Outline

GPU for HPC

Hardware Evolution and milestones

GPU x CPU Architecture

Development
- -Kernel
- - Execution and memory hierarchy
  - -threads and blocks
  - -Global memory
- - Warps and Coalesced i/o

Finite Diference Method
- -one dimension wave equation
  - -global and shared memory examples
- -Maxwell Equations, two dimension Yee method
  - -global, shared and texture examples

What is CUDA

Is a extension of C language that enable the use of GPU for general computation !

Why CUDA

- Easy to use
- Chip hardware
- Free to use
- Broaded adopted

Installing

http://developer.nvidia.com/cuda-toolkit-40

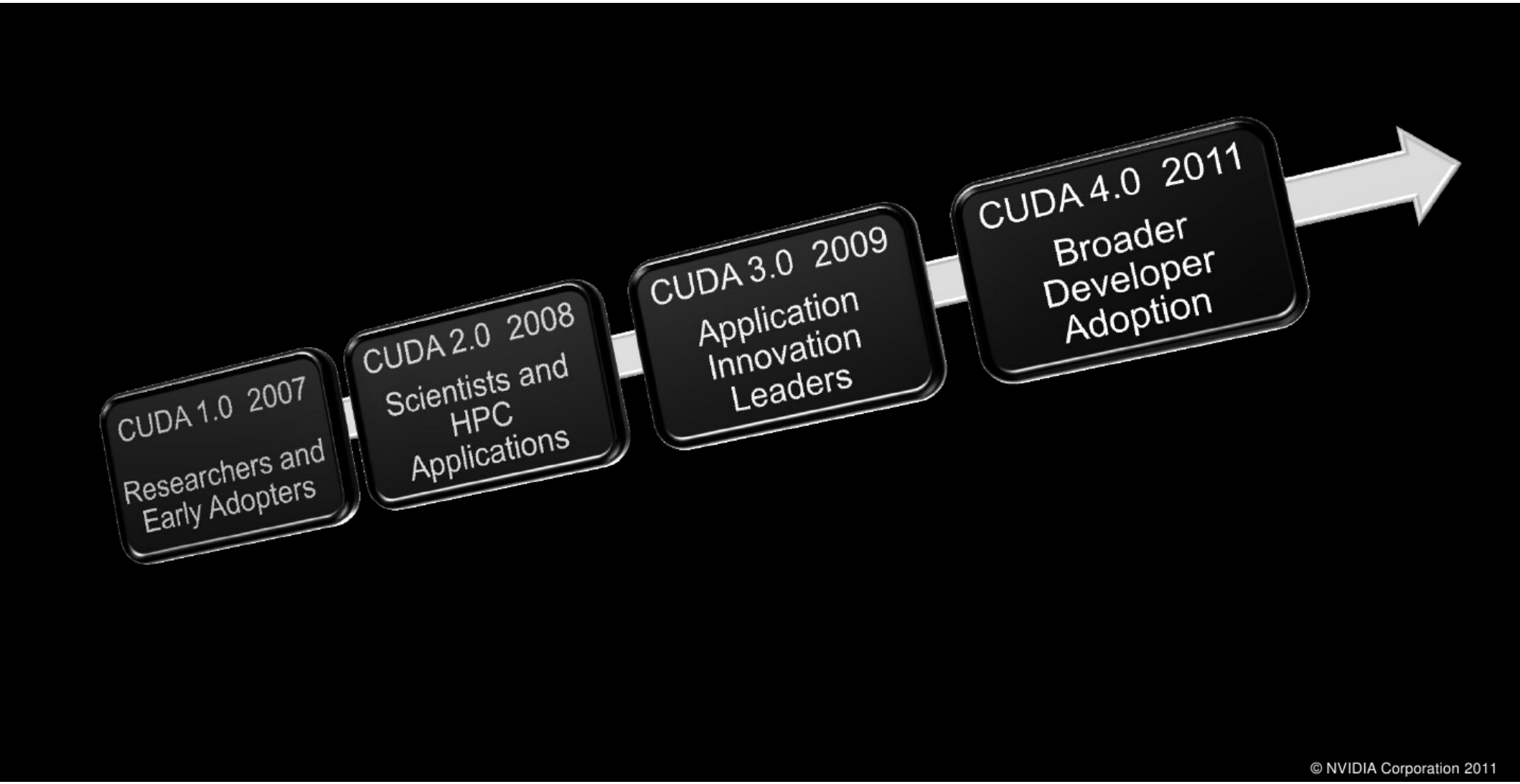
We need:

-Module: Developer Drivers for Linux (270.41.19)

-Toolkit: Cuda Toolkit for your distribution

Compiler: NVCC (same syntax as GCC)

Adoption time line



Adoption

SEARCH RESULTS

You searched for: GPU

You refined by:

Publication Year: 2007 - 2011

Set Search Alert | Download Citations | Email Selected Results | Print

Results per Page 25 Showing 1 - 25 of 2.395 results

Next»

YOU'RE A CREATIVE GENIUS. SO LET YOUR IDEAS FLOW

Unlock the power of Adobe® Creative Suite® 5.5 software with NVIDIA® GPUs and get blazing-fast performance and smooth, fluid interactivity.

If you're a creative artist, designer, or video professional, you can accelerate your full post production workflow and infuse your project with creative inspiration with NVIDIA® Quadro® graphics solutions.

ADOBE® PREMIERE® PRO CS5.5

Get an amazingly fluid, real-time video editing experience with the Adobe® Mercury Playback Engine, based on NVIDIA® CUDA™ technology, featured in the newly released Adobe® Premiere® Pro CS5.5.

> Learn More

ADOBE® AFTER EFFECTS® CS5.5

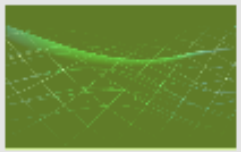
Create and design using real-time, NVIDIA GPU-accelerated visual effects in Adobe® After Effects CS5.5.

> Learn More

ADOBE® PHOTOSHOP® CS5

Now more than ever, you can connect with the digital canvas in a natural way. Experience a new level of interactivity while navigating through large images, retouching photos, or experimenting with effects.

> Learn More



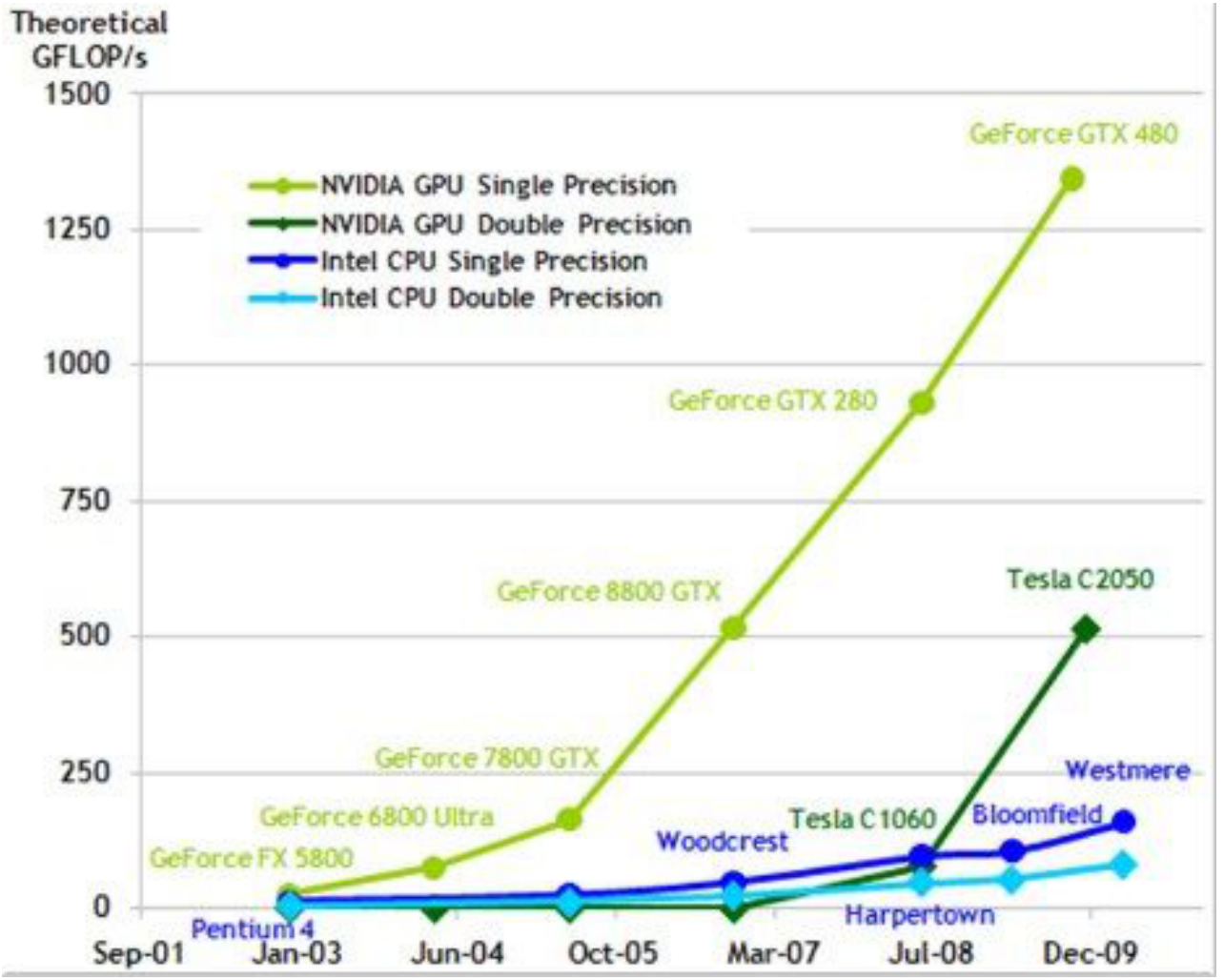
Numerical Analysis Tools

- MATLAB™
- Mathematica
- LabView
- Jacket® for MATLAB
- More..

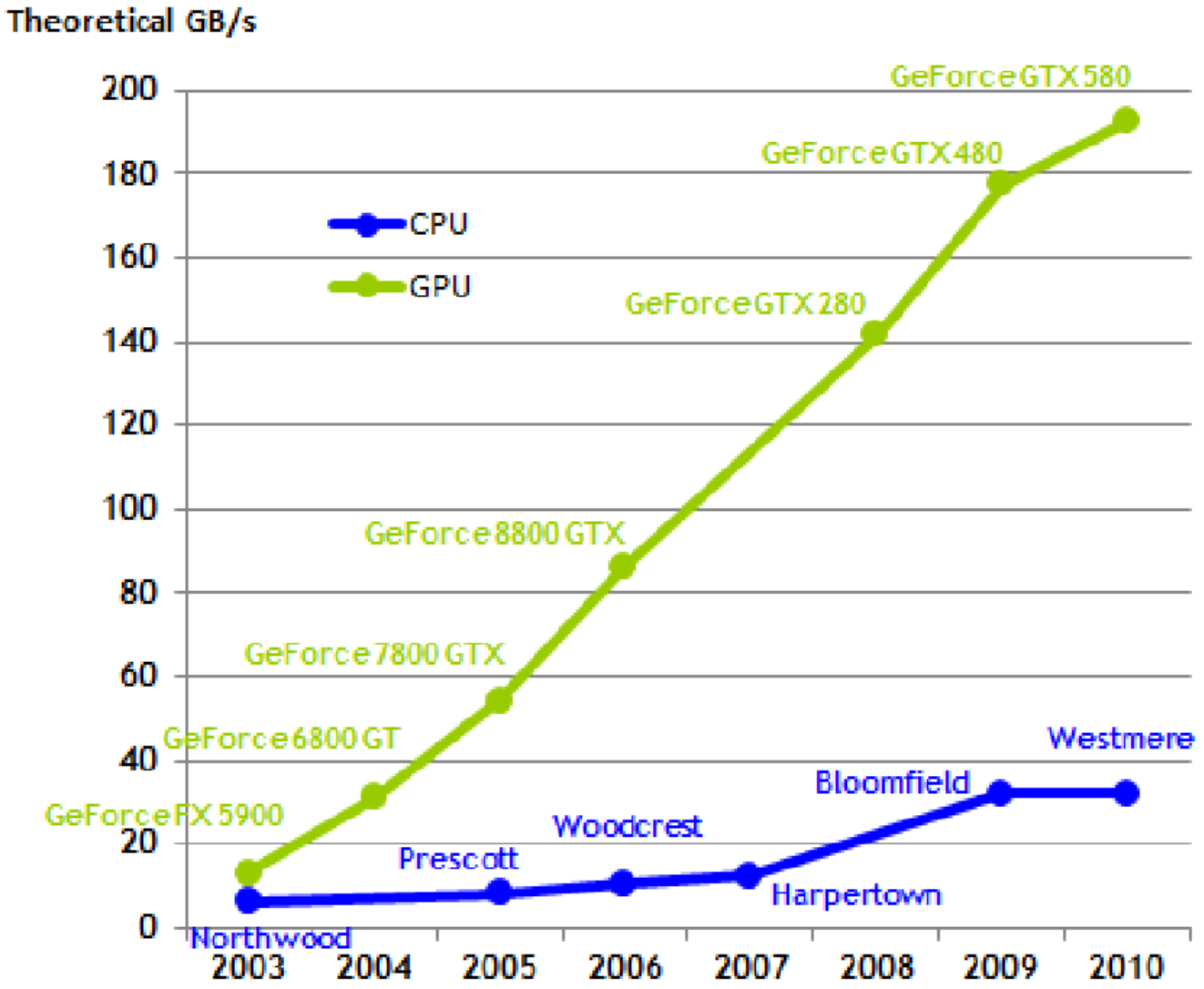
GPU Accelerated Libraries

- NVIDIA cuFFT
- NVIDIA cuBLAS
- NVIDIA Performance Primitives
- Thrust
- More..

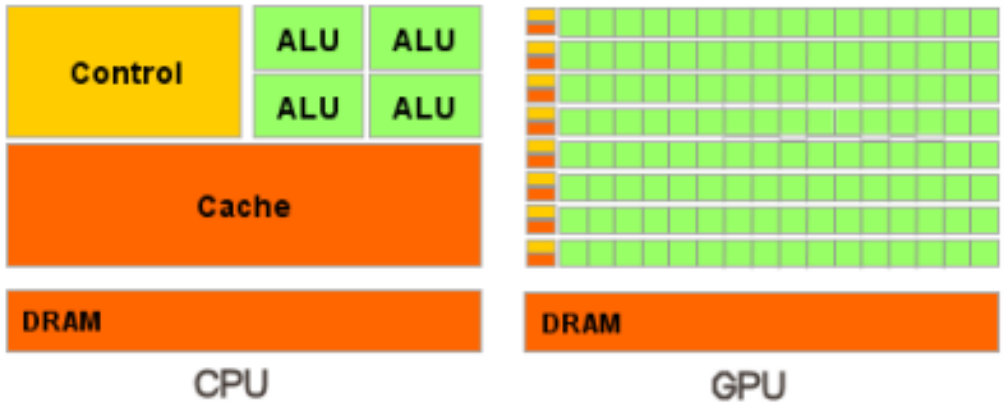
GFLOPs Evolution GPUxCPU

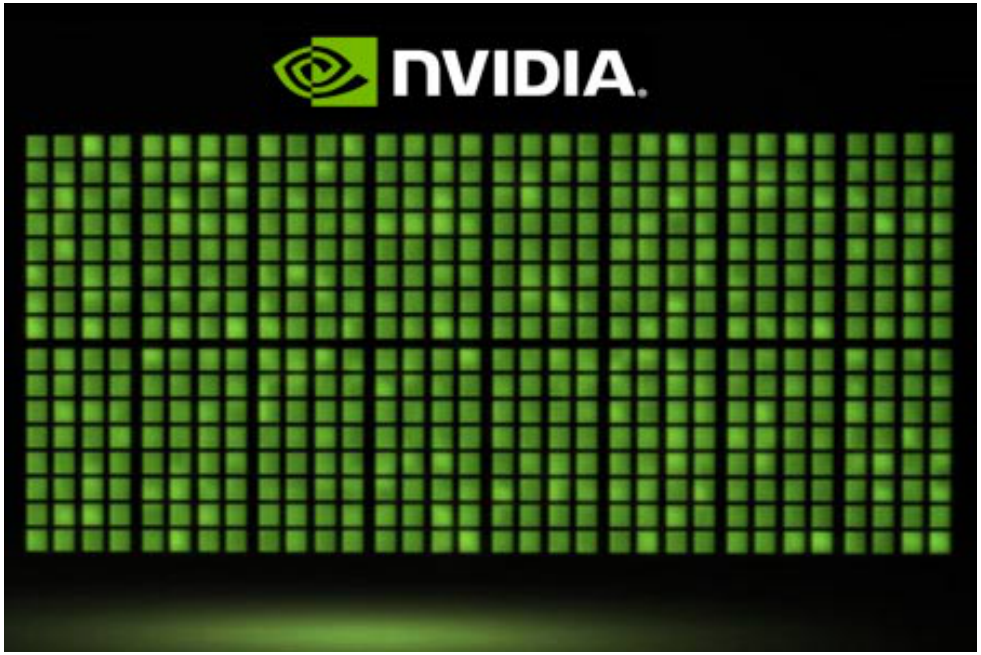


Bandwidth Evolution GPUxCPU



Schematic Architecture CPU x GPU

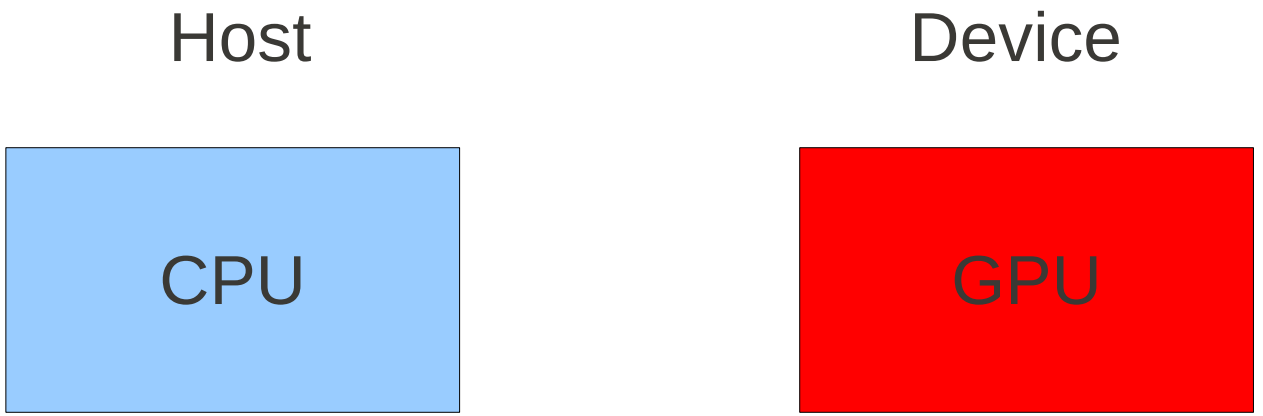




CPUs: great area dedicated to control
GPUs: great area dedicated to ALU(Aritmetic Logical Unit)

UNICAMP | FEEC | DMO

How to develop



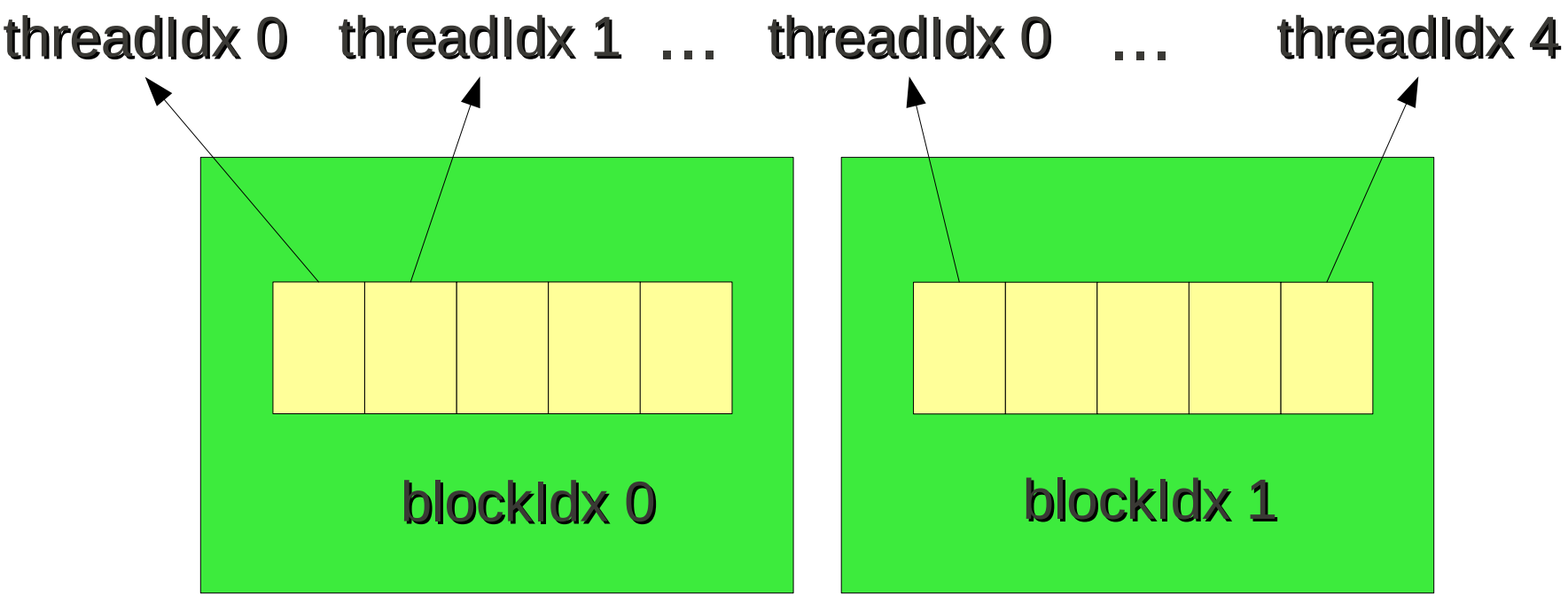
Alloc data on Host and Device:

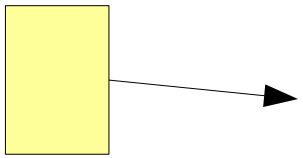
*cudaMalloc((void**)&A_d, memsize);*
*A_h = (float *)malloc(memsize);*

Copy data from Host and Device:

cudaMemcpy(A_d, A_h, memsize, cudaMemcpyHostToDevice);
cudaMemcpy(A_h, A_d, memsize, cudaMemcpyDeviceToHost);
cudaMemcpy(B_d, A_d, memsize, cudaMemcpyDeviceToDevice);

Threads and Blocks build-in variables





Execute one kernel instance and has a build-in index that points to data that should be updated !

Thread index = blockIdx.x*blockDim.x + threadIdx.x

Threads forming a Block can be syncronized !

Kernel

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
...
// Kernel invocation with N threads
VecAdd<<<nBlocks, BlockSize>>>(A, B, C);
```

Calling a Kernel

Main function – Host code:

```
kernel <<< nBlocks, BlockSize >>> (arguments);
```

nBlocks → number of blocks
BlockSize → number of threads for each Block

Example:

```
int BlockSize=4;
int nBlocks = N / BlockSize + (N % BlockSize == 0?0:1);
```

nBlocks defined at run time!

Example: Increment Array Elements

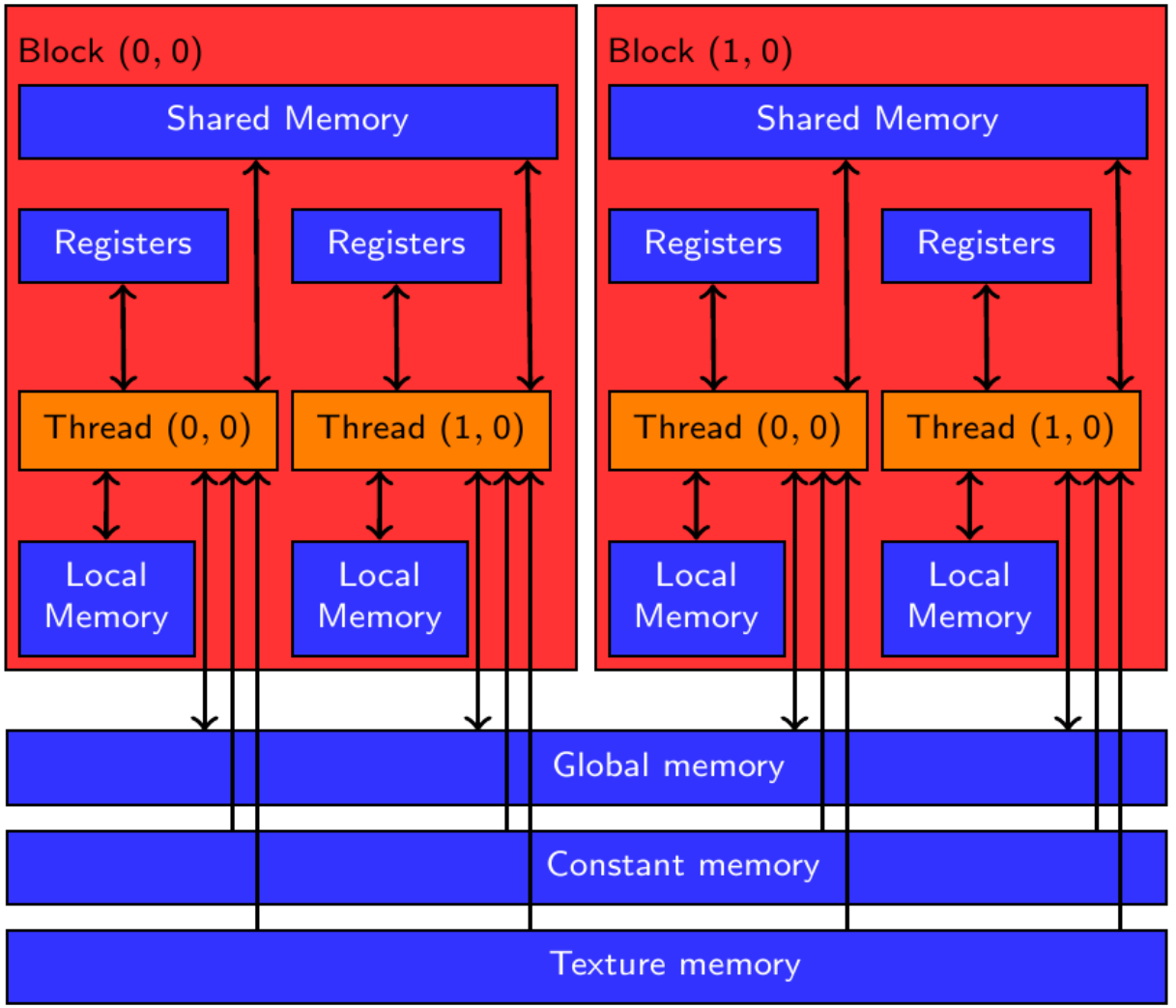
CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize)  );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Memory Hierarchy

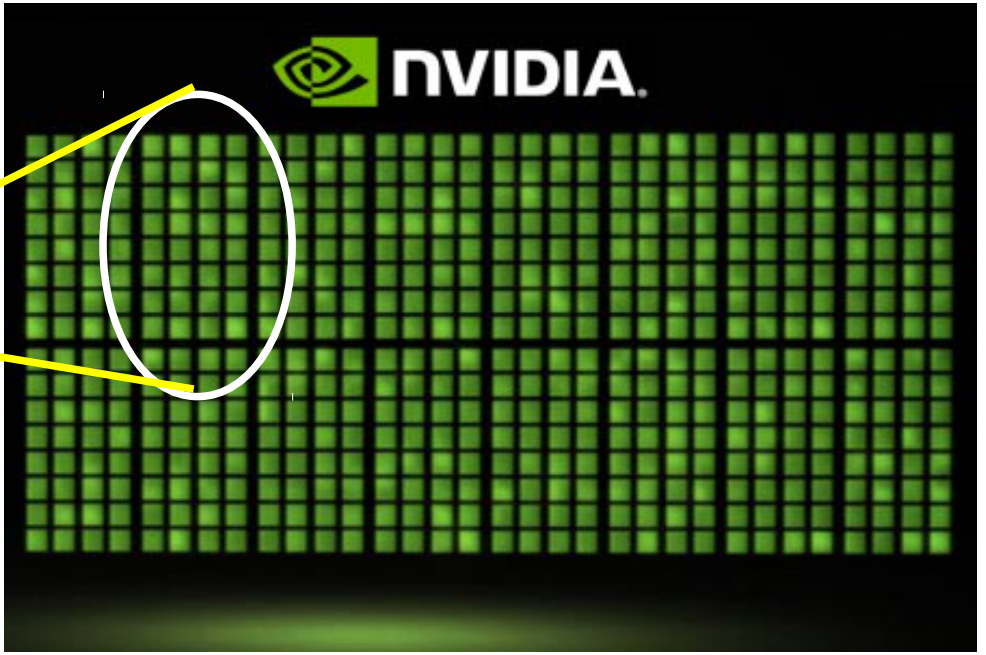


Memory Hierarchy

Registers	Per thread	Read-Write	
Local memory	Per thread	Read-Write	
Shared memory	Per block	Read-Write	For sharing data within a block
Global memory	Per grid	Read-Write	Not cached
Constant memory	Per grid	Read-only	Cached
Texture memory	Per grid	Read-only	Spatially cached

Hardware view

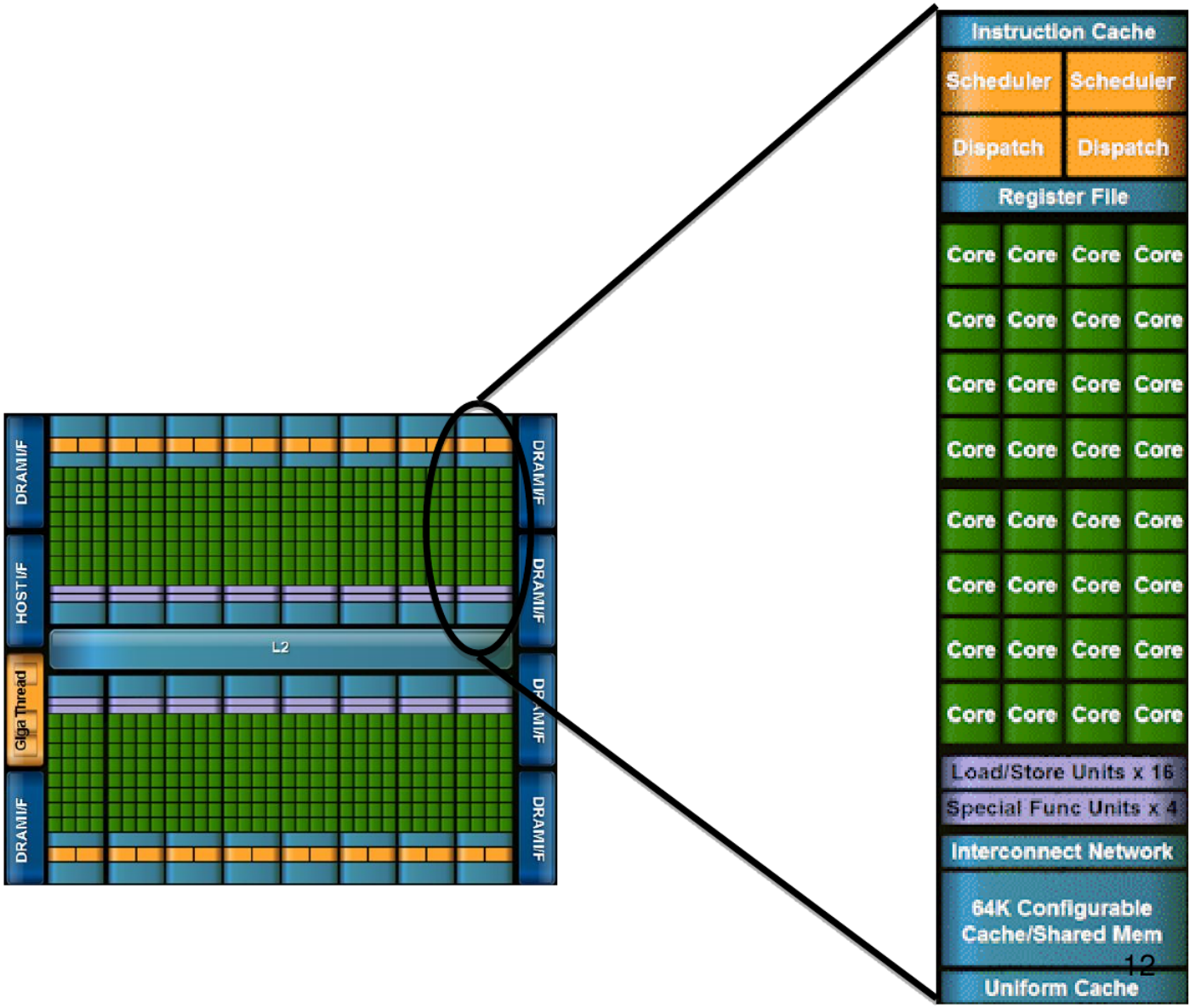
Multiprocessor



Each block is divided in 32 threads called Warp and queued to be executed in SIMD model on a Multiprocessor.

To warranty execute all threads in parallel in a warp the memory reads and writes must be **coalesced!**

Hardware view



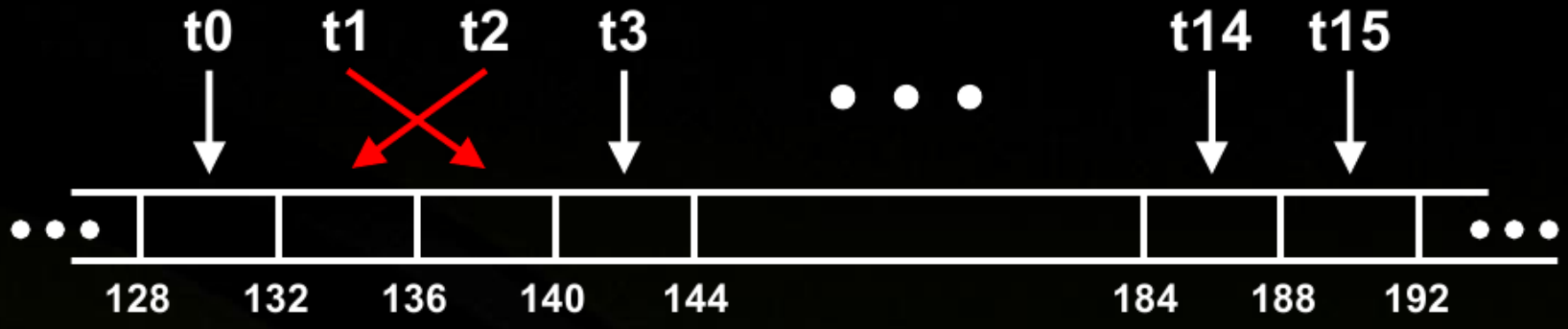
Coalesced Global Memory Accesses

Coalesced `float` memory access

Coalesced `float` memory access

(divergent warp)

Non-Coalesced Global Memory Accesses



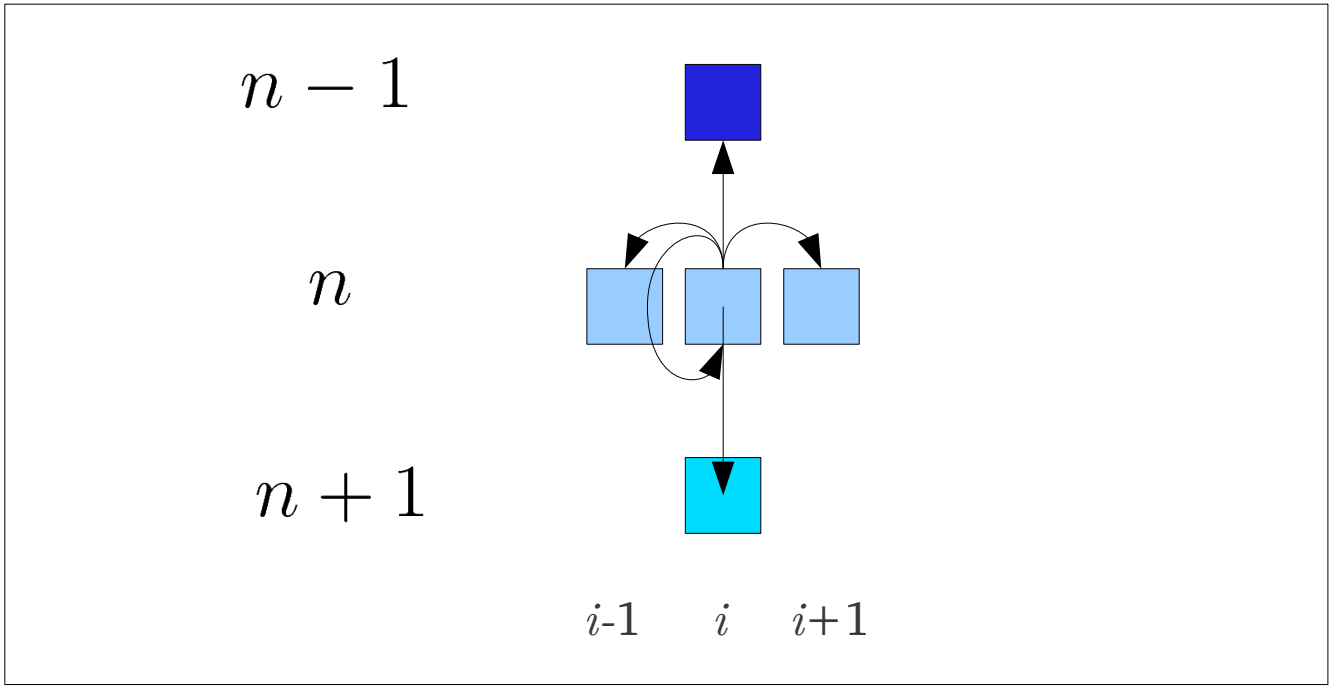
Non-sequential `float` memory access

Misaligned starting address

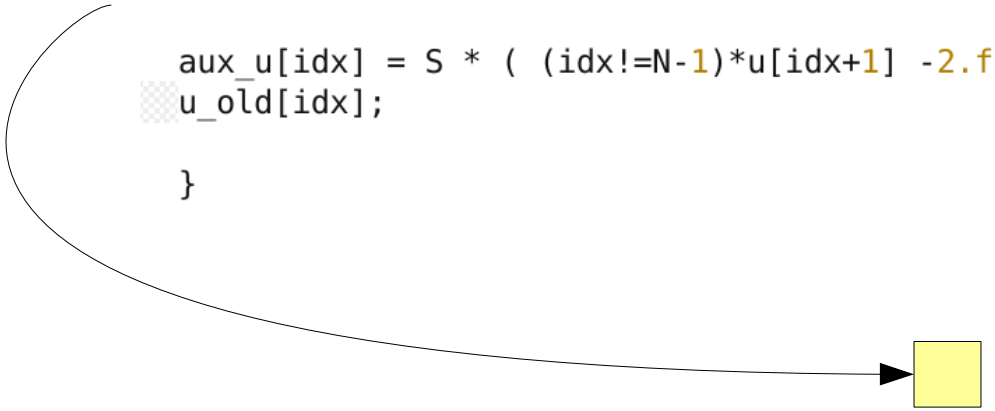
One-dimension FDTD Method

Wave equation solution via FDTD:

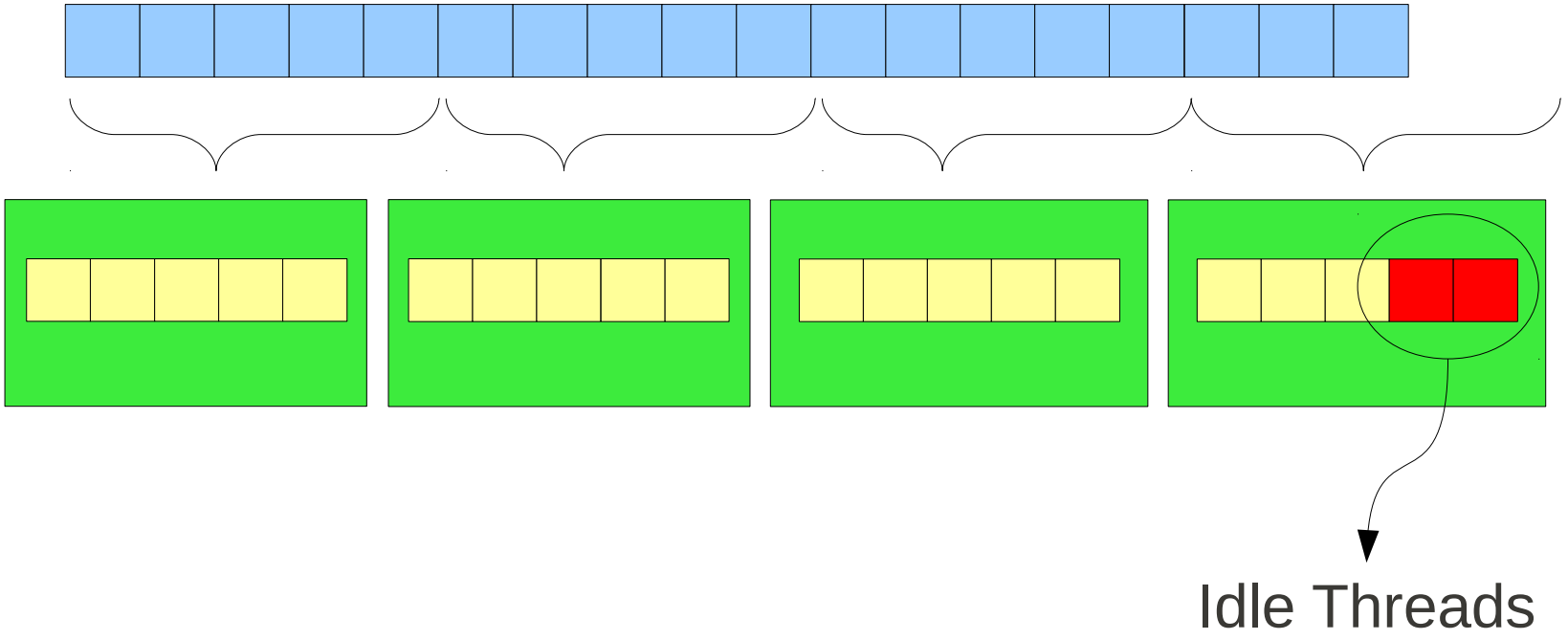
$$u_i^{n+1} = S\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + 2u_i^n - u_i^{n-1}$$



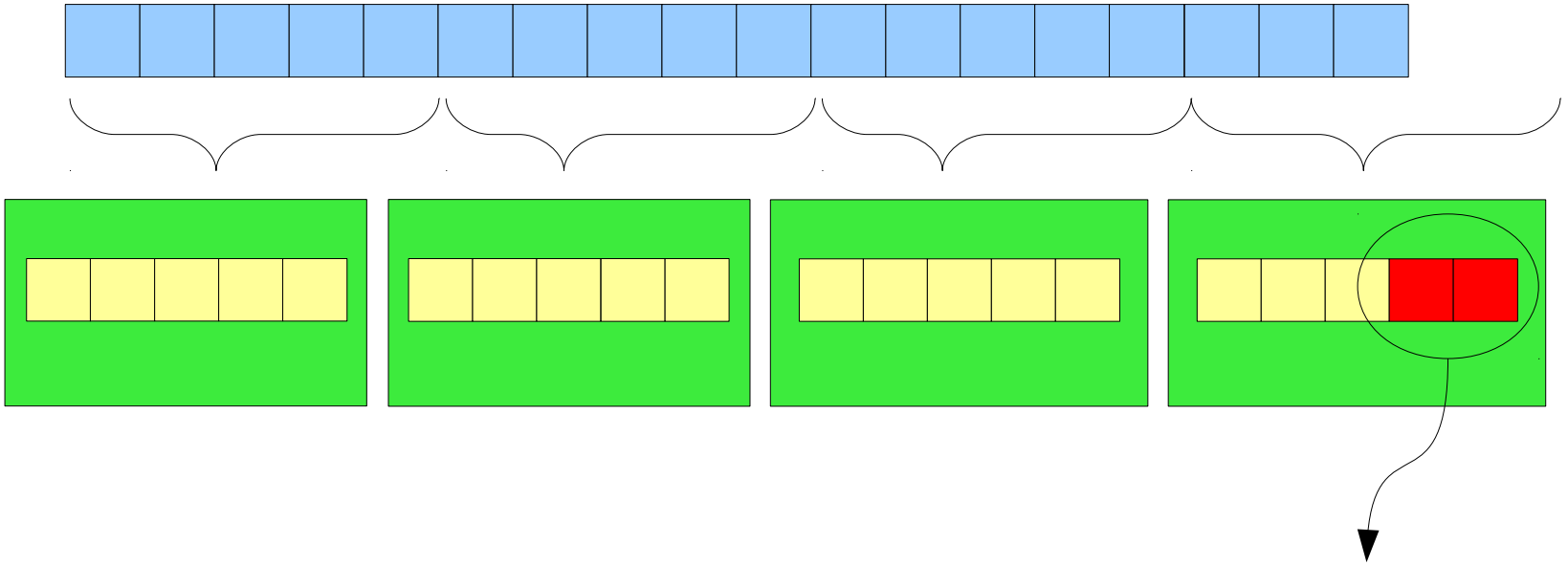
```
__global__ void WaveStep(float *u, float *u_old, float *aux_u, int N, float S, int t)
  {

  int idx = blockIdx.x*blockDim.x + threadIdx.x; // Global index


  aux_u[idx] = S * ( (idx!=N-1)*u[idx+1] -2.f * u[idx] + (idx!=0)*u[idx-1] ) + 2.f * u[idx] -
  u_old[idx];

  }
```



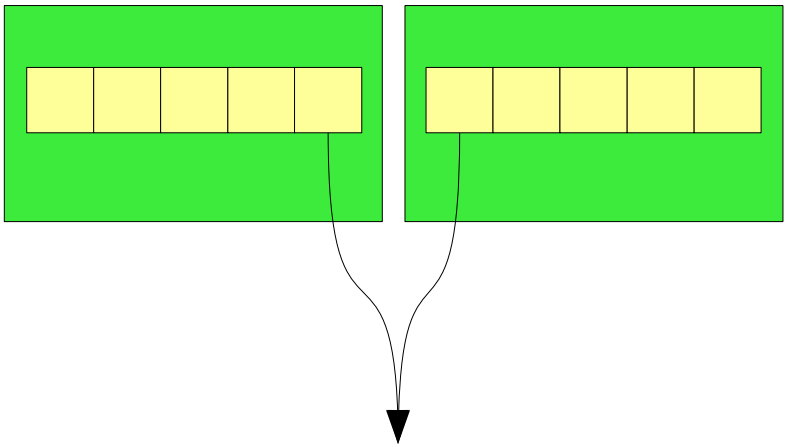
```
int blockSize=4; // block size. Must be taked carefully, so we can discover the bet value
int nBlocks = N/blockSize + (N%blockSize == 0?0:1); // minimal number of block for take the necessary
number of threads. ONE THREAD FOR EACH u ARRAY(ROPE) ELEMENT
```



```
int blockSize=4; // block size. Must be taked carefully, so we can discover the bet value
int nBlocks = N/blockSize + (N%blockSize == 0?0:1); // minimal number of block for take the necessary
number of threads. ONE THREAD FOR EACH u ARRAY(ROPE) ELEMENT
```



Syncronization !

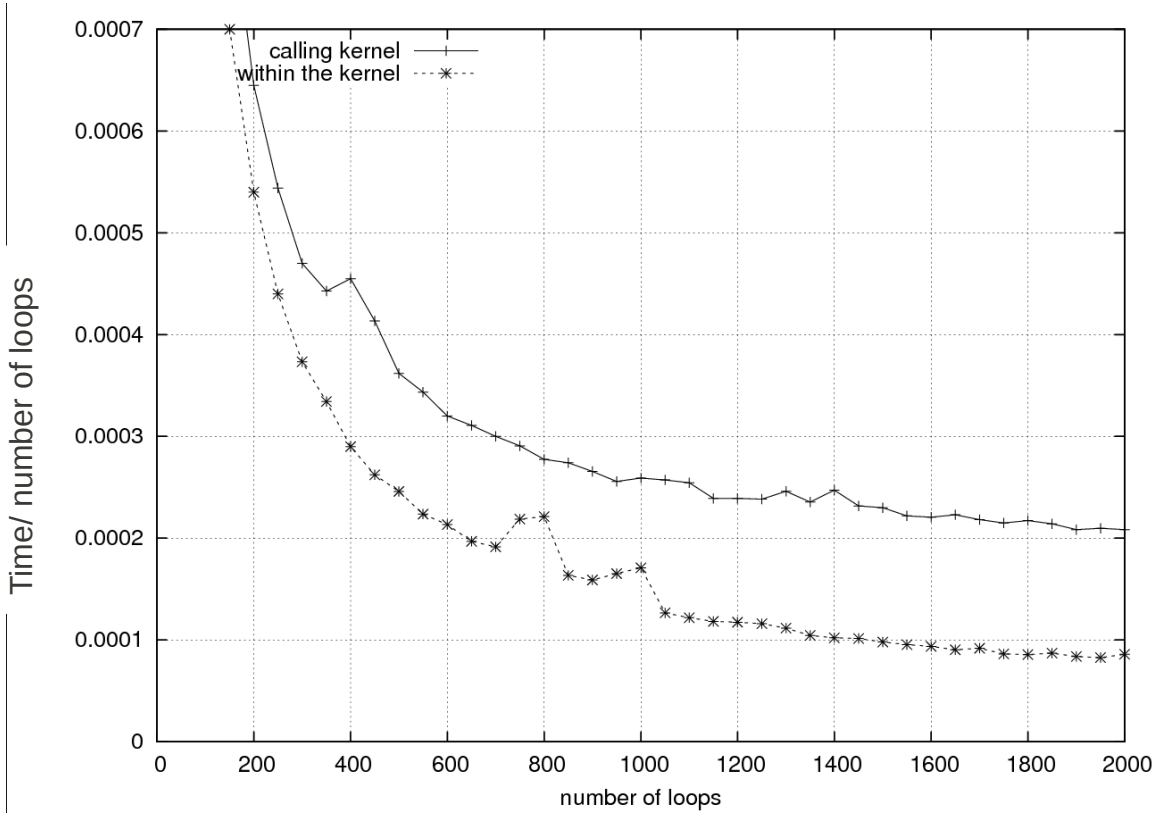


How to Sync this two Threads ?

You have to re-run the kernel !

```
__global__ void WaveStep(float *u, float *u_old, float *aux_u, int N, float S, int t)
  {

  int idx = blockIdx.x*blockDim.x + threadIdx.x; // Global index


  aux_u[idx] = S * ( (idx!=N-1)*u[idx+1] -2.f * u[idx] + (idx!=0)*u[idx-1] ) + 2.f * u[idx] -
  u_old[idx];

  }
```

Basic Example: Sum two arrays



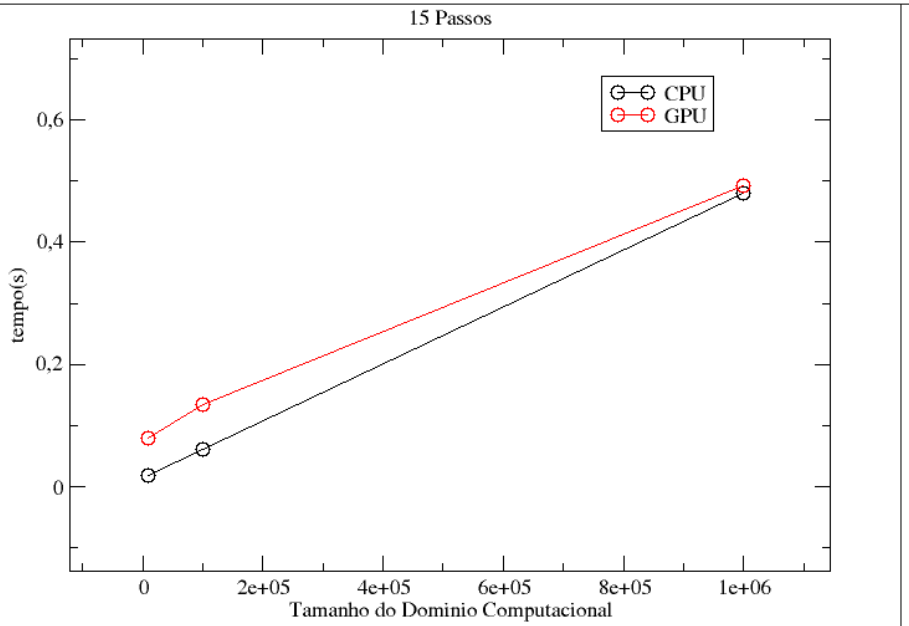
When adding two array of 10MB GPU expende half of the time for call the kernel !

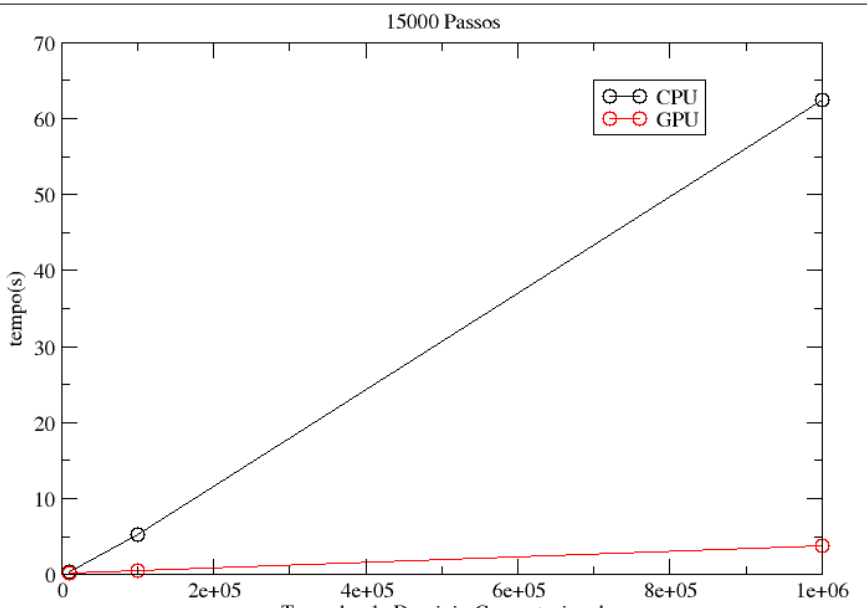
```
__global__ void WaveStep(float *u, float *u_old, float *aux_u, int N, float S, int t)
  {

  int idx = blockIdx.x*blockDim.x + threadIdx.x; // Global index


  aux_u[idx] = S * ( (idx!=N-1)*u[idx+1] -2.f * u[idx] + (idx!=0)*u[idx-1] ) + 2.f * u[idx] -
u_old[idx];

  }
```

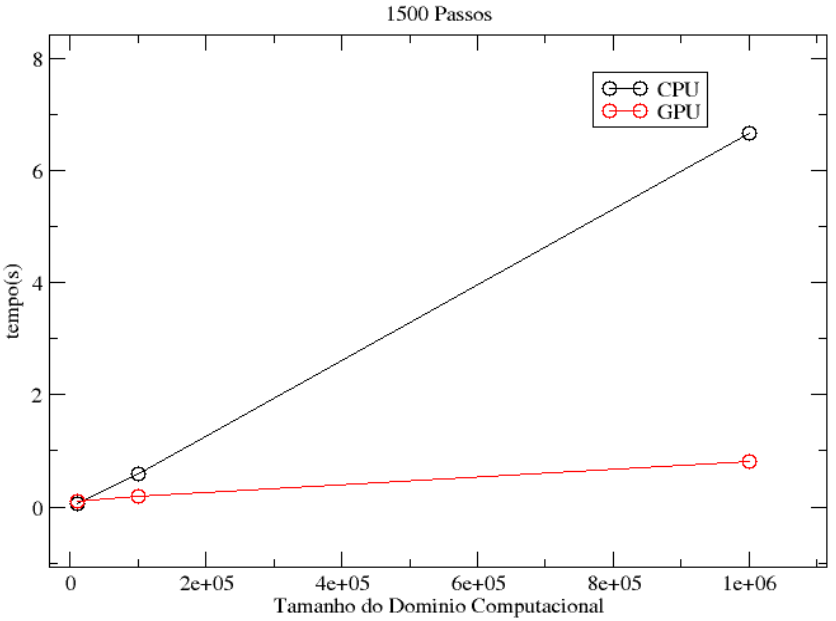
main()

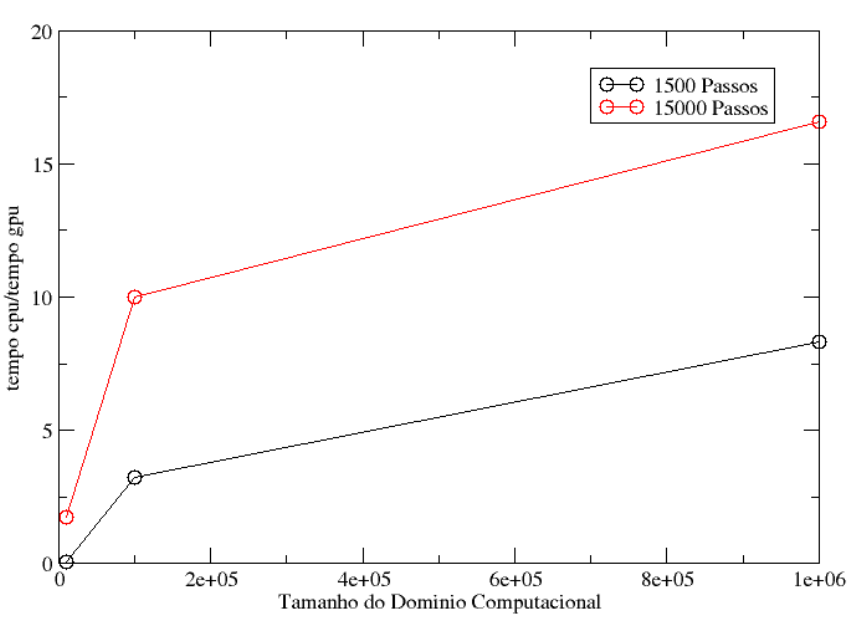
```
 for( int t=0; t < T+2;t=t+1) //time iteration
   {
   WaveStep <<< nBlocks, blockSize >>> (u_d, u_old_d, aux_u, N, S,t);
   checkCUDAError("kernel invocation");
   cudaMemcpy(u_old_d, u_d, size, cudaMemcpyDeviceToDevice);
   cudaMemcpy(u_d, aux_u, size, cudaMemcpyDeviceToDevice);
```

Resultados


15 Passos
CPU
GPU
tempo(s)
0,6
0,4
0,2
0
0
2e+05
4e+05
6e+05
8e+05
1e+06
Tamanho do Dominio Computacional


15000 Passos
CPU
GPU
tempo(s)
70
60
50
40
30
20
10
0
0
2e+05
4e+05
6e+05
8e+05
1e+06

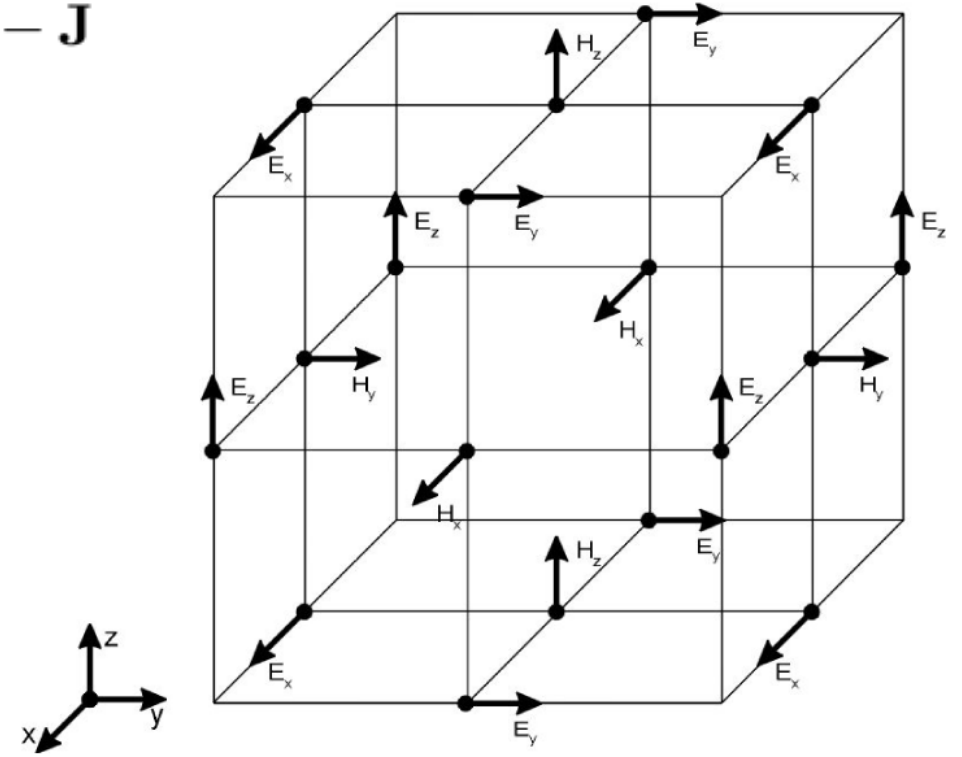

1500 Passos
CPU
GPU
tempo(s)
8
6
4
2
0
0
2e+05
4e+05
6e+05
8e+05
1e+06
Tamanho do Dominio Computacional


1500 Passos
15000 Passos
tempo cpu/tempo gpu
20
15
10
5
0
0
2e+05
4e+05
6e+05
8e+05
1e+06
Tamanho do Dominio Computacional

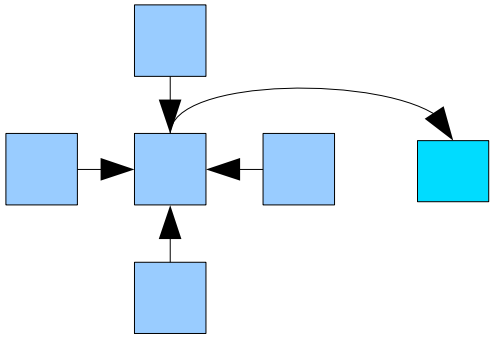
Maxwell's Equations and the Yee algorithm

Faraday's Law: $\mu \frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E}$

Ampere's Law: $\epsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H} - \mathbf{J}$



Two-dimension FDTD Method



TEz Mode

(LA)

$$E_x^n(i+1/2,j)=E_x^{n-1}(i+1/2,j)+\frac{\Delta t}{\varepsilon_0}\frac{H_z^{n-1/2}(i+1/2,j+1/2)-H_z^{n-1/2}(i+1/2,j-1/2)}{\Delta y}$$

$$E_y^n(i,j+1/2)=E_y^{n-1}(i,j+1/2)-\frac{\Delta t}{\varepsilon_0}\frac{H_z^{n-1/2}(i+1/2,j+1/2)-H_z^{n-1/2}(i-1/2,j+1/2)}{\Delta x}$$

(LF)

$$H_z^{n+1/2}(i+1/2,j+1/2)=H_z^{n-1/2}(i+1/2,j+1/2)+\frac{\Delta t}{\mu_0}\frac{E_x^n(i+1/2,j+1)-E_x^n(i+1/2,j)}{\Delta y}-\frac{\Delta t}{\mu_0}\frac{E_x^n(i+1,j+1/2)-E_x^n(i,j+1/2)}{\Delta x}$$

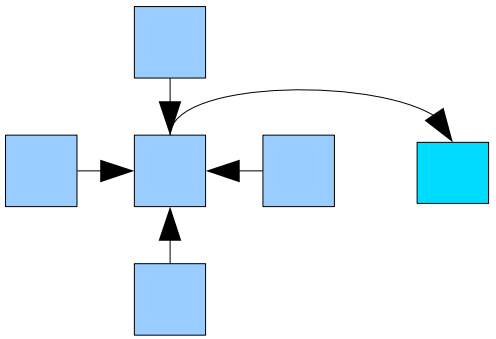
Two-dimension FDTD Method

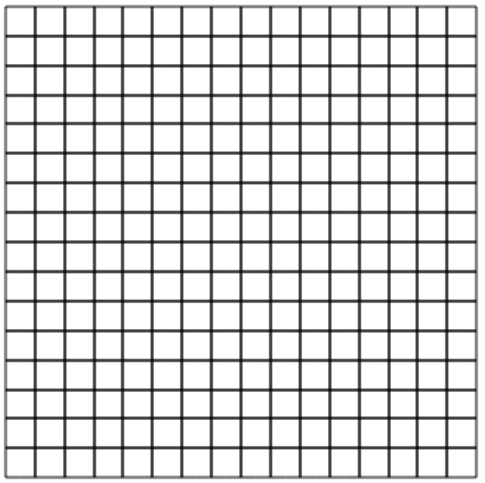
TMz Mode

$$E_z(i,j) = E_z(i,j) + \frac{\Delta t}{\epsilon_0 \Delta y}\left[H_x(i,j) - H_x(i,j+1) + H_y(i,j) - H_y(i+1,j)\right]$$

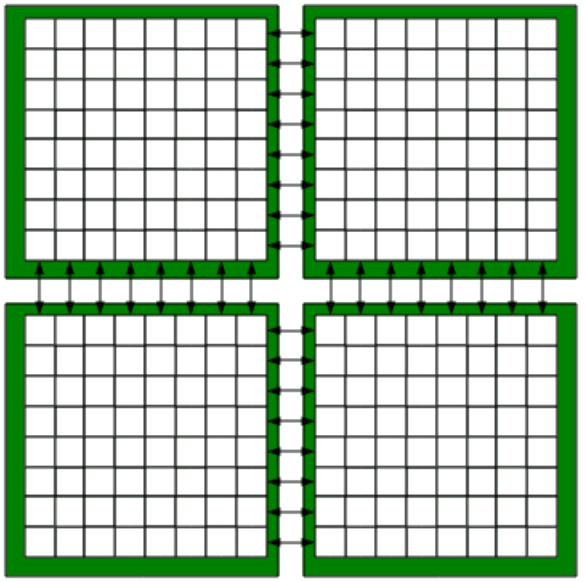
$$H_x(i,j) = H_x(i,j) + \frac{\Delta t}{\mu_0 \Delta y}\left[E_z(i,j-1) - E_z(i,j)\right]$$

$$H_y(i,j) = H_y(i,j) + \frac{\Delta t}{\mu_0 \Delta x}\left[E_z(i-1,j) - E_z(i,j)\right]$$

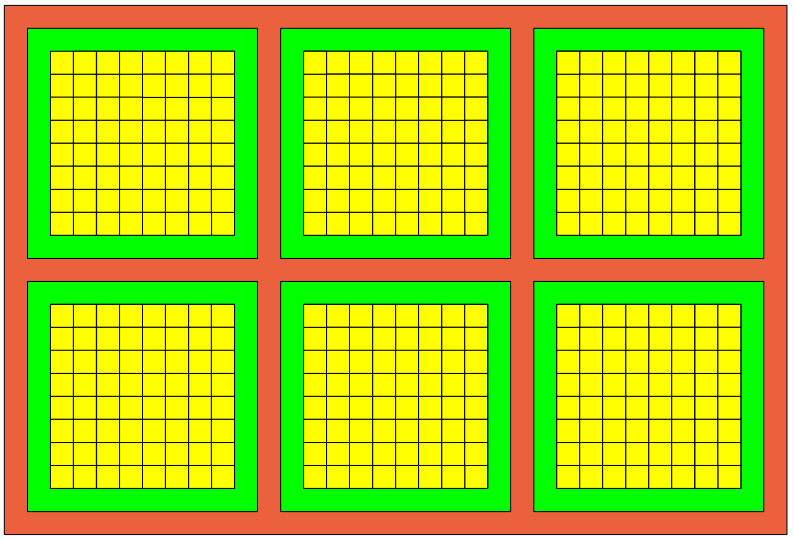




Example of a computational domain in two dimensions (16x16). Each box contains information about the fields and electromagnetic properties.



Division of the computational domain into four regions. The arrows indicate the process of communication between regions.



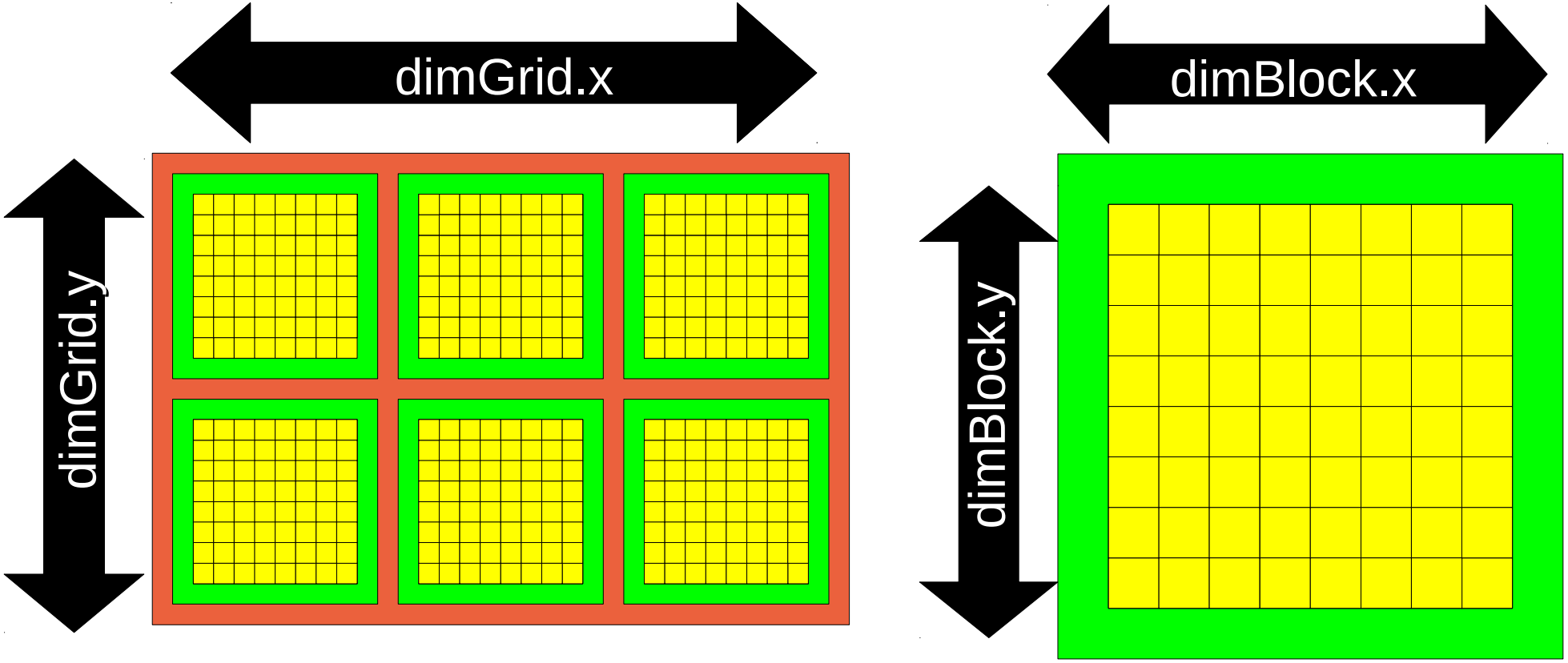
CUDA execution model representation. In blue, we have *threads* divided between the *blocks, in* green, and grouped into a *grid,* in red.

Syncronization issue !!!

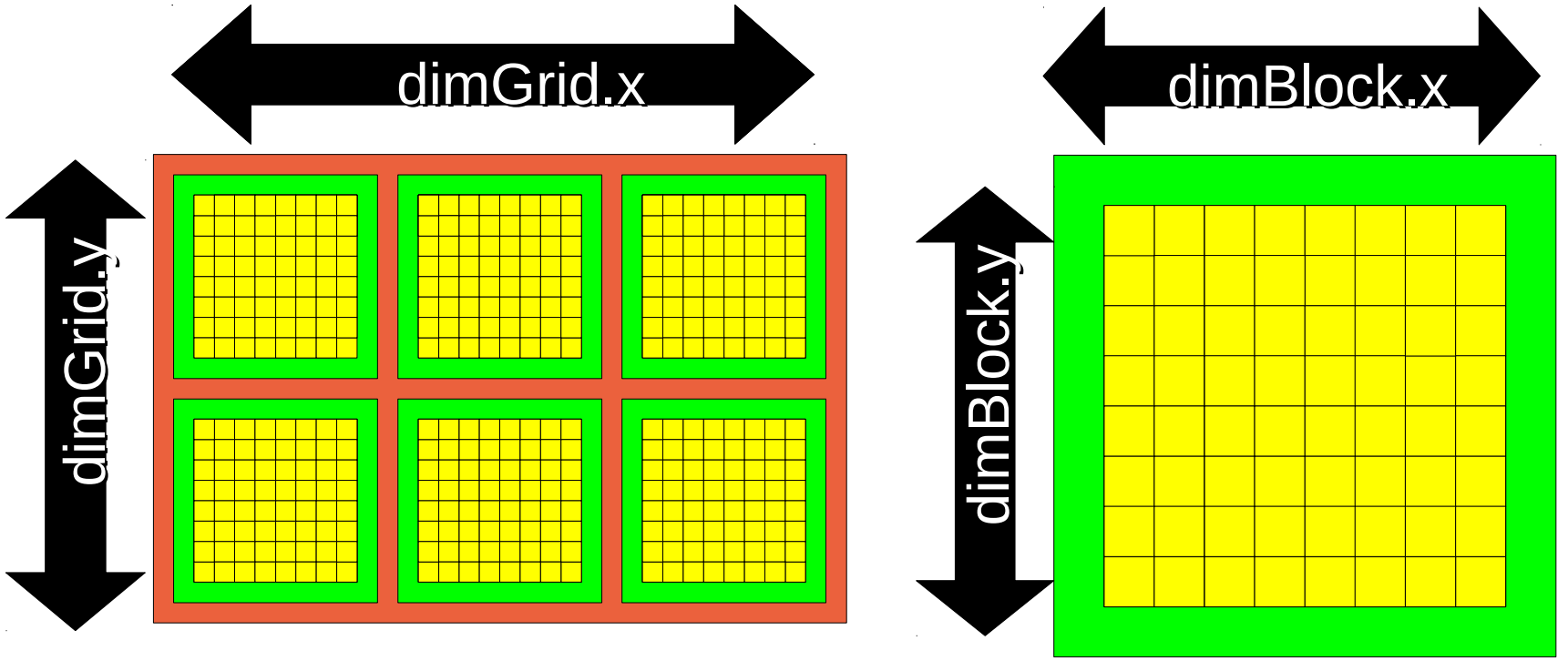
Set Threads and Blocks in 2D

dim3 dimBlock (BLOCKSIZEX, BLOCKSIZEY); //dim. of threads block

dim3 dimGrid (#of Blocks X direction, #of Blocks Y direction);



Mapping 2D computational domain in 1D arrays




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

Field[i+j*Lx] !

Setting Threads and Blocks

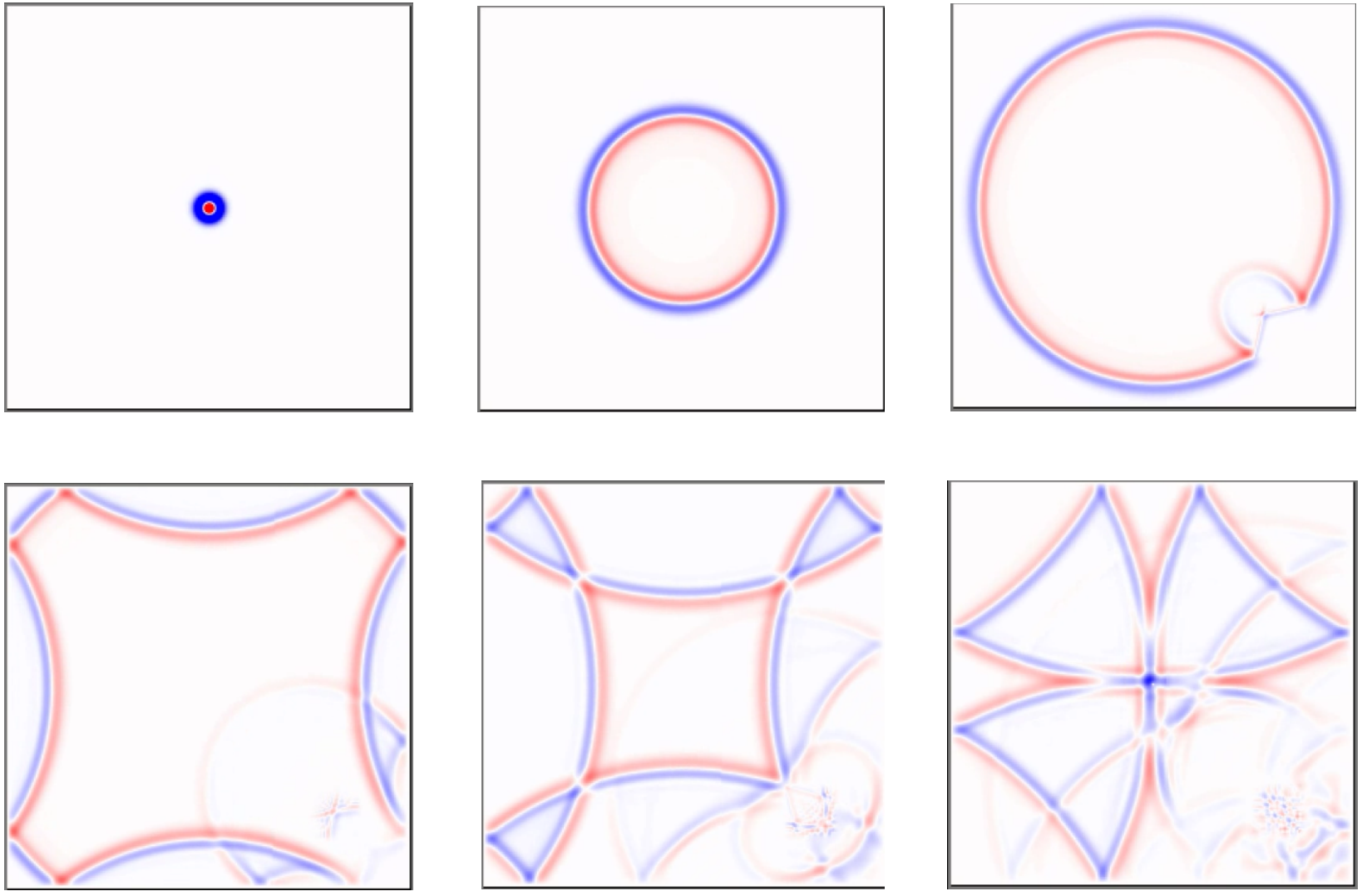
```
  int ly=lx;
  //computational y size set by user
  dim3 dimBlock (BLOCKSIZEX, BLOCKSIZEY);
  //dimensions of threads block
  dim3 dimGrid ((lx/(dimBlock.x) + (lx%(dimBlock.x) == 0?0:1)) , (ly/(dimBlock.y) + (ly%(dimBlock.y) == 0?0:1)));
  //grid size that fits the user domain

  int Lx=dimBlock.x*dimGrid.x;
  //computational x size
  int Ly=dimBlock.y*dimGrid.y;
  //computational y size
  int D=Lx*Ly;                                                                    //total
  computational domais.
  int Dsize=D*sizeof(float);
////////////////////////////////////////
```

Launching Kernel

```
  WaveStepE <<< dimGrid, dimBlock >>> (Hx,Hy,Ez, Lx, Ly, lx, ly, field,ez);
  WaveStepH <<< dimGrid, dimBlock >>> (Hx,Hy,Ez, Lx, Ly,mx,my);
```

Example !

Electric field time evolution



Video

Benchmark: Our Cuda code x Our CPU code x reference Cuda code[3]

Grid Points	CPU FDTD 2D Time(s)	Reference GPU FDTD 2D Time(s)	Our GPU FDTD 2D Time(s)
128x128	0,405	0,39	0,162
256x256	2,253	0,76	0,245
512x512	10,408	2,2	0,519
1024x1024	43,531	8,05	1,735
2048x2048	183,899	31,6	6,065
4096x4096	568,199		20,402

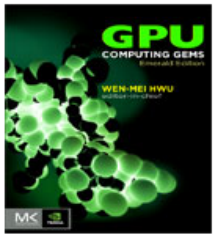
Table 1: Real elapsed time for the execution 2000 FDTD time iterations of the propagation of a Gaussian Pulse on a conductor box

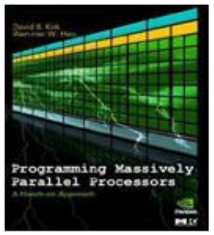
Speedups

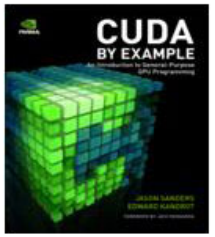
Grid Points	SpeedUP (our CPU/ our GPU)	SpeedUP (GPU ref. / our GPU)
128x128	2,5	2,4
256x256	9,2	3,1
512x512	20,1	4,2
1024x1024	25,1	4,6
2048x2048	30,3	5,2
4096x4096	27,9	

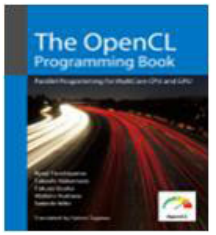
Table 2: Speedups for the execution 2000 FDTD time iterations of the propagation of a Gaussian Pulse on a conductor box

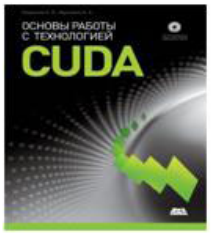
References











CUDA Webinars
CUDA Courses Online
CUDA Courses Around the World
CUDA Teaching Centers
CUDA Books
CUDA Activities

http://developer.nvidia.com/cuda-education-training

groups.google.com.br/group/gpubrasil

Dr. Dobbs: Supercomputing for masses