

# INTRODUÇÃO À PROGRAMAÇÃO PARA GPUs USANDO CUDA

---

Pedro Bruel

phrb@ime.usp.br

29 de Setembro de 2015



Instituto de Matemática e Estatística  
Universidade de São Paulo

## 1. Introdução

- Recapitulando: Um *template* para programas CUDA
- *Profilers* e *Debuggers*

## 1. Introdução

- Recapitulando: Um *template* para programas CUDA
- *Profilers e Debuggers*

## 2. Ferramentas

- `nvcc`
- `nvprof`
- `cuda-gdb`
- `cuda-memcheck`

Os *pdfs* com as aulas e todo o código fonte usado nos exemplos estão no [GitHub](#):

- [github.com/phrb/aulas-gpu](https://github.com/phrb/aulas-gpu)

Os *pdfs* com as aulas e todo o código fonte usado nos exemplos estão no [GitHub](#):

- [github.com/phrb/aulas-gpu](https://github.com/phrb/aulas-gpu)

Outros recursos:

- CUDA Toolkit Documentation: [docs.nvidia.com/cuda](https://docs.nvidia.com/cuda)
- GPU Teaching Kit: [syllabus.gputeachingkit.com](https://syllabus.gputeachingkit.com)
- iPython: [ipython.org/notebook.html](https://ipython.org/notebook.html)
- CUDA Toolkit: [developer.nvidia.com/cuda-toolkit](https://developer.nvidia.com/cuda-toolkit)
- Anaconda: [continuum.io/downloads](https://continuum.io/downloads)

# UM TEMPLATE PARA CUDA C

```
// Código em src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu  
#include <cuda_runtime.h>
```

# UM TEMPLATE PARA CUDA C

```
// Código em src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu  
#include <cuda_runtime.h>  
  
float *h_A = (float *)malloc(size);  
if (h_A == NULL) { ... };  
  
err = cudaMalloc((void **)&d_A, size);  
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);  
if (err != cudaSuccess) { ... };
```

# UM TEMPLATE PARA CUDA C

```
// Código em src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu
#include <cuda_runtime.h>

float *h_A = (float *)malloc(size);
if (h_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;
```



# UM TEMPLATE PARA CUDA C

```
// Código em src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu
#include <cuda_runtime.h>

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

# UM TEMPLATE PARA CUDA C

```
// Código em src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu
#include <cuda_runtime.h>

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

Exemplos no repositório:

- `src/cuda-samples/0_Simple`
- `src/cuda-samples/0_Simple/vectorAdd/vectorAdd.cu`
- `src/cuda-samples/0_Simple/template_runtime/template_runtime.cu`

*Profilers:*

- Ferramentas para análise de código em tempo de execução
- A intenção é otimizar o código

*Profilers:*

- Ferramentas para análise de código em tempo de execução
- A intenção é otimizar o código

Como?

- Instrumentação do Código
- Captura de eventos
- Geração de dados

## *Profilers:*

- Ferramentas para análise de código em tempo de execução
- A intenção é otimizar o código

## Como?

- Instrumentação do Código
- Captura de eventos
- Geração de dados

## O quê?

- Consumo de memória
- Frequência e duração de chamadas de função
- Uso de instruções específicas

## *Profilers:*

- Ferramentas para análise de código **em tempo de execução**
- A intenção é **otimizar o código**

## Como?

- Instrumentação do Código
- Captura de eventos
- Geração de dados

## O quê?

- Consumo de memória
- Frequência e duração de chamadas de função
- Uso de instruções específicas
- **Bottlenecks**, ou **Gargalos**, de desempenho

## *Debuggers:*

- Ferramentas para análise de código em tempo de execução
- As intenções são consertar bugs e realizar testes



*Debuggers:*

- Ferramentas para análise de código em tempo de execução
- As intenções são consertar bugs e realizar testes

Como?

- Instrumentação do Código
- Execução passo-a-passo
- Breakpoints

*Debuggers:*

- Ferramentas para análise de código em tempo de execução
- As intenções são consertar bugs e realizar testes

Como?

- Instrumentação do Código
- Execução passo-a-passo
- Breakpoints

O quê?

- Acessos e vazamentos de memória
- Pode ajudar com vários tipos de bugs ☺

Os próximos *slides* foram adaptados do material disponível no GPU Teaching Kit:

- [syllabus.gputeachingkit.com](https://syllabus.gputeachingkit.com)



## GPU Teaching Kit



The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).



GPU Teaching Kit

Accelerated Computing



# Lecture 2.4 – Introduction to CUDA C

Introduction to the CUDA Toolkit

# Objective

- To become familiar with some valuable tools and resources from the CUDA Toolkit
  - Compiler flags
  - Debuggers
  - Profilers

# GPU Programming Languages

---

Numerical analytics ►

MATLAB, Mathematica, LabVIEW

Fortran ►

CUDA Fortran

C ►

CUDA C

C++ ►

CUDA C++

Python ►

PyCUDA, Copperhead, Numba, NumbaPro

F# ►

Alea.cuBase

# CUDA - C

## Applications

Libraries

Easy to use  
Most Performance

Compiler  
Directives

Easy to use  
Portable code

Programming  
Languages

Most Performance  
Most Flexibility



# NVCC Compiler

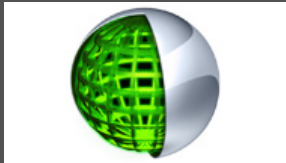
- NVIDIA provides a CUDA-C compiler
  - `nvcc`
- NVCC compiles device code then forwards code on to the host compiler (e.g. `g++`)
- Can be used to compile & link host only applications

# Compiler Flags

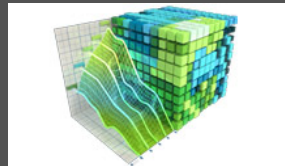
- Remember there are two compilers being used
  - NVCC: Device code
  - Host Compiler: C/C++ code
- NVCC supports some host compiler flags
  - If flag is unsupported, use `-Xcompiler` to forward to host
    - e.g. `-Xcompiler -fopenmp`
- Debugging Flags
  - `-g`: Include host debugging symbols
  - `-G`: Include device debugging symbols
  - `-lineinfo`: Include line information with symbols

# Developer Tools - Profilers

**NSIGHT**



**NVVP**

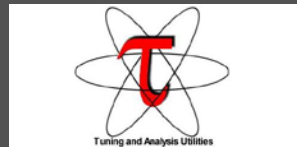


**NVPROF**

```
==205661== Profiling result:
Time(s)      Time      Calls      Avg      Min      Max      Name
49.88%      866.6ms      504758      1.727ms      1.560ms      2.815ms      void th
unt_thrust::detail::device_generate_function::thrust::detail::fill
25.33%      449.0ms      252662      1.741ms      1.536ms      2.308ms      void th
t_thrust::detail::device_generate_function::thrust::detail::fill_fo
17.87%      296.6ms      200      1.483ms      1.264ms      1.725ms      kerComp
2.98%      51.81ms      200      259.89us      246.97us      264.83us      kerTake
1.16%      26.17ms      501      48.26us      30us      17.477ms      [CUDA m
9.93%      16.19ms      200      80.99us      71.84us      96.75us      kerColl
0.73%      12.63ms      400      31.58us      14.72us      50.43us      [CUDA m
0.49%      12.07ms      200      60.37us      59.48us      62.36us      kerReme
0.63%      10.99ms      200      54.96us      52.68us      58.20us      kerTake
0.32%      5.554ms      200      27.76us      22.55us      33.15us      [CUDA m
0.12%      2.134ms      1      2.134ms      2.134ms      2.134ms      void th
```

**NVIDIA Provided**

**TAU**



Tuning and Analysis Utilities

**VampirTrace**



**3rd Party**

<https://developer.nvidia.com/performance-analysis-tools>

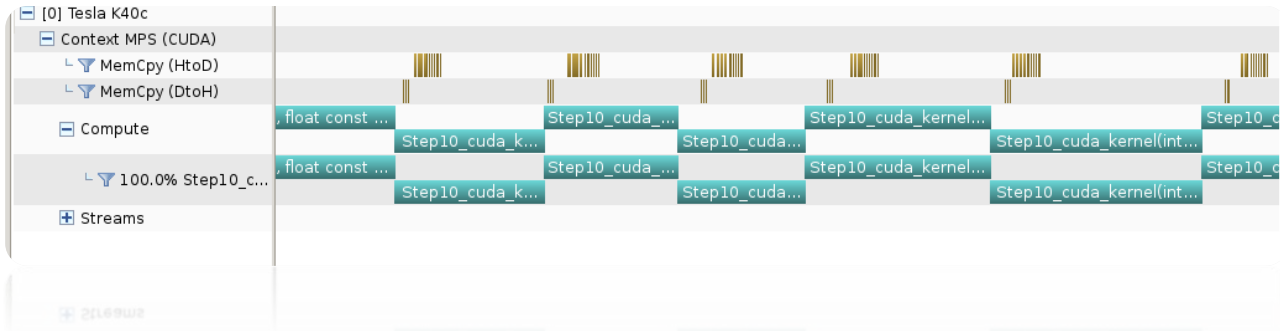
# NVPROF

## Command Line Profiler

- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

# NVIDIA's Visual Profiler (NVVP)

## Timeline



## Guided System

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

**3. Compute, Bandwidth, or Latency Bound**

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

**Perform Compute Analysis**

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

**Perform Latency Analysis**

**Perform Memory Bandwidth Analysis**

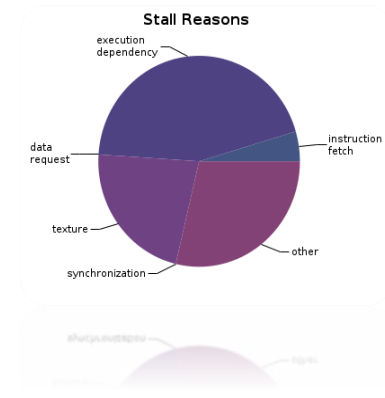
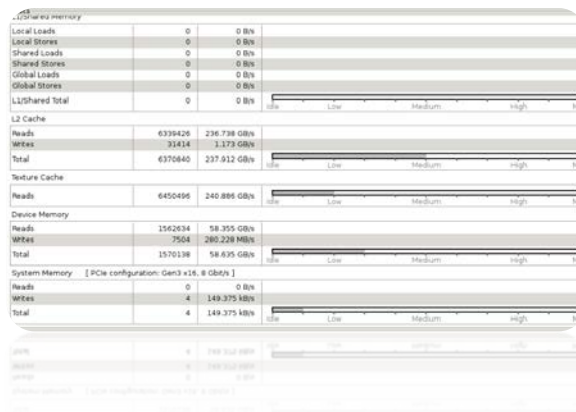
Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

**Rerun Analysis**

If you modify the kernel you need to rerun your application to update this analysis.

go to the next step

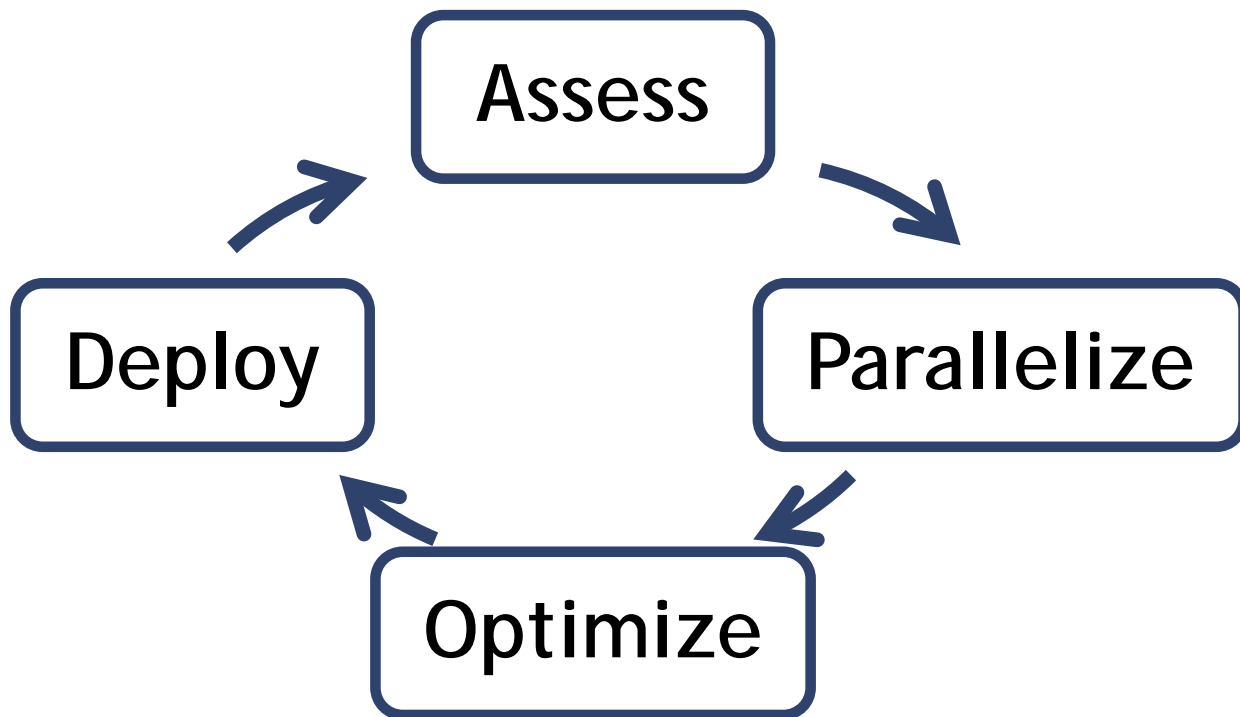
## Analysis



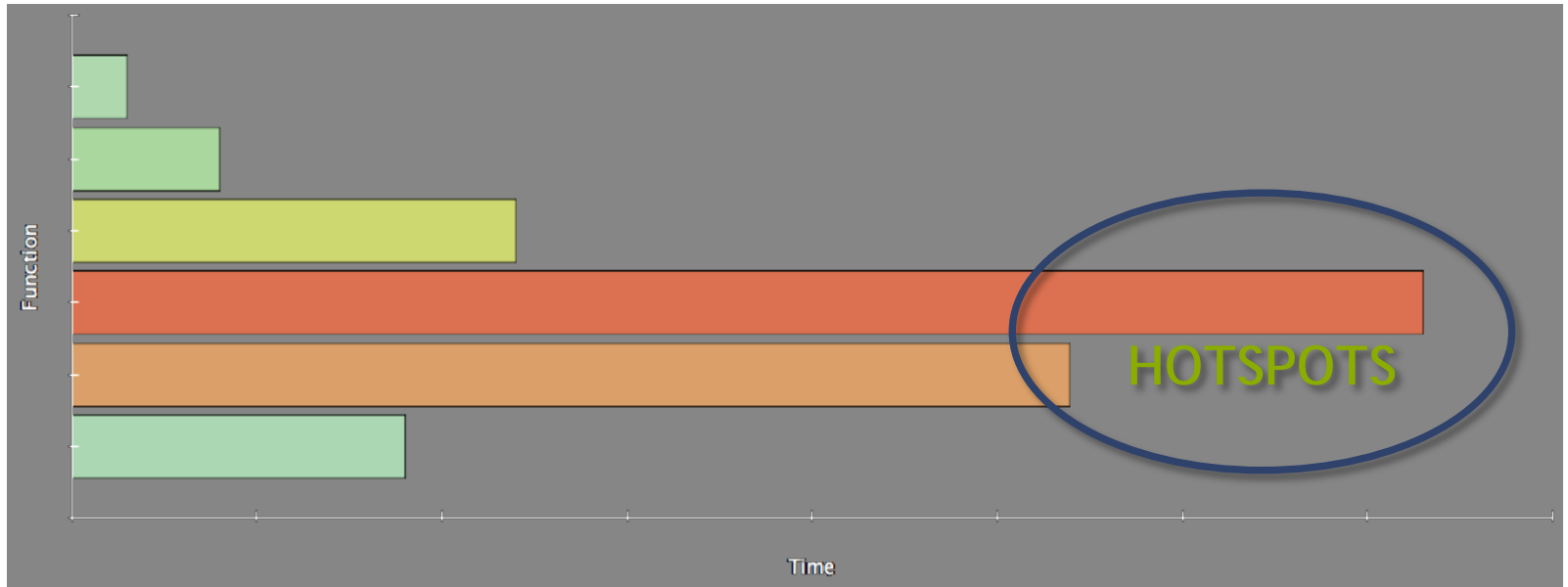
# Profiler Summary

- Many profile tools are available
- NVIDIA Provided
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- 3<sup>rd</sup> Party
  - TAU
  - VAMPIR

# Optimization



# Assess



- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit



# Parallelize

Applications

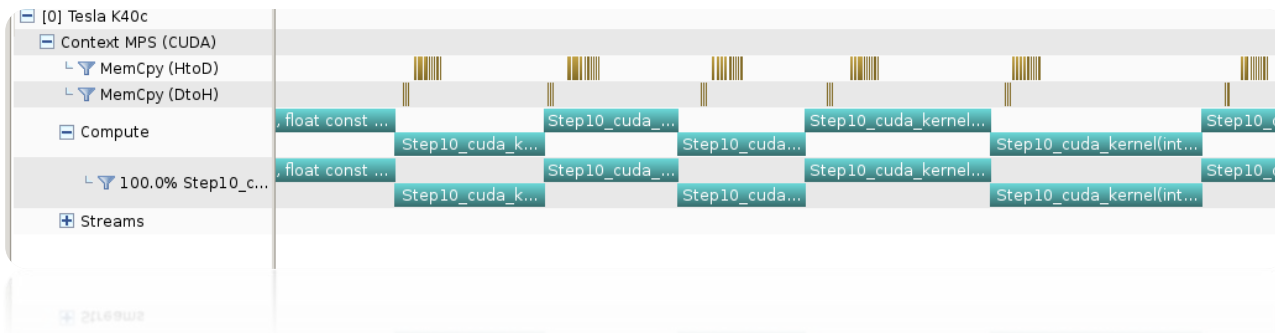
Libraries

Compiler  
Directives

Programming  
Languages

# Optimize

## Timeline



## Guided System

**1. CUDA Application Analysis**

**2. Performance-Critical Kernels**

**3. Compute, Bandwidth, or Latency Bound**

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10\_cuda\_kernel" is most likely limited by compute.

**Perform Compute Analysis**

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

**Perform Latency Analysis**

**Perform Memory Bandwidth Analysis**

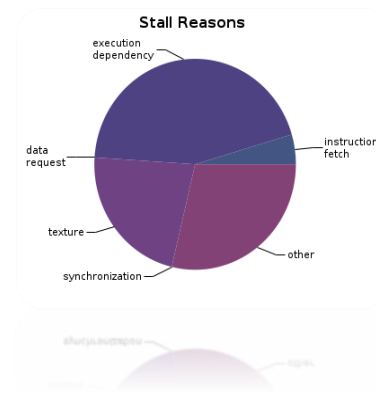
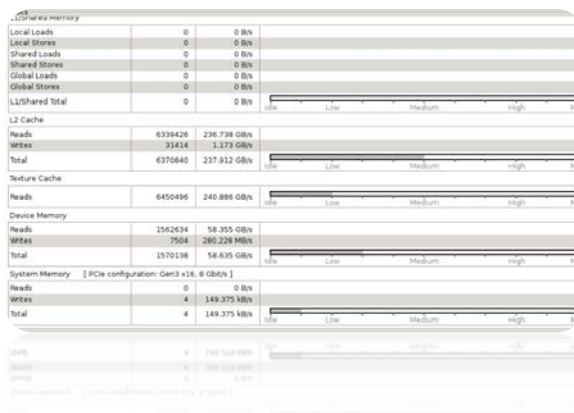
Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

**Rerun Analysis**

If you modify the kernel you need to rerun your application to update this analysis.

go to page 400 Workbench  
to learn more about the analysis tool and how to use it, visit the page  
go to page 400 Workbench

## Analysis



# Developer Tools - Debuggers

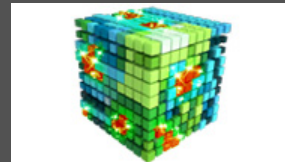
**NSIGHT**



**CUDA-GDB**



**CUDA MEMCHECK**



**NVIDIA Provided**

**allinea**  
DDT

**TotalView®**

**3<sup>rd</sup> Party**

<https://developer.nvidia.com/debugging-solutions>

# CUDA-GDB

- cuda-gdb is an extension of GDB
  - Provides seamless debugging of CUDA and CPU code
- Works on Linux and Macintosh
  - For a Windows debugger use NSIGHT Visual Studio Edition

<http://docs.nvidia.com/cuda/cuda-gdb>

# CUDA-MEMCHECK

- Memory debugging tool
  - No recompilation necessary
    - `%> cuda-memcheck ./exe`
- Can detect the following errors
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- For line numbers use the following compiler flags:
  - `-Xcompiler -rdynamic -lineinfo`

<http://docs.nvidia.com/cuda/cuda-memcheck>

Obrigado!

# INTRODUÇÃO À PROGRAMAÇÃO PARA GPUs USANDO CUDA

---

Pedro Bruel

phrb@ime.usp.br

29 de Setembro de 2015



Instituto de Matemática e Estatística  
Universidade de São Paulo