

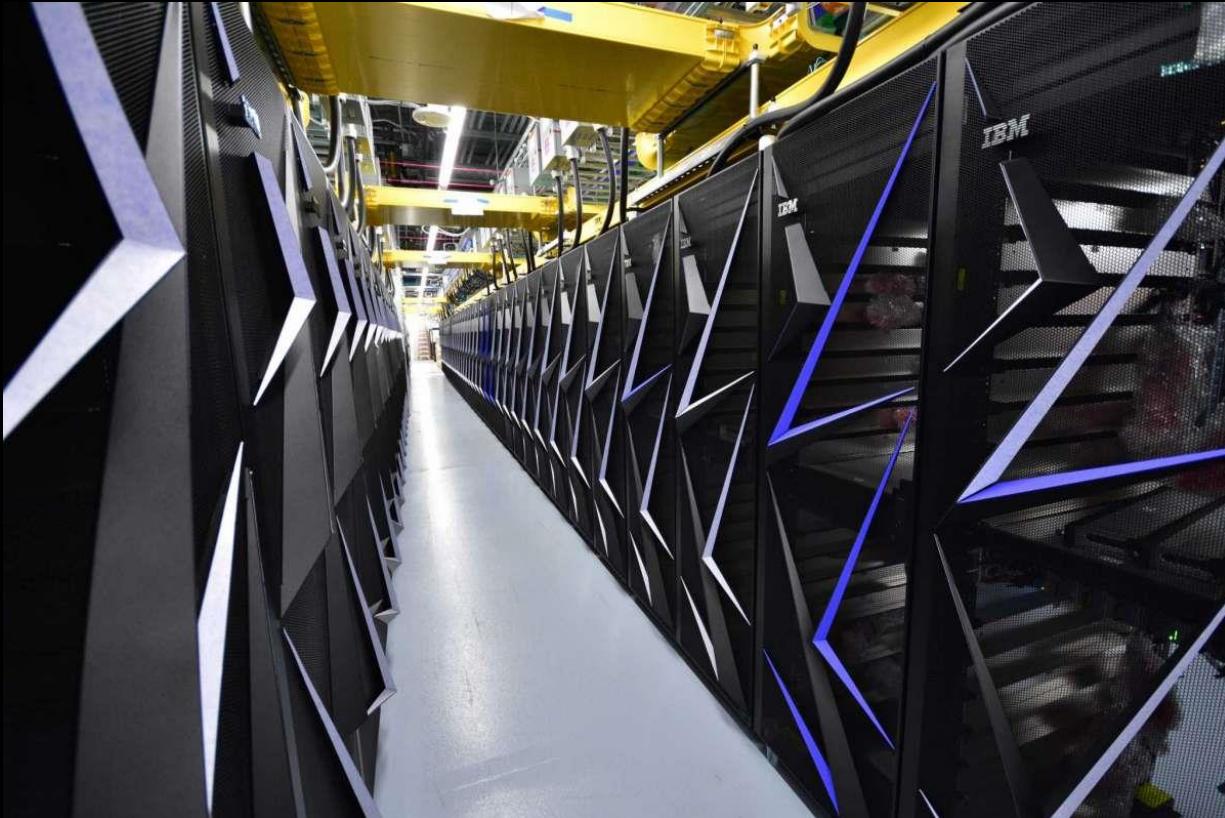
Arquitetura e Programação de GPUs

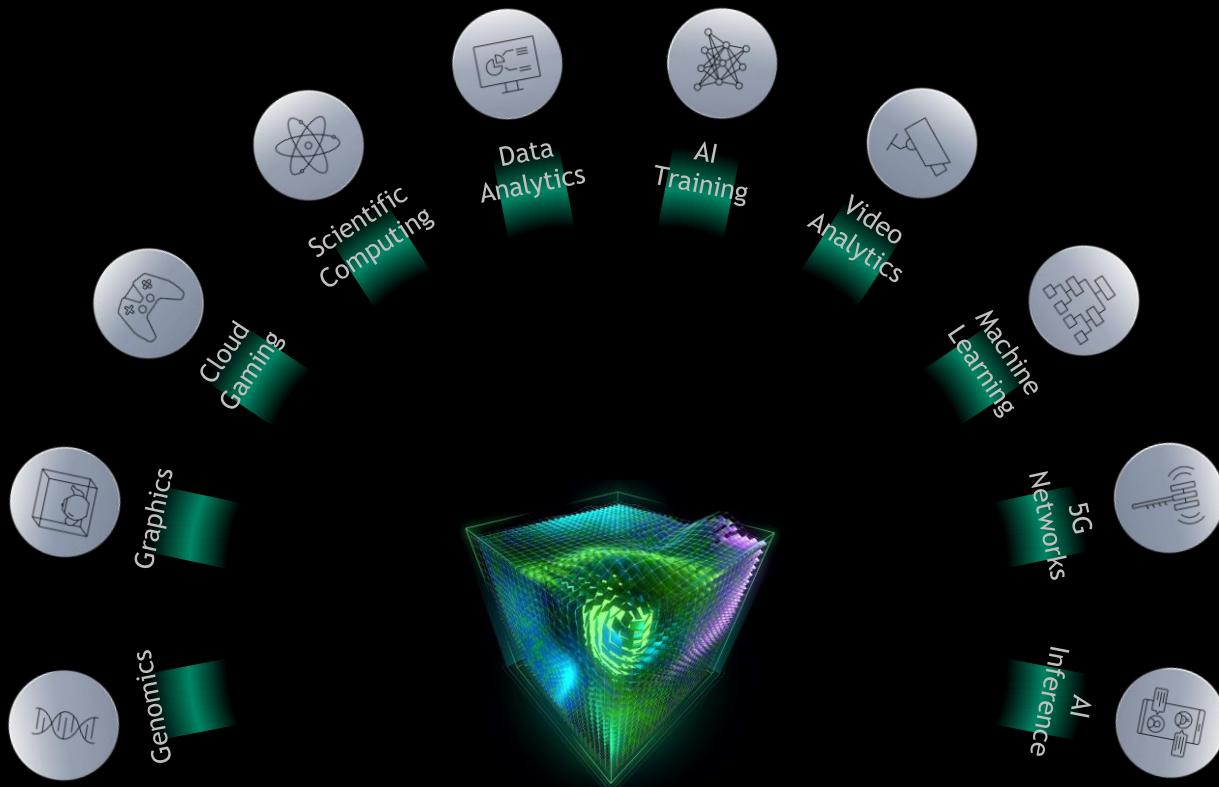
Prof. Esteban Walter Gonzalez Clua, Dr.

NVIDIA Cuda Fellow

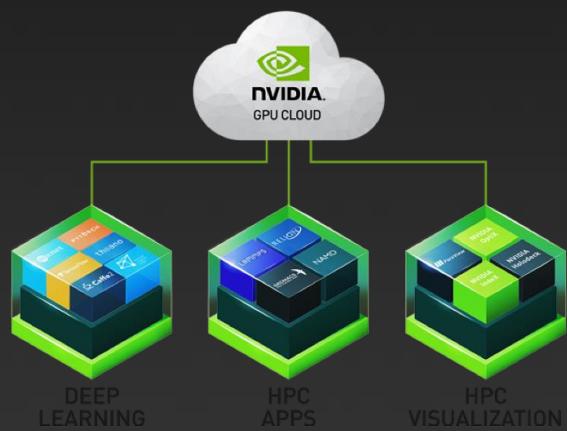
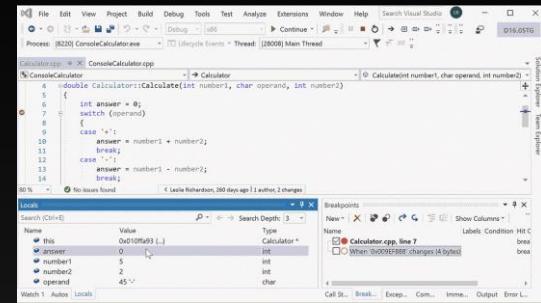
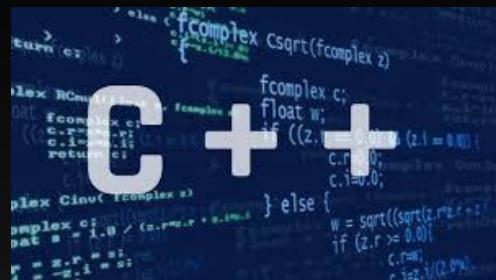
Computer Science Department

Universidade Federal Fluminense – Brazil





What do I need?



Universidade Federal Fluminense

Rio de Janeiro - Brasil



NVIDIA Center of Excellence

Current CCOEs

- Barcelona Supercomputing Center/ Universitat Politecnica de Catalunya
- Chinese Academy of Sciences, IPE
- Georgia Institute of Technology
- Harvard University
- Johns Hopkins University
- Moscow State University
- National Taiwan University
- National Tsing Hua University
- Shanghai Jiao Tong University
- Stanford University
- Technische Universität Dresden
- Tokyo Tech
- Tsinghua University
- Universidade Federal Fluminense
- University of Cambridge
- University of Illinois at Urbana-Champaign
- University of Maryland, College Park
- University of Oxford
- University of Tennessee
- University of Utah



GPU
CENTER OF
EXCELLENCE

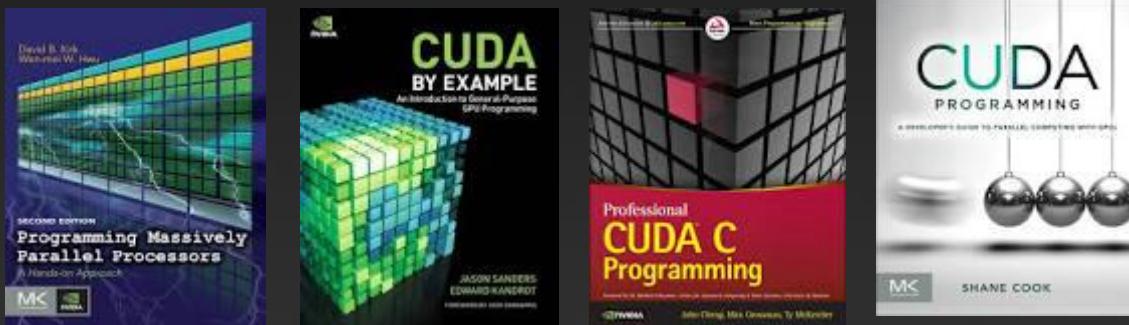


References

The screenshot shows the NVIDIA Developer website's navigation bar at the top, followed by a breadcrumb trail: Home > High Performance Computing > Accelerated Computing - Training. The main title is "Accelerated Computing - Training". Below it, a paragraph explains that the best way to get started is through hands-on courses offered by the NVIDIA Deep Learning Institute (DLI). It mentions that courses include dedicated access to a fully-configured GPU accelerated workstation in the cloud and require only a web browser and an internet connection – no GPU required! A note below states that once you've gotten started, you can dive deeper into the How-To guides for specific applications.

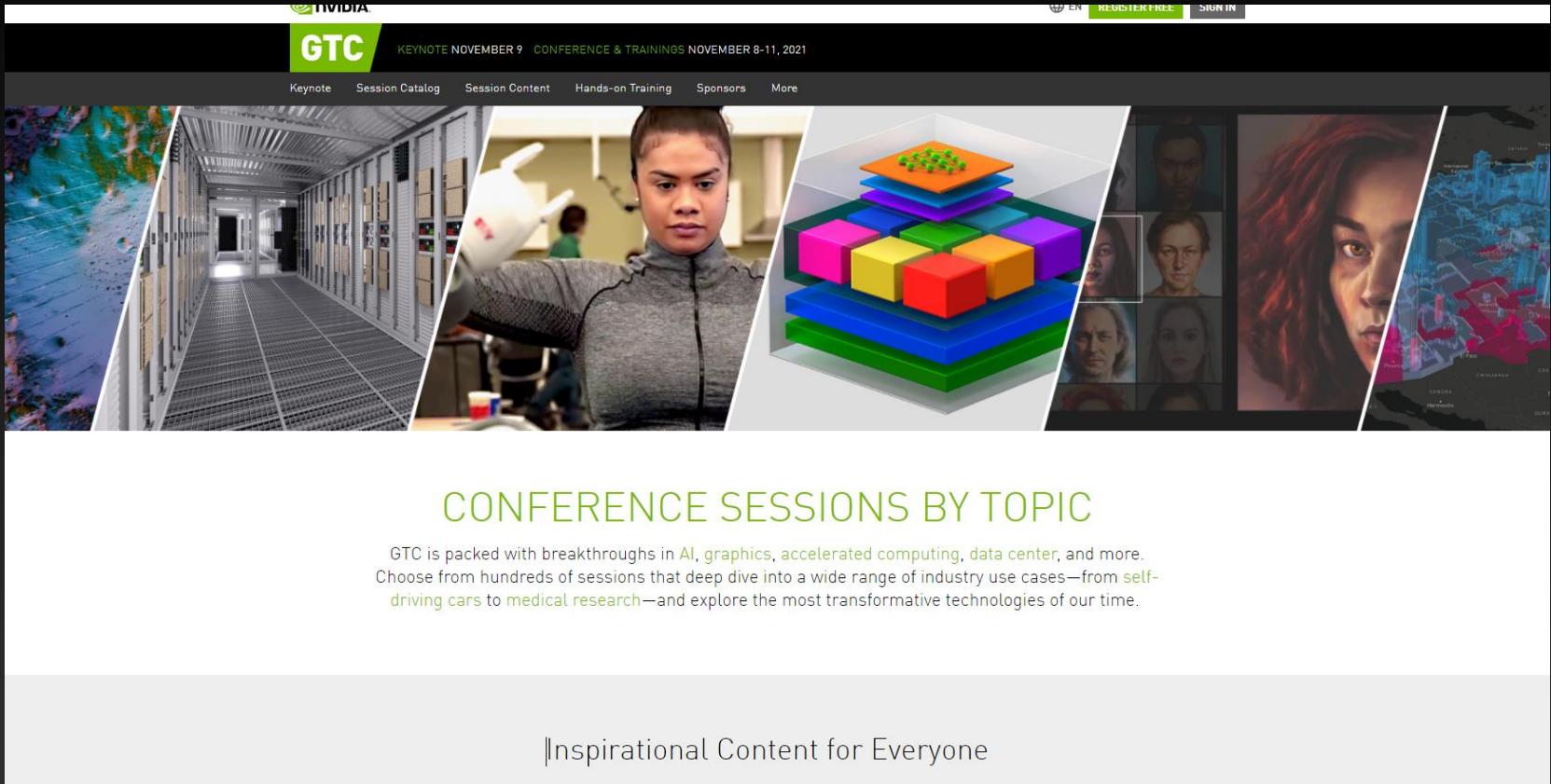
Accelerating your Applications

Optimized Libraries Drop-in, Industry standard libraries replace MKL, IPP, FFTW and other widely used libraries. Some feature automatic multi-GPU scaling. Get Started with GPU-Accelerated Libraries	Compiler Directives Use OpenACC – open standard directives for accelerated computing. Get Started With: <ul style="list-style-type: none">• C/C++ using CUDA C• Fortran using CUDA Fortran	Programming Languages Develop your own parallel applications and libraries using a programming language you already know. Get Started With: <ul style="list-style-type: none">• C/C++ using CUDA C• Fortran using CUDA Fortran
--	--	--



GTC

<https://www.nvidia.com/gtc/sessions/>



The screenshot shows the NVIDIA GTC website homepage. At the top, there's a navigation bar with the NVIDIA logo, a green "GTC" button, and links for "REGISTER FREE" and "SIGN IN". Below the navigation is a banner featuring several images: a server room, a woman in a lab coat, a 3D model of a neural network, a grid of faces, and a woman with red hair. The main heading "CONFERENCE SESSIONS BY TOPIC" is in large green text. Below it, a paragraph describes the conference's focus on AI, graphics, accelerated computing, data center, and more, with examples like self-driving cars and medical research.

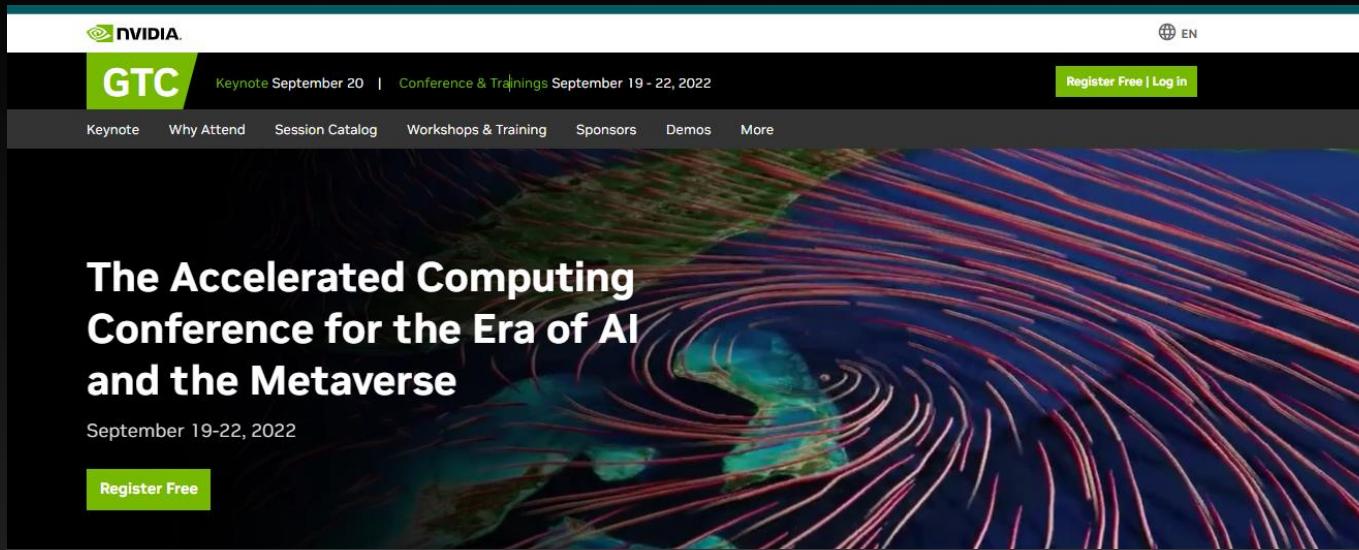
CONFERENCE SESSIONS BY TOPIC

GTC is packed with breakthroughs in [AI](#), [graphics](#), [accelerated computing](#), [data center](#), and more. Choose from hundreds of sessions that deep dive into a wide range of industry use cases—from [self-driving cars](#) to [medical research](#)—and explore the most transformative technologies of our time.

Inspirational Content for Everyone



References



Check out these recommended conference sessions.

I am interested in Accelerated Computing & Dev Tools ▾

Tuesday, September 20
11:00 a.m. - 11:50 a.m. PDT

CUDA Programming Model for Hopper Architecture

This session will introduce new features in CUDA for programming Hopper architecture. The new programming model for Hopper is more hierarchical and asynchronous. We'll also look at optimizing CUDA applications...

[Learn More](#)

Monday, September 19
10:00 a.m. - 10:50 a.m. PDT

How CUDA Programming Works

Come for an introduction to programming the GPU, led by the lead architect of CUDA. CUDA's unique in being a programming language designed and built hand-in-hand with the hardware that it runs on. Stepping up from...

[Learn More](#)

Wednesday, September 21
9:00 a.m. - 9:50 a.m. PDT

Developing HPC Applications with Standard C++, Fortran, and Python

Learn to develop for GPU and non-GPU systems using the latest features in the C++, Fortran, and Python programming languages. Applications written using standard programming languages can take advantage of GPU...

[Learn More](#)

[View More Sessions](#)



NVIDIA Deep Learning Institute

<https://www.nvidia.com/en-us/training/online/>

The screenshot shows the NVIDIA Deep Learning Institute website with a dark theme. At the top, there's a navigation bar with links for DEEP LEARNING INSTITUTE, ONLINE COURSES, INSTRUCTOR-LED WORKSHOPS, EDUCATOR PROGRAMS, ENTERPRISE SOLUTIONS, and RESOURCES. Below this is a secondary navigation bar for Self-Paced Online Training, with links for COURSES, BENEFITS, PARTNERS, and RESOURCES. A promotional banner at the top of the main content area says "Instructor-led DLI Workshops at GTC this November for just \$149. Seats are filling up fast. Enroll now." To the right of the banner is a close button (X). The main content area is titled "ONLINE TRAINING COURSES" and features a grid of course cards. The courses are categorized by topic: FUNDAMENTALS, AUTONOMOUS DRIVING, INTELLIGENT VIDEO ANALYTICS, and HEALTHCARE. Each card includes the course title, duration, cost, required tools, and whether a certificate is available. Some courses are marked as "NEW" or "POPULAR".

Category	Course Title	Duration	Cost	Tools	Certificate Available
FUNDAMENTALS	Building A Brain in 10 Minutes!	10 minutes	Free	Python	★ NEW
FUNDAMENTALS	Getting Started with Deep Learning	8 hours	\$90	TensorFlow 2 with Keras, Pandas	★ Certificate Available
FUNDAMENTALS	Getting Started with AI on Jetson Nano	8 hours	Free	PyTorch, NVIDIA® Jetson Nano™	★ POPULAR Certificate Available
AUTONOMOUS DRIVING	Integrating Sensors with NVIDIA DRIVE®	2 hours	\$30	0++, DriveWorks	★ NEW
FUNDAMENTALS	Optimization and Deployment of TensorFlow Models with TensorRT	2 hours	\$30	TensorFlow, Keras, NVIDIA TensorRT™	★ NEW
FUNDAMENTALS	Deep Learning at Scale with Horovod	2 hours	\$30	Horovod, TensorFlow, Keras, Python	★ POPULAR
FUNDAMENTALS	Getting Started with Image Segmentation	2 hours	\$30	TensorFlow	★ NEW
FUNDAMENTALS	Modeling Time-Series Data with Recurrent Neural Networks in Keras	2 hours	\$30	Keras	★ NEW
INTELLIGENT VIDEO ANALYTICS	AI Workflows for Intelligent Video Analytics with DeepStream	2 hours	\$30	NVIDIA DeepStream 3.0, C++, GStreamer	★ POPULAR
HEALTHCARE	Medical Image Classification Using Deep Learning	2 hours	\$30	TensorFlow, Keras	★ NEW
HEALTHCARE	Image Classification with Deep Learning	2 hours	\$30	TensorFlow, Keras	★ POPULAR

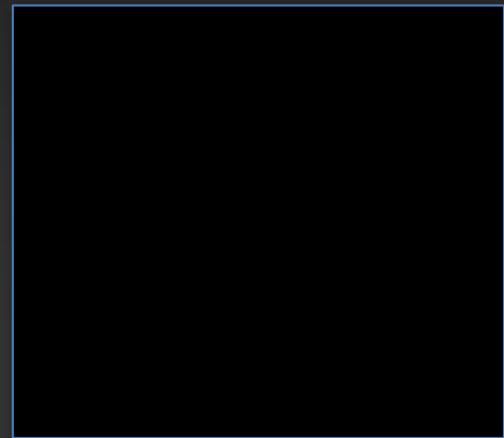
Why GPUs?



=



Introduction



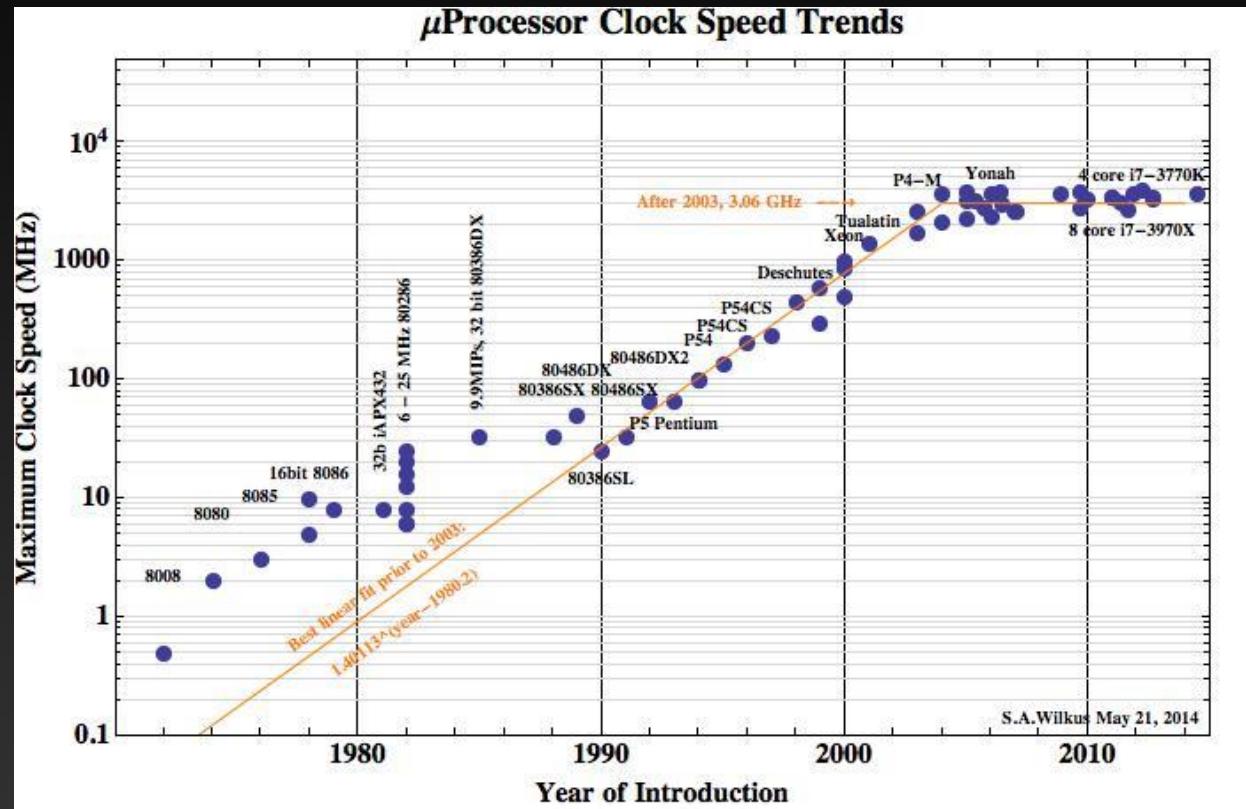
Why GPUs?



=

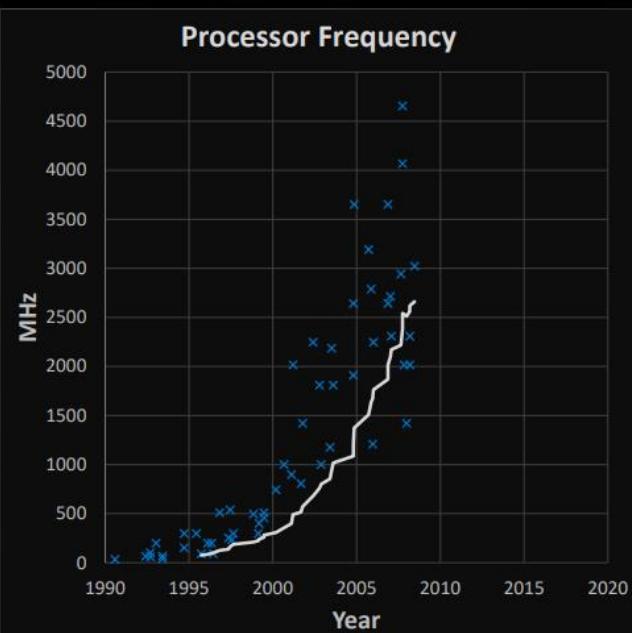


Clock Speed over the years...



Single-Thread Era (until ~2007)

Straight-Line Code

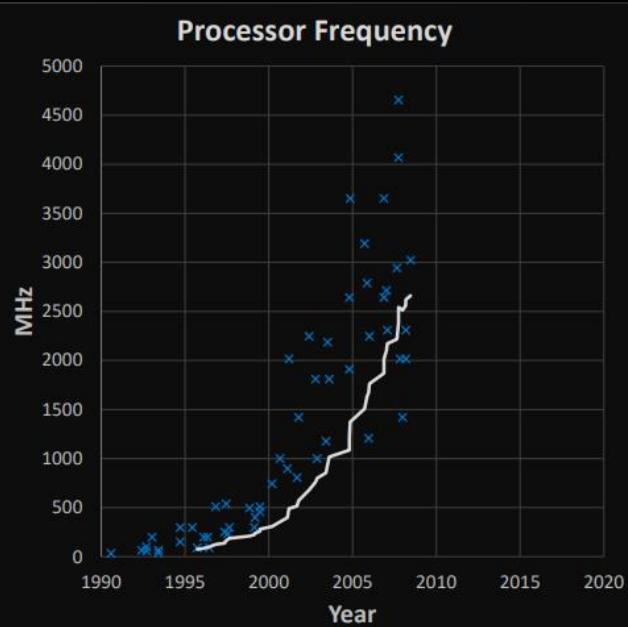


Source: Karl Rupp "40 years of microprocessor trend data"
<https://github.com/karlrupp/microprocessor-trend-data>



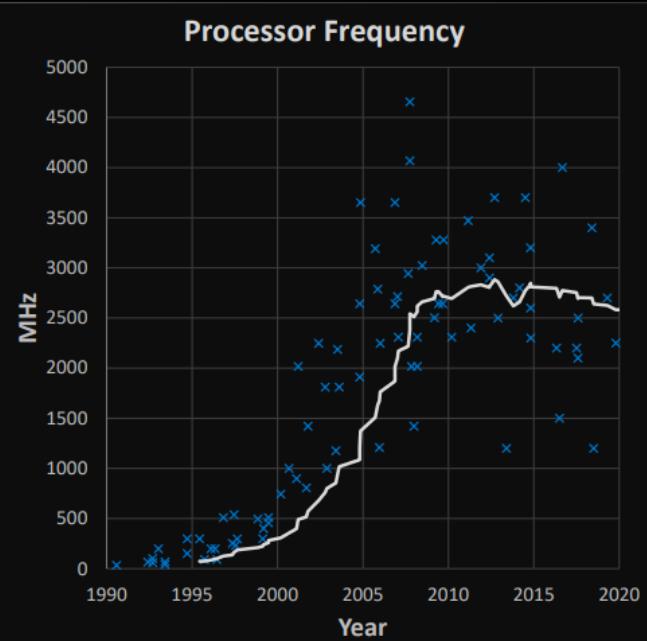
Single-Thread Era (until ~2007)

Straight-Line Code



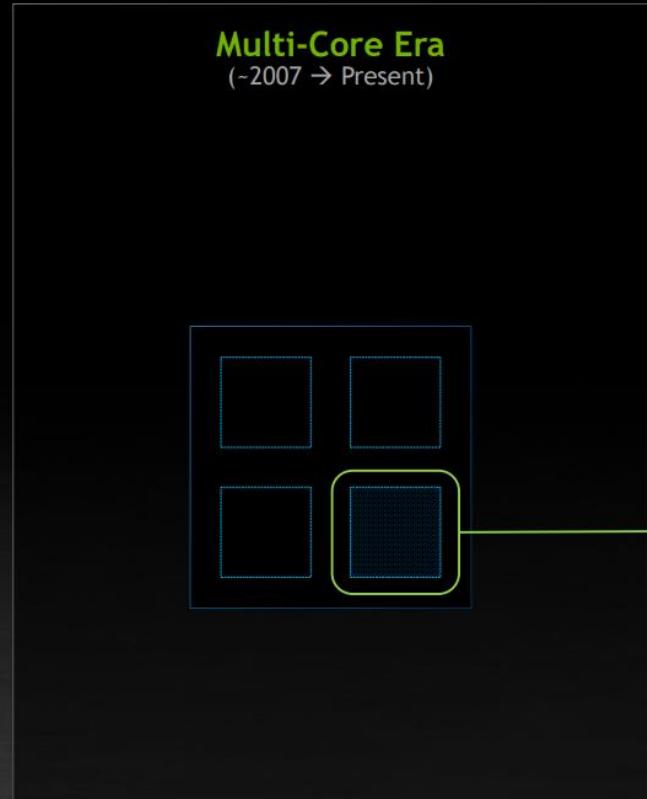
Single-Thread Era (until ~2007)

Straight-Line Code



Source: Karl Rupp "40 years of microprocessor trend data"
<https://github.com/karlrupp/microprocessor-trend-data>

Source: Karl Rupp "40 years of microprocessor trend data"
<https://github.com/karlrupp/microprocessor-trend-data>



Lesson #1

GPUs don´t want to do a lot of different things at the Same time.... They want to do a lot of jobs of the Same thing...



CPU and GPU: Different evolution chains...

CPU



Latency

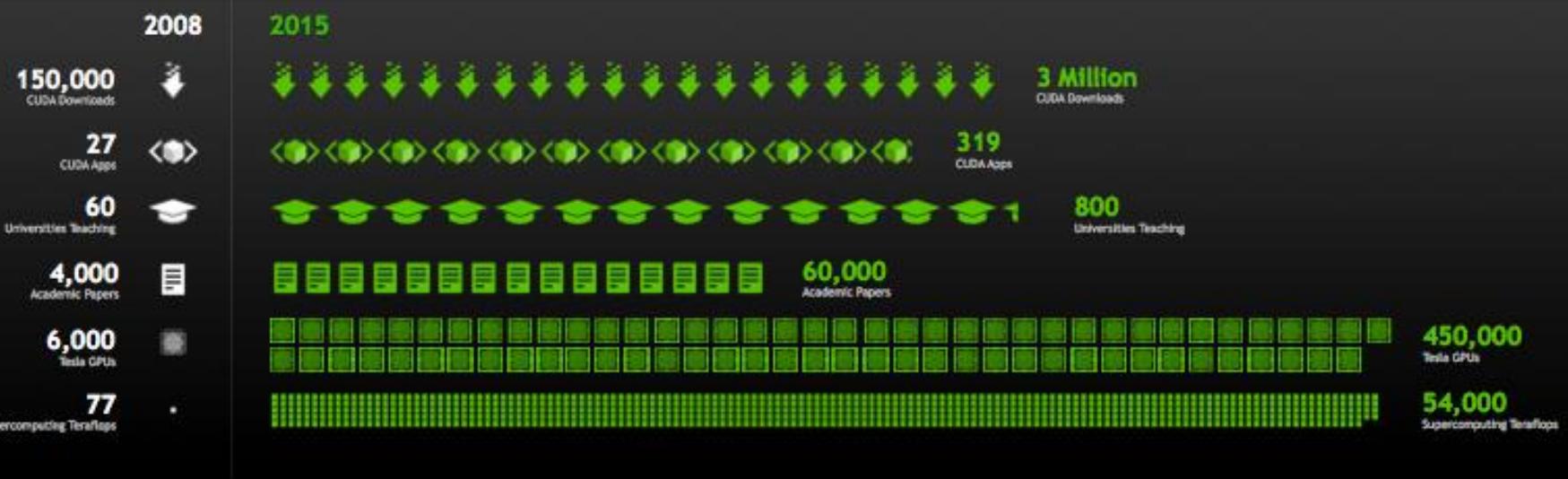


GPU



Throughput





TOP 10 Sites for November 2000

For more information about the sites and systems in the list, click on the links or view the [complete list](#).

Release

- [The List](#)
- [Press Release \(PDF\)](#)
- [Press Release](#)
- [List highlights](#)

Downloads

- [TOP500 List \(XML\)](#)
- [TOP500 List \(Excel\)](#)
- [TOP500 Poster](#)
- [Poster in PDF](#)

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



TOP 10 Sites for November 2000

For more information about the sites and systems in the list, click on the links or view the [complete list](#).

Release

- [The List](#)
- [Press Release \(PDF\)](#)
- [Press Release](#)
- [List highlights](#)

Downloads

- [TOP500 List \(XML\)](#)
- [TOP500 List \(Excel\)](#)
- [TOP500 Poster](#)
- [Poster in PDF](#)

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



1 Million Watts



TOP 10 Sites for November 2000

For more information about the sites and systems in the list, click on the links or view the [complete list](#).

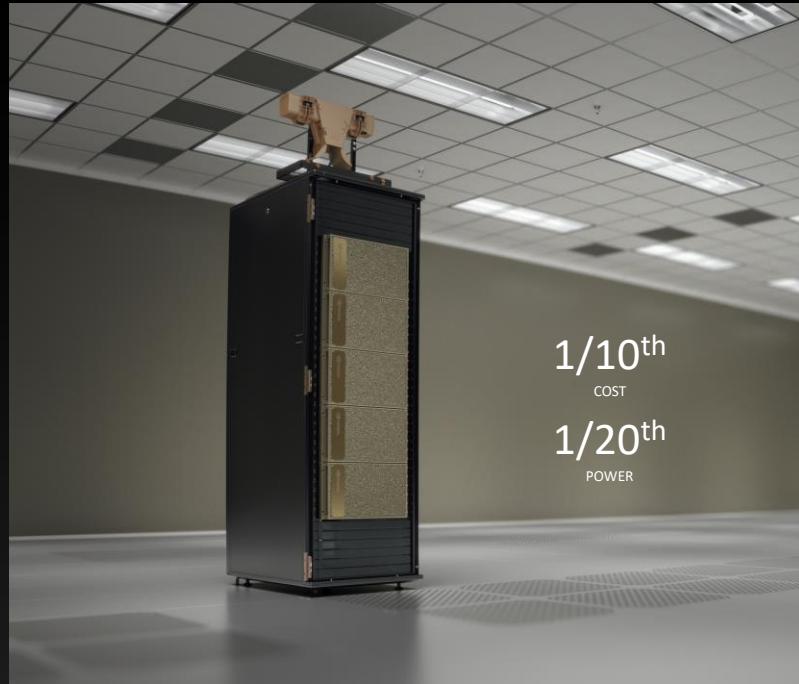
Release

- [The List](#)
- [Press Release \(PDF\)](#)
- [Press Release](#)
- [List highlights](#)

Downloads

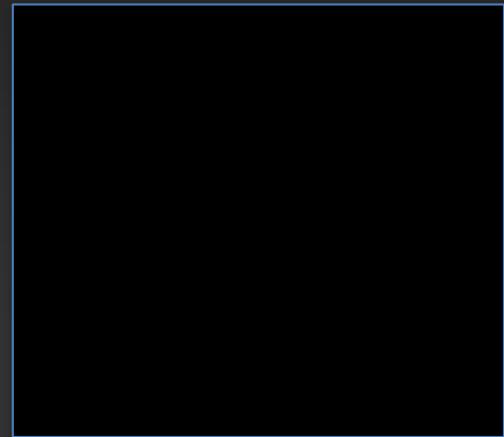
- [TOP500 List \(XML\)](#)
- [TOP500 List \(Excel\)](#)
- [TOP500 Poster](#)
- [Poster in PDF](#)

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



0.8 Pflops for HPC, 50 PFLOPS for Tensor cores
= 160 Laurences for HPC, 8000 Laurences for
Tensor 35 x less energy

Energy matters!



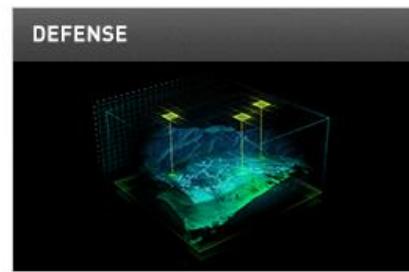
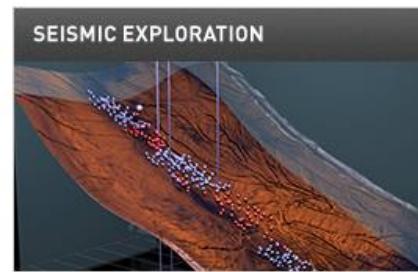
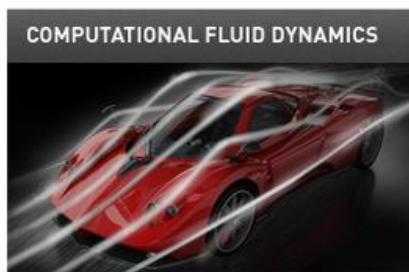
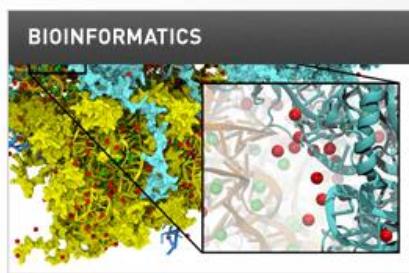
Energy Matters!...



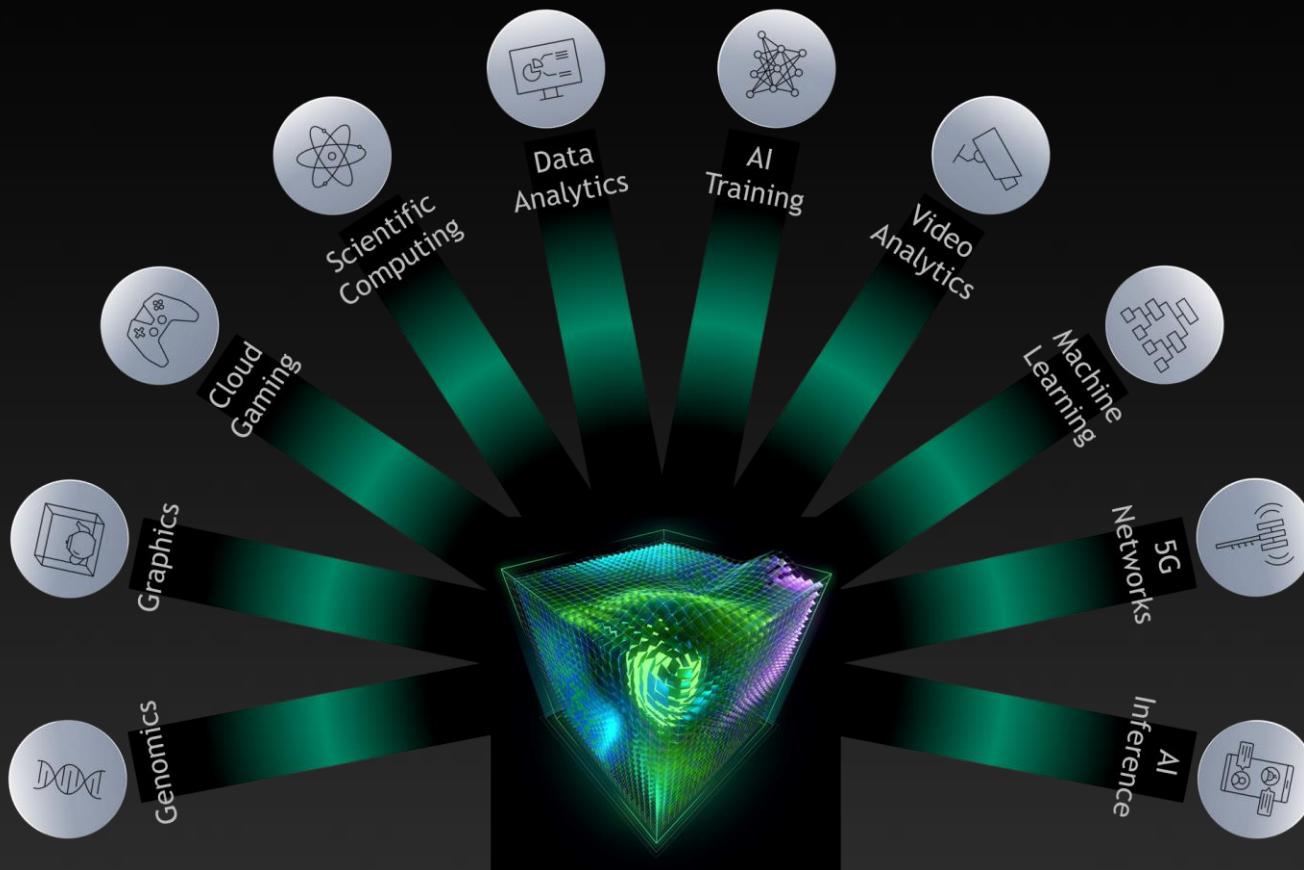
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438

1MW ~1000 casas

Broad Industry Adoption of GPUs



Broad Industry Adoption of GPUs



Broad Industry Adoption of GPUs



NVIDIA Xavier

NVIDIA Jetson Xavier Developer Kit



NVIDIA Jetson Xavier is the latest addition to the Jetson platform. It's an AI computer for autonomous machines, delivering the performance of a GPU workstation in an embedded module under 30W. With multiple operating modes at 10W, 15W, and 30W, Jetson Xavier has greater than 10x the energy efficiency and more than 20x the performance of its predecessor, the Jetson TX2.

Jetson Xavier is designed for robots, drones and other autonomous machines that need maximum compute at the edge to run modern AI workloads and solve problems in manufacturing, logistics, retail, service, agriculture and more. Jetson Xavier is also suitable for smart city applications and portable medical devices.

Broad Industry Adoption of GPUs



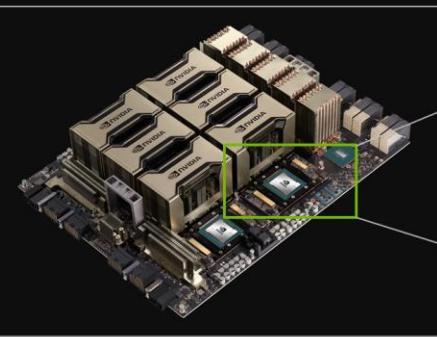
Super computing for all sizes



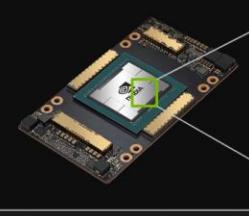
Hierarchy of Scales



Multi-System Rack
Unlimited Scale



Multi-GPU System
8 GPUs



Multi-SM GPU
108 Multiprocessors

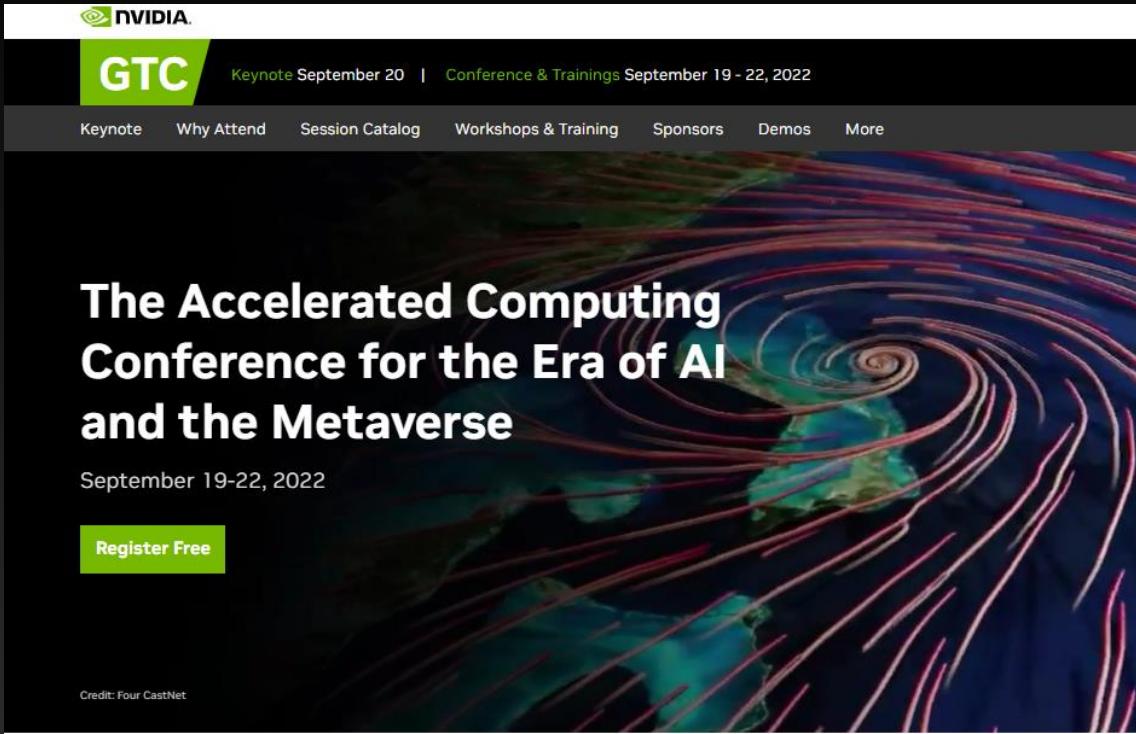


Multi-Core SM
2048 threads

Roadmap



Powering the AI and the Metaverse



The image shows a screenshot of the NVIDIA GTC 2022 website. At the top, the NVIDIA logo is on the left, followed by the letters "GTC" in white on a green background. To the right, it says "Keynote September 20 | Conference & Trainings September 19 - 22, 2022". Below this is a dark navigation bar with links: Keynote, Why Attend, Session Catalog, Workshops & Training, Sponsors, Demos, and More. The main content area features a large, abstract, swirling background image in shades of blue, green, and pink. Overlaid on this image is the text "The Accelerated Computing Conference for the Era of AI and the Metaverse" in white. Below this, the dates "September 19-22, 2022" are shown. At the bottom left of the main area is a green button with the text "Register Free". At the very bottom left of the entire page, there is small text that reads "Credit: Four CastNet".



NVIDIA Omniverse

 Products Solutions Industries For You

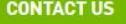
SHOP DRIVERS SUPPORT   

OMNIVERSE SOLUTIONS ▾ PLATFORM ▾ LEARN ▾ COMMUNITY ▾ SUPPORT ▾

NVIDIA OVX  OVERVIEW PORTFOLIO BENEFITS

NVIDIA OVX

Purpose-Built for the Demands of Omniverse Digital Twins





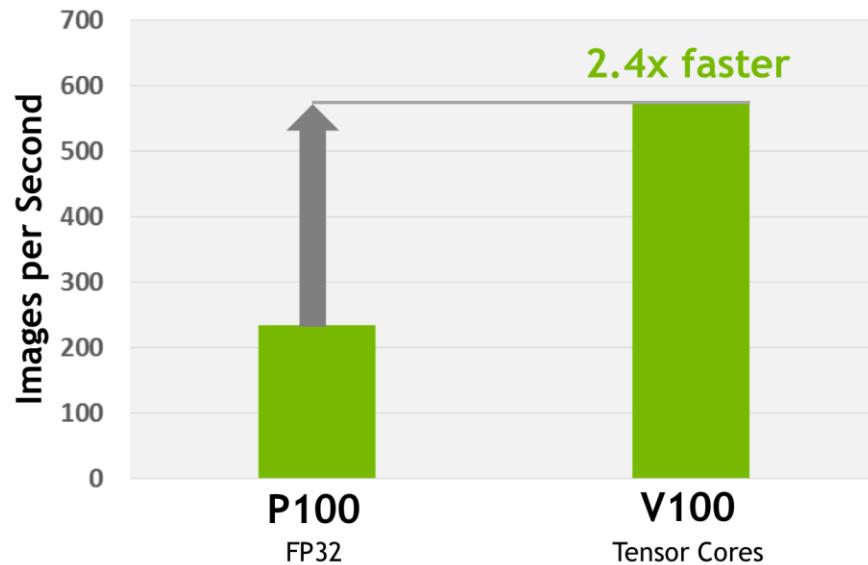
A Data Center Computing System Designed for Industrial Digital Twins

Digital twins revolutionize how enterprises design, test, and optimize complex systems and processes, requiring multiple autonomous systems interacting in the same time-space. NVIDIA® OVX™ is purpose-built to power large-scale industrial twins from the data center to create and

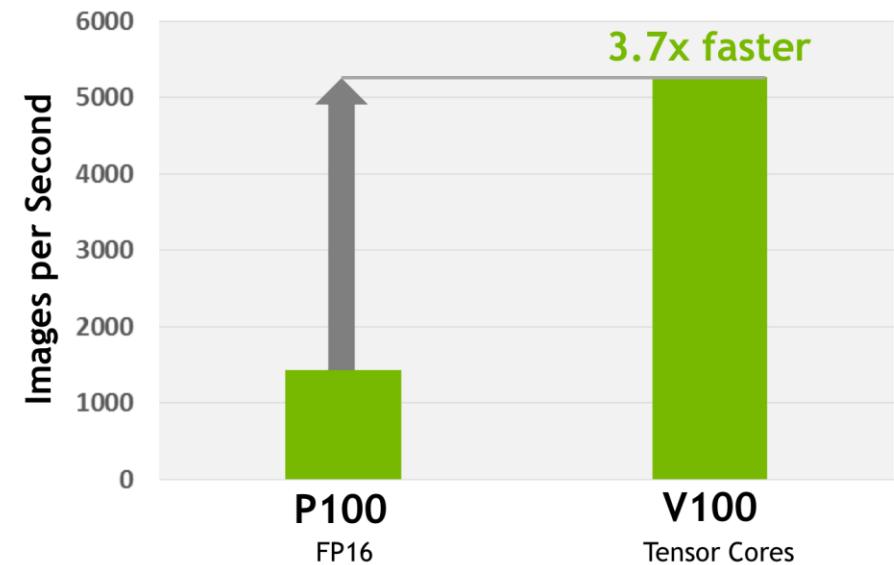


Architecture dedicated to Deep Learning

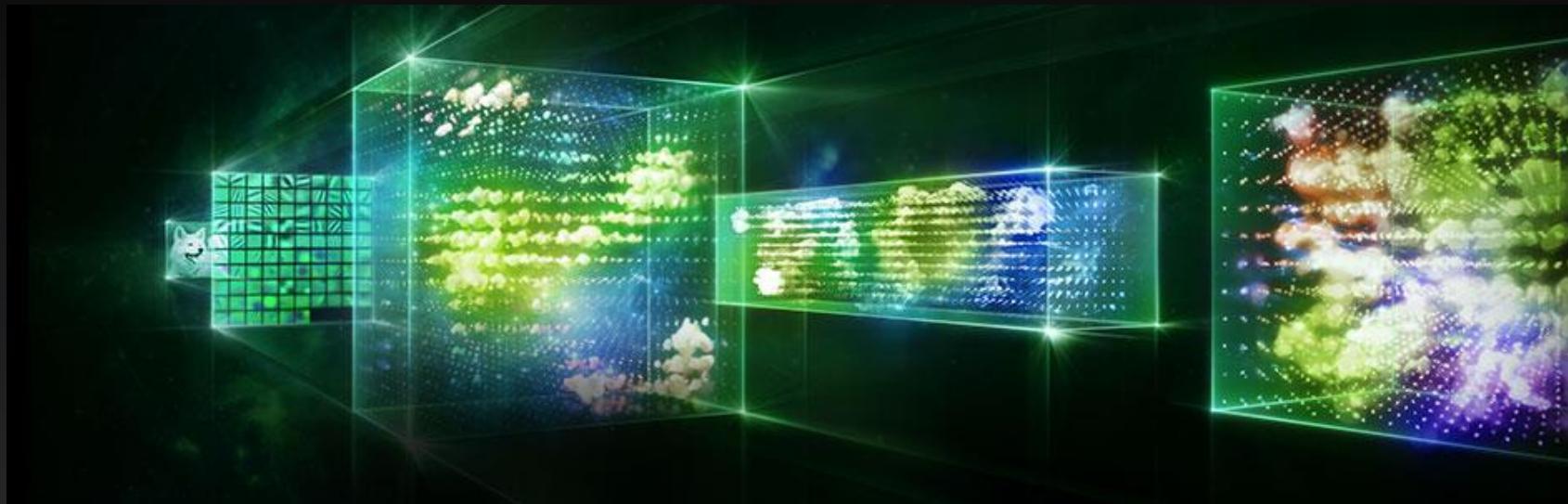
ResNet-50 Training



ResNet-50 Inference
TensorRT - 7ms Latency



Deep Learning Revolution

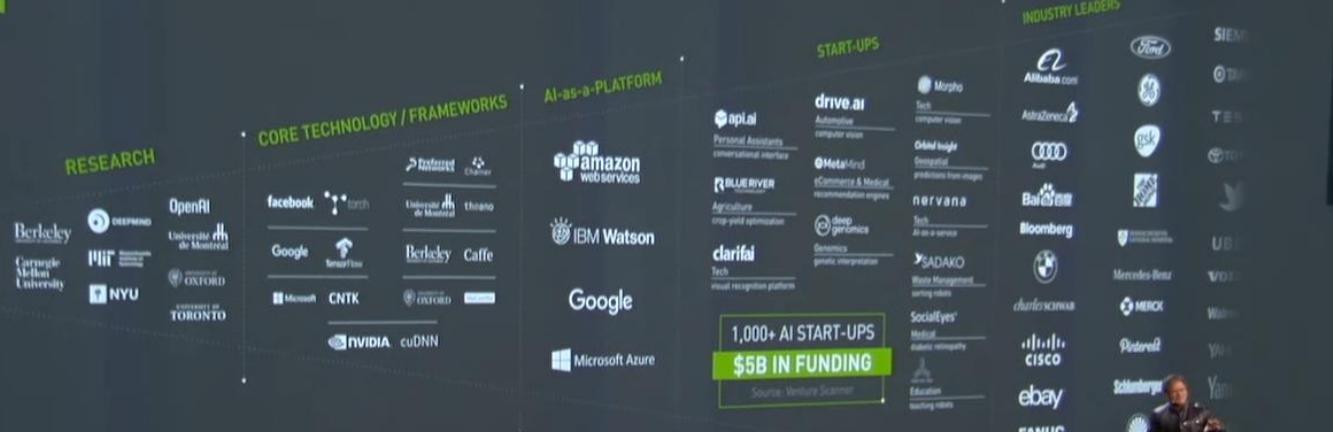


Deep Learning Revolution

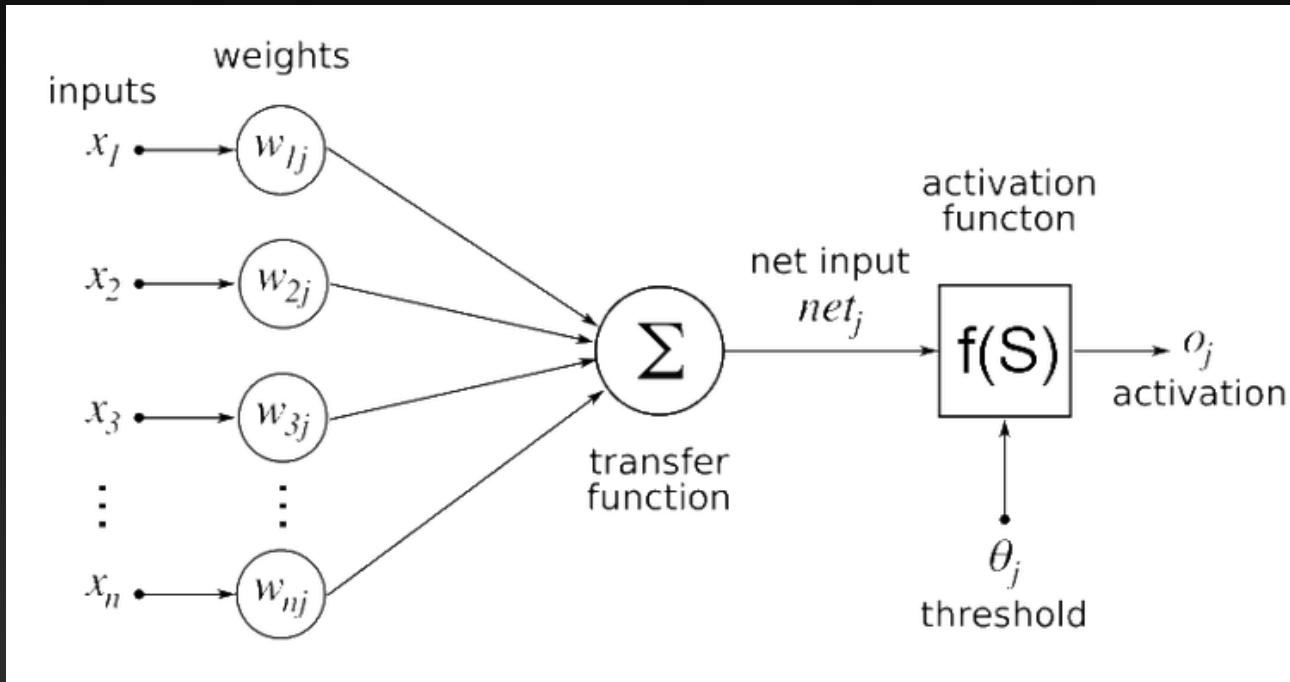
THE EXPANDING UNIVERSE OF MODERN AI

"THE BIG BANG"

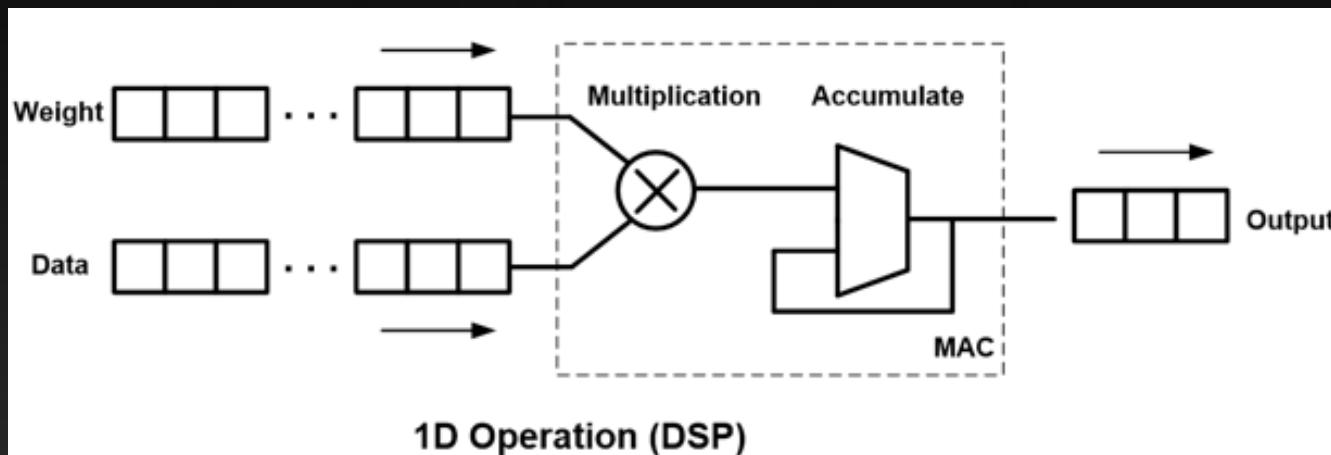
Big Data
GPU
Algorithms



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



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



GPU X-Ray



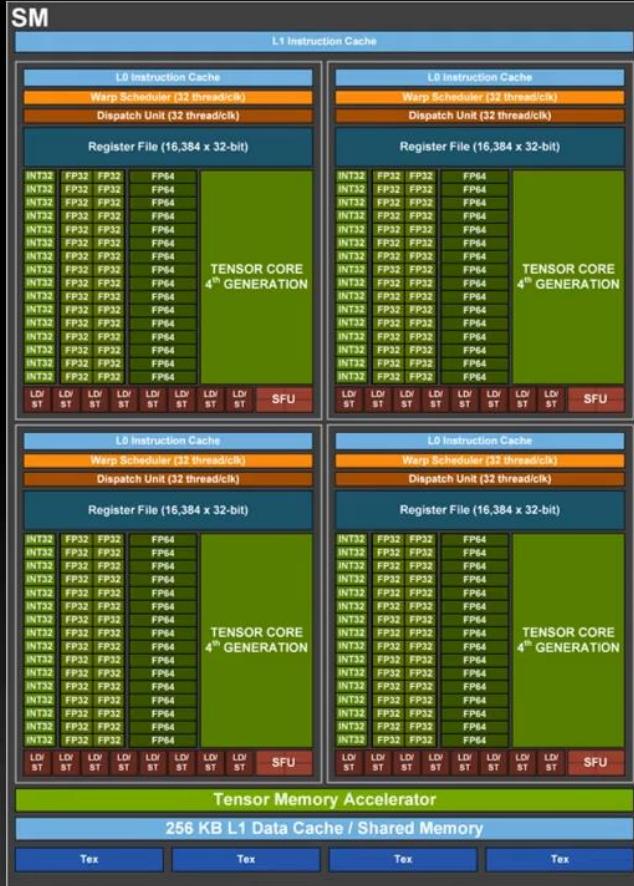
Ampere SM



Ampere SM



Hopper SM



H100 SM

256kB Shared Memory / L1-Cache

DPX Instruction Set

4th Generation Tensor Core

Fully-Asynchronous Architecture

Thread Block Clusters

Tensor Memory Accelerator

Asynchronous Transaction Barriers



Hopper SM



Hopper H100 Full Chip

132 SMs

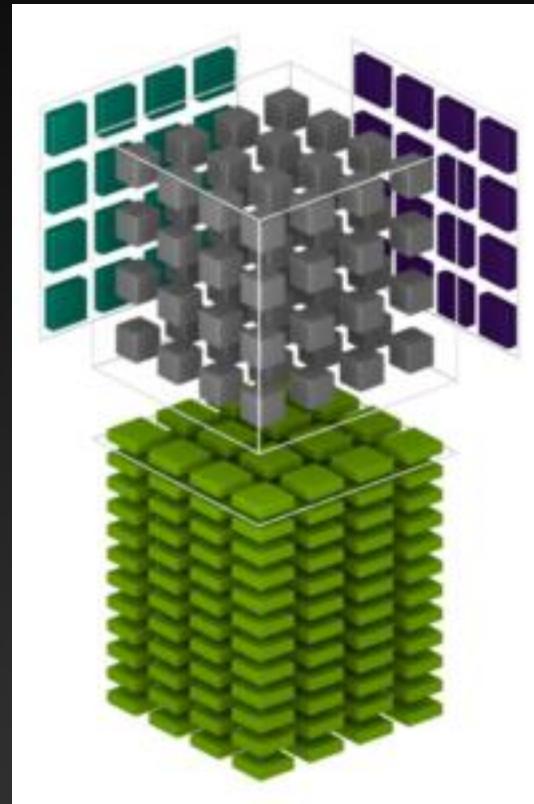


New Streaming Multiprocessor

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



Tensor Cores



Tensor Cores

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

64 FP operation per clock \rightarrow full process in 1 clock cycle

$$D = \left(\begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) \left(\begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) + \left(\begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right)$$

FP16 or FP32 FP16 FP16 or FP32

8 TC per SM \rightarrow 1024 FP per clock per SM

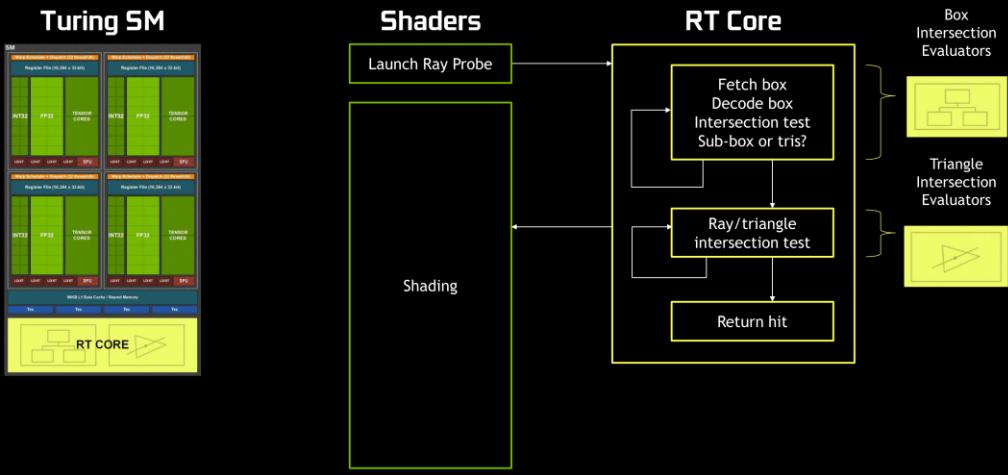
And what about Graphics?



And what about Graphics?

TURING RAY TRACING WITH RT CORES

Hardware Acceleration Replaces Software Emulation



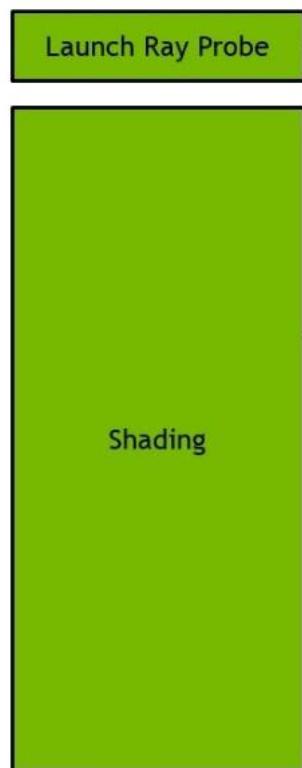
RT Cores

Hardware Acceleration Replaces Software Emulation

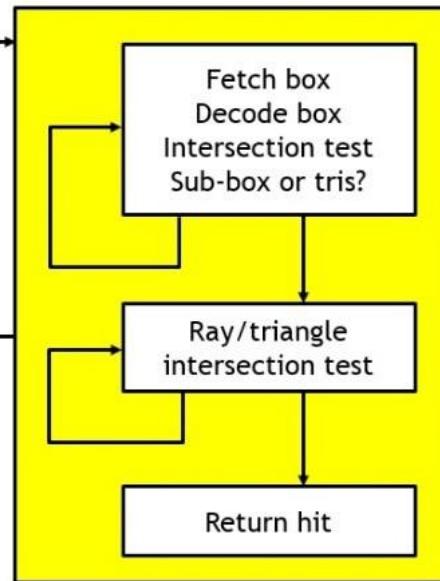
Turing SM



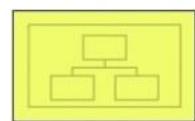
Shaders



RT Core



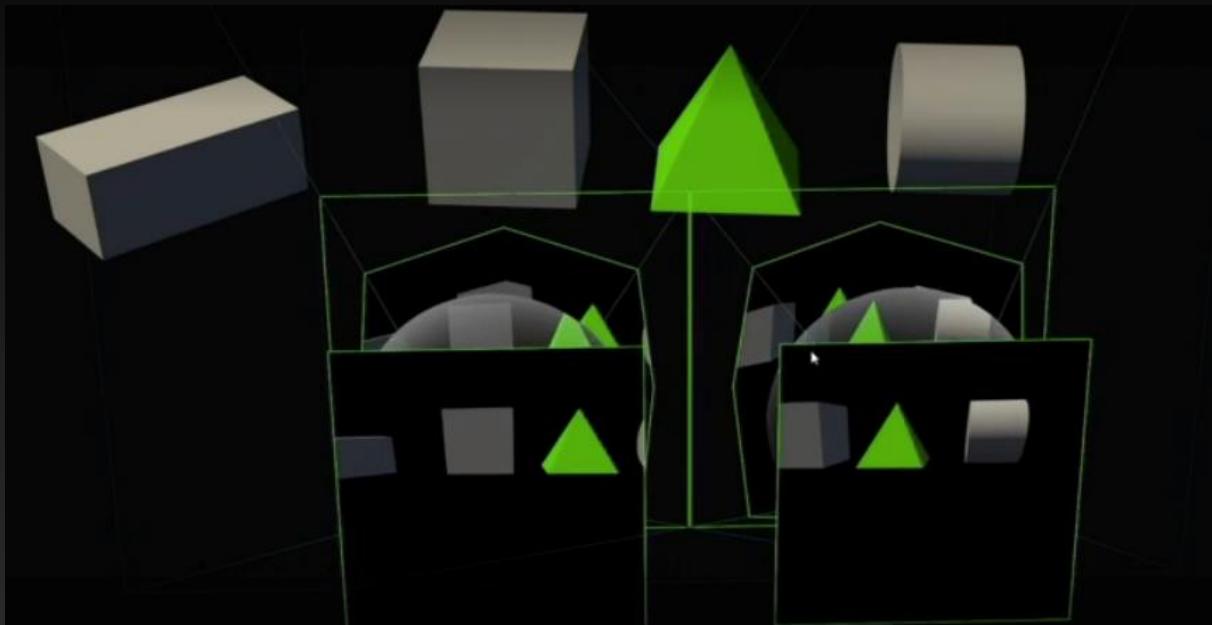
Box
Intersection
Evaluators



Triangle
Intersection
Evaluators



Pascal Multi-Res solution



Rendering- Latency problems

Human tolerance goes up to 20ms



CONTEXT
PRIORITY

Context Priority provides headset developers with control over GPU scheduling to support advanced virtual reality features such as asynchronous time warp, which cuts latency and quickly adjusts images as gamers move their heads, without the need to re-render a new frame.



DIRECT
MODE

With **Direct Mode**, the NVIDIA driver treats VR headsets as head mounted displays accessible only to VR applications, rather than a normal Windows monitor that your desktop shows up on, providing better plug and play support and compatibility for the VR headset.



FRONT BUFFER
RENDERING

Front Buffer Rendering enables the GPU to render directly to the front buffer reducing latency.

GPU Educational Kit

UFF NVIDIA CENTER OF EXCELLENCE

Follow us for daily updates:

MEDIALAB

NVIDIA

f

in

Home Learn GPU Computing Research Papers People Downloads Contact US Posts Search

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)

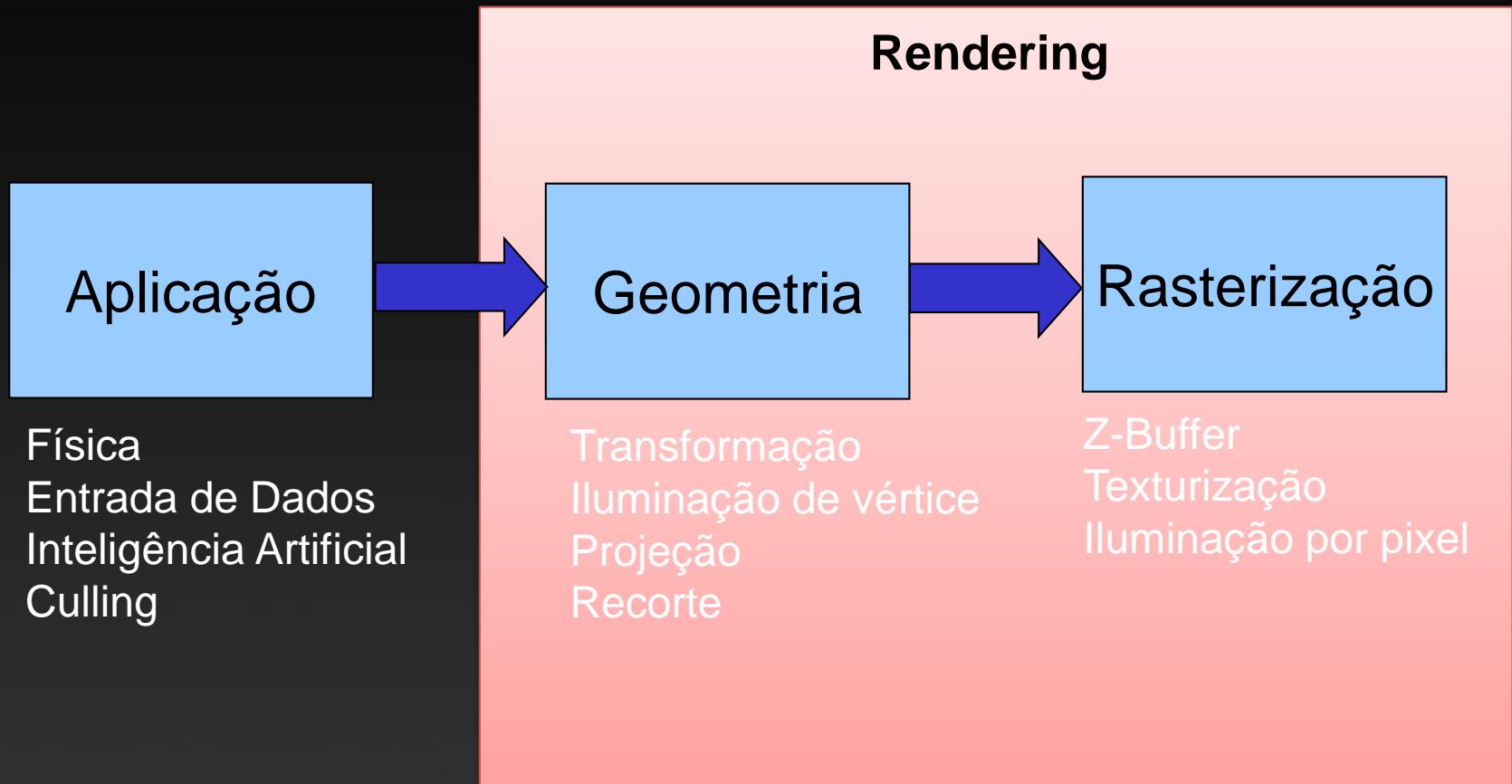
<http://www2.ic.uff.br/~gpu/kit-de-ensino-gpgpu/>

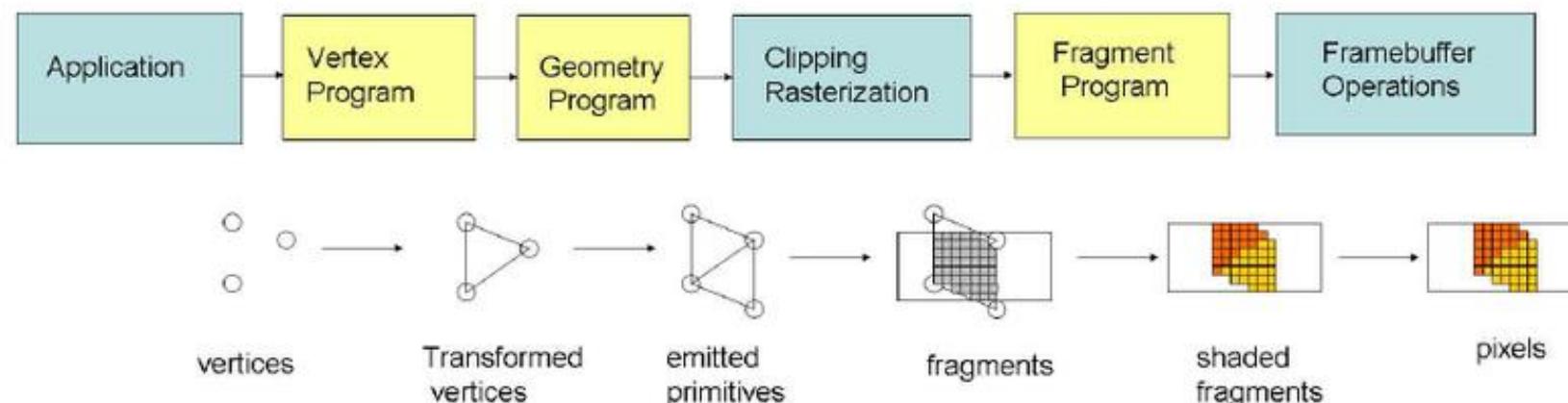


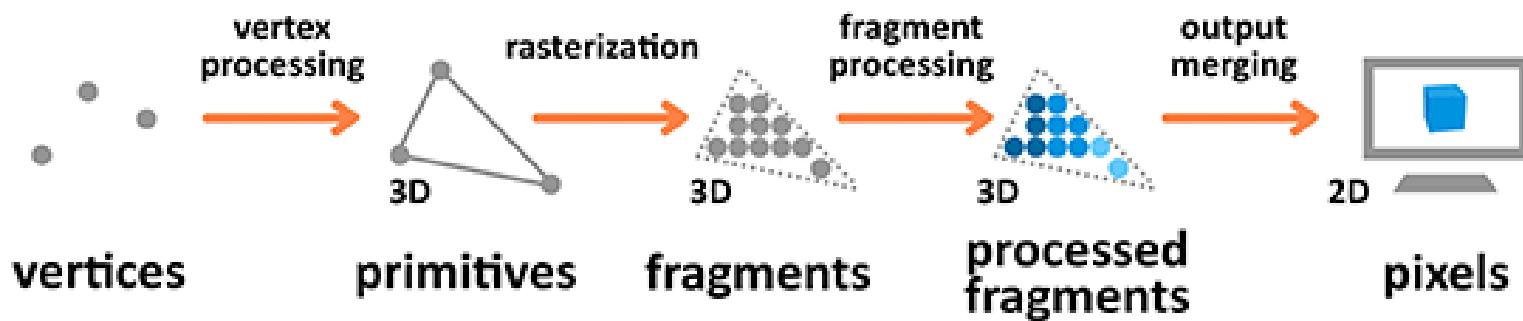
The GPU History



Graphic Pipeline

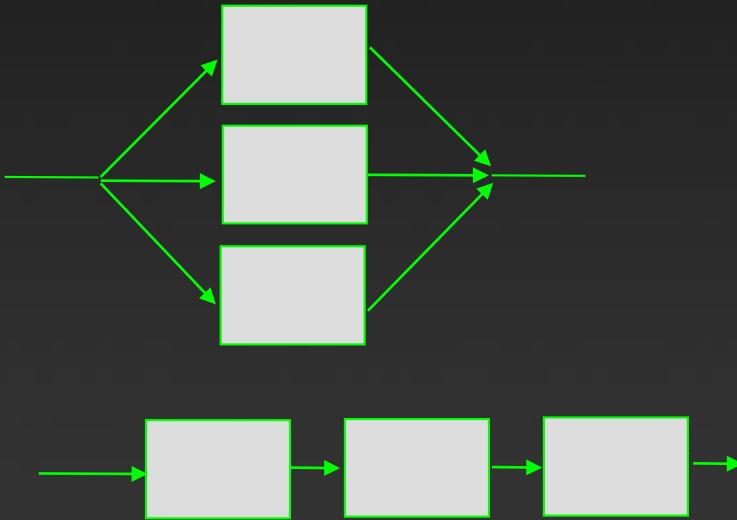
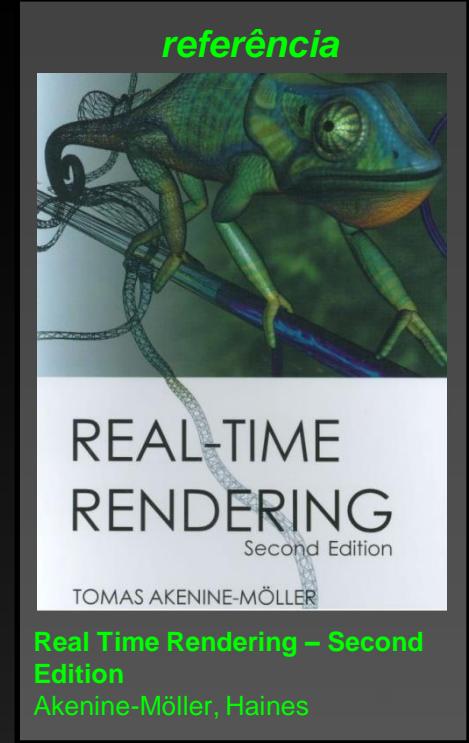




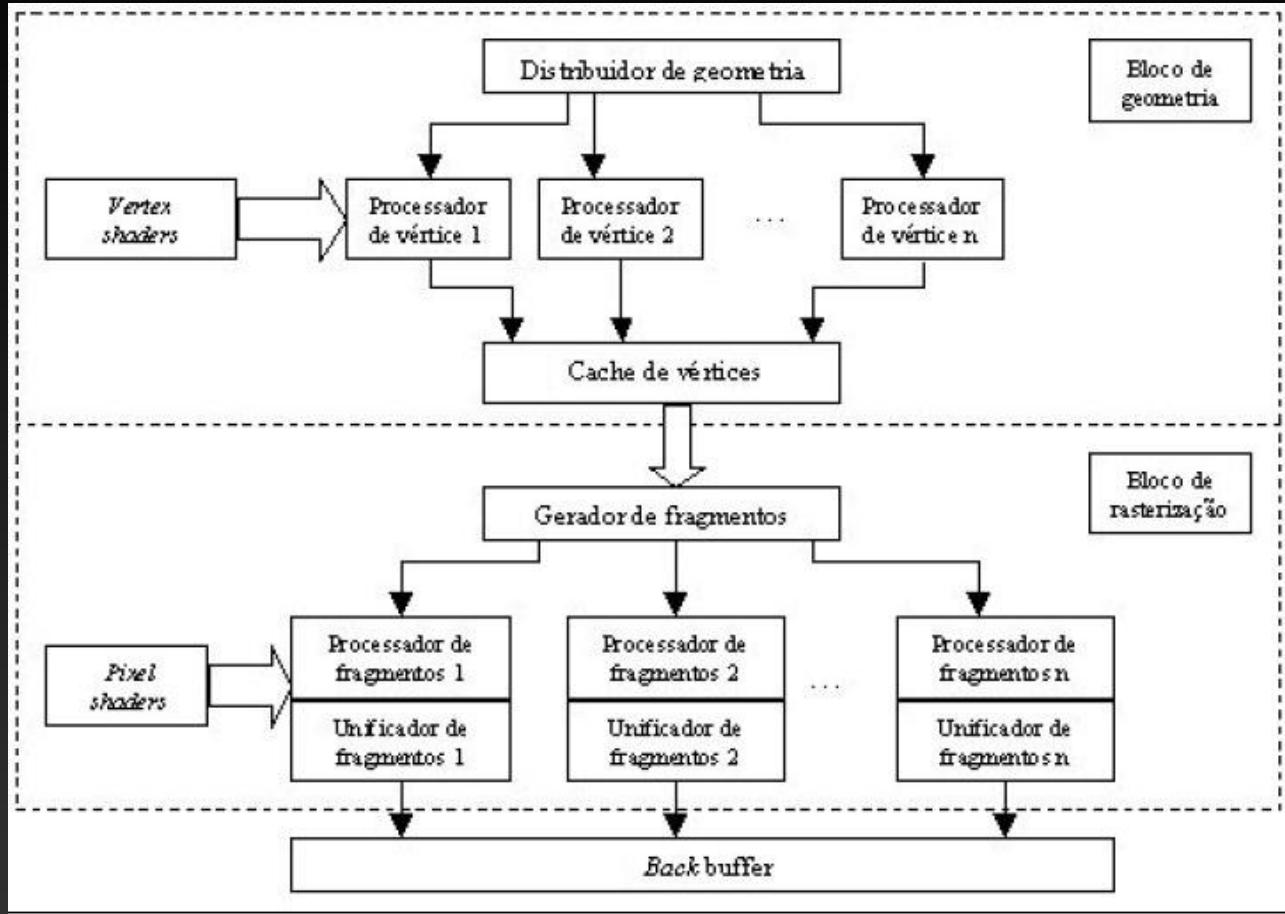


Pipeline Gráfico

- Pipeline / Estágios
- Gargalo
- Otimização
- Tipos de Processamento Paralelo



GPU Overview



```
// colored vertex lighting
Shader "Simple colored lighting" {
    // a single color property
    Properties {
        _Color ("Main Color", Color) = (1,.5,.5,1)
    }
    // define one subshader
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

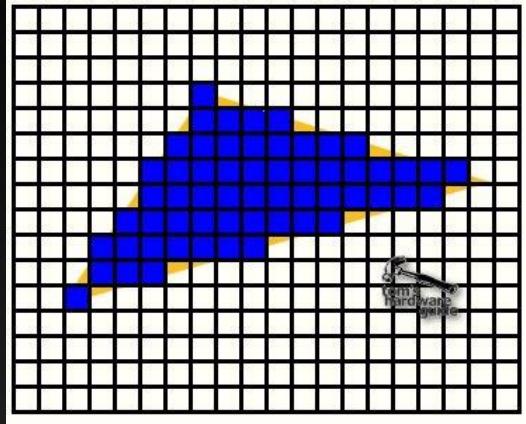


Once upon a time....

- Past
- Present
- Future



Pre-History – Rasterizer devices



~ U\$ 175.000,00

Fixed Function GPUs

- Impossible to be programed
- Nothing different than computer graphics
- No access to the processor
- APIs

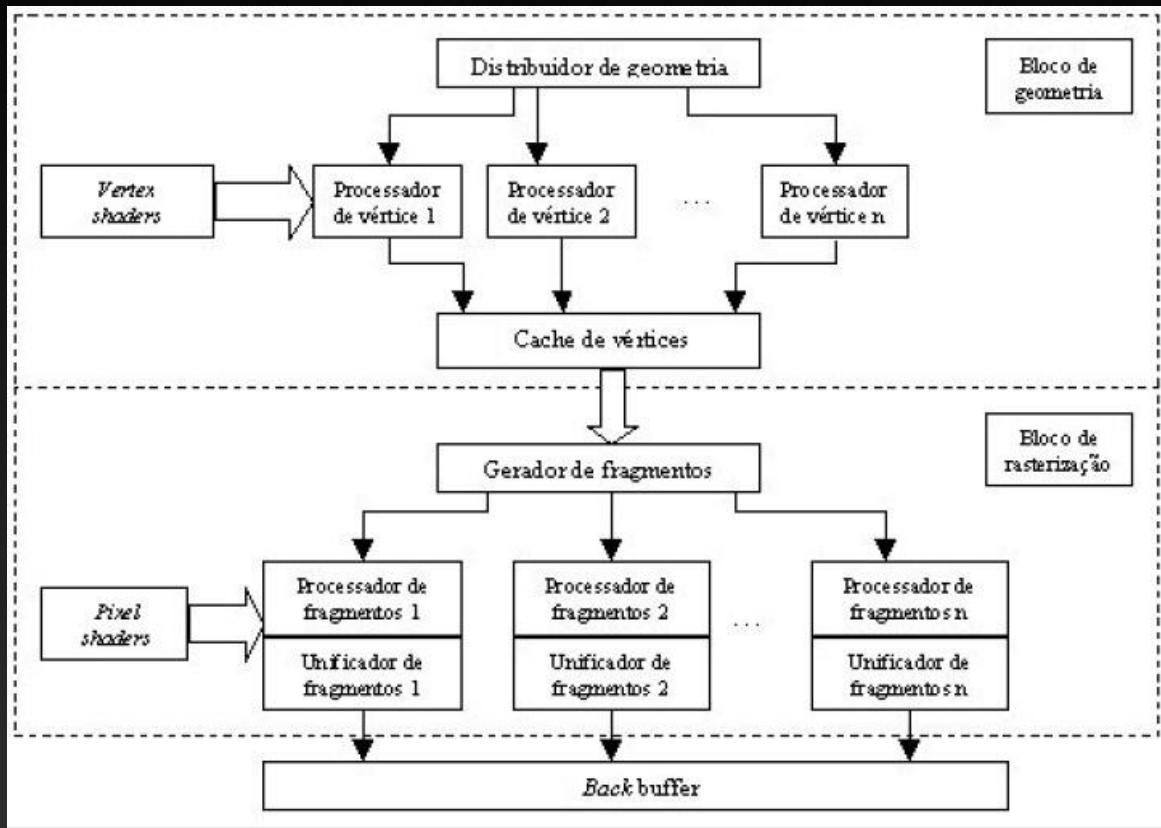


Programmable GPU

- Vertex and Pixel Shaders
- Completely oriented to Computer Graphics Data Structures



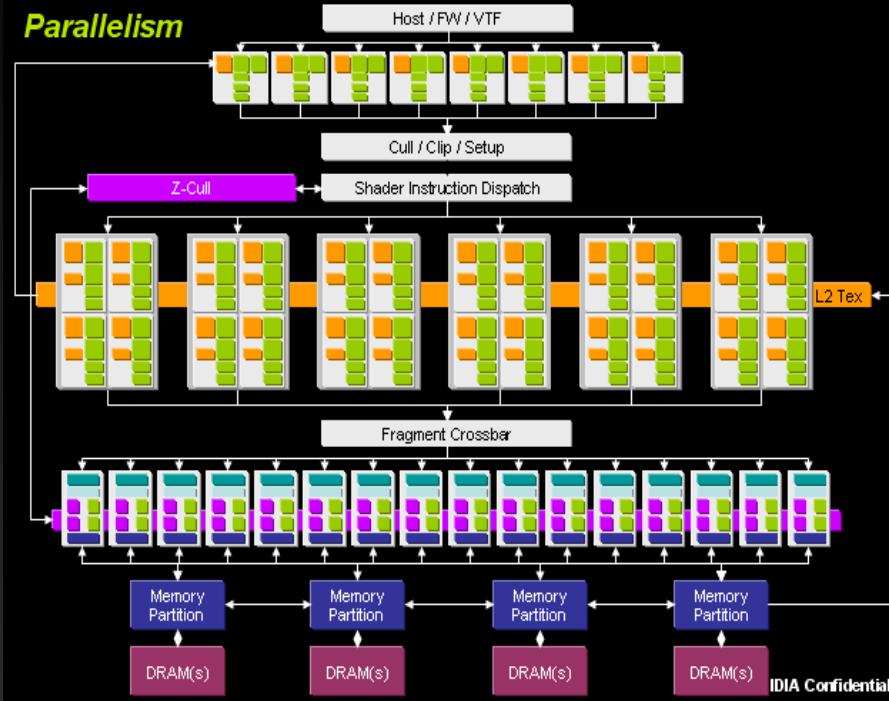
Programmable GPU



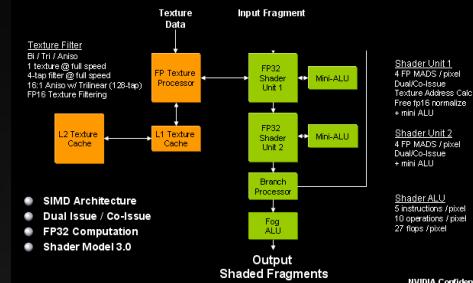
Programmable GPU

GeForce 7800

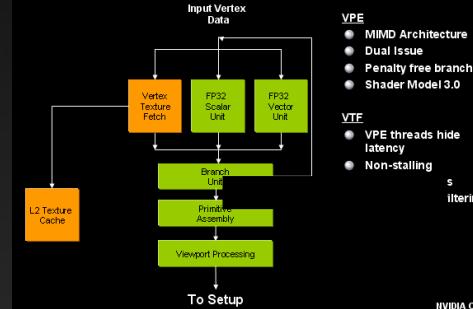
Parallelism



Detail of a single pixel shader pipeline



Detail of a single vertex shader pipeline



GPGPU

- Everything should be mapped to the graphics APIs
- OpenGL or DirectX languages for any kind of operation



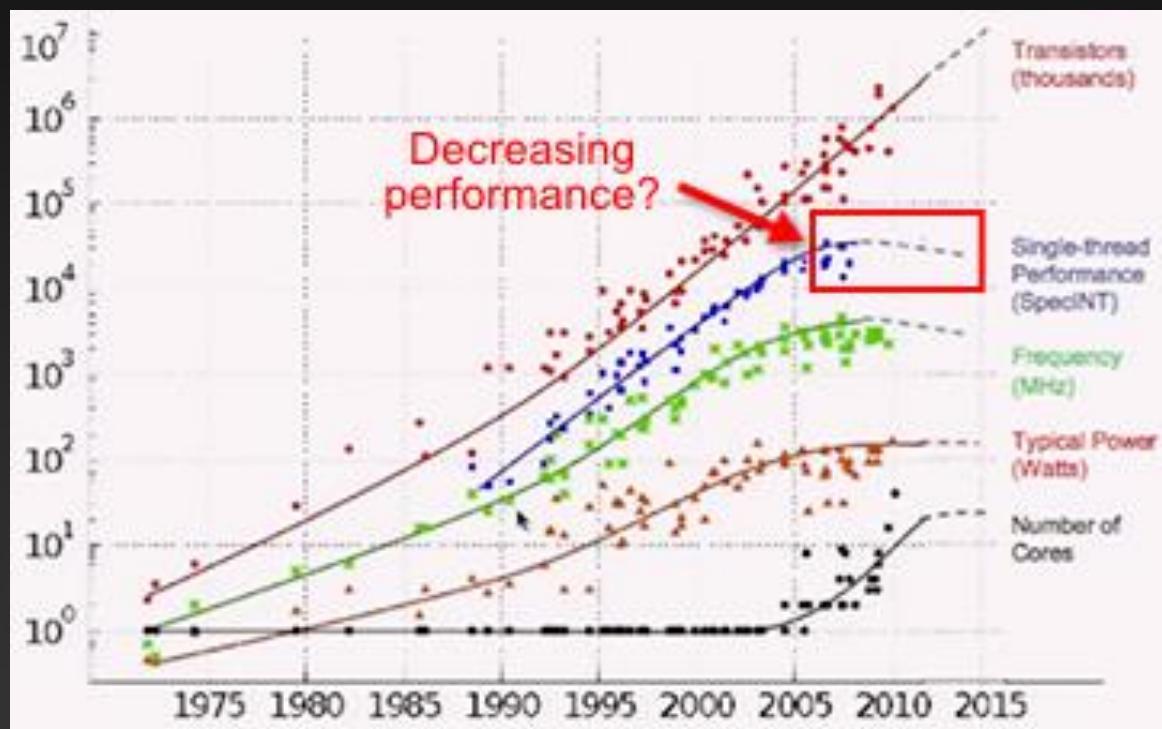
Unified Architectures



Some concepts about parallelism

Speeding up processors is not possible anymore for speeding the program...

Traditional single thread programming is becoming more and more an unreal situation in programs.



Amdahl's Law

Amdahl's law can be formulated in the following way:

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

where

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

If 30% of the execution time may be the subject of a speedup, p will be 0.3; if the improvement makes the affected part twice as fast, s will be 2.

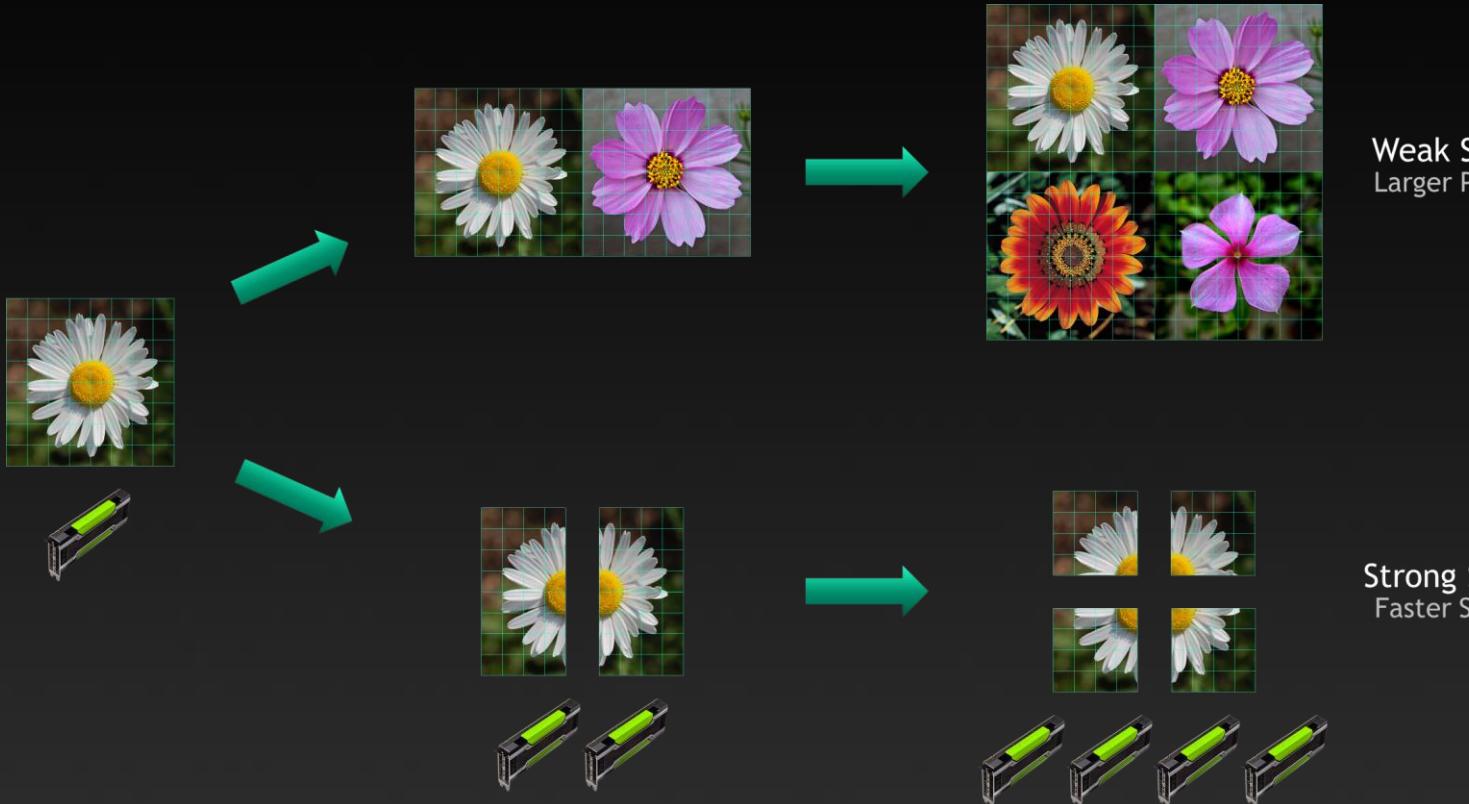
Amdahl's law states that the overall speedup of applying the improvement will be:

$$S_{\text{latency}} = \frac{1}{1 - p + \frac{p}{s}} = \frac{1}{1 - 0.3 + \frac{0.3}{2}} = 1.18.$$

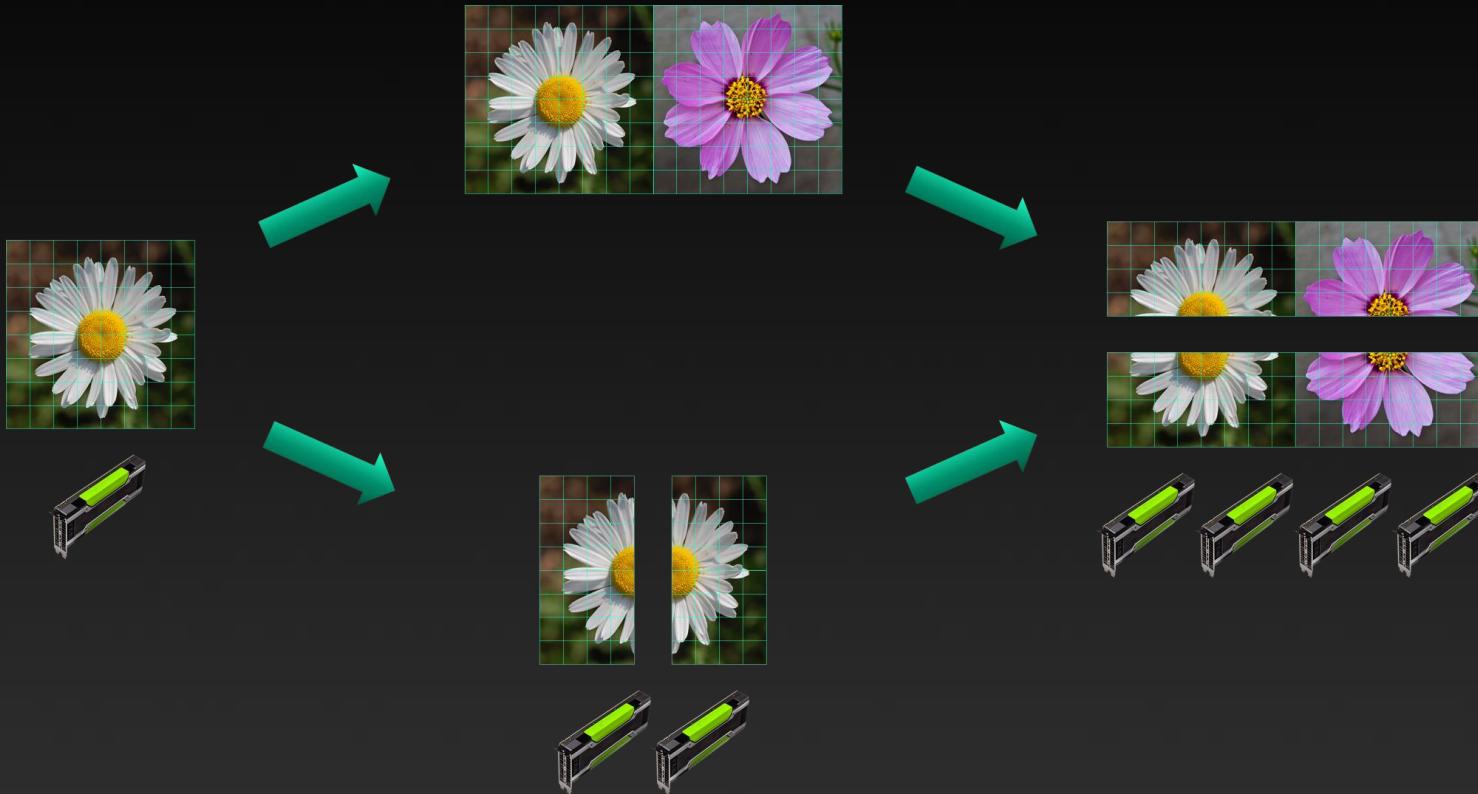
https://en.wikipedia.org/wiki/Amdahl%27s_law



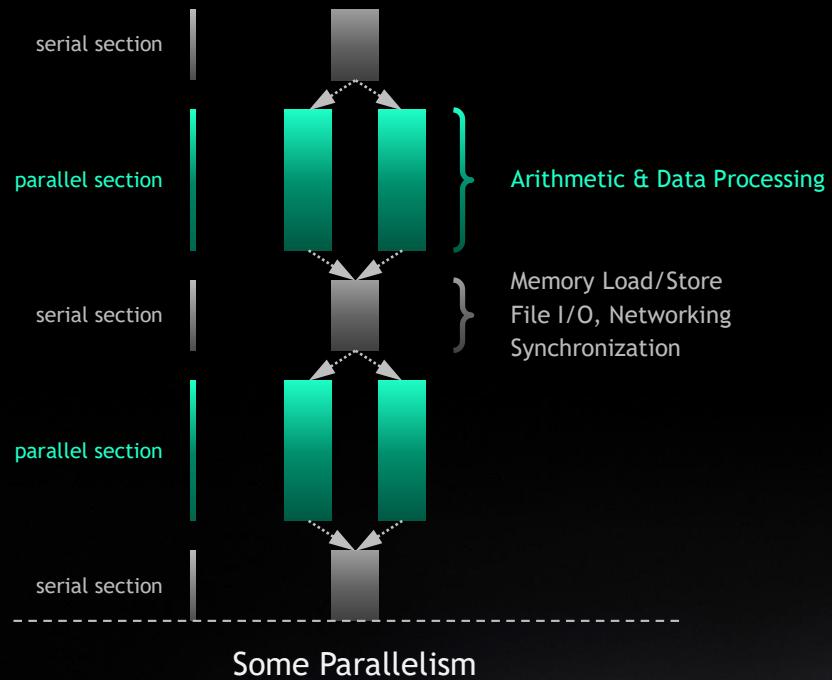
Amdahl's Law



Amdahl's Law

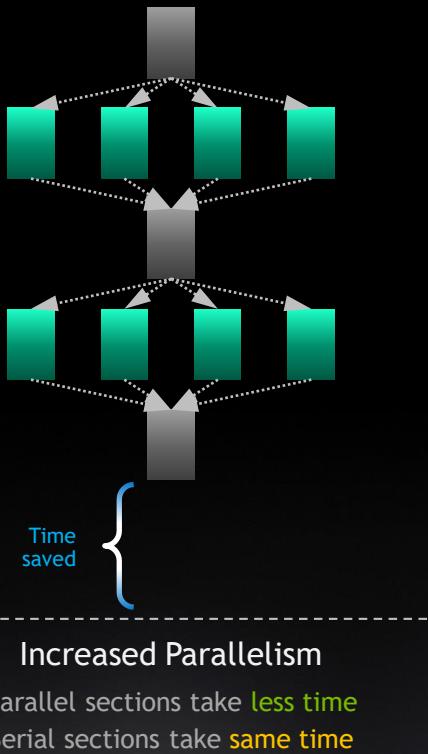
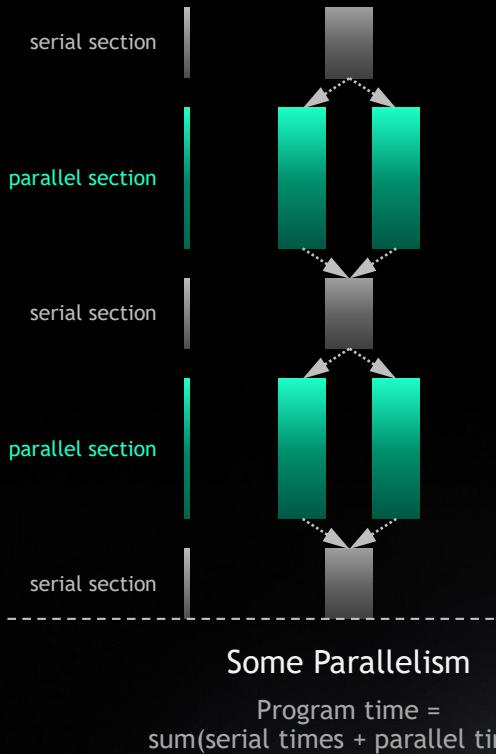


AMDAHL'S LAW

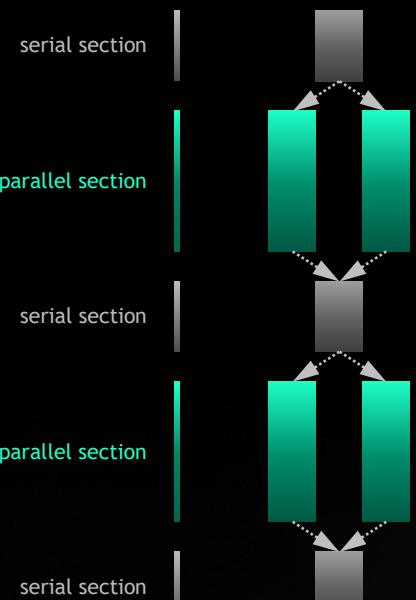


Program time =
sum(serial times + parallel times)

AMDAHL'S LAW

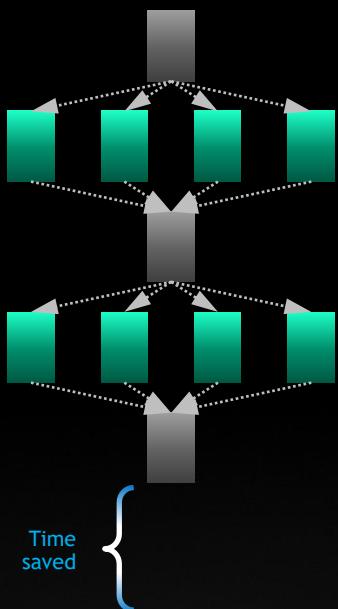


AMDAHL'S LAW



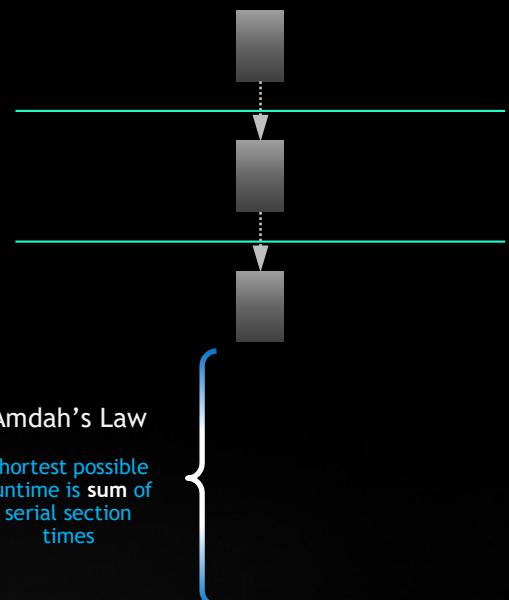
Some Parallelism

Program time =
sum(serial times + parallel times)



Increased Parallelism

Parallel sections take **less time**
Serial sections take **same time**

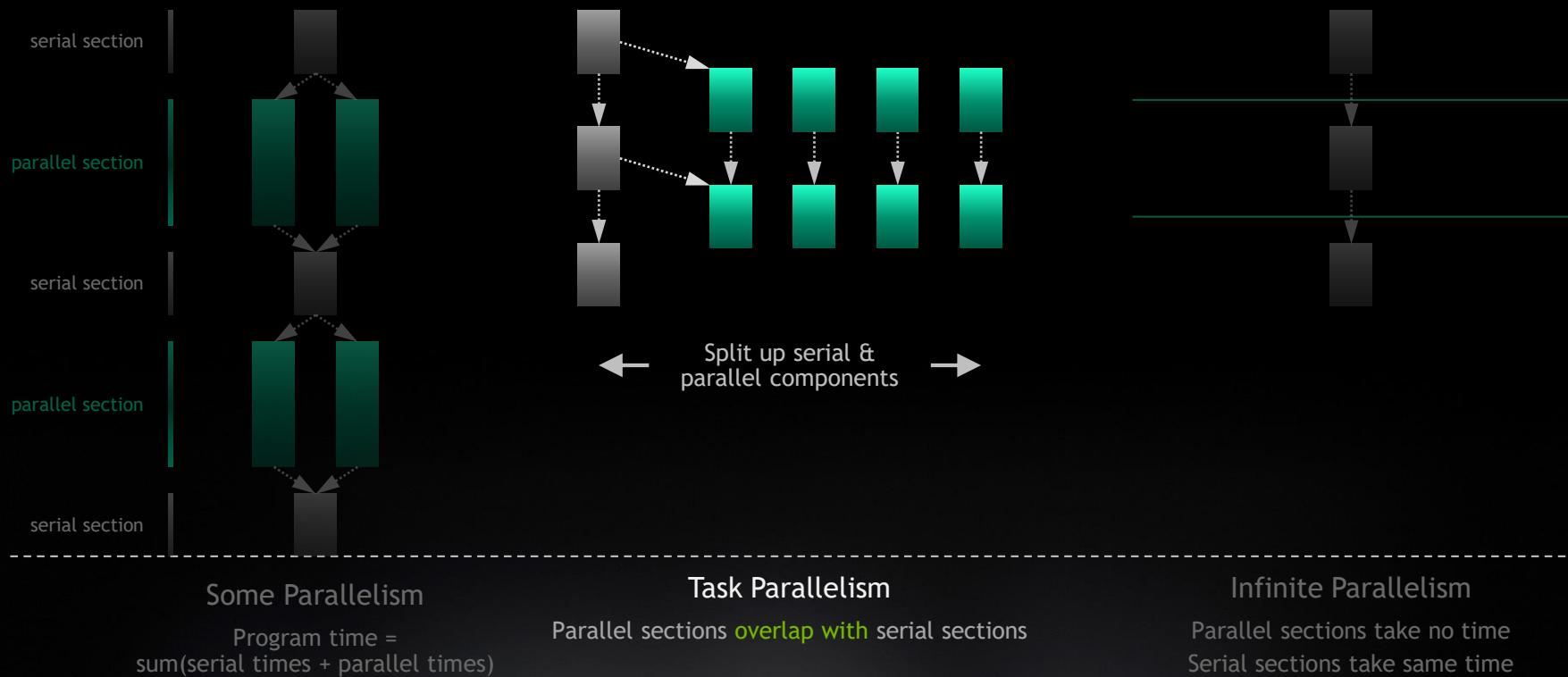


Infinite Parallelism

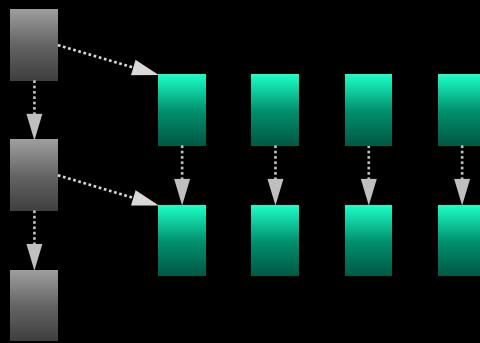
Parallel sections take **no time**
Serial sections take **same time**

Amdahl's Law
Shortest possible
runtime is **sum of
serial section
times**

OVERCOMING AMDAHL: ASYNCHRONY & LATENCY



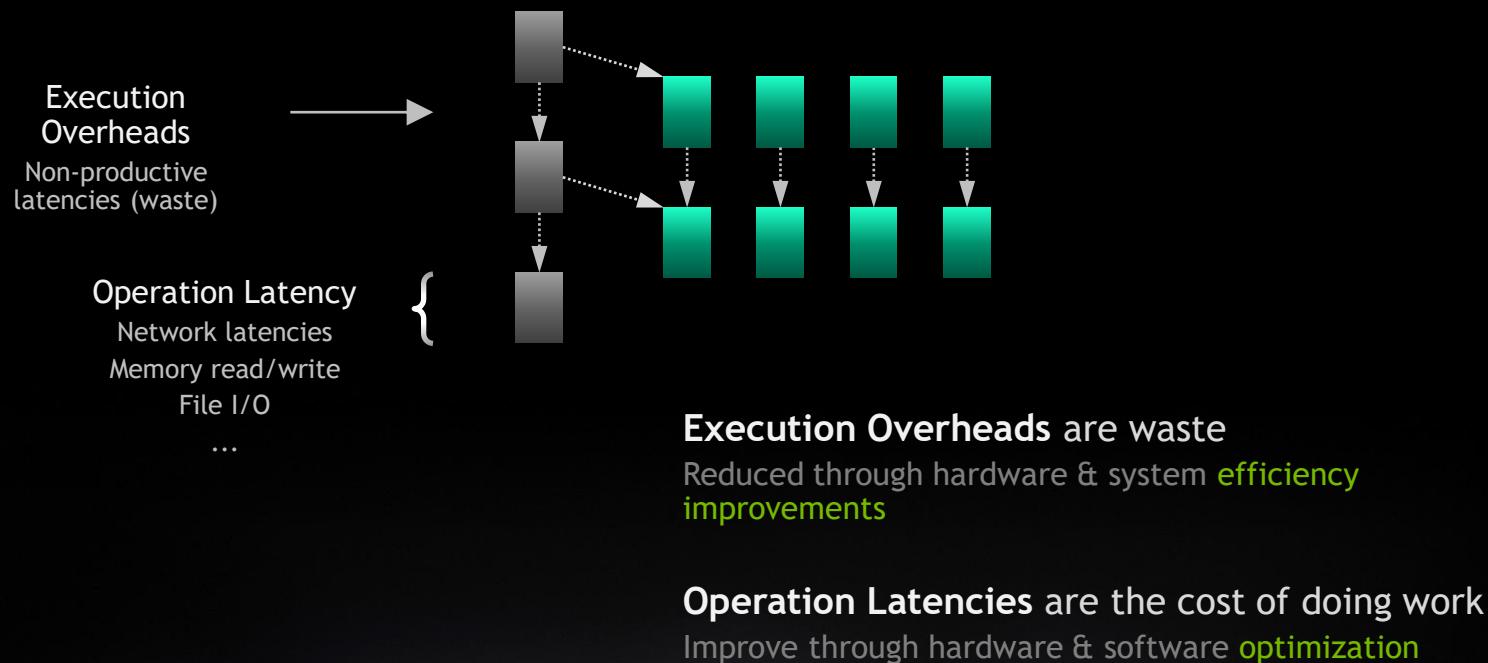
OVERCOMING AMDAHL: ASYNCHRONY & LATENCY



CUDA Concurrency Mechanisms At Every Scope

CUDA Kernel	Threads, Warps, Blocks, Barriers
Application	CUDA Streams, CUDA Graphs
Node	Multi-Process Service, GPU-Direct
System	NCCL, CUDA-Aware MPI, NVSHMEM

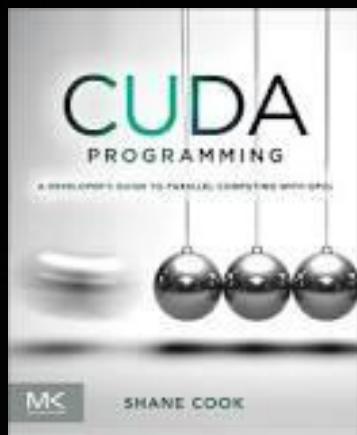
OVERCOMING AMDAHL: ASYNCHRONY & LATENCY



TASK

<https://usermanual.wiki/Pdf/Shane20CookCUDA20programming20A20developers20guide20to20parallel20computing20with20GPUsMorgan20Kaufmann202012.1739933505/html#pf4>

Ler Capítulo 1 – A Short History of Supercomputing



How to build a more powerful processor???

- 1) Decrease Latency (time for complete a task) - CPU
- 2) Increase Throughput (Things per second....) – GPU
CG: more pixels per second rather than less time per pixel...



Exercice:

We want do go from Rio to São Paulo – 400Km (by car or by Bus). A car transportates 5 persons in 100Km/h and a bus transportates 40 in 80Km/h

Car:

Latency: ____ Hours

Throughput: ____ people/hour

Bus:

Latency: ____ Hours

Throughput: ____ people/hour



Exercice:

We want do go from Rio to São Paulo – 400Km (by car or by Bus). A car transportates 5 persons in 100Km/h and a bus transportates 40 in 80Km/h

Car:

Latency: 4 Hours

Throughput: 1,25 people/hour

Bus:

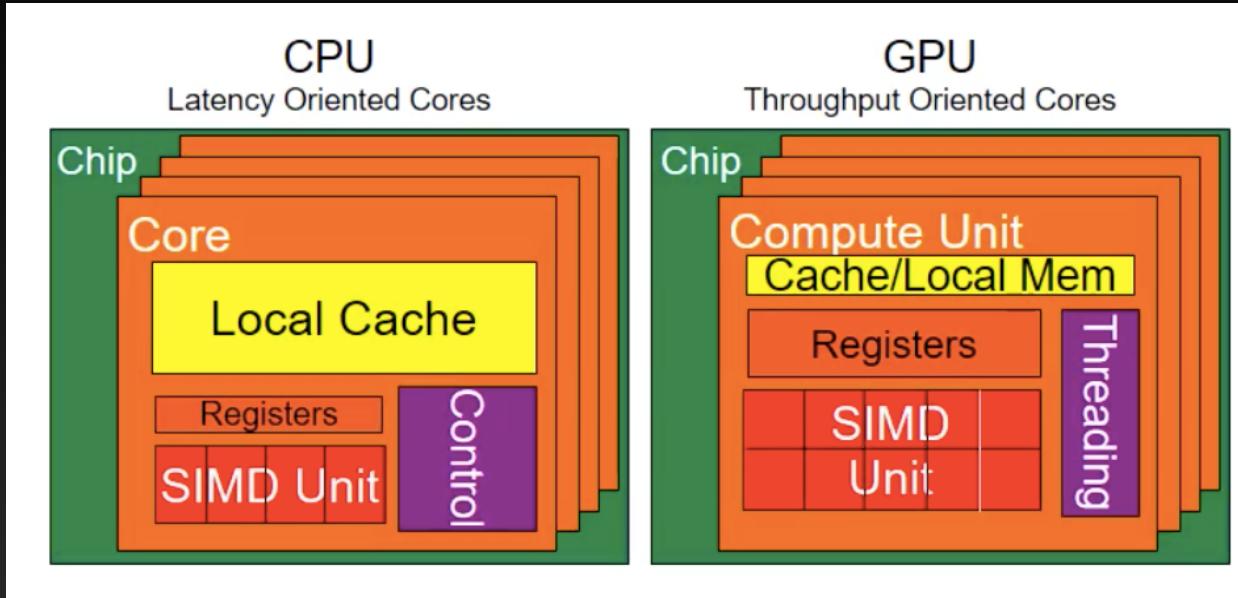
Latency: 5 Hours

Throughput: 8 people/hour

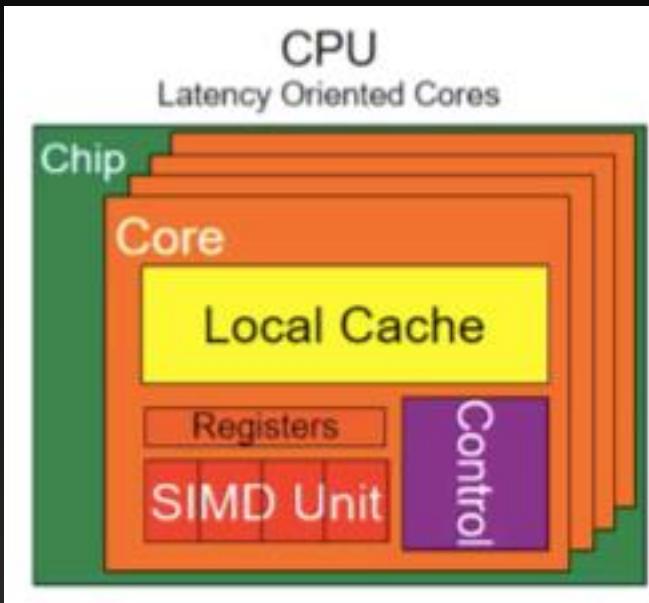


Latency devices (CPU)x

Throughput devices (GPU)

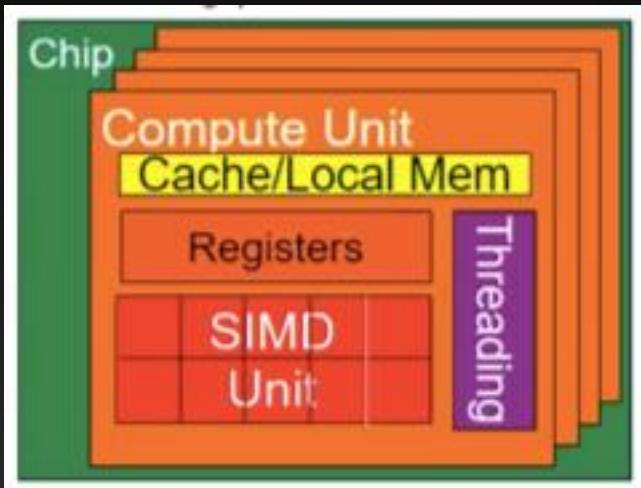


Latency devices (CPU) x Throughput devices (GPU)



- 1) Powerful ALU – reduce operation latency
- 2) Large Caches – convert long latency memory access to short memory cache access
- 3) Sophisticated control unit: reduce branch latency (branch prediction), data forwarding for reduce data latency

Latency devices (CPU) x Throughput devices (GPU)

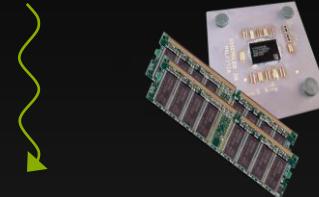


- 1) Small cache
- 2) Very simple control units (no branch prediction, no data forwarding)
- 3) Energy efficient ALU: long latency
- 4) Massive number of threads: small overhead for threads changes

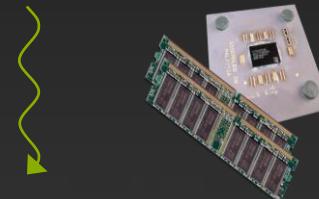
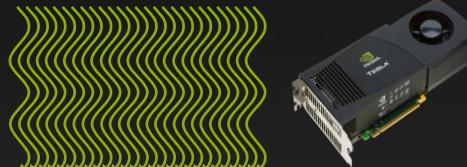
Latency devices (CPU)x

Throughput devices (GPU)

CPU is much more faster when latency matters



GPU is much more faster when throughput matters



Hybrid Computing

Parts of the code can depend of
optimizing Latency, parts of the code
can depend on optimizing throughput



Ideal Scaling

1 process takes 20h in a single processor

1 process takes 5h in 4 processors (20h of work in total)



Real Scaling

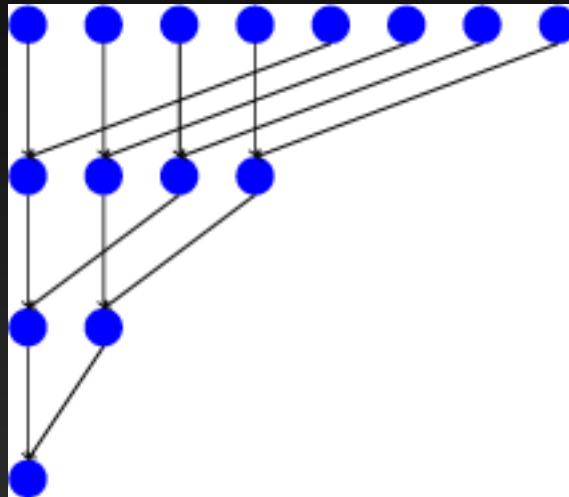
1 process takes 20h in a single processor

1 process takes 10h in 4 processors (40h of work in total)



Step and Work Complexity

Discussion about reduction Algorithm (summing vector elements)



4 step complexity

15 work complexity

Parallelism in single processors

OpenMP is an example of tool for showing places of possible parallelism

(Similar of OpenACC in GPUs)



Work Proposal - OpenMP



Parallelism in Clusters

MPI is an example for data management in parallel environments (such as clusters)



Work Proposal - MPI



Types of Parallelism

Task Based Parallelism: diverse and unrelated tasks. Pipeline Parallelism – one output is an input for the next stage. The bottleneck will be the slowest stage. (car assembly)



Types of Parallelism

Data-based Parallelism: Data is divided into different parts
(supermarket payment)



Flynn's Taxonomy (instructions x Data)

SIMD (Single instruction, multiple data)

MIMD (Multiple Instructions, multiple data)

SISD (Single Instruction, Single data) – Must use time slice for parallelism

MISD (Multiple Instruction, Single data)



Parallel Patterns

Loop-based Patterns: requires to broken iteration dependencies

Fork/join Patterns: A program arrives to a point that can fork different parallel works and then join again

Tiling/Grids

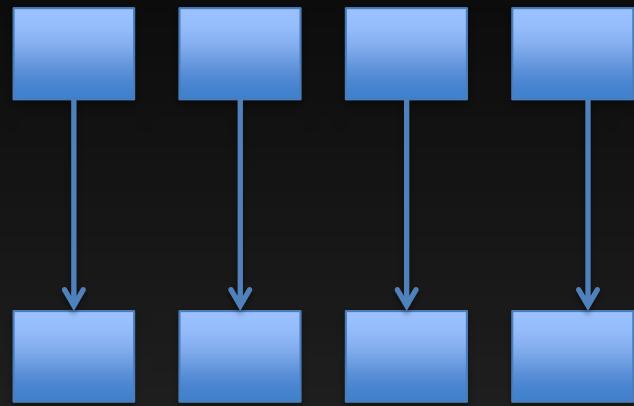
Divide and Conquer: typically used with recursive solutions



Communication Patterns

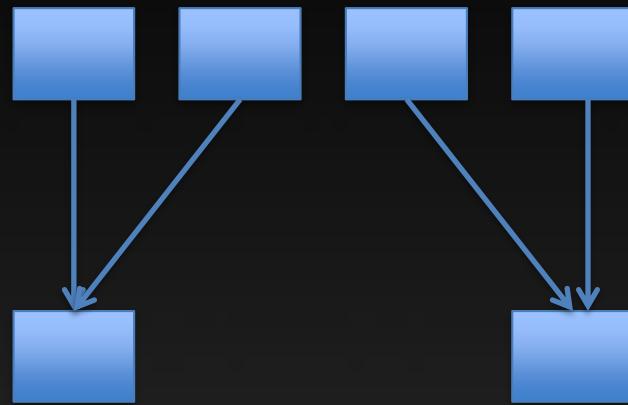


Map Operation (one to one)



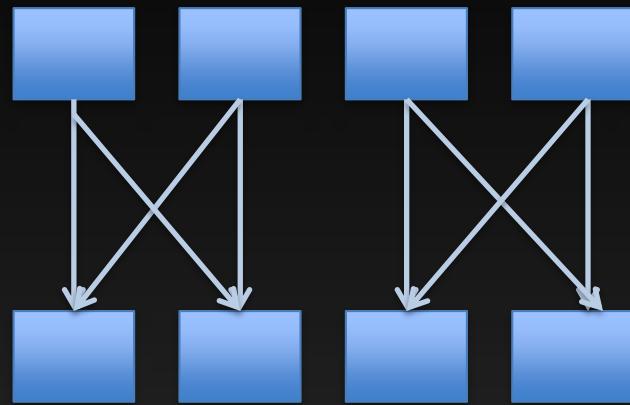
Examples...

Gather... (many to one)



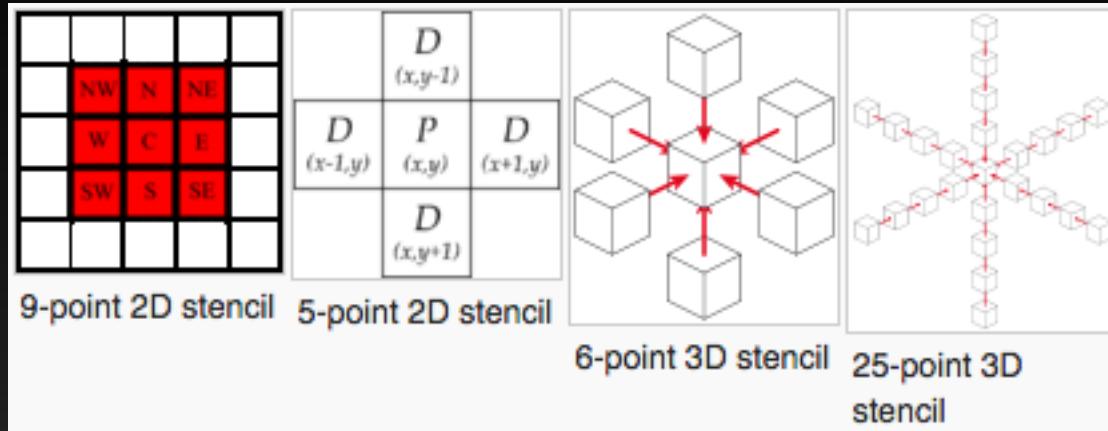
Examples: Blur image...

Scatter (one to many)



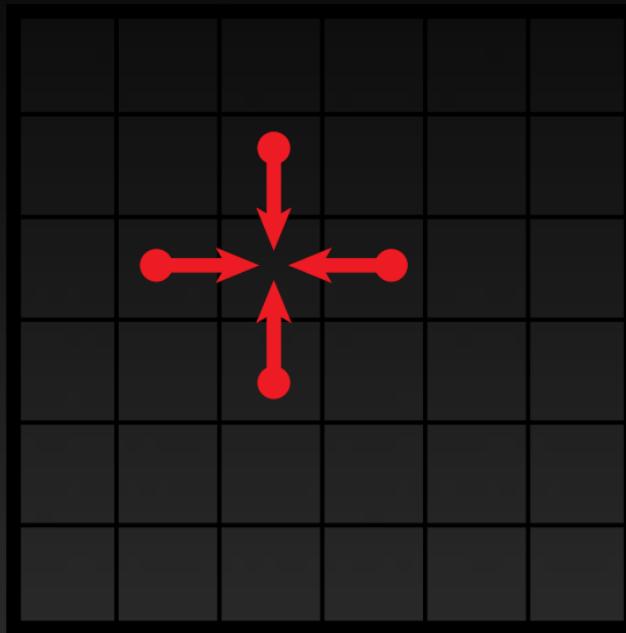
Examples: How to transform gatter into scatter. Problem of multiple threads writing in the same place...

Stencil Patterns (several to one)

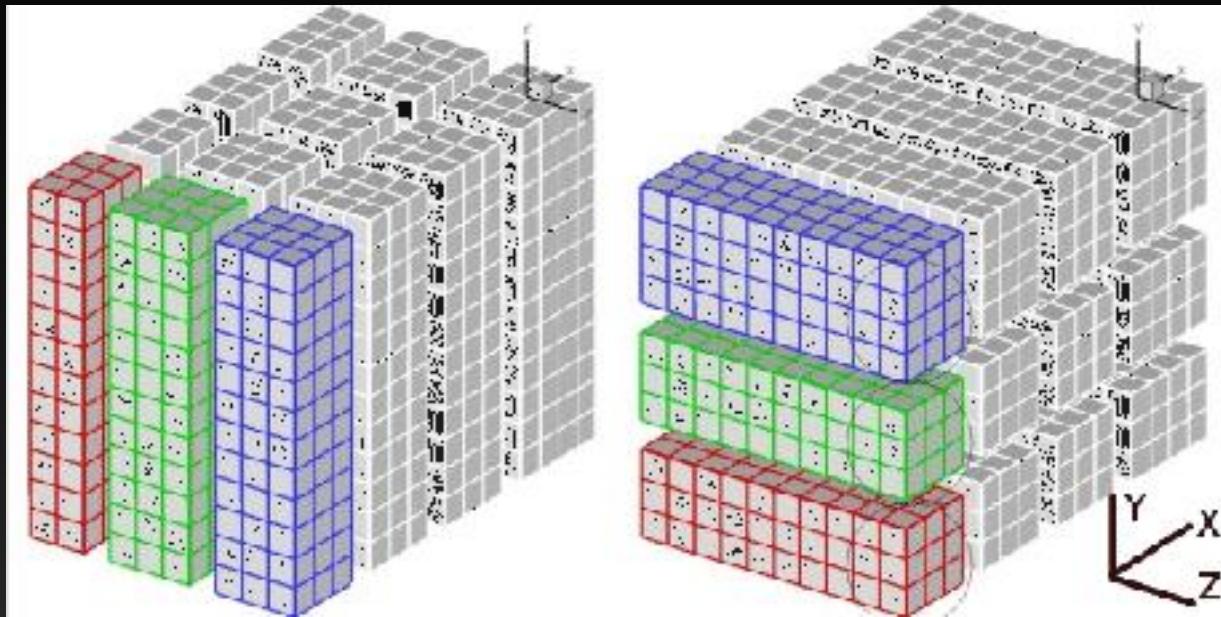


Examples of usage: fluid simulation, wave propagation, cellular automata...

Stencil Von Neumann



Transpose Operation?



Exercício:

Dar um exemplo para cada tipo de mapeamento

Thread Concept

A thread is a separated executed flow within a program that may diverge and converge as and when required with the main execution flow.

#threads ~ # cores

Threads typically shares resources



Process Concept

Processes are executed in independent memory regions and usually is not enabled to influence other processes memories.



Semaphore

Token or semaphore: who ever needs a resource, requires the token and others must wait for



Deadlocks

Thread1 takes token1 and thread2 takes token2. If thread1 needs to take token2 and thread2 needs token1, they may enter in deadlock...

This will happen If there is no special design in the code...

Exercise: Build and example...



Concurrency

$$X(t) = at + b$$

$$Y(t) = bt + c$$

$$F(t) = X(t) + Y(t)$$



Threads, warps, blocks and Grid

Grid is an Army, commanded by a General

Blocks are units, commanded by a Capitan

Units are composed of squads of 32 (warps), commanded by a sargent

Threads are soldiers

How they communicate with each other?...



Cache

Moving and delivering data became an important issue in modern architectures...

Cache can be addressed as spatial or temporal coherence

Cache hit: a number of tools are repeated constantly
(when preparing a food in kitchen: some ingredients are in the balcony
(cache L1) but others may be at the dispence (cache L2). Both memory
access have a specified time. However, in some cases he may need a
special ingredient and must drive to the supermarket (global memory).
This may take an unpredictable time... (rush, crowd, etc...) and have
an unpredictable latency



Triangular Matrix

Row = 1 + int ((sqrt (8*I +1) -1)/2))

Column = I – row(row-1)/2



Acesso remoto

Login: uffgpu2019

Senha: uffgpu@2019

IP público: 200.20.15.155

Porta externa 2022

Nome da máquina: Wolverine

GPUs: GTX 980 (device 0), K40 (device 1)

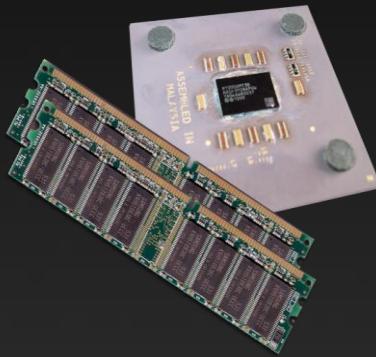


GPU Programming paradigms



#1 – we are talking about Heterogeneous Computing

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

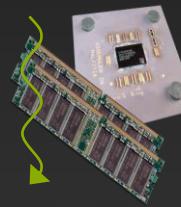
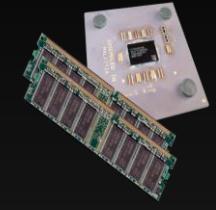
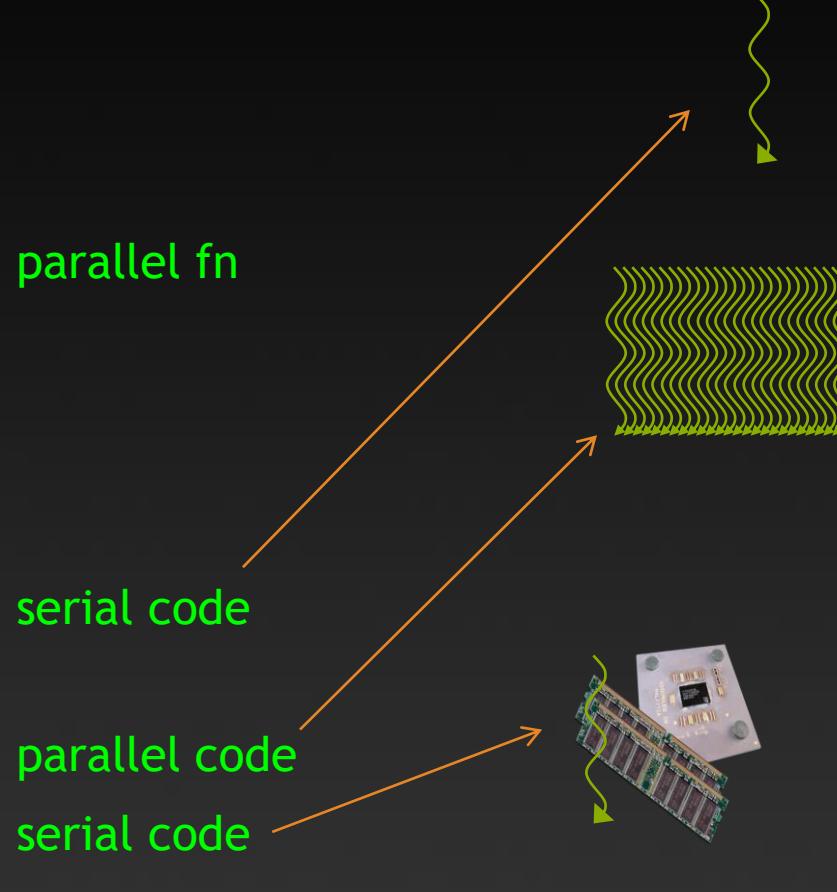
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

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

parallel fn

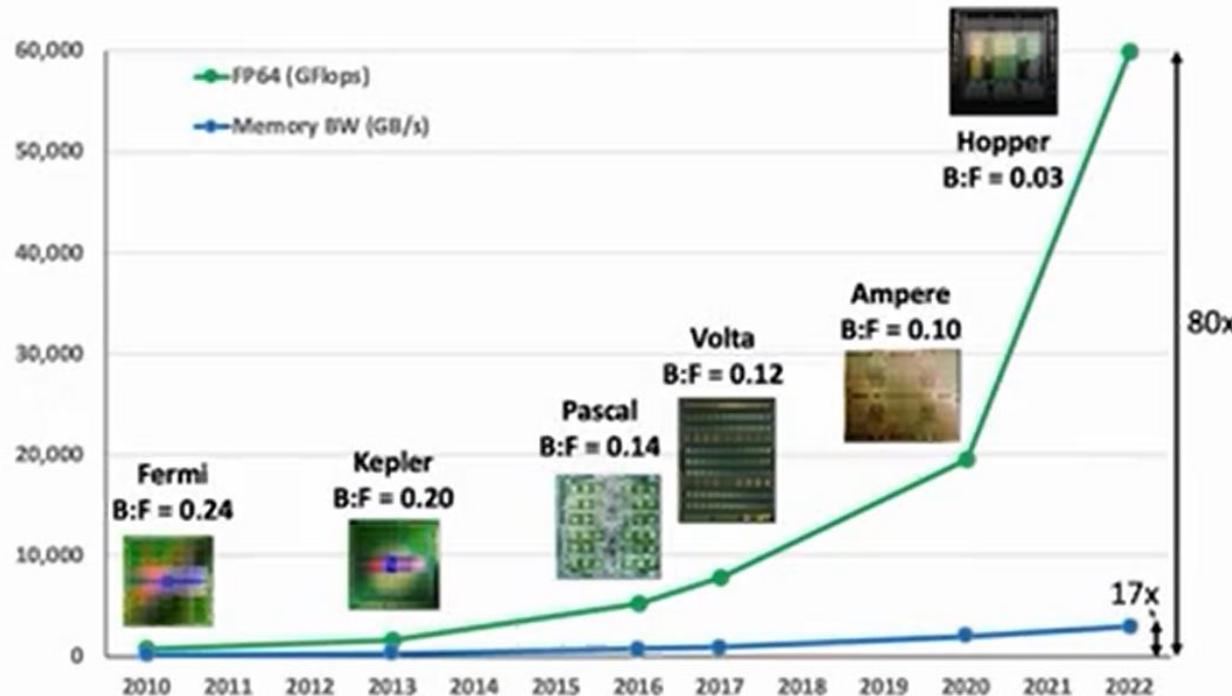
serial code

parallel code
serial code



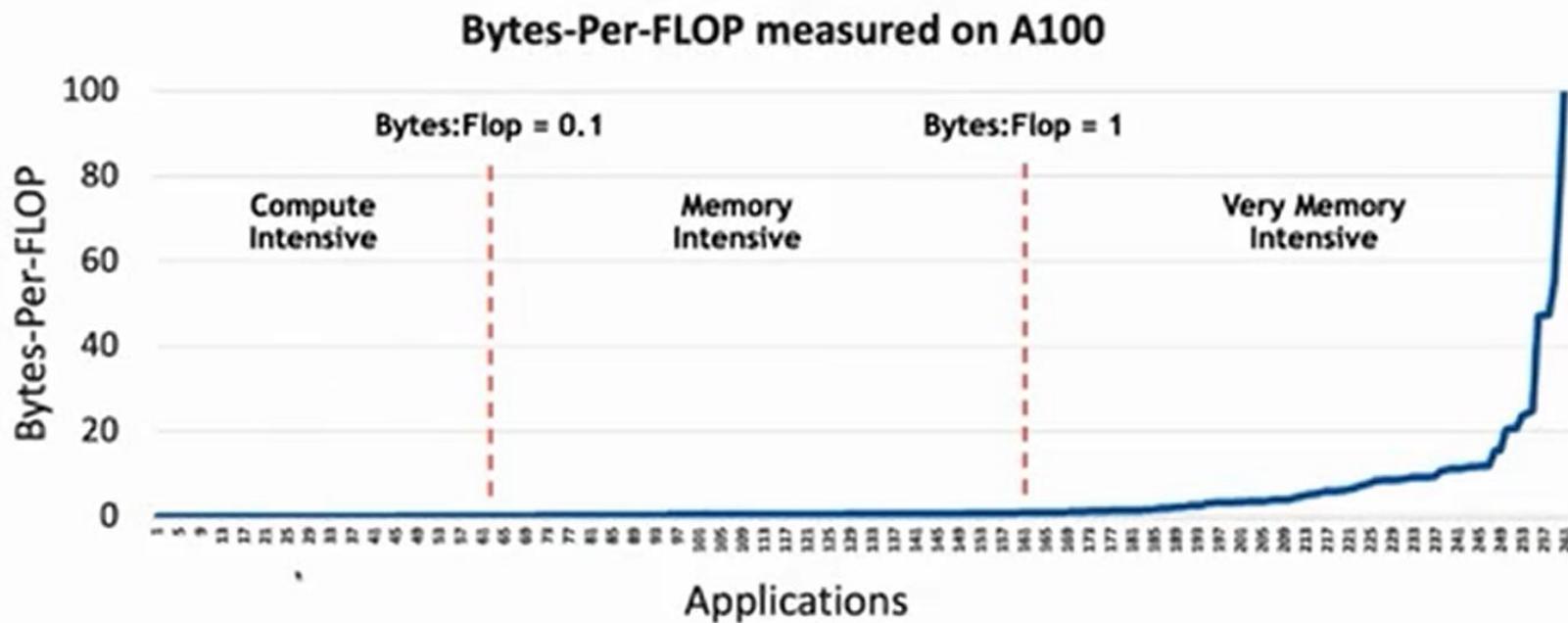
#2 – Memory bandwidth matters!...

Gap between Compute and Memory Bandwidth Increasing
Ratio of peak memory to compute bandwidth in Bytes:Flop (B:F) is dropping

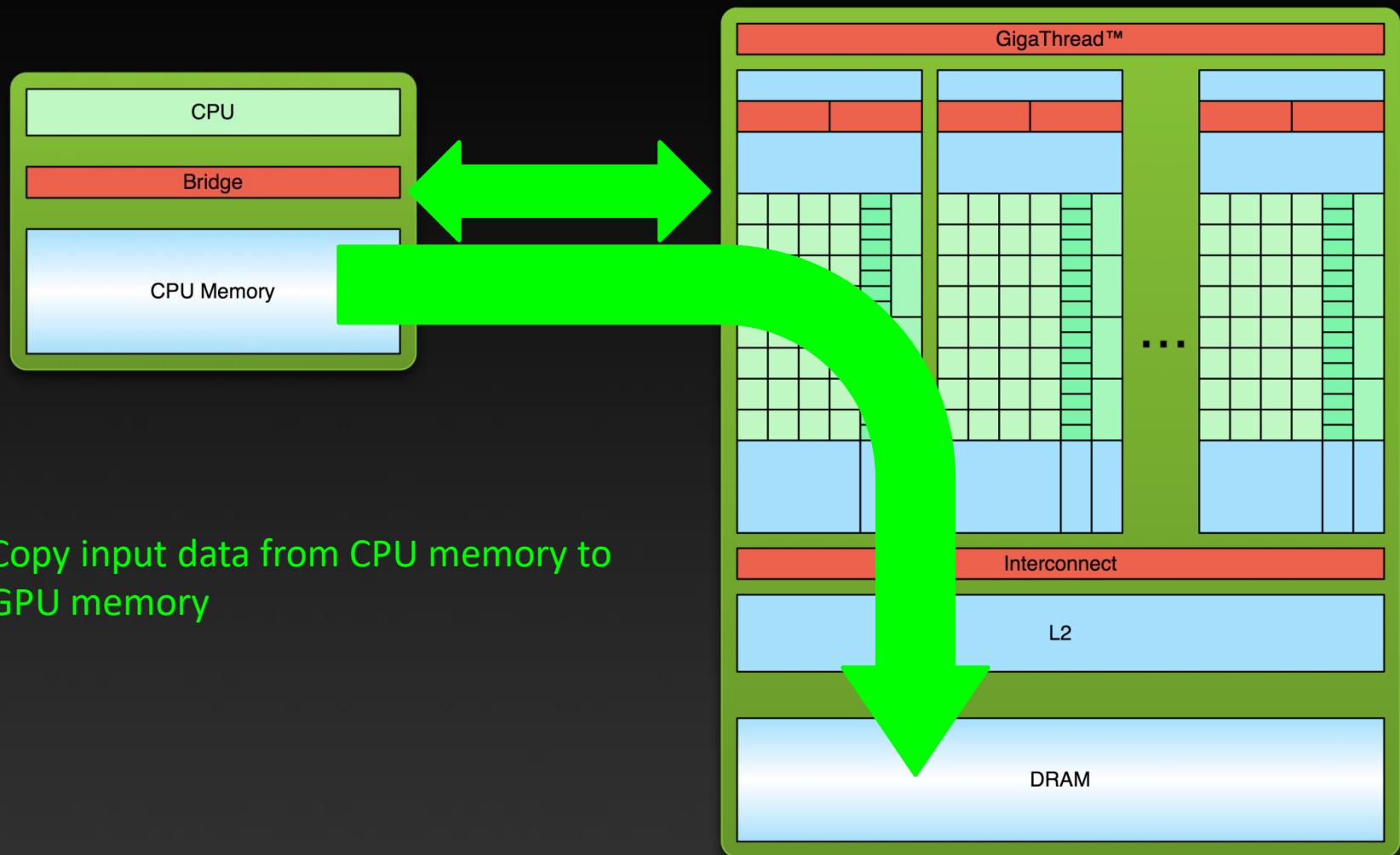


#2 – Memory bandwidth matters!...

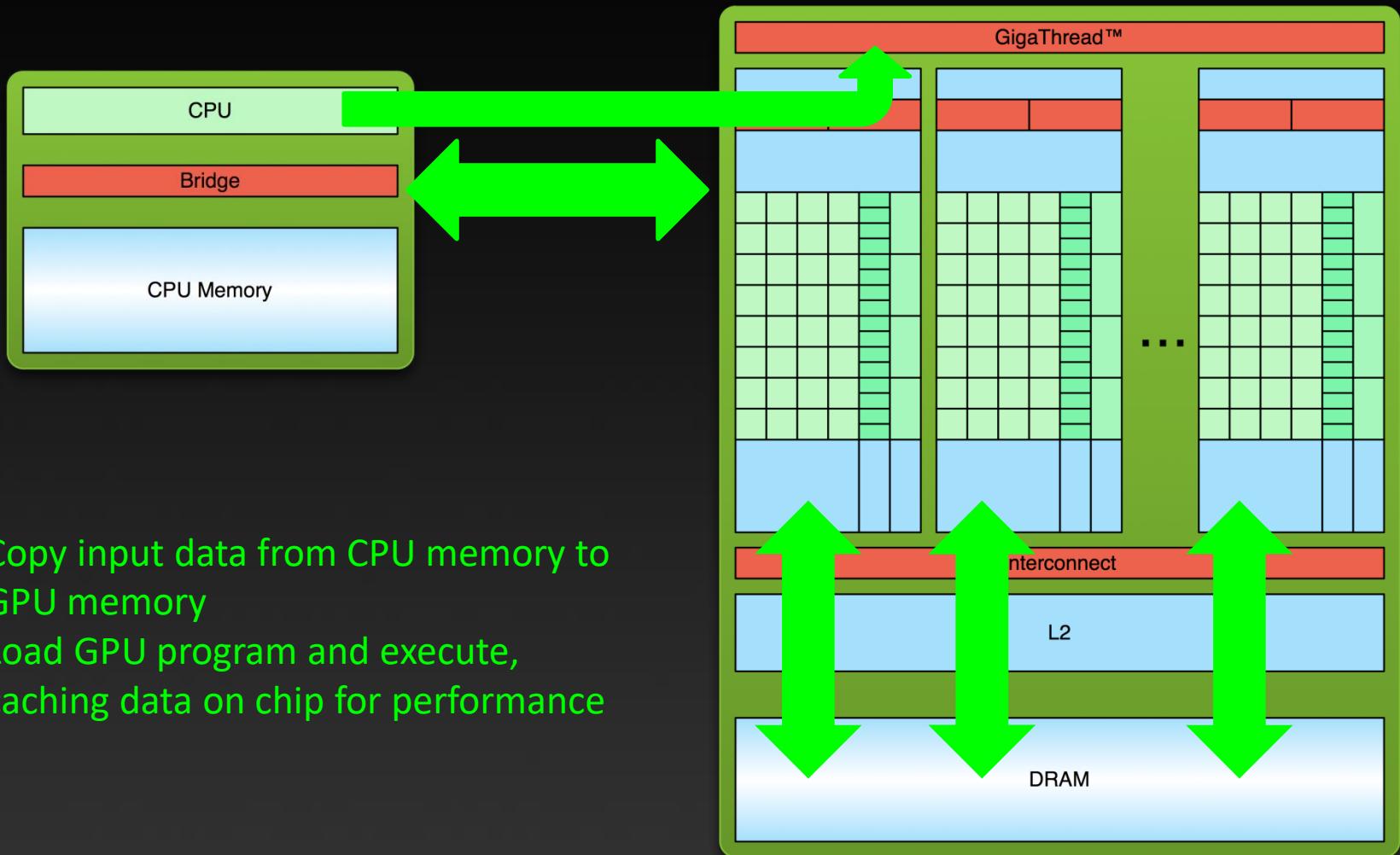
Many Applications are Bandwidth Sensitive



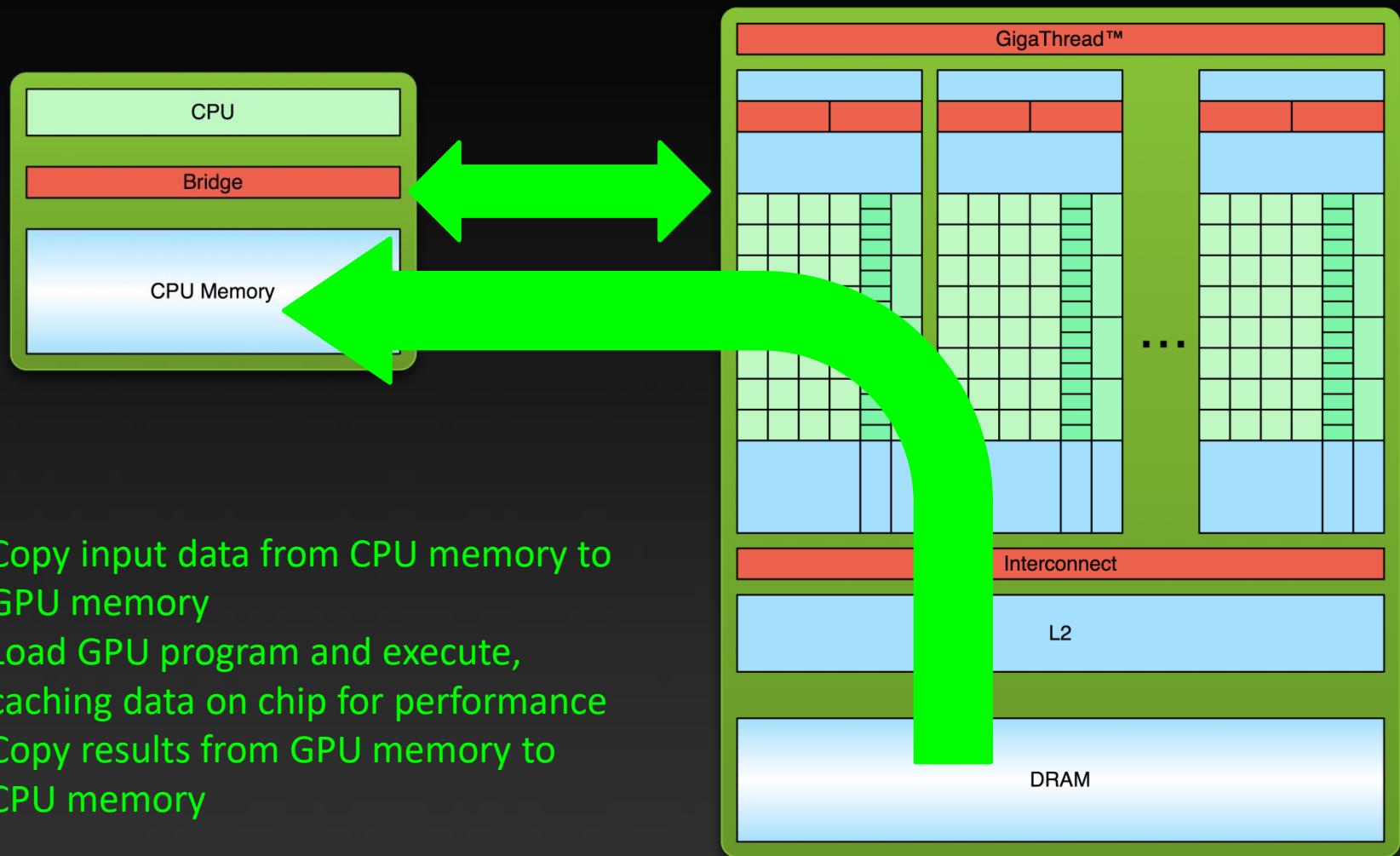
GPU Computing Flow



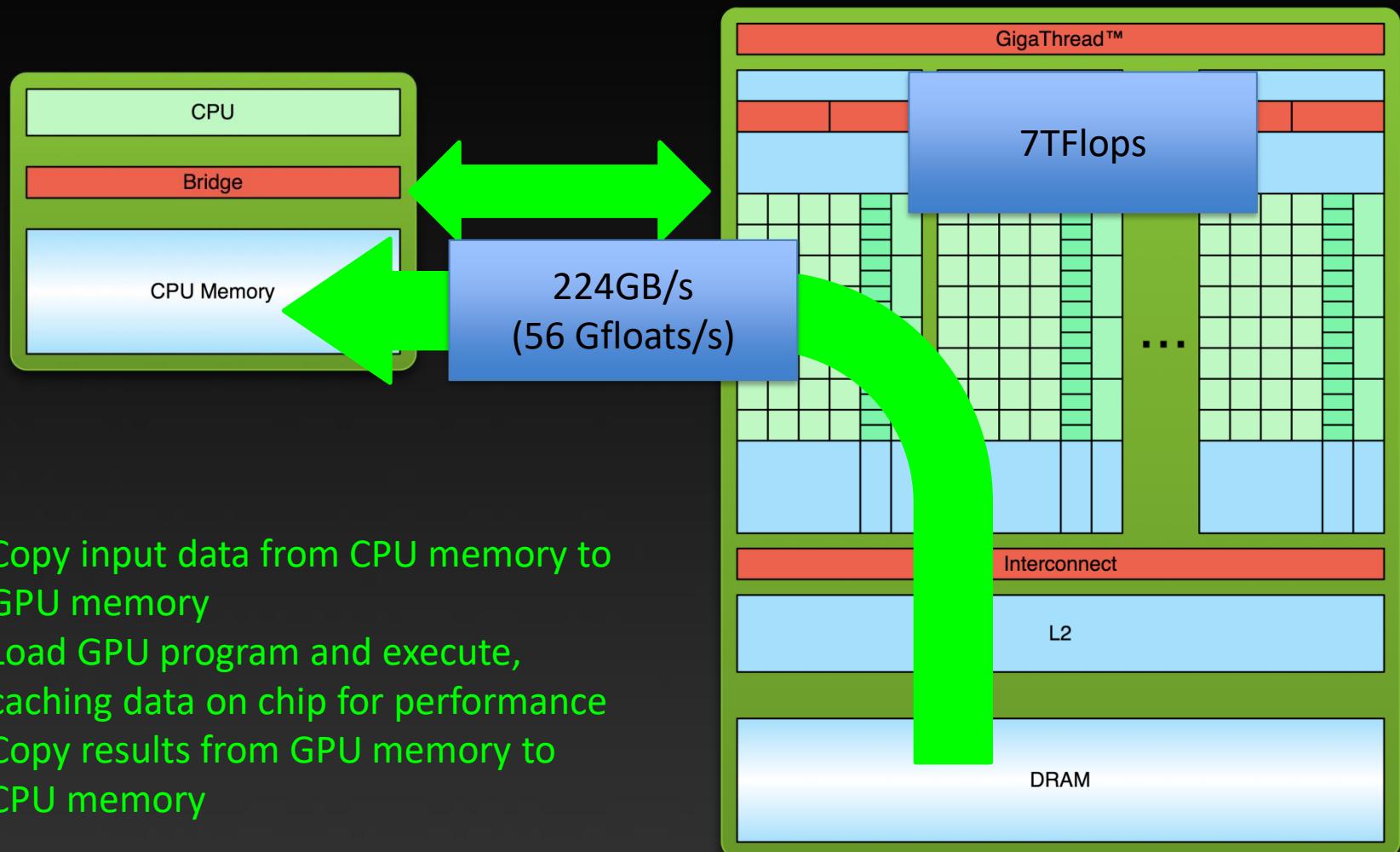
GPU Computing Flow



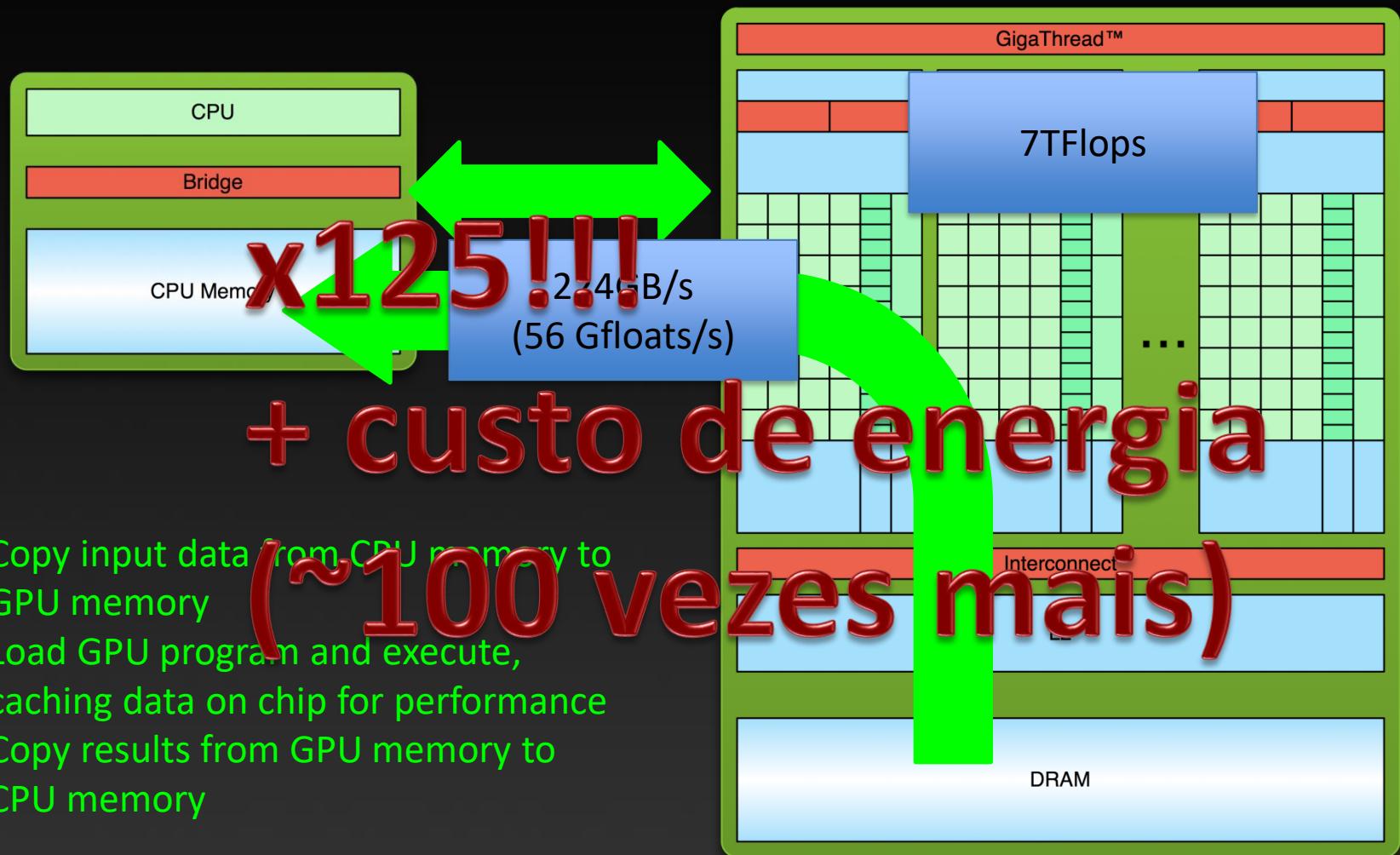
GPU Computing Flow



GPU Computing Flow

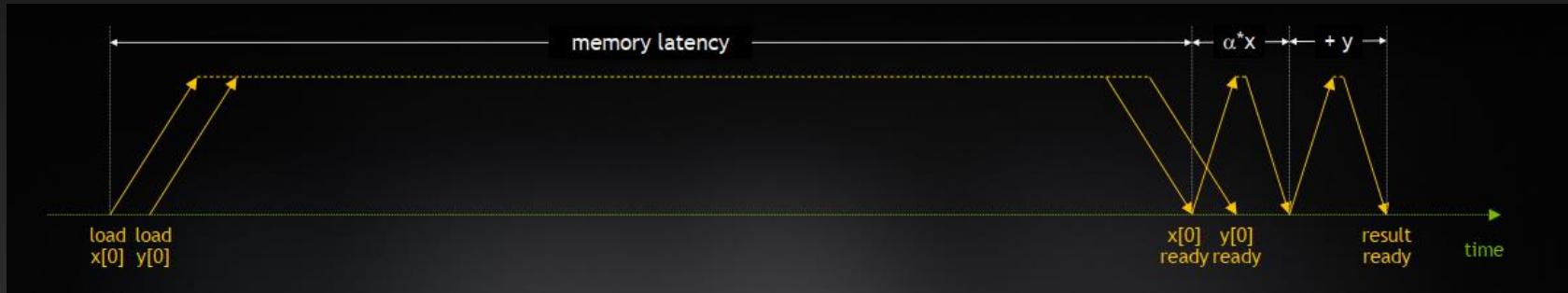


GPU Computing Flow



#2 – A Latência pode ser mais importante que os FLOPS!...

```
void daxpy(int n, double alpha, double *x, double *y)
{
    for( i = 0; i < n; i++ )
    {
        y[i] = alpha * x[i] + y[i];
    }
}
```



Computation x Memory

We want to maximize Arithmetic Operations:

Math / Memory

- Maximize Math operations per thread
- Minimize time spent on memory per thread





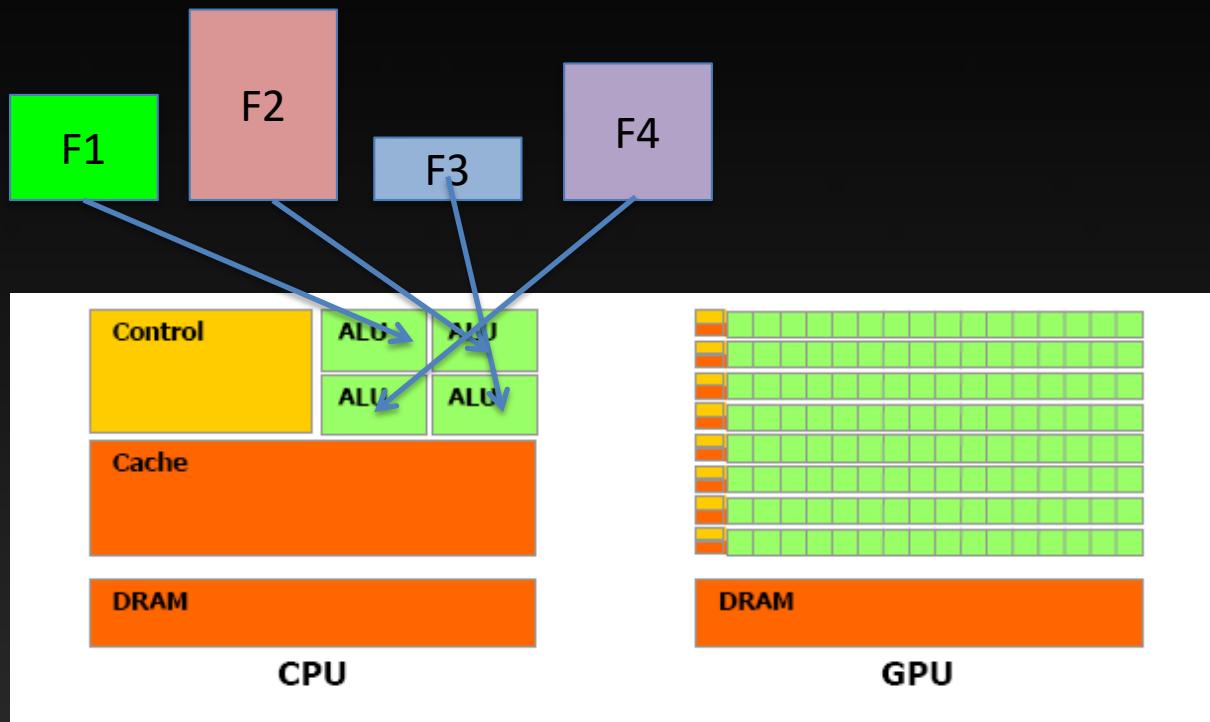
#3 – You can not pretend there is no parallelism...



#4 – 1 kernels, lots of threads...



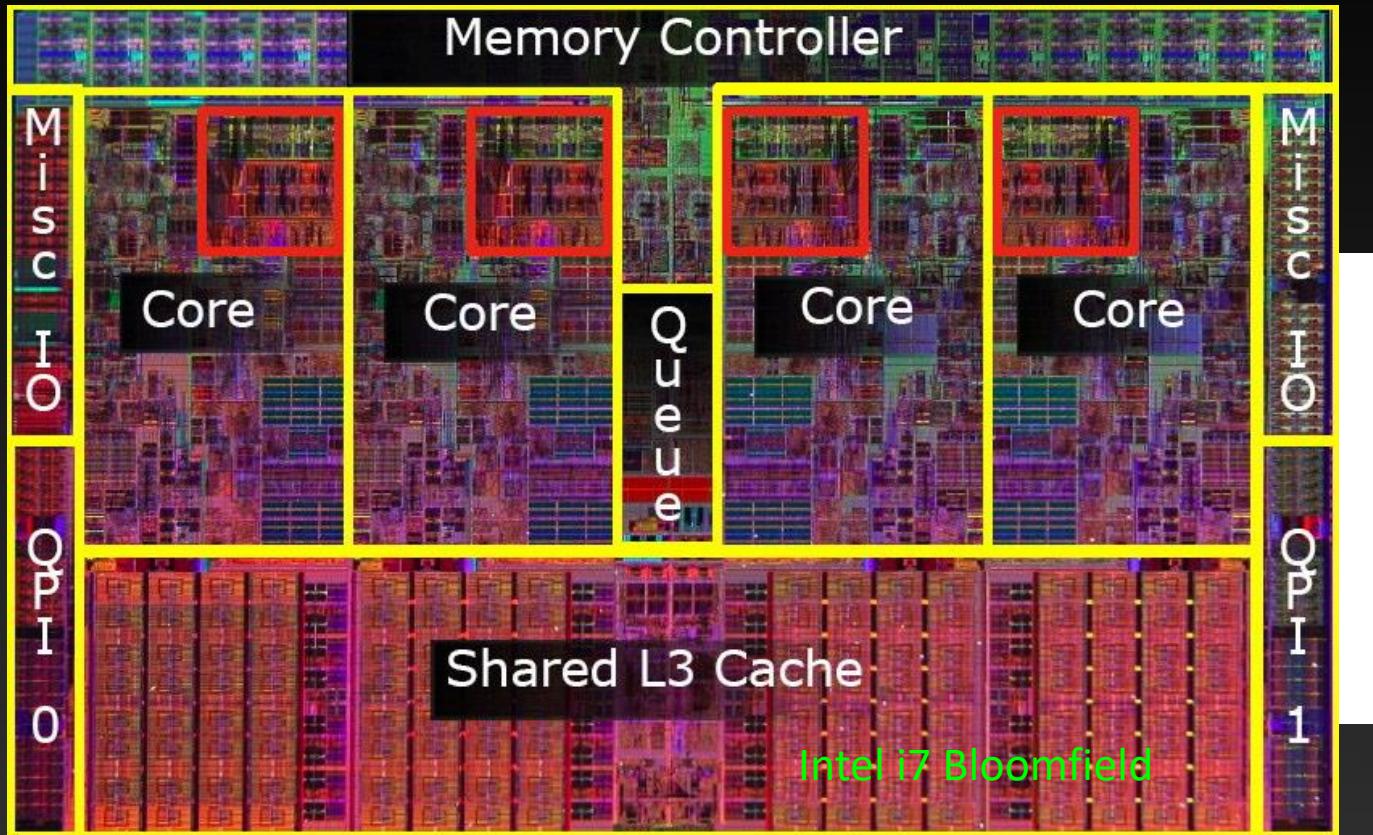
How things work at GPU x CPU



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

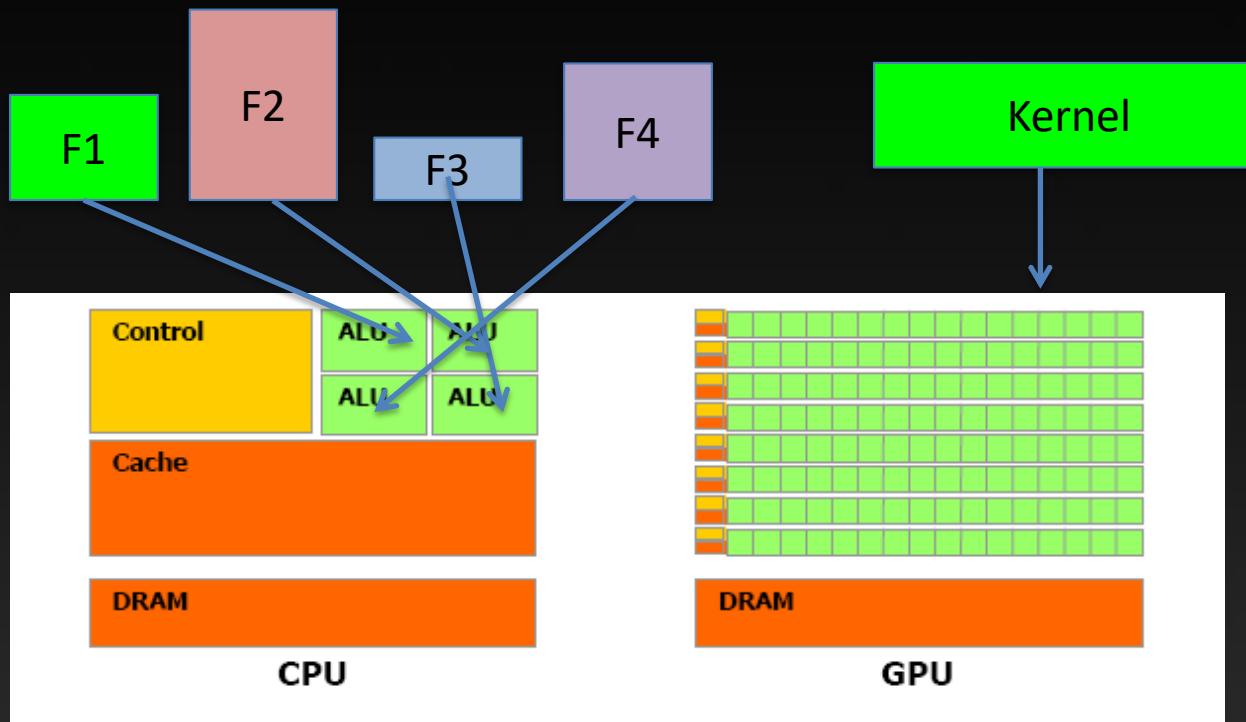


How things work at GPU x CPU



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

How things work at GPU x CPU



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



How things work at GPU x CPU

SIMT

means

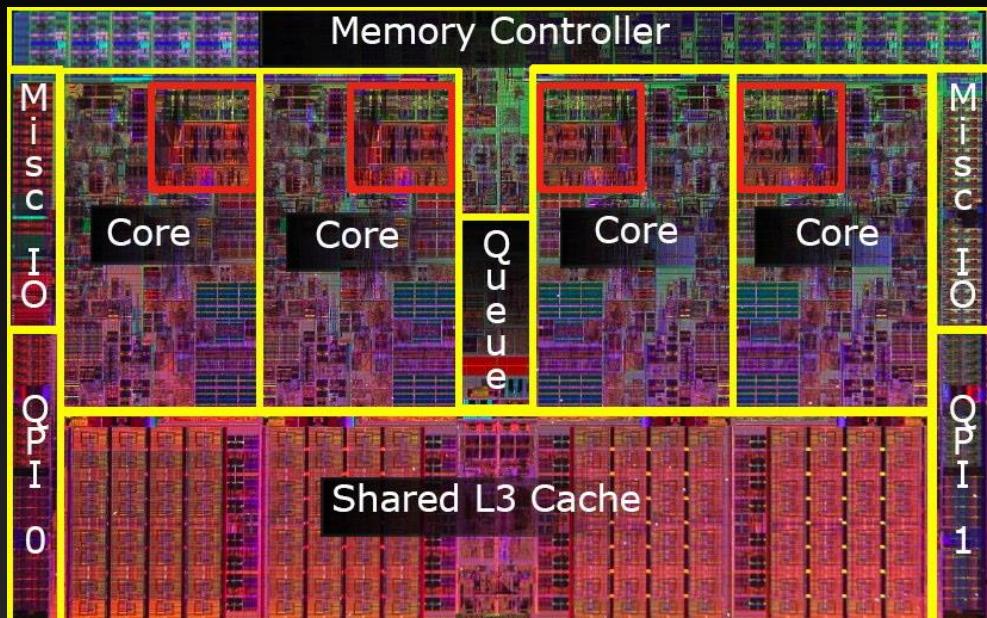
Single Instruction
Multiple Thread

...

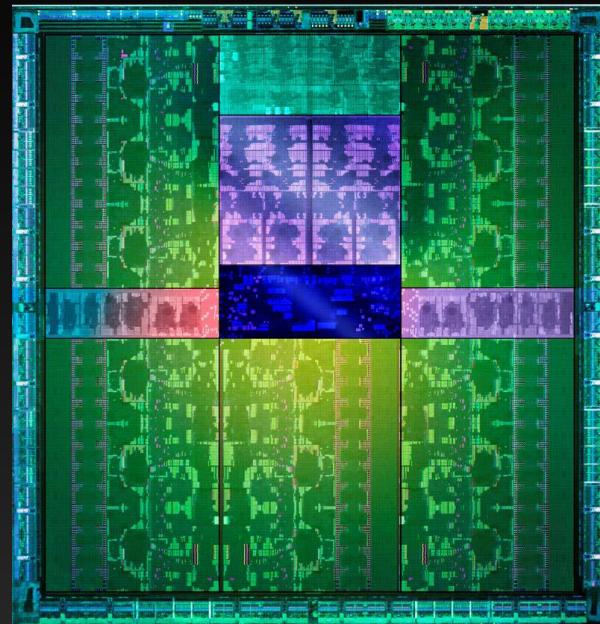
by allacronyms.com



How things work at GPU x CPU



Intel i7 Bloomfield



Kepler K10

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

Threads – Management Cost

In CPUs we use a small number of threads and it is very natural to spend 1000 instructions for changing from one thread to another

In GPUs there is another paradigm...

There is no thread management....

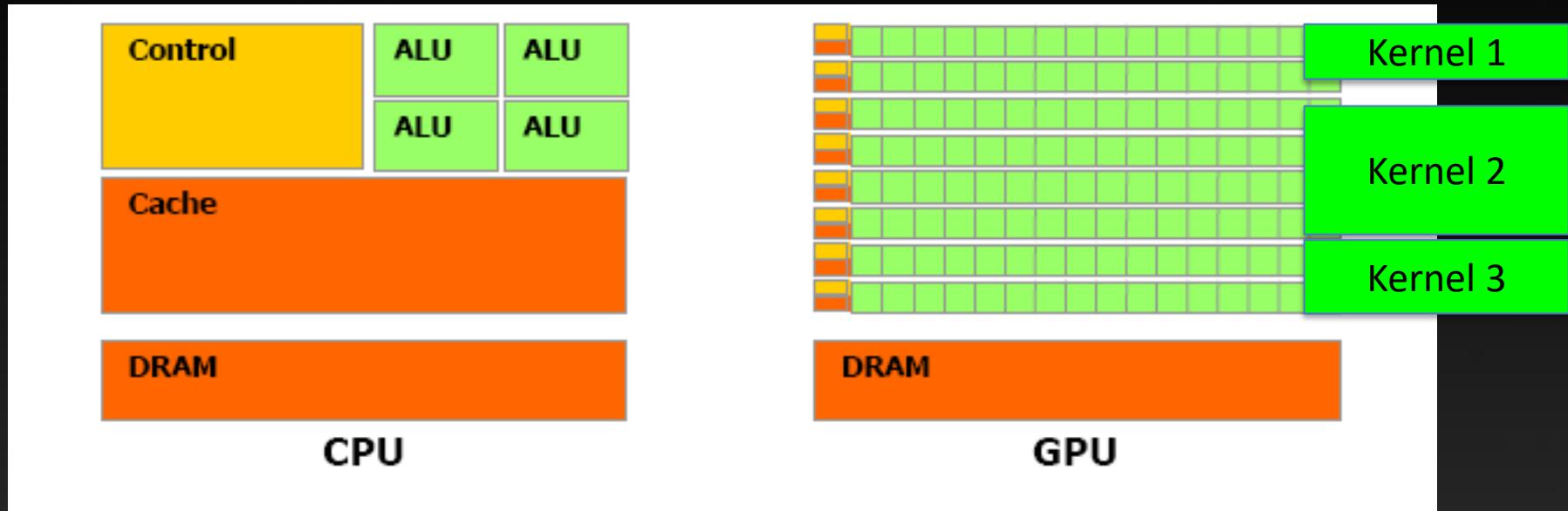


Kernel, Threads e Warps

SIMT Architecture



How things work at GPU x CPU



Concurrent Kernels



MPI can deliver different Kernels for the same GPU



Main CUDA concepts

Device: the GPU

Host: the CPU

Kernel – Program Instance

Thread – kernel Instance

Global Memory: main device memory

Main memory: main host memory

CUDA, PTX and Cubin





CudaPAD

File Edit Compiler Tools Help

>> PTX

Auto

Ready

```
/* Welcome to CudaPAD. */

//a modified version of Nvidia's timedReduction
extern "C" __global__ void timedReduction(
    const float * input,
    float * output,
    clock_t * timer)
{
    extern __shared__ float shared[];
    const char* some_string = "testing";
    const int tid = threadIdx.x;
    const int bid = blockIdx.x;

    if (tid == 0) timer[bid] = clock();

    // Copy input.
    shared[tid] = input[tid];
    shared[tid + blockDim.x] = input[tid + 1];

    // Perform reduction to find minimum.
    for(int d = blockDim.x; d > 0; d /= 2)
    {
        __syncthreads();

        if (tid < d)
        {
            float f0 = shared[tid];
            float f1 = shared[tid + d];
        }
    }
}
```

```
//del// .reg .s32      %r<28>;
//del// .reg .f32      %f<5>;

ld.param.u32      %r8, [timedReduct
/*new*/ ld.param.u32      %r9, [timedReduct
ld.param.u32      %r10, [timedReduct
/*new*/ cvta.to.global.u32  %r1, %r10
/*new*/ mov.u32       %r2, %ctaid.x;
mov.u32          %r3, %tid.x;
setp.ne.s32     %p2, %r3, 0;
@%p2 bra         BB5_2;

//del// cvta.to.global.u32  %r__, %r_
//del// mov.u32       %r__, %ctaid.x;
// inline asm
mov.u32          %r11, %clock;
// inline asm
shl.b32          %r12, %r2, 2;
add.s32          %r13, %r1, %r12;
st.global.u32   [%r13], %r11;

BB5_2:
cvta.to.global.u32  %r14, %r8
shl.b32          %r15, %r3, 2;
add.s32          %r16, %r14, %r15;
mov.u32          %r17, shared;
add.s32          %r4, %r17, %r15;
```

< >

Line Message

⚠ 9 variable "some_string" was declared but never referenced

< > Line: 11 Compile: 4496

Memory--- Global: 0 Constant:
N/ASpills--- Shared: N/A Stack Frame: 0
Loads: 0 Stores: 0

Regs--- Used: 8

^

^

^

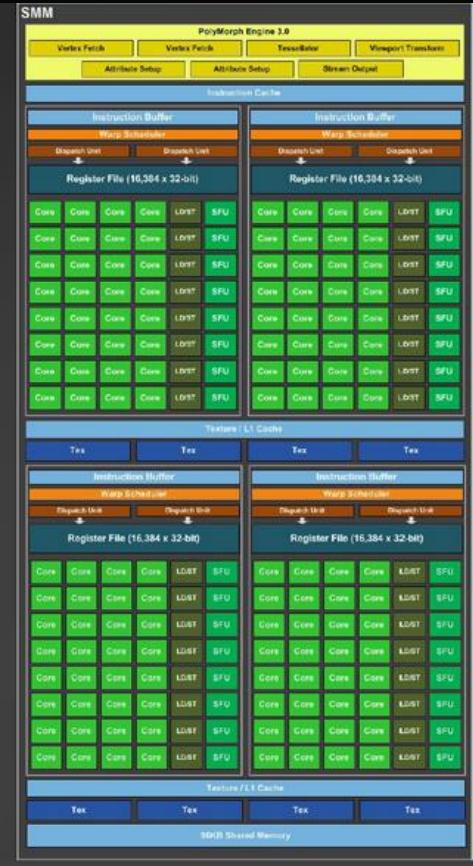
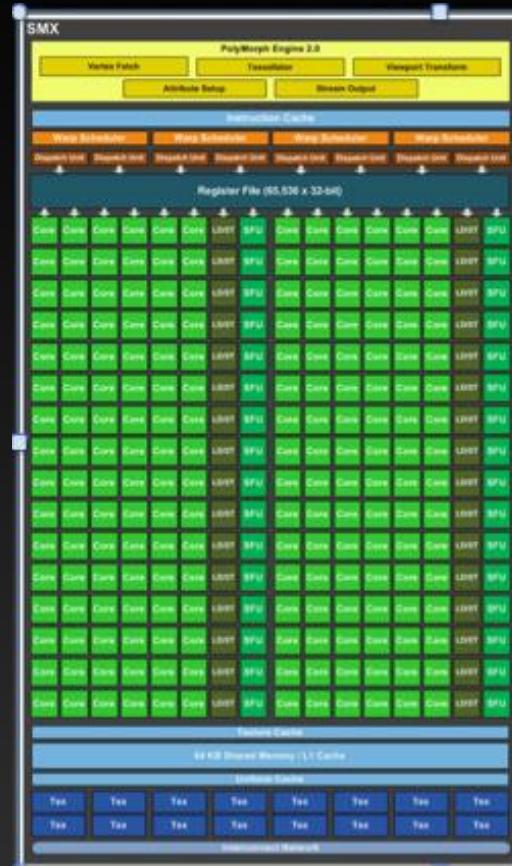
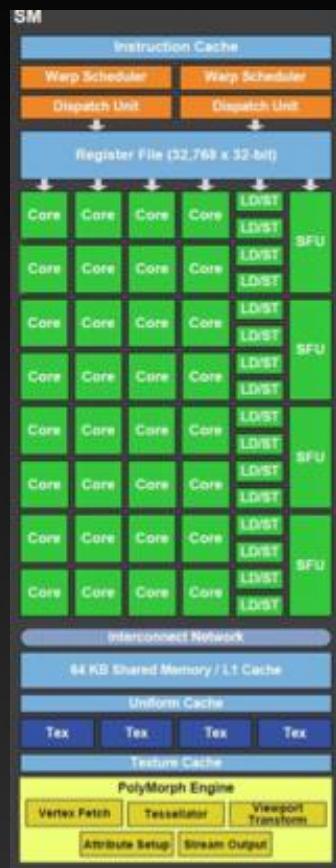
^

^

Grid



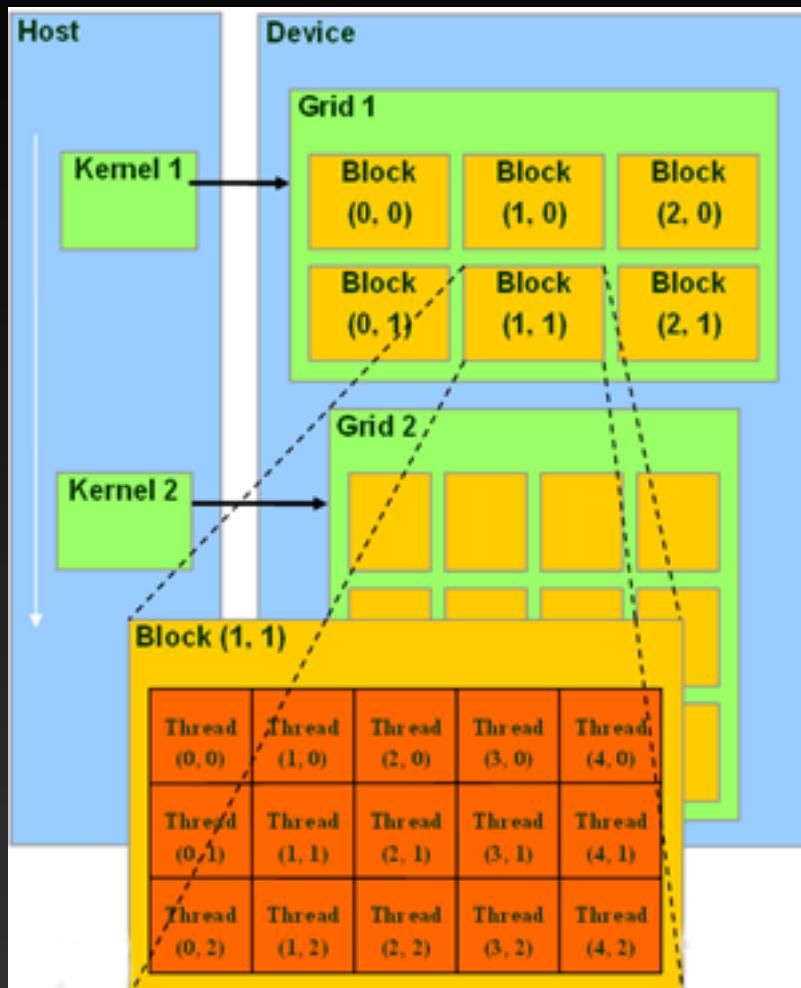
Stream Multiprocessor - Block



Stream Multiprocessor - Block



Threads, Blocos and Grids



One kernel will be executed in each Grid

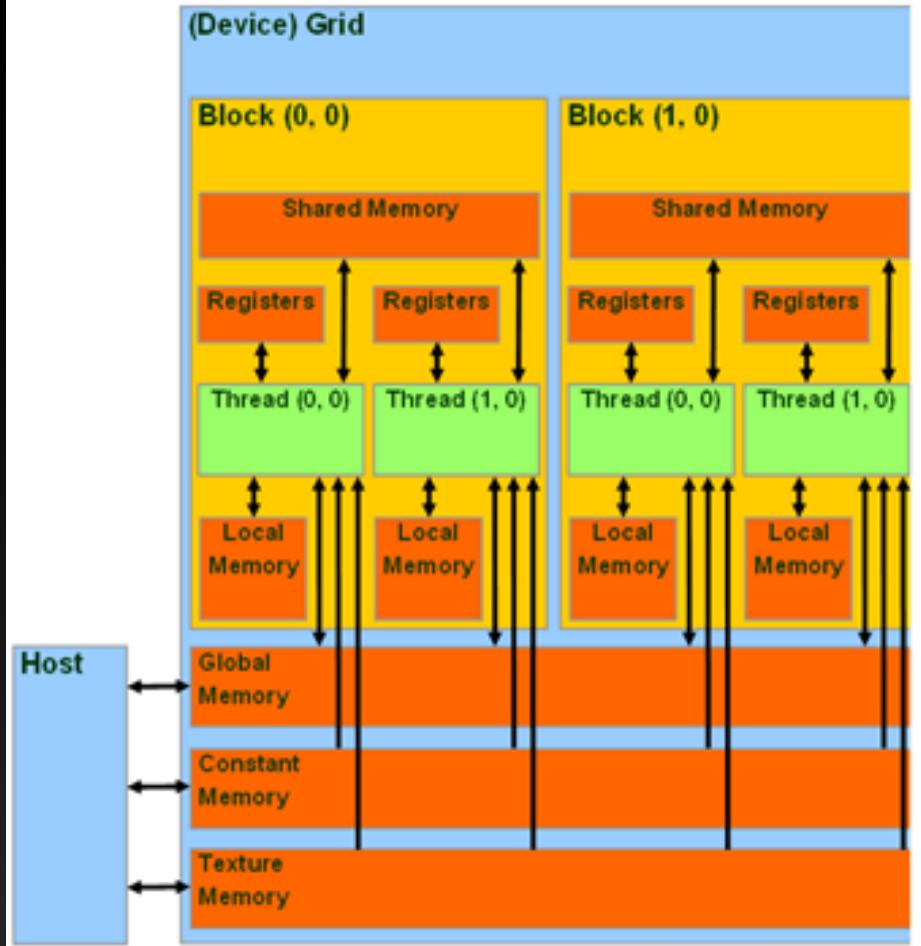
Each Block is composed by threads (1024)

All threads of a block can use the same shared memory

Threads of different Blocks can't share the same shared memory, but can use the same global memory

Memories...

- Memory Hierarchy
- Local
- Cache L1 and L2
- shared
- Constant
- Texture
- Global



How to use the GPU

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



How to use the GPU

Developer &
Application
Ecosystem

Frameworks

PyTorch, TensorFlow, Jax,
Modulus, Triton, ...

SDKs

Medical Devices, Energy,
Autonomous Vehicles, ...

NVIDIA
Accelerated
Libraries

Host Libraries

cuBLAS, cuFFT, cusolver,
cuTENSOR, NPP, cuRAND, ...

Source Libraries

libc++, CUB, Thrust,
CUTLASS, Warp, ...

Parallel
Languages

GPU-Native

CUDA C++, OpenCL,
CUDA FORTRAN

Standard Languages

OpenACC, OpenMP,
Std C++20

3rd Party

Numba, JuliaGPU,
MATLAB, XLA, ...

Compiler
Targets

NVVM / LLVM IR

PTX Assembly ISA



Work Proposal: OpenACC



Libraries

Linear Algebra

FFT, BLAS,
SPARSE, Matrix



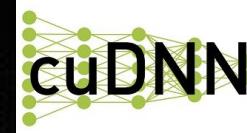
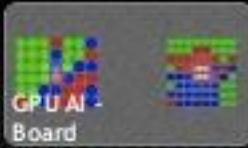
Numerical & Math

RAND, Statistics



Data Struct. & AI

Sort, Scan, Zero Sum



Visual Processing

Image & Video



- Don't require any GPU knowledge or even parallel computing experience
- EASY OF USE, DROP IN, QUALITY OF IMPLEMENTATION

Libraries – ex. Thrust

```
thrust::device_vector<float> deviceInput1(inputLength);
thrust::device_vector<float> deviceInput2(inputLength);
thrust::device_vector<float> deviceOutput(inputLength);

thrust::copy(hostInput1, hostInput1 + inputLength,
            deviceInput1.begin());
thrust::copy(hostInput2, hostInput2 + inputLength,
            deviceInput2.begin());

thrust::transform(deviceInput1.begin(), deviceInput1.end(),
                 deviceInput2.begin(), deviceOutput.begin(),
                 thrust::plus<float>());
```



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

EASY OF USE, PORTABLE, UNCERTAIN PERFORMANCE



CUDA Tools



CUDA DEVELOPER TOOLS: OVERVIEW & EXCITING NEW FEATURES

Rafael Campana, Software Engineering Director, NVIDIA



<https://on-demand.gputechconf.com/gtc/2020/s22043/>



Programming Language

More Efficient

Flexible: may be better optimized for specific platforms

Verbose: express more details



CUDA

```
//Vector size in elements
const int N = 1048576;
//Vector size in bytes
const int dataSize = N * sizeof(float);

//CPU memory allocation
float *h_A = (float *)malloc(dataSize);
float *h_B = (float *)malloc(dataSize);
float *h_C = (float *)malloc(dataSize);

//GPU memory allocation
float *d_A, *d_B, *d_C;
cudaMalloc((void **) &d_A, dataSize);
cudaMalloc((void **) &d_B, dataSize);
cudaMalloc((void **) &d_C, dataSize);

//Initialize h_A[], h_B[]...

//Copy input data to GPU for processing
cudaMemcpy(d_A, h_A, dataSize, cudaMemcpyHostToDevice) ;
cudaMemcpy(d_B, h_B, dataSize, cudaMemcpyHostToDevice) ;

//Run the core of N / 256 units, 256 streams each
//Assuming that N is multiple of 256
vectorAdd<<<N / 256, 256>>>(d_C, d_A, d_B);

//Read GPU results
cudaMemcpy(h_C, d_C, dataSize, cudaMemcpyDeviceToHost) ;
```



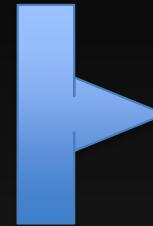
Funções em CUDA

	Executa no	Chamado no
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host



Hello World

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



GPU

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



CPU



Hello World

```
$ nvcc main.cu // Obs. Not main.cc
```

```
$ ./a.out
```

```
$ nvcc main.cu -o hello
```

```
$ ./hello
```



Compiling a GPU program

Name file as .cu

```
nvcc name.cu -o name2
```

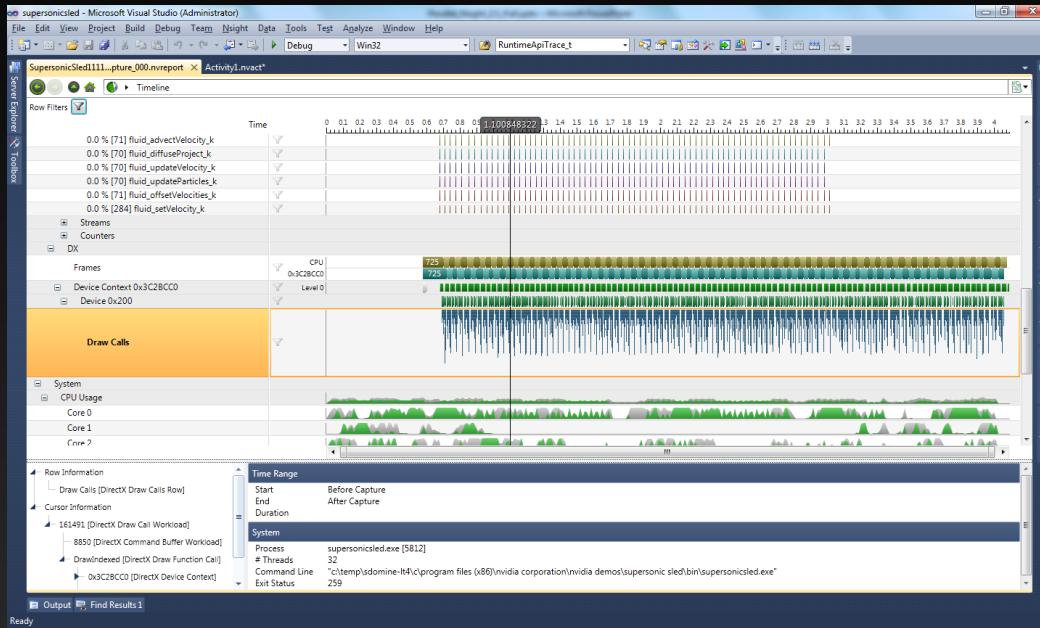
```
./name2
```

Voilá! . . .



Debuggers

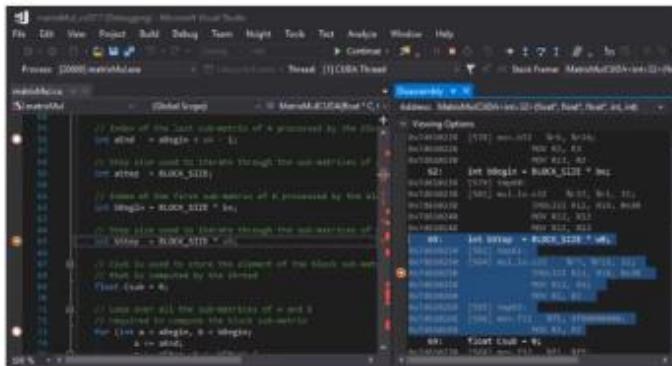
NSight



Debuggers

Developer Tools Ecosystem

Debuggers: cuda-gdb, Nsight Visual Studio Edition
Nsight Visual Studio **Code** Edition



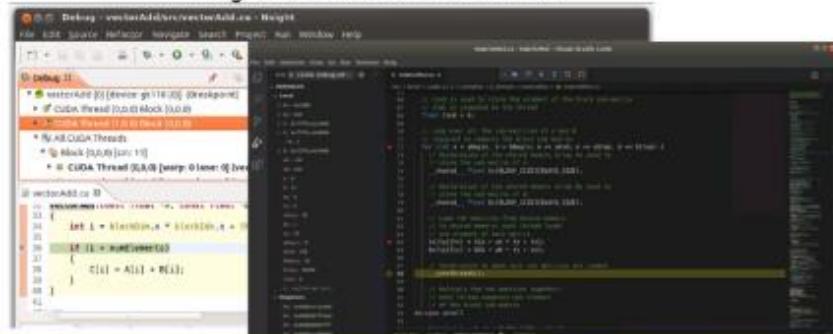
Correctness Checker: Compute Sanitizer

```
$ compute-sanitizer --leak-check full memcheck_demo
===== COMPUTE-SANITIZER
Allocating memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
===== Invalid __global__ write of size 4 bytes
===== at 0x60 in memcheck_demo.cu:6:unaligned_kernel(void)
===== by thread (0,0,0) in block (0,0,0)
===== Address 0x400100001 is misaligned
```

Profilers: Nsight Systems, Nsight Compute, CUPTI, NVIDIA Tools eXtension (NVTX)

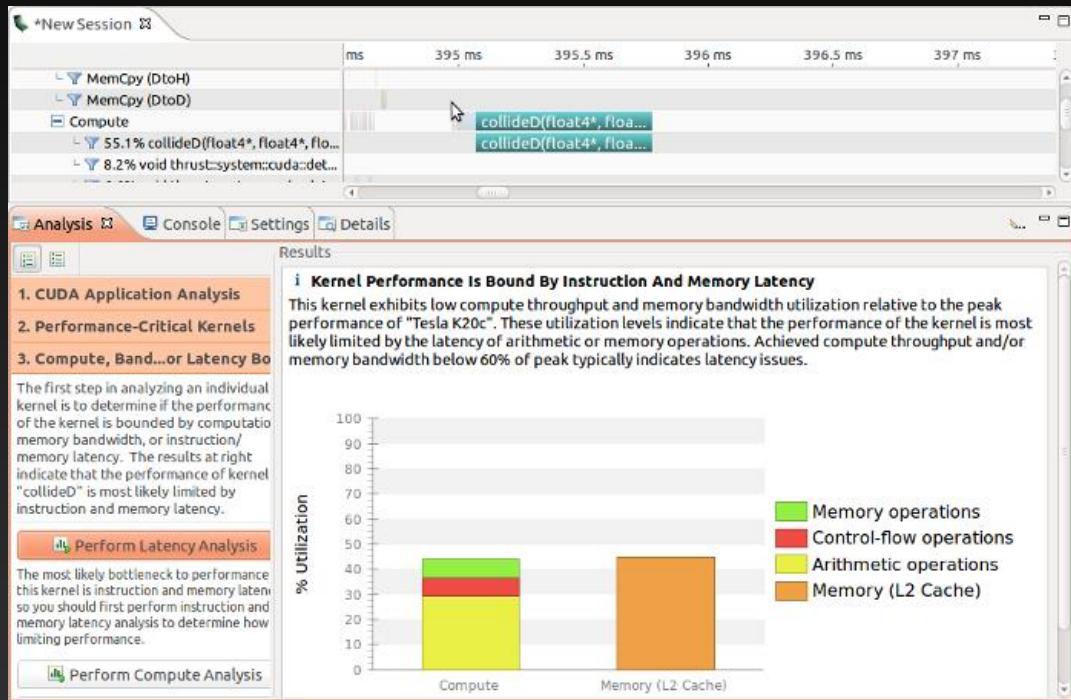


IDE integrations: Nsight Eclipse Edition
Nsight Visual Studio Edition
Nsight Visual Studio **Code** Edition



Debuggers

CUDA Memcheck



Profiler tools

NSIGHT

NVPROF



NVIDIA NVProf

```
Nvprof ./a.out
```

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



Trabalho

CUDA MEMCHECK



Compiler Flags

Host compiler: C/C++ code

-Xcompiler: forward flag to host compiler
Ex. -Xcompiler -fopenmp



Hello World

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

// __global__ → C Language Construct call: Declaration
Specifier (Declspec)
```



Feeding the GPU with Data...

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



Feeding the GPU with Data...

```
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;
```



Feeding the GPU with Data...

```
// 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) ;
```



cudaDeviceSynchronize ()



cudaDeviceSynchronize ()

CPU



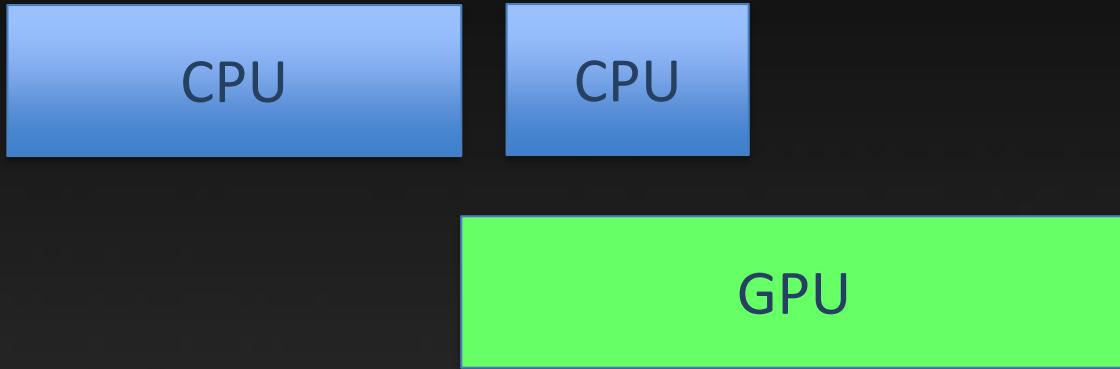
cudaDeviceSynchronize ()

CPU

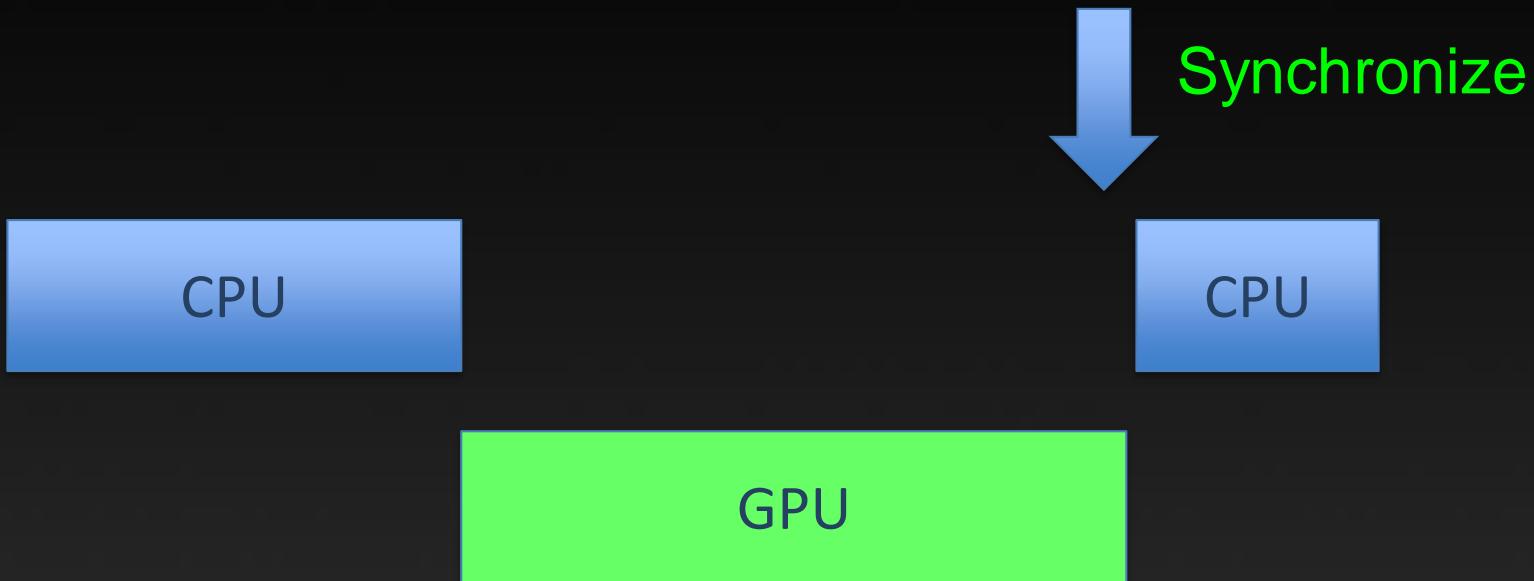
GPU



cudaDeviceSynchronize ()



cudaDeviceSynchronize ()



cudaMemset

```
// CPU -> GPU
cudaMalloc((void **) &d_a, size);
cudaMemset((void *) d_a, 0, size);

cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

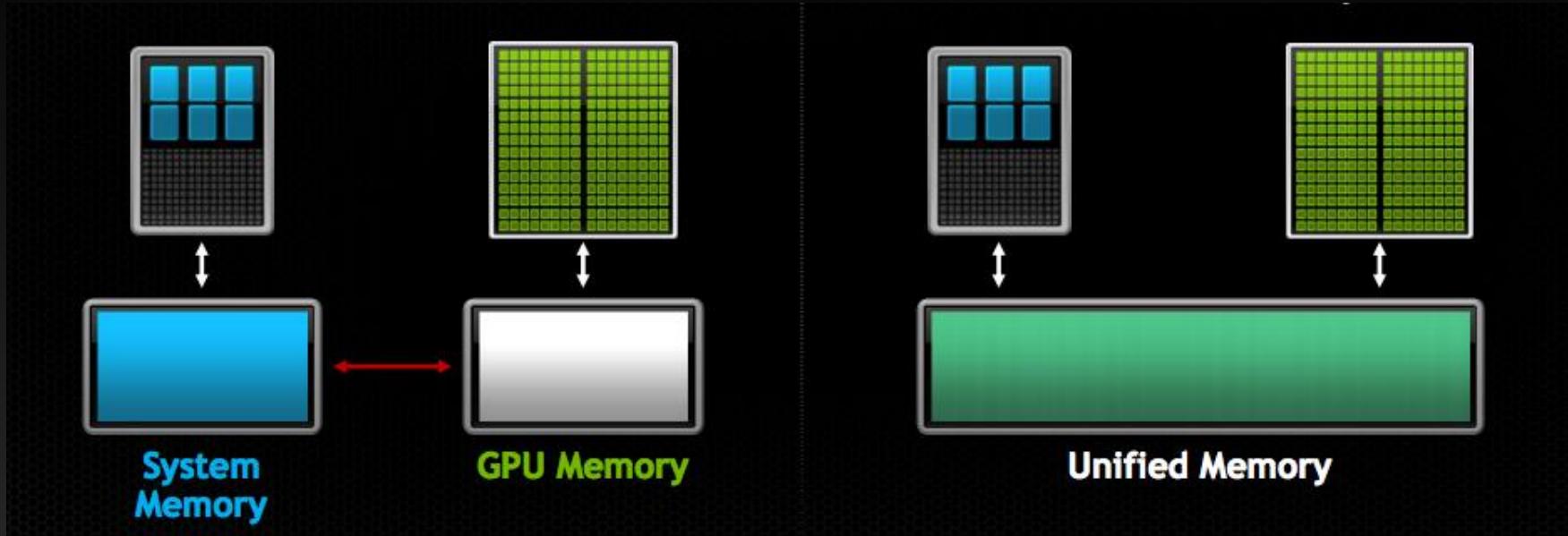
// kernel execution: 1 thread
add<<<1,1>>>(d_a, d_b, d_c);

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

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



Memória unificada



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);

// Syncrhonize
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
add<<<10,100>>>();
```

```
// Syncrhonize
cudaDeviceSynchronize();
```



Errors

```
cudaError_t error;

error = cudaMalloc((void **) &d_a, size);

if (error != cudaSuccess) {
    printf ("error: %s in ... ", cudaGetErrorString (error));
    exit (-1);
}
```



Errors types

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

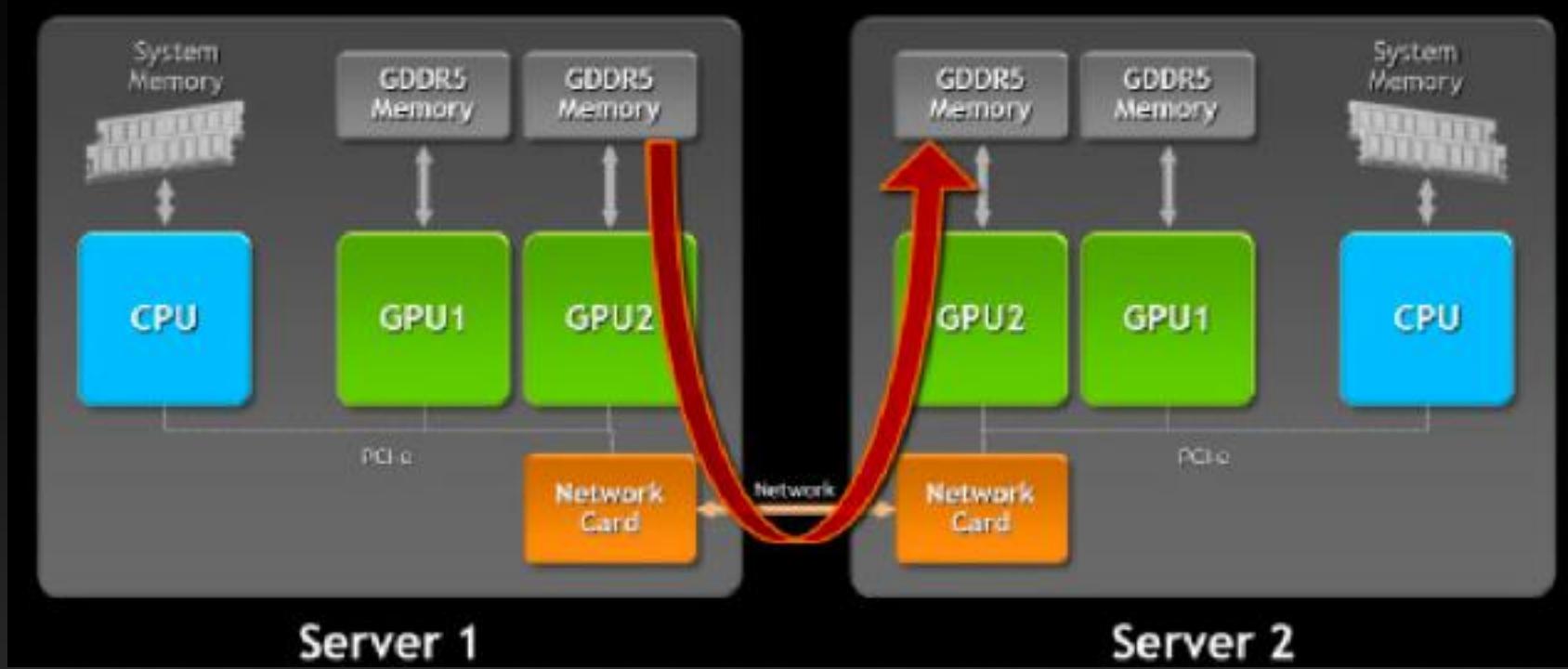
CUDA error types

Enumerator:

<i>cudaSuccess</i>	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 <code>cudaEventQuery()</code> and <code>cudaStreamQuery()</code>).
<i>cudaErrorMissingConfiguration</i>	The device function being invoked (usually via <code>cudaLaunch()</code>) was not previously configured via the <code>cudaConfigureCall()</code> function.
<i>cudaErrorMemoryAllocation</i>	The API call failed because it was unable to allocate enough memory to perform the requested operation.
<i>cudaErrorInitializationError</i>	The API call failed because the CUDA driver and runtime could not be initialized.
<i>cudaErrorLaunchFailure</i>	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 <code>cudaThreadExit()</code> is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.
<i>cudaErrorPriorLaunchFailure</i>	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.
<i>cudaErrorLaunchTimeout</i>	This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property <code>kernelExecTimeoutEnabled</code> for more information. The device cannot be used until <code>cudaThreadExit()</code> is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.



GPUDirect



<https://developer.nvidia.com/blog/gpudirect-storage/>



Finally... Exploring the Parallelism

```
Void add(int *d_a, int *d_b, int *d_c) {  
    for (i=0; i<N; i++)  
        d_c[i] = d_a[i] + d_b[i];  
}  
  
int main()  
{  
    ...  
    vecAdd (d_a, d_b, d_c);  
}
```



Finally... Exploring the Parallelism

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



Small fix...

```
__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  
}
```



Exploring the Parallelism: 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]  
}
```

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];

At the same time...



There is a limit for the number of threads... Per 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									



Tarefa:

Programe um kernel que multiplica
Todos os valores de um vetor por
um número real qualquer,
que é passado como parâmetro



If $N > 1024$???



Exploring the Parallelism: 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]  
}
```

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];

At the same time...



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];  
...  
...
```

...



We are still using 1 SM!...



Blocks

```
__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);  
}
```



Blocks

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

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

threadIdx.x threadIdx.x threadIdx.x threadIdx.x

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

{ } { } { } { }

blockIdx.x = blockIdx.x = blockIdx.x = blockIdx.x =

0

1

2

3



Danger: unreferenced indices...

```
__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
}

    vecAdd <<< ceil (N/M), M >>>
// ceil guarantees enought blocks...
// ceil N/M = (N-1)/M + 1
```



NVIDIA NVProf

```
Nvprof ./a.out
```

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



How many possible combinations we would have in this code...

```
__global__ void hello()
{
    printf("Hello world! I'm a thread in block %d\n",
blockIdx.x);
}

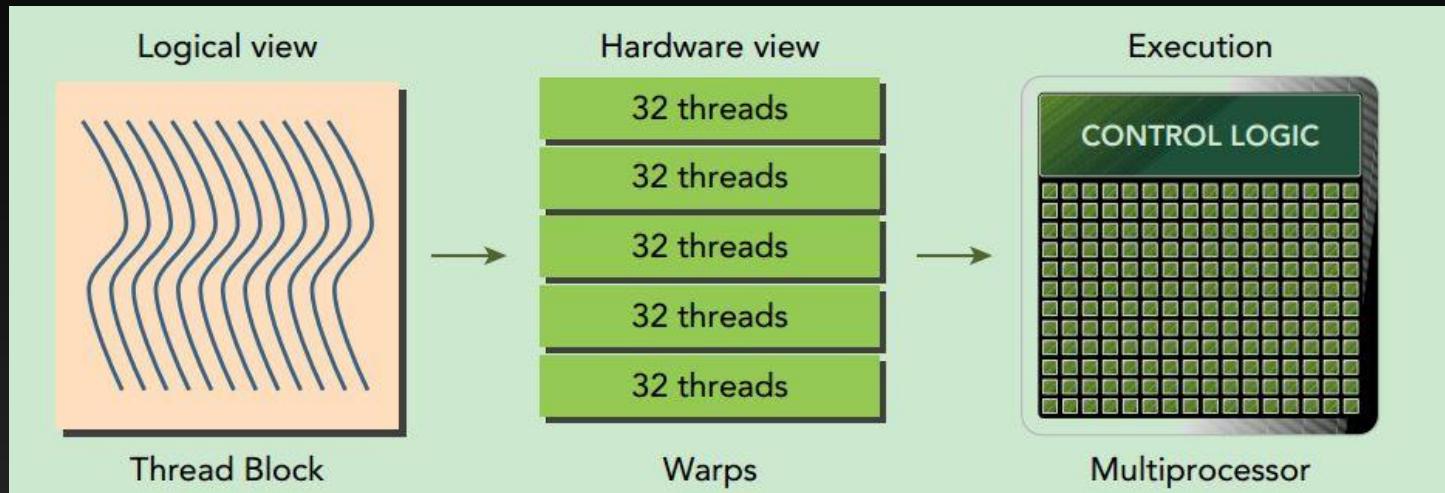
int main(int argc,char **argv)
{
    hello<<< 16, 1 >>>();
    // force the printf()s to flush
    cudaDeviceSynchronize();
    return 0;
}
```



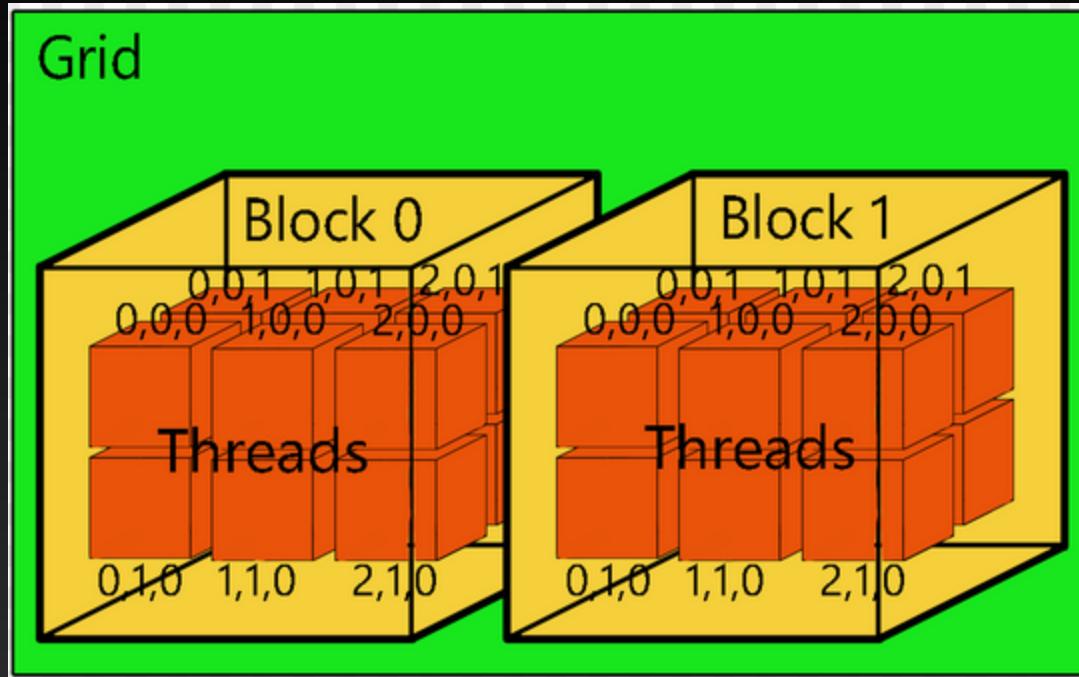
Melhor tamanho de um bloco...



Melhor tamanho de um bloco...

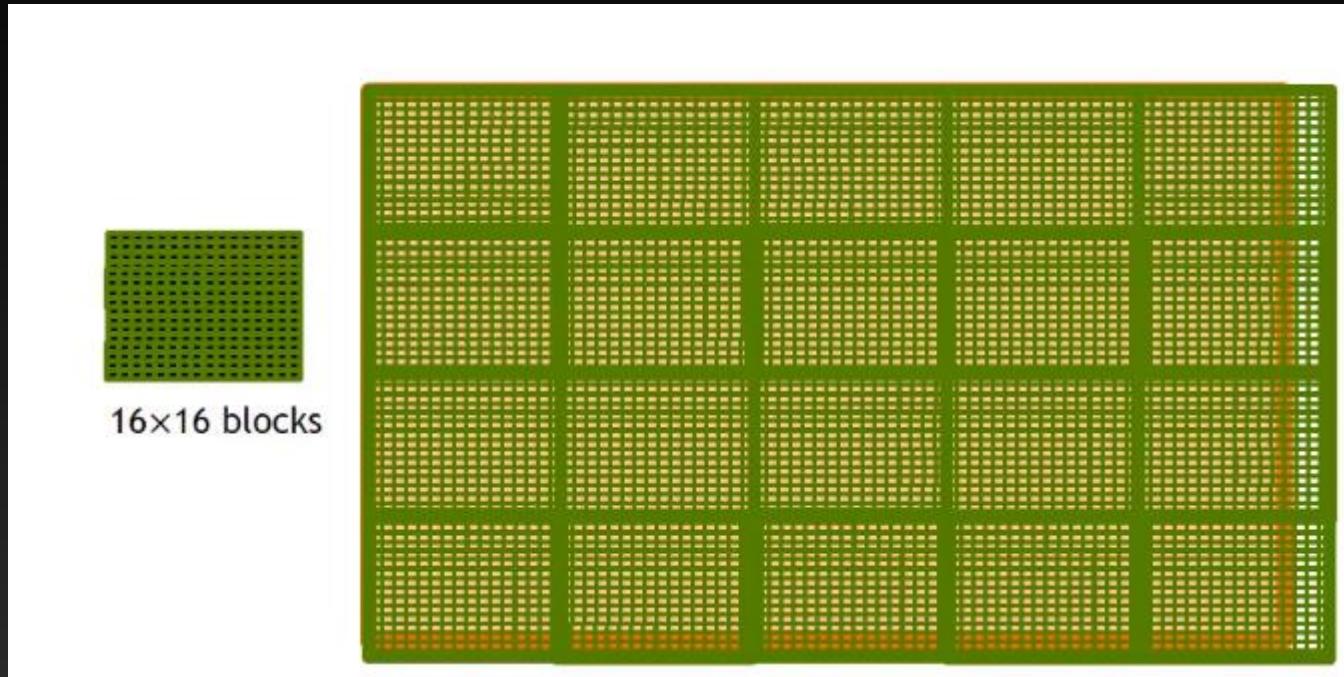


Threads can be indexed with 1, 2 or 3 dimensions



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

Dimension Design



Dim3 type

```
dim3 (x, y, z) = (x, y, z)  
dim3(10) = (10, 1, 1)
```



Threads can be indexed with 1, 2 or 3 dimensions

```
__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);  
}
```



Threads can be indexed with 1, 2 or 3 dimensions

```
__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);  
}
```

Question: how is the best kernel call for $N \times M > 1024$?



Dimension Design

Row-Major Layout in C/C++

M
↓

M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈	M ₉	M ₁₀	M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

M
↓



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}	M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}	M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}	M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}

Mapping pixels to kernels

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                               int height, int width)
{
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;

    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

```
...
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
...
```



Mapping pixels to kernels

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h) {
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    if (Col < w && Row < h) {
        int pixVal = 0;
        int pixels = 0;

        // Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
        for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
            for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

                int curRow = Row + blurRow;
                int curCol = Col + blurCol;
                // Verify we have a valid image pixel
                if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
                    pixVal += in[curRow * w + curCol];
                    pixels++; // Keep track of number of pixels in the accumulated total
                }
            }
        }

        // Write our new pixel value out
        out[Row * w + Col] = (unsigned char)(pixVal / pixels);
    }
}
```

Matrix Multiplication

(Exercice: write a 2D Grid and 2D Block)



$$C_{r,c} = AB_{r,c} = \sum_{i=1}^n A_{r,i} * B_{i,c}$$

Matrix Multiplication

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}

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}

c_{00}	c_{01}	c_{02}	c_{03}
c_{10}	c_{11}	c_{12}	c_{13}
c_{20}	c_{21}	c_{22}	c_{23}

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

tid_{00}	tid_{01}	tid_{02}	tid_{03}
tid_{10}	tid_{11}	tid_{12}	tid_{13}
tid_{20}	tid_{21}	tid_{22}	tid_{23}

bid_{00}

Matrix Multiplication

(very naive approach)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k] *N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

Matrix Multiplication

(Work: Best Matrix Multiplication for GPUs)



Image manipulation

Grayscaling an image



Gray Scaling an image

```
__global__ void rgba_to_greyscale(const uchar4* const rgbaImage,
                                  unsigned char* const greyImage,
                                  int numRows, int numCols)
{
    int col = threadIdx.x;
    int row = threadIdx.y;          // Include Blocks

    // if ((col < numCols) && (row < numRows)){
    uchar4 rgba = rgbaImage[r * numCols + c];
    float channelSum = .299f * rgba.x + .587f * rgba.y + .114f * rgba.z;
    greyImage[r * numCols + c] = channelSum;
    //}
}

}
```



Gray Scaling an image

```
Int col= threadIdx.x + blockIdx.x * blockDim.x;  
Int row= threadIdx.y + blockIdx.y * blockDim.y;  
  
If (col < Width) && (row < Height)  
    image[row*width + col] = DoSomething image[row*width + col];
```



Gray Scaling an image

```
__global__ void rgba_to_greyscale(const uchar4* const rgbaImage,
                                  unsigned char* const greyImage,
                                  int numRows, int numCols)
{
    int col = threadIdx.x + blockIdx.x*blockDim.x;
    int row = threadIdx.y + blockIdx.y*blockDim.y;

    // if ((col < numCols) && (row < numRows)){
        uchar4 rgba = rgbaImage[r * numCols + c];
        float channelSum = .299f * rgba.x + .587f * rgba.y + .114f * rgba.z;
        greyImage[r * numCols + c] = channelSum;
    //}
}

}
```



Gray Scaling an image

```
Void greyscale(const uchar4 * const h_rgbaImage, uchar4 * const d_rgbaImage,
               unsigned char* const d_greyImage, size_t numRows,
               size_t numCols)
{
    const dim3 blockSize(numCols, numRows, 1); //OPTIMIZE
    const dim3 gridSize( 1, 1, 1);           // OPTIMIZE
    rgba_to_greyscale<<<gridSize, blockSize>>> (d_rgbaImage, d_greyImage,
numRows, numCols);

    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
```



Ceil function

BlockSize = CONTANTE (por exemplo 32)

```
Kernel <<<(n/BlockSize) , BlockDim>>> () ; //Error  
Kernel <<<(n/256) , 256>>> () ; //Error
```

// Correct = (n-1)/256 + 1

```
Kernel <<< ceil (n/256.0) , 256>>> () ;
```



Gray Scaling an image

```
Void greyscale(const uchar4 * const h_In, uchar4 * const d_In,
               unsigned char* const d_Out, size_t numRows,
               size_t numCols)
{
    const dim3 blockSize(32, 32, 1);
    const dim3 gridSize( (Cols-1)/32 + 1, Rows-1)/32 + 1, 1);

    rgba_to_greyscale <<<gridSize, blockSize>>> (d_In, d_Out, Rows, Cols);

    cudaDeviceSynchronize();
    checkCudaErrors(cudaGetLastError());
}
```



Thread Divergence

```
__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  
        ...  
}
```



Thread Divergence - loop

```
__global__ void add(int *d_a) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
  
    for (int i=0; i<index; i++)  
        d_a[i] *=2;  
}
```



Memory Optimizations

- Move frequently-accessed data to the faster memory possible:

LOCAL

SHARED

GLOBAL

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application



Shared Memory

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



Synchronization Barriers

...

```
int i= threadIdx.x;  
__shared__ int array[128];  
array[i] = global_data[threadIdx.x];  
  
if (i < 127)  
    array[i] = array[i+1];  
...
```



Synchronization Barriers

...

```
int i= threadIdx.x;
__shared__ int array[128];
array[i] = global_data[threadIdx.x];
```

```
if (i < 127)
    array[i] = array[i+1];
```

...



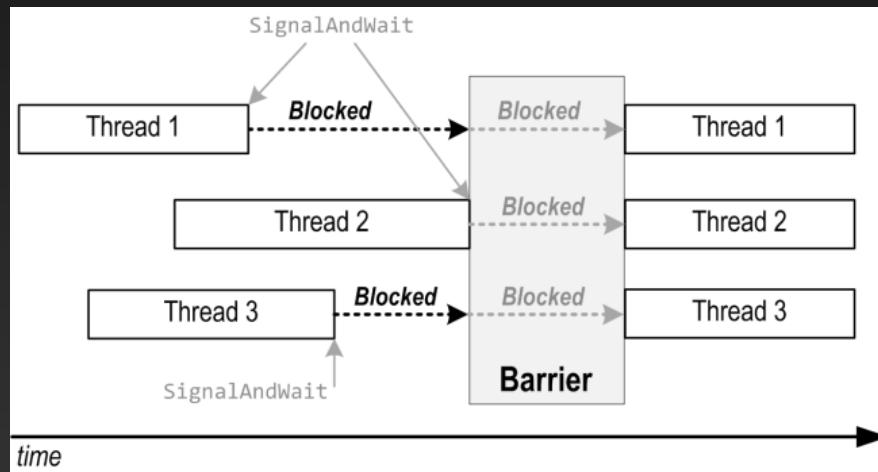
Synchronization

What happens if different threads must access the same memory space
(for read / write operation)?



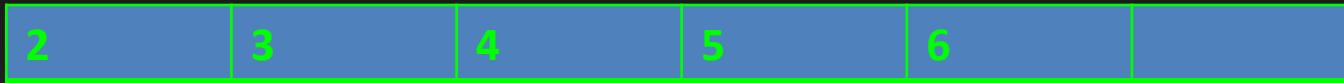
Barrier

A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.



Barrier – Exercise

Write a Kernel for shifting vector elements



Synchronization Barriers

```
MyKernel (...)

...
int i= threadIdx.x + blockIdx.x * blockDim.x;
__shared__ int array[128];
array[threadIdx.x] = global_data[i];

if (i < 127)
    array[i] = array[i+1];
...
```



Synchronization Barriers

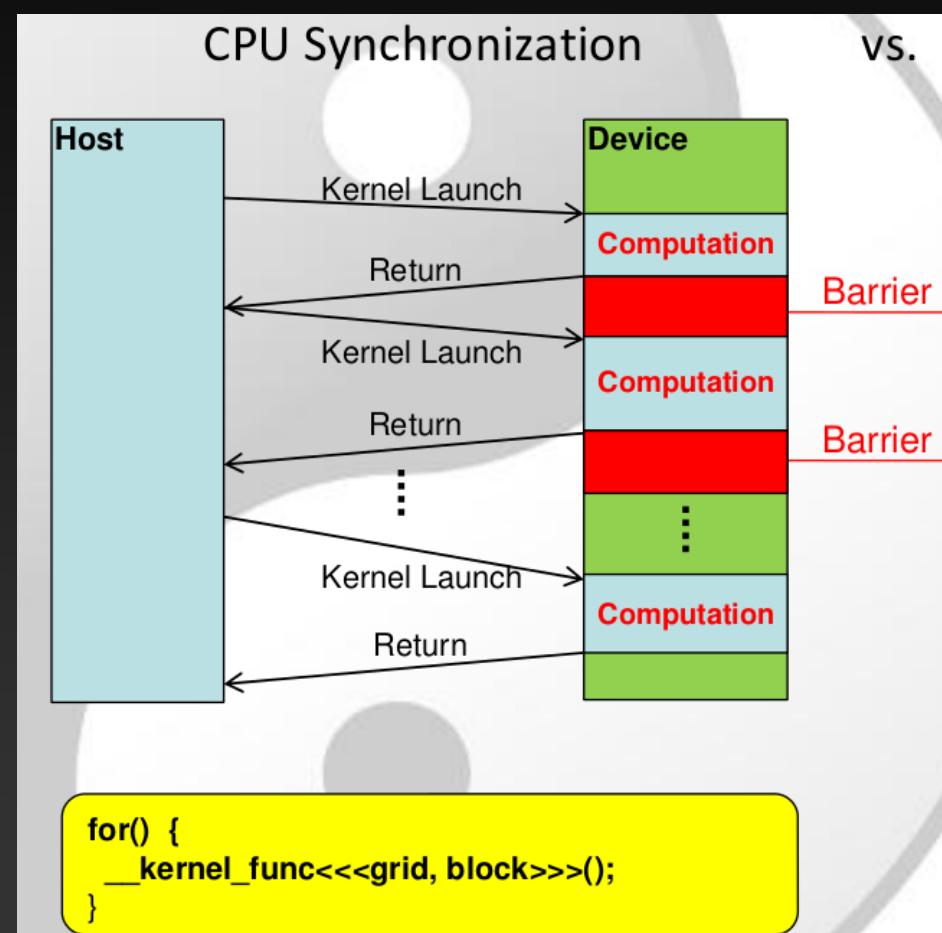
...

```
int i = threadIdx.x + blockIdx.x * blockDim.x;  
__shared__ int array[128];  
array[threadIdx.x] = global_data[i];  
__synchronize(); // Barrier 1  
  
if (threadIdx.x < 127)  
    int temp = array[threadIdx.x + 1];  
    __syncthread(); // Barrier 2  
    array[threadIdx.x] = temp;  
    __syncthread(); // Barrier 3
```



How to Synchronize in GRID level?

- Consecutive Kernel calls
- CPU Synchronization



Shared Memory and Synchronization

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

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

    __syncthread();

    ...
}
```



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???



Analyising the efficiency of Shared Memory

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

    __shared__ float temp_data[256];
    temp_data[threadIdx.x] = data[index];

    __syncthread();

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

...



Analyising the efficiency of Shared Memory

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

    __shared__ float temp_data[256];
    temp_data[threadIdx.x] = data[index];

    __syncthread();

    for (i=0; i<25; i++)
    {
        aux += temp_data[i];
    }
    data[threadIdx.x] = aux;
    temp_data[index] = 0; // what happens here???
```



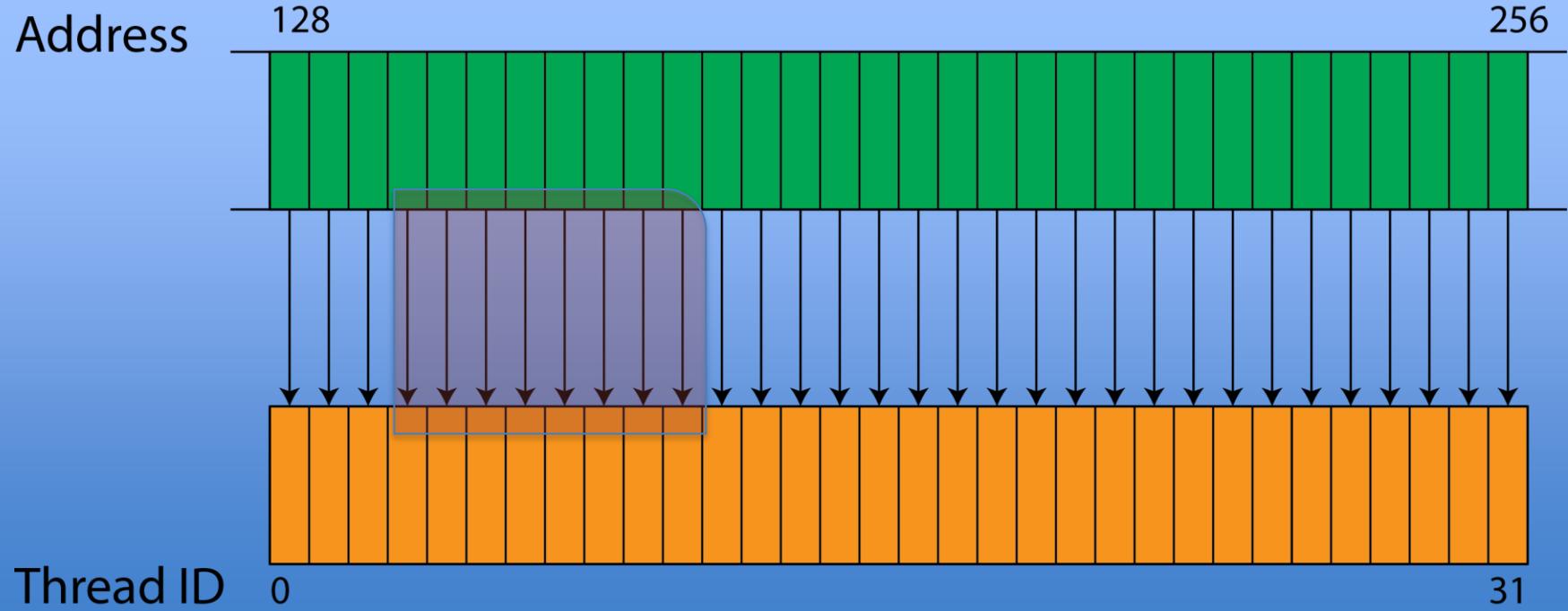
Events and Time

Events in CUDA are a GPU time stamp recorded by the user

```
cudaEvent_t start, stop;  
cudaEventCreate (&start);  
cudaEventCreate (&stop);  
cudaEventRecord (start, 0); // 0 is the stream number  
  
// do Work...  
  
cudaEventRecord (stop, 0);  
cudaEventSynchronize (stop);  
  
float elapsedTime;  
cudaEventElapsedTime (&elapsedTime, start, stop);  
printf ("Total GPU Time: %3.1f ms \n", elapsedtime);  
cudaEventDestroy(start);  
cudaEventDestroy(stop);
```

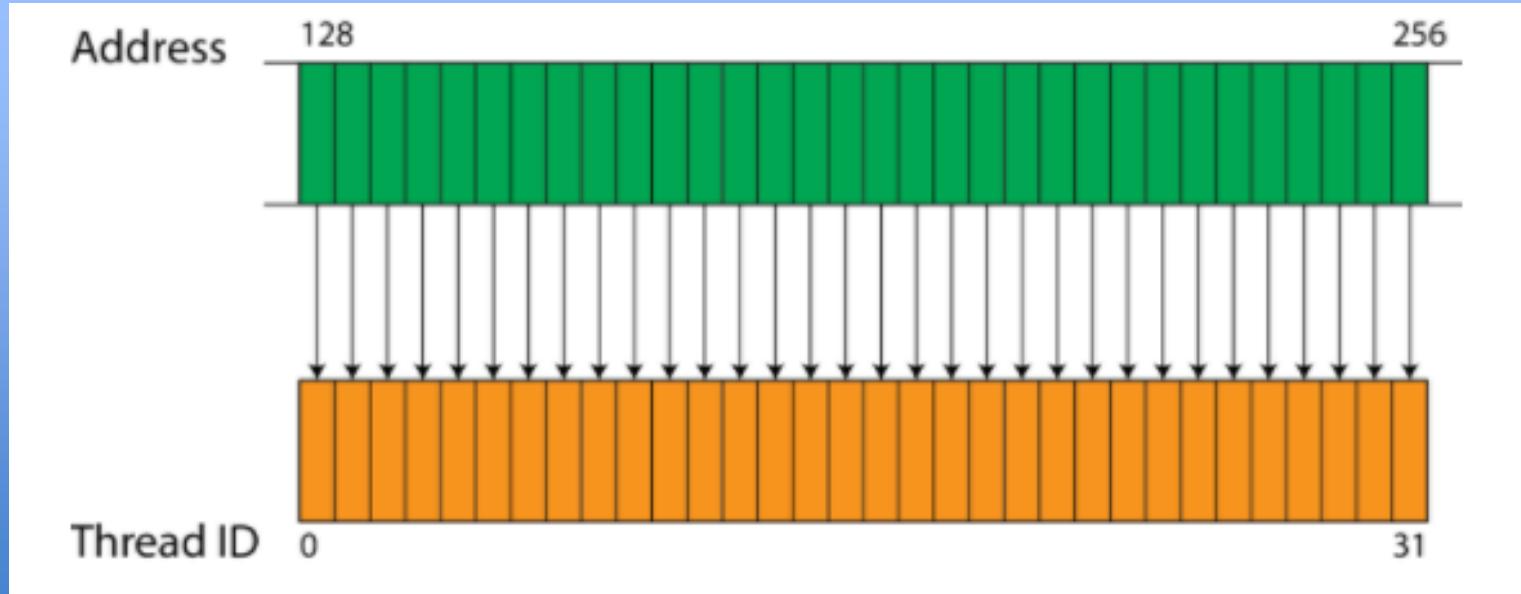


Coalescence reading

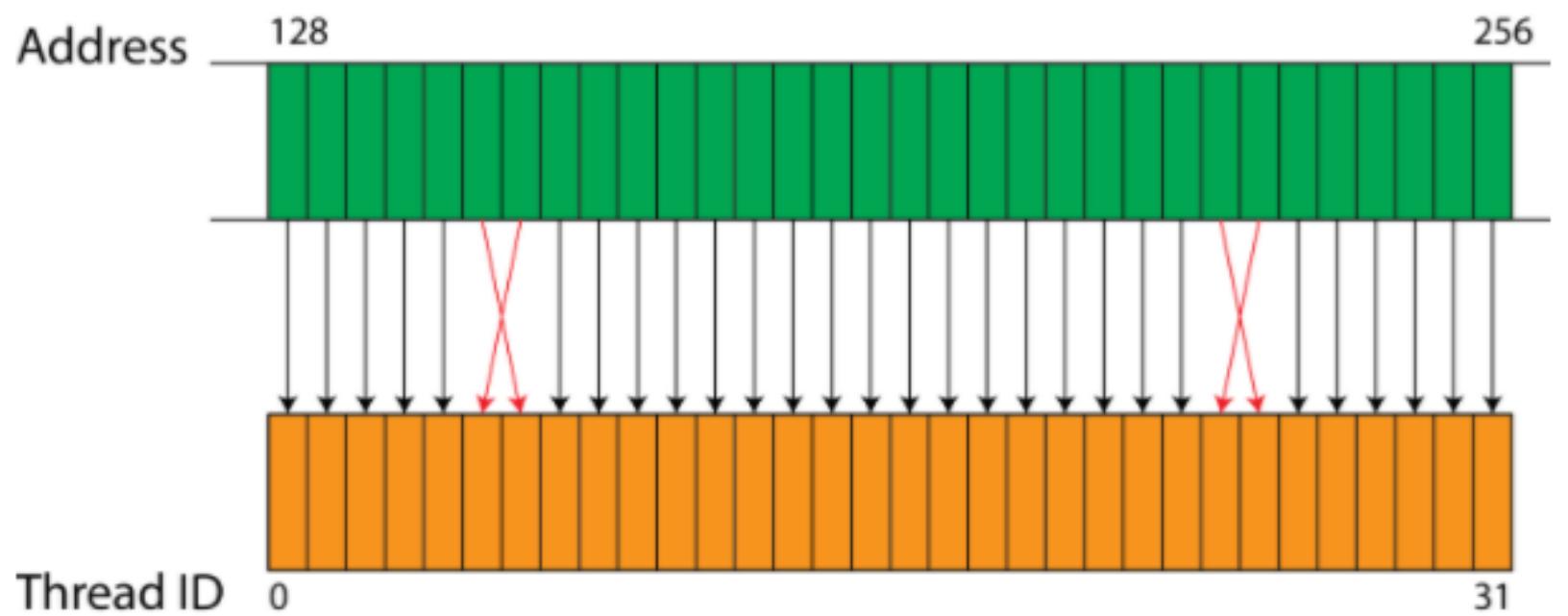


- DRAM (dynamic random access memories) uses parallel approach, due its big latency.

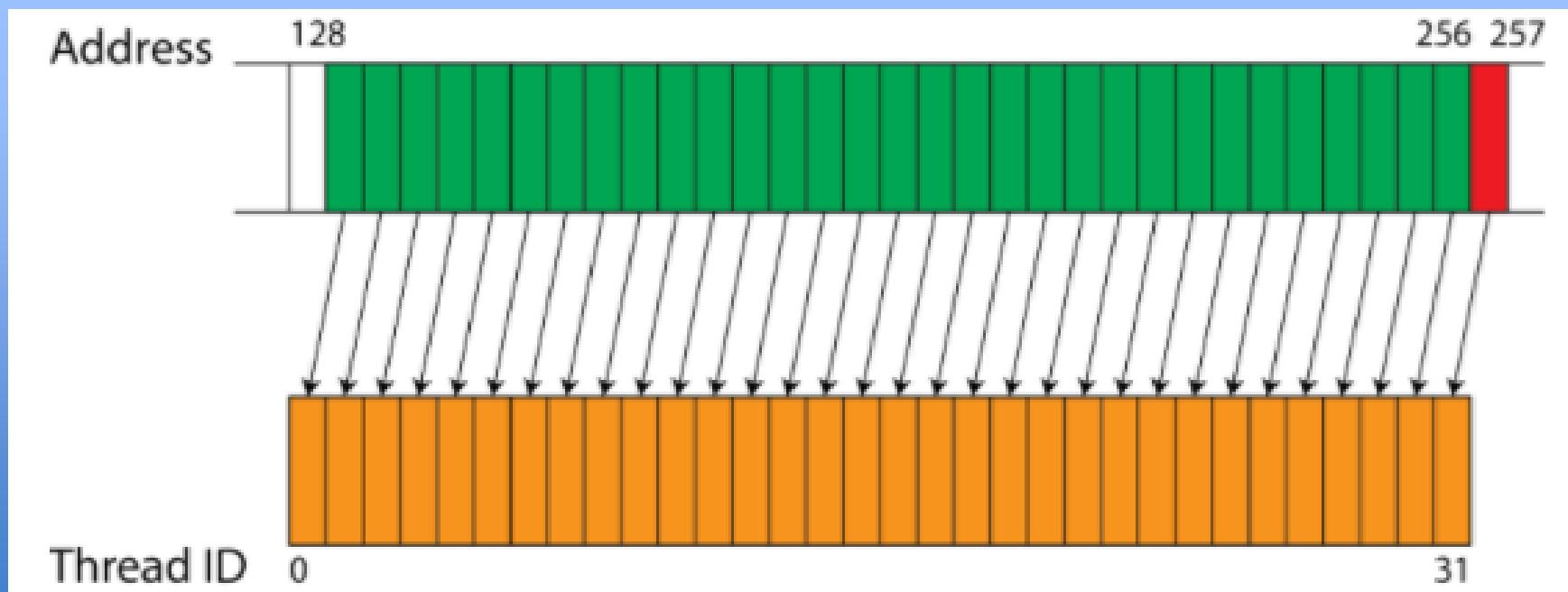
Aligned and Sequential



Aligned but not sequential



unaligned



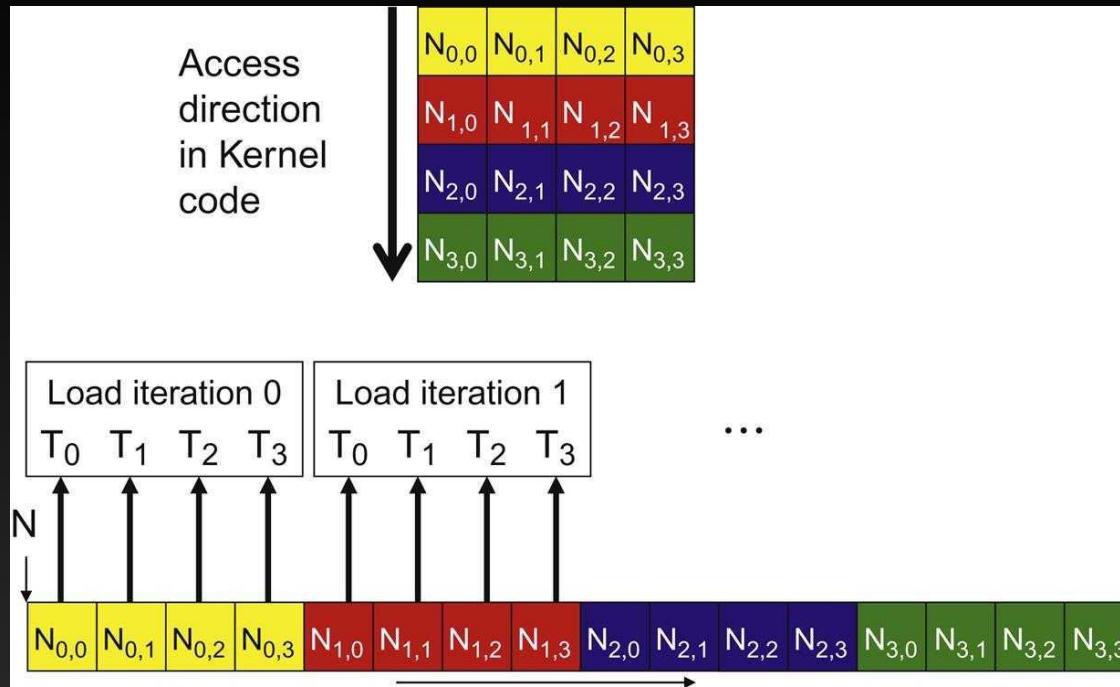
Example

```
shmem[threadIdx.x]=gmem[blockIdx.x*blockDim.x + threadIdx.x];
```

```
stride=4;  
shmem[threadIdx.x]=gmem[stride*blockIdx.x*blockDim.x + threadIdx.x*stride];
```



Example



Example of Coalescence

```
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.



Example of Coalescence

```
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 }
```



Exercice: write the same code
of the shared memory vector
copy with no coalescence

Stride: when there is no coalescence access



Atomic Operations

Many threads trying to write at the same
global memory space



Exercice: what happens in this code???

```
#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;
}
```



And now?

```
#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;
}
```

Run this program (with timer)



Exercice: Correct the code using barriers...

```
#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);
}
```



List of 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
```



Limitations of 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



Exercice: Write an Image Filter Kernel (convolution kernel)

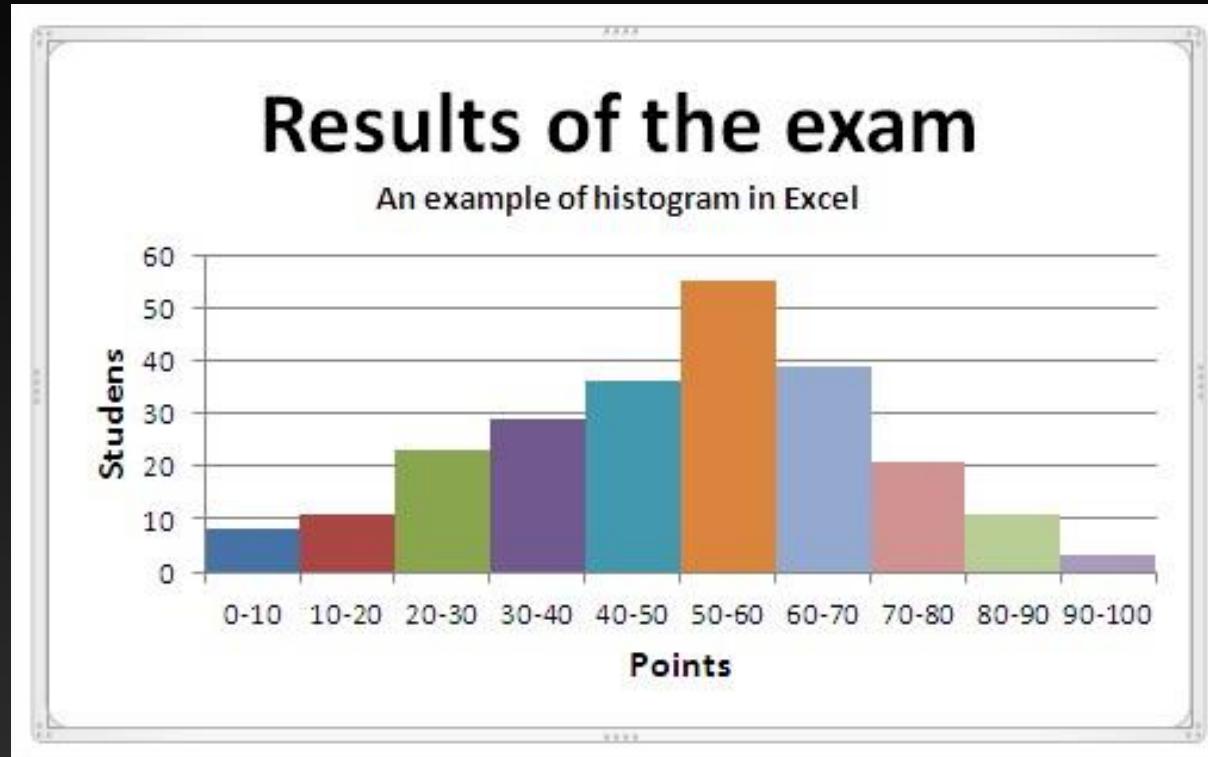


Exercice: Write an Image Filter Kernel (convolution kernel)

	0.000	0.000	0.001	0.001	0.001	0.000	0.000
	0.000	0.002	0.012	0.020	0.012	0.002	0.000
	0.001	0.012	0.068	0.109	0.068	0.012	0.001
	0.001	0.020	0.109	0.172	0.109	0.020	0.001
	0.001	0.012	0.068	0.109	0.068	0.012	0.001
	0.000	0.002	0.012	0.020	0.012	0.002	0.000
	0.000	0.000	0.001	0.001	0.001	0.000	0.000

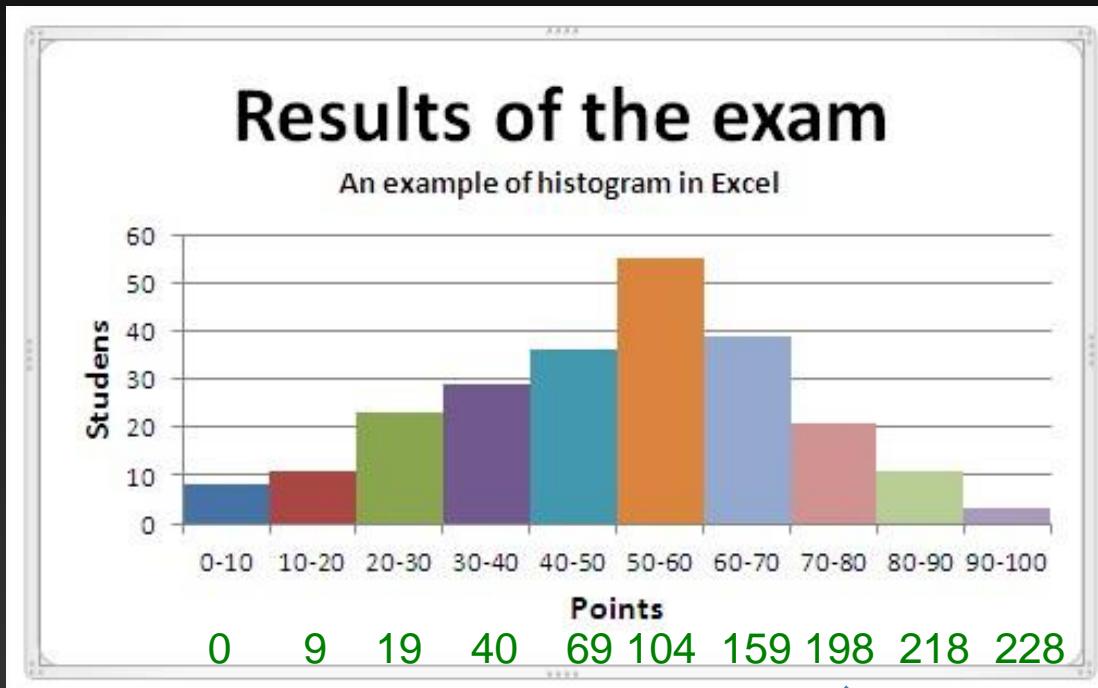


Histogram



Cumulative Distribution Function

If I received grade 70, how many students were worst than me? Exclusive Scan...



Histogram – serial implementation

```
...  
For (i=0, i< range_Size, i++)  
    Histogram [i]=0;  
  
For (i=0; i< data.size; i++)  
    Histogram [detectRange (data[i])]++;
```



Histogram – GPU naïve (wrong) implementation

```
__global__ void naïveHistogram (*int Histogram, const int * data, int)
{
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    int value = data[id];
    int Histvalue = DetectRange(value);
    Histogram[Histvalue]++;
}
```



Histogram – GPU naïve (wrong) implementation

```
__global__ void naïveHistogram (*int Histogram, const int * data, int)
{
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    int value = data[id];
    int Histvalue = DetectRange(value);
    Histogram[Histvalue]++;
}
```

What is wrong???



Race condition...

Thread is reading a value from GM to register
Register is operating the increment
Register writes the value back to the GM



Atomic Histogram

```
__global__ void atomicHistogram (*int Histogram, const int * data, int)
{
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    int value = data[id];
    int Histovalue = DetectRange(value);
    atomicAdd (&Histogram[Histvalue],1);
}
```

Efficiency Analysis (size of HISTOGRAMRANGE)



Device Properties

`cudaGetDevice()`

`cudaGetDeviceProperties ()`



Device Properties

```
cudaDeviceProp devProp;  
cudaGetDeviceProperties (&devProp, 0);
```

```
struct cudaDeviceProp {  
    char name[256];  
    size_t totalGlobalMem;  
    size_t sharedMemPerBlock;  
    int regsPerBlock;  
    int warpSize;  
    size_t memPitch;  
    int maxThreadsPerBlock;  
    int maxThreadsDim[3];  
    int maxGridSize[3];  
    size_t totalConstMem;  
    int major;  
    int minor;  
    int clockRate;  
    size_t textureAlignment;  
    int deviceOverlap;  
    int multiProcessorCount;  
    int kernelExecTimeoutEnabled;  
    int integrated;  
    int canMapHostMemory;  
    int computeMode;  
    int concurrentKernels;  
    int ECCEnabled;  
    int pciBusID;  
    int pciDeviceID;  
    int tccDriver;  
}
```



Some Basic Algorithms...



REDUCE

- 1) Set of Elements
- 2) Reduce Operation

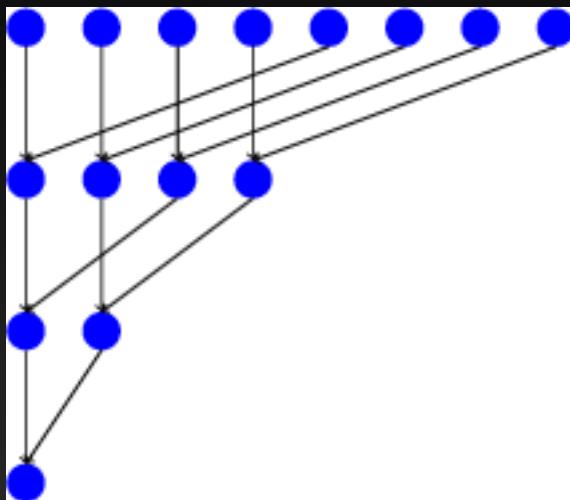
Example: Reduce ($\{1, 3, 5, 7\}$, $+$)



REDUCE

- Write a sequential algorithm
- Analyse works and steps complexity

Parallel REDUCE



- Analyse works and steps complexity

Work: Brent's Theorem

Implement a Parallel Reduce (Global Memory version)

Parallel Reduce (Global Memory version)

```
__global__ void reduceGlobal (float *s_data)
{
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    for (int stride = n/2; stride > 0; stride >>=1) {
        if (index < stride) {
            s_data [index] += s_data[index+stride];
        }
        __syncthread();
    }
}
```



Implement a Parallel Reduce (Shared Memory version)

```
__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[tid] = d_In[index]
    __syncthread();

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



Implement a Parallel Reduce (Shared Memory version)

```
__global__ void reduceShared (float *d_In, *d_Out)

reduceShared <<<blocks, threads, threads*sizeof(float)>>>
    (table, d_In);
```

```
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];
}
}
```



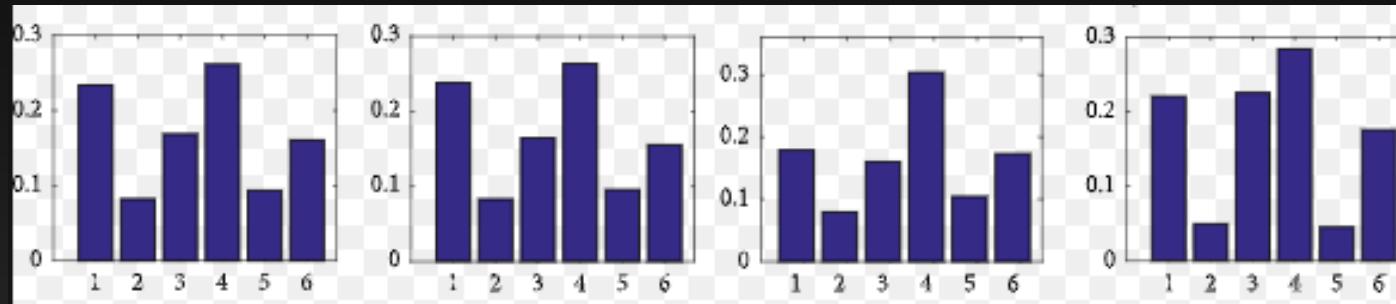
More efficient Parallel Solution for Histograms

Each thread computes a whole and complete histogram,
from a part of the data

After the first execution, all the sub-histograms must be joined. This can be done with Reduction strategy, in order to be also parallelized

EXERCISE: implement this version...

More efficient Parallel Solution for Histograms



SCAN

2	4	3	1	5	10	2
0	2	6	9	10	15	25

- Exclusive

- Example: bank account (first table: in/ou, second table: total balance (saldo))

SCAN

2	4	3	1	5	10	2
2	6	9	10	15	25	27

- Inclusive

SCAN

- Input Array, Operator (same properties as reduce), Identity operator
- Identity operator for +, max, *, min, or, and



SCAN

- Input Array, Operator f (same properties as reduce), Identity operator
- Identity operator for +, max, *, min, or, and
- Output: [Identity, $f A_0$, $f(A_0, A_1)$, $f(A_0, A_1, A_2)$, ...]



SCAN

- Serial Implementation
- Analyse steps and work complexity



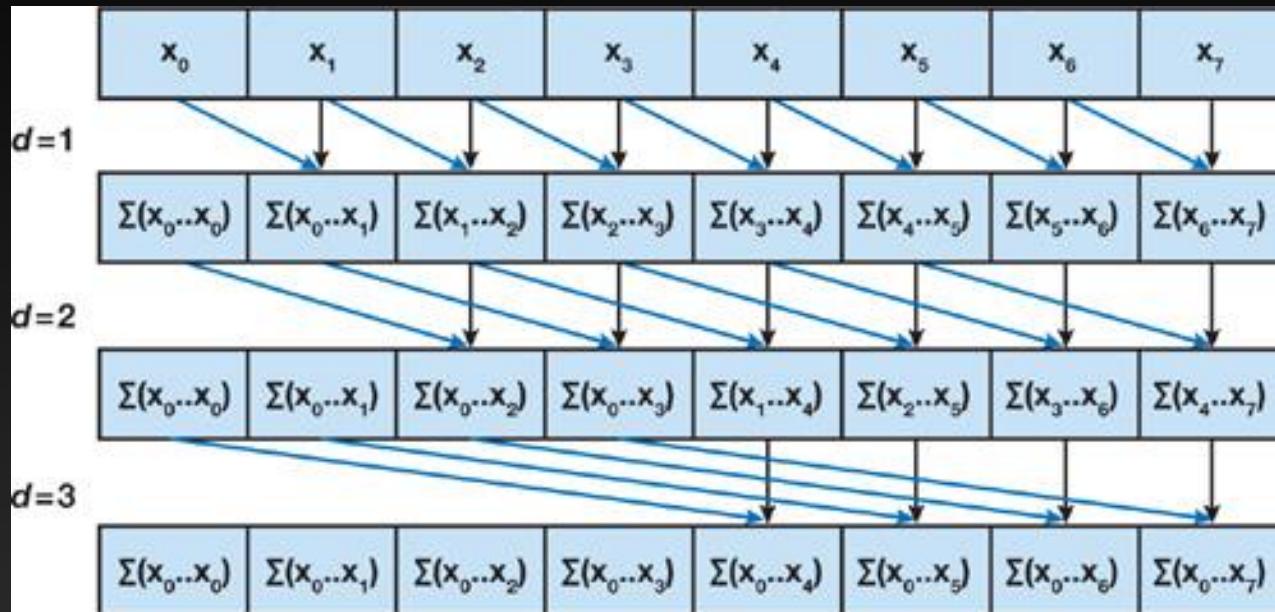
SCAN

- Hillis/Steele (Better step efficiency)
- Blelloch (better work efficiency)



Hillis/Steele SCAN

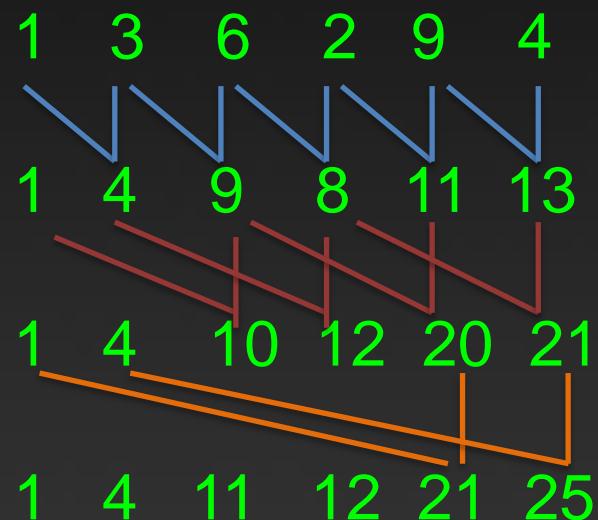
http://developer.nvidia.com/GPUGems3/gpugems3_ch39.htm



Hillis/Steele SCAN

Step = 0

Add yourself to your 2^{step} neighborhood



step = 0

step = 1

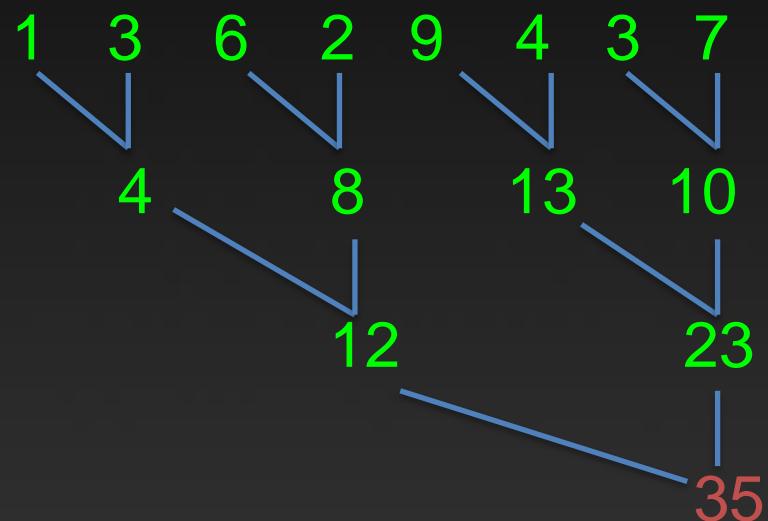
step = 2

final scan result

Blelloch Scan

Reduce + Downseep

REDUCE



step = 0

step = 1

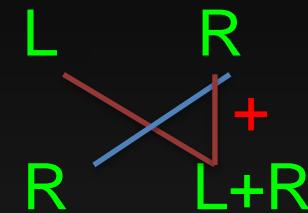
step = 2

Blelloch Scan

Downsweep

1	3	6	2	9	4	3	7
	4		8		13		10
			12			23	
1	4	6	12	9	13	3	35
						0	

operator

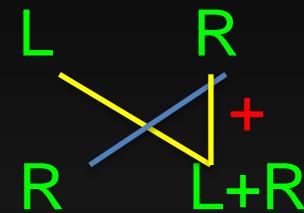


Start with the identity



Blelloch Scan

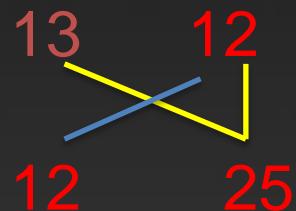
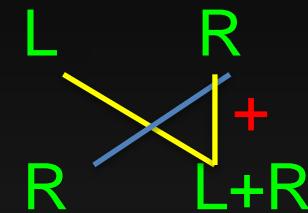
Downsweep



Blelloch Scan

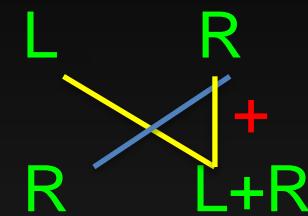
Downsweep

1	3	6	2	9	4	3	7
4		8		13	10		
		12			23		
					35		
					0		
			12			0	



Blelloch Scan

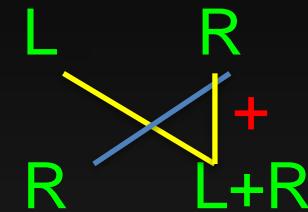
Downsweep



Blelloch Scan

Downsweep

1	3	6	2	9	4	3	7
4			8		13		10
				12		23	
					35		
					0		
			12			0	



4 0 13 12

1	0	6	4	9	12	3	25
0	1	4	10	12	21	25	28



Using and Optimizing Scan



Compact (selecting a sub-group of a group - Filter)

Input: $x_0 \quad x_1 \quad x_2 \quad \dots \quad x_n$
Predicate: $p(x_0) \ p(x_1) \ p(x_2) \ \dots \ p(x_n)$ // ex. P – é par?
Resultado: T F F T

Compact: $x_0 \quad \dots \quad \dots \quad x_n$ (saída esparsa)
Compact: $x_0 \ x_n$ (saída densa)

Compact (selecting a sub-group of a group - Filter)

Computing or calling threads only for the dense values is much more effective than sparse. In the sparse call many threads will be idle.

What is better: Apply a Filter into all elements (n filter) or compact then apply filter only at selected elements m (n compact + m filter) ?



Segmented Scan

1 2 3 4 5 6 7 8



Segmented Scan

1 2 3 4 5 6 7 8



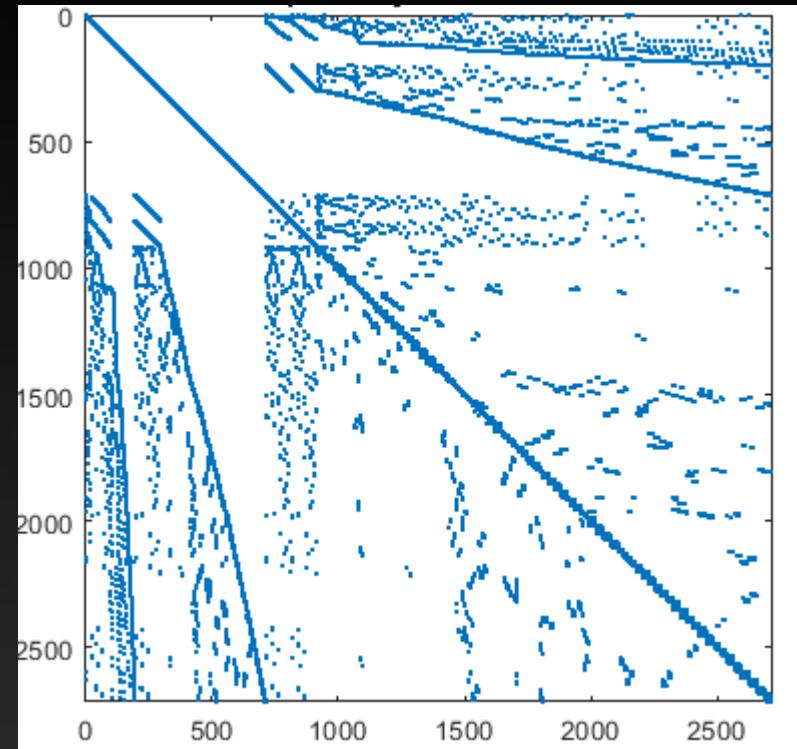
Segmented Scan

1 2 3 4 5 6 7 8

1 3 3 7 12 6 13 21



Sparse Matrix



$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

Sparse Matrix Representation

A	0	B
C	D	0
0	0	E

Value: A B C D E
Column: 0 2 0 1 2
Row pointer: 0 2 4

(Row Pointer = occurrence position in Value of
the element that starts a new line)



Calculating a Sparse Matrix by a vector

A	0	B		x
C	D	0	x	y
0	0	E		z



Calculating a Sparse Matrix by a vector

- 1) Create a segmented representation from values and Rowpointers:

A	0	B
C	D	0
0	0	E

Value: A B C D E
Column: 0 2 0 1 2
Row pointer: 0 2 4

A B C D E



Calculating a Sparse Matrix by a vector

2) Gather Vector values using Column

Value:	A	B	C	D	E
Column:	0	2	0	1	2

Value: A B C D E → Value: A B C D E

	x	z	x	y	z
	x	z	x	y	z

Calculating a Sparse Matrix by a vector

- 3) Multiply each pair of the vectors

$$\begin{array}{cc|cc|c} A & B & C & D & E \\ x & z & x & y & z \\ \hline A.x & B.z & C.x & D.y & E.z \end{array}$$

Calculating a Sparse Matrix by a vector

4) Exclusive Segmented Sum scan

$$\begin{array}{cc|cc|c} A & B & C & D & E \\ x & z & x & y & z \\ A.x & B.z & C.x & D.y & E.z \end{array}$$
$$A.x + B.z \quad C.x + D.y \quad E.z$$

Sparse Matrix in GPUs

 NVIDIA. HIGH PERFORMANCE COMPUTING

DOWNLOADS

TRAINING

ECOSYSTEM

FORUMS



ACCOUNT

cuSPARSE

[Home](#) > [High Performance Computing](#) > [Tools & Ecosystem](#) > [GPU Accelerated Libraries](#) > cuSPARSE

Basic Linear Algebra for Sparse Matrices on NVIDIA GPUs

[DOWNLOAD >](#)

[DOCUMENTATION >](#)

[SAMPLES >](#)

[SUPPORT >](#)

The cuSPARSE library provides GPU-accelerated basic linear algebra subroutines for sparse matrices that perform significantly faster than CPU-only alternatives. It provides functionality that can be used to build GPU accelerated solvers. cuSPARSE is widely used by engineers and scientists working on applications such as machine learning, computational fluid dynamics, seismic exploration and computational sciences. Using cuSPARSE, applications automatically benefit from regular performance improvements and new GPU architectures. The cuSPARSE library is included in both the [NVIDIA HPC SDK](#) and the [CUDA Toolkit](#).

[Explore what's new in the latest release...](#)



Race condition...



Atomic Histogram

```
__global__ void atomicHistogram (*int Histogram, const int * data, const int HISTOGRAMRANGE)
{
    int id = threadIdx.x + blockDim.x*blockIdx.x;
    int value = data[id];
    int Histvalue = value % HISTOGRAMRANGE;
    atomicAdd (&Histogram[Histvalue],1);
}
```

Efficiency Analysis (size of HISTOGRAMRANGE)



More efficient Parallel Solution

Each thread computes a whole and complete histogram, from a part of the data

Since each thread computes a complete histogram, it is no necessary atomic operation

After the first execution, all the sub-histograms must be joined. This can be done with Reduction strategy, in order to be also parallelised

EXERCISE: implement this version...



Exercise: Histogram Equalization



Streams

Task Parallelism: two or more completely different tasks in parallel



Streams

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

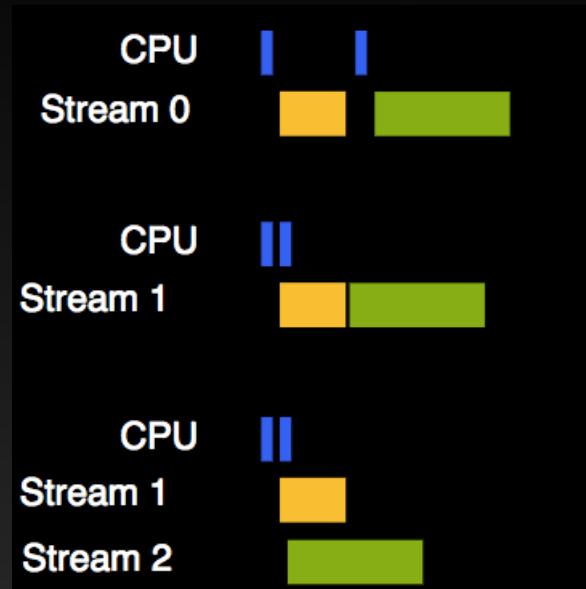


Streams

Synchronous

Asynchronous same stream

Asynchronous different streams



Streams

Before starting, we must check if the GPU supports streams... Older versions don't. Newer versions support 2 overlap in the memcpy engine.

```
...

cudaDeviceProp properties;
Int deviceHandler;

cudaGetDevice (&deviceHandler);
cudaGetDeviceProperties (&properties, deviceHandler);

If (!properties.deviceOverlap) {
    printf ("Sorry, no streams supported in this device... Buy a new one... ");
    return 0;
}

...
```



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));  
  
cudaMemcpyAsync (dev_a, a, size * sizeof(*dev_a) , cudaMemcpyHostToDevice));  
  
cudaFreeHost ( a );  
cudaFree       (dev_a);  
  
...
```



Streams

```
H_mem = malloc (size);  
cudaHostRegister (h_mem, size, 0);  
...  
cudaHostUnregister (h_mem);  
Free(h_mem);
```

Similar to
cudaMallocHost (&h_mem, size);
...
cudaFreeHost (h_mem);



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 synchronous . 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 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);
```

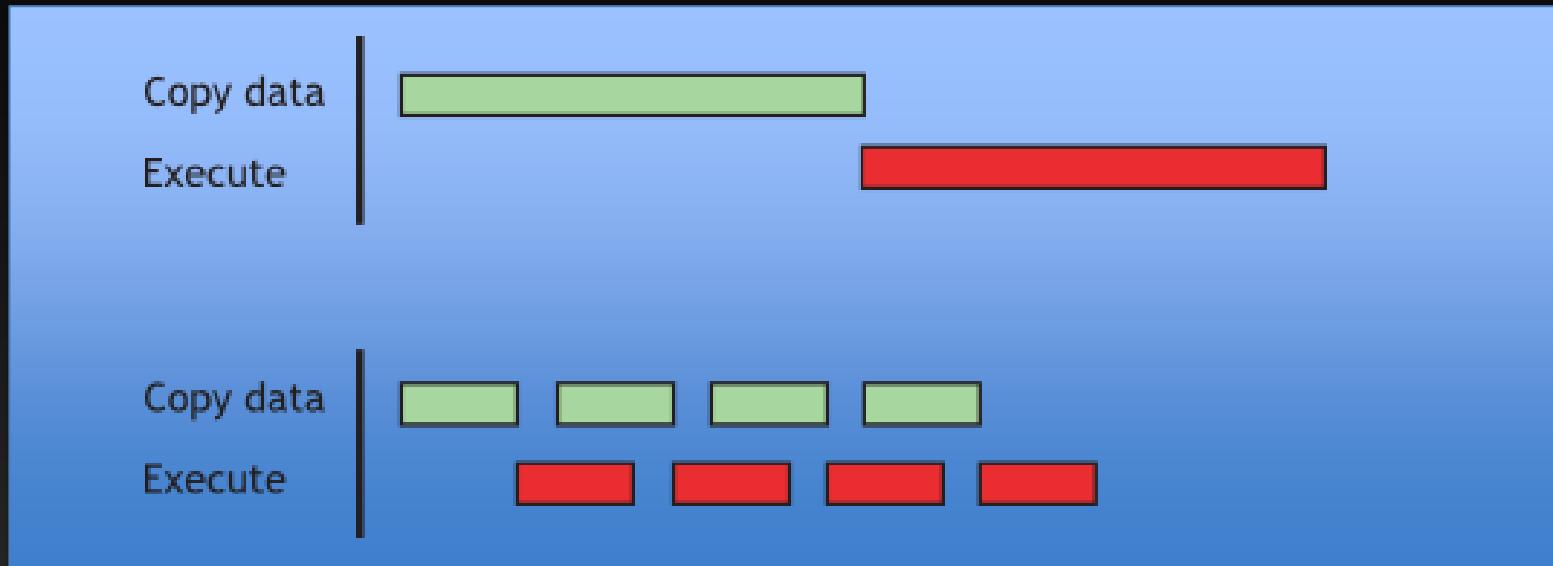


Streams

```
...  
  
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=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, stream1>>> (d_a1, d_b1, d_c1);

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

}
```



Stream overlaps

```
cudaStreamSynchronize (stream0);  
cudaStreamSynchronize (stream1);  
  
// frees and destroys...
```



Stream overlaps

```
cudaStreamSynchronize (stream0);  
cudaStreamSynchronize (stream1);  
  
// frees and destroys...
```

Possivel melhorias



Stream overlaps

Stream0 is the default and is always synchronized



Stream overlaps

```
For (int i=0; i < TOTAL_SIZE ; i+=N*2)
{
    cudaMemcpyAsync (dev_a0, h_a+i, N* sizeof(int), cudaMemcpyHostToDevice, stream1));
    cudaMemcpyAsync (dev_a1, h_a+i+N, N* sizeof(int), cudaMemcpyHostToDevice, stream2));

    cudaMemcpyAsync (dev_b0, h_b+i, N* sizeof(int), cudaMemcpyHostToDevice, stream1));
    cudaMemcpyAsync (dev_b1, h_b+i+N, N* sizeof(int), cudaMemcpyHostToDevice, stream2));

    kernel<<<N/256, 256, 0, stream1>>> (d_a0, d_b0, d_c0);
    kernel<<<N/256, 256, 0, stream2>>> (d_a1, d_b1, d_c1);

    cudaMemcpyAsync (h_c+i, dc_0, N* sizeof(int), cudaMemcpyDeviceToHost, stream1));
    cudaMemcpyAsync (h_c+i+N, dc_1, N* sizeof(int), cudaMemcpyDeviceToHost, stream2));
}
```



Stream overlaps

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset],
                   streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&a[offset], &d_a[offset],
                   streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost, stream[i]);
}
```



Concurrent Kernels

```
kernel1 <<<N/256, 256, 0, stream0>>> (d_a0, d_b0, d_c0);  
kernel2 <<<N/256, 256, 0, stream1>>> (d_a1, d_b1, d_c1);
```



Why using Stream:

Do at the same time memory transfer and Kernel execution

Fill the GPU with smaller pieces of data



Exercise:

Implement a simple vectorAdd for a huge vector with streams



Constant Memory

Special part of Memory, with the following properties:

- 1) Very small (for instance, 64KB);
- 2) A read operation in the constant memory can broadcast data to all half warps that are running;
- 3) Constant Memory is cached, so that consecutive reads at the same address will be optimized



Constant Memory

Special part of Memory, with the following properties:

- 1) Very small (64KB);
- 2) A read operation in the constant memory can broadcast data to all half warps that are running;
(if data is not the same, this operation will be serialized and can be 16 times heavier)
- 3) Constant Memory is cached, so that consecutive reads at the same address will be optimized

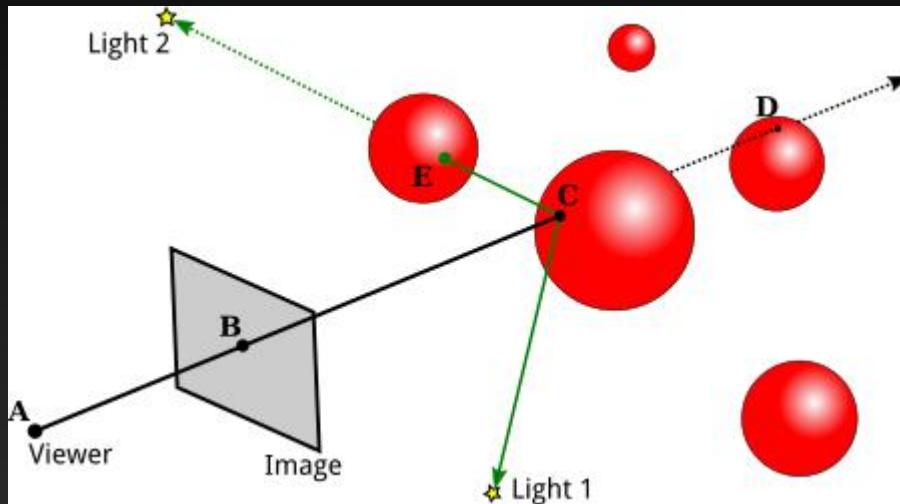


Constant Memory

```
...  
__constant__ Vector3 v[TOTAL_SIZE];  
  
...  
  
cudaMemcpyToSymbol (v, h_data, sizeof (vector3) * TOTAL_SIZE);  
  
// We don't have to specify the copy direction. Why??? (device→host,  
host→device)  
...
```



Exercice: implement a reduce using constant memory for vector



Constant Memory Example

Ray tracing in GPU

```
#define INF  2e10f; // Infinit

Struct Sphere {
    float r, g, b;
    float radius;
    float x, y, z;

__device__ float hit (float ox, float oy, float *n) {
    float dx = ox - x;
    float dy = oy - y;
    if (dx*dx + dy*dy < radius*radius){
        float dz = sqrtf (radius*radius - dx*dx - dy*dy);
        *n = dz / sqrtf (radius*radius);
        return dz+z;
    }
    return -INF;
}
```

Constant Memory Example

Ray tracing in GPU

```
#include "cuda.h"
#include "../common/cpu_bitmap.h"

#define rnd (x) (x*rand()/RAND_MAX)
#define SPHERE 20

__constant__ Sphere s[SPHERE];

Int main (void){
    CPUBitmap bitmap (DIM, DIM);
    unsigned char *dev_bitmap;

    cudaMalloc ((void**) &dev_bitmap, bitmap.image_size() ) ) ;

    // There is no CudaMalloc for the constant Memory
```



Constant Memory Example

Ray tracing in GPU

```
Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)*SPHERES);

for (int i=0; i<SPHERES; i++) {
    temp_s[i].r = rnd(1.0f);
    temp_s[i].g = rnd(1.0f);
    temp_s[i].b = rnd(1.0f);
    temp_s[i].x = rnd(1000.0f)-500;
    temp_s[i].y = rnd(1000.0f)-500;
    temp_s[i].z = rnd(1000.0f)-500;
    temp_s[i].radius = rnd(100.0f)+20;
}

CudaMemcpyToSymbol (s, temp_s, sizeof(Sphere)*SPHERES));
free (temp_s);
```



Constant Memory Example

Ray tracing in GPU

```
__global__ void kernel (unsigned char *ptr){  
  
    int x = threadIdx.x + blockIdx.x*blockDim.x;  
    int y = threadIdx.y + blockIdx.y*blockDim.y;  
    int offset = x + y *blockDim.x*gridDim.x;  
  
    float ox = (x - DIM/2);      // Mudanca de escala (centro da imagem no meio  
    float oy = (y - DIM/2);      // da tela  
  
    float r=0, g=0, b=0;  
    float maxz = -INF;
```



Constant Memory Example

Ray tracing in GPU

```
For (int i=0; i<SPHERES; i++) {
    float n, t = s[i].hit(ox, oy, &n);
    if (t > maxz) {
        float fscale = n;
        r = s[i].r * fscale;
        g = s[i].g * fscale;
        b = s[i].b * fscale;
        maxz = t;
    }
}

ptr[offset*4+0] = (int) (r * 255);
ptr[offset*4+1] = (int) (g * 255);
ptr[offset*4+2] = (int) (b * 255);
ptr[offset*4+3] = 255;
```



Constant Memory Example

Ray tracing in GPU

```
dim3      grids(DIM/16, DIM/16) ;
dim3      threads (16,16) ;
Kernel<<<grid, threads>>> (dev_bitmap) ;

cudaMemcpy (bitmap.get_ptr() , dev_bitmap , bitmap.image_size() ,
            cudaMemcpyDeviceToHost) ;

Bitmap.display_and_exit() ;
cudaFree(dev_bitmap) ;
}
```



Trabalho

NCCL – NVIDIA Collective Communication Library

cuRAND

CuSPARSE

CuDNN



Trabalho

cuRAND



Trabalho

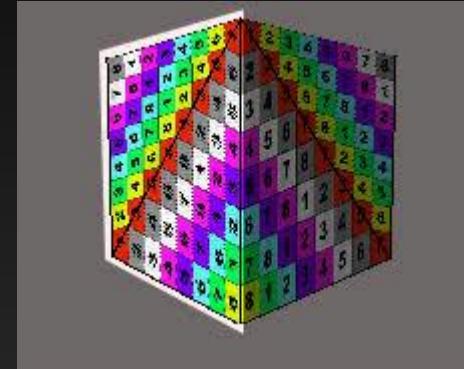
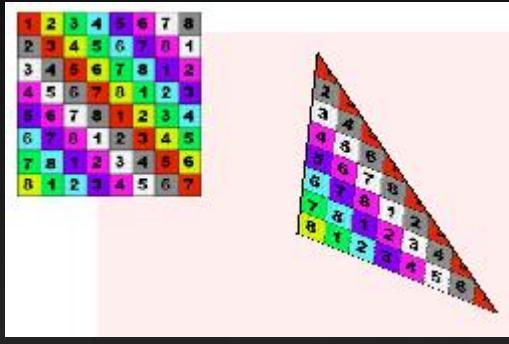
StarPU

PyCUDA



Texture memory

What is a texture in CG?



Texture memory

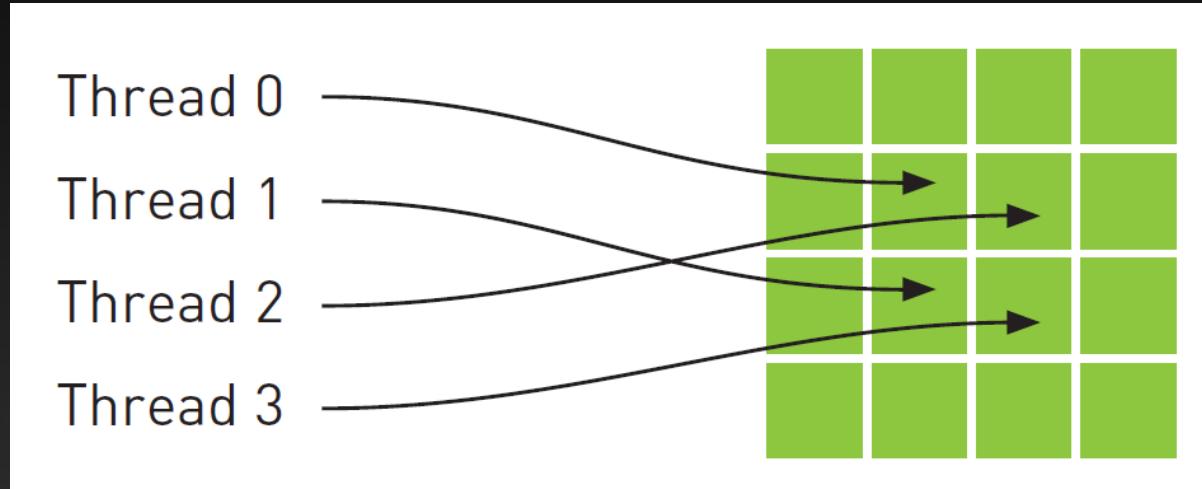
Special part of Memory, with the following properties:

- 1) Read-only memory
- 2) Originally designed for Graphic pipeline
- 3) Designed for 2D space locality



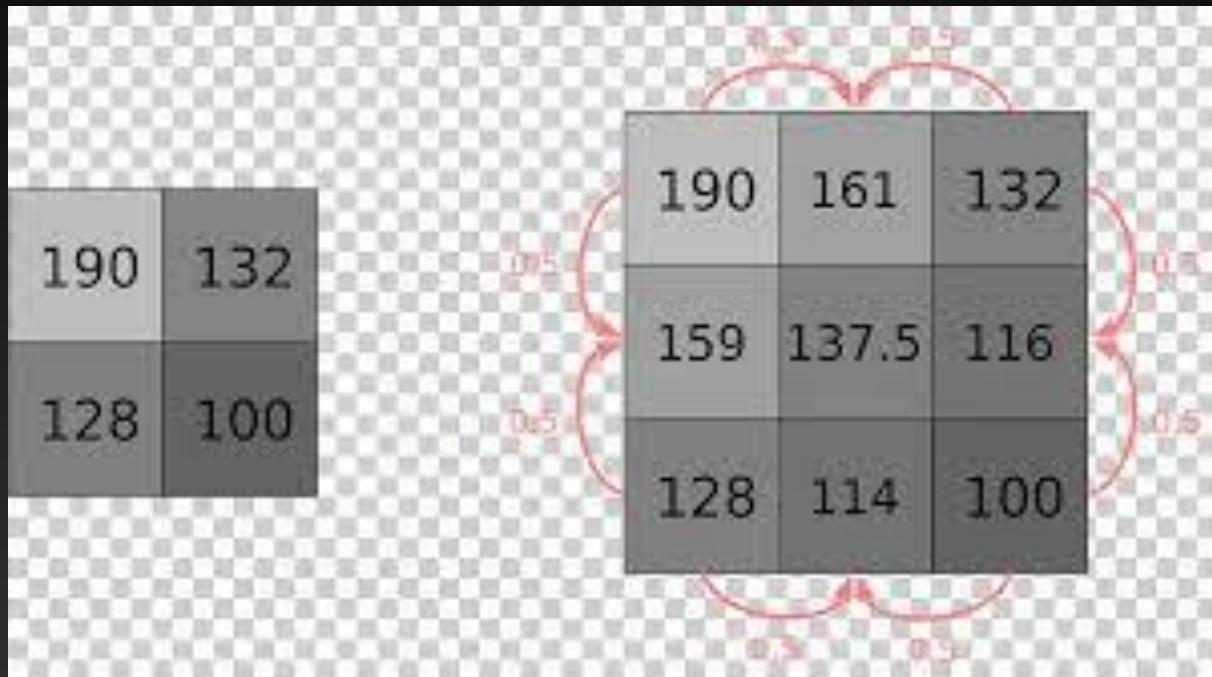
Texture memory

What is a texture in CG?



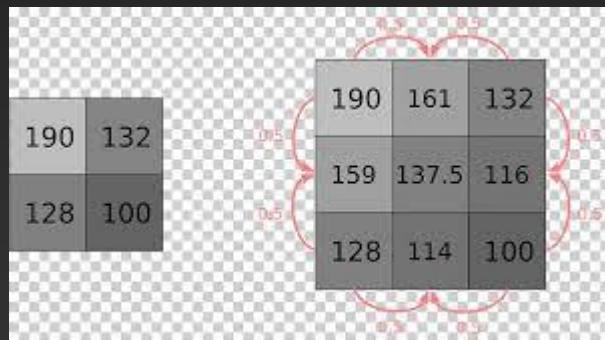
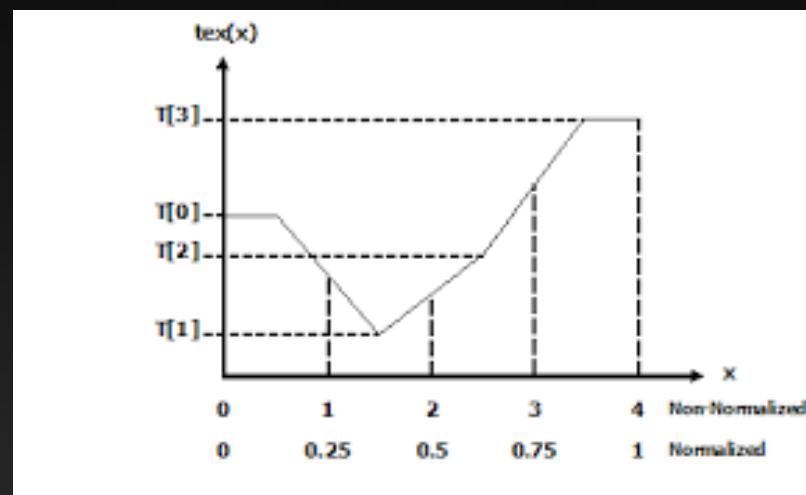
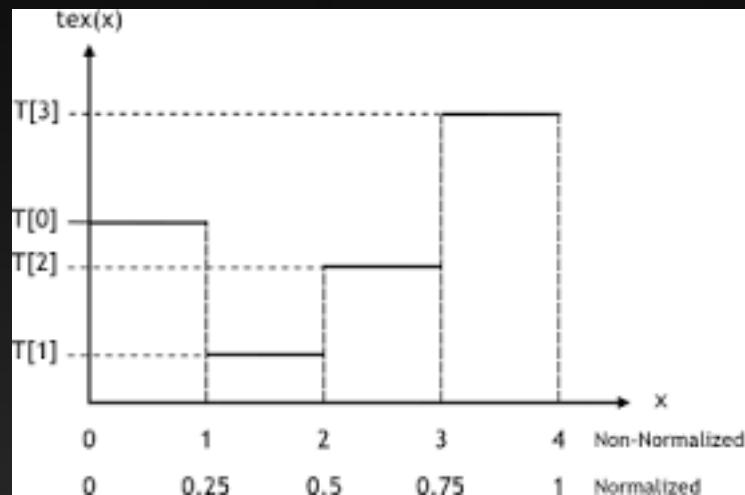
Texture memory

Free interpolation



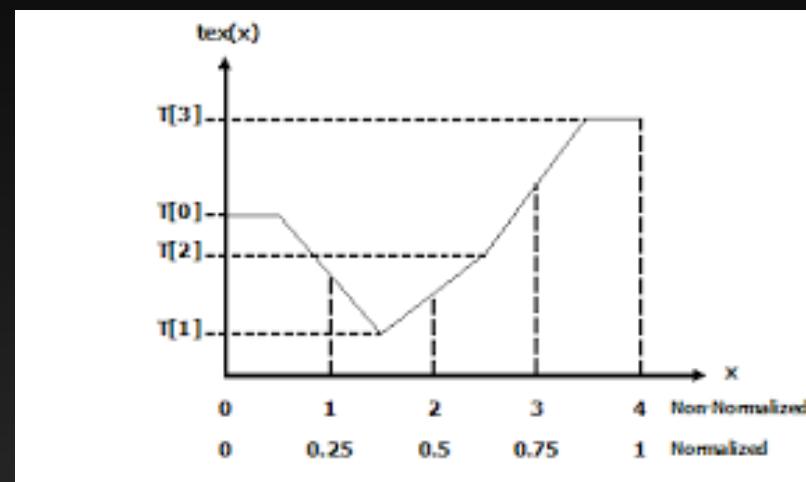
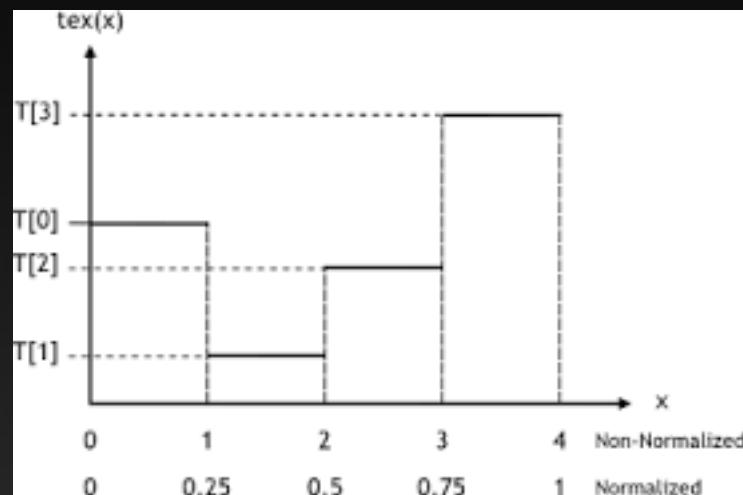
Texture memory

Free interpolation



Texture memory

Boundaries errors treatment



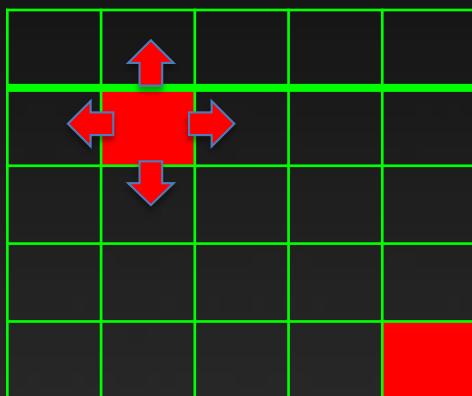
Texture memory Example

Energy propagation



Texture memory Example

Energy propagation



$$C^t = C^{t-1} + \text{Sum}^{\text{neighbors}}(k \cdot C^{n,t-1} - C^{t-1})$$

$$C^t = C^{t-1} + k.(C^{\text{top},t-1} + C^{\text{bottom},t-1} + C^{\text{right},t-1} + C^{\text{left},t-1} - 4 \cdot C^{t-1})$$

Energy propagation interation

1. Update Grid (copy constant emmiters at the output, that becomes input)
2. Propagate energy
3. Swap input and output for the next stage



1. Update entrance

```
__global__ void UpdateEntrance (float* input, const float * constantData){  
    int x = threadIdx.x + blockIdx.x * blockDim.x;  
    int y = threadIdx.y + blockIdx.y * blockDim.y;  
    int offset = x + y * blockDim.x*gridDim.x;  
  
    if (constantData[offset] != 0) input[offset] = constantData[offset];  
}
```



2. Energy propagation

```
__global__ void updateEnergy(float *output, const float *input) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x*gridDim.x;

    int left = offset - 1;
    int right = offset +1;
    if (x == 0) left++;                  // Border condition
    if (y == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;

    output[offset] = input[offset] + K * (input[top]+input[bottom]+
                                         input[right]+input[left] - 4*input[offset]);
}
```



Complete pipeline

...

```
For (int i=0; i<90; i++) {  
    UpdateEntrance <<<blocks, threads>>> (input, constantData);  
    updateEnergy <<<blocks, threads>>> (output, input);  
    swap (input, output);  
}  
...
```



Using Texture memory

Declaration of texture memory (references)

```
Texture<float> texConst;  
texture<float> texInput;  
texture<float> texOutput;  
  
cudaBindTexture (NULL, texConst, constantData, GridSize);  
cudaBindTexture (NULL, texInput, Input, GridSize);  
cudaBindTexture (NULL, texOutput, Output, GridSize);
```



Using Texture memory

Kernel with texture memory

```
__global__ void updateEnergy(float *output, const float *input) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x*gridDim.x;

    int left = offset - 1;
    int right = offset +1;
    if (x == 0) left++;           // Border condition
    if (y == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;
```



Using Texture memory

tex1Dfetch → Compiler Intrinsic (not function)

Texture reference must be declared globally at file scope and not as parameters.

```
Float t, l, c, r, b;

If (dstOut) {
    t= tex1Dfetch(texIn, top);
    l= tex1Dfetch(texIn, left);
    c= tex1Dfetch(texIn, offset);
    r= tex1Dfetch(texIn, right);
    b= tex1Dfetch(texIn, bottom);
}

Else{
    t= tex1Dfetch(texOut, top);
    l= tex1Dfetch(texOut, left);
    c= tex1Dfetch(texOut, offset);
    r= tex1Dfetch(texOut, right);
    b= tex1Dfetch(texOut, bottom);
}

Dst[offset]= c + K * (t + b + r + l - 4*c);
```



Update entrance

```
__global__ UpdateEntrance (float* iptr){  
    int x = threadIdx.x + blockIdx.x * blockDim.x;  
    int y = threadIdx.y + blockIdx.y * blockDim.y;  
    int offset = x + y * blockDim.x*gridDim.x;  
  
    float c = tex1Dfetch (texConstSrc, offset);  
    if (c !=0)  
        iptr[offset] = c;  
}
```



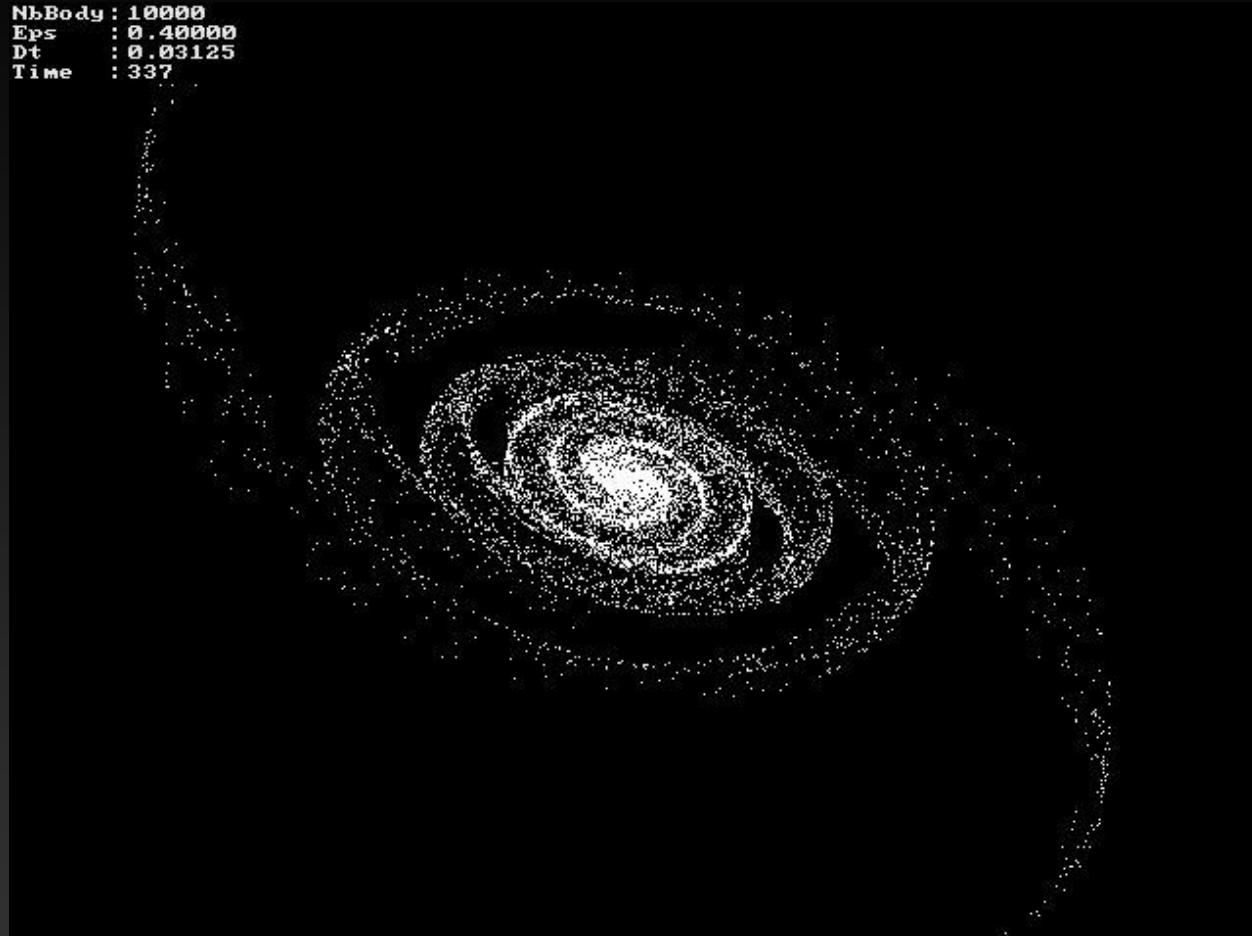
Complete pipeline

```
...  
Bool dstOut=true;  
  
For (int i=0; i<90; i++){ // 90 passos de tempo...  
    float *in, *out;  
    if (dstOut){  
        in = d→devinSrc;  
        out = d→dev_outSrc;  
    }  
    else {  
        out= d→devinSrc;  
        in= d→dev_outSrc;  
    }  
  
    UpdateEntrance <<<blocks, threads>>> (in);  
    updateEnergy <<<blocks, threads>>> (out,dstOut);  
    dstOut = !dstOut;  
}  
...
```



N-body

```
NbBody : 10000  
Eps   : 0.40000  
Dt    : 0.03125  
Time  : 337
```



N-body

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|},$$

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}.$$

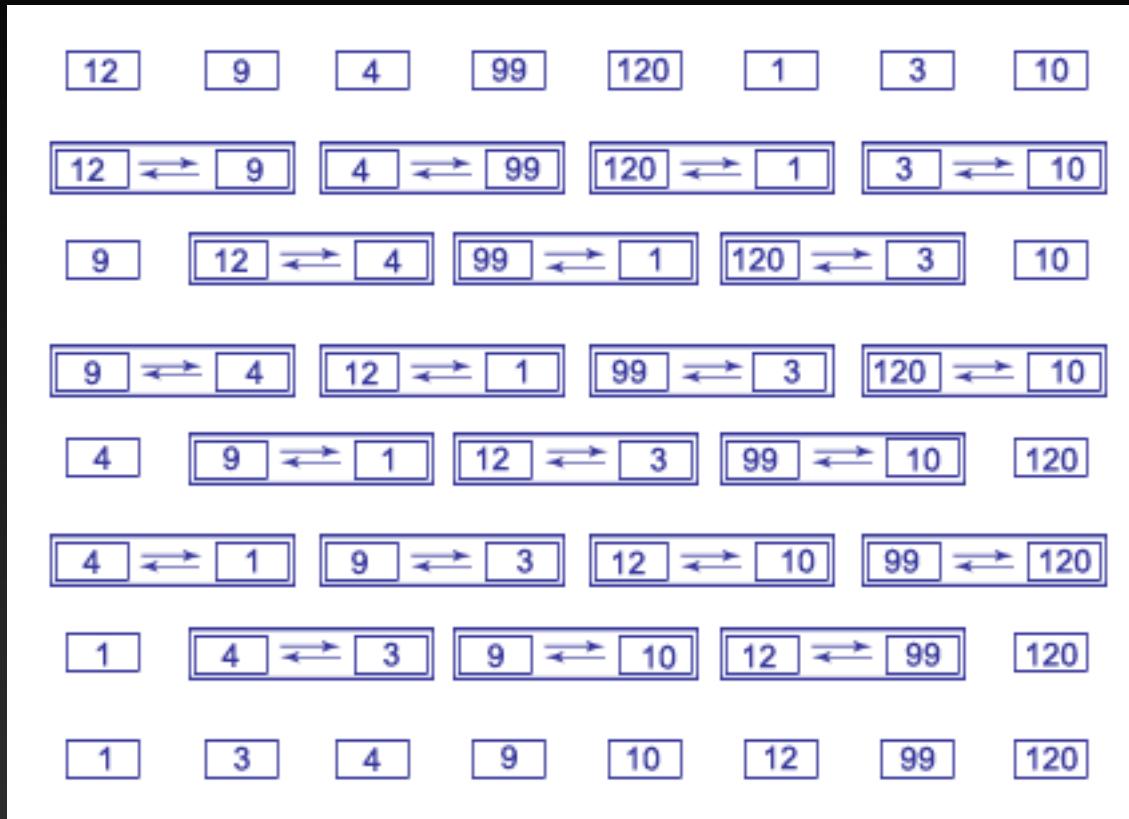
$$\mathbf{a}_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left(\|\mathbf{r}_{ij}\|^2 + \varepsilon^2\right)^{3/2}}.$$

SORT

Problems:

- Most algorithms are sequential
- “branchy” codes...
- Random Access (\neq Coalescence)

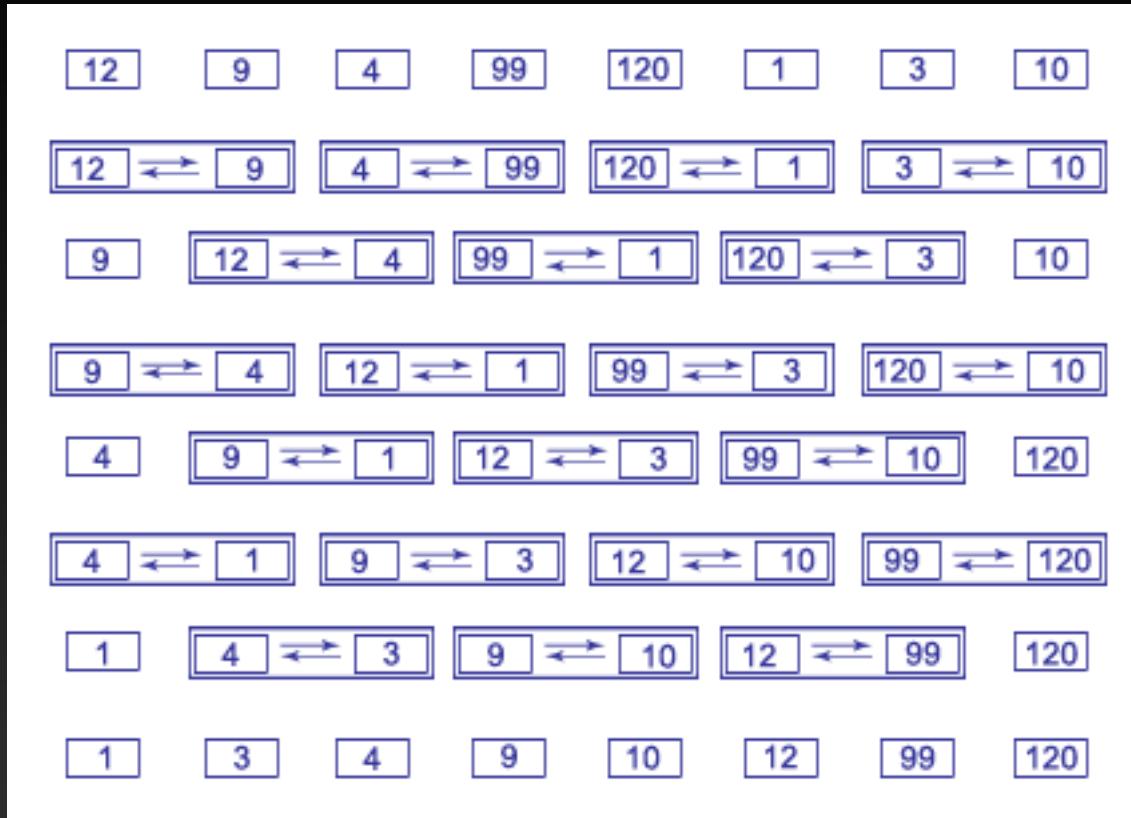
Brick SORT (also Odd-Even Sort)



Steps:

Work:

Brick SORT (also Odd-Even Sort)



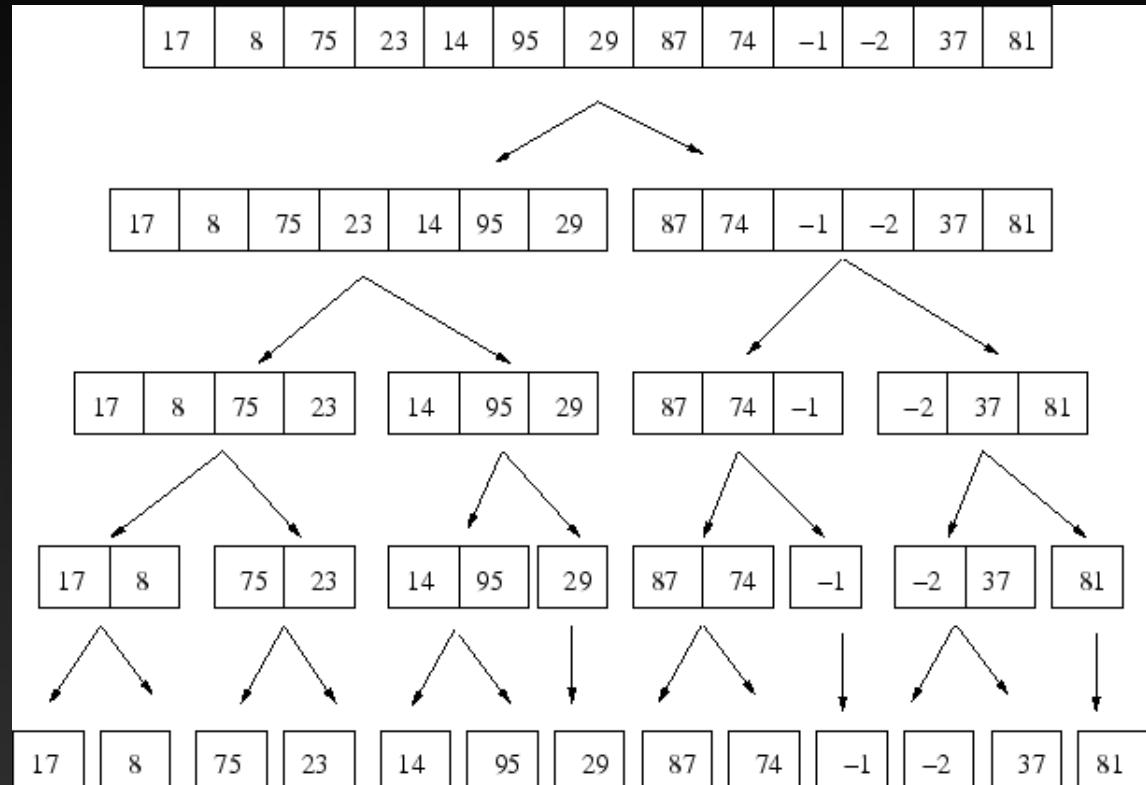
Steps: n

Work: n^2

Brick Sort

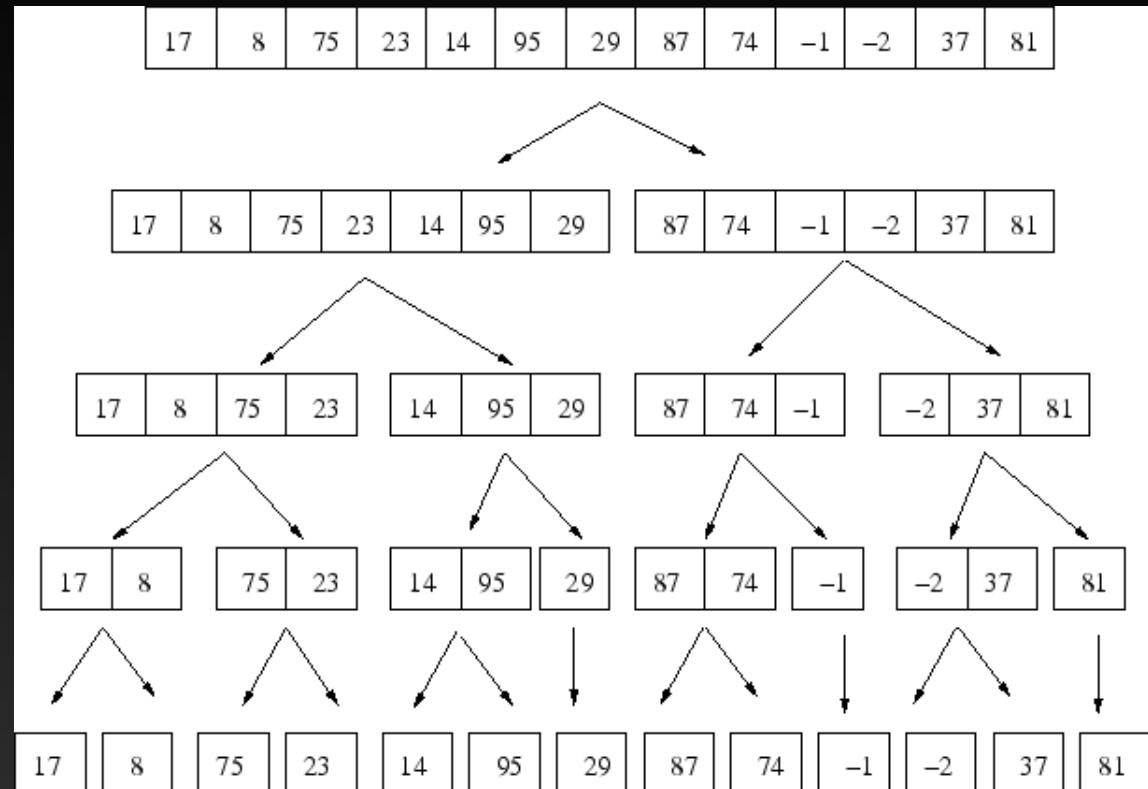
```
void OddEvenSort (T a[], int n)
{
    for (int i = 0 ; i < n ; i++)
    {
        if (i & 1) // 'i' is odd
        {
            for (int j = 2 ; j < n ; j += 2)
            {
                if (a[j] < a[j-1])
                    swap (a[j-1], a[j]) ;
            }
        }
        else
        {
            for (int j = 1 ; j < n ; j += 2)
            {
                if (a[j] < a[j-1])
                    swap (a[j-1], a[j]) ;
            }
        }
    }
}
```

Merge Sort



Order each little
Block starting with
The first, that has
Only 1 element

Merge Sort

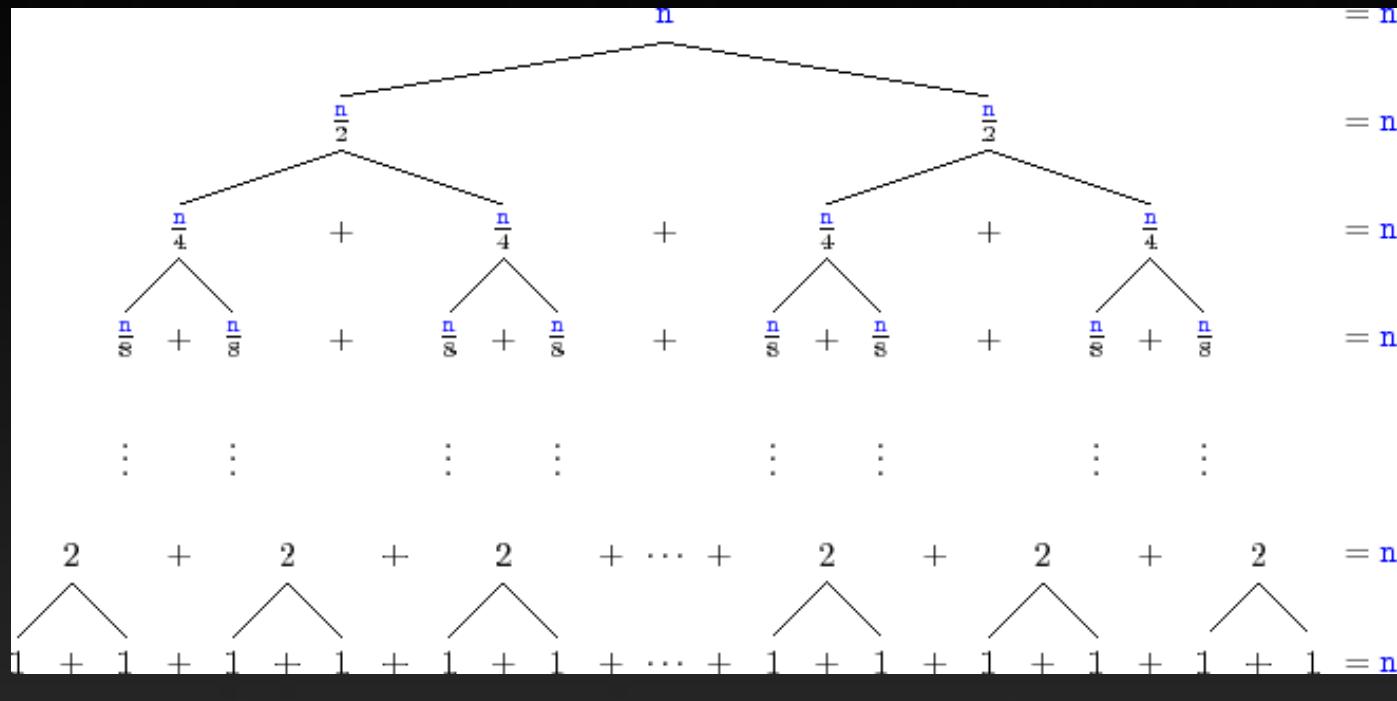


Log n steps

----- n itens -----

$O(n \log n)$

Merge Sort



End:

One single and big
merge...

Medium:
Lot's of small sorted
parts,

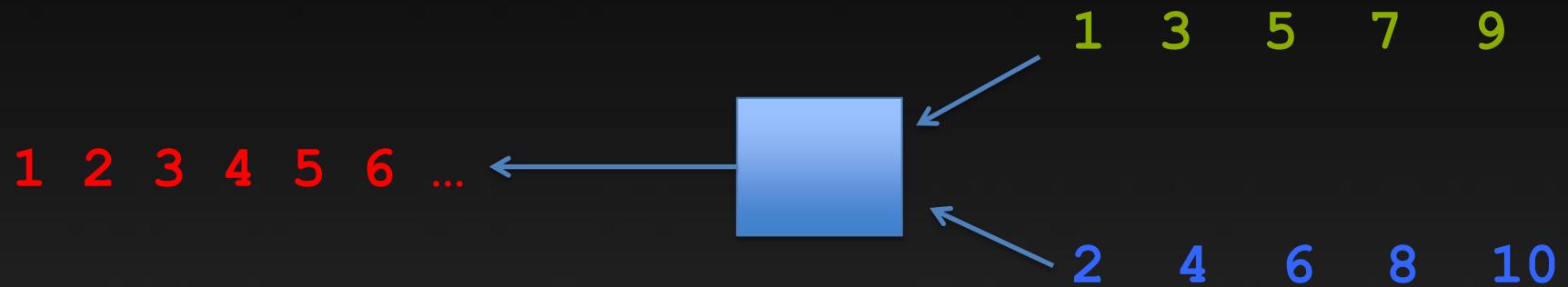
Start:

Lot's of work, with
Few elements (per
thread)

3 stages

- 1) Merge a large number of short sequences.
One merge task per thread. (for instance,
merge until small lists of 512 elements)
- 2) Merge Medium sized lists. Task per block
- 3) Merge a Big list. Task per Grid

Merging 2 sorted arrays (sequential)



Merging 2 sorted arrays (in parallel)

Input List 1: 1 3 5 7 9

Input List 2: 2 4 6 8 10

Each thread will find the position of its element
in the output list, which consist on a scatter
stage

Merging 2 sorted arrays (in parallel)

Input List 1: $\begin{matrix} 1 & 3 & 5 & 7 & 10 \\ 0 & 1 & 2 & 3 & 4 \end{matrix}$

0 1 2 3 5

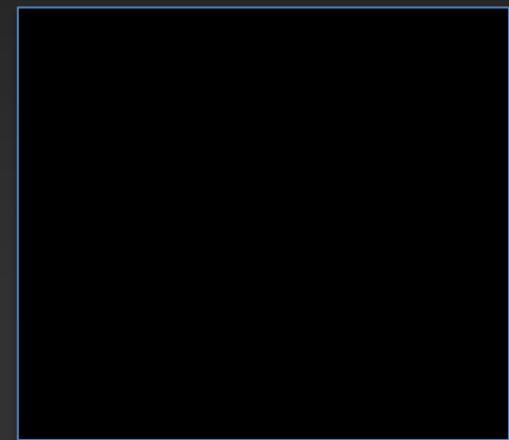
Input List 2: $\begin{matrix} 2 & 4 & 6 & 8 & 9 \\ 0 & 1 & 2 & 3 & 4 \end{matrix}$

0 1 2 3 4

1 2 3 4 4

1 2 3 4 5 6 7 8 9 10

0 1 2 3 4 5 6 7 8 9



Merging 2 sorted arrays (in parallel)



My position in list 1 is my thread ID

Find my position at list 2 with a binary search

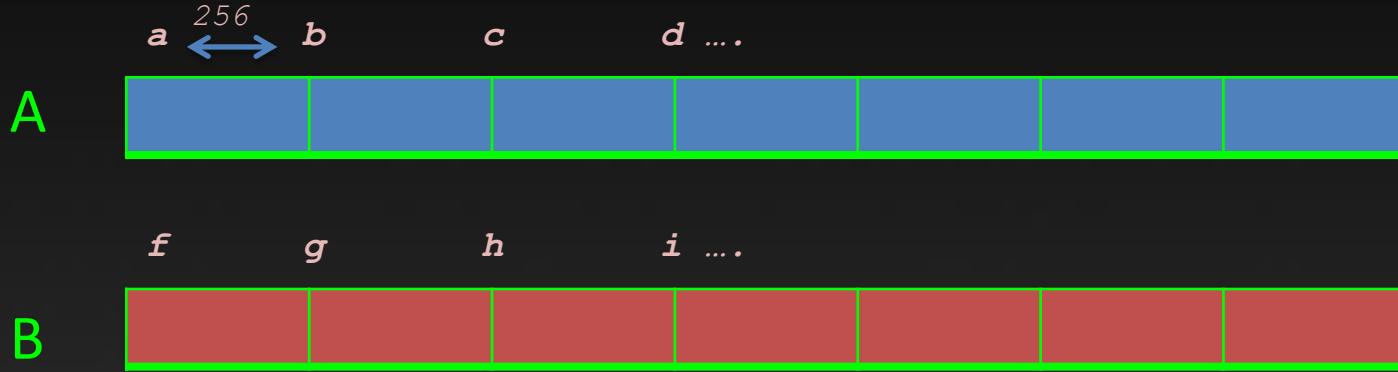
My final position will be the sum of both

Stage 3: Large Merge (in the last case, a single merge of two half complete lists)

Split a single merge activity into parts

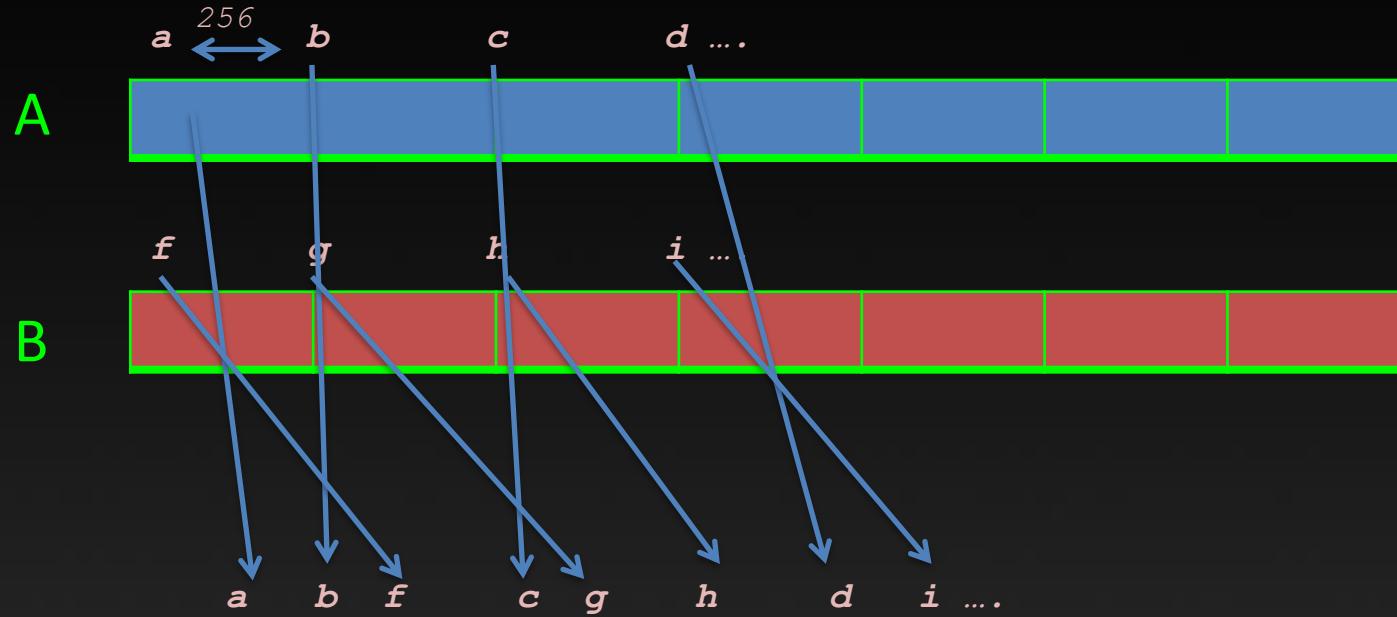
Stage 3: Large Merge (in the last case, a single merge of two half complete lists)

Split a single merge activity into parts



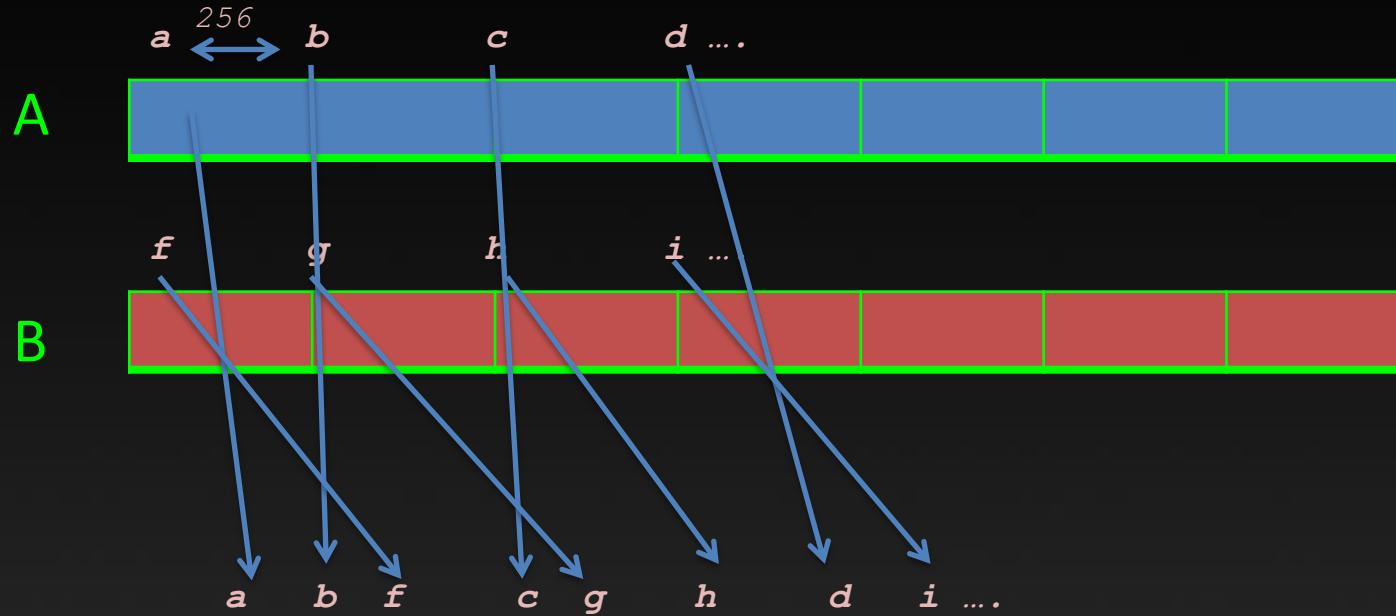
Choose a subset of each list, choosing every n^{th} element of each
For instance, take every 256 element of each list ($A[0], A[255], A[511]..., B[0], B[255], B[511]...$) → these elements are
called splitters

Stage 3: Large Merge



Merge these splitters into a list using the previous method

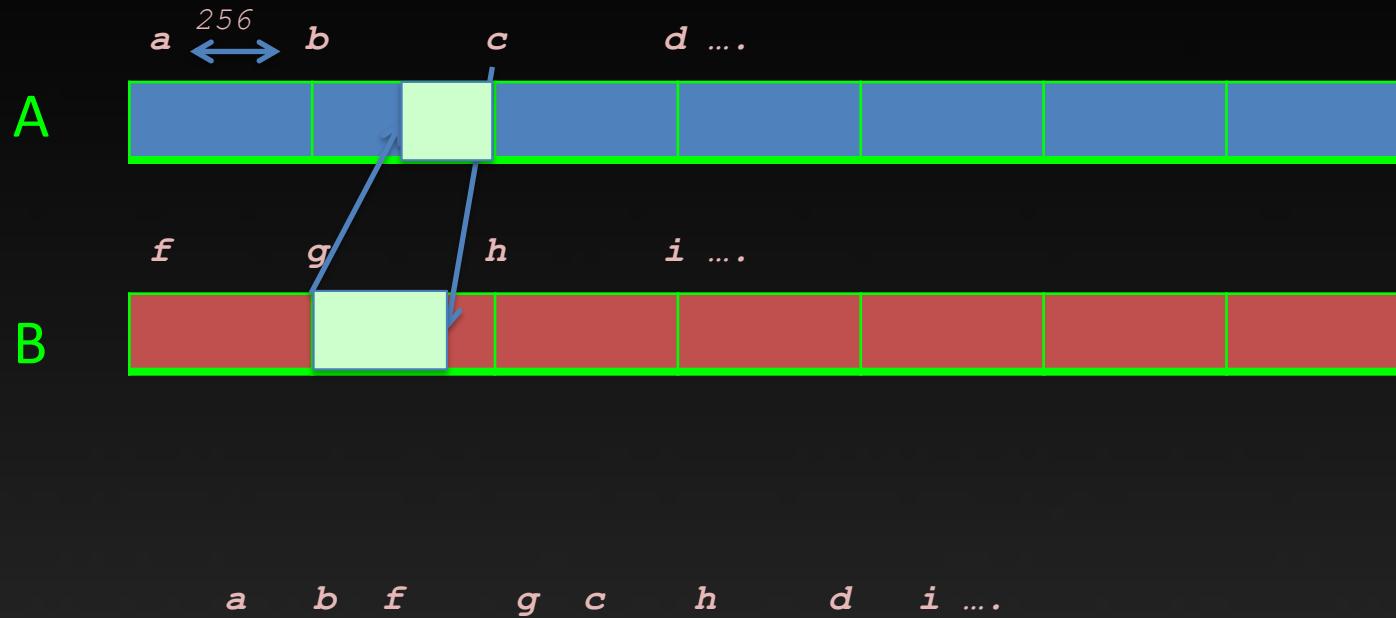
Stage 3: Large Merge



Create a block to merge each region. In the worst case, there will be 512 elements: $[a - b], [b - f], [f - c], \dots$

Order intervals or splitters using previous techniques: $a-b, b-f, f-c, \dots$

Stage 3: Large Merge



Create a block to merge each region. In the worst case, there will be 512 elements: [*a* – *b*], [*b* – *f*], [*f* – *c*], ...

Order intervals or splitters using previous techniques: *a*-*b*, *b*-*f*, *f*-*c*, ...

Compact

We want to select only a few elements from a big list and generate a new compacted list.

Ex.: only even numbers

1 3 4 9 8 10 7

4 8 10

Compact

1	3	4	9	8	10	7	
F	F	T	F	T	T	F	→ Predicate

-	-	4	-	8	10	-	→ Sparse
---	---	---	---	---	----	---	----------

4	8	10		→ Dense
---	---	----	--	---------

Sparse will generate a lot of empty threads...

Compact

1	3	4	9	8	10	7	
F	F	T	F	T	T	F	→ Predicate
0	0	1	0	1	1	0	

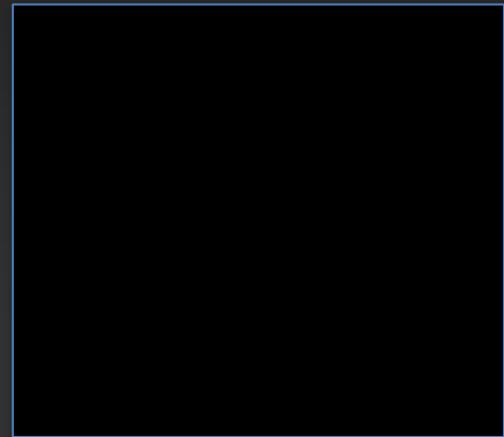
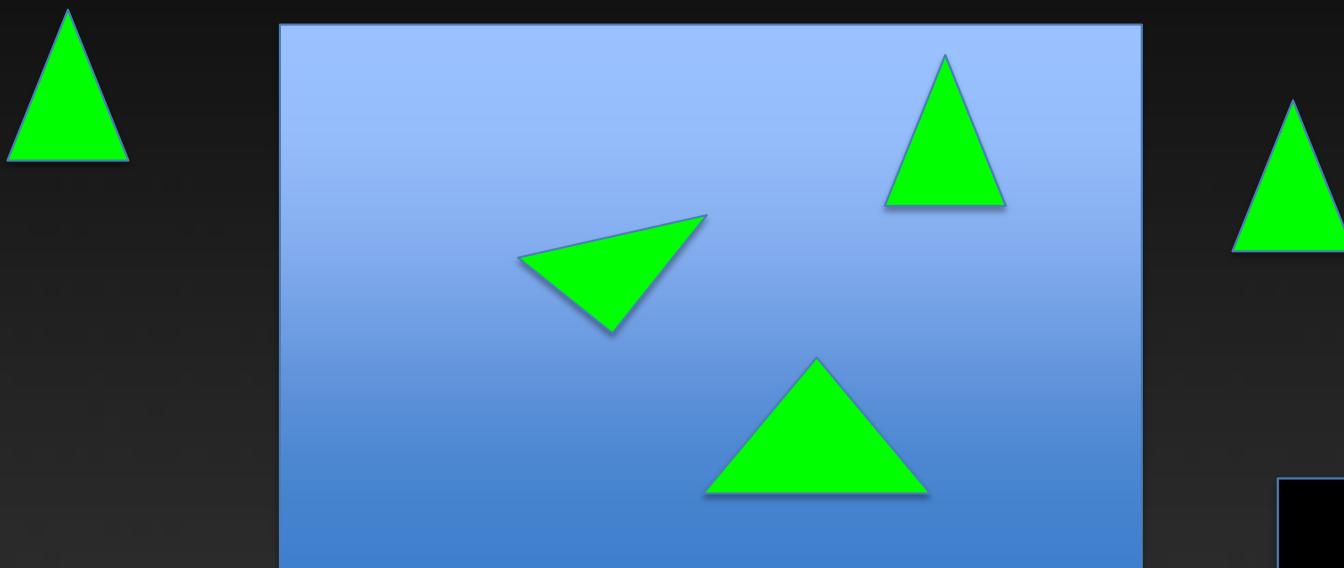
Exclusive Sum Scan:

0	0	0	1	1	2	3	
4	8	10					

Compact

- 1) Apply Predicate
- 2) Scan-in Array: true/false
- 3) Exclusive Sum Scan
- 4) Scatter input into output using generated addresses
(copy to position of sum scan vector, if predicate is true)

Compact Example: Clipping Triangles



Radix Sort

Good for number (int) ordering

Not based on comparison

Radix Sort

0	000
7	111
2	010
1	001

Colocar no topo os bits menos significativos e depois os mais:

0	000
2	010
7	111
1	001

Radix Sort

0	000
2	010
7	111
1	001

Colocar no topo os bits menos significativos e depois os mais:

0	000
1	001
2	010
7	111

$O(k \cdot n)$, onde k é o numero de bits usados para representar o numero

Radix Sort

Which operation separates the vector into 2 groups?

0	000
1	001
2	010
7	111

Compact, predicate: $(i \& 1) == 0$

Cooperative Threads

Historically, the CUDA programming model has provided a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block, as implemented with the `__syncthreads()` function.

Cooperative Threads

The Cooperative Groups programming model consists of the following elements:

- Data types representing groups of cooperating threads and their properties;
- Intrinsic groups defined by the CUDA launch API (e.g., thread blocks);
- Group partitioning operations;
- A group barrier synchronization operation;
- Group-specific collectives.

Using Tensor Cores

```
if (aRow < M && aCol < K && bRow < K && bCol < N) {  
    // Load the inputs  
    wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);  
    wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);  
  
    // Perform the matrix multiplication  
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);  
}
```

GPU Device Properties

CUDA Compute Capability

```
cudaDeviceProp prop;  
  
cudaGetDeviceProperties (&prop, 0);
```

GPU Device Properties

CUDA Compute Capability

```
cudaDeviceProp prop;

Int count;
cudaGetDeviceCount(&count);
For (int i=0; i<count; i++) {
    cudaGetDeviceProperties (&prop, i);

    printf("Name: %s\n", prop.name);
    printf("Compute Capability: %d.%d\n", prop.major, prop.minor);
...
}
```

GPU Device Properties

CudaDeviceProp

```
Char name[256]
Size_t totalGlobalMem
Size_t sharedMemPerBlock
Int regsPerBlock
Int warpSize
Size_t maxThreadsPerBlock
Int maxThreadsDim[3]
Int maxGridSize[3]
Size_t totalConstMem
Int major / int minor (compute capability major.minor)
...
```

GPU Device Properties

CUDA Compute Capability

```
cudaDeviceProp prop;  
Int dev;  
  
Memset (&prop, 0, sizeof (cudaDeviceProp) );  
  
// Indicates CUDA to choose a device that satisfies the  
cudaDeviceProp structure  
  
cudaChooseDevice (&dev, &prop) ;
```

GPU Device Properties

CUDA Compute Capability

```
cudaDeviceProp prop;  
Int dev;  
  
Memset (&prop, 0, sizeof (cudaDeviceProp) );  
  
Prop.minor=2;  
  
cudaChooseDevice (&dev, &prop) ;
```

GPU Device Properties

Compiling for Specific Compute Capability

```
Nvcc -arch=sm_11
```

Graphics Interoperability

Memory correspondent in OpenGL and CUDA

```
#include "GL/glut.h"
#include "cuda.h"
#include "cuda_gl_interop.h"

GLuint bufferObj;
cudaGraphicsResource *resource;
```

Graphics Interoperability

Memory correspondent in OpenGL and CUDA

```
cudaDeviceProp prop;  
Int dev;  
  
Memset (&prop, 0, sizeof (cudaDeviceProp) );  
Prop.minor=2;  
cudaChooseDevice (&dev, &prop) ;  
  
cudaGLSetGLDevice (dev) ;  
  
// This indicates that we intend to use both CUDA and  
OpenGL at the same device dev
```

Graphics Interoperability

Initialize OpenGL

```
glutInit (&argc, argv);  
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);  
glutInitWindowSize (DIM, DIM);  
glutCreateWindow ("application");
```

Graphics Interoperability

Creating Common Buffers:

1. Generate Buffer Handle
2. Bind Buffer to a Pixel Buffer
3. Allocate Buffer in memory

```
glBuffers(1, &bufferObj);  
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj);  
glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, DIM*DIM*4,  
             NULL, GL_DYNAMIC_DRAW_ARB);
```

Graphics Interoperability

Connect the Buffer and the CUDA memory (both are the same, but with different names and scopes)

```
cudaGraphicsGLRegisterBuffer (&resource, bufferObj,  
    cudaGraphicsMapFlagsNone) ;  
  
// cudaGraphicsMapFlagsNone (no particular behavior)  
// cudaGraphicsMapFlagsReadOnly (buffer is read only)  
// cudaGraphicsMapFlagsWriteDiscard (previous content  
will be discarded)
```

Graphics Interoperability

Create the address of resource in order to be passed, operated by the kernel

```
Uchar4* devPtr;  
Size_t size;  
  
cudaGraphicsMapResource (1, &resource, NULL);  
cudaGraphicsResourceGetMappedPointer ( (void**) &devPtr,  
&size, resource);
```

Graphics Interoperability

Proceed CUDA normally...

```
Dim3 grid(DIM/16,DIM/16) ;  
Dim3 block(16,16) ;  
  
Kernel <<<grid, block>>> (devPtr) ;  
  
cudaGraphicsUnmapResources (1, &resources, NULL) ;  
  
glutDisplayFun (draw_func) ;  
glutMainLoop () ;
```

Graphics Interoperability

Proceed CUDA normally...

```
__global__ void kernel (uchar4 *ptr)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // DO SOMETHING

    Ptr[offset].x = ???;
    Ptr[offset].y = ???;
    Ptr[offset].z = ???;
    Ptr[offset].w = ???;
}
```

Graphics Interoperability

Call Traditional OpenGL draw operation

```
Static void draw_func (void){  
    glDrawPixels (DIM, DIM, GL_RGBA, GL_UNSIGNED_BYTE,  
0);  
    glutSwapBuffers ();  
}
```

Warp Shuffle

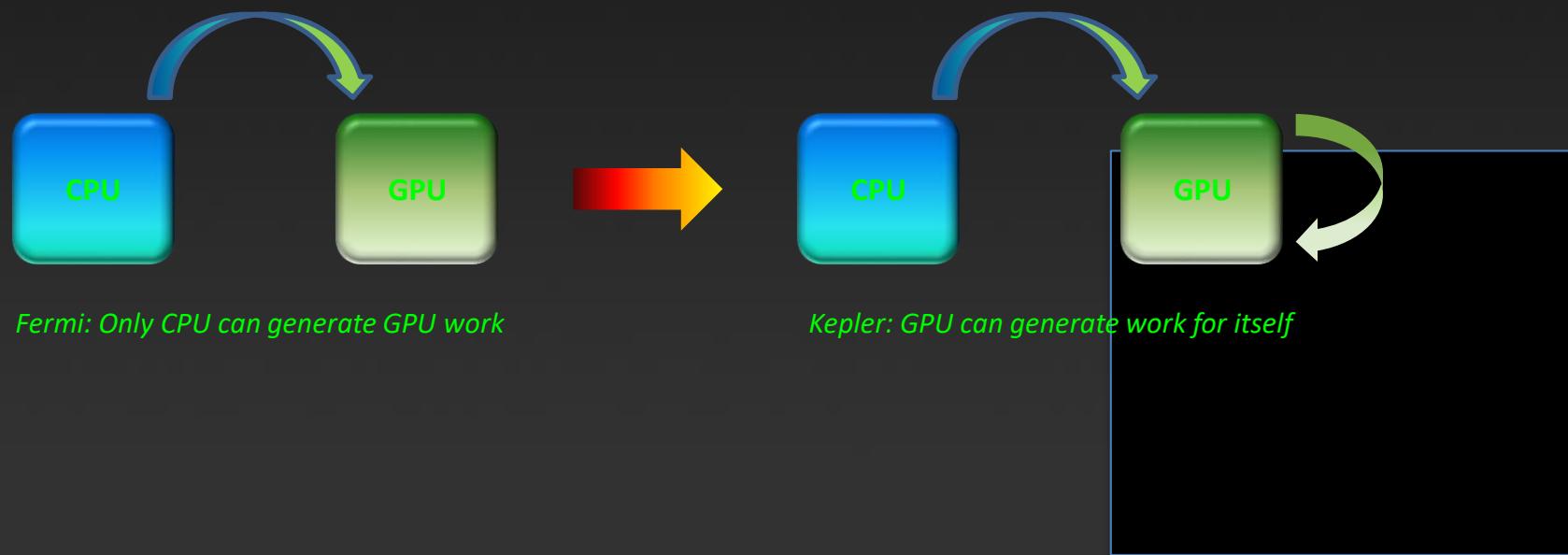
Threads of the same warp can read registers of other threads

```
x= __shfl (x, lane)  
  
// laneid: coordinate of the thread in a warp  
// laneid = threadIdx.x % 32  
// identifier in ptx: %laneid
```

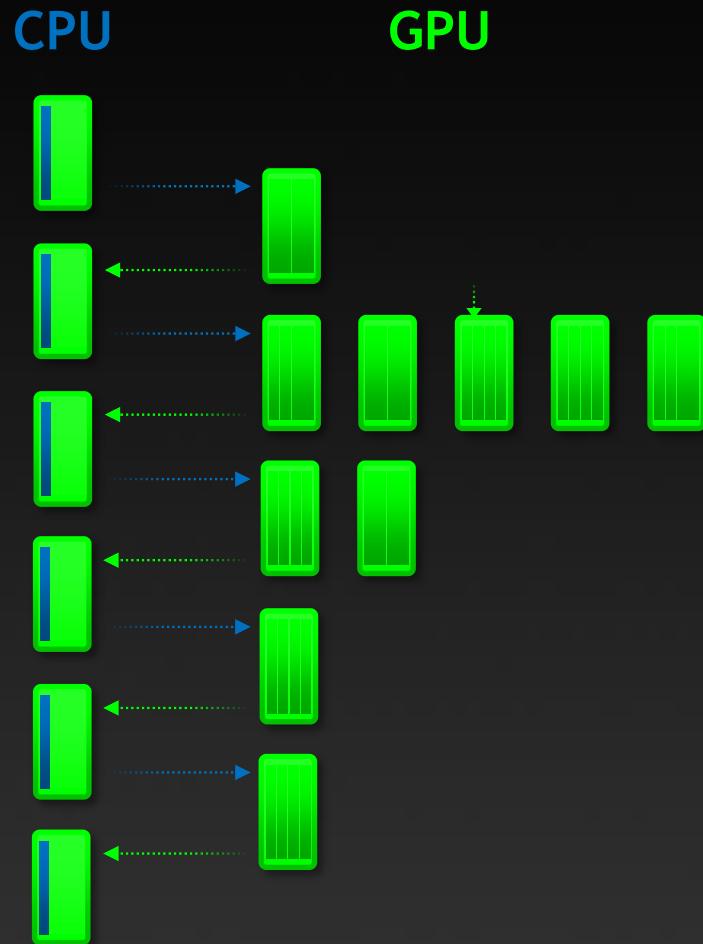
What is Dynamic Parallelism?

The ability to launch new grids from the GPU

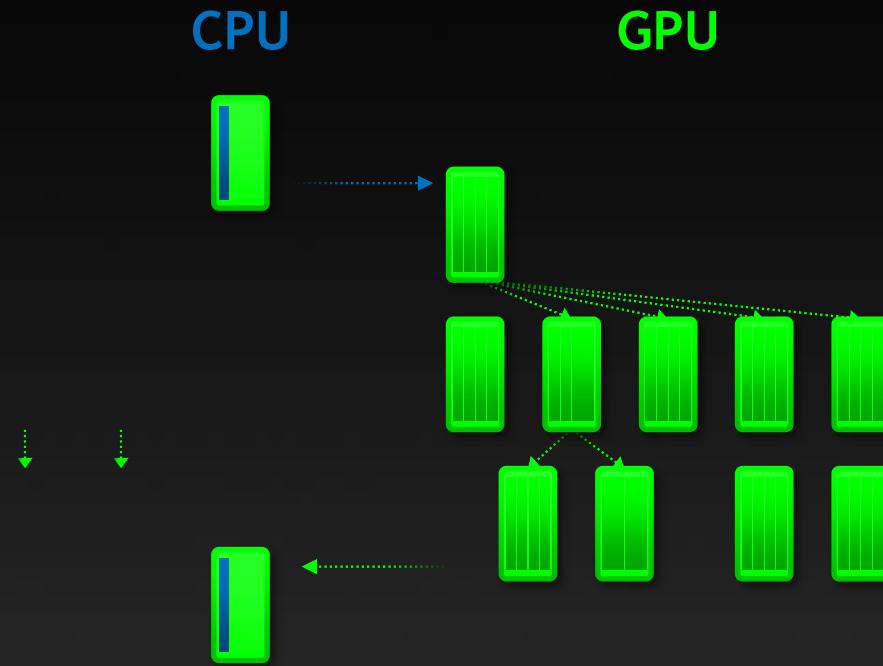
- Dynamically
- Simultaneously
- Independently



What Does It Mean?



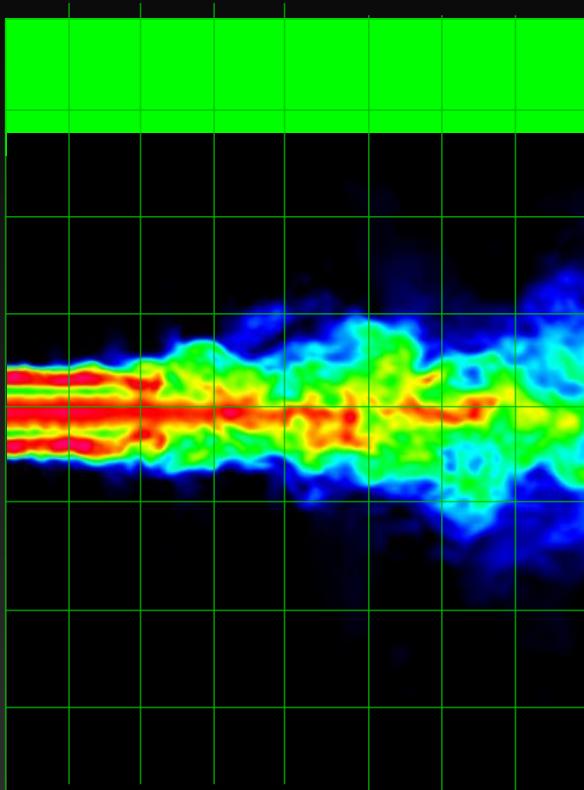
GPU as Co-Processor



Autonomous, Dynamic Parallelism

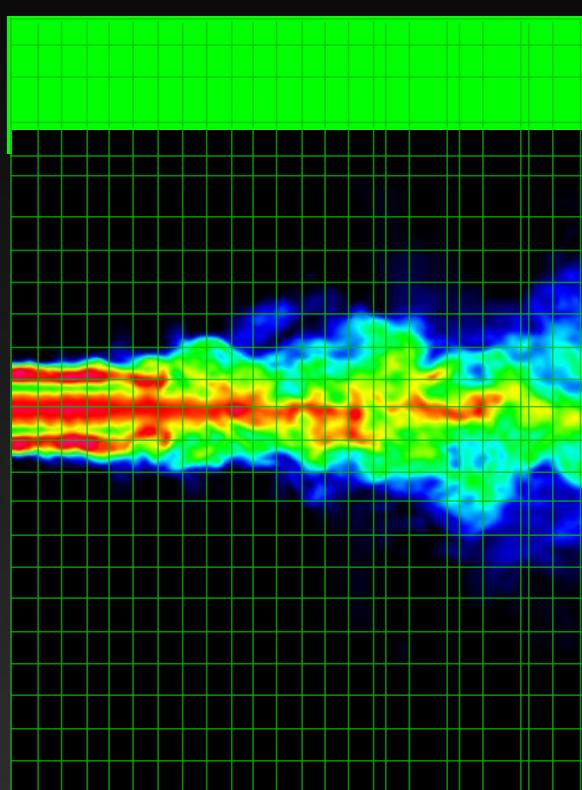
Dynamic Work Generation

Coarse grid



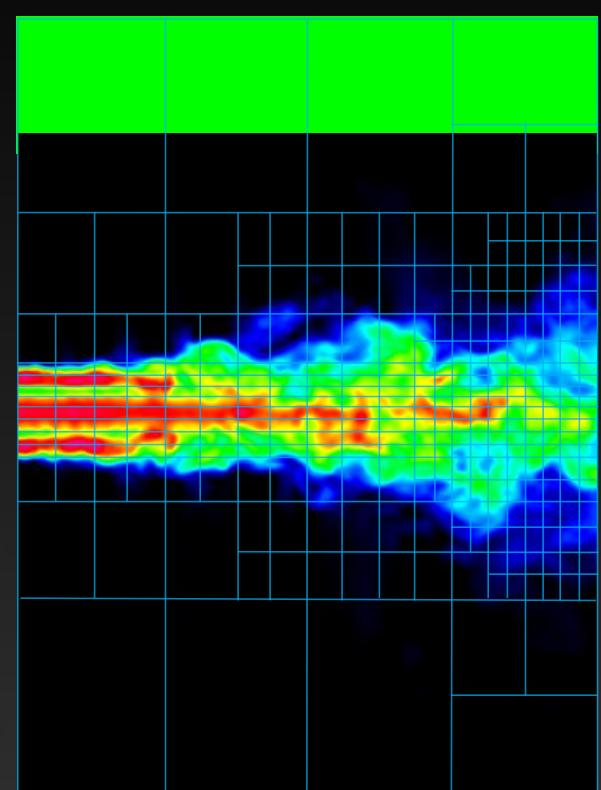
Higher Performance
Lower Accuracy

Fine grid



Lower Performance
Higher Accuracy

Dynamic grid



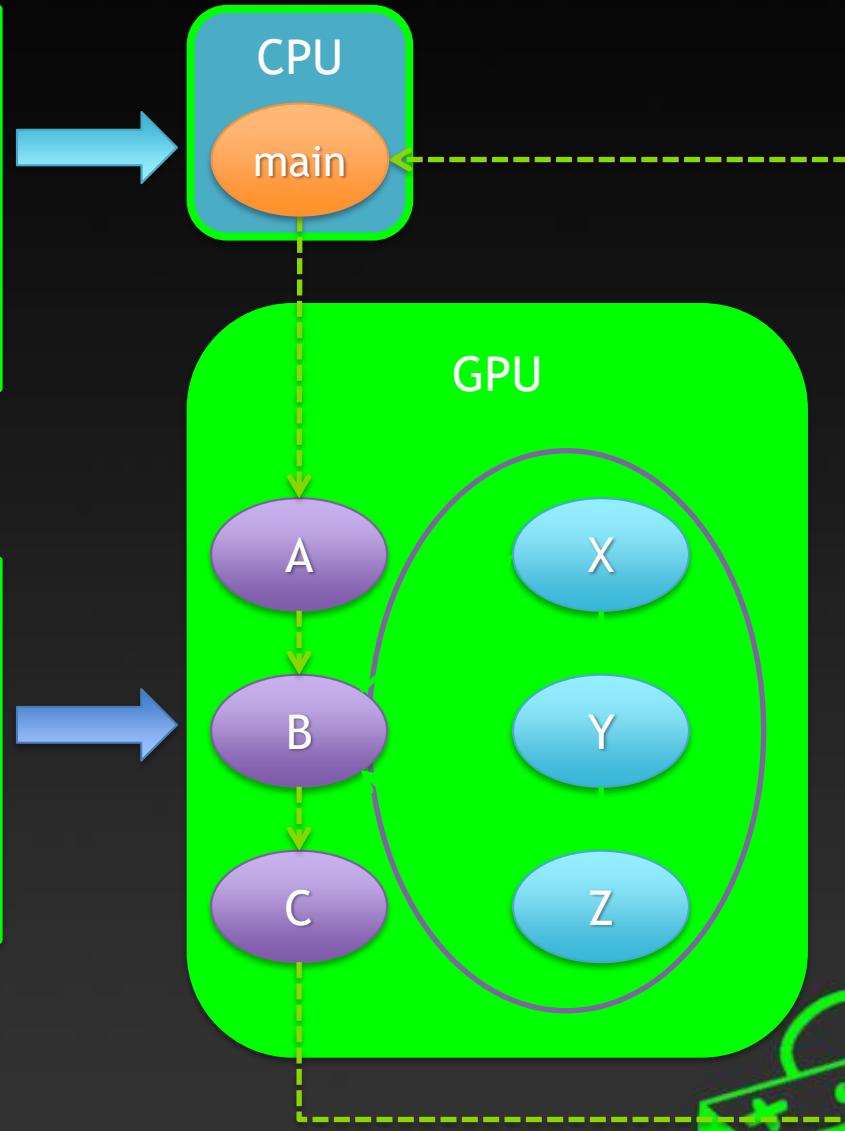
*Target performance where
accuracy is required*



Familiar Syntax and Programming Model

```
int main() {  
    float *data;  
    setup(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
  
    cudaDeviceSynchronize();  
    return 0;  
}
```

```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



Turing Architecture

- ▶ 14.2 TFLOPS¹ of peak single precision (FP32) performance
- ▶ 28.5 TFLOPS¹ of peak half precision (FP16) performance
- ▶ 14.2 TIPS¹ concurrent with FP, through independent integer execution units
- ▶ 113.8 Tensor TFLOPS^{1,2}
- ▶ 10 Giga Rays/sec
- ▶ 78 Tera RTX-OPS

The Quadro RTX 6000 provides superior computational performance designed for professional workflows:

- ▶ 16.3 TFLOPS¹ of peak single precision (FP32) performance
- ▶ 32.6 TFLOPS¹ of peak half precision (FP16) performance
- ▶ 16.3 TIPS¹ concurrent with FP, through independent integer execution units
- ▶ 130.5 Tensor TFLOPS^{1,2}
- ▶ 10 Giga Rays/sec
- ▶ 84 Tera RTX-OPS

RTX 2080



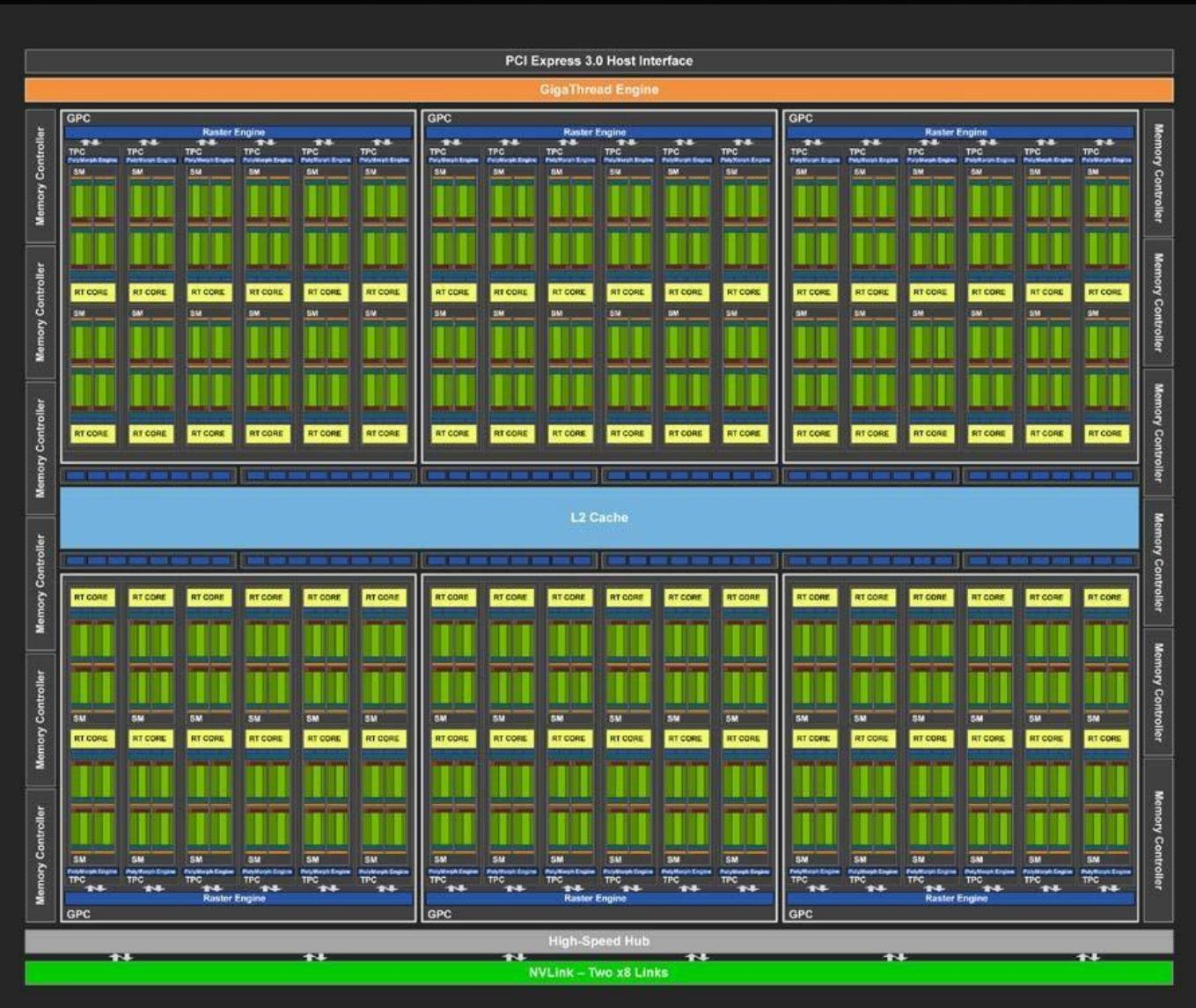
Turing Architecture

The TU102 GPU includes six Graphics Processing Clusters (GPCs), 36 Texture Processing Clusters (TPCs), and 72 Streaming Multiprocessors (SMs). (See Figure 2 for an illustration of the TU102 full GPU with 72 SM units.) Each GPC includes a dedicated raster engine and six TPCs, with each TPC including two SMs. Each SM contains 64 CUDA Cores, eight Tensor Cores, a 256 KB register file, four texture units, and 96 KB of L1/shared memory which can be configured for various capacities depending on the compute or graphics workloads. RTX 2080

- ▶ 4,608 CUDA Cores
- ▶ 72 RT Cores
- ▶ 576 Tensor Cores
- ▶ 288 texture units
- ▶ 12 32-bit GDDR6 memory controllers (384-bits total).



Turing Architecture



Turing Architecture



Concurrent INT/FLOAT execution



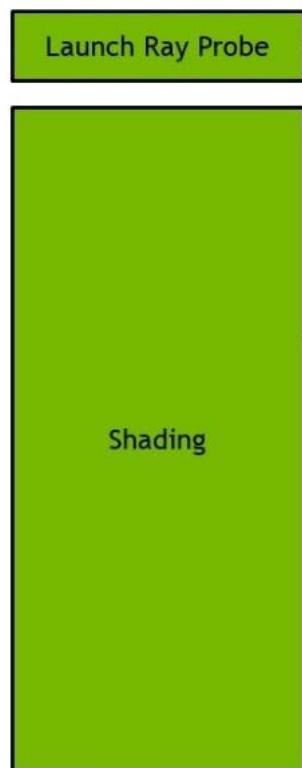
RT Cores

Hardware Acceleration Replaces Software Emulation

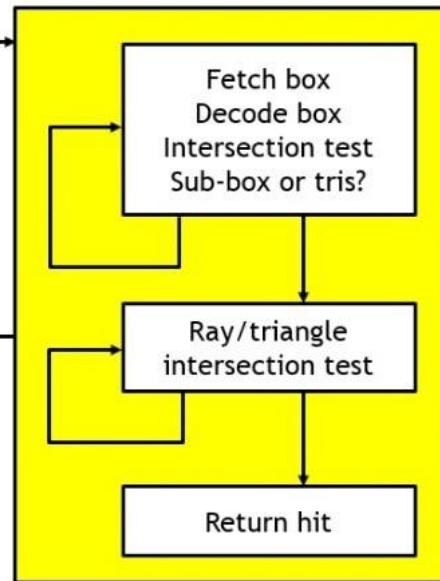
Turing SM



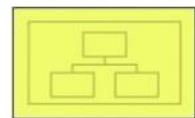
Shaders



RT Core



Box
Intersection
Evaluators



Triangle
Intersection
Evaluators



Acesso Remoto a máquinas com Suporte CUDA

Instruções para uso das máquinas remotas para programação em CUDA.

<http://tinyurl.com/q5mafjx>



Instruções Importantes.

- Você irá utilizar a máquina remota somente para compilar e executar o programa.
- Existe um diretório chamado Alunos na raíz 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 varias pessoas da mesma turma utilizando a mesma conta.

Instrucoes 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/>

Divisão dos Grupos de Acesso

- Dividir a sala em QUATRO grupos de acesso
 - Grupo 01
 - Grupo 02
 - Grupo 03
 - Grupo 04
- Essa divisão é necessário para equilibrar a carga dos usuarios, para não deixar a máquina lenta.
- Caso a máquina do seu grupo não esteja disponível ou com algum problema, utilizar a maquina do próximo grupo.

Divisao de Acceso das máquinas

- Grupo 01 devem se conectar a máquina:
 - 200.20.15.157 – Porta: 2022 ou 22 (tentar as duas)
- Grupo 02 devem se conectar a máquina:
 - 200.20.15.115 – Porta: 2022/22 (tentar as duas)
- Grupo 03 devem se conectar a máquina:
 - 200.20.15.116 – Porta: 2022/22 (tentar as duas)
- Grupo 04 devem se conectar a máquina:
 - 200.20.15.117 – Porta: 2022/22 (tentar as duas)

usuario e senha

u: cudateaching

p:

ProgrammingCUDANVIDI@20172

0172017

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

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-7.0_Samples que está na raíz da máquina, é a pasta da NVIDIA contendo todos os exemplos de desenvolvimento do SDK. Voce 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 seguinte programa através do link
<https://dl.dropboxusercontent.com/u/198995/static/VectorAdd-SampleCode.zip>
- Esse código nada mais é que o exemplo padrão da NVIDIA de soma de dois vetores em paralelo.
- Voce pode baixar este software ZIP através do comando
- “wget
<https://dl.dropboxusercontent.com/u/198995/static/VectorAdd-SampleCode.zip>” e depois deve descompactar o arquivo

Rodando um Hello World em CUDA

01

- Para compilar o projeto basta digitar MAKE e pressionar enter.
- Para limpar o projeto basta digitar MAKE CLEAN e pressionar enter.
- Fique a vontade para modificar a copia do projeto da maneira que lhe for mais conveniente.

Rodando um Hello World em CUDA

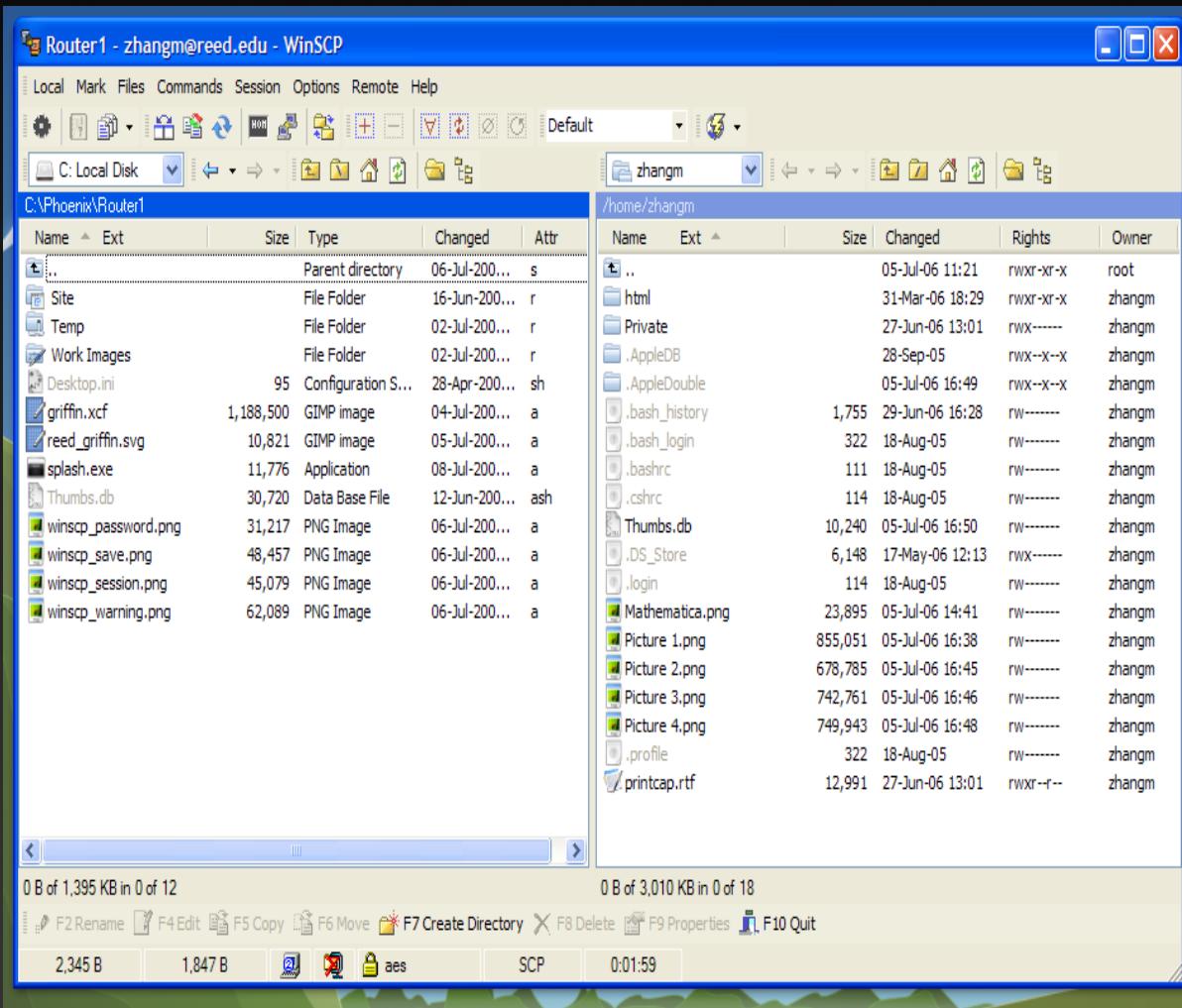
03

- Alternativamente este mesmo software que soma dois vetores, também está disponível na pasta NVIDIA_CUDA-7.0_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 maquina remota a partir de Windows

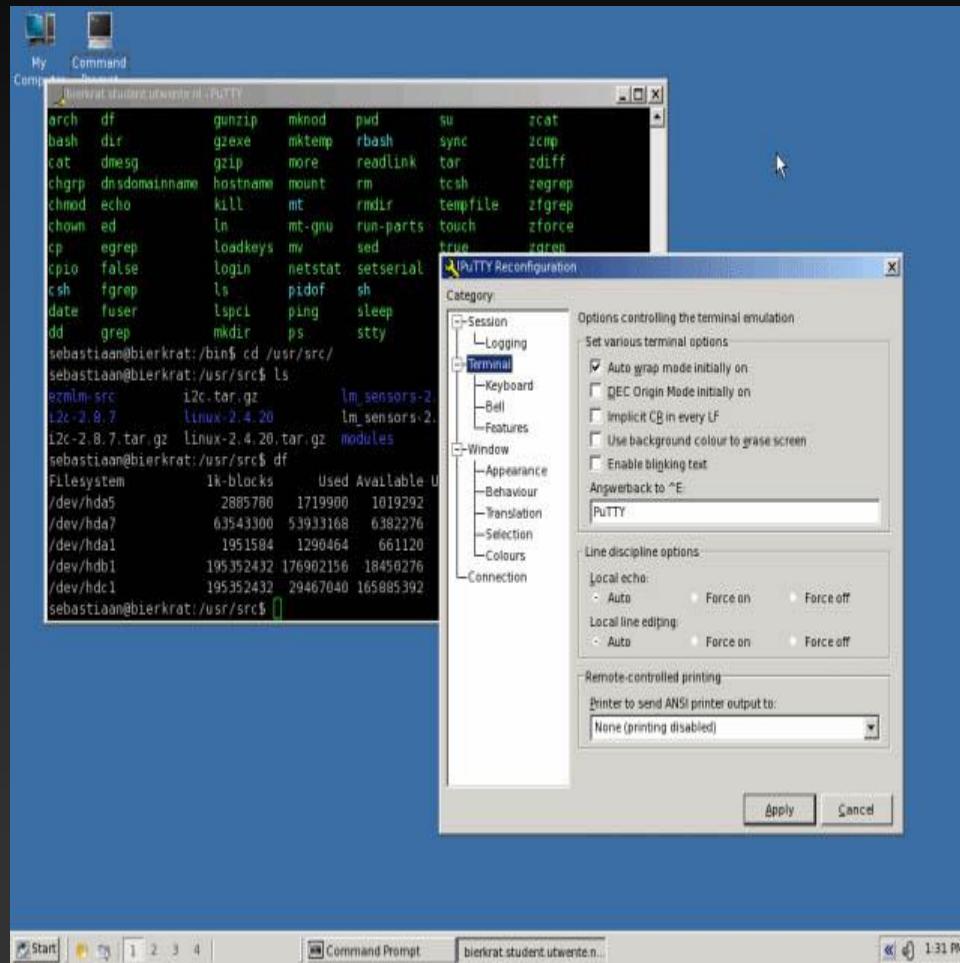
- MOBAXTERM -
<http://mobaxterm.mobatek.net/>
- Através desse software voce se conecta a maquina desejada, e ele dispoe de todas as ferramentas necesarias, desde transferencia de arquivo através de interface, até terminal e editor de texto.

winSCP- Troca de Arquivos- <http://winscp.net/>



Putty - terminal de acesso remoto

– SSH Client -www.putty.org – para windows

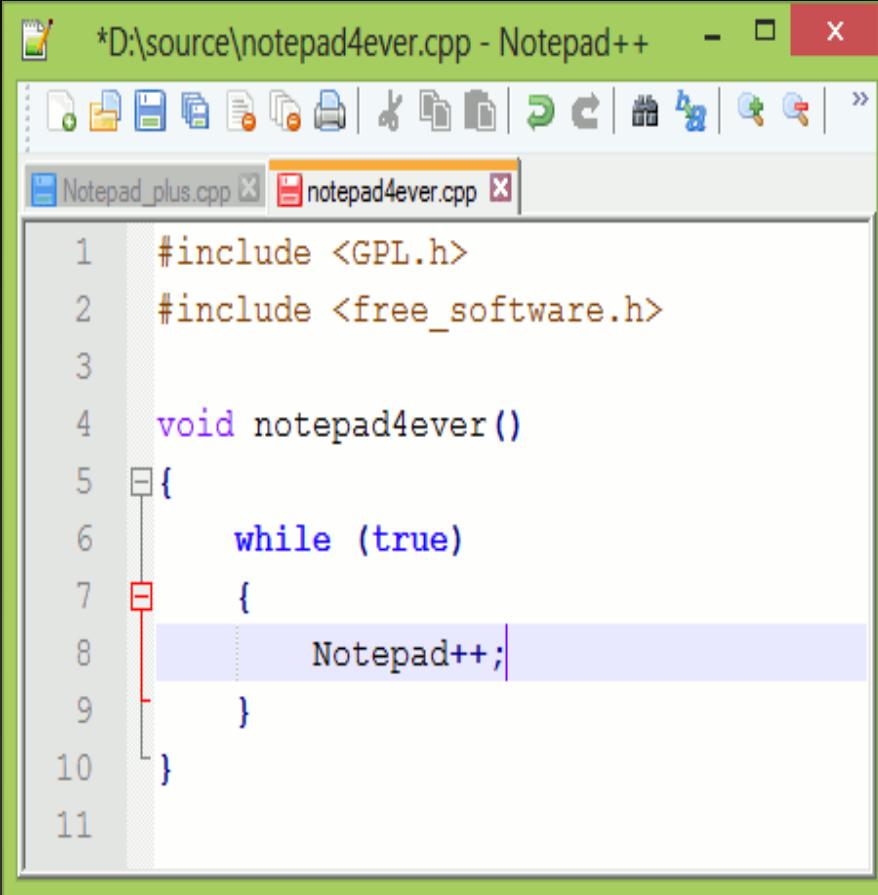


Terminal (MAC)

ssh cudateaching@200.20.15.157

Notepad++ - Editar código fonte.

<http://notepad-plus-plus.org/>



The screenshot shows the Notepad++ interface with a dark theme. The title bar reads "*D:\source\notepad4ever.cpp - Notepad++". The menu bar includes File, Edit, View, Insert, Search, Tools, Plugins, Settings, Help, and Exit. The toolbar contains icons for new file, open file, save file, cut, copy, paste, find, and search. The tabs at the top show "Notepad_plus.cpp" and "notepad4ever.cpp". The code editor displays the following C++ code:

```
1 #include <GPL.h>
2 #include <free_software.h>
3
4 void notepad4ever()
5 {
6     while (true)
7     {
8         Notepad++;
9     }
10 }
```

The word "Notepad" is highlighted in blue, indicating it is a variable or identifier being searched or selected. The code editor has a light gray background with dark gray horizontal and vertical lines for the code structure. The status bar at the bottom is visible but contains no text.

Em caso de dúvida entre em contato através do email do laboratório:

suportemedialab@gmail.com

Dica Final

Para entender na prática como utilizar uma função da biblioteca CUDA, somente tem que escrever o nome da função desejada no google, seguida do comando “filetype:cu”.

Veja o exemplo ao lado na figura com cudaMalloc.

Voce vai achar varios exemplos de códigos de uso da função desejada.

The screenshot shows a Google search results page with the query "filetype:cu cudaMalloc" entered in the search bar. The results are filtered to show only CUDA source code files (".cu"). There are approximately 13,700 results found in 0.41 seconds. The results list several examples of how to use the cudaMalloc function, such as in matrix multiplication programs and genetic algorithms.

- CUDA Matrix Multiplication program**
www.arc.vt.edu/resources/software/cuda/docs/MatMul.cu ▾
... bytes float *dA, *dB, *dC; cudaMalloc(&dA, size); cudaMalloc(&dB, size);
cudaMalloc(&dC, size); dim3 threadBlock(BLOCK_SIZE,BLOCK_SIZE); dim3 grid(K,K); ...
- bisect_large.cu**
www.naic.edu/~phil/hardware/nvidia/doc/src/.../bisect_large.cu ▾
... eigenvalue after the first step cutlSafeCall(cudaMalloc((void**) &result.g_num_one, sizeof(unsigned int))); cutlSafeCall(cudaMemcpy(result.g_num_one, ...
- genetic_algorithm/cudaMalloc.cu at master · goghino ...**
https://github.com/goghino/genetic_algorithm/blob/.../cudaMalloc.cu ▾
Thrust kernels allocate some GPU memory for internal use. // This file implements
cudaMalloc/cudaFree hooks to re-use a single. // GPU memory allocation for ...
- ppc.cu - IceCube**
icecube.wisc.edu/~dima/work/WISC/ppc/bkp/old/gpu-old/v4/ppc.cu ▾
n", size, NBLK*NTHR); } { unsigned int size=d.size*sizeof(unsigned int);
checkError(cudaMalloc((void**) &e.rm, size)); checkError(cudaMemcpy(e.rm, d.rm, size, ...
- cuDPP -- CUDA Data Parallel Primitives library ...**
cudpp.googlecode.com/svn-history/r289/branches/.../compress_app.cu ▾
... CUDA_SAFE_CALL(cudaMalloc((void**) &(plan->m_d_keys), numElts*sizeof(unsigned int))); CUDA_SAFE_CALL(cudaMalloc((void**) &(plan->m_d_values) ...

site de acesso:
<https://codeanywhere.com/edit>
or/

Programei as portas 2022,

u: esteban@gazolla.com

p: gpuuff2016

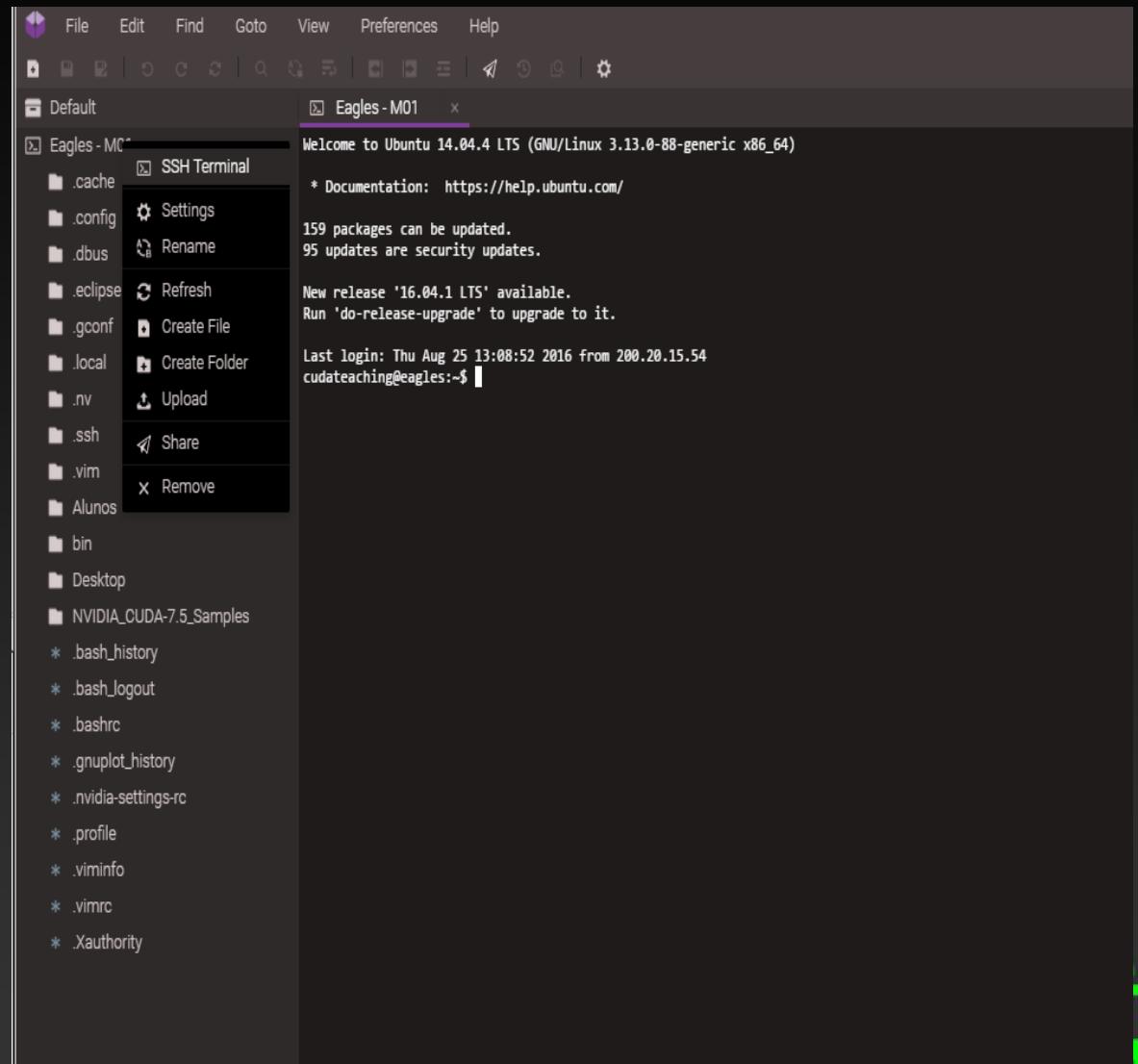


**So da pra cadastrar uma maquina
remota na versao gratis.**

Ia cadastrar todas as maquinas, mas na
versao gratis so pode apenas uma por vez.



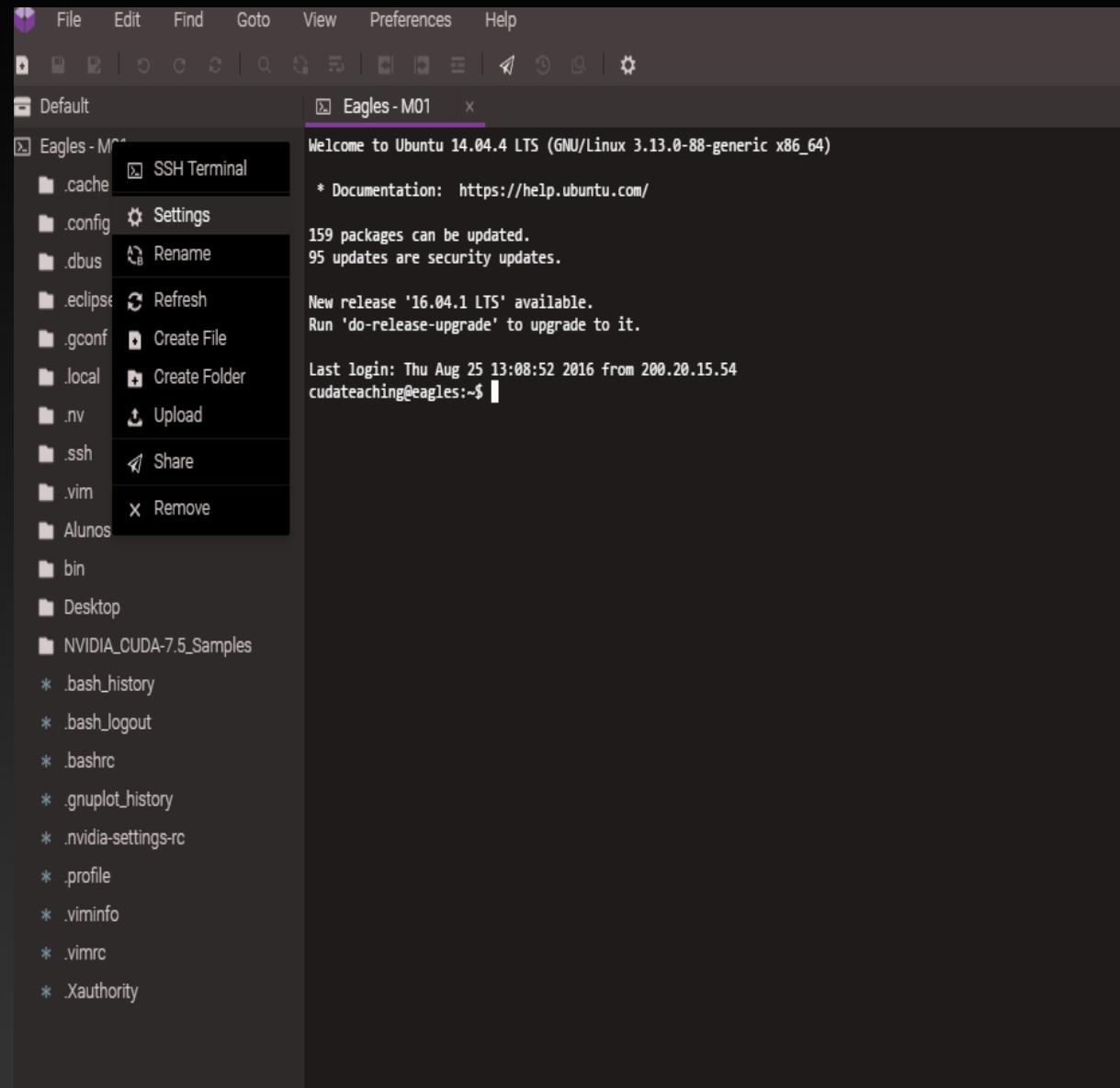
Pra abrir o
terminal
Clica em cima
da maquina
E clica em SSH
TERMINAL



Sobre as Portas

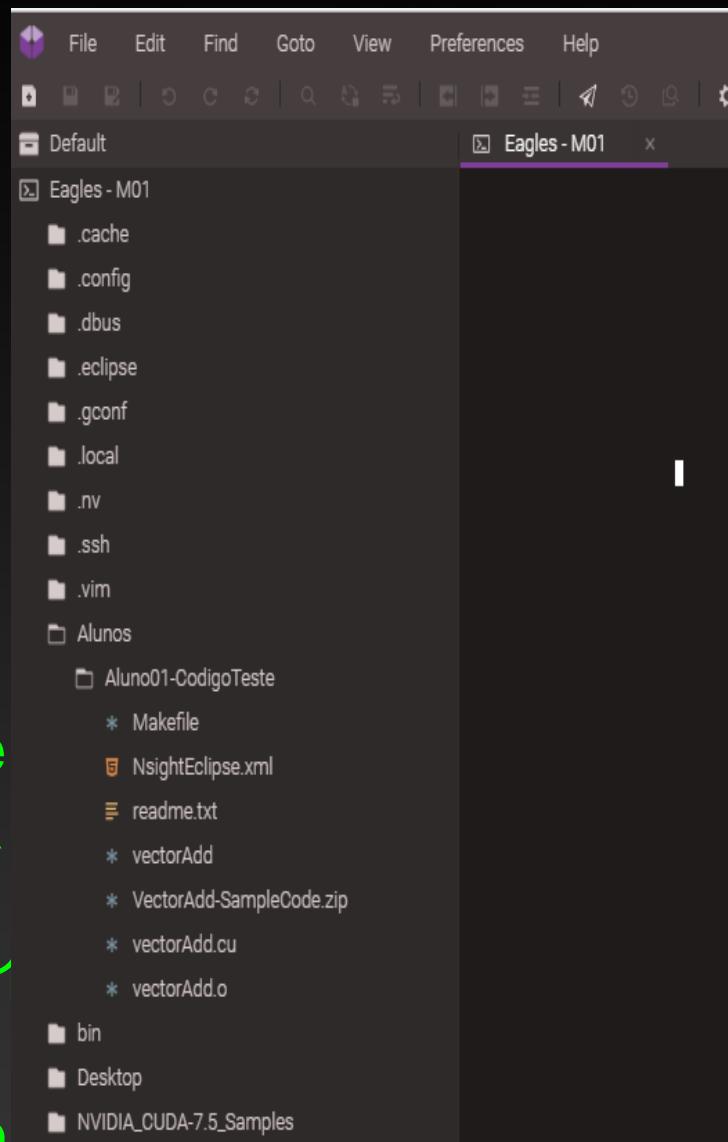
Pra quem
acessa de for a
da UFF
A porta de SSH
eh 2022
De dentro eh

22



Onde esta o projeto

O projeto teste
Esta dentro da pasta
Alunos/aluno0'-codigoteste
Basta clicar no arquivo que



Dividindo as telas

Na aba view
voce consegue
Mudar o layout
da IDE

