GPGPU 2015: High Performance Computing with CUDA

Department of Computer Science. University of Cape Town

April 20th-24th, 2015

Manuel Ujaldón

Associate Professor @ Univ. of Malaga (Spain)
Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)
CUDA Fellow @ Nvidia

NVIDIA

Tutorial contents for today [118 slides]

1. Introduction. [17 slides]
2. Architecture. [21]
    1. CUDA hardware model. [3]
    2. The first generation: Tesla (2007-2009). [3]
    3. The second generation: Fermi (2010-2011). [3]
    4. The third generation: Kepler (2012-2014). [6]
    5. The fourth generation: Maxwell (2015-?). [5]
    6. Summary by generation. [1]
3. Programming. [15]
4. Syntax. [16]
    1. Basic elements. [10]
    2. A couple of preliminary examples. [6]
5. Compilation and tools [12]
6. Examples: VectorAdd, Stencil, MxM. [25]
7. Bibliography, resources and tools. [12]

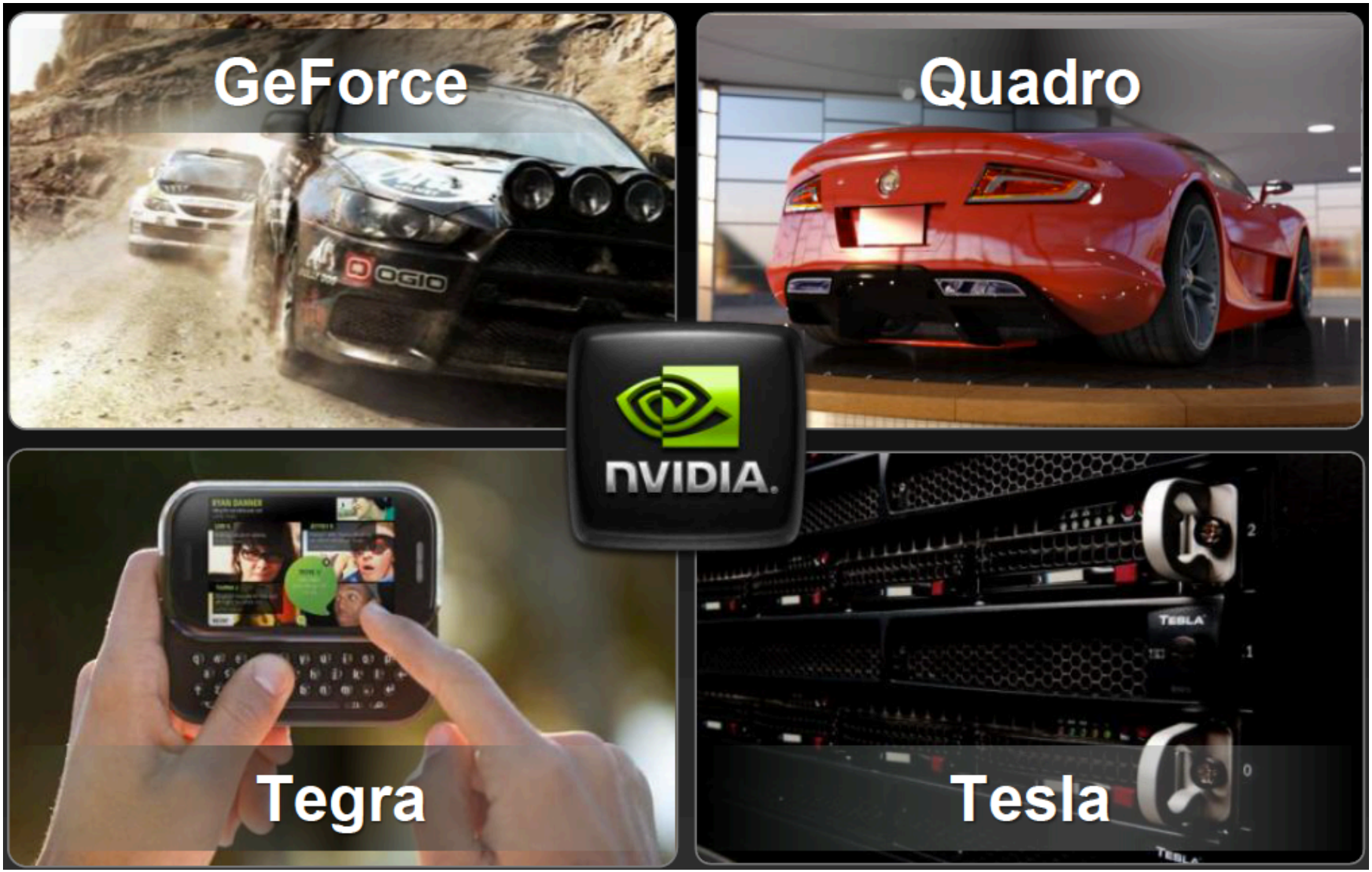
Prerequisites for this tutorial

- 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, ...

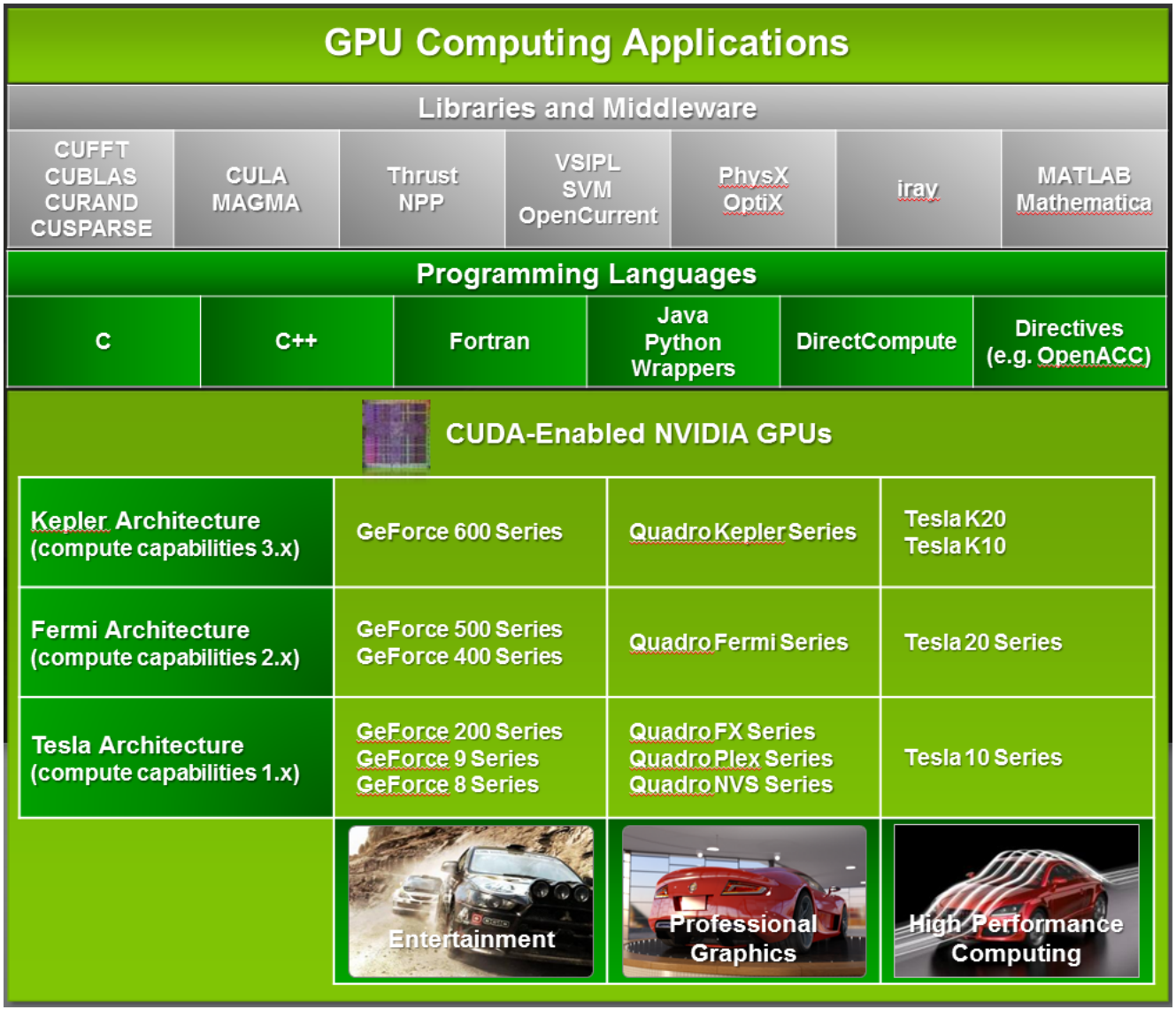
I. Introduction

NVIDIA

Welcome to the GPU world



The characters of this story: The CUDA family picture



The impressive evolution of CUDA

Year 2008	Year 2015
100.000.000 CUDA-capable GPUs (6.000 Teslas only)	600.000.000 CUDA-capable GPUs (and 450.000 Tesla high-end GPUs)
150.000 CUDA downloads	3.000.000 CUDA downloads per year (that is, one every 9 seconds)
1 supercomputer in top500.org (77 TFLOPS)	75 supercomputers in TOP500.org (aggregate 54.000 TFLOPS)
60 university courses	840 university courses
4.000 academic papers	60.000 academic papers

Worldwide distribution of CUDA university courses

Summary of GPU evolution

- 2001: First many-cores (vertex and pixel processors).
- 2003: Those processor become programmable (with 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.
- 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.

What is CUDA?
"Compute Unified Device Architecture"

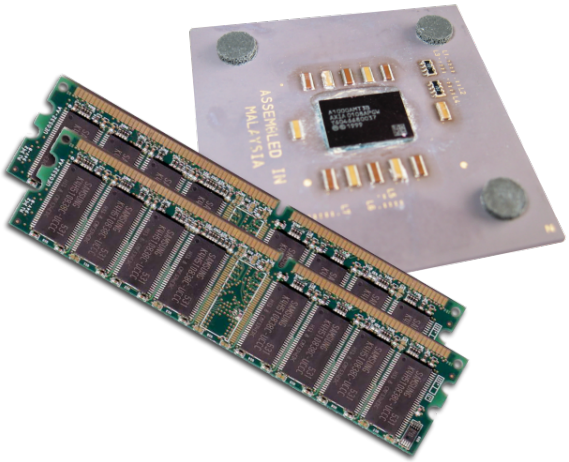
- A platform designed jointly at software and hardware levels to make use of the GPU computational power in general-purpose applications at three levels:
  - Software: It allows to program the GPU with minimal but powerful SIMD extensions to enable heterogeneous programming and attain an efficient and scalable execution.
  - Firmware: It offers a driver oriented to GPGPU programming, which is compatible with that used for rendering. Straightforward APIs to manage devices, memory, etc.
  - Hardware: It exposes GPU parallelism for general-purpose computing via a number of 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-grained 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 as of 2013].
  - Device: The graphics card [GPU + video memory]:
    - GPU: Nvidia GeForce/Tesla.
    - Video memory: GDDR5 as of 2015.

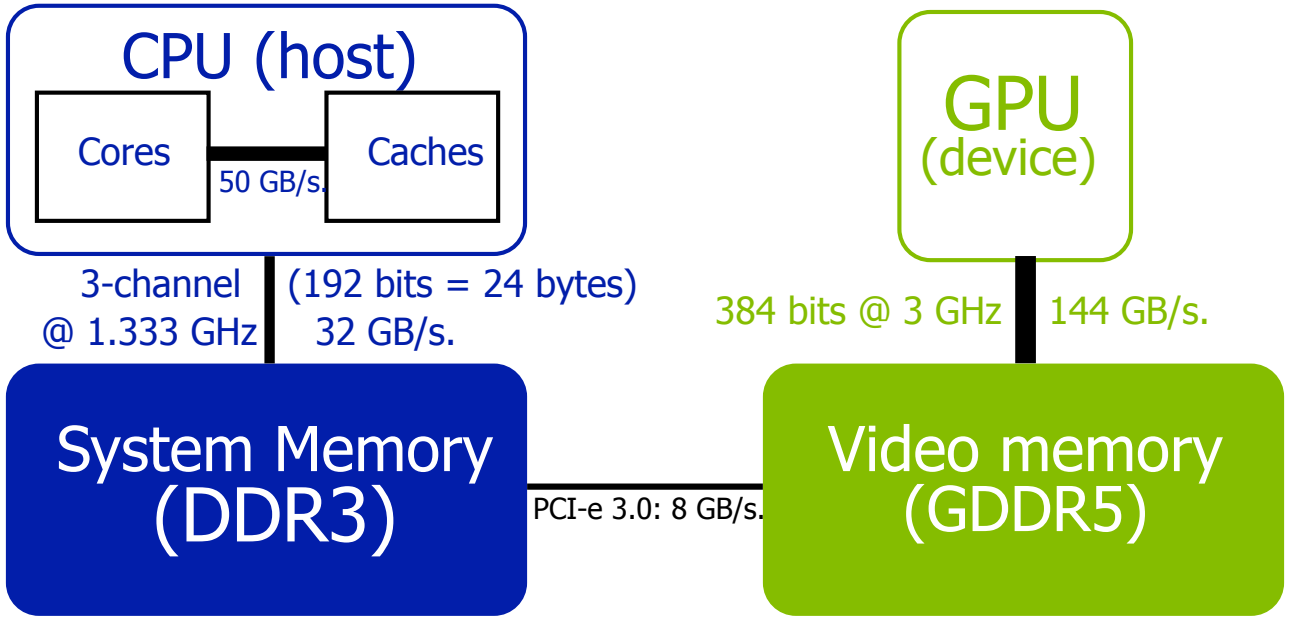


Host

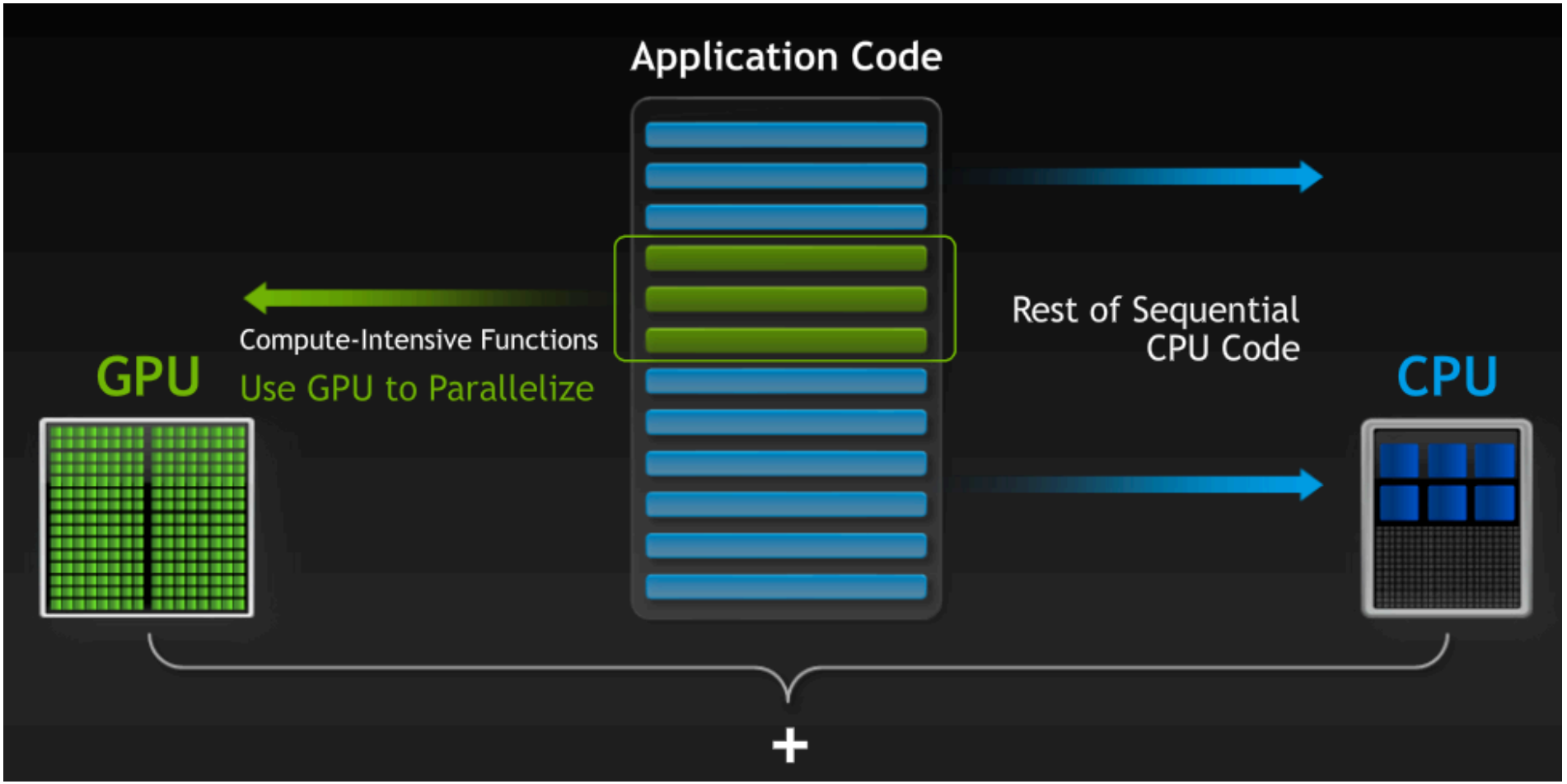
Device

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.

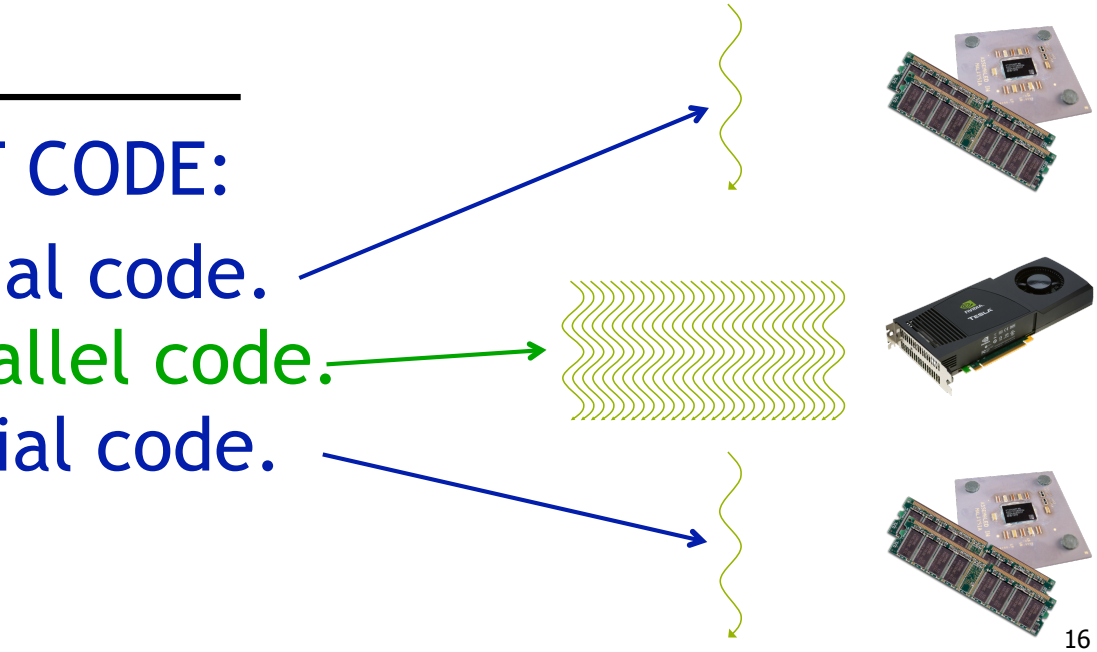
Heterogeneous Computing (4/4)

```
#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);
}

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;
}
```

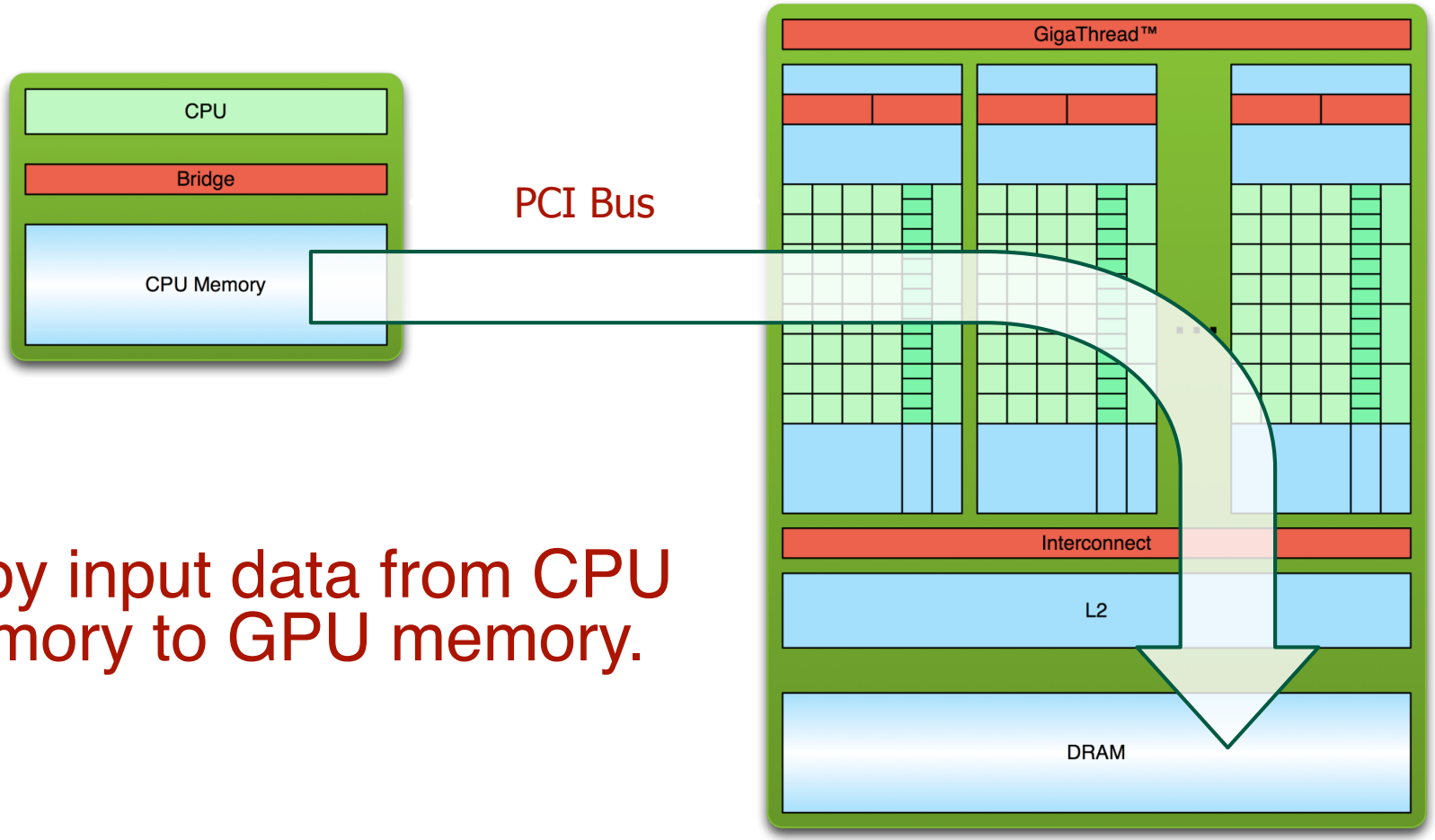
DEVICE CODE:
Parallel function written in CUDA.

HOST CODE:

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

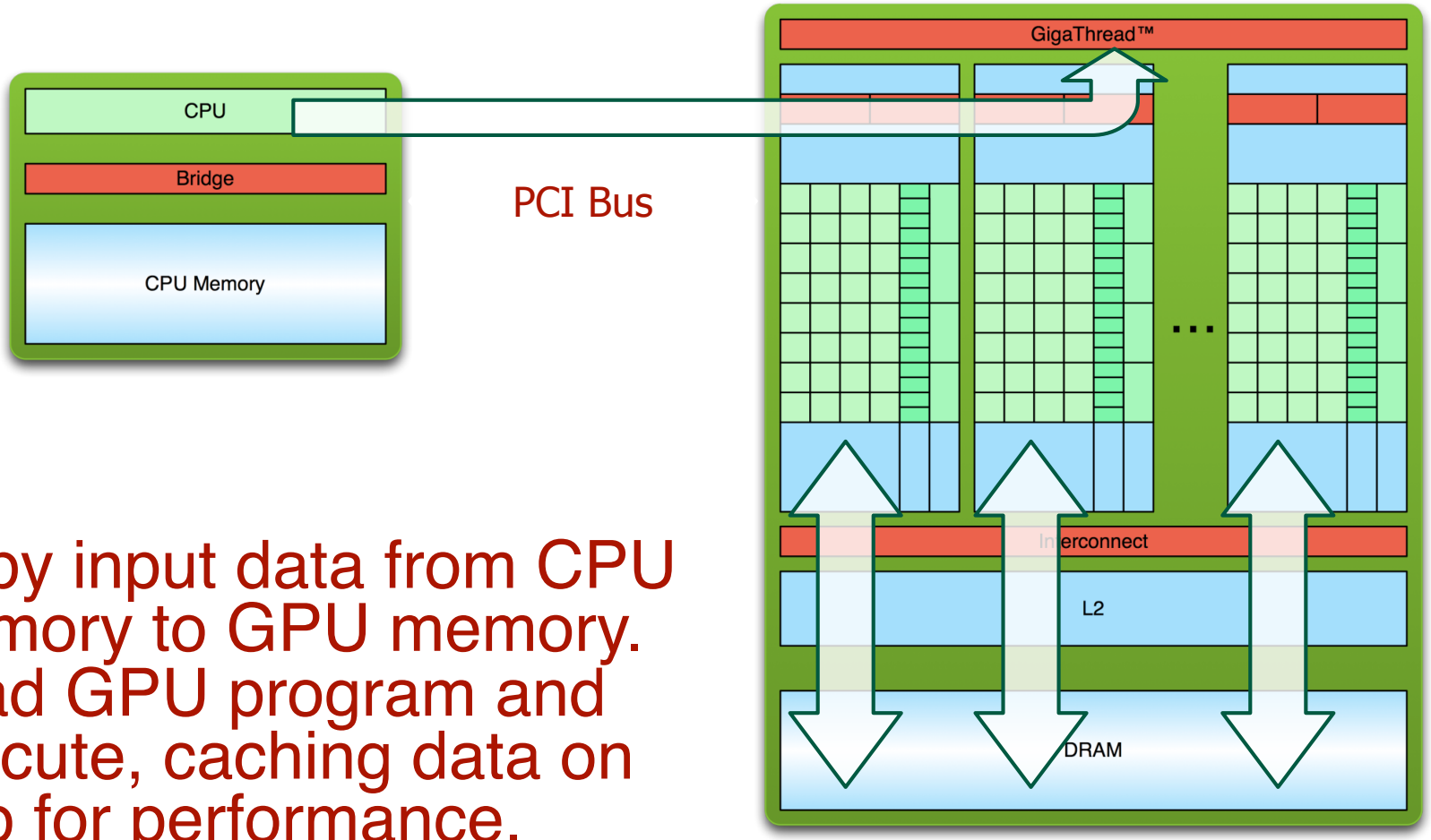


Simple Processing Flow (1/3)



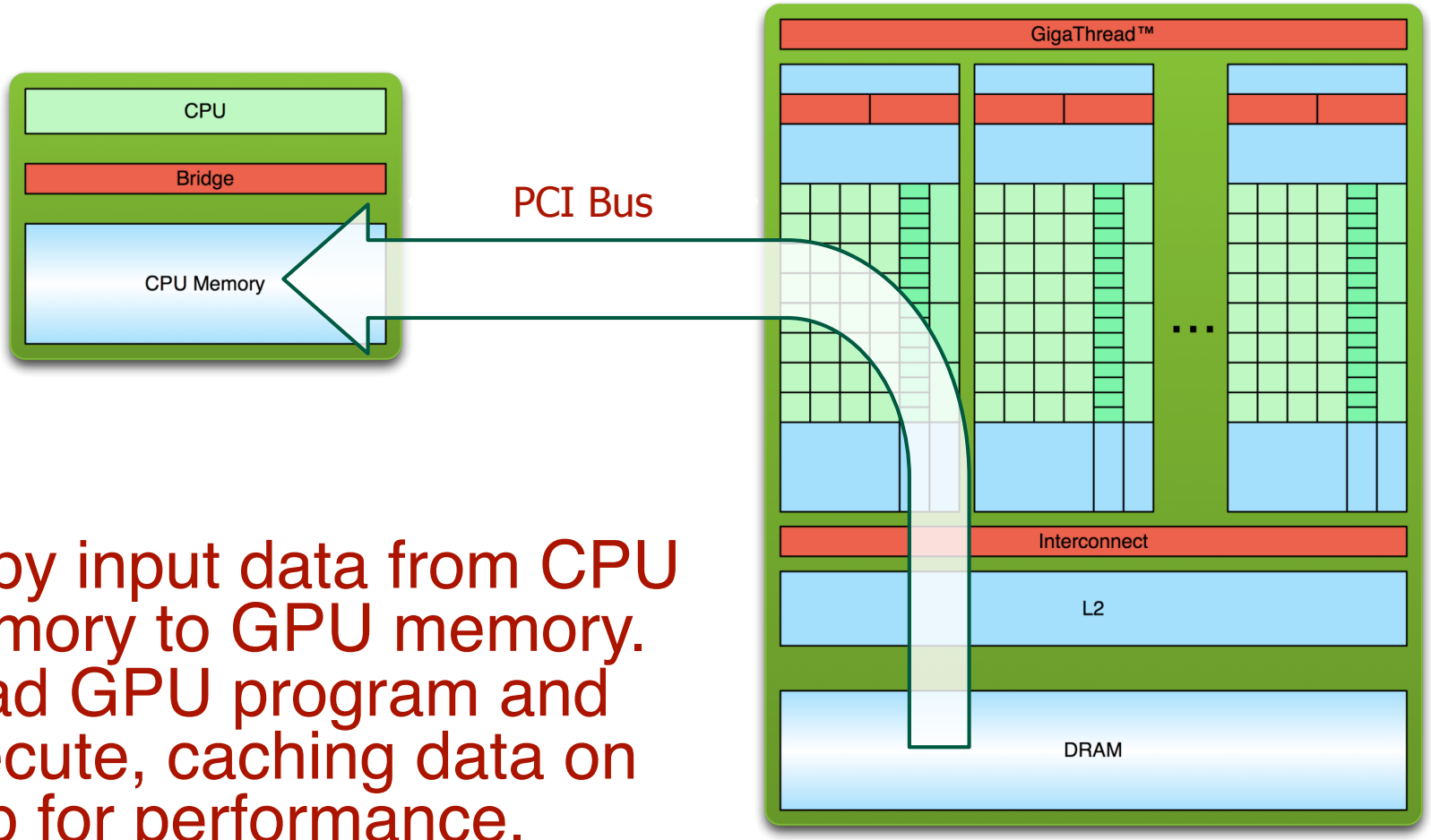
1. Copy input data from CPU memory to GPU memory.

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

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

Salida:

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

II. Architecture

NVIDIA

``... 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

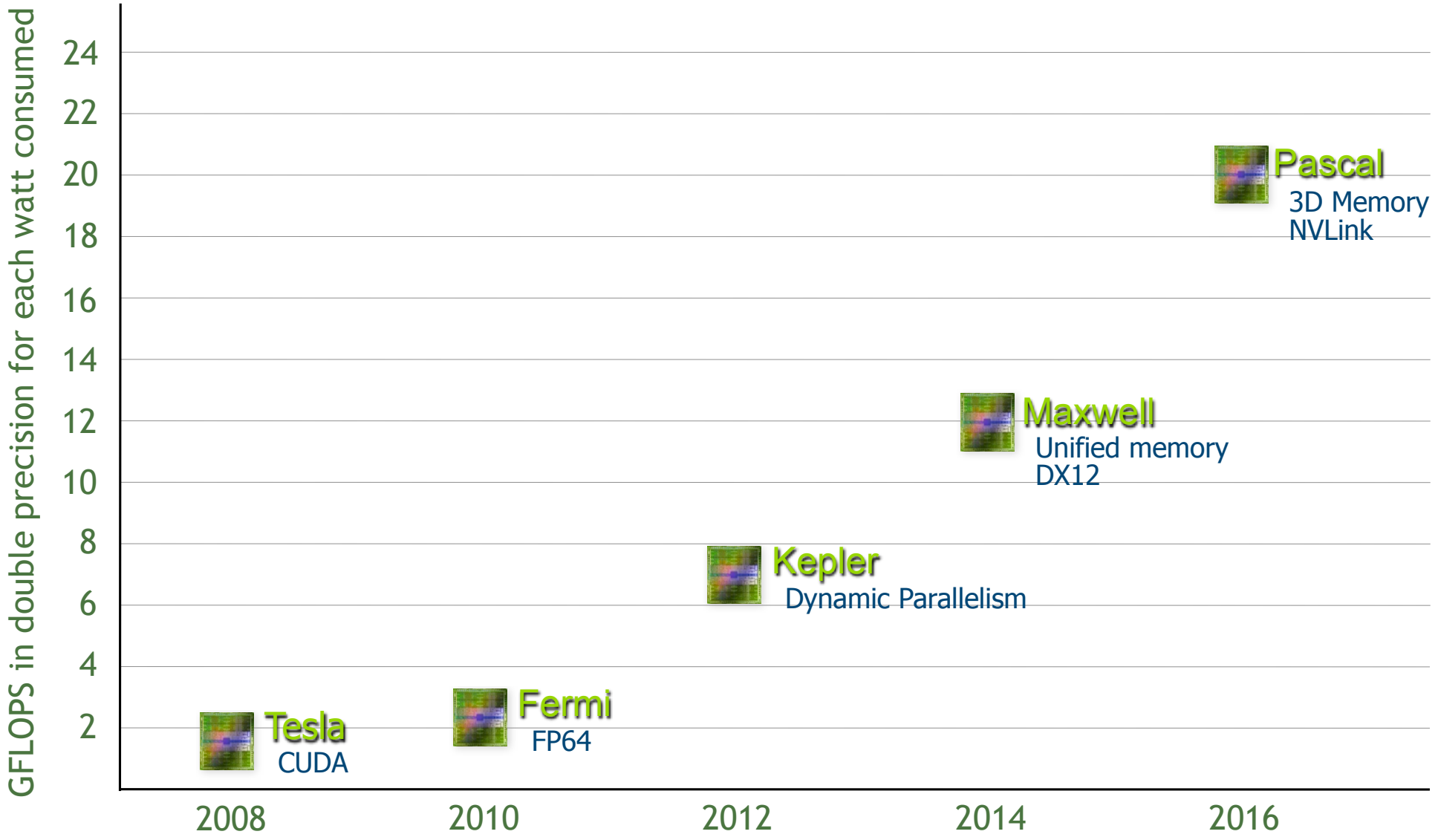
Mc-Graw-Hill (1995)

Chapter 9, page 569

II.1. CUDA hardware model

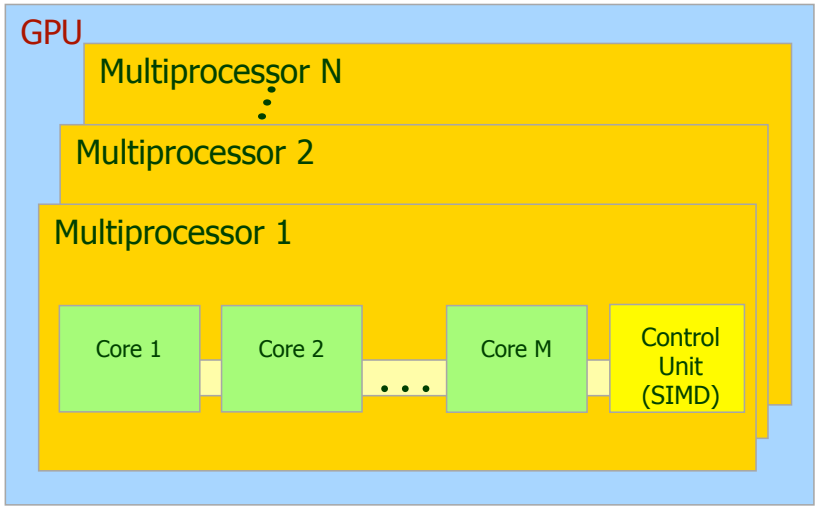
NVIDIA

Overview of CUDA hardware generations



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

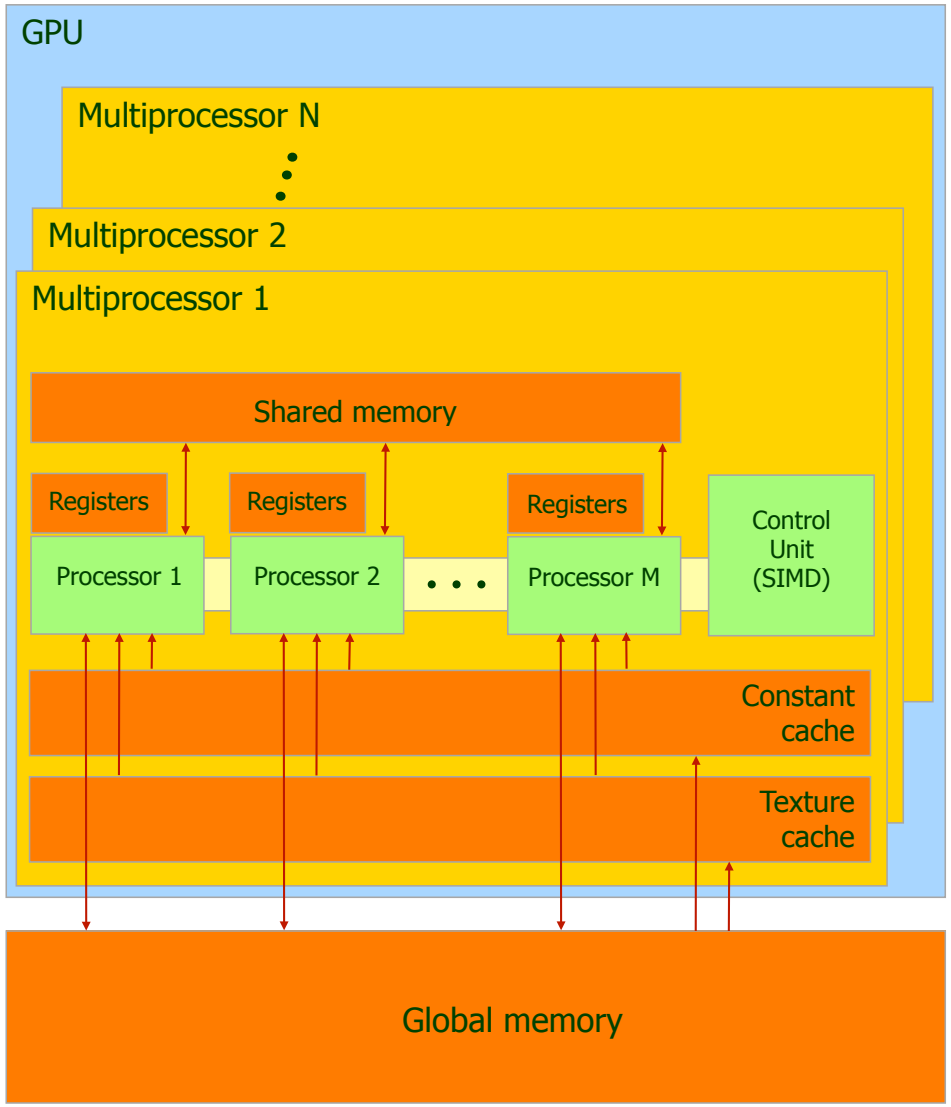
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)	GK110 (Kepler)
Time period	2006-07	2008-09	2010-11	2012-13
N (multiprocs.)	16	30	14-16	13-15
M (cores/multip.)	8	8	32	192
Number of cores	128	240	448-512	2496-2880

Memory hierarchy

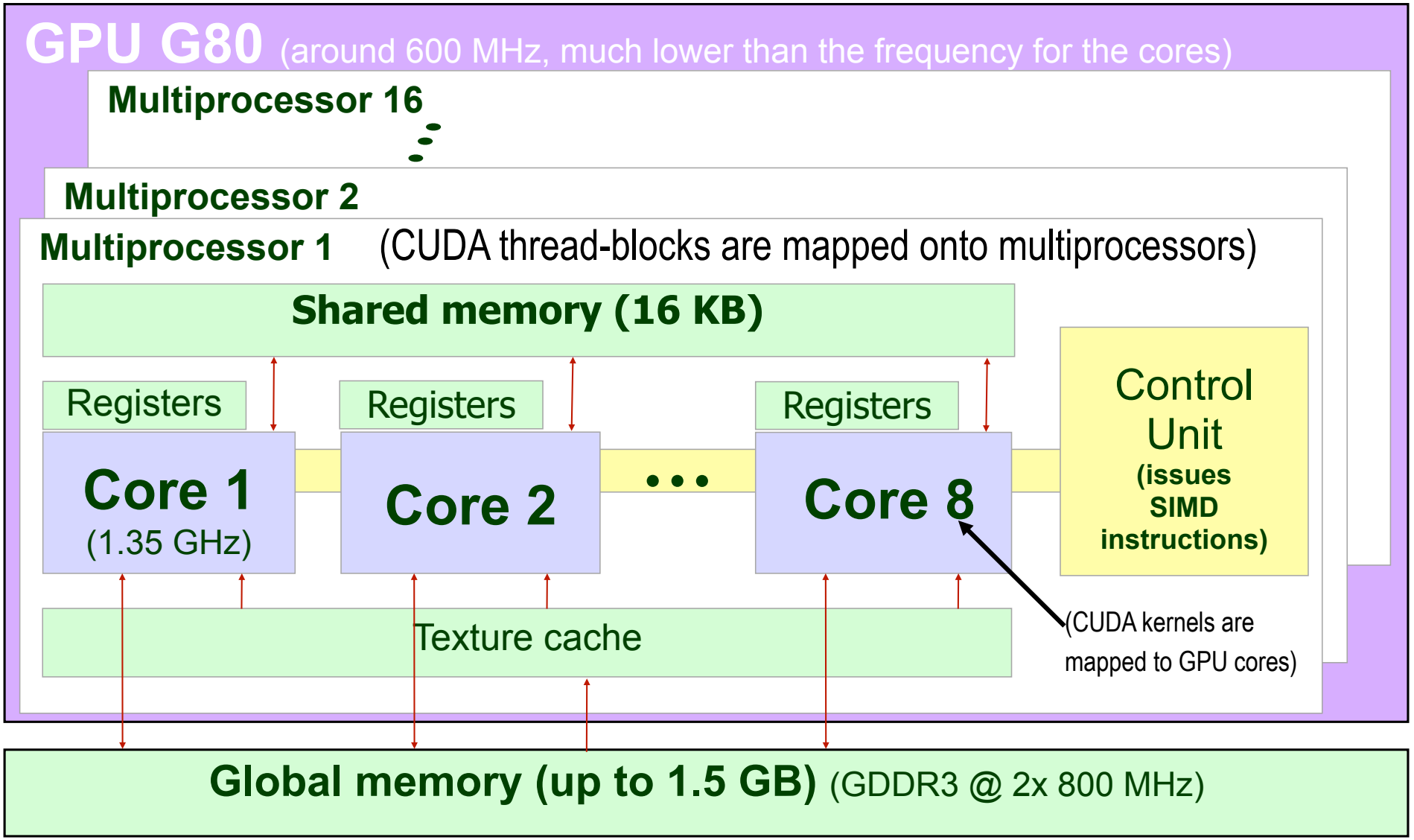
- Each multiprocessor has:
  - A register file.
  - Shared memory.
  - A constant cache and a texture cache, both read-only.
- Global memory is the actual video memory (GDDR5):
  - Three times faster than the DDR3 used by the CPU, but...
  - ... around 500 times slower than shared memory! (DRAM versus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



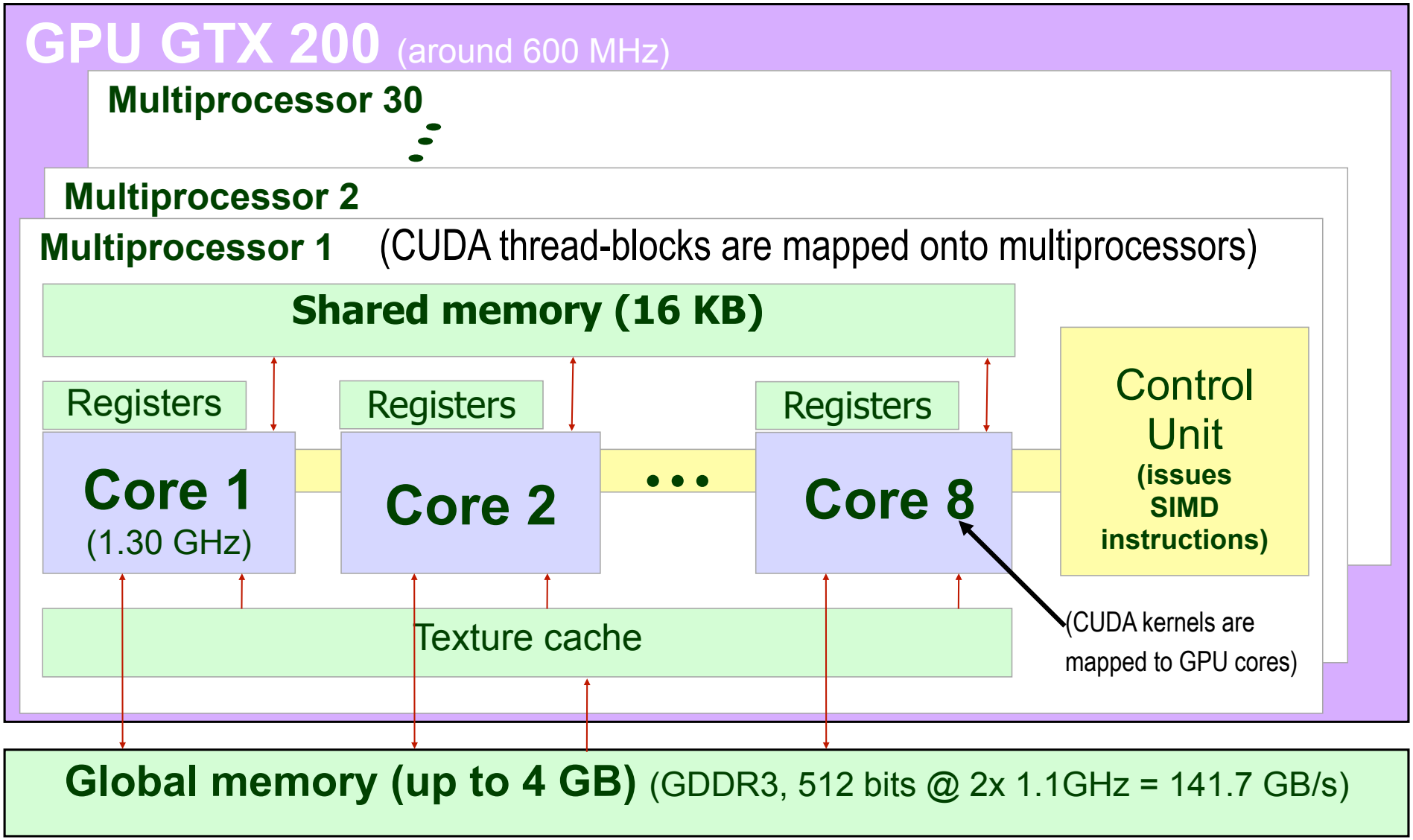
II.2. The first generation: Tesla (G80 and GT200)

NVIDIA

The first generation: G80 (GeForce 8800)

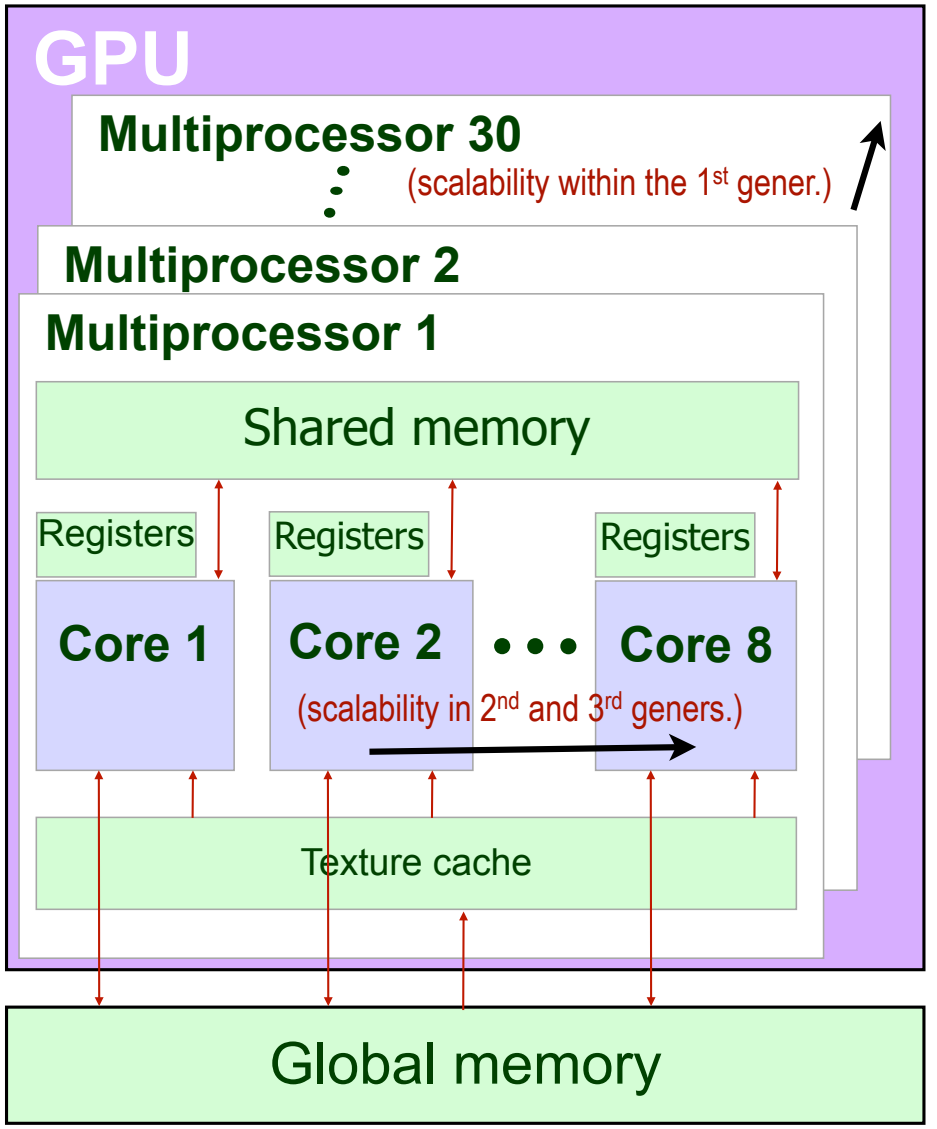


The first generation: GT200 (GTX 200)



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. The second generation: Fermi (GF1xx)

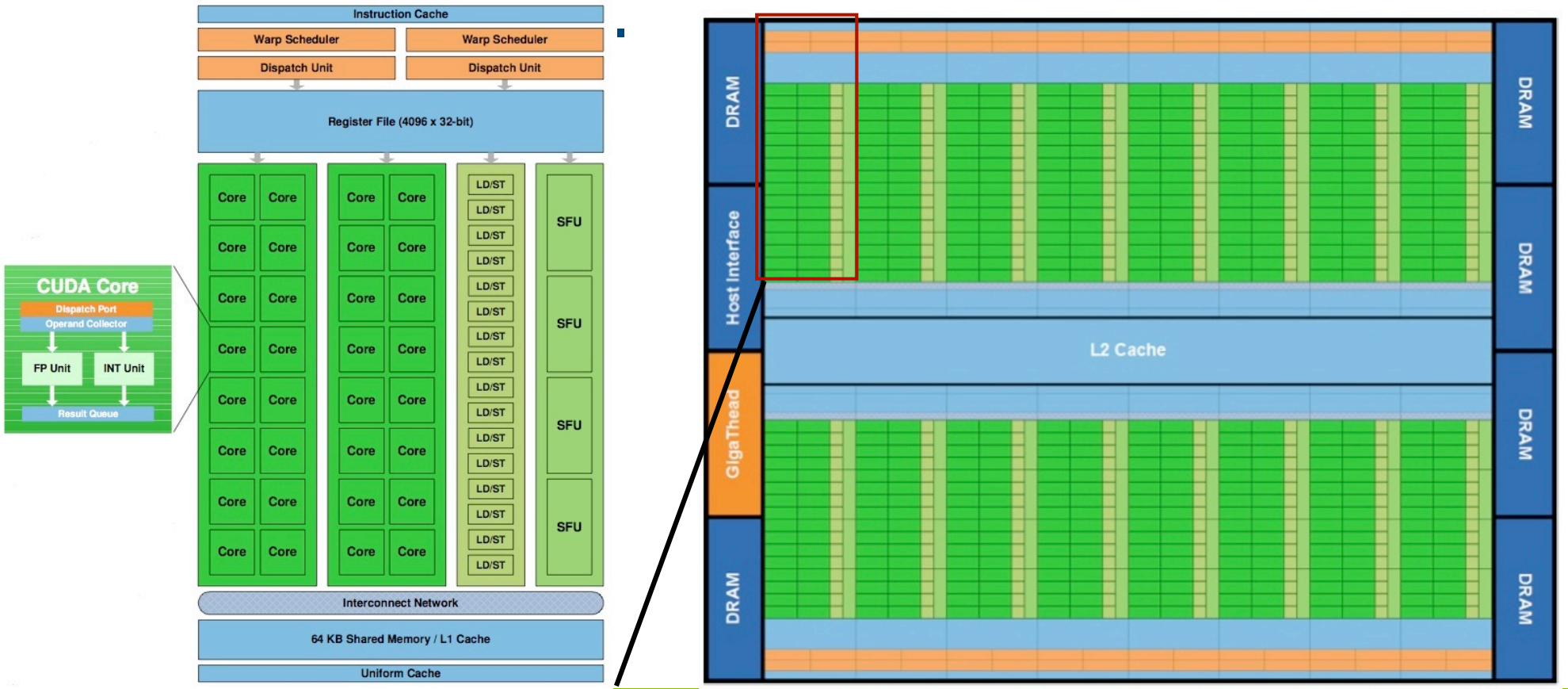
NVIDIA

Fermi hardware compared to its predecessors

GPU architecture	G80	GT200	GF100 (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

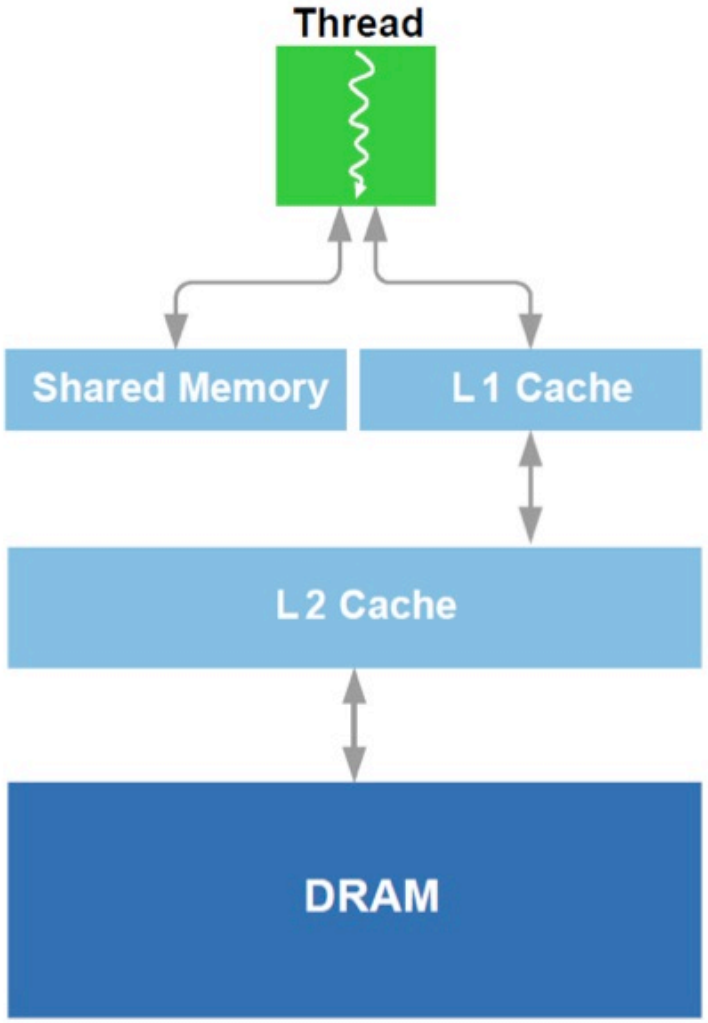
Fermi: An architectural overview

- 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.



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 conherence shared by all multiprocessors (SMs).

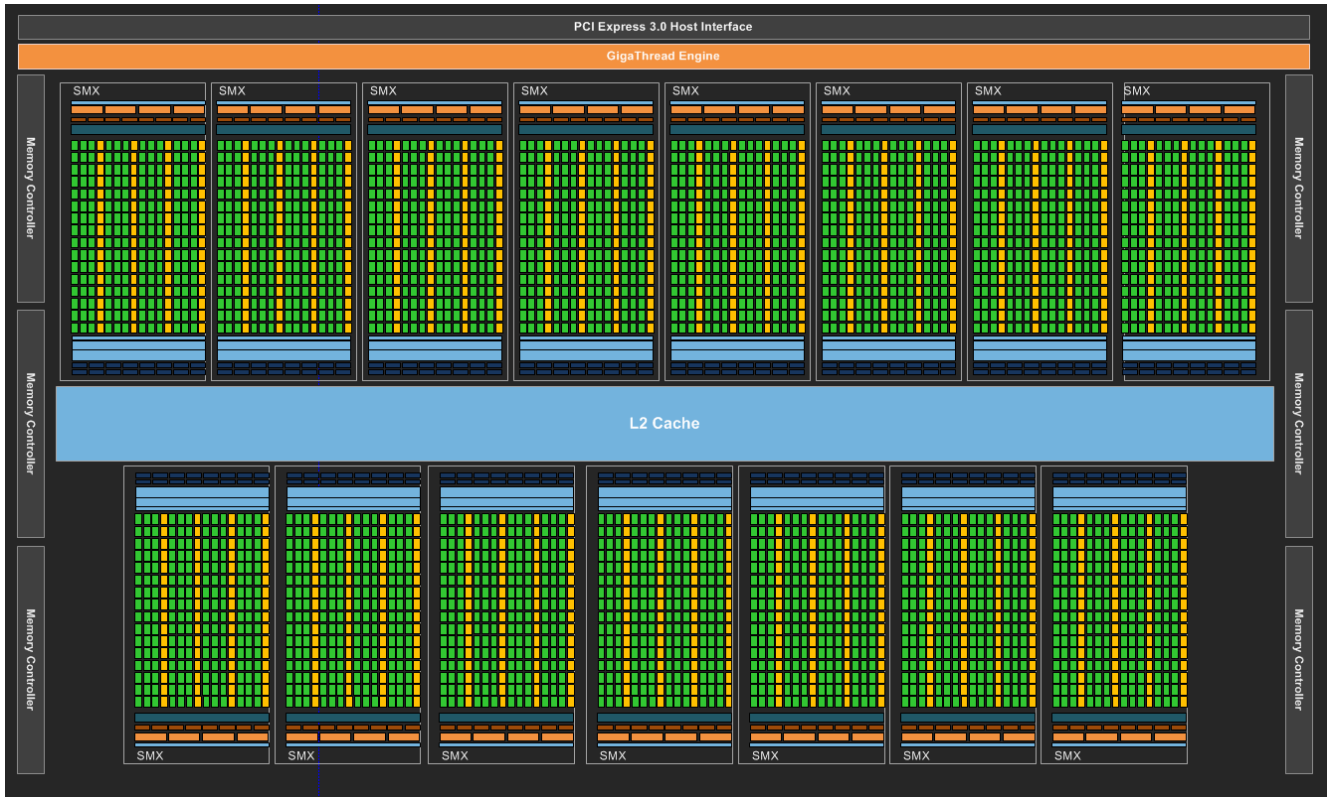


II. 4. The third generation: Kepler (GK1xx)

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.

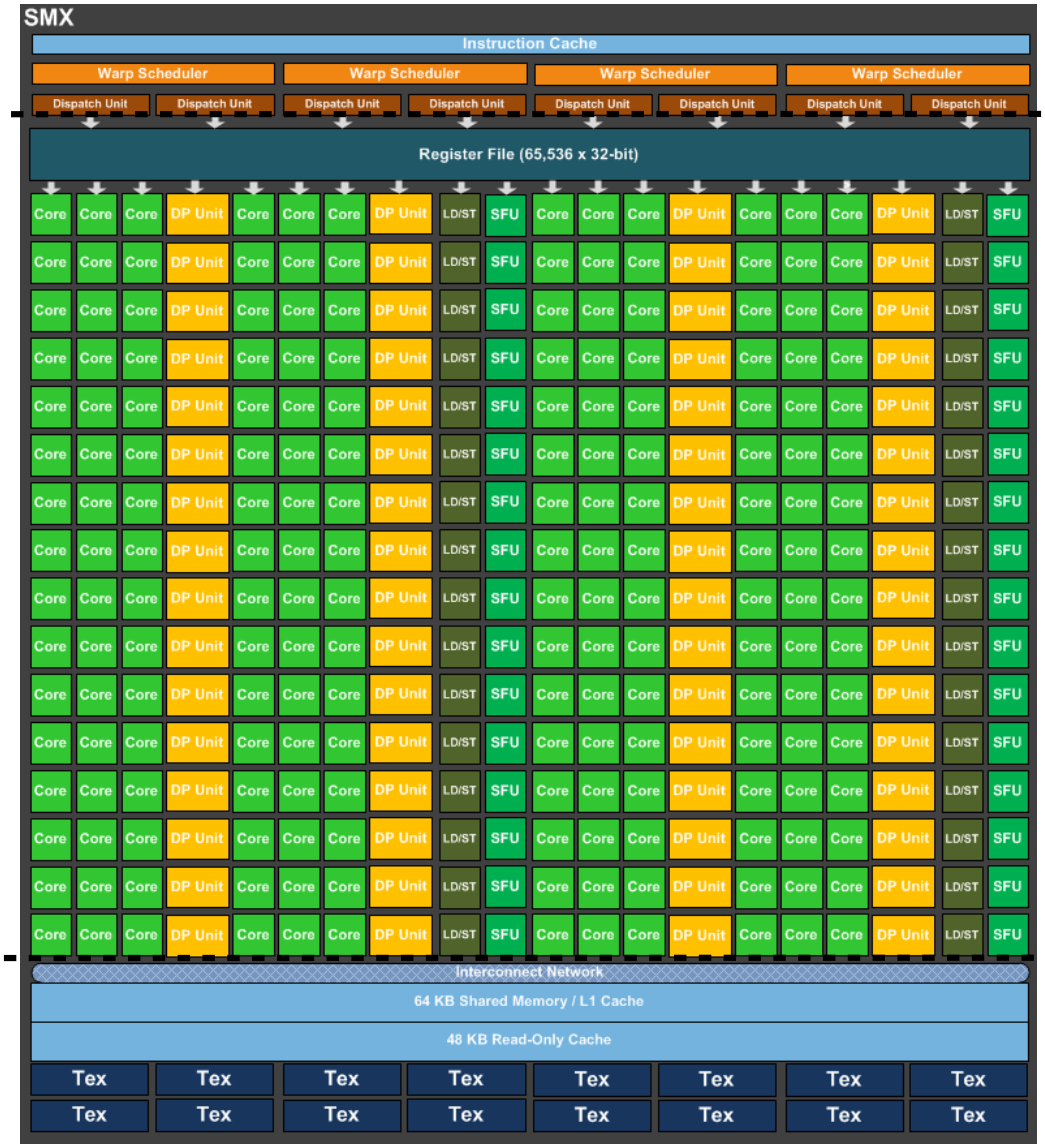


Multiprocessor evolution: From SMs in Fermi to SMXs in Kepler

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, ...)

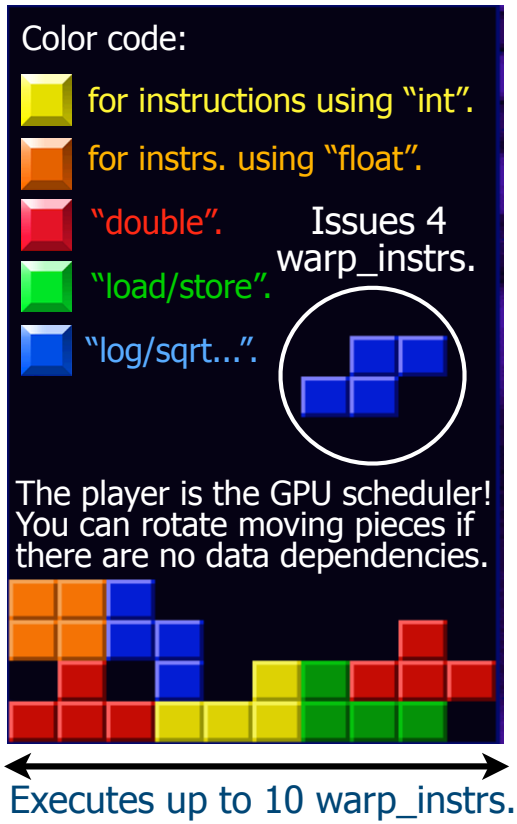
Back-end

Memory access

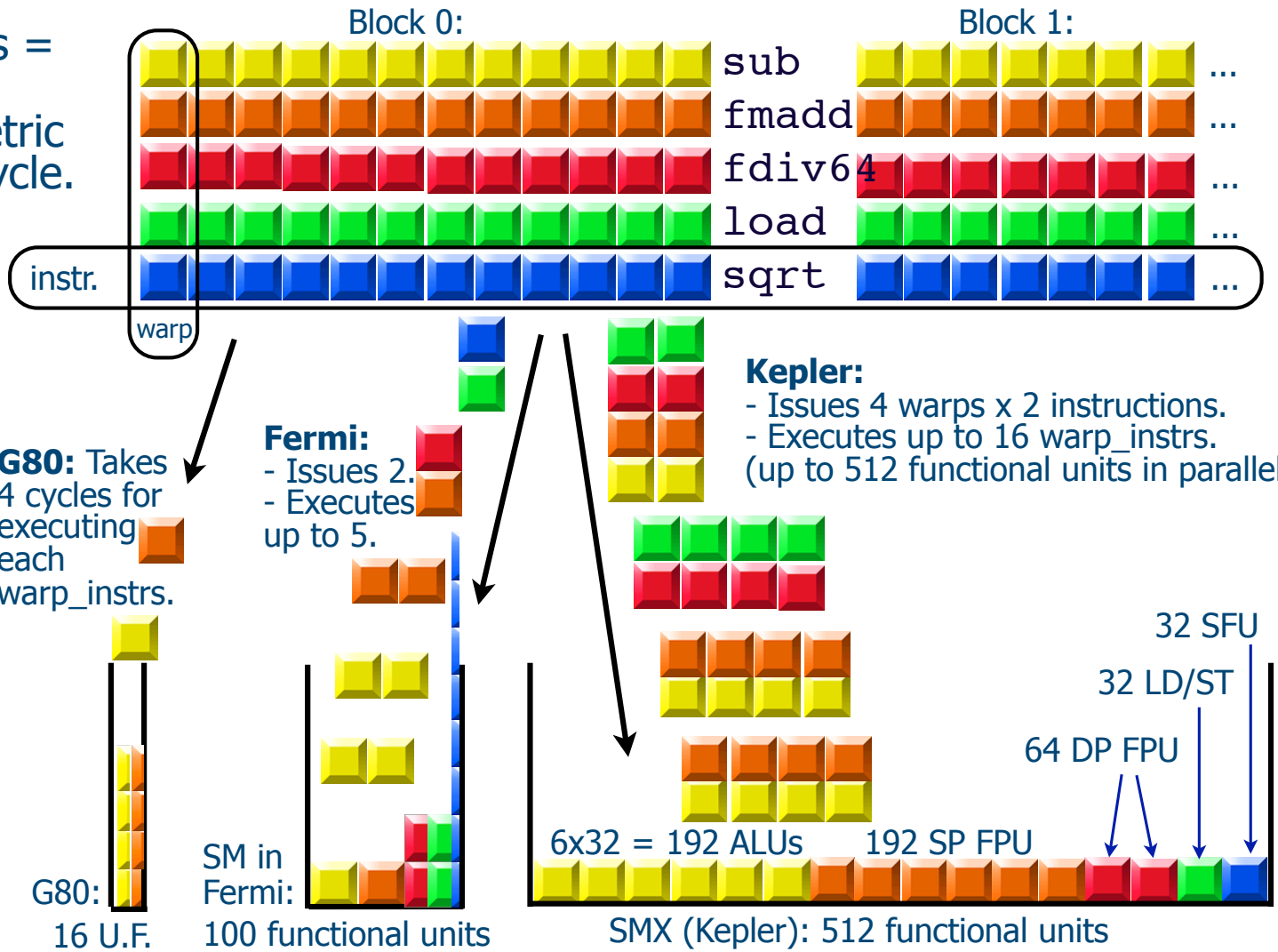
Interface

Express as much parallelism as possible: SMXs (Kepler) are wider than SMs (Fermi)

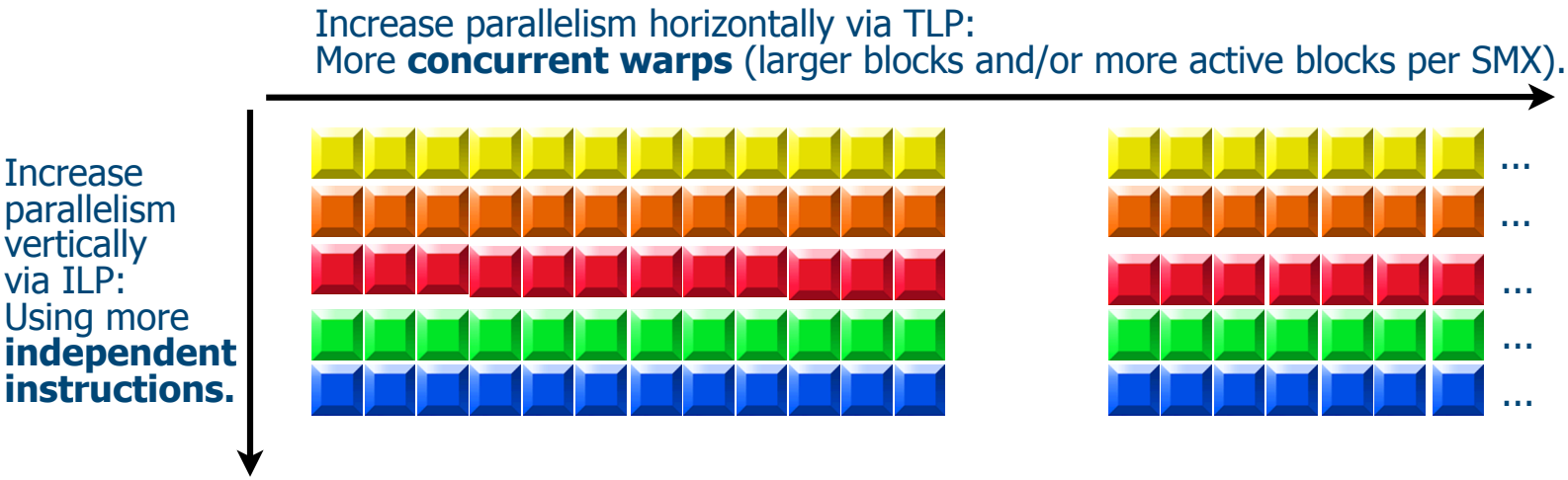
Tetris (tile = warp_instr.):
- Issues 4 warp_instrs.
- Executes up to 10 warps = 320 threads.
- Warp_instrs. are symmetric and executed all in one cycle.



Example: Kernel with blocks of 384 threads (12 warps).

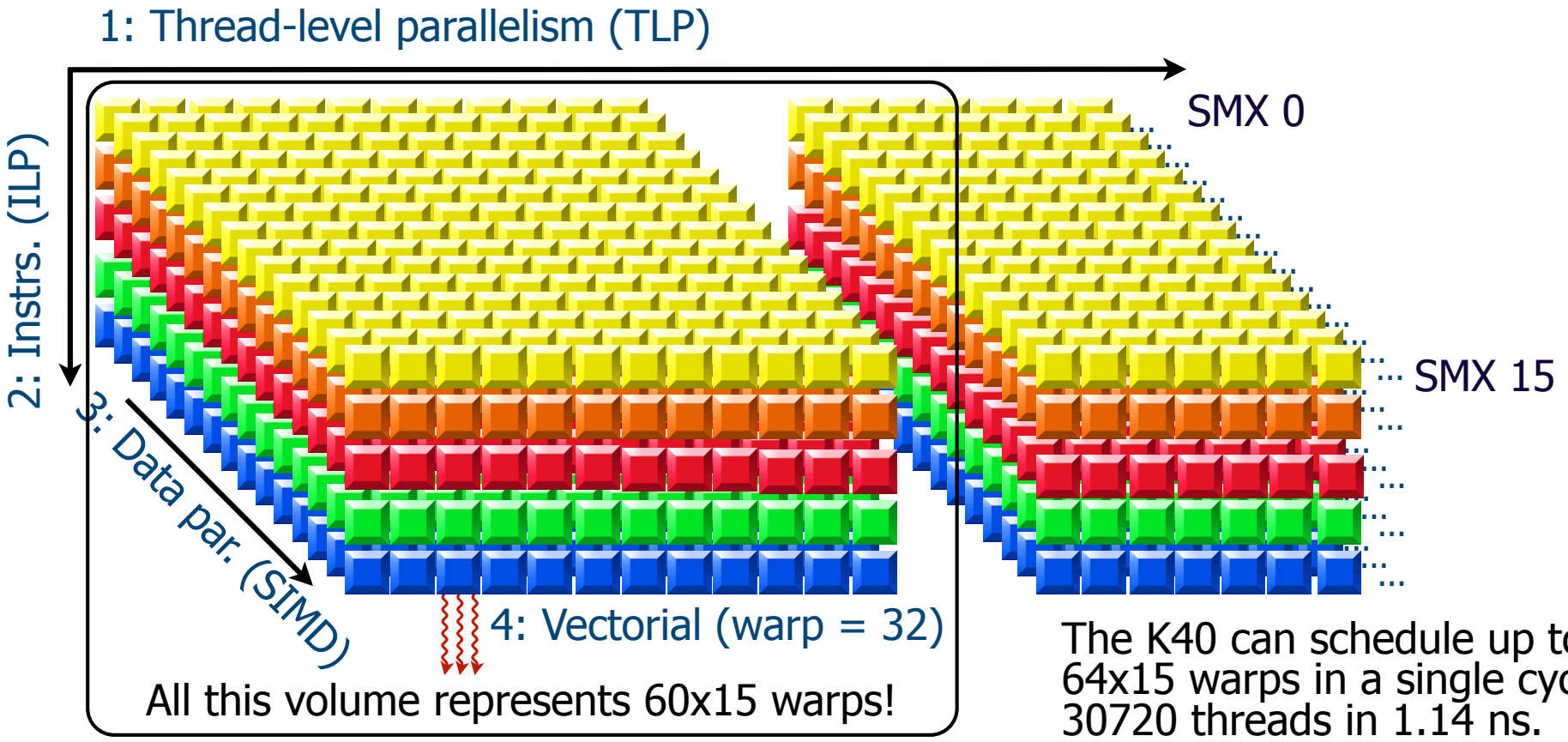


Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP)



- SMXs can leverage available ILP interchangeably with TLP:
  - It is much better at this than Fermi.
- Sometimes is easier to increase ILP than TLP (for example, a small loop unrolling):
  - # of threads may be limited by algorithm or hardware limits.
- We need ILP for attaining a high IPC (Instrs. Per Cycle).

Kepler GPUs can hold together **all** forms of parallelism. Example: K40.



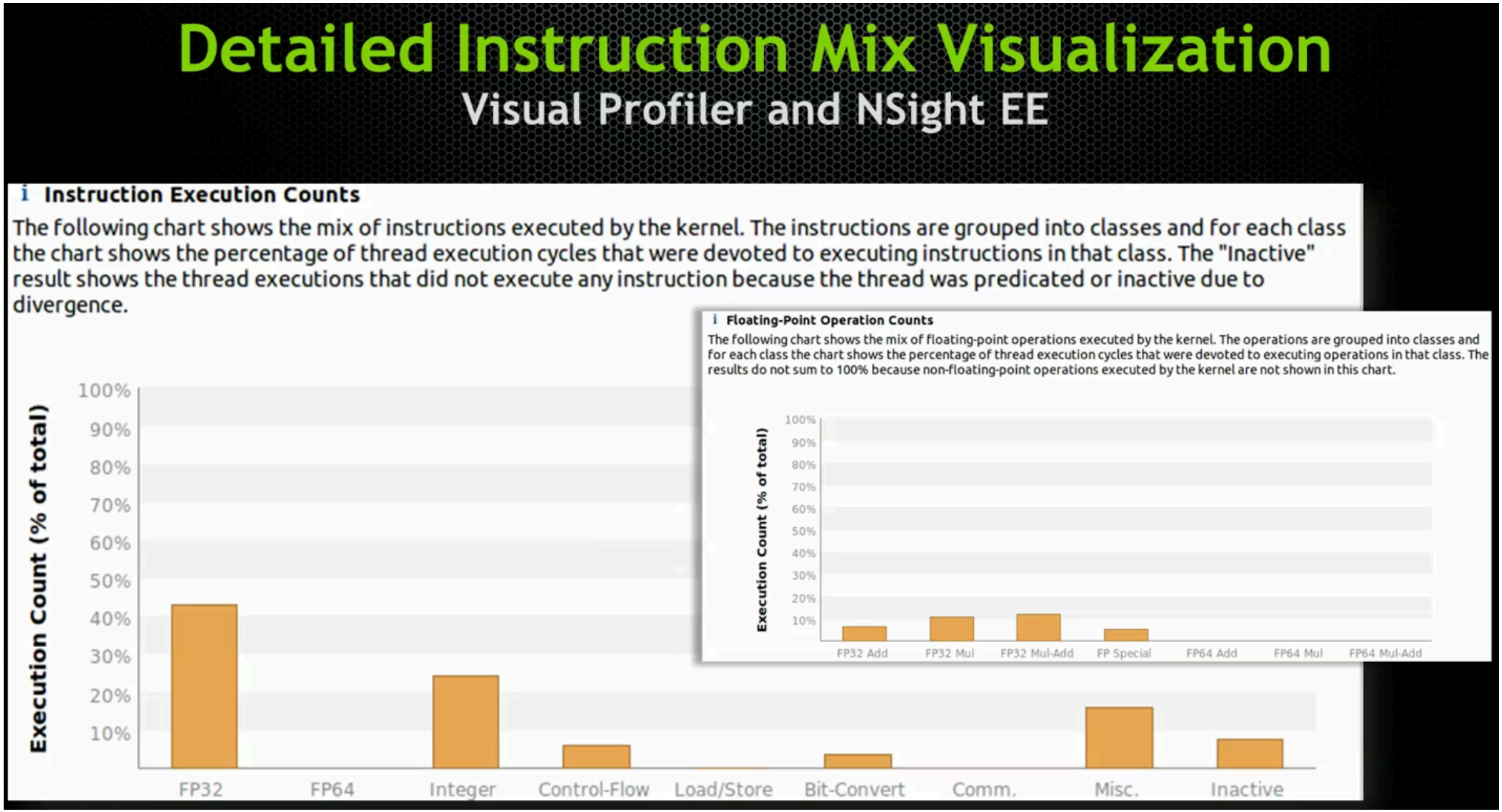
- Imagine a 3D tetris with 15 boxes and up to 64 pieces falling down simultaneously on each of them, because that is the way K40 works when all parallelism is deployed.

Case study: Zernike moments

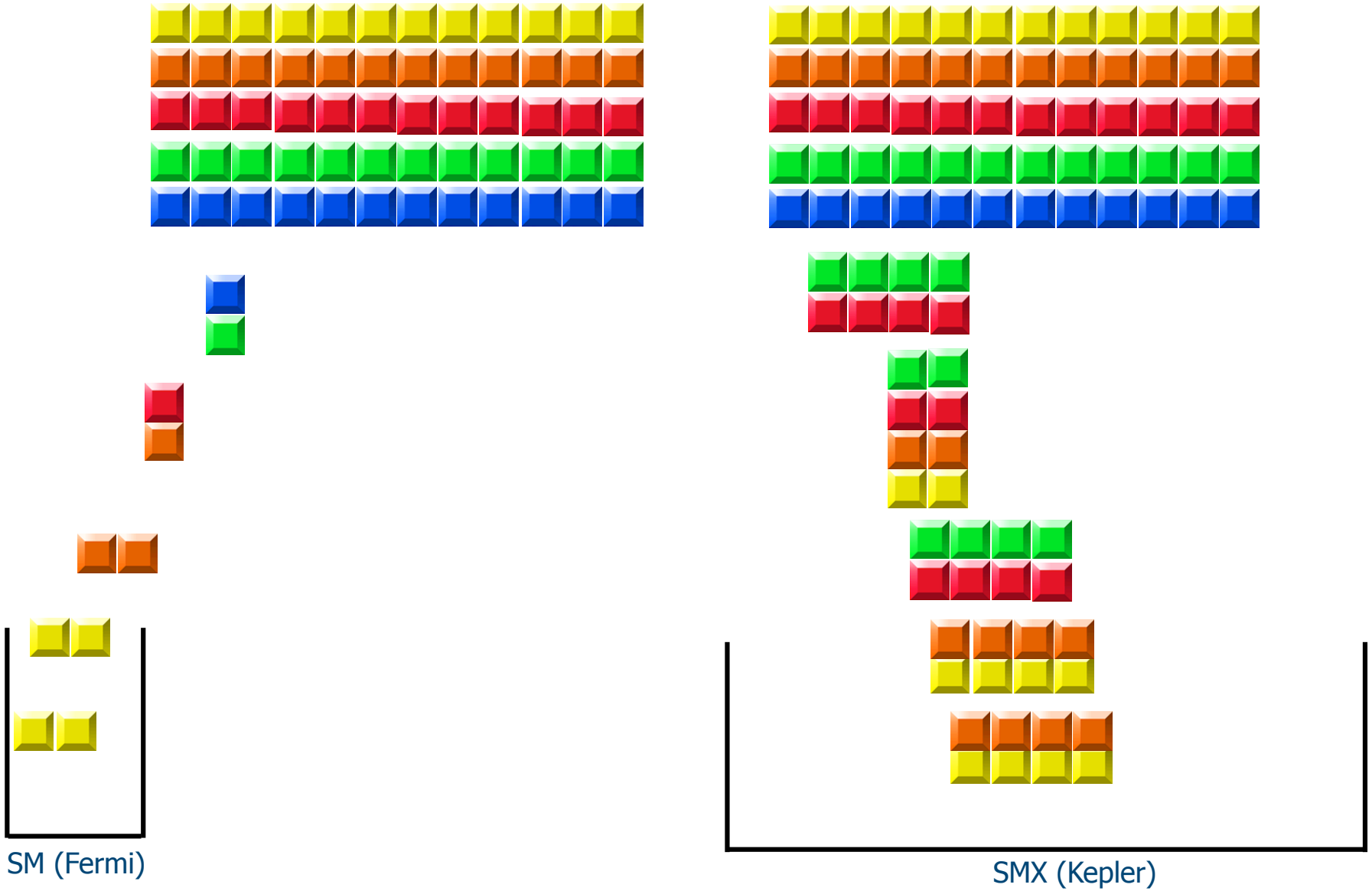
GPU resources	ALU	32-bits FPU	64-bits FPU	Load/store	SFU
Fermi	32%	32%	16%	16%	4%
Kepler	37.5%	37.5%	12.5%	6.25%	6.25%
Kernel for Zernike	54%	21%	0%	25%	0%
Better	Kepler	Fermi	Kepler	Fermi	Fermi

- Fermi is more balanced in this case.
- With the resources distribution in Kepler, the execution of integer arithmetic improves, but the floating-point arithmetic and the load/store worsens. All the others are not used.

Use the CUDA Visual Profiler to know how good your application adapts to resources

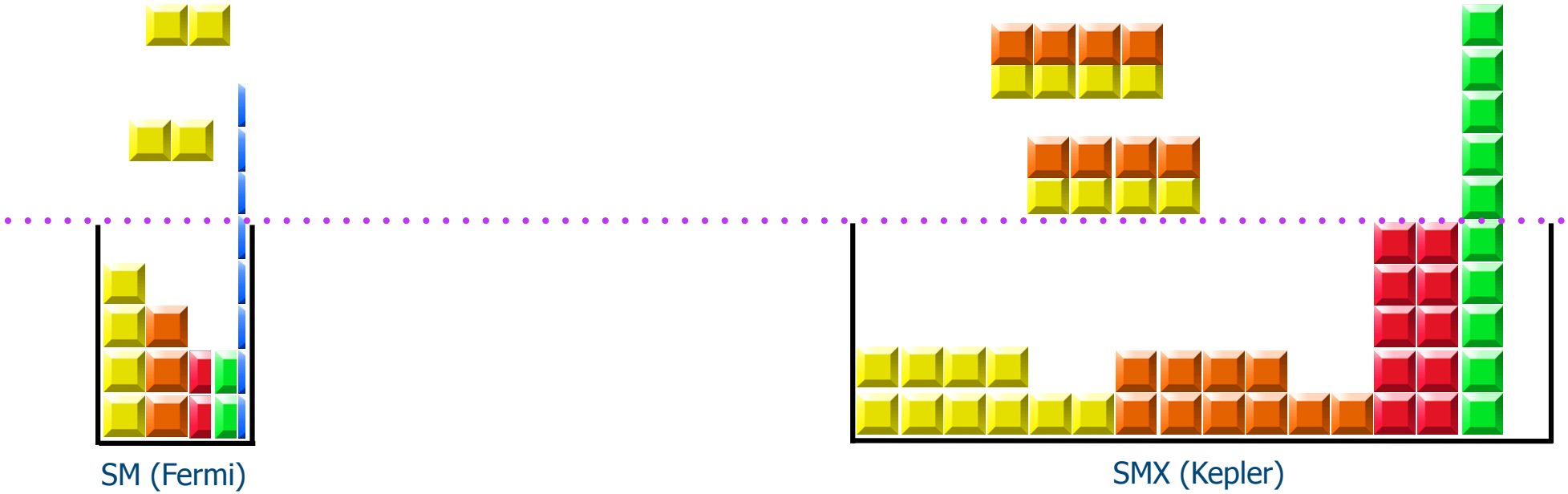


The way the GPU front-end works: (1) How warps are scheduled



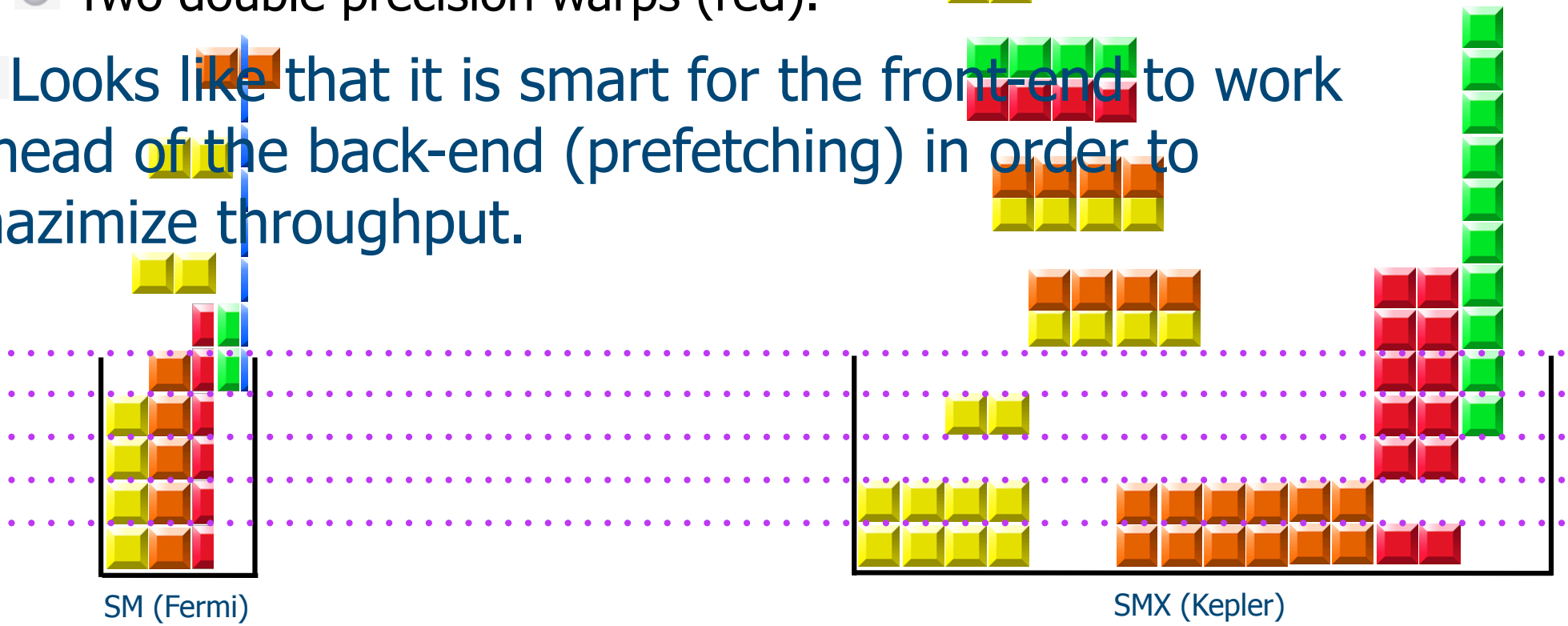
The interface between front-end & back-end: (2) How warps are issued

- In the 5 cycles shown, we could have executed all this work.
  - In Fermi, there is a deficit in SFUs (blue), whereas in Kepler, the deficit goes to load/store units (green).
  - Kepler balances double precision (red) and has a good surplus in "int" and "float" computations, an evidence that real codes have more presence of orange and, overall, yellow instructions.



The way the GPU back-end works: (3) Warps execution

- Let us assume that when we start the execution there are few warps pending to be executed:
  - Two single precision warps (orange).
  - Two double precision warps (red).
- Looks like that it is smart for the front-end to work ahead of the back-end (prefetching) in order to mazimize throughput.

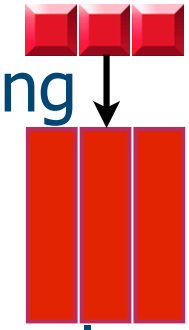


Some remarks about the "tetris" model

- In Fermi, red tiles are not allowed to be combined with others.

- In Kepler, we cannot take 8 warp_instrs. horizontally, bricks must have a minimum height of 2.

- Instructions must have different latency, so those consuming more than one cycle (i.e. double precision floating-point) should expand vertically.



- If the warp faces divergencies, it will take more than one cycle. We can represent that case similarly to the one above.
- Codes usually have more yellow tiles ("int" predominates).
- Some bricks are not complete because the scheduler cannot find 4x2 structures free of dependencies.

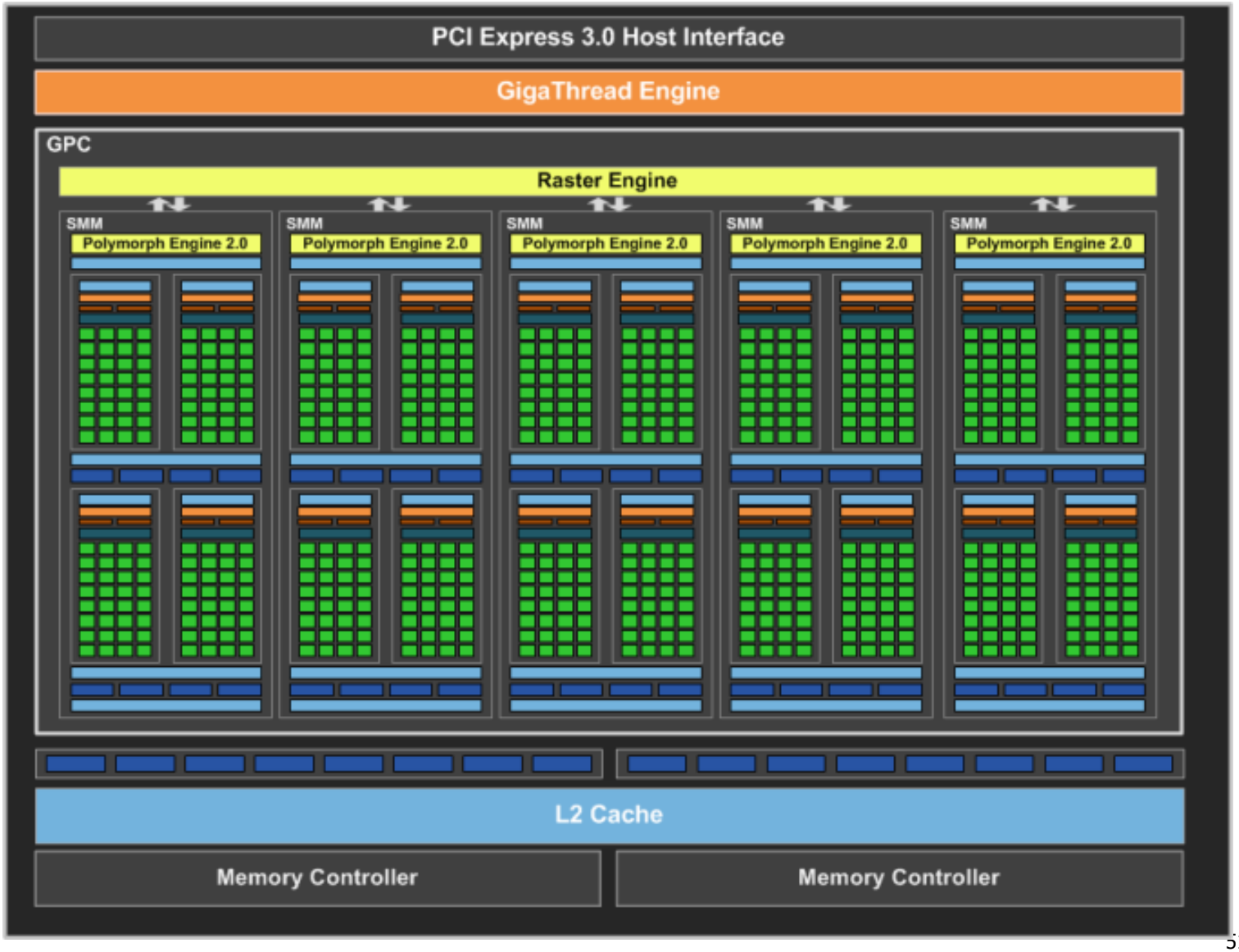
- Bricks can assemble non-contiguous tiles.

II. 5. The fourth generation: Maxwell (GM1xx)

NVIDIA

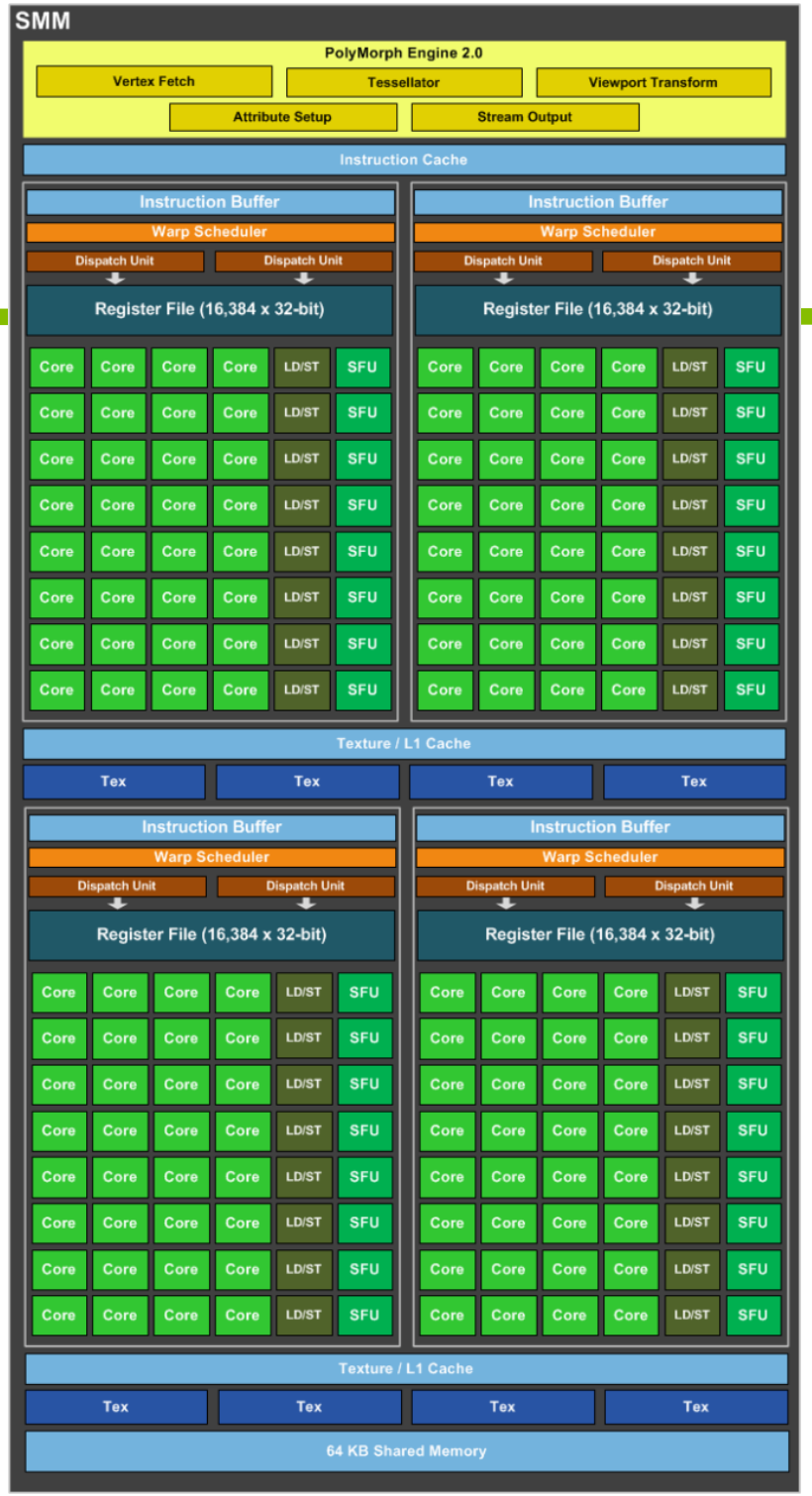
Maxwell and SMM multiprocessors (for GeForce GTX 750 Ti, GM107 with 5 SMM)

- 1870 Mt.
- 148 mm^2.



The SMMs

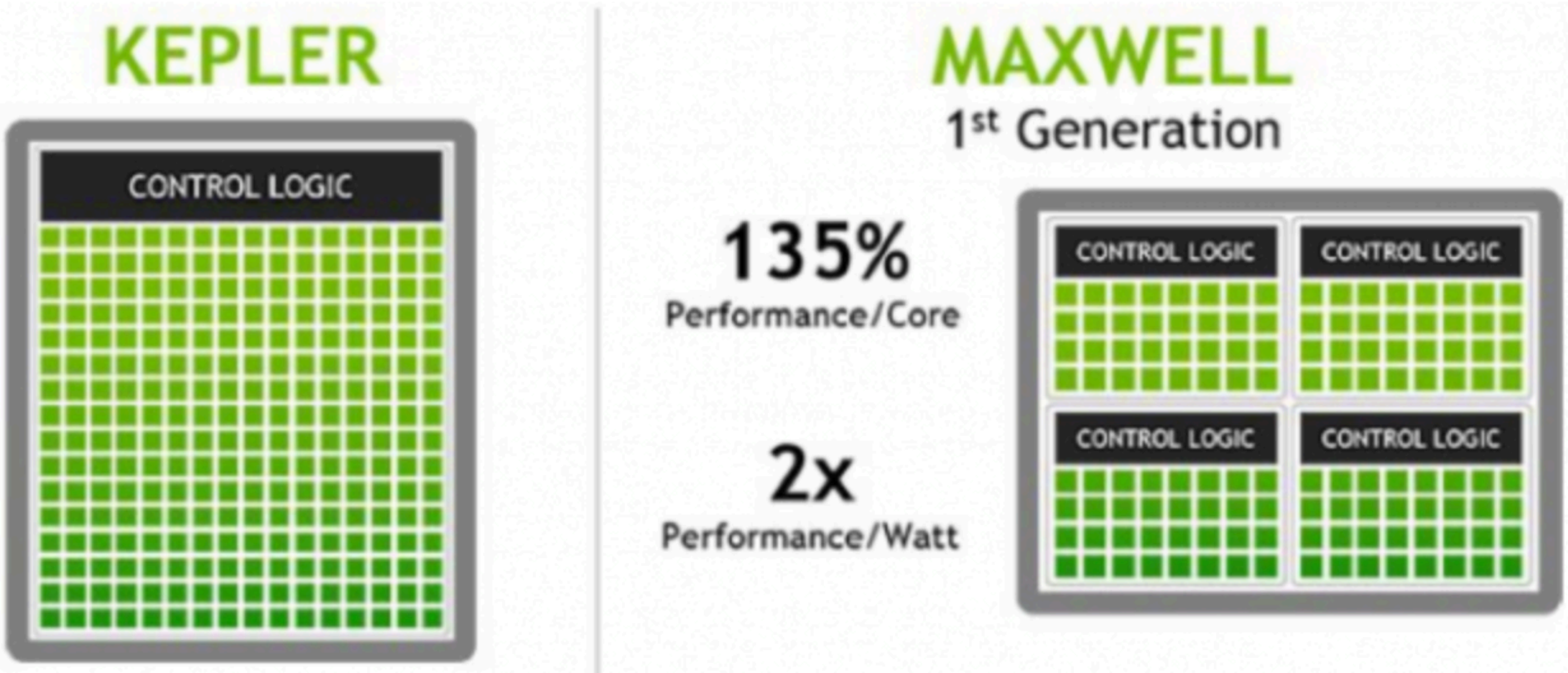
- Keep the same 4 warp schedulers, and the same LD/ST and SFU units.
- Reduce the number of cores for `int` and `float`: from 192 to 128 units.



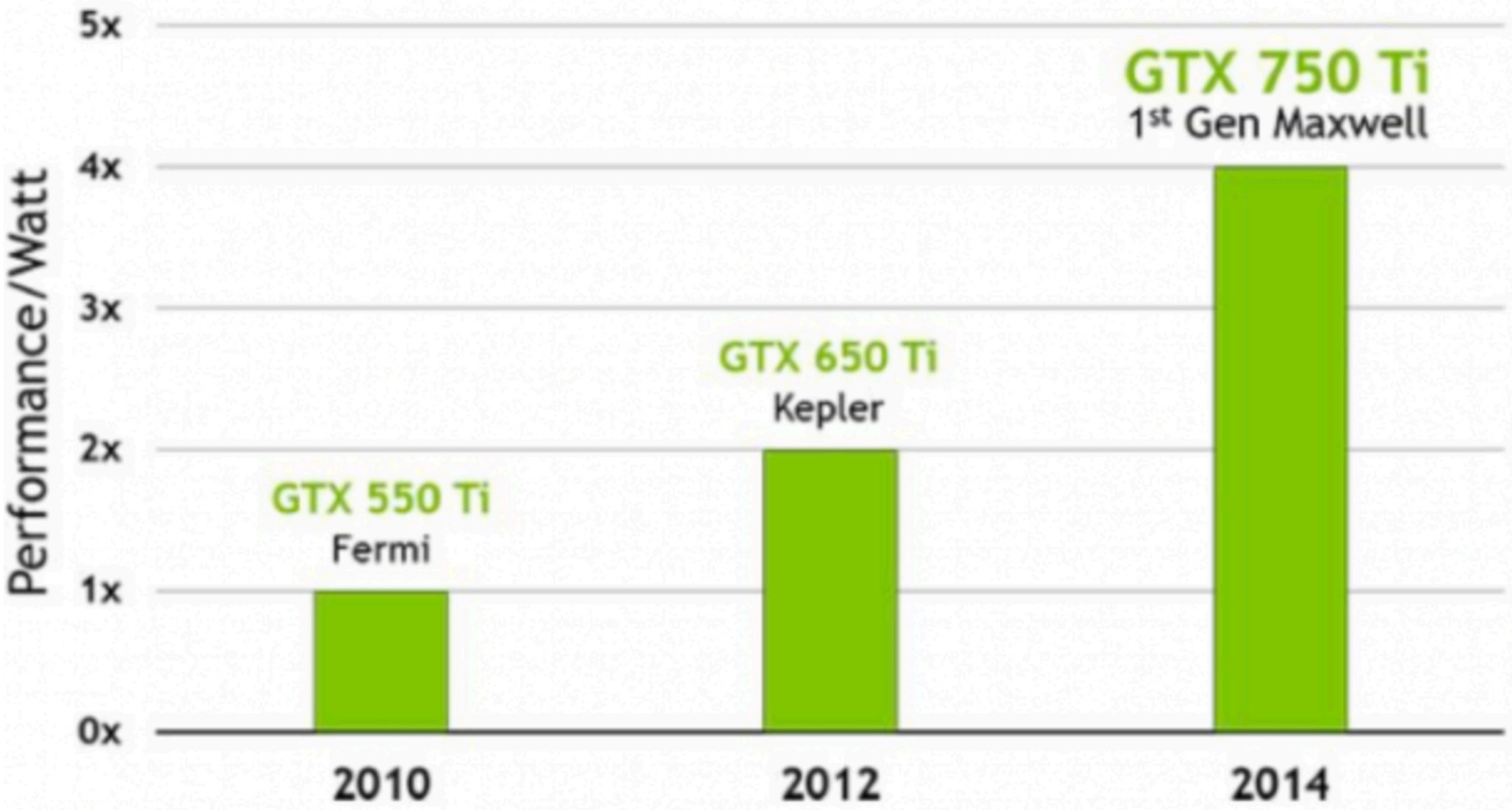
Some commercial models on 28 nm.

	GeForce GTX 650	GeForce GTX 650 Ti	GeForce GTX 750 Ti	GeForce GTX 660
GPU	GK107	GK106	GM107	GK106
Architecture	Kepler	Kepler	Maxwell	Kepler
Multiprocessors	2 SMX	4 SMX	5 SMM	5 SMX
Number of cores	192 x 2 = 384	192 x 4 = 768	128 x 5 = 640	192 x 5 = 960
Frequency of cores	1058 MHz	925 MHz	1020 - **1085 MHz**	980 - 1033 MHz
DRAM bus width	128 bits	128 bits	128 bits	192 bits
DRAM frequency	2x 2500 MHz	2x 2700 MHz	2x 2700 MHz	2x 3000 MHz
RAM bandwidth	80 Gbytes/s.	86.4 Gbytes/s.	86.4 Gbytes/s.	144 Gbytes/s.
GDDR5 memory size	1 or 2 Gbytes	1 or 2 Gbytes	1 or 2 Gbytes	2 Gbytes
Power connector	6 pins	6 pins	**None**	6 pins
Maximum TDP	64 W.	110 W.	**60 W.**	140 W.
Million transistors	1300	2540	1870	3175 mm2 ?
Die size	118 mm2	221 mm2	148 mm2	275 mm2 ?
Approx. cost (2 GB)	100 €	110 €	110 €	150 €

Major enhancements



Power efficiency



II. 6. A summary of four generations

NVIDIA

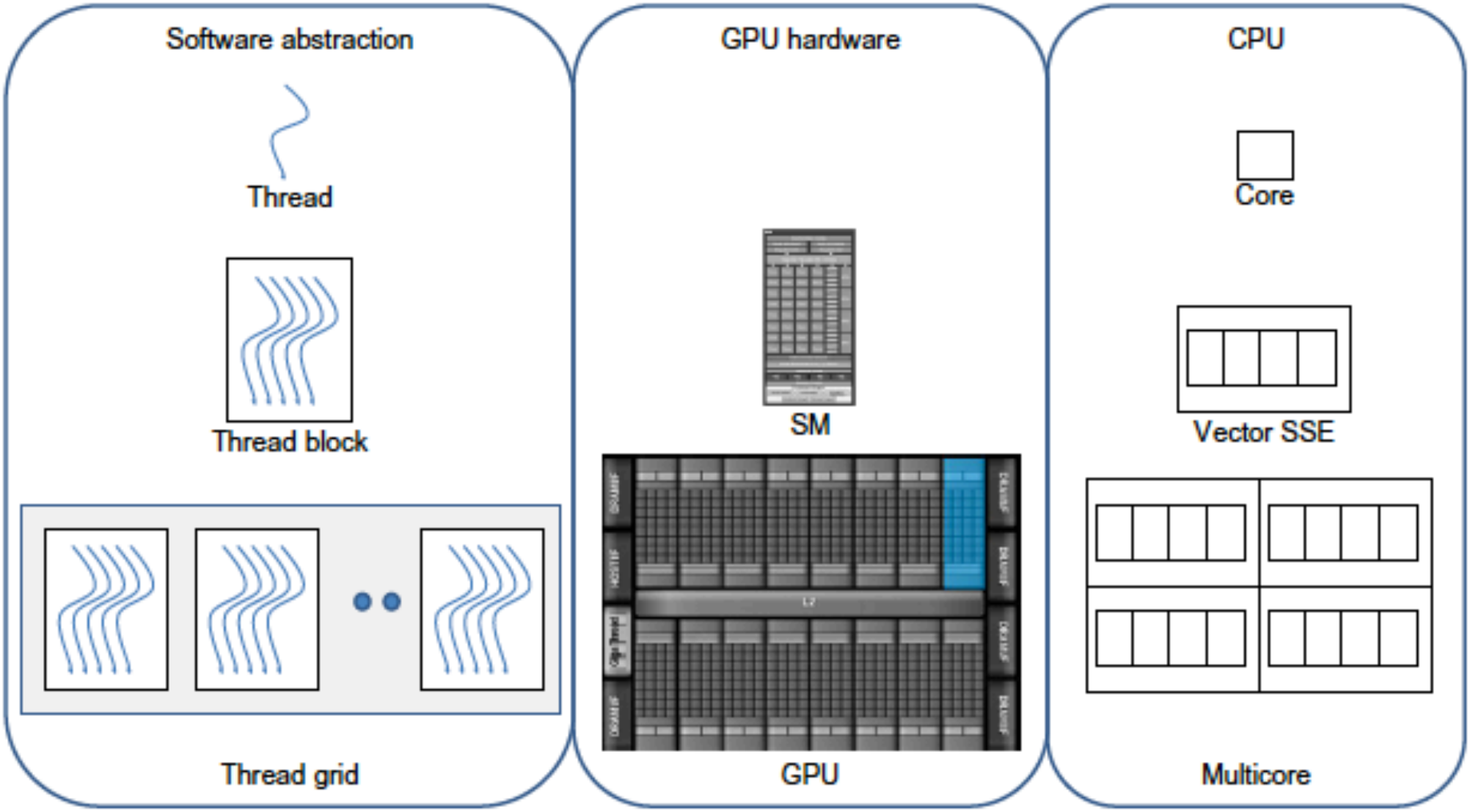
Scalability for the architecture: A summary of four generations

	Tesla		Fermi		Kepler				Maxwell	
Architecture	**G80**	**GT200**	**GF100**	**GF104**	**GK104 (K10)**	**GK110 (K20X)**	**GK110 (K40)**	**GK110 (K80)**	**GM107 (GTX760)**	**GM204 (GTX980)**
Time frame	**2006 /07**	**2008 /09**	**2010**	**2011**	**2012**	**2013**	**2013 /14**	**2014 /15**	**2014 /15**	**2014 /15**
CUDA Compute Capability	**1.0**	**1.2**	**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

III. Programming

NVIDIA

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.

The model instanciates over few features to produce the commercial catalog

- We may expect higher differences for these features between models associated to different generations.
- Differences will also grow when graphics cards aim to different ends:
  - $300-500 high-end graphics card.
  - $150-250 mid-end.
  - $60-120 low-end.
- Video memory may also differ when a new technology emerges. Last step forward: GDDR5 vs. GDDR3.
- Graphical features are different too, but that is out of the scope of this tutorial.

Structure of a CUDA program

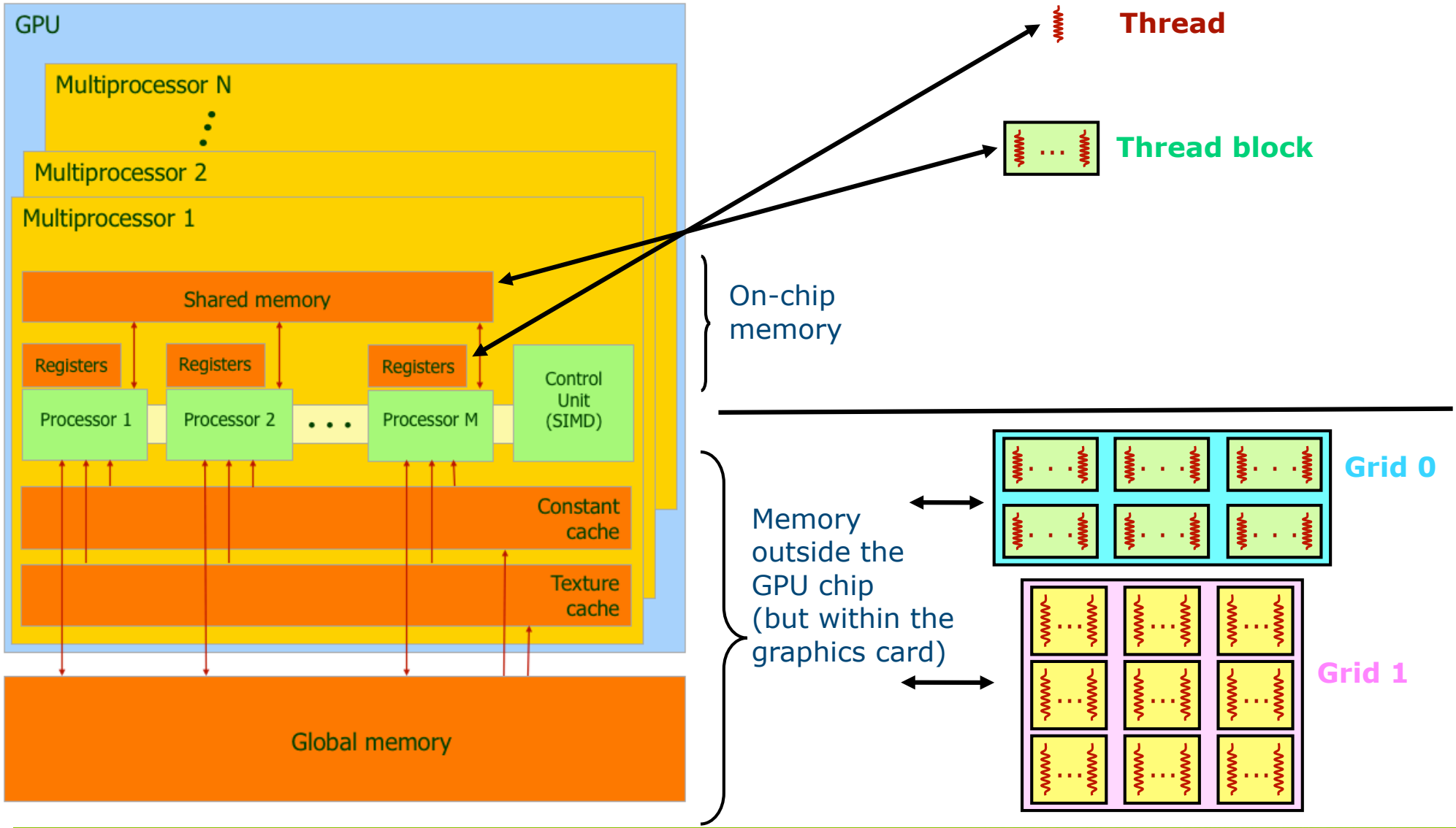
- Each multiprocessor (SM) processes batches of blocks one after another.
  - Active blocks = blocks processed by one multiprocessor in one batch.
  - Active threads = all the threads from the active blocks.
- Registers and shared memory within a multiprocessor are split among the active threads. Therefore, 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.

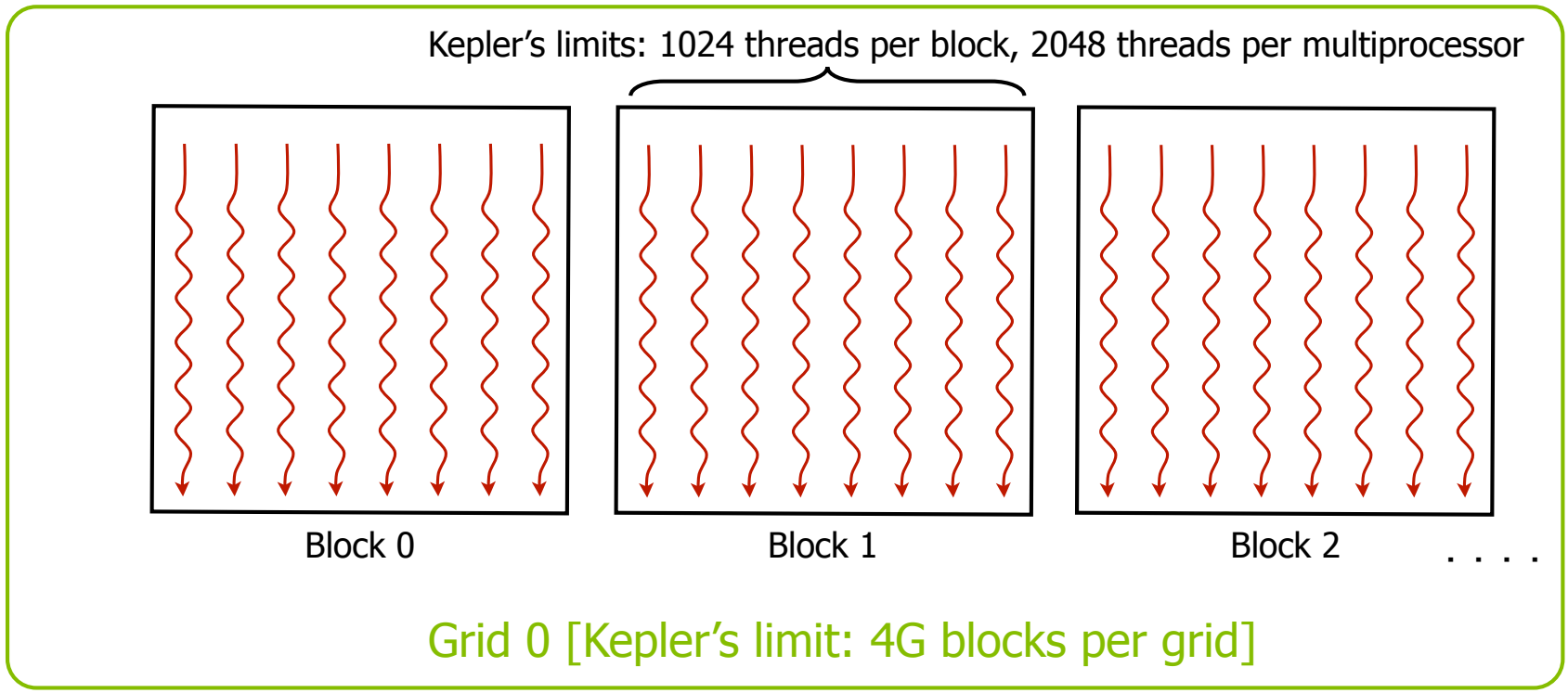
The relation between hardware and software from a memory access perspective



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		
Multiprocessors / GPU	16	30	14-16	13-16	4, 5, ...	Hardware	Scalability
Cores / Multiprocessor	8	8	32	192	128	Hardware	Scalability
Threads / Warp	32	32	32	32	32	Software	Throughput
Blocks / Multiprocessor	8	8	8	16	32	Software	Throughput
Threads / Block	512	512	1024	1024	1024	Software	Parallelism
Threads / Multiprocessor	768	1 024	1 536	2048	2048	Software	Parallelism
32 bits registers / Multip.	8K	16K	32K	64K	64K	Hardware	Working set
Shared memory / Multip.	16K	16K	16K 48K	16K, 32K, 48K	64K, 96K	Hardware	Working set

GPU threads and blocks

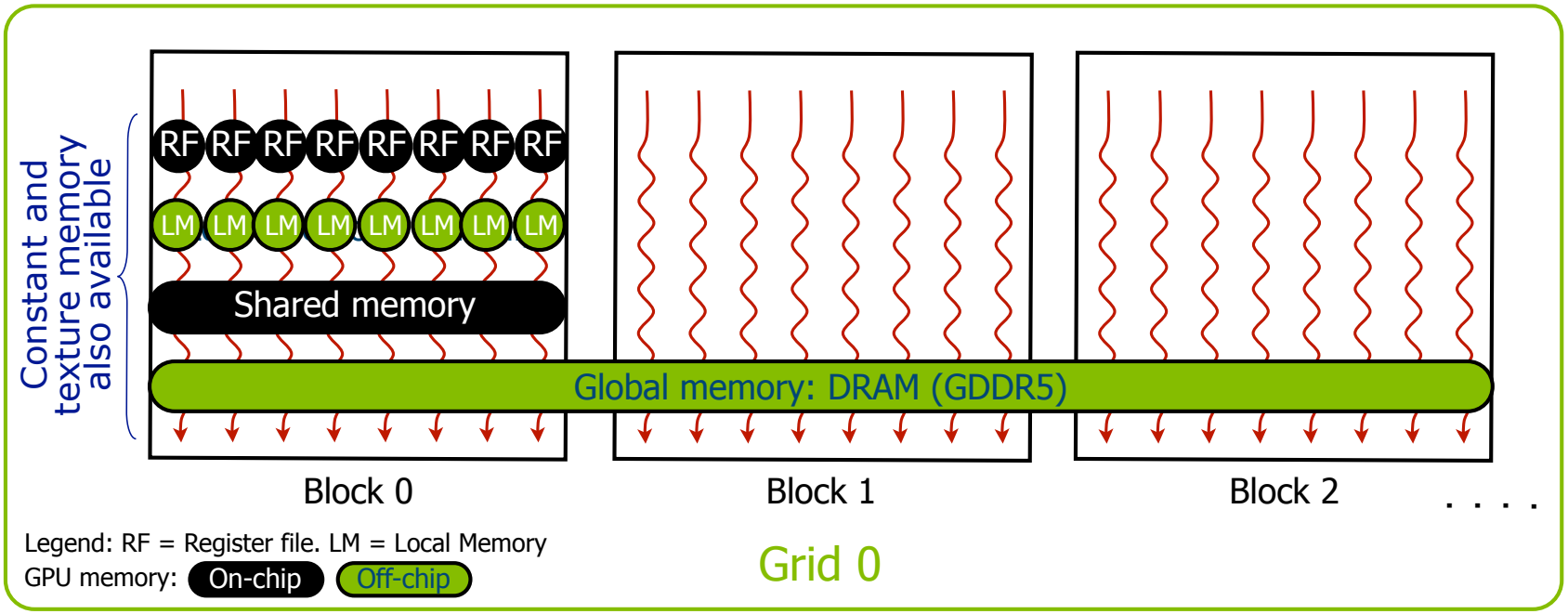


- Threads are assigned to multiprocessors in blocks, and to cores via warps, which is the scheduling unit (32 threads).
- Threads of a block share information via shared memory, and can synchronize via `syncthreads()` calls.

Playing with parallel constrainsts in Kepler to maximize concurrency

- Limits within a multiprocessor: [1] 16 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 16 multiprocessors.

GPU memory: Scope and location



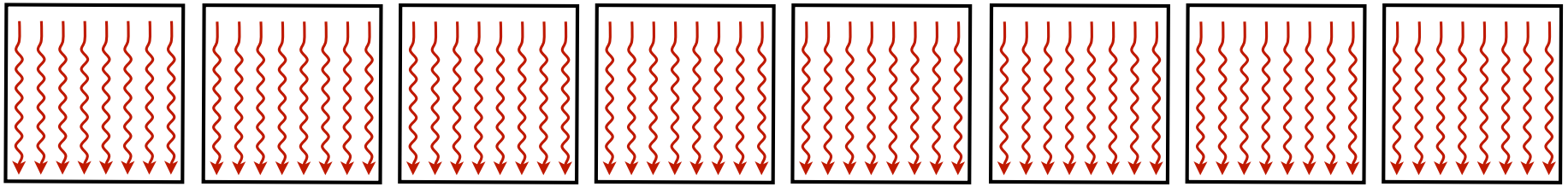
- Threads within a block can use the shared memory to perform tasks in a more cooperative and faster manner.
- Global memory is the only visible to threads, blocks and kernels.

Playing with memory constraints in Kepler to maximize the use of resources

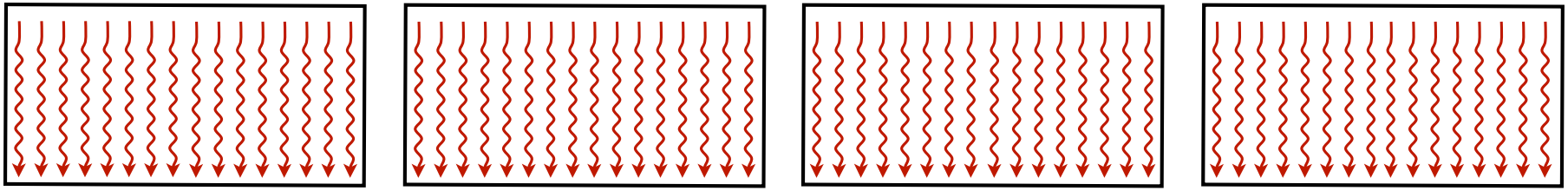
- Limits within a multiprocessor (SMX): 64 Kregs. and 48 KB. of shared memory. That way:
  - To allow a second block to execute on the same multiprocessor, each block must use at most 32 Kregs. and 24 KB of shared memory.
  - To allow a third block to execute on the same multiprocessor, each block must use at most 21.3 Kregs. and 16 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 8-16 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:
  - 1K on Fermi and Kepler.
- Maximum number of blocks:
  - 64K on Fermi, 4G on Kepler.
- 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.
  - 128K blocks of 512 threads each (only feasible in Kepler).
  - 256K blocks of 256 threads each (only feasible in Kepler).
  - ... and so on.

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 (this situation is changing in CUDA 5.0, where the memory accessed from both processors is unified).

IV. Syntax

NVIDIA

IV. 1. Basic elements

NVIDIA

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

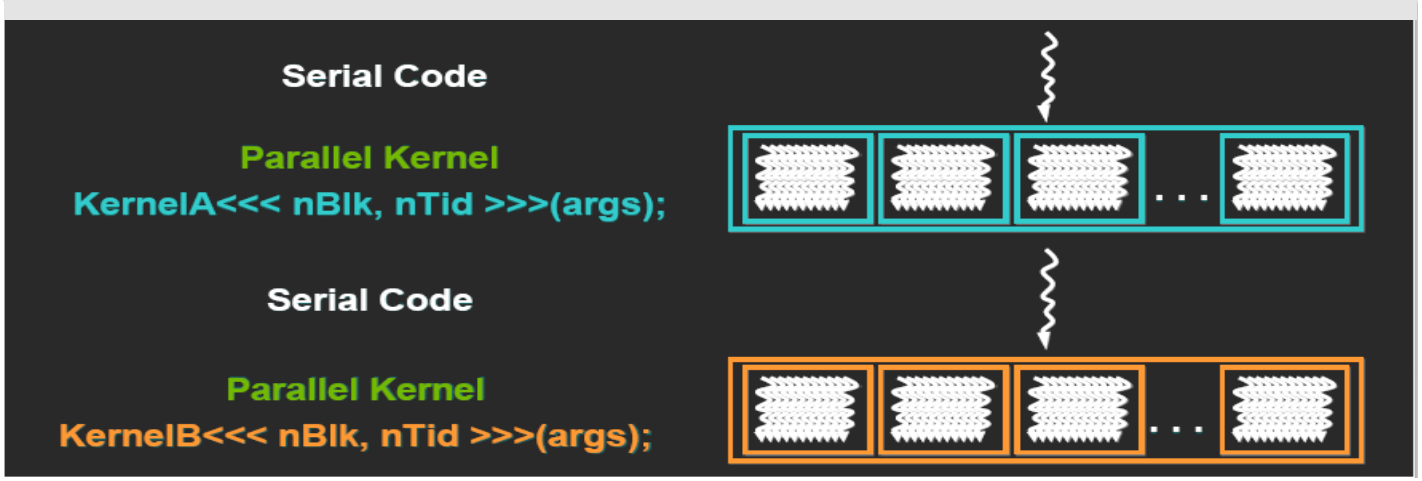
- CUDA extends the C language with a new type of function, kernel, which executes code in parallel on all active threads within GPU. Remaining code is native C executed on CPU.
- The typical `main()` of C combines the sequential execution on CPU and the parallel execution on GPU of CUDA kernels.
- A 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.)

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

Execution

CPU
GPU
CPU
GPU
CPU



- A kernel does not start until all previous kernels are over.
- A **stream** is a new concept used for concurrent kernels.

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

NVIDIA

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.

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

Language extensions

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((void**)&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);
```

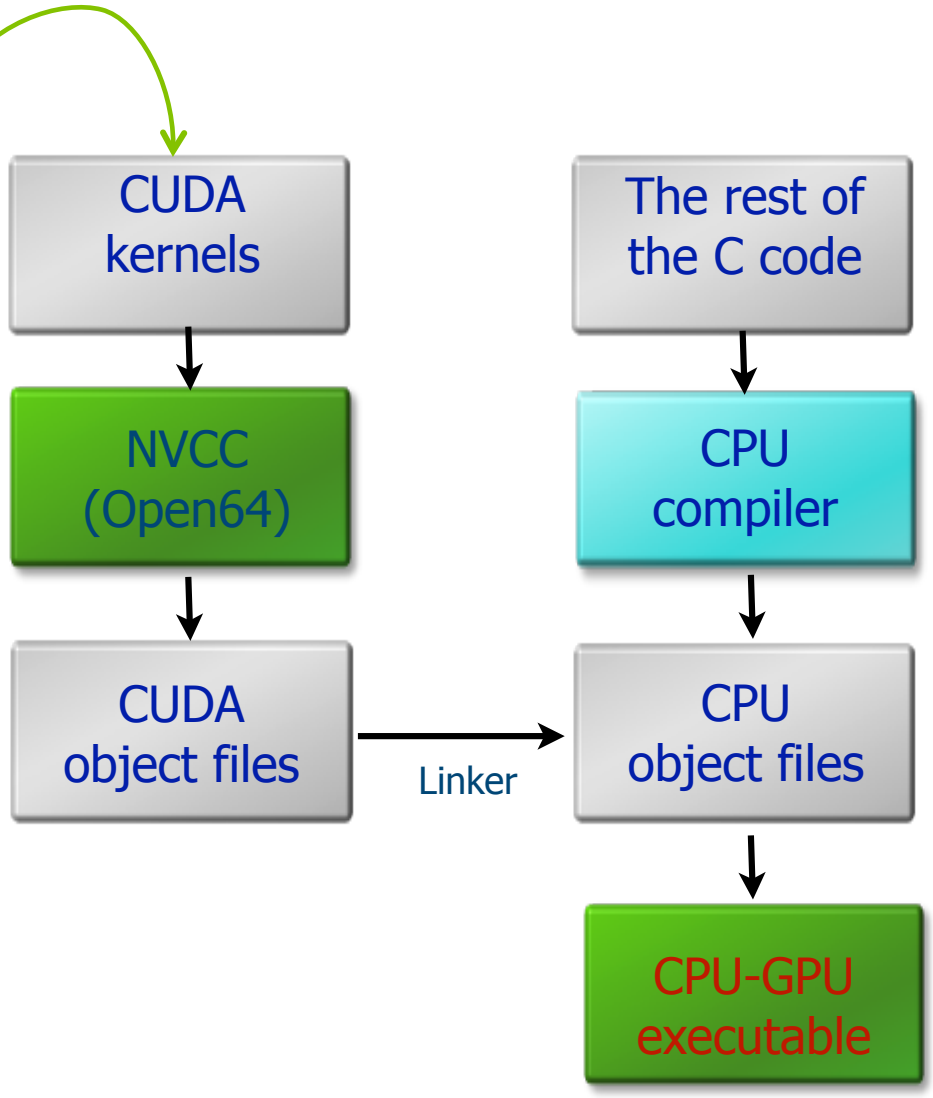
V. Compilation

NVIDIA

The global process

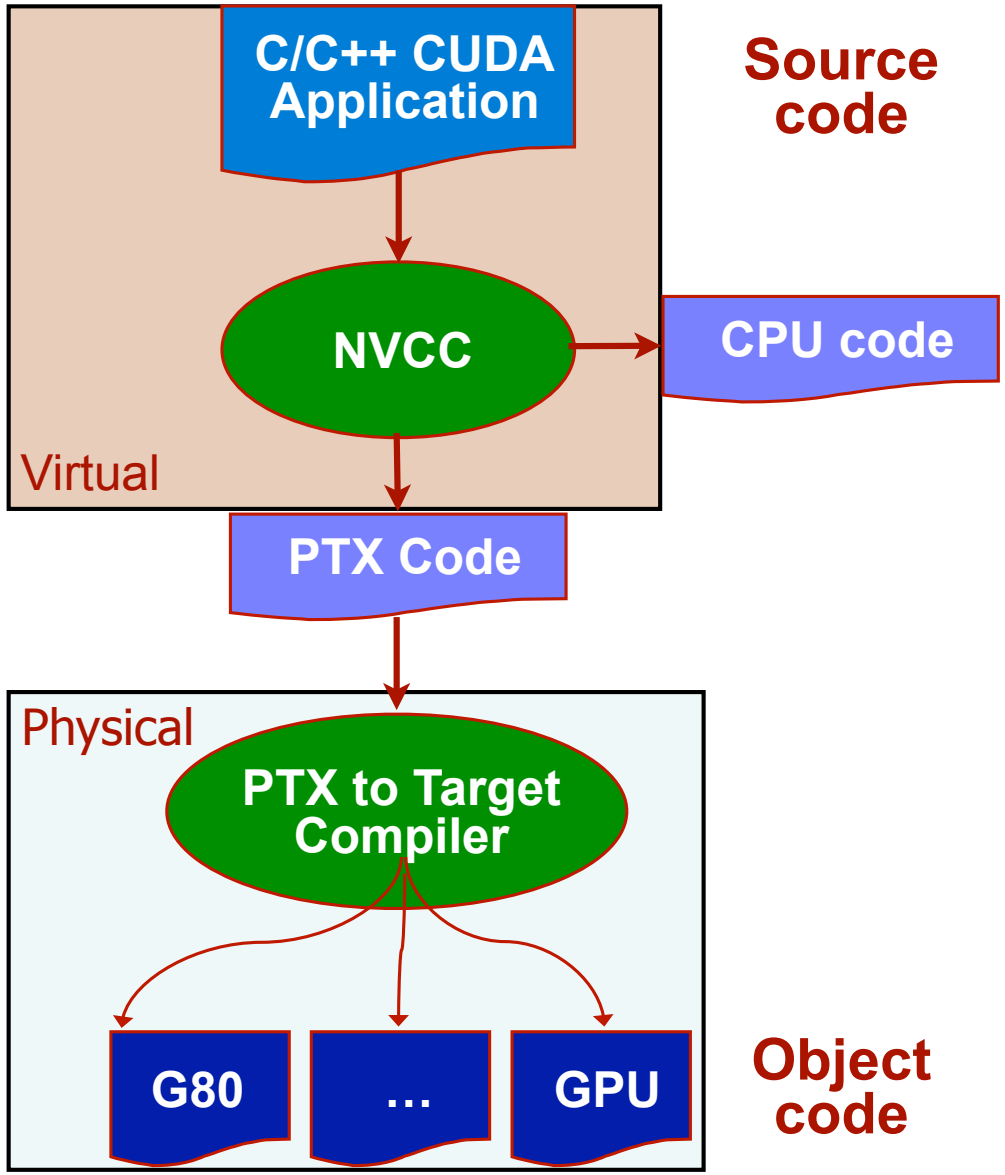
```
void function_in_CPU(… )
{
  ...
}
void other_funcs_CPU(int ...)
{
  ...
}

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

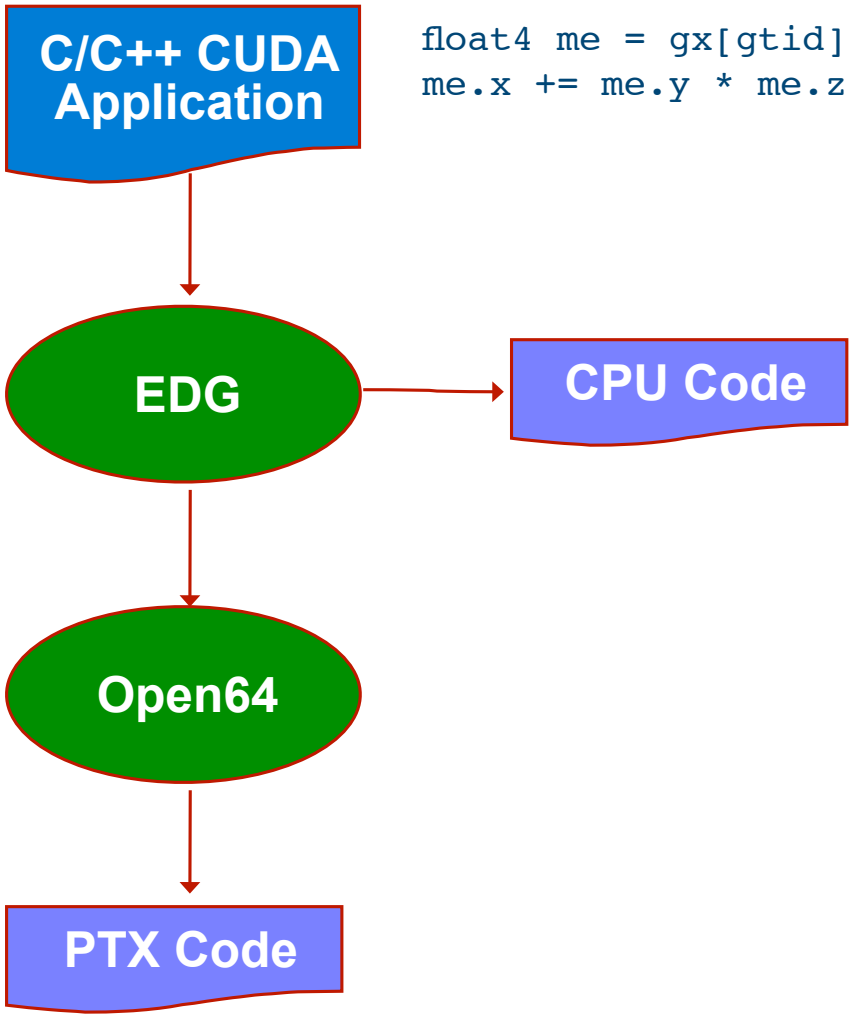


Compilation modules

- 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;
```

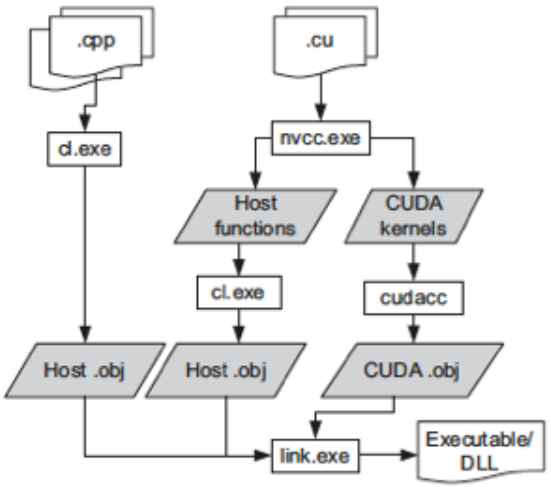
- EDG
  - Separates GPU and CPU code.
- Open64
  - Generates PTX assembler.
- Parallel Thread eXecution (PTX)
  - Virtual machine and ISA.
  - Programming model.
  - Resources and execution states.

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

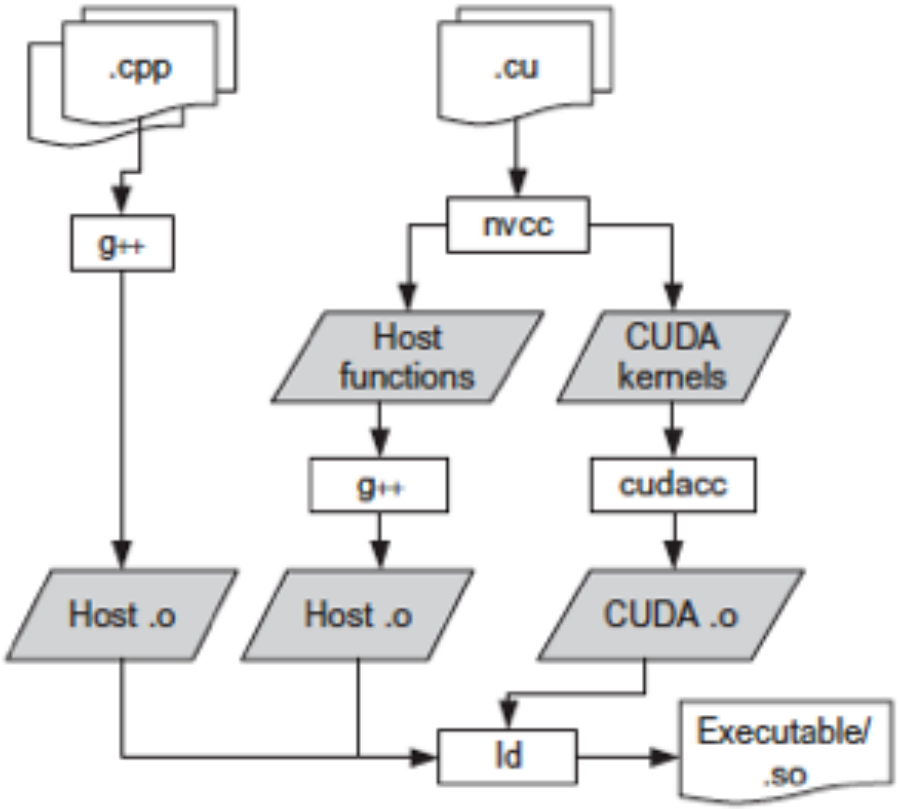
NVCC (NVidia CUDA Compiler)

- 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.)

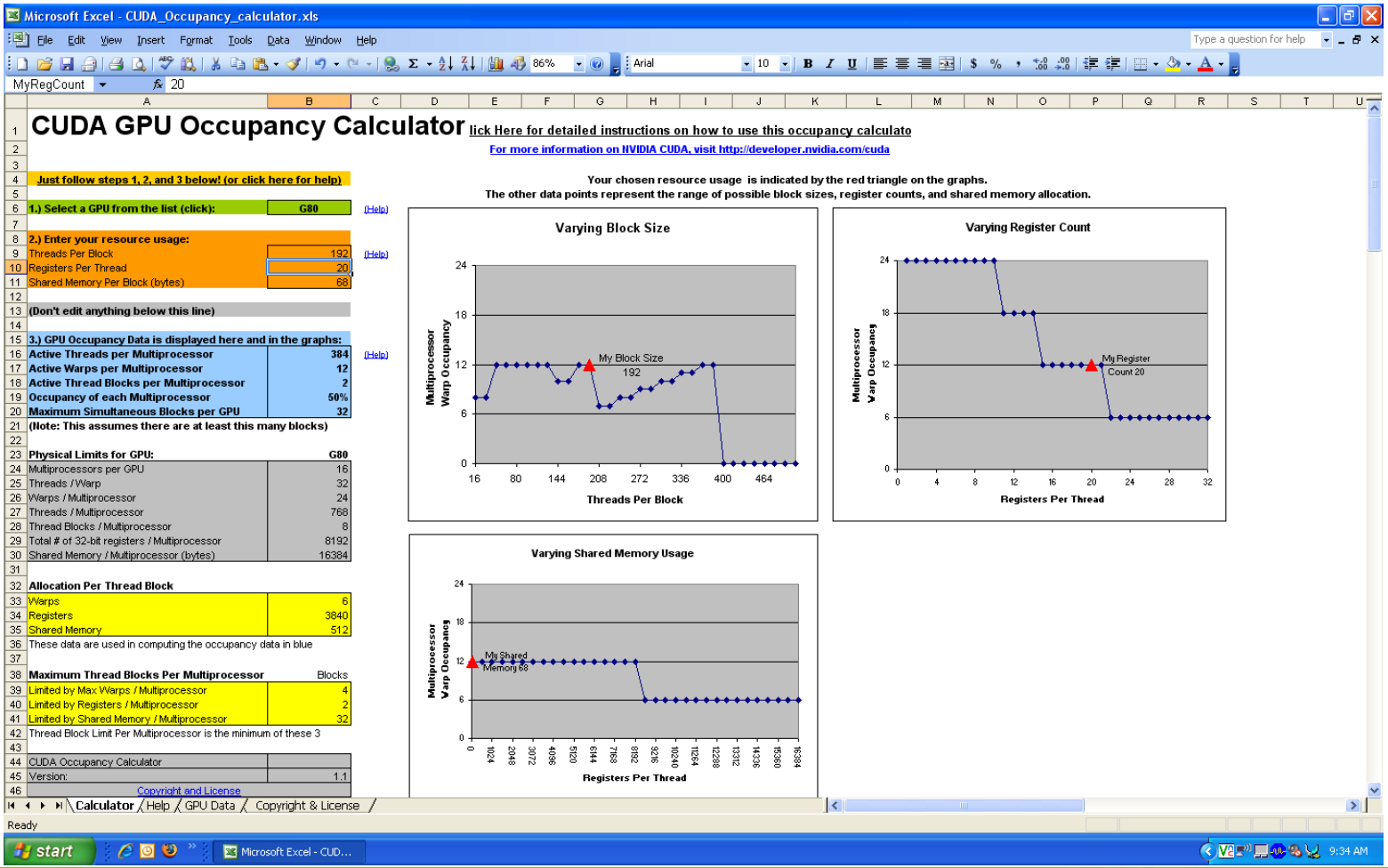
- General rules for the code to be scalable in future generations and for the blocks stream to be processed within a pipeline:
  - (1) Think big for the number of blocks.
  - (2) Think small for the size of threads.
- Tradeoff: More threads per block means better memory latency hiding, but also means 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



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 = 22480 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.

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

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 read 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

NVIDIA

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 = threadIdx.x + (blockDim.x* blockIdx.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

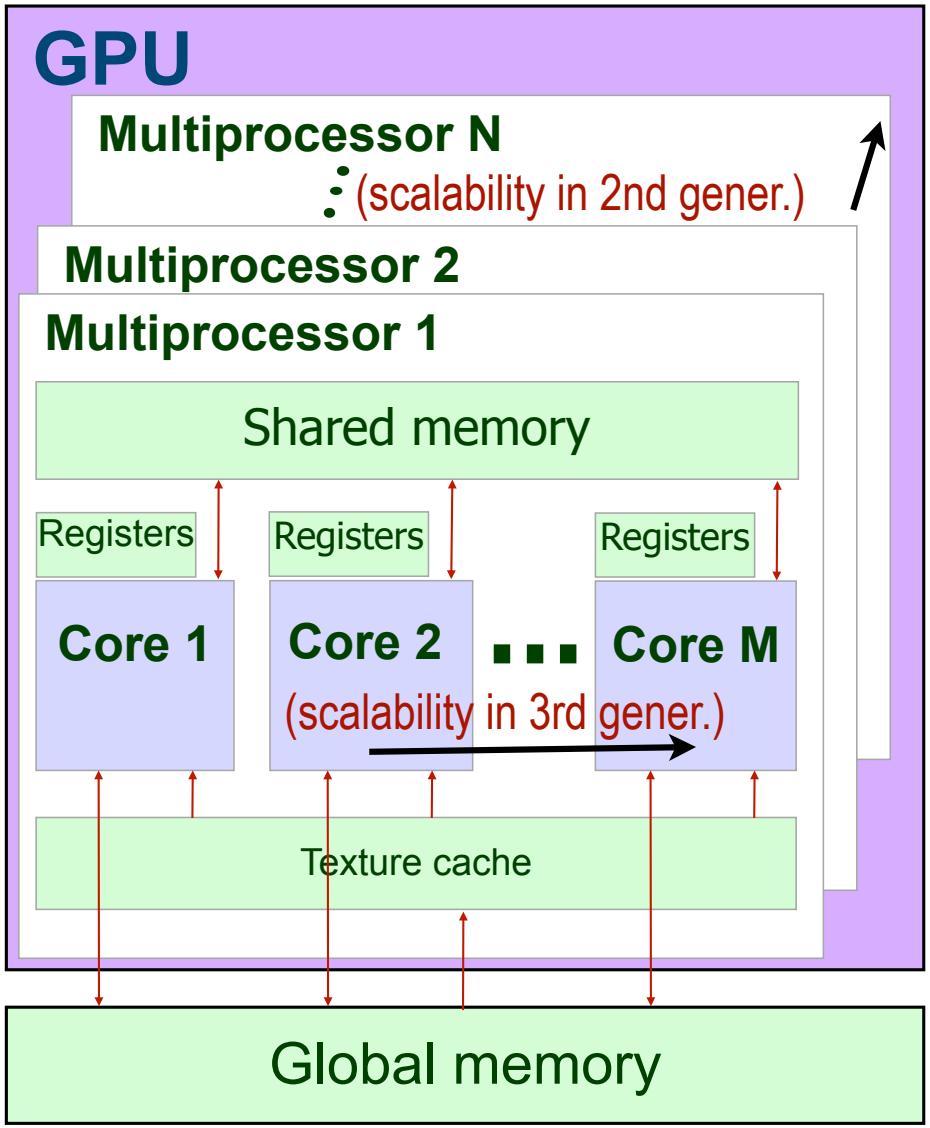
- The `__global__` prefix indicates that `vecAdd()` will execute on device (GPU) and will be called from host (CPU).
- `A`, `B` and `C` are pointers to device memory, so we need to:
  - Allocate/free memory on GPU, using `cudaMalloc()`/`cudaFree()`.
  - These pointers cannot be dereferenced in host code.

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);
```

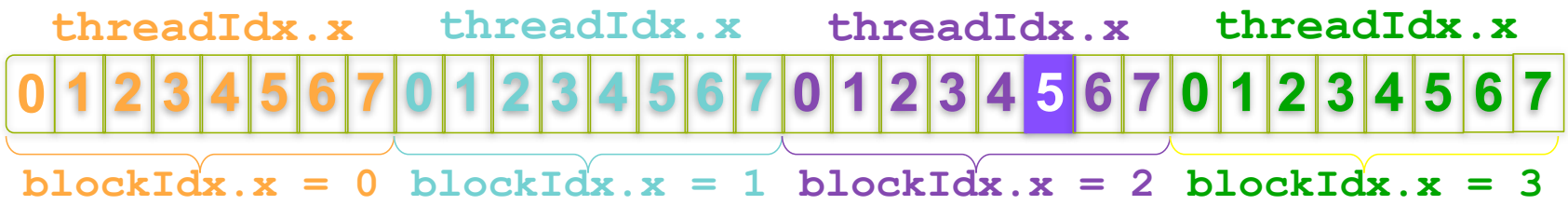
Running in parallel (regardless of hardware generation)

- `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 = threadIdx.x+ blockDim.x* blockIdx.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:



- 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 = 5 + (8 * 2) = 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 = threadIdx.x + (blockDim.x * blockIdx.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

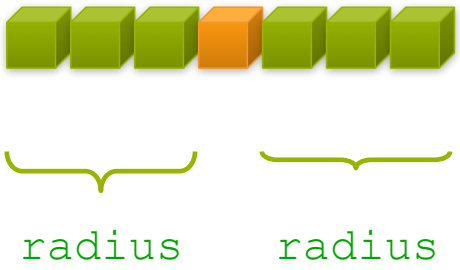
NVIDIA

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

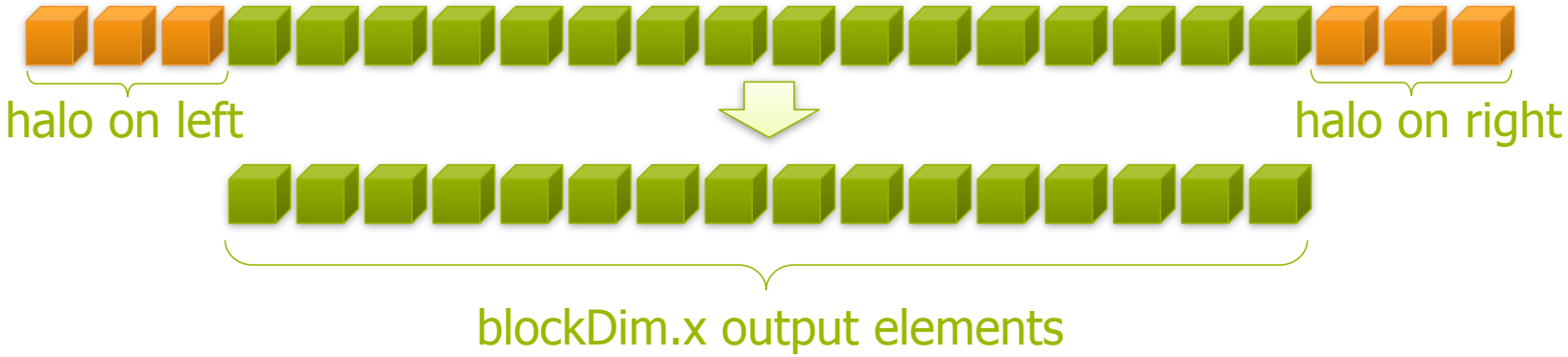
- Threads within a block can share data via shared memory.
  - Shared memory is user-managed: Declare with __shared__ prefix.
  - Data is allocated per block.
  - Shared memory is extremely fast:
    - 500 times faster than global memory (video memory - GDDR5). The difference is technology: static (built 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.

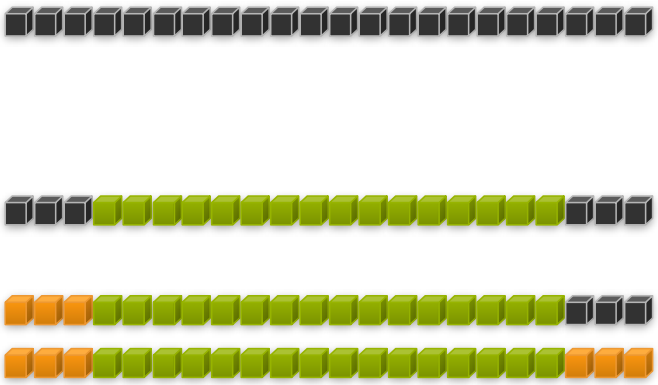
Using Shared Memory

- 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 = threadIdx.x
             + blockIdx.x * blockDim.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 = threadIdx.x + blockIdx.x * blockDim.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

NVIDIA

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

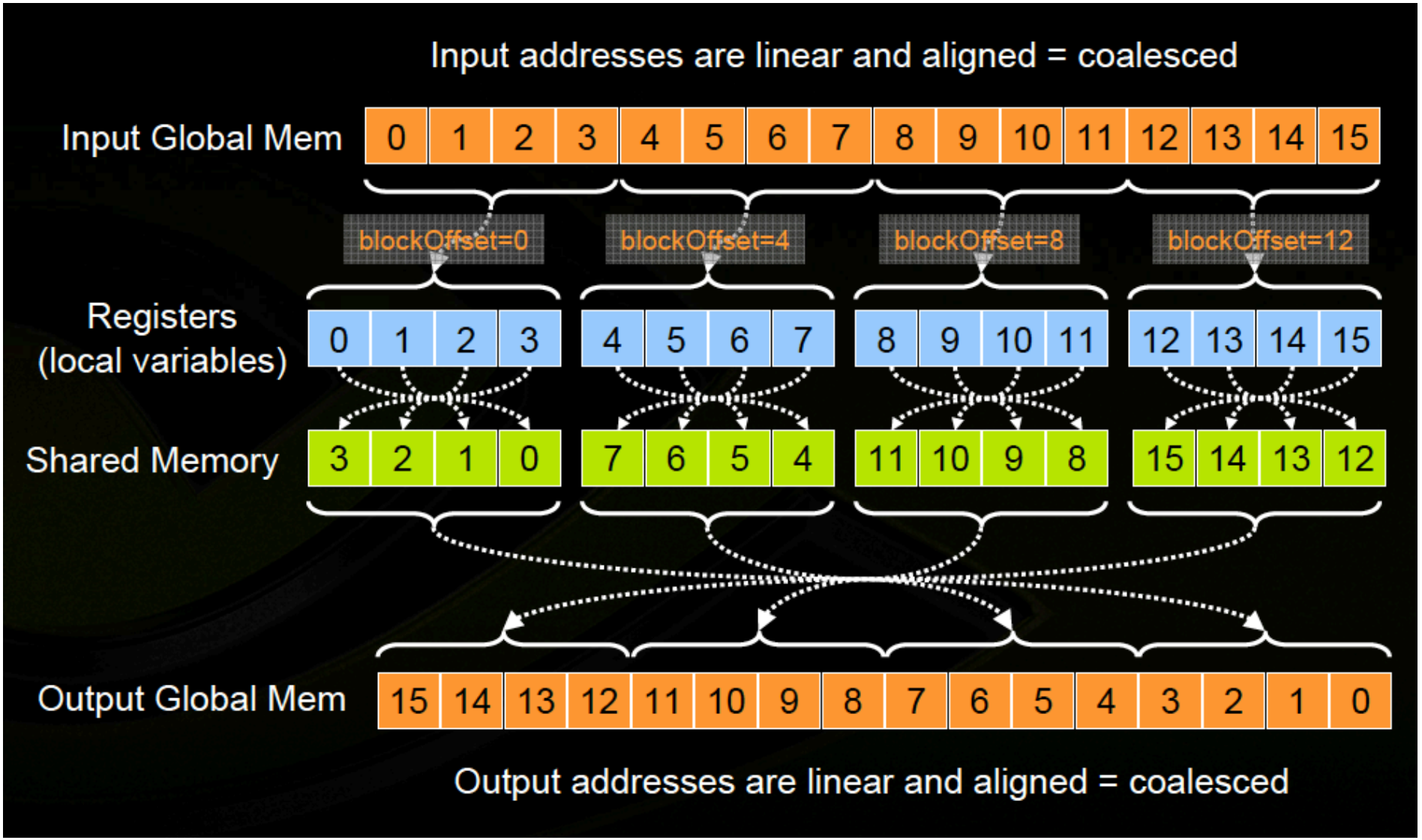
```
__global__ void reverseArray(int *in, int *out) {
  int index_in = threadIdx.x;
  int index_out = blockDim.x – 1 – threadIdx.x;

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

- It is a solution too naive, 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) {
  int in_offset  =                    blockIdx.x  * blockDim.x;
  int out_offset = (gridDim.x – 1 – blockIdx.x) * blockDim.x;
  int index_in  =  in_offset +                     threadIdx.x;
  int index_out = out_offset + (blockDim.x – 1 – threadIdx.x);

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

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 = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  syncthreads();

  // Reverse local arrays within each block
  temp[lindex] = temp[blockDim.x-lindex-1];
  syncthreads();

  // Reverse contents in chunks of whole blocks
  out[threadIdx.x + ((N/blockDim.x)-blockIdx.x-1) * blockDim.x] = temp[lindex];
}
```

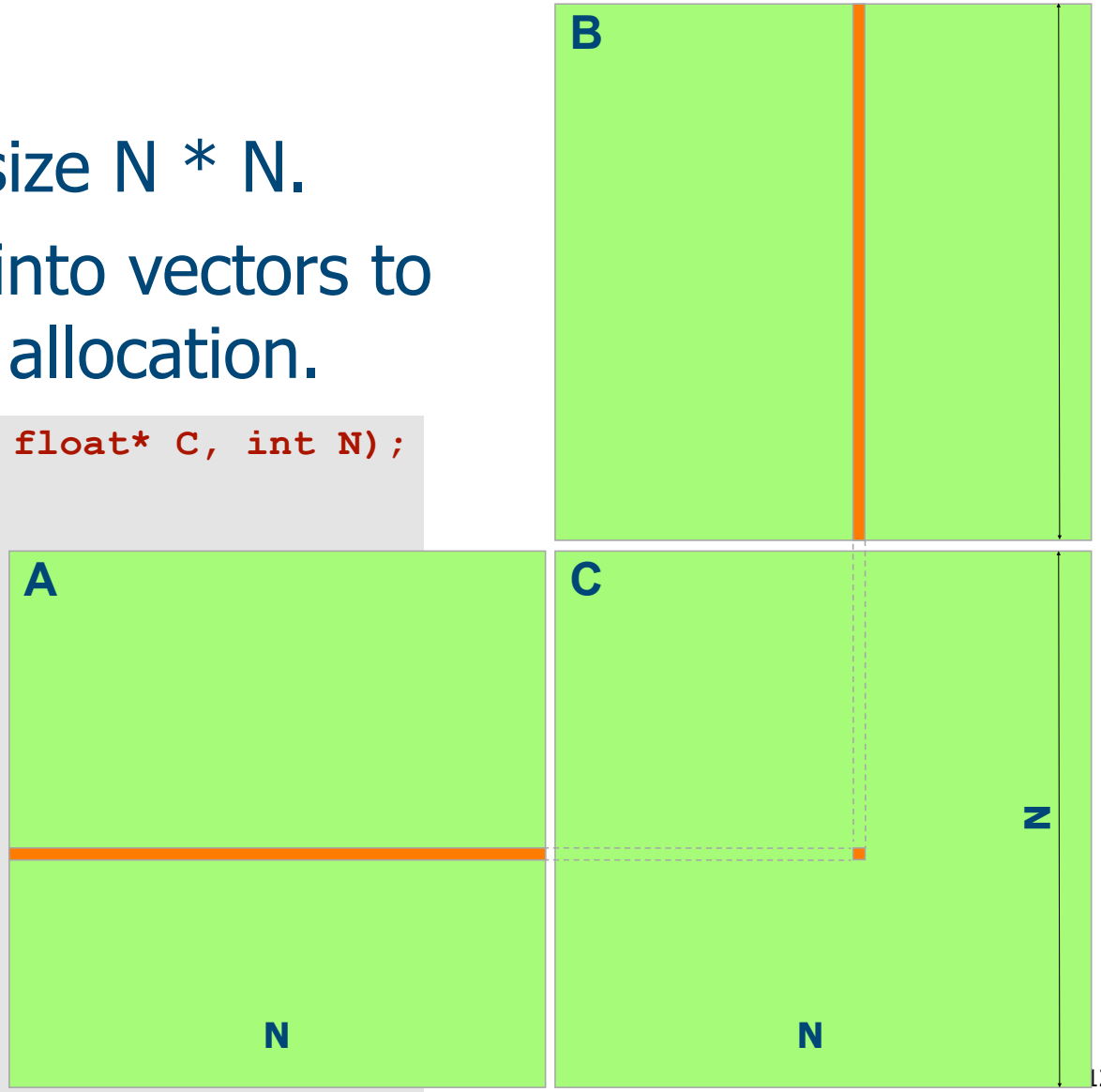
VI. 4. Matrix product

NVIDIA

Typical CPU code written in C language

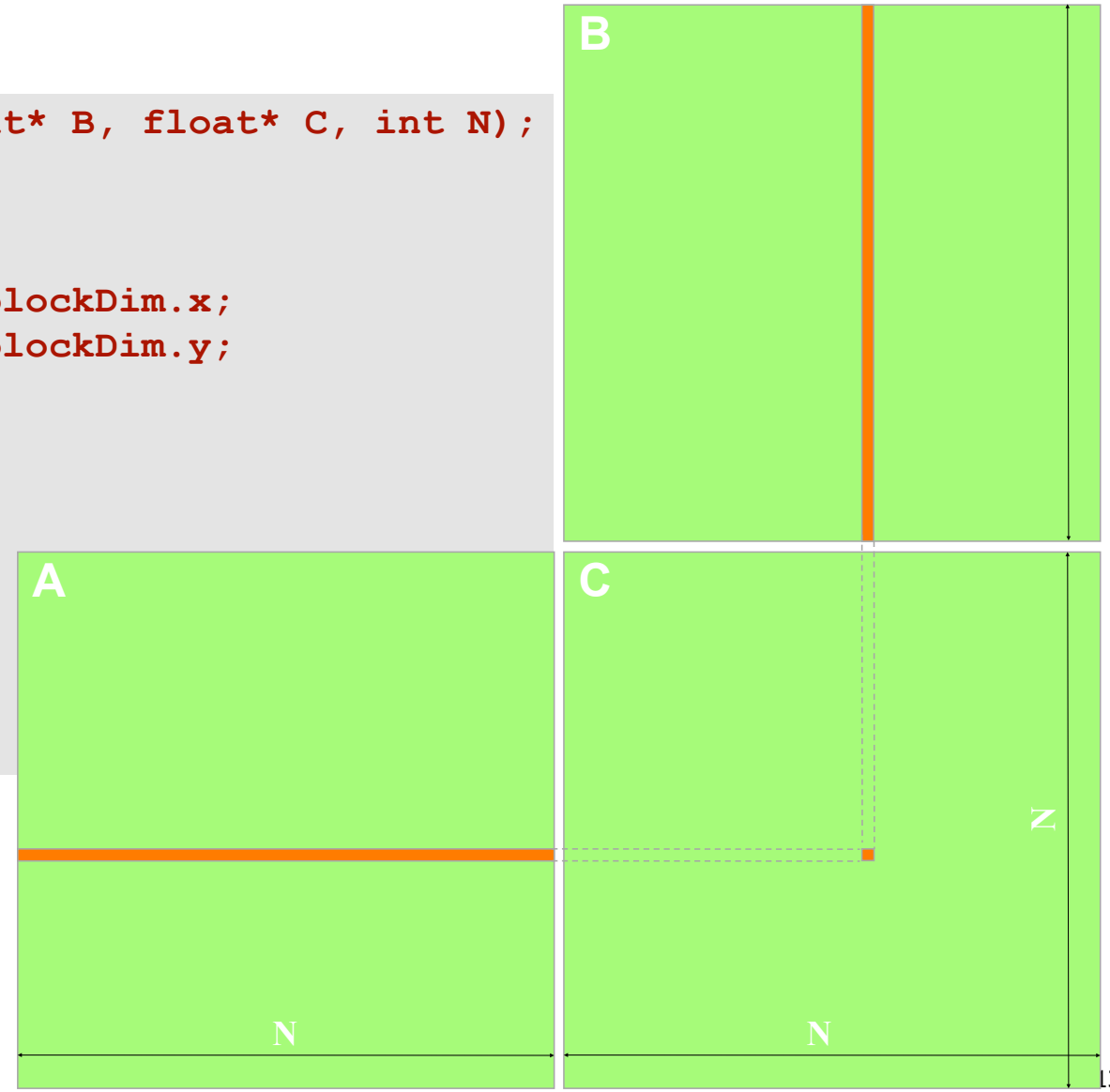
- C = A * B.
- All square matrices of size N * N.
- Matrices are serialized into vectors to simplify dynamic memory allocation.

```
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;
    }
}
```



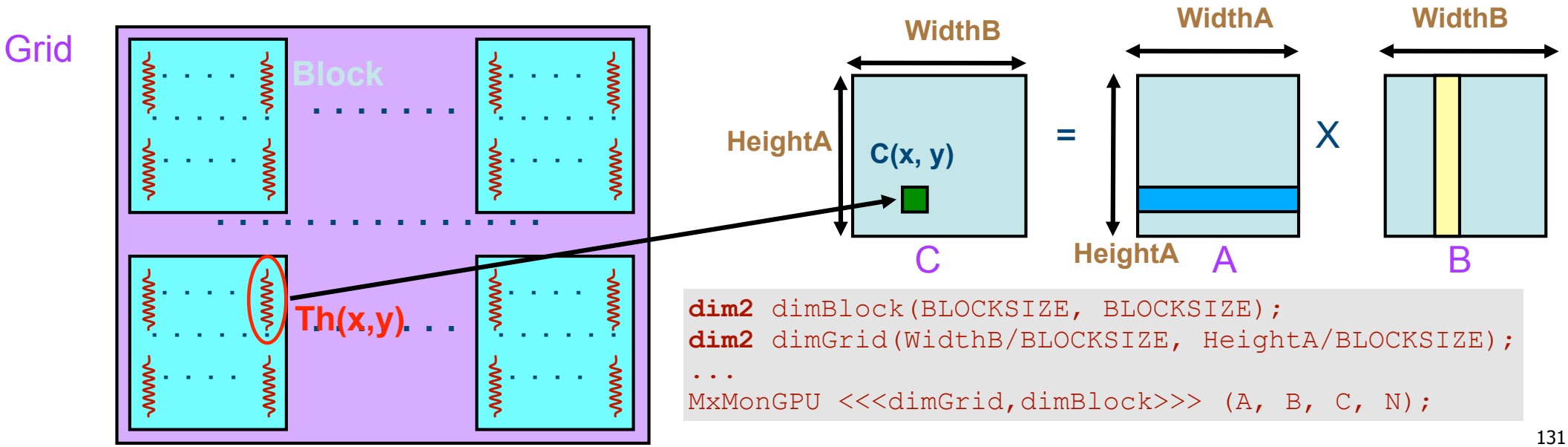
CUDA version for the matrix product: A draft for the parallel code

```
__global__ MxMonGPU(float* A, float* B, float* C, int N);
{
  float sum=0;
  int i, j;
  i = threadIdx.x + blockIdx.x * blockDim.x;
  j = threadIdx.y + blockIdx.y * blockDim.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.



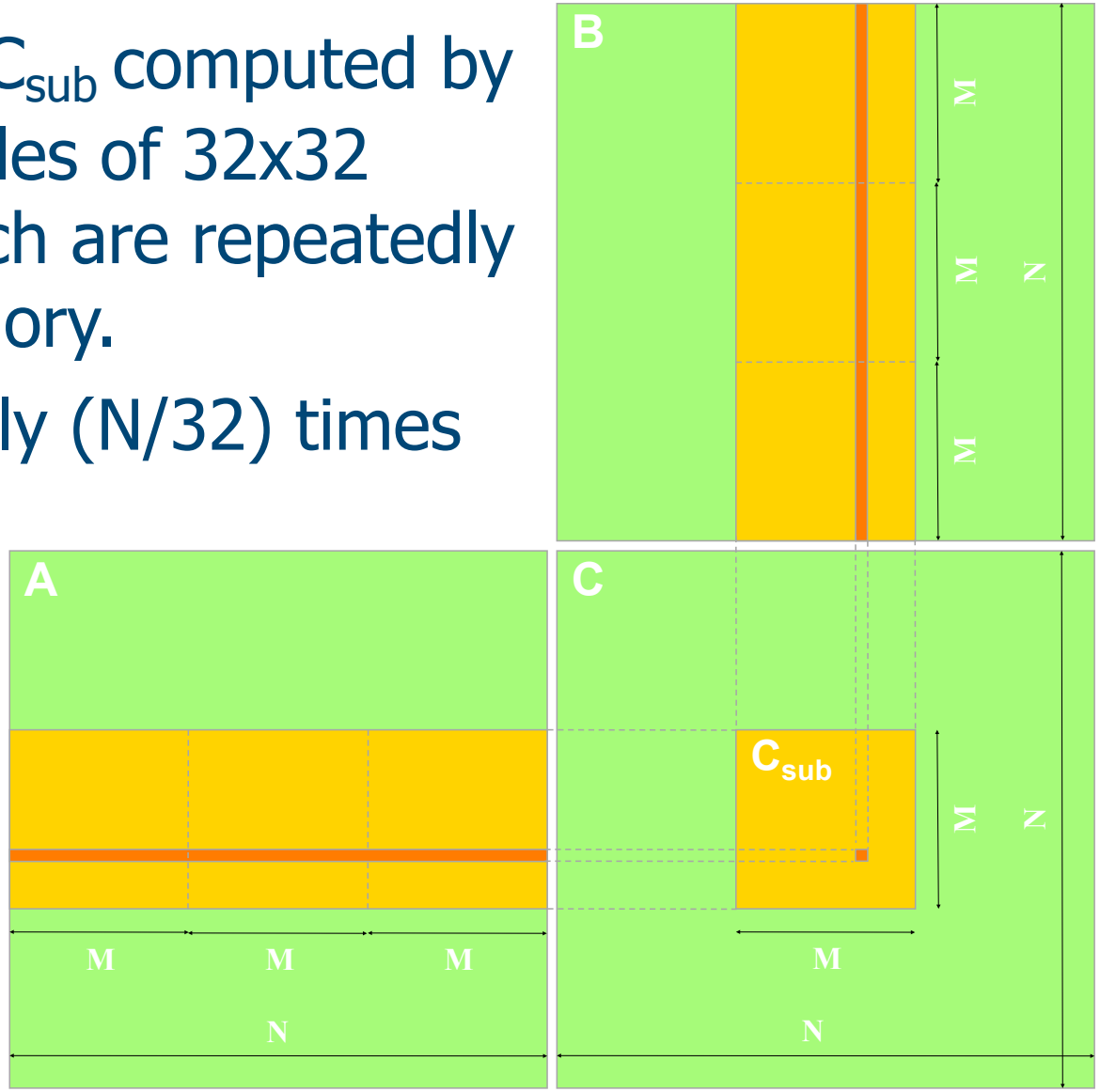

```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(WidthB/BLOCKSIZE, HeightA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid,dimBlock>>> (A, B, C, N);
```

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: shData[threadIdx.x][threadIdx.y], threads within a half-warp will be reading from the same column, that is, from the same shared memory bank.
- However, using shData[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 in shared memory transposed.

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 = tx + blockIdx.x*blockDim.x;      j = ty + blockIdx.y*blockDim.y;
  __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

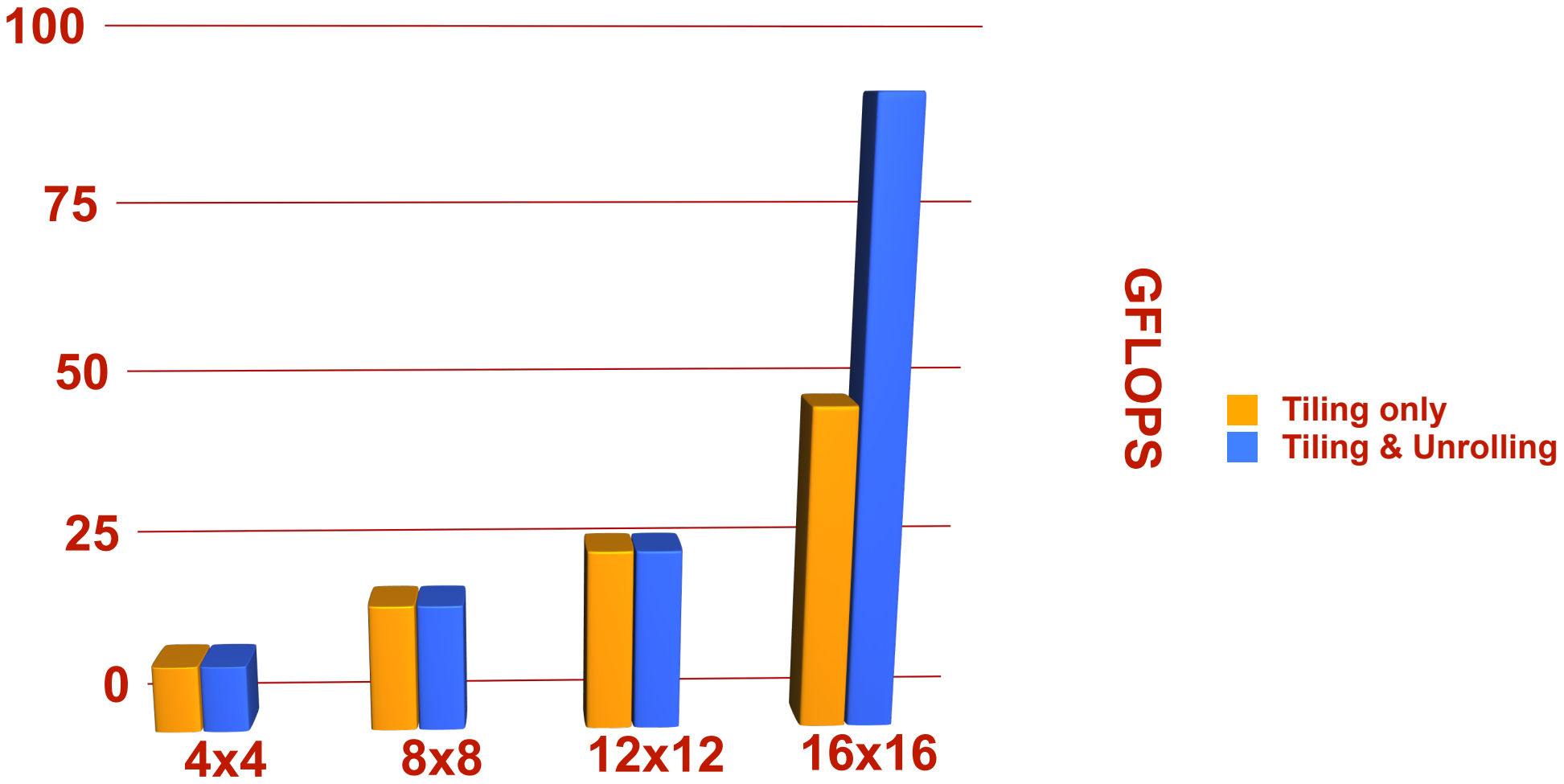
Without loop unrolling:

```
    ...
    __syncthreads();

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

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

Unrolling the loop:

```
    ...
    __syncthreads();

    // Compute results for that 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



Tile size (32x32 unfeasible on G80 hardware)

VII. Bibliography and tools

NVIDIA

CUDA Zone:
Basic web resource for a CUDA programmer

EXPLORE CUDA ZONE

WHAT IS CUDA

Learn more about the CUDA parallel computing platform and programming model.

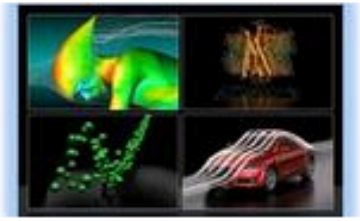
GET STARTED - PARALLEL COMPUTING

Find out about different paths and options for deploying CUDA and GPU Computing in your project

- Languages (C/C++, Python).
- Libraries (cuBLAS, cuFFT).
- Directives (OpenACC).
- Templates (thrust).

CUDA IN ACTION - RESEARCH & APPS

Supercomputing is now accessible for every researcher and scientist.
Find latest research, applications and links to how CUDA is transforming the industry



CUDA TOOLKIT

The NVIDIA CUDA Toolkit provides a comprehensive development environment for C and C++ developers building GPU-accelerated applications.

- Compiler (NVCC).
- Debugger (GDB).
- Profiler (cudaprof and Visual).
- Development envir. (Nsight).
- Code examples.

CUDA EDUCATION & TRAINING

Get educated with online courses, webinars, University Courses and wealth of technical papers & documentation

CUDA TOOLS & ECOSYSTEM

Learn more about powerful CUDA tools, libraries, languages, and other development aids available from NVIDIA & partners.

- Eclipse.
- Matlab.
- CUDA Fortran.
- GPUDirect.
- SDK for the LLVM compiler.

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

CUDA 6 Production Release.
Free download for all platforms and users

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

Windows | Linux | Mac OSX

Version		64-bit	32-bit
Windows 8.1 Windows 7 Windows Vista	Notebook	EXE	EXE
	Desktop	EXE	EXE
Windows XP	Desktop	EXE	EXE

Getting Started Guide

Windows | Linux | Mac OSX

OSX Release	Package
10.8	PKG
10.9	

Getting Started Guide

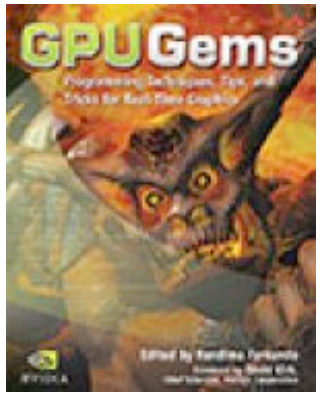
Windows | Linux | Mac OSX

Distribution	x86 64-bit		x86 32-bit		ARMv7
Fedora 19	RPM	RUN			
OpenSUSE 12.3	RPM	RUN			
RHEL 6 CentOS 6	RPM	RUN			
RHEL 5 CentOS 5		RUN			
SLES 11 (SP2 & SP3)	RPM	RUN			
Ubuntu 13.04	DEB	RUN		RUN	DEB
Ubuntu 12.04	DEB*	RUN	DEB	RUN	
L4T, Linux for Tegra				LINK	LINK

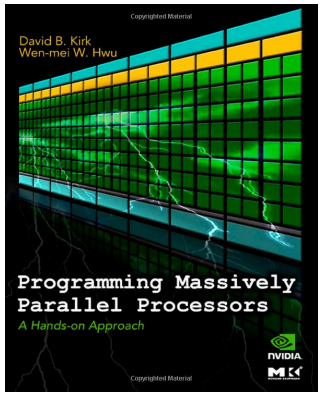
Getting Started Guide

CUDA books: From 2007 to 2013

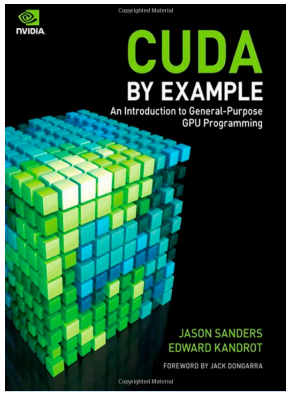
- GPU Gems series [developer.vidia.com/content/GPUGems3/gpugems3_part01.html]
- List of CUDA books in [www.nvidia.com/object/cuda_books.html]



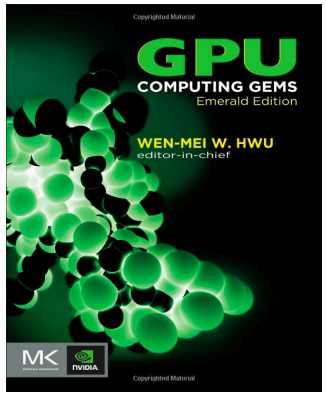
Sep'07



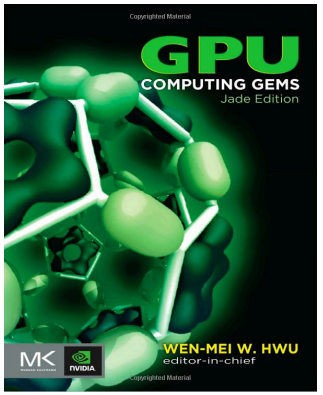
Feb'10



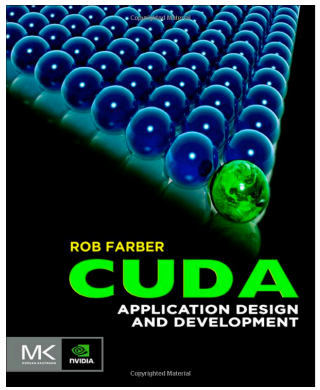
Jul'10



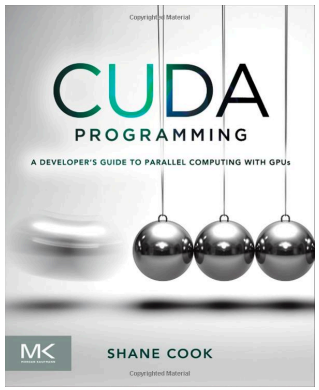
Abr'11



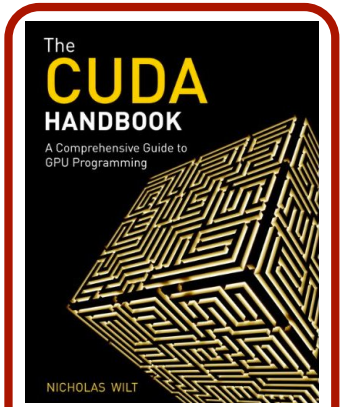
Oct'11



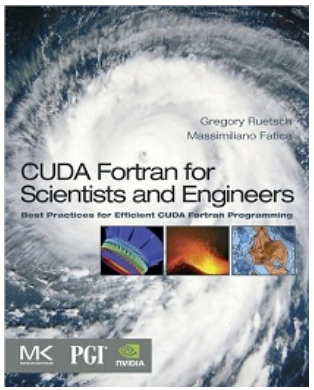
Nov'11



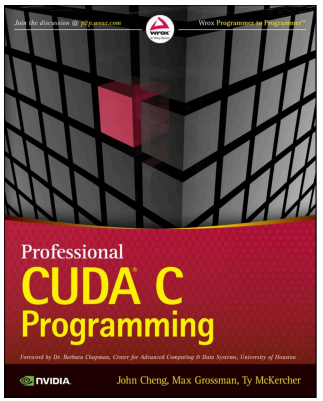
Dic'12



Jun'13



Oct'13



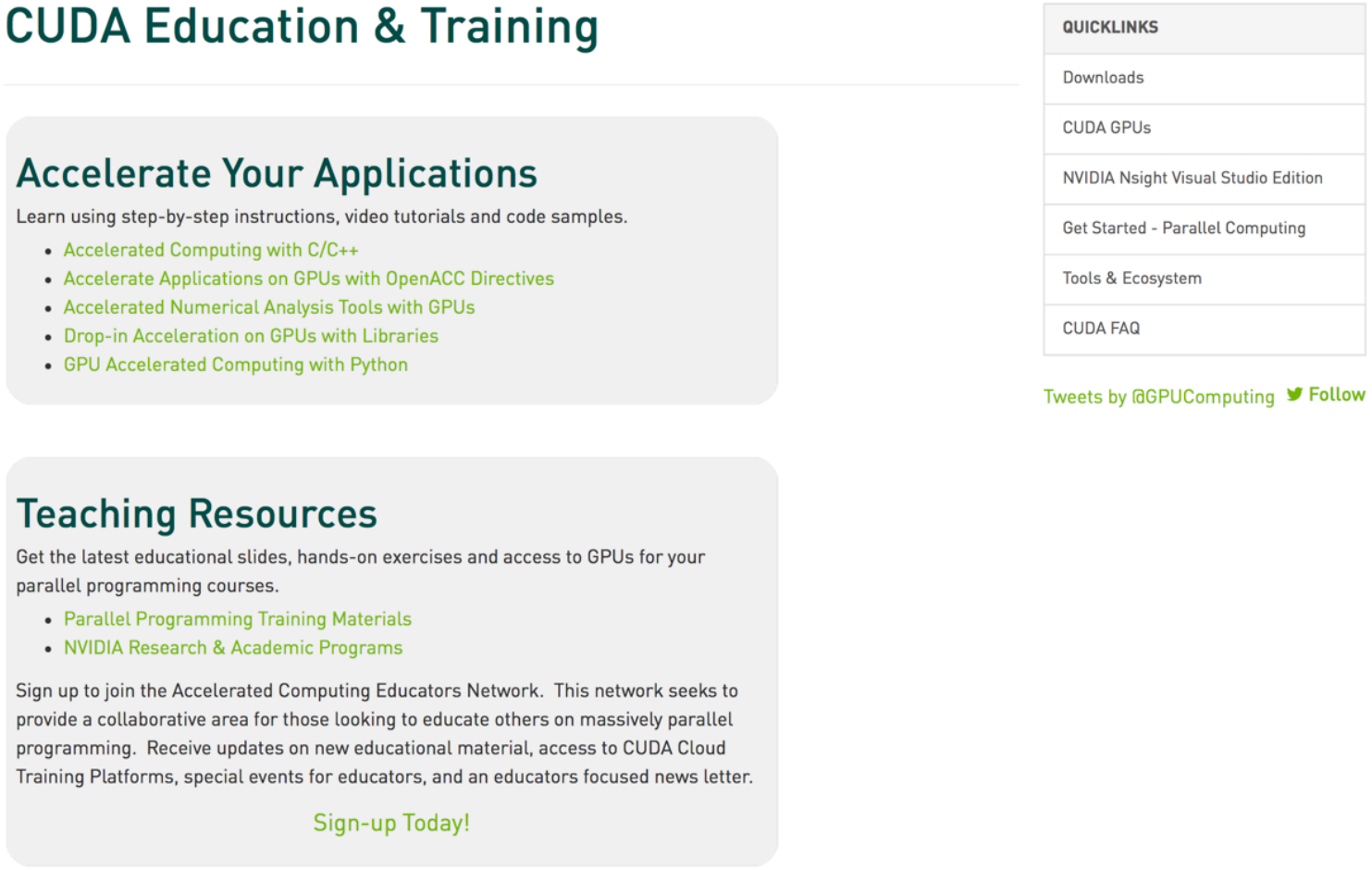
Sep'14

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 platforms.
  - 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).

Choices to accelerate your applications on GPUs and material for teaching CUDA

- [developer.nvidia.com/cuda-education-training] (also available from the left lower corner of the CUDA Zone)

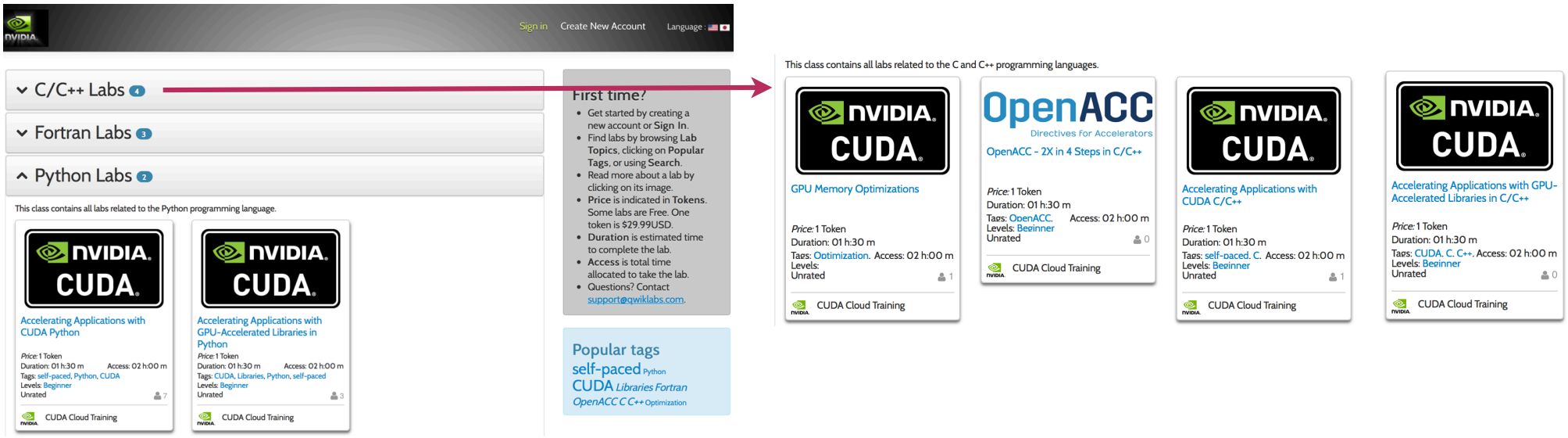


Courses on-line (free access)

- More than 50.000 registered users from 127 countries over the last 6 months. An opportunity to learn from CUDA masters:
  - Prof. Wen-Mei Hwu (Univ. of Illinois).
  - Prof. John Owens (Univ. of California at Davis).
  - Dr. David Luebke (Nvidia Research).
- There are two basic options, both recommended:
  - Introduction to parallel programming: UDACITY
    - 7 units of 3 hours = 21 hours.
    - Provides high-end GPUs to carry out the proposed assignments.
    - [https://developer.nvidia.com/udacity-cs344-intro-parallel-programming]
  - Heterogeneous Parallel Programming: coursera
    - 9 weeks, each with classes (20' video), quizzes and programming assignments.
    - [https://www.coursera.org/course/hetero]

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.



Talks and webinars

- Talks recorded at GTC (Graphics Technology Conference):
  - 383 talks from 2013.
  - More than 500 available from 2014.
  - [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]
- CUDACasts:
  - [bit.ly/cudacasts]

Examples of webinars about CUDA 6.0

GTC Express Webinar Program

GTC Express is a year-round extension of the great content available at GTC. Each month, top developers, scientists, researchers, and industry practitioners present innovative and thought-provoking webinars focused on the GPU-enabled work they're doing and sharing how GPUs are transforming their fields.

Register below for upcoming webinars and explore recordings of previous webinars.

GTC EXPRESS SCHEDULE AT-A-GLANCE

Date	Title	Speaker	
June 3, 2014, 9:00 AM PDT	The Next Steps for Folding@home	Vijay Pande, Professor, Stanford University	> Register Now
May 14, 2014, 10:00 AM PDT	CUDA 6: Performance Overview	Jonathan Cohen, Senior Manager, CUDA Libraries and Algorithms, NVIDIA	> Register Now
May 13, 2014, 9:00 AM PDT	An Overview of AMBER 14 - Creating the World's Fastest Molecular Dynamics Software Package	Ross C. Walker, University of California San Diego, Scott Le Grand, Amazon Web Services, and Adrian Roitberg, University of Florida.	> Register Now
May 7, 2014, 10:00 AM PDT	CUDA 6: Drop-in Performance Optimized Libraries	NVIDIA DevTech Team	> Register Now
May 1, 2014, 10:00 AM PDT	CUDA 6: Unified Memory	Mark Ebersole, CUDA Educator, NVIDIA	> Register Now
April 23, 2014, 11:00 AM PDT	CUDA 6 Features Overview	Ujval Kapasi, CUDA Product Manager, NVIDIA	> Register Now

Developers

- Sign up as a registered developer:
  - [www.nvidia.com/paralleldeveloper]
  - Access to exclusive developer downloads.
  - Exclusive access to pre-release CUDA installers like CUDA 6.0.
  - Exclusive activities an special offers.
- Meeting point with many other developers:
  - [www.gpucomputing.net]
- GPU news and events:
  - [www.gpgpu.org]
- 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]
- And a a last tool for tuning code: The CUDA Occupancy Calculator
  - [developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls]

Future developments

- Nvidia's blog contains articles unveiling future technology to be used within CUDA. It is the most reliable source about what's next (subscription recommended):
  - [devblogs.nvidia.com/parallelforall]
- Some recommended articles:
  - "5 Powerful New Features in CUDA 6", by Mark Harris.
  - "CUDA 6.0 Unified Memory", by Mark Ebersole.
  - "CUDA Dynamic Parallelism API and Principles", by Andrew Adinetz.
  - "NVLINK, Pascal and Stacked Memory: Feeding the Appetite for Big Data", by Denis Foley.
  - "CUDA Pro Tip: Increase Application Performance with NVIDIA GPU Boost", by Mark Harris.

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