Introduction to GPU Programming with the CUDA Platform

Pedro Bruel
phrb@ime.usp.br
4th INFIERI, 2017

Instituto de Matemática e Estatística
Universidade de São Paulo

About

Pedro Bruel
phrb@ime.usp.br

Alfredo Goldman
gold@ime.usp.br

Marco Dimas Gubitoso

gubi@ime.usp.br

Part I

1. Introduction
2. Heterogeneous Computing
3. GPUs
4. CUDA Platform
5. CUDA C

Part II

Slides

This presentation and all source code are available at GitHub:

- `github.com/phrb/intro-cuda`

Hardware Acceleration

The use of devices to accelerate computations on large datasets:

Hardware Acceleration

The use of devices to accelerate computations on large datasets:

- Associated with a host processor

Hardware Acceleration

The use of devices to accelerate computations on large datasets:

- Associated with a host processor
- Separate control and memory

Hardware Acceleration

The use of devices to accelerate computations on large datasets:

- Associated with a host processor
- Separate control and memory
- Differ on specialization and configurability

Hardware Acceleration

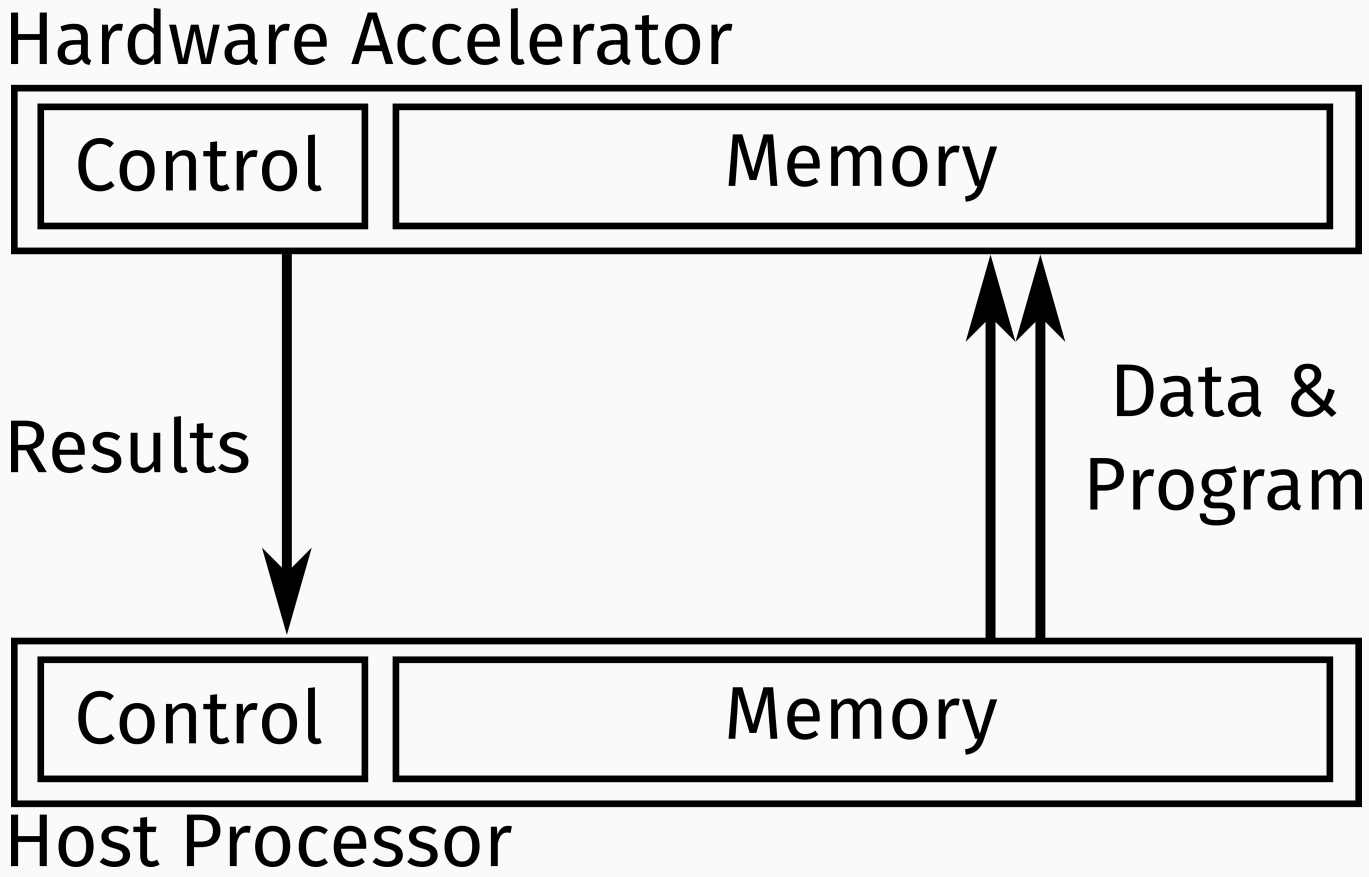
The use of devices to accelerate computations on large datasets:

- Associated with a host processor
- Separate control and memory
- Differ on specialization and configurability
- GPUs, DSPs, FPGAs, ASICs

Hardware Acceleration

Use cases:

- Machine Learning
- Digital Signal Processing
- Bioinformatics
- Cryptography
- Meteorology
- Simulations
- ...


Hardware Accelerator
Control
Memory
Results
Data &
Program
Control
Memory
Host Processor

Percentage of systems with accelerators in the *Top500*:

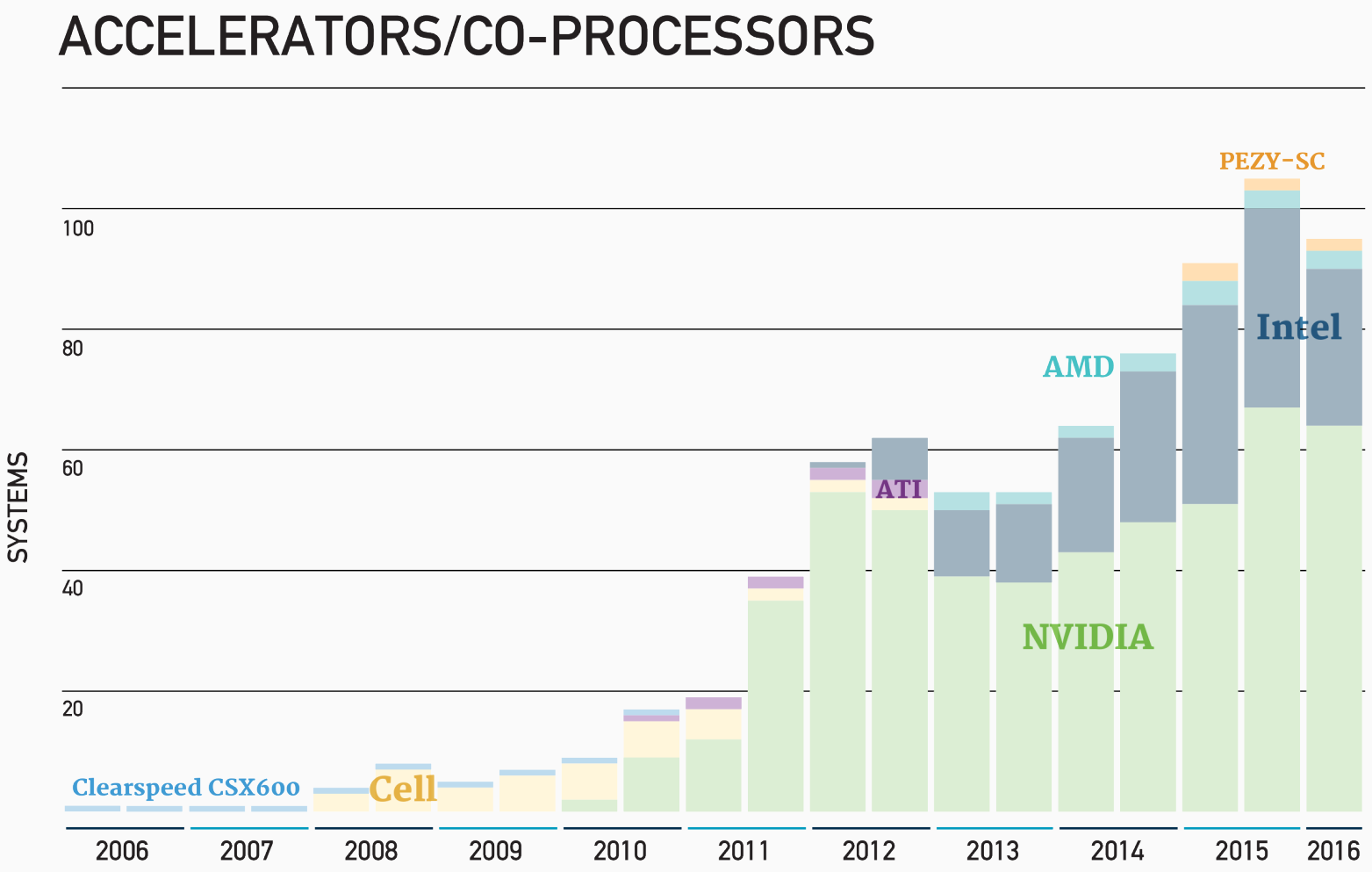


Image: `top500.org/lists/2016/06/download/TOP500_201606_Poster.pdf` [Accessed in 29/07/16]

Heterogeneous Computing

Heterogeneous Computing

Given heterogeneous computational resources, datasets and a set of computations, how to distribute data and computations to optimize resource usage?

Image: `olcf.ornl.gov/titan` [Accessed in 29/07/16]

Heterogeneous Computing

Heterogeneous computational resources:

Heterogeneous Computing

Heterogeneous computational resources:

- Low latency: CPUs

Heterogeneous Computing

Heterogeneous computational resources:

- Low latency: CPUs
- High Throughput: GPUs

Heterogeneous Computing

Heterogeneous computational resources:

- Low latency: CPUs
- High Throughput: GPUs
- Reconfigurable: FPGAs

Heterogeneous Computing

Heterogeneous computational resources:

- Low latency: CPUs
- High Throughput: GPUs
- Reconfigurable: FPGAs
- Specialized: ASICs, DSPs

Heterogeneous Computing

Heterogeneous computational resources:

- Low latency: CPUs
- High Throughput: GPUs
- Reconfigurable: FPGAs
- Specialized: ASICs, DSPs
- Memory?

Datasets:

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

Heterogeneous Computing

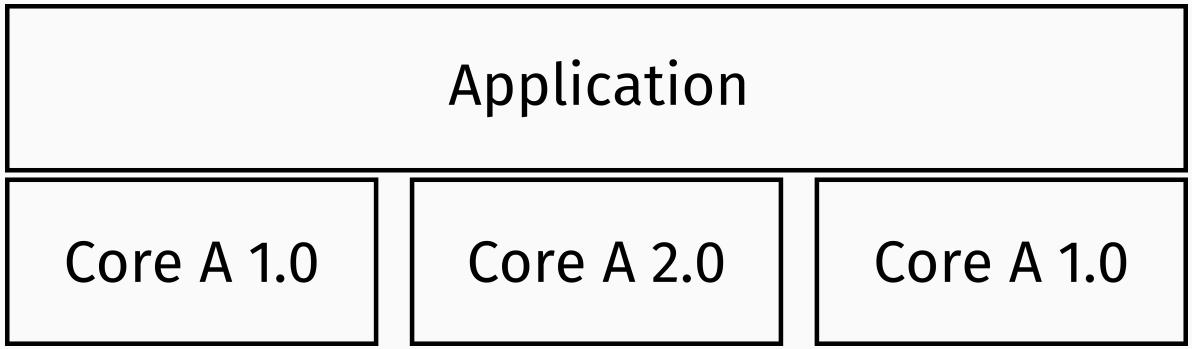
Datasets:

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

Programming models (computations):

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

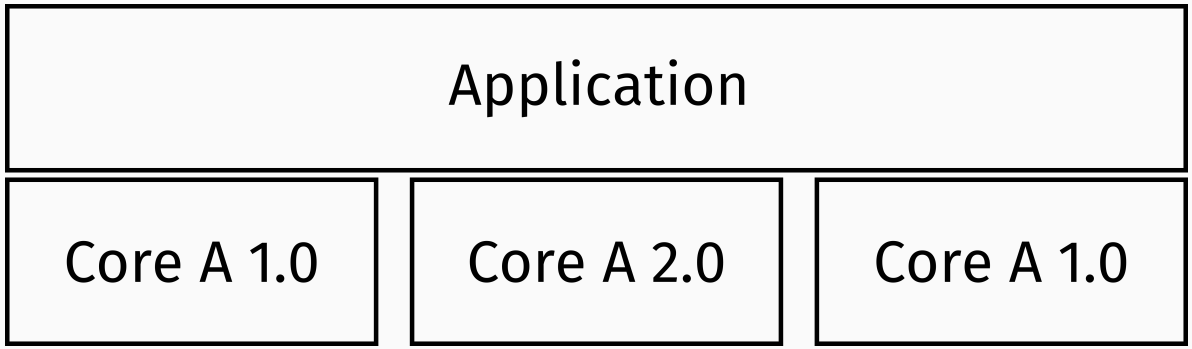
Heterogeneous Computing: Scalability



Scalability means not losing performance running in:

- Multiple identical cores

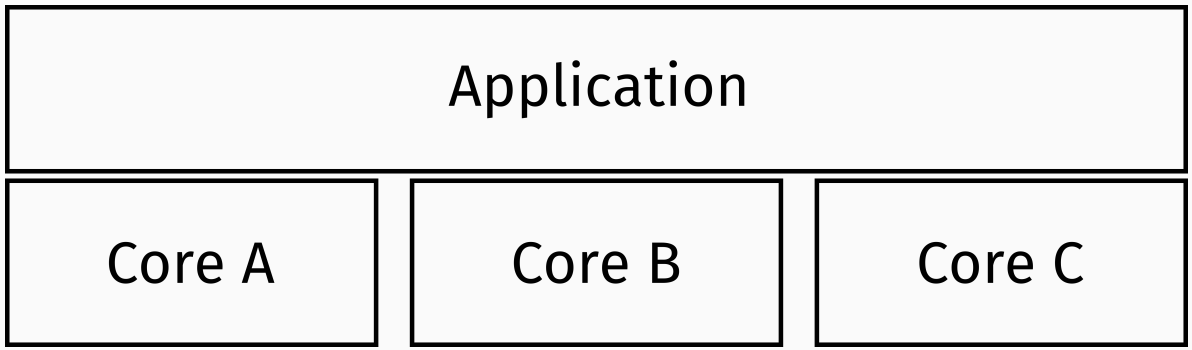
Heterogeneous Computing: Scalability



Scalability means not losing performance running in:

- Multiple identical cores
- New versions of the same core

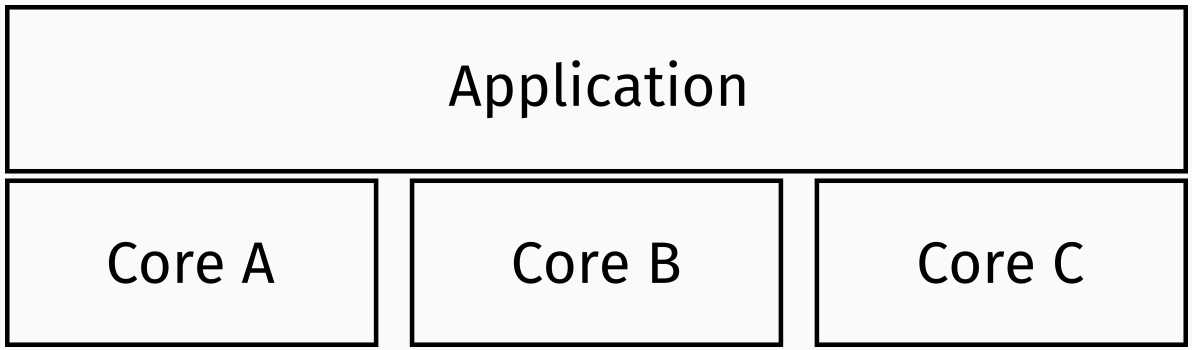
Heterogeneous Computing: Portability



Portability means not losing performance running in different:

- Cores or accelerators

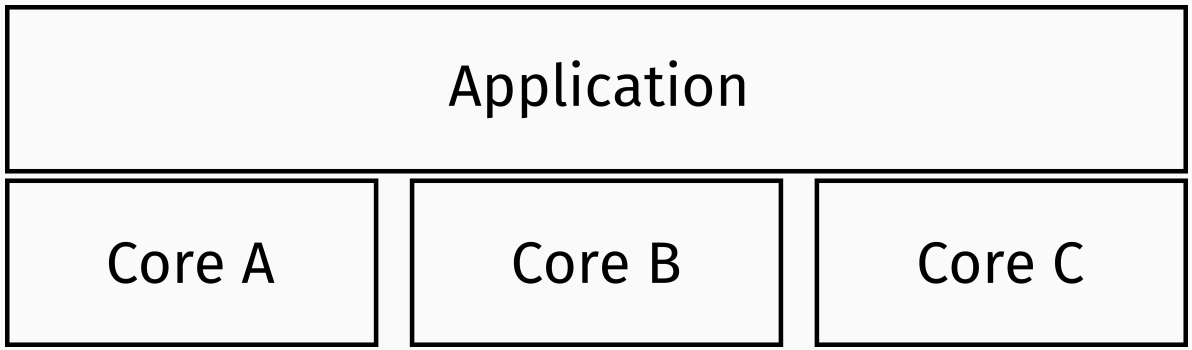
Heterogeneous Computing: Portability



Portability means not losing performance running in different:

- Cores or accelerators
- Memory models

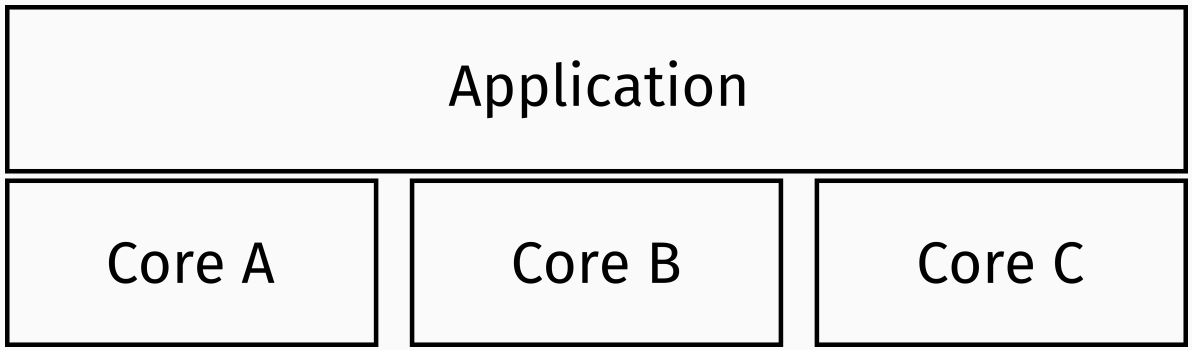
Heterogeneous Computing: Portability



Portability means not losing performance running in different:

- Cores or accelerators
- Memory models
- Parallelism models

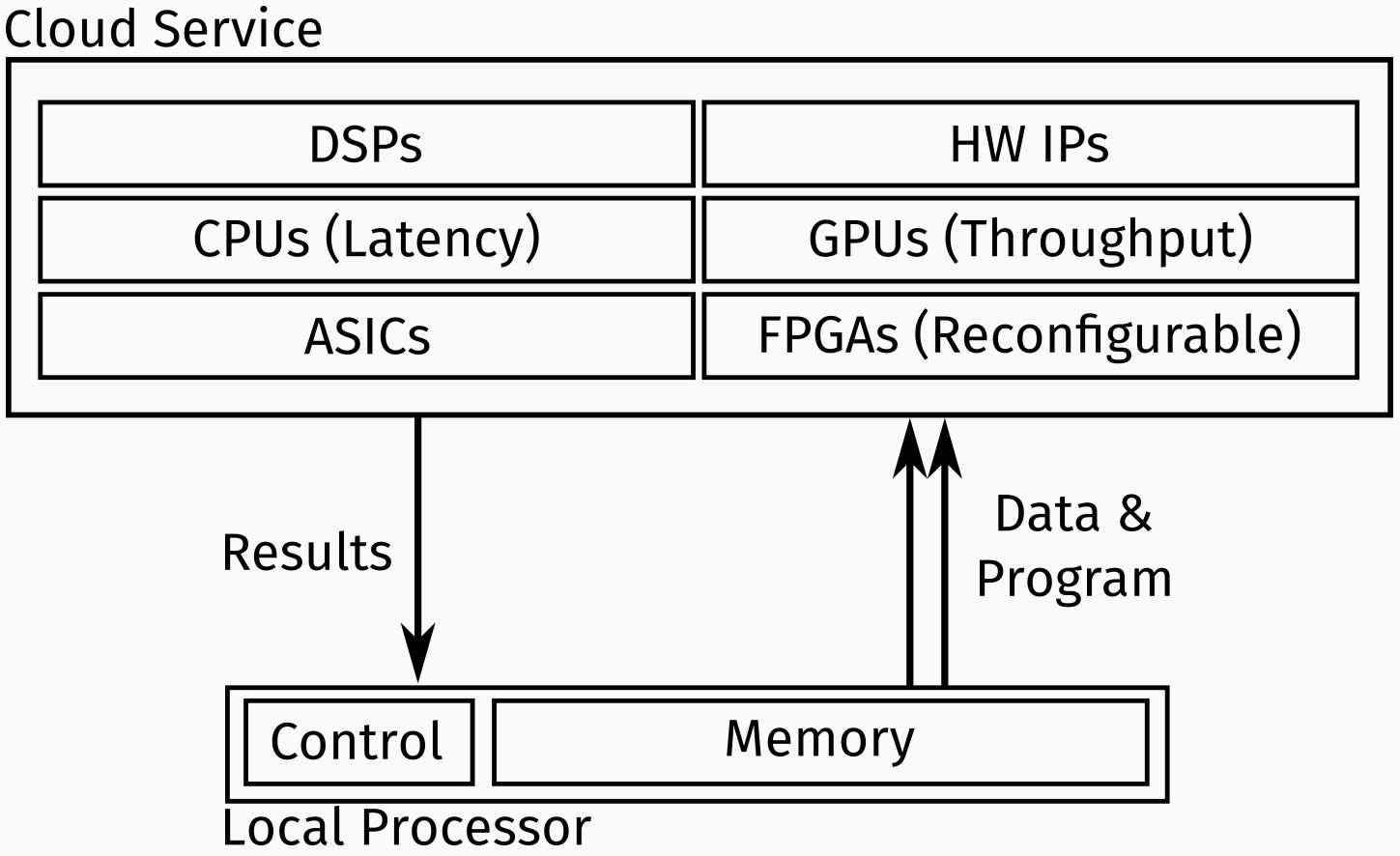
Heterogeneous Computing: Portability



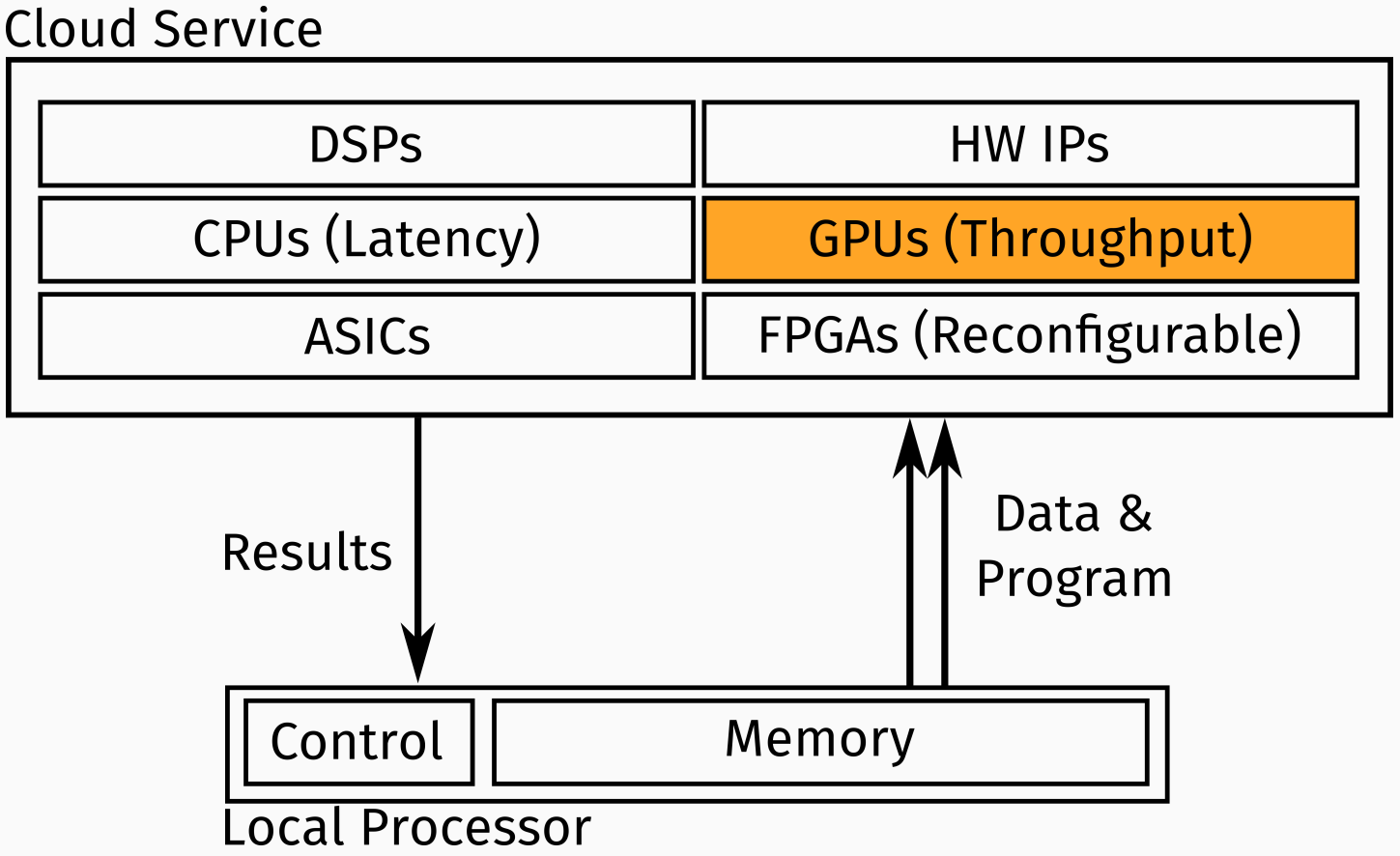
Portability means not losing performance running in different:

- Cores or accelerators
- Memory models
- Parallelism models
- Instruction sets

Heterogeneous Computing



Heterogeneous Computing



Graphics Processing Units

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

- Small caches (*kilobytes*)

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

- Small caches (*kilobytes*)
- No branch prediction

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

- Small caches (*kilobytes*)
- No branch prediction
- Thousands of higher latency ALUs

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

- Small caches (*kilobytes*)
- No branch prediction
- Thousands of higher latency ALUs
- Execution pipelines

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

- Small caches (*kilobytes*)
- No branch prediction
- Thousands of higher latency ALUs
- Execution pipelines
- 114 688 concurrent threads in Pascal architecture

Graphics Processing Units

Now, largely used in general-purpose computing, also called *General Purpose GPUs* (GPGPUs):

Graphics Processing Units

Now, largely used in general-purpose computing, also called *General Purpose GPUs* (GPGPUs):

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

Hardware Model

Flynn's taxonomy:

- *Single Instruction Multiple Data* (SIMD)

Hardware Model

Flynn's taxonomy:

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

Hardware Model

Flynn's taxonomy:

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

Scheduling and execution:

- *Streaming Multiprocessor* (SM)

Hardware Model

Flynn's taxonomy:

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

Scheduling and execution:

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

Hardware Model

Flynn's taxonomy:

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

Scheduling and execution:

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

Hardware Model

Higher-Level scheduling:

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

Hardware Model

Higher-Level scheduling:

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

Memory:

- Shared: L1, L2 caches; *megabytes* (NVIDIA Pascal)
- Global: volatile, GDDR5; *gigabytes*

Hardware Model

Higher-Level scheduling:

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

Memory:

- Shared: L1, L2 caches; *megabytes* (NVIDIA Pascal)
- Global: volatile, GDDR5; *gigabytes*
- SSD: non-volatile; *terabytes* (AMD SSG Fiji, 2017)

AMD SSG: `anandtech.com/show/10518/amd-announces-radeon-pro-ssg-fiji-with-m2-ssds-onboard` [Accessed in 30/07/16]

Compute Capability and SIMT

The Compute Capability specifies *Single Instruction Multiple Thread* (SIMT) architectures:

Compute Capability and SIMT

The Compute Capability specifies *Single Instruction Multiple Thread* (SIMT) architectures:

- *Warp*: Group of *threads* that runs in parallel
- *Streaming Multiprocessor* (SM): Creates, schedules and runs *warps*

Compute Capability and SIMT

The Compute Capability specifies *Single Instruction Multiple Thread* (SIMT) architectures:

- *Warp*: Group of *threads* that runs in parallel
- *Streaming Multiprocessor* (SM): Creates, schedules and runs *warps*
- Multiple *warps* run concurrently in a SM

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation` [Accessed in 29/07/16]

Compute Capability and SIMT

Warps:

Compute Capability and SIMT

Warps:

- Unit of execution and scheduling
- 32 *threads*

Compute Capability and SIMT

Warps:

- Unit of execution and scheduling
- 32 *threads*
- 1 common instruction at a time

Compute Capability and SIMT

Warps:

- Unit of execution and scheduling
- 32 *threads*
- 1 common instruction at a time
- *Active threads*: in the current execution branch
- *Inactive threads*: not in the current execution branch

Compute Capability and SIMT

Warps:

- Unit of execution and scheduling
- 32 *threads*
- 1 common instruction at a time
- *Active threads*: in the current execution branch
- *Inactive threads*: not in the current execution branch
- Inactive threads run sequentially

Compute Capability and SIMT

Warps:

- Unit of execution and scheduling
- 32 *threads*
- 1 common instruction at a time
- *Active threads*: in the current execution branch
- *Inactive threads*: not in the current execution branch
- Inactive threads run sequentially
- Efficiency: no execution divergence inside *Warps*

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation` [Accessed in 29/07/16]

Pascal GP100 Architecture

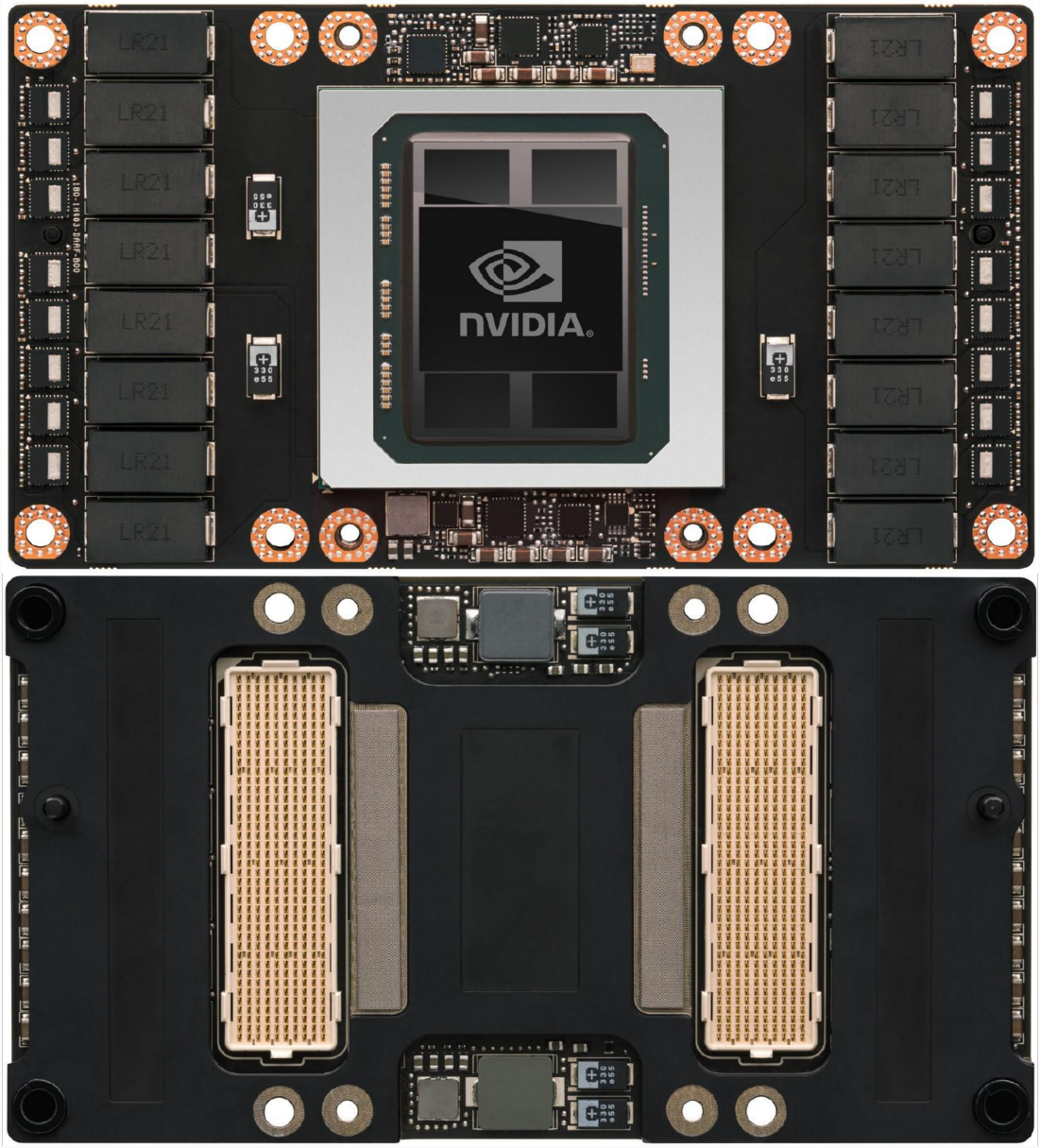


Image: images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf [Accessed in 29/07/16]

Pascal GP100 Architecture: *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

Image: `images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` [Accessed in 29/07/16]

Pascal GP100 Architecture: SM

Image: `images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` [Accessed in 29/07/16]

Pascal GP100 Architecture

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (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.

Image: `images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` [Accessed in 29/07/16]

Pascal GP100 Architecture

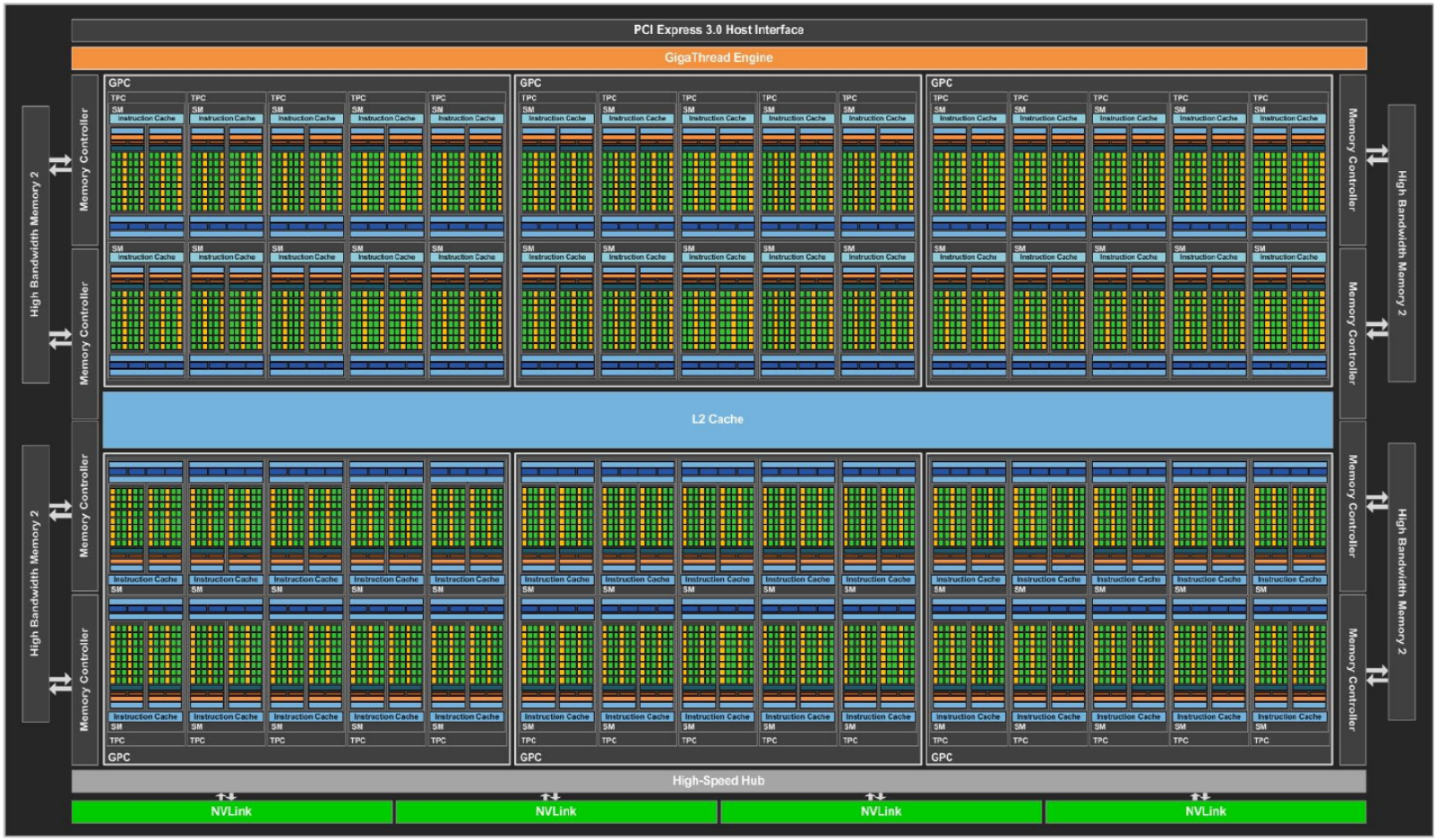
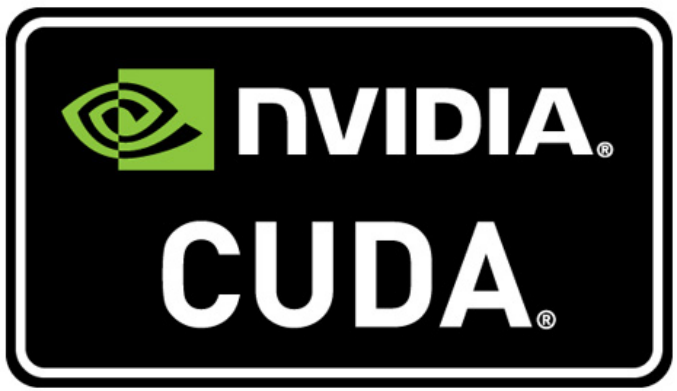
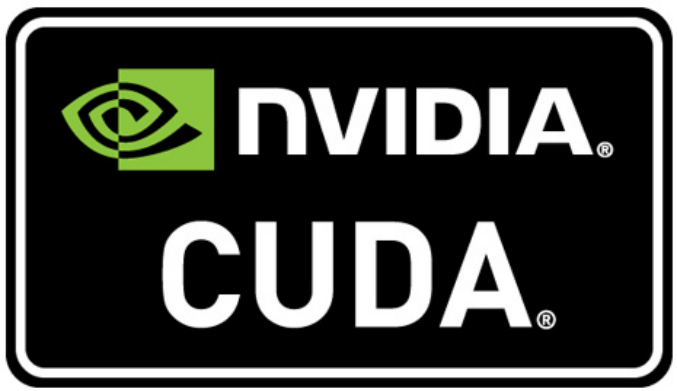


Image: `images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` [Accessed in 29/07/16]

CUDA Platform

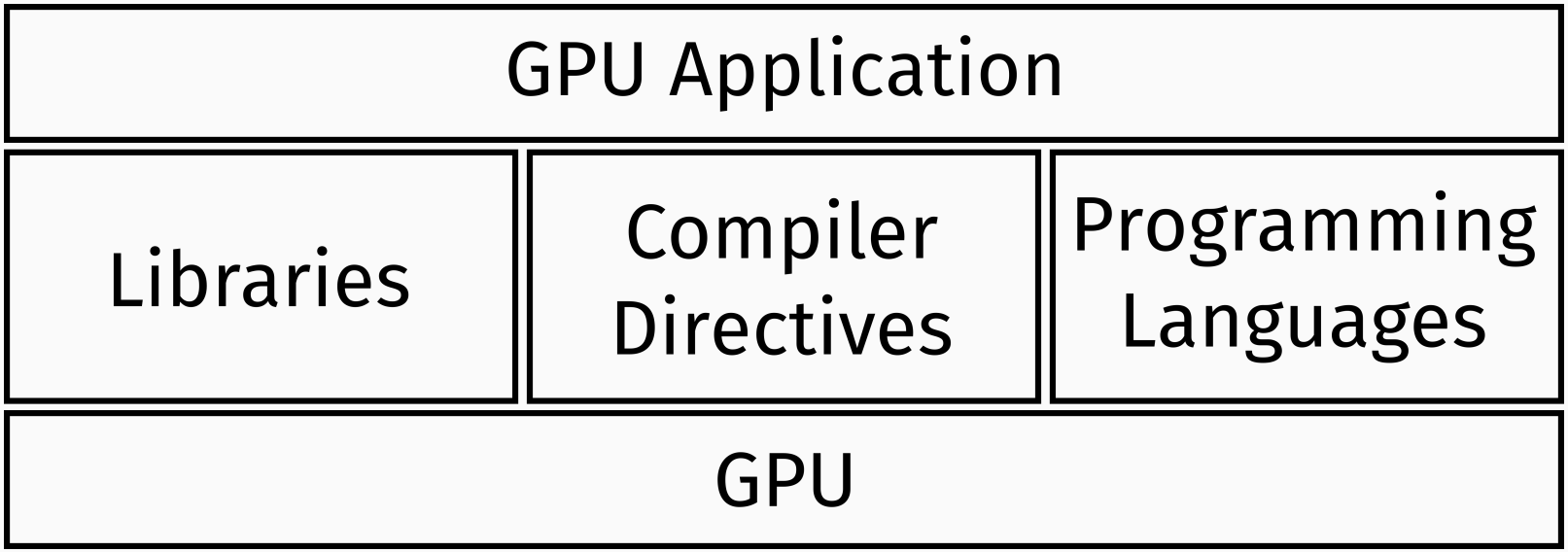


CUDA Platform

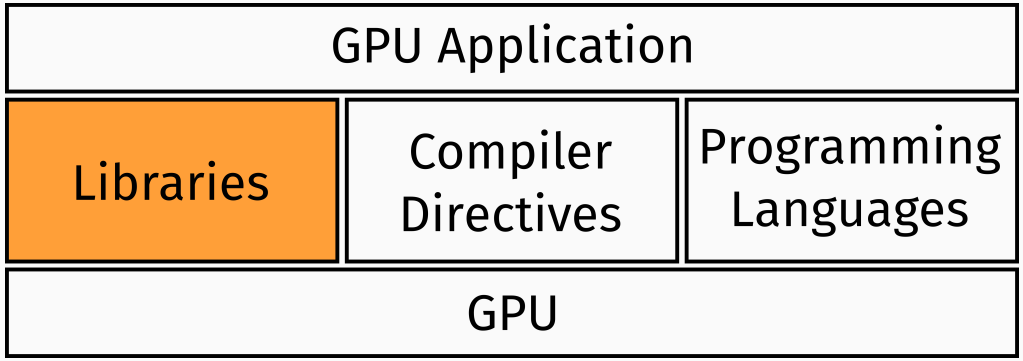


- A platform for parallel computing
- *Application Programming Interface* (API)
- CUDA *Toolkit*

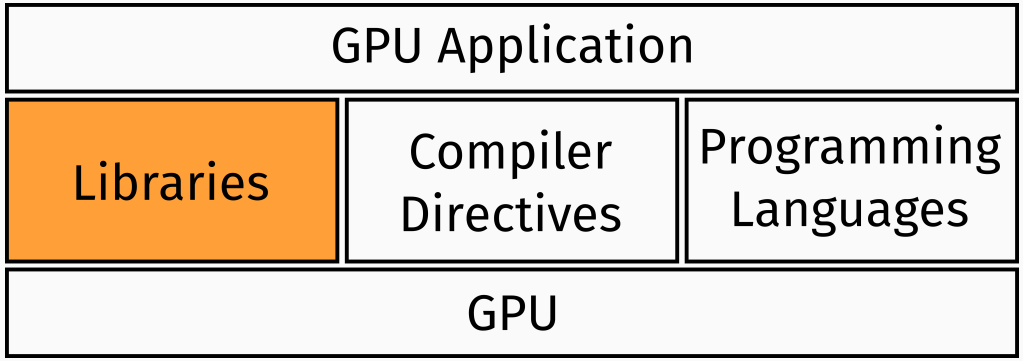
Software Acceleration



Libraries

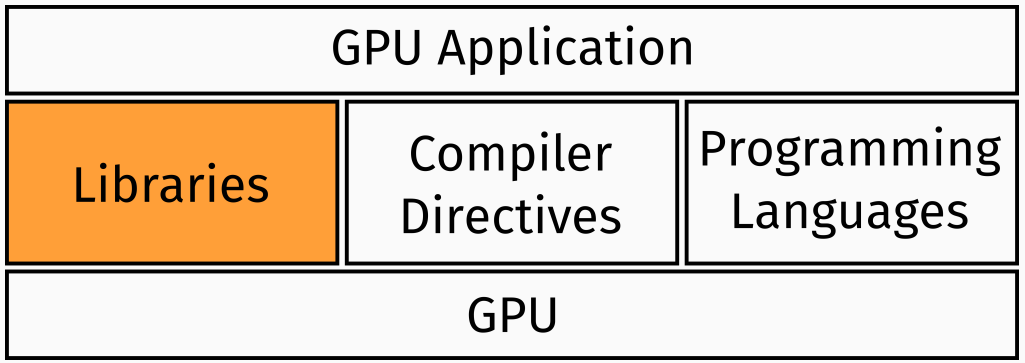

GPU Application
Libraries
Compiler Directives
Programming Languages
GPU

Libraries



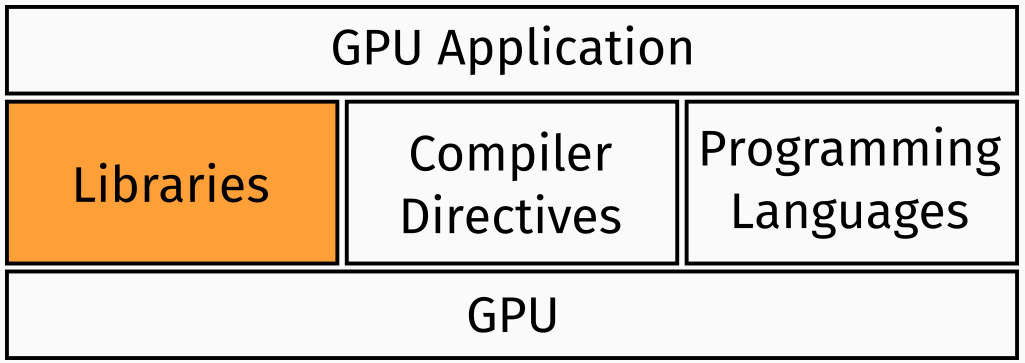
- Easy to use

Libraries



- Easy to use
- Drop-in acceleration

Libraries

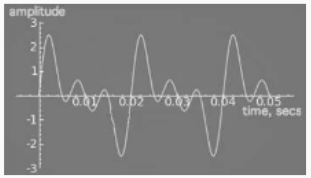


- Easy to use
- Drop-in acceleration
- Optimized by specialists

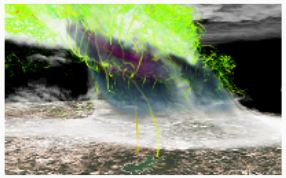
Libraries

AmgX

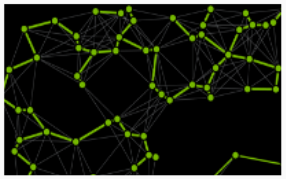
cuDNN



cuFFT



IndeX Framework



nvGRAPH

GIE

NPP

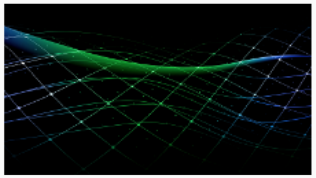
FFmpeg

CHOLMOD

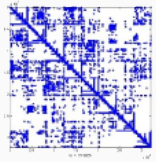
CULA Tools

MAGMA

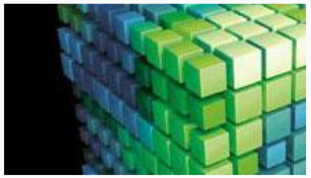
IMSL Fortran Numerical Library



cuSOLVER

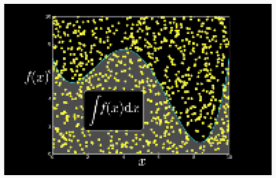


cuSPARSE

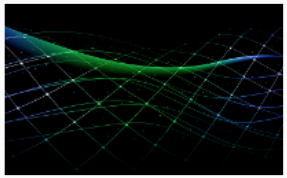


cuBLAS

ArrayFire

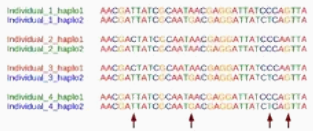


cuRAND



CUDA Math Library

Thrust



NVBIO

Image: `developer.nvidia.com/gpu-accelerated-libraries` [Accessed in 29/07/16]

Vector sum with the Thrust library:

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

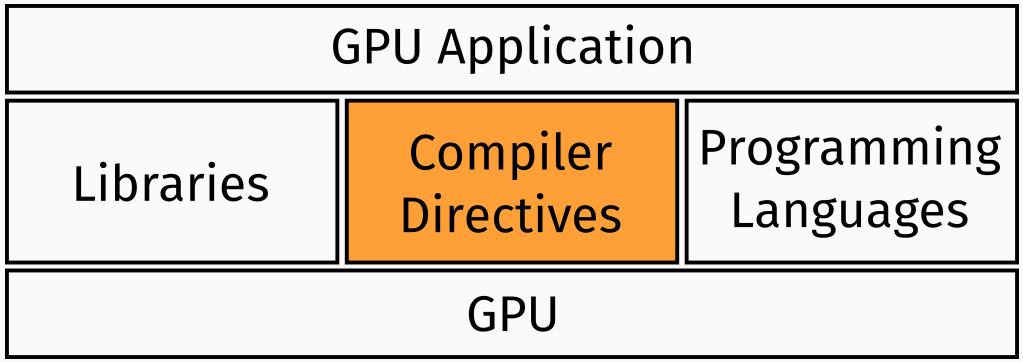
Vector sum with the Thrust library:

```
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());
```

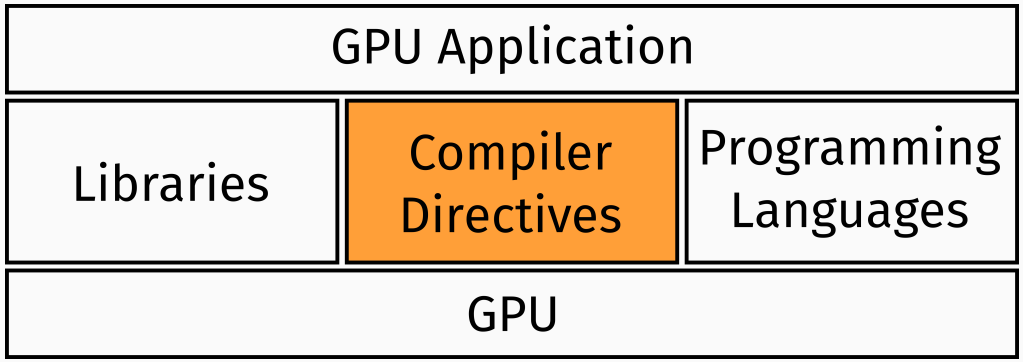
Vector sum with the Thrust library:

```
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>());
```

Compiler Directives

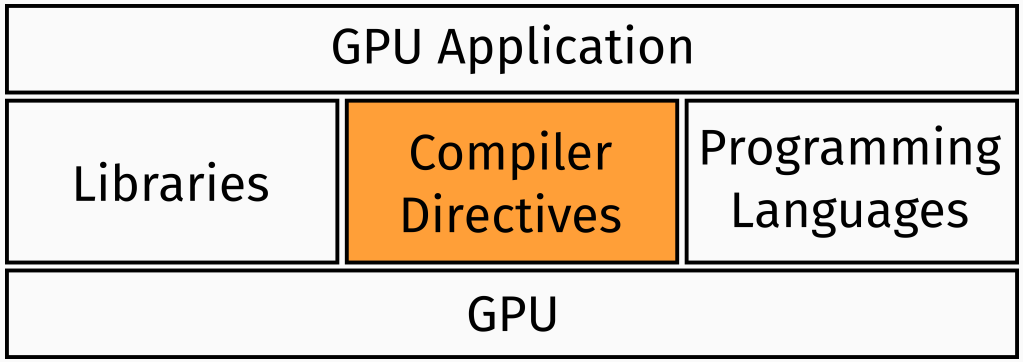


Compiler Directives



- Easy to use

Compiler Directives



- Easy to use
- Portable, but performance depends on compiler

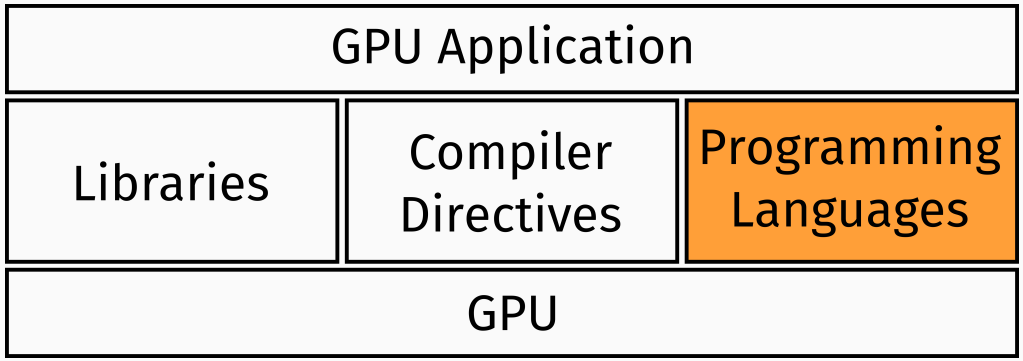
Vector sum with OpenACC:

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

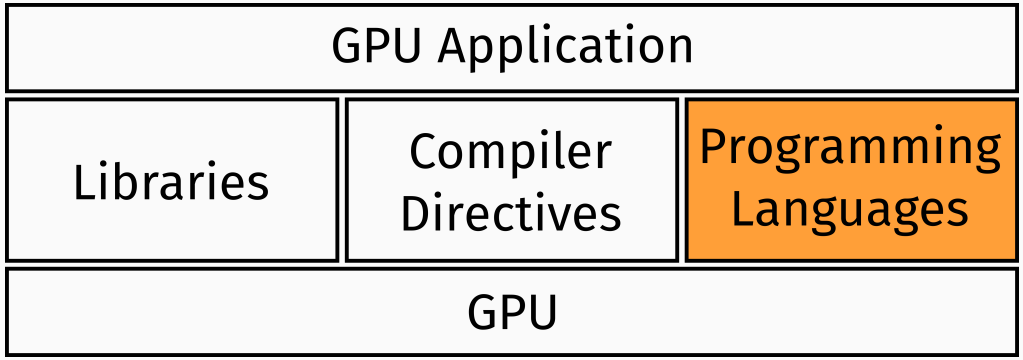
Vector sum with OpenACC:

```
#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];
    }
}
```

Programming Languages

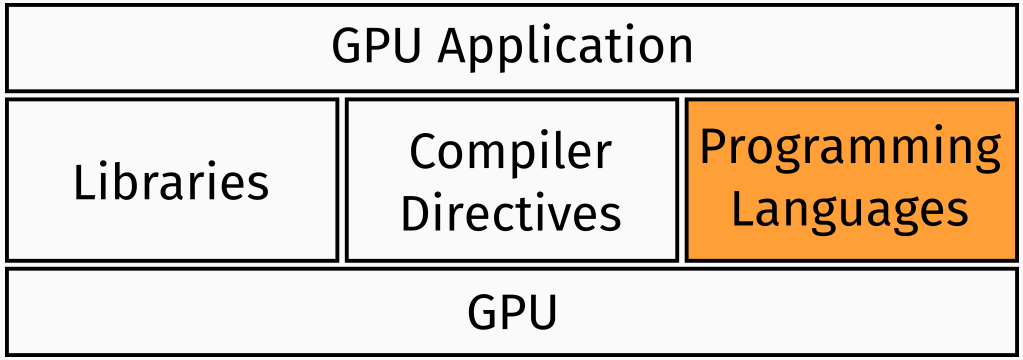


Programming Languages



- Best performance, but hard to master

Programming Languages



- Best performance, but hard to master
- Flexible

Programming Languages

CUDA implementations in:

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

CUDA C

CUDA C

Programming Model:

CUDA C

Programming Model:

- Kernels

CUDA C

Programming Model:

- Kernels
- Threads hierarchy

CUDA C

Programming Model:

- Kernels
- Threads hierarchy
- Memory hierarchy

CUDA C

Programming Model:

- Kernels
- Threads hierarchy
- Memory hierarchy
- Heterogeneous Programming

CUDA C: Kernel

Kernels:

- Kernels are C functions executed by threads
- *N threads* run *N kernels*

CUDA C: Kernel

Kernels:

- Kernels are C functions executed by threads
- *N threads* run *N kernels*
- Access thread IDs using the `threadIdx` variable

CUDA C: Kernel

Kernels:

- Kernels are C functions executed by threads
- *N threads* run *N kernels*
- Access thread IDs using the `threadIdx` variable
- Defined within the `__global__` scope
- Its type must be `void`

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

CUDA C: Kernel

Kernels:

- Configured and launched using the syntax:
  - `kernel_name<<<...>>>(...);`

CUDA C: Kernel

Kernels:

- Configured and launched using the syntax:
  - `kernel_name<<<...>>>(...);`
- Ideally executed in parallel

CUDA C: Kernel

Kernels:

- Configured and launched using the syntax:
  - `kernel_name<<<...>>>(...);`
- Ideally executed in parallel
- In practice, parallelism depends on:
  - Number of threads in a warp

CUDA C: Kernel

Kernels:

- Configured and launched using the syntax:
  - `kernel_name<<<...>>>(...);`
- Ideally executed in parallel
- In practice, parallelism depends on:
  - Number of threads in a warp
  - Execution branch coherence

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

CUDA C: Kernel

Writing and launching a 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

Writing and launching a 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);
    ...
}
```

Source: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model [Accessed in 29/07/16]

Thread Block:

- Grouping of threads

CUDA C: Thread Hierarchy

Thread Block:

- Grouping of threads
- Tridimensional: `Db = (Dx, Dy, Dz)`

CUDA C: Thread Hierarchy

Thread Block:

- Grouping of threads
- Tridimensional: `Db = (Dx, Dy, Dz)`
- Maximum size of 1024 threads

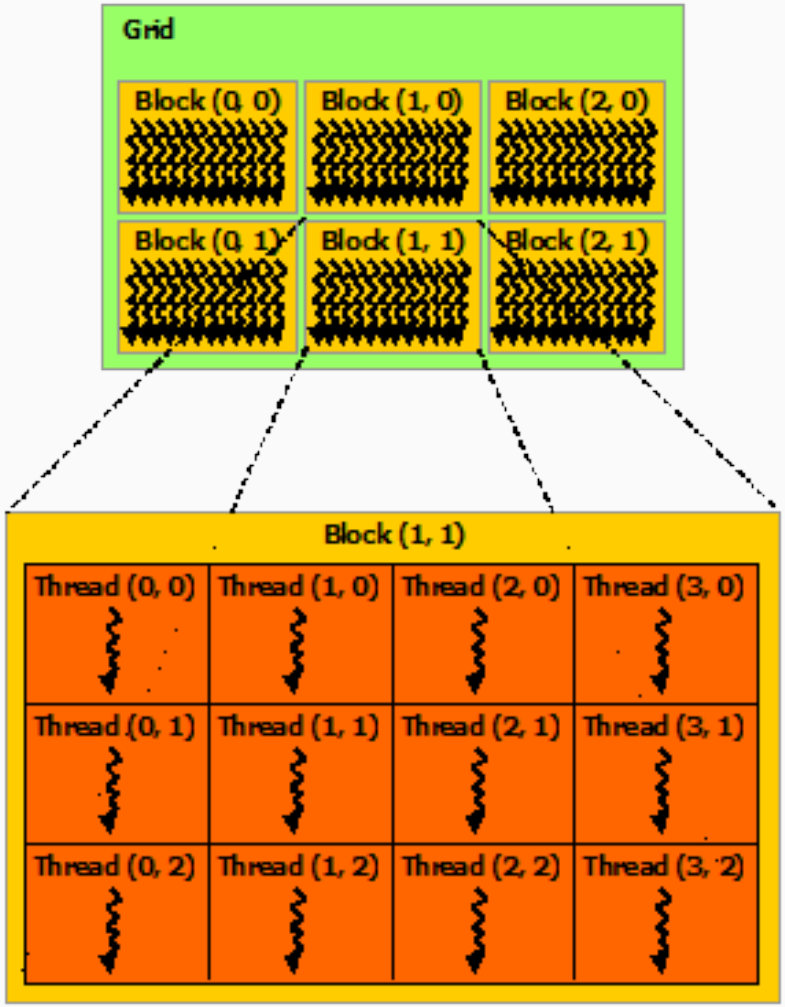
CUDA C: Thread Hierarchy

Thread Block:

- Grouping of threads
- Tridimensional: `Db = (Dx, Dy, Dz)`
- Maximum size of 1024 threads
- A Grid is the tridimensional grouping of *blocks*

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

CUDA C: Thread Hierarchy



Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

Remembering: Pascal GP100 SM

Image: `images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` [Accessed in 29/07/16]

CUDA C: Thread Hierarchy

```
__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: Thread Hierarchy

```
__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);
    ...
}
```

Source: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model [Accessed in 29/07/16]

CUDA C: Thread Hierarchy

```
__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: Thread Hierarchy

```
__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);
    ...
}
```

Source: docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model [Accessed in 29/07/16]

CUDA C: Thread Hierarchy

Configuration and Launching syntax `<<< Dg, Db, Ns, S >>>`:

- `dim3 Dg` grid dimension and size:

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

CUDA C: Thread Hierarchy

Configuration and Launching syntax `<<< Dg, Db, Ns, S >>>`:

- `dim3` `Dg` grid dimension and size:

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

- `dim3` `Db` block dimension and size:

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

CUDA C: Thread Hierarchy

Configuration and Launching syntax `<<< Dg, Db, Ns, S >>>`:

- `dim3 Dg` grid dimension and size:

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

- `dim3 Db` block dimension and size:

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

- `size_t Ns = 0`: extra bytes in shared memory
- `cudaStream_t S = 0`: CUDA stream

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

CUDA C: Memory Hierarchy

Threads access multiple memory addresses during a kernel's execution:

- Local

CUDA C: Memory Hierarchy

Threads access multiple memory addresses during a kernel's execution:

- Local
- Shared with its *block*

CUDA C: Memory Hierarchy

Threads access multiple memory addresses during a kernel's execution:

- Local
- Shared with its *block*
- Global

CUDA C: Memory Hierarchy

Threads access multiple memory addresses during a kernel's execution:

- Local
- Shared with its *block*
- Global: persists between kernels of the same application

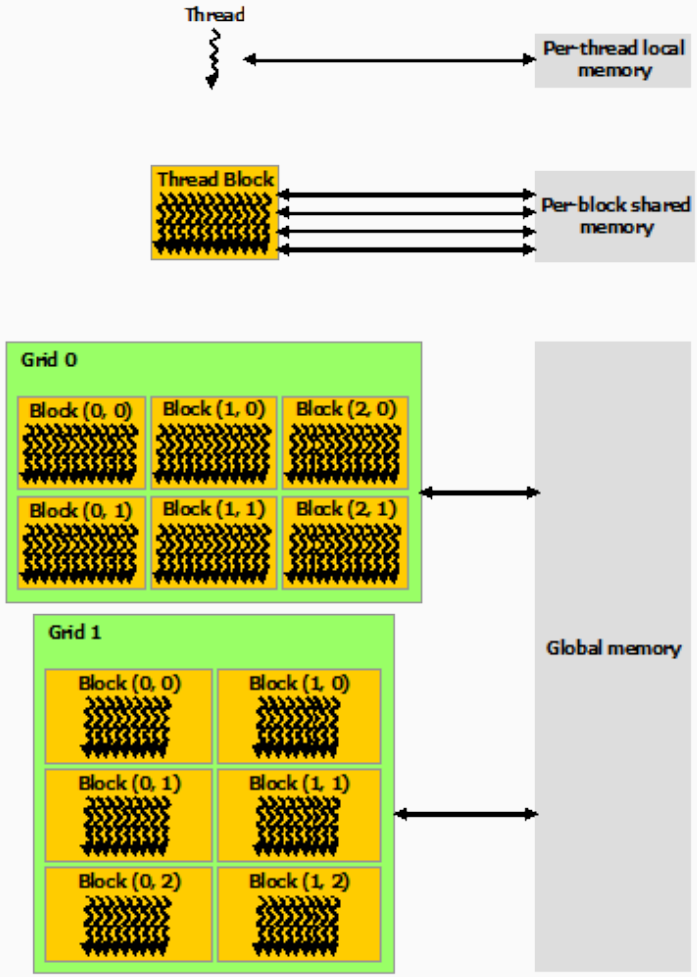
CUDA C: Memory Hierarchy

Threads access multiple memory addresses during a kernel's execution:

- Local
- Shared with its *block*
- Global: persists between kernels of the same application
- *Read-only*

Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

CUDA C: Memory Hierarchy



Source: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model` [Accessed in 29/07/16]

Host's tasks:

- Allocate device memory and resources
- Move data (bottleneck!)

Host's tasks:

- Allocate device memory and resources
- Move data (bottleneck!)
- Launch kernel

Host's tasks:

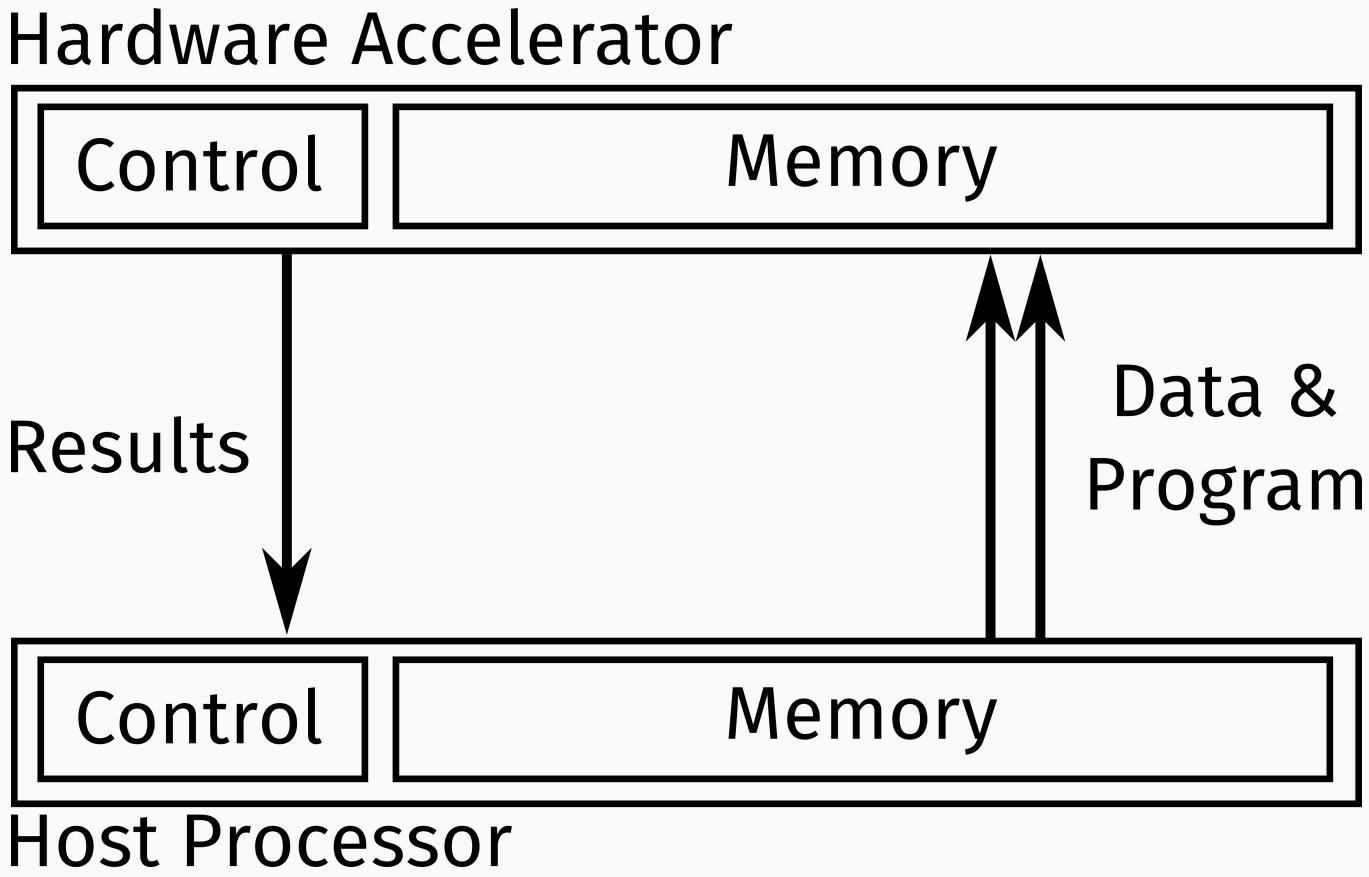
- Allocate device memory and resources
- Move data (bottleneck!)
- Launch kernel
- Move results (bottleneck!)
- Release device memory and resources

Host's tasks:

- Allocate device memory and resources
- Move data (bottleneck!)
- Launch kernel
- Move results (bottleneck!)
- Release device memory and resources

Device's tasks:

- Run kernel


Hardware Accelerator
Control
Memory
Results
Data &
Program
Control
Memory
Host Processor

Example: Vector Sum

```
__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];
    }
}
```

Source: `github.com/phrb/intro-cuda` [Accessed in 29/07/16]

Example: Vector Sum

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

Example: Vector Sum

```
#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) { ... };
```

Example: Vector Sum

```
#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;
```

Example: Vector Sum

```
#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) { ... };
```

Example: Vector Sum

```
#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) { ... };
```

Source: `github.com/phrb/intro-cuda` [Accessed in 29/07/16]

Example: Practical Tip

Always check for errors!

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

err = cudaGetLastError()

err = cudaDeviceSynchronize()
```

Example: Practical Tip

Always check for errors!

```
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);
}
```

Source: `github.com/phrb/intro-cuda` [Accessed in 29/07/16]

Hands on!

We'll run some CUDA C samples available at `github.com/phrb/intro-cuda`:

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

Hands on!

We'll run some CUDA C samples available at `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`

Hands on!

We'll run some CUDA C samples available at `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`

Resources

This presentation and all source code are available at GitHub:

- `github.com/phrb/intro-cuda`

More resources:

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

Introduction to GPU Programming with the CUDA Platform

Pedro Bruel
phrb@ime.usp.br
4th INFIERI, 2017

Instituto de Matemática e Estatística
Universidade de São Paulo

About

Pedro Bruel
phrb@ime.usp.br

Alfredo Goldman
gold@ime.usp.br

Marco Dimas Gubitoso

gubi@ime.usp.br

Part I

1. Introduction
2. Heterogeneous Computing
3. GPUs
4. CUDA Platform
5. CUDA C

Part II

6. Recapitulation
7. Compiling CUDA Applications
8. Optimization Best Practices
9. Analysing CUDA Applications
10. Otimizing CUDA Applications
11. Conclusion

Slides

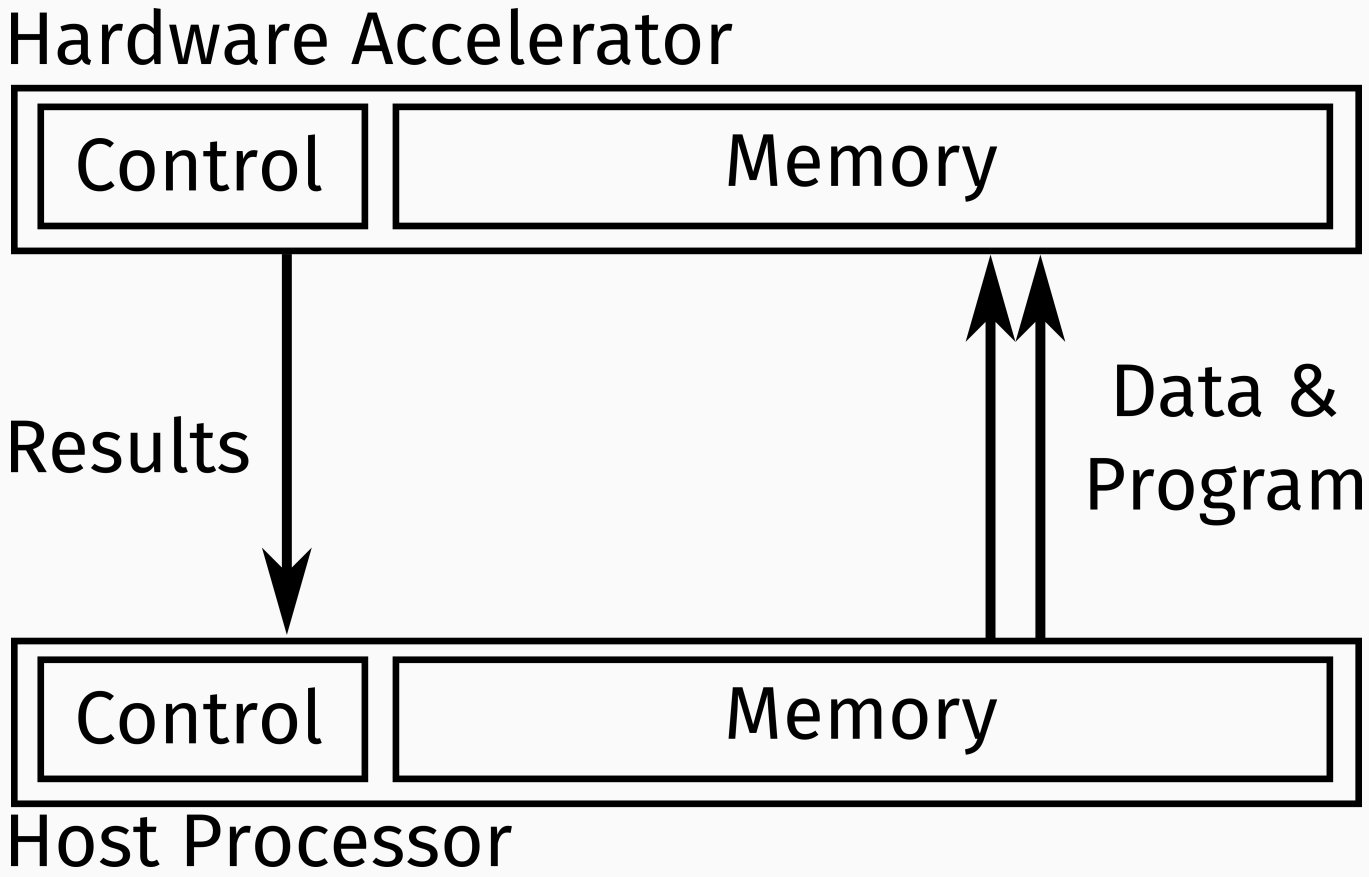
This presentation and all source code are available at GitHub:

- `github.com/phrb/intro-cuda`

Hardware Acceleration

The use of devices to accelerate computations on large datasets:

- Associated with a host processor
- Separate control and memory
- Differ on specialization and configurability
- GPUs, DSPs, FPGAs, ASICs


Hardware Accelerator
Control
Memory
Results
Data &
Program
Control
Memory
Host Processor

Percentage of systems with accelerators in the *Top500*:

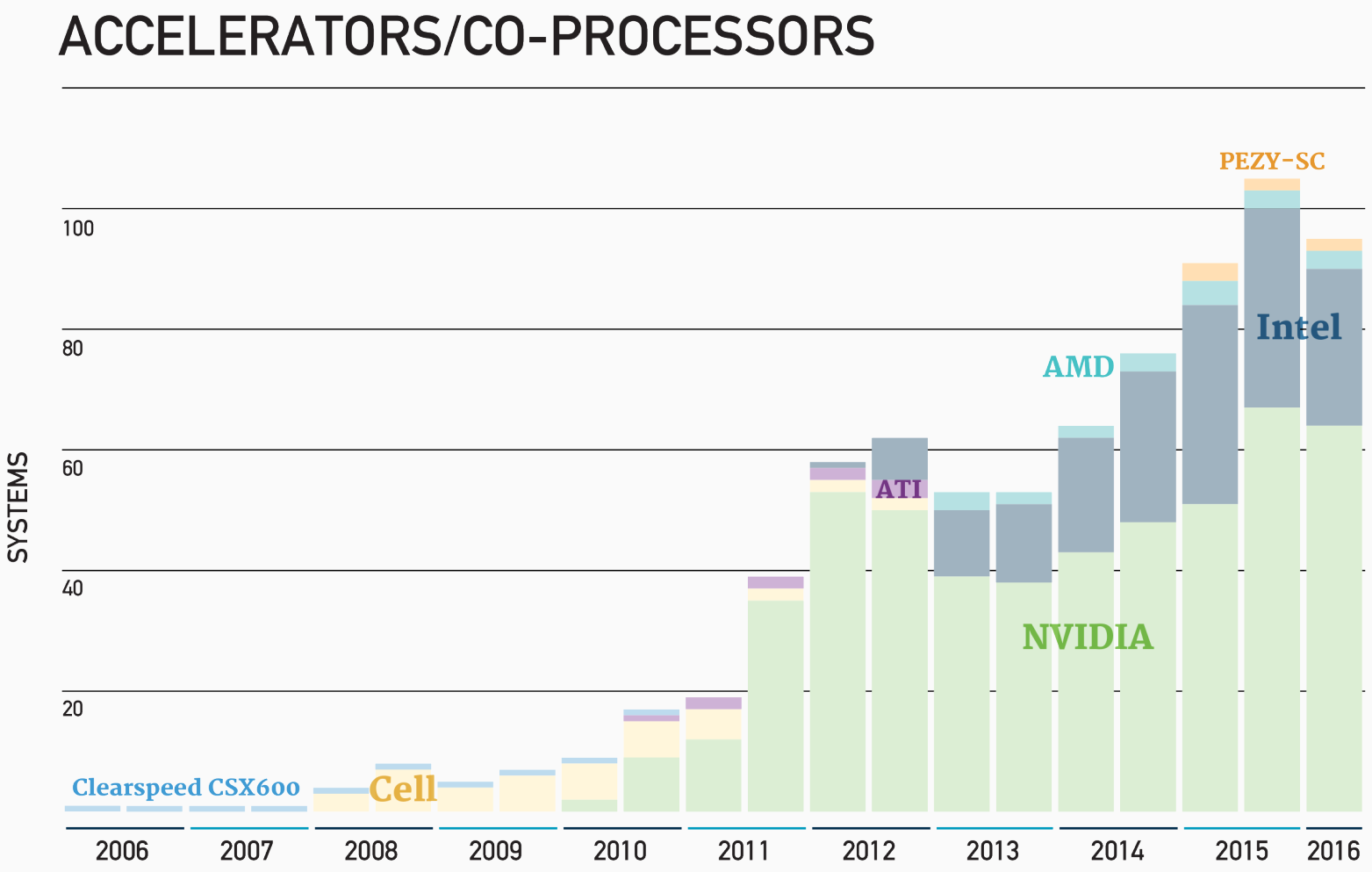


Image: `top500.org/lists/2016/06/download/TOP500_201606_Poster.pdf` [Accessed in 29/07/16]

Heterogeneous Computing

Given heterogeneous computational resources, datasets and a set of computations, how to distribute data and computations to optimize resource usage?

Image: `olcf.ornl.gov/titan` [Accessed in 29/07/16]

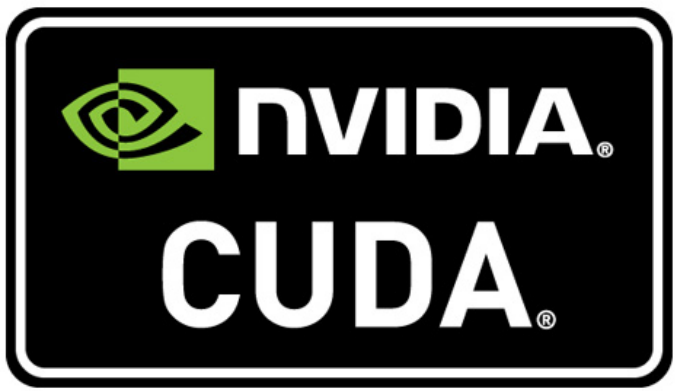
Graphics Processing Units

Graphics Processing Units

Originally specialized for graphical processing, optimized for large datasets and high throughput:

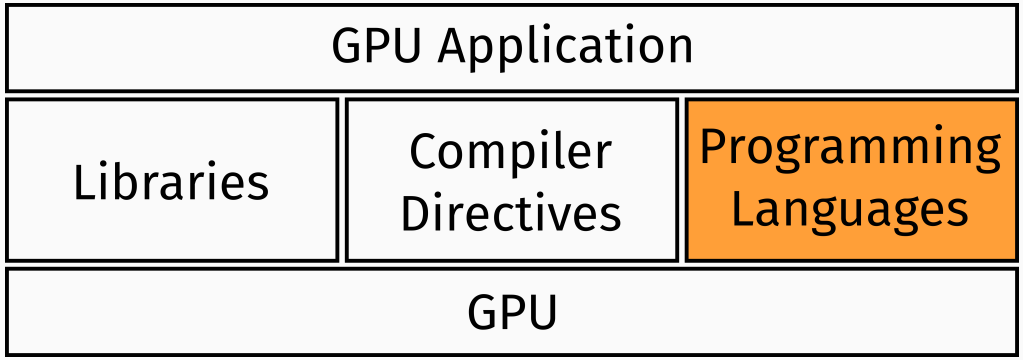
- Small caches (*kilobytes*)
- No branch prediction
- Thousands of higher latency ALUs
- Execution pipelines
- 114 688 concurrent threads in Pascal architecture

CUDA Platform



- A platform for parallel computing
- *Application Programming Interface* (API)
- CUDA *Toolkit*

Programming Languages



- Best performance, but hard to master
- Flexible

CUDA C

Programming Model:

- Kernels
- Threads hierarchy
- Memory hierarchy
- Heterogeneous Programming

Example: Vector Sum

```
#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) { ... };
```


Source: github.com/phrb/intro-cuda [Accessed in 29/07/16]

Hands on!

We'll run some CUDA C samples available at `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`

Example: Practical Tip

Always check for errors!

```
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);
}
```

Source: `github.com/phrb/intro-cuda` [Accessed in 29/07/16]

Nvidia CUDA Compiler (nvcc)

Nvidia CUDA Compiler (nvcc)

- Proprietary system

Nvidia CUDA Compiler (nvcc)

- Proprietary system
- Uses the host's C++ compiler

Nvidia CUDA Compiler (nvcc)

- Proprietary system
- Uses the host's C++ compiler
- Compiles CUDA code to *Parallel Thread Execution* (PTX ISA)

Nvidia CUDA Compiler (nvcc)

- Proprietary system
- Uses the host's C++ compiler
- Compiles CUDA code to *Parallel Thread Execution* (PTX ISA)
- Host's code + Device's code $\rightarrow$ fatbinary

nvcc: CUDA programming model

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

nvcc: CUDA programming model

- SIMD, SIMT
- *Single Program Multiple Data* (SPMD)
- Device: parallel and independent threads
- *Host*: does not interfere

nvcc: Compilation Trajectory

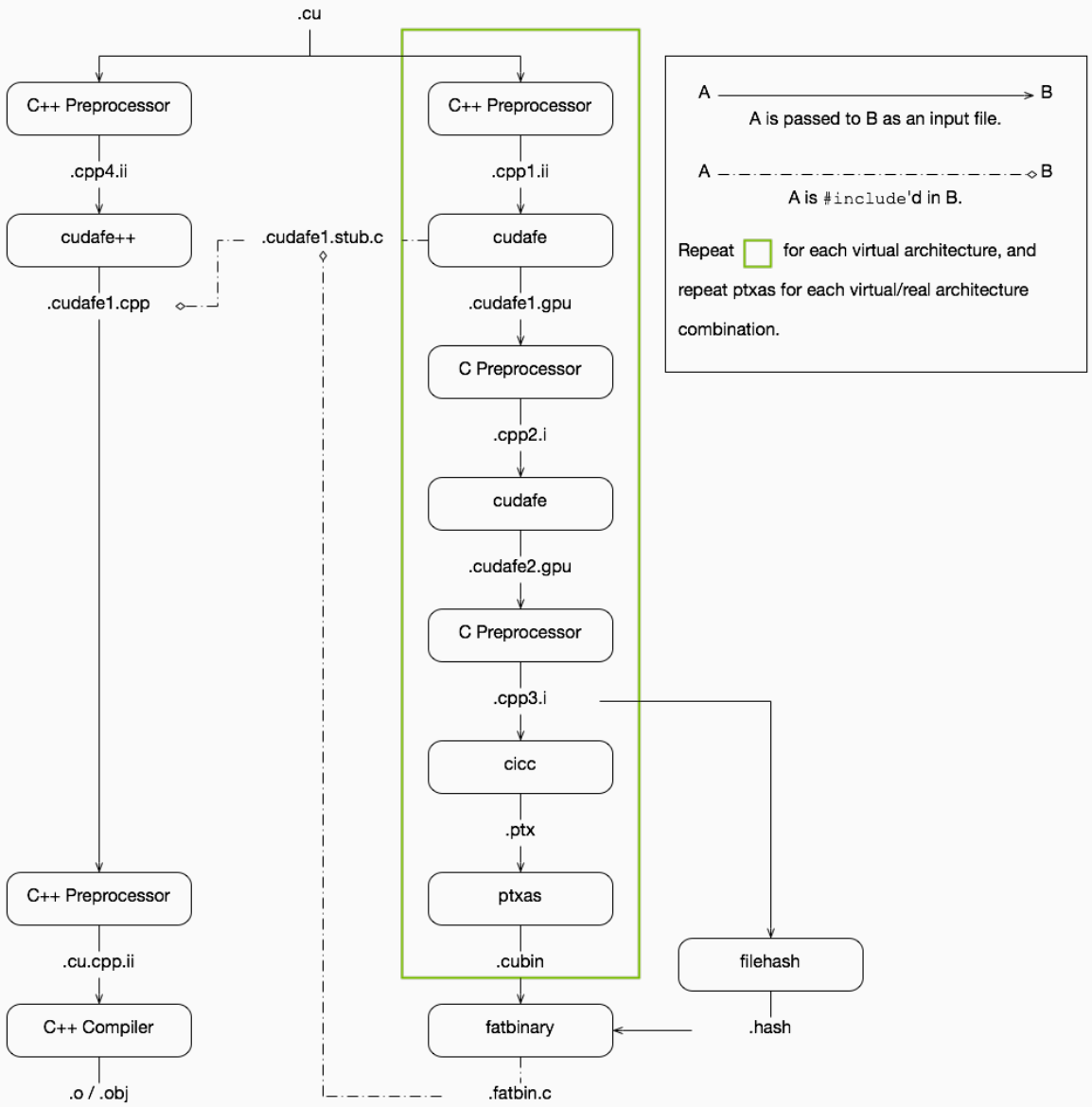
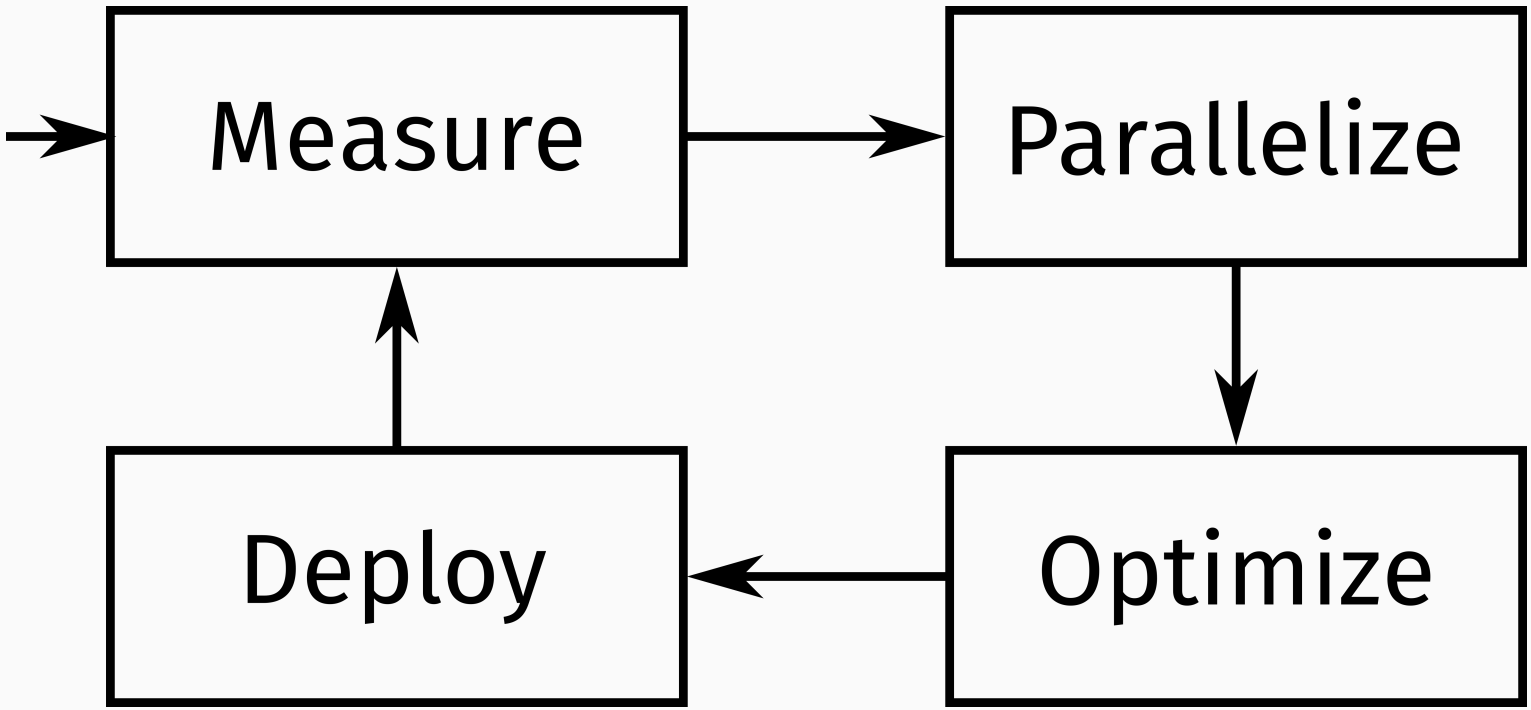


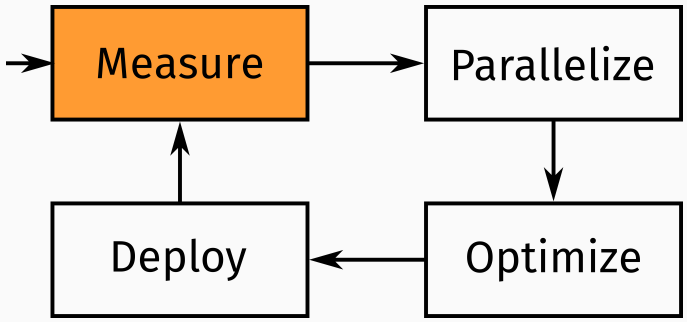
Image: `docs.nvidia.com/cuda/cuda-compiler-driver-nvcc` [Accessed in 29/07/16]

Measure, Parallelize, Optimize, Deploy

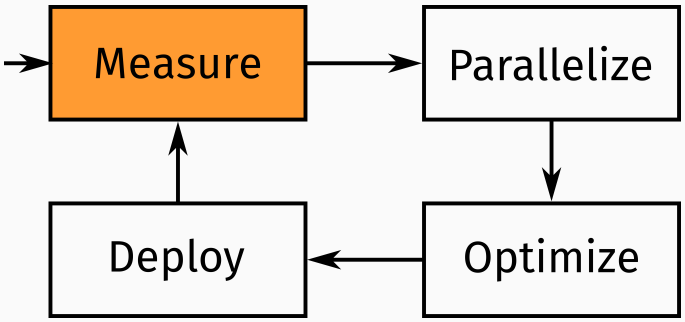


Source: `docs.nvidia.com/cuda/cuda-c-best-practices-guide` [Accessed in 29/07/16]

Measure



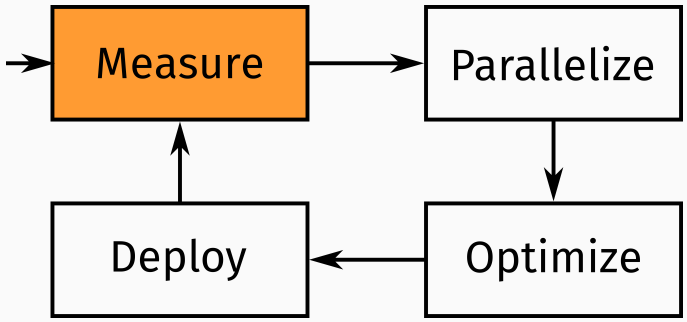
Measure



Given a sequential program:

- Which functions do the heavy lifting (bottlenecks)?

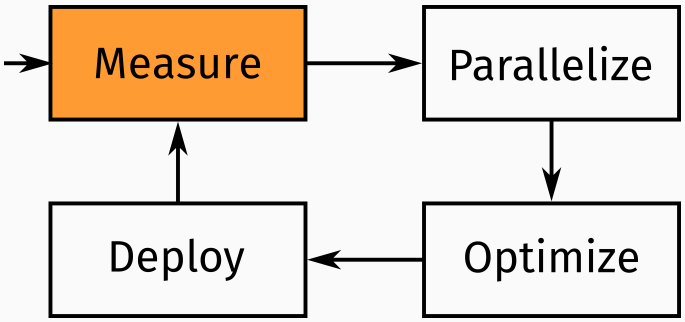
Measure



Given a sequential program:

- Which functions do the heavy lifting (bottlenecks)?
- Can they be parallelized?

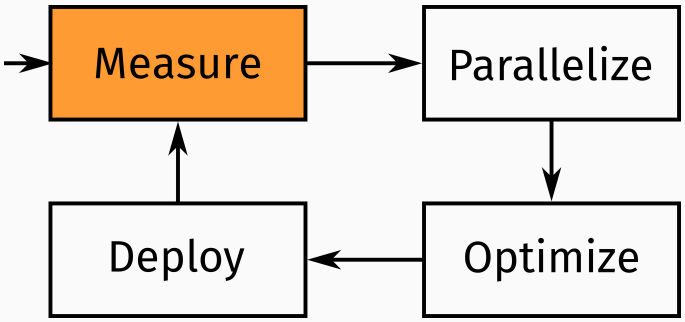
Measure



Given a sequential program:

- Which functions do the heavy lifting (bottlenecks)?
- Can they be parallelized?
- How does your program behaves when:
  - The work is divided between more processes? (strong scaling)

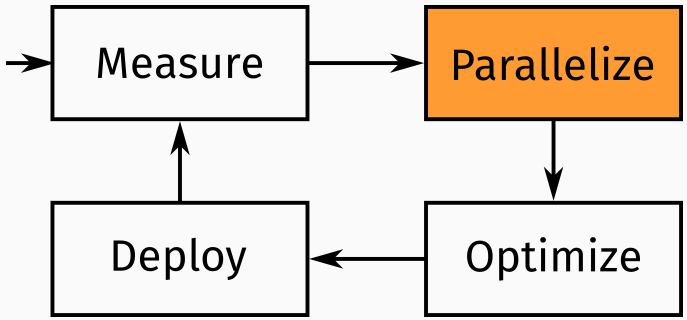
Measure



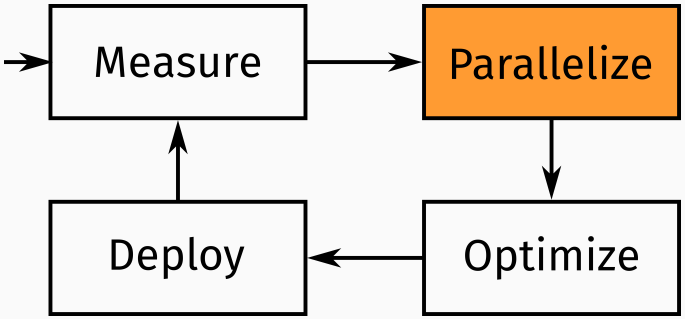
Given a sequential program:

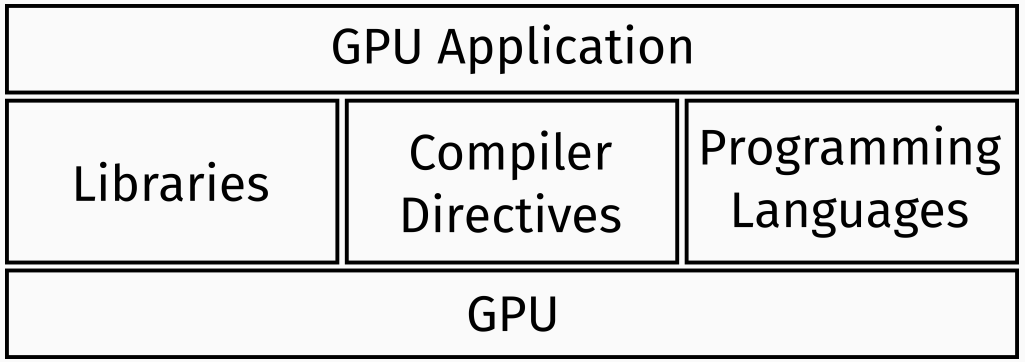
- Which functions do the heavy lifting (bottlenecks)?
- Can they be parallelized?
- How does your program behaves when:
  - The work is divided between more processes? (strong scaling)
  - There is more work to be done? (weak scaling)

Parallelize

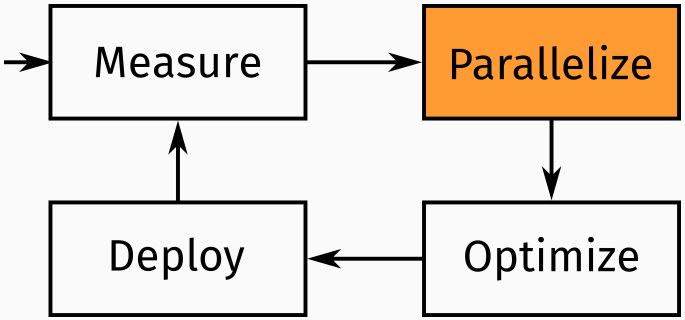


Parallelize





Parallelize



AmgX

cuDNN

cuFFT

IndeX Framework

nvGRAPH

GIE

NPP

FFmpeg

CHOLMOD

CULA Tools

MAGMA

IMSL Fortran Numerical Library

cuSOLVER

cuSPARSE

cuBLAS

ArrayFire

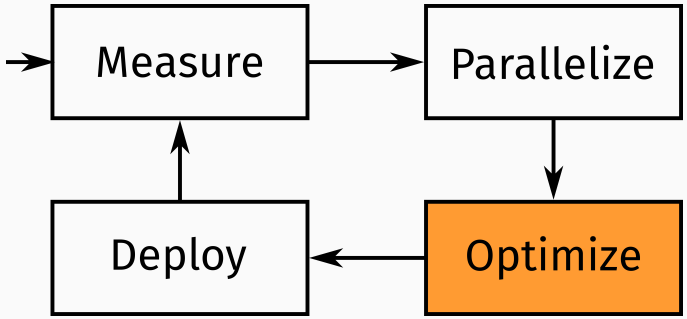
cuRAND

CUDA Math Library

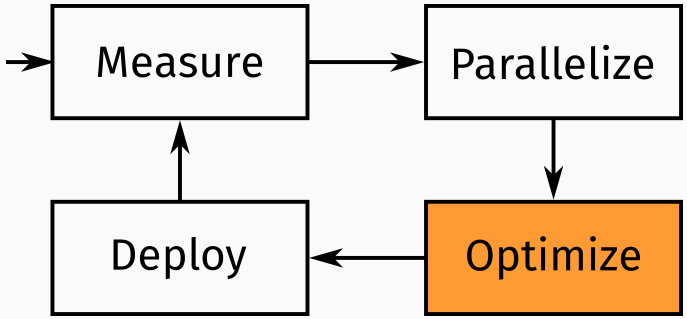
Thrust

NVBIO

Optimize



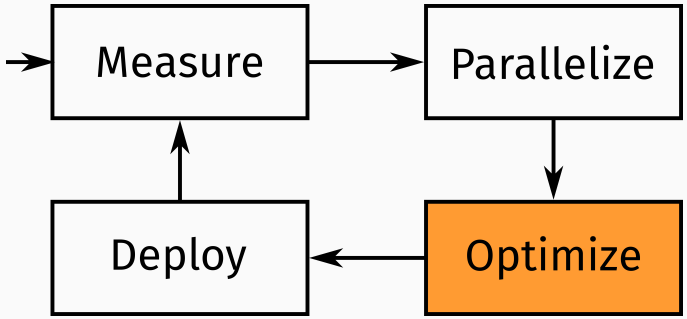
Optimize



Optimization:

- Cyclical

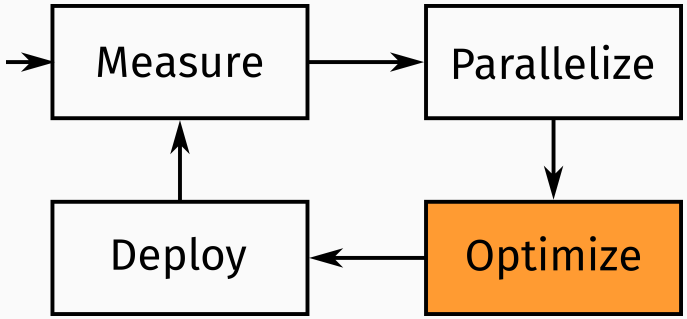
Optimize



Optimization:

- Cyclical and incremental

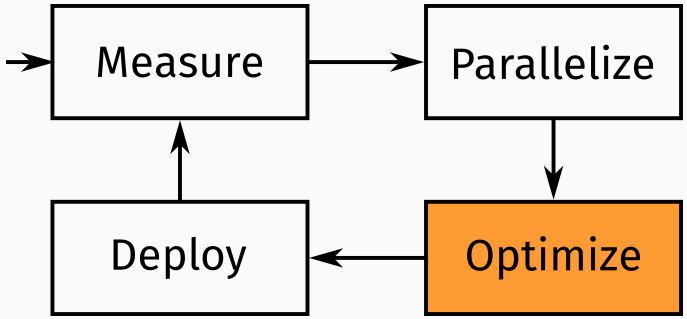
Optimize



Optimization:

- Cyclical and incremental
- Applicable to different levels

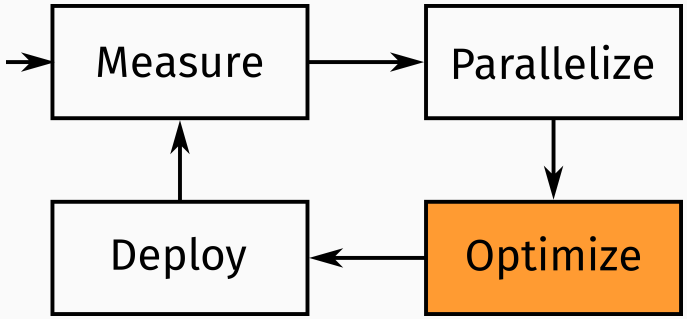
Optimize



Optimization:

- Cyclical and incremental
- Applicable to different levels
- Tools are very important

Optimize

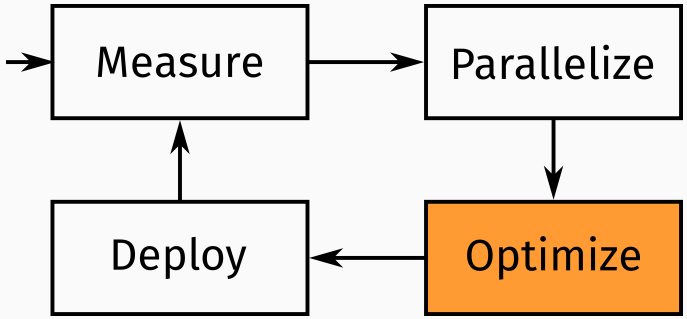


Optimization:

- Cyclical and incremental
- Applicable to different levels
- Tools are very important

Efficiency with algorithms,

Optimize

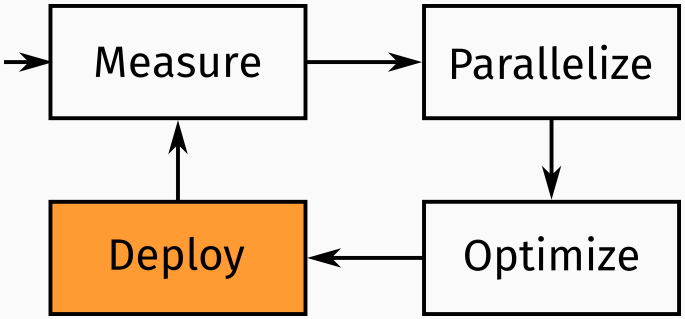


Optimization:

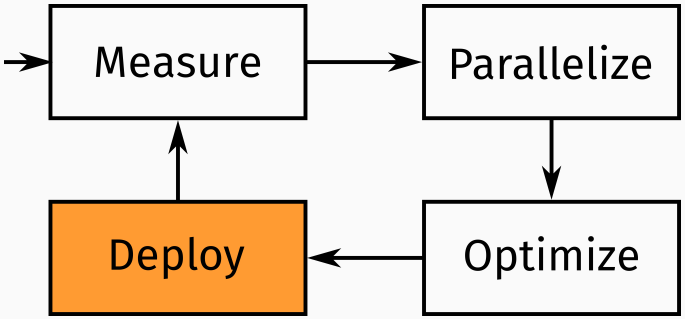
- Cyclical and incremental
- Applicable to different levels
- Tools are very important

Efficiency with algorithms, performance with data structures

Deploy

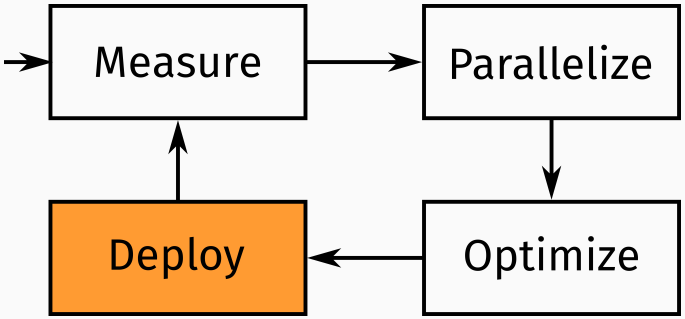


Deploy



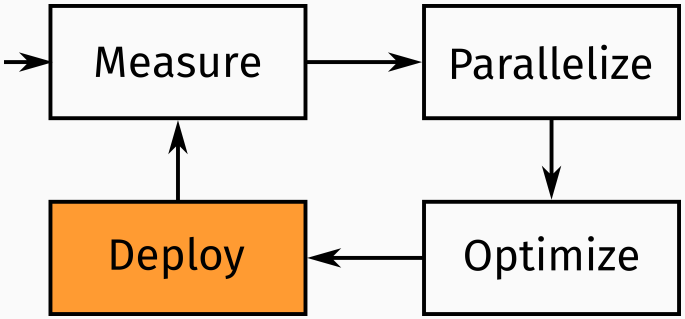
- Prepare application for measurement

Deploy



- Prepare application for measurement
- Incremental optimizations and changes

Deploy



- Prepare application for measurement
- Incremental optimizations and changes
- How much is it still possible to optimize?

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential
- Coarse grained

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential
- Coarse grained

Analysis or profiling:

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential
- Coarse grained

Analysis or profiling:

- What are the hotspots (*bottlenecks*)?

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential
- Coarse grained

Analysis or profiling:

- What are the hotspots (*bottlenecks*)?
- Can they be parallelized?

Analysing CUDA Applications

Parallel and distributed computing is increasingly important, but legacy code is often:

- Sequential
- Coarse grained

Analysis or profiling:

- What are the hotspots (*bottlenecks*)?
- Can they be parallelized?
- What are the relevant workloads?

Host and Device

Differences between host and device:

- Threading model

Host and Device

Differences between host and device:

- Threading model
- Memory

Host and Device

Differences between host and device:

- Threading model
- Memory

What to run on GPUs?

Host and Device

Differences between host and device:

- Threading model
- Memory

What to run on GPUs?

- Large datasets

Host and Device

Differences between host and device:

- Threading model
- Memory

What to run on GPUs?

- Large datasets
- Independent computations

Host and Device

Differences between host and device:

- Threading model
- Memory

What to run on GPUs?

- Large datasets
- Independent computations

Always consider the impact of data transferences!

Profiling

Monitoring code during runtime obtaining information for optimization

Monitoring code during runtime obtaining information for optimization

Code instrumentation:

- Capture events

Profiling

Monitoring code during runtime obtaining information for optimization

Code instrumentation:

- Capture events
- Generate data

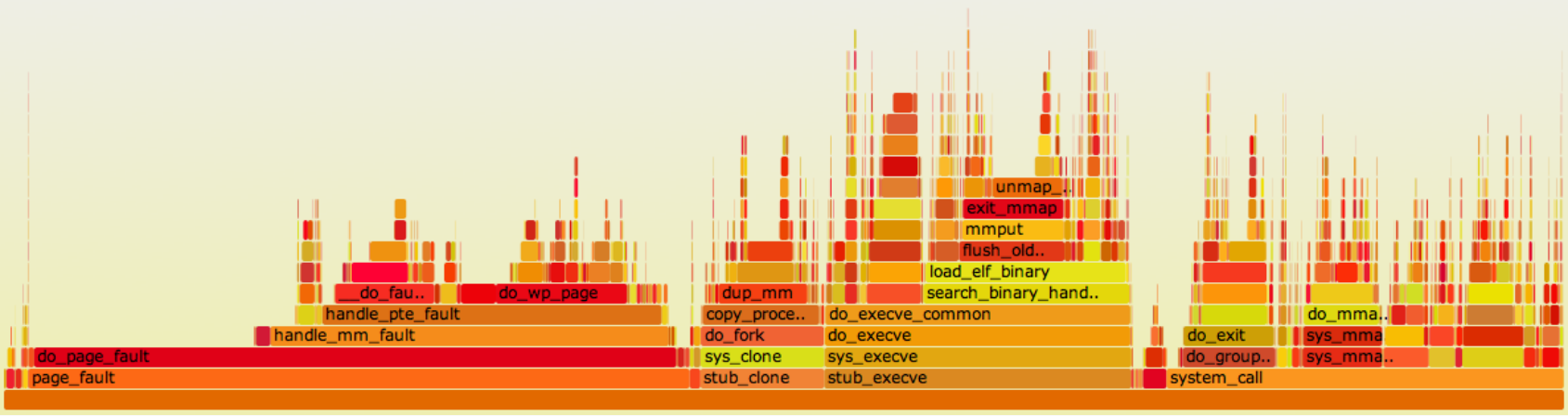
Profiling

Monitoring code during runtime obtaining information for optimization

Code instrumentation:

- Capture events
- Generate data
- Tools

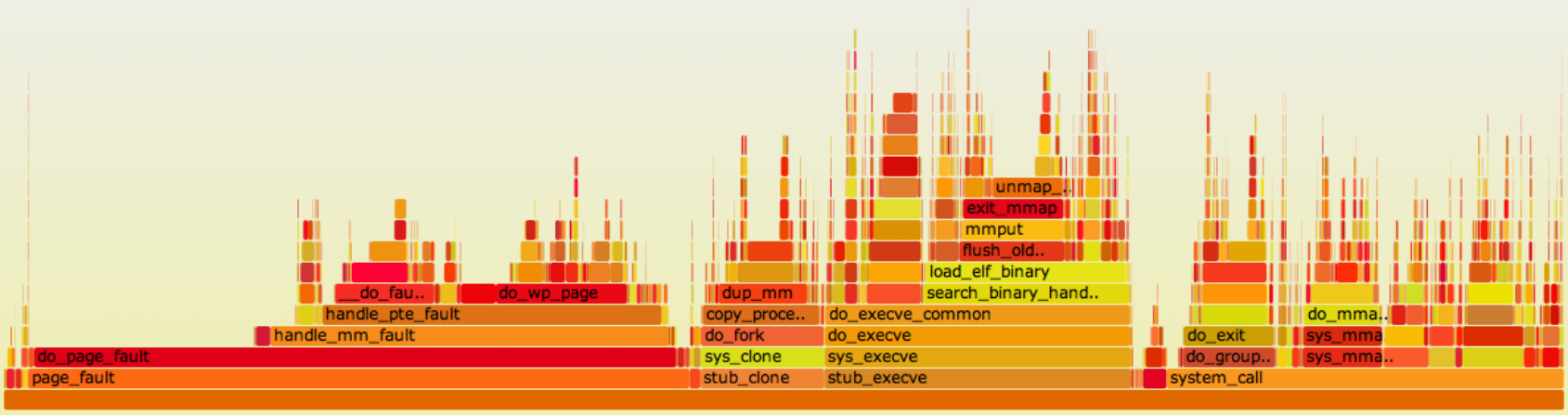
Profiling: Flamegraphs



Help analyse:

- Performance *bottlenecks*, *hotspots*

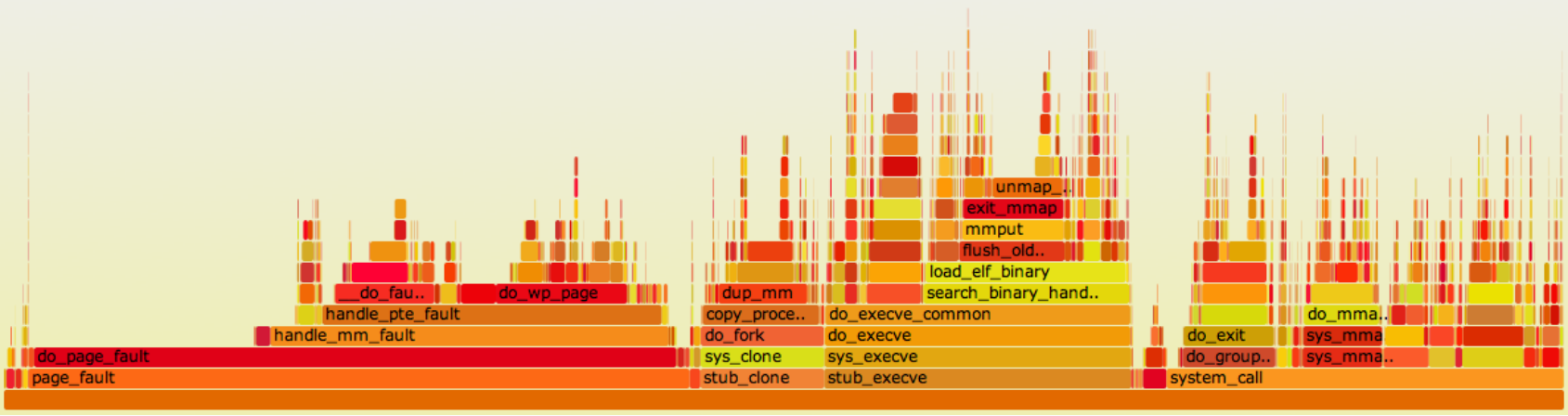
Profiling: Flamegraphs



Help analyse:

- Performance *bottlenecks*, *hotspots*
- Frequency and duration of function calls: *On-CPU*

Profiling: Flamegraphs



Help analyse:

- Performance *bottlenecks*, *hotspots*
- Frequency and duration of function calls: *On-CPU*
- Idle time (*waiting*): *Off-CPU*

Profiling: nvprof

A command line profiler that enables collecting CUDA-related events:

Profiling: nvprof

A command line profiler that enables collecting CUDA-related events:

- CPU and GPU

Profiling: nvprof

A command line profiler that enables collecting CUDA-related events:

- CPU and GPU
- Kernel execution

Profiling: nvprof

A command line profiler that enables collecting CUDA-related events:

- CPU and GPU
- Kernel execution
- Memory usage and transferences

Profiling: nvprof

A command line profiler that enables collecting CUDA-related events:

- CPU and GPU
- Kernel execution
- Memory usage and transferences
- CUDA API

Profiling: nvprof

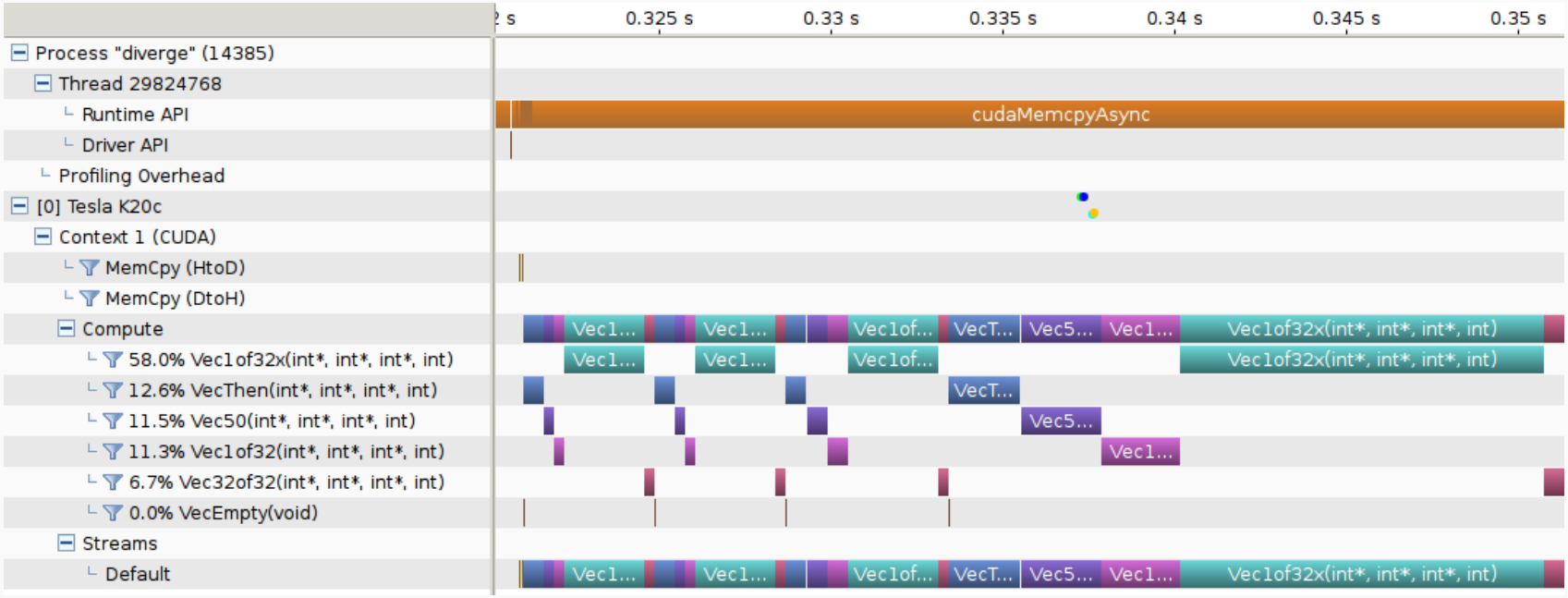
A command line profiler that enables collecting CUDA-related events:

- CPU and GPU
- Kernel execution
- Memory usage and transferences
- CUDA API

Generates data for posterior visualization

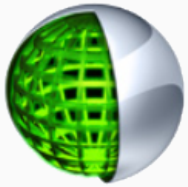
Source: `docs.nvidia.com/cuda/profiler-users-guide` [Accessed in 29/07/16]

Profiling: nvvp

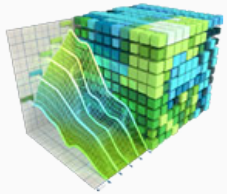


Source: `docs.nvidia.com/cuda/profiler-users-guide` [Accessed in 29/07/16]

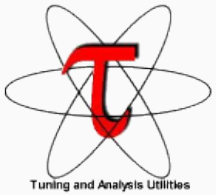
Profiling: Tools



NVIDIA® Nsight™



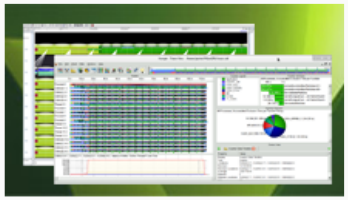
NVIDIA Visual Profiler



TAU Performance System®

VampirTrace

The PAPI CUDA Component



The NVIDIA CUDA Profiling Tools Interface

Source: `developer.nvidia.com/performance-analysis-tools` [Accessed in 29/07/16]

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints
- Different threads

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints
- Different threads
- CUDA `gdb` and `memcheck`

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints
- Different threads
- CUDA `gdb` and `memcheck`

Help fixing:

- Memory leaks

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints
- Different threads
- CUDA `gdb` and `memcheck`

Help fixing:

- Memory leaks
- Control flow problems

Debugging

Monitoring code during runtime obtaining information for error correction (bugs)

Code instrumentation:

- Step-by-step execution
- Breakpoints
- Different threads
- CUDA `gdb` and `memcheck`

Help fixing:

- Memory leaks
- Control flow problems
- ...

Different abstraction and priority levels:

Optimizing CUDA Applications

Different abstraction and priority levels:

- Memory

Optimizing CUDA Applications

Different abstraction and priority levels:

- Memory
- Control flow

Optimizing CUDA Applications

Different abstraction and priority levels:

- Memory
- Control flow
- Execution configuration

Optimizing CUDA Applications

Different abstraction and priority levels:

- Memory
- Control flow
- Execution configuration
- Instruction selection

Memory Optimizations

The most important for high performance

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations
  - Create and destroy intermediate data structures

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations
  - Create and destroy intermediate data structures
  - Grouped (*coalescent*) memory accesses

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations
  - Create and destroy intermediate data structures
  - Grouped (*coalescent*) memory accesses
- Host:

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations
  - Create and destroy intermediate data structures
  - Grouped (*coalescent*) memory accesses
- Host:
  - Delegate computations

Memory Optimizations

The most important for high performance

Objectives:

- Maximize bandwidth
- Minimize transferences between device and host

Best practices:

- Device:
  - Prioritize computations
  - Create and destroy intermediate data structures
  - Grouped (*coalescent*) memory accesses
- Host:
  - Delegate computations
  - Minimize data transferences

Control Flow Optimizations

High priority for high performance

Control Flow Optimizations

High priority for high performance

Objectives:

- Avoid Control Flow divergence inside warps

Control Flow Optimizations

High priority for high performance

Objectives:

- Avoid Control Flow divergence inside warps
- `if`, `switch`, `do`, `for`, `while`

Control Flow Optimizations

High priority for high performance

Objectives:

- Avoid Control Flow divergence inside warps
- `if`, `switch`, `do`, `for`, `while`
- Use `threadIdx` efficiently

Execution Configuration Optimizations

How to use the device's potential?

Execution Configuration Optimizations

How to use the device's potential?

- Maximize occupancy

Execution Configuration Optimizations

How to use the device's potential?

- Maximize occupancy
- Efficiently allocate blocks and threads

Execution Configuration Optimizations

How to use the device's potential?

- Maximize occupancy
- Efficiently allocate blocks and threads
- Independent kernel executions

Instruction Selection Optimizations

Low priority for high performance:

Instruction Selection Optimizations

Low priority for high performance:

- Low level

Instruction Selection Optimizations

Low priority for high performance:

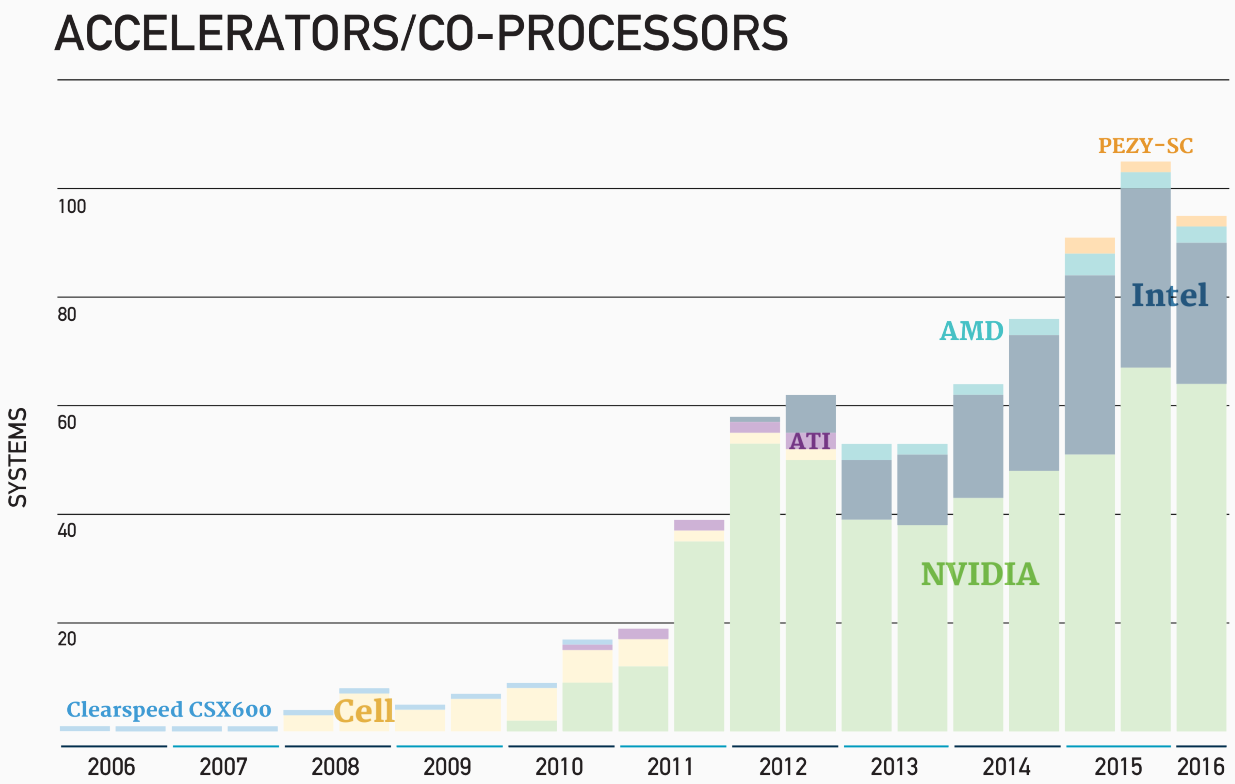
- Low level
- Applied to hotspots

Instruction Selection Optimizations

Low priority for high performance:

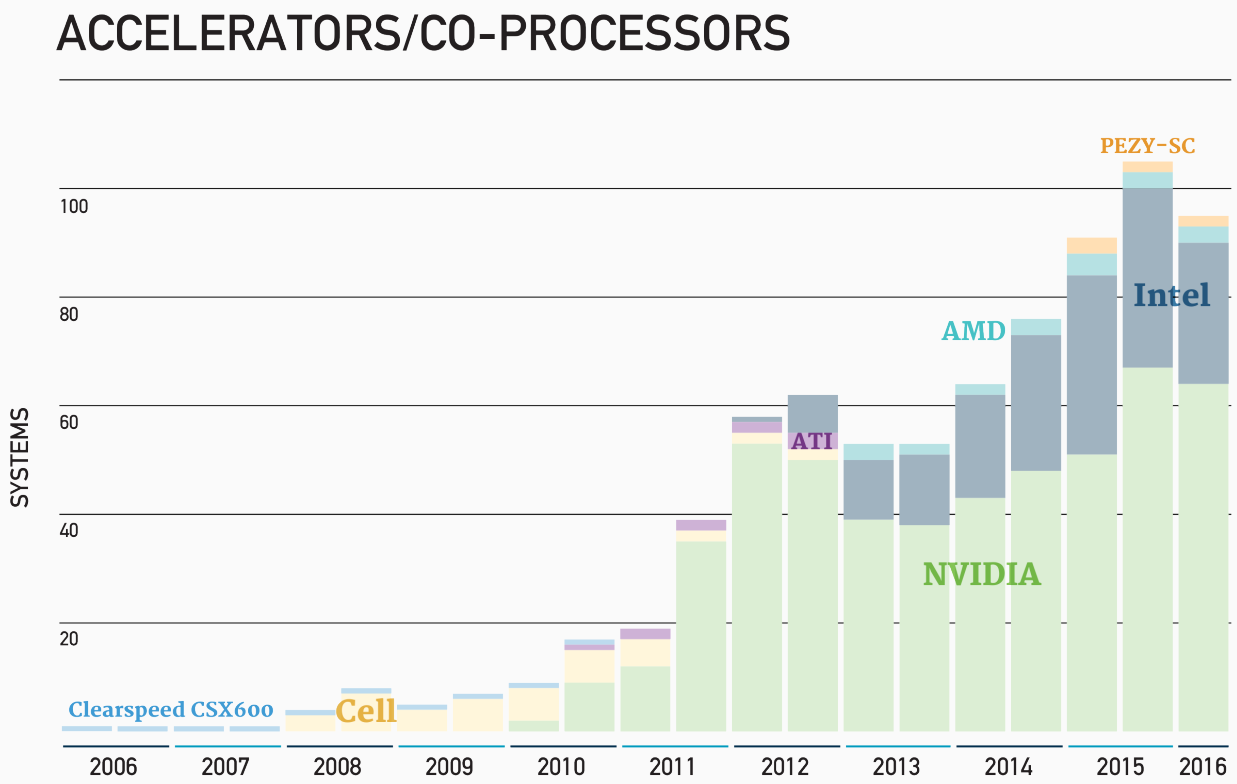
- Low level
- Applied to hotspots
- Applied by experts
- Only after exhausting other optimizations

Conclusion



Achieve high performance using accelerators

Conclusion



Achieve high performance using accelerators (GPUs)

Conclusion

Conclusion

Implementation, analysis and optimization costs

Conclusion

Implementation, analysis and optimization costs

CUDA Platform tools

Resources

This presentation and all source code are available at GitHub:

- `github.com/phrb/intro-cuda`

More resources:

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

Introduction to GPU Programming with the CUDA Platform

Pedro Bruel
phrb@ime.usp.br
4th INFIERI, 2017

Instituto de Matemática e Estatística
Universidade de São Paulo