Hands-on CUDA

Prof. Esteban Walter Gonzalez Clua, Dr.

Cuda Fellow

Computer Science Department

Universidade Federal Fluminense – Brazil

Universidade Federal Fluminense
Rio de Janeiro - Brasil

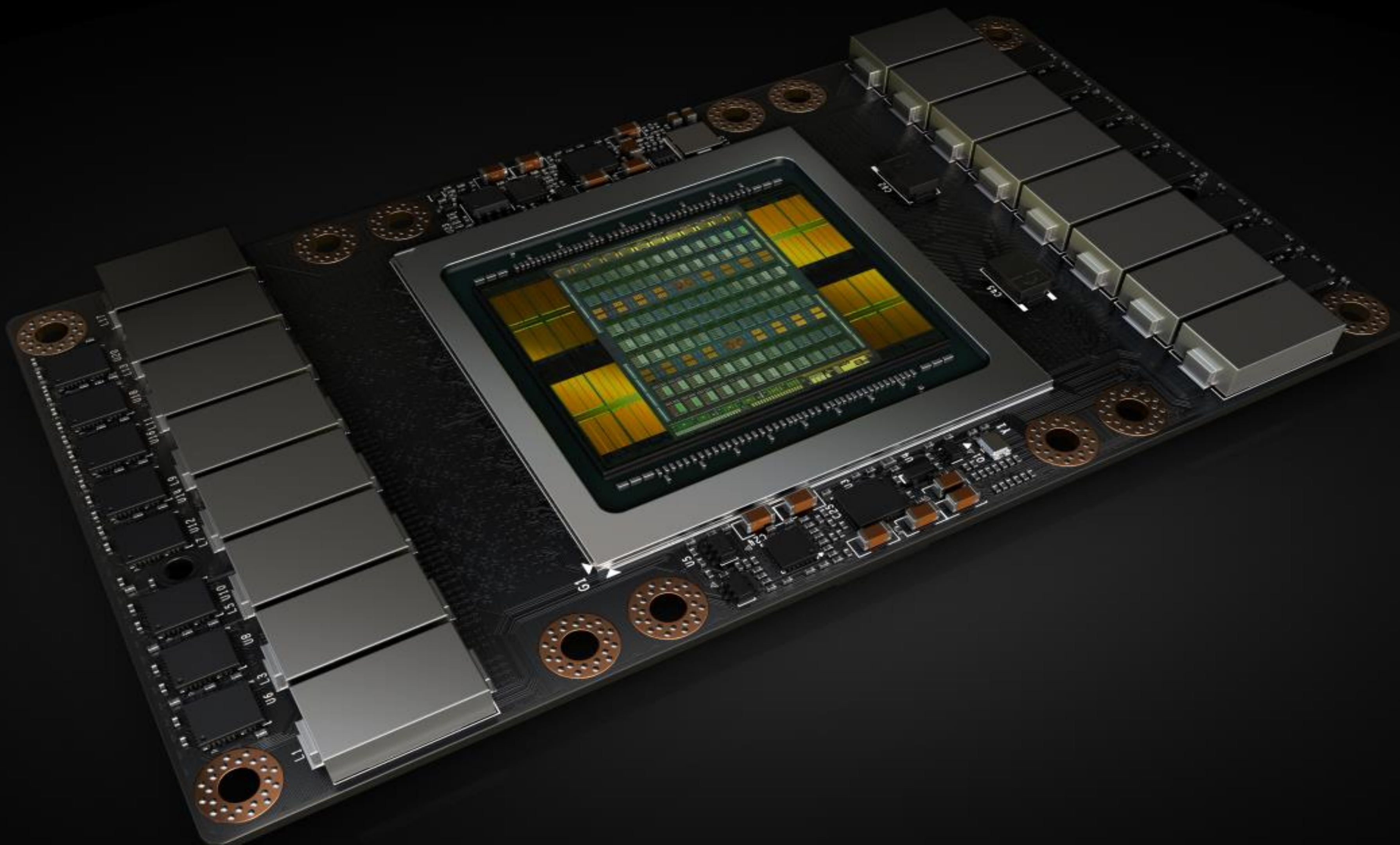
Porque CUDA?

...futuro das GPUs

```
For each n (parallel, 0, n, [&](int i )
     dado[i] = tarefa (fonte[i]);
```

Volta

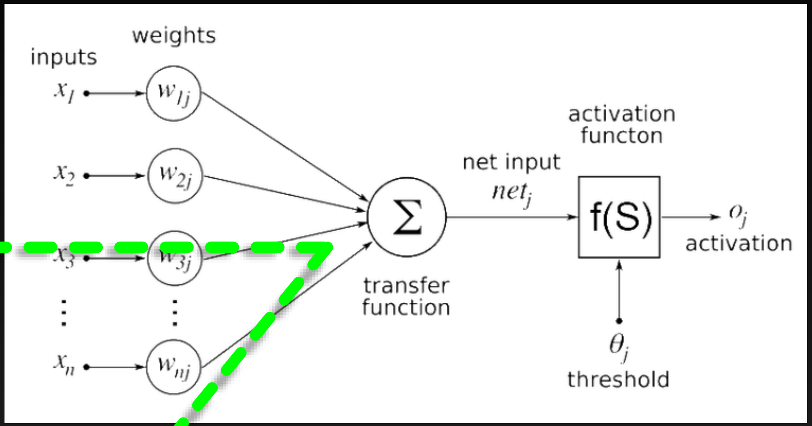
The most advanced accelerator ever built



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.*

Big Bang of IA

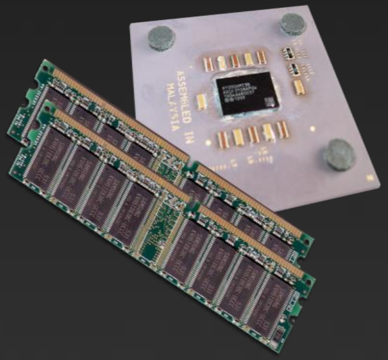


Paradigmas de GPU Programming

3 coisas que voce deve saber de cor!

#1 – Estamos falando de computação heterogênea

- *Host* CPU e sua memória (host memory)
- *Device* GPU e sua memória (Global memory)



Host

Device

Heterogeneous Computing

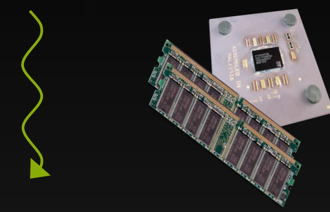
```
#include <iostream>
#include <algorithm>

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;
}
```

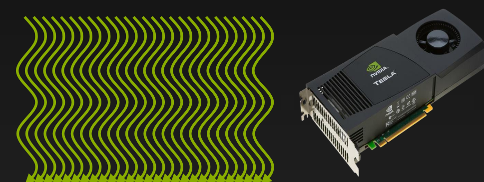
parallel fn

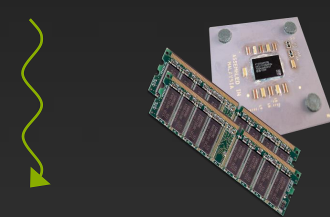
serial code

parallel code

serial code



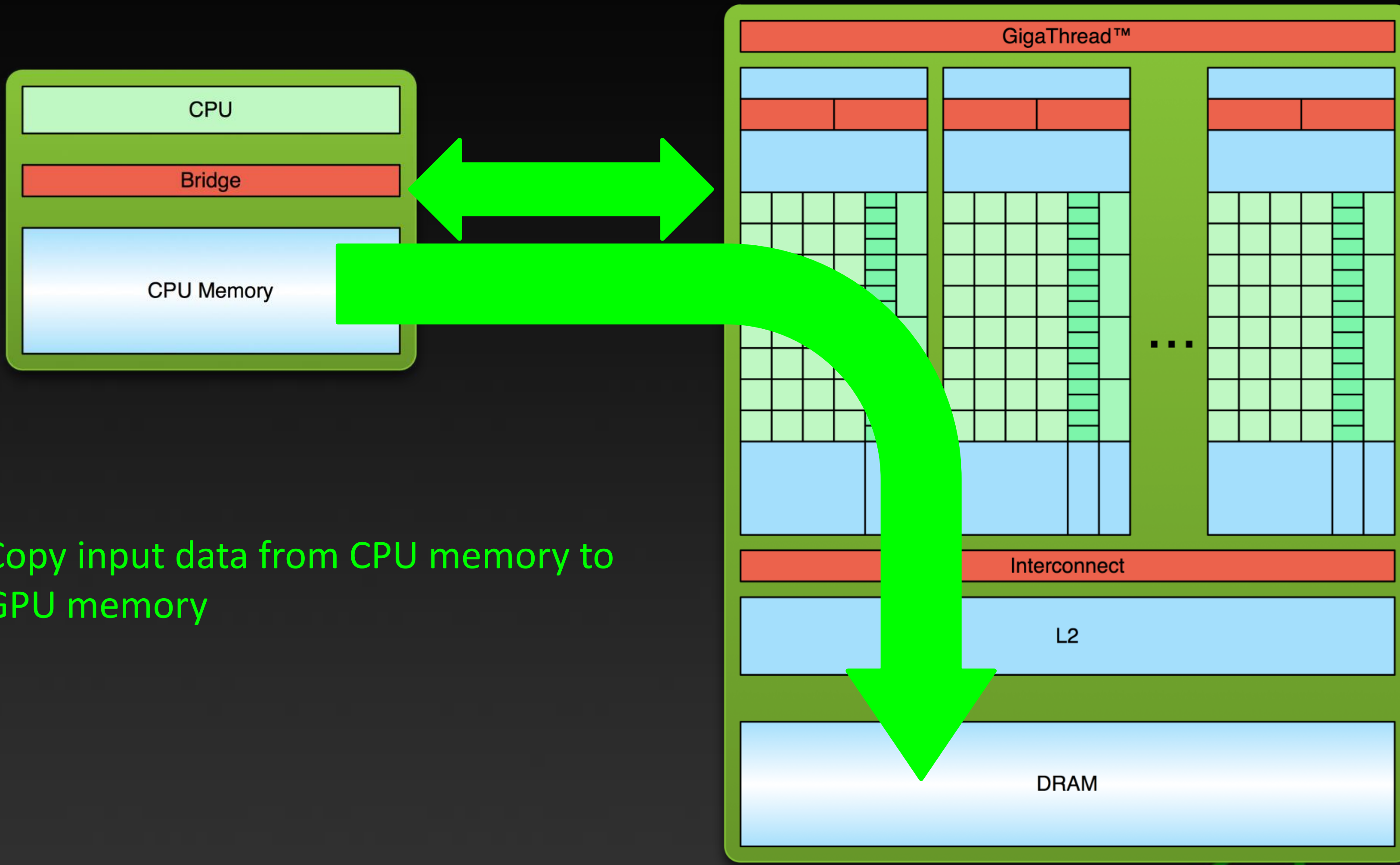




This slide is credited to Mark Harris (nvidia)

#2 – Tráfego de memória importa muito!...

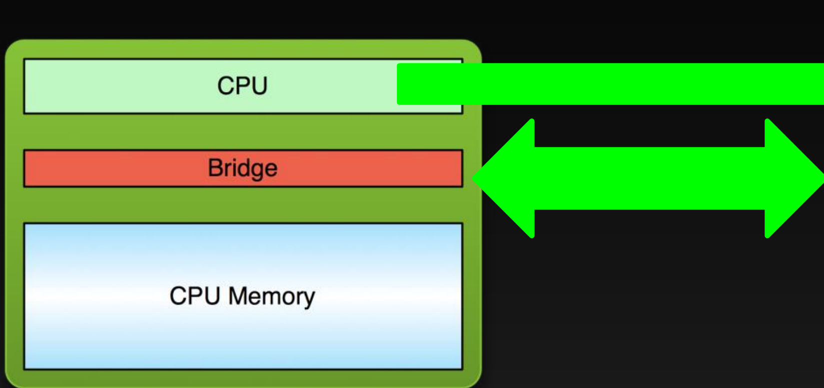
GPU Computing Flow

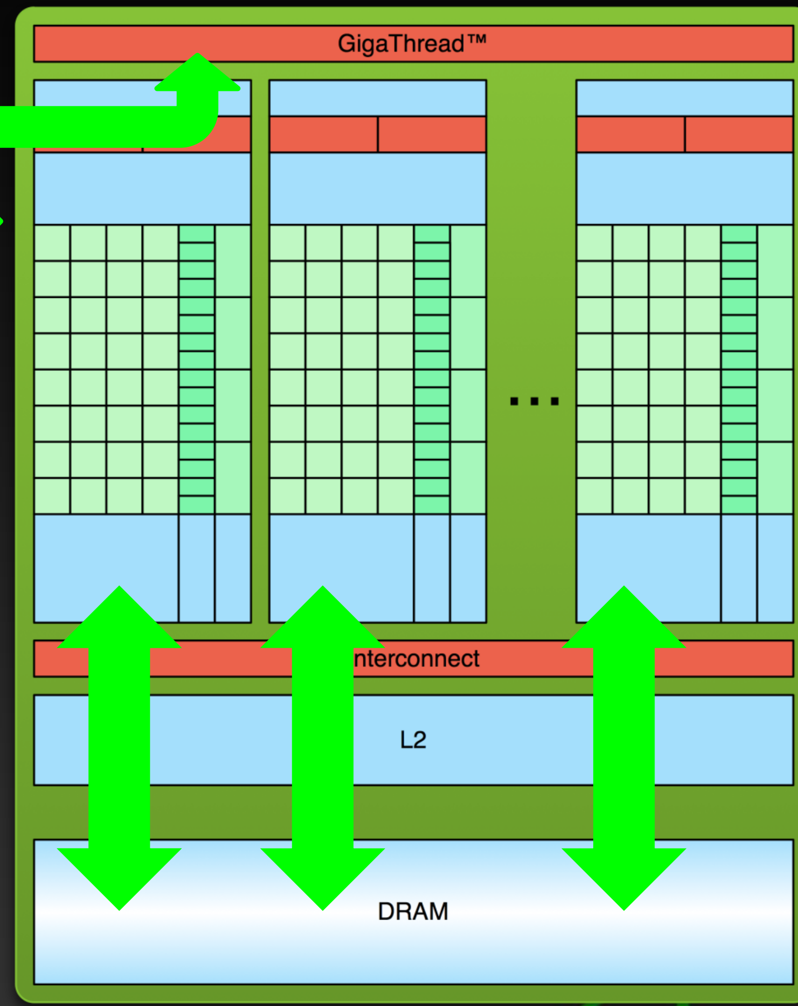


1. Copy input data from CPU memory to GPU memory

This slide is credited to Mark Harris (nvidia)

GPU Computing Flow

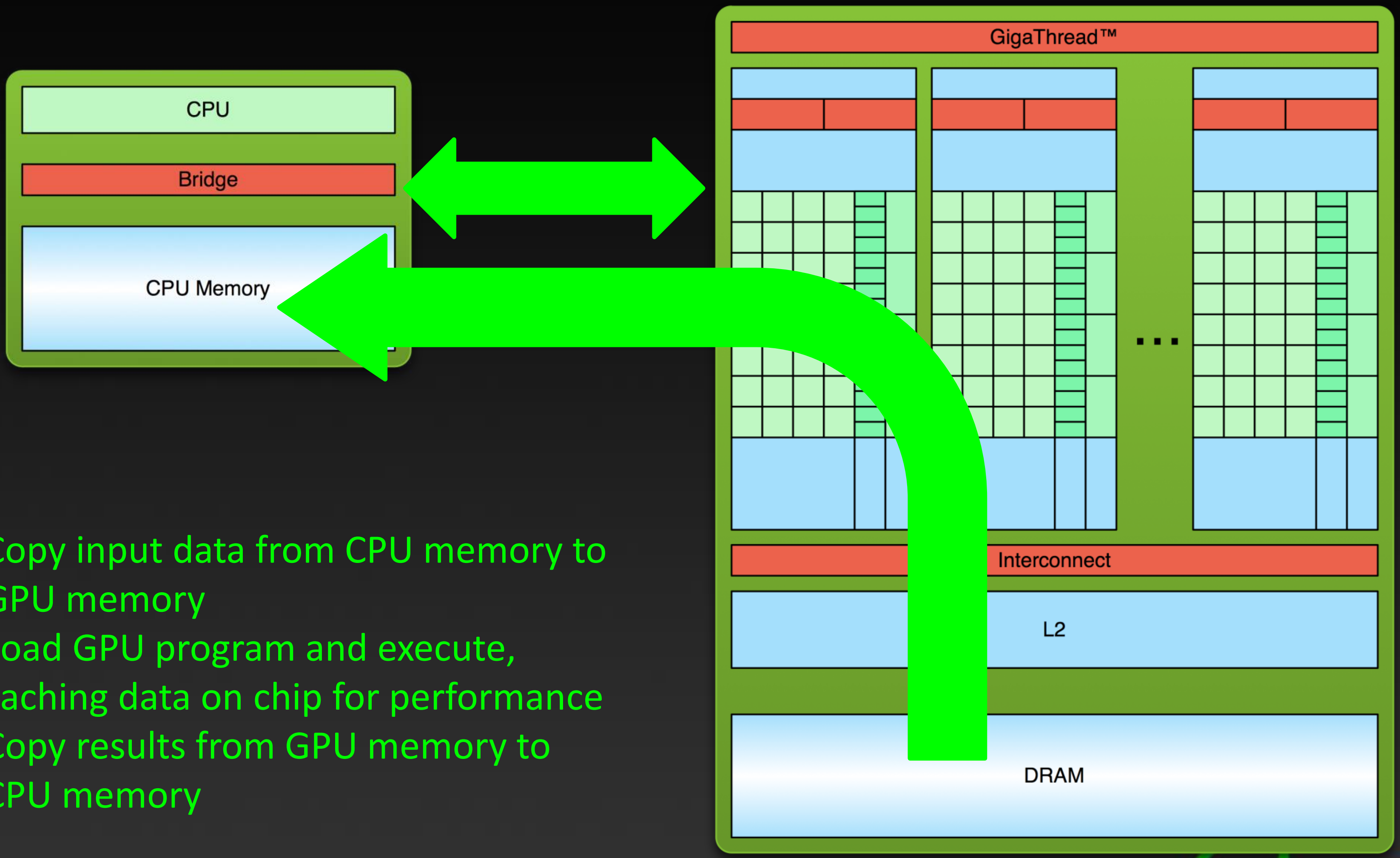




1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

This slide is credited to Mark Harris (nvidia)

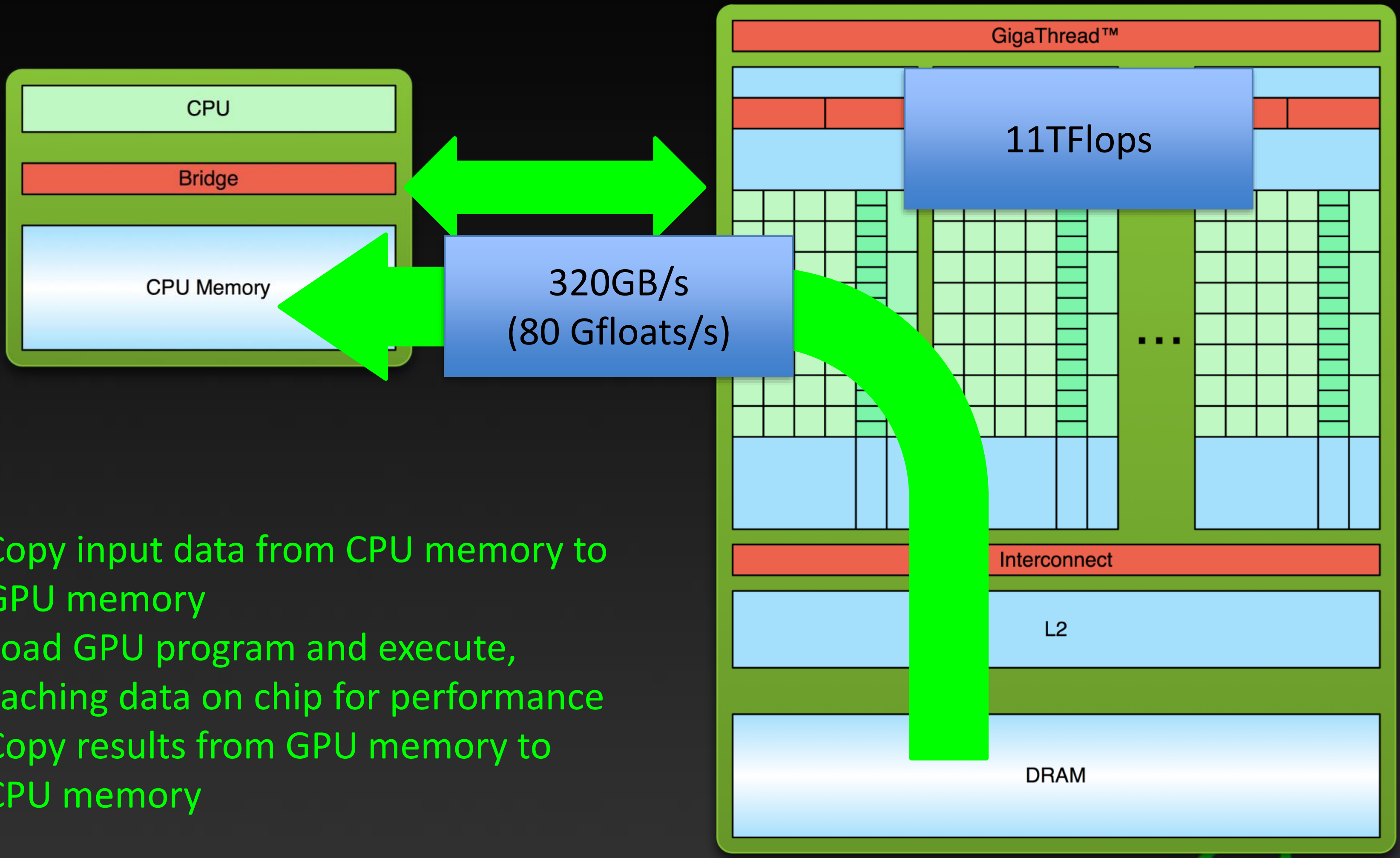
GPU Computing Flow



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

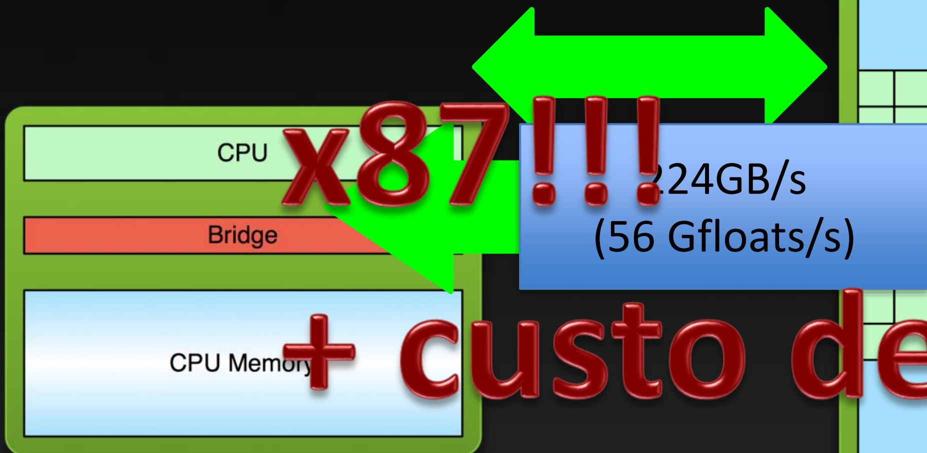
This slide is credited to Mark Harris (nvidia)

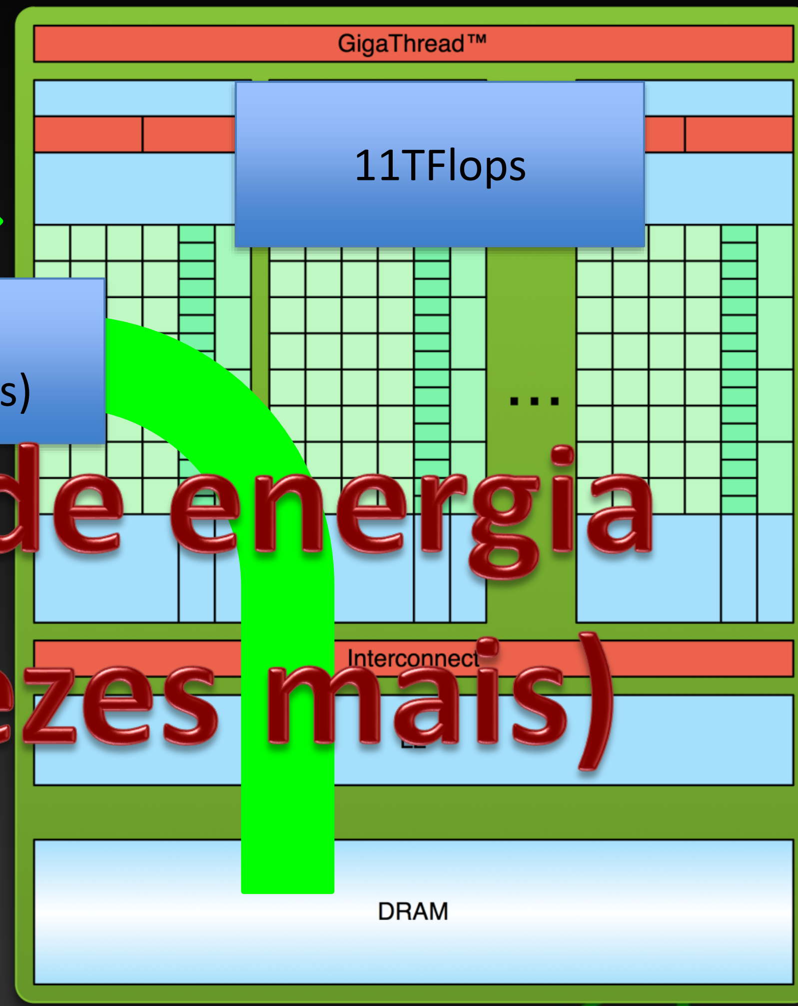
GPU Computing Flow



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





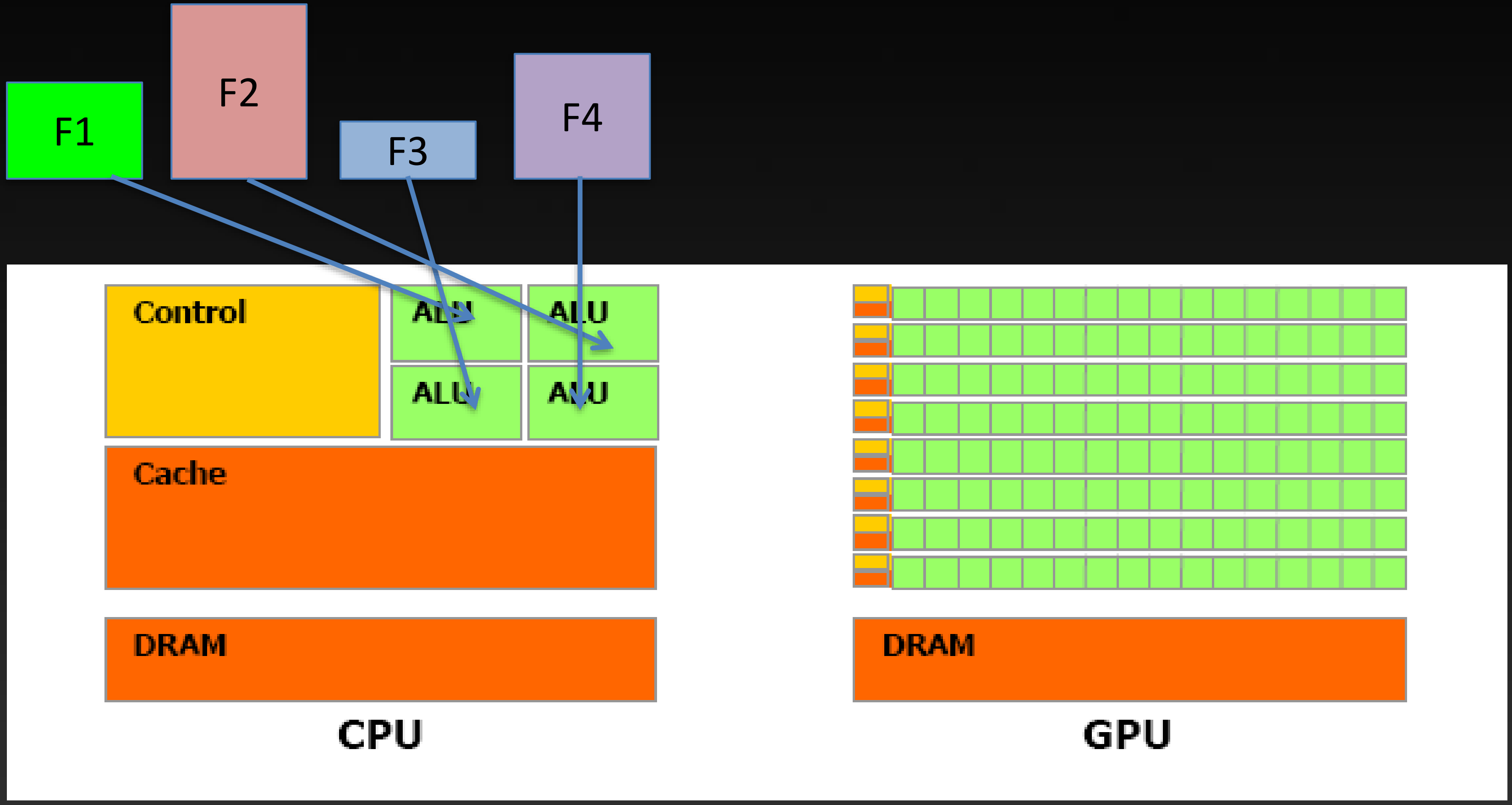
x87!!!

+ custo de energia (~100 vezes mais)

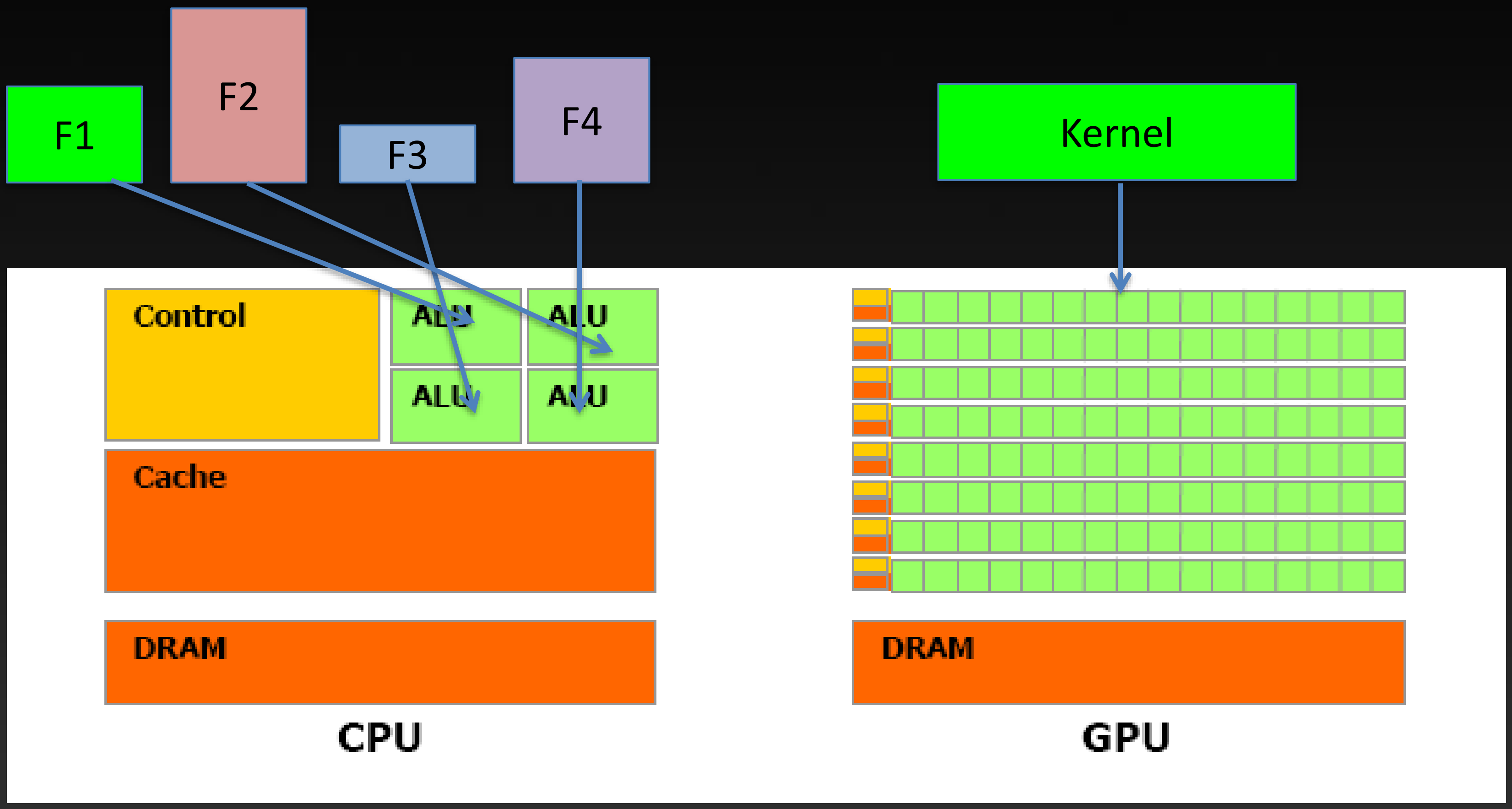
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

#3 – 1 kernels, muitos threads...

Threads em GPU x CPU



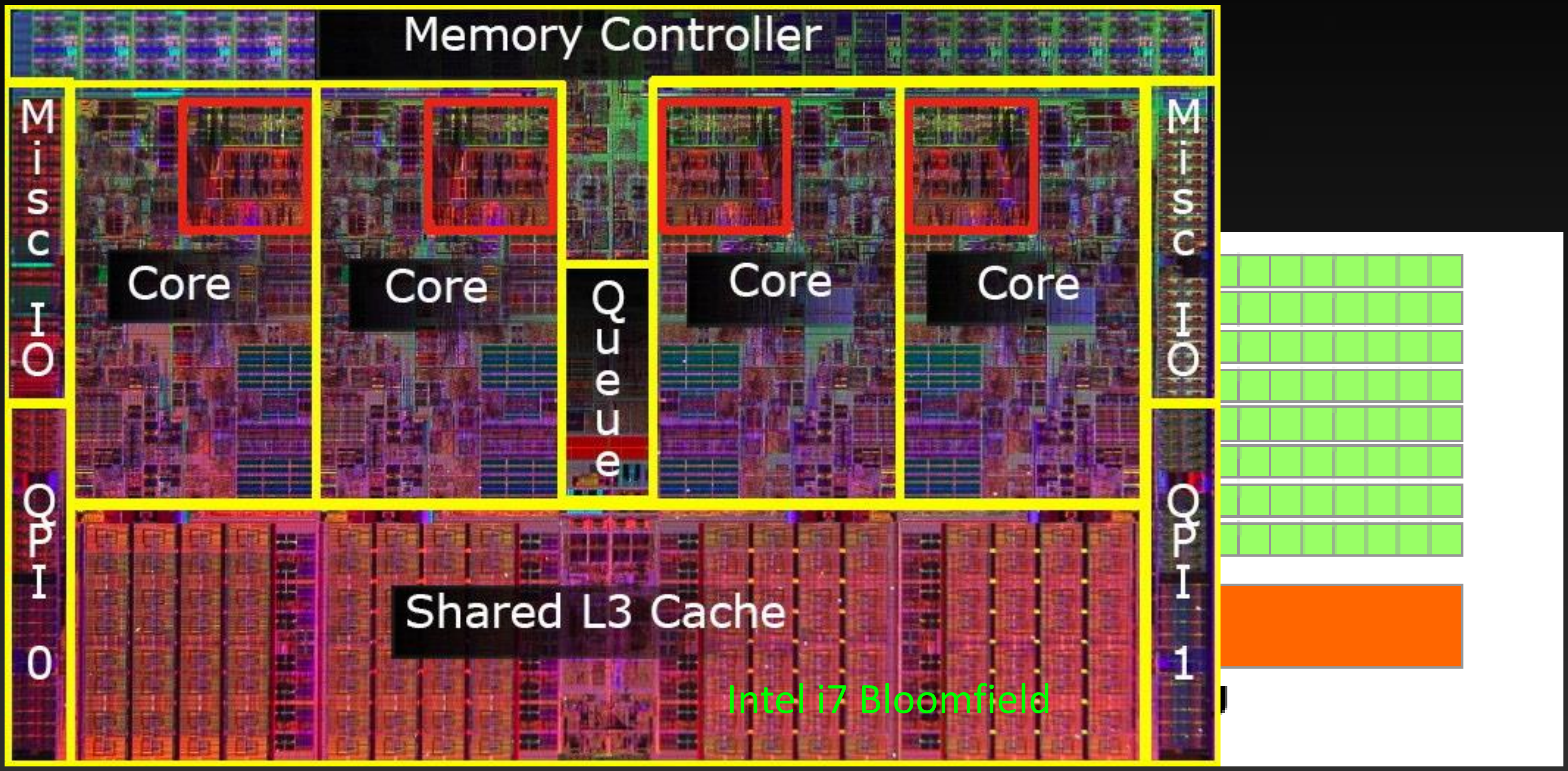
Threads em GPU x CPU



Modelo SIMT

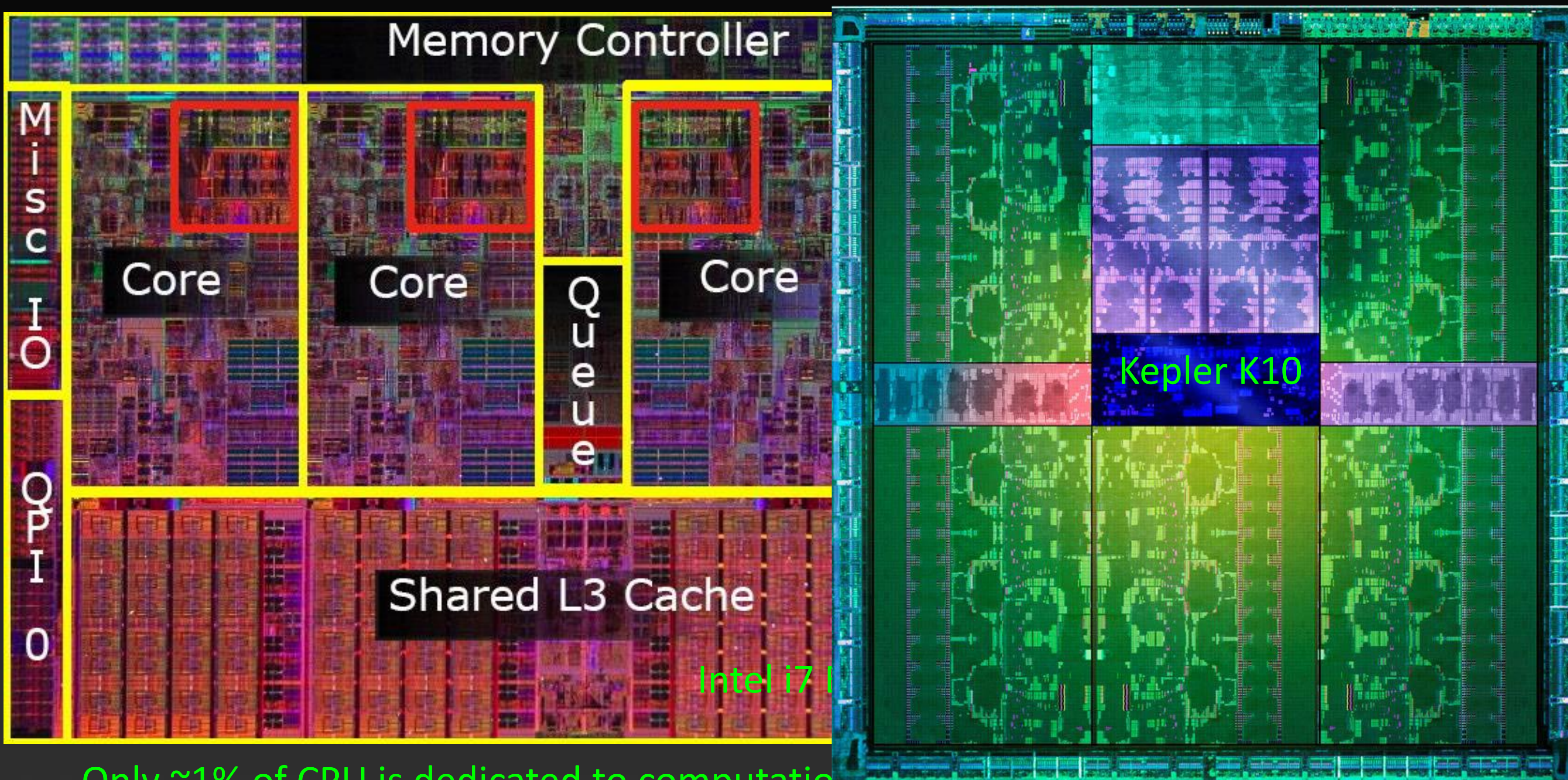

SIMT
means
Single Instruction
Multiple Thread
by allacronyms.com
ALLACRONYMS

GPU x CPU



Only ~1% of CPU is dedicated to computation,
99% to moving/storing data to combat latency.

GPU x CPU



Only ~1% of CPU is dedicated to computation,
99% to moving/storing data to combat latency.

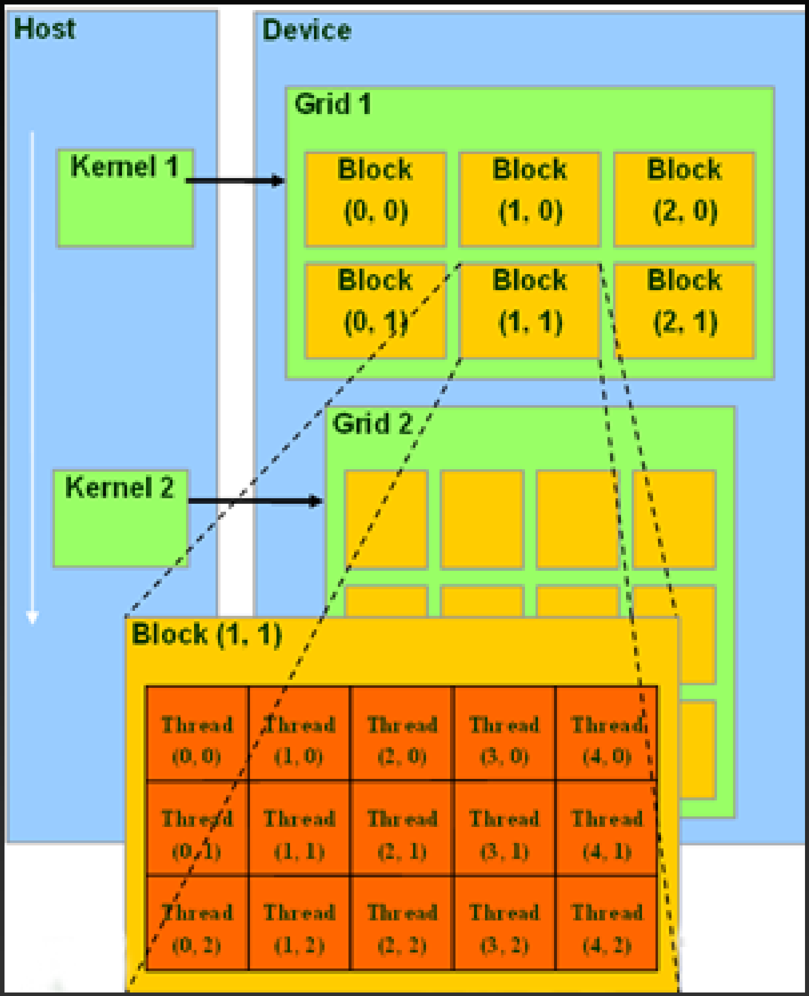
GRID


PCI Express 3.0 Host Interface
GigaThread Engine
SMX
Memory Controller
L2 Cache

Principais conceitos de CUDA

Device: a GPU
Host: a CPU
Kernel – Programa que vai para a GPU
Thread –Instancias do kernel
Global Memory: memória principal da GPU
Main memory: memória principal da CPU
CUDA, PTX and Cubin

Threads, Blocks e Grids



Um kernel é executado numa GRID

Cada bloco é composto por threads (1024)

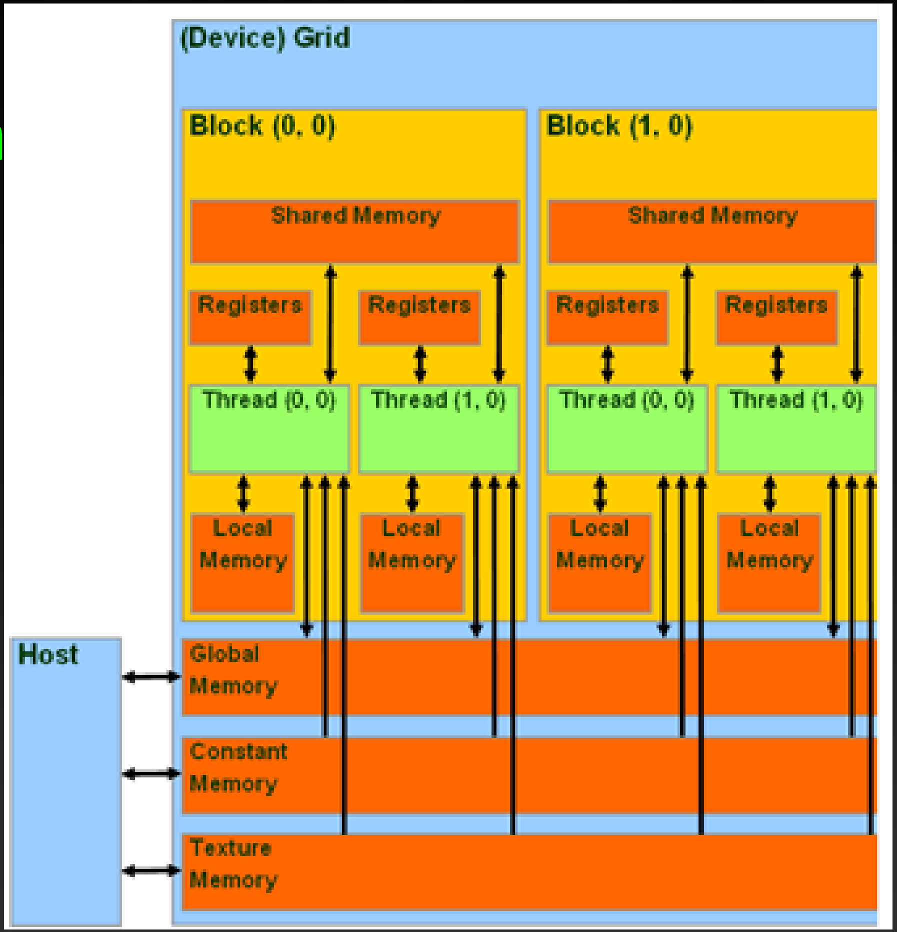
Todas as threads de um bloco podem usar a mesma shared memory

Threads de blocos diferentes não podem compartilhar a mesma shared memory, mas podem compartilhar dados pela memória global

Kernel, Threads e Warps

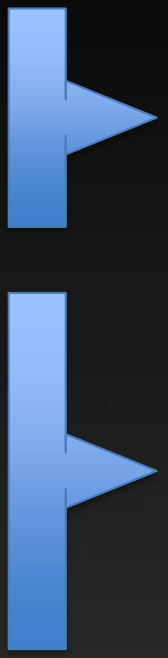
Memórias...

- Hierarquia de memória
- Local
- Cache L1 and L2
- shared
- Constant
- Texture
- Global



Hello World

```
__global__ void mykernel(void){
}

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



GPU

CPU

Hello World

```
__global__ void add(int *a, int *b, int *c)
{
      *c = *a + *b;
}
```

Alimentando a GPU com dados...

- **`Malloc()      ~  cudaMalloc()`**
- **`Free()        ~  cudaFree()`**
- **`cudaMemcpy() ~  memcpy()`**

Alimentando a GPU com dados...

```
int main(void) {
        int a, b, c;                  // CPU
        int *d_a, *d_b, *d_c;         // GPU
        int size = sizeof(int);

        // Allocate space for device
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 10;
        b = 20;
```

Alimentando a GPU com dados...

```
// CPU -> GPU
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// kernel execution: 1 thread
    add<<<1,1>>>(d_a, d_b, d_c);

// GPU -> CPU
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Clean memory
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

Memoria unificada

cudaMallocManaged() → igual a cudaMalloc(), porém permite unificar as duas memórias de forma conceitual.

```
    cudaMallocManaged ((void **)&a, size);
    cudaMallocManaged ((void **)&b, size);
    cudaMallocManaged ((void **)&c, size);

// kernel execution: 1 thread
    add<<<1,1>>>(a, b, c);

// Synchrhonize
    cudaDeviceSynchronize();

// Clean memory
    cudaFree(a); cudaFree(b); cudaFree(c);
```

Memoria unificada
Global Variable

__managed__

```
__device__ __managed__ int a[1000];
__device__ __managed__ int b[1000];
__device__ __managed__ int c[1000];

// kernel execution: 1 thread
    add<<<10,100>>>();

// Synchronize
    cudaDeviceSynchronize();
```

Para compilar os programas

Compiling a GPU program

Name file as .cu

Nvcc name.cu

./a.out

Voilá!...

Errors types

https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/group__CUDART__TYPES_g3f51e3575c2178246db0a94a430e0038.html

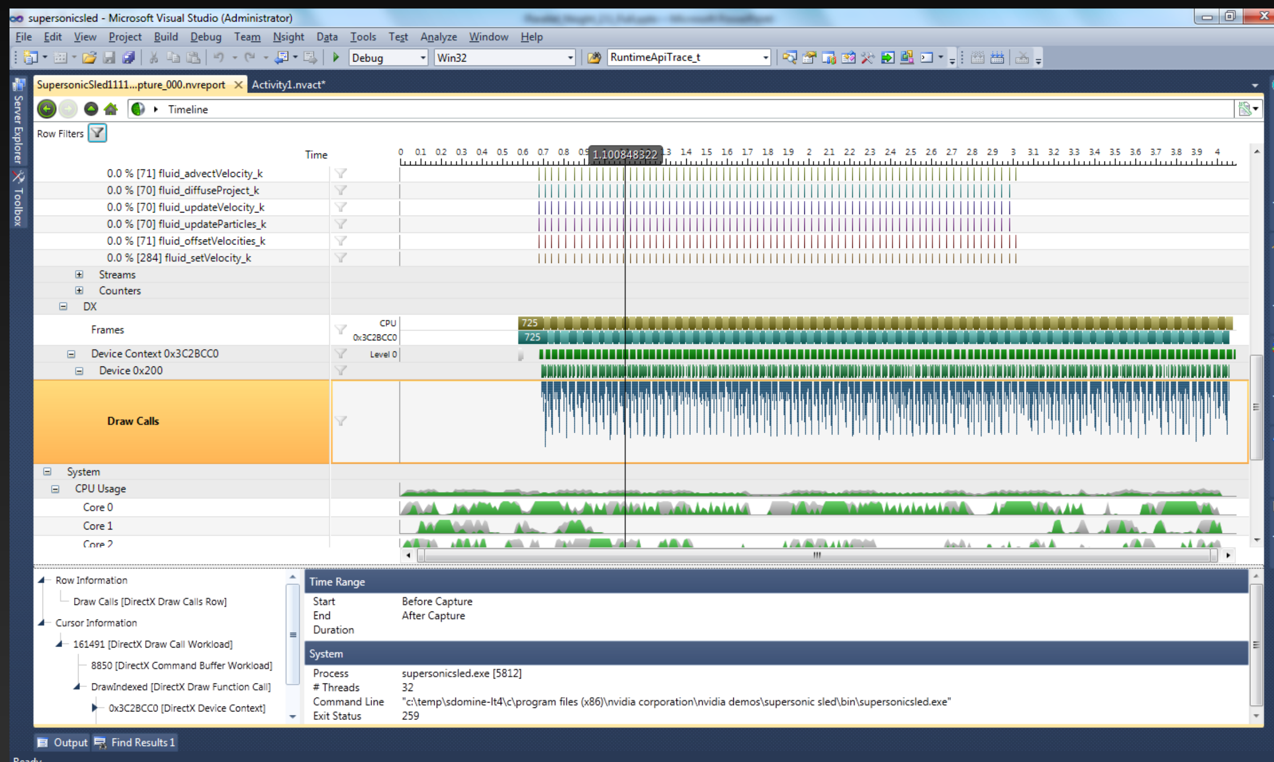
CUDA error types

Enumerator:

cudaSuccess	The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see **cudaEventQuery()** and **cudaStreamQuery()**).
cudaErrorMissingConfiguration	The device function being invoked (usually via **cudaLaunch()**) was not previously configured via the **cudaConfigureCall()** function.
cudaErrorMemoryAllocation	The API call failed because it was unable to allocate enough memory to perform the requested operation.
cudaErrorInitializationError	The API call failed because the CUDA driver and runtime could not be initialized.
cudaErrorLaunchFailure	An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until **cudaThreadExit()** is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.
cudaErrorPriorLaunchFailure	This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches. **Deprecated:** This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.
cudaErrorLaunchTimeout	This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property **kernelExecTimeoutEnabled** for more information. The device cannot be used until **cudaThreadExit()** is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

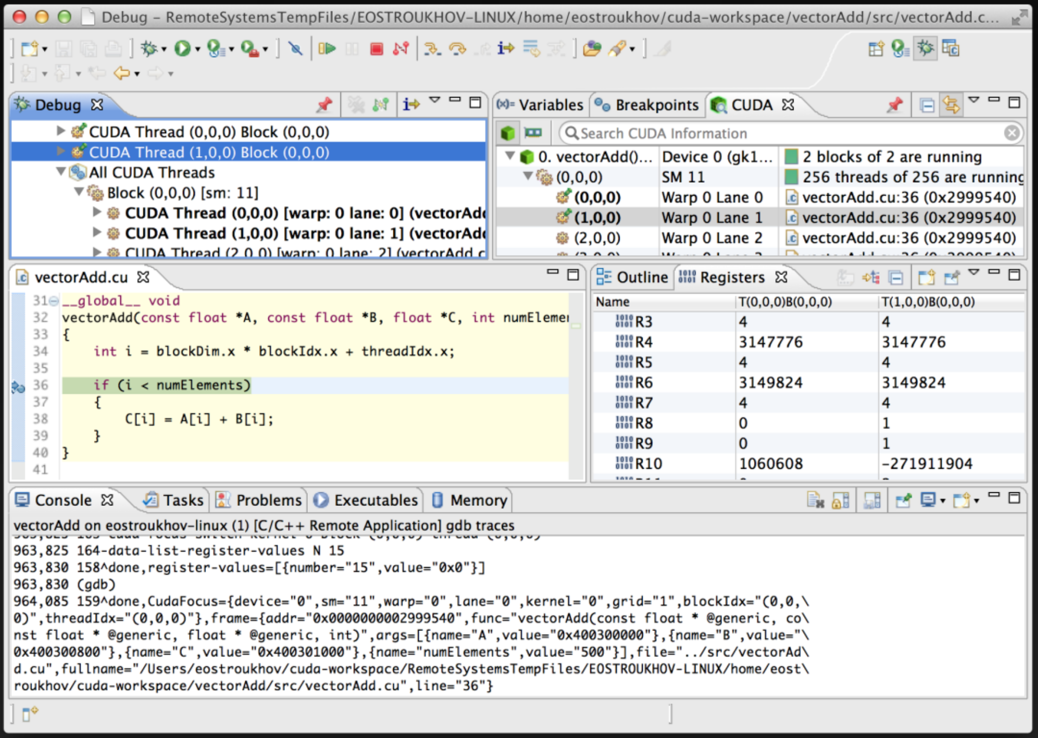
Debuggers

NSight



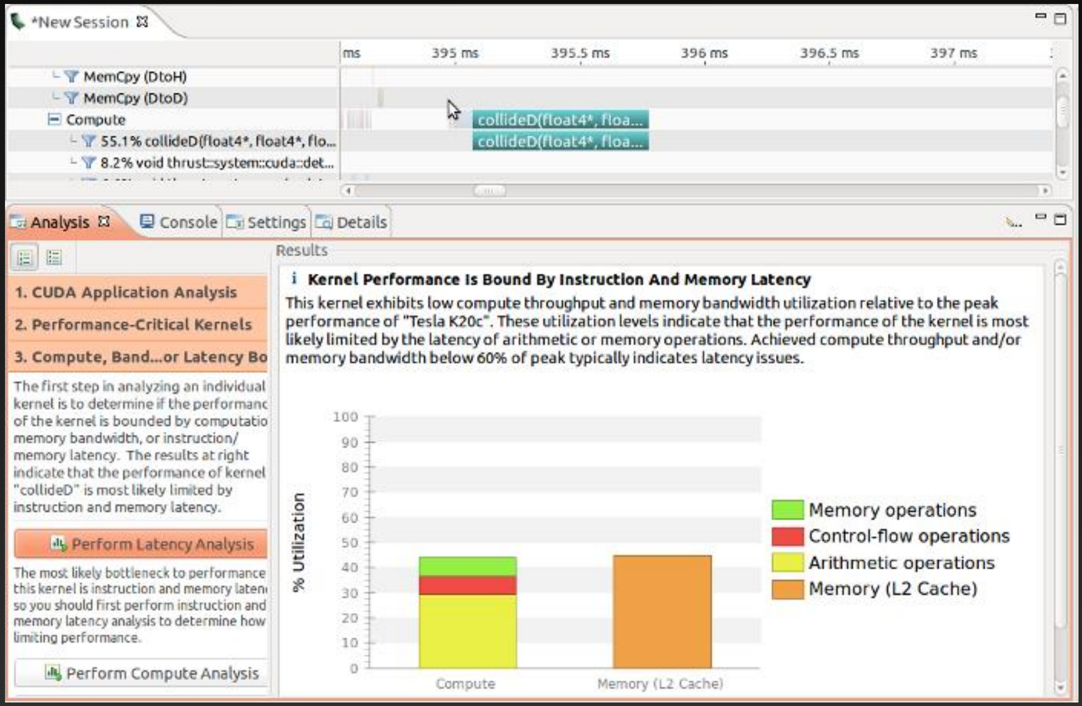
Debuggers

CUDA GDB



Debuggers

CUDA Memcheck



Profiler tools

NSIGHT

NVPP

NVPROF

NVIDIA NVProf

Nvprof ./a.out

Make some tests... Changing vector size...

Finalmente... O paralelismo

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

int main()
{
    ...
vecAdd<<<1, N>>>(d_a, d_b, d_c);
}
```

Pequeno concerto..

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

```
int main()
{
    ...
vecAdd<<<1, N>>>(d_a, d_b, d_c);    // blockDim.x = N
}
```

Explorando o paralelismo: Threads

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

At the same time...

```
c[0]  = a[0] + b[0];
c[1]  = a[1] + b[1];
c[2]  = a[2] + b[2];
...
C[N-1]  = a[N-1] + b[N-1];
```

Há um limite de threads... Por bloco...

Technical specifications	Compute capability (version) 1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2
Maximum dimensionality of grid of thread blocks	2				3					
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1				
Maximum y-, or z-dimension of a grid of thread blocks	65535									
Maximum dimensionality of thread block	3									
Maximum x- or y-dimension of a block	512				1024					
Maximum z-dimension of a block	64									
Maximum number of threads per block	512				1024					
Warp size	32									
Maximum number of resident blocks per multiprocessor	8					16			32	
Maximum number of resident warps per multiprocessor	24		32		48	64				
Maximum number of resident threads per multiprocessor	768		1024		1536	2048				
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		128 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255			
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			112 KB	64 KB	96 KB
Number of shared memory banks	16				32					
Amount of local memory per thread	16 KB				512 KB					

If N > 1024 ???

If N > 1024 ???

```
__global__ void add(int *d_a, int *d_b, int *d_c) {
    int i = threadIdx.x;
While (i < N)
   {
     d_c[i] = d_a[i] + d_b [i];
     i += blockDim.x;
}
}
```

```
c[0]   = a[0]   + b[0];
C[1024]= a[1024]+ b[1024];
C[2048]= a[2048]+ b[2048];
…
```

```
C[1]   = a[1]   + b[1];
C[1025]= a[1025]+ b[1025];
C[2049]= a[2049]+ b[2049];
…
```

…

Apenas estamos usando 1 SM!...

Blocos

```
__global__ void add(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);
}
```

Blocos

```
__global__ void add(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];
}
```

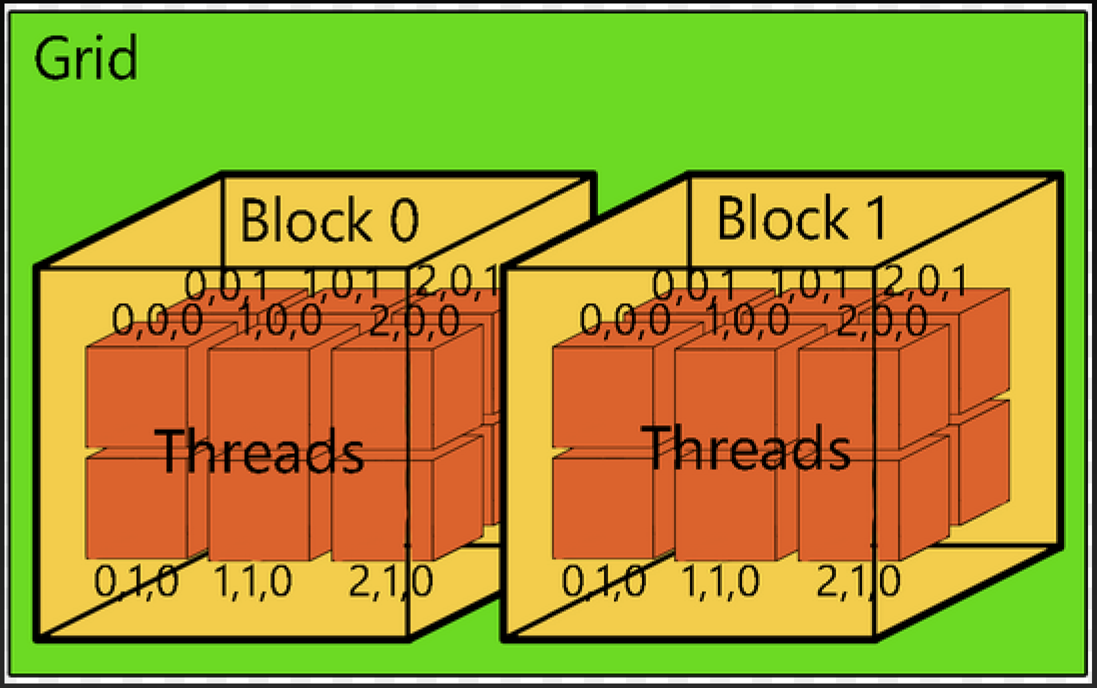
```
int main()
{
    vecAdd <<<K,M>>>(A, B, C);
}
```

threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

Perigo: indices não referenciados...

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


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

Threads podem ser indexados em 1, 2 ou 3 dimensões



(threadIdx.x, threadIdx.y, threadIdx.z)

Threads podem ser indexados em 1, 2 ou 3 dimensões

```
__global__ void MatAdd(int *d_a, int *d_b, int *d_c) {
    int i= threadIdx.x;
    int j= threadIdx.y;

    d_c[i][j] = d_a[i][j] + d_b[i][j];
}

int main()
{
    dim3 threadsPerBlock (N,M)              // N*M < 1024
    vecAdd <<<1,threadsPerBlock>>>(A, B, C);
}
```

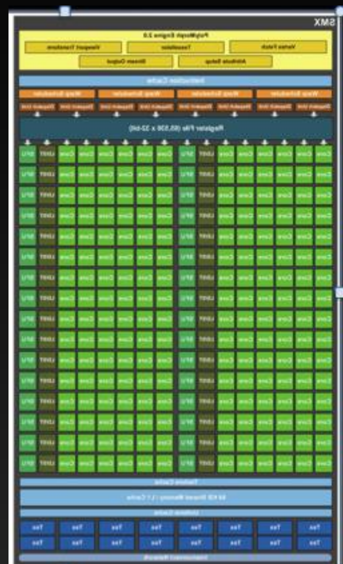
Overview

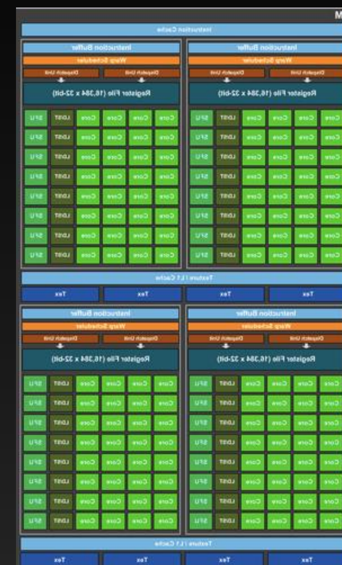
6 GPCs, 84 Volta SMs, 42 TPCs (each including two SMs), and eight 512-bit memory controllers (4096 bits total). Each SM has 64 FP32 Cores, 64 INT32 Cores, 32 FP64 Cores, and 8 new Tensor Cores. Each SM also includes four texture units. 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores,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

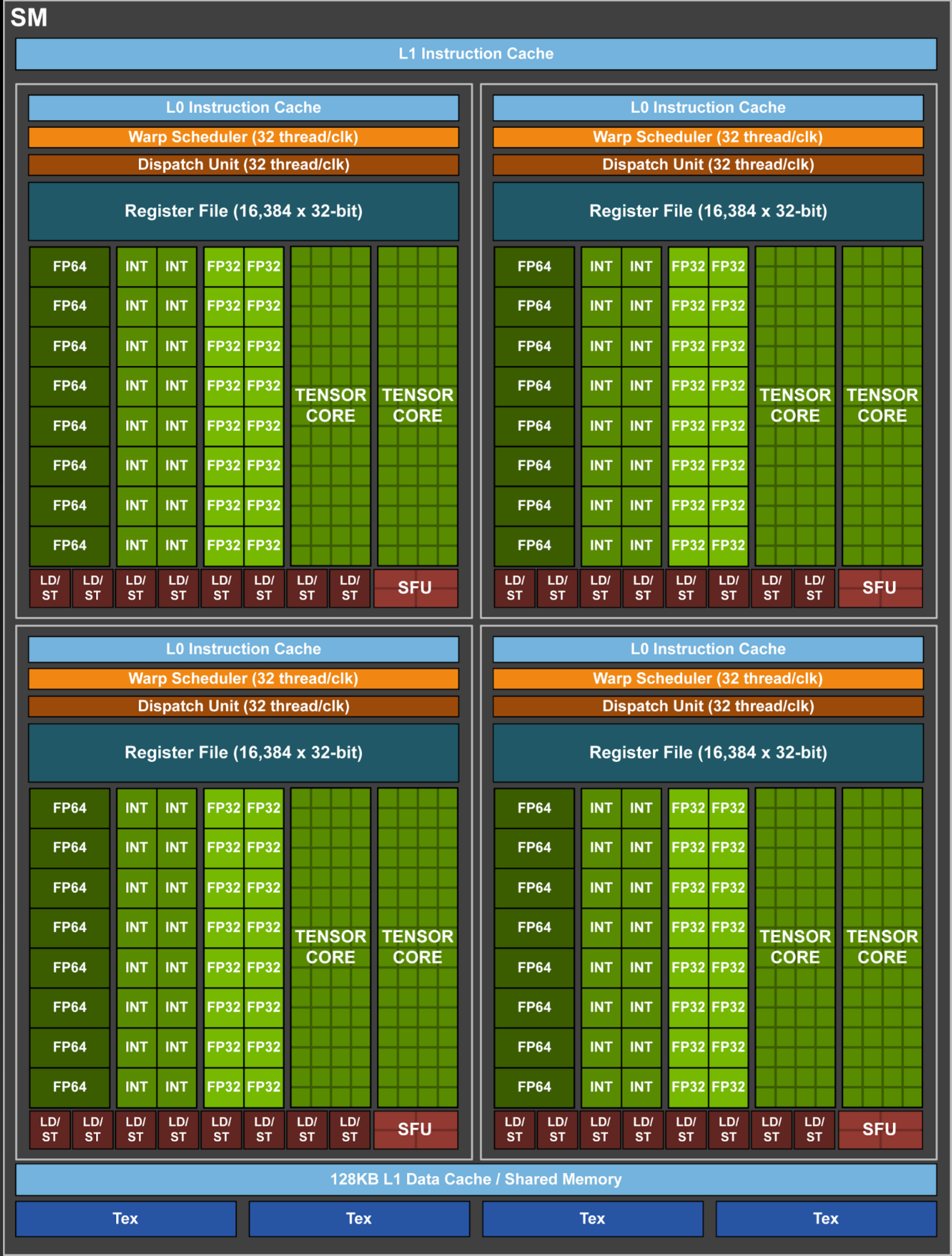
Volta SM





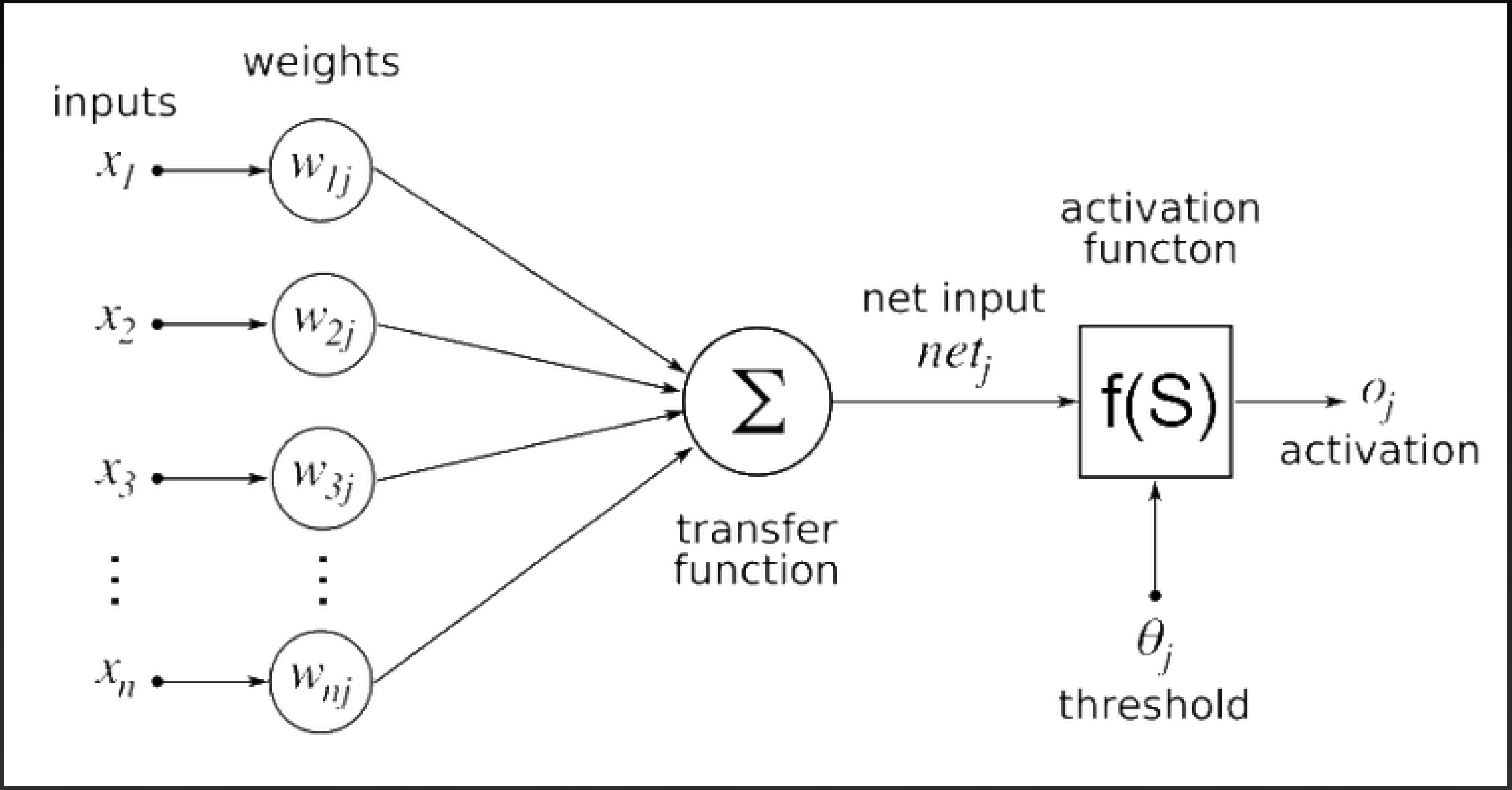
Compute capability

Volta SM

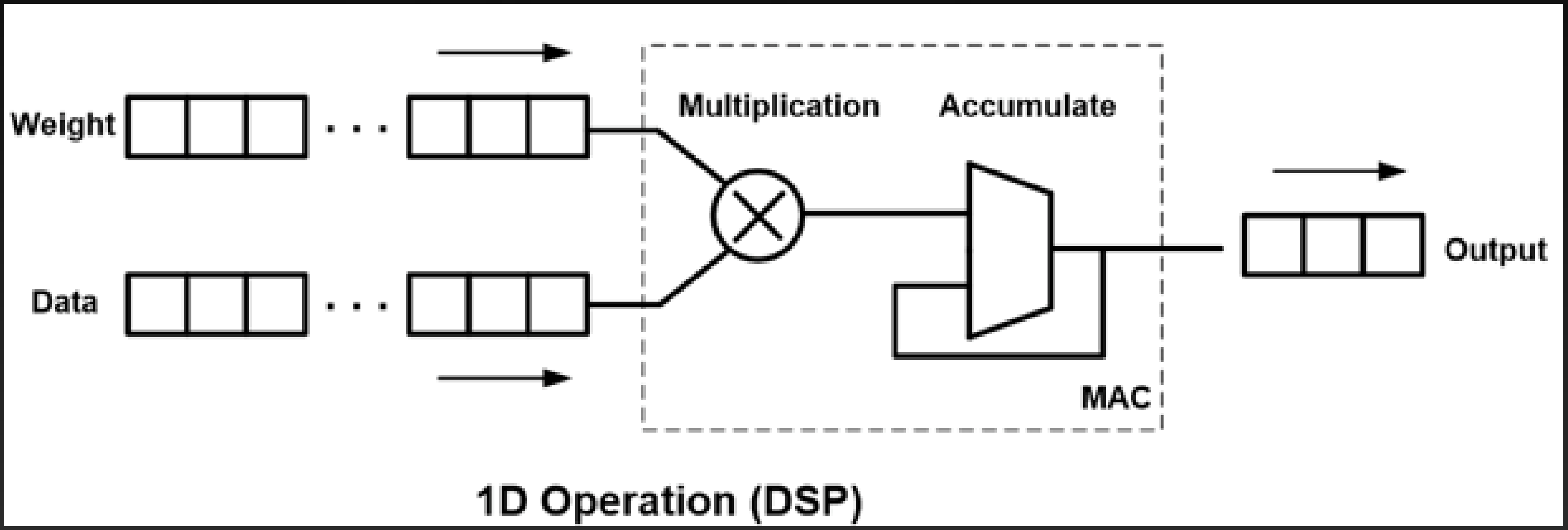

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex

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

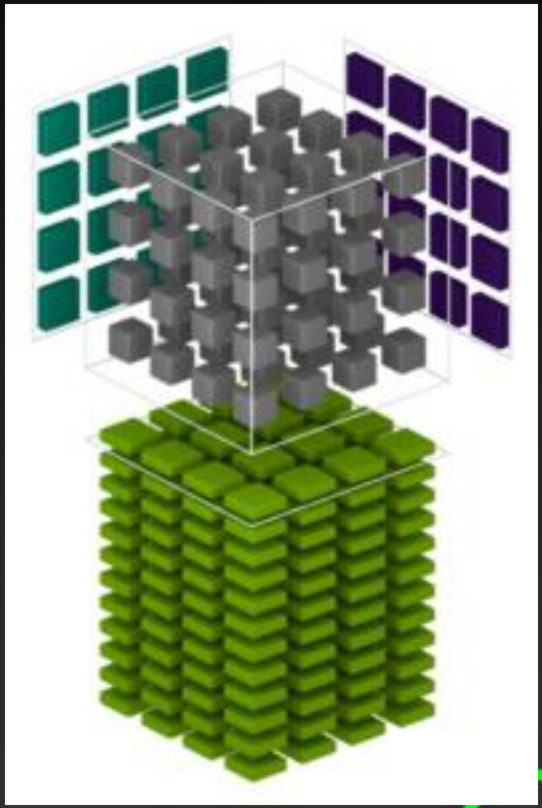
Why GPUs became as powerfull (and indispensable) to Deep Learning as they are for Rendering?



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



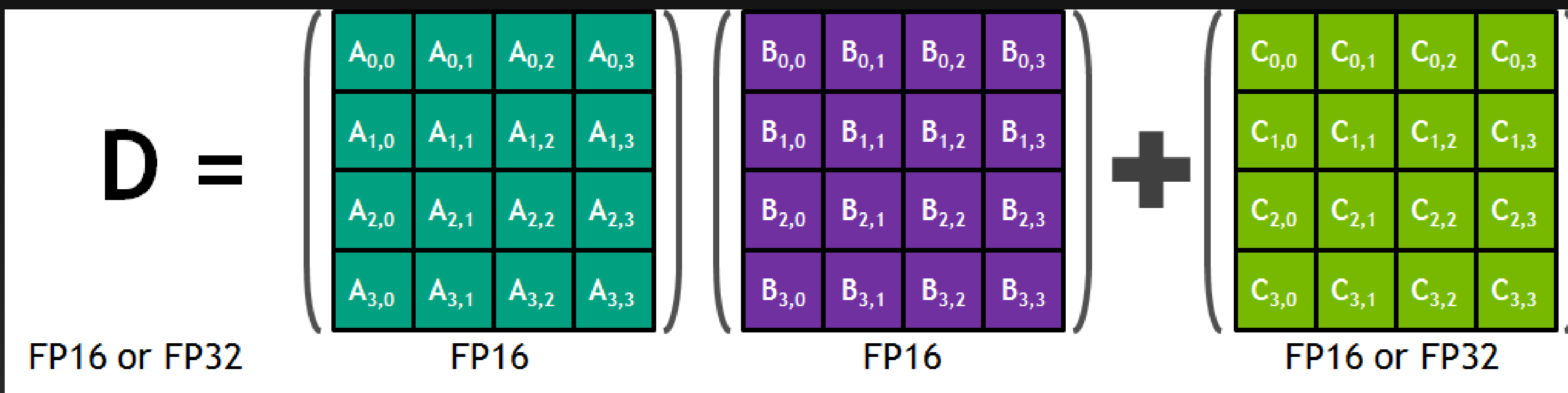
Tensor Cores



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

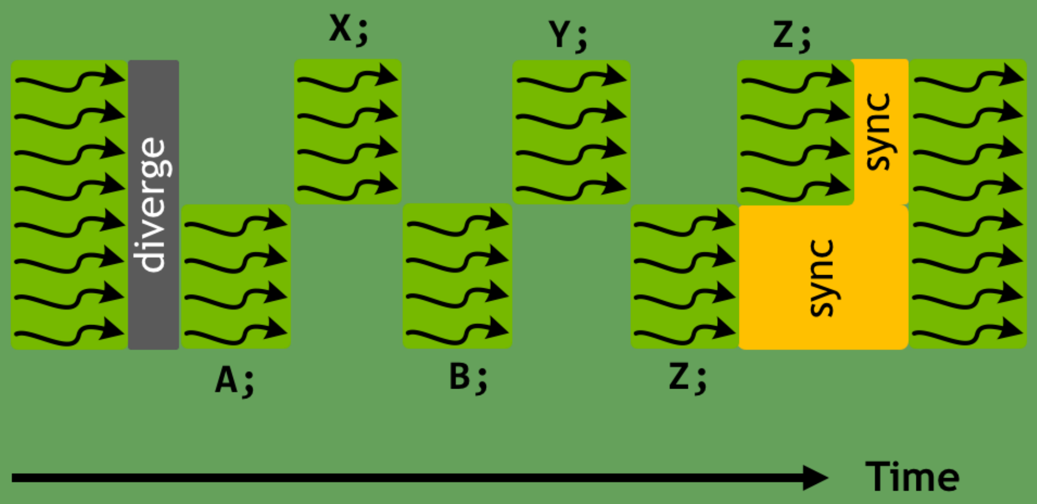


8 TC per SM → 1024 FP per clock per SM

New SIMT model

Volta allows to group threads at a warp level

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

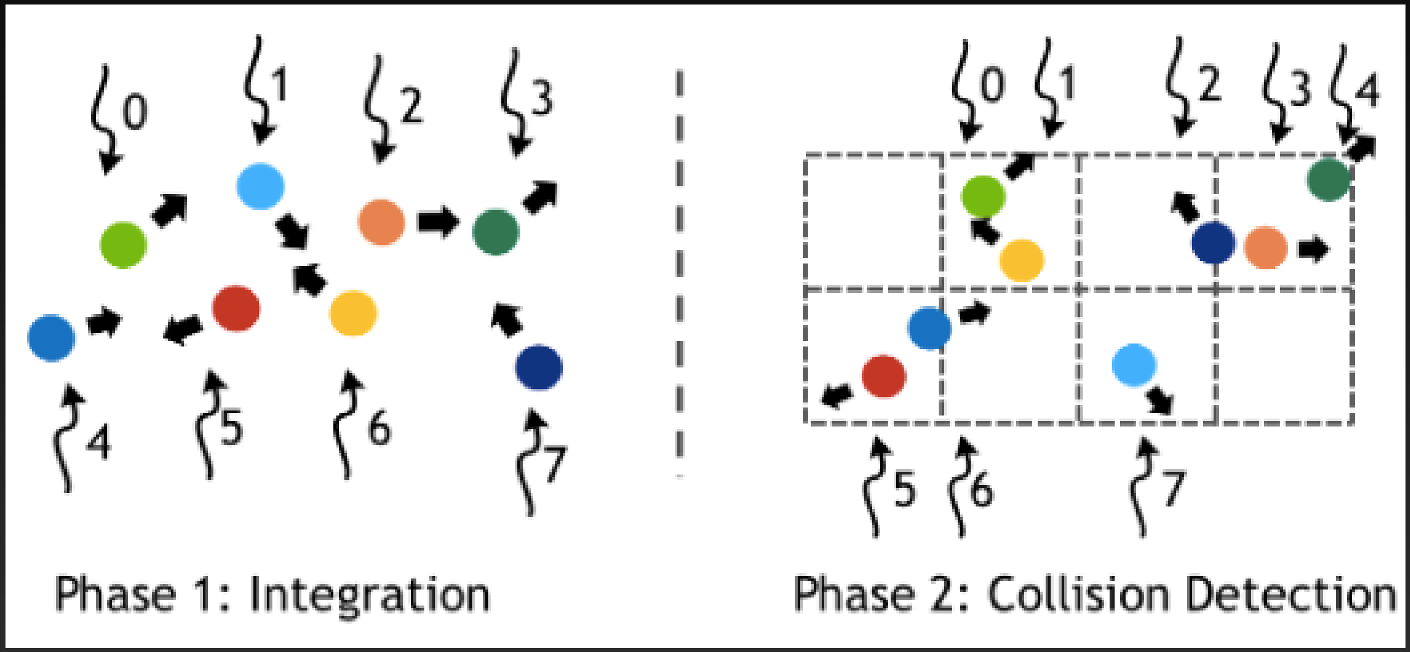


There is no control in the thread level sync at the divergence, in the same warp

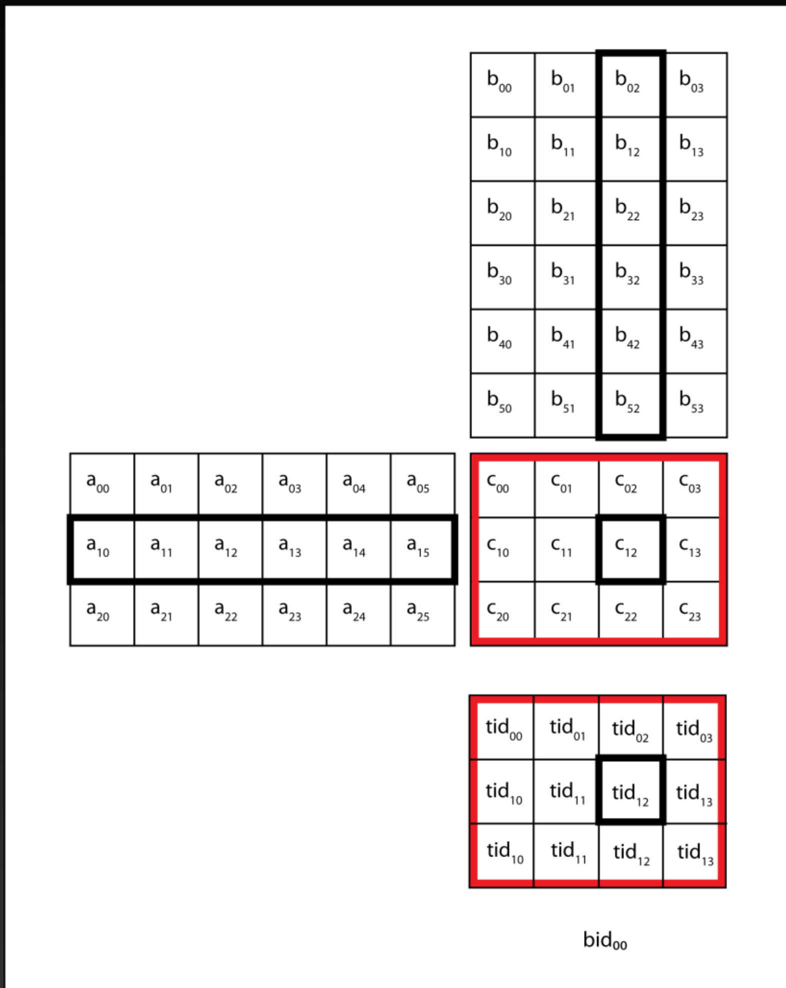
Cooperative Groups

```
__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



Multiplicação de Matrizes



$$A * B = C = \begin{bmatrix} \sum_{i=1}^{m} a_{1i} * b_{i1} & \sum_{i=1}^{m} a_{1i} * b_{i2} & \cdots & \sum_{i=1}^{m} a_{1i} * b_{ir} \\ \sum_{i=1}^{m} a_{2i} * b_{i1} & \sum_{i=1}^{m} a_{2i} * b_{i2} & & \sum_{i=1}^{m} a_{1i} * b_{ir} \\ \vdots & & \ddots & \vdots \\ \sum_{i=1}^{m} a_{ni} * b_{i1} & \sum_{i=1}^{m} a_{ni} * b_{i2} & \cdots & \sum_{i=1}^{m} a_{ni} * b_{ir} \end{bmatrix}$$

Multiplicação de Matrizes
(implementação trivial)

```
__global__ void add(int *d_a, int *d_b, int *d_c, int K) {
    int col= threadIdx.x + blockIdx.x * blockDim.x;
    int row= threadIdx.y + blockIdx.y * blockDim.y;
    cValue = 0.0f;

    for (int k = 0; k < K; k++)
        cValue += d_a[col][k] * d_b[k][row];

    d_c[col][row]= cValue
}
```

Multiplicação de Matrizes

(implementação trivial)

```
__global__ void add(int *d_a, int *d_b, int *d_c, int K) {
    int i= threadIdx.x + blockIdx.x * blockDim.x;
    int j= threadIdx.y + blockIdx.y * blockDim.y;
    cValue = 0;

    for (int k = 0; k < K; k++)
        cValue += d_a[i][k] * d_b[k][j];

    d_c[i][j]= cValue
}
```

PORQUE NÃO É UMA BOA SOLUÇÃO???

Divergencia de Threads

```
__global__ void add(int *d_a) {
   int i= threadIdx.x + blockIdx.x * blockDim.x;

    if ((i%2) != 0)                  //i is odd
        d_a[i] *=2;
    else                             // i is even
        d_a[i] /=2;
}
```

Warps

Independent of the Architecture, it consists on 32 threads per warp. Thread multiple of 32 will optimize the occupacy rate

Coalescence is storng in the same warp

Thread Divergence is also strong in the same warp

Aviso importante

Respect the amount of register size for each Warp

Technical specifications	Compute capability (version)									
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2
Maximum dimensionality of grid of thread blocks	2				3					
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1				
Maximum y-, or z-dimension of a grid of thread blocks	65535									
Maximum dimensionality of thread block	3									
Maximum x- or y-dimension of a block	512				1024					
Maximum z-dimension of a block	64									
Maximum number of threads per block	512				1024					
Warp size	32									
Maximum number of resident blocks per multiprocessor	8					16			32	
Maximum number of resident warps per multiprocessor	24		32		48	64				
Maximum number of resident threads per multiprocessor	768		1024		1536	2048				
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		128 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255			
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			112 KB	64 KB	96 KB
Number of shared memory banks	16				32					
Amount of local memory per thread	16 KB				512 KB					

Kernels concorrentes

```
cudaMalloc ( &dA, sizeA ) ;
cudaMalloc ( &dB, sizeB ) ;
…
cudaMemcpy ( dA, A, size, H2D ) ;
cudaMemcpy ( dB, B, size, H2D ) ;
…
kernelA <<< gridA, blockA>>> ( …, dA, … ) ;
kernelB <<< gridB, blockB>>> ( …, dB, … ) ;
…
```

Shared Memory

- Available for a complete Block. Can only be manipulated by the Device...

- Kepler and newer support banks of 8 bytes of shared memory. Previous architectures accepted 4.

Shared Memory

```
__global__ void copy_vector(float *data)
{
    int index = threadIdx.x;

    __shared__ float temp_data[256];
    temp_data[index] = data[index];
```

…

Shared Memory

```
__global__ void copy_vector(float *data)
{
    int index = threadIdx.x;

    __shared__ float temp_data[256];
    temp_data[index] = data[index];

    __syncthread();

…
```

Is this code more efficient than only using the global memory???

Analisando a eficiência da Shared Memory

```
__global__ void copy_vector(float *data)
{
    int index = threadIdx.x;
    int i, aux=0;

    __shared__ float temp_data[256];
    temp_data[index] = data[index];

    __syncthread();

    for (i=0; i<25; i++)
    {
        aux += temp_data[i];
    }
    data[index] = aux;

...
```

Aviso importante

Respect the amount of shared memory available for each Block

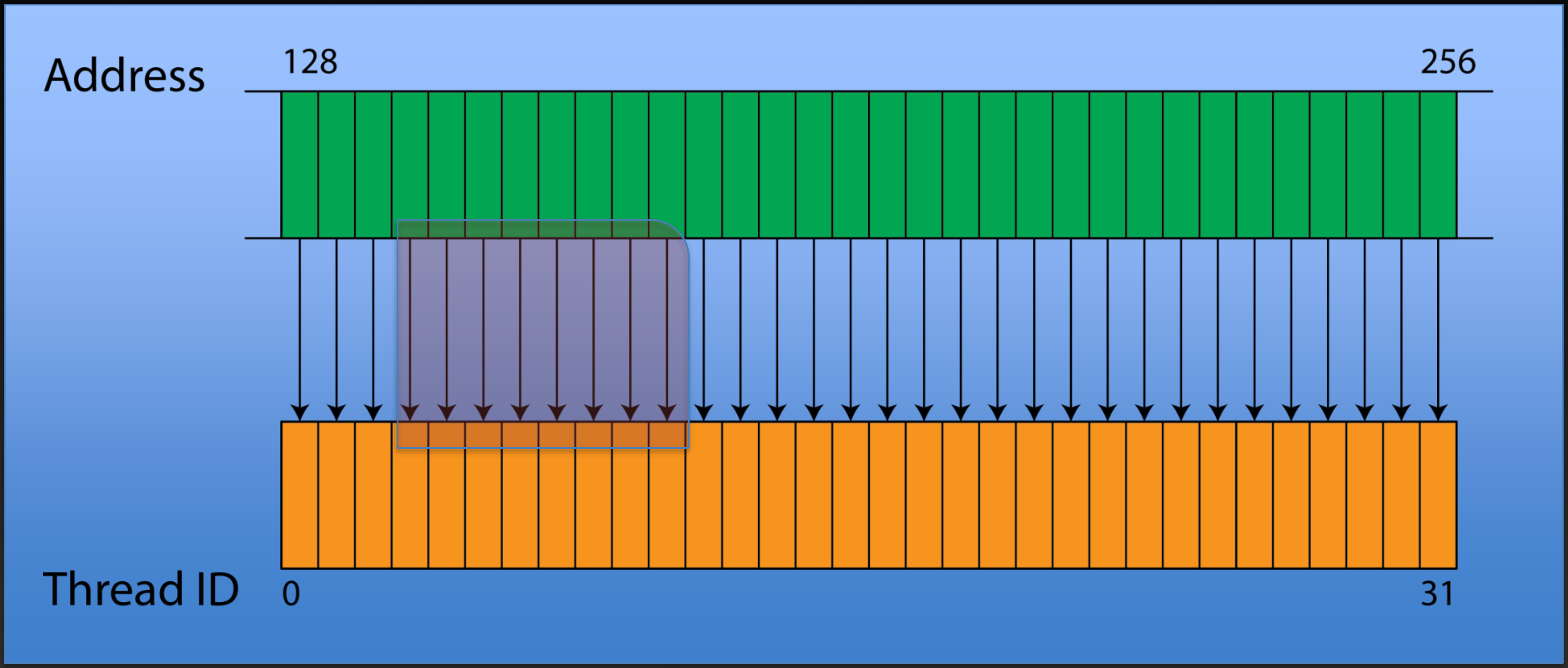
Technical specifications	Compute capability (version)									
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2
Maximum dimensionality of grid of thread blocks	2				3					
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1				
Maximum y-, or z-dimension of a grid of thread blocks	65535									
Maximum dimensionality of thread block	3									
Maximum x- or y-dimension of a block	512				1024					
Maximum z-dimension of a block	64									
Maximum number of threads per block	512				1024					
Warp size	32									
Maximum number of resident blocks per multiprocessor	8					16			32	
Maximum number of resident warps per multiprocessor	24		32		48	64				
Maximum number of resident threads per multiprocessor	768		1024		1536	2048				
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		128 K	64 K	
Maximum number of 32-bit registers per thread	128				63		255			
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			112 KB	64 KB	96 KB
Number of shared memory banks	16				32					
Amount of local memory per thread	16 KB				512 KB					

Implementando o Parallel Reduce

(Shared Memory)

```
__global__ void reduceShared (float *d_In, *d_Out)
{
    external __shared __ s_data[];
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    int tid = threadIdx.x;

    s_data = d_In[index]
    __syncthread();

    for (int stride = blockDim.x/2; stride > 0; stride >>==1) {
        if (tid < stride){
            s_data [index] += s_data[index+s];
        }
    __syncthread();
    if (tid == 0){
        d_Out[blockIdx.x] = s_data[0];
    }
}
```

Coalescencia



Otimizando o código

Each SM fetches 128 bytes per memory access.

Good optimization is obtained when reading 32 bytes .
Reading 64 bits requires one fetch finish for making another.

Examplo de Coalescence

Array of Structures

```
struct st_particle{
   float3 p;
   float3 v;
   float3 a;
};

__global__ void K_Particle_01(st_particle *vet){

   int i = blockDim.x * blockIdx.x + threadIdx.x;
   vet[i].p.x = vet[i].p.x + vet[i].v.x * vet[i].a.x;
   vet[i].p.y = vet[i].p.y + vet[i].v.y * vet[i].a.y;
   vet[i].p.z = vet[i].p.z + vet[i].v.z * vet[i].a.z;

}
```

Data of particle #0 begins in position 0 of the memory, the attributes of particle #2 starts in position 96 bytes of memory and so on.

Examplo de Coalescencia

```
__global__ void K_Particle_02(float *vet_px, float *vet_py, float *vet_pz,
                              float *vet_vx, float *vet_vy, float *vet_vz,
                              float *vet_ax, float *vet_ay, float *vet_az){

    int i = blockDim.x * blockIdx.x + threadIdx.x;
    vet_px[i] = vet_px[i] + vet_vx[i] * vet_ax[i];
    vet_py[i] = vet_py[i] + vet_vy[i] * vet_ay[i];
    vet_pz[i] = vet_pz[i] + vet_vz[i] * vet_az[i];


}

```

Structure of Arrays

Atomic Operations

Exercicio: o que acontece com este código???

```
#define BLOCKS 1000
#define THREADSPERBLOCK 1000
#define size 10

__global__ void incrementVector (float *data)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    data[index] = data[index] + 1;
}
```

E agora?

```
#define BLOCKS 1000
#define THREADSPERBLOCK 1000
#define size 10

__global__ void incrementVector (float *data)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    index = index % size;
    data[index] = data[index] + 1;
}
```

Exercicio: corrigir usando barreiras...

```
#define BLOCKS 1000
#define THREADSPERBLOCK 1000
#define size 10

__global__ void incrementVector (float *data)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    index = index % size;
    data[index] = data[index] + 1;
}
```

Atomic Operation

```
#define BLOCKS 1000
#define THREADSPERBLOCK 1000
#define size 10

__global__ void incrementVector (float *data)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    index = index % size;
    atomicAdd(&data[index], 1);
}
```

Lista de Atomic Operation

```
int atomicAdd(int* address, int val);

int atomicSub(int* address, int val);

int atomicExch(int* address, int val);

 int atomicMin(int* address, int val);

 int atomicMax(int* address, int val);

 unsigned int atomicInc(unsigned int* address, unsigned int val);  // old >= val ? 0 : (old+1)

 unsigned int atomicDec(unsigned int* address, unsigned int val);

int atomicAnd(int* address, int val);  // Or and Xor also available
```

Works fine for int . Only add and exchange work for float and double

Limitações de Atomic Operation

1. only a set of operations are supported
2. Restricted to data types
3. Random order in operation
4. Serialize access to the memory (there is no magic!)

Great improvements on latest archictectures

Streams

Task Parallelism: two or more completely different tasks in parallel

Streams

cudaHostAlloc: malloc memory in the Host

Differs from traditional malloc() since it guarantees that the memory will be page-locked, i.e., it will never be paged to memory out to disk (assures that data will allways be resident at physical memory)

Constraint: doing so the memory may run out much faster that when using malloc…

Streams

Knowing the physical adress buffer allows the GPU to use the DMA (Direct Memory Access), which proceeds without the intervention of the CPU

Streams

```
…
int *a, *dev_a;

a = (int*)malloc(size*sizeof(*a));
cudaMalloc ( (void**)&dev_a, size * sizeof (*dev_a)));

cudaMemcpy (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice));

…
```

Streams

```
…
int *a, *dev_a;

cudaHostAlloc ( (void**)&a    , size * sizeof (*a), cudaHostAllocDefault ));
cudaMalloc    ( (void**)&dev_a, size * sizeof (*dev_a)));

cudaMemcpy (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice));

cudaFreeHost ( a );
cudaFree     (dev_a);

…
```

Streams

GPU allow to create specific order of operations using streams. In some situations it allows to create parallel tasks.

Streams

```
…
int *a, *dev_a;
cudaStream_t stream;

cudaStreamCreate(&stream);

cudaMalloc    ( (void**)&dev_a, size * sizeof (*dev_a)));
cudaHostAlloc ( (void**)&a    , size * sizeof (*a), cudaHostAllocDefault ));

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));

// A stream operation is Asynchronous. Each stram opeartion only starts
// after the previous stream operation have finished

Kernel <<<GridDim, BlockDim, stream>>> (dev_a);

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));

cudaStreamDEstroy (stream);
```

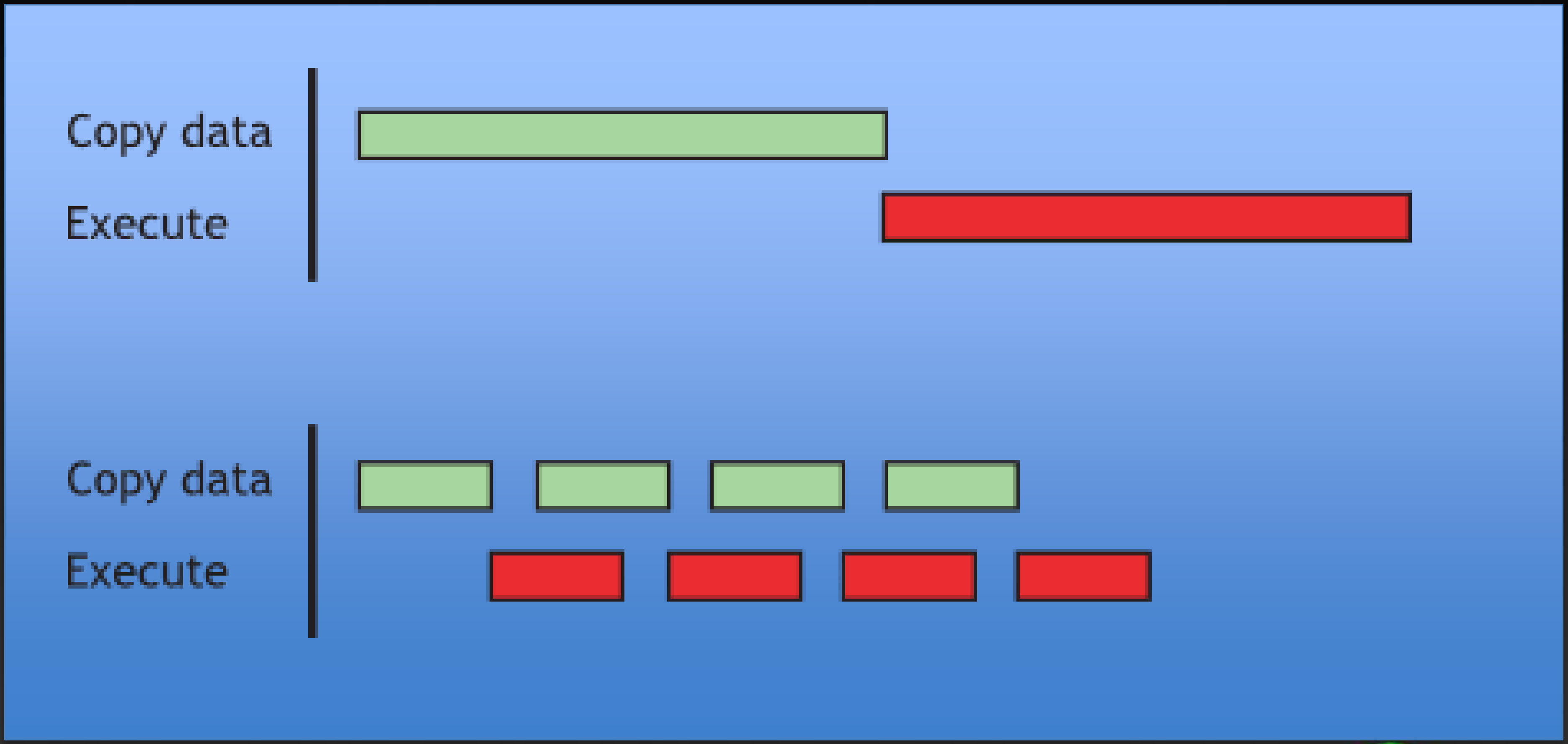
Streams

```
…
int *a, *dev_a;
cudaStream_t stream;

cudaStreamCreate(&stream);

cudaMalloc    ( (void**)&dev_a, size * sizeof (*dev_a)));
cudaHostAlloc ( (void**)&a    , size * sizeof (*a), cudaHostAllocDefault ));

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));    // Async copy only works with page locked memory

// A stream operation is Asynchronous. Each stram opeartion only starts
// after the previous stream operation have finished

Kernel <<<GridDim, BlockDim, stream>>> (dev_a);

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));

cudaStreamDEstroy (stream);
```

Streams

```
…
int *a, *dev_a;
cudaStream_t stream;

cudaStreamCreate(&stream);

cudaMalloc    ( (void**)&dev_a, size * sizeof (*dev_a)));
cudaHostAlloc ( (void**)&a    , size * sizeof (*a), cudaHostAllocDefault ));

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));

// A stream operation is Asynchronous. Each stram opeartion only starts
// after the previous stream operation have finished

Kernel <<<GridDim, BlockDim, stream>>> (dev_a);

cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a), cudaMemcpyHostToDevice,
stream));

cudaStreamDEstroy (stream);
```

Optimizing code with Asynchronous operations



Stream overlaps

```
…
#define N (1024 * 1024)
#define TOTAL_SIZE (N*20)

Int *h_a, *h_b, *h_c;

Int *d_a0, *d_b0, *d_c0;
Int *d_a1, *d_b1, *d_c1;

cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

cudaMalloc    ( (void**)&d_a0, N*sizeof (int));
cudaMalloc    ( (void**)&d_b0, N*sizeof (int));
cudaMalloc    ( (void**)&d_c0, N*sizeof (int));
cudaMalloc    ( (void**)&d_a1, N*sizeof (int));
cudaMalloc    ( (void**)&d_b1, N*sizeof (int));
cudaMalloc    ( (void**)&d_c1, N*sizeof (int));
```

Stream overlaps

...

```
cudaHostAlloc ( (void**)&h_a, TOTAL_SIZE*sizeof (int), cudaHostAllocDefault ));
cudaHostAlloc ( (void**)&h_b, TOTAL_SIZE*sizeof (int), cudaHostAllocDefault ));
cudaHostAlloc ( (void**)&h_c, TOTAL_SIZE*sizeof (int), cudaHostAllocDefault ));

For (int i=0; i<TOTAL_SIZE; i++){
    h_a[i] = rand();
    h_b[i] = rand();
}
```

Stream overlaps

```
For (int i=o; i < TOTAL_SIZE ; i+=N*2)
{

    cudaMemcpyAsync (dev_a0, h_a+i, N* sizeof(int), cudaMemcpyHostToDevice,
                stream0));
    cudaMemcpyAsync (dev_b0, h_b+i, N* sizeof(int), cudaMemcpyHostToDevice,
                stream0));
    kernel<<<N/256, 256, 0, stream0>>> (d_a0, d_b0, d_c0);

    cudaMemcpyAsync (h_c+i, dc_0, N* sizeof(int), cudaMemcpyDeviceToHost,
                stream0));



    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                    stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                    stream1));
    kernel<<<N/256, 256, 0, stream0>>> (d_a1, d_b1, d_c1);

    cudaMemcpyAsync (h_c+i+N, dc_1, N* sizeof(int), cudaMemcpyDeviceToHost,
                stream1));
}
```

Stream overlaps

```
    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    kernel<<<N/256, 256, 0, stream0>>> (d_a1, d_b1, d_c1);

    cudaMemcpyAsync (h_c+i+N, dc_1, N* sizeof(int), cudaMemcpyDeviceToHost,
                 stream1));
}

cudaStreamSynchronize (stream0);
cudaStreamSynchronize (stream1);

// frees and destroys…
```

Stream overlaps

```
    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    kernel<<<N/256, 256, 0, stream0>>> (d_a1, d_b1, d_c1);

    cudaMemcpyAsync (h_c+i+N, dc_1, N* sizeof(int), cudaMemcpyDeviceToHost,
                 stream1));
}

cudaStreamSynchronize (stream0);
cudaStreamSynchronize (stream1);

// frees and destroys…
```

Esta versão ainda não traz otimizações:
Sobrecarga do engine de memória e kernel

Improving Stream

```
For (int i=o; i < TOTAL_SIZE ; i+=N*2)
{

    cudaMemcpyAsync (dev_a0, h_a+i, N* sizeof(int), cudaMemcpyHostToDevice,
                stream0));
    cudaMemcpyAsync (dev_b0, h_b+i, N* sizeof(int), cudaMemcpyHostToDevice,
                stream0));

    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                    stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                    stream1));

    kernel<<<N/256, 256, 0, stream0>>> (d_a0, d_b0, d_c0);
    kernel<<<N/256, 256, 0, stream0>>> (d_a1, d_b1, d_c1);

    cudaMemcpyAsync (h_c+i, dc_0, N* sizeof(int), cudaMemcpyDeviceToHost,
                stream0));
    cudaMemcpyAsync (h_c+i+N, dc_1, N* sizeof(int), cudaMemcpyDeviceToHost,
                stream1));
}
```

Optimizing with compiler directives

CUDA capability	Features
2.0	Fermi architecture
3.0	Kepler architecture
3.2	Unified memory programming
3.5	Dynamic parallelism
5.0, 5.2 and 5.3	Maxwell

Directives

nvcc -arch=compute_20 -code=sm_20,sm_32, sm_35, sm_50,sm_52,sm_53 foo.cu -o foo

nvcc -arch=compute_35 -code=sm_35 foo.cu -o foo

nvcc -use_fast_math foo.cu -o foo

Last advices…

- Find ways to parallelize sequential code,
- Minimize data transfers between the host and the device,
- Adjust kernel launch configuration to maximize device utilization,
- Ensure global memory accesses are coalesced,
- Minimize redundant accesses to global memory whenever possible,
- Avoid different execution paths within the same warp.

Read more at: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#ixzz3jGmjoXLj

NVLink

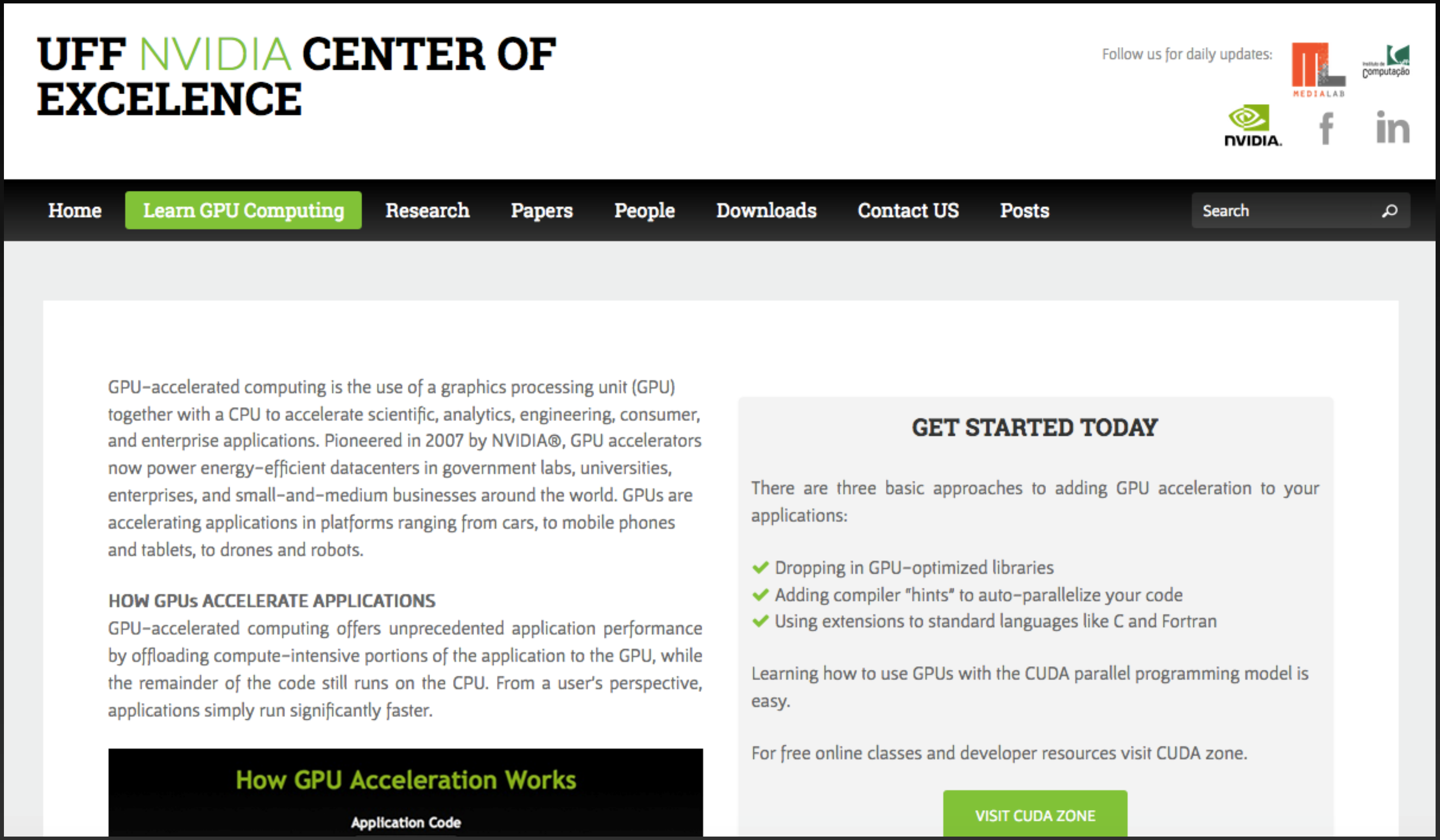
Mixed Precision

“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.”

P100: 21.2 Tflops for Half precision

half a, b …

GPU Educational Kit



GPU Educational Kit

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

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

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

esteban@ic.uff.br