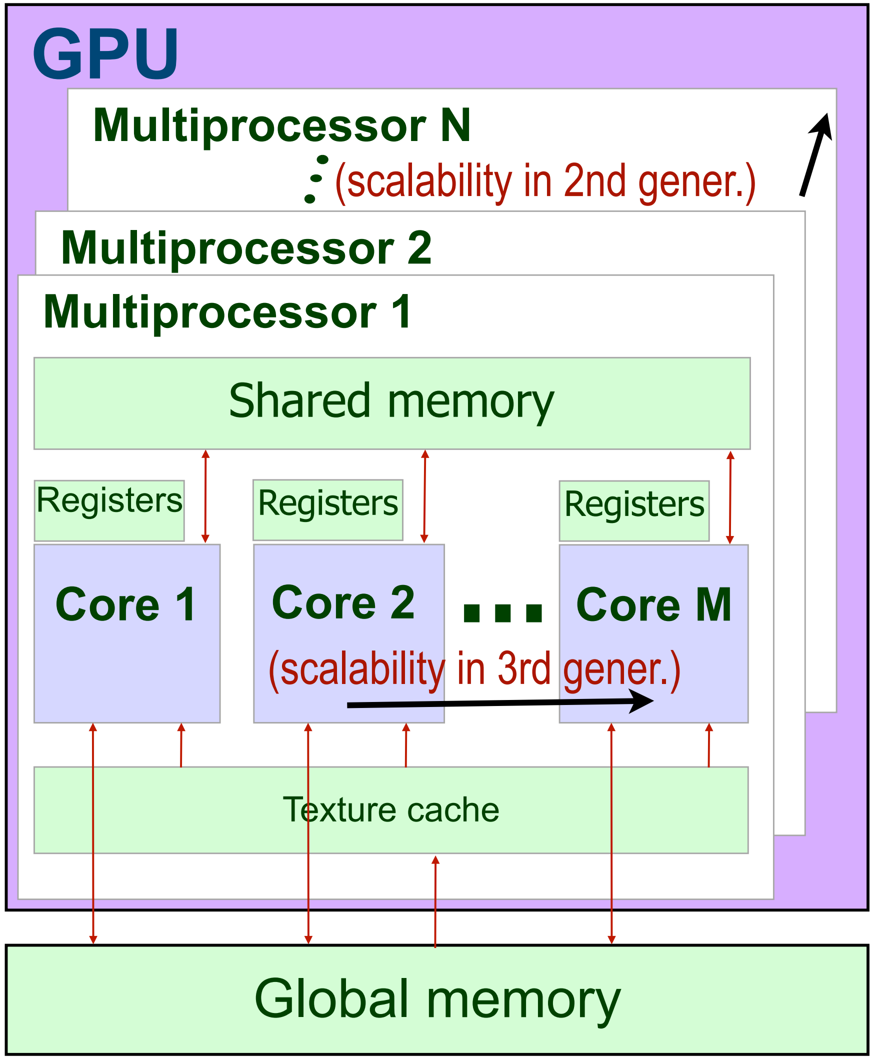
- The `__global__` prefix indicates that `vecAdd()` will execute on device (GPU) and will be called from host (CPU).
- `A`, `B` and `C` are pointers to device memory, so we need to:
  - Allocate/free memory on GPU, using `cudaMalloc()/cudaFree()`.
  - These pointers cannot be dereferenced in host code.

CPU code to handle memory and gather results from the GPU

```
unsigned int numBytes = N * sizeof(float);
// Allocates CPU memory
float* h_A = (float*) malloc(numBytes);
float* h_B = (float*) malloc(numBytes);
... initializes h_A and h_B ...
// Allocates GPU memory
float* d_A = 0;  cudaMalloc((void**)&d_A, numBytes);
float* d_B = 0;  cudaMalloc((void**)&d_B, numBytes);
float* d_C = 0;  cudaMalloc((void**)&d_C, numBytes);
// Copy input data from CPU into GPU
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
... CALL TO THE VecAdd KERNEL IN THE PREVIOUS SLIDE HERE...
// Copy results from GPU back to CPU
float* h_C = (float*) malloc(numBytes);
cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);
// Free video memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

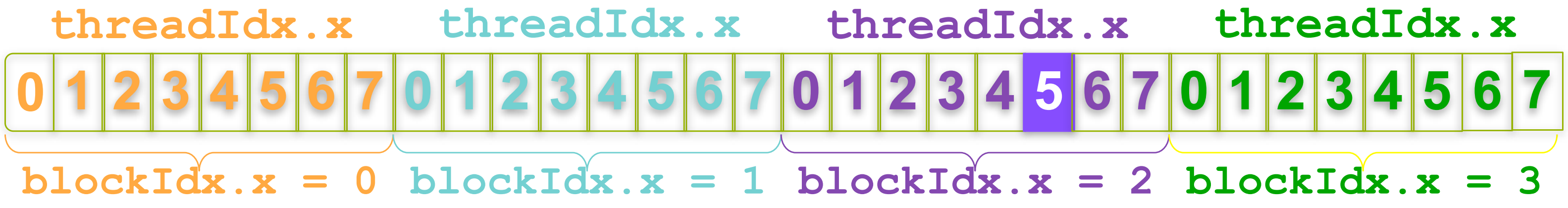
Running in parallel (regardless of hardware generation)

- `vecAdd <<< 1, 1 >>> ()` Executes 1 block composed of 1 thread - no parallelism.
- `vecAdd <<< B, 1 >>> ()` Executes B blocks composed on 1 thread. Inter-multiprocessor parallelism.
- `vecAdd <<< B, M >>> ()` Executes B blocks composed of M threads each. Inter- and intra-multiprocessor parallelism.



Indexing arrays with blocks and threads

- With M threads per block, a unique index is given by:
  - `tid = blockIdx.x*blockDim.x + threadIdx.x;`
- Consider indexing an array of one element per thread (because we are interested in fine-grained parallelism), B=4 blocks of M=8 threads each:



threadIdx.x threadIdx.x threadIdx.x threadIdx.x

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

blockIdx.x = 0 blockIdx.x = 1 blockIdx.x = 2 blockIdx.x = 3

- Which thread will compute the 22nd element of the array?
  - gridDim.x is 4. blockDim.x is 8. blockIdx.x = 2. threadIdx.x = 5.
  - tid = (2 * 8) + 5 = 21 (we start from 0, so this is the 22nd element).

Handling arbitrary vector sizes

- Typical problems are not friendly multiples of blockDim.x, so we have to prevent accessing beyond the end of arrays:

```
// Add two vectors of size N: C[1..N] = A[1..N] + B[1..N]
__global__ void vecAdd(float* A, float* B, float* C, N) {
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < N)
      C[tid] = A[tid] + B[tid];
}
```

- And now, update the kernel launch to include the "incomplete" block of threads:

```
vecAdd<<< (N + M-1)/256, 256>>>(d_A, d_B, d_C, N);
```

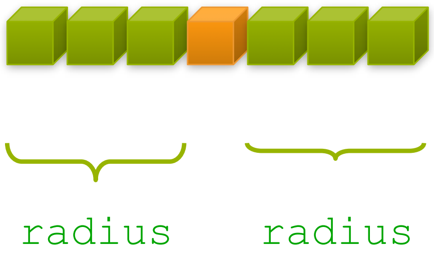
VI. 2. Stencil kernels

Rationale

- Looking at the previous example, threads add a level of complexity without contributing with new features.
- However, unlike parallel blocks, threads can:
  - Communicate (via shared memory).
  - Synchronize (for example, to preserve data dependencies).
- We need a more sophisticated example to illustrate all this...

1D Stencil

- Consider applying a 1D stencil to a 1D array of elements.
  - Each output element is the sum of input elements within a radius.
- If radius is 3, then each output element is the sum of 7 input elements:



- Again, we apply fine-grained parallelism for each thread to process a single output element.
- Input elements are read several times:
  - With radius 3, each input element is read seven times.

Sharing data between threads. Advantages

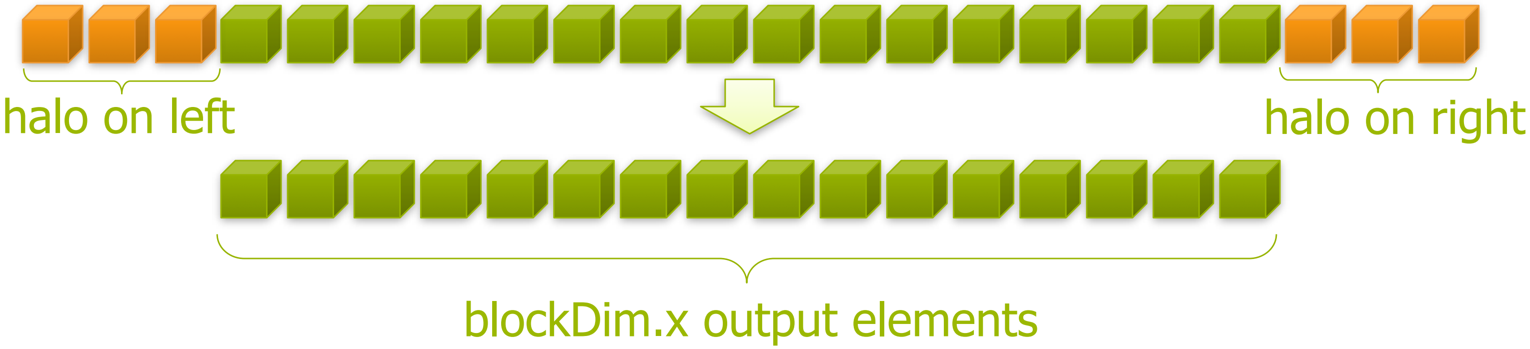
- Threads within a block can share data via shared memory.
  - Shared memory is user-managed: Declare with __shared__ prefix.
  - Data is allocated per block.
  - Shared memory is extremely fast:
    - 500 times faster than global memory (video memory - GDDR5). The difference is technology: static (built with transistors) versus dynamic (capacitors).
    - Programmer can see it like an extension of the register file.
  - Shared memory is more versatile than registers:
    - Registers are private to each thread, shared memory is private to each block.

Sharing data between threads. Limitations

- Shared memory and registers usage limit parallelism.
    - If we leave room for a second block, register file and shared memory are partitioned (even though blocks do not execute simultaneously, **context switch is immediate**).
- Examples for Kepler were shown before (for a max. of 64K registers and 48 Kbytes of shared memory per multiproc.):
    - To allocate two blocks per multiprocessor: The block cannot use more than 32 Kregisters and 24 Kbytes of shared memory.
    - To allocate three blocks per multiprocessor: The block cannot use more than 21.3 Kregisters and 16 Kbytes of shared memory.
    - To allocate four blocks per multiprocessor: The block cannot use more than 16 Kregisters and 12 Kbytes of shared memory.
    - ... and so on. Use the CUDA Occupancy Calculator to figure it out.

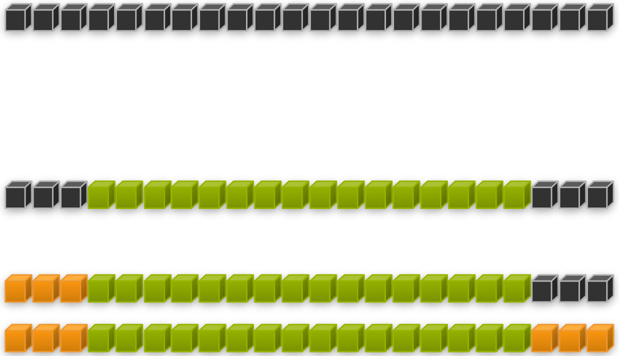
Using Shared Memory

- Steps to cache data in shared memory:
  - Read (`blockDim.x + 2 * radius`) input elements from global memory to shared memory.
  - Compute `blockDim.x` output elements.
  - Write `blockDim.x` output elements to global memory.
- Each block needs a halo of `radius` elements at each boundary.



Stencil kernel

```
__global__ void stencil_1d(int *d_in, int *d_out)
{
  __shared__ int temp[BLOCKSIZE + 2 * RADIUS];
  int gindex = blockIdx.x * blockDim.x + threadIdx.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = d_in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex-RADIUS] = d_in[gindex-RADIUS];
    temp[lindex+blockDim.x]=d_in[gindex+blockDim.x];
  }

  // Apply the stencil
  int result = 0;
  for (int offset=-RADIUS; offset<=RADIUS; offset++) {
    result += temp[lindex + offset];
  }
  // Store the result
  d_out[gindex] = result;
}
```



But we have to prevent race conditions. For example, last thread reads the halo before first thread (from a different warp) has fetched it. Synchronization among threads is required!

Threads synchronization

- Use `__syncthreads()` to synchronize all threads within a block:
  - All threads must reach the barrier before progressing.
  - This can be used to prevent RAW / WAR / WAW hazards.
  - In conditional code, the condition must be uniform across the block.

```
__global__ void stencil_1d(...)
{
  < Declare variables and indices >
  < Read input elements into shared memory >

  __syncthreads();

  < Apply the stencil >
  < Store the result >
}
```

Summary of major concepts applied during this example

- Launch N blocks with M threads per block to execute threads in parallel. Use:
  - `kernel <<< N, M >>> ();`
- Access block index within grid and thread index within block:
  - `blockIdx.x` and `threadIdx.x;`
- Calculate global indices where each thread has to work depending on data partitioning. Use:
  - `int index = blockIdx.x * blockDim.x + threadIdx.x;`
- Declare a variable/array in shared memory. Use:
  - `__shared__` (as prefix to the data type).
- Synchronize threads to prevent data hazards. Use:
  - `__syncthreads();`

VI. 3. Reverse the order of a vector of elements

GPU code for the ReverseArray kernel (1) using a single block

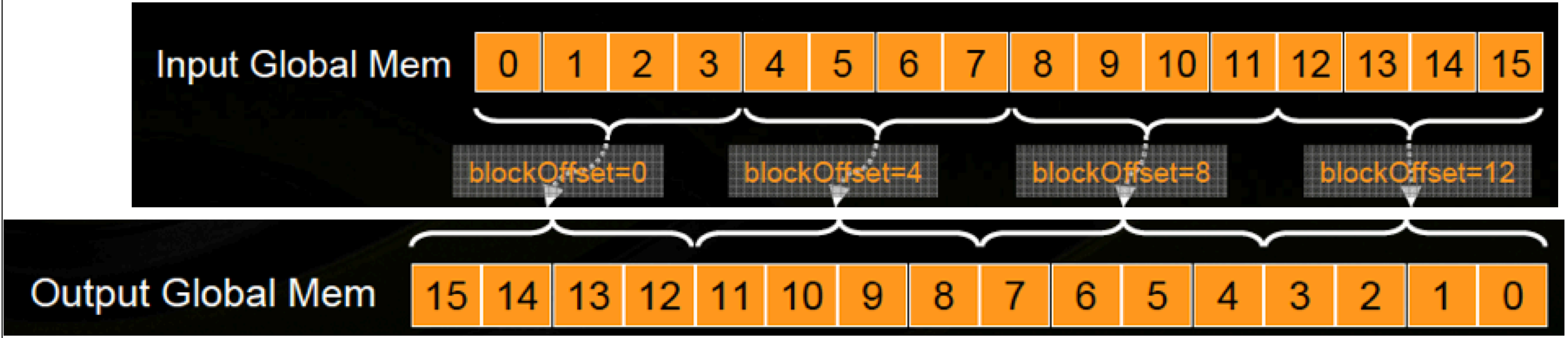
```
__global__ void reverseArray(int *in, int *out) {
  int index_in = threadIdx.x;
  int index_out = blockDim.x – 1 – threadIdx.x;

  // Reverse array contents using a single block
  out[index_out] = in[index_in];
}
```

- It is a naive solution which does not aspire to apply massive parallelism. The maximum block size is 1024 threads, so that is the largest vector that this code would accept as input.

GPU code for the ReverseArray kernel (2) using multiple blocks

```
__global__ void reverseArray(int *in, int *out) { // For thread 0 within block 0:
  int in_offset  =                     blockIdx.x  * blockDim.x; // in_offset =   0;
  int out_offset = (gridDim.x – 1 – blockIdx.x) * blockDim.x; // out_offset = 12;
  int index_in   =  in_offset +                   threadIdx.x;  // index_in =   0;
  int index_out = out_offset + (blockDim.x – 1 – threadIdx.x); // index_out = 15;

  // Reverse contents in chunks of whole blocks
  out[index_out] = in[index_in];
}
```

- For an example of 4 blocks, each composed of 4 threads:

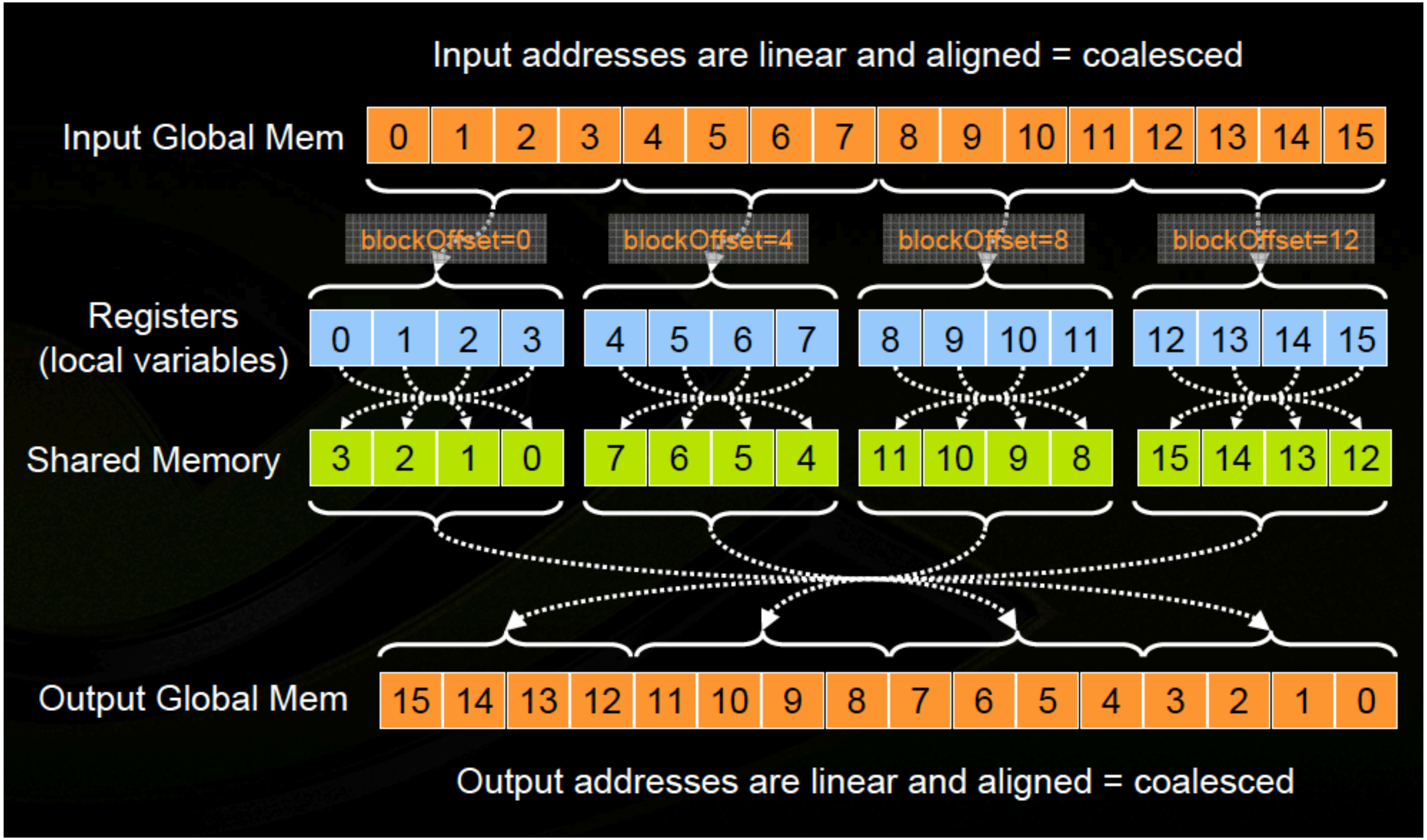


Input Global Mem 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

blockOffset=0 blockOffset=4 blockOffset=8 blockOffset=12

Output Global Mem 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

A more sophisticated version using shared memory



GPU code for the ReverseArray kernel (3) using multiple blocks and shared memory

```
__global__ void reverseArray(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE];
  int gindex = blockIdx.x * blockDim.x + threadIdx.x;
  int lindex = threadIdx.x;

  temp[lindex] = in[gindex];               // Load the input vector into shared memory
  syncthreads();                                                                // (i1)
  temp[lindex] = temp[blockDim.x-lindex-1]; // Reverse local arrays within blocks (i2)
  syncthreads();                                                                // (i3)
  // Reverse contents in chunks of whole blocks                                    (i4)
  out[threadIdx.x + (((N/blockDim.x)-blockIdx.x-1) * blockDim.x)] = temp[lindex];
}
```

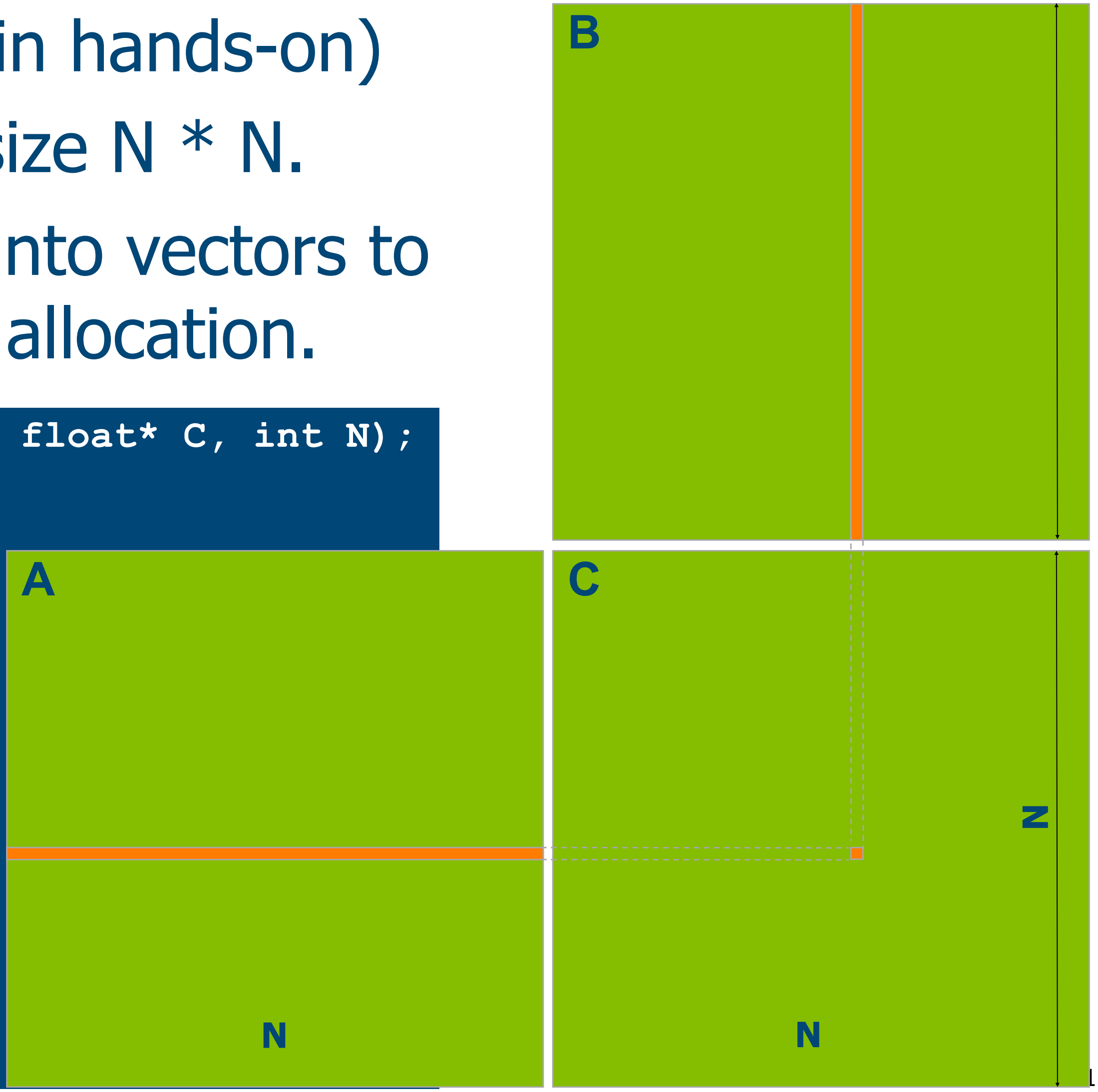
- Dependency: In (i2), values written by a warp, have to be read (before) by another warp.
- Solution: Use a `temp2[BLOCK_SIZE]` array to store intermediate results (also in (i4)).
- Improvement: (i3) is not required. Also, if you swap indices within `temp[]` and `temp2[]` in (i2), then (i1) is not required (but (i3) becomes mandatory).
- If you substitute all `temp` and `temp2` instances by their equivalent expressions, you converge into the previous CUDA version.
- Every array element is accessed once, so using shared memory does not improve anyway!

VI. 4. Matrix product

Typical CPU code written in C language

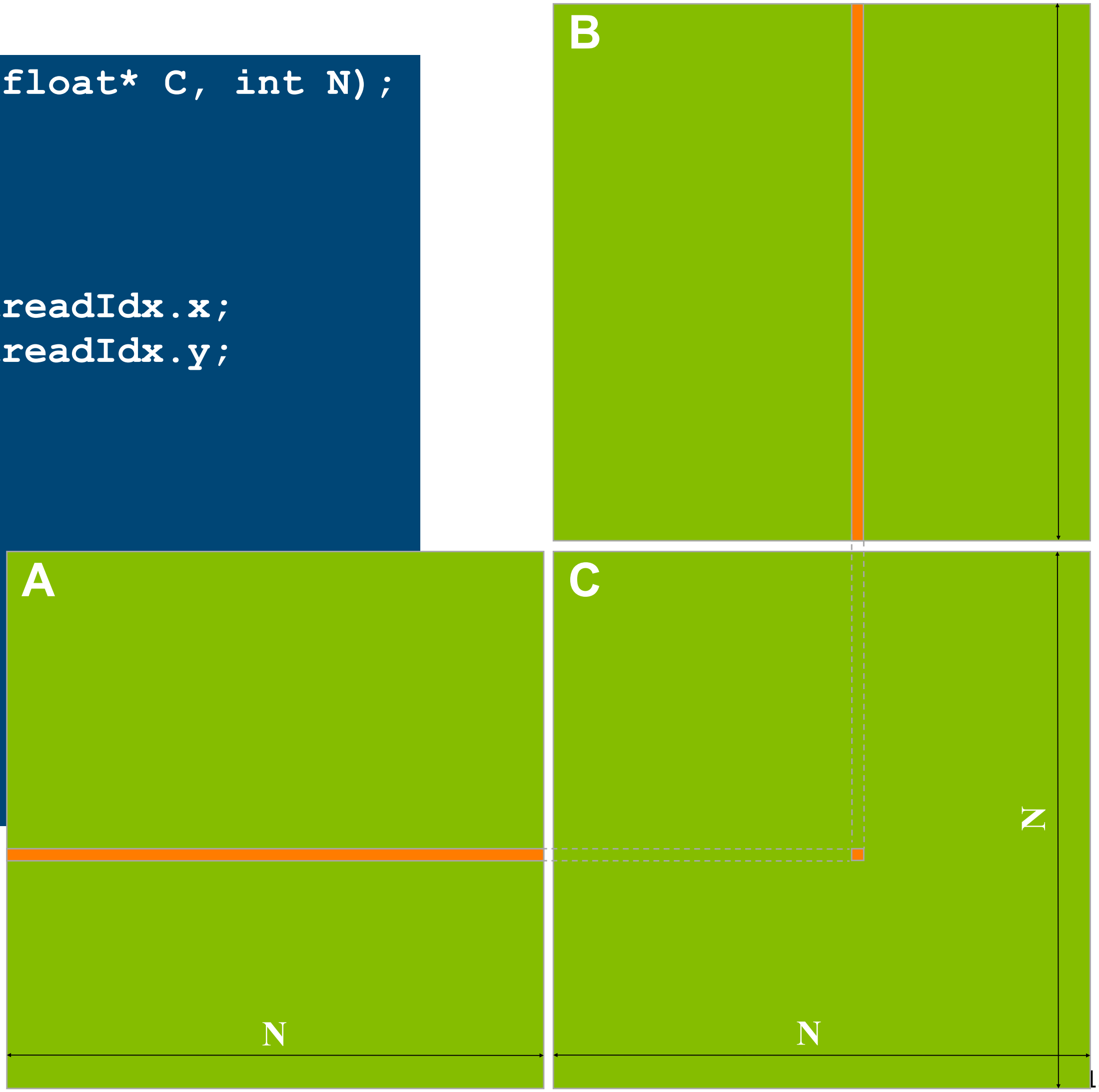
- C = A * B. (P = M * N in hands-on)
- All square matrices of size N * N.
- Matrices are serialized into vectors to simplify dynamic memory allocation.

```
void MxMonCPU(float* A, float* B, float* C, int N);
{
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
    {
      float sum=0;
      for (int k=0; k<N; k++)
      {
        float a = A[i*N + k];
        float b = B[k*N + j];
        sum += a*b;
      }
      C[i*N + j] = sum;
    }
}
```



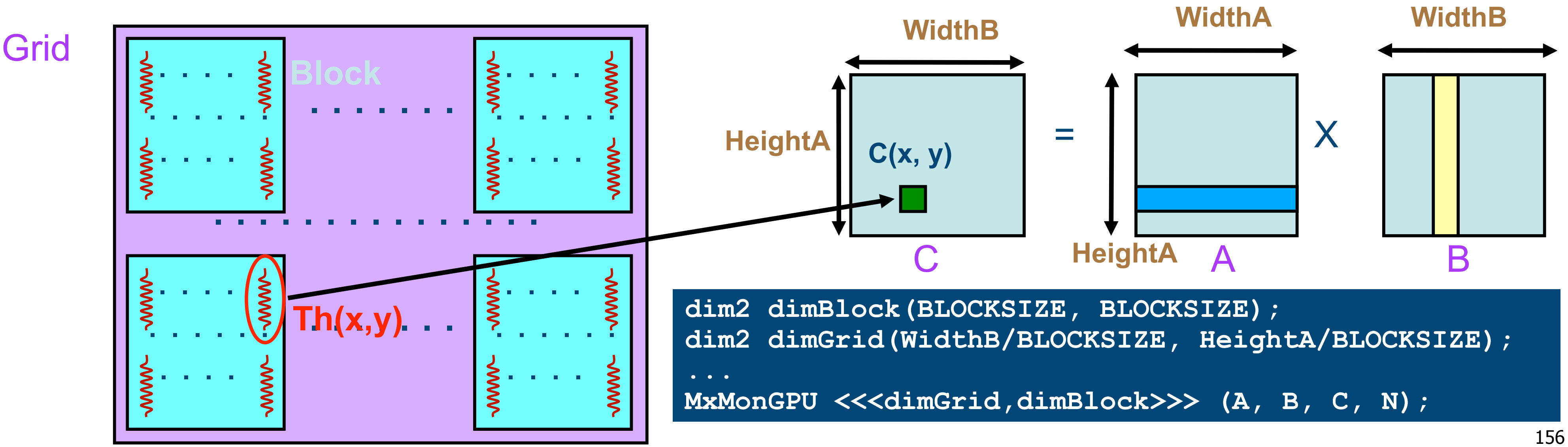
CUDA version for the matrix product: A draft for the parallel code

```
void MxMonGPU(float* A, float* B, float* C, int N);
{
  float sum=0;
  int i, j;

  i = blockIdx.x * blockDim.x + threadIdx.x;
  j = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k=0; k<N; k++)
  {
    float a = A[i*N + k];
    float b = B[k*N + j];
    sum += a*b;
  }
  C[i*N + j] = sum;
}
```



CUDA version for the matrix product: Explaining parallelization

- Each thread computes a single element of C.
    - Matrices A and B are loaded N times from video memory.
- Blocks accomodate threads in groups of 1024 threads (internal CUDA constraint in Fermi and Kepler). That way, we may use 2D blocks composed of 32x32 threads each.



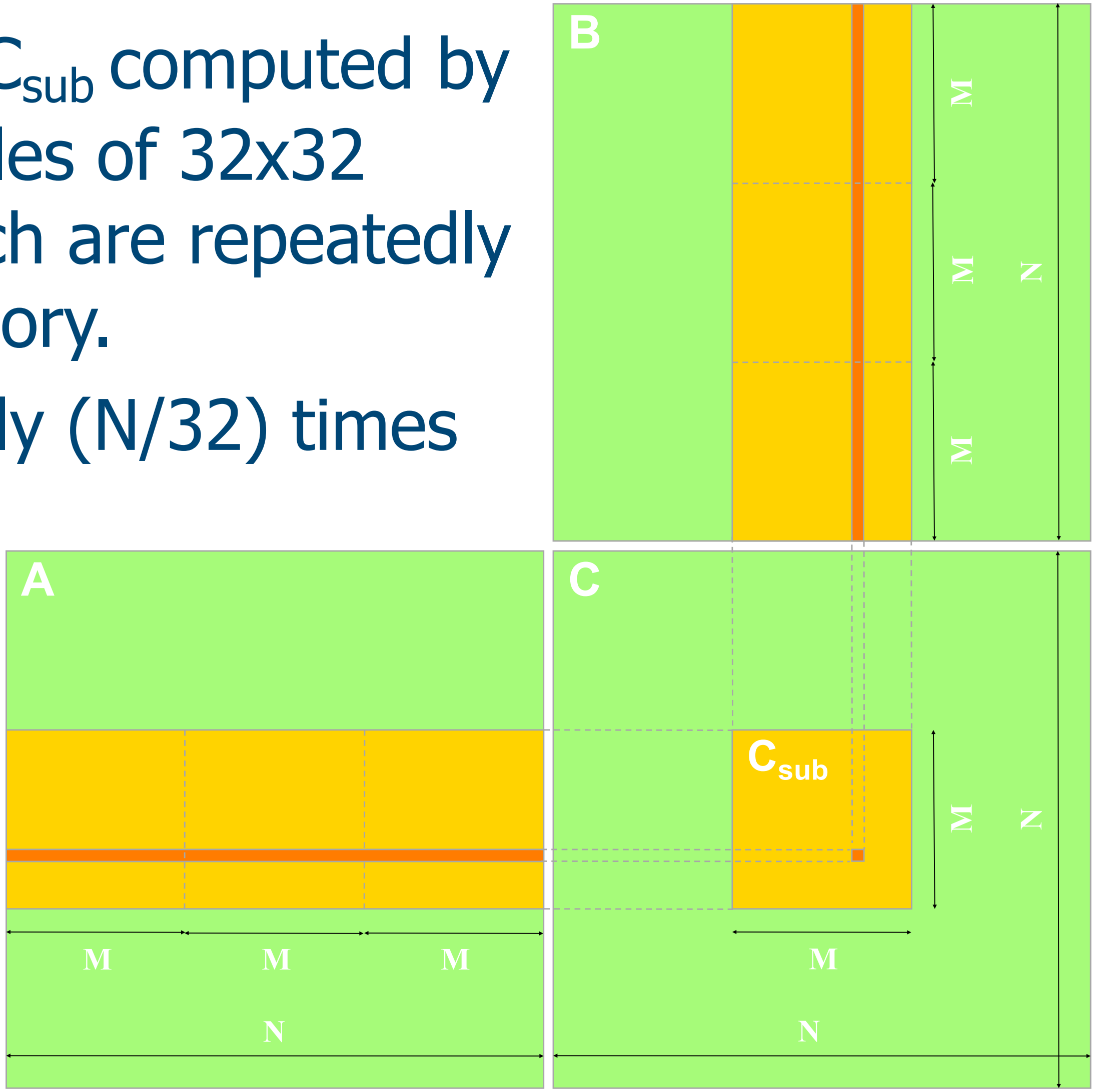
```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(WidthB/BLOCKSIZE, HeightA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid,dimBlock>>> (A, B, C, N);
```

CUDA version for the matrix product: Analysis

- Each thread requires 10 registers, so we can reach the maximum amount of parallelism in Kepler:
  - 2 blocks of 1024 threads (32x32) on each SMX. (2x1024x10 = 20480 registers, which is lower than 65536 registers available).
- Problems:
  - Low arithmetic intensity.
  - Demanding on memory bandwidth, which becomes the bottleneck.
- Solution:
  - Use shared memory on each multiprocessor.

Using shared memory: Version with tiling for A and B

- The 32x32 submatrix C_{sub} computed by each thread block uses tiles of 32x32 elements of A and B which are repeatedly allocated on shared memory.
- A and B are loaded only (N/32) times from global memory.
- Achievements:
  - Less demanding on memory bandwidth.
  - More arithmetic intensity.



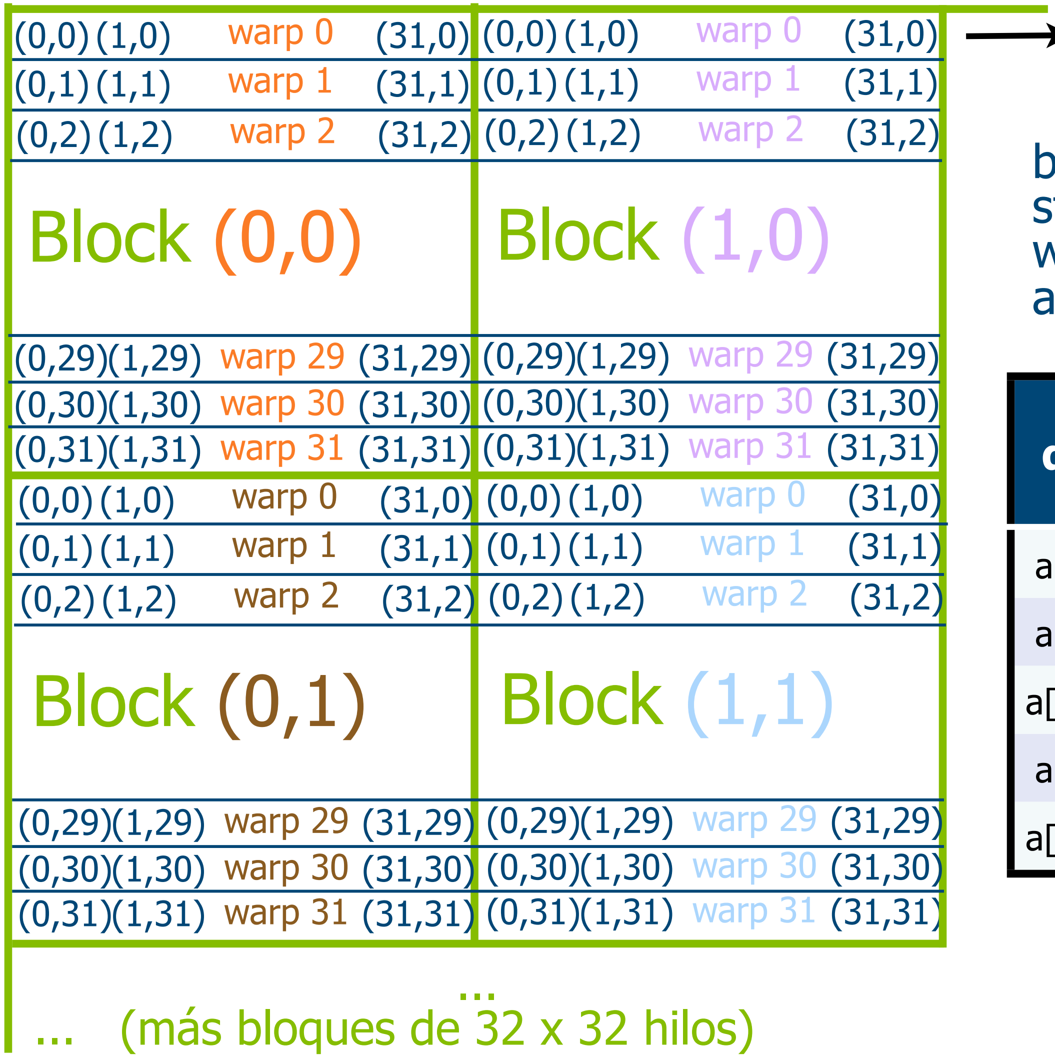
Tiling: Implementation details

- We have to manage all tiles involved within a thread block:
  - Load **in parallel** (all threads contribute) the input tiles (A and B) from global memory into shared memory. Tiles reuse the shared memory space.
  - `__syncthreads()` (to make sure we have loaded matrices before starting the computation).
  - Compute all products and sums for C using tiles within shared memory.
    - Each thread can now iterate independently on tile elements.
  - `__syncthreads()` (to make sure that the computation with the tile is over before loading, in the same memory space within share memory, two new tiles of A and B in the next iteration).

A trick to avoid shared memory bank conflicts

- Rationale:
  - The shared memory is structured into 16 (pre-Fermi) or 32 banks.
  - Threads within a block are numbered in column major order, that is, the x dimension is the fastest varying.
- When using the regular indexing scheme to shared memory arrays: **`As[threadIdx.x][threadIdx.y]`**, threads within a half-warp will be reading from the same column, that is, from the same bank in shared memory.
- However, using **`As[threadIdx.y][threadIdx.x]`**, threads within a half-warp will be reading from the same row, which implies reading from a different bank each.
- So, tiles store/access data **transposed** in shared memory.

An example for solving conflicts to banks in shared memory



(0,0) (1,0) warp 0 (31,0) (0,0) (1,0) warp 0 (31,0)

(0,1) (1,1) warp 1 (31,1) (0,1) (1,1) warp 1 (31,1)

(0,2) (1,2) warp 2 (31,2) (0,2) (1,2) warp 2 (31,2)

Block (0,0) Block (1,0)

(0,29)(1,29) warp 29 (31,29) (0,29)(1,29) warp 29 (31,29)

(0,30)(1,30) warp 30 (31,30) (0,30)(1,30) warp 30 (31,30)

(0,31)(1,31) warp 31 (31,31) (0,31)(1,31) warp 31 (31,31)

(0,0) (1,0) warp 0 (31,0) (0,0) (1,0) warp 0 (31,0)

(0,1) (1,1) warp 1 (31,1) (0,1) (1,1) warp 1 (31,1)

(0,2) (1,2) warp 2 (31,2) (0,2) (1,2) warp 2 (31,2)

Block (0,1) Block (1,1)

(0,29)(1,29) warp 29 (31,29) (0,29)(1,29) warp 29 (31,29)

(0,30)(1,30) warp 30 (31,30) (0,30)(1,30) warp 30 (31,30)

(0,31)(1,31) warp 31 (31,31) (0,31)(1,31) warp 31 (31,31)

...

... (más bloques de 32 x 32 hilos)

→ Consecutive threads within a warp differ in the first dimension.

but consecutive positions of memory store data of a bidimensional matrix which differ in the second dimension: a[0][0], a[0][1], a[0][2], ...

data	It is stored in bank	If thread (x,y) uses a[x][y], warp access to	If thread (x,y) uses a[y][x], warp access to
a[0][0]	0	X	X
a[0][1]	1		X
a[0][31]	31		X
a[1][0]	0	X	
a[31][0]	0	X	

↓ 100% conflicts

↓ No conflicts

Tiling: The CUDA code for the GPU kernel

```
__global__ void MxMonGPU(float *A, float *B, float *C, int N)
{
  int sum=0, tx, ty, i, j;
  tx = threadIdx.x;                      ty = threadIdx.y;
  i = blockIdx.x * blockDim.x + tx;     j = blockIdx.y * blockDim.y + ty;
  __shared__ float As[32][32], float Bs[32][32];

  // Traverse tiles of A and B required to compute the block submatrix for C
  for (int tile=0; tile<(N/32); tile++)
  {
    // Load tiles (32x32) from A and B in parallel (and store them transposed)
    As[ty][tx]= A[(i*N) + (ty+(tile*32))];
    Bs[ty][tx]= B[((tx+(tile*32))*N) + j];
    __syncthreads();
    // Compute results for the submatrix of C
    for (int k=0; k<32; k++) // Data have to be read from tiles transposed too
      sum += As[k][tx] * Bs[ty][k];
    __syncthreads();
  }
  // Write all results for the block in parallel
  C[i*N+j] = sum;
}
```

A compiler optimization: Loop unrolling

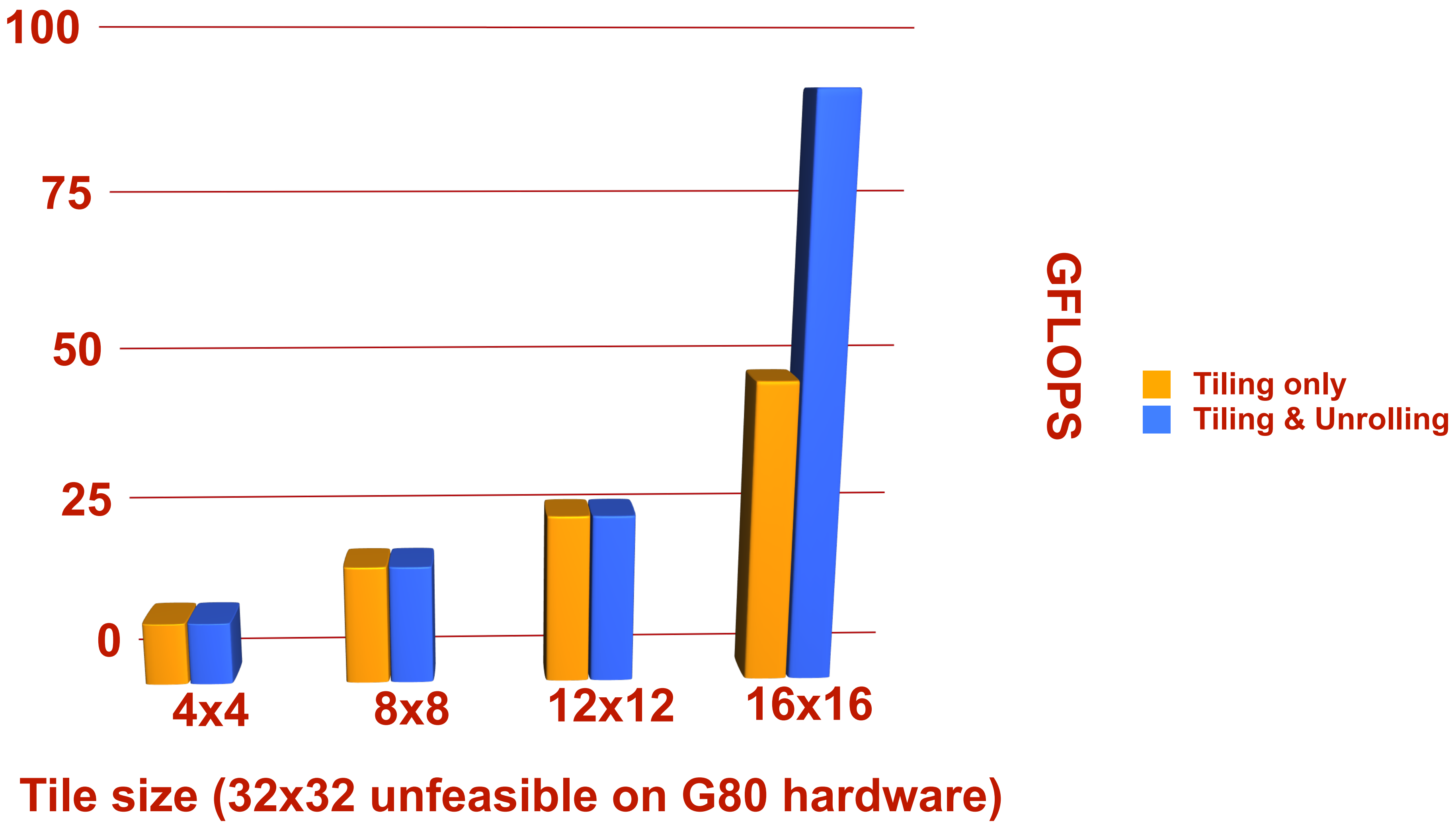
Without loop unrolling:

```
   ...
    __syncthreads();

   // Compute the tile
   for (k=0; k<32; k++)
     sum += As[tx][k]*Bs[k][ty];

    __syncthreads();
}
C[indexC] = sum;
```

Unrolling the loop:

```
    __syncthreads();

   // Compute the tile
    sum += As[tx][0]*Bs[0][ty];
    sum += As[tx][1]*Bs[1][ty];
    sum += As[tx][2]*Bs[2][ty];
    sum += As[tx][3]*Bs[3][ty];
    sum += As[tx][4]*Bs[4][ty];
    sum += As[tx][5]*Bs[5][ty];
    sum += As[tx][6]*Bs[6][ty];
    sum += As[tx][7]*Bs[7][ty];
    sum += As[tx][8]*Bs[8][ty];
    ...
    sum += As[tx][31]*Bs[31][ty];
    __syncthreads();
}
C[indexC] = sum;
```

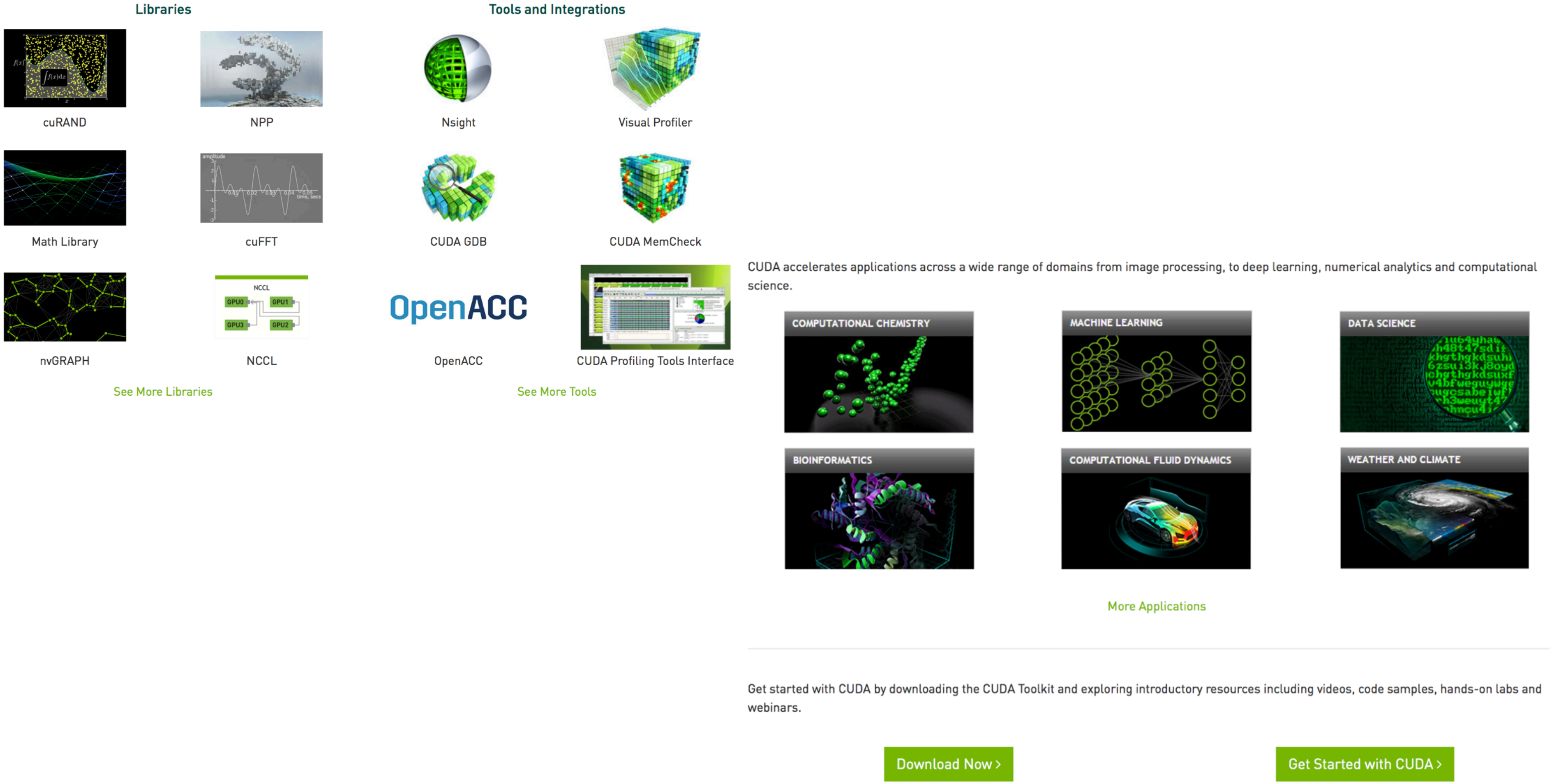
Performance on the G80 for tiling & unrolling



VII. Bibliography and tools

CUDA Zone:
The root web for a CUDA programmer

- [developer.nvidia.com/cuda-zone]



Libraries

Tools and Integrations

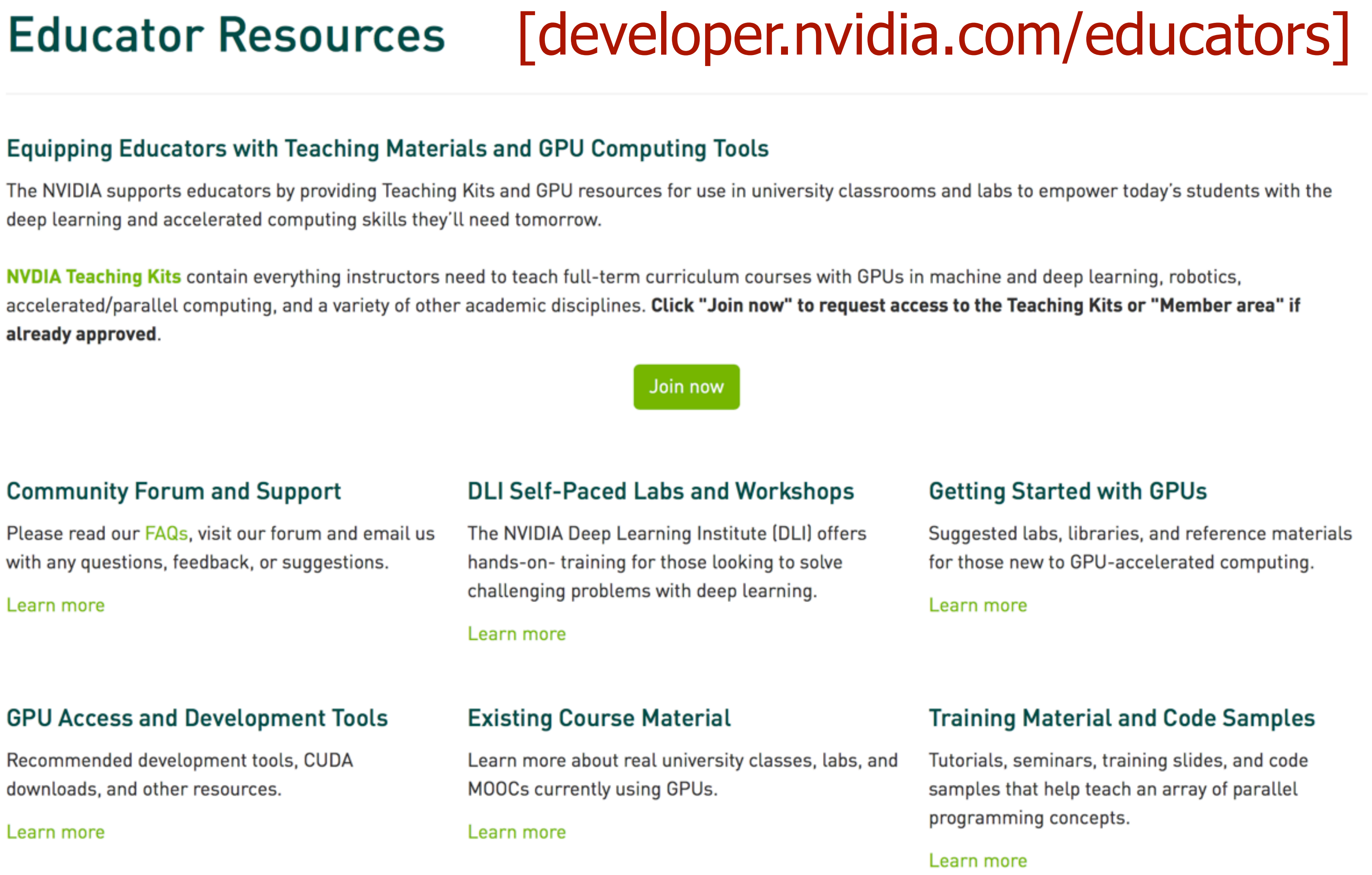
CUDA accelerates applications across a wide range of domains from image processing, to deep learning, numerical analytics and computational science.

Get started with CUDA by downloading the CUDA Toolkit and exploring introductory resources including videos, code samples, hands-on labs and webinars.

Guides for developers and more documents

- Getting started with CUDA C: Programmers guide.
  - [docs.nvidia.com/cuda/cuda-c-programming-guide]
- For tough programmers: The best practices guide.
  - [docs.nvidia.com/cuda/cuda-c-best-practices-guide]
- The root web collecting all CUDA-related documents:
  - [docs.nvidia.com/cuda]
- where we can find, additional guides for:
  - Installing CUDA on Linux, MacOS and Windows.
  - Optimize and improve CUDA programs on Kepler and Maxwell GPUs.
  - Check the CUDA API syntax (runtime, driver and math).
  - Learn to use libraries like cuBLAS, cuFFT, cuRAND, cuSPARSE, ...
  - Deal with basic tools (compiler, debugger, profiler).

Educator Resources



Educator Resources [developer.nvidia.com/educators]

Equipping Educators with Teaching Materials and GPU Computing Tools

The NVIDIA supports educators by providing Teaching Kits and GPU resources for use in university classrooms and labs to empower today's students with the deep learning and accelerated computing skills they'll need tomorrow.

NVDIA Teaching Kits contain everything instructors need to teach full-term curriculum courses with GPUs in machine and deep learning, robotics, accelerated/parallel computing, and a variety of other academic disciplines. **Click "Join now" to request access to the Teaching Kits or "Member area" if already approved**.

Join now

Community Forum and Support

Please read our FAQs, visit our forum and email us with any questions, feedback, or suggestions.

Learn more

DLI Self-Paced Labs and Workshops

The NVIDIA Deep Learning Institute (DLI) offers hands-on- training for those looking to solve challenging problems with deep learning.

Learn more

Getting Started with GPUs

Suggested labs, libraries, and reference materials for those new to GPU-accelerated computing.

Learn more

GPU Access and Development Tools

Recommended development tools, CUDA downloads, and other resources.

Learn more

Existing Course Material

Learn more about real university classes, labs, and MOOCs currently using GPUs.

Learn more

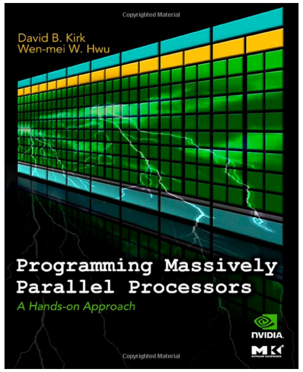
Training Material and Code Samples

Tutorials, seminars, training slides, and code samples that help teach an array of parallel programming concepts.

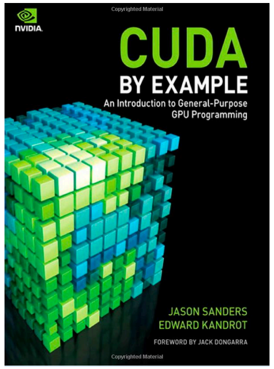
Learn more

CUDA books: From 2007 to 2015

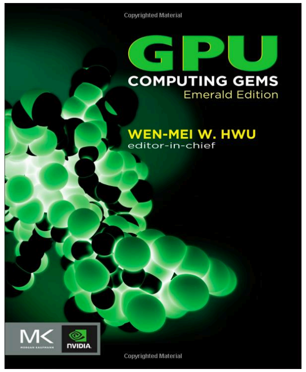
Sep'07



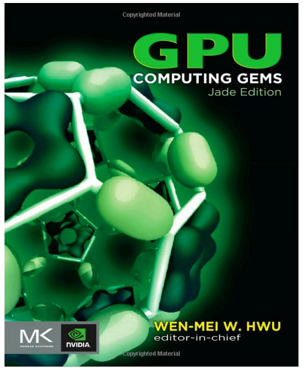
Feb'10


CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA

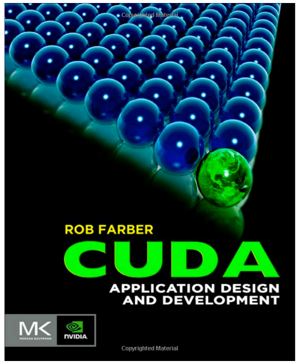
Jul'10



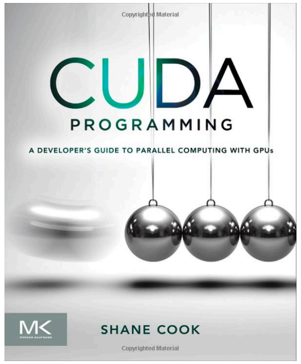
Abr'11



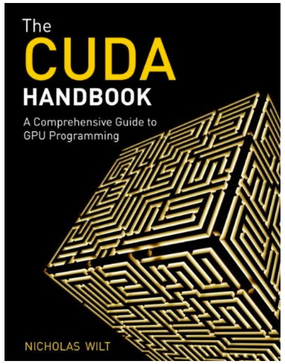
Oct'11



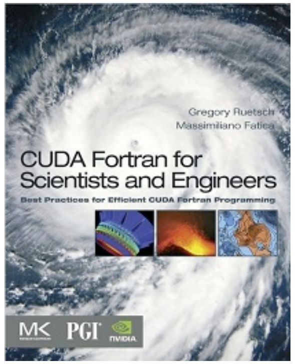
Nov'11



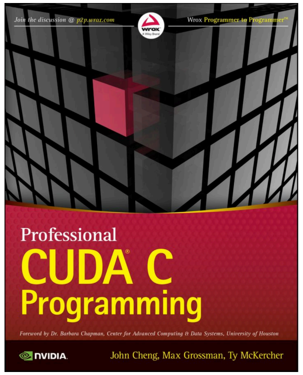
Dic'12



Jun'13


Gregory Ruetsch
Massimiliano Fatica
CUDA Fortran for Scientists and Engineers
Best Practices for Efficient CUDA Fortran Programming

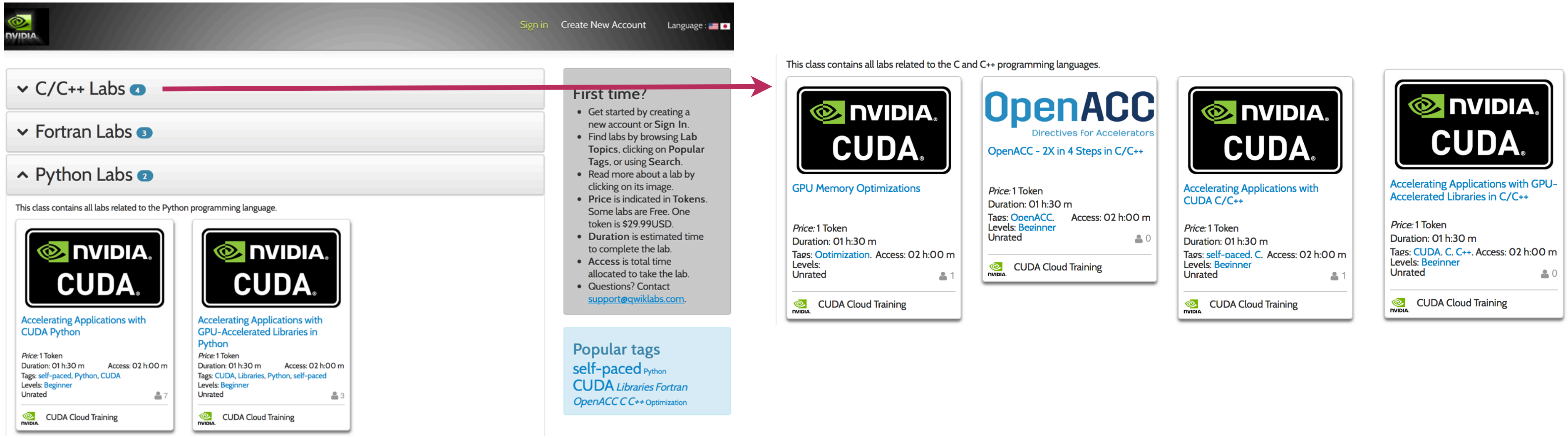
Oct'13



Sep'14

Tutorials about C/C++, Fortran and Python

- You have to register on the Amazon EC2 services available on the Web (cloud computing): [nvidia.qwiklab.com]
  - They are usually sessions of 90 minutes.
  - Only a Web browser and SSH client are required.
  - Some tutorials are free, other require tokens of $29.99.



C/C++ Labs 4

Fortran Labs 3

Python Labs 2

This class contains all labs related to the Python programming language.

Accelerating Applications with CUDA Python
Price: 1 Token
Duration: 01 h:30 m Access: 02 h:00 m
Tags: self-paced, Python, CUDA
Levels: Beginner
Unrated 7
CUDA Cloud Training

Accelerating Applications with GPU-Accelerated Libraries in Python
Price: 1 Token
Duration: 01 h:30 m Access: 02 h:00 m
Tags: CUDA, Libraries, Python, self-paced
Levels: Beginner
Unrated 3
CUDA Cloud Training

First time?
- Get started by creating a new account or **Sign In**.
- Find labs by browsing **Lab Topics**, clicking on **Popular Tags**, or using **Search**.
- Read more about a lab by clicking on its image.
- **Price** is indicated in **Tokens**. Some labs are Free. One token is $29.99USD.
- **Duration** is estimated time to complete the lab.
- **Access** is total time allocated to take the lab.
- Questions? Contact support@qwiklabs.com.

Popular tags
self-paced Python
CUDA *Libraries Fortran*
OpenACC C C++ Optimization

This class contains all labs related to the C and C++ programming languages.

GPU Memory Optimizations
Price: 1 Token
Duration: 01 h:30 m
Tags: Optimization. Access: 02 h:00 m
Levels:
Unrated 1
CUDA Cloud Training

OpenACC - 2X in 4 Steps in C/C++
Price: 1 Token
Duration: 01 h:30 m
Tags: OpenACC. Access: 02 h:00 m
Levels: Beginner
Unrated 0
CUDA Cloud Training

Accelerating Applications with CUDA C/C++
Price: 1 Token
Duration: 01 h:30 m
Tags: self-paced. C. Access: 02 h:00 m
Levels: Beginner
Unrated 1
CUDA Cloud Training

Accelerating Applications with GPU-Accelerated Libraries in C/C++
Price: 1 Token
Duration: 01 h:30 m
Tags: CUDA. C. C++. Access: 02 h:00 m
Levels: Beginner
Unrated 0
CUDA Cloud Training

Talks and webinars

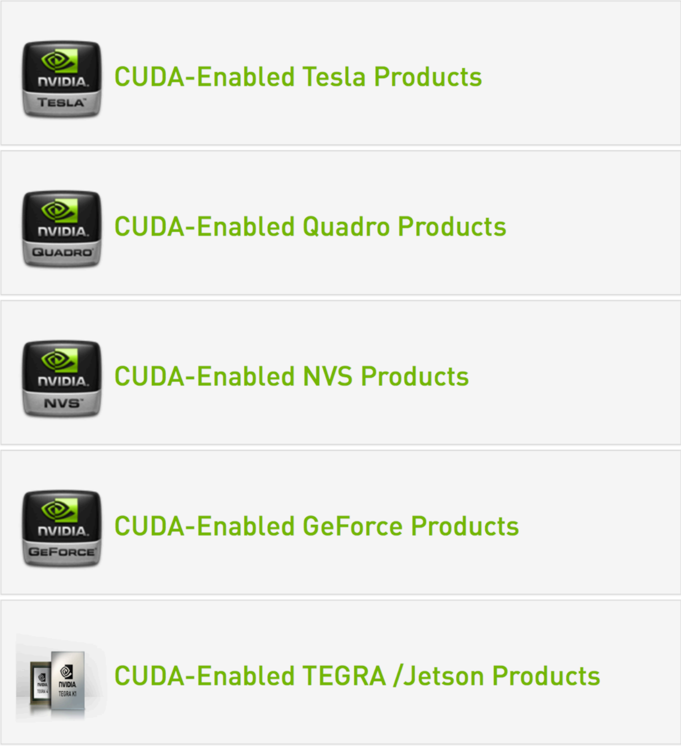
- Talks recorded at GTC (Graphics Technology Conference):
  - More than 500 talks available on each of the last editions (2013-18):
  - [www.gputechconf.com/gtcnew/on-demand-gtc.php]
- Webinars about GPU computing:
  - List of past talks on video (mp4/wmv) and slides (PDF).
  - List of incoming on-line talks to be enrolled.
  - [developer.nvidia.com/gpu-computing-webinars]

Developers

- Sign up as a registered developer:
  - [www.nvidia.com/developer-program]
  - Access to exclusive developer downloads.
  - Exclusive access to pre-release CUDA installers like CUDA 8.0.
  - Exclusive activities an special offers.
- Technical questions on-line:
  - NVIDIA Developer Forums: [devtalk.nvidia.com]
  - Search or ask on: [stackoverflow.com/tags/cuda]

Developers (2)

- List of CUDA-enabled GPUs:
  - [developer.nvidia.com/cuda-gpus]



CUDA-Enabled Tesla Products

CUDA-Enabled Quadro Products

CUDA-Enabled NVS Products

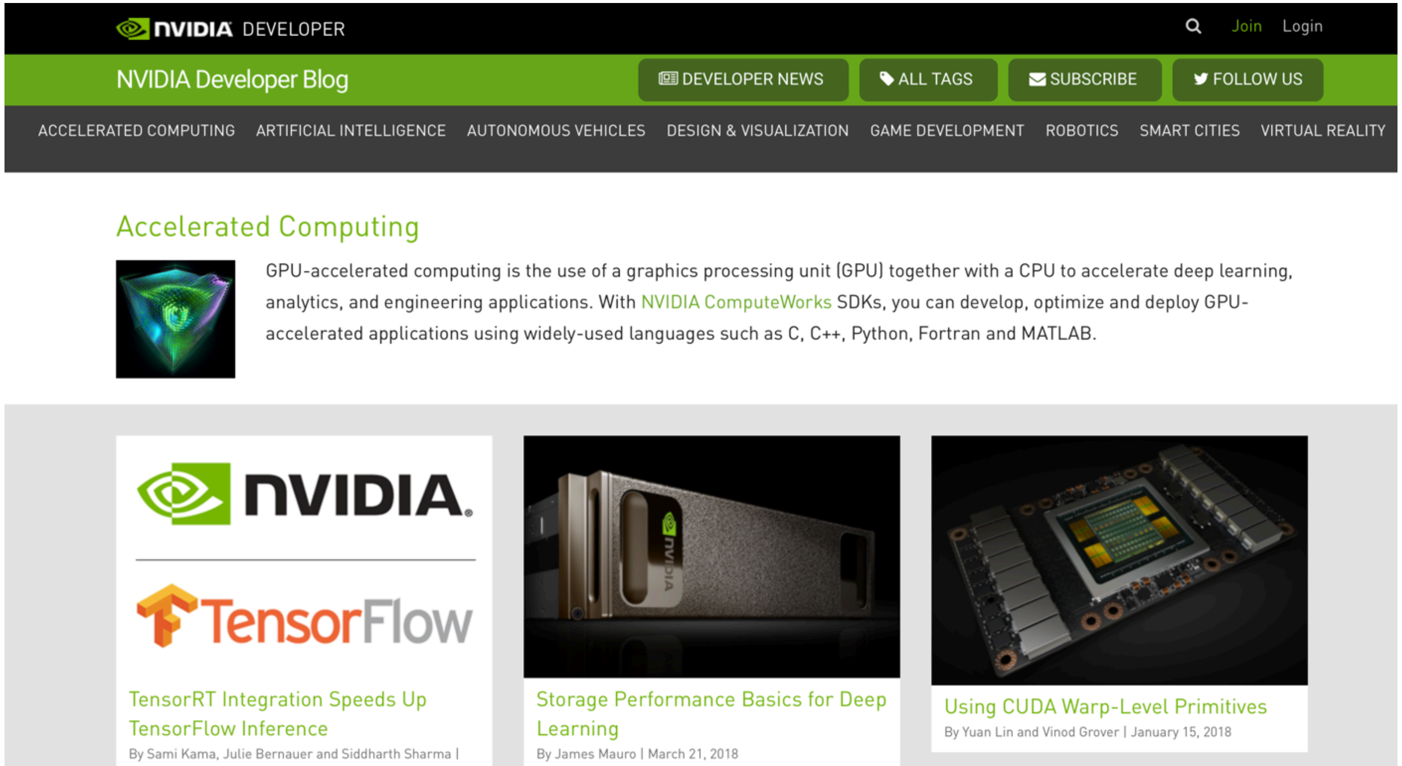
CUDA-Enabled GeForce Products

CUDA-Enabled TEGRA /Jetson Products

- And a last tool for tuning code: CUDA Occupancy Calculator
  - [developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls]

Articles about recent topics on CUDA and accelerated computing on GPU

- Nvidia's blog with state-of-the-art posts about CUDA and accelerated computing on GPUs:
  - https://devblogs.nvidia.com



NVIDIA DEVELOPER

Join Login

NVIDIA Developer Blog

DEVELOPER NEWS | ALL TAGS | SUBSCRIBE | FOLLOW US

ACCELERATED COMPUTING | ARTIFICIAL INTELLIGENCE | AUTONOMOUS VEHICLES | DESIGN & VISUALIZATION | GAME DEVELOPMENT | ROBOTICS | SMART CITIES | VIRTUAL REALITY

Accelerated Computing

GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications. With NVIDIA ComputeWorks SDKs, you can develop, optimize and deploy GPU-accelerated applications using widely-used languages such as C, C++, Python, Fortran and MATLAB.

TensorRT Integration Speeds Up TensorFlow Inference

By Sami Kama, Julie Bernauer and Siddharth Sharma |

Storage Performance Basics for Deep Learning

By James Mauro | March 21, 2018

Using CUDA Warp-Level Primitives

By Yuan Lin and Vinod Grover | January 15, 2018

Thanks for your attention!

- You can always reach me in Spain at the Computer Architecture Department of the University of Malaga:
  - e-mail: ujaldon@uma.es
  - Phone: +34 952 13 28 24.
  - Web page: http://manuel.ujaldon.es (english/spanish versions available).

- Or, more specifically on GPUs, visit my web page as Nvidia CUDA Fellow:
  - http://research.nvidia.com/users/manuel-ujaldon