Programming GPUs with CUDA

IX Conference on Articulated Motion and Deformable Objects

4 hours tutorial. Palma de Mallorca (Spain), July, 13th-15th, 2016

Manuel Ujaldón

A/Prof. @ University of Malaga (Spain)

CUDA Fellow @ NVIDIA

Agenda

1. 10:00 - 11:30 : First session. The hardware of the GPU
2. 11:30 - 12:00 : Coffee break.
3. 12:00 - 13:30 : Second session. CUDA Programming

Contents [80 slides]

Prerequisites for this tutorial

Prerequisites for this tutorial

- You (probably) need experience with C.

Prerequisites for this tutorial

- You (probably) need experience with C.
- You do not need parallel programming background (but it helps if you have it).

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.

Prerequisites for this tutorial

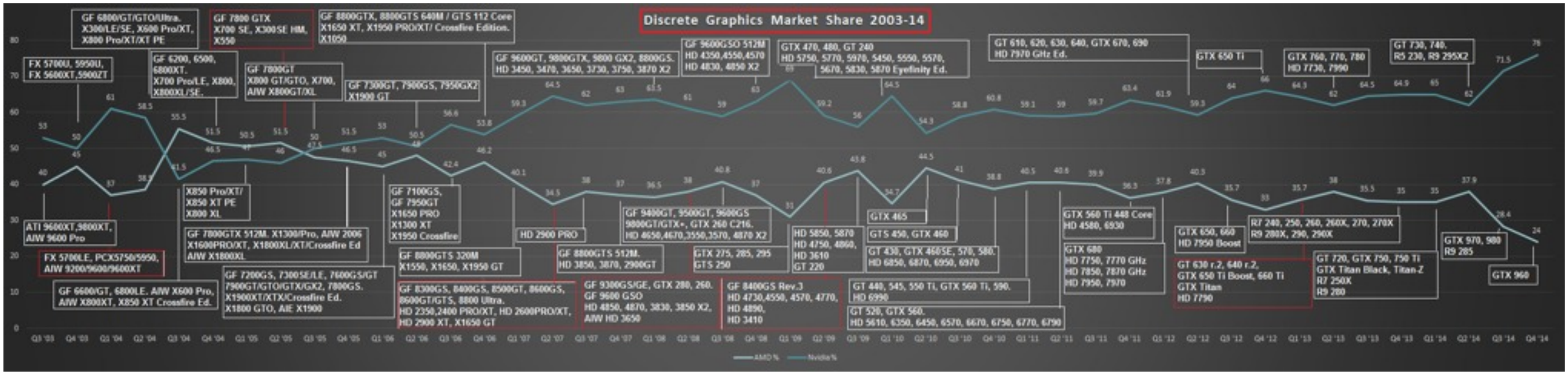
- You (probably) need experience with C.
- You do not need parallel programming background (but it helps if you have it).
- You do not need knowledge about the GPU architecture: We will start with the basic pillars.
- You do not need graphics experience. Those were the old times (shaders, Cg). With CUDA,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

I. Introduction

NVIDIA

Past:
The GPU market share

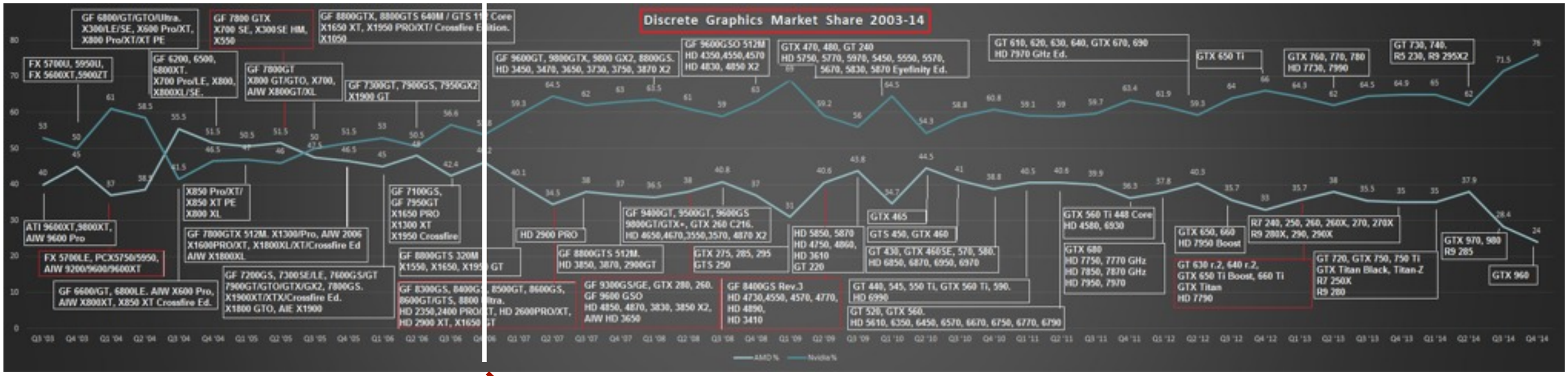
Source: Jon Peddie Research consulting



	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

Past:
The GPU market share

Source: Jon Peddie Research consulting

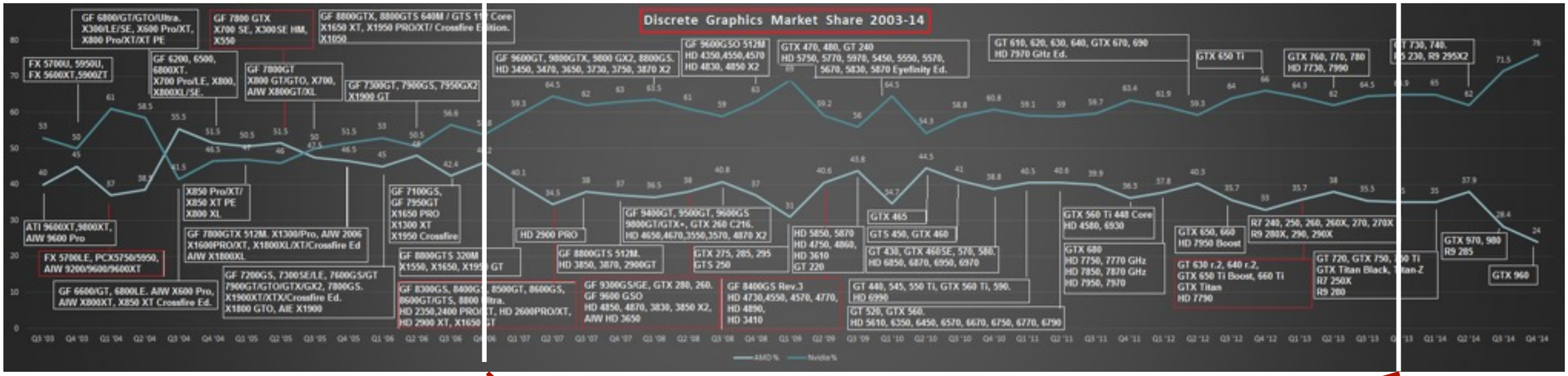


	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

Pre-CUDA era: 1-1

Past:
The GPU market share

Source: Jon Peddie Research consulting

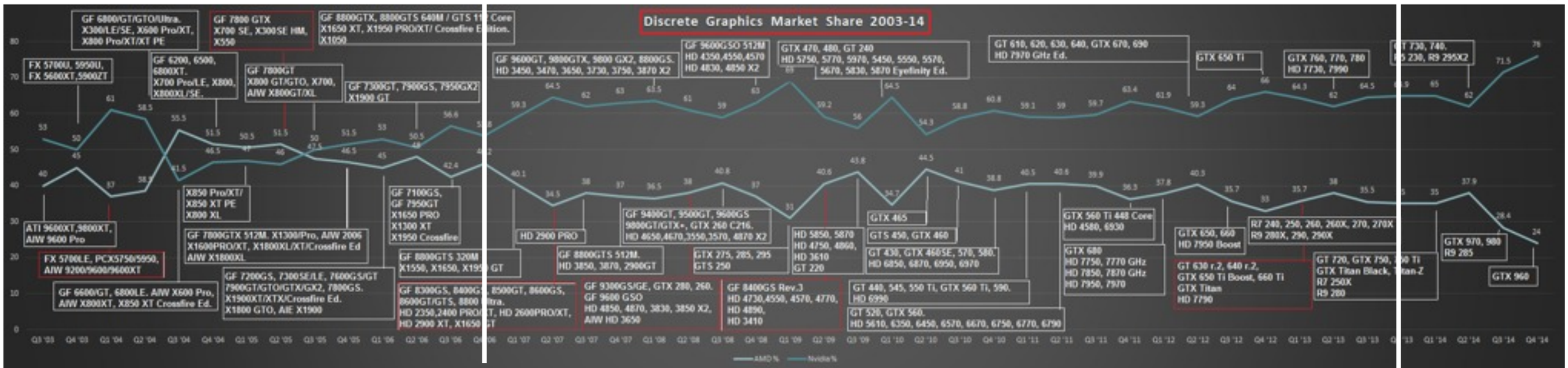


	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

Pre-CUDA era: 1-1 | Stable period of 7 years: 2-1

Past:
The GPU market share

Source: Jon Peddie Research consulting

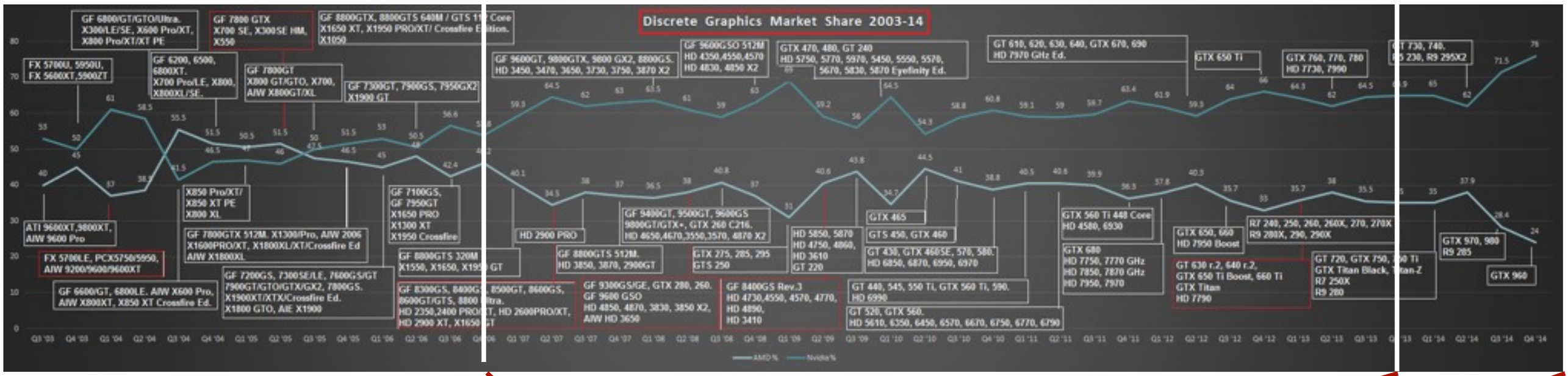


	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

Pre-CUDA era: 1-1 | Stable period of 7 years: 2-1 | 3-1

Past:
The GPU market share

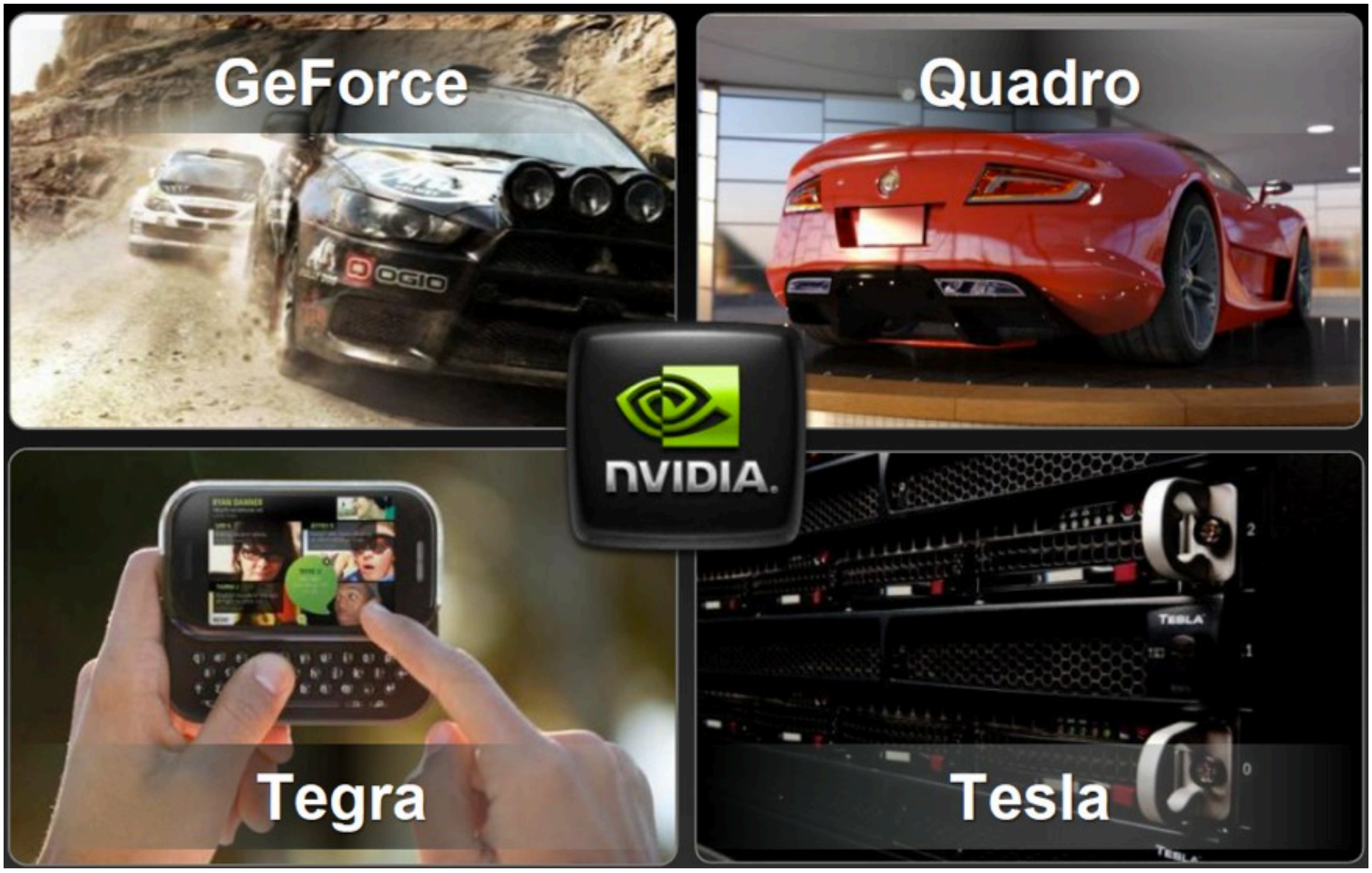
Source: Jon Peddie Research consulting



	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	1Q15	2Q15
Nvidia	50%	53%	51%	55%	63%	63%	64%	61%	64%	66%	65%	76%	77%	81%
AMD	45%	46%	45%	46%	37%	37%	36%	39%	36%	33%	35%	24%	22%	18%

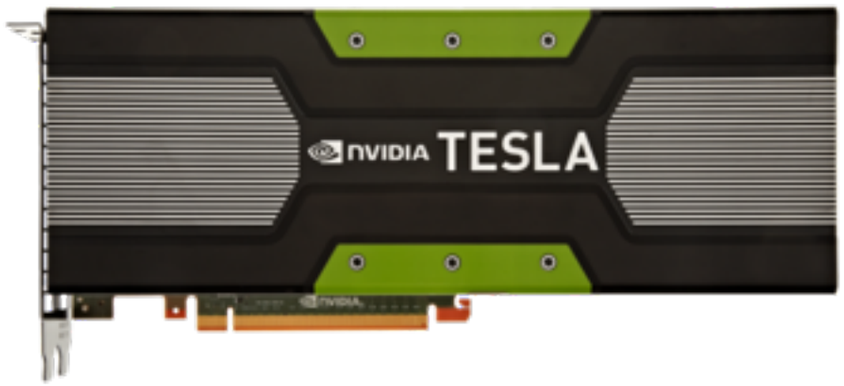
Pre-CUDA era: 1-1 | Stable period of 7 years: 2-1 | 3-1 | 4-1

Welcome to the GPU world



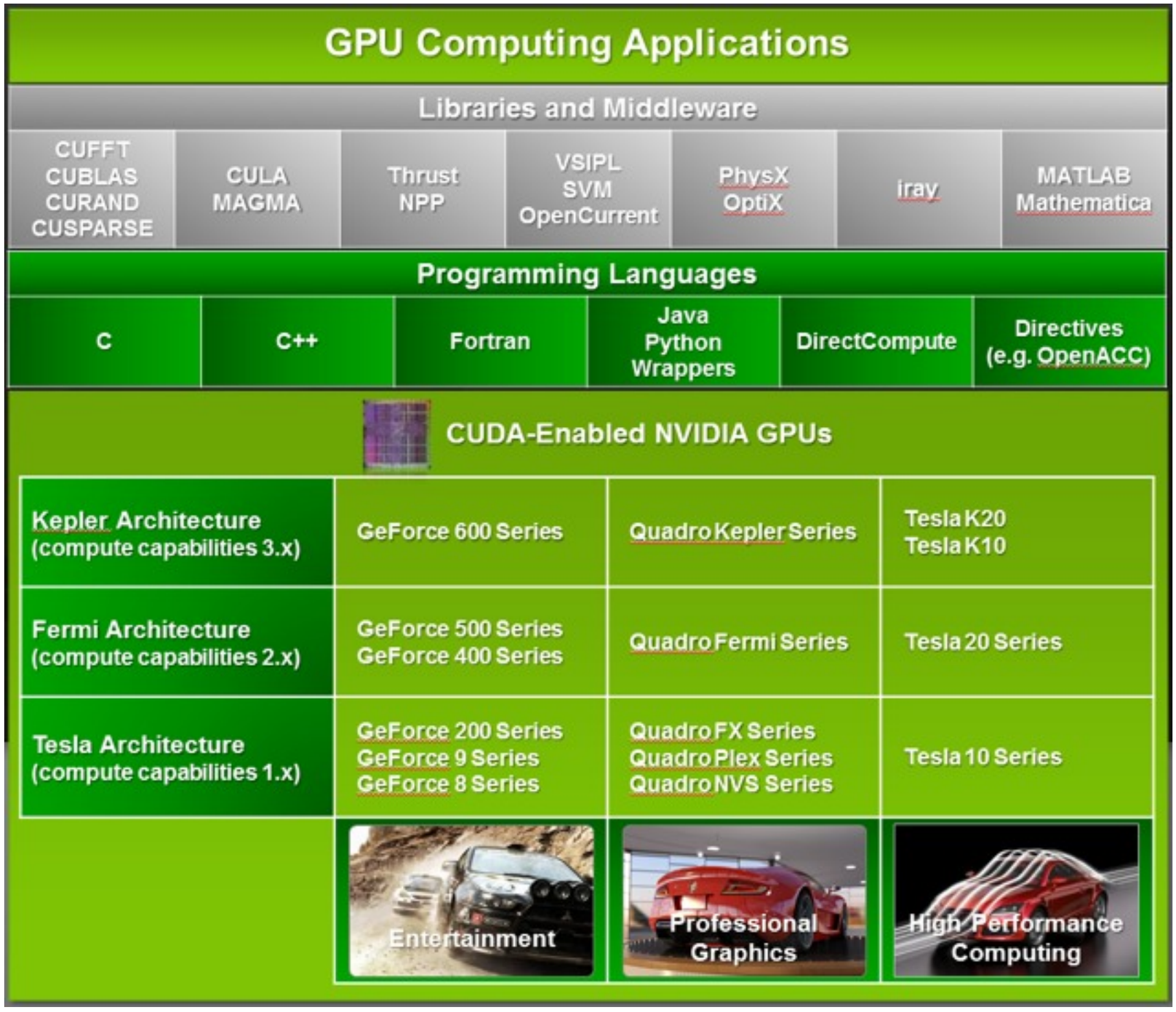
Commercial models available for Kepler: GeForce vs. Tesla

- Designed for gamers:
  - Price is a priority (<500€).
  - Availability and popularity.
  - Small video memory (1-2 GB.).
  - Frequency slightly ahead.
  - Hyper-Q only for CUDA streams.
  - 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).
  - Hyper-Q for MPI.
  - GPUDirect (RDMA) and other features for GPU clusters.

The characters of this story: The CUDA family picture



The impressive evolution of CUDA

Year 2008

100.000.000 CUDA-capable GPUs (6.000 Teslas only)

150.000 CUDA downloads

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)

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)

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)

The impressive evolution of CUDA

Year 2008	Year 2016
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)	63 supercomputers in TOP500.org Agregate: 80.000 TFLOPS (more than 14% of the 567 PFLOPs in top500)
60 university courses	840 university courses

The impressive evolution of CUDA

Year 2008	Year 2016
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)	63 supercomputers in TOP500.org Agregate: 80.000 TFLOPS (more than 14% of the 567 PFLOPs in top500)
60 university courses	840 university courses
4.000 academic papers	60.000 academic papers

Summary of GPU evolution

Summary of GPU evolution

- 2001: First many-cores (vertex and pixel processors).

Summary of GPU evolution

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

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.

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.

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.

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.

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.

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.

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

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**).

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.

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**.

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

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.

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.

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:

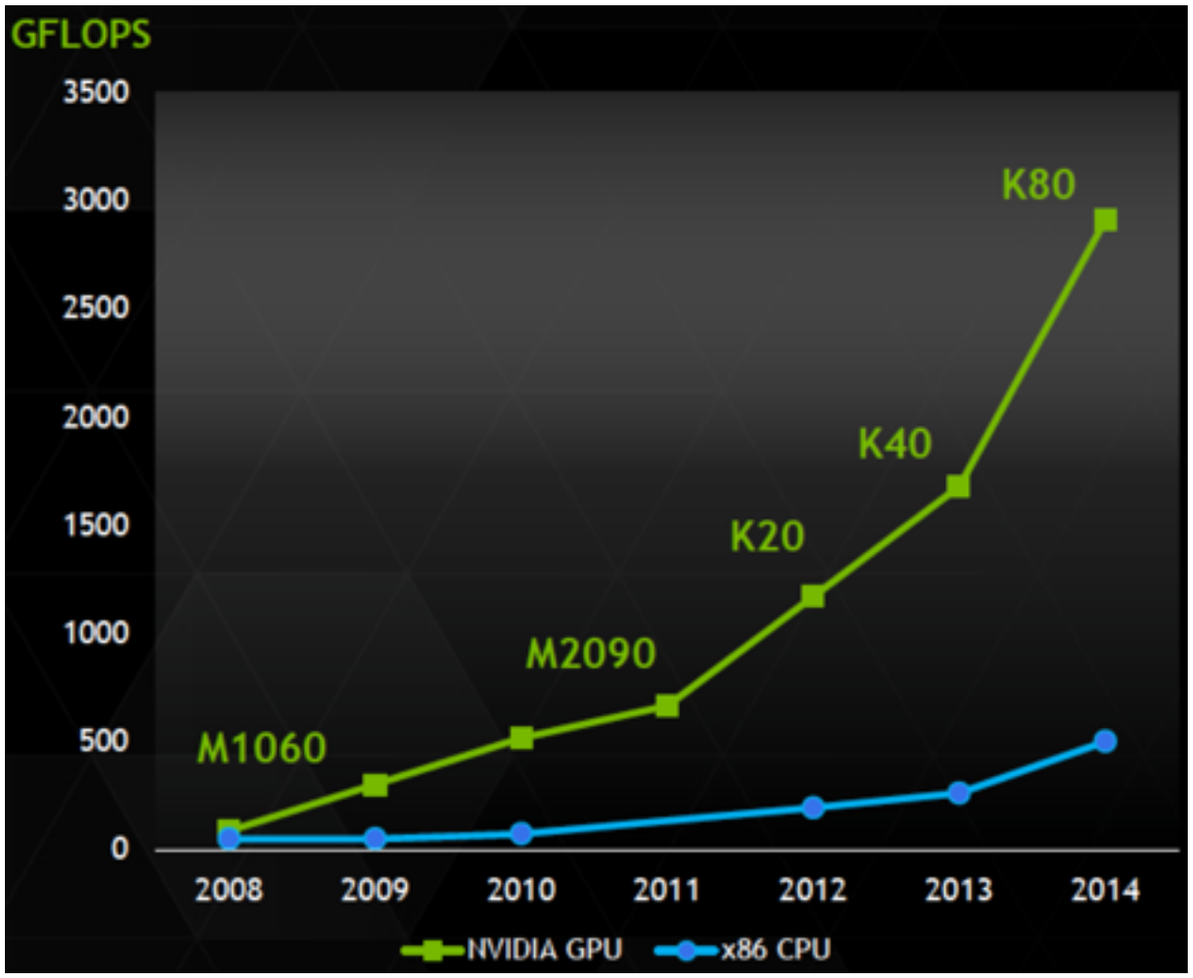
Three reason for feeling attracted to GPUs

- 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

GPU peak performance vs. CPU

Peak GFLOPS (fp64)



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

Memory Bandwidth

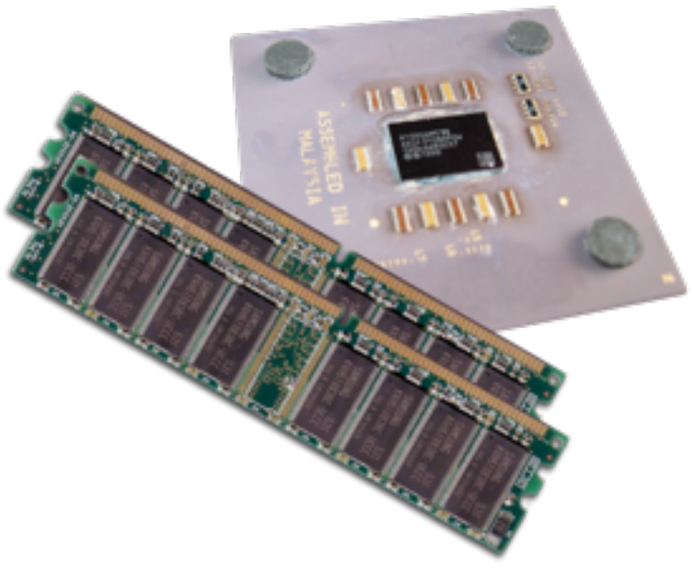
- GPU 6x more bandwidth:
  - 7 GHz x 48 bytes = 336 GB/s.
  - 2 GHz x 32 bytes = 64 GB/s.

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.

Heterogeneous Computing (1/3)

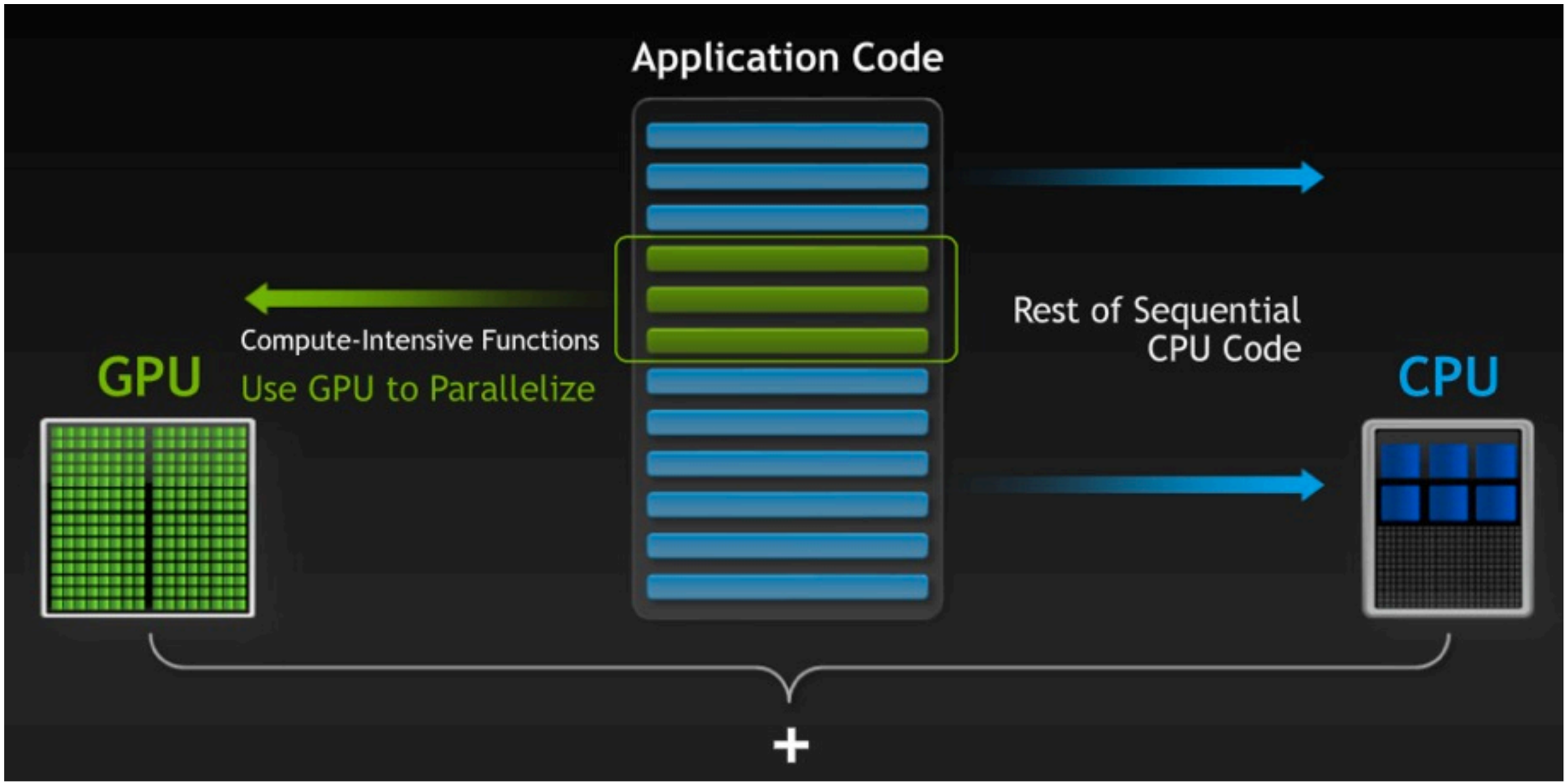
- Terminology:
  - Host: The CPU and the memory on motherboard [DDR3 as of 2013].
  - Device: The graphics card [GPU + video memory]:
    - GPU: Nvidia GeForce/Tesla.
    - Video memory: GDDR5 as of 2015.



Host

Device

Heterogeneous Computing (2/3)



- 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 (3/3)

```
#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;
}
```

Heterogeneous Computing (3/3)

```
#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;
}
```

HOST CODE:

- Serial code.

Heterogeneous Computing (3/3)

```
#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.

Heterogeneous Computing (3/3)

```
#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.

Heterogeneous Computing (3/3)

```
#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.

Heterogeneous Computing (3/3)

```
#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.

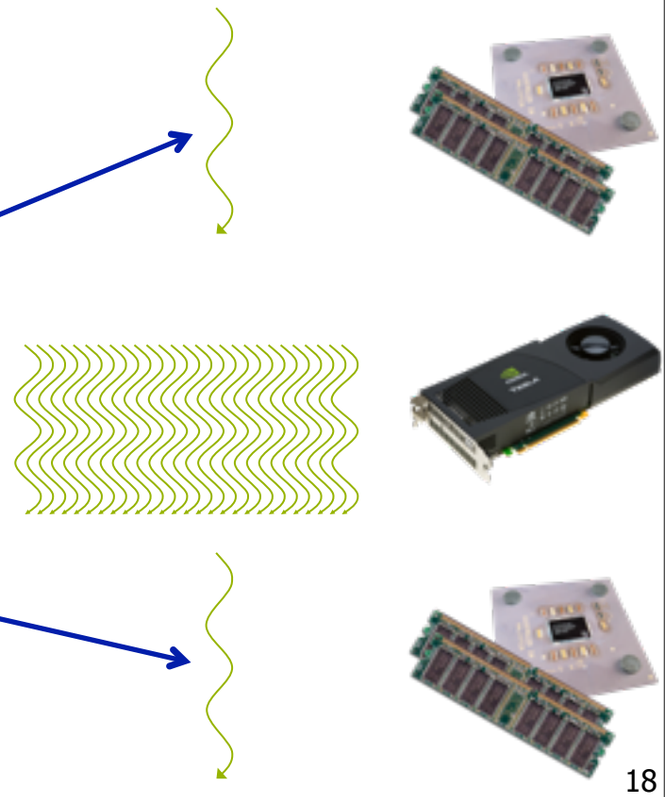
Heterogeneous Computing (3/3)

```
#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;
}
```

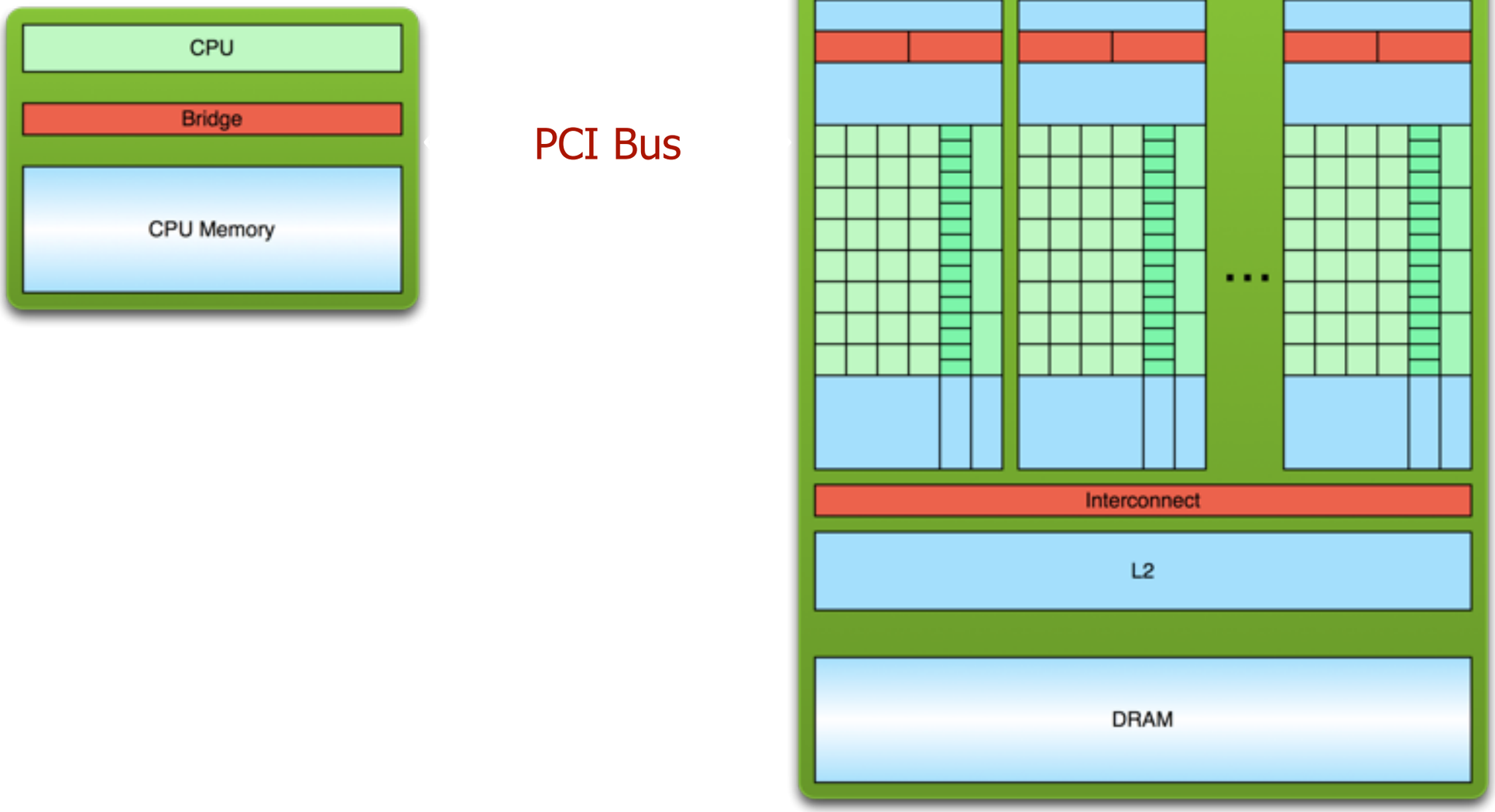
DEVICE CODE:
Parallel function written in CUDA.

HOST CODE:

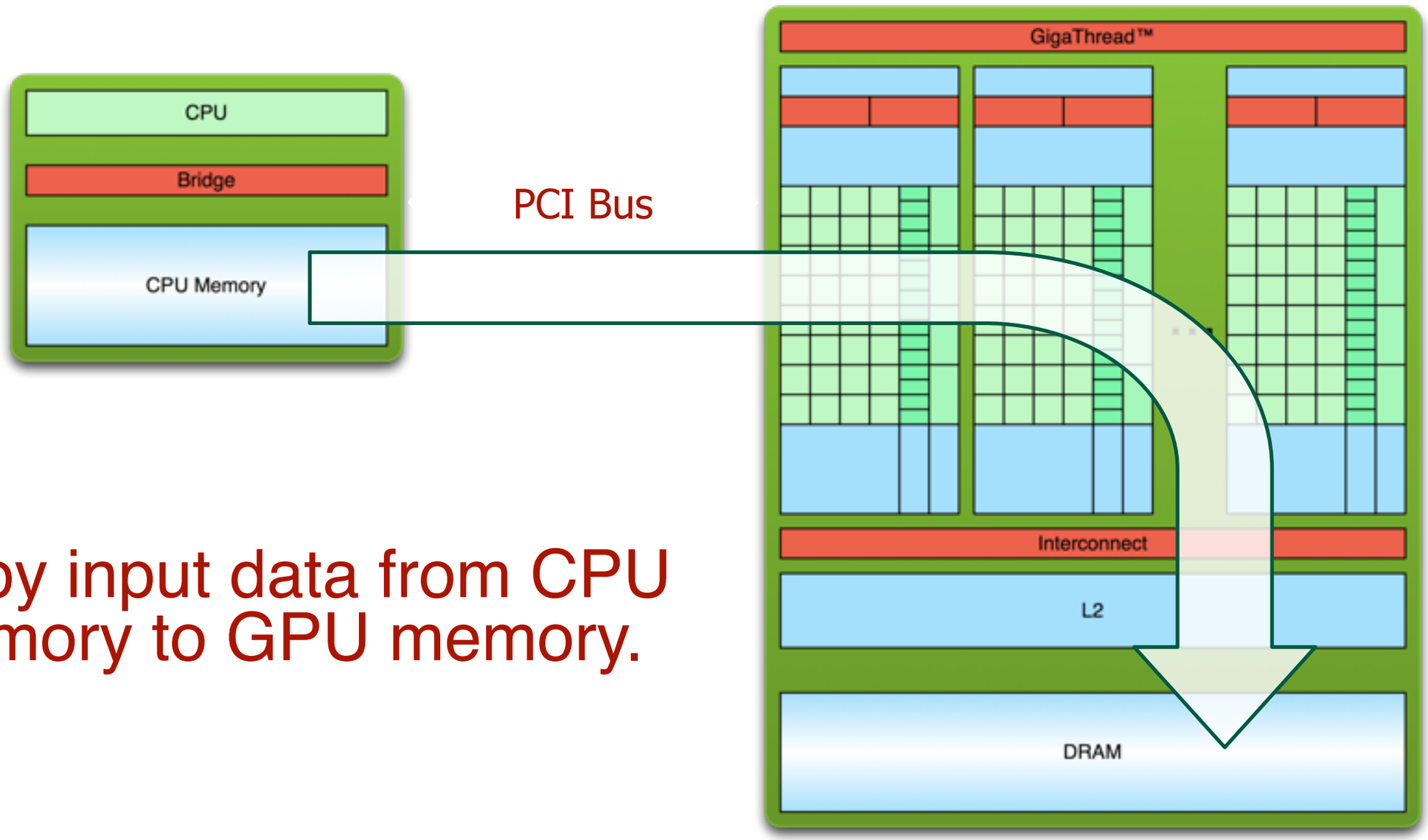
- Serial code.
- Parallel code.
- Serial code.



Simple Processing Flow (1/3)

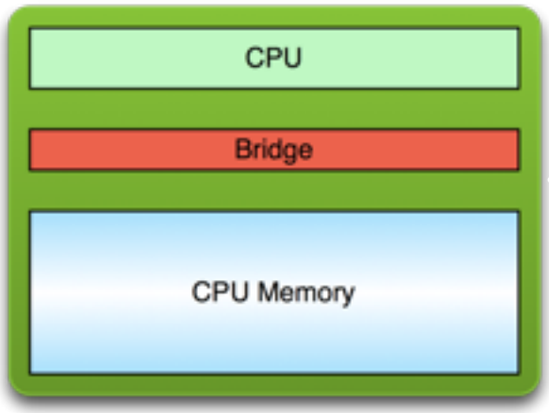


Simple Processing Flow (1/3)

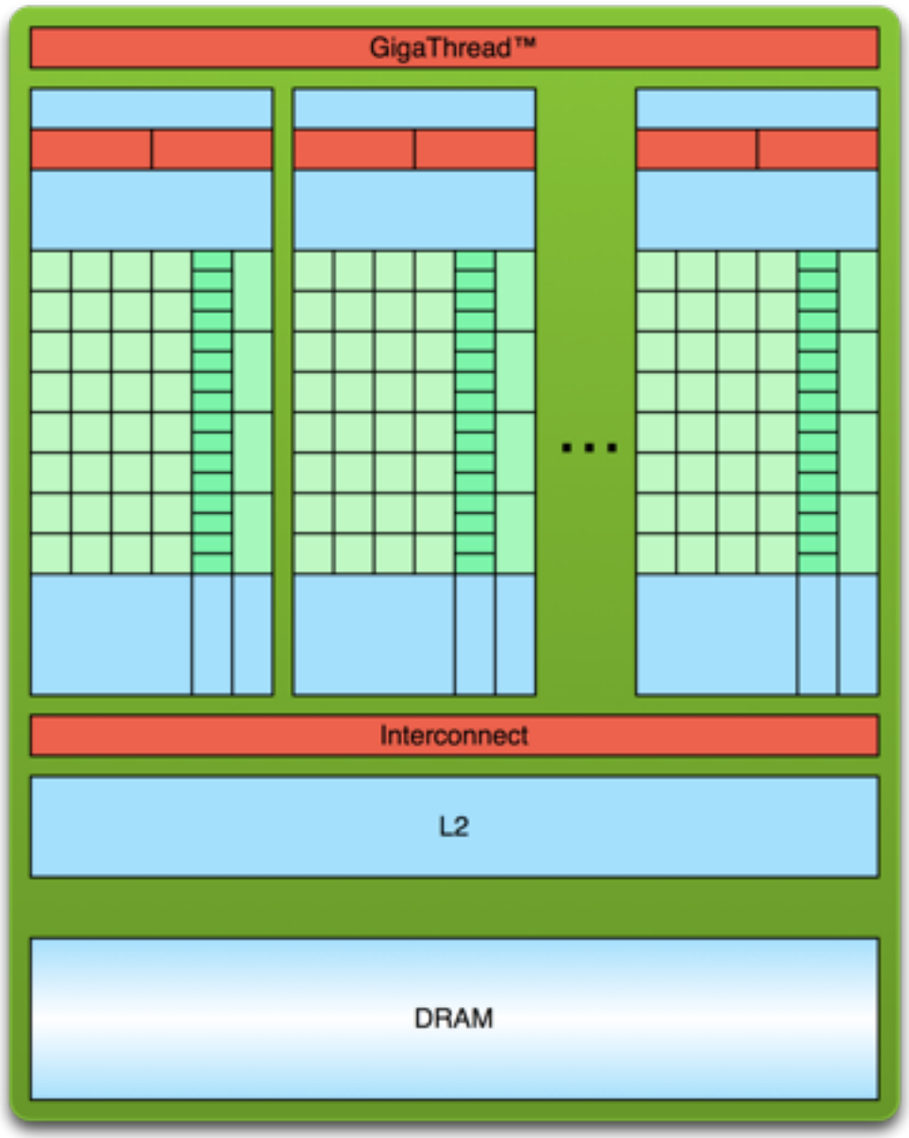


1. Copy input data from CPU memory to GPU memory.

Simple Processing Flow (2/3)

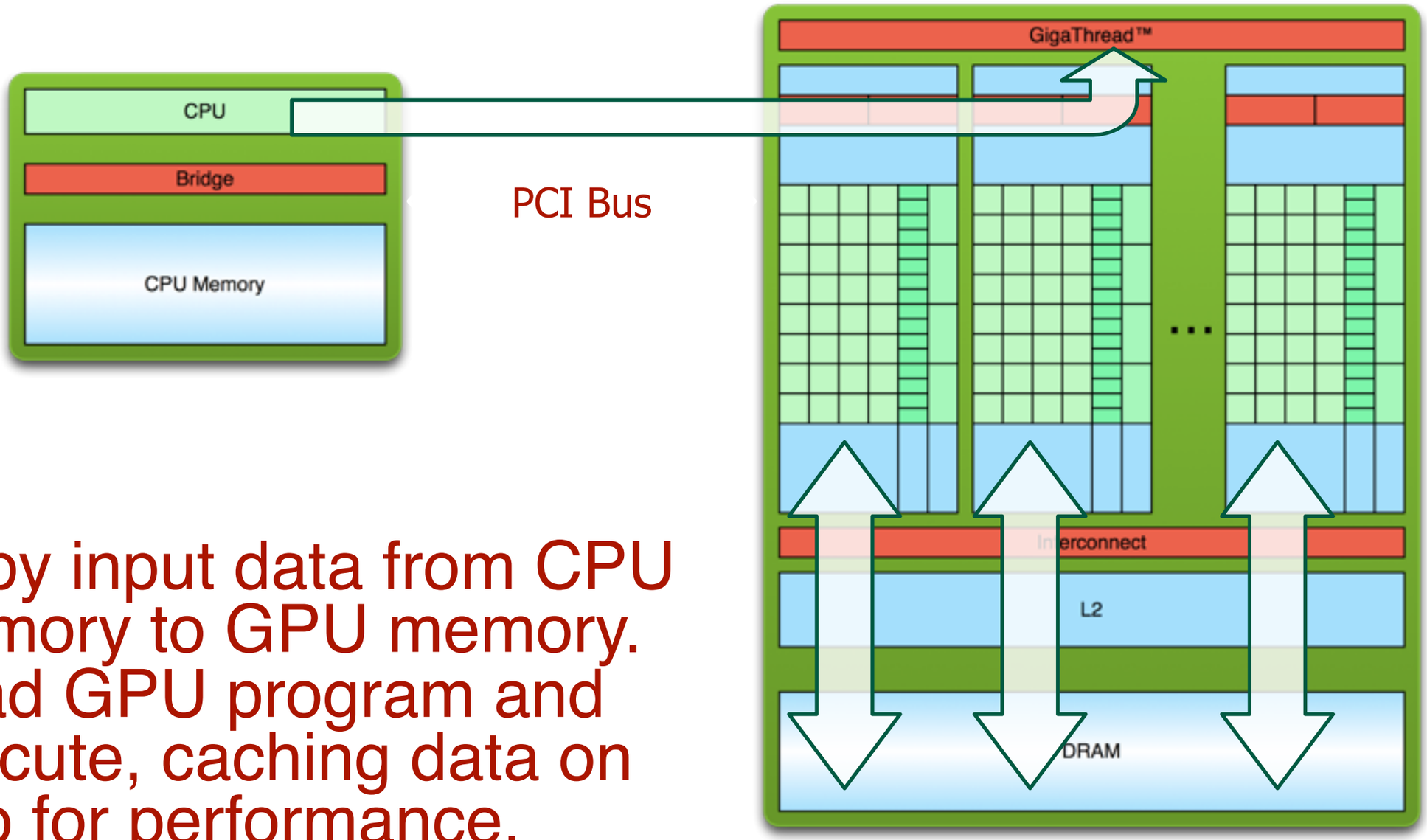


PCI Bus



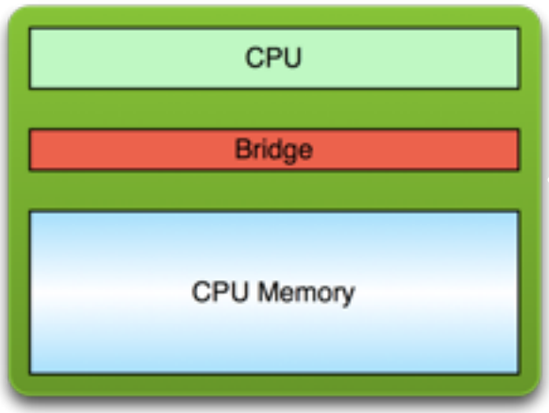
1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.

Simple Processing Flow (2/3)

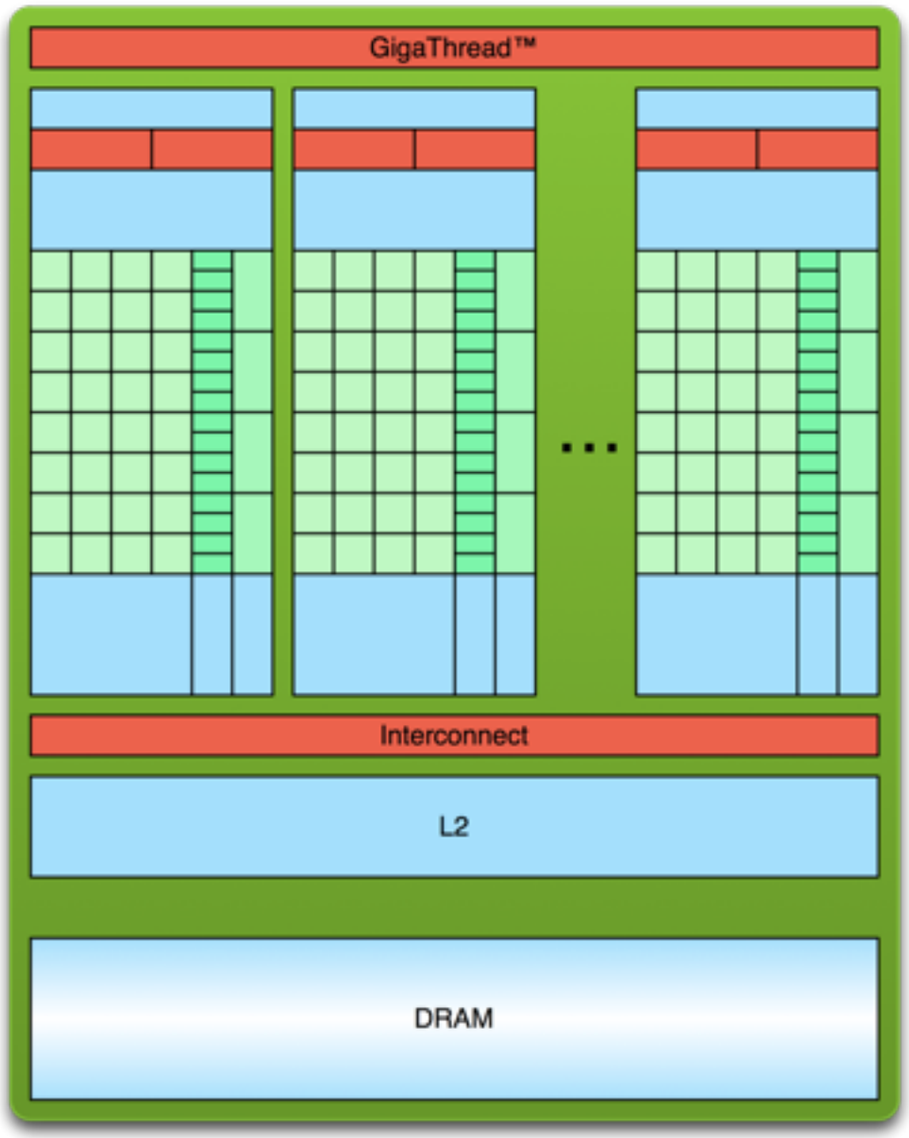


1. Copy input data from CPU memory to GPU memory.
2. Load GPU program and execute, caching data on chip for performance.

Simple Processing Flow (3/3)

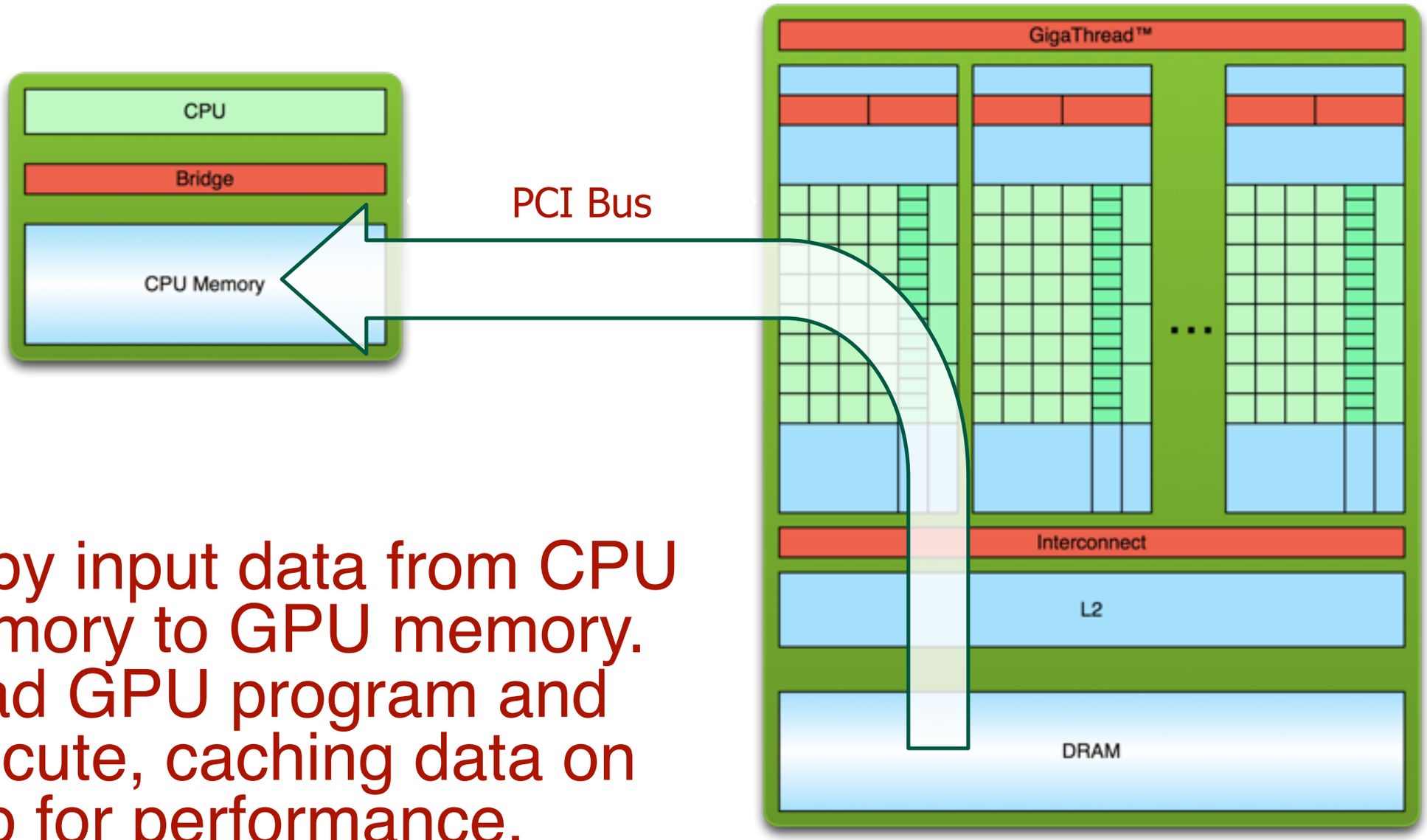


PCI Bus



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.

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;
}
```

The classic example

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

Output:

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

The classic example

```
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;
}
```

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!

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;
}
```

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!
$
```

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.

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.

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”.

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 ...''

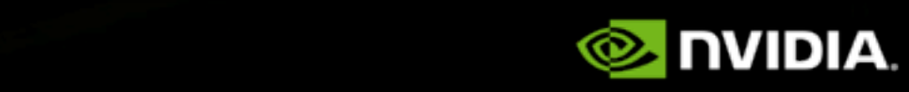
David A. Patterson & John Hennessy

Organization and Computer Design

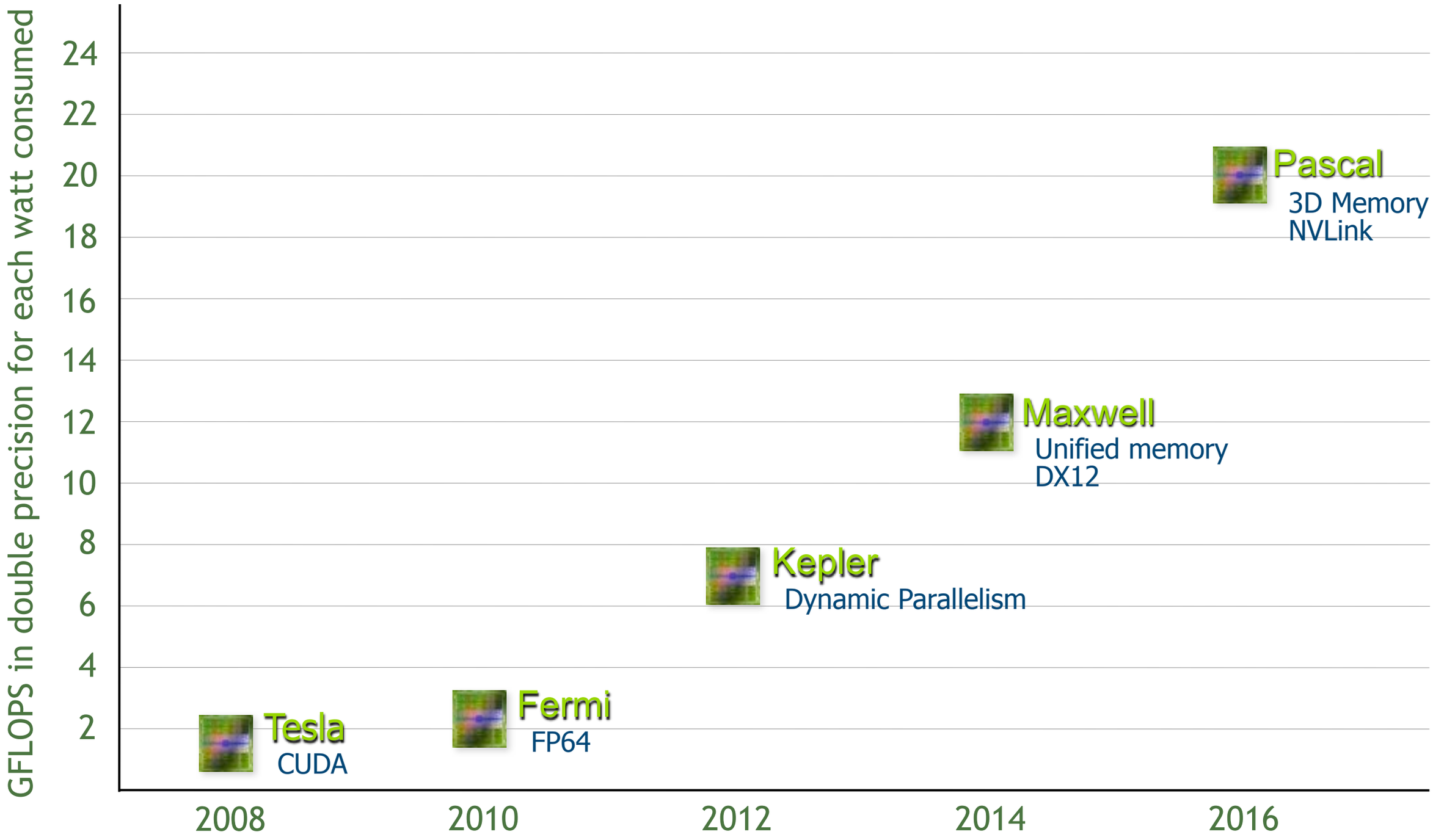
Mc-Graw-Hill (1995)

Chapter 9, page 569

II.1. CUDA hardware model

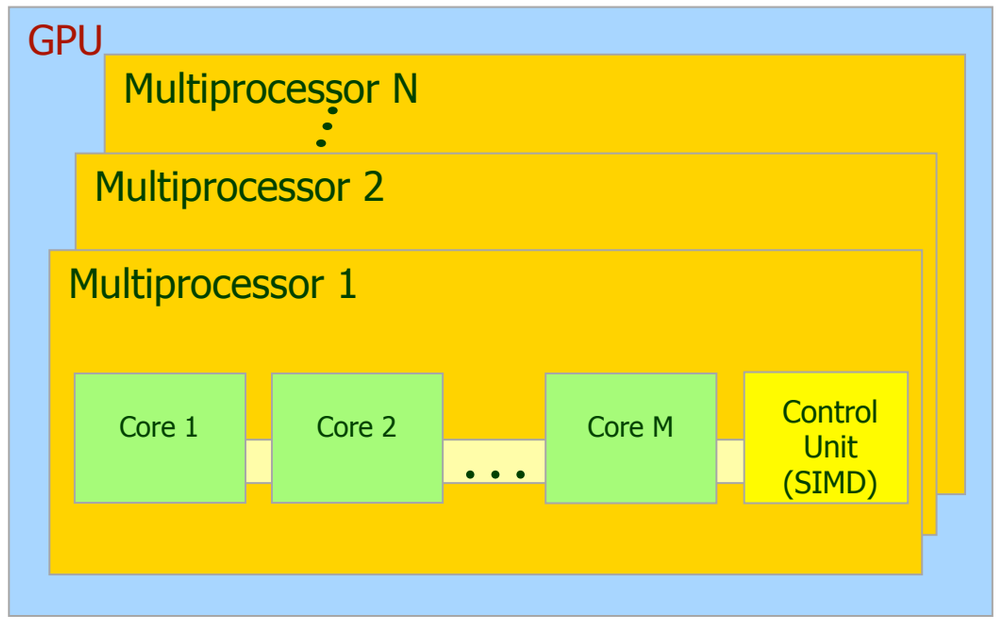


Overview of CUDA hardware generations



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

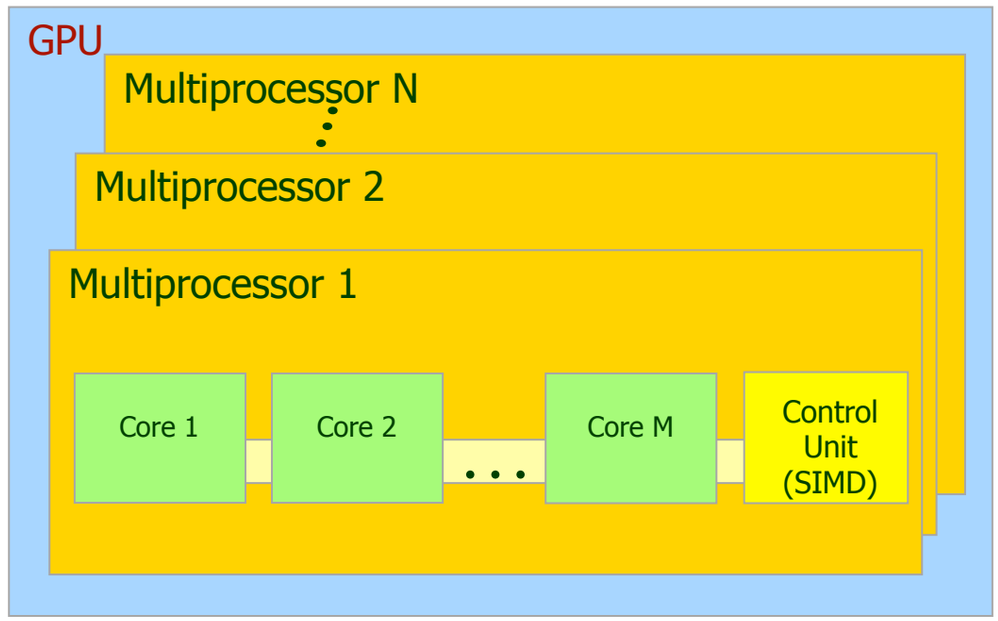
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



	G80 (Tesla)
Period	2006-07
N (multip.)	16
M (cores/mult.)	8
# cores	128

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

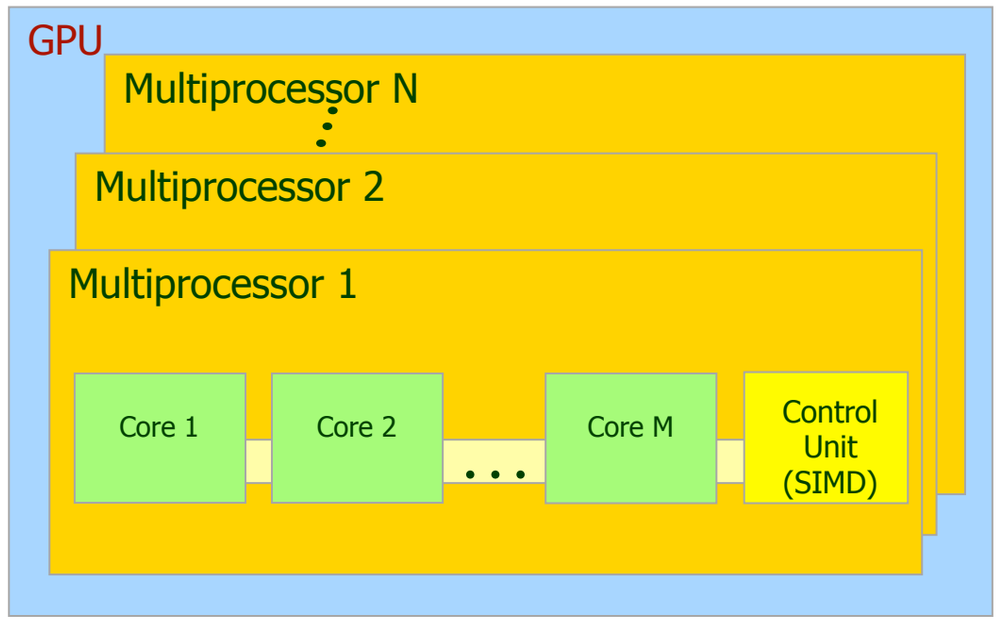
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



	G80 (Tesla)	GT200 (Tesla)
Period	2006-07	2008-09
N (multip.)	16	30
M (cores/mult.)	8	8
# cores	128	240

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

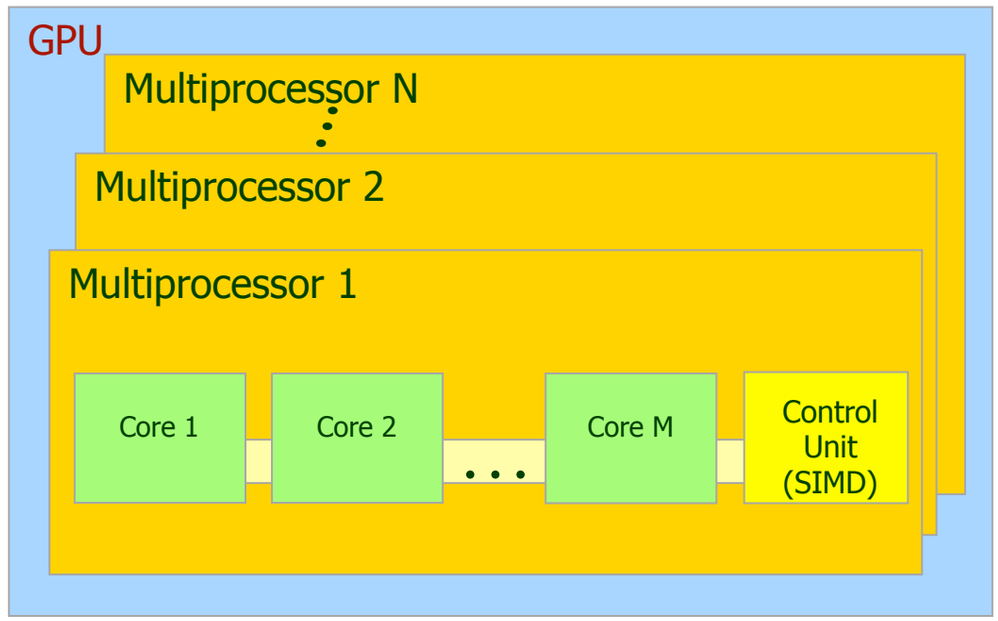
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



	G80 (Tesla)	GT200 (Tesla)	GF100 (Fermi)
Period	2006-07	2008-09	2010-11
N (multip.)	16	30	14-16
M (cores/mult.)	8	8	32
# cores	128	240	448-512

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

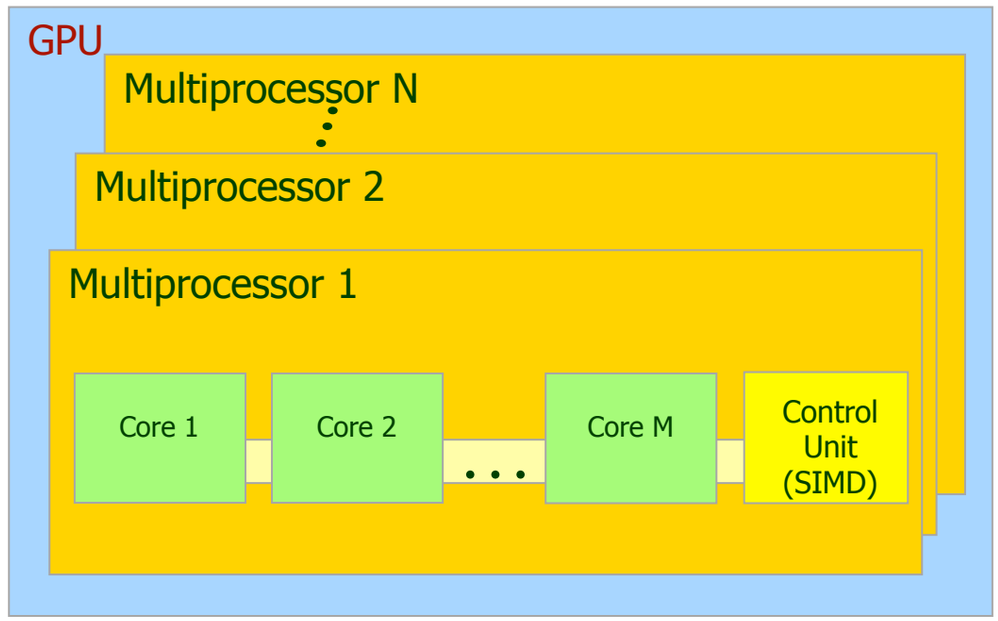
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



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

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

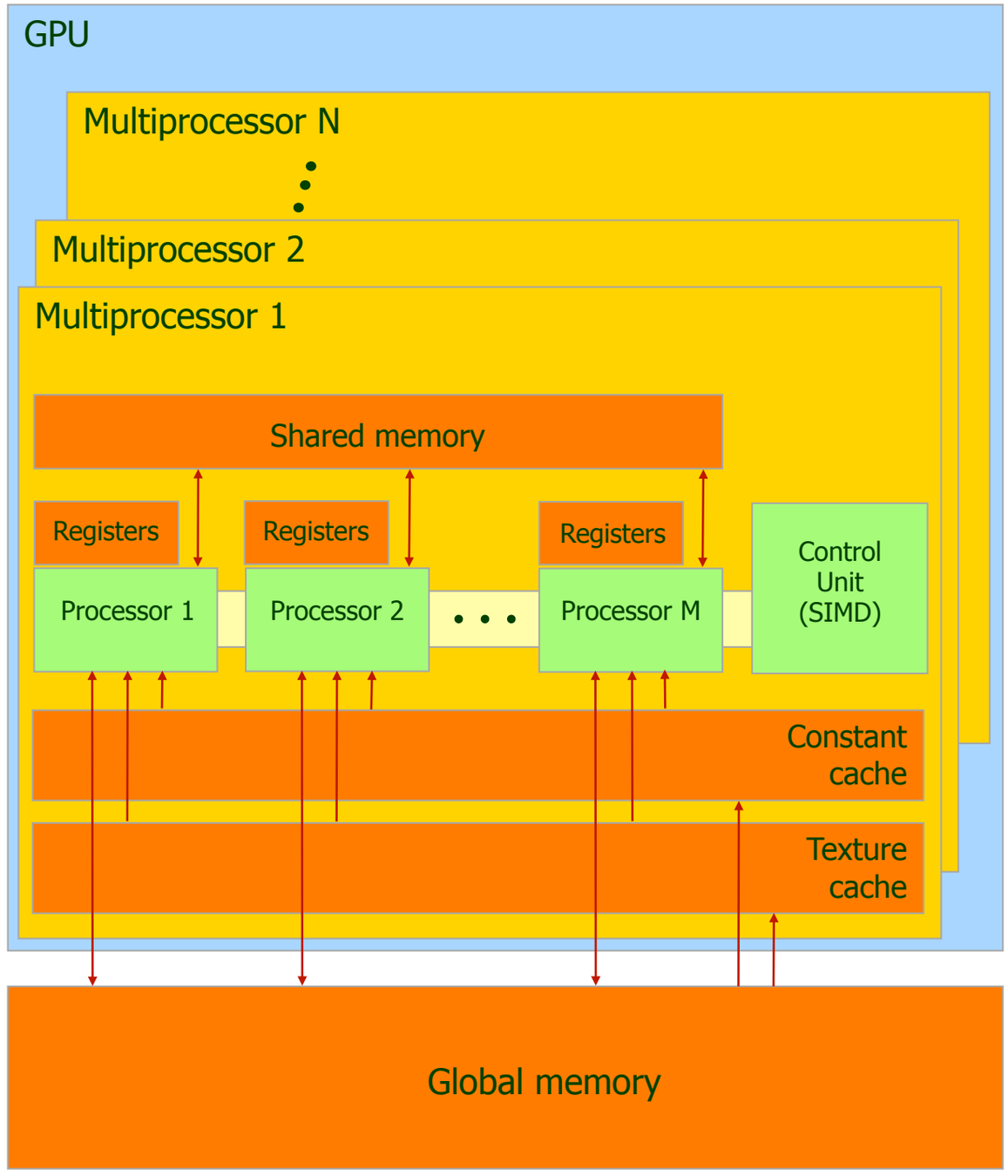
- A GPU consists of:
  - N multiprocessors (or SMs), each containing M cores (or stream procs).
- Massive parallelism:
  - Applied to thousands of threads.
  - Sharing data at different levels.
- Heterogeneous computing:
  - GPU:
    - Data intensive.
    - Fine-grain parallelism.
  - CPU:
    - Control/management.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



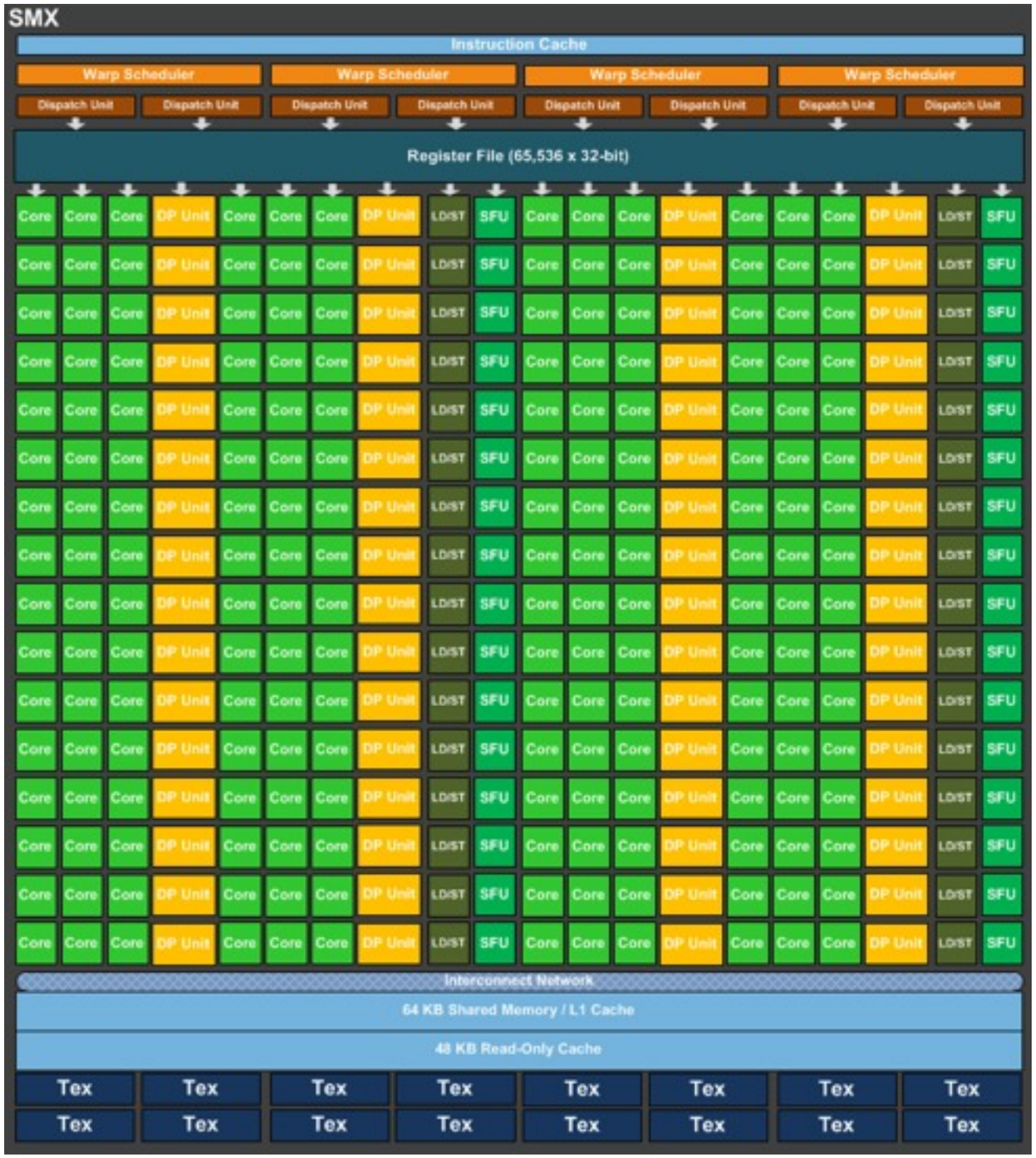
II. 4. The third generation: Kepler (GKxxx)

NVIDIA

Kepler GK110 Block Diagram

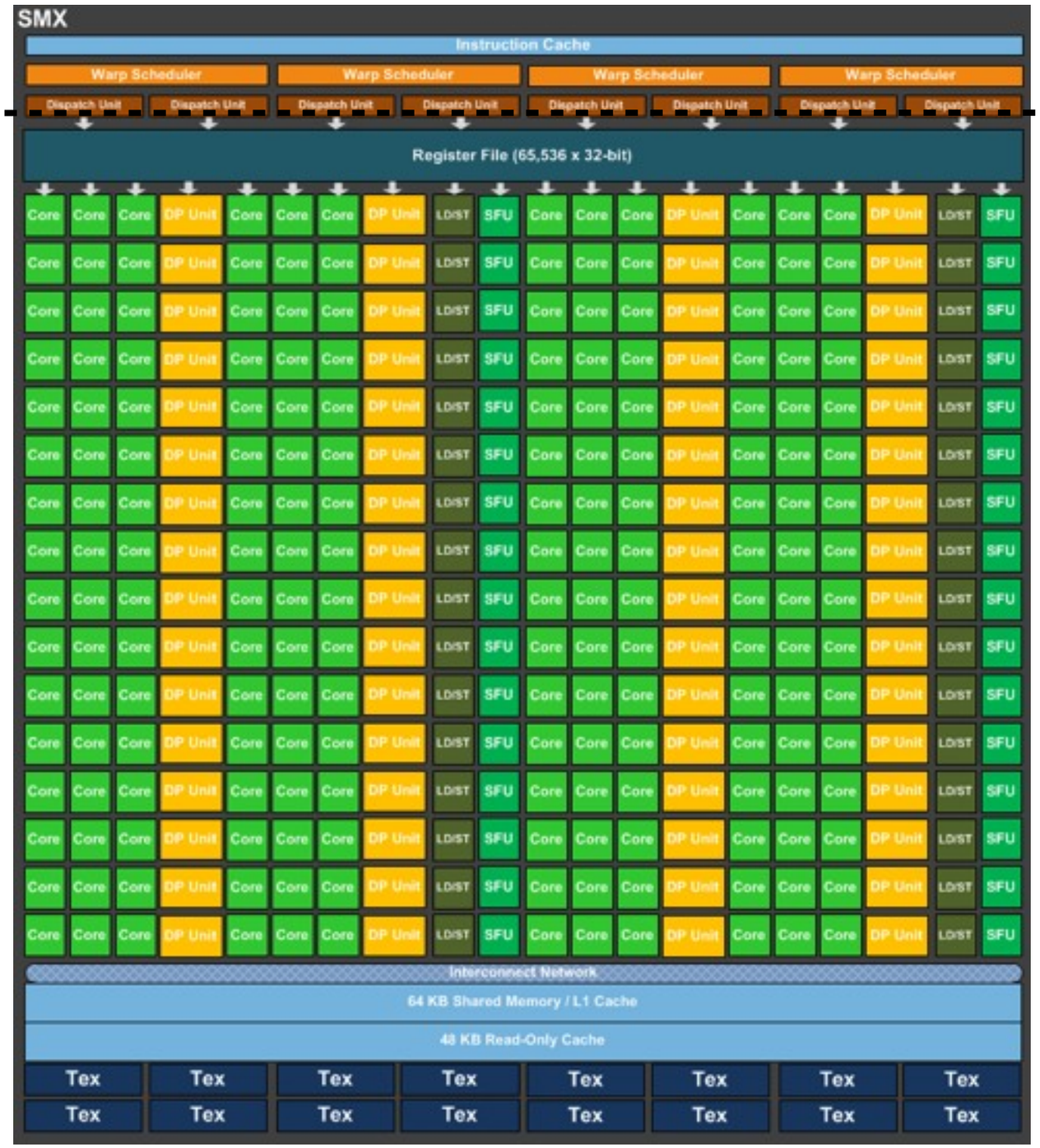
- 7.1 billion transistors.
- 15 SMX multiprocs.
- > 1 TFLOP FP64.
- 1.5 MB L2 Cache.
- 384-bit GDDR5.
- PCI Express Gen3.

The SMX multiprocessor



The SMX multiprocessor

Instruction scheduling and issuing in **warps**

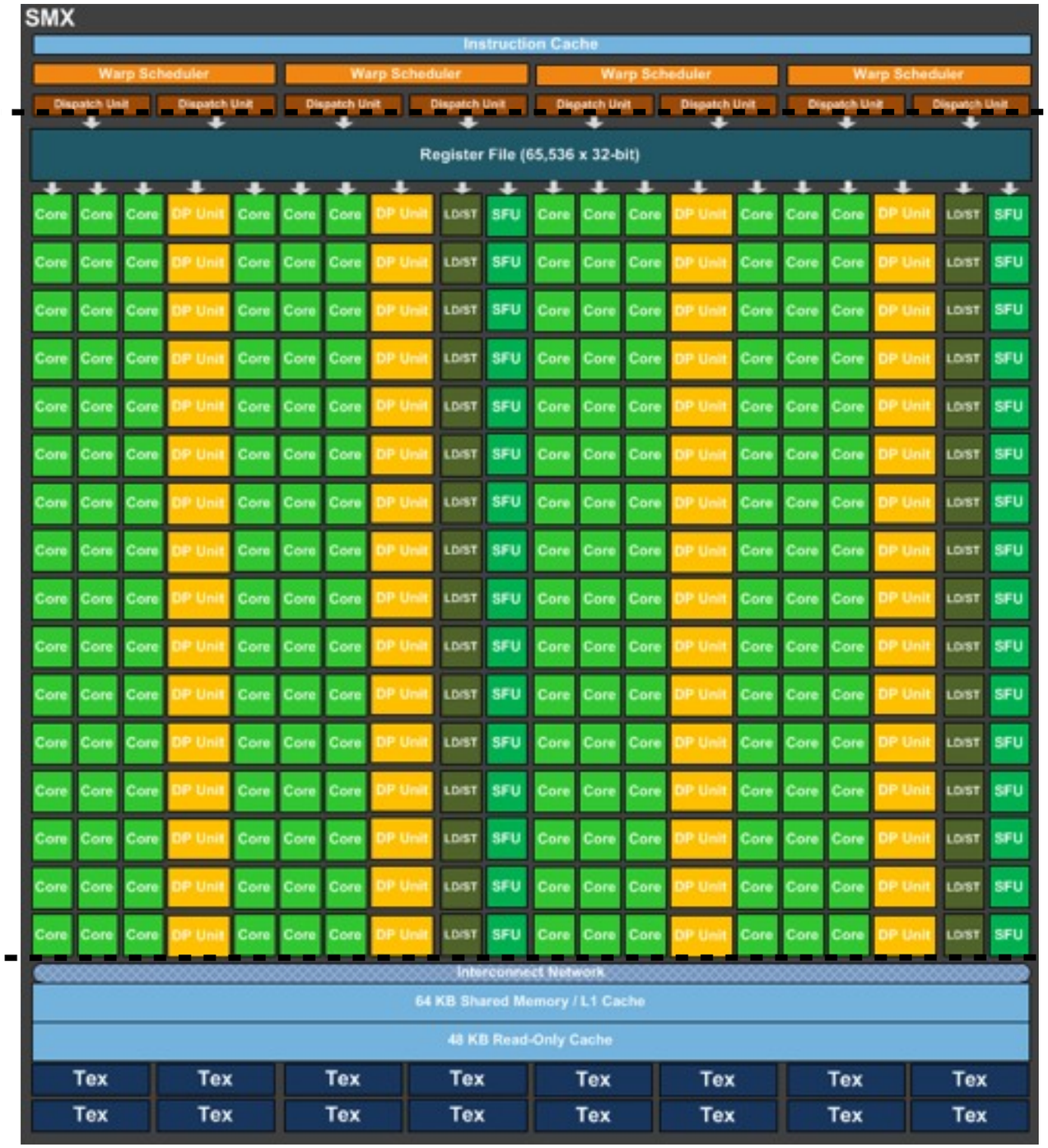


Front-end

The SMX multiprocessor

Instruction scheduling and issuing in **warps**

Front-end



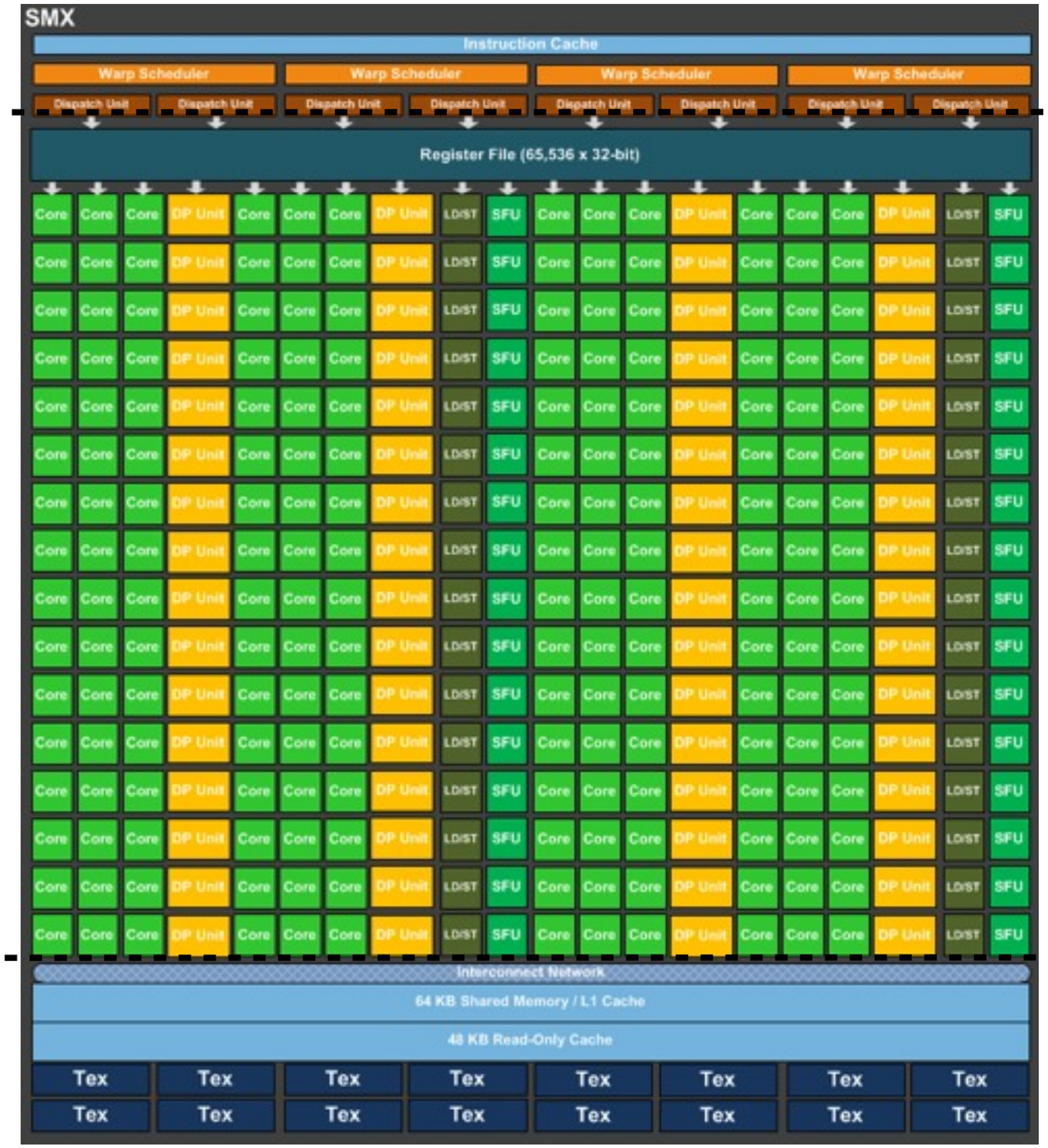
Instructions execution.
512 functional units:
- 192 for ALUs.
- 192 for FPUs S.P.
- 64 for FPUs D.P.
- 32 for load/store.
- 32 for SFUs (log,sqrt, ...)

Back-end

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

SMX Balance of Resources

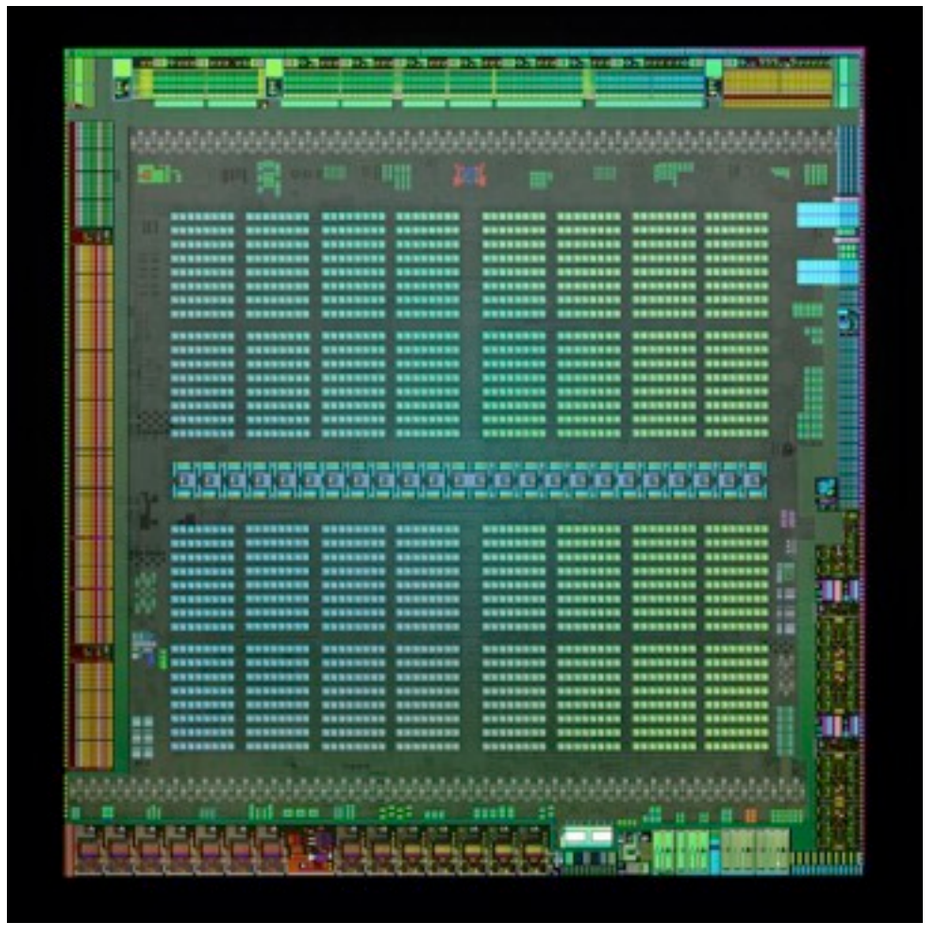
Resource	Kepler GK110 vs. Fermi GF100
Floating-point throughput	2-3x
Maximum number of blocks per SMX	2x
Maximum number of threads per SMX	1.3x
Register file bandwidth	2x
Register file capacity	2x
Shared memory bandwidth	2x
Shared memory capacity	1x
L2 bandwidth	2x
L2 cache capacity	2x

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

NVIDIA

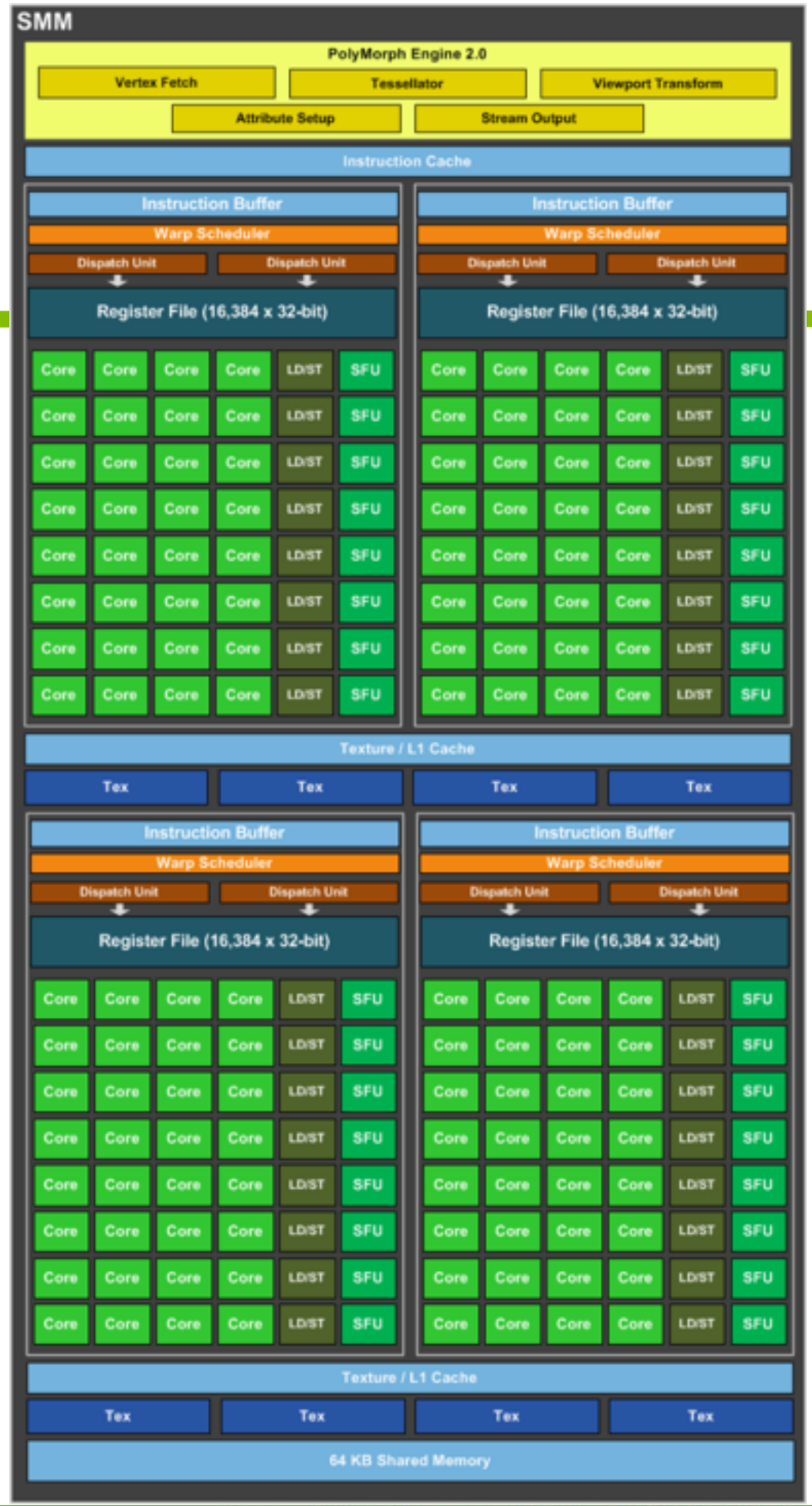
Maxwell and SMM multiprocessors (for GeForce GTX 980, 16 SMMs)

- 1870 Mt.
- 148 mm^2.



The SMMs

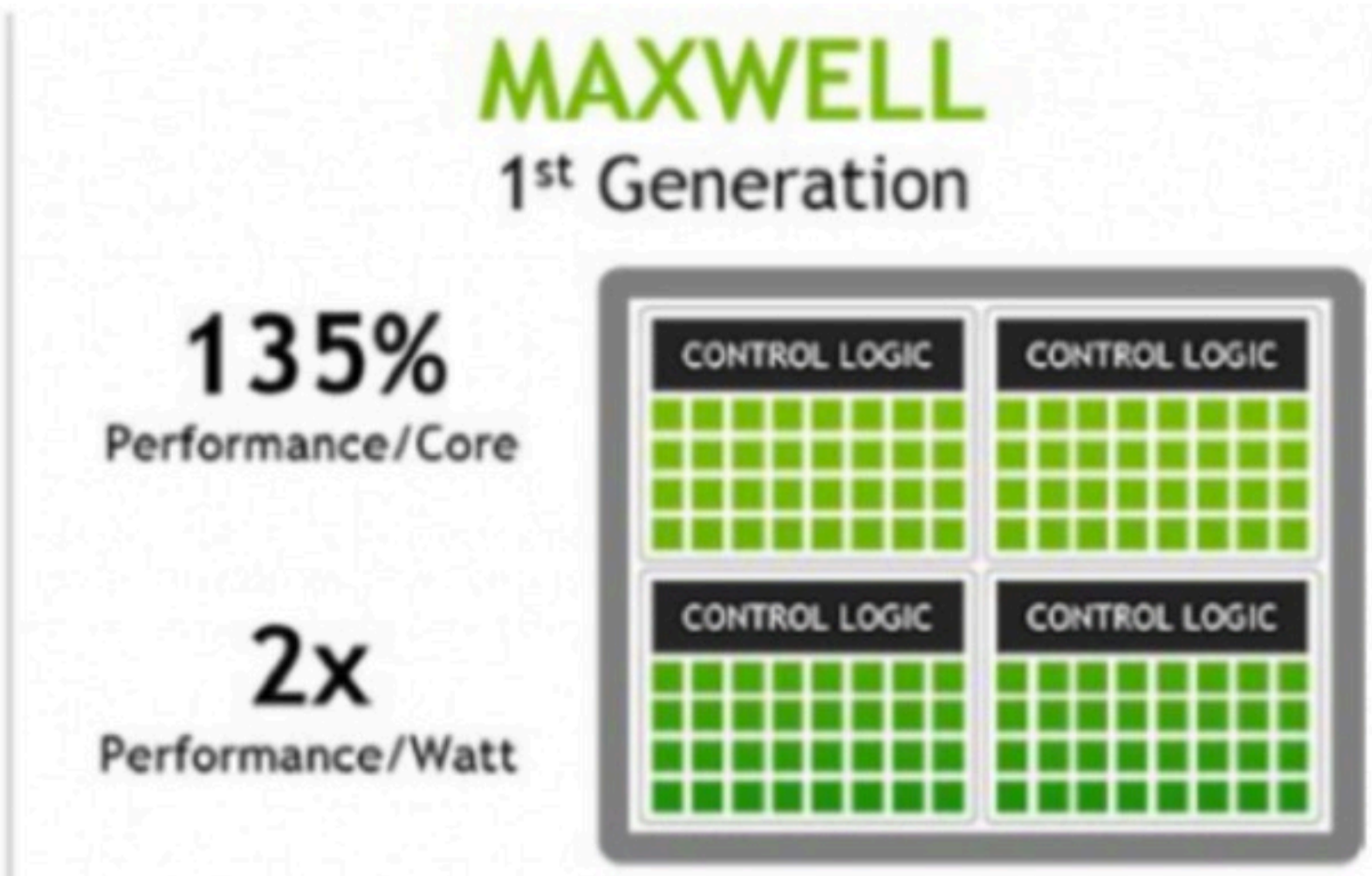
- Keep the same 4 warp schedulers, and the same LD/ST and SFU units.
- Reduce the number of cores for `int` and `float`: from 192 to 128 units.



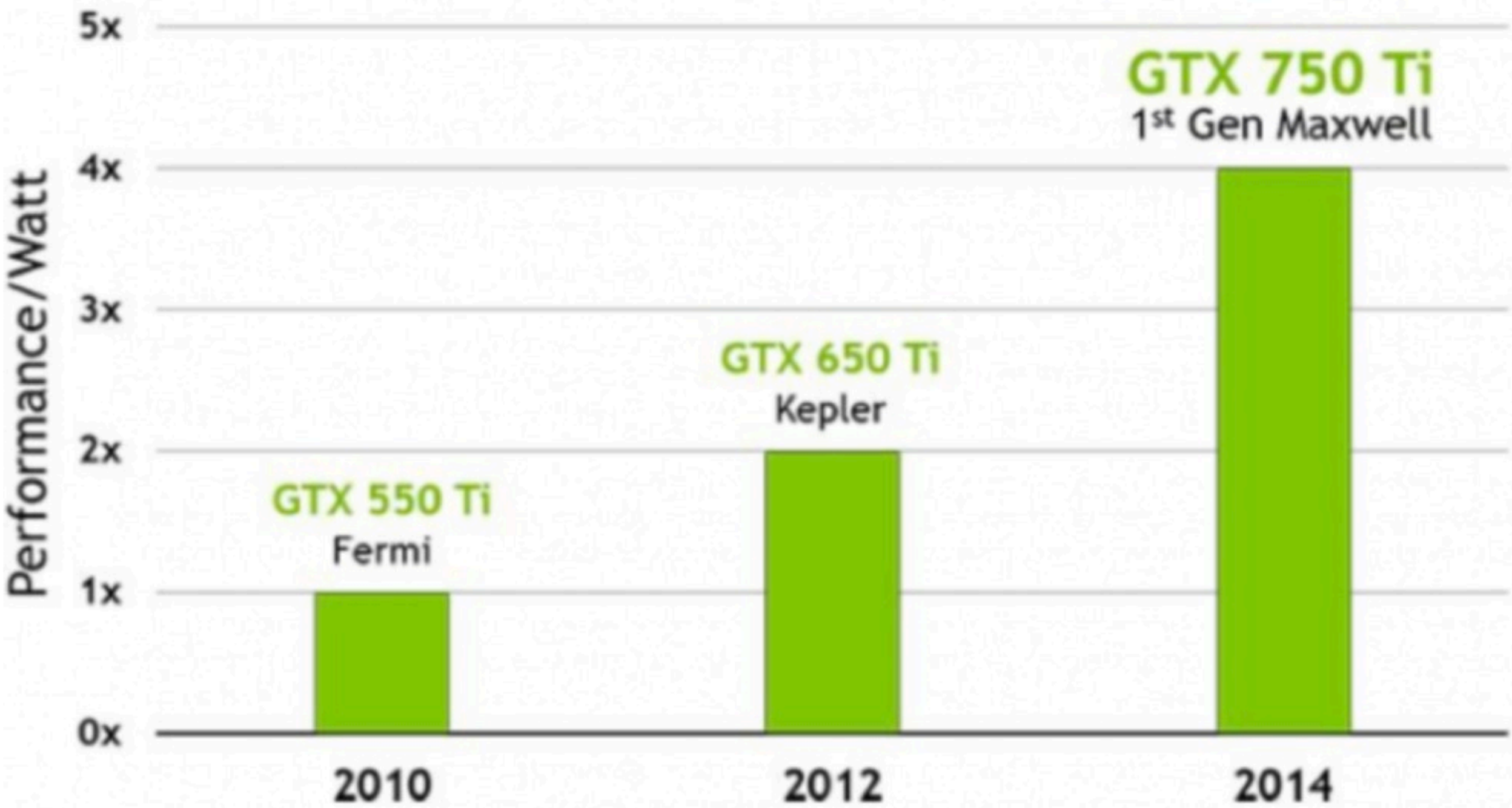
Some commercial models for CCC 5.2 (all @ 28 nm)

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



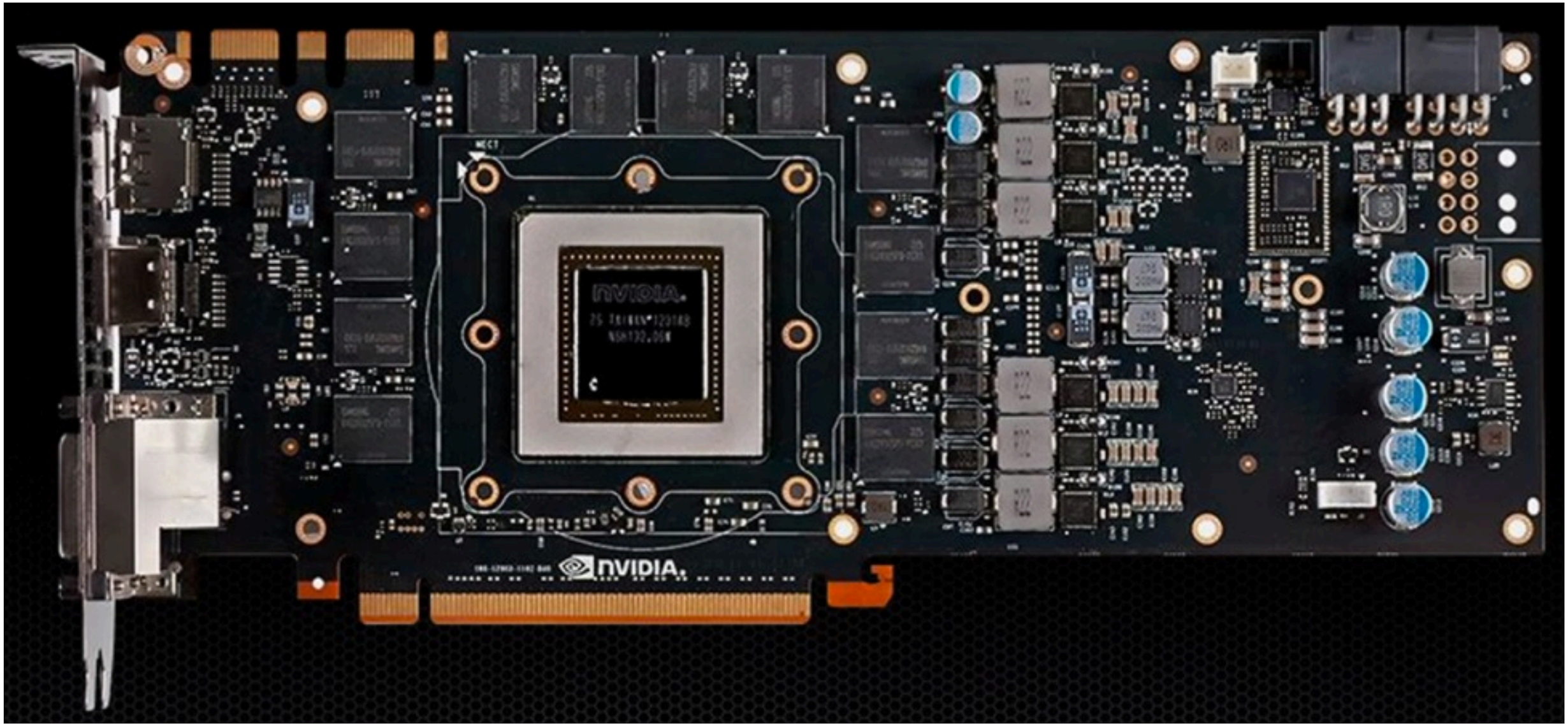
Power efficiency



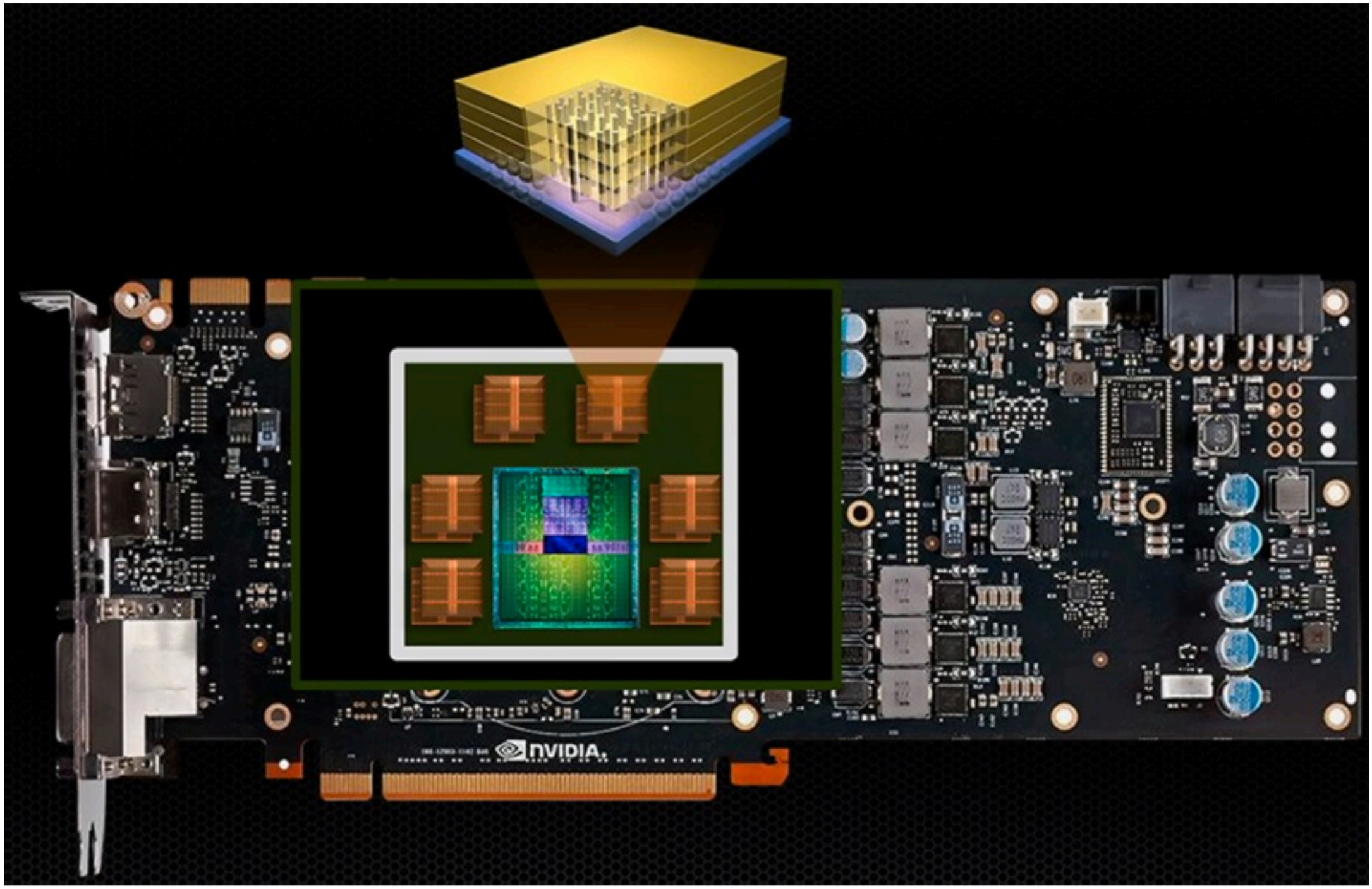
II. 6. The fifth generation: Pascal (GPxxx)

NVIDIA

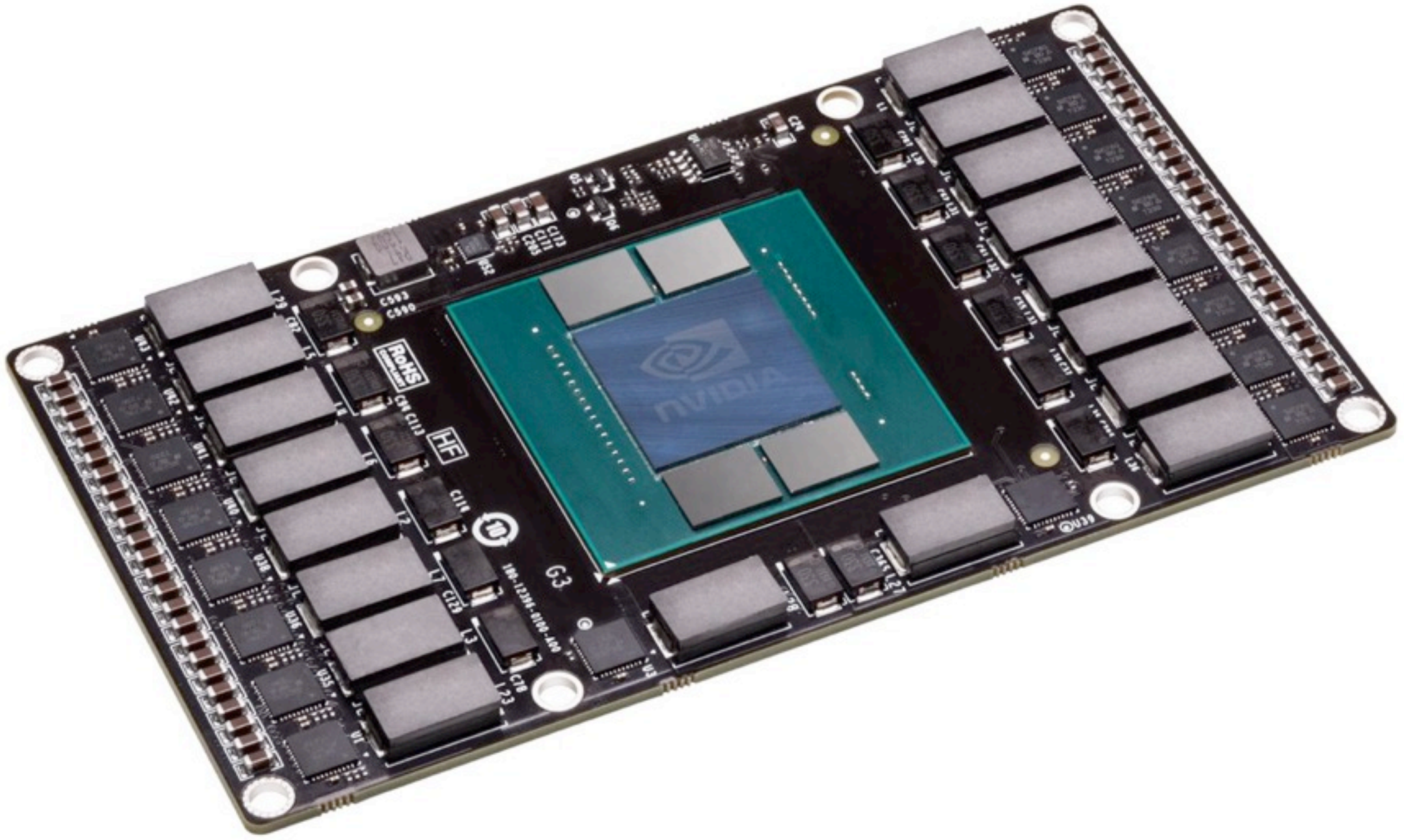
A 2015 graphics card:
Kepler/Maxwell GPU with GDDR5 memory



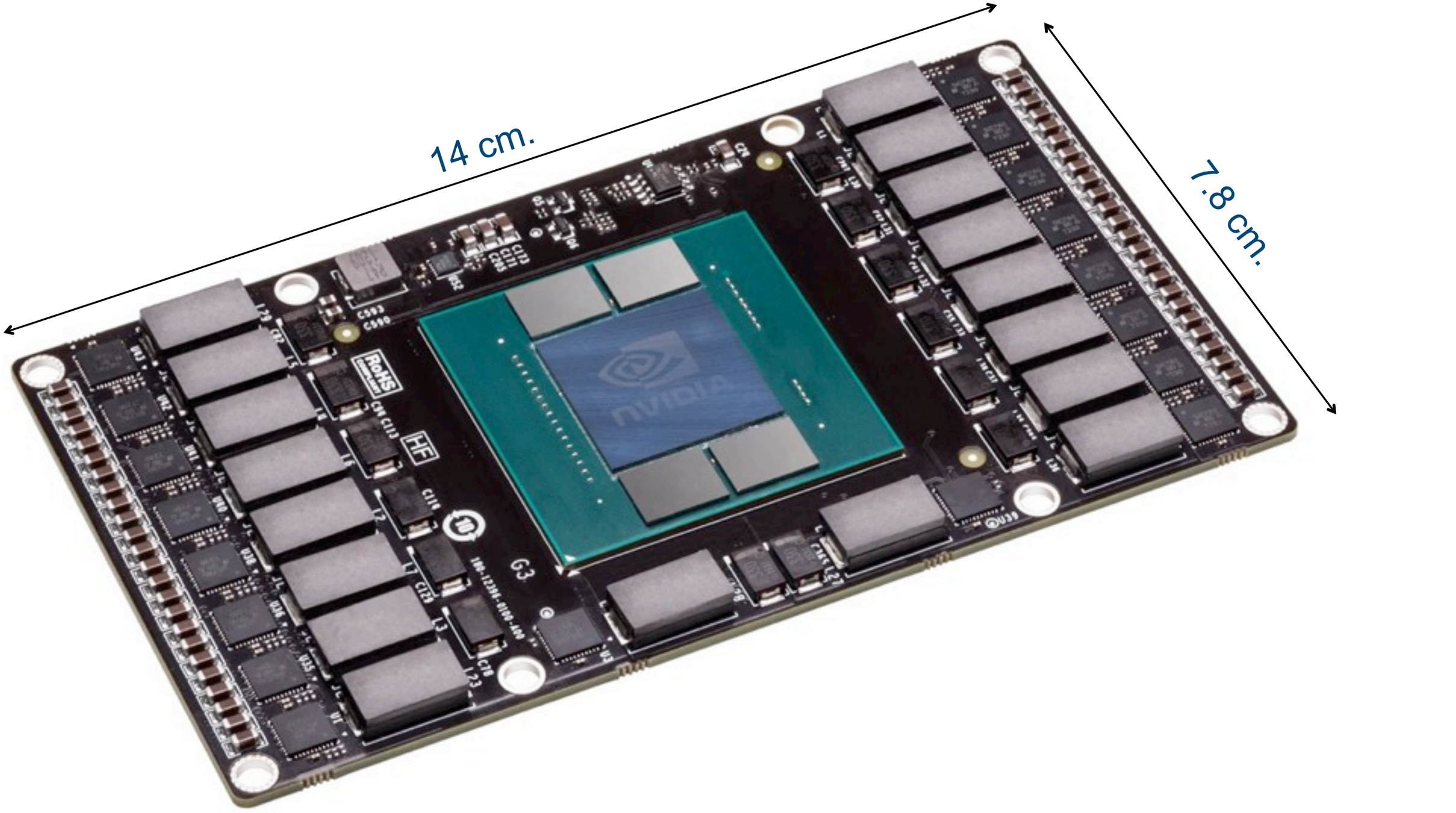
A 2016 graphics card: Pascal GPU with Stacked DRAM



A Pascal GPU prototype



A Pascal GPU prototype



The printed circuit board for Pascal as introduced in GTC'16

- 3x performance density.
- 18 billion FinFET 16 nm transistors on a 600 mm^2 die:
  - Manufactured by TSMC.
- HBM2 memory cubes with 4000 wires:
  - Manufactured by Samsung.

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

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.

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

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	1625 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	512 KB	2048 KB	

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	1625 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	512 KB	2048 KB	
DRAM memory: Interface	256-bit GDDR5	256-bit GDDR5	256-bit GDDR5X

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	1625 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	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

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	1625 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	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

Commercial models for Tesla P100 (Pascal) and comparative with 2 previous generations

Commercial models for Tesla P100 (Pascal) and comparative with 2 previous generations

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 w. NV-link	P100 w. PCI-e
Release date	2012	2014	2016	
Transistors	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET	

Commercial models for Tesla P100 (Pascal) and comparative with 2 previous generations

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 w. NV-link	P100 w. PCI-e
Release date	2012	2014	2016	
Transistors	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET	
# 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)	

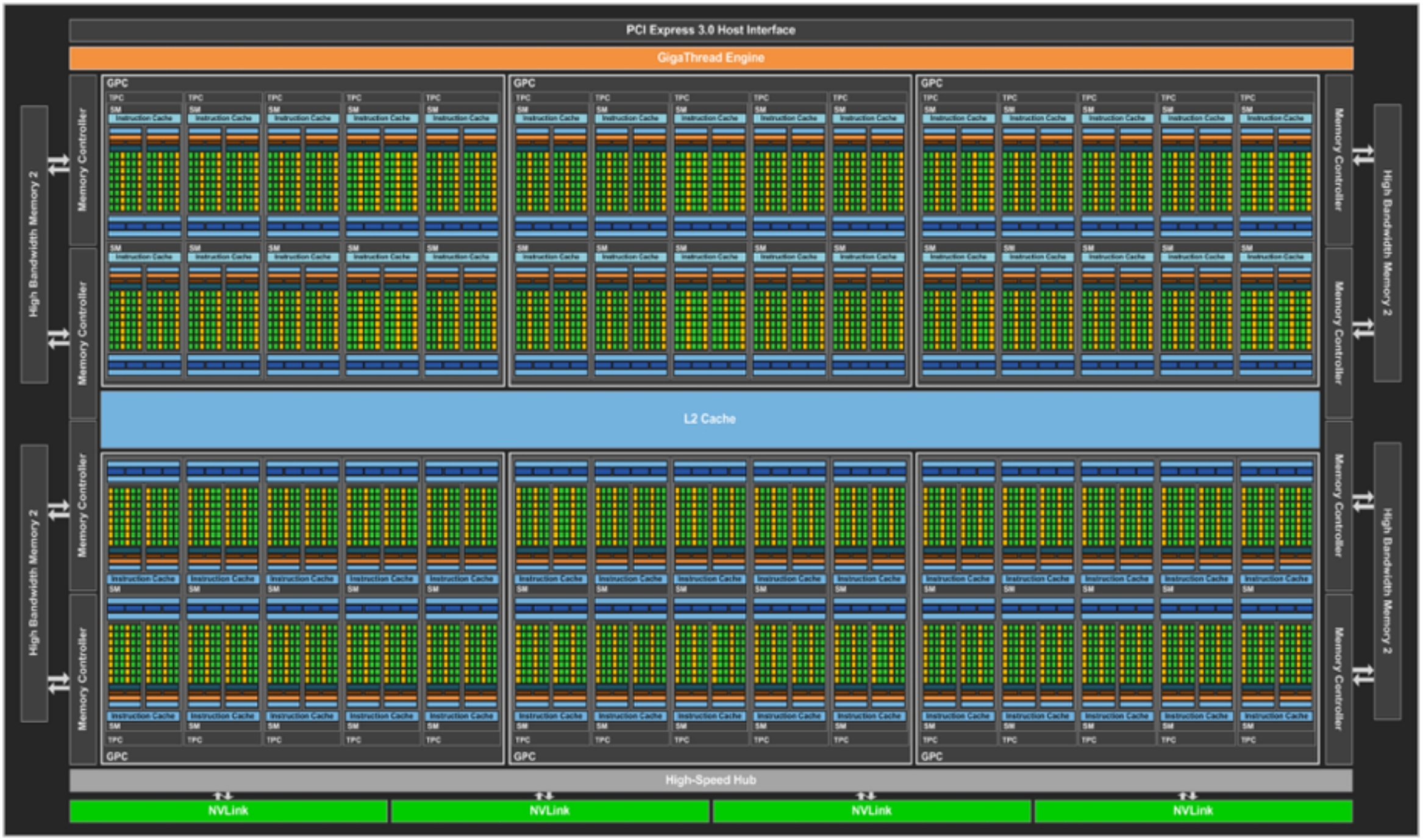
Commercial models for Tesla P100 (Pascal) and comparative with 2 previous generations

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 w. NV-link	P100 w. PCI-e
Release date	2012	2014	2016	
Transistors	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET	
# 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	

Commercial models for Tesla P100 (Pascal) and comparative with 2 previous generations

	Tesla K40 (Kepler)	Tesla M40 (Maxwell)	P100 w. NV-link	P100 w. PCI-e
Release date	2012	2014	2016	
Transistors	7.1 B @ 28 nm.	8 B @ 28 nm.	15.3 B @ 16 nm. FinFET	
# 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 physical layout for multiprocessors, memory controllers and buses

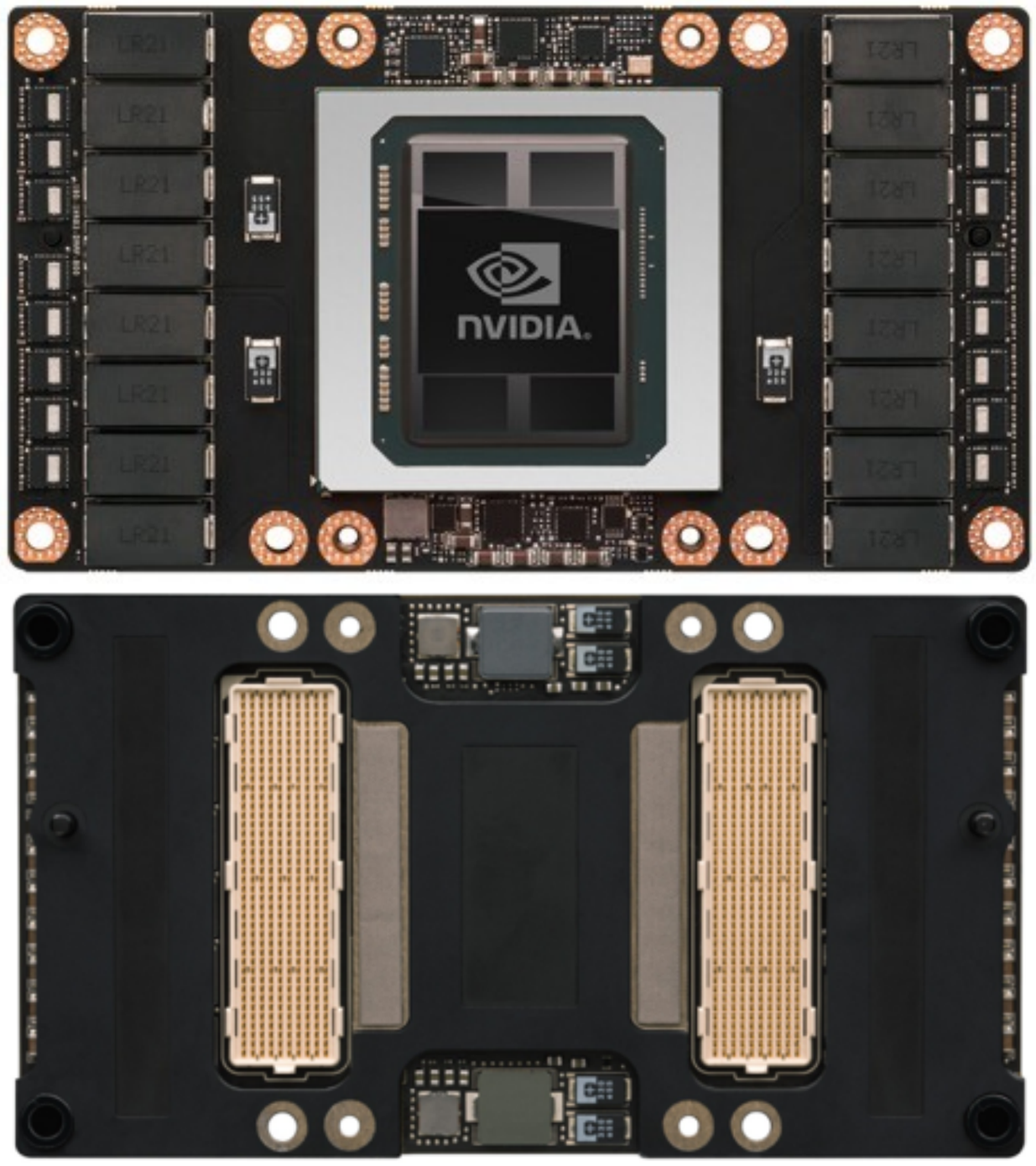


Pascal multiprocessor

Pascal and its 4 3D memory cubes: Front and back

- The Tesla P100 model:
  - GPU: 56 SMs x 64 cores.

    Peak performance:
    - 5.3 TFLOPS (FP64).
    - 10.6 TFLOPS (FP32).
    - 21.2 TFLOPS (FP16).
- Memory:
  - More register files and shared memory than Maxwell (same sizes but now there are 56 SMX2, versus 24 in Maxwell Titan X).
- NVLINK bus: 160 GB/s.
  - 80 GB/s. bidirectional.



II. 7. A summary of four generations

NVIDIA

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

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

	Tesla	Fermi	Kepler	Maxwell

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)

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

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**

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

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

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

New models for 2016/17

	Maxwell	Pascal

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)

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**

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**

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

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

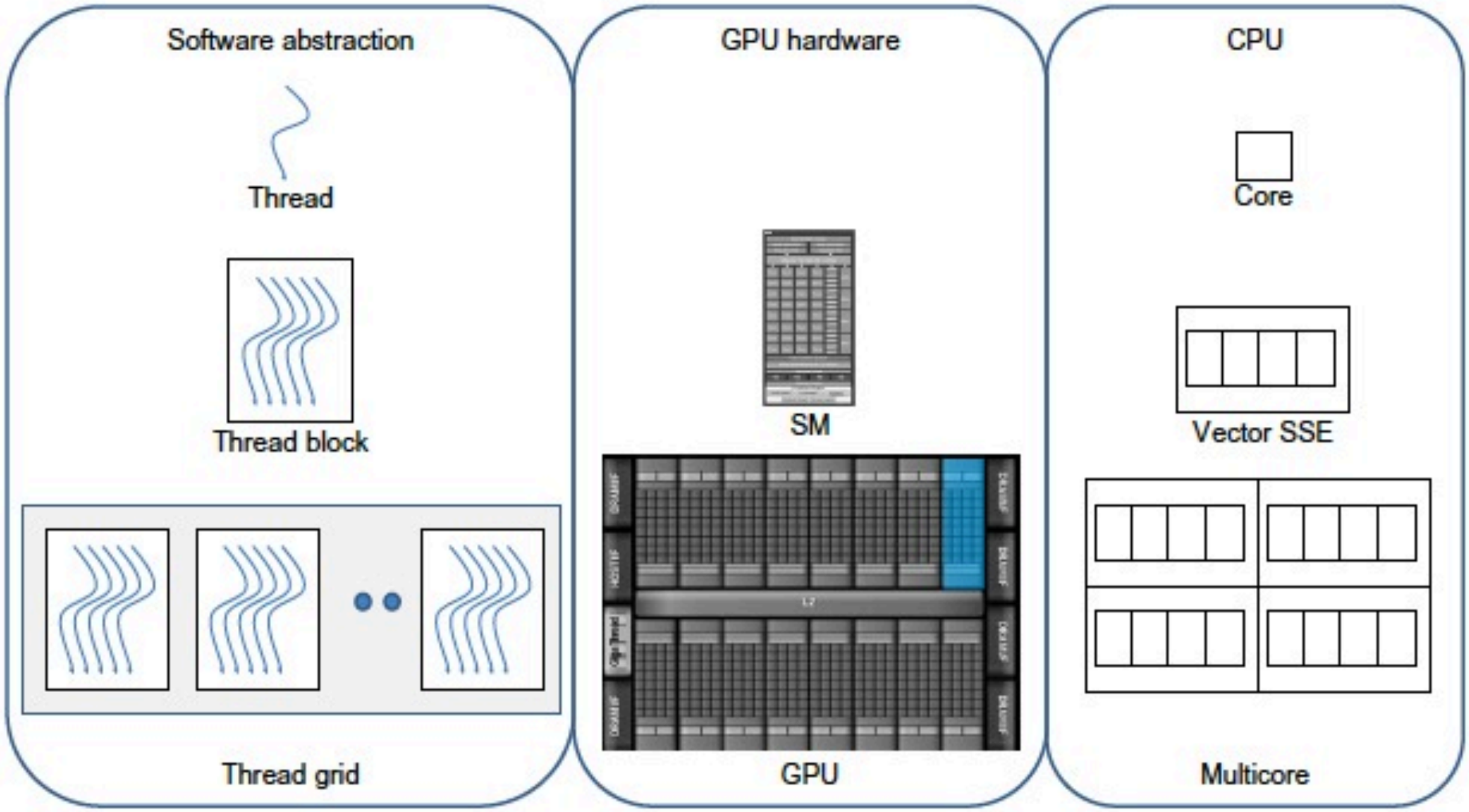
New models for 2016/17

	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

NVIDIA

Comparing the GPU and the CPU



From POSIX threads in CPU to CUDA threads in GPU

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);
}
```

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;
}
```

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

Preliminary definitions

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

Preliminary definitions

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

- Device = GPU = Set of multiprocessors.

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.

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.

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.

Preliminary definitions

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

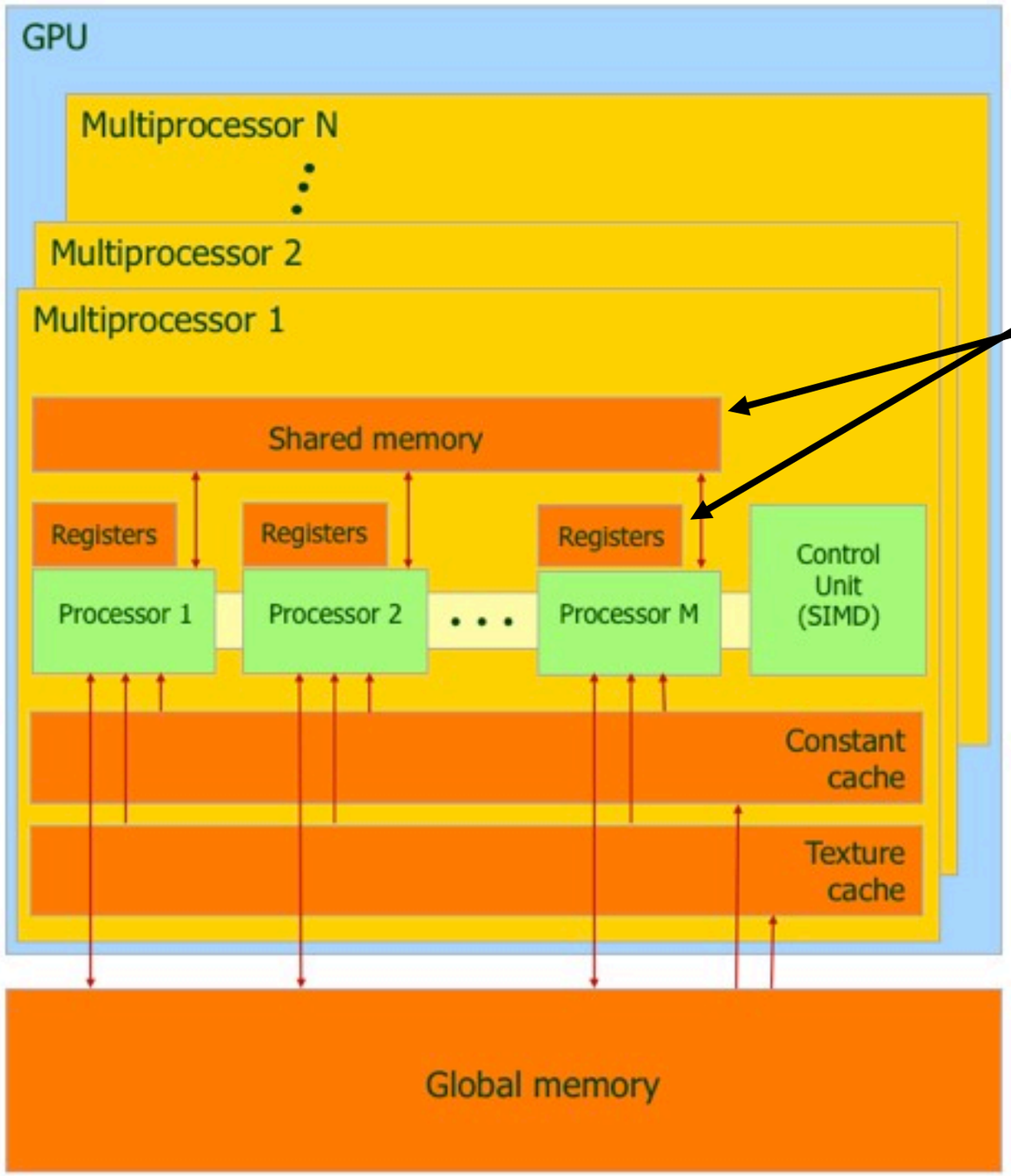
- Device = GPU = Set of multiprocessors.
- Multiprocessor = Set of processors + shared memory.
- Kernel = Program ready to run on GPU.
- Grid = Array of thread blocks that execute a kernel.
- Thread block = Group of SIMD threads that:
  - Execute a kernel on different data based on threadID and blockID.
  - Can communicate via shared memory.

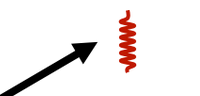
Preliminary definitions

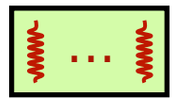
Programmers face the challenge of exposing parallelism for thousands cores using the following elements:

- 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

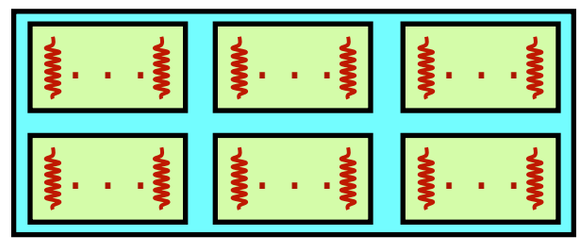


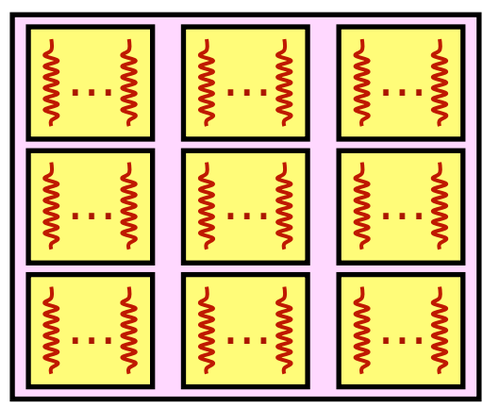




On-chip memory

Memory outside the GPU chip (but within the graphics card)



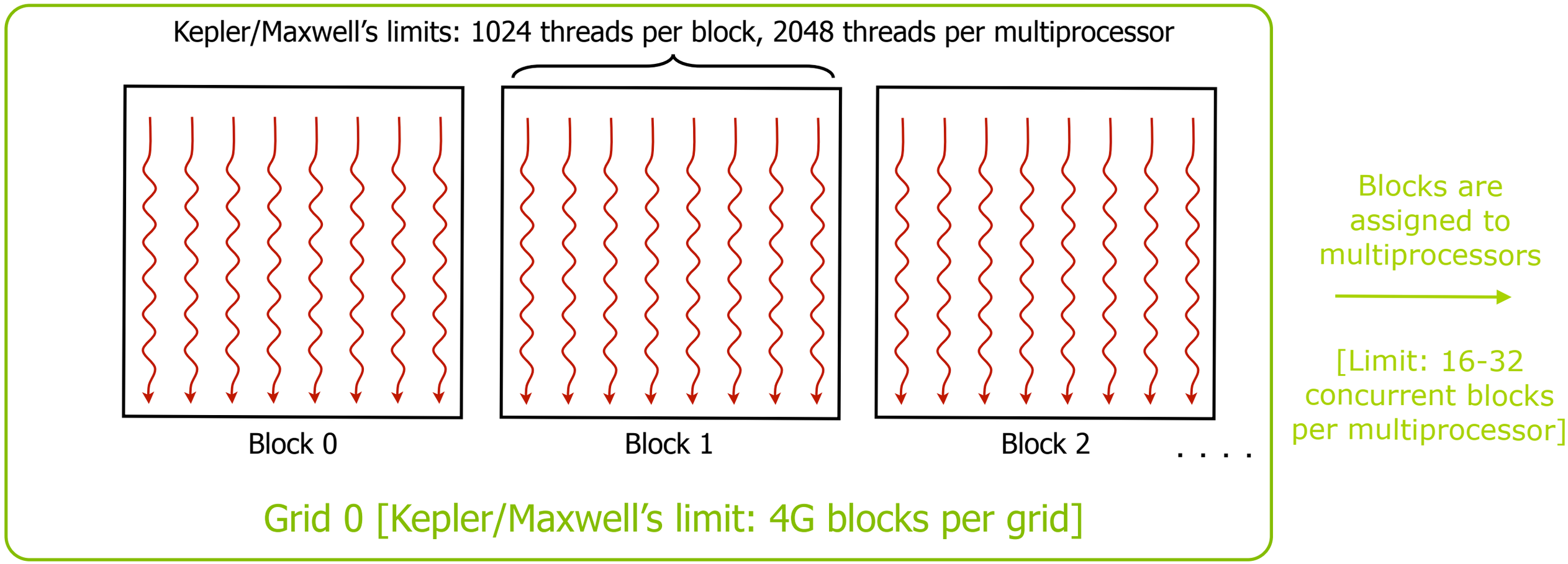


Resources and limitations depending on CUDA hardware generation (CCC)

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, ...	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	16K 48K	16K, 32K, 48K	64K (5.0) 96K (5.2)	64KB.		

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 Maxwell to maximize concurrency

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.

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].

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.

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.

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.

Playing with parallel constrainsts in Maxwell to maximize concurrency

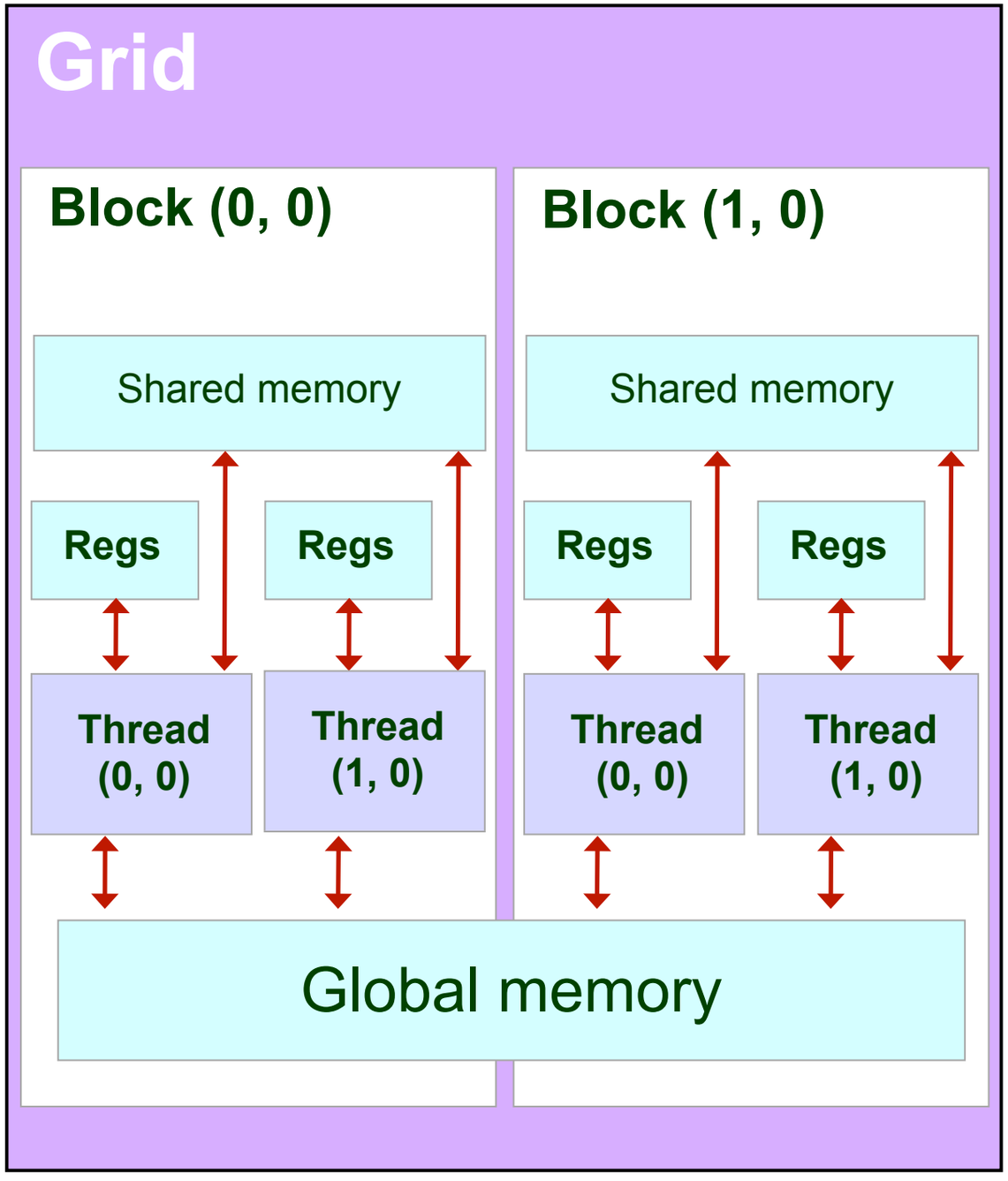
- Limits within a SMM multiprocessor: [1] 32 concurrent blocks, [2] 1024 threads/block and [3] 2048 threads total.
- 1 block of 2048 threads. Forbidden by [2].
- 2 blocks of 1024 threads. Feasible on the same multiproc.
- 4 blocks of 512 threads. Feasible on the same multiproc.
- 4 blocks of 1024 threads.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

Playing with parallel constrainsts in Maxwell to maximize concurrency

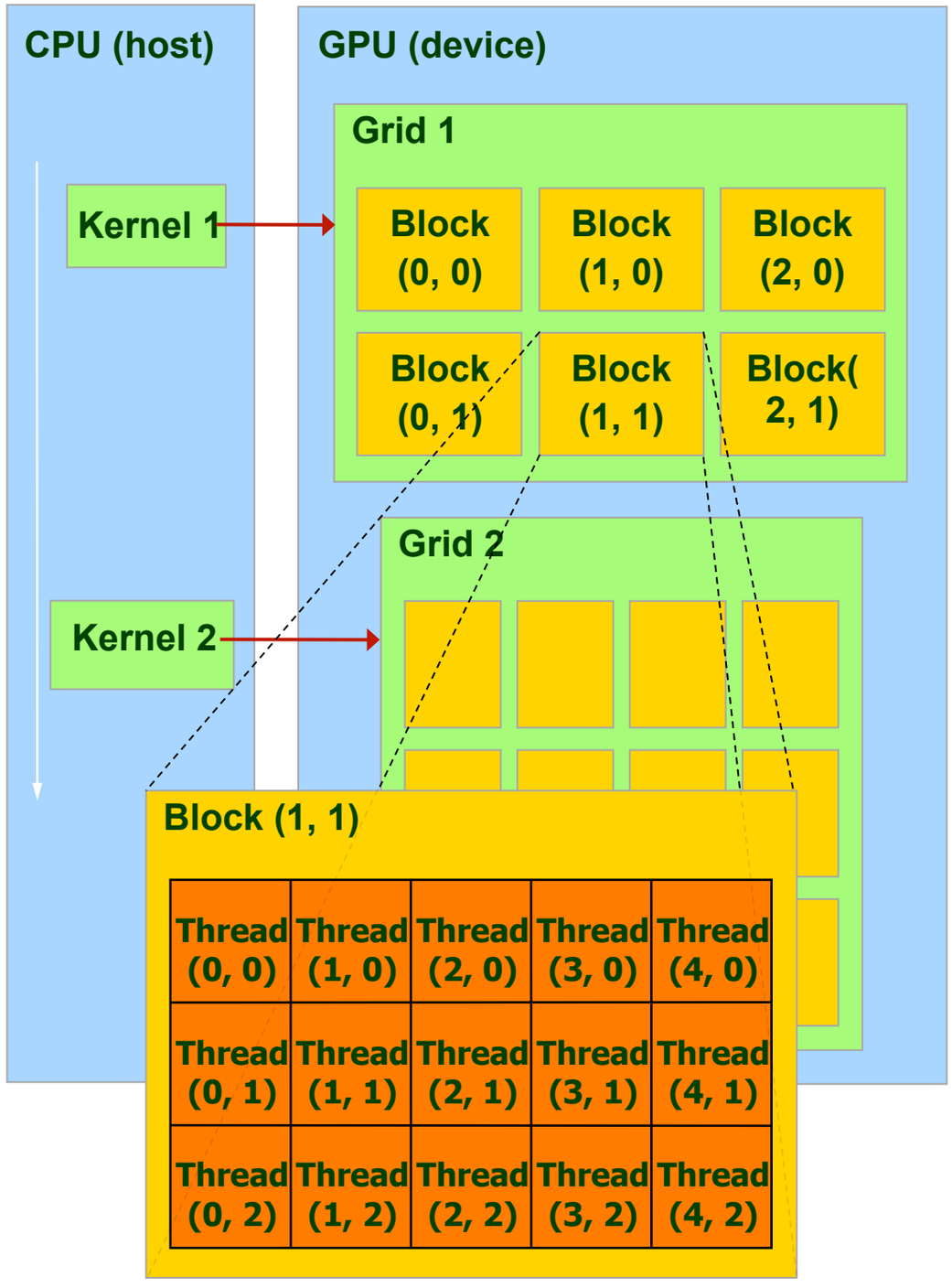
- Limits within a SMM multiprocessor: [1] 32 concurrent blocks, [2] 1024 threads/block and [3] 2048 threads total.
- 1 block of 2048 threads. Forbidden by [2].
- 2 blocks of 1024 threads. Feasible on the same multiproc.
- 4 blocks of 512 threads. Feasible on the same multiproc.
- 4 blocks of 1024 threads.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

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).

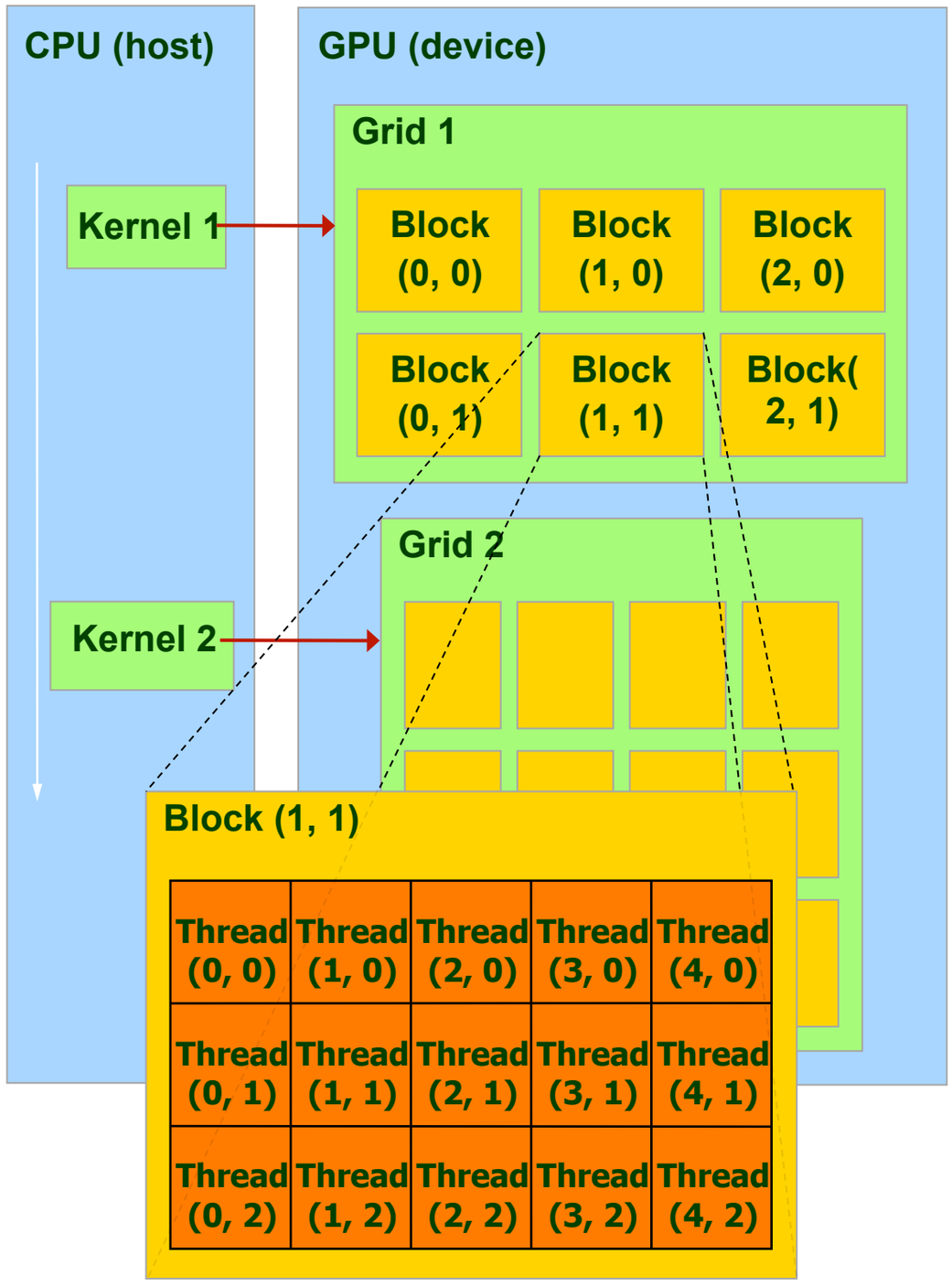


Partitioning data and computations



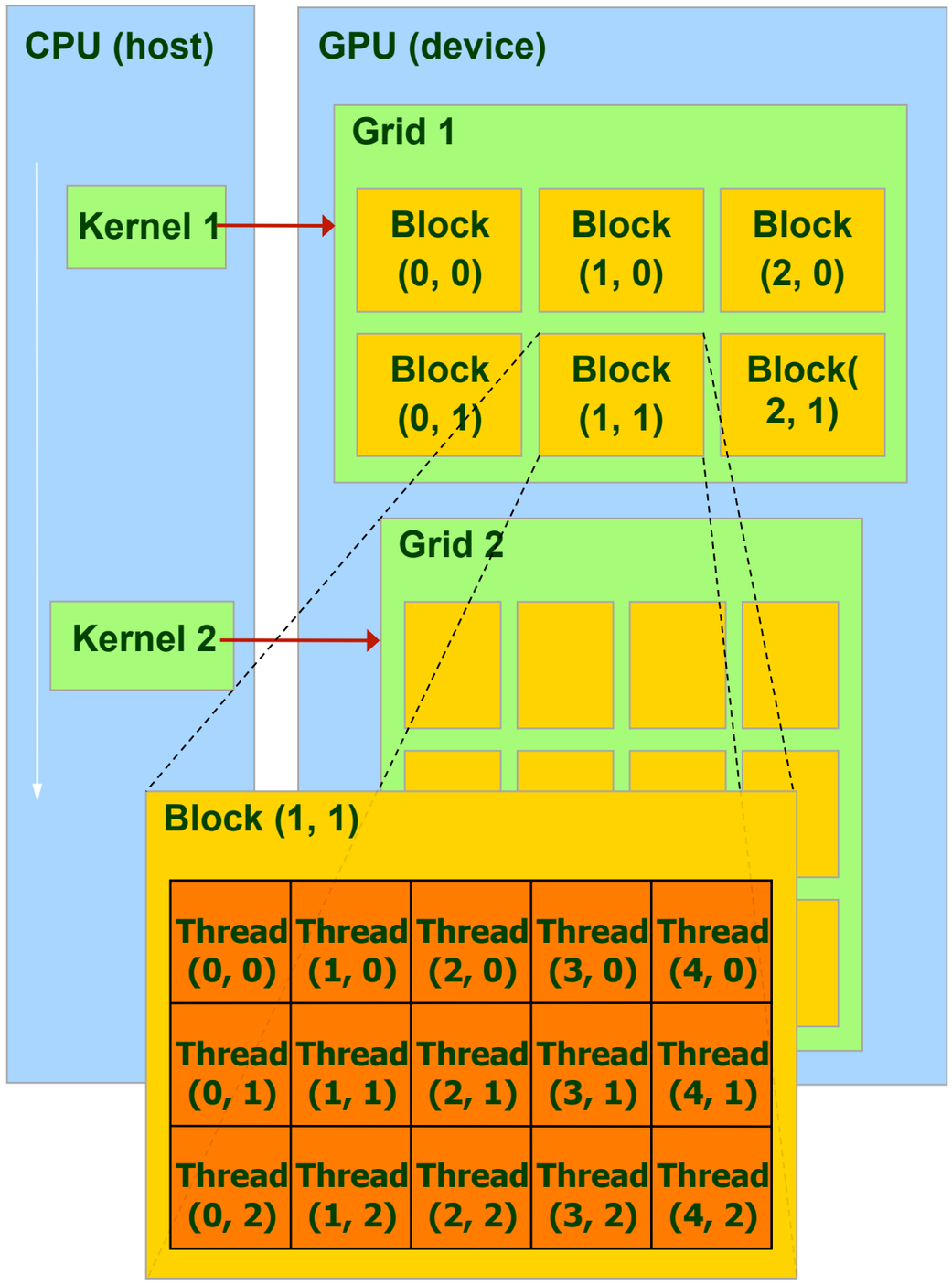
Partitioning data and computations

- A block is a batch of threads which can cooperate by:
  - Sharing data via shared memory.
  - Synchronizing their execution for hazard-free memory accesses.



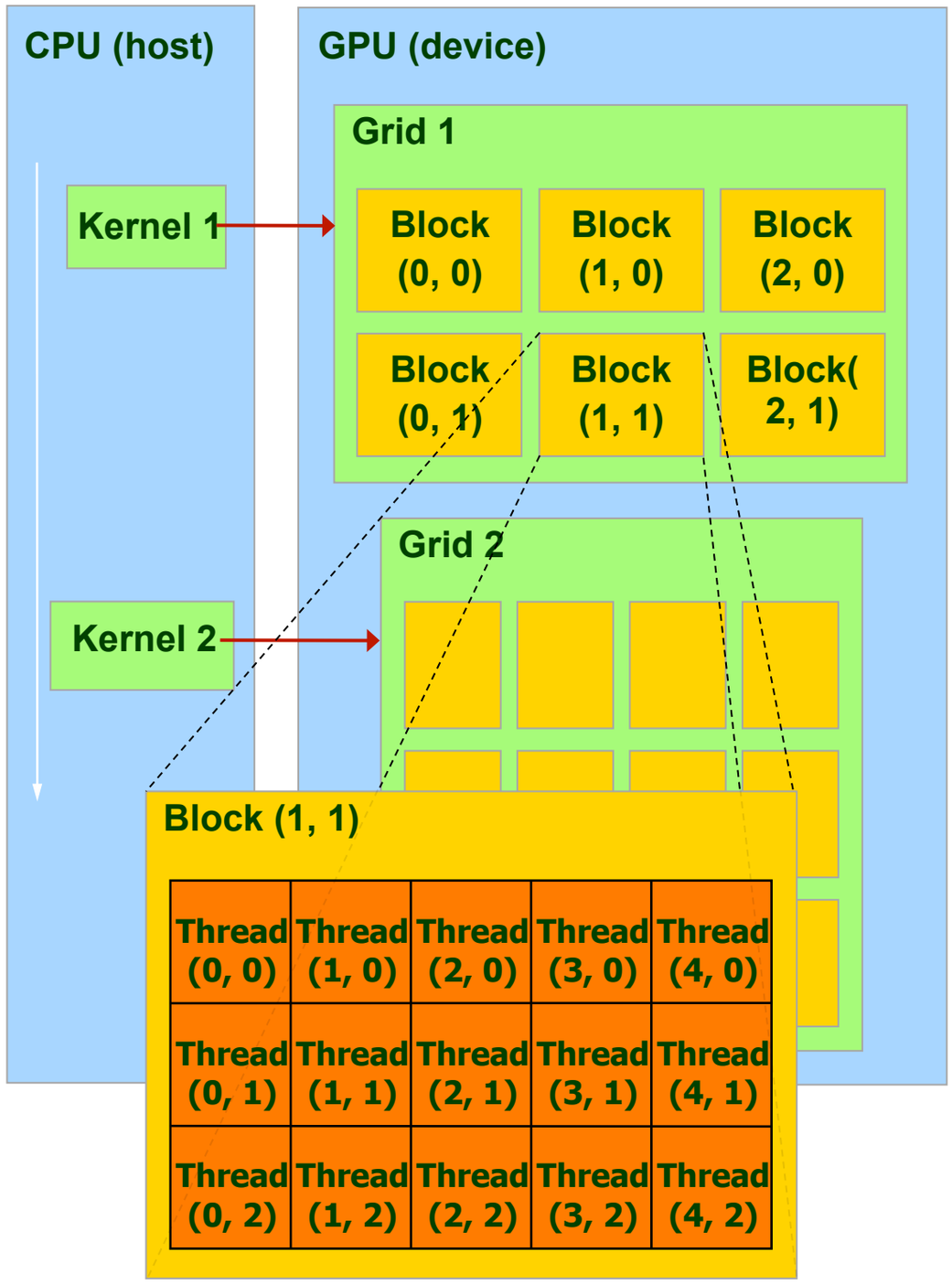
Partitioning data and computations

- 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.



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.



IV. Syntax

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

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.

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.

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.

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.

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

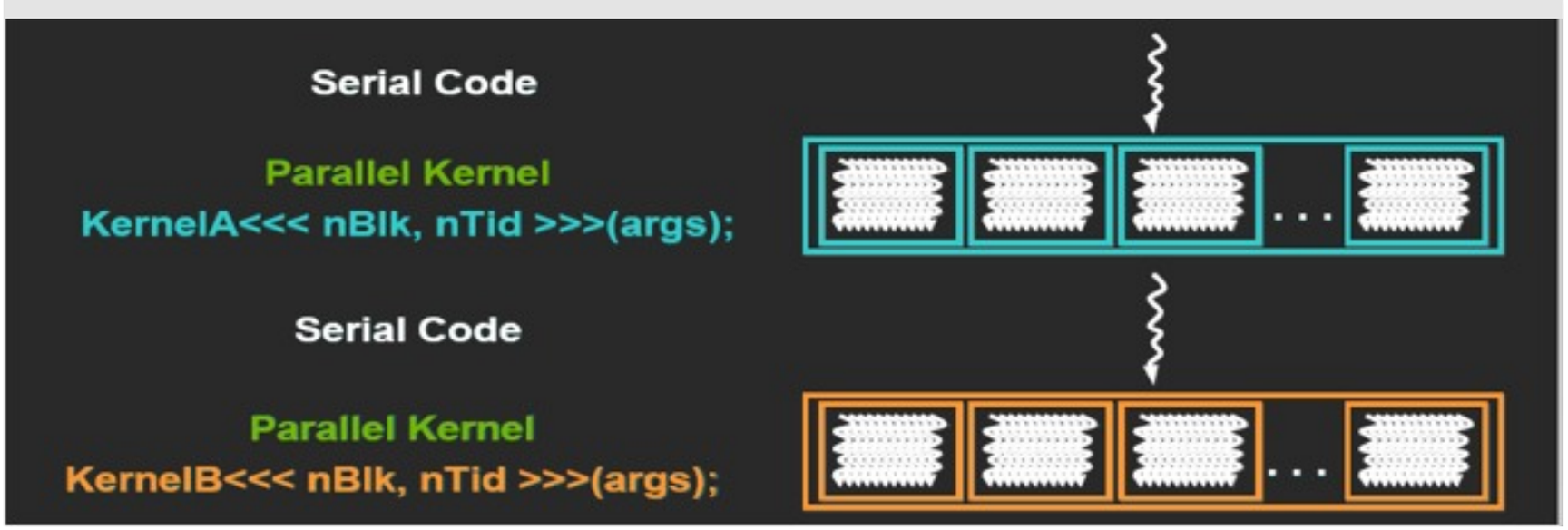
CPU

GPU

CPU

GPU

CPU



- A kernel does not start until all previous kernels are over.
- **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

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.

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);`

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);`

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);`

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);`

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. More 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

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:

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
```

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();`

V. 1. Adding two vectors

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

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

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 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

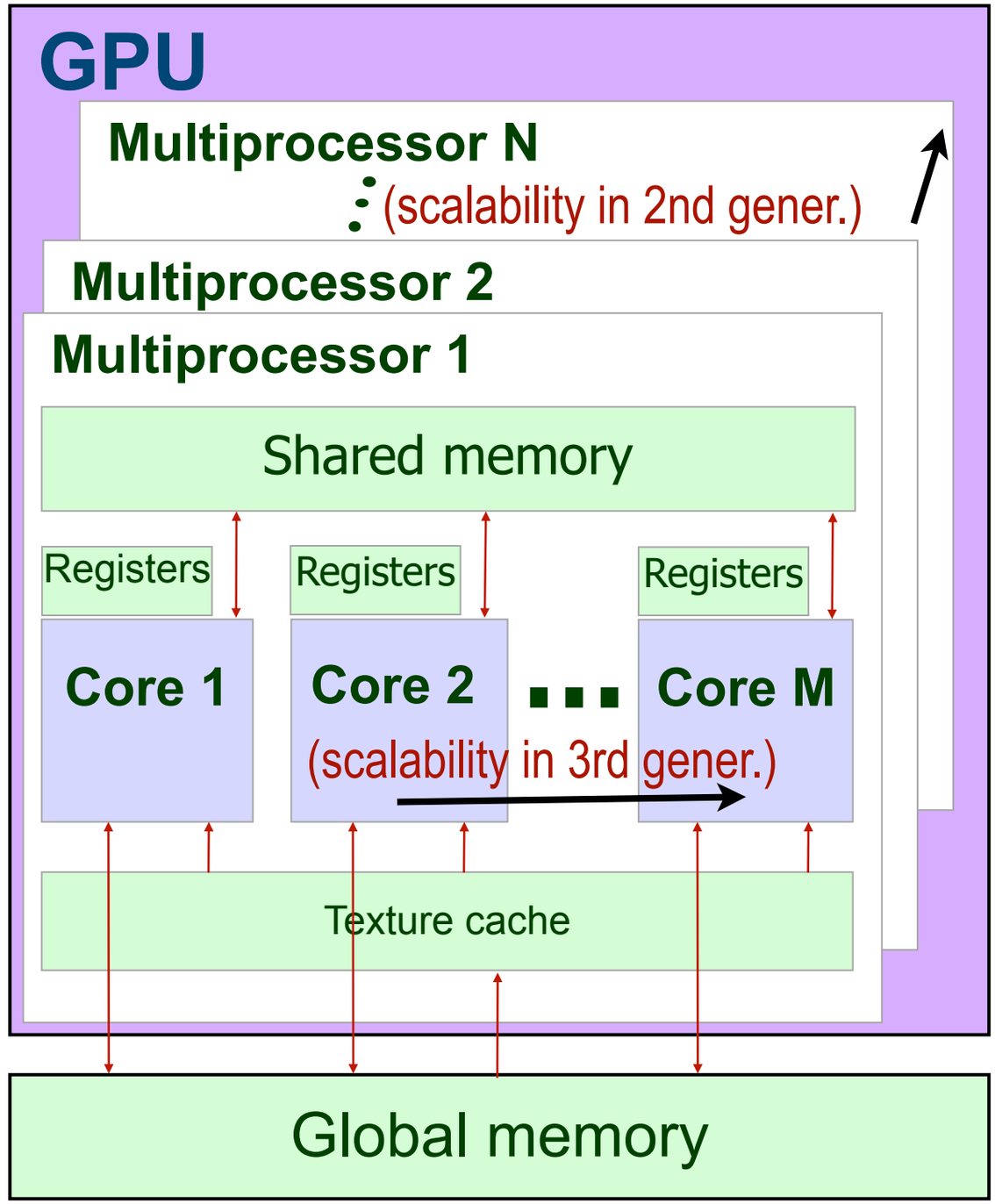
- The `__global__` prefix indicates that `vecAdd()` will execute on device (GPU) and will be called from host (CPU).
- `A`, `B` and `C` are pointers to device memory, so we need to:
  - Allocate/free memory on GPU, using `cudaMalloc()`/`cudaFree()`.
  - These pointers cannot be dereferenced in host code.

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);
```

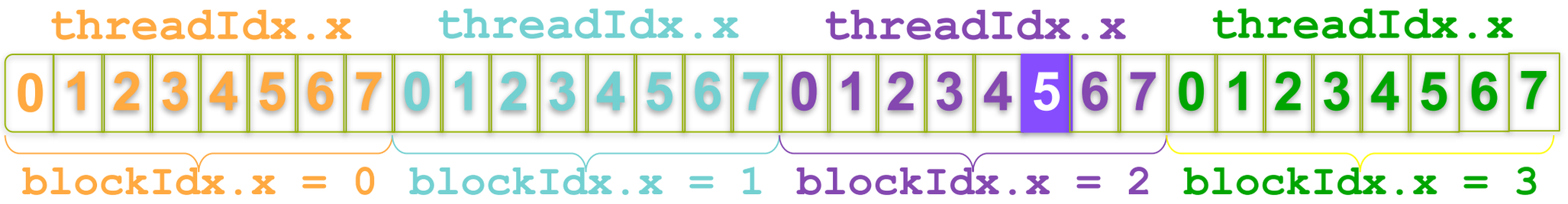
Running in parallel (regardless of hardware generation)

- `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:



- 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);
```

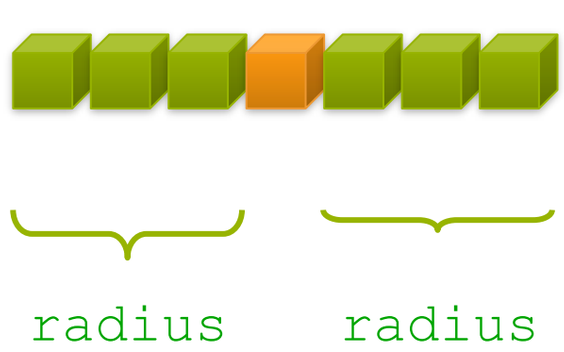
VI. 2. Stencil kernels

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

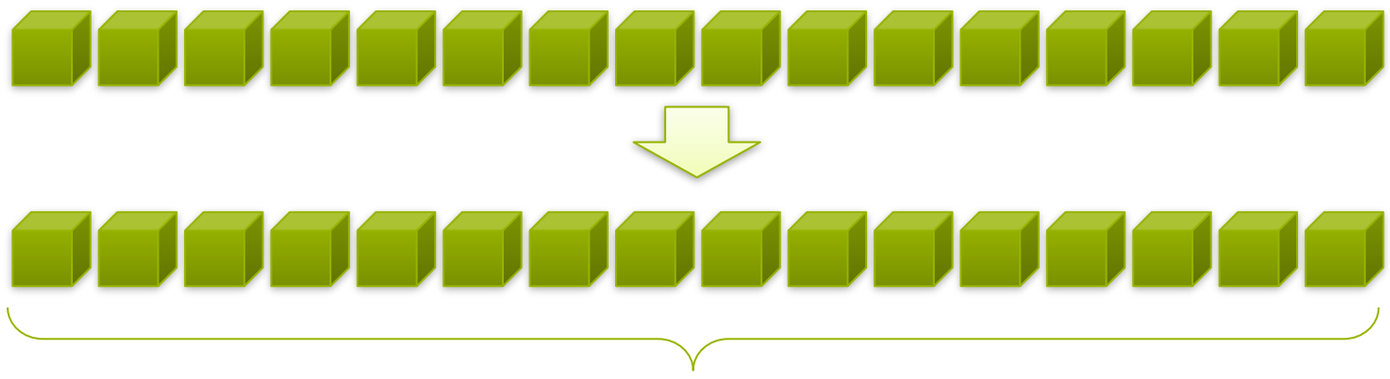
- Threads within a block can share data via shared memory.
  - Shared memory is user-managed: Declare with __shared__ prefix.
  - Data is allocated per block.
  - Shared memory is extremely fast:
    - 500 times faster than global memory (video memory - GDDR5). The difference is technology: static (built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

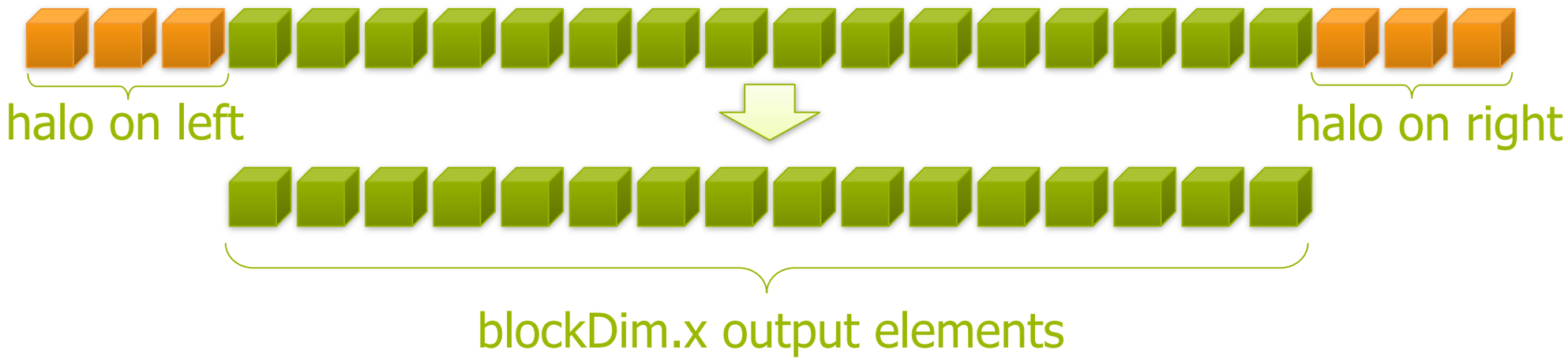
Using Shared Memory

- 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.



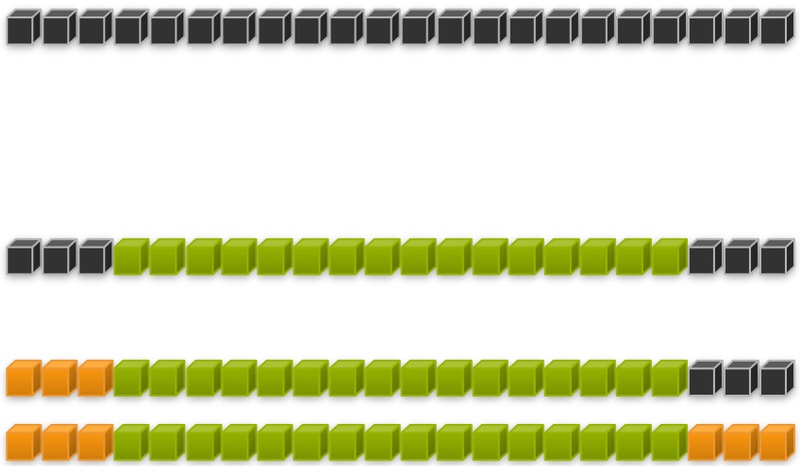
Using Shared Memory

- 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.



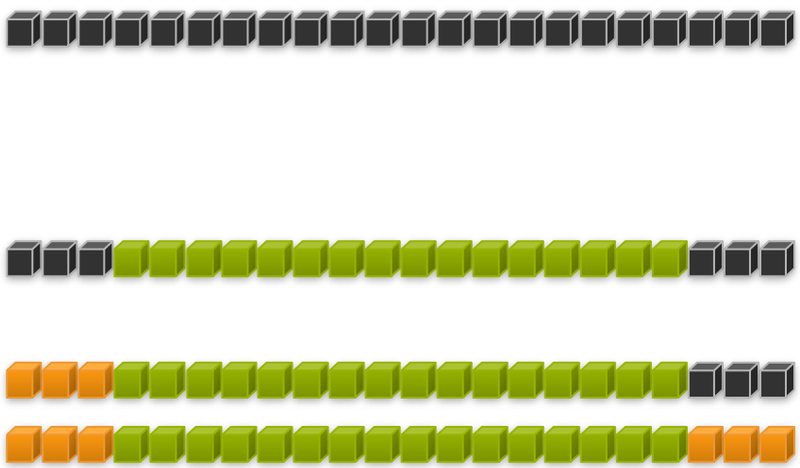
Stencil kernel

```
__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;
}
```



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

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 >>> ();`

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;`

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;`

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).

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

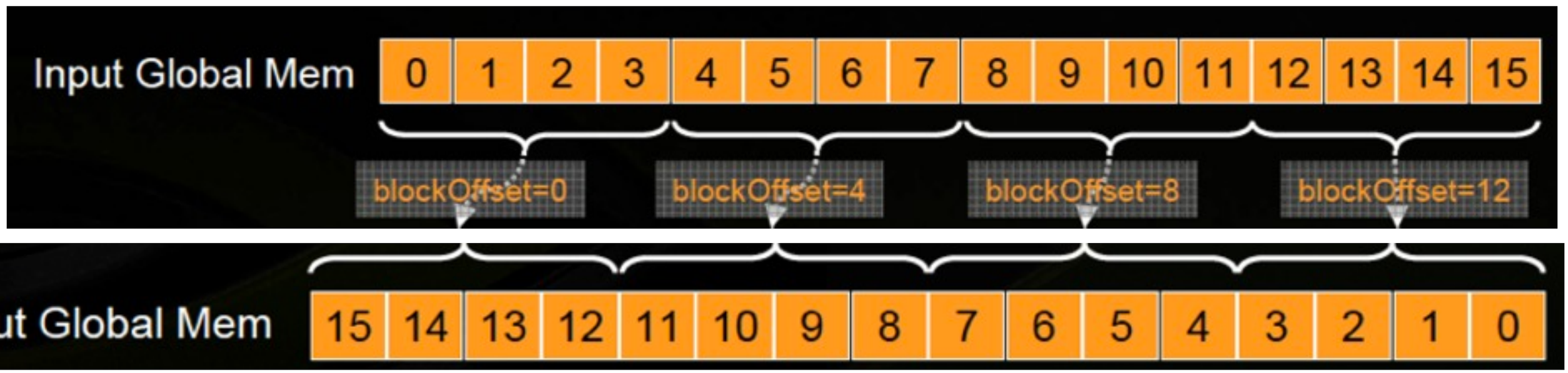
```
__global__ void reverseArray(int *in, int *out) {
  int index_in = threadIdx.x;
  int index_out = blockDim.x – 1 – threadIdx.x;

  // 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.

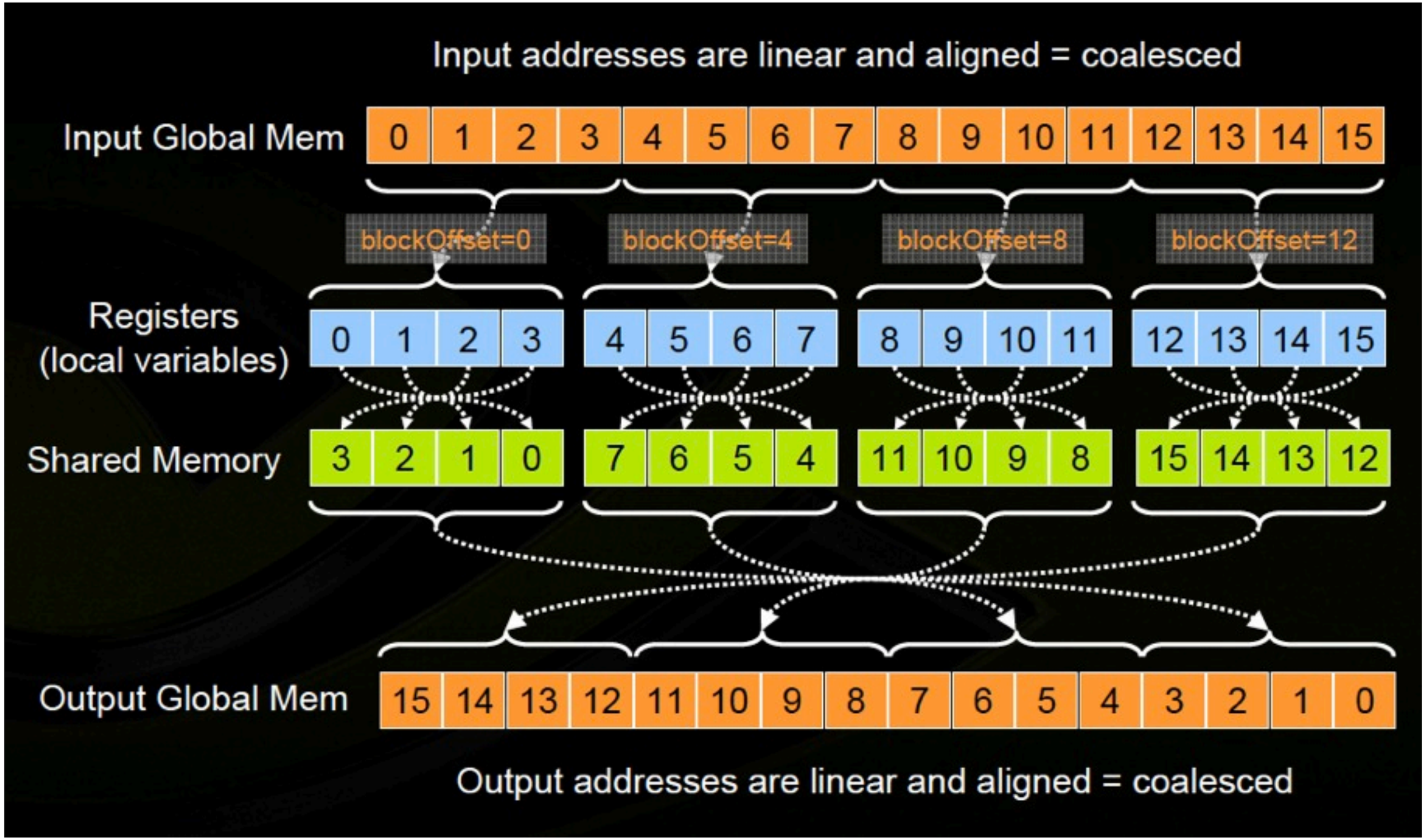
GPU code for the ReverseArray kernel (2) using multiple blocks

```
__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:



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];
}
```

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.

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)).

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).

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.

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];
}
```

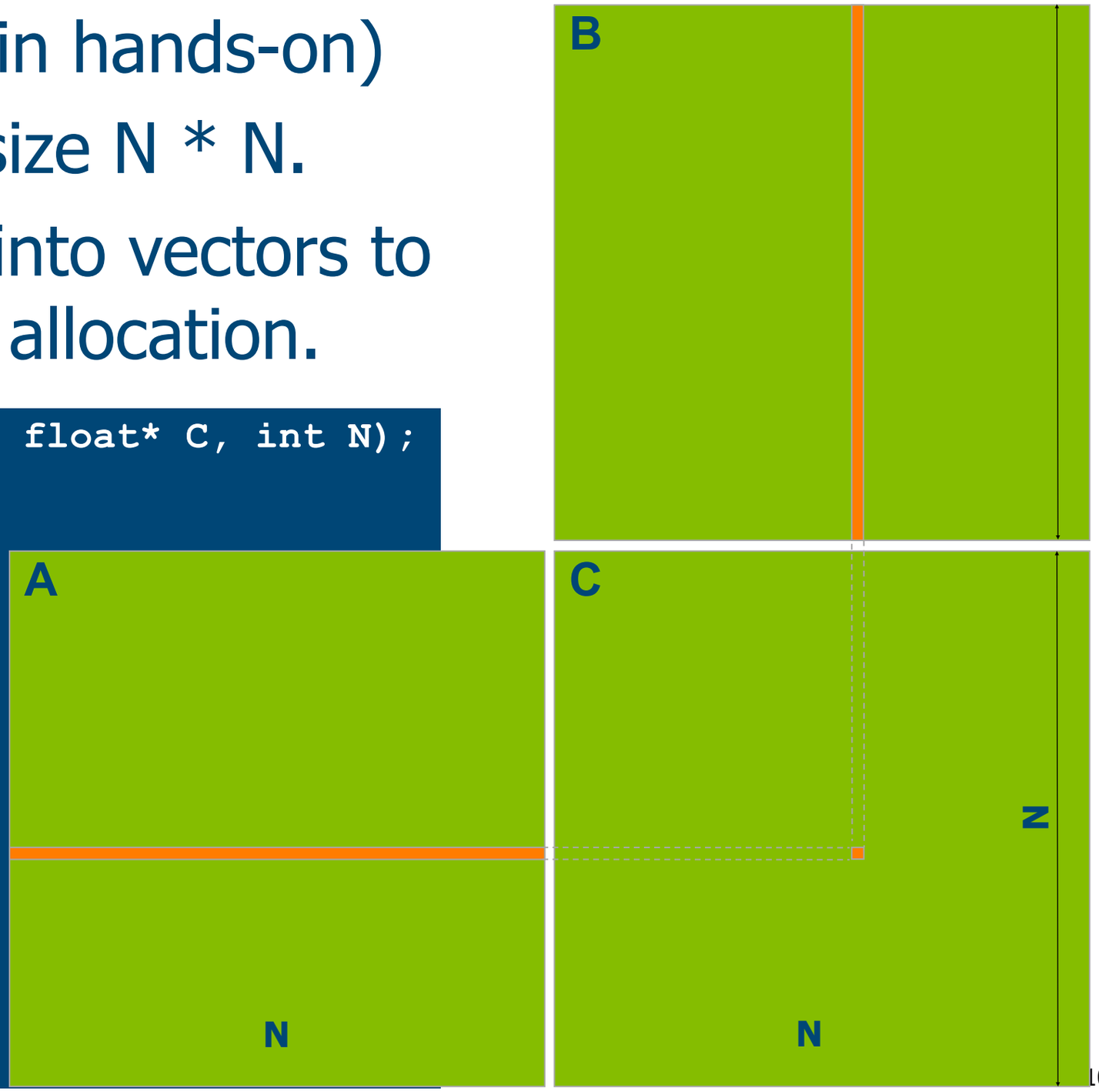
- Dependency: In (i2), values written by a warp, have to be read (before) by another warp.
- Solution: Use a `temp2[BLOCK_SIZE]` array to store intermediate results (also in (i4)).
- Improvement: (i3) is not required. Also, if you swap indices within `temp[]` and `temp2[]` in (i2), then (i1) is not required (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!

V. 4. Matrix product

Typical CPU code written in C language

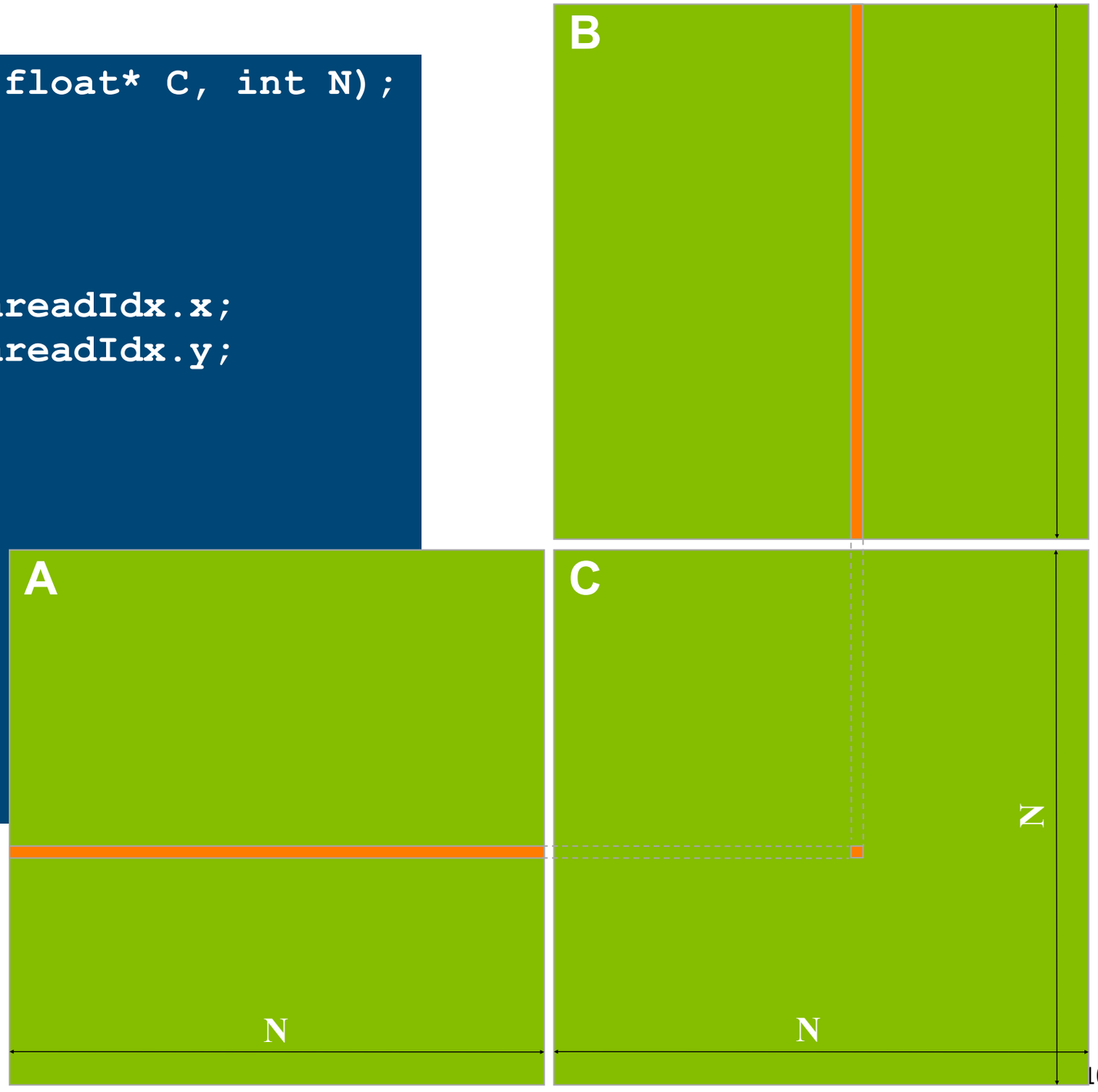
- C = A * B. (P = M * N in hands-on)
- All square matrices of size N * N.
- Matrices are serialized into vectors to simplify dynamic memory allocation.

```
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;
    }
}
```



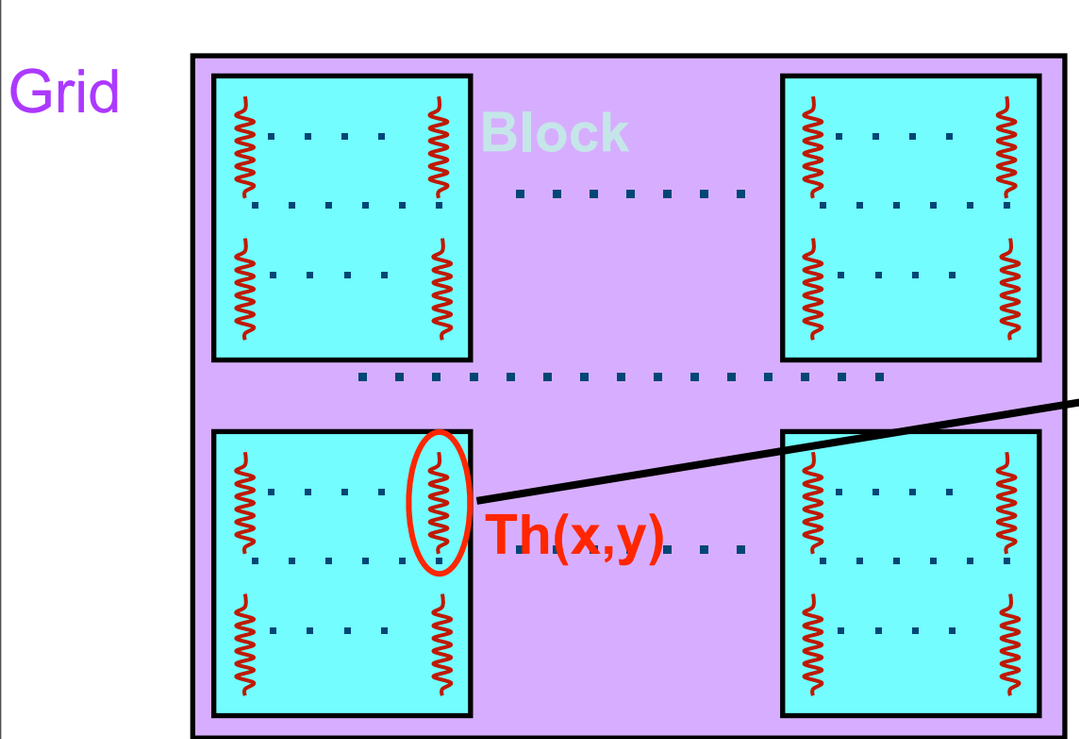
CUDA version for the matrix product: A draft for the parallel code

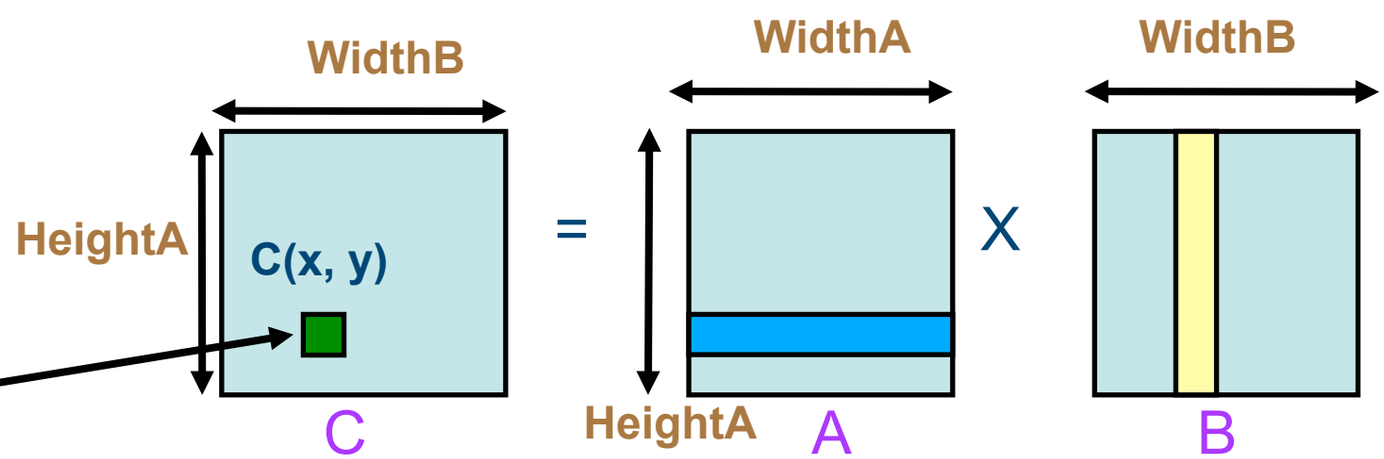
```
void MxMonGPU(float* A, float* B, float* C, int N);
{
  float sum=0;
  int i, j;

  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.





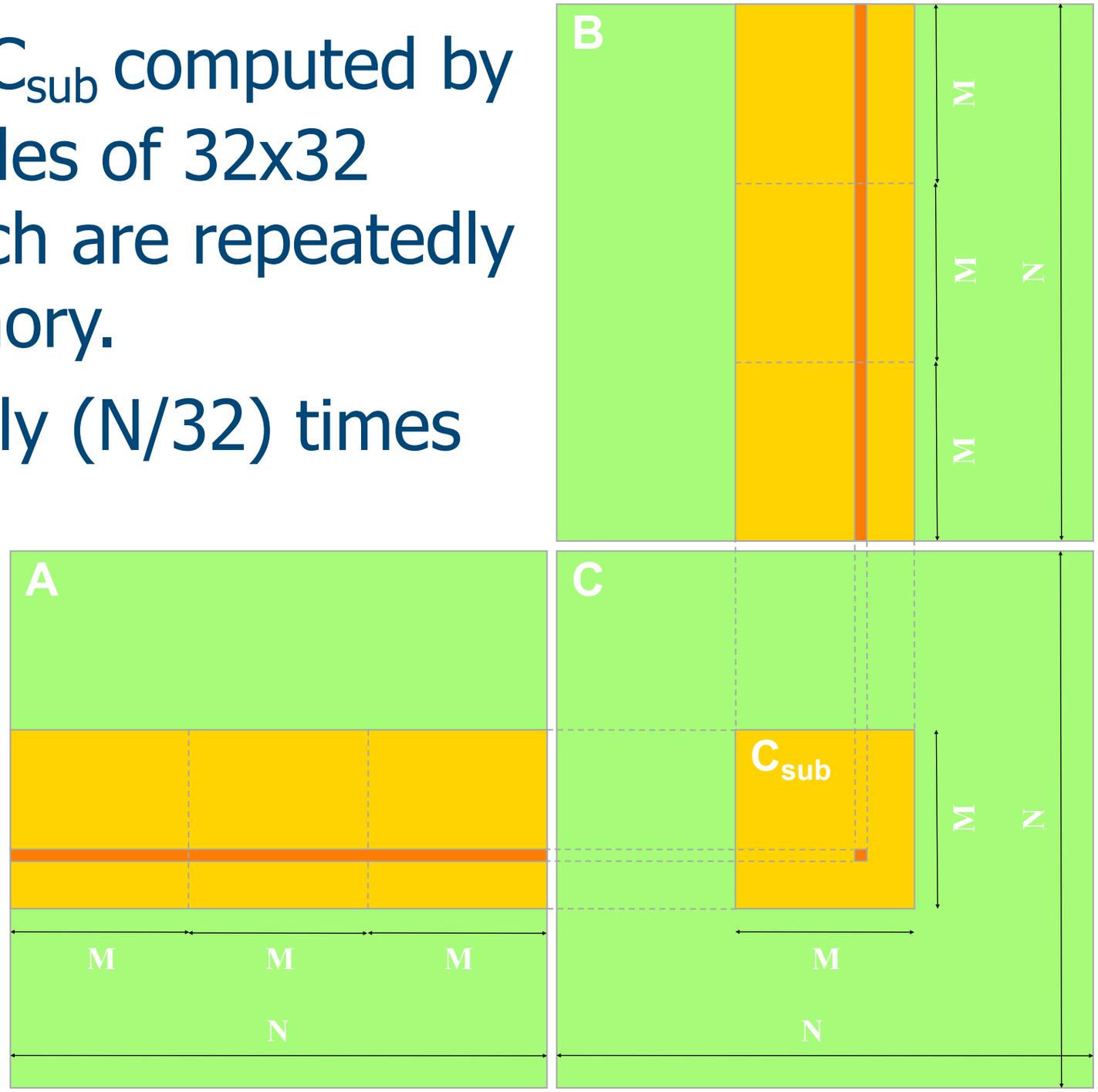

```
dim2 dimBlock(BLOCKSIZE, BLOCKSIZE);
dim2 dimGrid(WidthB/BLOCKSIZE, HeightA/BLOCKSIZE);
...
MxMonGPU <<<dimGrid,dimBlock>>> (A, B, C, N);
```

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.



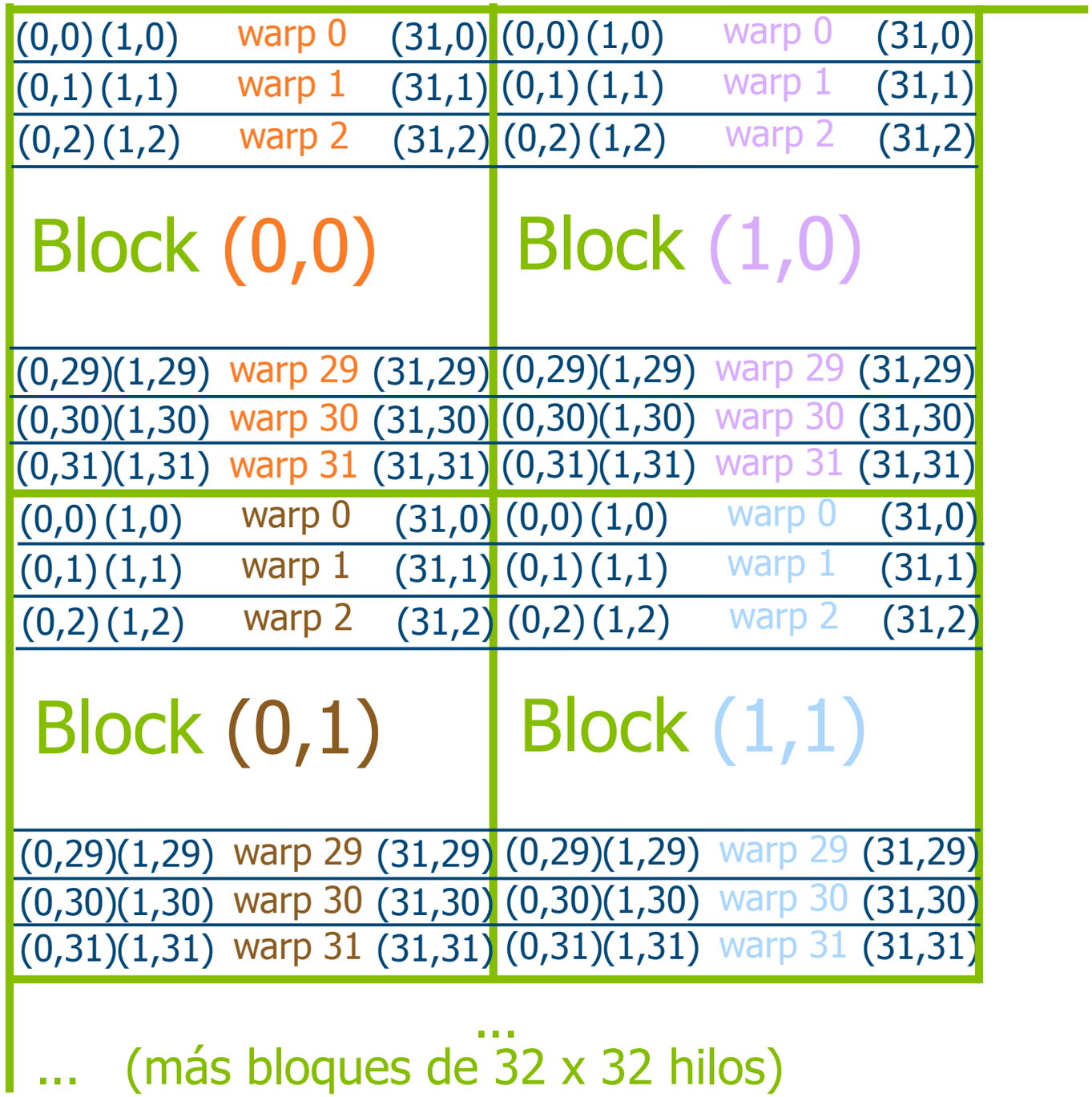
Tiling: Implementation details

- 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).

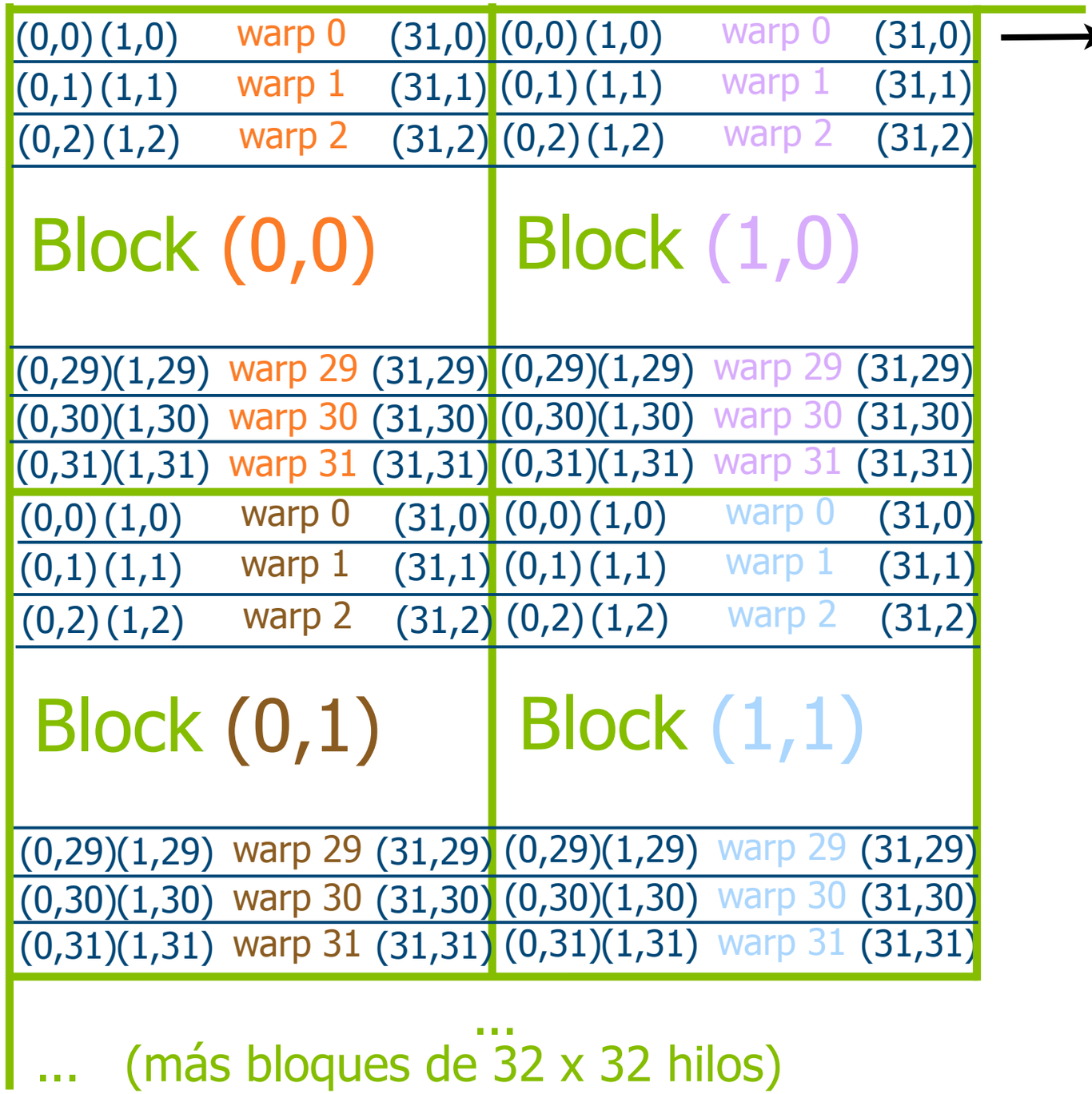
A trick to avoid shared memory bank conflicts

- 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

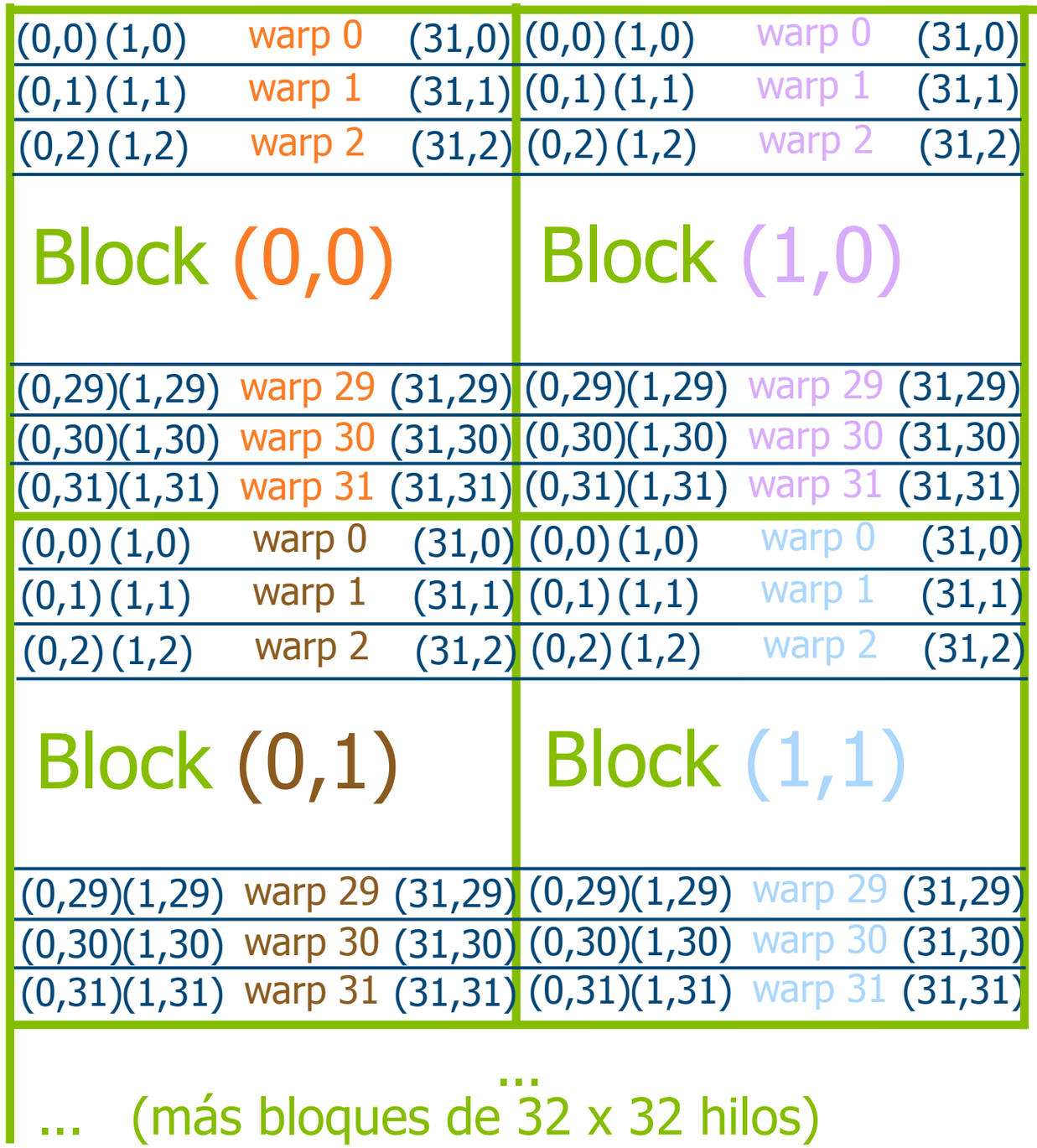


An example for solving conflicts to banks in shared memory



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

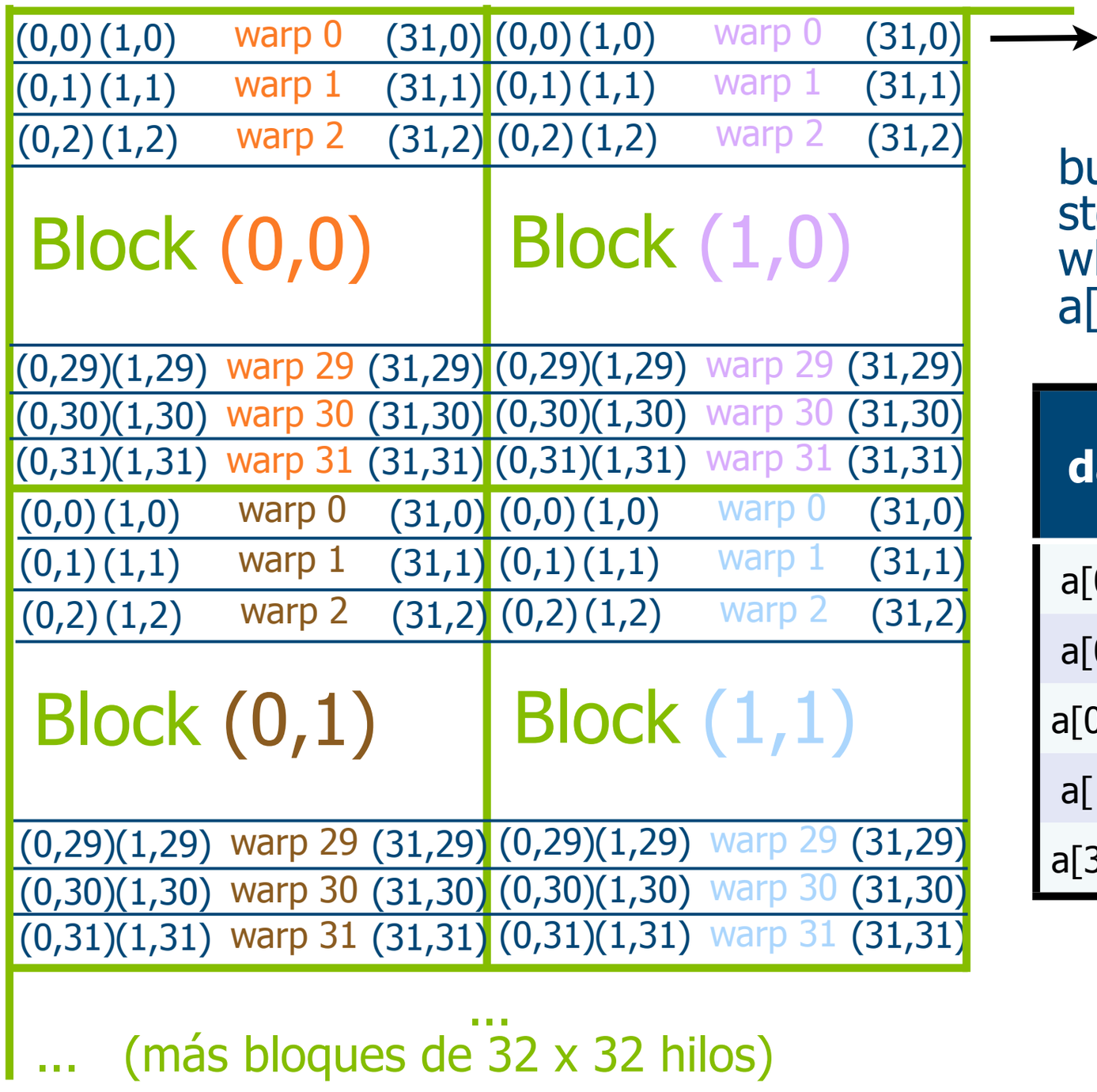
An example for solving conflicts to banks in shared memory



→ 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], ...

An example for solving conflicts to banks in shared memory

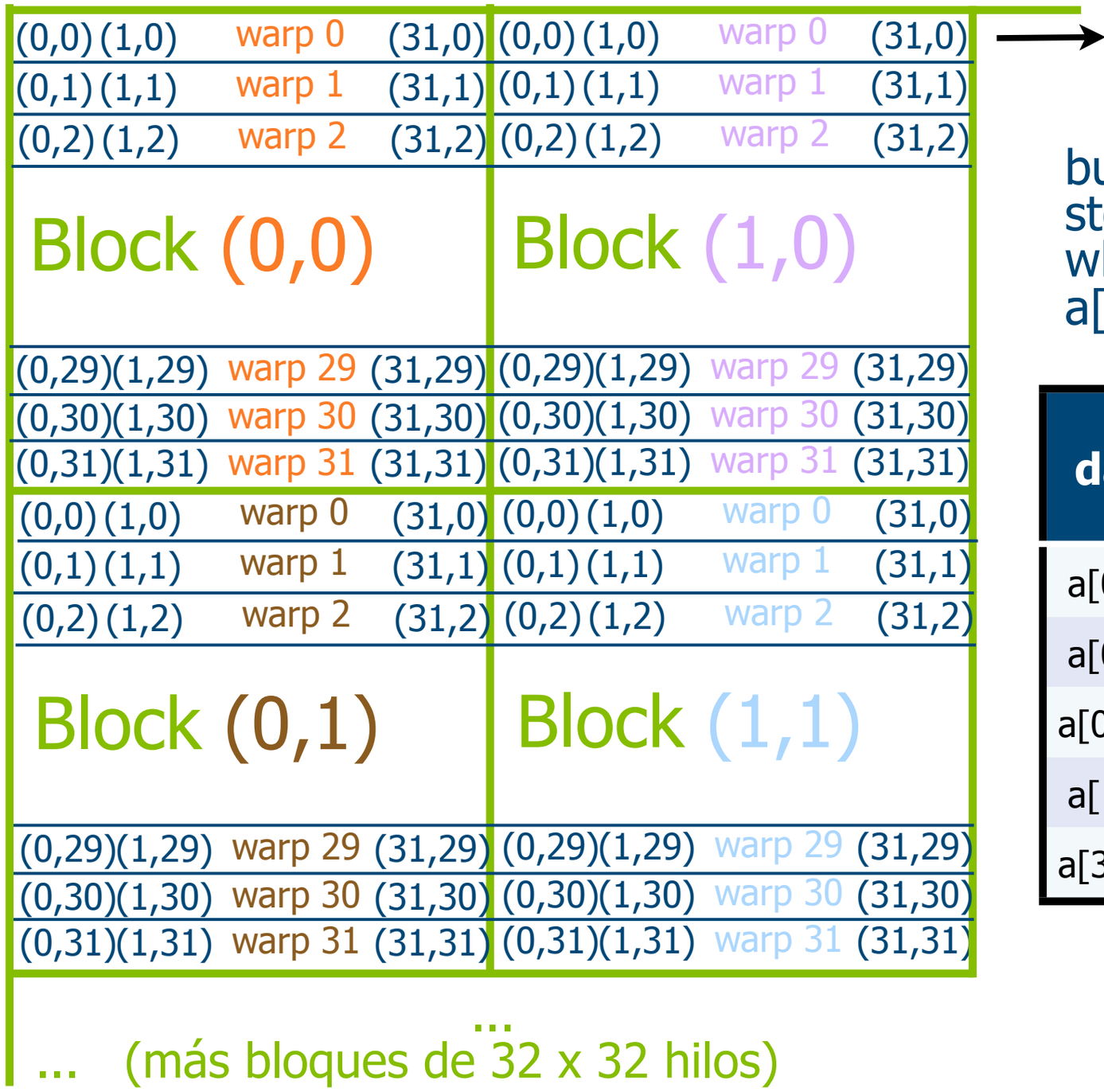


→ Consecutive threads within a warp differ in the first dimension.

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
a[0][0]
a[0][1]
a[0][31]
a[1][0]
a[31][0]

An example for solving conflicts to banks in shared memory

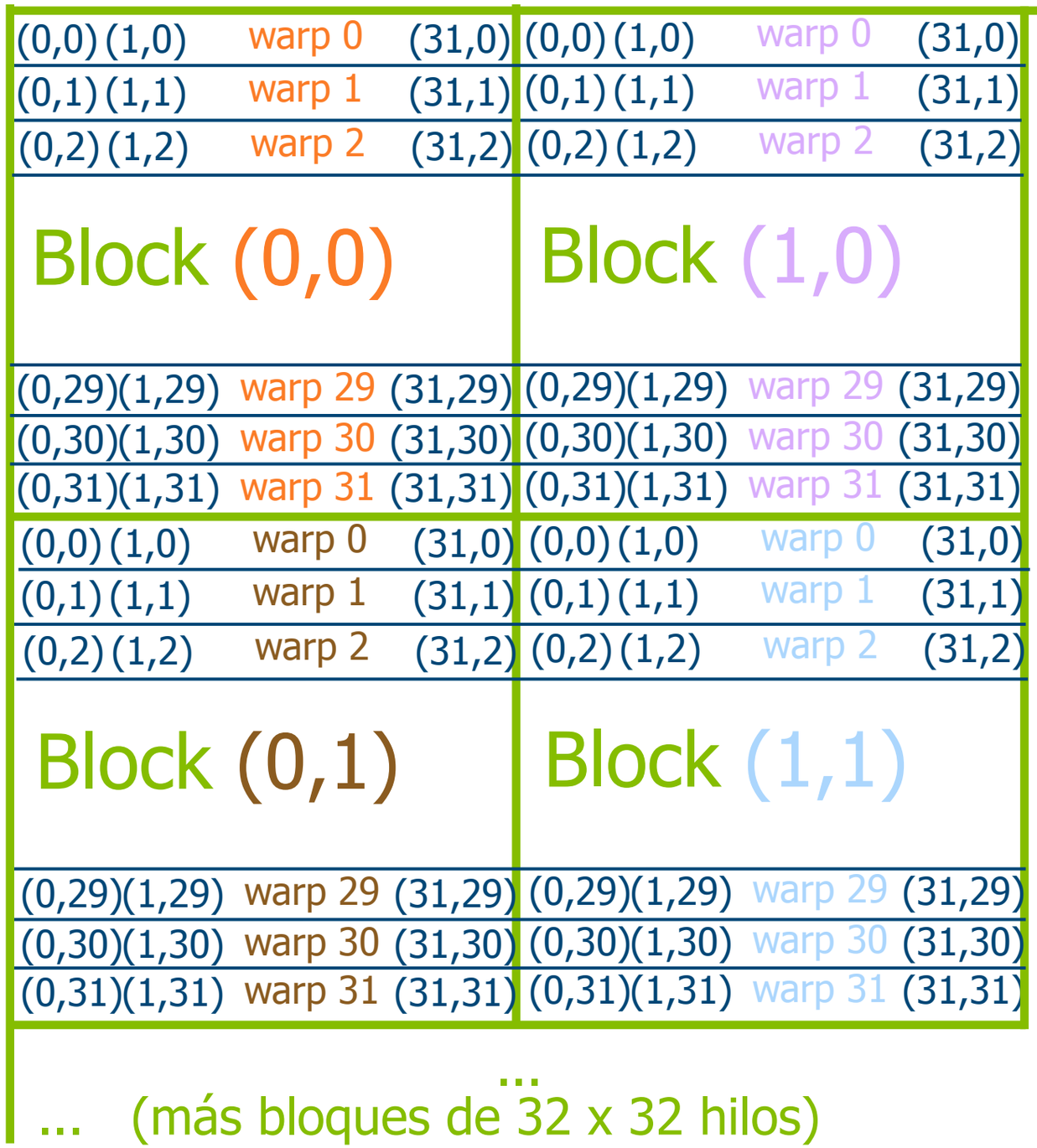


→ Consecutive threads within a warp differ in the first dimension.

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
a[0][0]	0
a[0][1]	1
a[0][31]	31
a[1][0]	0
a[31][0]	0

An example for solving conflicts to banks in shared memory

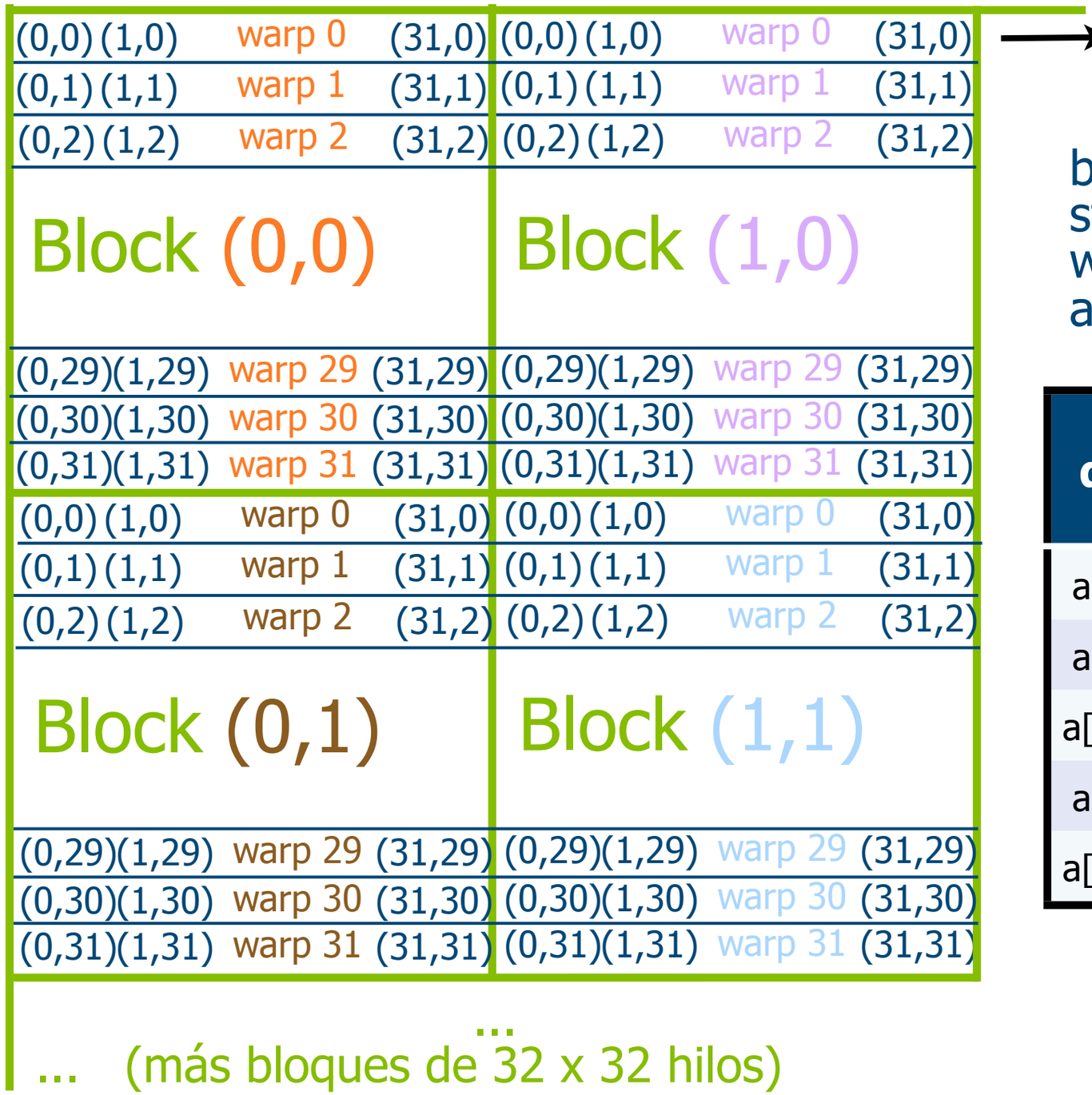


→ Consecutive threads within a warp differ in the first dimension.

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
a[0][0]	0	X
a[0][1]	1	
a[0][31]	31	
a[1][0]	0	X
a[31][0]	0	X

An example for solving conflicts to banks in shared memory

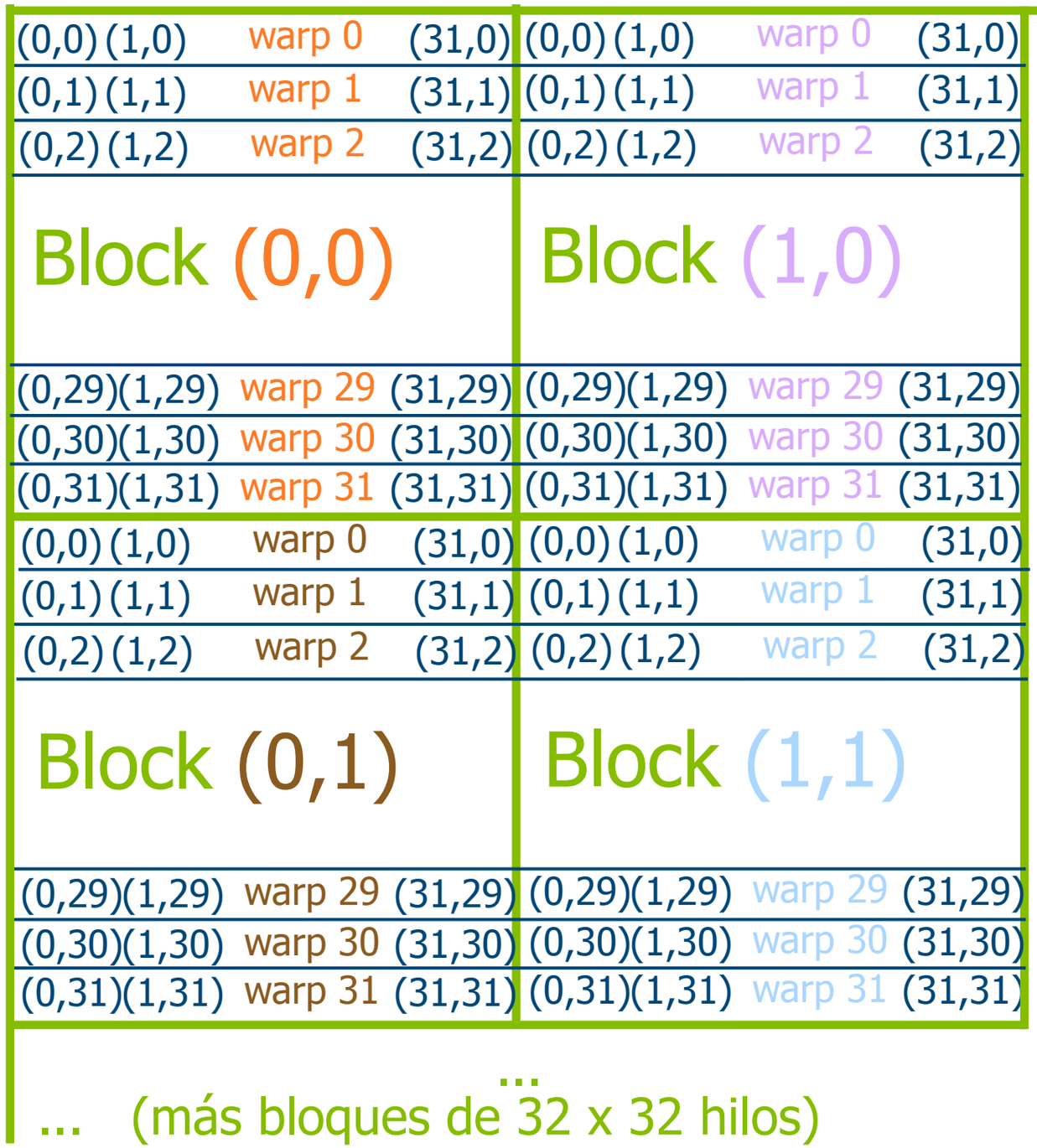


→ Consecutive threads within a warp differ in the first dimension.

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	

An example for solving conflicts to banks in shared memory



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

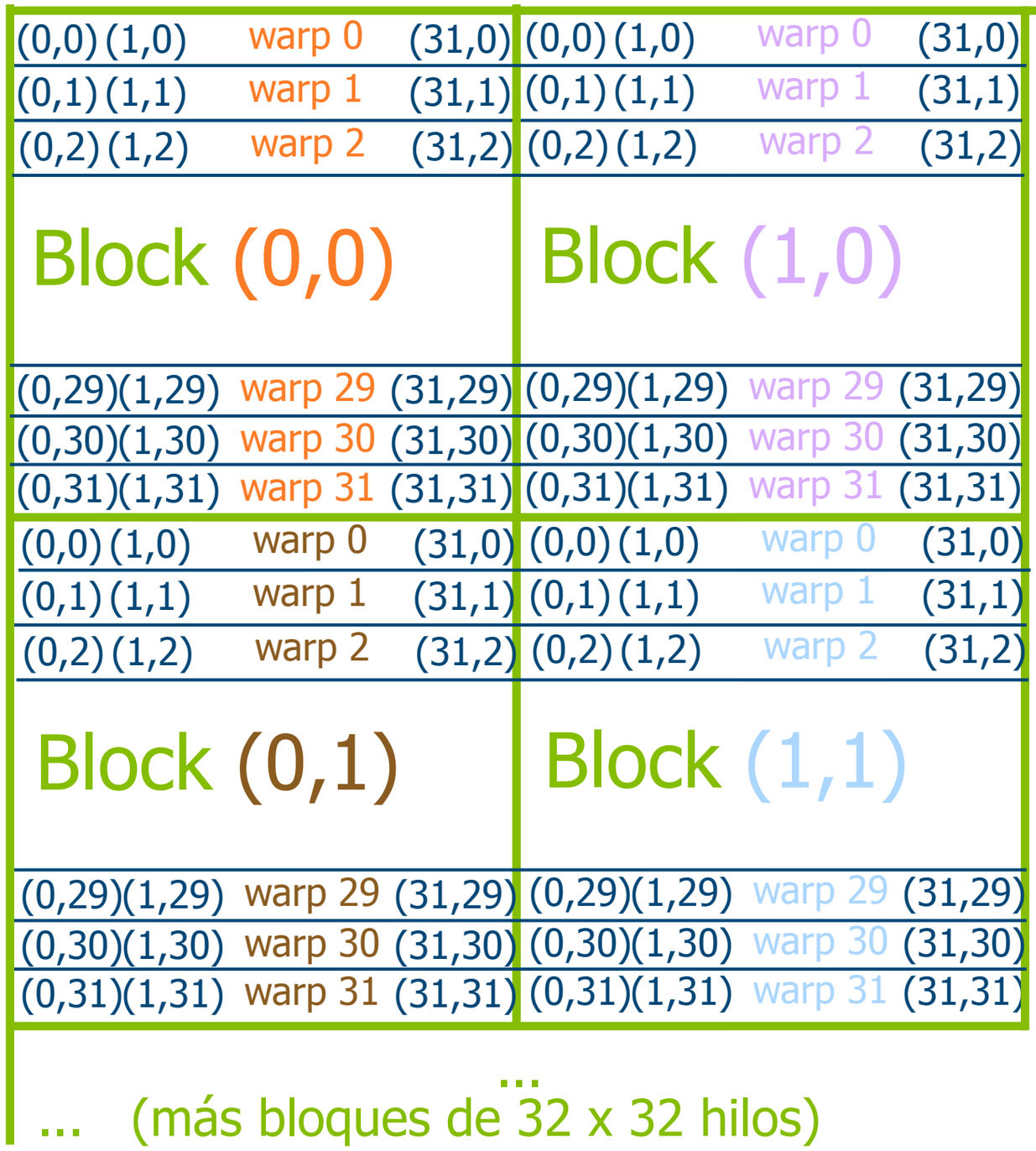
but consecutive positions of memory store data of a bidimensional matrix which differ in the second dimension: a[0][0], a[0][1], a[0][2], ...

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

An example for solving conflicts to banks in shared memory



→ 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

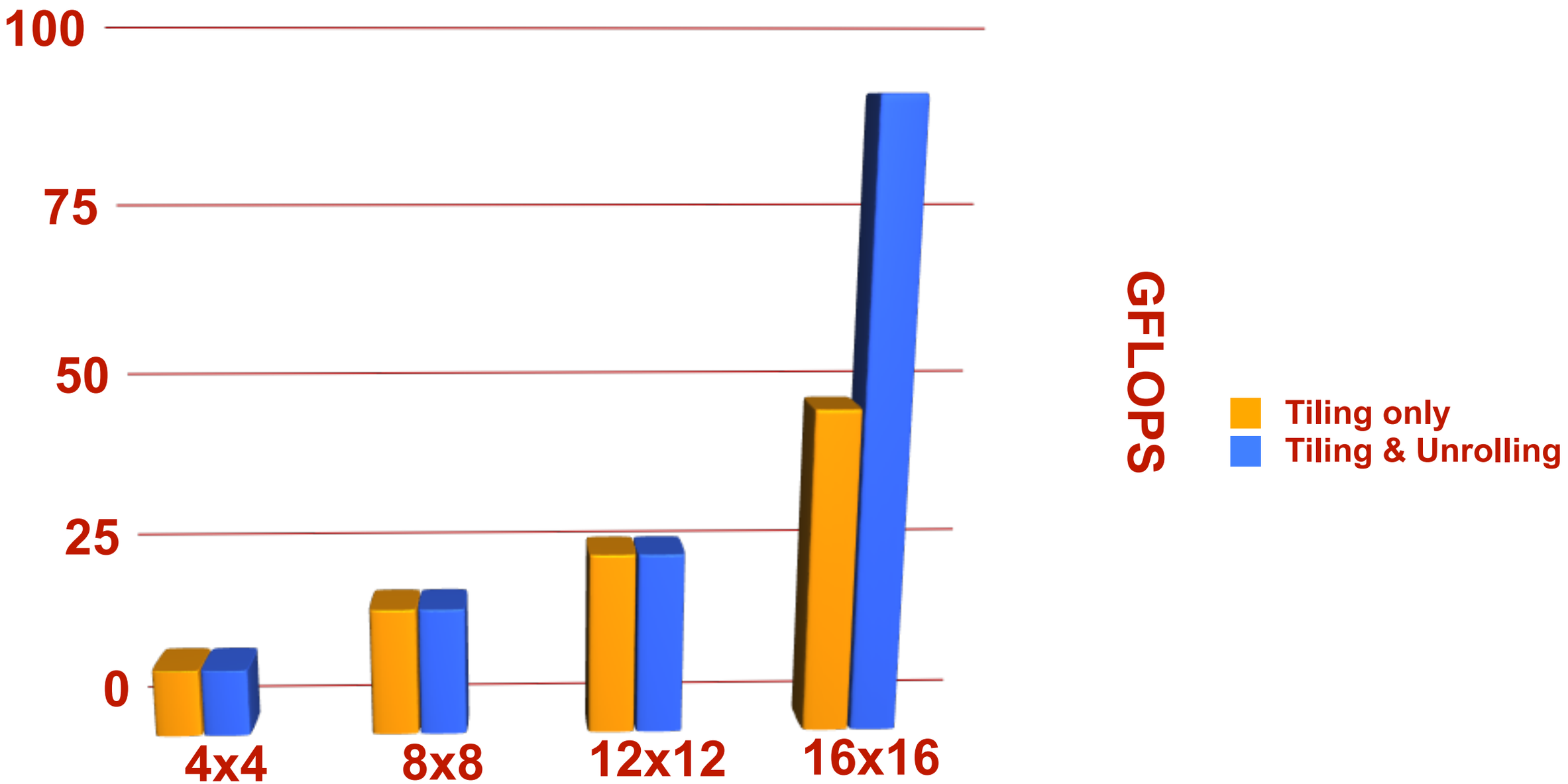
Without loop unrolling:

```
...
 __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



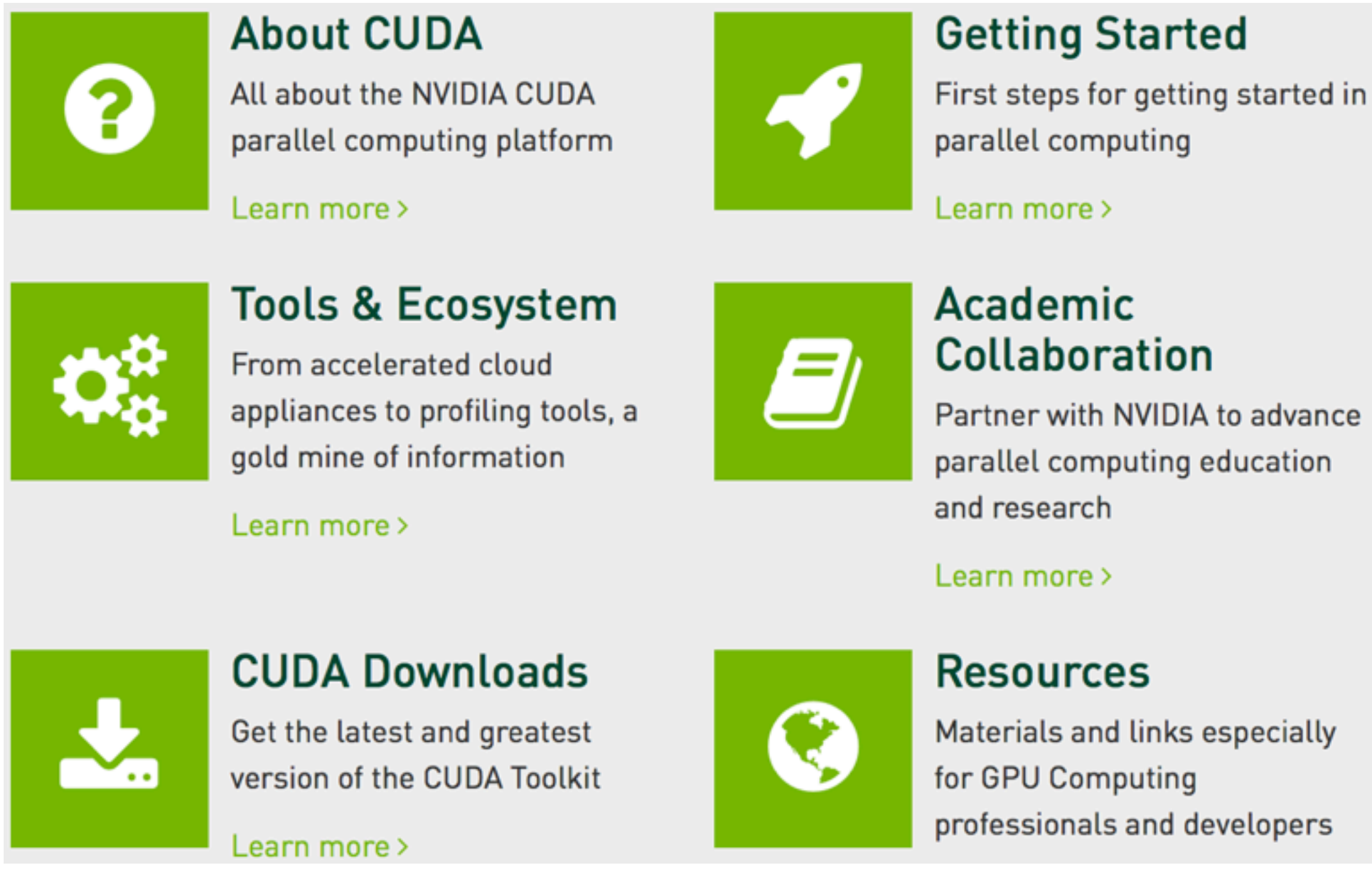
Tile size (32x32 unfeasible on G80 hardware)

VI. Bibliography and tools

NVIDIA

CUDA Zone:
The root Web for CUDA programmers

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



Getting Started

First steps for getting started in parallel computing

Learn more >

Optimized Libraries

Drop-in, Industry standard libraries replace MKL, IPP, FFTW and other widely used libraries. Some feature automatic multi-GPU scaling,

Get Started with GPU-Accelerated Libraries

Compiler Directives

Easy: simply insert hints in your code

Open: run on either CPU or GPU

Powerful: tap into the power of GPUs within minutes

Get Started with Directives

Programming Language

Develop your own parallel applications and libraries using a programming language you already know.

Get Started With:

- C/C++ using CUDA C
- Fortran using CUDA Fortran
- Python

Tools & Ecosystem

From accelerated cloud appliances to profiling tools, a gold mine of information

Accelerated Solutions

GPUs are accelerating many applications across numerous industries.

Learn more >

Numerical Analysis Tools

Applications with high arithmetic density can enjoy amazing GPU acceleration.

Learn more >

GPU-Accelerated Libraries

Adding acceleration to your application can be as easy as calling a library function.

Learn more >

Language and APIs

GPU acceleration can be accessed from most popular programming languages.

Learn more >

Performance Analysis Tools

Find the best solutions for analyzing your application's performance profile.

Learn more >

Debugging Solutions

Powerful tools can help debug complex parallel applications in intuitive ways.

Learn more >

Key Technologies

Learn more about parallel computing technologies and architectures.

Cluster Management

Managing your GPU cluster will help achieve maxium performance.

Job Scheduling

Scheduling jobs on your GPU Cluster can be simple and intuitive.

Academic Collaboration

Partner with NVIDIA to advance parallel computing education and research

Academic Programs

All about our investment in academia through our four core programs.

Learn more >

GPU Centers

A showcase of all our GPU Centers – check for your institution.

Learn more >

CUDA Fellows

Our partners who are committed to leading the use and adoption of CUDA.

Learn more >

Educators Network

A collaborative area for those looking to educate others on massively parallel programming.

Learn more >

Curriculum & Teaching Resources

Hands-on exercises and access to GPUs for your parallel programming courses.

Learn more >

Additional Resources

Materials and links especially for academia.

Learn more >

CUDA Downloads

Get the latest and greatest version of the CUDA Toolkit

NVIDIA CUDA ZONE | Getting Started | Downloads | Training | Ecosystem | Forum | Register Now | Login

Home > CUDA ZONE > Tools & Ecosystem > CUDA Toolkit > CUDA 7 Downloads

CUDA 7 Downloads

Check out:

- CUDA 7 Performance Report and Webinar Recording
- An informative webinar by Ujval Kapasi, NVIDIA's CUDA Product Manager CUDA 7 Features and Overview
- The Power of C++11 in CUDA 7, another technical blog on Parallel Forall.

If you find any issues please file a bug (requires membership of the CUDA Registered Developer Program).

Please Note: There is a recommended patch for CUDA 7.0 which resolves an issue in the cuFFT library that can lead to incorrect results for certain inputs sizes less than or equal to 1920 in any dimension when cufftSetStream() is passed a non-blocking stream (e.g., one created using the cudaStreamNonBlocking flag of the CUDA Runtime API or the CU_STREAM_NON_BLOCKING flag of the CUDA Driver API).

Windows | Linux x86 | Linux POWER8 | Mac OSX

Version	Network Installer	Local Installer
Windows 8.1 Windows 7 Win Server 2012 R2 Win Server 2008 R2	EXE (8.0MB)	EXE (939MB)
cuFFT Patch	ZIP (52MB) , README	

Windows Getting Started Guide

Windows FAQ

Q: Where is the notebook installer?
A: Previous releases of the CUDA Toolkit had separate installation packages for notebook and desktop

- Documentation
- Release Notes
- End User License Agreement
- Online Documentation
- CUDA Toolkit Overview
- Checksums

Windows | Linux x86 | Linux POWER8 | Mac OSX

Version	Network Installer	Local Package Installer	Runfile Installer
Fedora 21	Coming Soon	Coming Soon	Coming Soon
OpenSUSE 13.2	RPM (3KB)	RPM (1GB)	RUN (1.1GB)
OpenSUSE 13.1	RPM (3KB)	RPM (1GB)	RUN (1.1GB)
RHEL 7 CentOS 7	RPM (10KB)	RPM (1GB)	RUN (1.1GB)
RHEL 6 CentOS 6	RPM (18KB)	RPM (1GB)	RUN (1.1GB)
SLES 12	RPM (3KB)	RPM (1.1GB)	RUN (1.1GB)
SLES 11 (SP3)	RPM (3KB)	RPM (1.1GB)	RUN (1.1GB)
SteamOS 1.0-beta			RUN (1.1GB)
Ubuntu 14.10	DEB (3KB)	DEB (1.5GB)	RUN (1.1GB)
Ubuntu 14.04*	DEB (10KB)	DEB (902MB)	RUN (1.1GB)
Ubuntu 12.04	DEB (3KB)	DEB (1.3GB)	RUN (1.1GB)
GPU Deployment Kit	Included in Installer	Included in Installer	RUN (4MB)
cuFFT Patch		TAR (122MB) , README	

Linux Getting Started Guide

Windows | Linux x86 | Linux POWER8 | Mac OSX

Version	Network Installer	Local Package Installer	Runfile Installer
Ubuntu 14.10	DEB (3KB)	DEB (588MB)	
Ubuntu 14.04	DEB (3KB)	DEB (588MB)	
GPU Deployment Kit	n/a	n/a	RUN (1.7MB)
cuFFT Patch		TAR (105MB) , README	

Windows | Linux x86 | Linux POWER8 | Mac OSX

Version	Network Installer	Local Installer
10.9 10.10	DMG (0.4MB)	PKG (977MB)
cuFFT Patch		TAR (104MB) , README

Resources

Materials and links especially for GPU Computing professionals and developers

Downloads

- CUDA Toolkit
- CUDA Downloads
- CUDA Archives
- CUDA Developer Home
- CUDA Developer Sign Up

Docs and References

- Online Documentation
- Architecture References

Education & Training

- Training Materials
- GTC Express Webinars
- GTC Presentations
- Udacity -Free Courses
- Coursera

CUDA Powered Processors

- Tesla
- Quadro
- GeForce
- GRID
- Tegra

Community

- GPU Computing Forums
- Meetups in Your City
- Parallel Forall Blog
- YouTube
- Stackoverflow
- gpgpu.org
- gpucomputing.net

Keep Informed

- Twitter
- Facebook
- CUDA Newsletter
- Parallel Forall – RSS feed

Partners & Ecosystem

- Tools & Ecosystem
- Accelerated Libraries
- Programming Language

Success Stories

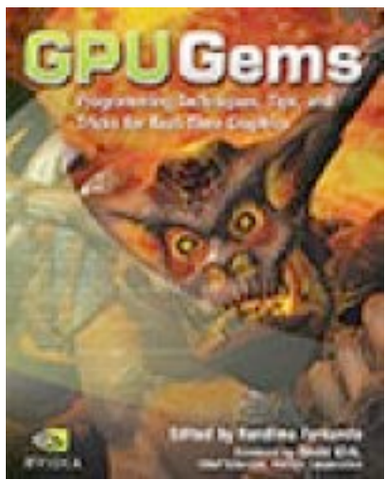
- Industry Domains
- Industry Applications
- Industry Success Stories
- CUDA Spotlights

Contact Us

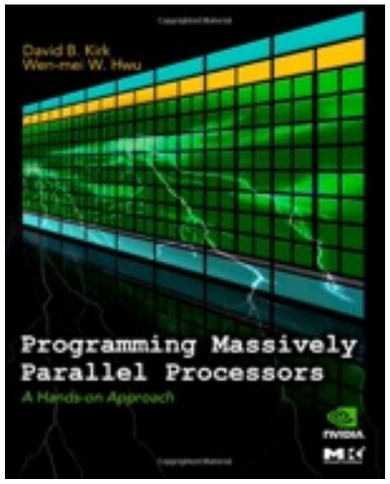
- Contact Form
- Submit Bugs
- View Your Submitted Bugs
- Forums
- Academic Programs

CUDA books: From 2007 to 2015

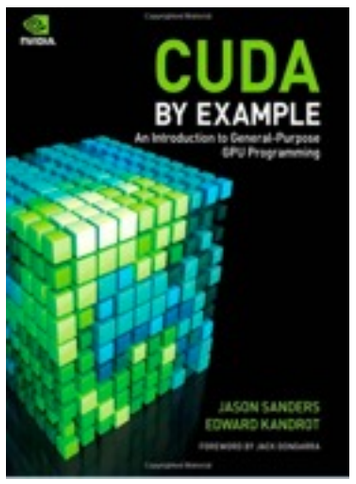
- GPU Gems series: 1, 2, 3 [developer.nvidia.com/gpugems]
- List of CUDA books in [developer.nvidia.com/suggested-reading]



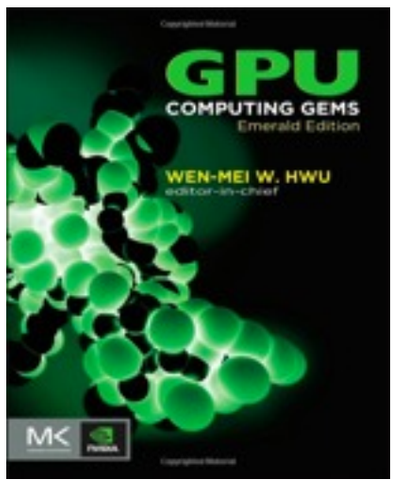
Sep'07



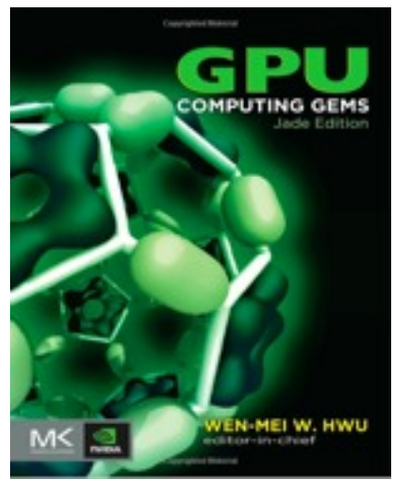
Feb'10



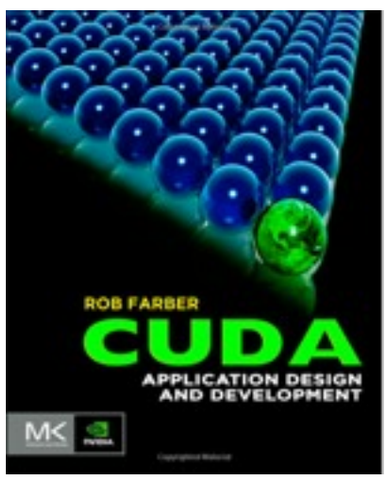
Jul'10



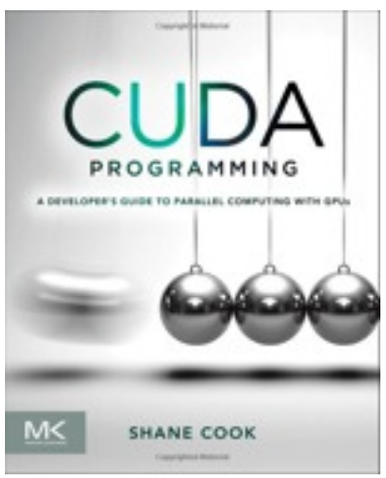
Abr'11



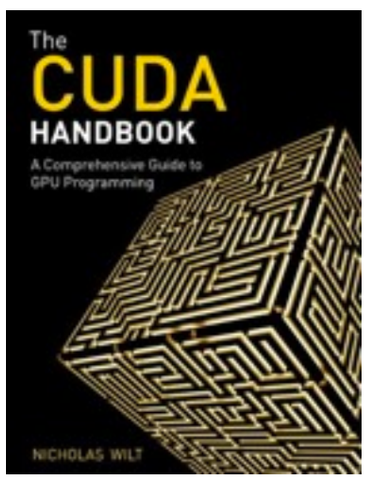
Oct'11



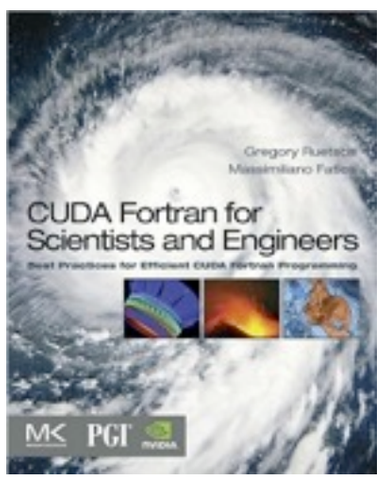
Nov'11



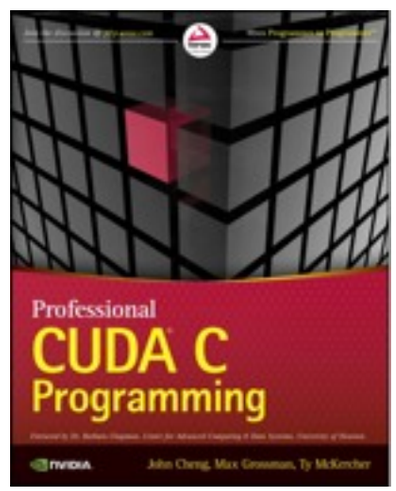
Dic'12



Jun'13



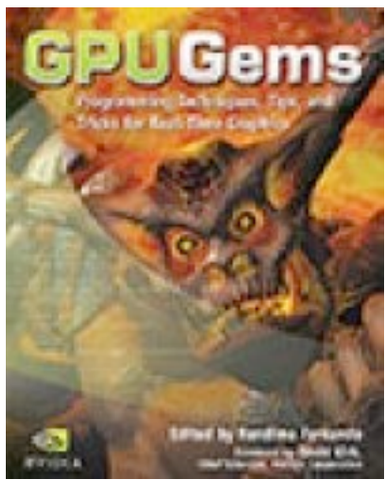
Oct'13



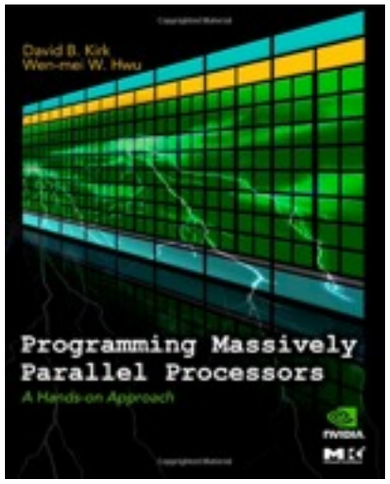
Sep'14

CUDA books: From 2007 to 2015

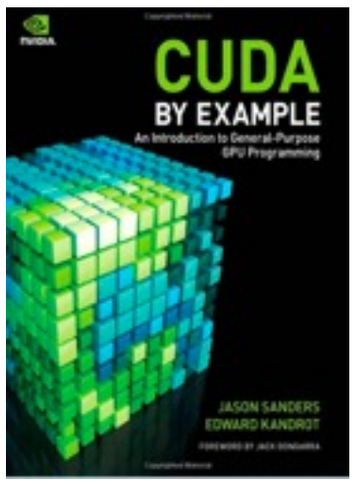
- GPU Gems series: 1, 2, 3 [developer.nvidia.com/gpugems]
- List of CUDA books in [developer.nvidia.com/suggested-reading]



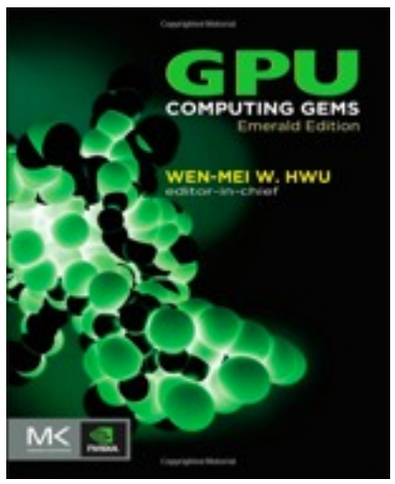
Sep'07



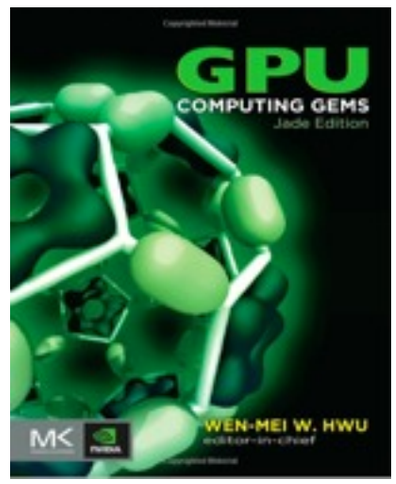
Feb'10



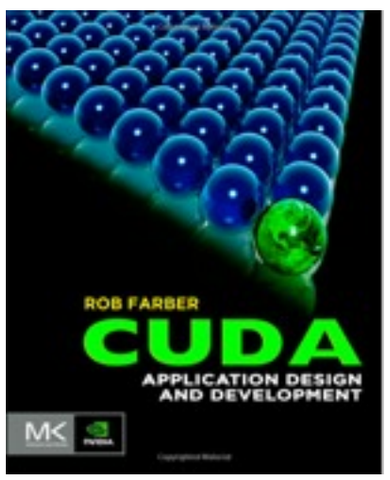
Jul'10



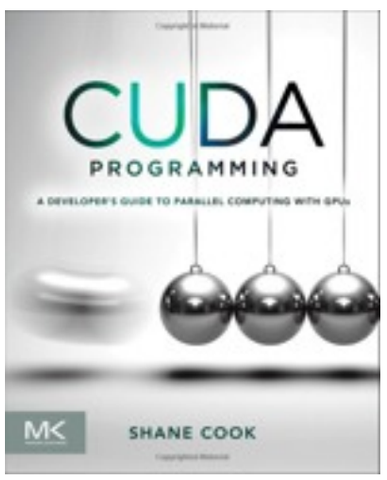
Abr'11



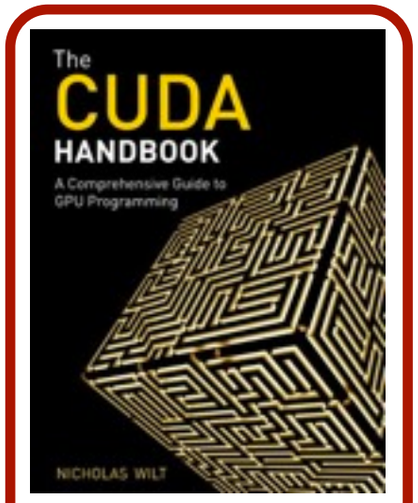
Oct'11



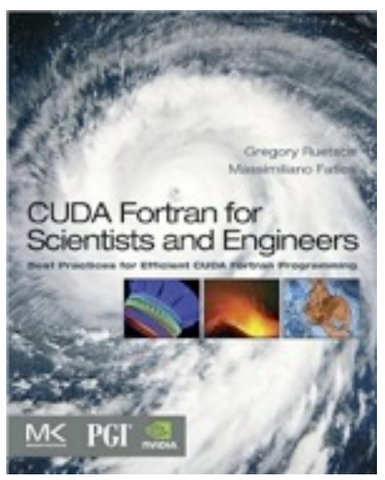
Nov'11



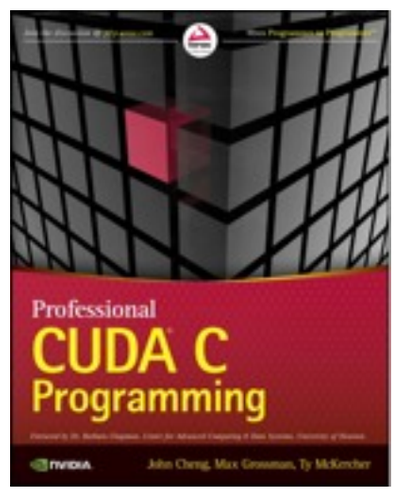
Dic'12



Jun'13



Oct'13



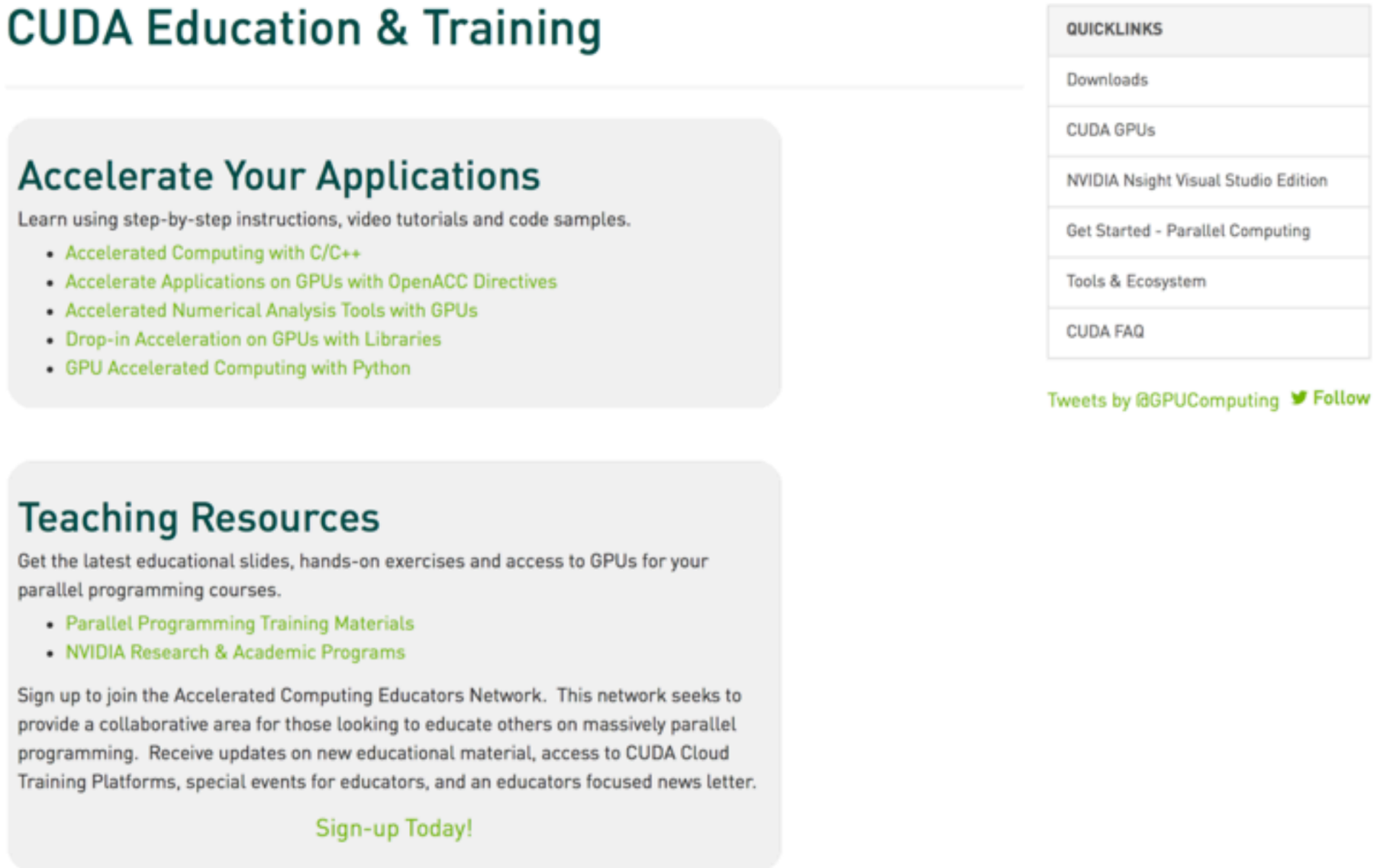
Sep'14

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 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).

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

- [developer.nvidia.com/cuda-education-training] (also available from CUDA Zone -> Resources -> Training materials)

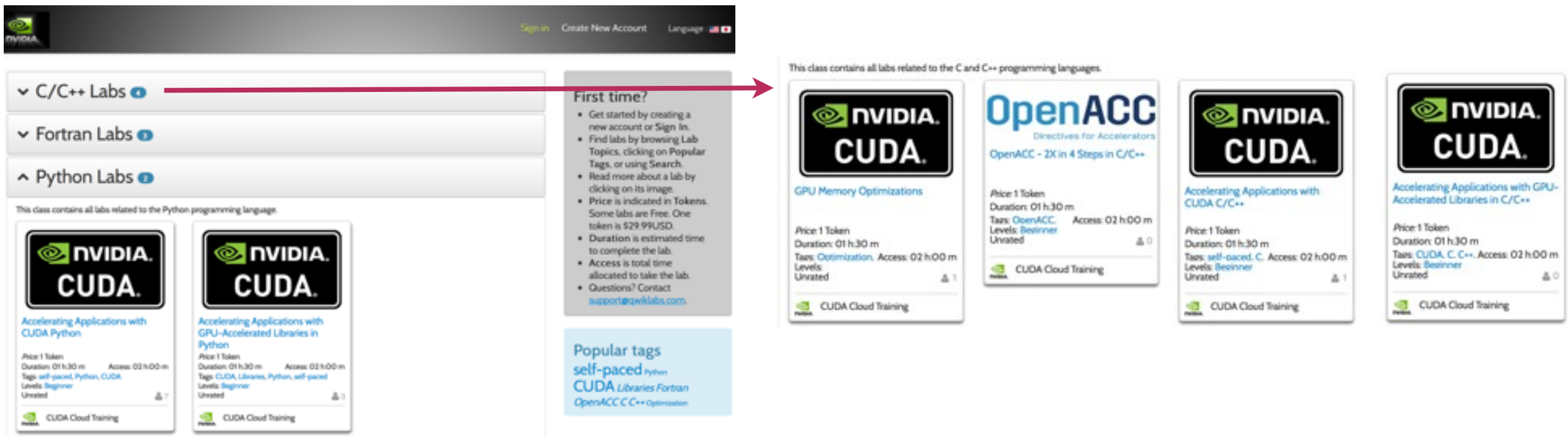


Courses on-line (free access)

- 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 (at UIUC): 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

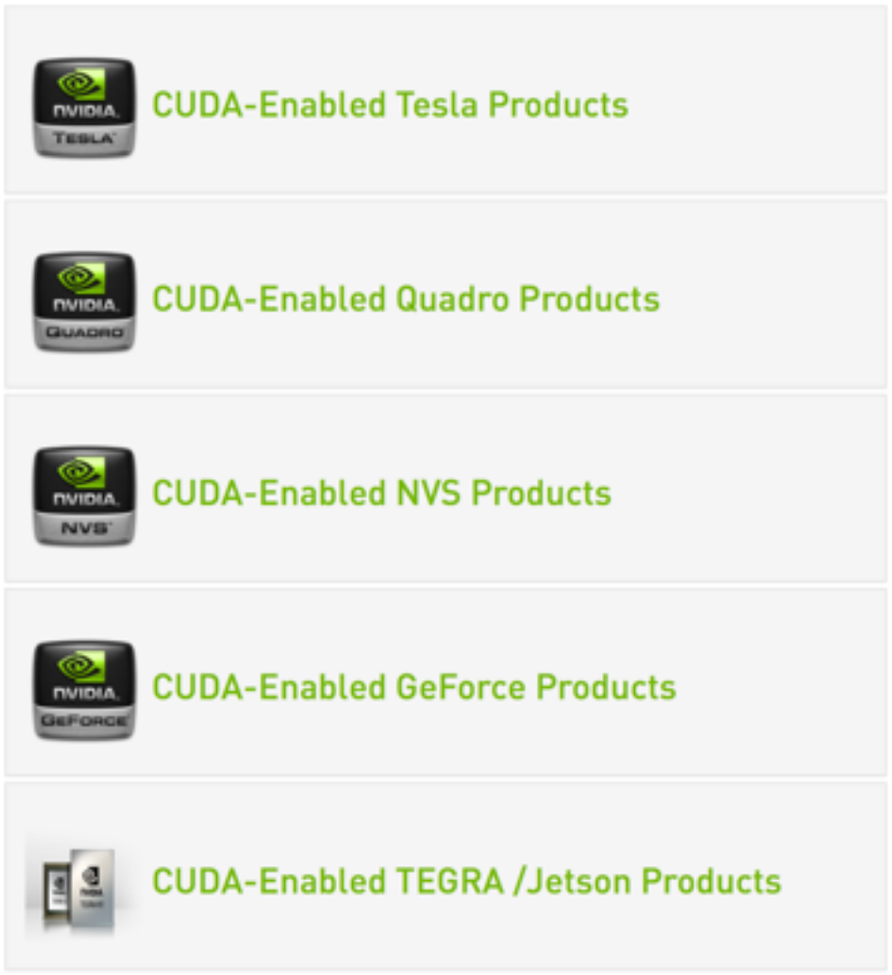
- Talks recorded at GTC (Graphics Technology Conference):
  - More than 500 talks available on each of the last four editions.
  - [www.gputechconf.com/gtcnew/on-demand-gtc.php]
- Webinars about GPU computing:
  - List of past talks on video (mp4/wmv) and slides (PDF).
  - List of incoming on-line talks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

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 8.0.
  - 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 last tool for tuning code: 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:
  - "Getting Started with OpenACC", by Jeff Larkin.
  - "New Features in CUDA 7.5", by Mark Harris.
  - "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