

# INTRODUÇÃO À PROGRAMAÇÃO DE GPUs COM A PLATAFORMA CUDA

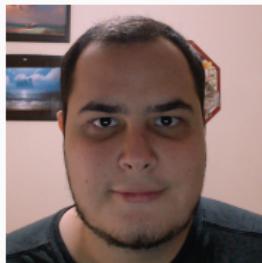
---

Pedro Bruel  
[phrb@ime.usp.br](mailto:phrb@ime.usp.br)  
04 de Agosto de 2016



Instituto de Matemática e Estatística  
Universidade de São Paulo

# SOBRE



Pedro Bruel



Alfredo Goldman

- [phrb@ime.usp.br](mailto:phrb@ime.usp.br)
- [www.ime.usp.br/~phrb](http://www.ime.usp.br/~phrb)
- [github.com/phrb](https://github.com/phrb)
- [gold@ime.usp.br](mailto:gold@ime.usp.br)
- [www.ime.usp.br/~gold](http://www.ime.usp.br/~gold)

# PESQUISA

Meus interesses de **pesquisa**:

- *Autotuning*
- *Stochastic Local Search*
- *Model Based Search*

# PESQUISA

Meus interesses de **pesquisa**:

- *Autotuning*
- *Stochastic Local Search*
- *Model Based Search*
- GPUs, FPGAs, *cloud*
- Julia Language

# PESQUISA

Meus interesses de **pesquisa**:

- *Autotuning*
- *Stochastic Local Search*
- *Model Based Search*
- GPUs, FPGAs, *cloud*
- Julia Language
- Colaboração! ([phrb@ime.usp.br](mailto:phrb@ime.usp.br))

# PARTE I

1. Introdução
2. Computação Heterogênea
3. GPUs
4. Plataforma CUDA
5. CUDA C

## PARTE II

6. Retomada
7. Compilação de Aplicações CUDA
8. Boas Práticas em Otimização
9. Análise de Aplicações CUDA
10. Otimização de Aplicações CUDA
11. Conclusão

# SLIDES



O *pdf* com as aulas e todo o código fonte estão no [GitHub](#):

- [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda)

# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a grandes conjuntos de dados:

# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a grandes conjuntos de dados:

- Associação a um processador **hospedeiro** (*host*)

# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a grandes conjuntos de dados:

- Associação a um processador **hospedeiro** (*host*)
- Controle e **memória** próprios

# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a grandes conjuntos de dados:

- Associação a um processador **hospedeiro** (*host*)
- **Controle e memória** próprios
- Diferem em **especialização** e **configurabilidade**

# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (*devices*) para acelerar computações aplicadas a grandes conjuntos de dados:

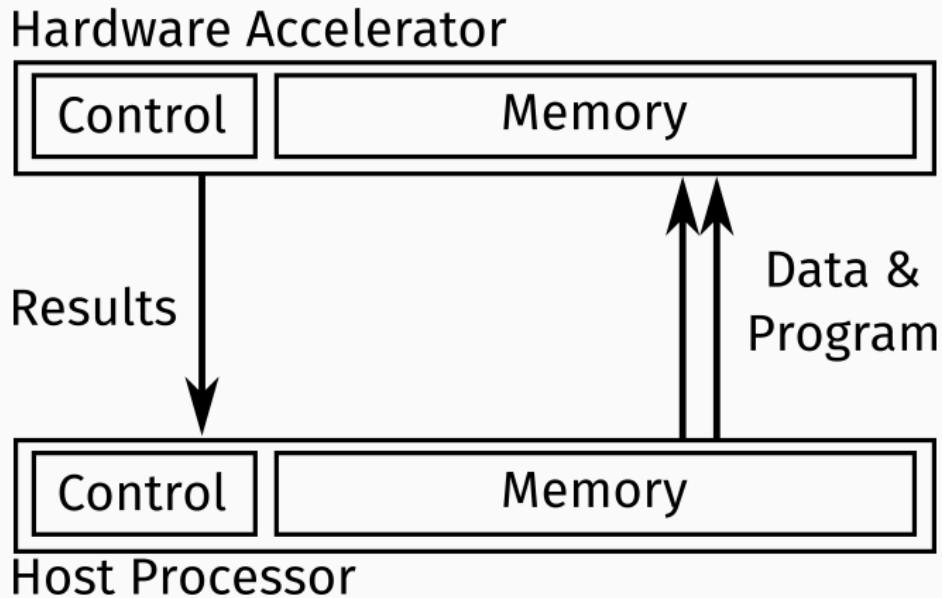
- Associação a um processador **hospedeiro** (*host*)
- **Controle e memória** próprios
- Diferem em **especialização** e **configurabilidade**
- **GPUs**, DSPs, FPGAs, ASICs

# ACELERAÇÃO POR *HARDWARE*

Casos de uso:

- Aprendizagem Computacional
- Processamento Digital de Sinais e Imagens
- Bioinformática
- Criptografia
- Meteorologia
- Simulações
- ...

# ACELERAÇÃO POR *HARDWARE*



# ACELERAÇÃO POR HARDWARE

Porcentagem de sistemas com aceleradores na Top500:

## ACCELERATORS/CO-PROCESSORS

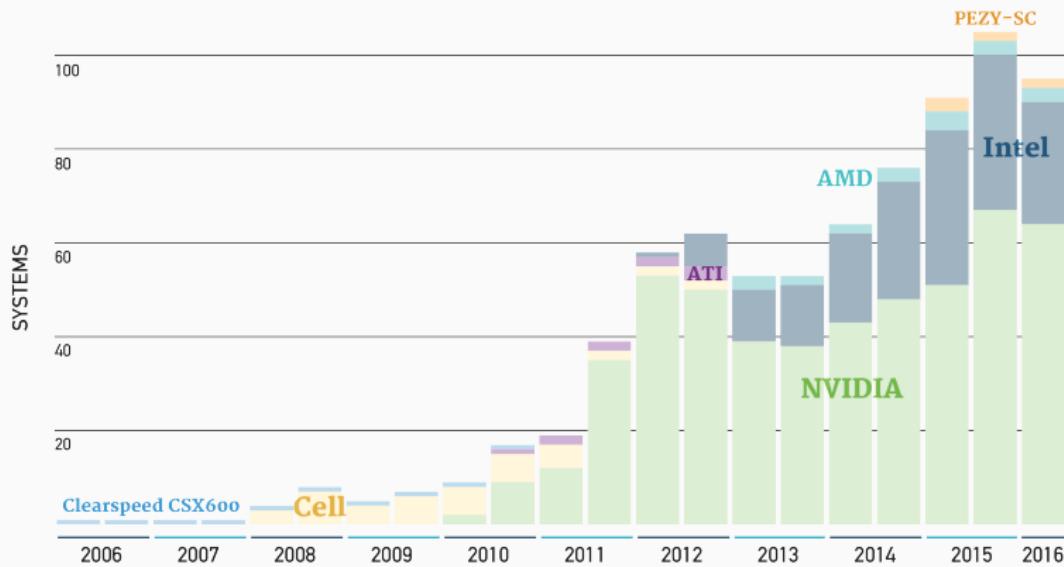
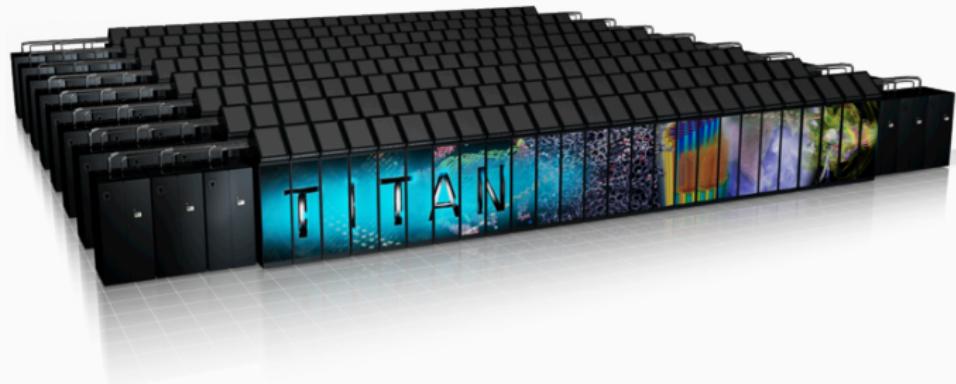


Imagen: [top500.org/lists/2016/06/download/TOP500\\_201606\\_Poster.pdf](http://top500.org/lists/2016/06/download/TOP500_201606_Poster.pdf) [Acessado em 29/07/16]

# COMPUTAÇÃO HETEROGÊNEA



# COMPUTAÇÃO HETEROGÊNEA



Dados recursos computacionais heterogêneos, conjuntos de dados e computações, como distribuir computações e dados de forma a otimizar o uso dos recursos?

Imagen: [olcf.ornl.gov/titan](http://olcf.ornl.gov/titan) [Acessado em 29/07/16]

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais heterogêneos:

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais heterogêneos:

- Baixa latência: CPUs

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais heterogêneos:

- Baixa latência: CPUs
- Alta vazão: GPUs

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais **heterogêneos**:

- Baixa latência: CPUs
- Alta vazão: GPUs
- Reconfiguráveis: FPGAs

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais **heterogêneos**:

- Baixa latência: CPUs
- Alta vazão: GPUs
- Reconfiguráveis: FPGAs
- Especializados: ASICs, DSPs

# COMPUTAÇÃO HETEROGÊNEA

Recursos computacionais **heterogêneos**:

- Baixa latência: CPUs
- Alta vazão: GPUs
- Reconfiguráveis: FPGAs
- Especializados: ASICs, DSPs
- Memória?

# COMPUTAÇÃO HETEROGÊNEA

Conjuntos de **dados**:

- *Big Data*
- *Data Streams*
- ...

# COMPUTAÇÃO HETEROGÊNEA

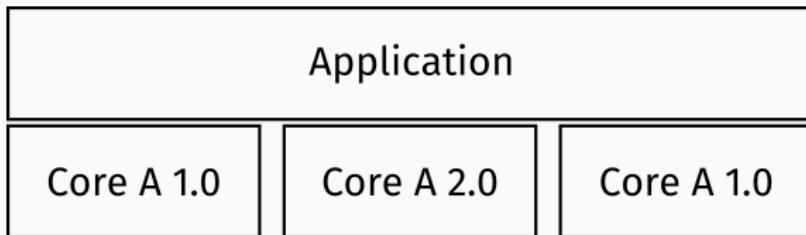
Conjuntos de **dados**:

- *Big Data*
- *Data Streams*
- ...

Modelos de programação (**computações**):

- *MapReduce*
- *Task Parallelism*
- ...

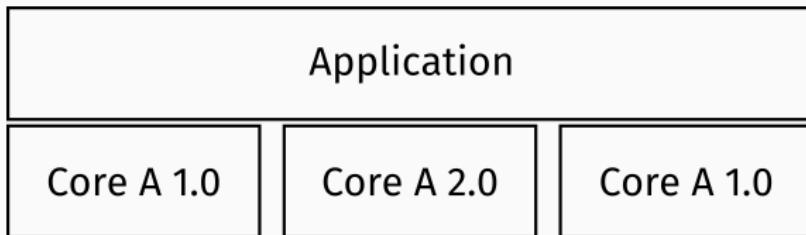
# COMPUTAÇÃO HETEROGÊNEA: ESCALABILIDADE



Escalabilidade significa não perder desempenho executando em:

- Múltiplos *cores* idênticos

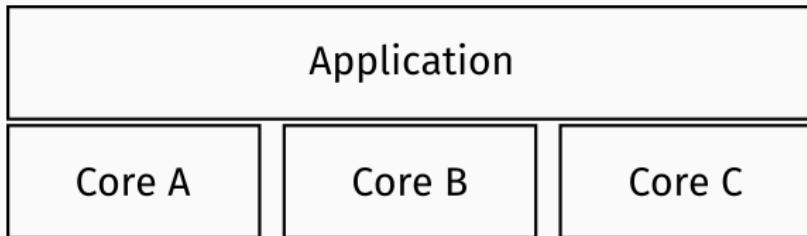
# COMPUTAÇÃO HETEROGÊNEA: ESCALABILIDADE



Escalabilidade significa não perder desempenho executando em:

- Múltiplos *cores* idênticos
- Novas versões do mesmo *core*

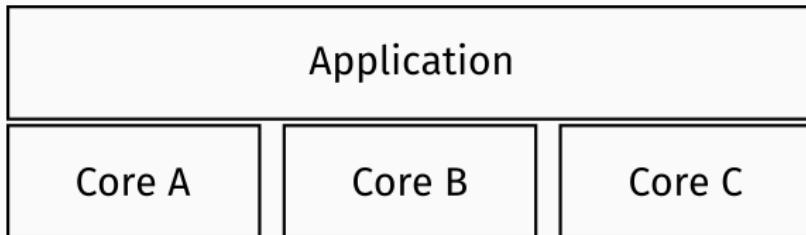
# COMPUTAÇÃO HETEROGÊNEA: PORTABILIDADE



Portabilidade significa não perder desempenho executando em diferentes:

- Cores ou aceleradores

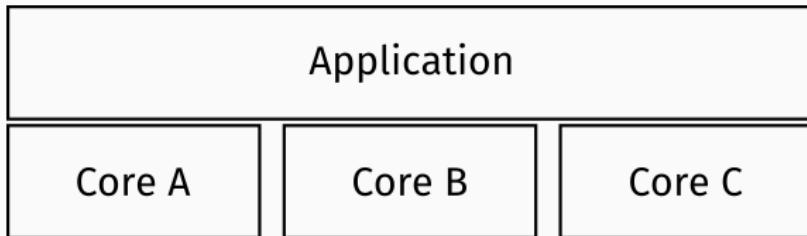
# COMPUTAÇÃO HETEROGÊNEA: PORTABILIDADE



Portabilidade significa não perder desempenho executando em diferentes:

- Cores ou aceleradores
- Modelos de memória

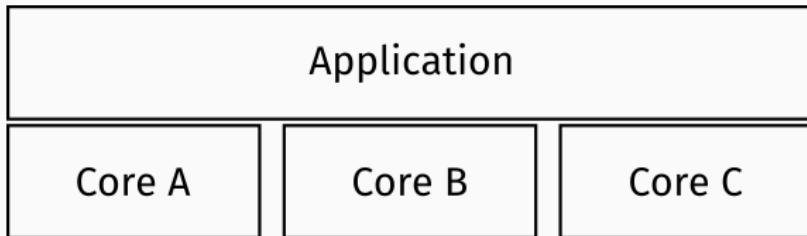
# COMPUTAÇÃO HETEROGÊNEA: PORTABILIDADE



Portabilidade significa não perder desempenho executando em diferentes:

- Cores ou aceleradores
- Modelos de memória
- Modelos de paralelismo

# COMPUTAÇÃO HETEROGÊNEA: PORTABILIDADE

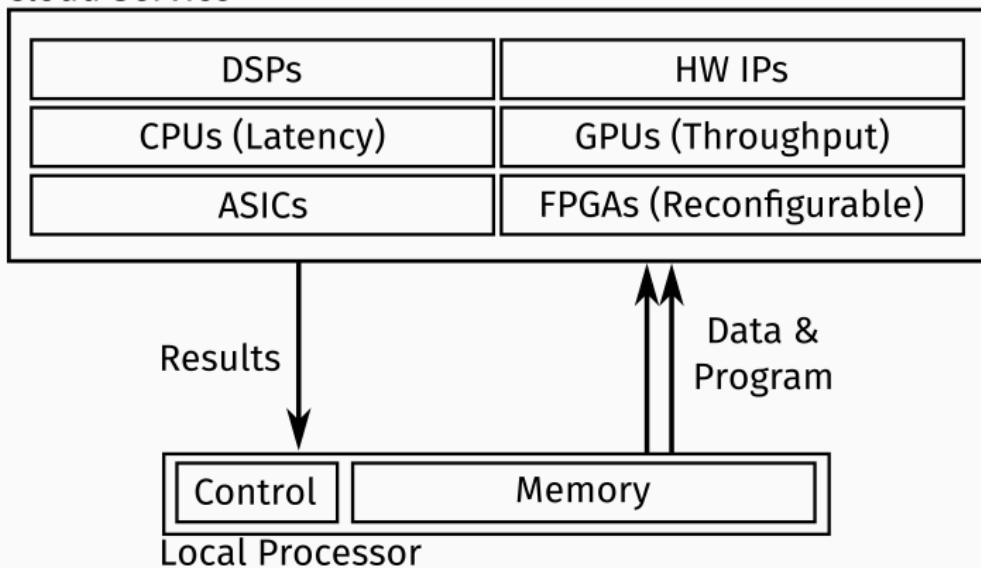


Portabilidade significa não perder desempenho executando em diferentes:

- Cores ou aceleradores
- Modelos de memória
- Modelos de paralelismo
- Conjuntos de instruções

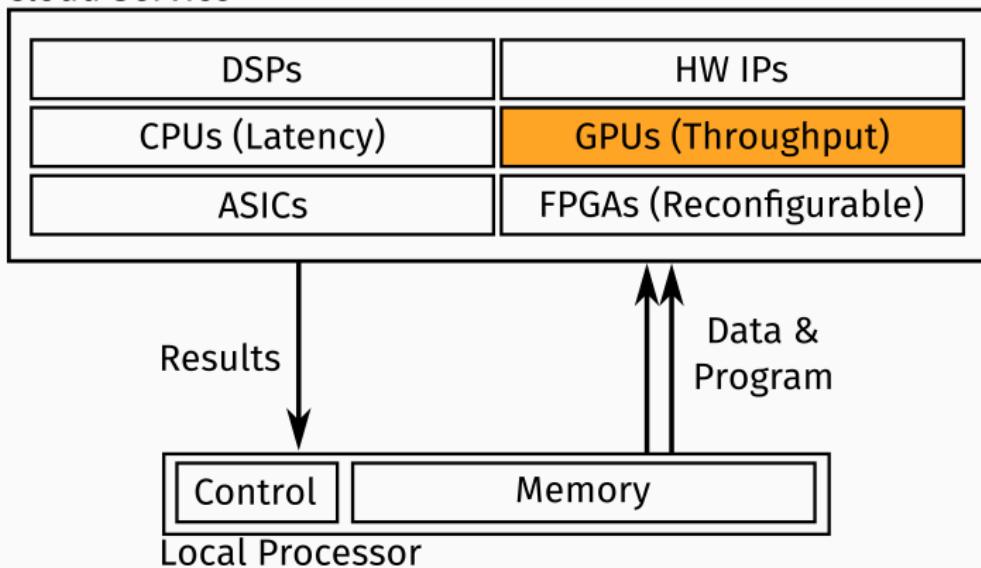
# COMPUTAÇÃO HETEROGÊNEA

Cloud Service



# COMPUTAÇÃO HETERÔGENEA

Cloud Service



# GRAPHICS PROCESSING UNITS



# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

- Caches pequenos (*kilobytes*)

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

- Caches pequenos (*kilobytes*)
- Sem branch prediction

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

- Caches pequenos (*kilobytes*)
- Sem branch prediction
- Milhares de ALUs de maior latência

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

- Caches pequenos (*kilobytes*)
- Sem branch prediction
- Milhares de ALUs de maior latência
- Pipelines de execução

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

- Caches pequenos (*kilobytes*)
- Sem branch prediction
- Milhares de ALUs de maior latência
- Pipelines de execução
- 114 688 threads concorrentes na arquitetura Pascal

# GRAPHICS PROCESSING UNITS

Hoje são usadas em computação de propósito geral, podem ser chamadas de *General Purpose GPUs* (GPGPUs):

# GRAPHICS PROCESSING UNITS

Hoje são usadas em computação de propósito geral, podem ser chamadas de *General Purpose GPUs* (GPGPUs):

- *OpenGL*: 1992
- *DirectX*: 1995
- **CUDA**: 2007
- *OpenCL*: 2009
- *Vulkan*: 2016

# MODELO DE *HARDWARE*

Taxonomia de Flynn:

- *Single Instruction Multiple Data (SIMD)*

# MODELO DE *HARDWARE*

Taxonomia de Flynn:

- *Single Instruction Multiple Data* (SIMD)
- *Single Instruction Multiple Thread* (SIMT)?

# MODELO DE *HARDWARE*

Taxonomia de Flynn:

- *Single Instruction Multiple Data* (SIMD)
- *Single Instruction Multiple Thread* (SIMT)?

Escalonamento e execução:

- *Streaming Multiprocessor* (SM)

# MODELO DE *HARDWARE*

Taxonomia de Flynn:

- *Single Instruction Multiple Data* (SIMD)
- *Single Instruction Multiple Thread* (SIMT)?

Escalonamento e execução:

- *Streaming Multiprocessor* (SM)
- *Warps*

# MODELO DE *HARDWARE*

Taxonomia de Flynn:

- *Single Instruction Multiple Data* (SIMD)
- *Single Instruction Multiple Thread* (SIMT)?

Escalonamento e execução:

- *Streaming Multiprocessor* (SM)
- *Warps*
- *Grids, blocks e threads*

## MODELO DE *HARDWARE*

Escalonamento de mais alto-nível:

- *Pipelines*
- *Texture Processing Cluster (TPC)*
- *Graphics Processing Cluster (GPC)*

# MODELO DE *HARDWARE*

Escalonamento de mais alto-nível:

- *Pipelines*
- *Texture Processing Cluster (TPC)*
- *Graphics Processing Cluster (GPC)*

Memória:

- Compartilhada: Cache L1, L2; *megabytes* (Nvidia Pascal)
- Global: volátil, GDDR5; *gigabytes*

# MODELO DE *HARDWARE*

Escalonamento de mais alto-nível:

- *Pipelines*
- *Texture Processing Cluster (TPC)*
- *Graphics Processing Cluster (GPC)*

Memória:

- Compartilhada: Cache L1, L2; *megabytes* (Nvidia Pascal)
- Global: volátil, GDDR5; *gigabytes*
- SSD:  **não-volátil; *terabytes***  (AMD SSG Fiji, 2017)

AMD SSG: [anandtech.com/show/10518/amd-announces-radeon-pro-ssg-fiji-with-m2-ssds-onboard](http://anandtech.com/show/10518/amd-announces-radeon-pro-ssg-fiji-with-m2-ssds-onboard) [Acessado em 30/07/16]

## *COMPUTE CAPABILITY E SIMT*

A **Compute Capability** especifica características de arquiteturas *Single Instruction Multiple Thread* (SIMT):

## *COMPUTE CAPABILITY E SIMT*

A **Compute Capability** especifica características de arquiteturas *Single Instruction Multiple Thread* (SIMT):

- *Warp*: Grupo de *threads* executadas em **paralelo**
- *Streaming Multiprocessor* (SM): Cria, escalona e executa *warps*

# COMPUTE CAPABILITY E SIMT

A **Compute Capability** especifica características de arquiteturas *Single Instruction Multiple Thread* (SIMT):

- *Warp*: Grupo de *threads* executadas em **paralelo**
- *Streaming Multiprocessor* (SM): Cria, escalona e executa *warps*
- Múltiplas *warps concorrentes* no mesmo SM

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation) [Acessado em 29/07/16]

## *COMPUTE CAPABILITY E SIMT*

*Warps:*

## *COMPUTE CAPABILITY E SIMT*

*Warps:*

- Unidade de execução e escalonamento
- 32 *threads*

## *COMPUTE CAPABILITY E SIMT*

*Warps:*

- Unidade de execução e escalonamento
- 32 *threads*
- 1 **instrução em comum** por vez

## *COMPUTE CAPABILITY E SIMT*

*Warps:*

- Unidade de execução e escalonamento
- 32 *threads*
- 1 **instrução em comum** por vez
- *Threads ativas*: estão no *branch* atual de execução
- *Threads inativas*: **não** estão no *branch* atual de execução

## *COMPUTE CAPABILITY E SIMT*

*Warps:*

- Unidade de execução e escalonamento
- 32 *threads*
- 1 **instrução em comum** por vez
- *Threads ativas*: estão no *branch* atual de execução
- *Threads inativas*: **não** estão no *branch* atual de execução
- *Threads* fora do *branch* atual executadas **sequencialmente**

# COMPUTE CAPABILITY E SIMT

Warps:

- Unidade de execução e escalonamento
- 32 *threads*
- 1 **instrução em comum** por vez
- *Threads ativas*: estão no *branch* atual de execução
- *Threads inativas*: **não** estão no *branch* atual de execução
- *Threads* fora do *branch* atual executadas **sequencialmente**
- Eficiência: sem divergência de execução dentro de Warps

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation) [Acessado em 29/07/16]

# ARQUITETURA PASCAL GP100

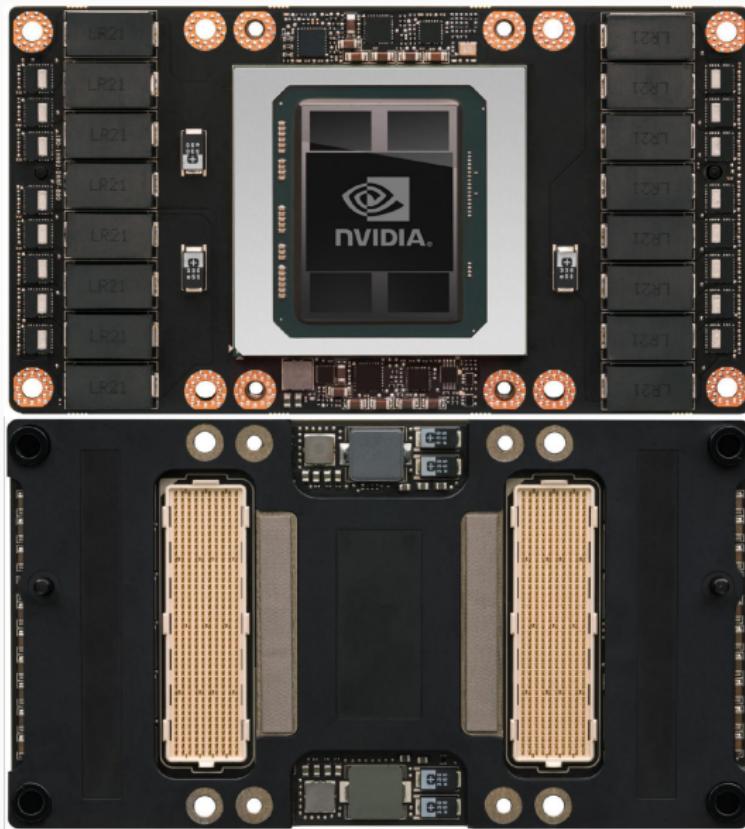


Imagen: [images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf](http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf) [Acessado em 29/07/16]

# ARQUITETURA PASCAL GP100: COMPUTE CAPABILITY 6.0

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

Imagen: [images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf](http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf) [Acessado em 29/07/16]

# ARQUITETURA PASCAL GP100: SM



Imagen: [images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf](http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf) [Acessado em 29/07/16]

# ARQUITETURA PASCAL GP100

Tesla Products	Tesla K40	Tesla M40	Tesla P100
<b>GPU</b>	JK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
<b>SMs</b>	15	24	56
<b>TPCs</b>	15	24	28
<b>FP32 CUDA Cores / SM</b>	192	128	64
<b>FP32 CUDA Cores / GPU</b>	2880	3072	3584
<b>FP64 CUDA Cores / SM</b>	64	4	32
<b>FP64 CUDA Cores / GPU</b>	960	96	1792
<b>Base Clock</b>	745 MHz	948 MHz	1328 MHz
<b>GPU Boost Clock</b>	810/875 MHz	1114 MHz	1480 MHz
<b>Peak FP32 GFLOPs<sup>1</sup></b>	5040	6840	10600
<b>Peak FP64 GFLOPs<sup>1</sup></b>	1680	210	5300
<b>Texture Units</b>	240	192	224
<b>Memory Interface</b>	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
<b>Memory Size</b>	Up to 12 GB	Up to 24 GB	16 GB
<b>L2 Cache Size</b>	1536 KB	3072 KB	4096 KB
<b>Register File Size / SM</b>	256 KB	256 KB	256 KB
<b>Register File Size / GPU</b>	3840 KB	6144 KB	14336 KB
<b>TDP</b>	235 Watts	250 Watts	300 Watts
<b>Transistors</b>	7.1 billion	8 billion	15.3 billion
<b>GPU Die Size</b>	551 mm <sup>2</sup>	601 mm <sup>2</sup>	610 mm <sup>2</sup>
<b>Manufacturing Process</b>	28-nm	28-nm	16-nm FinFET

<sup>1</sup> The GFLOPS in this chart are based on GPU Boost Clocks.

# ARQUITETURA PASCAL GP100

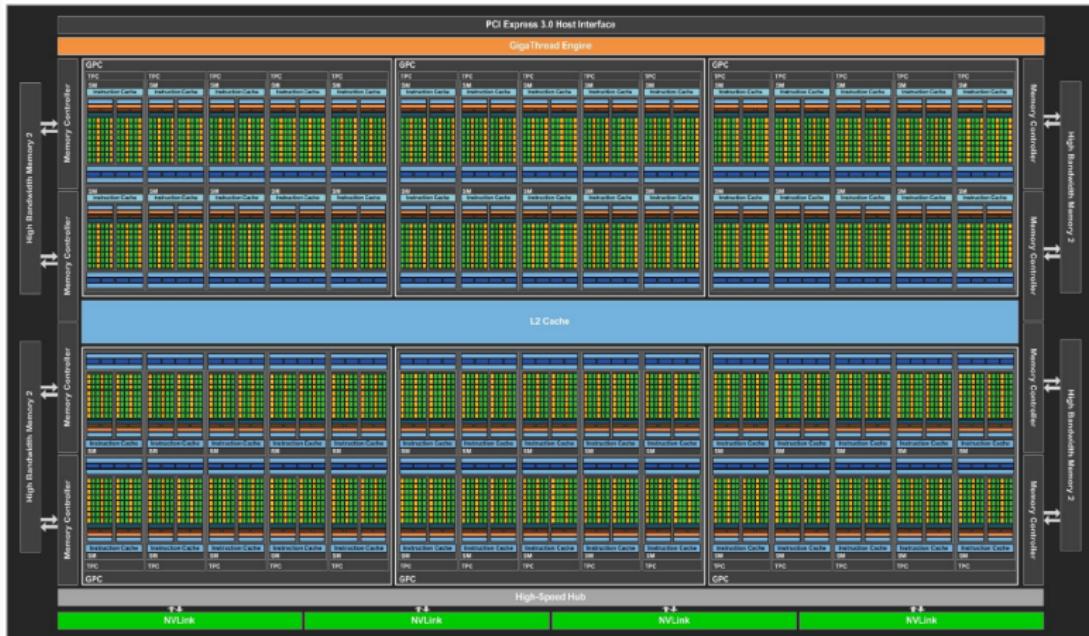


Imagen: [images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf](http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf) [Acessado em 29/07/16]

# PLATAFORMA CUDA

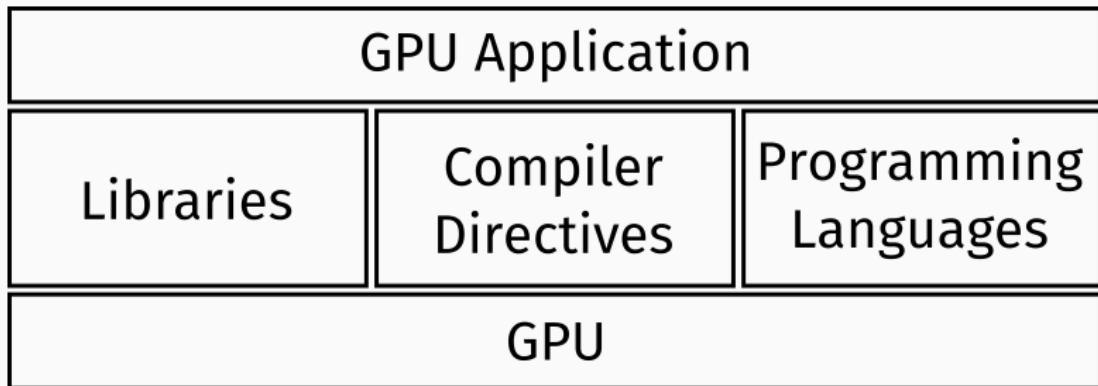


# PLATAFORMA CUDA

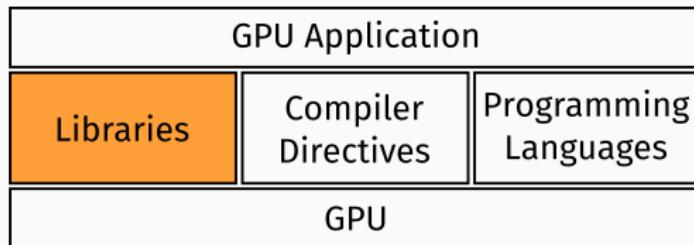


- Plataforma para **computação paralela**
- *Application Programming Interface* (API)
- *CUDA Toolkit*

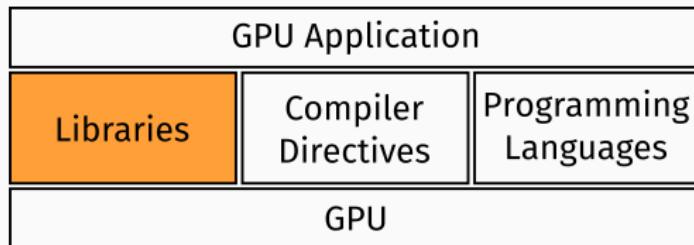
# ACELERAÇÃO POR SOFTWARE



# BIBLIOTECAS

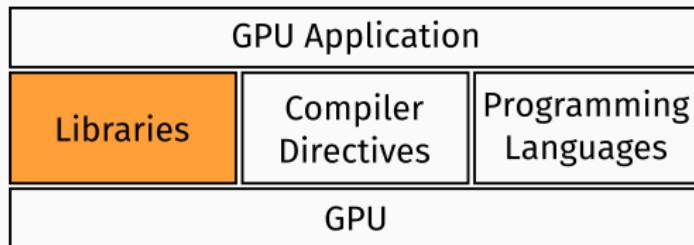


# BIBLIOTECAS



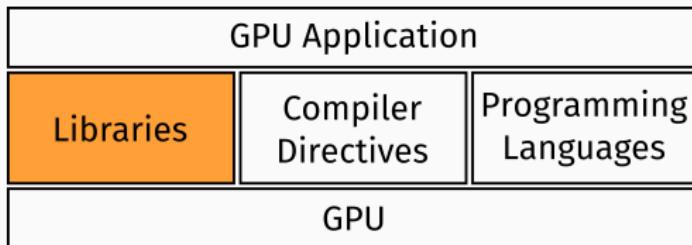
- Fáceis de usar

# BIBLIOTECAS



- Fáceis de usar
- Aceleração *Drop-in*

# BIBLIOTECAS



- Fáceis de usar
- Aceleração *Drop-in*
- Otimizadas por especialistas



# BIBLIOTECAS

Soma de vetores com a biblioteca Thrust:

```
thrust::device_vector<float> device_input1(input_lenght);
thrust::device_vector<float> device_input2(input_lenght);
thrust::device_vector<float> device_output(input_lenght);
```

# BIBLIOTECAS

Soma de vetores com a biblioteca Thrust:

```
thrust::device_vector<float> device_input1(input_lenght);
thrust::device_vector<float> device_input2(input_lenght);
thrust::device_vector<float> device_output(input_lenght);

thrust::copy(host_input1, host_input1 + input_lenght,
            device_input1.begin());

thrust::copy(host_input2, host_input2 + input_lenght,
            device_input2.begin());
```

# BIBLIOTECAS

Soma de vetores com a biblioteca Thrust:

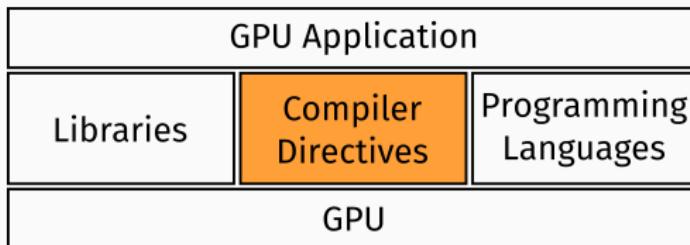
```
thrust::device_vector<float> device_input1(input_lenght);
thrust::device_vector<float> device_input2(input_lenght);
thrust::device_vector<float> device_output(input_lenght);

thrust::copy(host_input1, host_input1 + input_lenght,
            device_input1.begin());

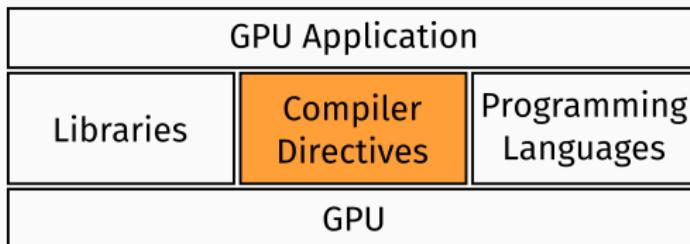
thrust::copy(host_input2, host_input2 + input_lenght,
            device_input2.begin());

thrust::transform(device_input1.begin(),
                 device_input1.end(), device_input2.begin(),
                 device_output.begin(), thrust::plus<float>());
```

# DIRETIVAS DE COMPILAÇÃO

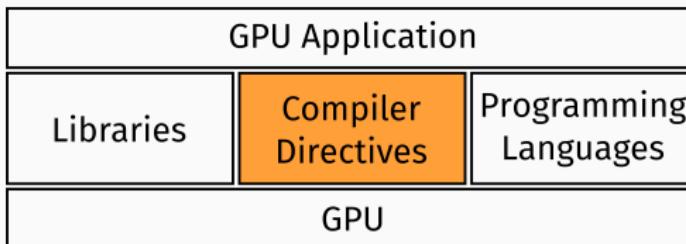


# DIRETIVAS DE COMPILAÇÃO



- Fáceis de usar

# DIRETIVAS DE COMPILAÇÃO



- Fáceis de usar
- Portáveis, mas desempenho depende do compilador

# DIRETIVAS DE COMPILAÇÃO

Soma de vetores com OpenACC:

```
#pragma acc data
    copyin(input1[0:input_length],input2[0:input_length]),
    copyout(output[0:input_length])
```

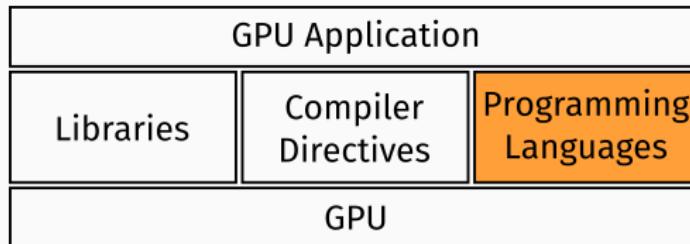
# DIRETIVAS DE COMPILAÇÃO

Soma de vetores com OpenACC:

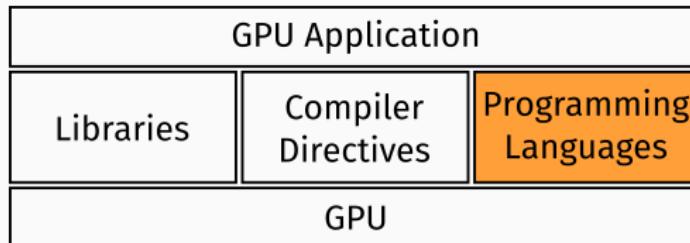
```
#pragma acc data
    copyin(input1[0:input_lenght],input2[0:input_lenght]),
    copyout(output[0:input_lenght])

{
    #pragma acc kernels loop independent
    for(i = 0; i < input_lenght; i++) {
        output[i] = input1[i] + input2[i];
    }
}
```

# LINGUAGENS DE PROGRAMAÇÃO

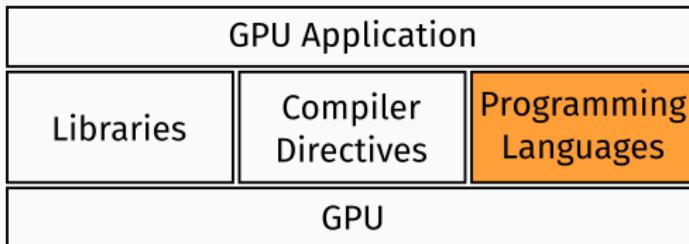


# LINGUAGENS DE PROGRAMAÇÃO



- Melhor desempenho, mas mais difíceis de usar

# LINGUAGENS DE PROGRAMAÇÃO



- Melhor desempenho, mas mais difíceis de usar
- Flexíveis

# LINGUAGENS DE PROGRAMAÇÃO

Implementações de CUDA em:

- C
- Fortran
- C++
- Python
- F#

# CUDA C





Modelo de Programação:



Modelo de Programação:

- Kernels



Modelo de Programação:

- Kernels
- Hierarquia de Threads



Modelo de Programação:

- Kernels
- Hierarquia de Threads
- Hierarquia de Memória



Modelo de Programação:

- Kernels
- Hierarquia de Threads
- Hierarquia de Memória
- Programação Heterogênea

# CUDA C: KERNEL

Kernels:

- Kernels são funções C executadas por threads
- $N$  threads executam  $N$  kernels

# CUDA C: KERNEL

Kernels:

- Kernels são funções C executadas por threads
- $N$  threads executam  $N$  kernels
- Acessam ID de suas threads pela variável threadIdx

# CUDA C: KERNEL

## Kernels:

- Kernels são funções C executadas por threads
- $N$  threads executam  $N$  kernels
- Acessam ID de suas threads pela variável threadIdx
- Definidos usando a palavra-chave \_\_global\_\_
- Seu tipo deve ser void

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: KERNEL

### Kernels:

- Lançados e configurados usando a sintaxe:
  - `kernel_name<<<...>>>( ... );`

## CUDA C: KERNEL

### Kernels:

- Lançados e configurados usando a sintaxe:
  - `kernel_name<<<...>>>( ... );`
- Idealmente, são executados em **paralelo**

## CUDA C: KERNEL

### Kernels:

- Lançados e configurados usando a sintaxe:
  - `kernel_name<<<...>>>( ... );`
- Idealmente, são executados em **paralelo**
- Na prática, o paralelismo depende:
  - Número de *threads* em relação a uma *warp*

# CUDA C: KERNEL

## Kernels:

- Lançados e configurados usando a sintaxe:
  - `kernel_name<<<...>>>( ... );`
- Idealmente, são executados em **paralelo**
- Na prática, o paralelismo depende:
  - Número de *threads* em relação a uma *warp*
  - Coerência entre ramos de execução

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: KERNEL

Escrevendo e lançando (launching) um kernel:

```
#include <cuda_runtime.h>

__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

## CUDA C: KERNEL

Escrevendo e lançando (launching) um kernel:

```
#include <cuda_runtime.h>

__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    ...
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: HIERARQUIA DE *THREADS*

Thread Block:

- Agrupamentos de *Threads*

## CUDA C: HIERARQUIA DE *THREADS*

### Thread Block:

- Agrupamentos de *Threads*
- Tridimensionais:  $D_b = (D_x, D_y, D_z)$

## CUDA C: HIERARQUIA DE *THREADS*

### Thread Block:

- Agrupamentos de *Threads*
- Tridimensionais:  $D_b = (D_x, D_y, D_z)$
- Tamanho **máximo** de 1024 *threads*

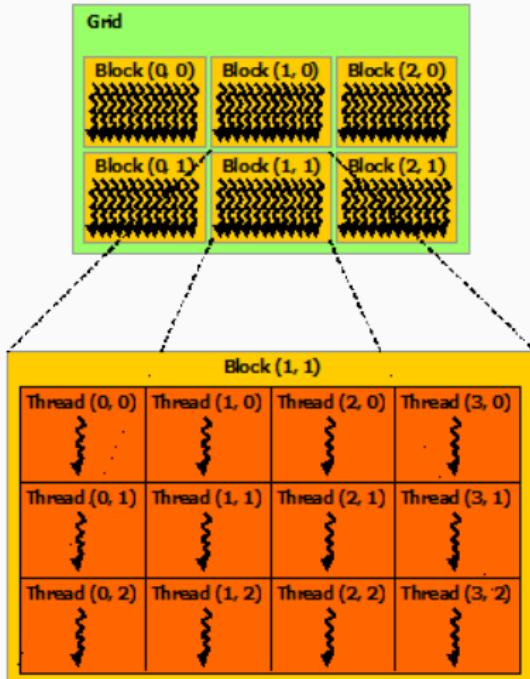
# CUDA C: HIERARQUIA DE *THREADS*

## Thread Block:

- Agrupamentos de *Threads*
- Tridimensionais:  $D_b = (D_x, D_y, D_z)$
- Tamanho **máximo** de 1024 *threads*
- Um **Grid** é um agrupamento tridimensional de *blocks*

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

# CUDA C: HIERARQUIA DE THREADS



Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

# RELEMBRANDO: SM DA ARQUITETURA PASCAL GP100



Imagen: [images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf](http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf) [Acessado em 29/07/16]

## CUDA C: HIERARQUIA DE THREADS

```
__global__ void MatAdd(float A[N][N], float B[N][N], float
C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

## CUDA C: HIERARQUIA DE THREADS

```
__global__ void MatAdd(float A[N][N], float B[N][N], float
C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: HIERARQUIA DE THREADS

```
__global__ void MatAdd(float A[N][N], float B[N][N], float
C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) {
        C[i][j] = A[i][j] + B[i][j];
    }
}
```

## CUDA C: HIERARQUIA DE THREADS

```
__global__ void MatAdd(float A[N][N], float B[N][N], float
C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) {
        C[i][j] = A[i][j] + B[i][j];
    }
}

int main() {
    ...
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N /
                    threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: HIERARQUIA DE THREADS

Sobre a sintaxe de lançamento e configuração `<<< Dg, Db, Ns, S >>>`:

- **dim3** Dg determina dimensão e tamanho do *grid*:

$$Dg.x * Dg.y * Dg.z = \text{numBlocks}$$

## CUDA C: HIERARQUIA DE THREADS

Sobre a sintaxe de lançamento e configuração `<<< Dg, Db, Ns, S >>>`:

- **dim3** `Dg` determina dimensão e tamanho do *grid*:

`Dg.x * Dg.y * Dg.z = numBlocks`

- **dim3** `Db` determina dimensão e tamanho de cada *block*:

`Db.x * Db.y * Db.z = threadsPerBlock`

## CUDA C: HIERARQUIA DE THREADS

Sobre a sintaxe de lançamento e configuração `<<< Dg, Db, Ns, S >>>`:

- **dim3** `Dg` determina dimensão e tamanho do *grid*:

$$Dg.x * Dg.y * Dg.z = \text{numBlocks}$$

- **dim3** `Db` determina dimensão e tamanho de cada *block*:

$$Db.x * Db.y * Db.z = \text{threadsPerBlock}$$

- **size\_t** `Ns` = 0: Bytes extras na memória compartilhada

- **cudaStream\_t** `S` = 0: CUDA stream

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

## CUDA C: HIERARQUIA DE MEMÓRIA

Threads acessam **múltiplos espaços de memória** durante a execução de um *kernel*:

- Local

## CUDA C: HIERARQUIA DE MEMÓRIA

Threads acessam **múltiplos espaços de memória** durante a execução de um *kernel*:

- Local
- Compartilhada com o *block*

## CUDA C: HIERARQUIA DE MEMÓRIA

Threads acessam **múltiplos espaços de memória** durante a execução de um *kernel*:

- Local
- Compartilhada com o *block*
- Global

## CUDA C: HIERARQUIA DE MEMÓRIA

Threads acessam múltiplos espaços de memória durante a execução de um *kernel*:

- Local
- Compartilhada com o *block*
- Global: persistente entre *kernels* da mesma aplicação

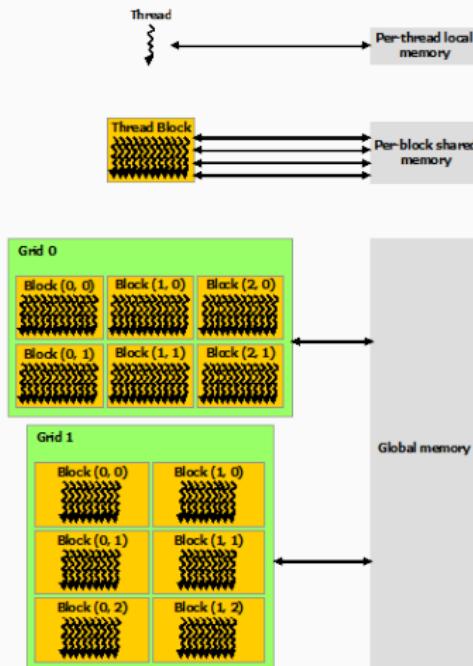
# CUDA C: HIERARQUIA DE MEMÓRIA

Threads acessam **múltiplos espaços de memória** durante a execução de um *kernel*:

- Local
- Compartilhada com o *block*
- Global: persistente entre *kernels* da **mesma aplicação**
- *Read-only*

Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

# CUDA C: HIERARQUIA DE MEMÓRIA



Fonte: [docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model) [Acessado em 29/07/16]

# CUDA C: PROGRAMAÇÃO HETEROGÊNEA

Tarefas do **host**:

- Alocar memória e recursos do **device**
- Mover **dados** (**gargalo!**)

# CUDA C: PROGRAMAÇÃO HETEROGRÊNEA

Tarefas do **host**:

- Alocar memória e recursos do **device**
- Mover **dados** (**gargalo!**)
- Lançar *kernel*

# CUDA C: PROGRAMAÇÃO HETEROGRÉNA

Tarefas do **host**:

- Alocar memória e recursos do **device**
- Mover **dados (gargalo!)**
- Lançar *kernel*
- Mover **resultados (gargalo!)**
- Liberar memória do **device**

# CUDA C: PROGRAMAÇÃO HETEROGRÉNA

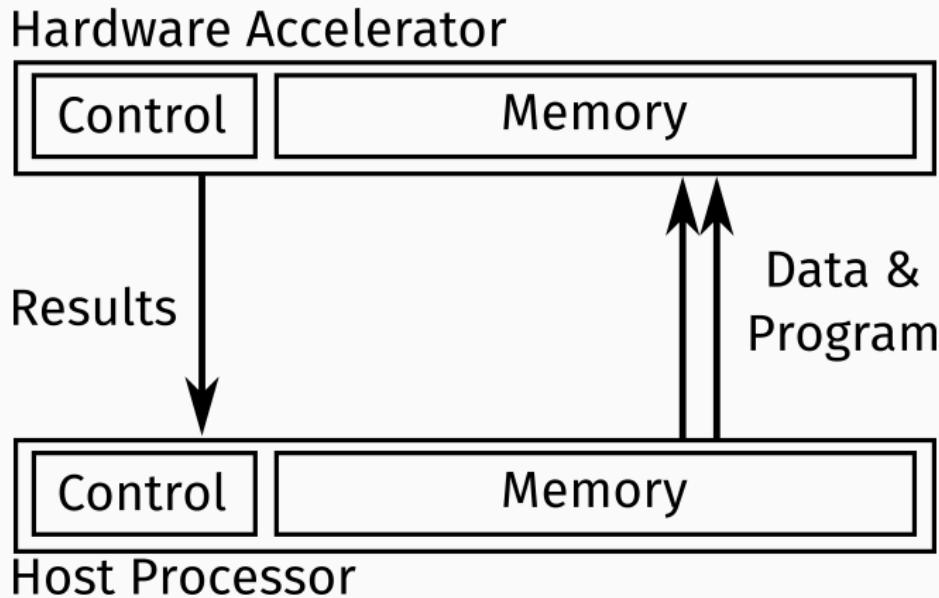
Tarefas do **host**:

- Alocar memória e recursos do **device**
- Mover **dados (gargalo!)**
- Lançar *kernel*
- Mover **resultados (gargalo!)**
- Liberar memória do **device**

Tarefas do **device**:

- Executar *kernel*

# CUDA C: PROGRAMAÇÃO HETEROGRÊNEA



## EXEMPLO: ADIÇÃO DE VETORES

```
--global__ void vectorAdd(const float *A, const float *B,
    float *C, int numElements) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

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

Fonte: [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda) [Acessado em 29/07/16]

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>
```

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>

float *h_A = (float *) malloc(size);
if (h_A == NULL) { ... };

float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess) { ... };
```

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>

float *h_A = (float *) malloc(size);
if (h_A == NULL) { ... };

float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess) { ... };

int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) /
    threadsPerBlock;
```

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>

float *h_A = (float *) malloc(size);
if (h_A == NULL) { ... };

float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess) { ... };

int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) /
    threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
    numElements);

err = cudaGetLastError()
err = cudaDeviceSynchronize()
if (err != cudaSuccess) { ... };
```

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>

float *h_A = (float *) malloc(size);
if (h_A == NULL) { ... };

float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess) { ... };

int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) /
    threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
    numElements);

err = cudaGetLastError()
err = cudaDeviceSynchronize()
if (err != cudaSuccess) { ... };

err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
err = cudaFree(d_A);
if (err != cudaSuccess) { ... };
```

## EXEMPLO: DICA PRÁTICA

Sempre procure por erros!

```
float *d_A = NULL;  
err = cudaMalloc((void **) &d_A, size);  
  
err = cudaGetLastError()  
  
err = cudaDeviceSynchronize()
```

## EXEMPLO: DICA PRÁTICA

Sempre procure por erros!

```
float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);

err = cudaGetLastError()

err = cudaDeviceSynchronize()

if (err != cudaSuccess) {
    fprintf(stderr, "Failed to allocate device vector A
(error code %s)!\n", cudaGetStringError(err));
    exit(EXIT_FAILURE);
}
```

Fonte: [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda) [Acessado em 29/07/16]

# MÃO NA MASSA!

Vamos executar alguns exemplos em CUDA C, disponíveis em  
[github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda):

- `src/cuda-samples/0_Simple/vectorAdd`

# MÃO NA MASSA!

Vamos executar alguns exemplos em CUDA C, disponíveis em [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda):

- `src/cuda-samples/0_Simple/vectorAdd`
- `src/cuda-samples/5_Simulations/fluidsGL`
- `src/cuda-samples/5_Simulations/oceanFFT`
- `src/cuda-samples/5_Simulations/smokeParticles`

# MÃO NA MASSA!

Vamos executar alguns exemplos em CUDA C, disponíveis em [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda):

- `src/cuda-samples/0_Simple/vectorAdd`
- `src/cuda-samples/5_Simulations/fluidsGL`
- `src/cuda-samples/5_Simulations/oceanFFT`
- `src/cuda-samples/5_Simulations/smokeParticles`
- `src/mandelbrot_numba`

# RECURSOS

O *pdf* com as aulas e todo o código fonte estão no [GitHub](#):

- [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda)

Outros recursos:

- CUDA C: [docs.nvidia.com/cuda/cuda-c-programming-guide](https://docs.nvidia.com/cuda/cuda-c-programming-guide)
- CUDA Toolkit: [developer.nvidia.com/cuda-toolkit](https://developer.nvidia.com/cuda-toolkit)
- Guia de Boas Práticas:
  - [docs.nvidia.com/cuda/cuda-c-best-practices-guide](https://docs.nvidia.com/cuda/cuda-c-best-practices-guide)
- GPU Teaching Kit: [syllabus.gputeachingkit.com](https://syllabus.gputeachingkit.com)
- iPython: [ipython.org/notebook.html](https://ipython.org/notebook.html)
- Anaconda: [continuum.io/downloads](https://continuum.io/downloads)

# INTRODUÇÃO À PROGRAMAÇÃO DE GPUs COM A PLATAFORMA CUDA

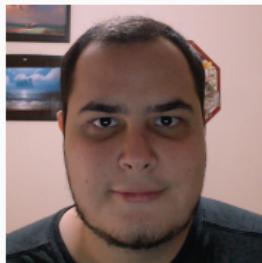
---

Pedro Bruel  
[phrb@ime.usp.br](mailto:phrb@ime.usp.br)  
04 de Agosto de 2016



Instituto de Matemática e Estatística  
Universidade de São Paulo

# SOBRE



Pedro Bruel



Alfredo Goldman

- [phrb@ime.usp.br](mailto:phrb@ime.usp.br)
- [www.ime.usp.br/~phrb](http://www.ime.usp.br/~phrb)
- [github.com/phrb](https://github.com/phrb)
- [gold@ime.usp.br](mailto:gold@ime.usp.br)
- [www.ime.usp.br/~gold](http://www.ime.usp.br/~gold)

# PESQUISA

Meus interesses de **pesquisa**:

- *Autotuning*
- *Stochastic Local Search*
- *Model Based Search*
- GPUs, FPGAs, *cloud*
- Julia Language
- Colaboração! ([phrb@ime.usp.br](mailto:phrb@ime.usp.br))

# PARTE I

1. Introdução
2. Computação Heterogênea
3. GPUs
4. Plataforma CUDA
5. CUDA C

## PARTE II

6. Retomada
7. Compilação de Aplicações CUDA
8. Boas Práticas em Otimização
9. Análise de Aplicações CUDA
10. Otimização de Aplicações CUDA
11. Conclusão

# SLIDES



O *pdf* com as aulas e todo o código fonte estão no [GitHub](#):

- [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda)

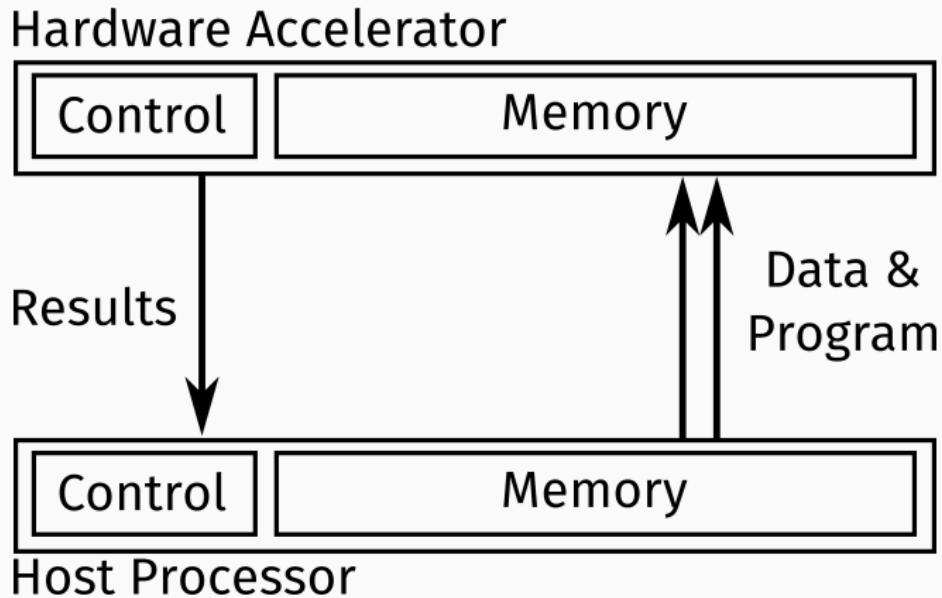
# ACELERAÇÃO POR *HARDWARE*



Uso de **dispositivos** (devices) para acelerar computações aplicadas a grandes conjuntos de dados:

- Associação a um processador **hospedeiro** (host)
- Controle e memória próprios
- Diferem em especialização e configurabilidade
- **GPUs**, DSPs, FPGAs, ASICs

# ACELERAÇÃO POR *HARDWARE*



# ACELERAÇÃO POR HARDWARE

Porcentagem de sistemas com aceleradores na Top500:

## ACCELERATORS/CO-PROCESSORS

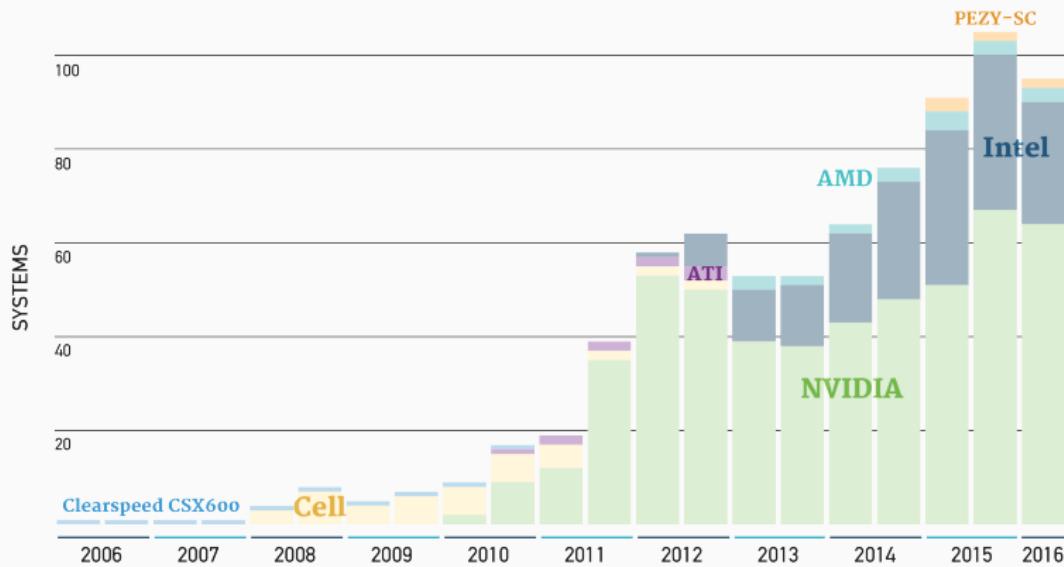


Imagen: [top500.org/lists/2016/06/download/TOP500\\_201606\\_Poster.pdf](http://top500.org/lists/2016/06/download/TOP500_201606_Poster.pdf) [Acessado em 29/07/16]

# COMPUTAÇÃO HETEROGÊNEA



Dados recursos computacionais **heterogêneos**, conjuntos de **dados** e **computações**, como distribuir computações e dados de forma a **otimizar o uso** dos recursos?

Imagen: [olcf.ornl.gov/titan](http://olcf.ornl.gov/titan) [Acessado em 29/07/16]

# GRAPHICS PROCESSING UNITS



Originalmente especializadas em processamento gráfico, trabalham com muitos dados e têm alta vazão:

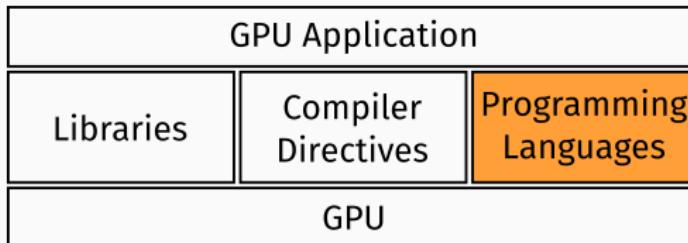
- Caches pequenos (*kilobytes*)
- Sem branch prediction
- Milhares de ALUs de maior latência
- Pipelines de execução
- 114 688 threads concorrentes na arquitetura Pascal

# PLATAFORMA CUDA



- Plataforma para **computação paralela**
- *Application Programming Interface* (API)
- *CUDA Toolkit*

# LINGUAGENS DE PROGRAMAÇÃO



- Melhor desempenho, mas mais difíceis de usar
- Flexíveis



Modelo de Programação:

- Kernels
- Hierarquia de Threads
- Hierarquia de Memória
- Programação Heterogênea

## EXEMPLO: ADIÇÃO DE VETORES

```
#include <cuda_runtime.h>

float *h_A = (float *) malloc(size);
if (h_A == NULL) { ... };

float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
if (err != cudaSuccess) { ... };

int threadsPerBlock = 256;
int blocksPerGrid = (numElements + threadsPerBlock - 1) /
    threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
    numElements);

err = cudaGetLastError()
err = cudaDeviceSynchronize()
if (err != cudaSuccess) { ... };

err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
err = cudaFree(d_A);
if (err != cudaSuccess) { ... };
```

# MÃO NA MASSA!

Vamos executar alguns exemplos em CUDA C, disponíveis em [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda):

- `src/cuda-samples/0_Simple/vectorAdd`
- `src/cuda-samples/5_Simulations/fluidsGL`
- `src/cuda-samples/5_Simulations/oceanFFT`
- `src/cuda-samples/5_Simulations/smokeParticles`
- `src/mandelbrot_numba`

# DICA PRÁTICA

Sempre procure por erros!

```
float *d_A = NULL;
err = cudaMalloc((void **) &d_A, size);

err = cudaGetLastError()

err = cudaDeviceSynchronize()

if (err != cudaSuccess) {
    fprintf(stderr, "Failed to allocate device vector A
(error code %s)!\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

Fonte: [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda) [Acessado em 29/07/16]

# NVIDIA CUDA COMPILER (nvcc)



# NVIDIA CUDA COMPILER (nvcc)



- Sistema proprietário

# NVIDIA CUDA COMPILER (nvcc)



- Sistema proprietário
- Usa o compilador C++ do host

# NVIDIA CUDA COMPILER (nvcc)



- Sistema proprietário
- Usa o compilador C++ do host
- Compila código CUDA para *Parallel Thread Execution* (PTX ISA)

# NVIDIA CUDA COMPILER (nvcc)



- Sistema **proprietário**
- Usa o compilador C++ do *host*
- Compila código CUDA para *Parallel Thread Execution* (**PTX** ISA)
- Código do *host* + Código do *device* → **fatbinary**

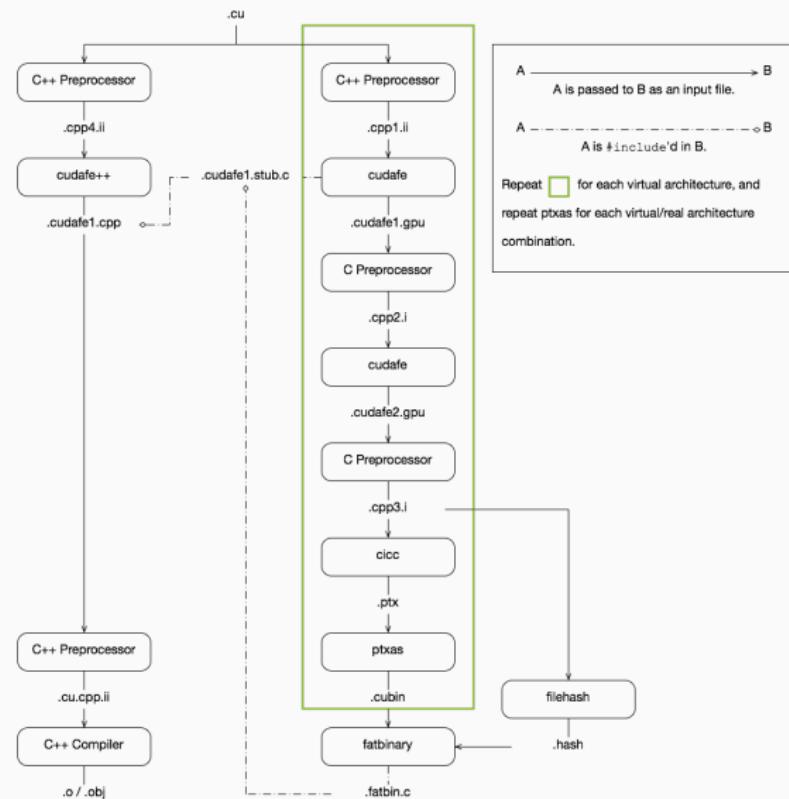
# nvcc: MODELO DE PROGRAMAÇÃO CUDA

- SIMD, SIMT
- *Single Program Multiple Data* (SPMD)

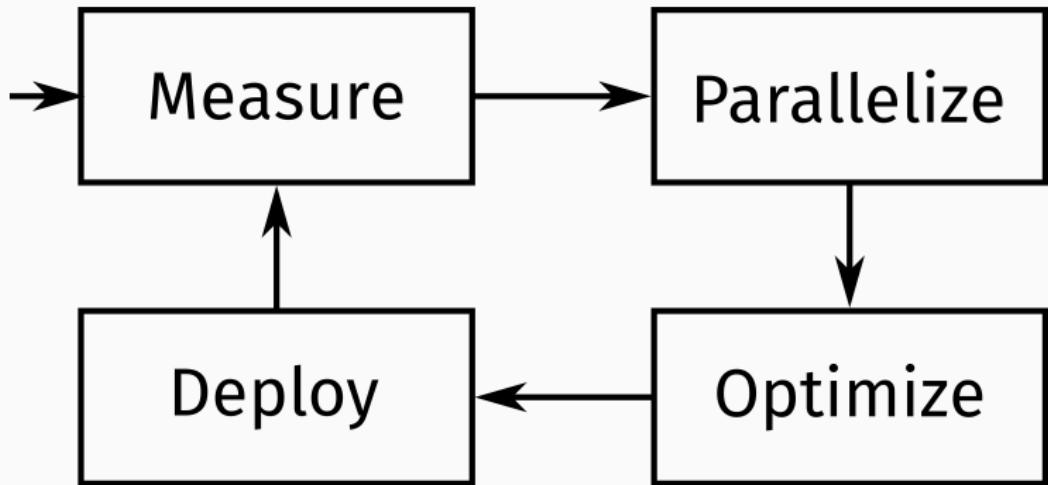
# nvcc: MODELO DE PROGRAMAÇÃO CUDA

- SIMD, SIMT
- *Single Program Multiple Data* (SPMD)
- *Device*: threads paralelas e independentes
- *Host*: não interfere

# nvcc: TRAJETÓRIA DE COMPILAÇÃO

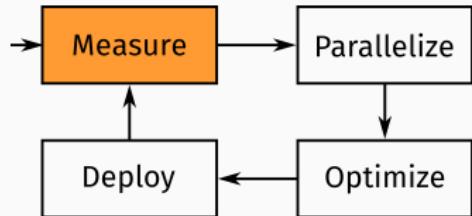


# MEASURE, PARALLELIZE, OPTIMIZE, DEPLOY

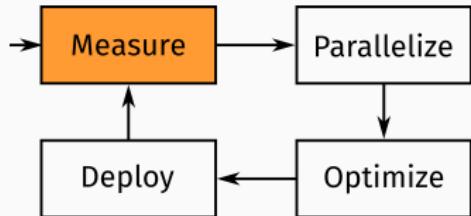


Fonte: [docs.nvidia.com/cuda/cuda-c-best-practices-guide](http://docs.nvidia.com/cuda/cuda-c-best-practices-guide) [Acessado em 29/07/16]

# MEASURE



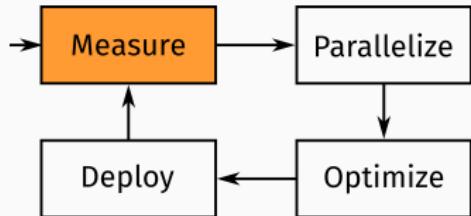
# MEASURE



Dado um programa sequencial:

- Quais funções fazem o trabalho pesado (gargalos)?

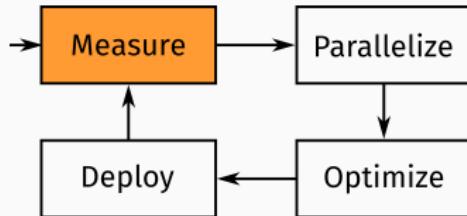
# MEASURE



Dado um programa sequencial:

- Quais funções fazem o trabalho pesado (gargalos)?
- É possível paralelizá-las?

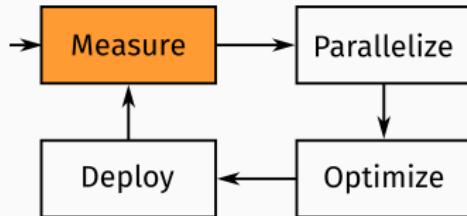
# MEASURE



Dado um **programa sequencial**:

- Quais funções fazem o **trabalho pesado (gargalos)**?
- É possível **paralelizá-las**?
- Como o programa se comporta quando:
  - O trabalho é dividido entre mais processos? (**strong scaling**)

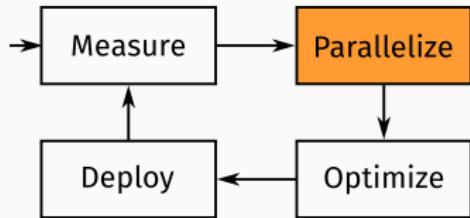
# MEASURE



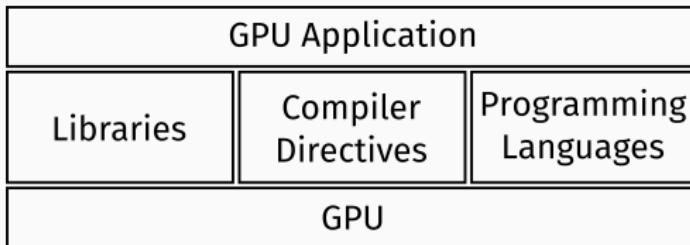
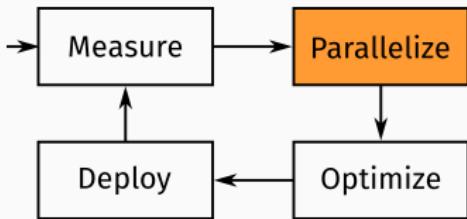
Dado um **programa sequencial**:

- Quais funções fazem o **trabalho pesado (gargalos)**?
- É possível **paralelizá-las**?
- Como o programa se comporta quando:
  - O trabalho é dividido entre mais processos? (**strong scaling**)
  - Há mais trabalho a ser feito? (**weak scaling**)

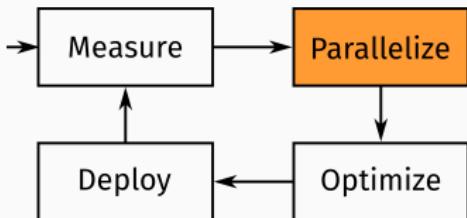
# PARALLELIZE



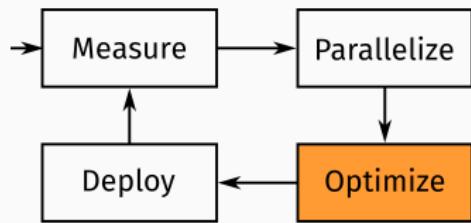
# PARALLELIZE



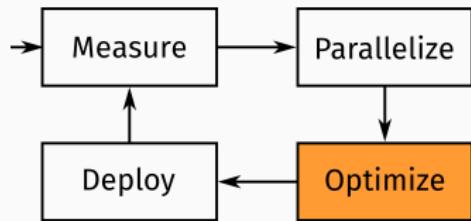
# PARALLELIZE



# OPTIMIZE



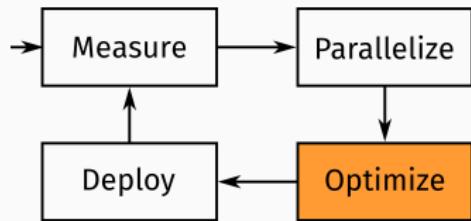
# OPTIMIZE



Otimização:

- Processo **cíclico**

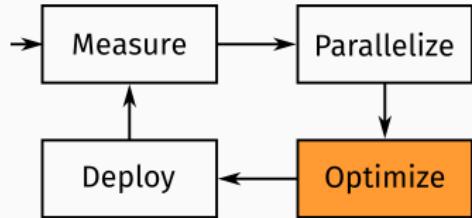
# OPTIMIZE



Otimização:

- Processo **cíclico** e **incremental**

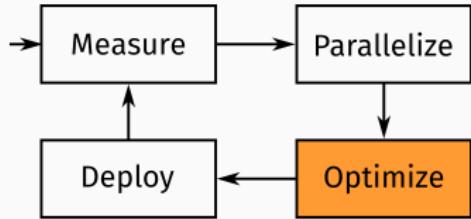
# OPTIMIZE



Otimização:

- Processo **cíclico** e **incremental**
- Aplicável a **diferentes níveis**

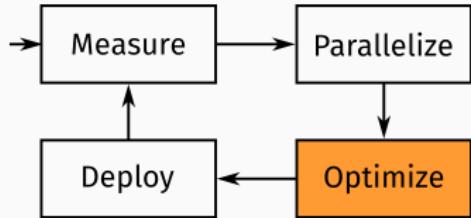
# OPTIMIZE



Otimização:

- Processo **cíclico** e **incremental**
- Aplicável a **diferentes níveis**
- **Ferramentas** são muito importantes

# OPTIMIZE

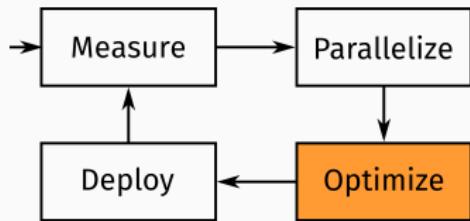


Otimização:

- Processo **cíclico** e **incremental**
- Aplicável a **diferentes níveis**
- **Ferramentas** são muito importantes

Eficiência com algoritmos,

# OPTIMIZE

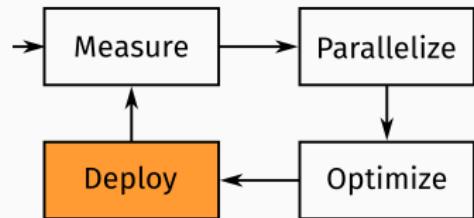


Otimização:

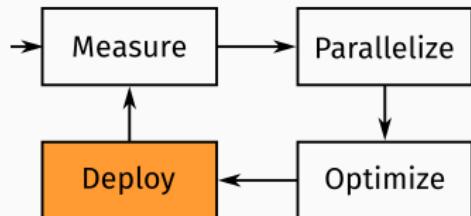
- Processo **cíclico** e **incremental**
- Aplicável a **diferentes níveis**
- **Ferramentas** são muito importantes

Eficiência com algoritmos, desempenho com estruturas de dados

# DEPLOY

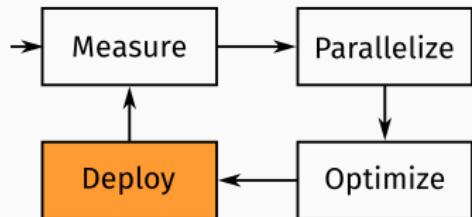


# DEPLOY



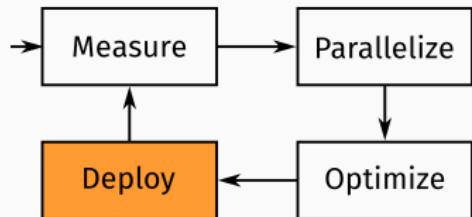
- Preparar a aplicação para a **medição**

# DEPLOY



- Preparar a aplicação para a **medição**
- Otimizações e mudanças **incrementais**

# DEPLOY



- Preparar a aplicação para a **medição**
- Otimizações e mudanças **incrementais**
- O quanto ainda é **possível** otimizar?

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial
- Paralelismo de baixa granularidade

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial
- Paralelismo de baixa granularidade

Análise ou **profiling**:

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial
- Paralelismo de baixa granularidade

Análise ou **profiling**:

- Quais são os **hotspots** (*gargalos, bottlenecks*)?

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial
- Paralelismo de baixa granularidade

Análise ou **profiling**:

- Quais são os **hotspots** (*gargalos, bottlenecks*)?
- Podem ser paralelizados?

# ANÁLISE DE APLICAÇÕES CUDA



Computação paralela e distribuída é cada vez mais importante, mas código legado:

- Sequencial
- Paralelismo de baixa granularidade

Análise ou **profiling**:

- Quais são os **hotspots** (*gargalos, bottlenecks*)?
- Podem ser paralelizados?
- Quais **workloads** são relevantes?

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*
- Memória

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*
- Memória

O que executar em **GPUs**?

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*
- Memória

O que executar em **GPUs**?

- Grande quantidade de **dados**

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*
- Memória

O que executar em **GPUs**?

- Grande quantidade de **dados**
- Operações **independentes**

## *HOST E DEVICE*

Diferenças entre *host* e *device*:

- Modelo de *threading*
- Memória

O que executar em **GPUs**?

- Grande quantidade de **dados**
- Operações **independentes**

Qual o impacto das **transferências de dados**?

# *PROFILING*

Monitoramento de código **em tempo de execução**, obtendo informação para **otimizar o código**

# PROFILING

Monitoramento de código **em tempo de execução**, obtendo informação para **otimizar o código**

**Instrumentação** do Código:

- Captura de **eventos**

# PROFILING

Monitoramento de código **em tempo de execução**, obtendo informação para **otimizar o código**

**Instrumentação** do Código:

- Captura de **eventos**
- Geração de **dados**

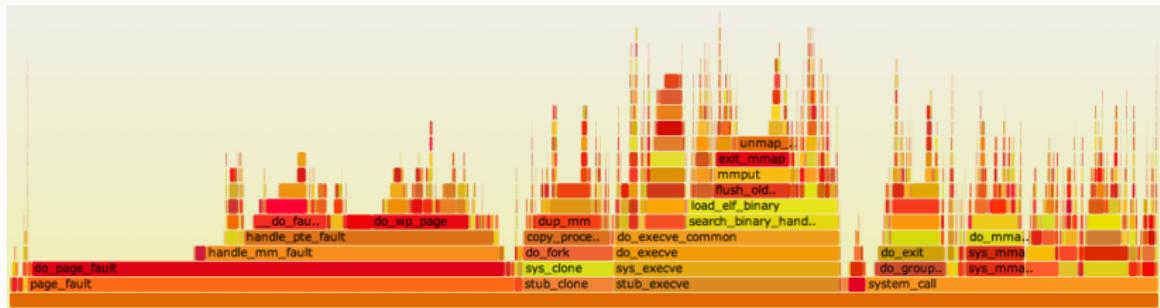
# PROFILING

Monitoramento de código **em tempo de execução**, obtendo informação para **otimizar o código**

**Instrumentação** do Código:

- Captura de **eventos**
- Geração de **dados**
- Ferramentas

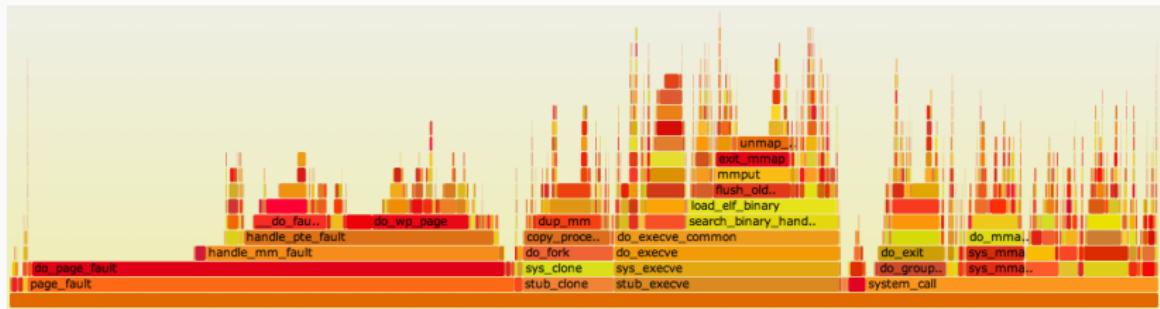
# PROFILING: FLAMEGRAPHS



Facilitam a análise de:

- Bottlenecks, hotspots, ou gargalos de desempenho

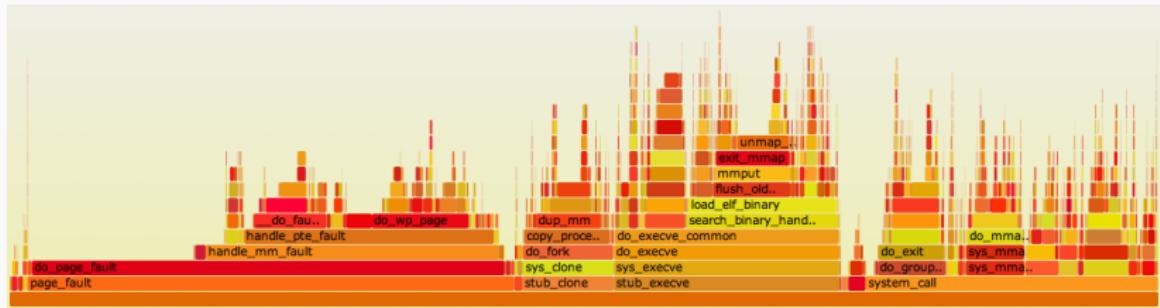
# PROFILING: FLAMEGRAPHS



Facilitam a análise de:

- Bottlenecks, hotspots, ou gargalos de desempenho
- Frequência e duração de **chamadas de função**: On-CPU

# PROFILING: FLAMEGRAPHS



Facilitam a análise de:

- Bottlenecks, hotspots, ou gargalos de desempenho
- Frequência e duração de **chamadas de função**: *On-CPU*
- Tempo **ocioso** (*espera*): *Off-CPU*

## PROFILING: nvprof

*Profiler de linha de comando*, permite a coleta de eventos relacionados a CUDA:

## PROFILING: nvprof

*Profiler de linha de comando*, permite a coleta de eventos relacionados a CUDA:

- CPU e GPU

## PROFILING: nvprof

Profiler de **linha de comando**, permite a coleta de eventos relacionados a CUDA:

- CPU e GPU
- Execução do *kernel*

## PROFILING: nvprof

Profiler de **linha de comando**, permite a coleta de eventos relacionados a CUDA:

- CPU e GPU
- Execução do *kernel*
- Uso e transferência de memória

## PROFILING: nvprof

Profiler de **linha de comando**, permite a coleta de eventos relacionados a CUDA:

- CPU e GPU
- Execução do *kernel*
- Uso e transferência de memória
- CUDA API

# PROFILING: nvprof

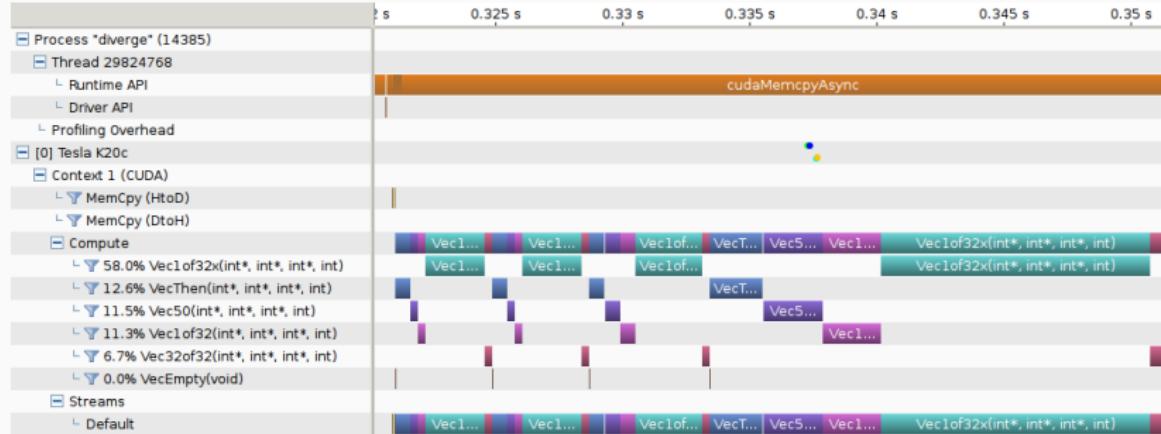
Profiler de **linha de comando**, permite a coleta de eventos relacionados a CUDA:

- CPU e GPU
- Execução do *kernel*
- Uso e transferência de memória
- CUDA API

Gera dados para **posterior visualização**

Fonte: [docs.nvidia.com/cuda/profiler-users-guide](http://docs.nvidia.com/cuda/profiler-users-guide) [Acessado em 29/07/16]

# PROFILING: nvvp

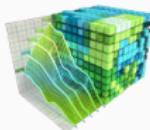


Fonte: [docs.nvidia.com/cuda/profiler-users-guide](http://docs.nvidia.com/cuda/profiler-users-guide) [Acessado em 29/07/16]

# PROFILING: FERRAMENTAS



NVIDIA® Nsight™



NVIDIA Visual Profiler



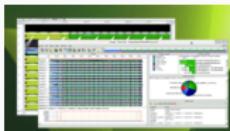
TAU Performance System®



VampirTrace



The PAPI CUDA Component



The NVIDIA CUDA Profiling Tools Interface

Fonte: [developer.nvidia.com/performance-analysis-tools](http://developer.nvidia.com/performance-analysis-tools) [Acessado em 29/07/16]

## *DEBUGGING*

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

# *DEBUGGING*

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

**Instrumentação do Código:**

- Execução **passo-a-passo**

# *DEBUGGING*

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

**Instrumentação do Código:**

- Execução **passo-a-passo**
- **Breakpoints**

# *DEBUGGING*

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

**Instrumentação do Código:**

- Execução **passo-a-passo**
- **Breakpoints**
- Diferentes **threads**

# *DEBUGGING*

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

**Instrumentação do Código:**

- Execução **passo-a-passo**
- **Breakpoints**
- Diferentes **threads**
- CUDA **gdb** e **memcheck**

# DEBUGGING

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

Instrumentação do Código:

- Execução **passo-a-passo**
- **Breakpoints**
- Diferentes **threads**
- CUDA **gdb** e **memcheck**

Facilitam a solução de:

- **Vazamentos** de memória

# DEBUGGING

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

Instrumentação do Código:

- Execução **passo-a-passo**
- **Breakpoints**
- Diferentes **threads**
- CUDA **gdb** e **memcheck**

Facilitam a solução de:

- **Vazamentos** de memória
- Problemas com **fluxo de controle**

# DEBUGGING

Monitoramento de código **em tempo de execução**, obtendo informação para **consertar erros (bugs)**

Instrumentação do Código:

- Execução **passo-a-passo**
- **Breakpoints**
- Diferentes **threads**
- CUDA **gdb** e **memcheck**

Facilitam a solução de:

- **Vazamentos** de memória
- Problemas com **fluxo de controle**
- ...

# OTIMIZAÇÃO DE APLICAÇÕES CUDA



Diferentes níveis de abstração e prioridade:

# OTIMIZAÇÃO DE APLICAÇÕES CUDA



Diferentes níveis de abstração e prioridade:

- Memória

# OTIMIZAÇÃO DE APLICAÇÕES CUDA



Diferentes níveis de abstração e prioridade:

- Memória
- Fluxo de controle

# OTIMIZAÇÃO DE APLICAÇÕES CUDA



Diferentes níveis de abstração e prioridade:

- Memória
- Fluxo de controle
- Configuração de execução

# OTIMIZAÇÃO DE APLICAÇÕES CUDA



Diferentes níveis de abstração e prioridade:

- Memória
- Fluxo de controle
- Configuração de execução
- Instruções

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações
  - Criar e destruir estruturas de dados intermediárias

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações
  - Criar e destruir estruturas de dados intermediárias
  - Acessos agrupados (coalescentes) à memória

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações
  - Criar e destruir estruturas de dados intermediárias
  - Acessos agrupados (coalescentes) à memória
- No *host*:

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações
  - Criar e destruir estruturas de dados intermediárias
  - Acessos agrupados (coalescentes) à memória
- No *host*:
  - Delegar execução de computações

# OTIMIZAÇÕES DE MEMÓRIA

São as mais importantes para atingir alto desempenho

Objetivos:

- Maximizar largura de banda (*bandwidth*)
- Minimizar transferências entre *device* e *host*

Boas práticas:

- No *device*:
  - Priorizar execução de computações
  - Criar e destruir estruturas de dados intermediárias
  - Acessos agrupados (coalescentes) à memória
- No *host*:
  - Delegar execução de computações
  - Minimizar o número de transferências de dados

# OTIMIZAÇÕES DE FLUXO DE CONTROLE

Alta prioridade para atingir alto desempenho

# OTIMIZAÇÕES DE FLUXO DE CONTROLE

Alta prioridade para atingir alto desempenho

Objetivos:

- Evitar divergência de fluxo de controle dentro de warps

# OTIMIZAÇÕES DE FLUXO DE CONTROLE

Alta prioridade para atingir alto desempenho

Objetivos:

- Evitar divergência de fluxo de controle dentro de warps
- **if, switch, do, for, while**

# OTIMIZAÇÕES DE FLUXO DE CONTROLE

Alta prioridade para atingir alto desempenho

Objetivos:

- Evitar divergência de fluxo de controle dentro de warps
- `if, switch, do, for, while`
- Fazer bom uso do `threadIdx`

# OTIMIZAÇÕES DE CONFIGURAÇÃO DE EXECUÇÃO

Como aproveitar totalmente o potencial do **device**?

# OTIMIZAÇÕES DE CONFIGURAÇÃO DE EXECUÇÃO

Como aproveitar totalmente o potencial do **device**?

- Maximizar a **ocupância**

# OTIMIZAÇÕES DE CONFIGURAÇÃO DE EXECUÇÃO

Como aproveitar totalmente o potencial do **device**?

- Maximizar a **ocupância**
- Escolher bem os **blocks** e **threads**

# OTIMIZAÇÕES DE CONFIGURAÇÃO DE EXECUÇÃO

Como aproveitar totalmente o potencial do **device**?

- Maximizar a **ocupância**
- Escolher bem os **blocks** e **threads**
- Execução independente de **kernels**

# OTIMIZAÇÕES DE INSTRUÇÕES

Baixa prioridade para atingir alto desempenho:

# OTIMIZAÇÕES DE INSTRUÇÕES

Baixa prioridade para atingir alto desempenho:

- Baixo nível

# OTIMIZAÇÕES DE INSTRUÇÕES

Baixa prioridade para atingir alto desempenho:

- Baixo nível
- Aplicadas a hotspots

# OTIMIZAÇÕES DE INSTRUÇÕES

Baixa prioridade para atingir alto desempenho:

- Baixo nível
- Aplicadas a **hotspots**
- Feitas por **experts**
- Após **exaurir** outras possibilidades de otimização

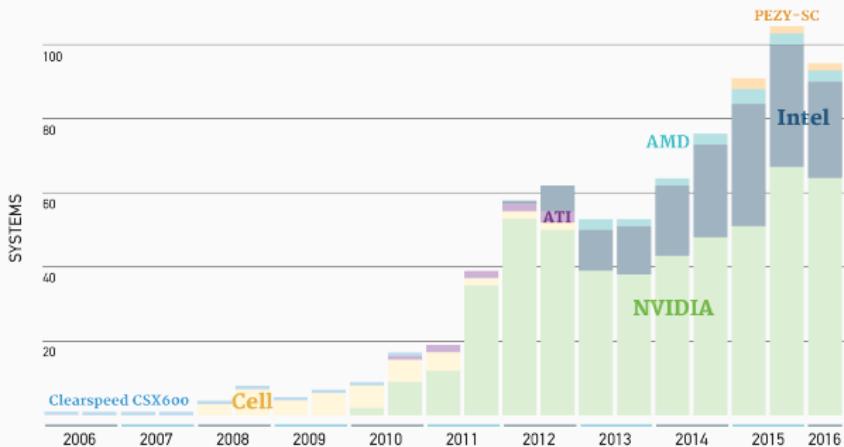
# OTIMIZAÇÕES DE INSTRUÇÕES

Baixa prioridade para atingir alto desempenho:

- Baixo nível
- Aplicadas a **hotspots**
- Feitas por **experts**
- Após **exaurir** outras possibilidades de otimização
- “**Escovar bits**” ☺

# CONCLUSÃO

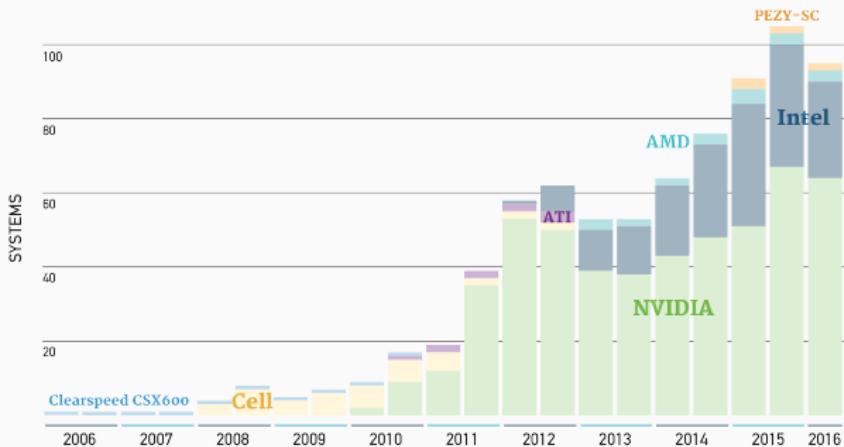
## ACCELERATORS/CO-PROCESSORS



Atingir alto desempenho através do uso de aceleradores

# CONCLUSÃO

## ACCELERATORS/CO-PROCESSORS



Atingir alto desempenho através do uso de aceleradores (GPUs)

# CONCLUSÃO



# CONCLUSÃO



Custo de implementação, análise e otimização

# CONCLUSÃO



Custo de implementação, análise e otimização

Ferramentas da plataforma CUDA

# RECURSOS

O *pdf* com as aulas e todo o código fonte estão no [GitHub](#):

- [github.com/phrb/intro-cuda](https://github.com/phrb/intro-cuda)

Outros recursos:

- CUDA C: [docs.nvidia.com/cuda/cuda-c-programming-guide](https://docs.nvidia.com/cuda/cuda-c-programming-guide)
- CUDA Toolkit: [developer.nvidia.com/cuda-toolkit](https://developer.nvidia.com/cuda-toolkit)
- Guia de Boas Práticas:
  - [docs.nvidia.com/cuda/cuda-c-best-practices-guide](https://docs.nvidia.com/cuda/cuda-c-best-practices-guide)
- GPU Teaching Kit: [syllabus.gputeachingkit.com](https://syllabus.gputeachingkit.com)
- iPython: [ipython.org/notebook.html](https://ipython.org/notebook.html)
- Anaconda: [continuum.io/downloads](https://continuum.io/downloads)

# INTRODUÇÃO À PROGRAMAÇÃO DE GPUs COM A PLATAFORMA CUDA

---

Pedro Bruel  
[phrb@ime.usp.br](mailto:phrb@ime.usp.br)  
04 de Agosto de 2016



Instituto de Matemática e Estatística  
Universidade de São Paulo