# Hands-on CUDA

Prof. Esteban Walter Gonzalez Clua, Dr.

Cuda Fellow

Computer Science Department

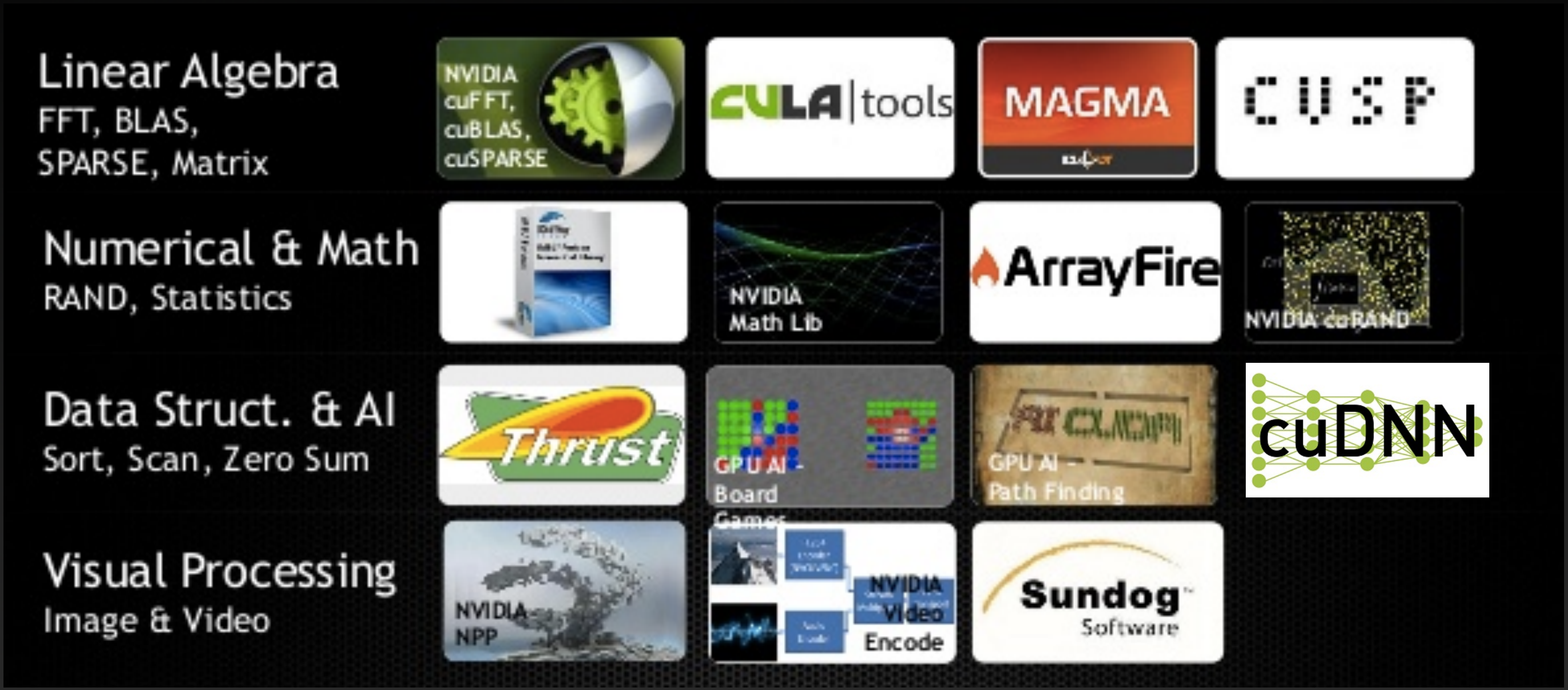
Universidade Federal Fluminense – Brazil

## How to use the GPU

1) Libraries (ex. Physics) : ready to use
2) Compiler Directives (ex. OpenACC)
3) Programming Language (CUDA, OpenCL)

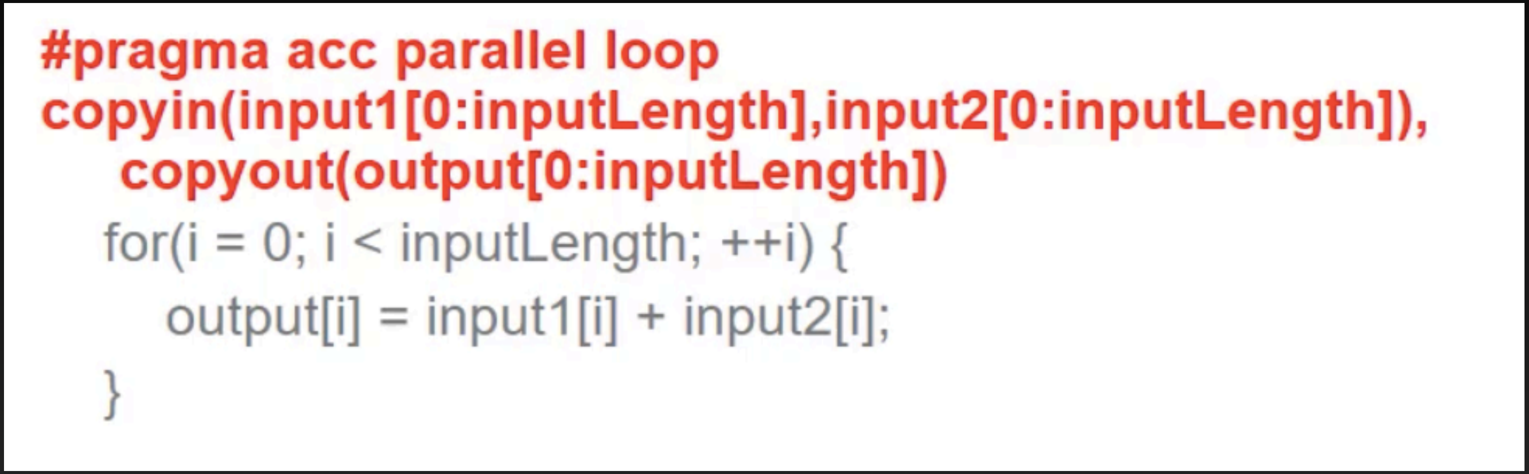
## Work Proposal: OpenACC

# Libraries



- Don’t require any GPU knowledge or even parallel computing experience

# Compiler directives



```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
  copyout(output[0:inputLength])
  for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
  }
```

Don’t requires GPU knowledge, but a little bit of parallel programming

## Programming Language

More Efficient

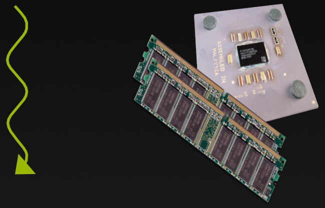
Flexible: may be better optimized for specific platforms

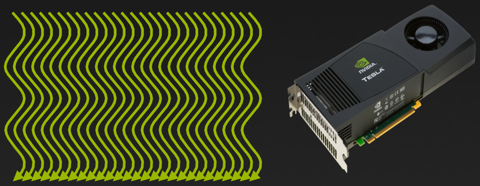
Verbose: express more details

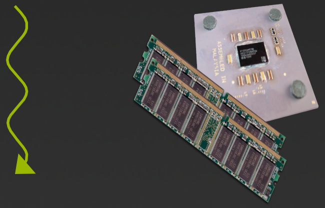
# Latency devices (CPU)x Throughput devices (GPU)

CPU is much more faster when latency matters

GPU is much more faster when througput matters





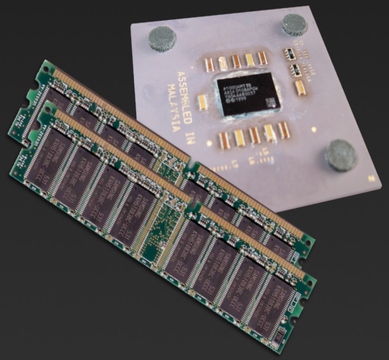


## Paradigmas de GPU Programming

3 coisas que voce deve saber de cor!

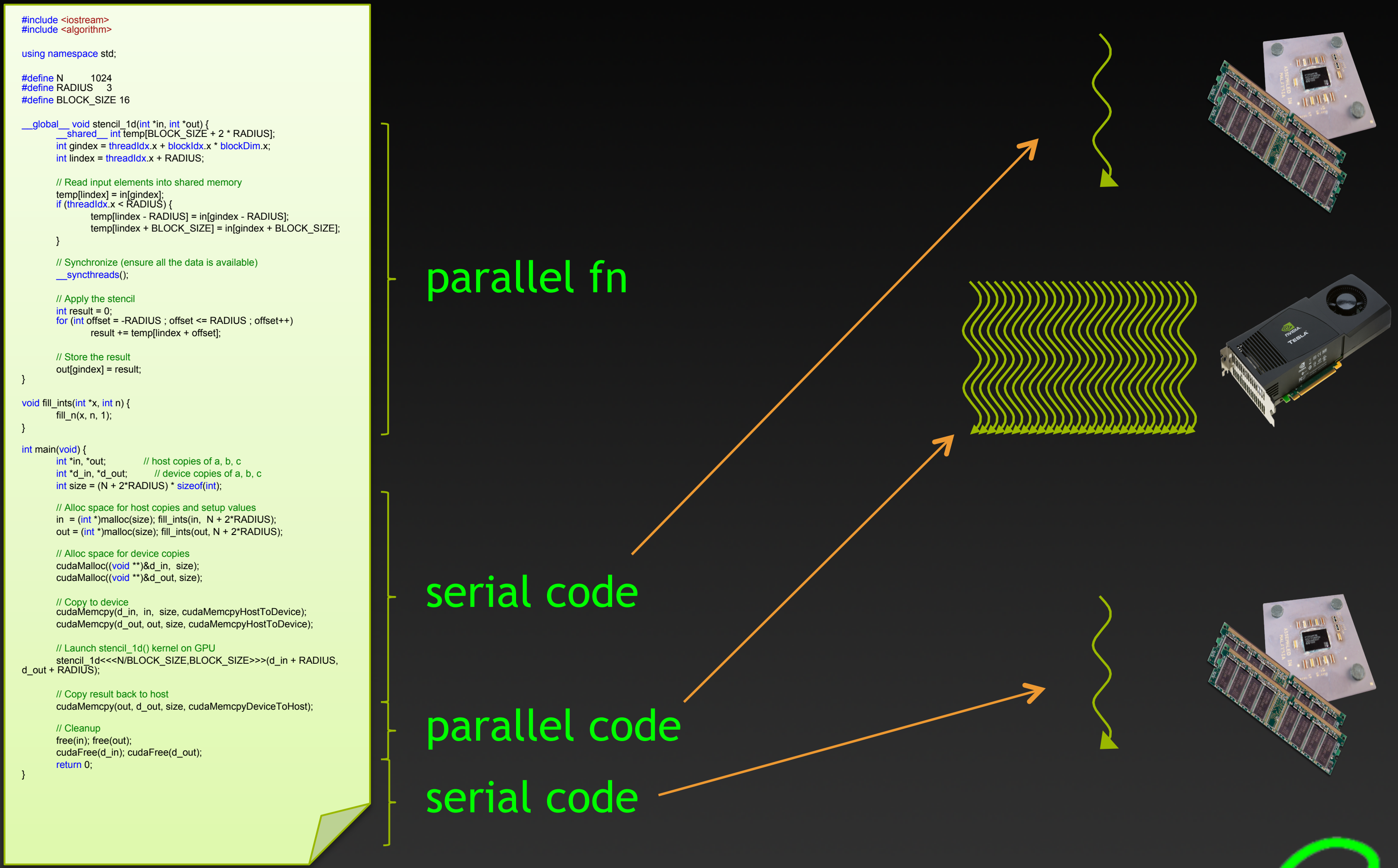
## #1 – Estamos falando de computação heterogênea

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


Host

Device

## Heterogeneous Computing

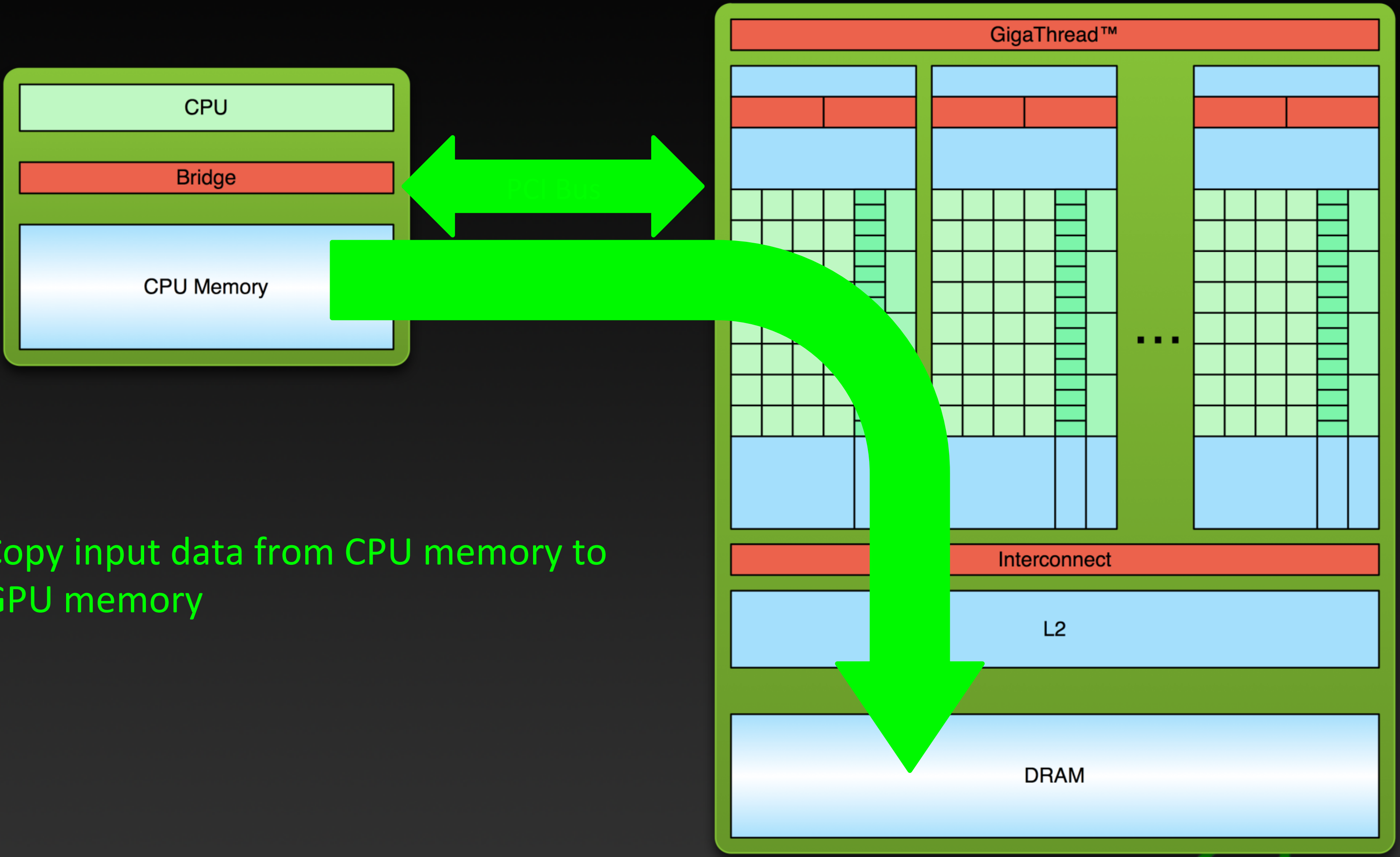


```
#include <iostream>
#include <algorithm>

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;
}
```

This slide is credited to Mark Harris (nvidia)

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

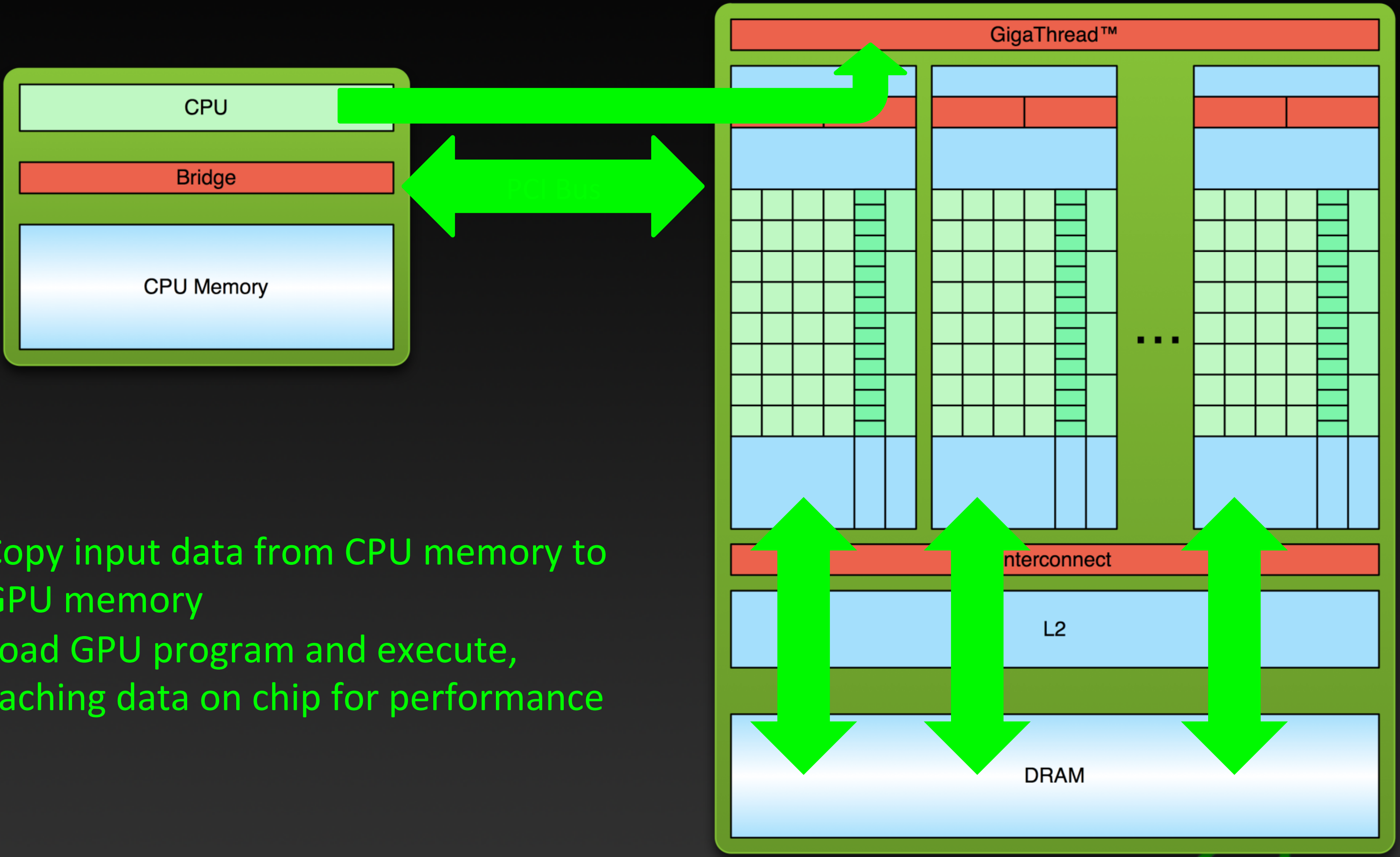
# GPU Computing Flow



1. Copy input data from CPU memory to GPU memory

This slide is credited to Mark Harris (nvidia)

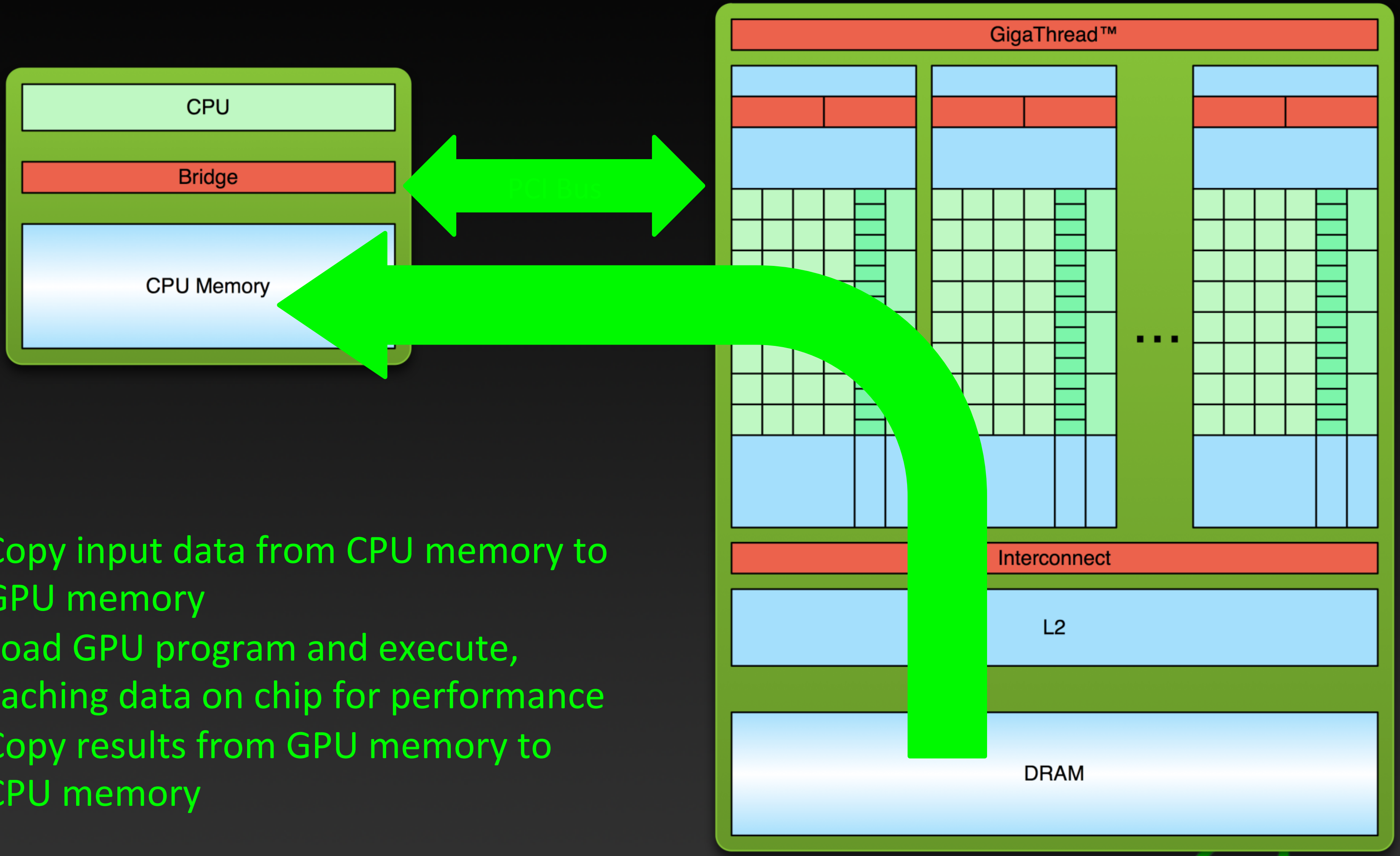
# GPU Computing Flow



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)

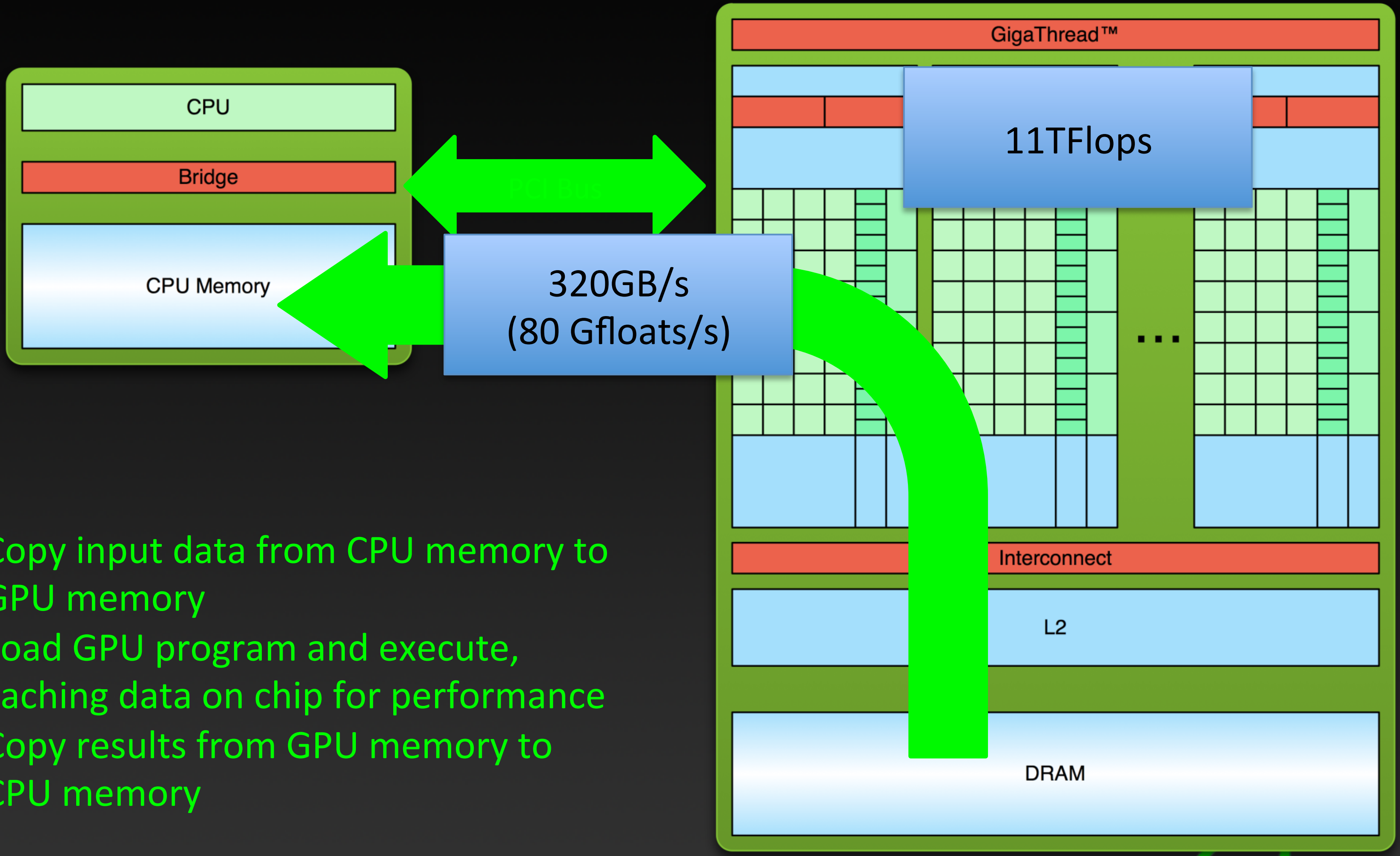
# GPU Computing Flow



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

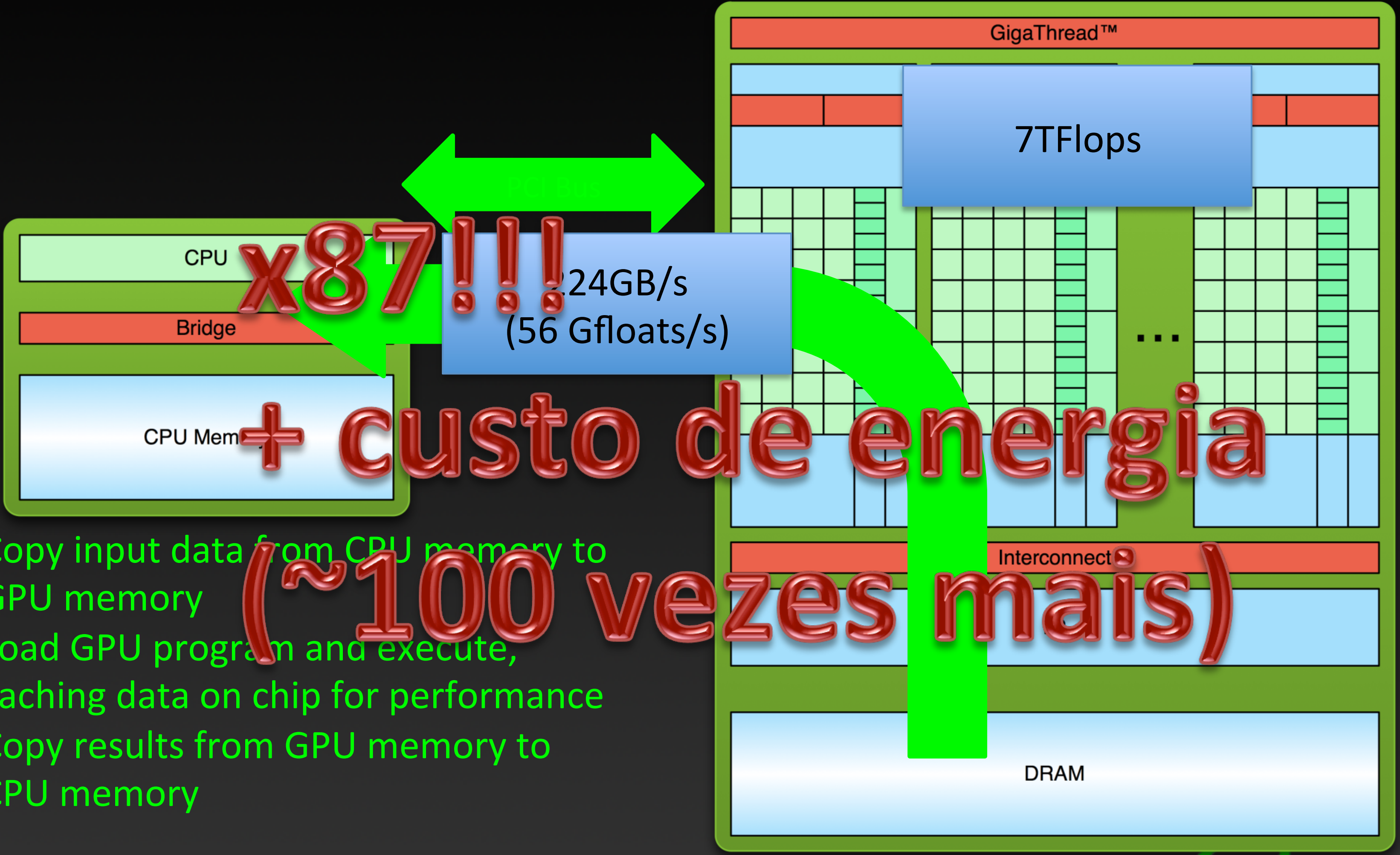
This slide is credited to Mark Harris (nvidia)

# 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

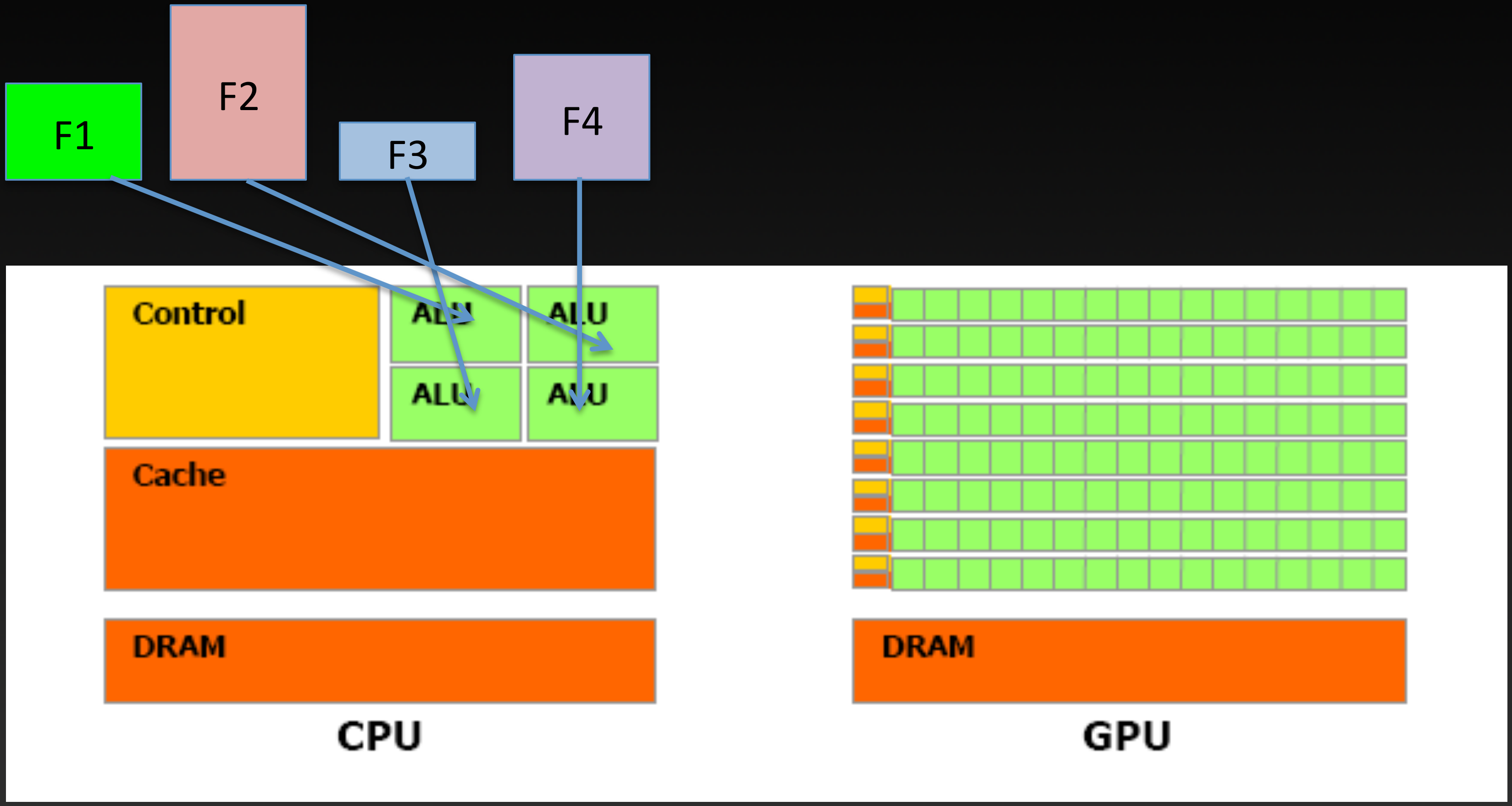
# GPU Computing Flow



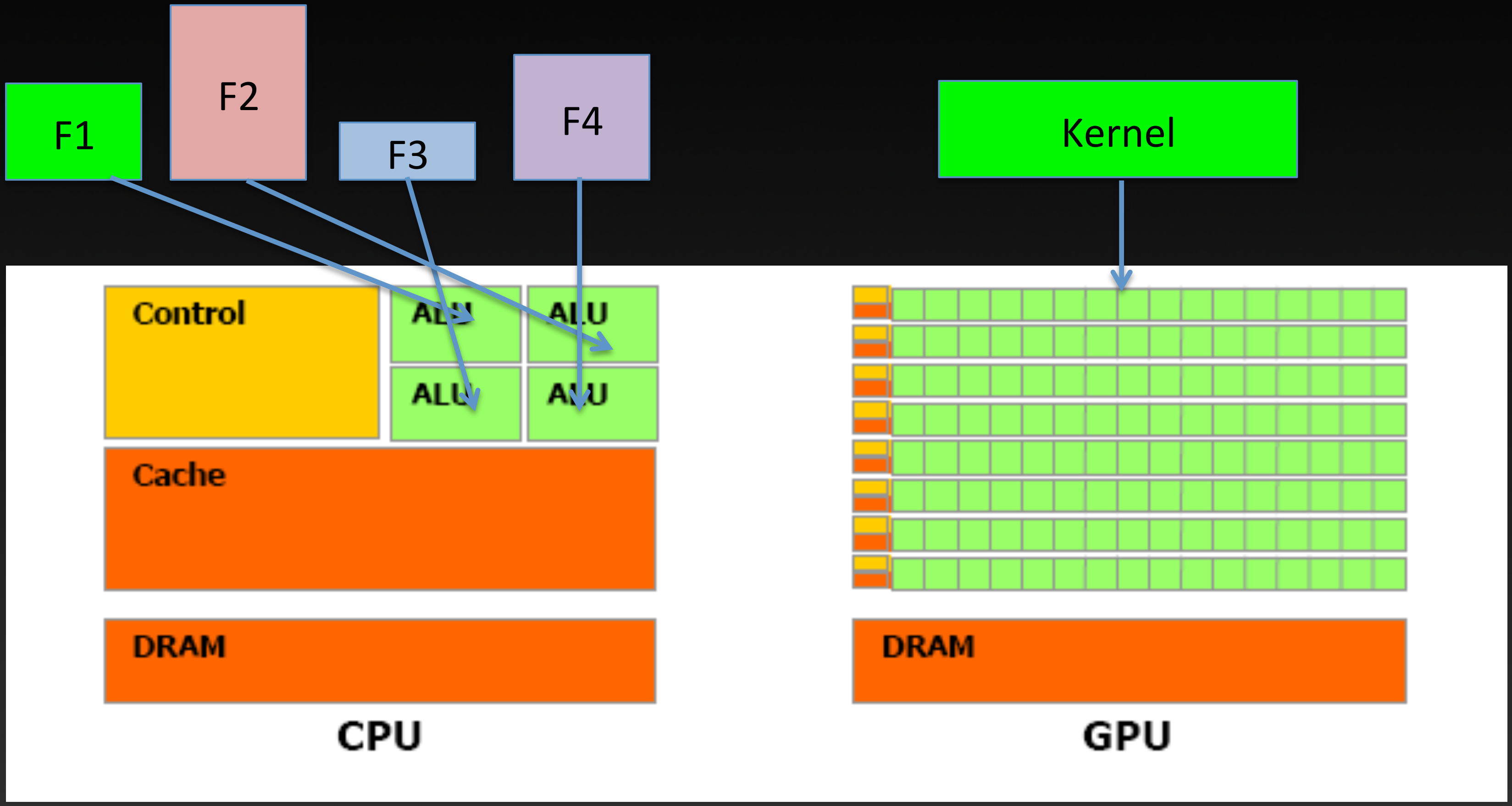
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

## #3 – 1 kernels, muitos threads...

# Threads em GPU x CPU



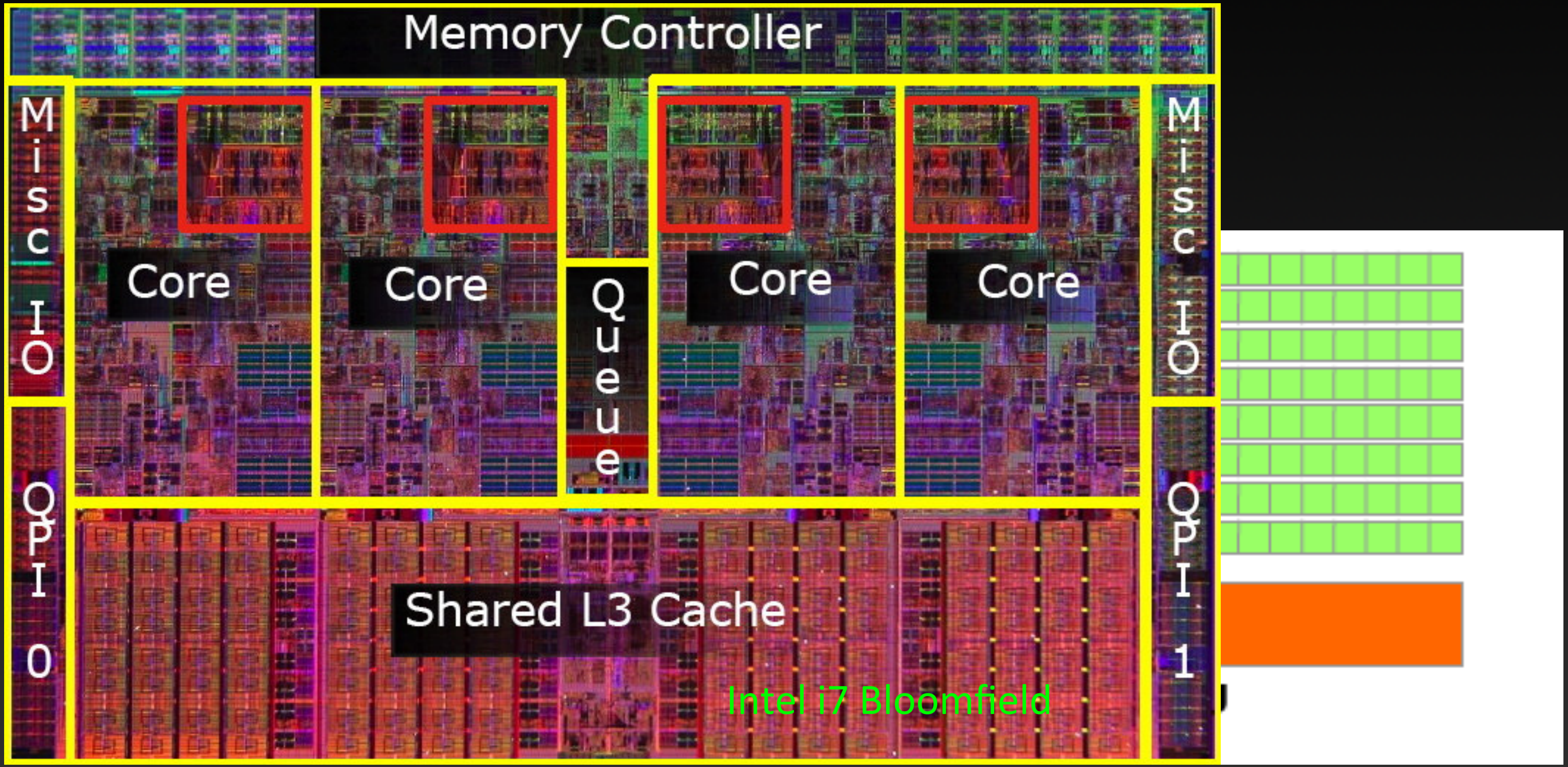
# Threads em GPU x CPU



## Modelo SIMT

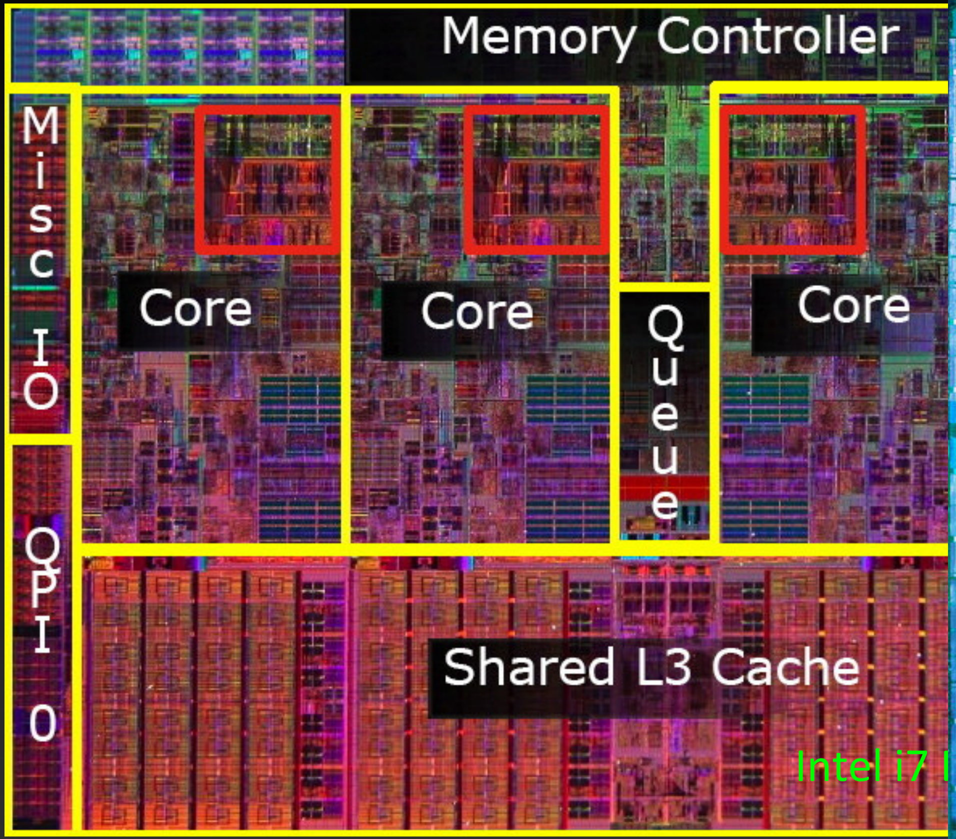
SIMT
means
Single Instruction
Multiple Thread
...
by allacronyms.com
ALLACRONYMS

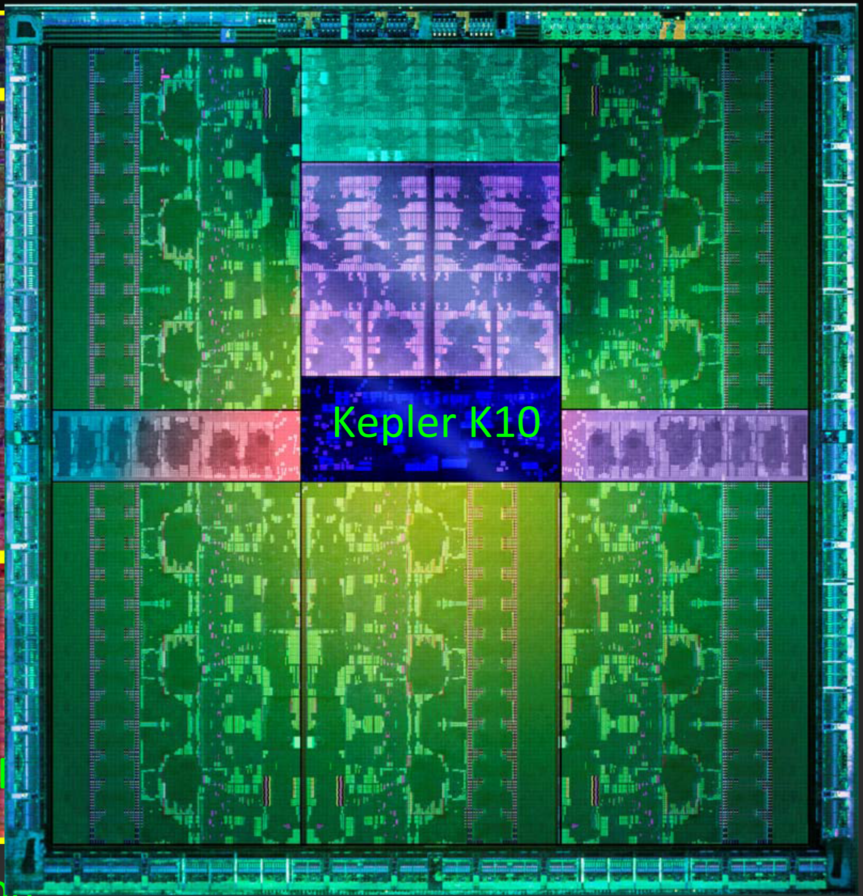
# GPU x CPU



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

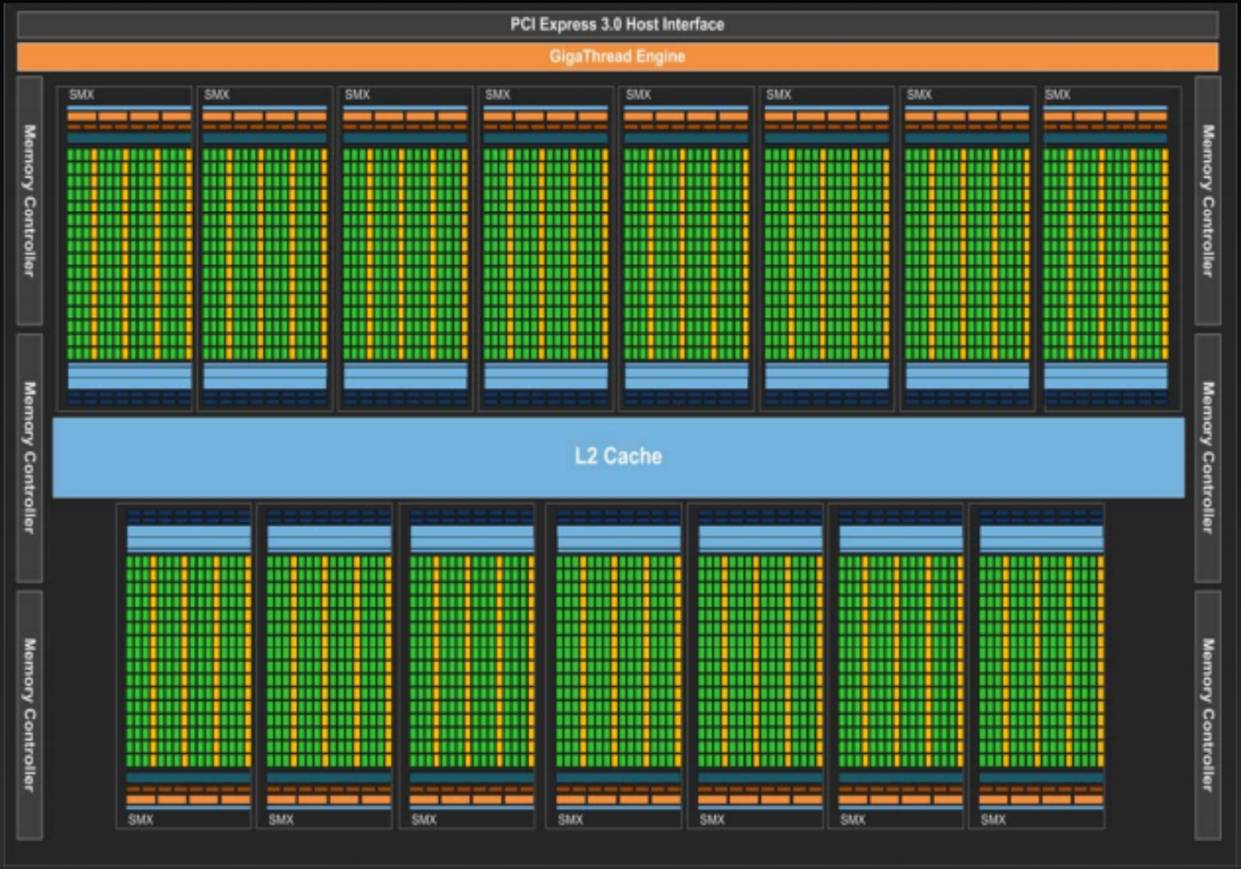
# GPU x CPU





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

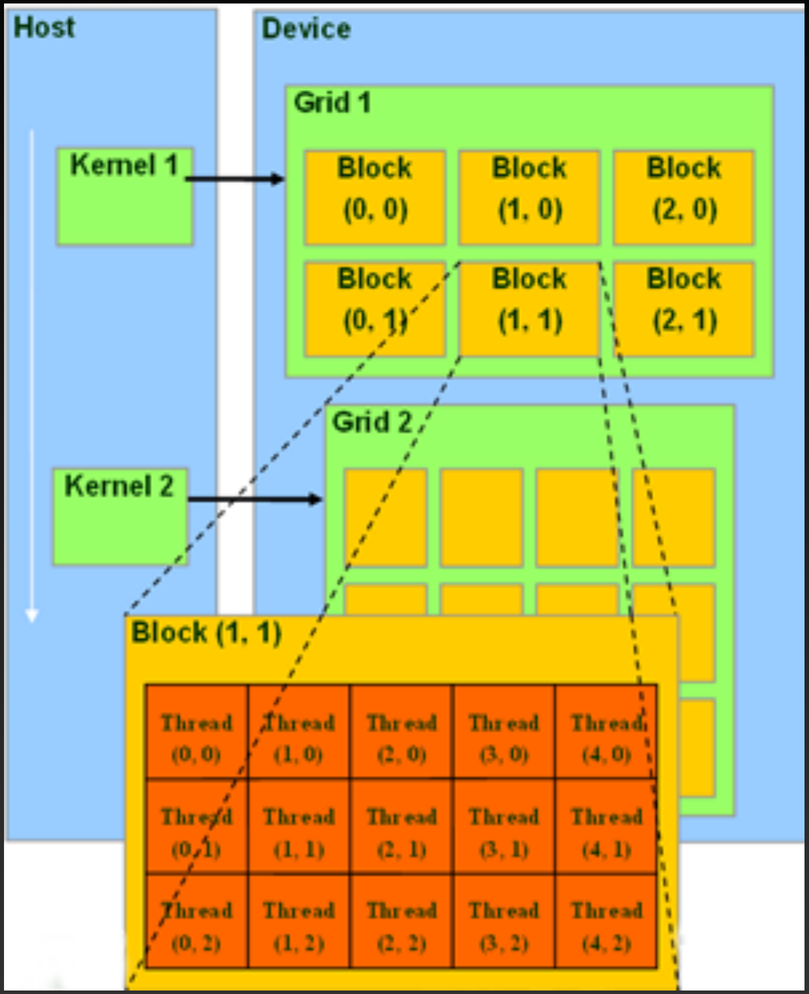
# GRID


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
SMX
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)

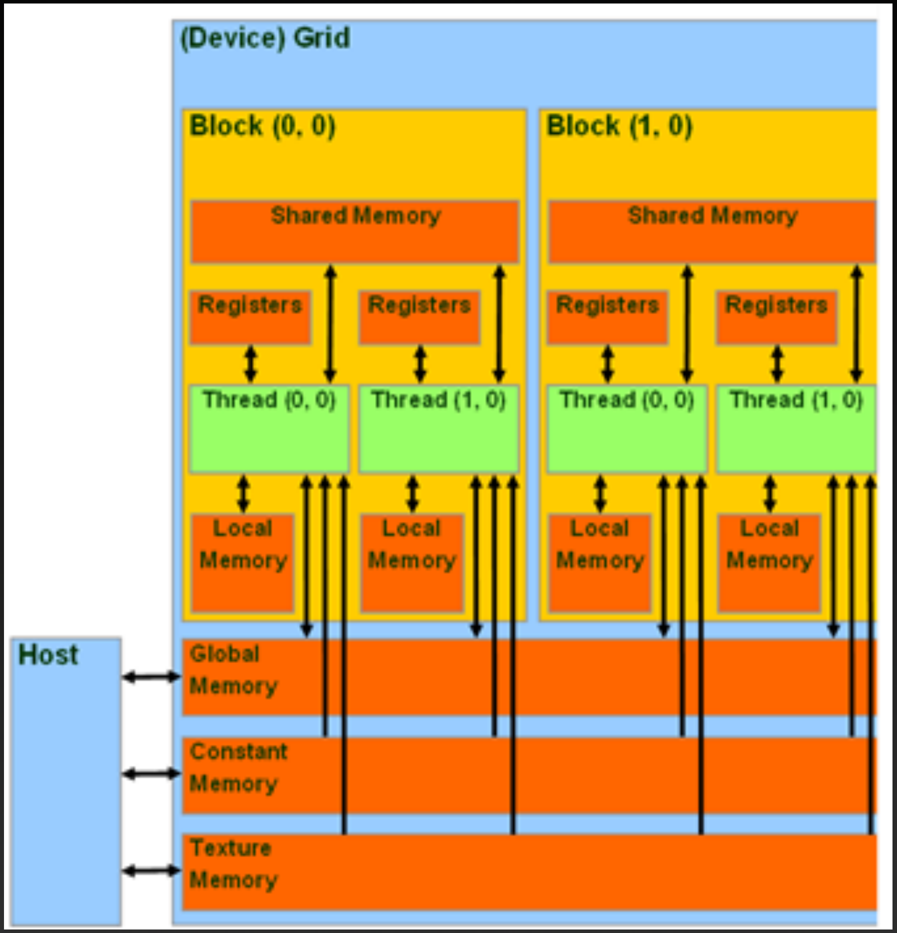
Todas as threads de um bloco podem usar a mesma shared memory

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

## Kernel, Threads e Warps

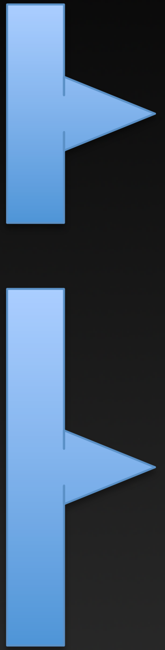
# Memórias...

- 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;
}
```



## 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

# Acesso Remoto a máquinas com Suporte CUDA

Instruções para uso das máquinas remotas para programação em CUDA.

## Instruções Importantes.

- Você irá utilizar a máquina remota somente para compilar e executar o programa.
- Existe um diretório chamado **Alunos** na raiz do usuário, onde cada um deve criar uma pasta para si.
- Criar um diretório com o seu nome e renomeá-lo.
- A criação dessas pastas individuais será necessária pois existem várias pessoas da mesma turma utilizando a mesma conta.

## Instruções Importantes

- Para compilar o projeto, digite o comando **make**
- Para limpar os códigos de máquina gerados digite o comando **make clean**.
- Manual CUDA está disponível neste link- http://docs.nvidia.com/cuda/cuda-c-programming-guide/

## Usuário e senha

u: cudateaching

p: teachingCUDA2018

Esta conta é uma conta temporária, e será suspensa ao final do curso.

## Divisão de Acceso da máquina

- Devem se conectar à máquina:
  - 200.20.15.153 – Porta: 22

- Para utilizar uma determinada GPU é preciso escrever na frente do comando de execução a variável de ambiente seguinte:

```
CUDA_VISIBLE_DEVICES=1
```

- No caso anterior o 1 é o índice da GPU 1, sendo que a DGX tem 8 GPUs, portanto:
  - Grupo 01 devem rodar os seus testes na GPU 1
  - Grupo 02 devem rodar os seus testes na GPU 2
  - Grupo 03 devem rodar os seus testes na GPU 3

## Orientações Importantes

- Devido a esta conta ser uma conta compartilhada, você deve criar uma pasta sua ou para sua equipe dentro da pasta **Alunos**, para que assim os arquivos de cada um fiquem organizados.
- A pasta **NVIDIA_CUDA-9.0_Samples** que está na raiz da máquina, é a pasta da NVIDIA contendo todos os exemplos de desenvolvimento do SDK. Você não deve modificar esta pasta!
- Sempre que quiser utilizar algum código desta pasta, faça uma cópia para sua pasta que você criou.

## Rodando um Hello World em CUDA 01

- Baixe PARA SUA PASTA o seguinte programa através do link https://www.dropbox.com/s/3qay3sp2wpfhull/vectorAddSample.zip

- Esse código nada mais é que o exemplo padrão da NVIDIA de soma de dois vetores em paralelo.

# Rodando um Hello World em CUDA 02

- Voce pode baixar este software através do comando "wget https://www.dropbox.com/s/3qay3sp2wpfhull/vectorAddSample.zip"

- Depois deve descompactar o arquivo com o comando "unzip vectorAddSample.zip"

# Rodando um Hello World em CUDA 03

- Para compilar o projeto basta digitar **make** e pressionar enter.

- Para executar lembre-se sempre de adicionar a variável de ambiente para selecionar a GPU do seu grupo (por exemplo para os alunos do Grupo 2):

```
CUDA_VISIBLE_DEVICES=2 ./vectorAdd
```

- Para limpar o projeto basta digitar **make clean** e pressionar enter.

## Rodando um Hello World em CUDA 04

- Fique à vontade para modificar a cópia do projeto da maneira que lhe for mais conveniente.

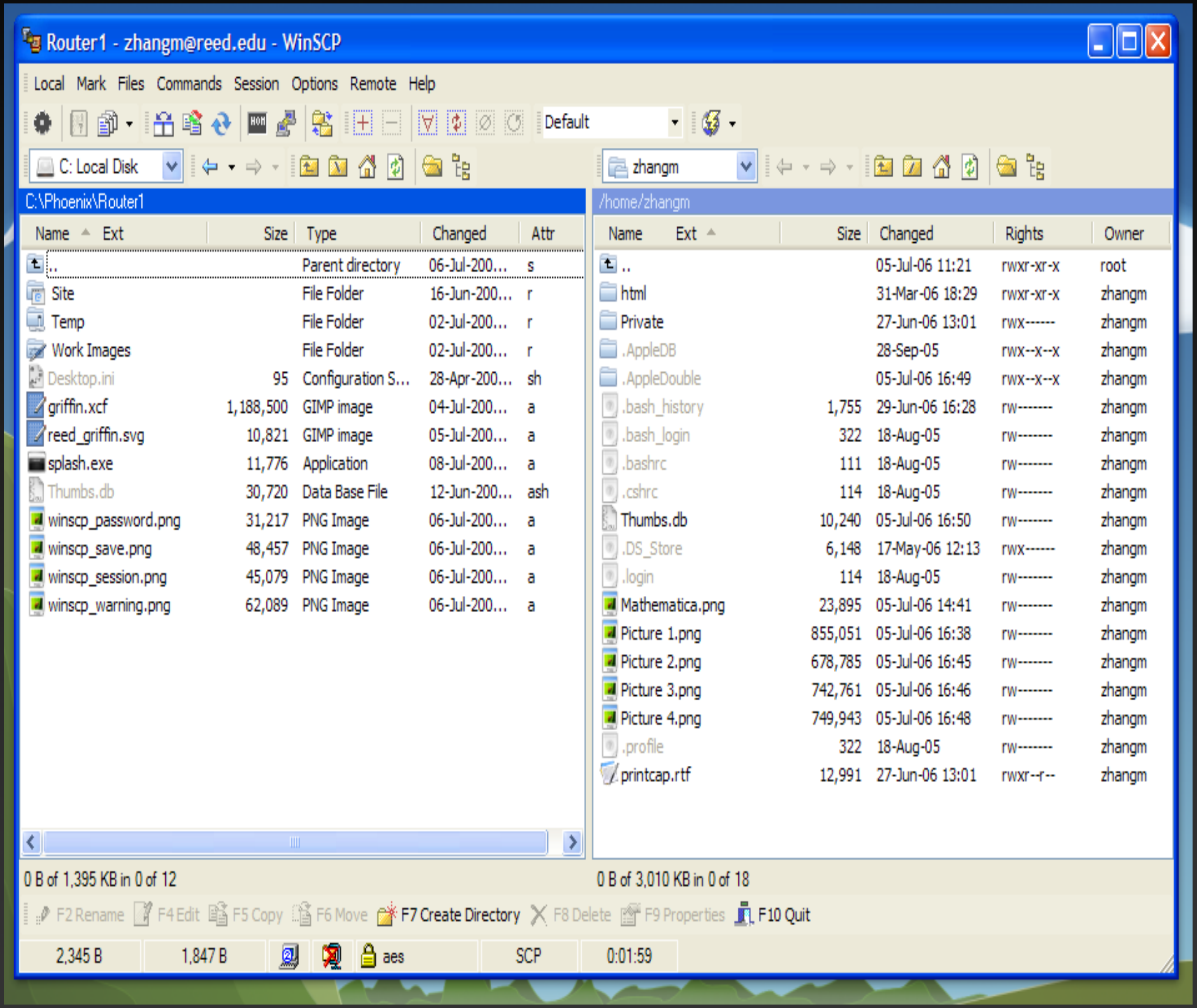
- Alternativamente este mesmo software que soma dois vetores, também está disponível na pasta **NVIDIA_CUDA-9.0_Samples** na subpasta **0_Simple**, na pasta **vectorAdd**.

- Você pode copiar esta pasta com o código fonte para a pasta pessoal sua que você criou.

## Programas sugeridos para acessar a máquina remota a partir de Windows

- MOBAXTERM - http://mobaxterm.mobatek.net/
- Através desse software voce se conecta a maquina desejada, e ele dispoe de todas as ferramentas necesarias, desde transferencia de arquivo através de interface, até terminal e editor de texto.

# winSCP- Troca de Arquivos- http://winscp.net/



Router1 - zhangm@reed.edu - WinSCP
Local Mark Files Commands Session Options Remote Help
Default
C: Local Disk
zhangm

C:\Phoenix\Router1

| Name | Ext | Size | Type | Changed | Attr |
|---|---|---|---|---|---|
| .. | | | Parent directory | 06-Jul-200... | s |
| Site | | | File Folder | 16-Jun-200... | r |
| Temp | | | File Folder | 02-Jul-200... | r |
| Work Images | | | File Folder | 02-Jul-200... | r |
| Desktop.ini | | 95 | Configuration S... | 28-Apr-200... | sh |
| griffin.xcf | | 1,188,500 | GIMP image | 04-Jul-200... | a |
| reed_griffin.svg | | 10,821 | GIMP image | 05-Jul-200... | a |
| splash.exe | | 11,776 | Application | 08-Jul-200... | a |
| Thumbs.db | | 30,720 | Data Base File | 12-Jun-200... | ash |
| winscp_password.png | | 31,217 | PNG Image | 06-Jul-200... | a |
| winscp_save.png | | 48,457 | PNG Image | 06-Jul-200... | a |
| winscp_session.png | | 45,079 | PNG Image | 06-Jul-200... | a |
| winscp_warning.png | | 62,089 | PNG Image | 06-Jul-200... | a |

/home/zhangm

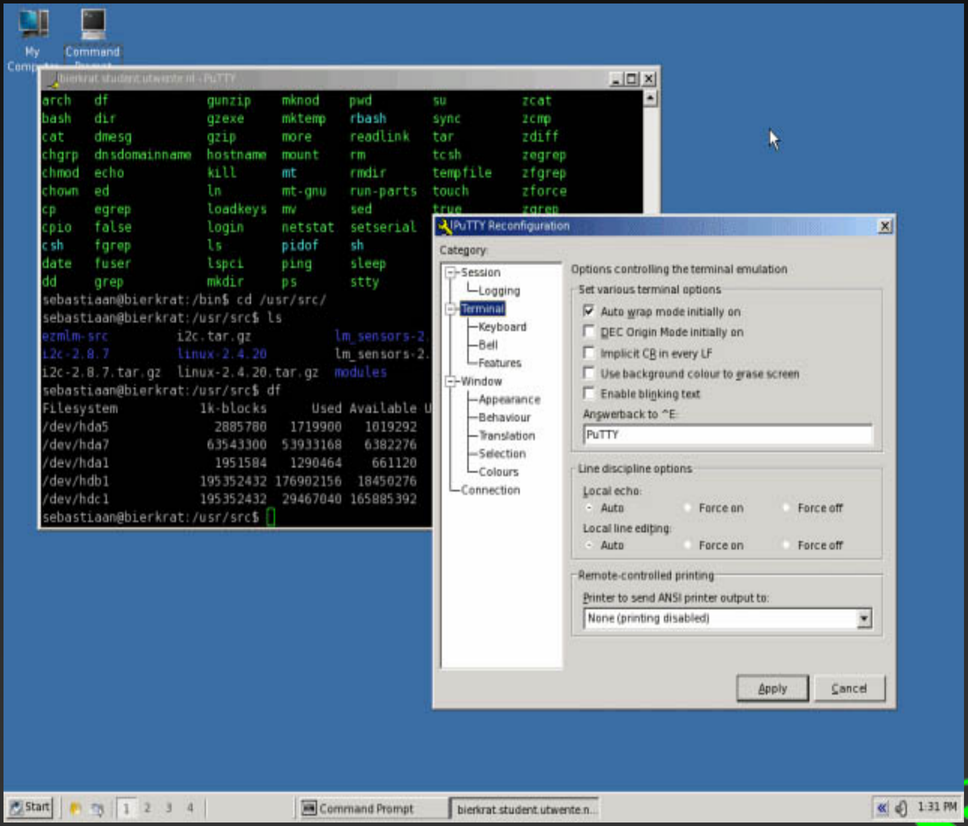
| Name | Ext | Size | Changed | Rights | Owner |
|---|---|---|---|---|---|
| .. | | | 05-Jul-06 11:21 | rwxr-xr-x | root |
| html | | | 31-Mar-06 18:29 | rwxr-xr-x | zhangm |
| Private | | | 27-Jun-06 13:01 | rwx------ | zhangm |
| .AppleDB | | | 28-Sep-05 | rwx--x--x | zhangm |
| .AppleDouble | | | 05-Jul-06 16:49 | rwx--x--x | zhangm |
| .bash_history | | 1,755 | 29-Jun-06 16:28 | rw------- | zhangm |
| .bash_login | | 322 | 18-Aug-05 | rw------- | zhangm |
| .bashrc | | 111 | 18-Aug-05 | rw------- | zhangm |
| .cshrc | | 114 | 18-Aug-05 | rw------- | zhangm |
| Thumbs.db | | 10,240 | 05-Jul-06 16:50 | rw------- | zhangm |
| .DS_Store | | 6,148 | 17-May-06 12:13 | rwx------ | zhangm |
| .login | | 114 | 18-Aug-05 | rw------- | zhangm |
| Mathematica.png | | 23,895 | 05-Jul-06 14:41 | rw------- | zhangm |
| Picture 1.png | | 855,051 | 05-Jul-06 16:38 | rw------- | zhangm |
| Picture 2.png | | 678,785 | 05-Jul-06 16:45 | rw------- | zhangm |
| Picture 3.png | | 742,761 | 05-Jul-06 16:46 | rw------- | zhangm |
| Picture 4.png | | 749,943 | 05-Jul-06 16:48 | rw------- | zhangm |
| .profile | | 322 | 18-Aug-05 | rw------- | zhangm |
| printcap.rtf | | 12,991 | 27-Jun-06 13:01 | rwxr--r-- | zhangm |

0 B of 1,395 KB in 0 of 12 | 0 B of 3,010 KB in 0 of 18

F2 Rename | F4 Edit | F5 Copy | F6 Move | F7 Create Directory | F8 Delete | F9 Properties | F10 Quit

2,345 B | 1,847 B | aes | SCP | 0:01:59

# **Putty** - Terminal de acesso remoto – SSH Client -www.putty.org – para Windows


My Computer
Command Prompt
bierkrat.student.utwente.nl - PuTTY
Start
Command Prompt
bierkrat.student.utwente.n...
1:31 PM

```
arch   df             gunzip    mknod    pwd        su        zcat
bash   dir            gzexe     mktemp   rbash      sync      zcmp
cat    dmesg          gzip      more     readlink   tar       zdiff
chgrp  dnsdomainname  hostname  mount    rm         tcsh      zegrep
chmod  echo           kill      mt       rmdir      tempfile  zfgrep
chown  ed             ln        mt-gnu   run-parts  touch     zforce
cp     egrep          loadkeys  mv       sed        true      zgrep
cpio   false          login     netstat  setserial
csh    fgrep          ls        pidof    sh
date   fuser          lspci     ping     sleep
dd     grep           mkdir     ps       stty
sebastiaan@bierkrat:/bin$ cd /usr/src/
sebastiaan@bierkrat:/usr/src$ ls
ezmlm-src          i2c.tar.gz           lm_sensors-2
i2c-2.8.7          linux-2.4.20         lm_sensors-2.
i2c-2.8.7.tar.gz  linux-2.4.20.tar.gz  modules
sebastiaan@bierkrat:/usr/src$ df
Filesystem           1k-blocks      Used Available U
/dev/hda5              2885780   1719900   1019292
/dev/hda7             63543300  53933168   6382276
/dev/hda1              1951584   1290464    661120
/dev/hdb1            195352432 176902156  18450276
/dev/hdc1            195352432  29467040 165885392
sebastiaan@bierkrat:/usr/src$ 
```

PuTTY Reconfiguration

Category:
- Session
  - Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection

Options controlling the terminal emulation

Set various terminal options
- [✓] Auto wrap mode initially on
- [ ] DEC Origin Mode initially on
- [ ] Implicit CR in every LF
- [ ] Use background colour to erase screen
- [ ] Enable blinking text

Answerback to ^E: PuTTY

Line discipline options

Local echo: [✓ Auto] Force on Force off

Local line editing: [✓ Auto] Force on Force off

Remote-controlled printing

Printer to send ANSI printer output to: None (printing disabled)

Apply Cancel

## Terminal (MAC)

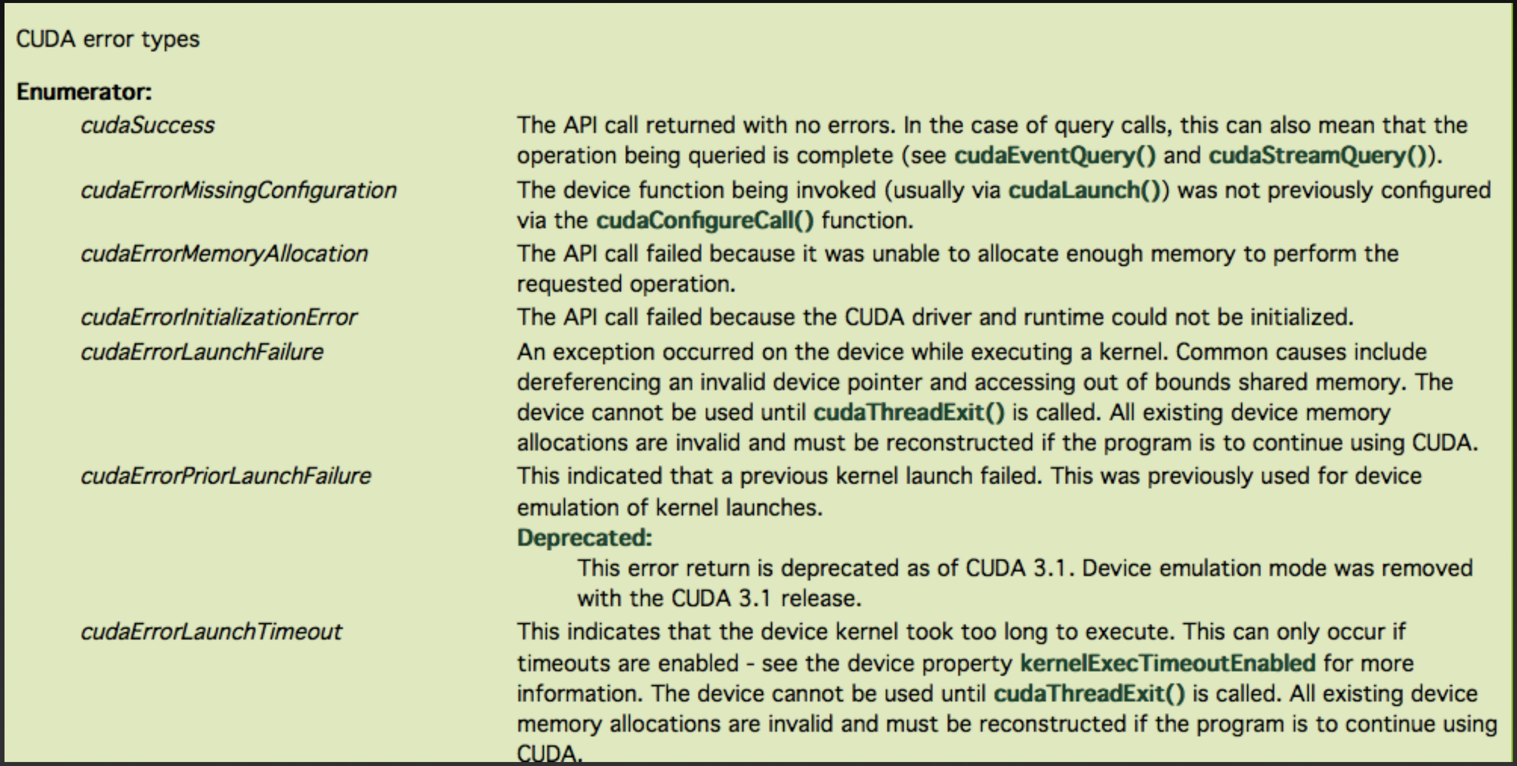
```
ssh cudateaching@200.20.15.153
```

## 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



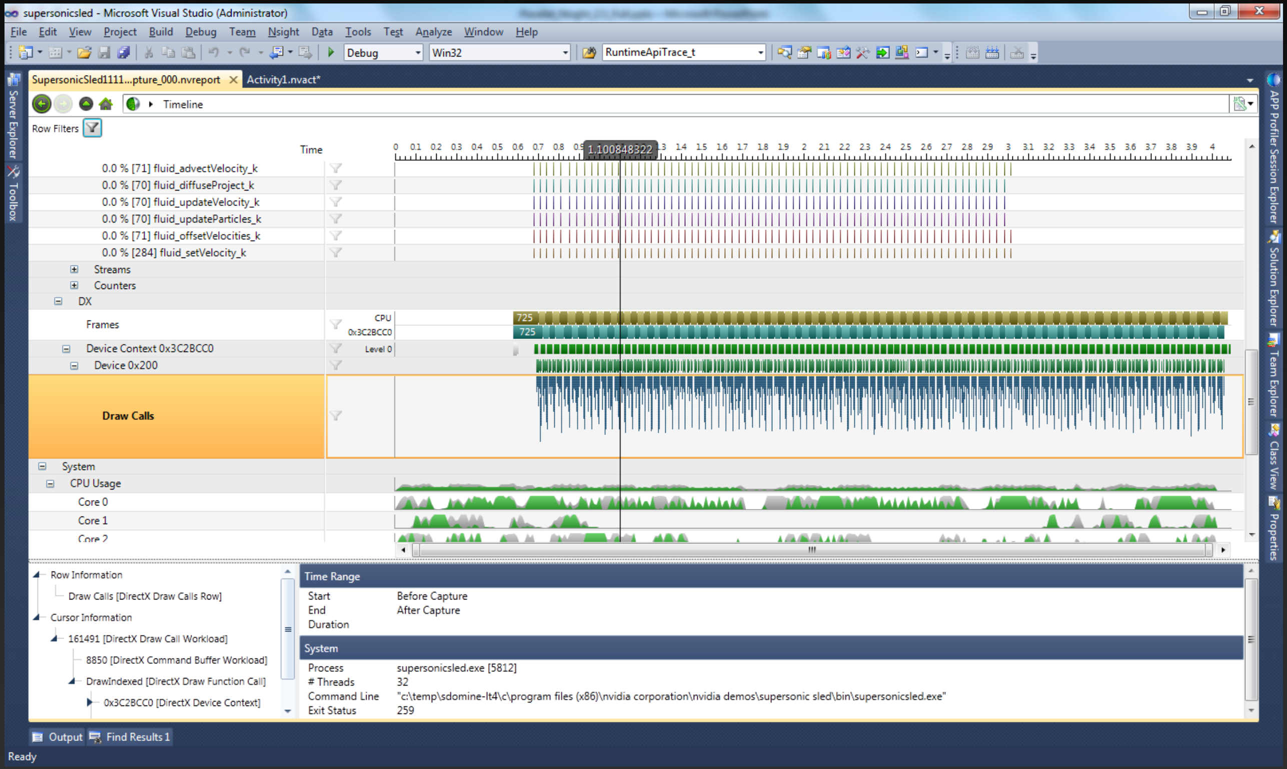
CUDA error types

**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**


supersonicsled - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help
Debug
Win32
RuntimeApiTrace_t
SupersonicSled1111...pture_000.nvreport
Activity1.nvact*
Timeline
Row Filters
Time
1.100848322
0.0 % [71] fluid_advectVelocity_k
0.0 % [70] fluid_diffuseProject_k
0.0 % [70] fluid_updateVelocity_k
0.0 % [70] fluid_updateParticles_k
0.0 % [71] fluid_offsetVelocities_k
0.0 % [284] fluid_setVelocity_k
Streams
Counters
DX
Frames
CPU
0x3C2BCC0
725
725
Device Context 0x3C2BCC0
Level 0
Device 0x200
Draw Calls
System
CPU Usage
Core 0
Core 1
Core 2
Server Explorer
Toolbox
APP Profiler Session Explorer
Solution Explorer
Team Explorer
Class View
Properties
Output
Find Results 1
Ready

Row Information
- Draw Calls [DirectX Draw Calls Row]

Cursor Information
- 161491 [DirectX Draw Call Workload]
  - 8850 [DirectX Command Buffer Workload]
  - DrawIndexed [DirectX Draw Function Call]
    - 0x3C2BCC0 [DirectX Device Context]

Time Range

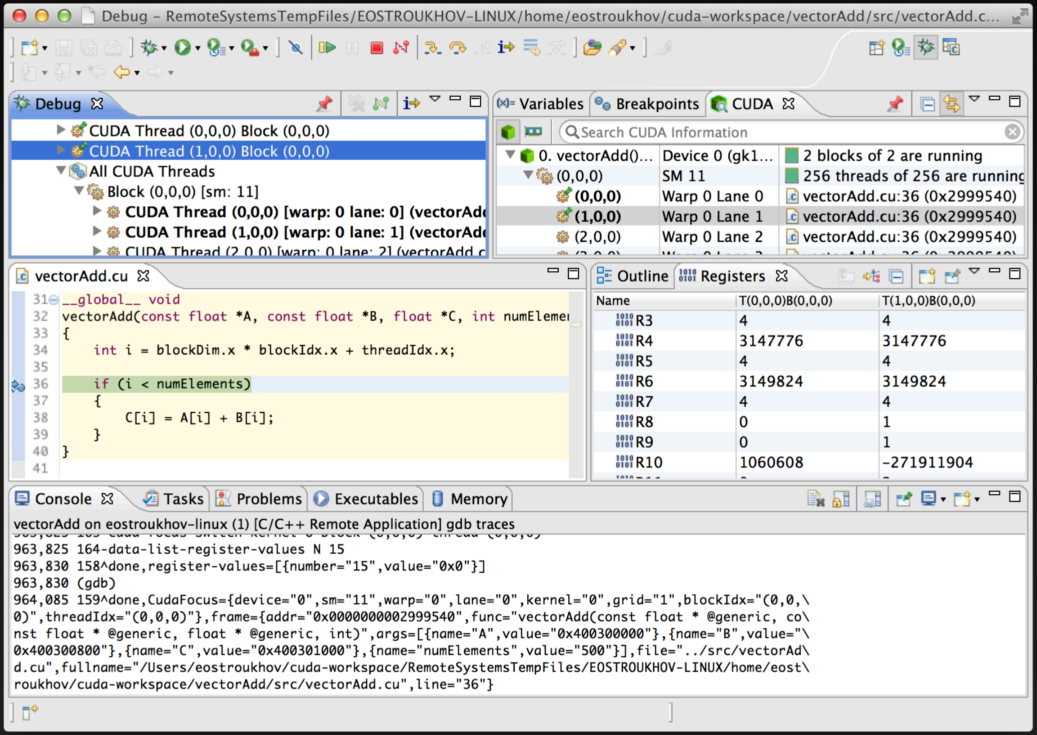
| | |
|---|---|
| Start | Before Capture |
| End | After Capture |
| Duration | |

System

| | |
|---|---|
| Process | supersonicsled.exe [5812] |
| # Threads | 32 |
| Command Line | "c:\temp\sdomine-lt4\c\program files (x86)\nvidia corporation\nvidia demos\supersonic sled\bin\supersonicsled.exe" |
| Exit Status | 259 |

# Debuggers

**CUDA GDB**



Debug – RemoteSystemsTempFiles/EOSTROUKHOV-LINUX/home/eostroukhov/cuda-workspace/vectorAdd/src/vectorAdd.c…

```
CUDA Thread (0,0,0) Block (0,0,0)
CUDA Thread (1,0,0) Block (0,0,0)
All CUDA Threads
  Block (0,0,0) [sm: 11]
    CUDA Thread (0,0,0) [warp: 0 lane: 0] (vectorAdd
    CUDA Thread (1,0,0) [warp: 0 lane: 1] (vectorAdd
    CUDA Thread (2,0,0) [warp: 0 lane: 2] (vectorAdd.c
```

| | | |
|---|---|---|
| 0. vectorAdd()… | Device 0 (gk1… | 🟩 2 blocks of 2 are running |
| (0,0,0) | SM 11 | 🟩 256 threads of 256 are running |
| (0,0,0) | Warp 0 Lane 0 | vectorAdd.cu:36 (0x2999540) |
| (1,0,0) | Warp 0 Lane 1 | vectorAdd.cu:36 (0x2999540) |
| (2,0,0) | Warp 0 Lane 2 | vectorAdd.cu:36 (0x2999540) |

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElemen
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

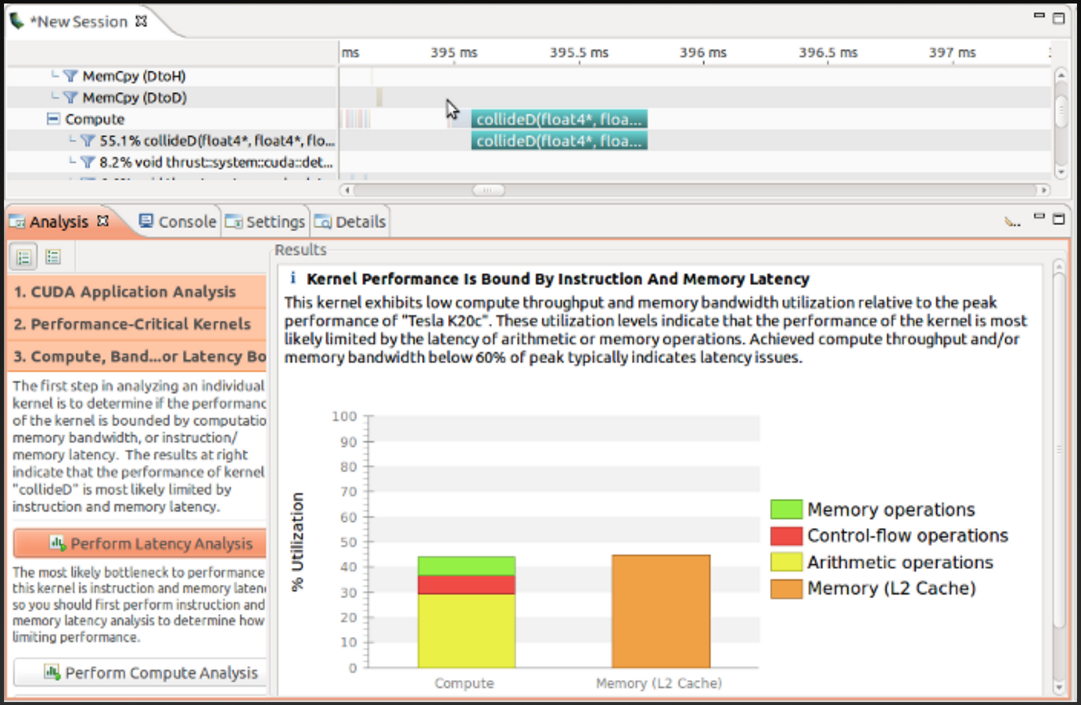
| Name | T(0,0,0)B(0,0,0) | T(1,0,0)B(0,0,0) |
|---|---|---|
| R3 | 4 | 4 |
| R4 | 3147776 | 3147776 |
| R5 | 4 | 4 |
| R6 | 3149824 | 3149824 |
| R7 | 4 | 4 |
| R8 | 0 | 1 |
| R9 | 0 | 1 |
| R10 | 1060608 | -271911904 |

vectorAdd on eostroukhov-linux (1) [C/C++ Remote Application] gdb traces

```
963,825 164-data-list-register-values N 15
963,830 158^done,register-values=[{number="15",value="0x0"}]
963,830 (gdb)
964,085 159^done,CudaFocus={device="0",sm="11",warp="0",lane="0",kernel="0",grid="1",blockIdx="(0,0,\
0)",threadIdx="(0,0,0)"},frame={addr="0x0000000002999540",func="vectorAdd(const float * @generic, co\
nst float * @generic, float * @generic, int)",args=[{name="A",value="0x400300000"},{name="B",value="\
0x400300800"},{name="C",value="0x400301000"},{name="numElements",value="500"}],file="../src/vectorAd\
d.cu",fullname="/Users/eostroukhov/cuda-workspace/RemoteSystemsTempFiles/EOSTROUKHOV-LINUX/home/eost\
roukhov/cuda-workspace/vectorAdd/src/vectorAdd.cu",line="36"}
```

# Debuggers

**CUDA Memcheck**



**Kernel Performance Is Bound By Instruction And Memory Latency**

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla K20c". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

The first step in analyzing an individual kernel is to determine if the performanc of the kernel is bounded by computatio memory bandwidth, or instruction/ memory latency. The results at right indicate that the performance of kernel "collideD" is most likely limited by instruction and memory latency.

The most likely bottleneck to performance this kernel is instruction and memory laten so you should first perform instruction and memory latency analysis to determine how limiting performance.

# 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
}
```

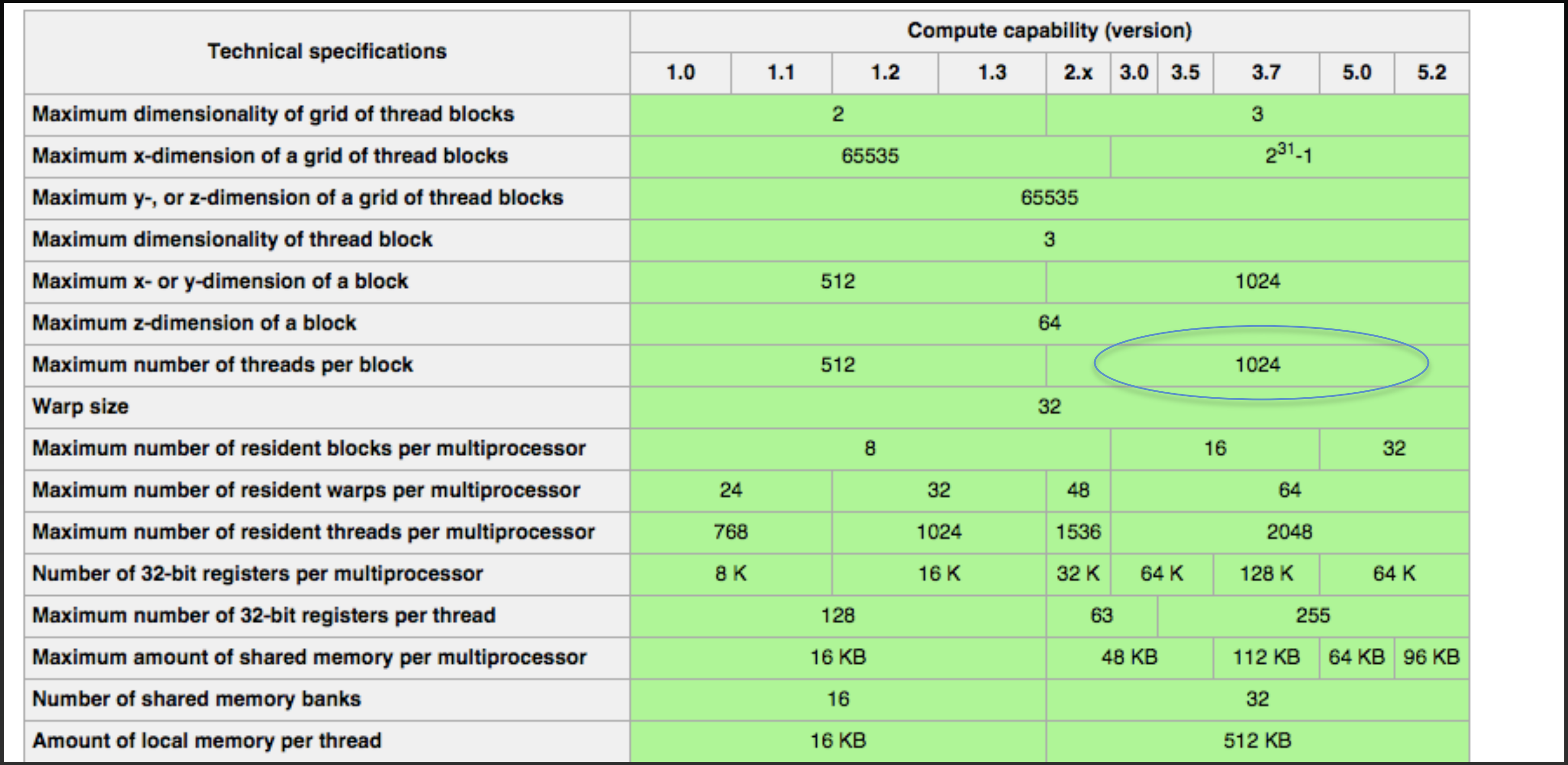
# Explorando o paralelismo: Threads

```
__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];
…
```

…

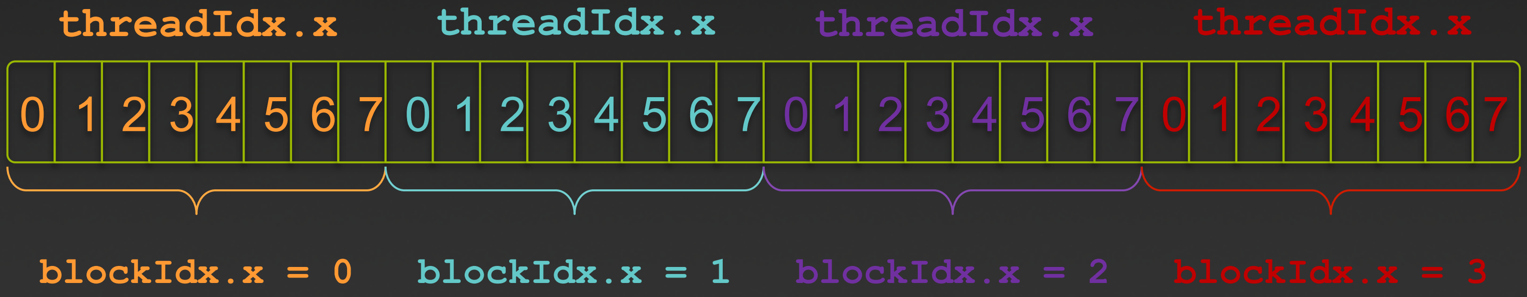
## Apenas estamos usando 1 SM!...

# 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

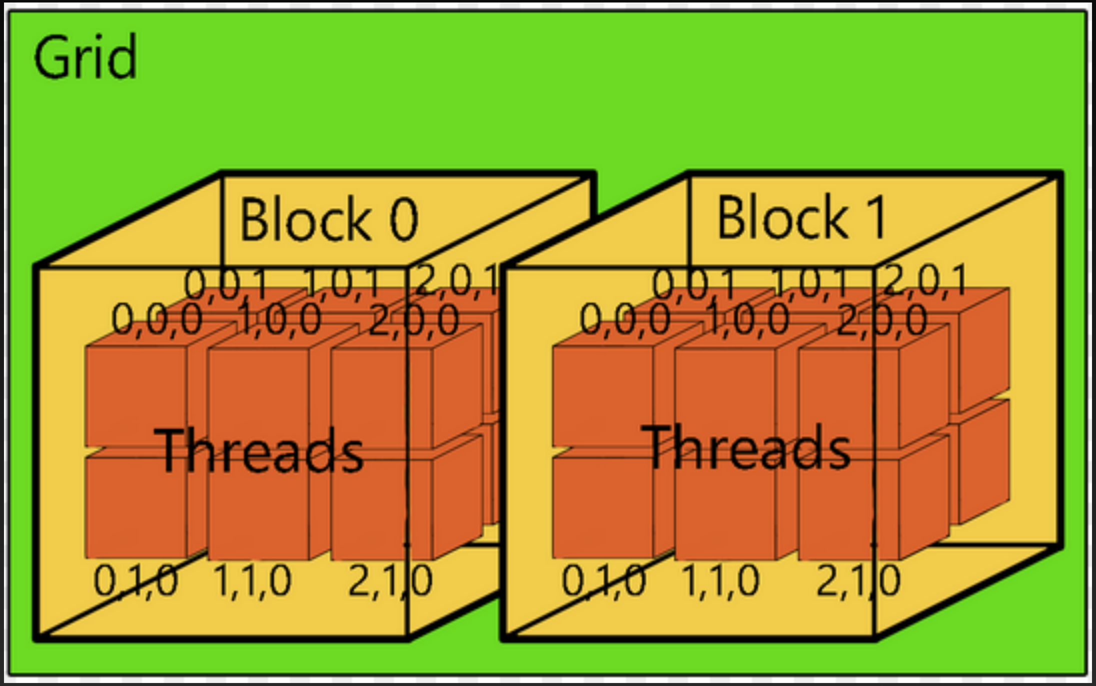
```
__global__ void add(int *d_a, int *d_b, int *d_c) {
    int i= threadIdx.x + blockIdx.x * blockDim.x;

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

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



## Perigo: indices não referncciados...

```
__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

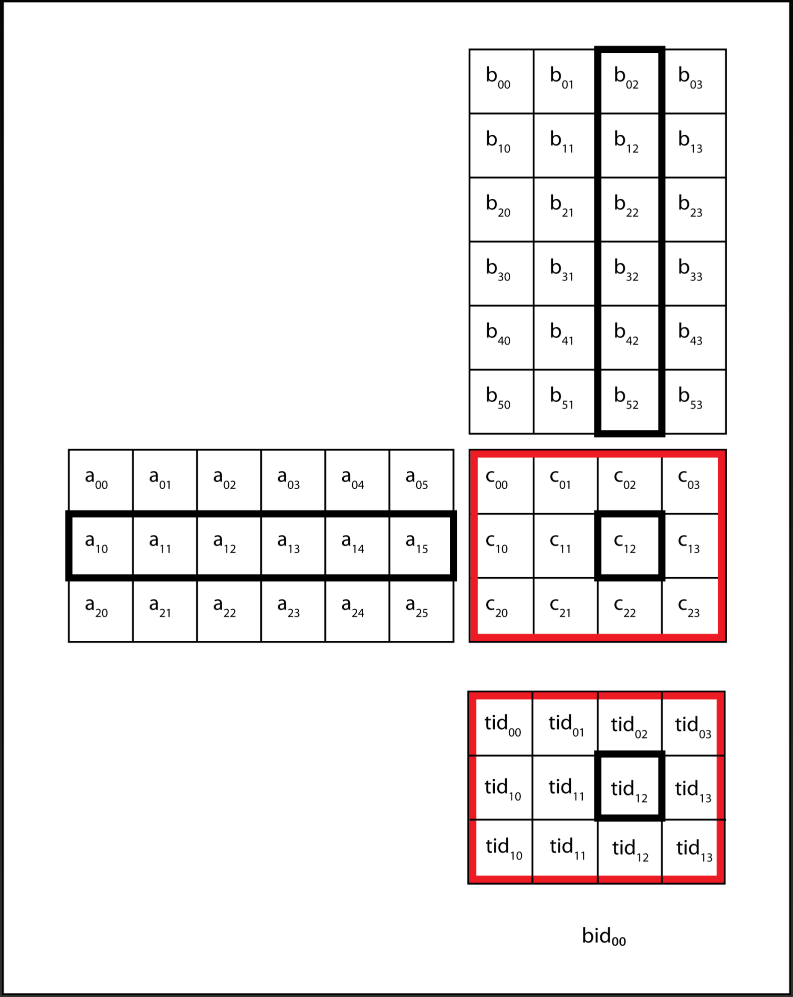


`(threadIdx.x, threadIdx.y, threadIdx.z)`

# 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



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

# 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

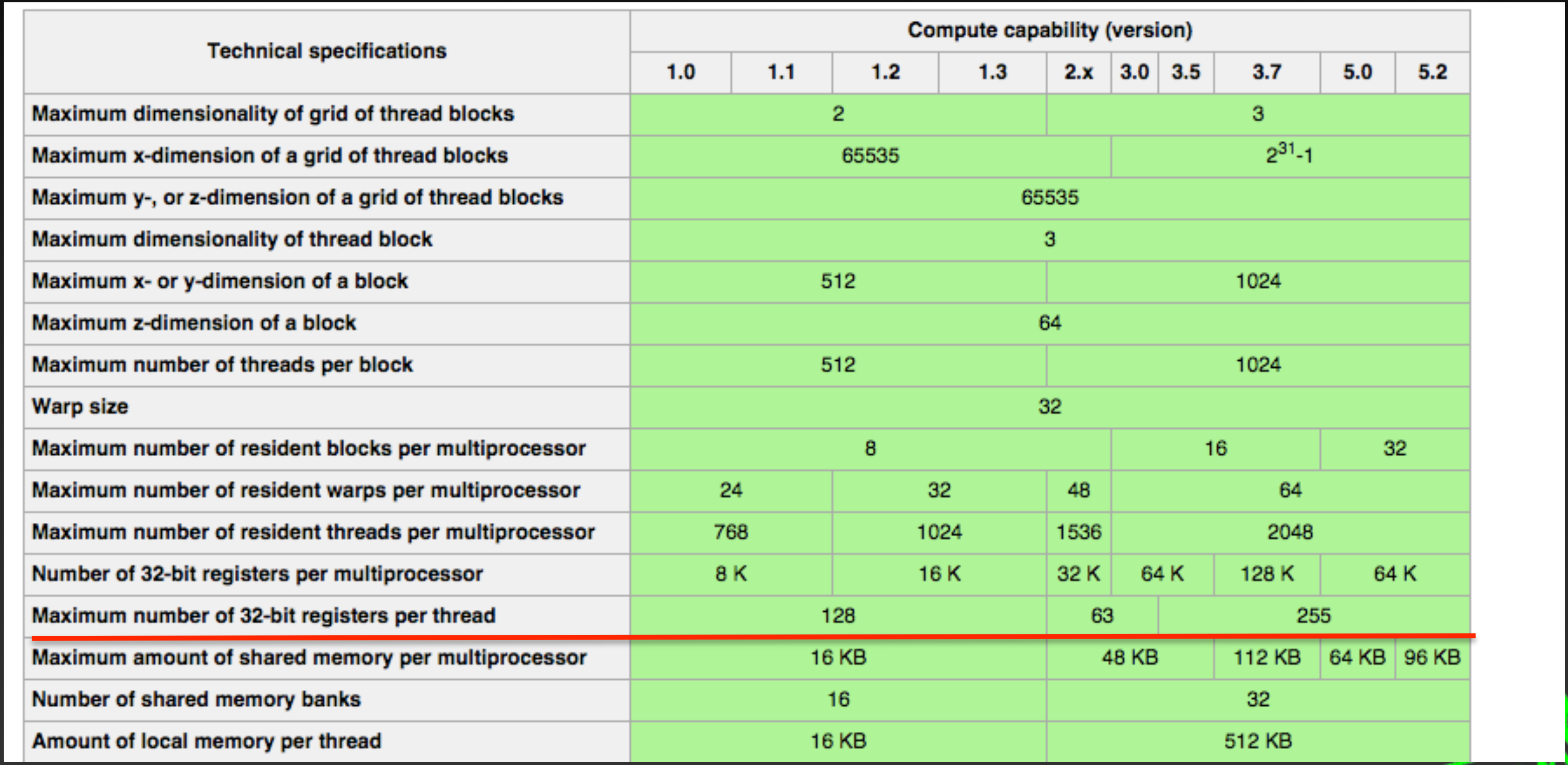
Independent of the Architecture, it consists on 32 threads per warp. Thread multiple of 32 will optimize the occupacy rate

Coalescence is storng in the same warp

Thread Divergence is also strong in the same warp

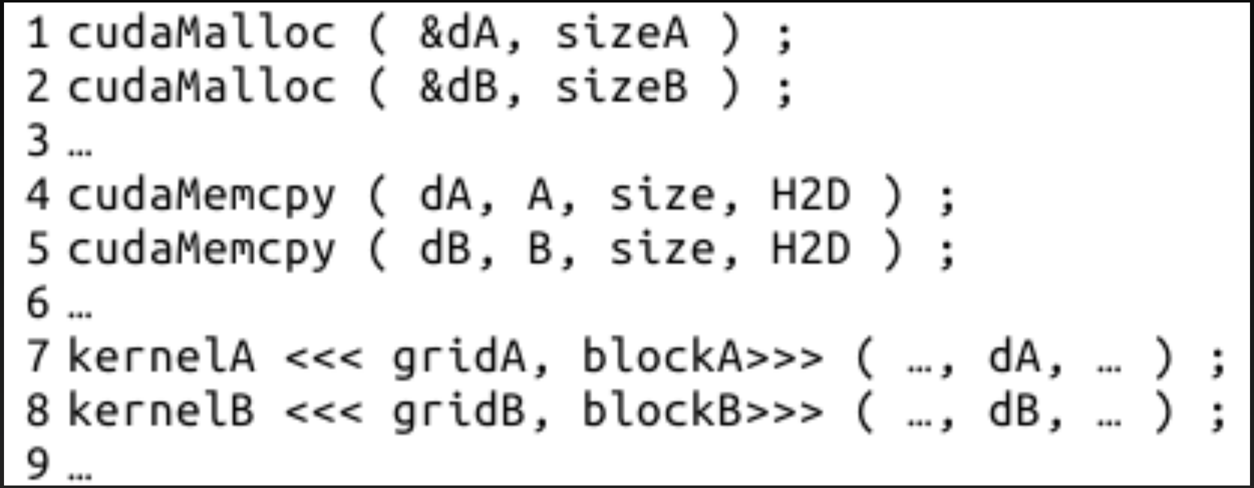
# Aviso importante

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 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();

…
```

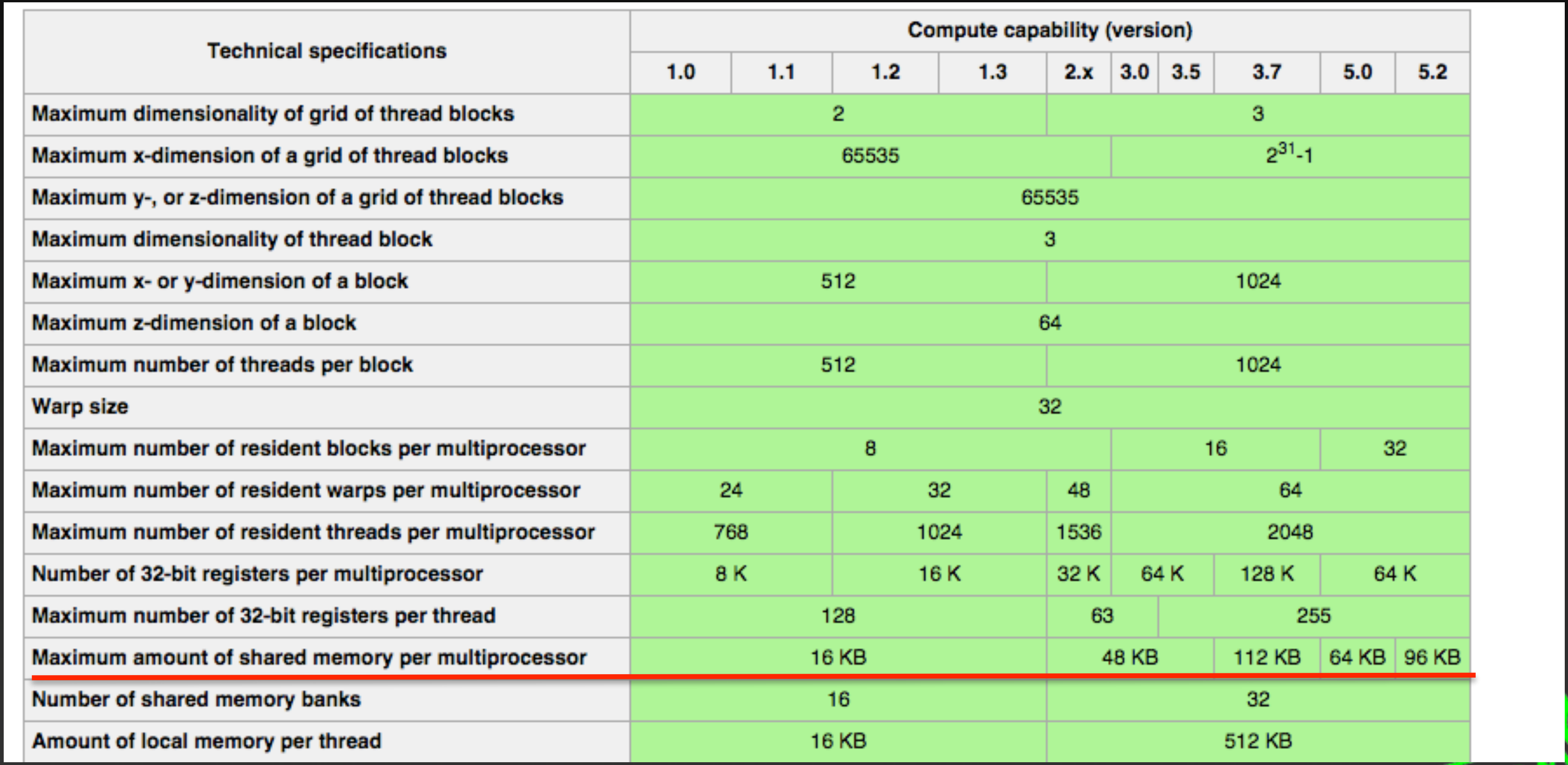
Is this code more efficient than only using the global memory???

## 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**

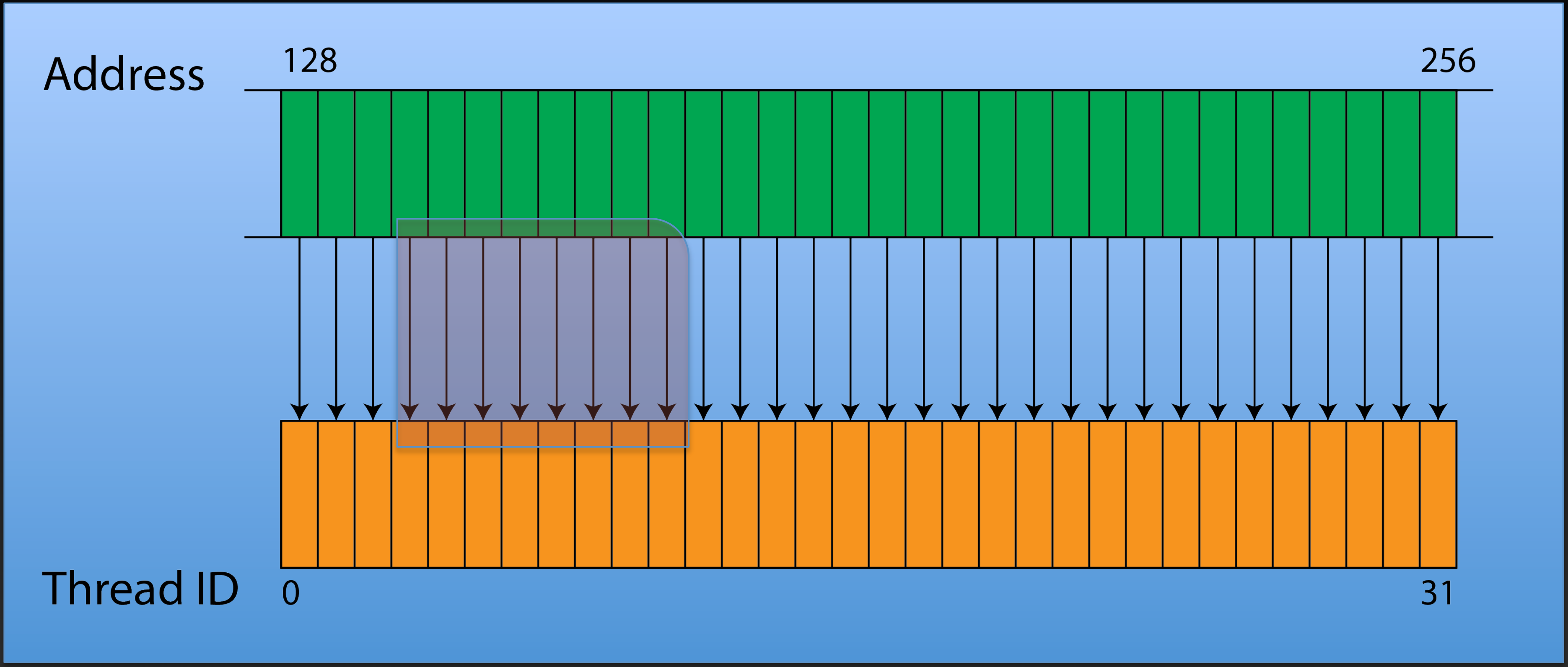


| Technical specifications | Compute capability (version) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.5 | 3.7 | 5.0 | 5.2 |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | | | | |
| Maximum x-dimension of a grid of thread 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



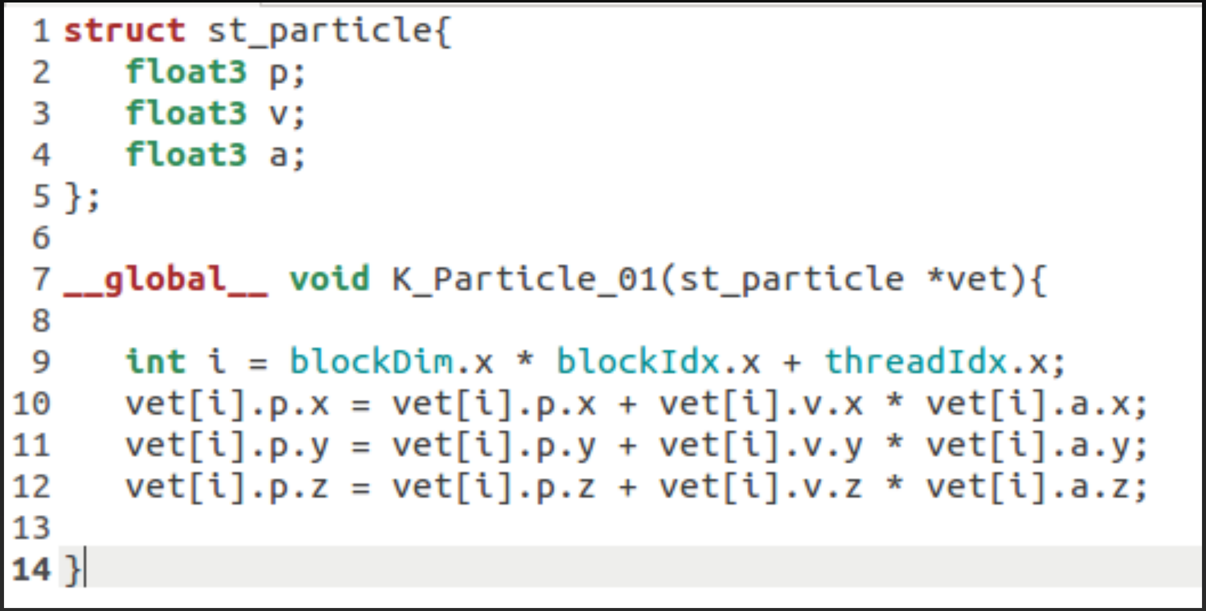
## Otimizando o código

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

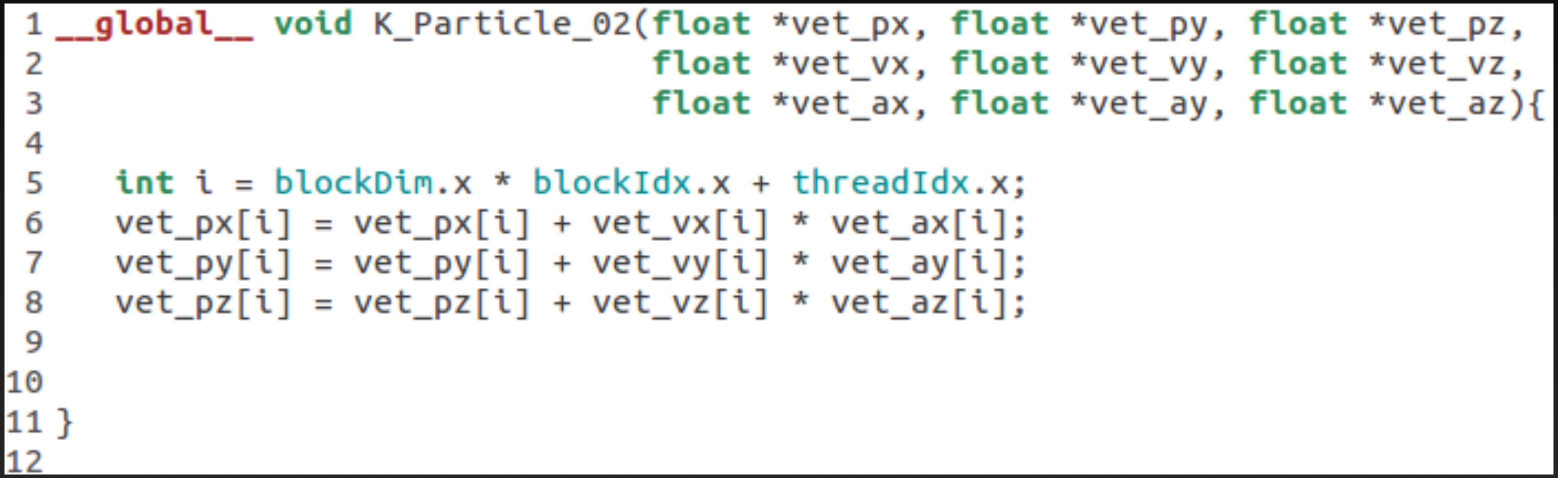
Array of Structures



```
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);
```

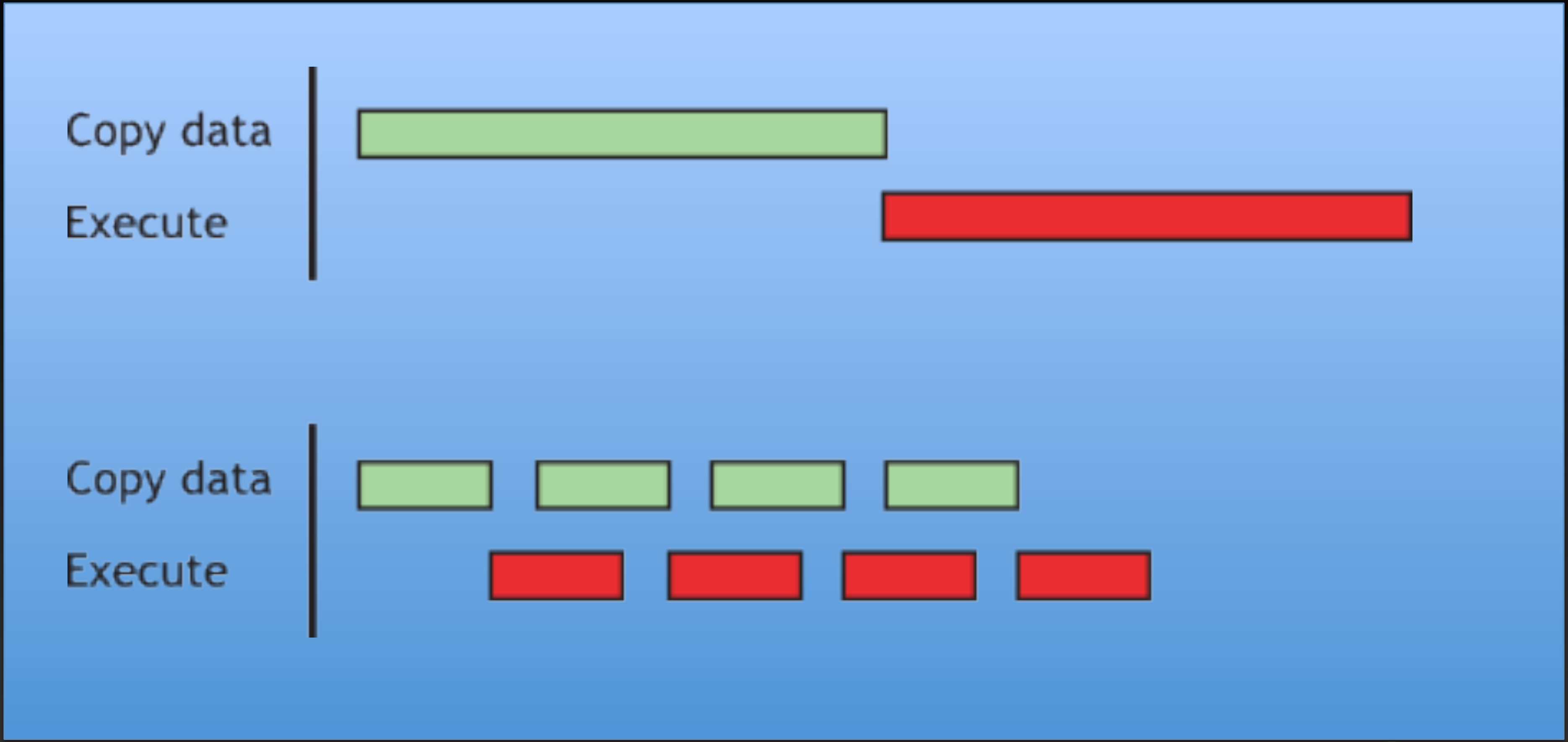
# Streams

```
…
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

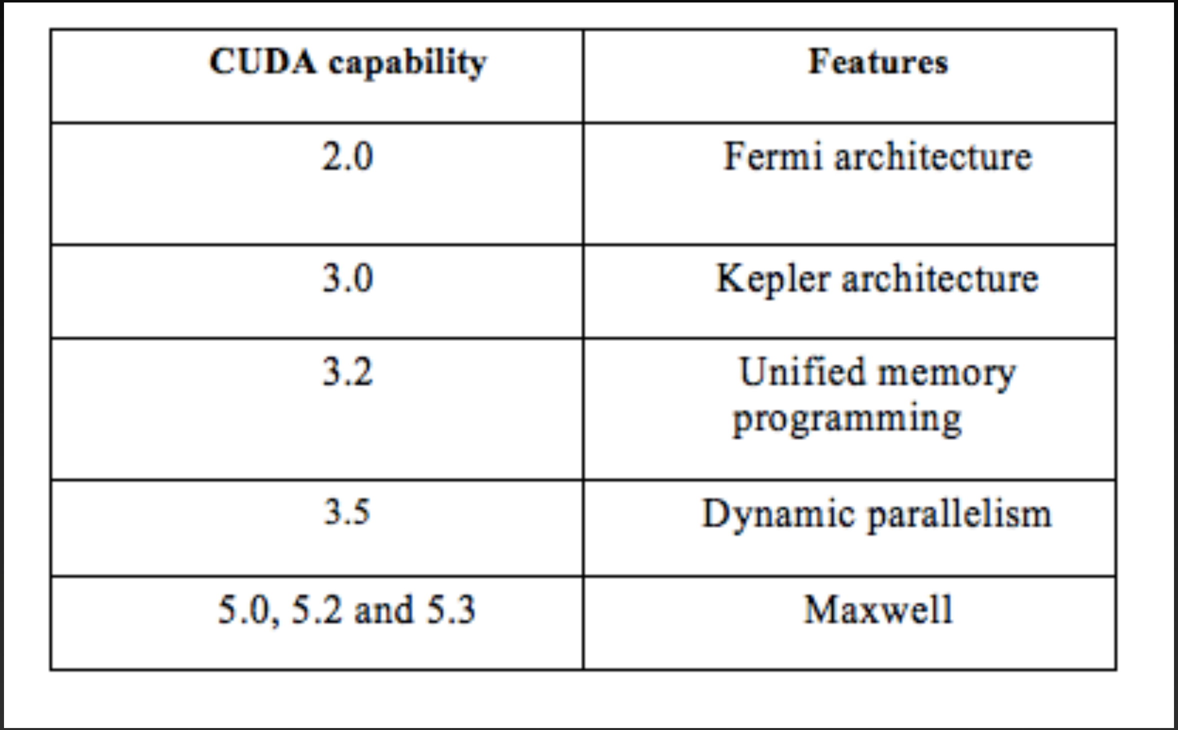
```
    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice,
                     stream1));
    kernel<<<N/256, 256, 0, stream0>>> (d_a1, d_b1, d_c1);

    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

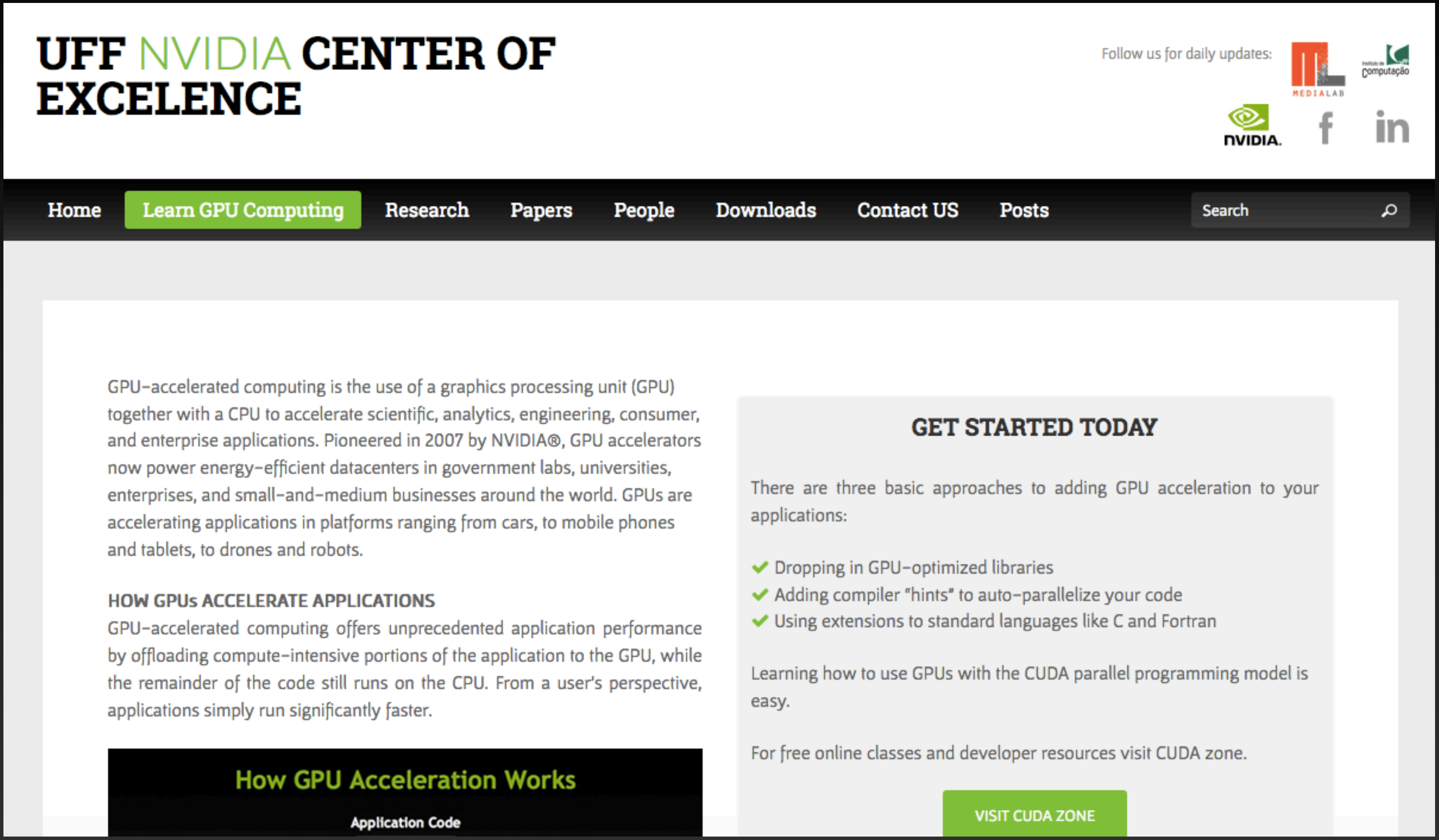
# Mixed Precision

“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-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.

GET STARTED TODAY

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

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

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

For free online classes and developer resources visit CUDA zone.

# GPU Educational Kit

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

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

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

*esteban@ic.uff.br*