

Introdução à Programação de GPUs com a Plataforma CUDA

Pedro Bruel

phrb@ime.usp.br

21 de Maio de 2021



Instituto de Matemática e Estatística
Universidade de São Paulo

Introdução

Parte I

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

Parte II

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

Computação Heterogênea

Aceleração por *Hardware*



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

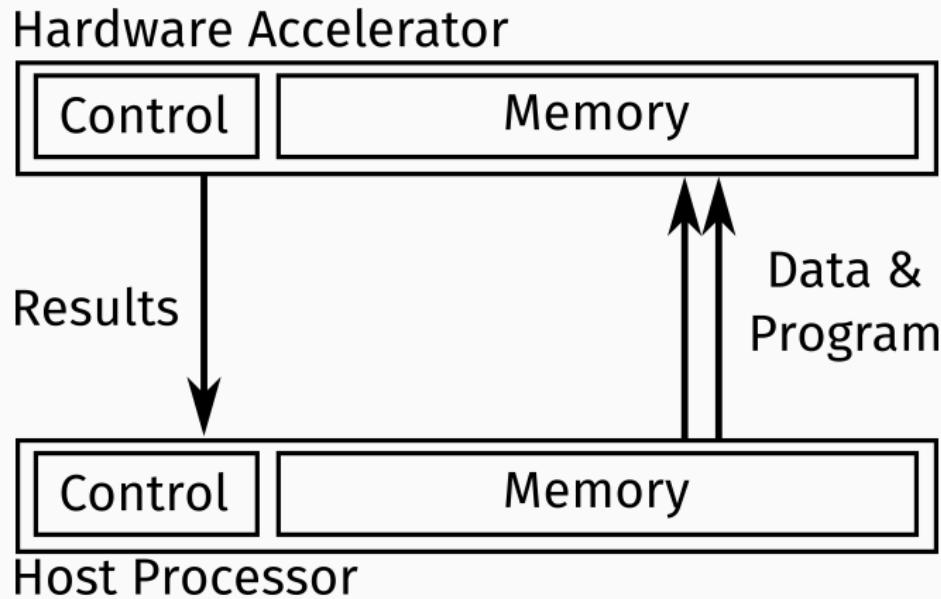
- Associação a um processador **hospedeiro** (*host*)
- **Controle e memória** próprios
- Diferem em **especialização e configurabilidade**
- **GPUs, DSPs, FPGAs, ASICs**

Aceleração por *Hardware*

Casos de uso:

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

Aceleração por *Hardware*



Aceleração por Hardware

Sistemas com aceleradores na *Top500*:

ACCELERATORS/CO-PROCESSORS

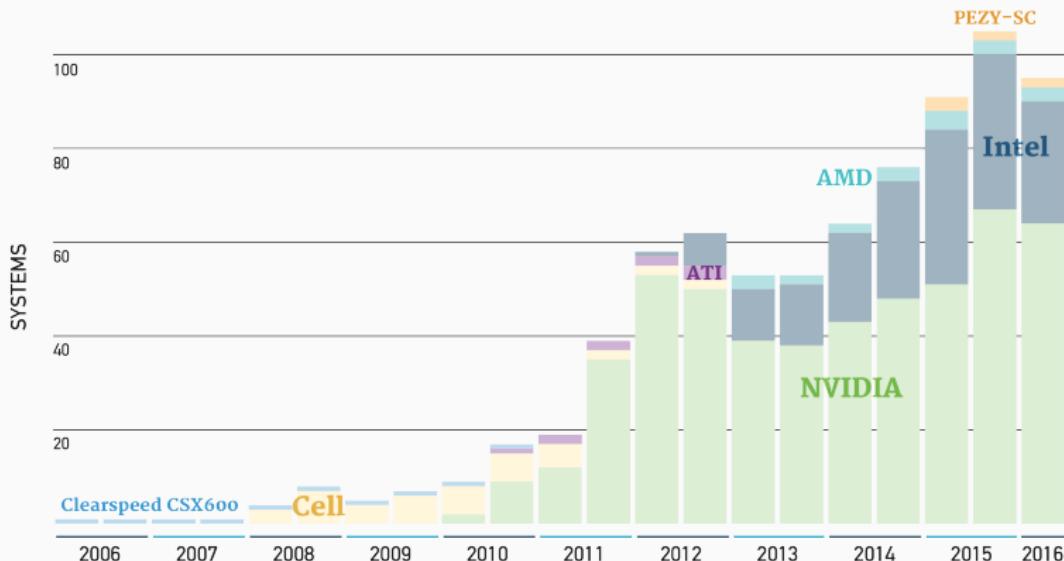
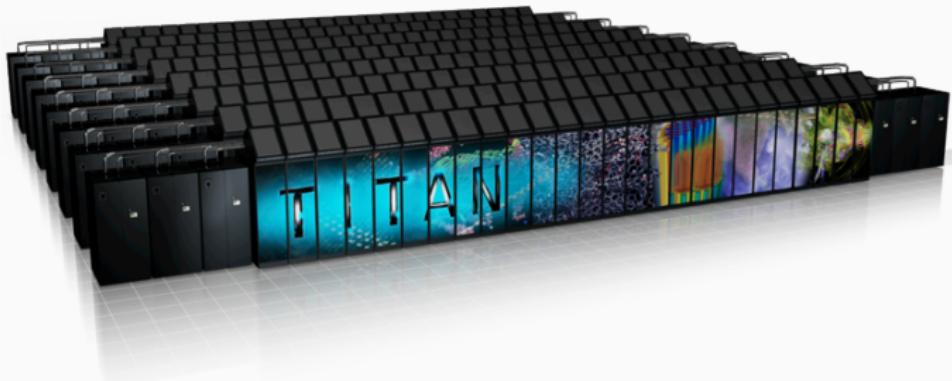


Imagem: top500.org/lists/2016/06/download/TOP500_201606_Poster.pdf [Acessado em 29/07/16]

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 [Acessado em 29/07/16]

Computação Heterogênea

Recursos computacionais **heterogêneos**:

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

Computação Heterogênea

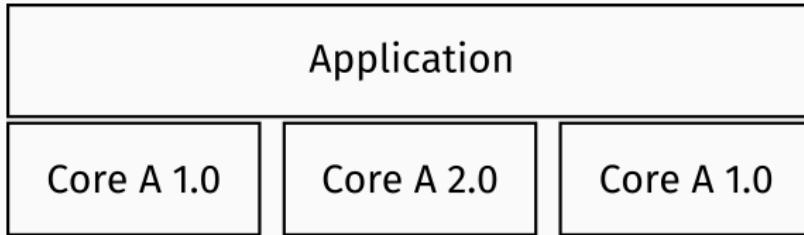
Conjuntos de **dados**:

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

Modelos de programação (**computações**):

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

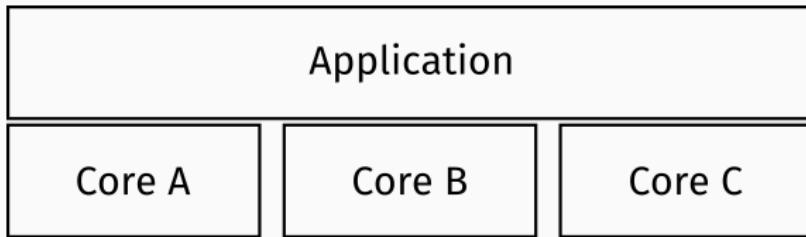
Computação Heterogênea: Escalabilidade



Escalabilidade significa não perder desempenho executando em:

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

Computação Heterogênea: Portabilidade

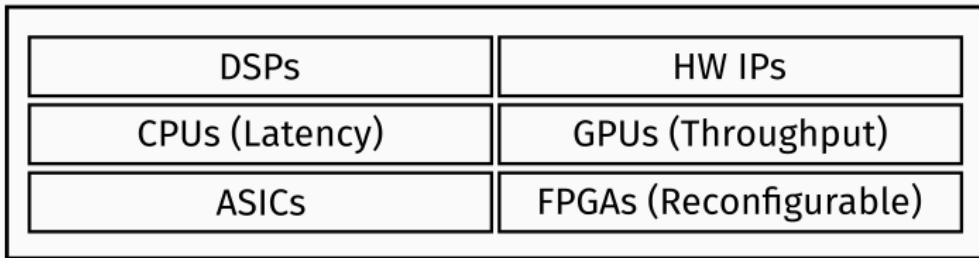


Portabilidade significa não perder desempenho executando em diferentes:

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

Computação Heterogênea

Cloud Service



Results

Data &
Program

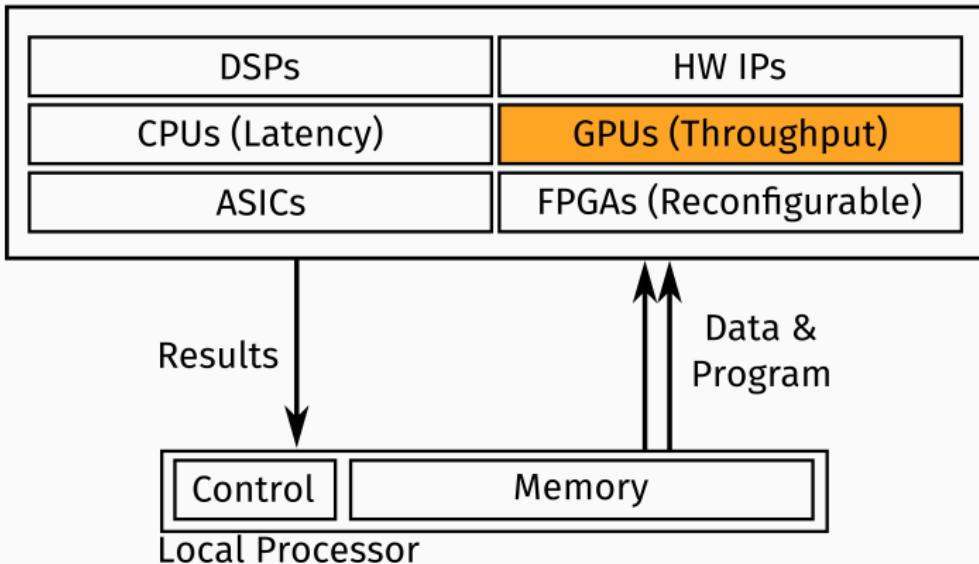
Control

Memory

Local Processor

Computação Heterogênea

Cloud Service



GPUs

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):

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

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)

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 [Acessado em 30/07/16]

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 [Acessado em 29/07/16]

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 [Acessado em 29/07/16]

Arquitetura Pascal GP100

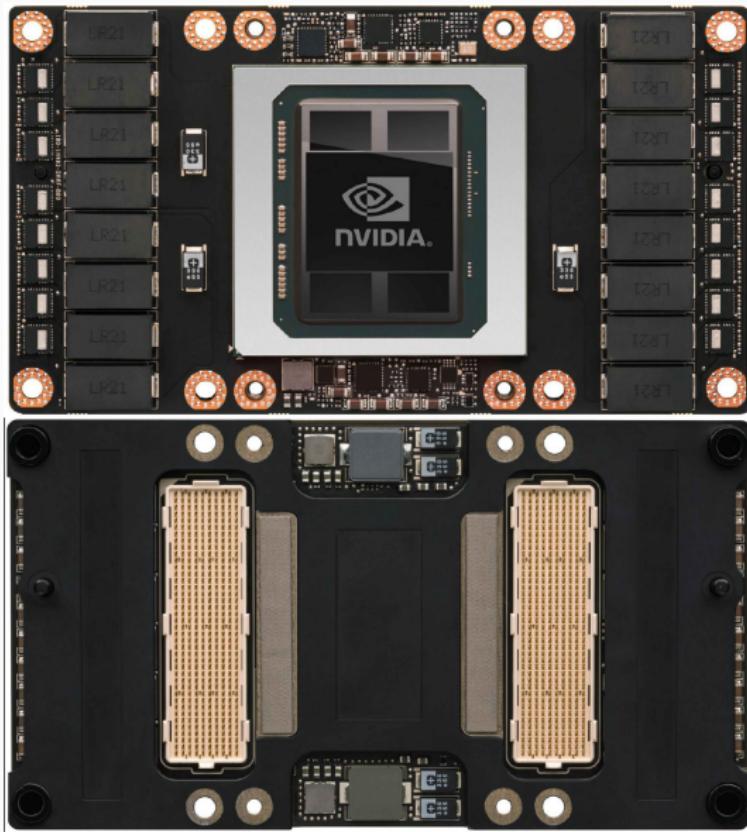


Imagen: images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf [Acessado em 29/07/16]

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 [Acessado em 29/07/16]

Arquitetura Pascal GP100: SM



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

¹ 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 [Acessado em 29/07/16]

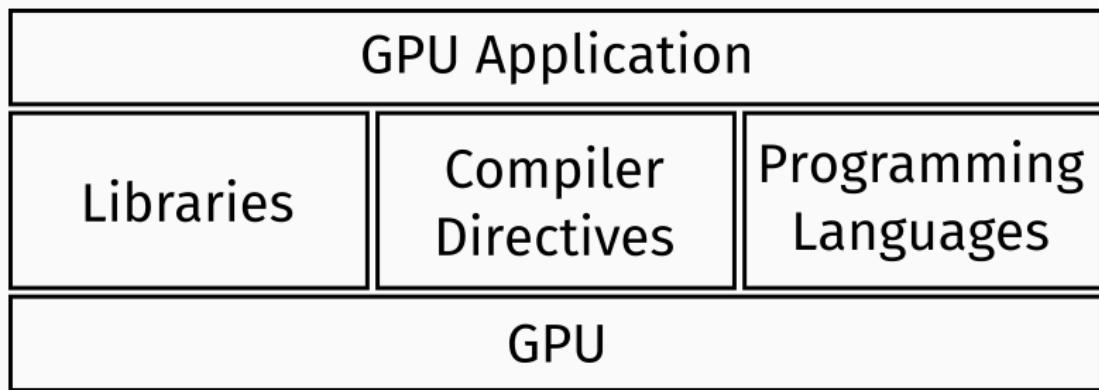
Plataforma CUDA

Plataforma CUDA

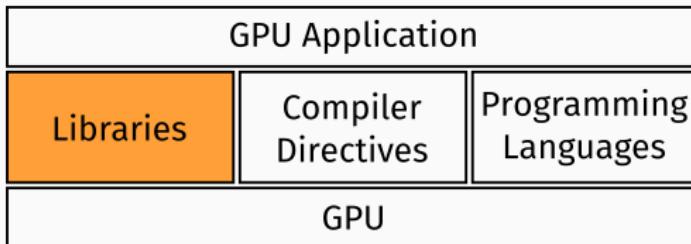


- Plataforma para **computação paralela**
- *Application Programming Interface* (API)
- *CUDA Toolkit*

Aceleração por *Software*



Bibliotecas



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

Bibliotecas



AmgX



cuDNN



cuFFT



IndeX Framework



nvGRAPH



GIE



NPP



FFmpeg



CHOLMOD



CULA Tools



MAGMA



cuSOLVER



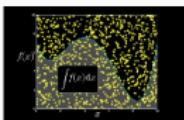
cuSPARSE



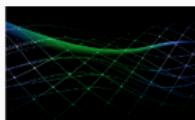
cuBLAS



ArrayFire



cuRAND



CUDA Math Library



Thrust

IMSL Fortran Numerical Library
RogueWave Software

NVBIO

Bibliotecas

Soma de vetores com a biblioteca **Thrust**:

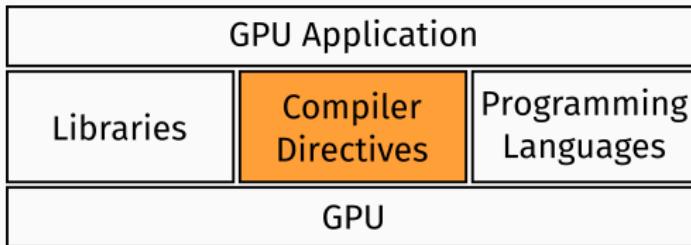
```
thrust::device_vector<float> device_input1(input_length);
thrust::device_vector<float> device_input2(input_length);
thrust::device_vector<float> device_output(input_length);
```

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

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

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

Diretivas de Compilação



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

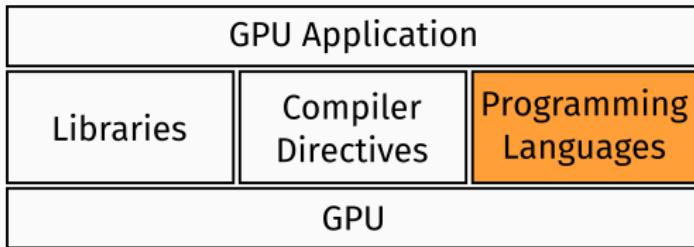
Diretivas de Compilação

Soma de vetores com OpenACC:

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

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

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:

- 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
- 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 [Acessado em 29/07/16]

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 [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];
}

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

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

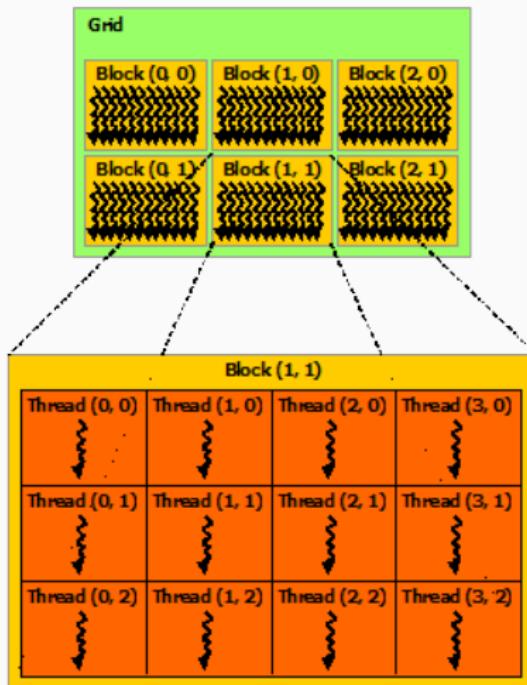
CUDA C: Hierarquia de *Threads*

Thread Block:

- Agrupamentos de *Threads*
- 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 [Acessado em 29/07/16]

CUDA C: Hierarquia de *Threads*



Fonte: 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 [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];
}

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 [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];
    }
}
```

```
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 [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 = numBlocks
```

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

```
Db.x * Db.y * Db.z = 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 [Acessado em 29/07/16]

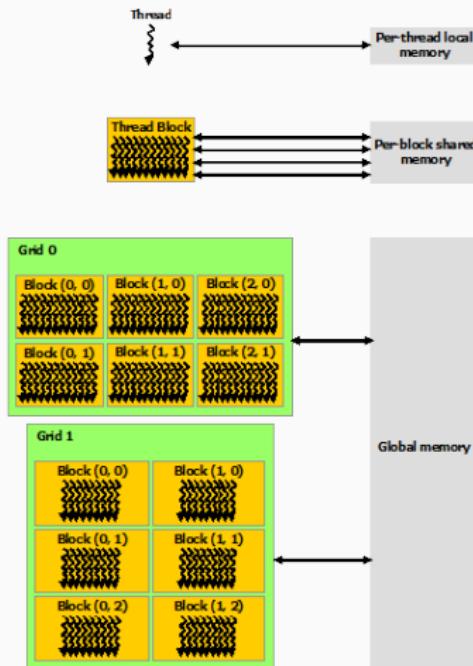
CUDA C: Hierarquia de Memória

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 [Acessado em 29/07/16]

CUDA C: Hierarquia de Memória



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

CUDA C: Programação Heterogênea

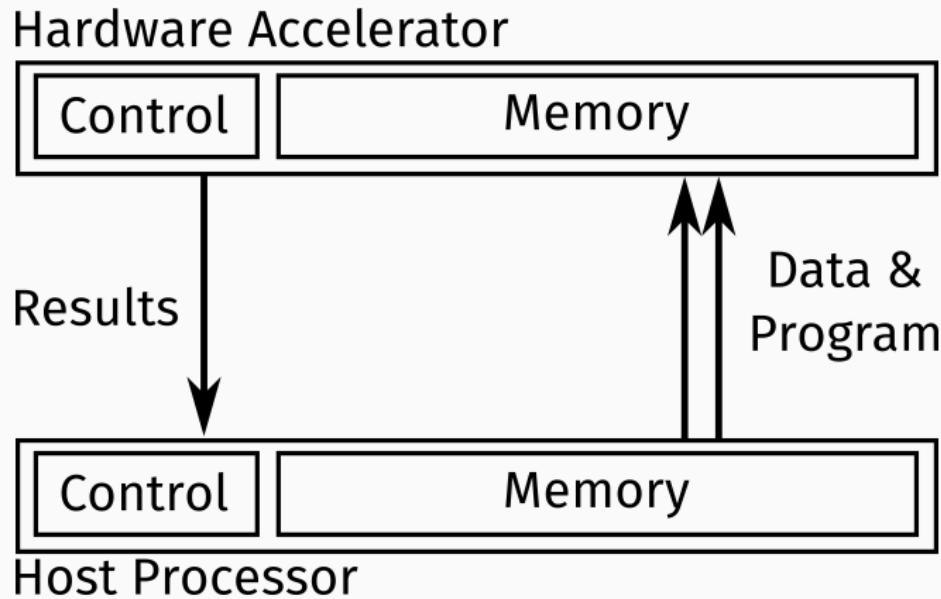
Tarefas do **host**:

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

Tarefas do **device**:

- Executar *kernel*

CUDA C: Programação Heterogênea



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 [Acessado em 29/07/16]

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()

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 [Acessado em 29/07/16]

Mão na massa!

Vamos executar alguns exemplos em CUDA C, disponíveis em
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

Outros recursos:

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

Introdução à Programação de GPUs com a Plataforma CUDA

Pedro Bruel

phrb@ime.usp.br

21 de Maio de 2021



Instituto de Matemática e Estatística
Universidade de São Paulo

Retomada

Parte I

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

Parte II

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



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:

- `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()

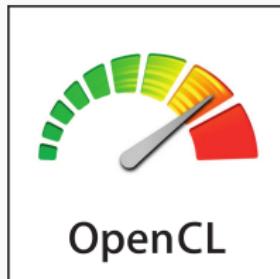
err = cudaDeviceSynchronize()

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 [Acessado em 29/07/16]

Introdução a OpenCL

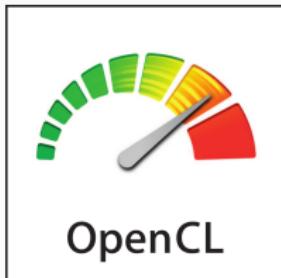
Introdução a OpenCL



OpenCL é um **framework** para programação **paralela e heterogênea**:

- Padrão **aberto**
- Mantido pelo **Khronos Group**
- Implementado por **diferentes empresas**
- CPUs, **GPUs**, DSPs, FPGAs, ...

Modelo de Programação OpenCL

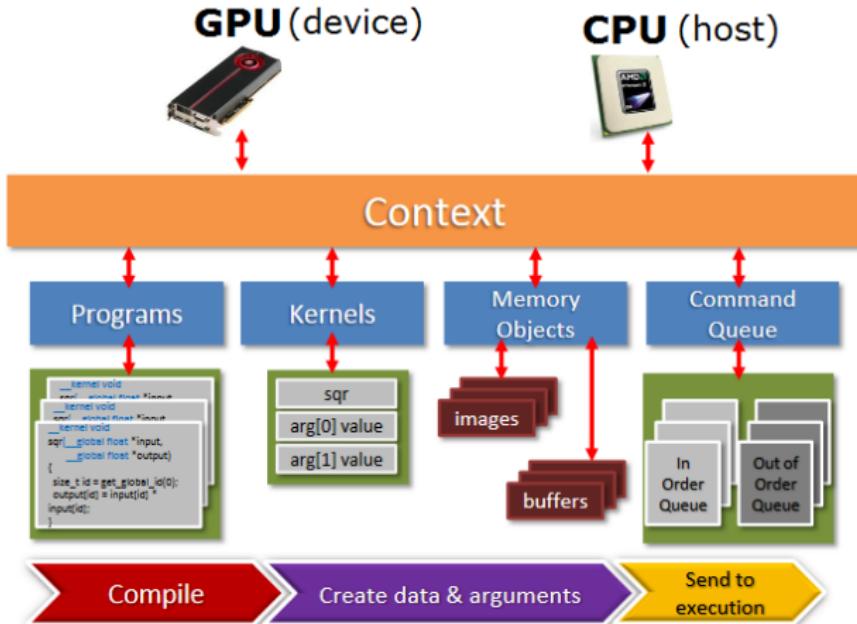


Modelo de programação:

- Compute devices
- Kernels
- Extensões de C/C++
- API
- Kernels compilados em tempo de execução
- Hierarquia de memória

Fluxo de um Programa OpenCL

OpenCL™ Program Flow



Mão na massa!

Vamos executar um exemplo em OpenCL, disponível em
github.com/phrb/intro-cuda:

- `src/opencl`

Compilação de Aplicações CUDA

Nvidia CUDA Compiler (nvcc)

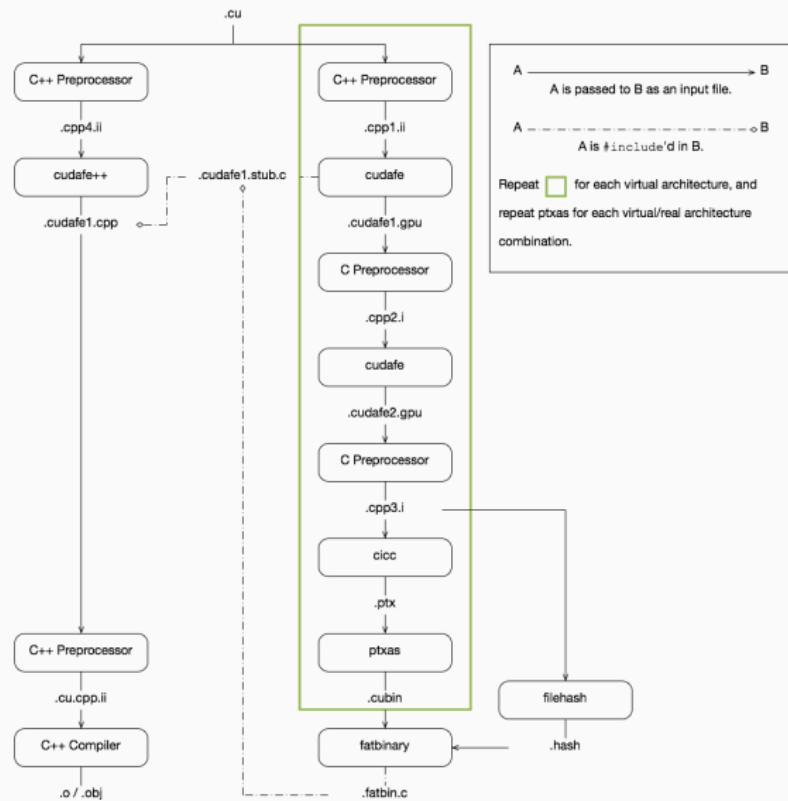


- Sistema **proprietário**
- Usa o compilador C++ do *host*
- Compila código CUDA para *Parallel Thread Execution* (**PTX ISA**)
- Código do *host* + Código do *device* → **fatbinary**

nvcc: Modelo de programação CUDA

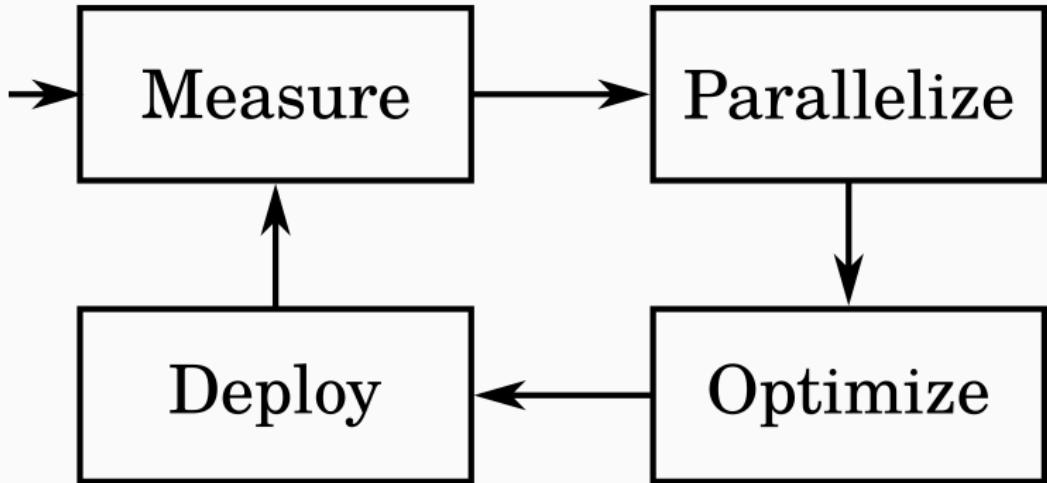
- SIMD, SIMT
- *Single Program Multiple Data* (SPMD)
- *Device*: threads paralelas e independentes
- *Host*: não interfere

nvcc: Trajetória de Compilação



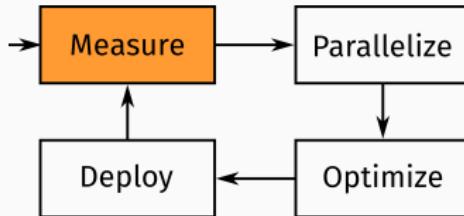
Boas Práticas em Otimização

Measure, Parallelize, Optimize, Deploy



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

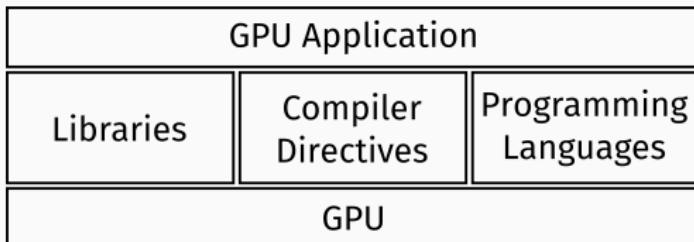
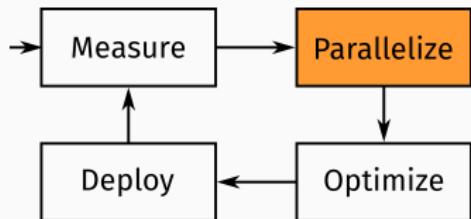
Measure



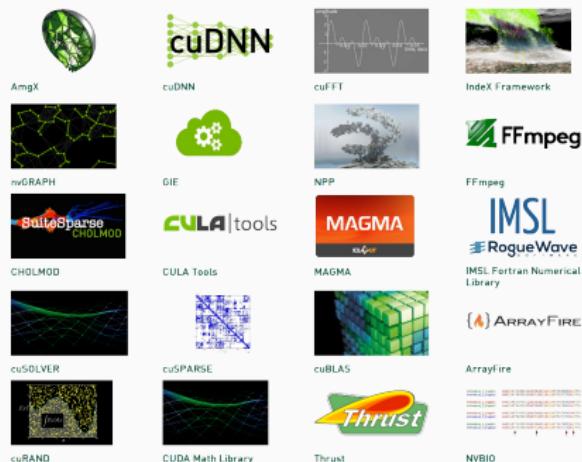
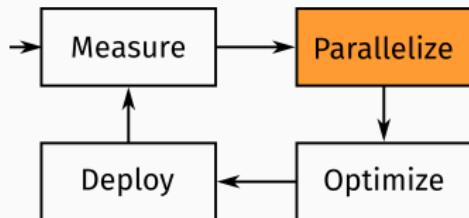
Dado um **programa sequencial**:

- 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**)

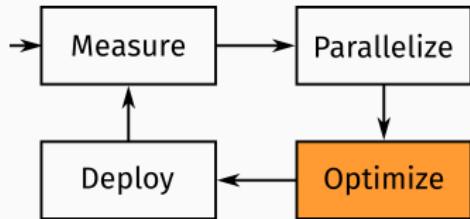
Parallelize



Parallelize



Optimize

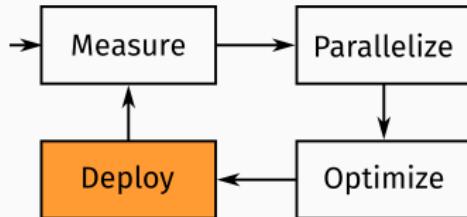


Otimização:

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

Eficiência com algoritmos, desempenho com estruturas de dados

Deploy



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

Análise de Aplicações CUDA

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*
- Memória

O que executar em **GPUs**?

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

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

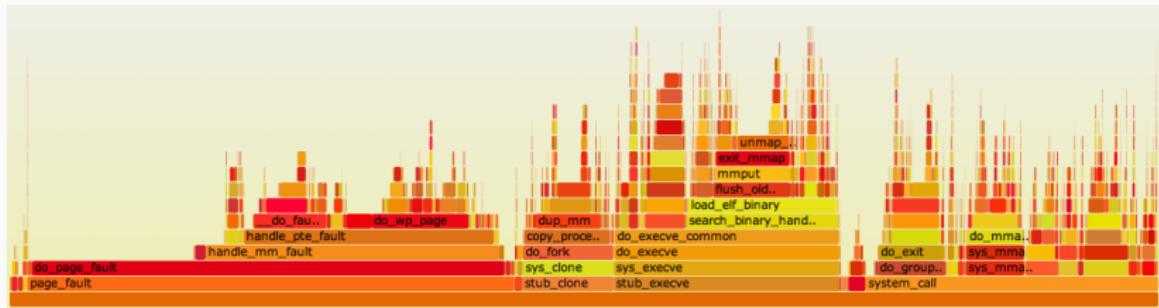
Profiling

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

Instrumentação do Código:

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

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:

- 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 [Acessado em 29/07/16]

Profiling: nvvp

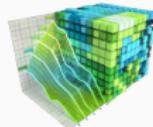


Fonte: docs.nvidia.com/cuda/profiler-users-guide [Acessado em 29/07/16]

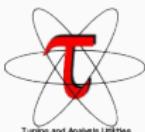
Profiling: Ferramentas



NVIDIA® Nsight™



NVIDIA Visual Profiler



Tuning and Analysis Utilities

TAU Performance System®



VampirTrace



The PAPI CUDA Component



The NVIDIA CUDA Profiling Tools Interface

Fonte: 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)**

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

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

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

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**?

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

Otimizações de Instruções

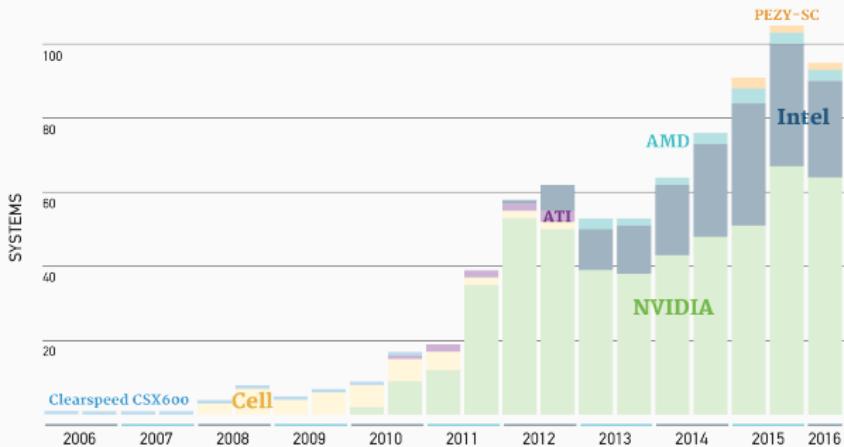
Baixa prioridade para atingir alto desempenho:

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

Conclusão

Conclusão

ACCELERATORS/CO-PROCESSORS



Atingir **alto desempenho** através do uso de **aceleradores (GPUs)**

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

Outros recursos:

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

Introdução à Programação de GPUs com a Plataforma CUDA

Pedro Bruel

phrb@ime.usp.br

21 de Maio de 2021



Instituto de Matemática e Estatística
Universidade de São Paulo