

JAI 6 - Deep Learning

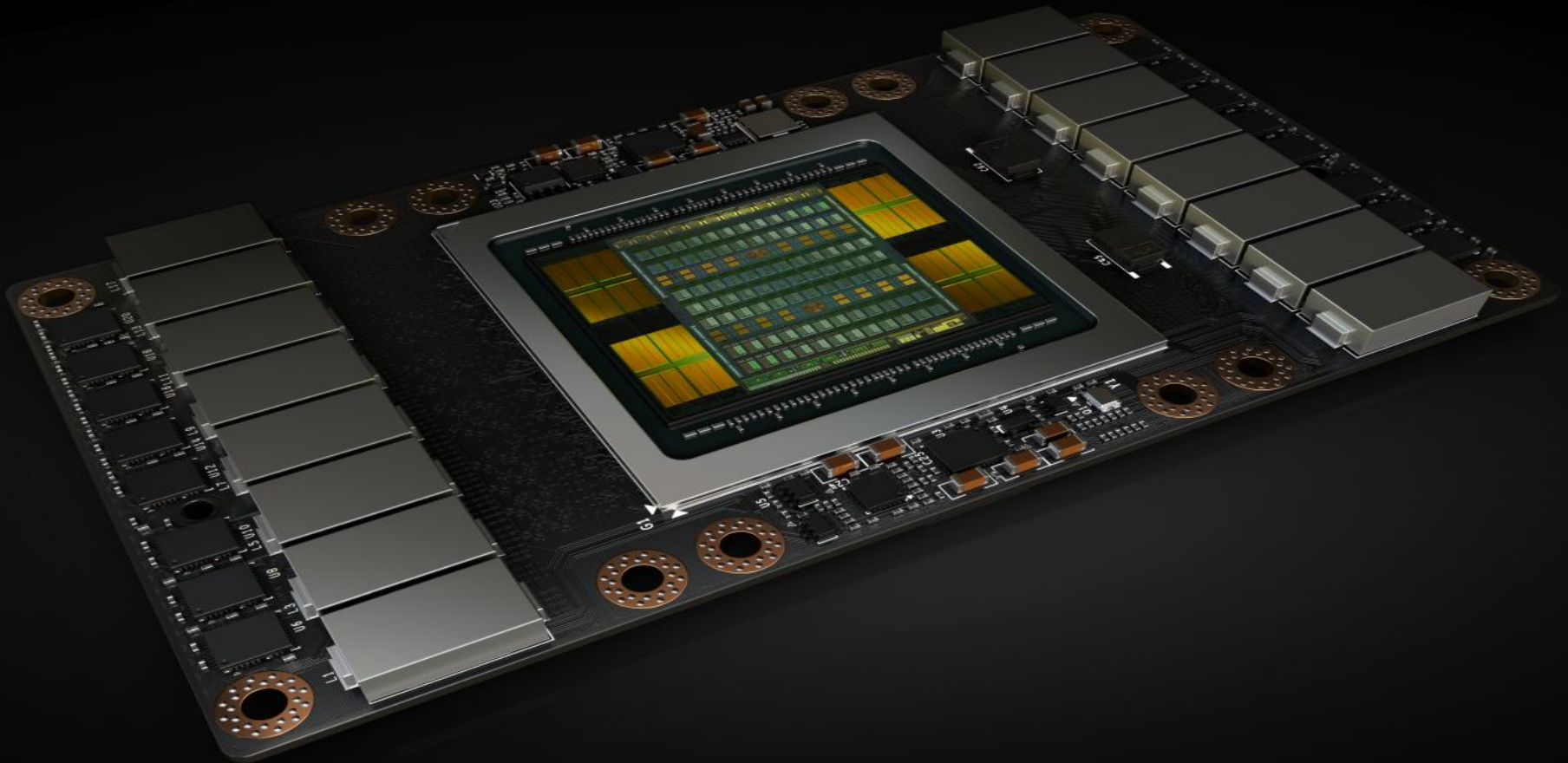
Teoria e Prática

Esteban Clua e Cristina Nader Vasconcelos
Universidade Federal Fluminense

GPUs

Volta

- Acelerador mais poderoso construído até hoje.
 - E é Dedicado a Deep Learning!



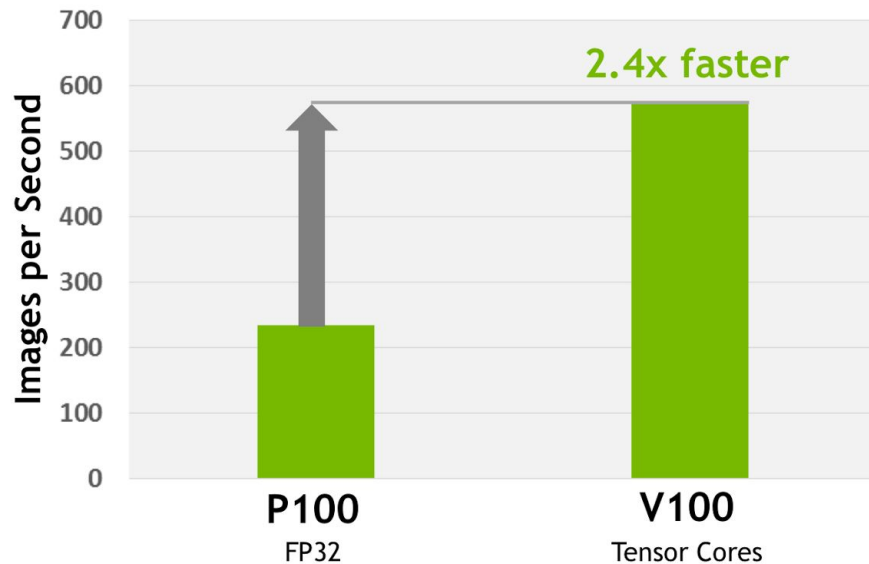
Capacidade

- *7.5 TFLOP/s de precisao dupla para floating-point (FP64) performance;*
- *15 TFLOP/s em precisão simples (FP32);*
- *120 Tensor TFLOP/s para mixed precision em operações matrix-multiply-and-accumulate.*



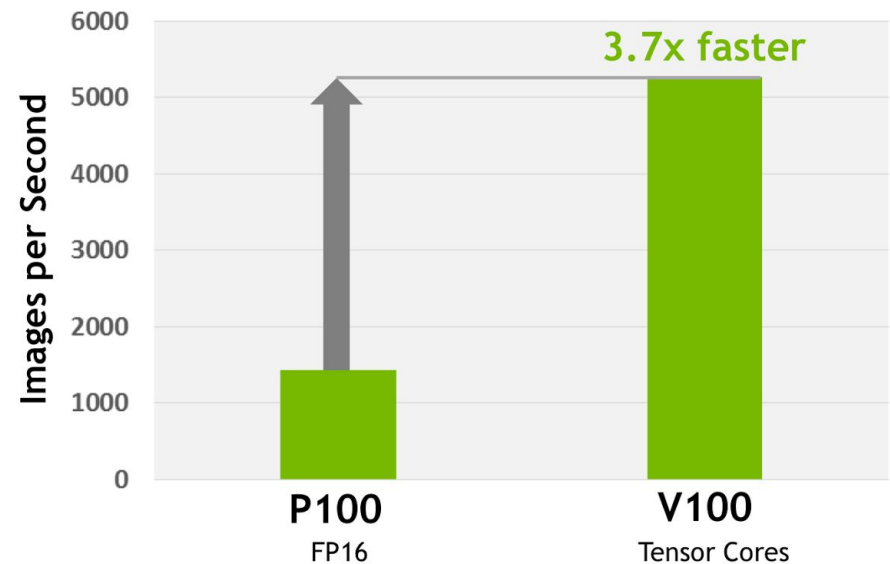
Arquitetura dedicada a Deep Learning

ResNet-50 Training



ResNet-50 Inference

TensorRT - 7ms Latency



Arquitetura Volta



Novos Streaming Multiprocessor

- *Otimizados para deep learning*
- *Tensor Cores (ganhos de até 12x para treinamento)*
- *Capacidade de escalonamento independente de threads*
- *Nova arquitetura para L1 Data Cache e Shared Memory*



Memória mais rápida

*900 GB/sec peak bandwidth
NVLink 2.0*



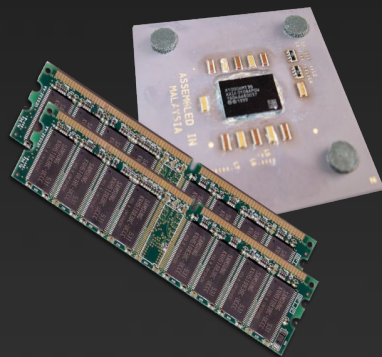
Paradigmas de GPU Programming

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);

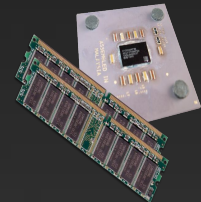
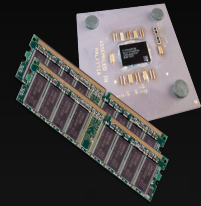
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

parallel code

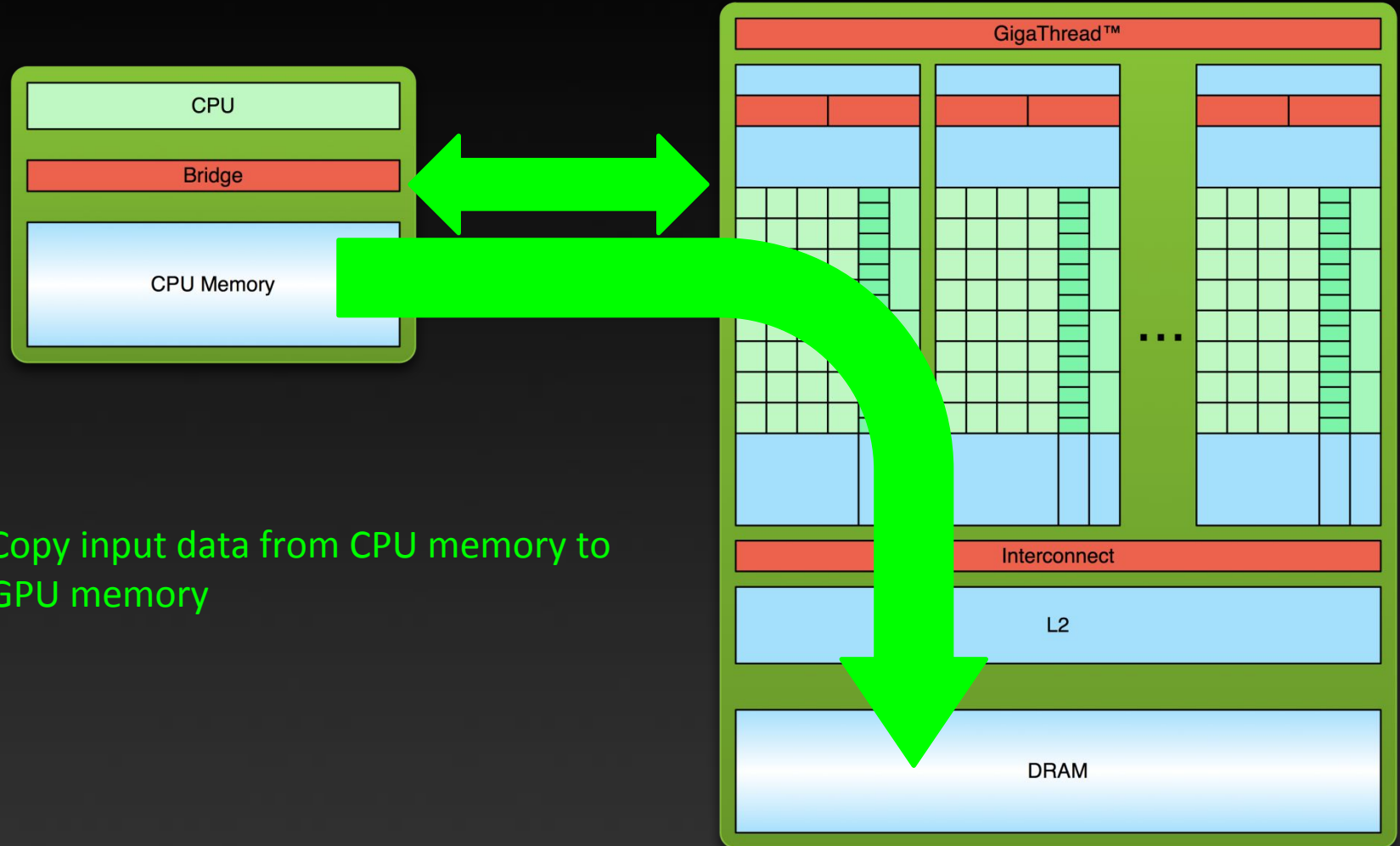
serial code



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

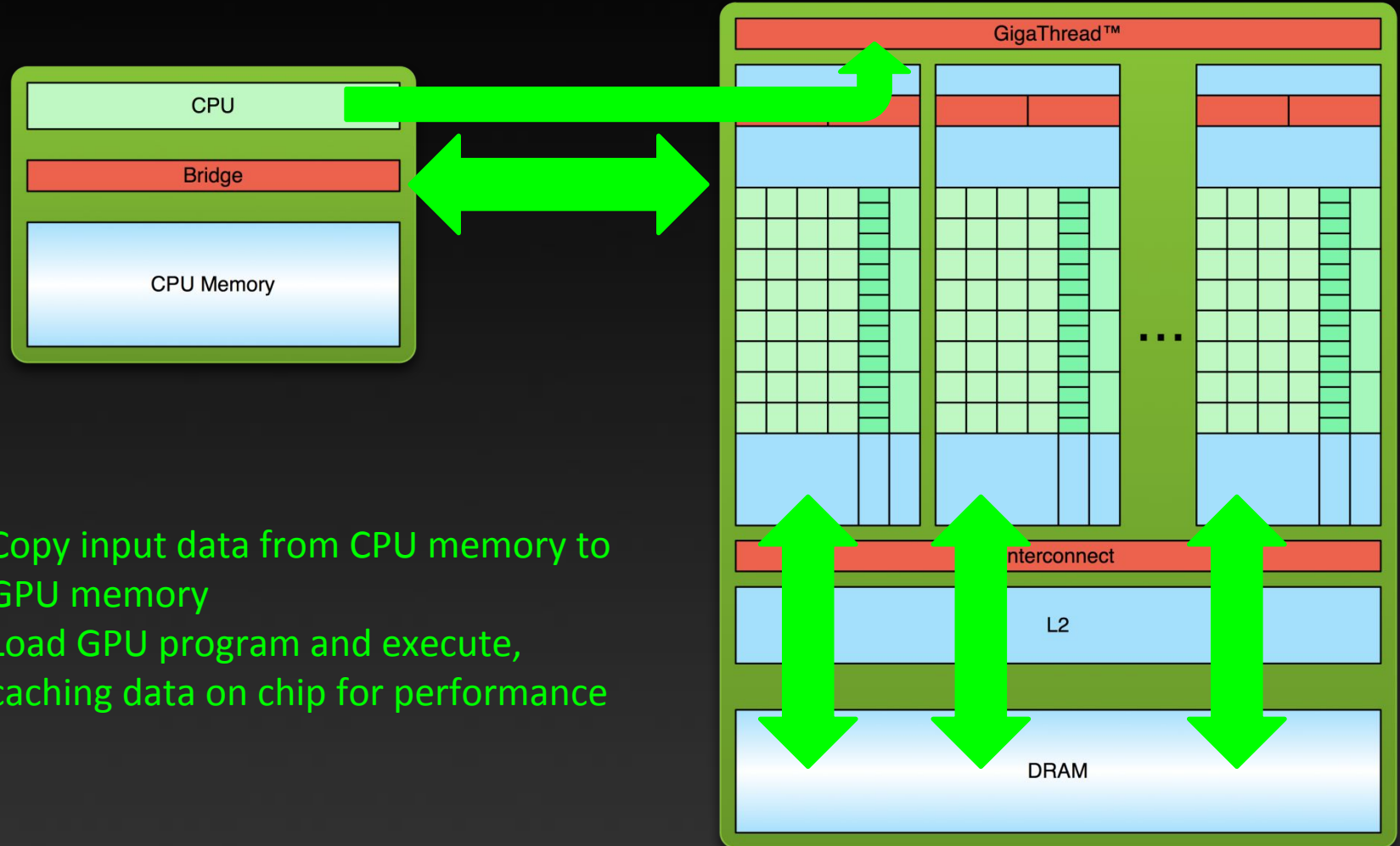


GPU Computing Flow



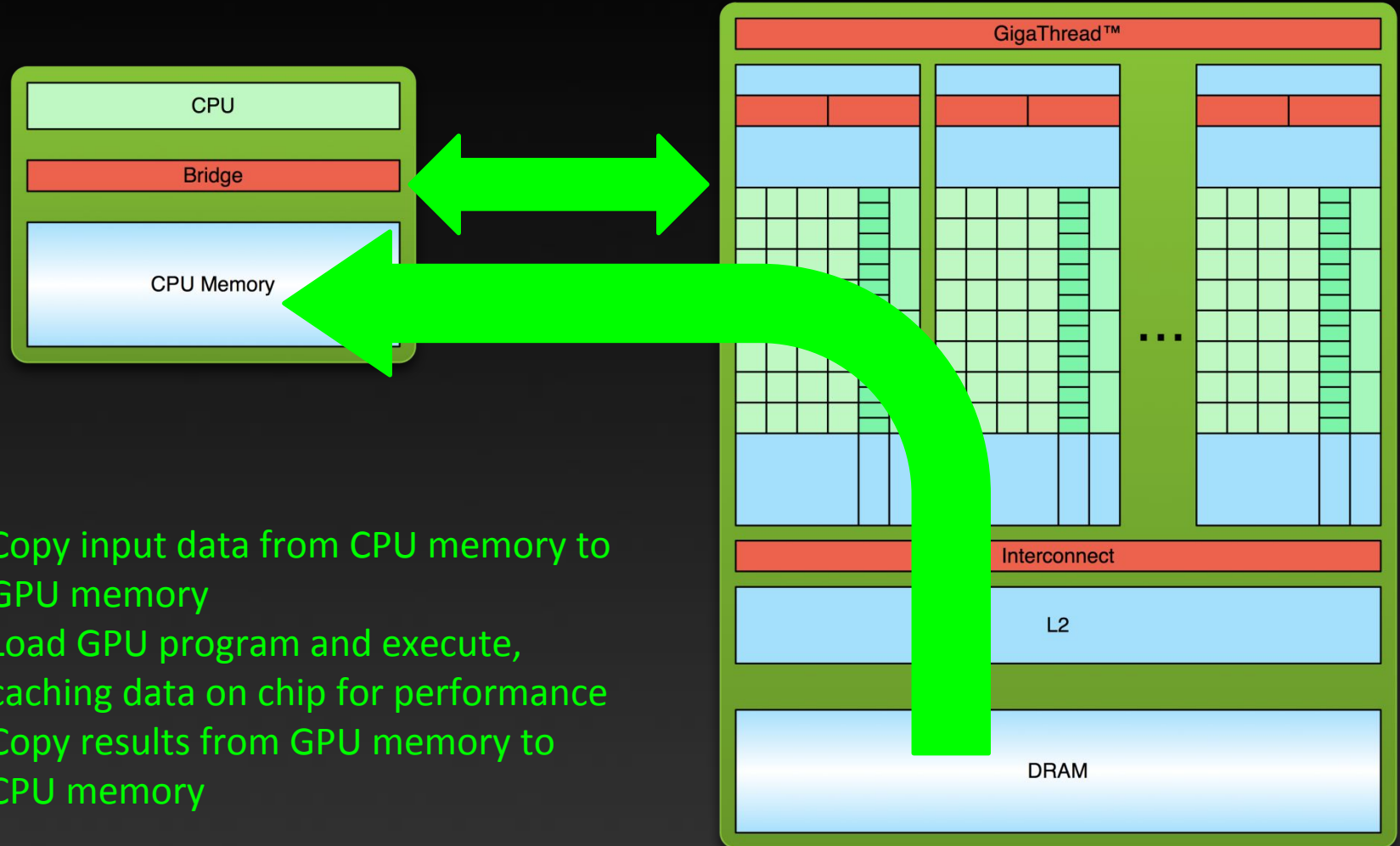
1. Copy input data from CPU memory to GPU memory

GPU Computing Flow



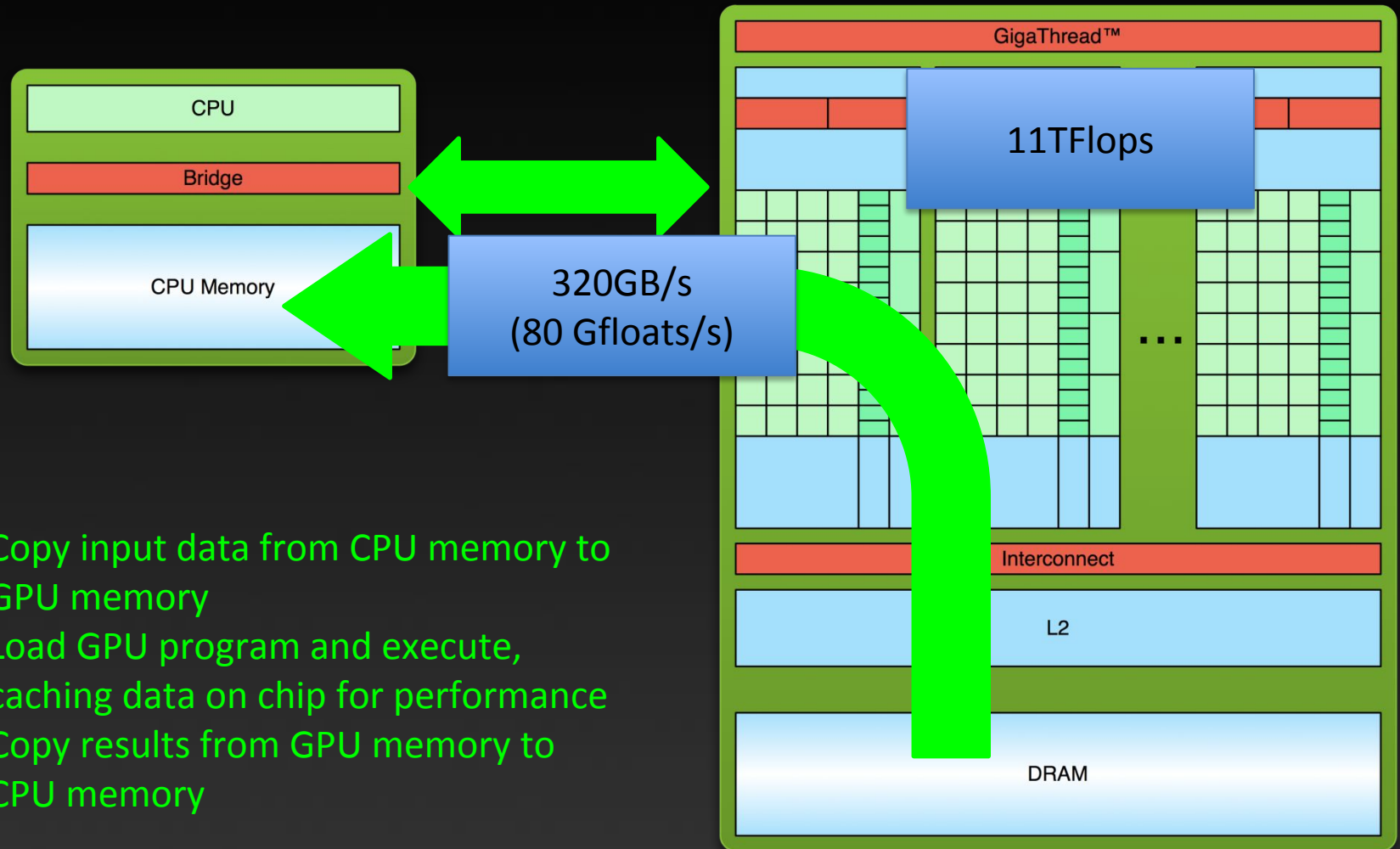
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

GPU Computing Flow

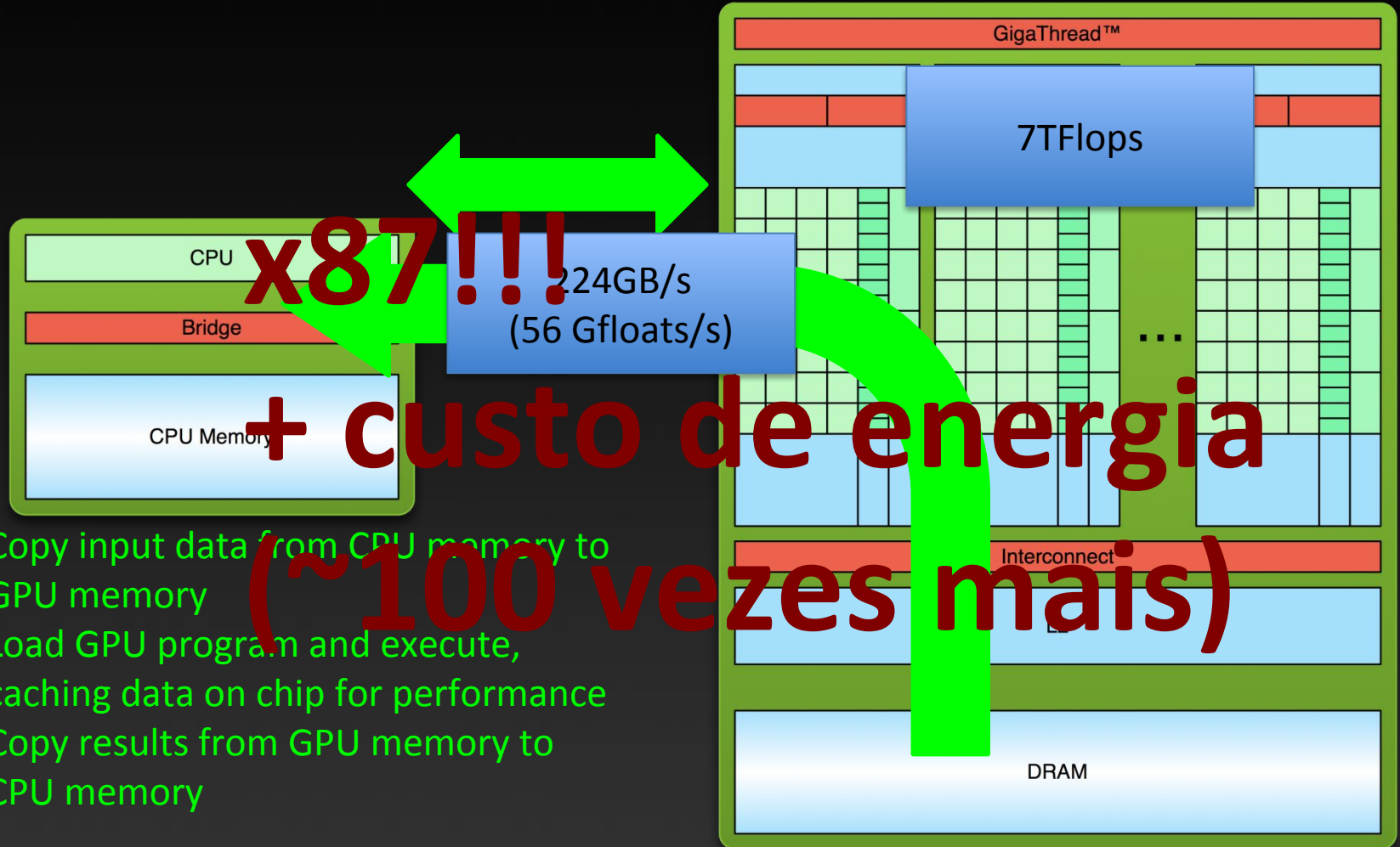


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

GPU Computing Flow



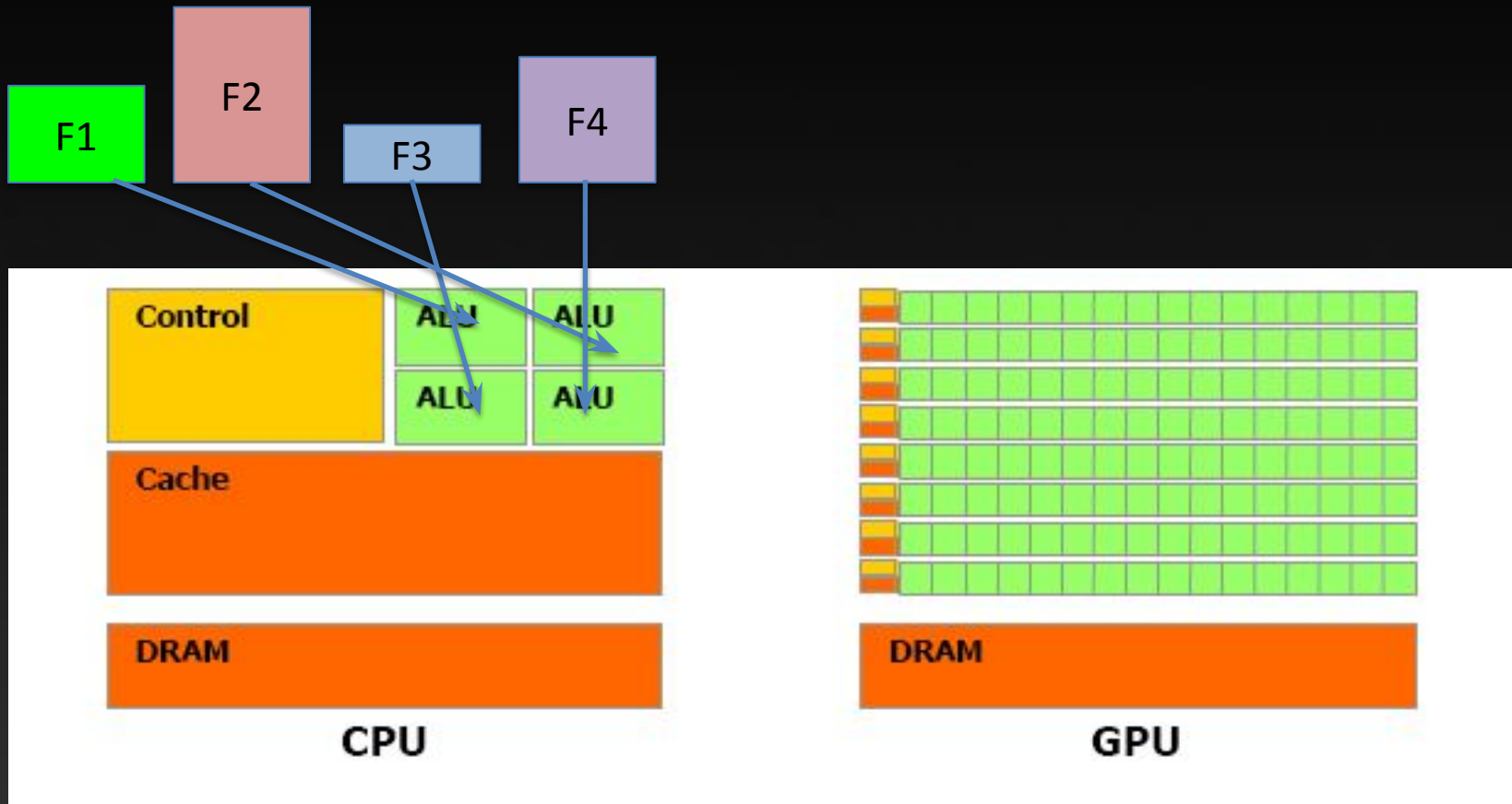
GPU Computing Flow



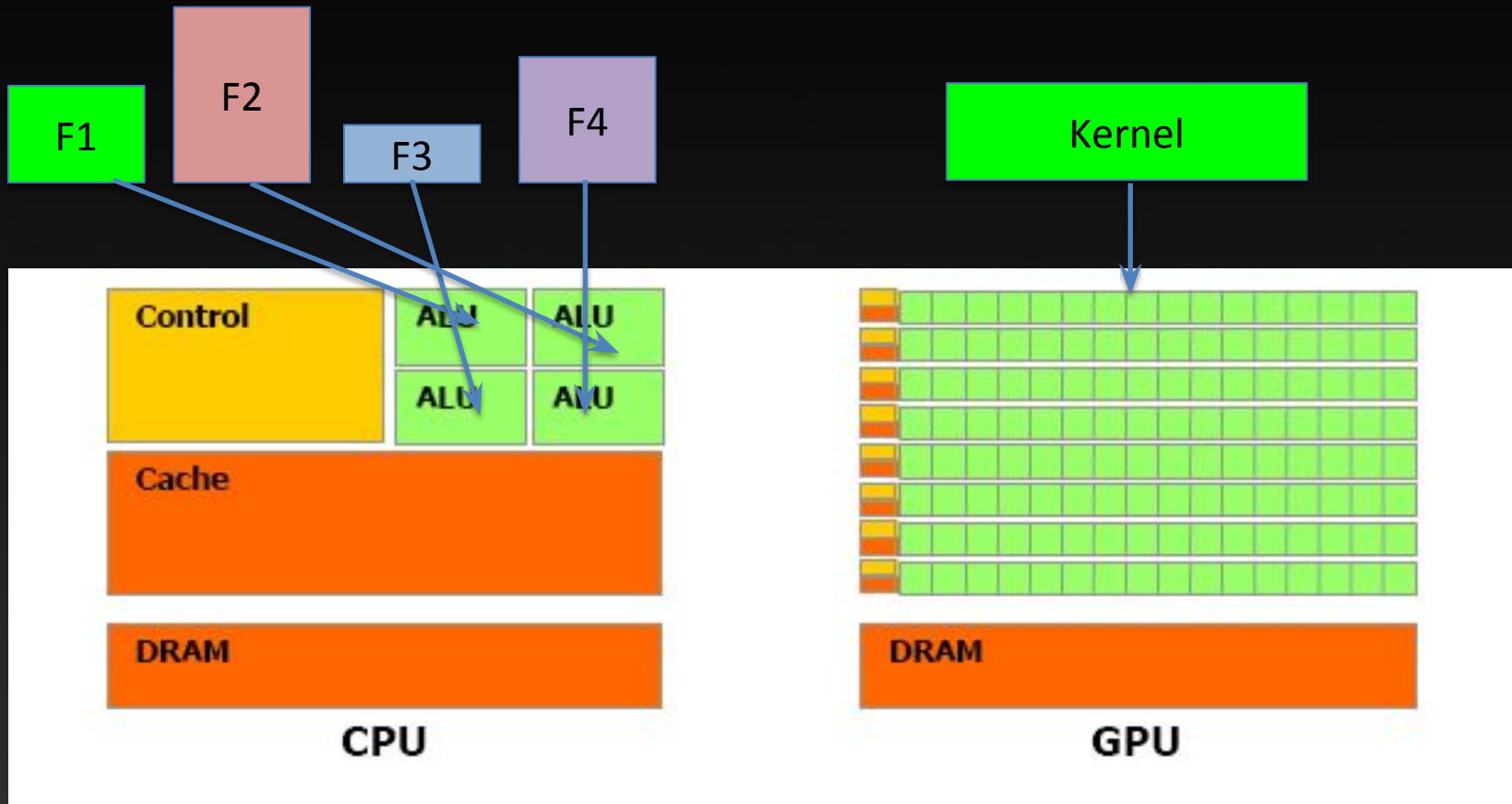
#3 – 1 kernels, muitos threads...



Threads em GPU x CPU



Threads em GPU x CPU



Modelo SIMT

SIMT

means

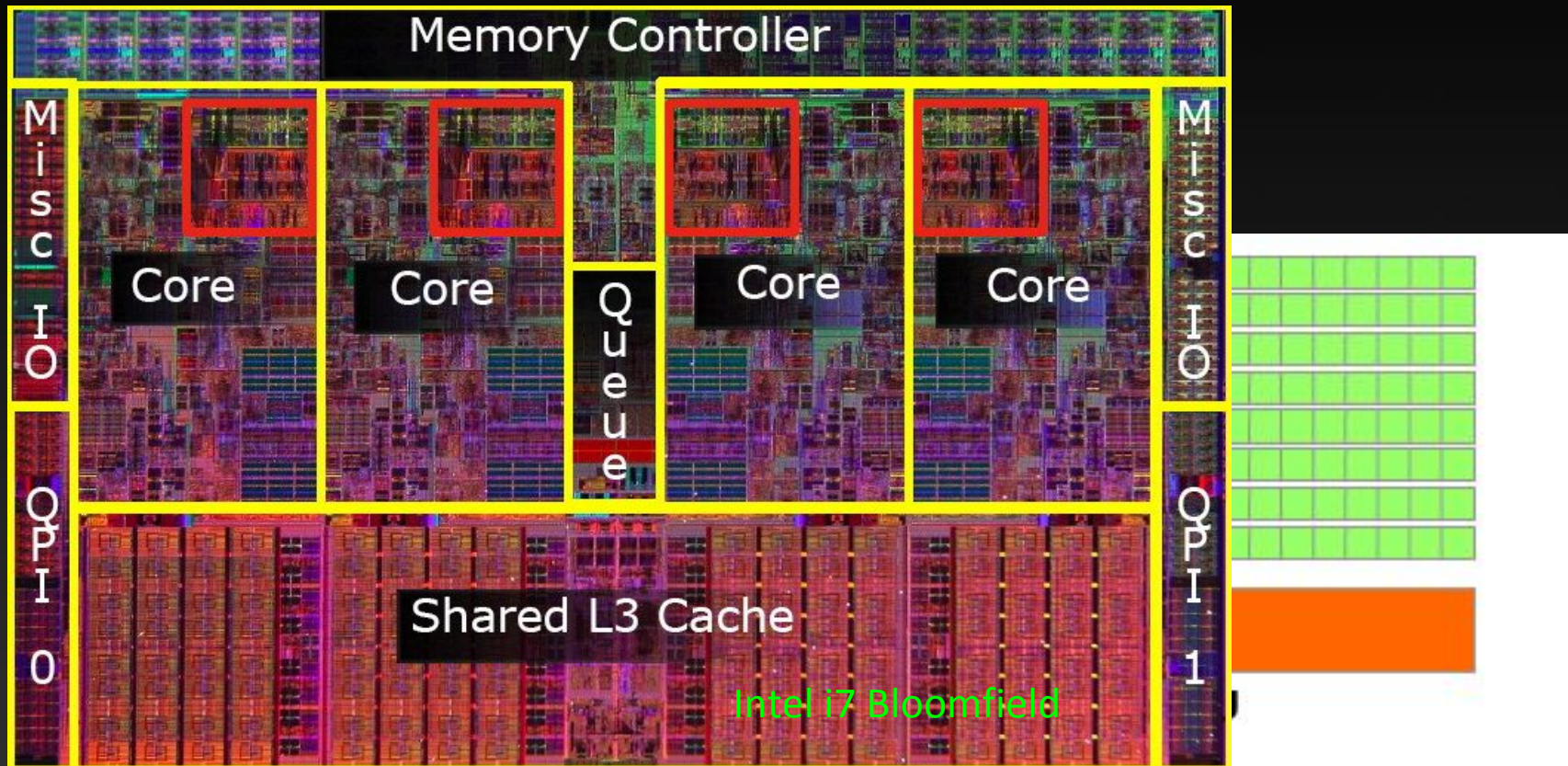
Single Instruction
Multiple Thread

...

by allacronyms.com



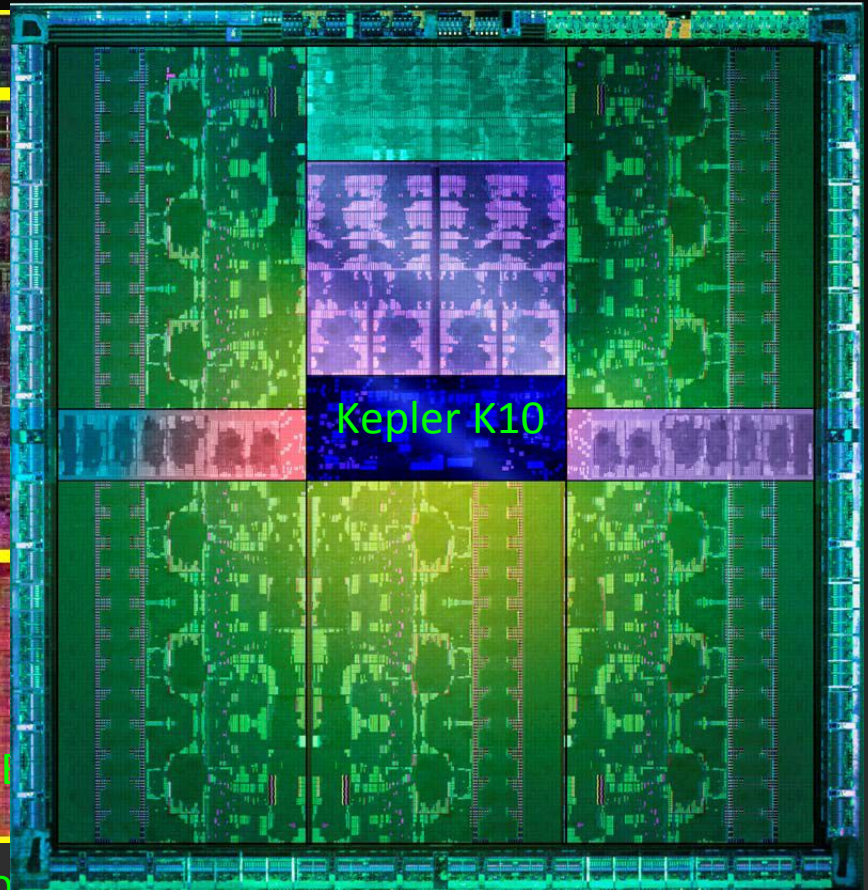
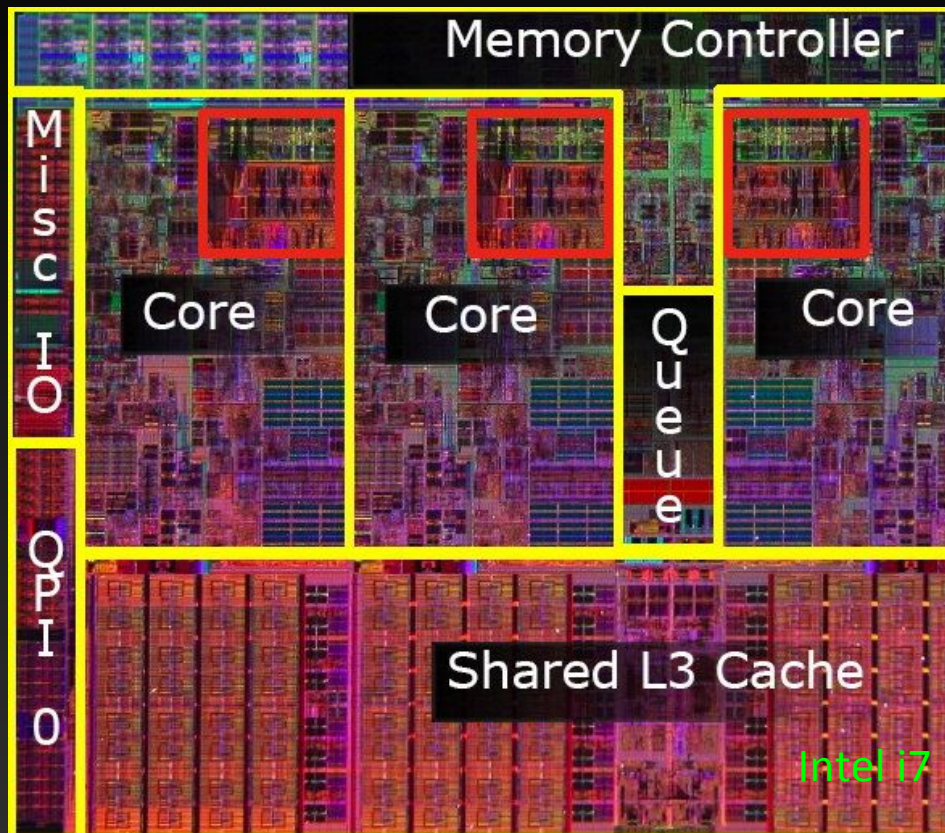
GPU x CPU



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



Principais conceitos de CUDA

Device: a GPU

Host: a CPU

Kernel – Programa que vai para a GPU

Thread – Instancias do kernel

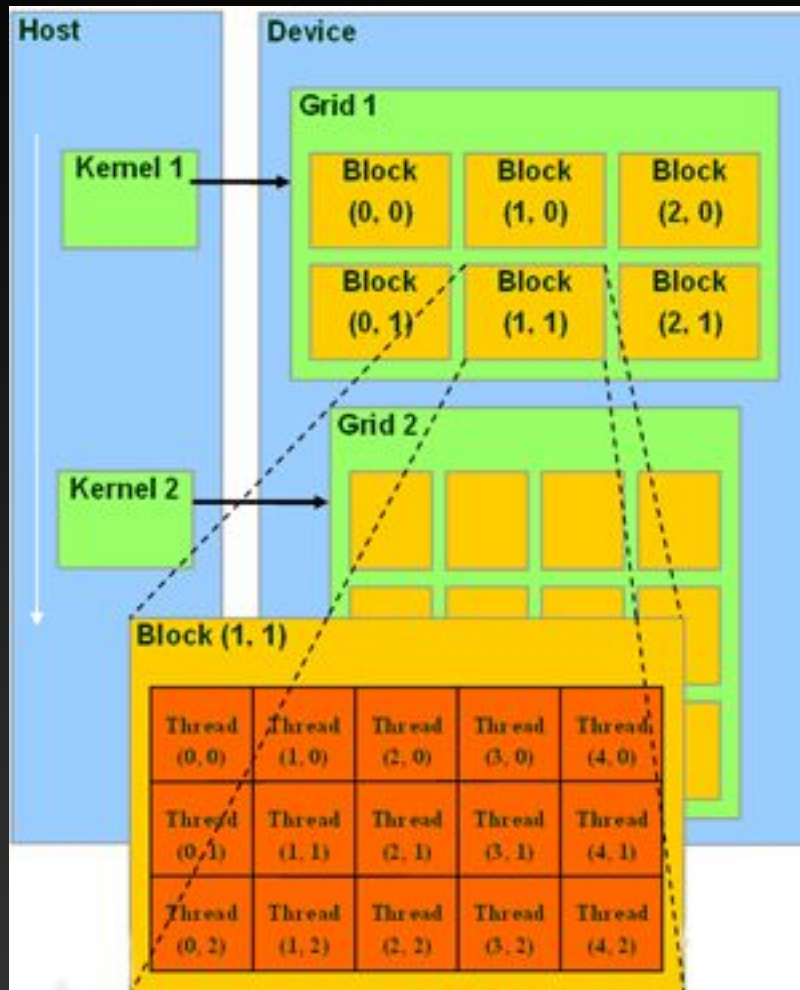
Global Memory: memória principal da GPU

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

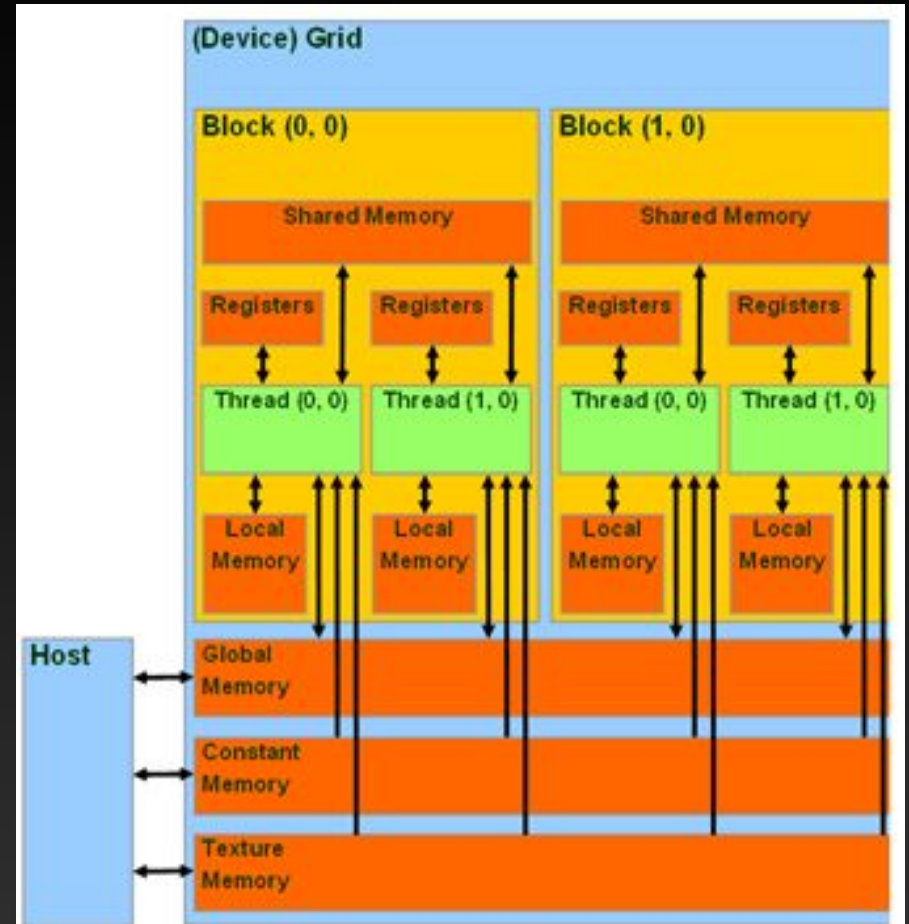


Kernel, Threads e Warps



Memórias...

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



Hello World

```
__global__ void mykernel(void) {  
}
```



GPU

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



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);
```



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!...



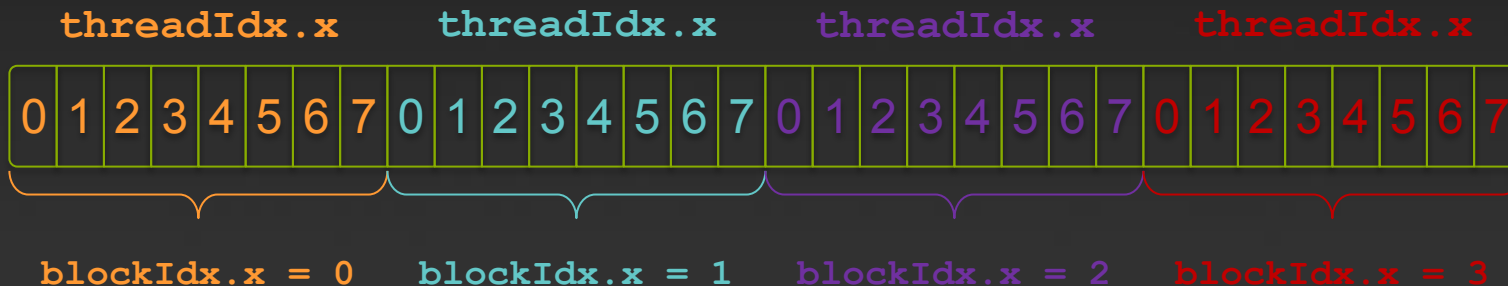
Blocos

```
__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);  
}
```



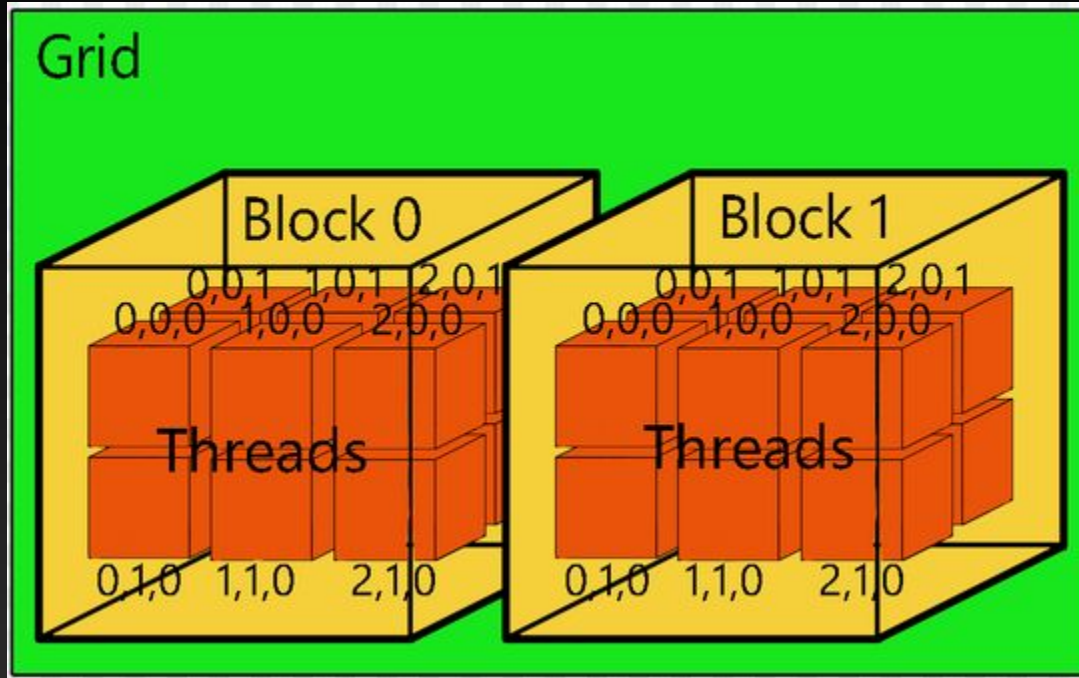
Perigo: indices não referenciados...

```
__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);  
}
```



Multiplicação de Matrizes

a ₀₀	a ₀₁	a ₀₂	a ₀₃	a ₀₄	a ₀₅
a ₁₀	a ₁₁	a ₁₂	a ₁₃	a ₁₄	a ₁₅
a ₂₀	a ₂₁	a ₂₂	a ₂₃	a ₂₄	a ₂₅

b ₀₀	b ₀₁	b ₀₂	b ₀₃
b ₁₀	b ₁₁	b ₁₂	b ₁₃
b ₂₀	b ₂₁	b ₂₂	b ₂₃
b ₃₀	b ₃₁	b ₃₂	b ₃₃
b ₄₀	b ₄₁	b ₄₂	b ₄₃
b ₅₀	b ₅₁	b ₅₂	b ₅₃

c ₀₀	c ₀₁	c ₀₂	c ₀₃
c ₁₀	c ₁₁	c ₁₂	c ₁₃
c ₂₀	c ₂₁	c ₂₂	c ₂₃

tid ₀₀	tid ₀₁	tid ₀₂	tid ₀₃
tid ₁₀	tid ₁₁	tid ₁₂	tid ₁₃
tid ₁₀	tid ₁₁	tid ₁₂	tid ₁₃

bid₀₀

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



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 occupancy rate

Coalescence is strong 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 concurrentes

```
1 cudaMalloc ( &dA, sizeA ) ;  
2 cudaMalloc ( &dB, sizeB ) ;  
3 ...  
4 cudaMemcpy ( dA, A, size, H2D ) ;  
5 cudaMemcpy ( dB, B, size, H2D ) ;  
6 ...  
7 kernelA <<< gridA, blockA>>> ( ..., dA, ... ) ;  
8 kernelB <<< gridB, blockB>>> ( ..., dB, ... ) ;  
9 ...
```



Shared Memory

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



Analizando 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)

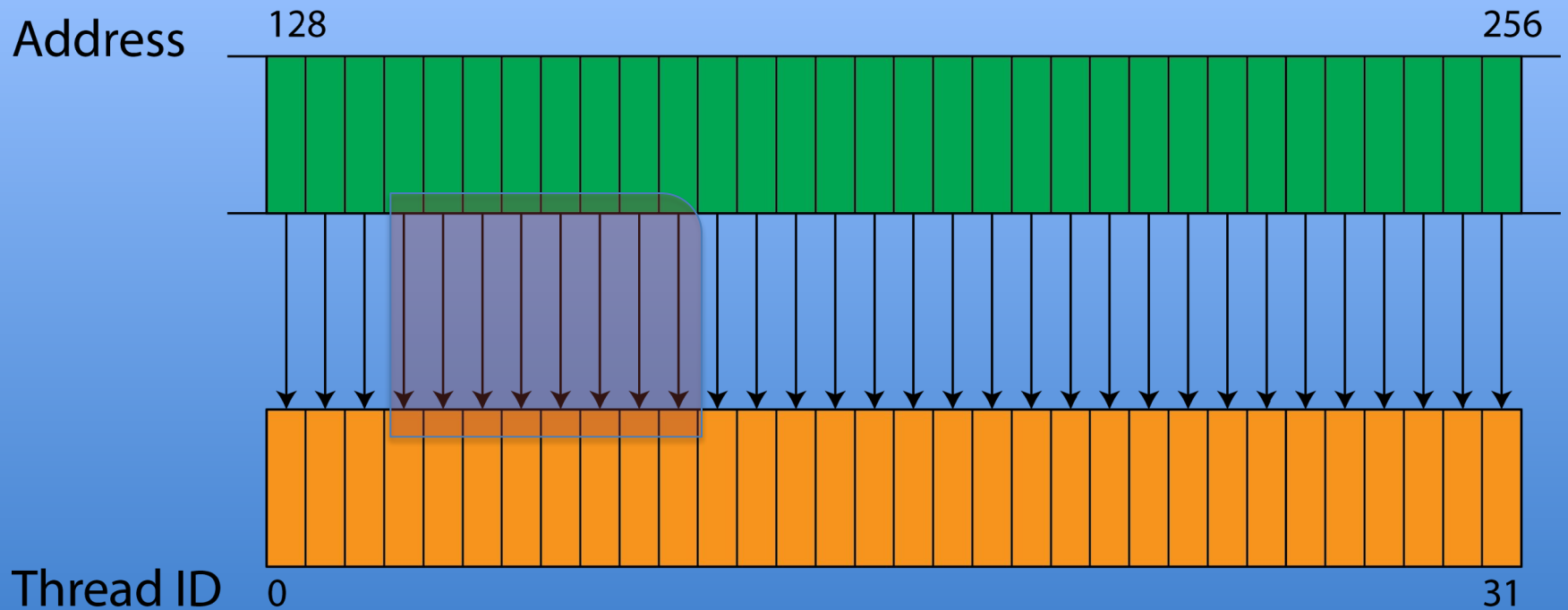
```
__global__ void reduceShared (float *d_In, *d_Out)
{
    external __shared__ s_data[];
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    int tid = threadIdx.x;

    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.



Exemplo de Coalescence

Array of Structures

```
1 struct st_particle{
2     float3 p;
3     float3 v;
4     float3 a;
5 };
6
7 __global__ void K_Particle_01(st_particle *vet){
8
9     int i = blockDim.x * blockIdx.x + threadIdx.x;
10    vet[i].p.x = vet[i].p.x + vet[i].v.x * vet[i].a.x;
11    vet[i].p.y = vet[i].p.y + vet[i].v.y * vet[i].a.y;
12    vet[i].p.z = vet[i].p.z + vet[i].v.z * vet[i].a.z;
13
14 }
```

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.



Exemplo de Coalescencia

```
1 __global__ void K_Particle_02(float *vet_px, float *vet_py, float *vet_pz,  
2                               float *vet_vx, float *vet_vy, float *vet_vz,  
3                               float *vet_ax, float *vet_ay, float *vet_az){  
4  
5     int i = blockDim.x * blockIdx.x + threadIdx.x;  
6     vet_px[i] = vet_px[i] + vet_vx[i] * vet_ax[i];  
7     vet_py[i] = vet_py[i] + vet_vy[i] * vet_ay[i];  
8     vet_pz[i] = vet_pz[i] + vet_vz[i] * vet_az[i];  
9  
10  
11 }  
12
```

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 architectures



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 always be resident at physical memory)

Constraint: doing so the memory may run out much faster than when using `malloc`...



Streams

Knowing the physical address 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 stream operation 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 stream operation 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 stream operation only starts
// after the previous stream operation have finished

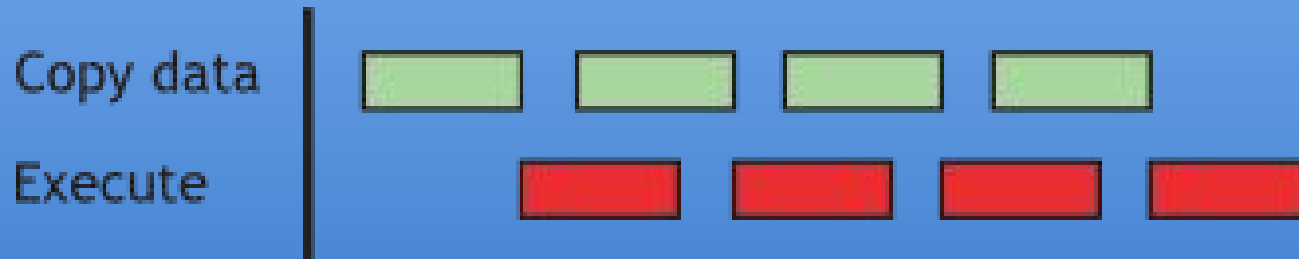
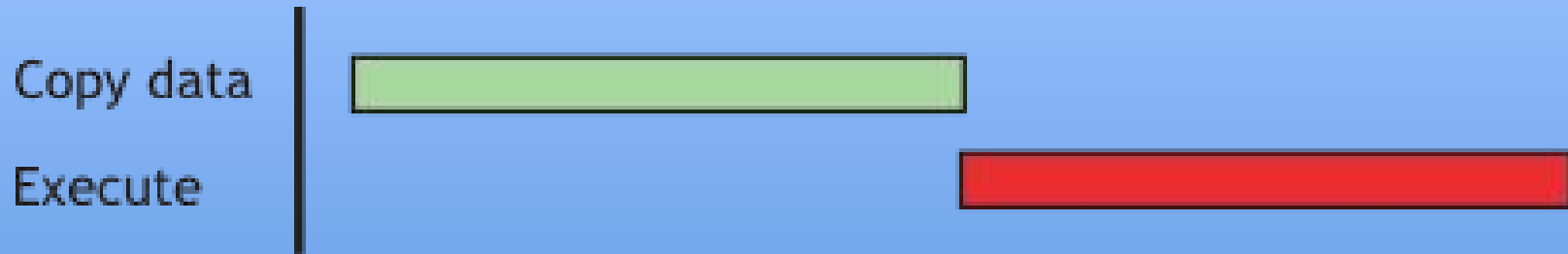
Kernel <<<GridDim, BlockDim, stream>>> (dev_a);

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=0; 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=0; 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...

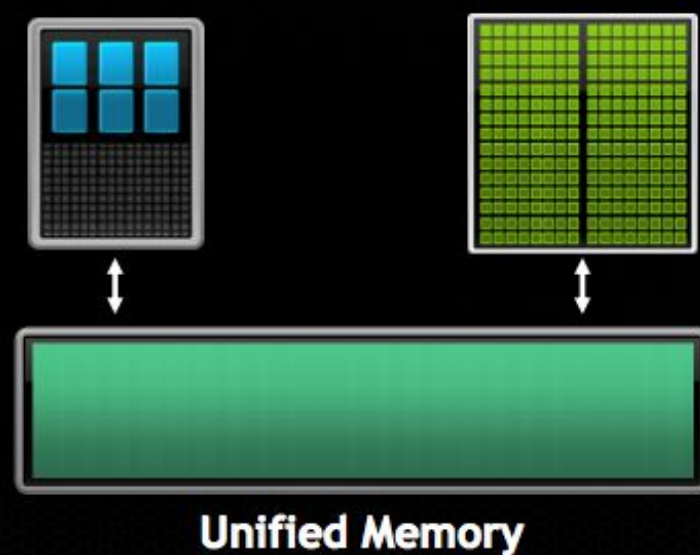
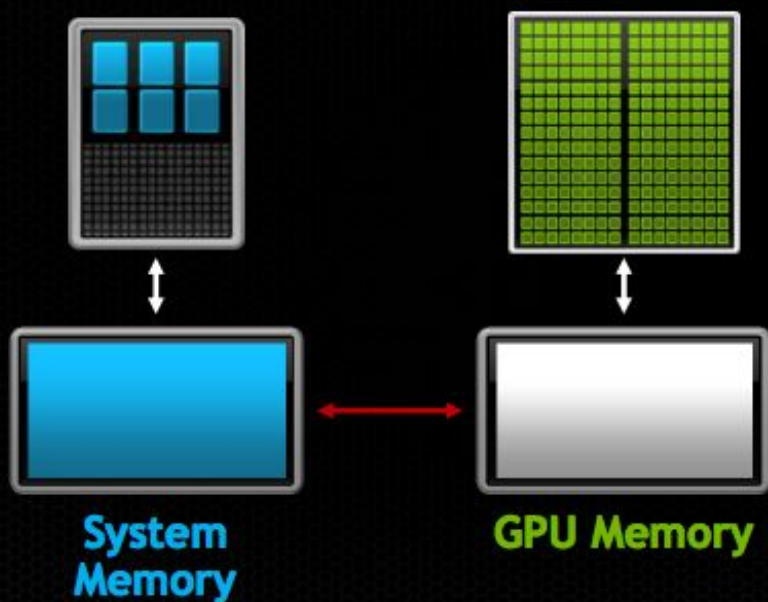
- Find ways to parallelize sequential code,
- Minimize data transfers between the host and the device,
- Adjust kernel launch configuration to maximize device utilization,
- Ensure global memory accesses are coalesced,
- Minimize redundant accesses to global memory whenever possible,
- Avoid different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

Read more at:

<http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#ixzz3jGmjoXLj>



Memória Unificada / CUDA 8 / PASCAL



Exemplo de memória unificada

```
void foo() {  
  
    // Allocate 64 GB  
    char *data;  
    size_t size = 64*1024*1024*1024;  
    cudaMallocManaged(&data, size);  
}
```

- Allocating more memory than available at the GPU...



Exemplo de memória unificada

```
__global__ void mykernel(char *data) {  
    data[1] = 'g';  
}  
  
void foo() {  
    char *data;  
    cudaMallocManaged(&data, 2);  
  
    mykernel<<<...>>>(data);  
    // no synchronize here  
    data[0] = 'c';  
  
    cudaFree(data);  
}
```

Both CPU and GPU
accessing the data



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.”

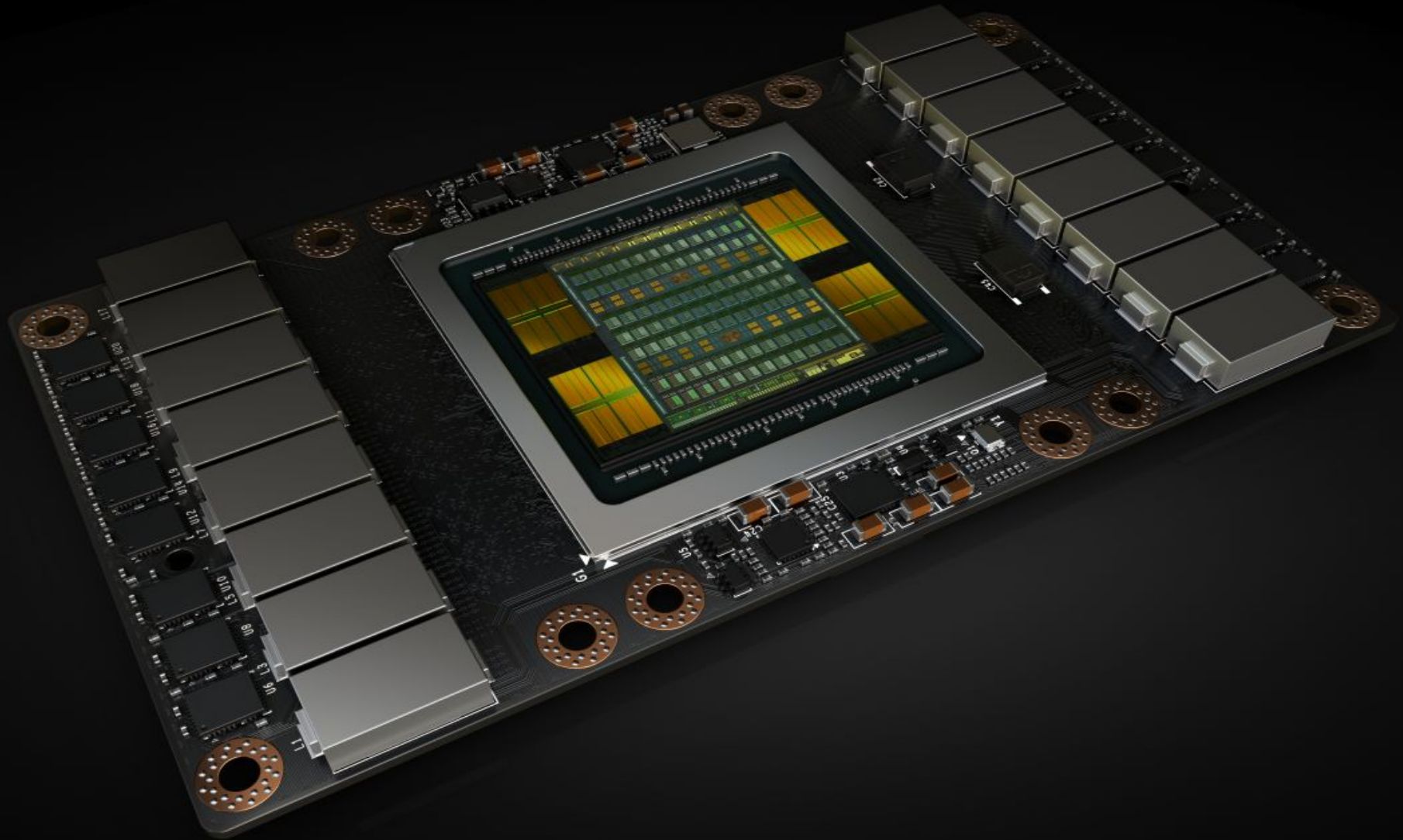
P100: 21.2 Tflops for Half precision

half a, b ...



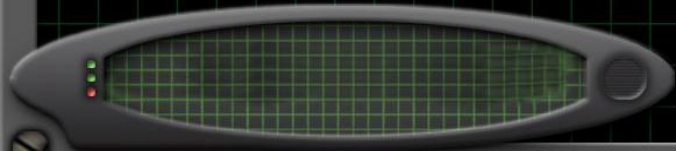
Volta

The most advanced accelerator ever built



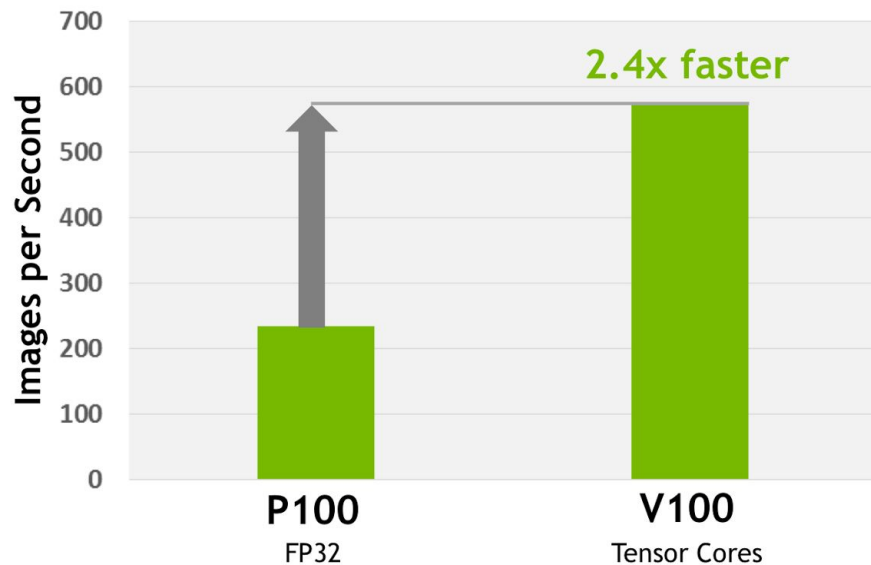
Capacity

- *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.*



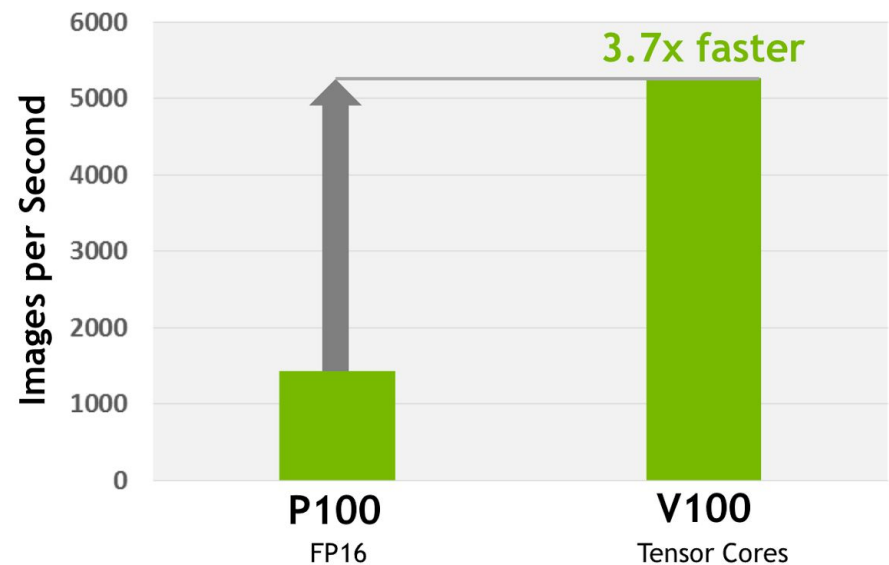
Architecture dedicated to Deep Learning

ResNet-50 Training



ResNet-50 Inference

TensorRT - 7ms Latency

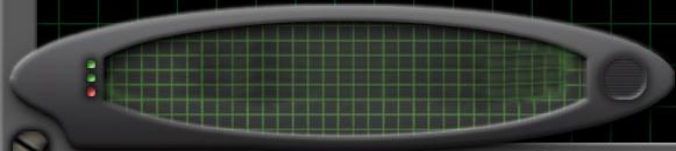


Volta Architecture



New Streaming Multiprocessor

- *Optimized for deep learning*
- *Tensor Cores (up to 12x for training)*
- *Independent thread scheduling capacity*
- *New combined L1 Data Cache and Shared Memory*



Faster Memory

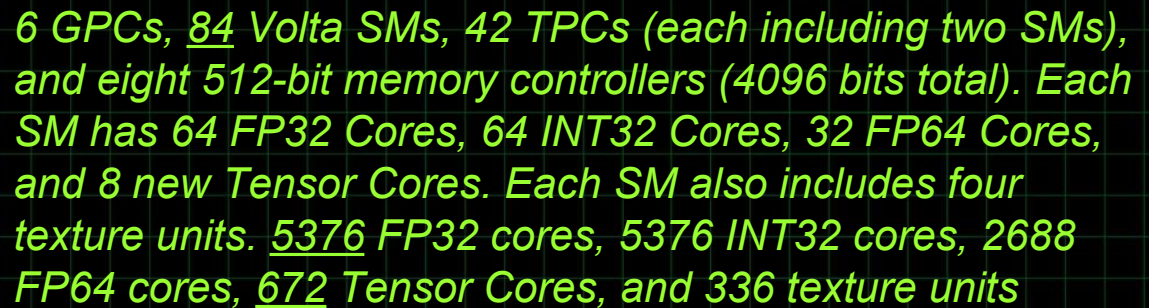
900 GB/sec peak bandwidth
NVLink 2.0



Cooperative Groups

Groups of Communicating threads





Overview

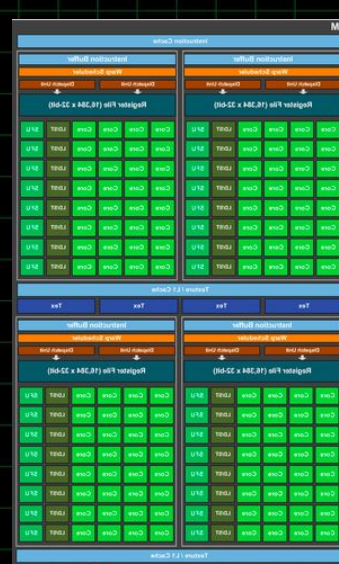
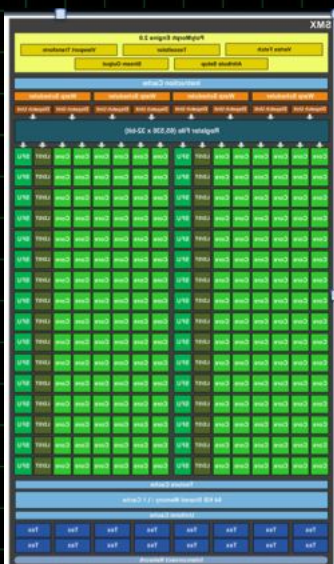
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	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

The image displays four architectural diagrams of NVIDIA's Volta architecture, showing the internal components and data flow of different processing units:

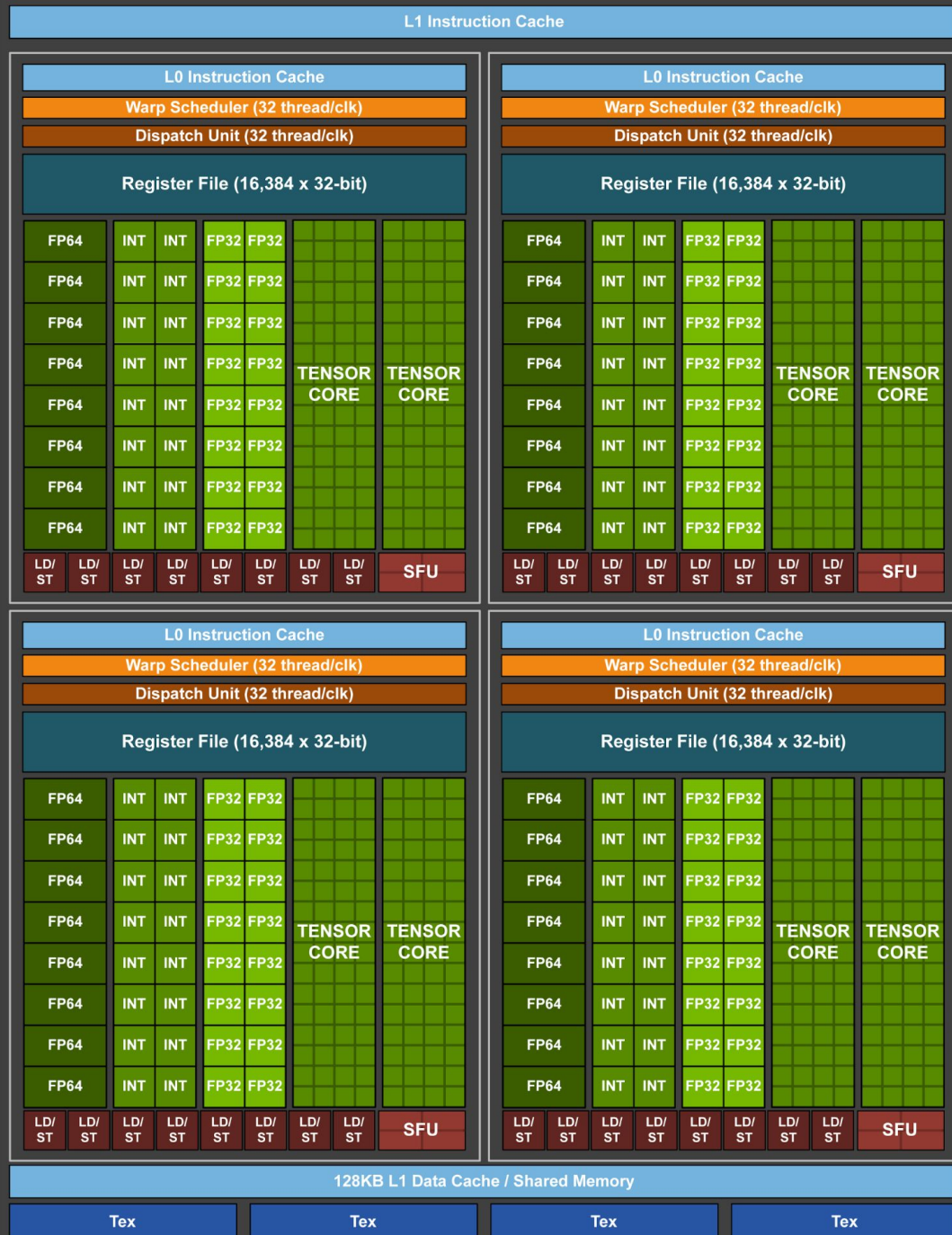
- MB (Memory Block):** Shows a memory controller at the top, followed by a Register File (32,768 x 32-bit), a large array of cores (16x16 grid), and a Memory Controller at the bottom.
- MM2 (Memory Management Unit):** Shows a memory controller at the top, followed by a Register File (32,768 x 32-bit), a large array of cores (16x16 grid), and a Memory Controller at the bottom.
- M2 (Memory Management Unit):** Shows a memory controller at the top, followed by a Register File (32,768 x 32-bit), a large array of cores (16x16 grid), and a Memory Controller at the bottom.
- SM (Streaming Multiprocessor):** Shows a memory controller at the top, followed by a Register File (32,768 x 32-bit), a large array of cores (16x16 grid), and a Memory Controller at the bottom.

Each diagram illustrates the internal components and data flow of the respective unit, including memory controllers, register files, and core arrays.

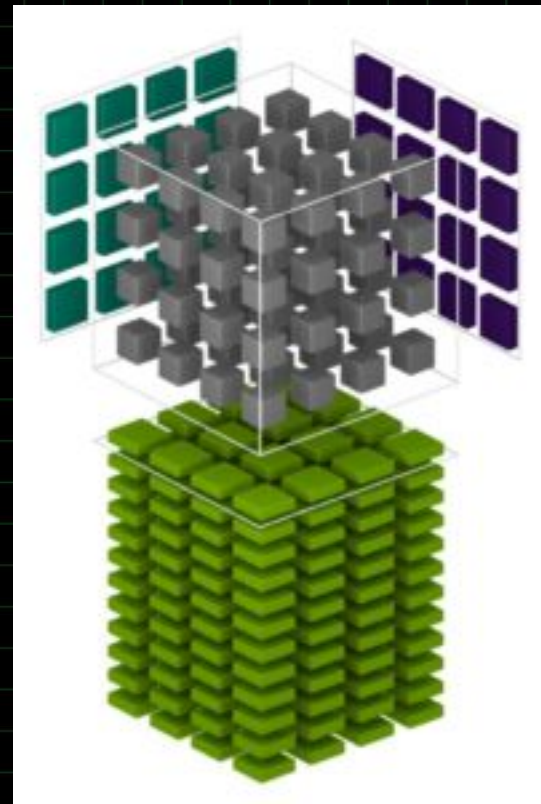


Volta SM

SM



Tensor Cores



Tensor Cores

$$(FP16/FP32) D = (FP16) A \times B + C \quad (4 \times 4 \times 4)$$

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

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

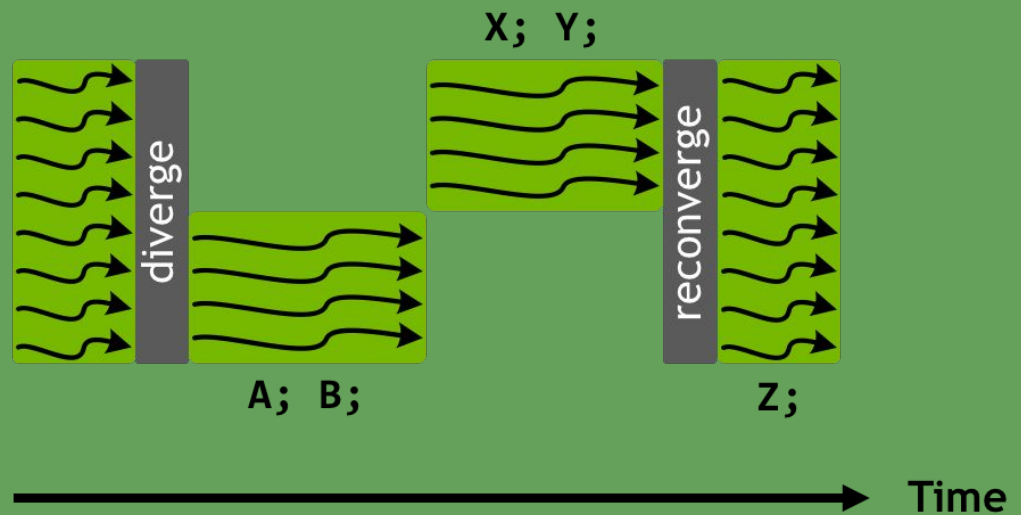
FP16 or FP32 FP16 FP16 FP16 or FP32

8 TC per SM → 1024 FP per clock per SM

New SIMT model

Until Pascal: 32 threads per warp in SIMT scheme

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

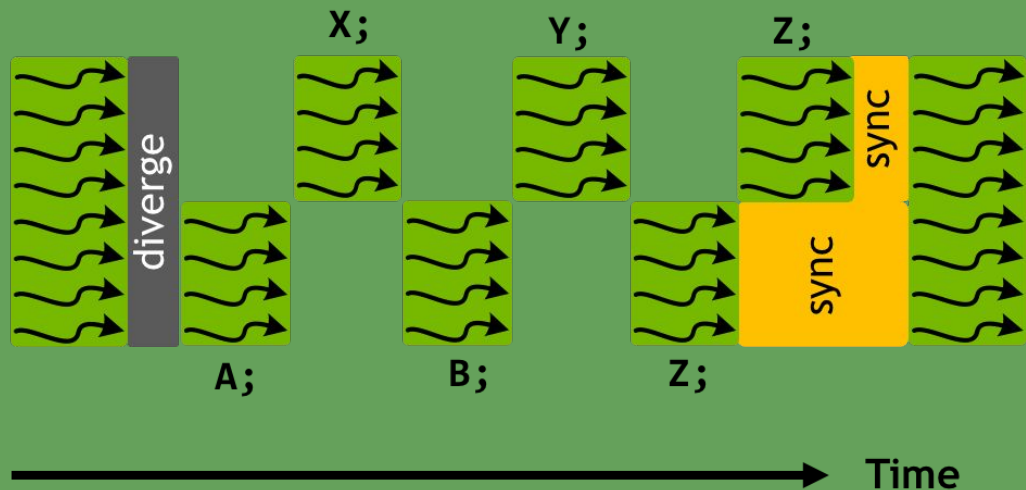


There is no control in the thread level sync at the divergence, in the same warp

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

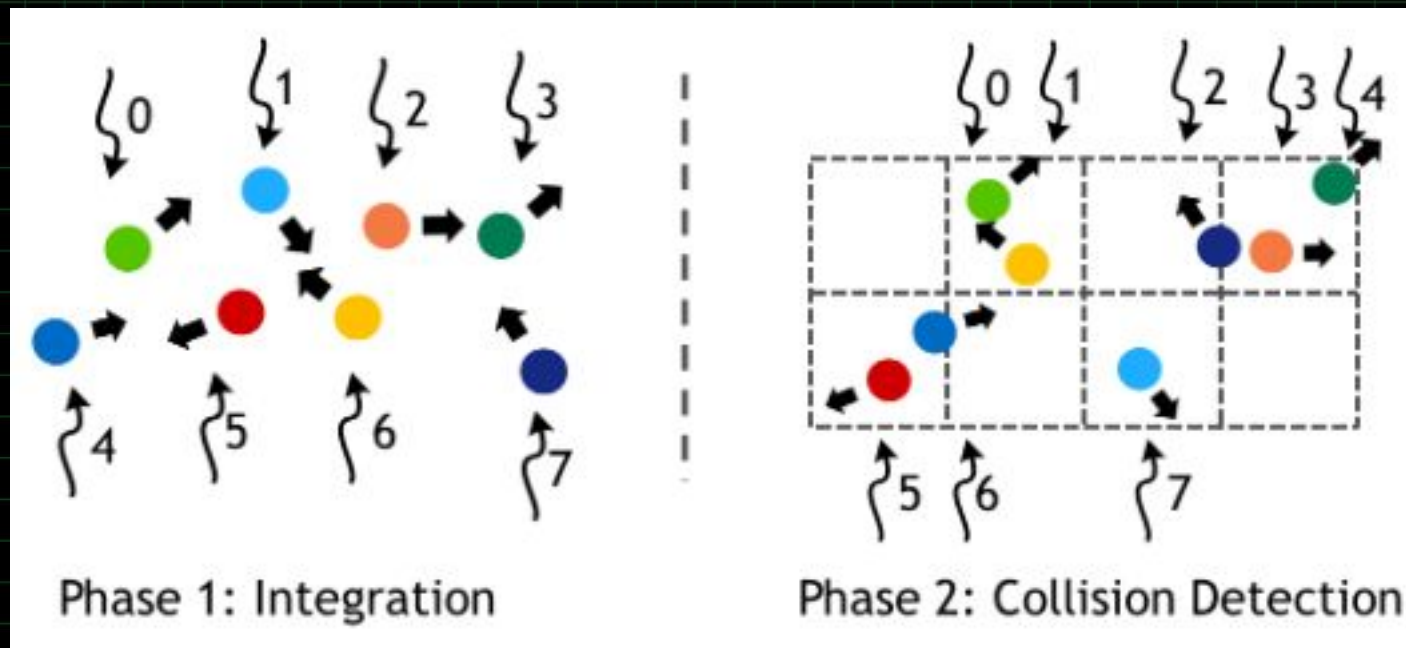
```
__global__ void cooperative_kernel(...)
{

    // 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 size
    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);  
}
```

GPU Educational Kit

UFF NVIDIA CENTER OF EXCELLENCE

Follow us for daily updates:  
  

[Home](#) [Learn GPU Computing](#) [Research](#) [Papers](#) [People](#) [Downloads](#) [Contact US](#) [Posts](#)

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.

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/>

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



esteban@ic.uff.br

