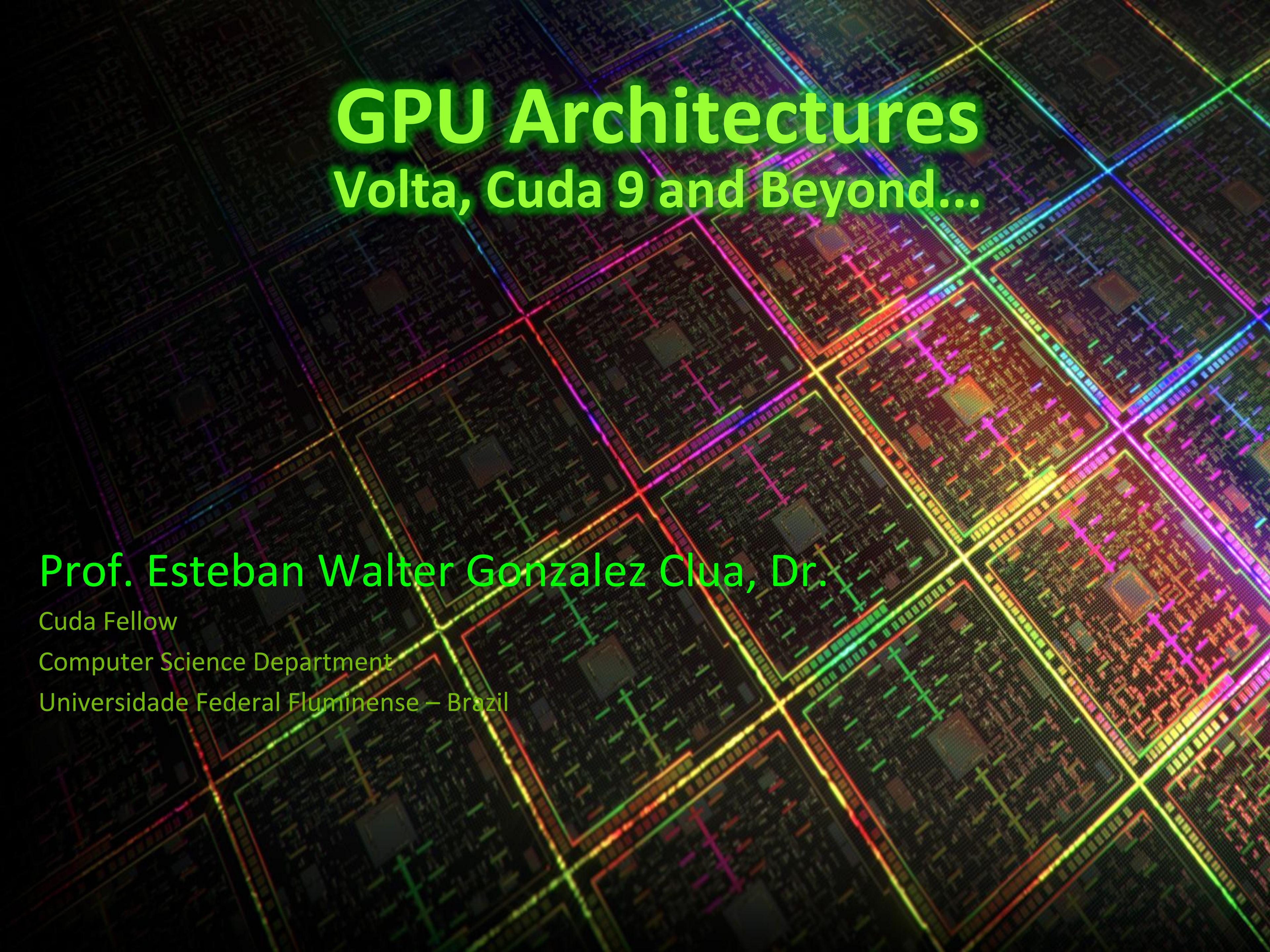


GPU Architectures
Volta, Cuda 9 and Beyond...

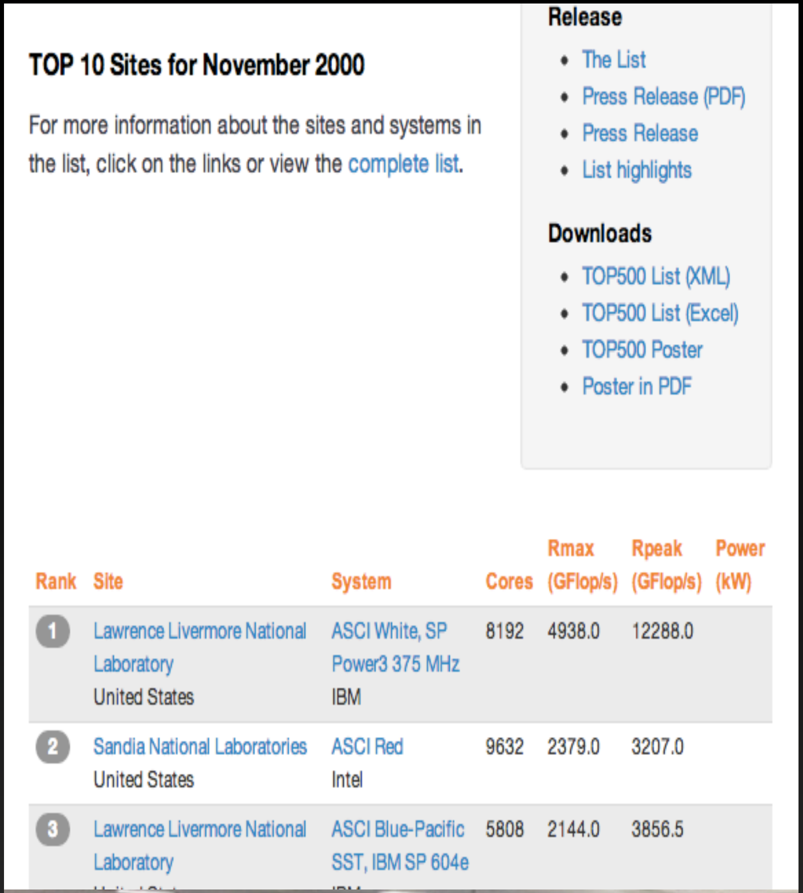
Prof. Esteban Walter Gonzalez Clua, Dr.

Cuda Fellow

Computer Science Department

Universidade Federal Fluminense – Brazil

Universidade Federal Fluminense
Rio de Janeiro - Brasil



TOP 10 Sites for November 2000

For more information about the sites and systems in the list, click on the links or view the complete list.

Release

- The List
- Press Release (PDF)
- Press Release
- List highlights

Downloads

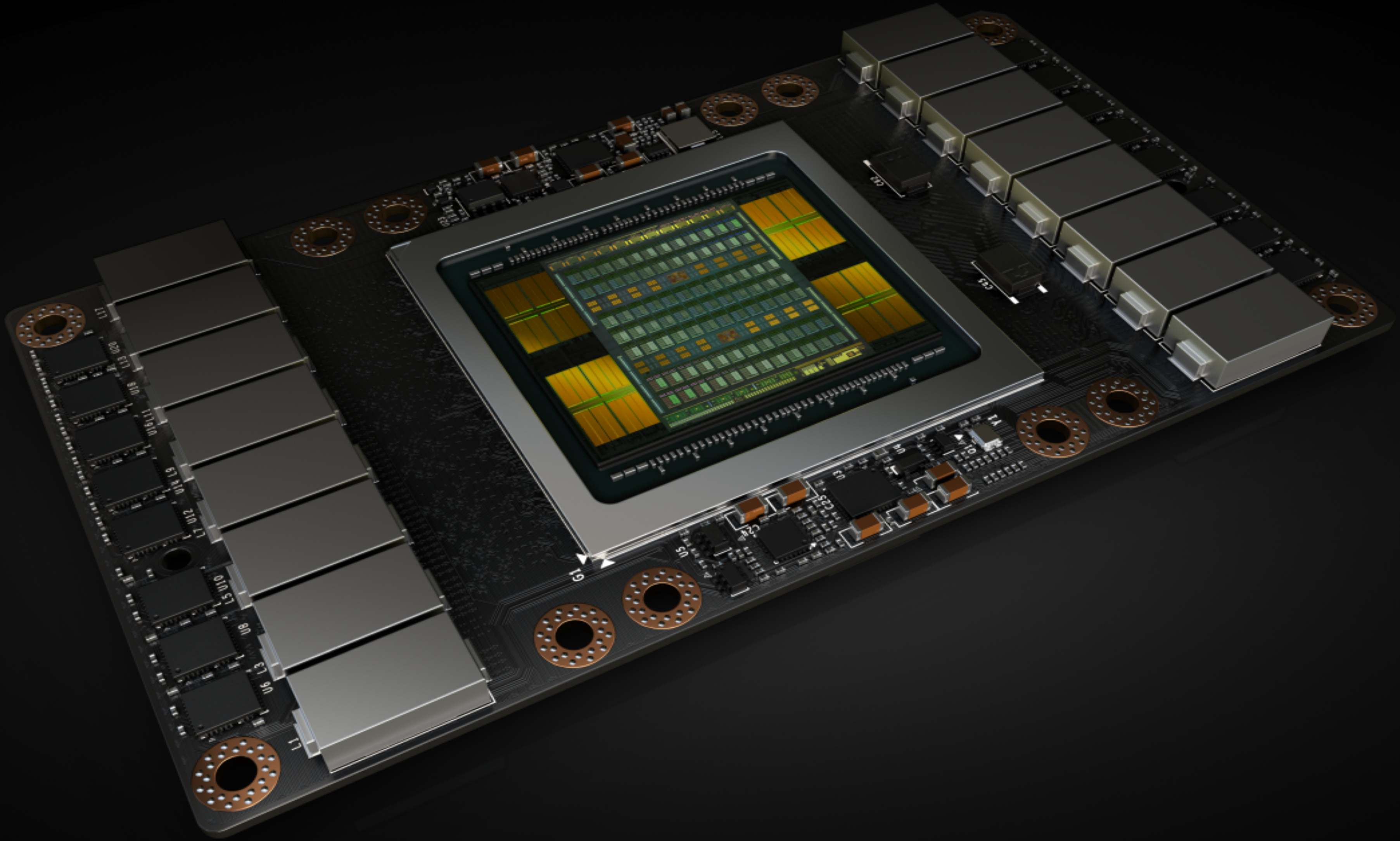
- TOP500 List (XML)
- TOP500 List (Excel)
- TOP500 Poster
- Poster in PDF

Rank	Site	System	Cores	Rmax (GFlop/s)	Rpeak (GFlop/s)	Power (kW)
1	Lawrence Livermore National Laboratory United States	ASCI White, SP Power3 375 MHz IBM	8192	4938.0	12288.0	
2	Sandia National Laboratories United States	ASCI Red Intel	9632	2379.0	3207.0	
3	Lawrence Livermore National Laboratory	ASCI Blue-Pacific SST, IBM SP 604e	5808	2144.0	3856.5	

1 Million Watts

Volta

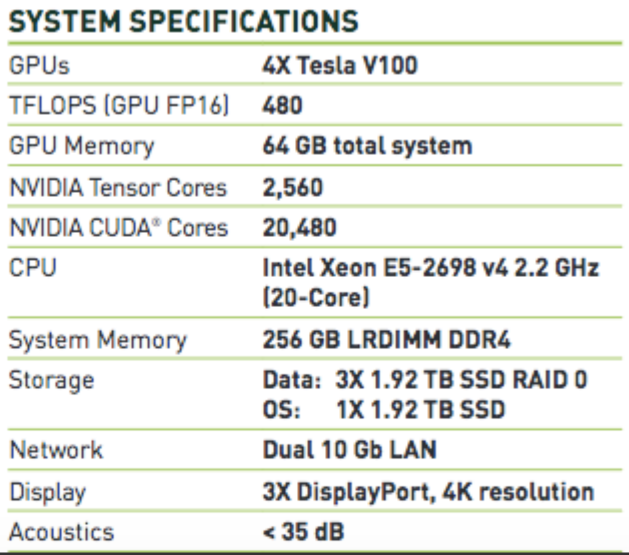
The most advanced accelerator ever built



Capacity

- *7.5 TFLOP/s of double precision floating-point (FP64) performance;*
- *15 TFLOP/s of single precision (FP32) performance;*
- *120 Tensor TFLOP/s of mixed-precision matrix-multiply-and-accumulate.*

Personal Clusters

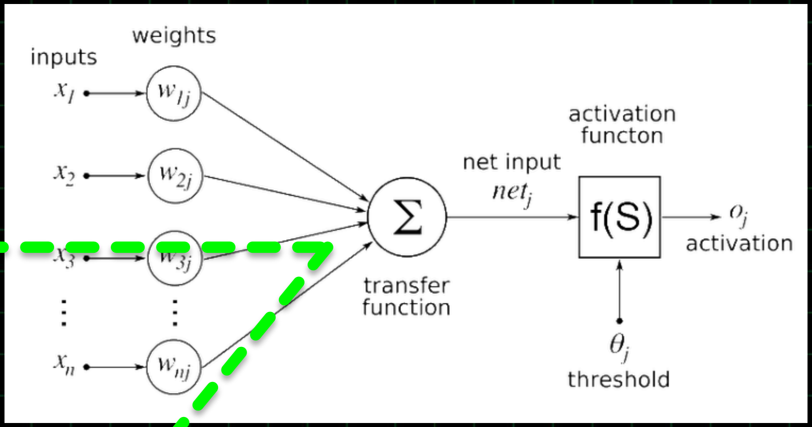


SYSTEM SPECIFICATIONS

GPUs	4X Tesla V100
TFLOPS (GPU FP16)	480
GPU Memory	64 GB total system
NVIDIA Tensor Cores	2,560
NVIDIA CUDA® Cores	20,480
CPU	Intel Xeon E5-2698 v4 2.2 GHz (20-Core)
System Memory	256 GB LRDIMM DDR4
Storage	Data: 3X 1.92 TB SSD RAID 0 OS: 1X 1.92 TB SSD
Network	Dual 10 Gb LAN
Display	3X DisplayPort, 4K resolution
Acoustics	< 35 dB

More than the sum of all Top 500 systems of the year 2000

Big Bang of IA



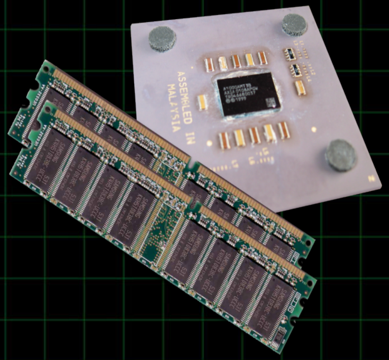
Volta Architecture

GPU Programming paradigms

[illegible]

#1 – we are talking about Heterogeneous Computing

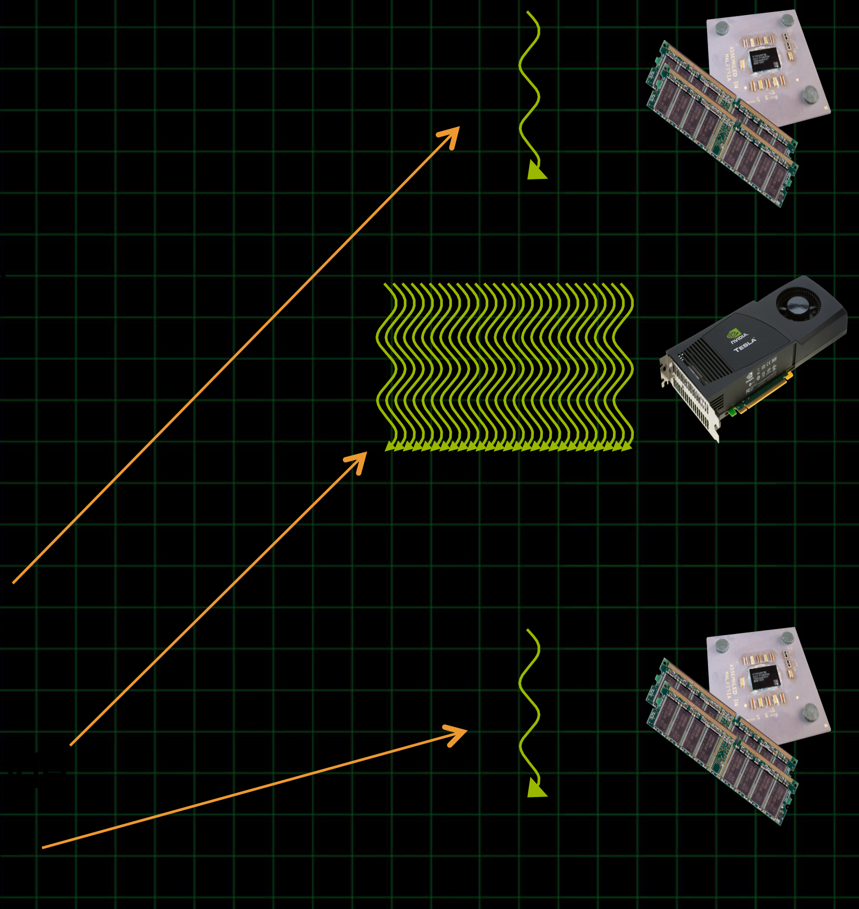
- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)


Host

Device

Heterogeneous Computing

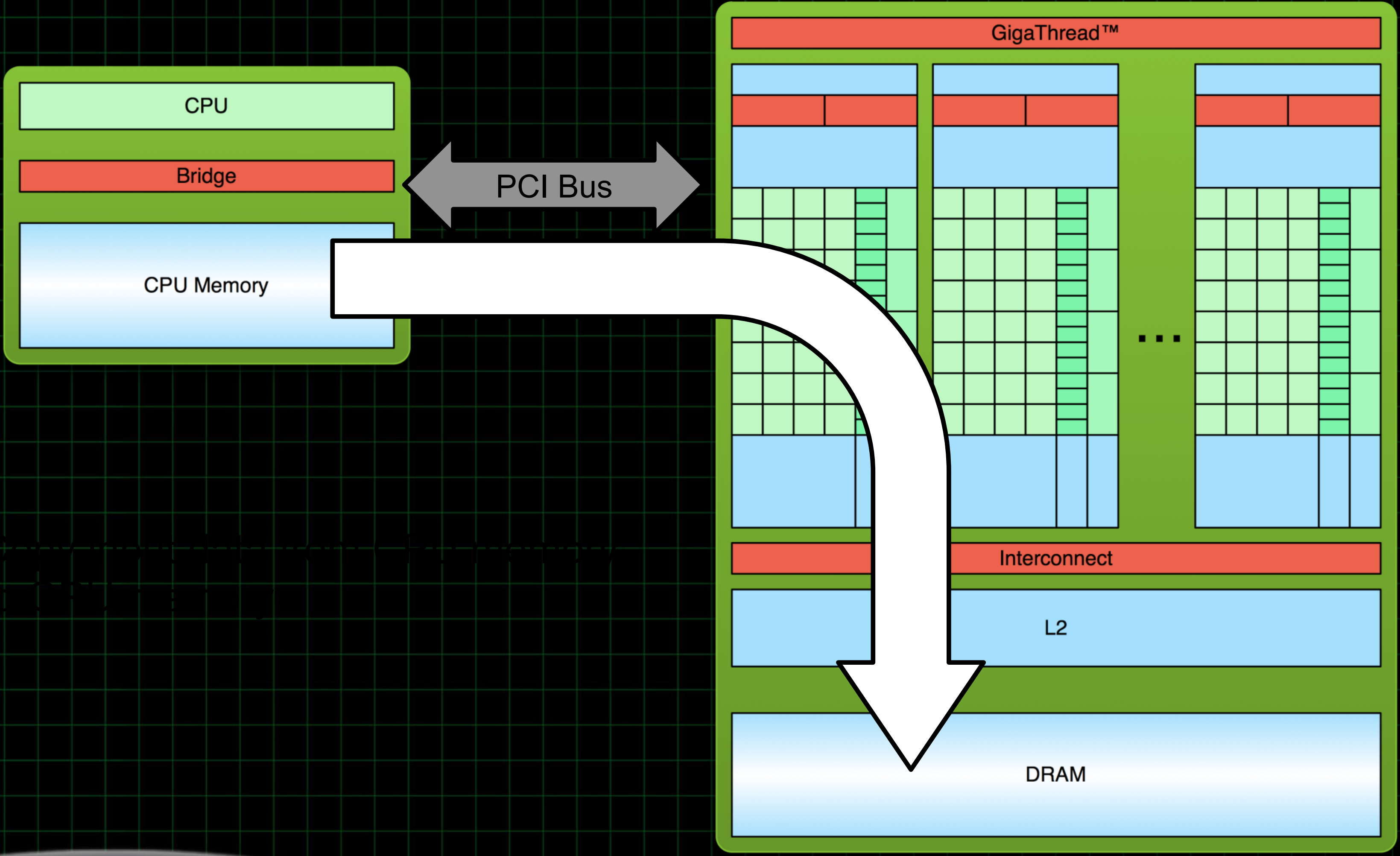
```
#include <iostream>
#include <algorithm>

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;
}
```



This slide is credited to Mark Harris (nvidia)

#2 – Memory bandwidth matters!...

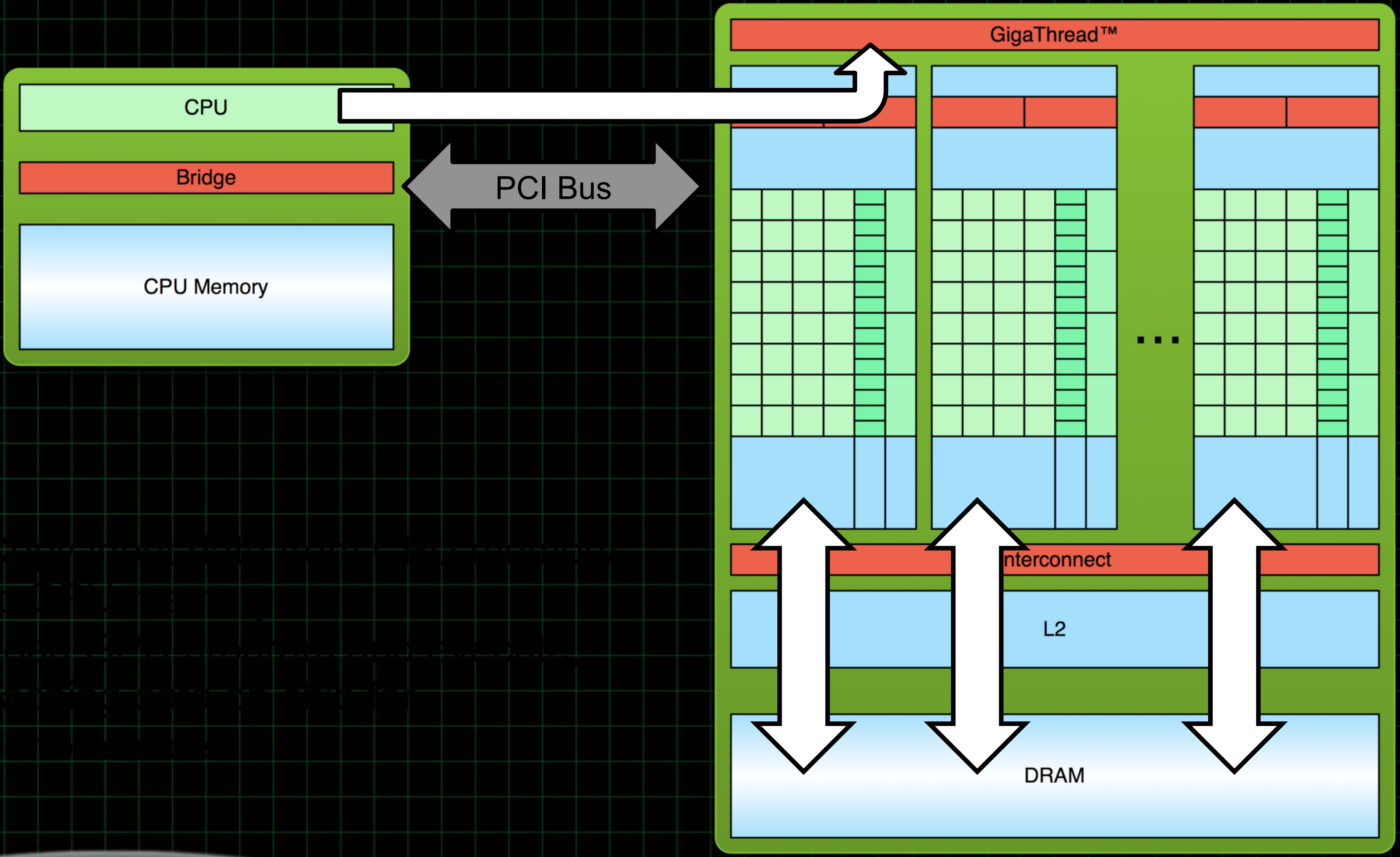
GPU Computing Flow



1. Copy input data from CPU memory to GPU memory

This slide is credited to Mark Harris (nvidia)

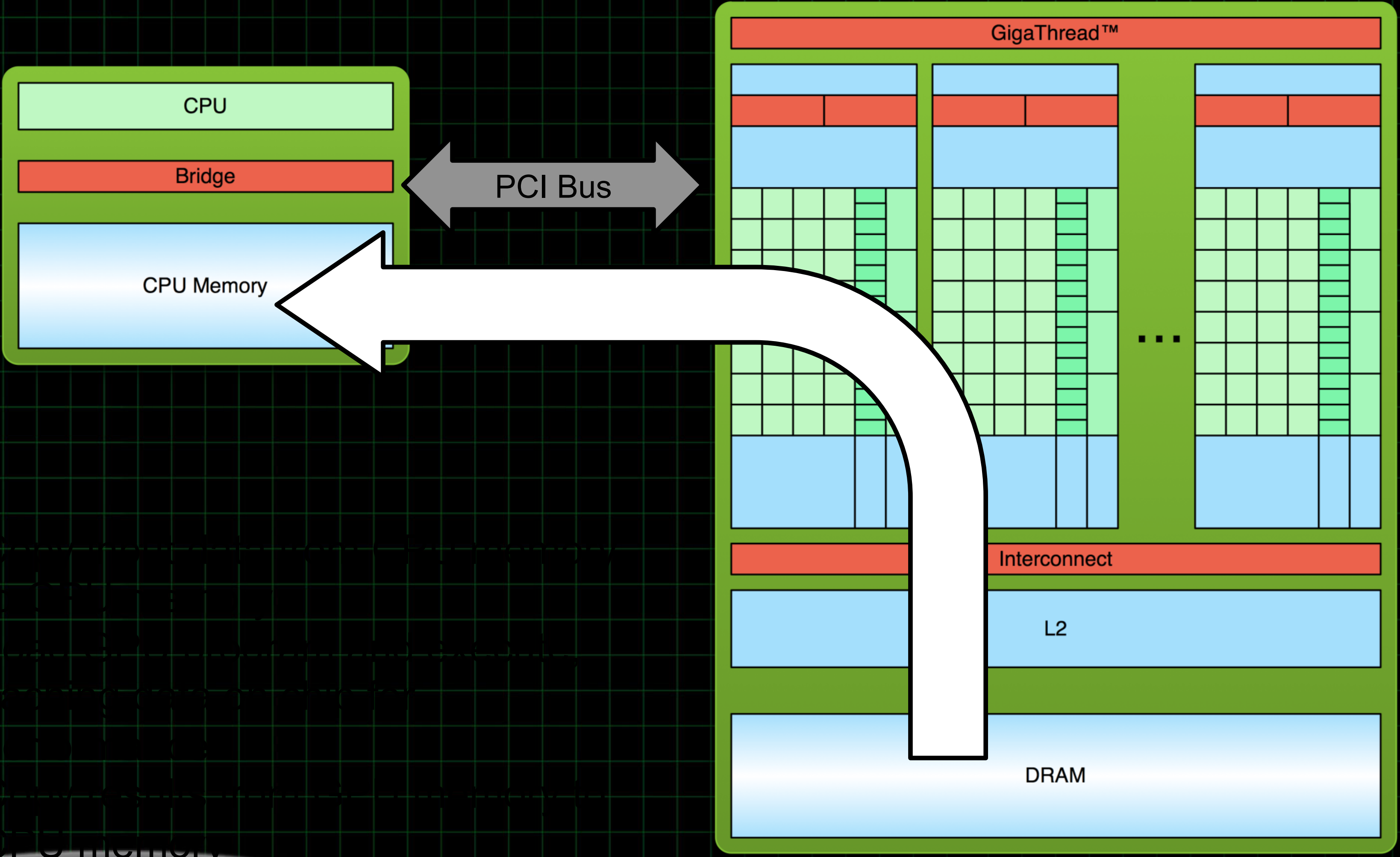
GPU Computing Flow



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

This slide is credited to Mark Harris (nvidia)

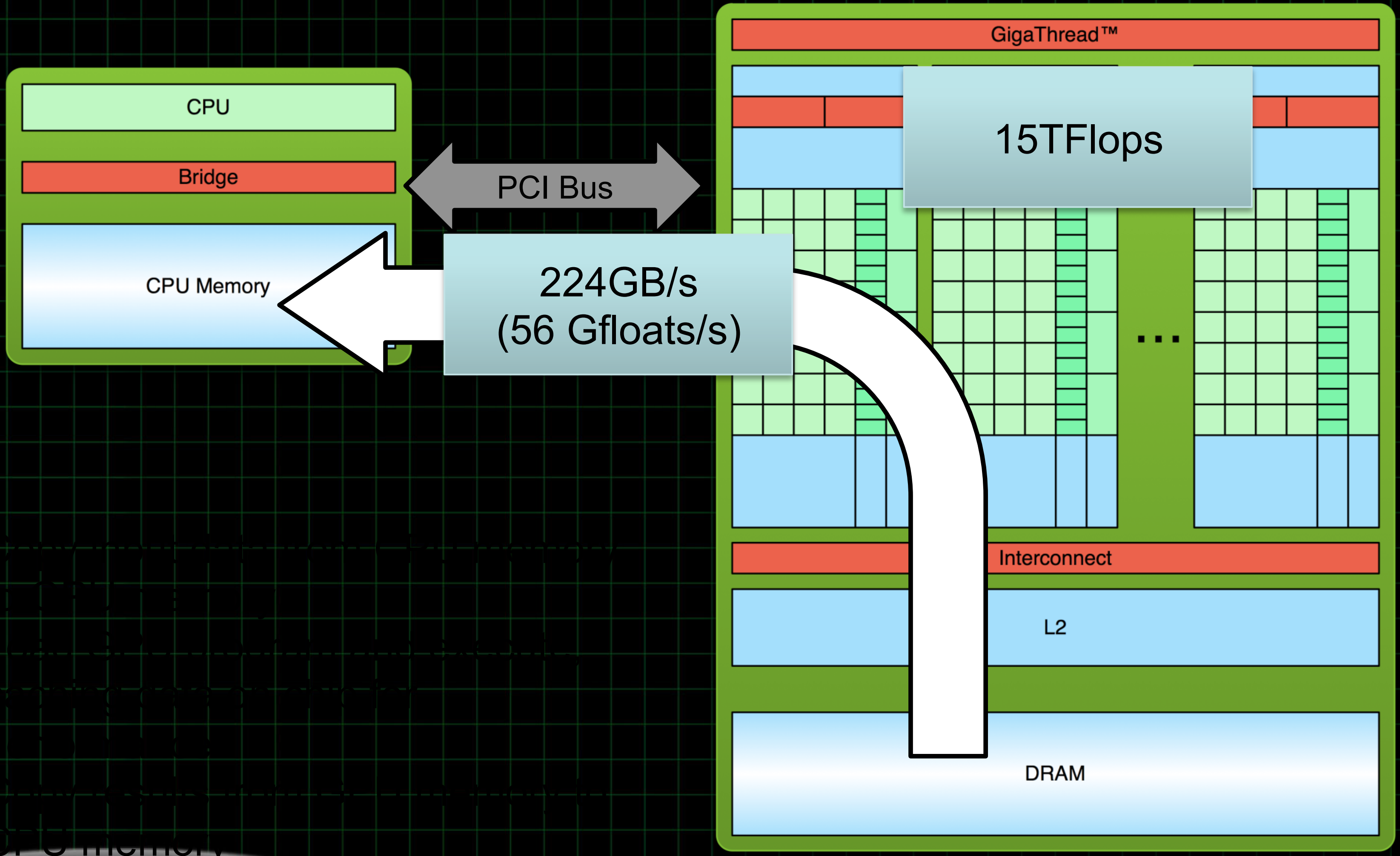
GPU Computing Flow



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

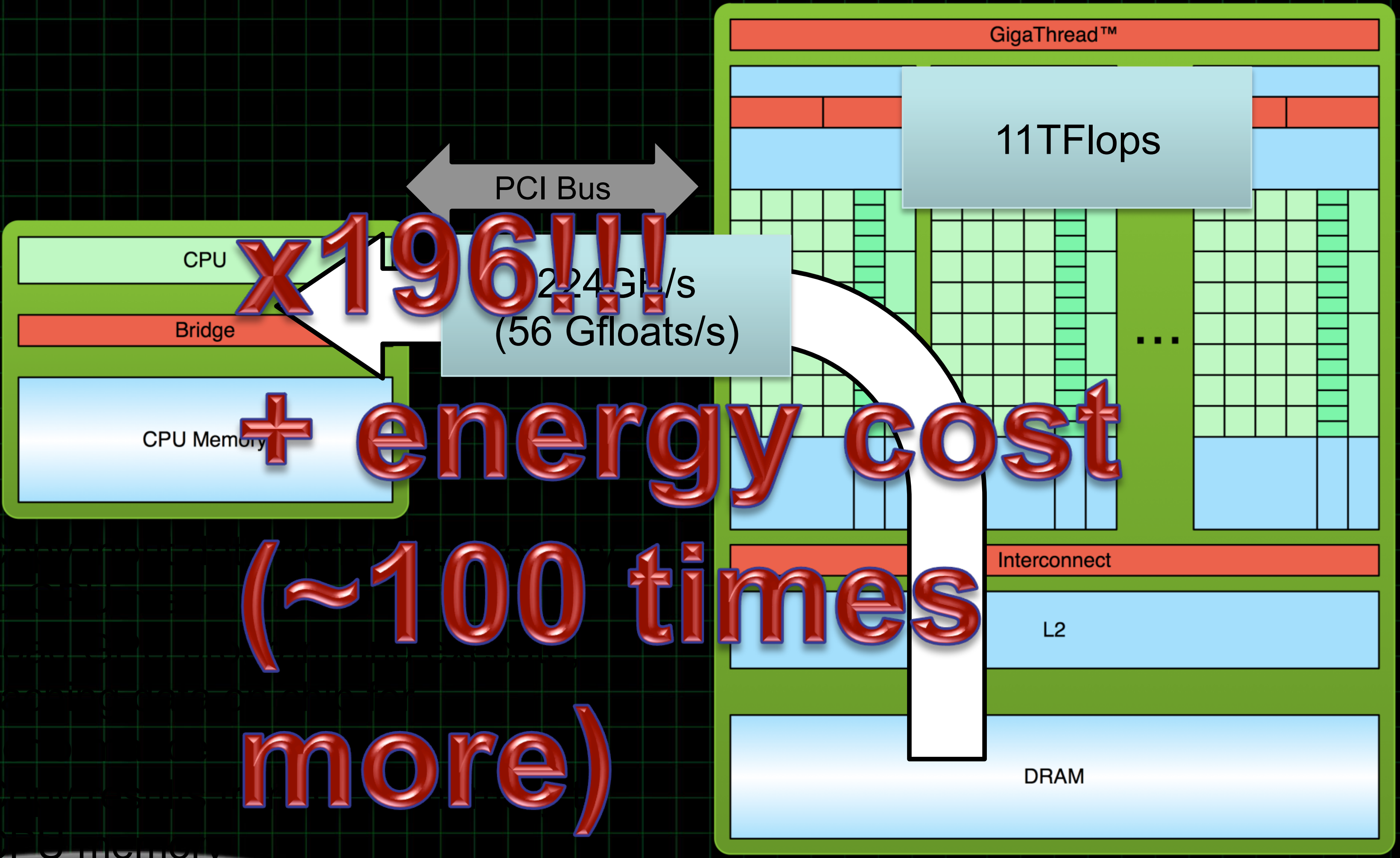
This slide is credited to Mark Harris (nvidia)

GPU Computing Flow



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

GPU Computing Flow

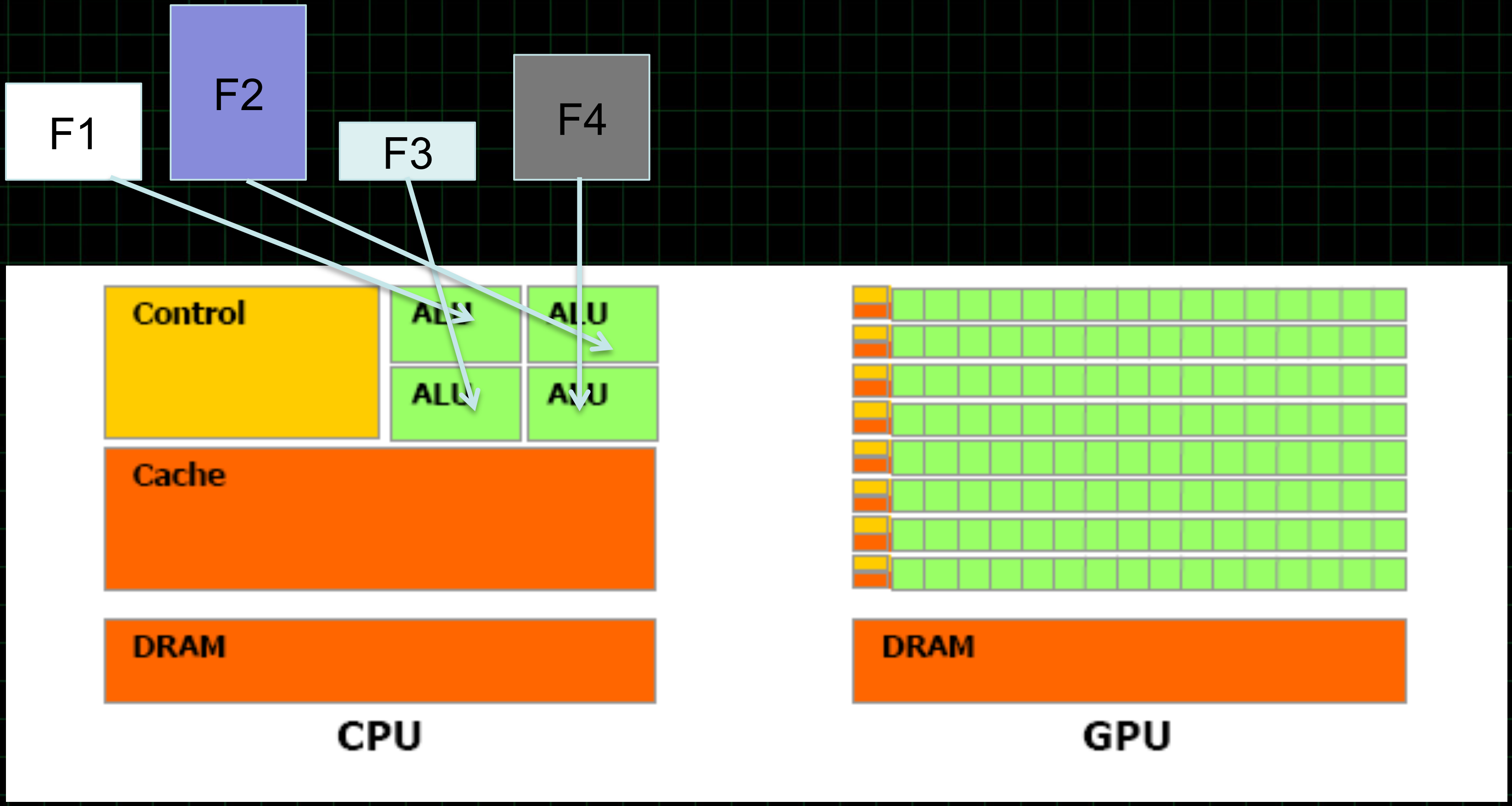


x196!!!

+ energy cost (~100 times more)

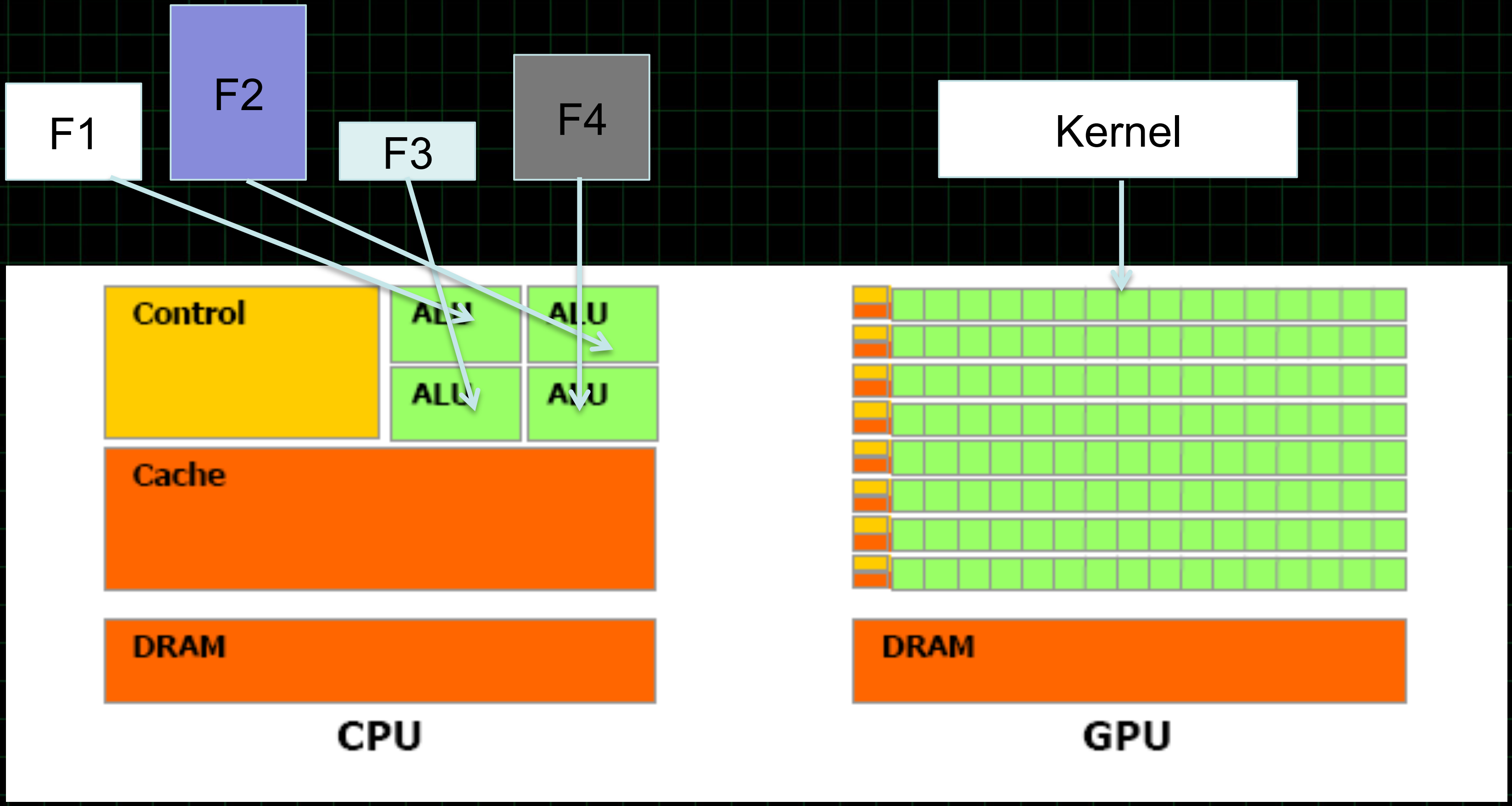
#3 – 1 kernels, lots of threads...

How things work at GPU x CPU



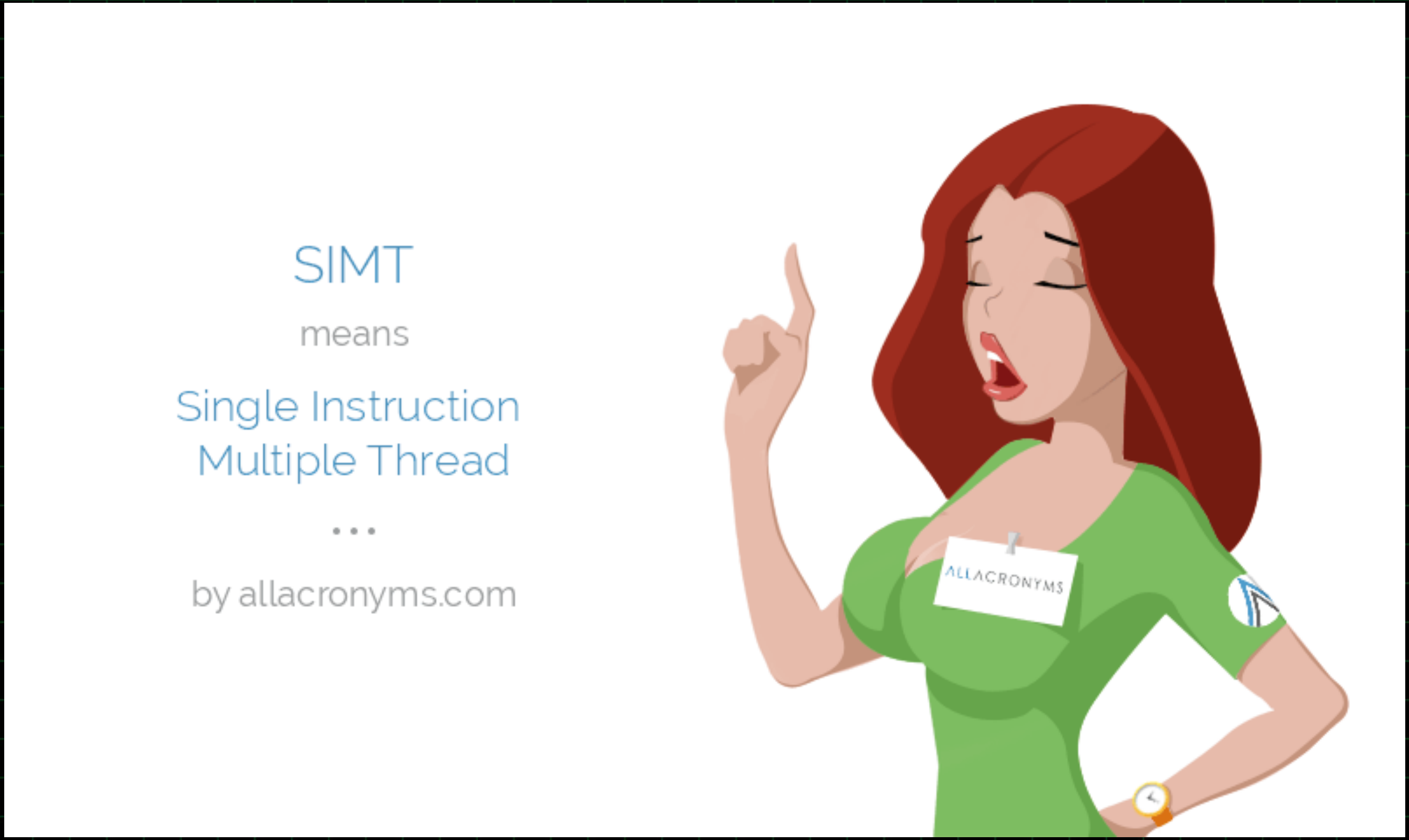
[illegible] computation, 99% to [illegible] to combat latency.

How things work at GPU x CPU



[illegible]
computation, 99% to [illegible]
to combat latency.

Modelo SIMT



SIMT

means

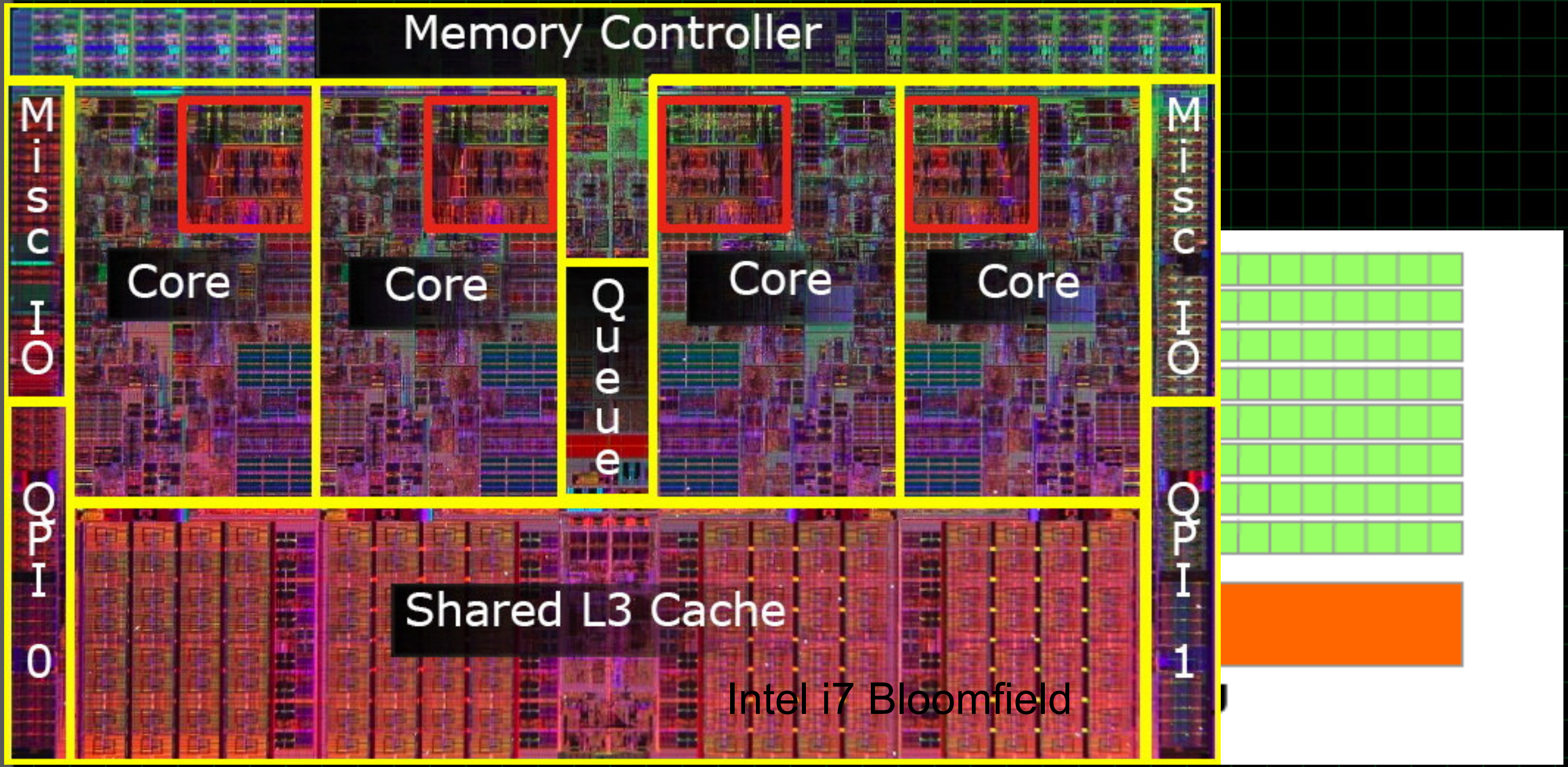
Single Instruction
Multiple Thread

• • •

by allacronyms.com

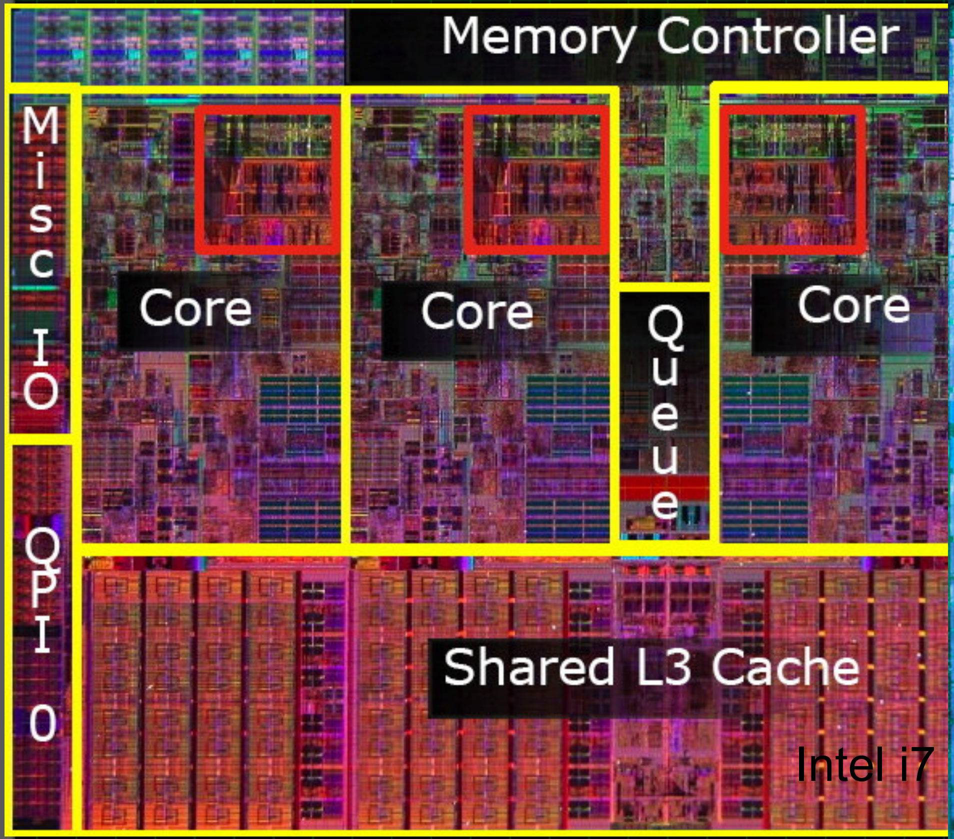
ALLACRONYMS

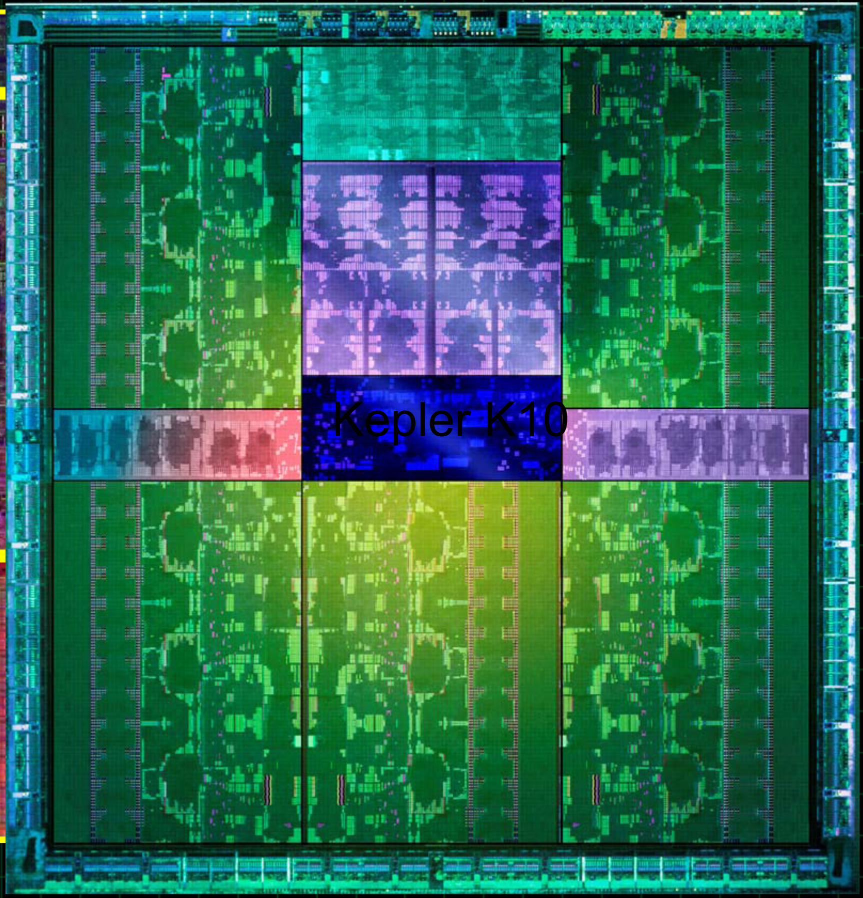
GPU x CPU



[illegible]
computation, 99% to [illegible]
to combat latency.

GPU x CPU



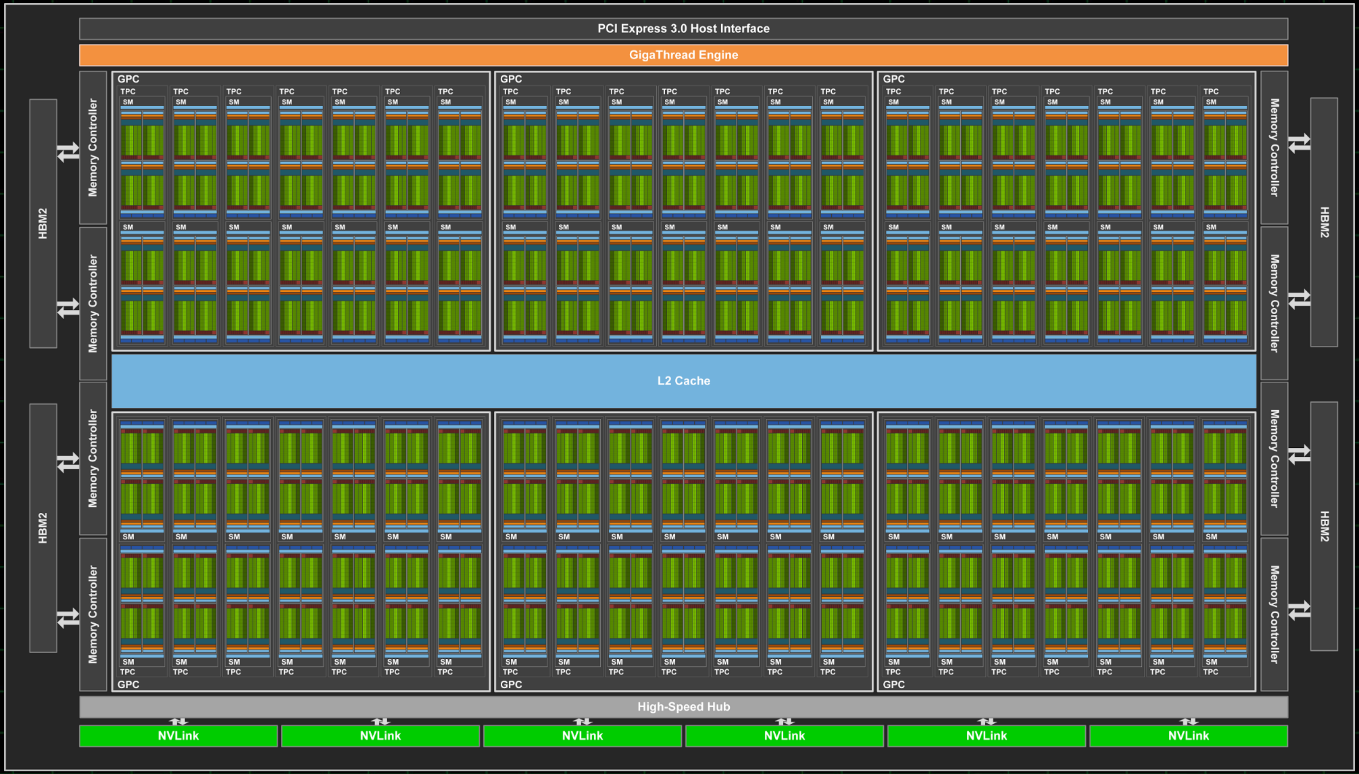


[illegible]
computation, 99% to [illegible]
to combat latency.

How CUDA works?

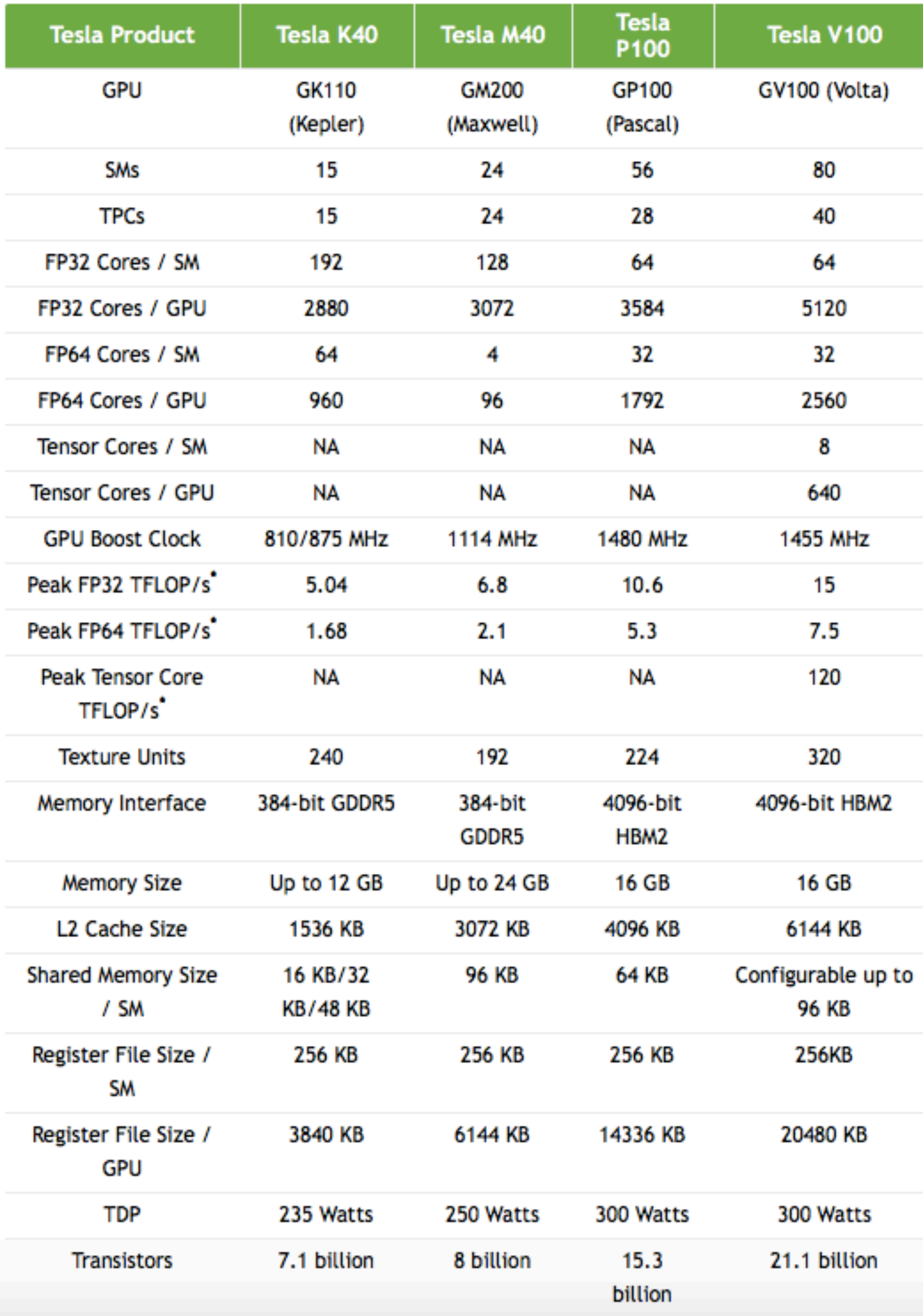
```
__global__ void vecAdd(int *d_a, int *d_b, int *d_c){
    int i= threadIdx.x + blockIdx.x * blockDim.x;
    d_c[i] = d_a[i] + d_b[i];
}


int main()
{
        vecAdd <<<K,M>>>(A, B, C);          //  K*M >= N
}
```

Overview



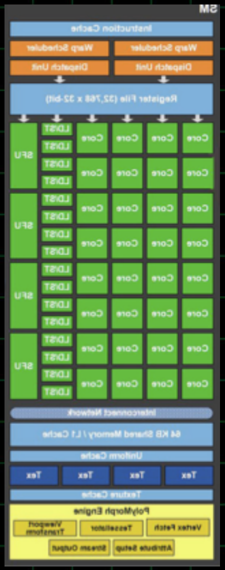
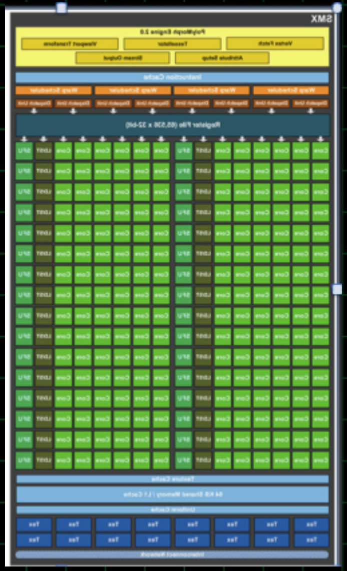
6 GPCs, 84 Volta SMs, 42 TPCs (each including two SMs), and eight 512-bit memory controllers (4096 bits total). Each SM has 64 FP32 Cores, 64 INT32 Cores, 32 FP64 Cores, and 8 new Tensor Cores. Each SM also includes four texture units. 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores, and 336 texture units

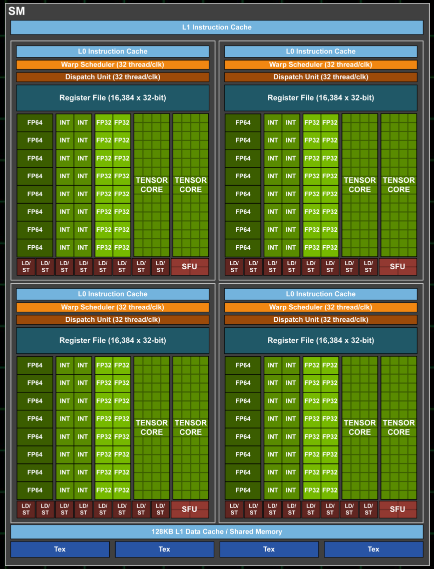
Overview



Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1455 MHz
Peak FP32 TFLOP/s*	5.04	6.8	10.6	15
Peak FP64 TFLOP/s*	1.68	2.1	5.3	7.5
Peak Tensor Core TFLOP/s*	NA	NA	NA	120
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion

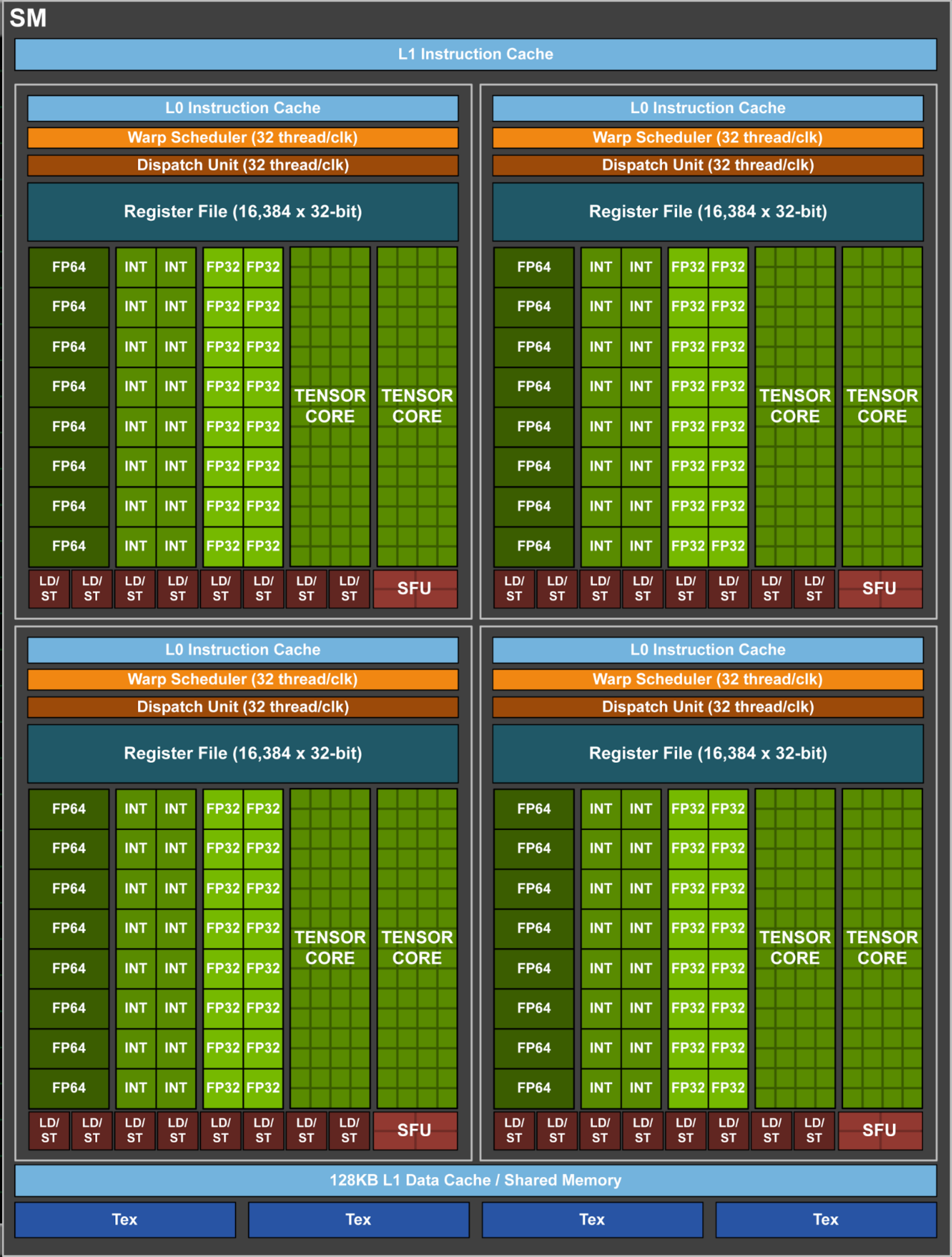
Volta SM


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

Compute capability

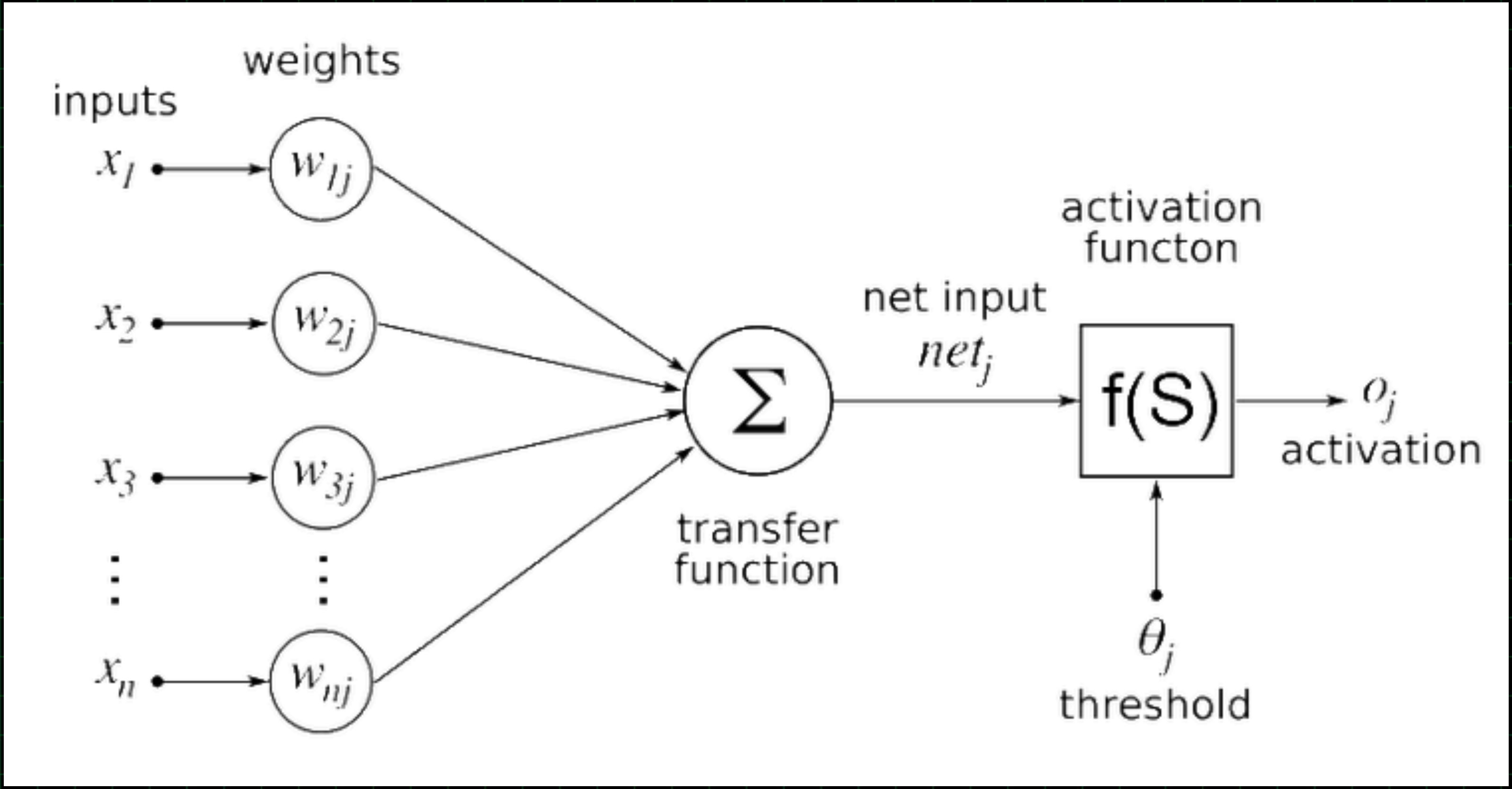
Volta SM



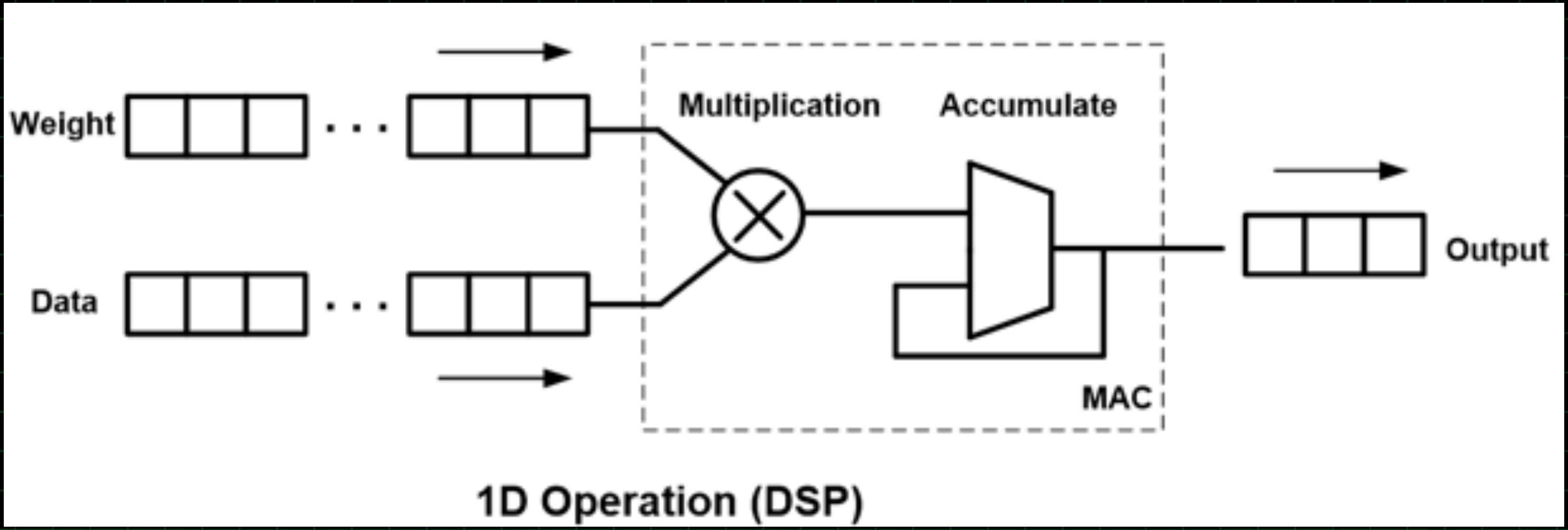
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Why GPUs became as powerfull (and indispensable) to Deep Learning as they are for Rendering?

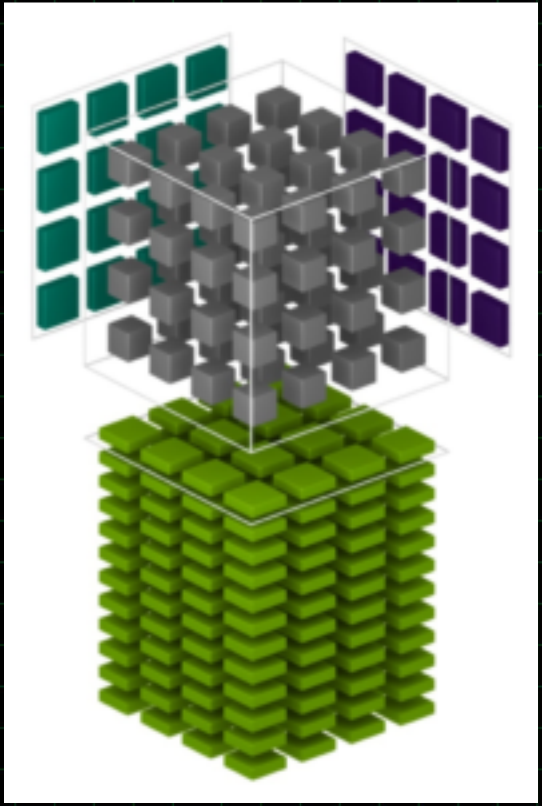
Why GPUs became as powerfull (and indispensable) to Deep Learning as they are for Rendering?



Why GPUs became as powerfull (and indispensable) to Deep Learning as they are for Rendering?



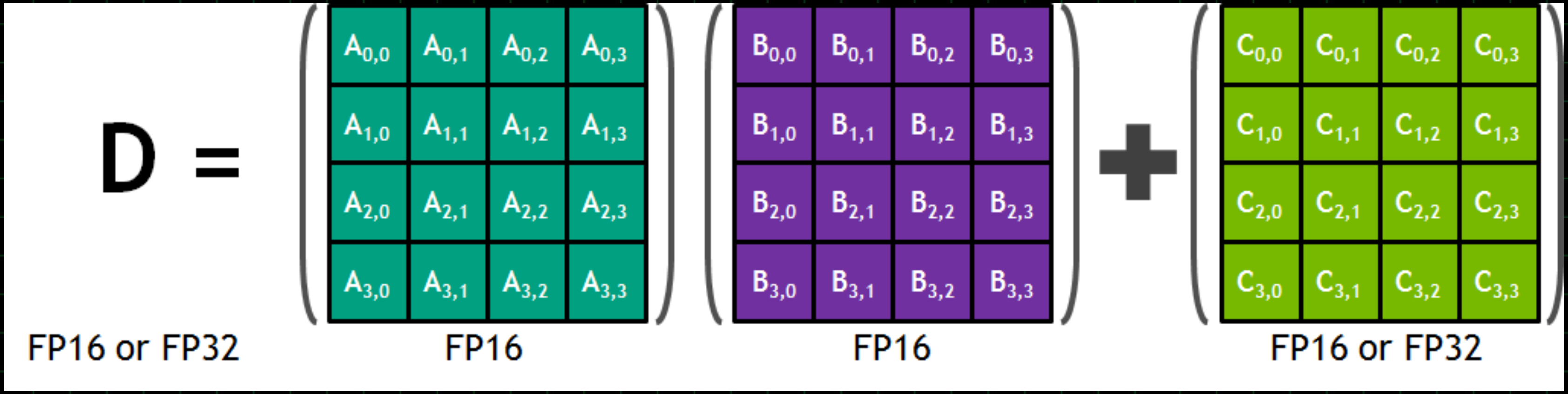
Tensor Cores



Tensor Cores

(FP16/FP32) D = (FP16) A x B + C (4 x 4 x 4)

64 FP operation per clock → full process in 1 clock cycle



$D =$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$+$
$C_{0,0}$ $C_{0,1}$ $C_{0,2}$ $C_{0,3}$
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$
$C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$
$C_{3,0}$ $C_{3,1}$ $C_{3,2}$ $C_{3,3}$
FP16 or FP32
FP16
FP16
FP16 or FP32

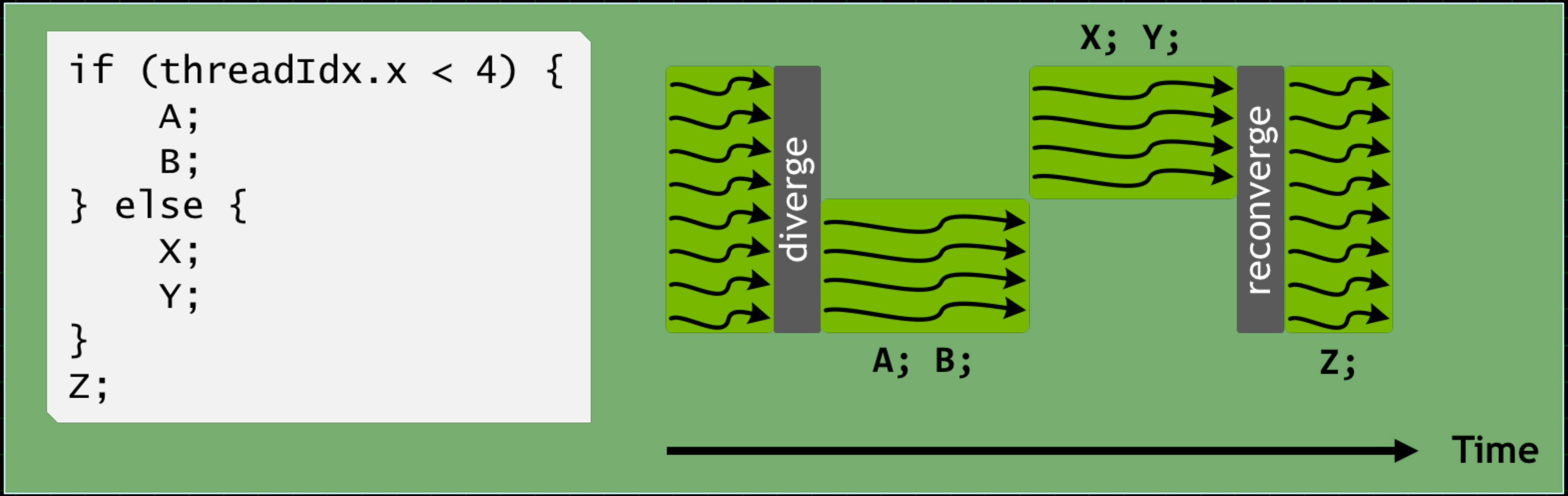
8 TC per SM → 1024 FP per clock per SM

Mixed Precision

“Deep learning have found that deep neural network architectures have a natural resilience to errors due to the backpropagation algorithm used in training them, and some developers have argued that 16-bit floating point (half precision, or FP16) is sufficient for training neural networks.”

New SIMT model

Until Pascal: 32 threads per warp in SIMT scheme

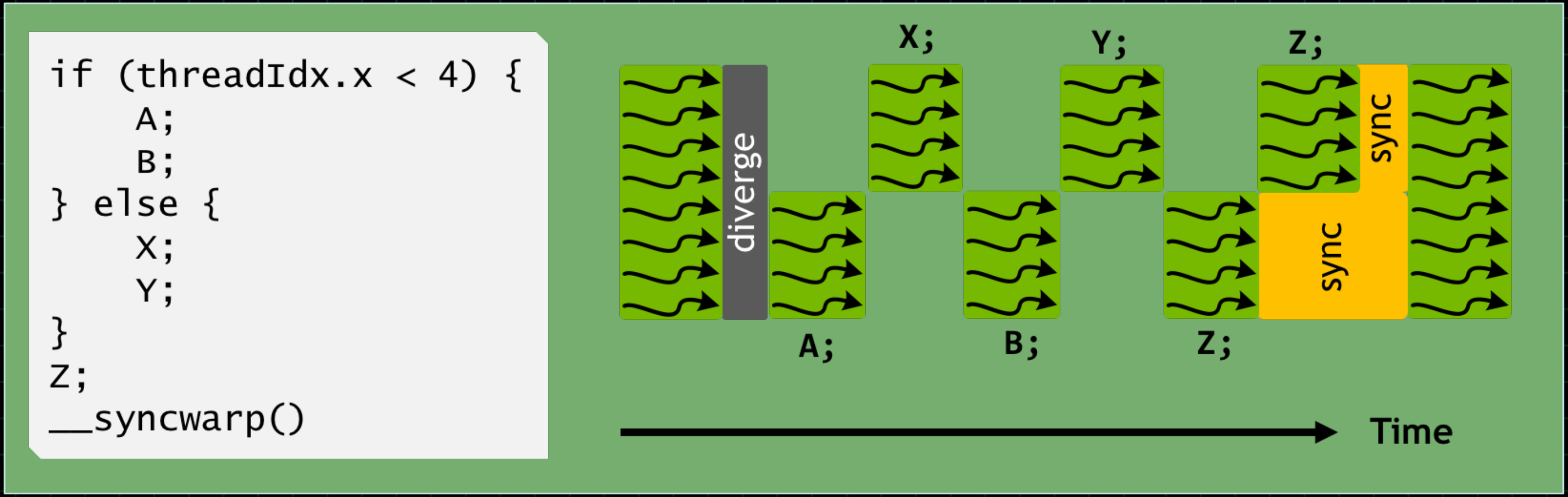


```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

There is no control in the thread level sync at the divergence, in the same warp

New SIMT model

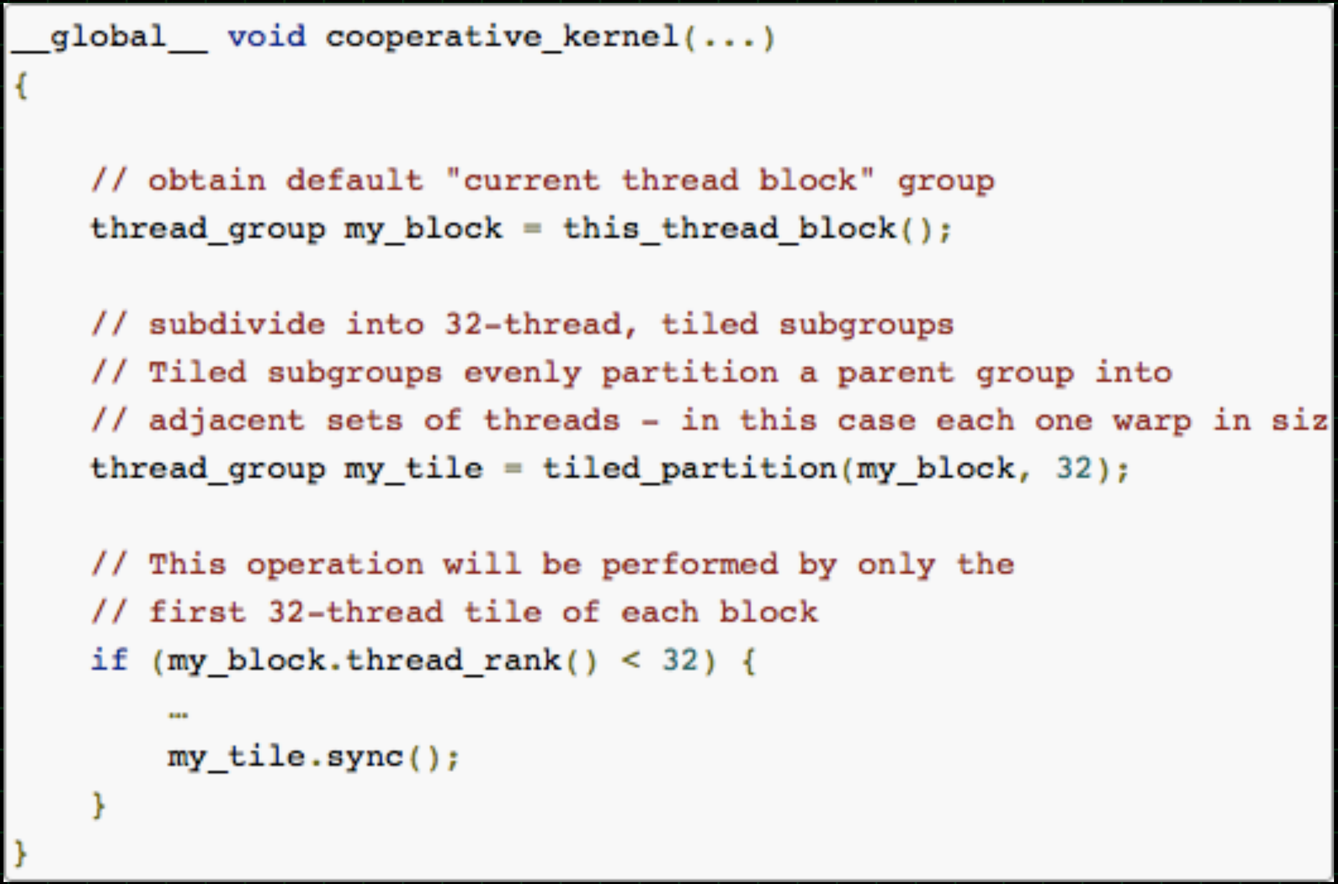
Volta allows to group threads at a warp level



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

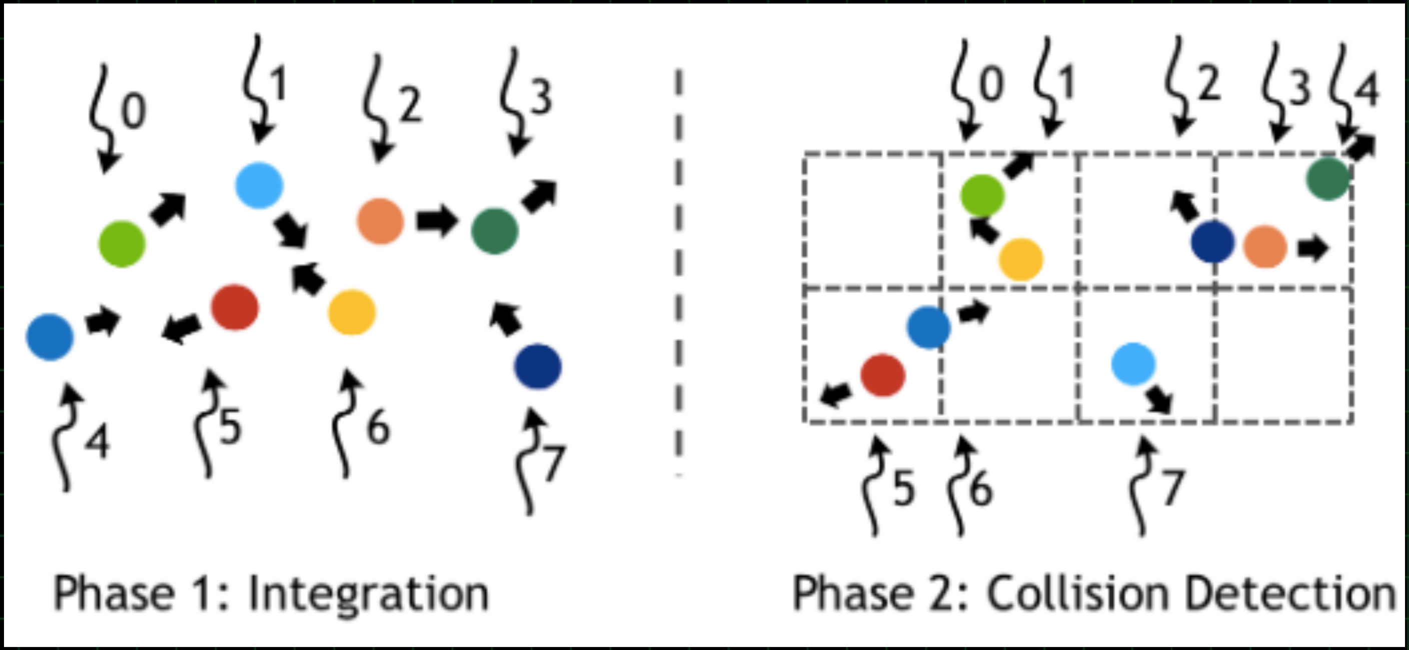
There is no control in the thread level sync at the divergence, in the same warp

Cooperative Groups

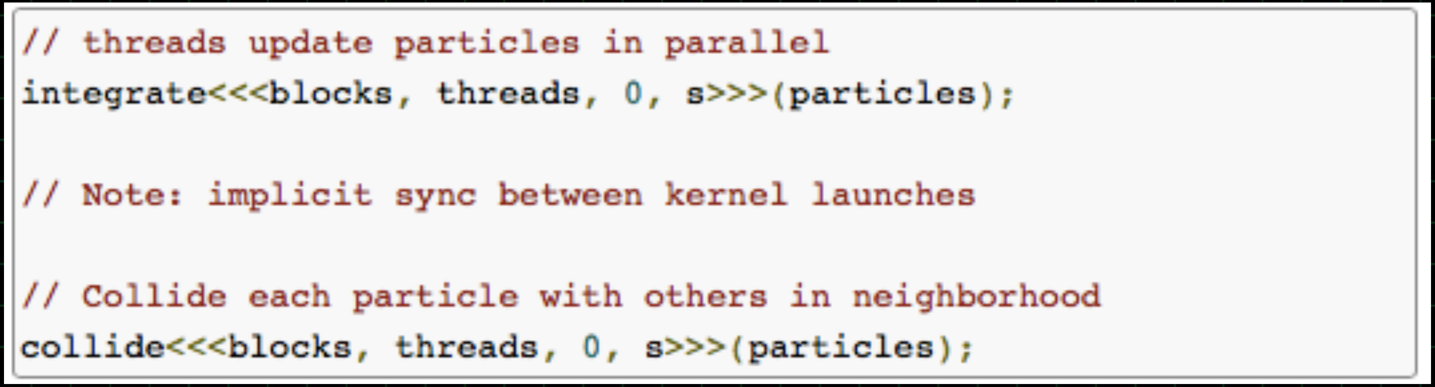


```
__global__ void cooperative_kernel(...)
{

    // obtain default "current thread block" group
    thread_group my_block = this_thread_block();

    // subdivide into 32-thread, tiled subgroups
    // Tiled subgroups evenly partition a parent group into
    // adjacent sets of threads - in this case each one warp in siz
    thread_group my_tile = tiled_partition(my_block, 32);

    // This operation will be performed by only the
    // first 32-thread tile of each block
    if (my_block.thread_rank() < 32) {
        …
        my_tile.sync();
    }
}
```

Cooperative Groups - Example

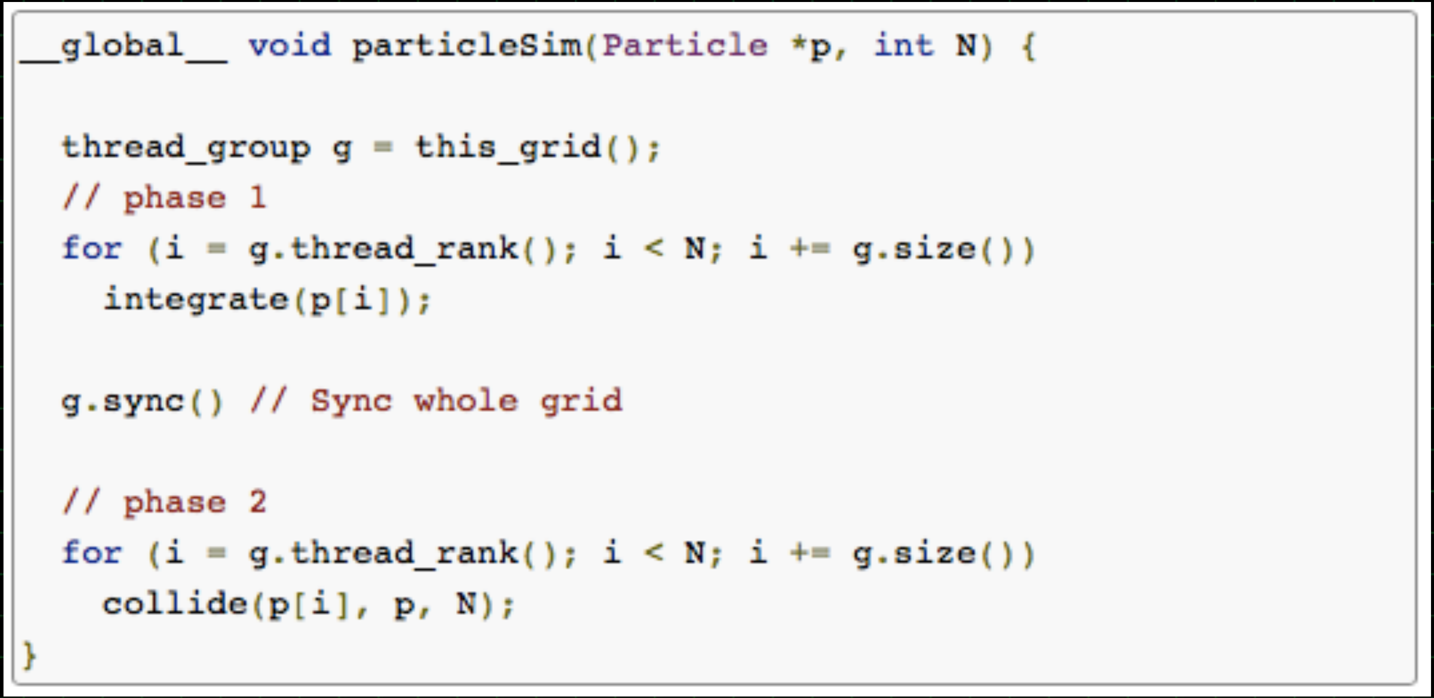


Cooperative Groups - Example



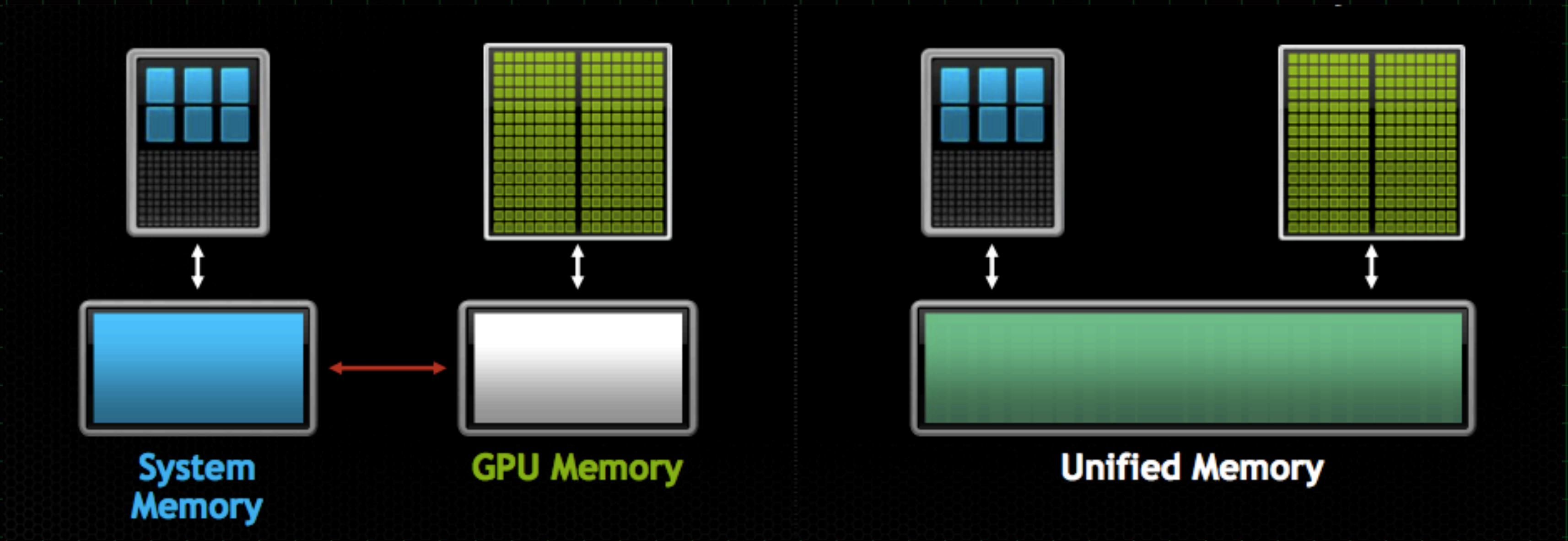
```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);

// Note: implicit sync between kernel launches

// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

Cooperative Groups - Example



```
__global__ void particleSim(Particle *p, int N) {

  thread_group g = this_grid();
  // phase 1
  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // Sync whole grid

  // phase 2
  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```

Closer to Unified Memory

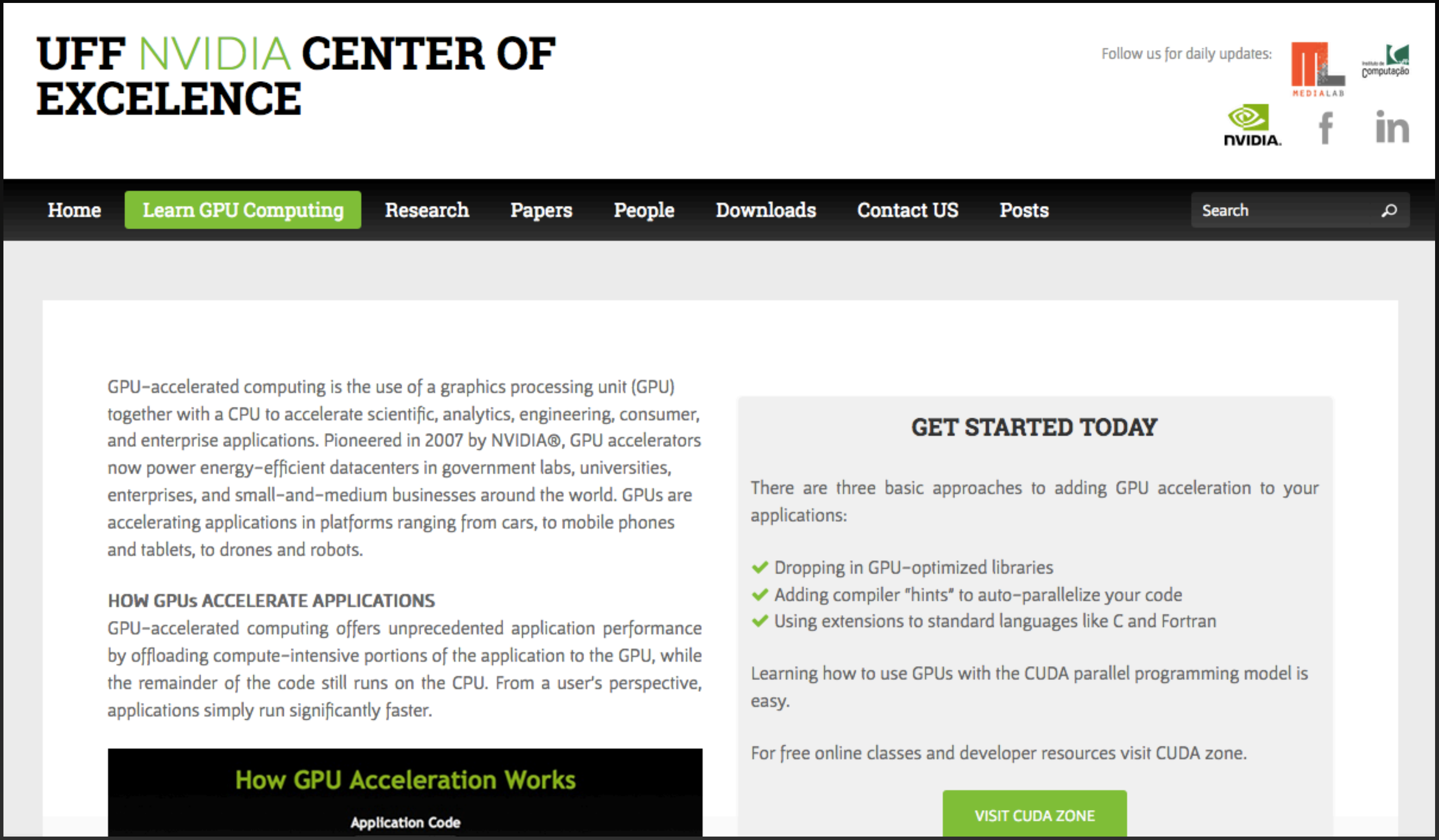


Faster Memory

900 GB/sec peak bandwidth

NVLink 2.0

GPU Educational Kit



UFF NVIDIA CENTER OF EXCELENCE

GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications. Pioneered in 2007 by NVIDIA®, GPU accelerators now power energy-efficient datacenters in government labs, universities, enterprises, and small-and-medium businesses around the world. GPUs are accelerating applications in platforms ranging from cars, to mobile phones and tablets, to drones and robots.

HOW GPUs ACCELERATE APPLICATIONS
GPU-accelerated computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run significantly faster.

How GPU Acceleration Works

Application Code

GET STARTED TODAY

There are three basic approaches to adding GPU acceleration to your applications:

- ✓ Dropping in GPU-optimized libraries
- ✓ Adding compiler "hints" to auto-parallelize your code
- ✓ Using extensions to standard languages like C and Fortran

Learning how to use GPUs with the CUDA parallel programming model is easy.

For free online classes and developer resources visit CUDA zone.

GPU Educational Kit

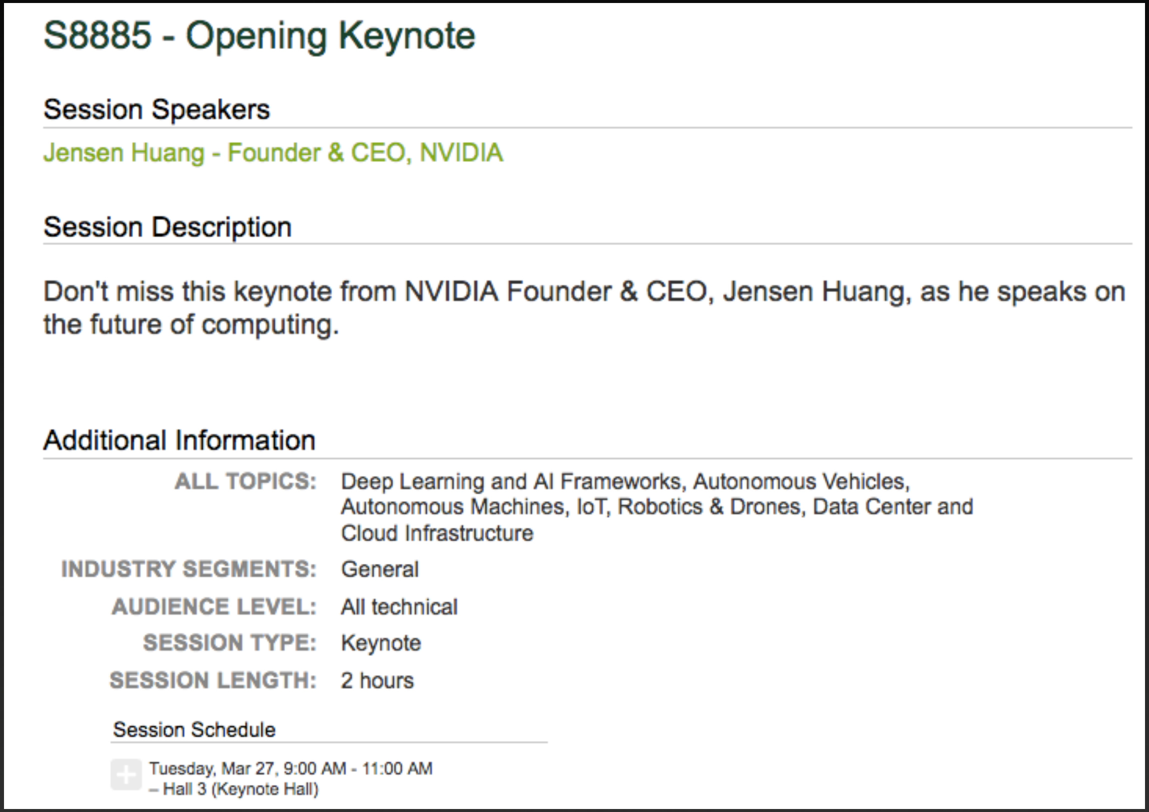
Curso completo de Programação em GPUs:
(legendado para Português)

http://www2.ic.uff.br/~gpu/kit-de-ensino-gpgpu/

Curso de Deep Learning em GPUs:
(Português)

http://www2.ic.uff.br/~gpu/learn-gpu-computing/deep-learning/

Save the date



S8885 - Opening Keynote

Session Speakers

Jensen Huang - Founder & CEO, NVIDIA

Session Description

Don't miss this keynote from NVIDIA Founder & CEO, Jensen Huang, as he speaks on the future of computing.

Additional Information

ALL TOPICS:	Deep Learning and AI Frameworks, Autonomous Vehicles, Autonomous Machines, IoT, Robotics & Drones, Data Center and Cloud Infrastructure
INDUSTRY SEGMENTS:	General
AUDIENCE LEVEL:	All technical
SESSION TYPE:	Keynote
SESSION LENGTH:	2 hours

Session Schedule

Tuesday, Mar 27, 9:00 AM - 11:00 AM
– Hall 3 (Keynote Hall)

27 de março
9AM San Jose, CA
2PM Brasil/Argentina