Introduction to CUDA

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

What?

What?

[1] Graphics Processing Unit

What?

- **C**ompute **U**nified **D**evice **A**rchitecture
- A C/C++ SDK to accelerate algorithms on NVIDIA GPUs[1]
- A SIMT (Single Instruction Multiple Thread) model
  - SIMD combinded with multithreading
- First released in 2006

[1] Graphics Processing Unit

GPUs

Figure 1: 1993 - Hercules VESA card

GPUs

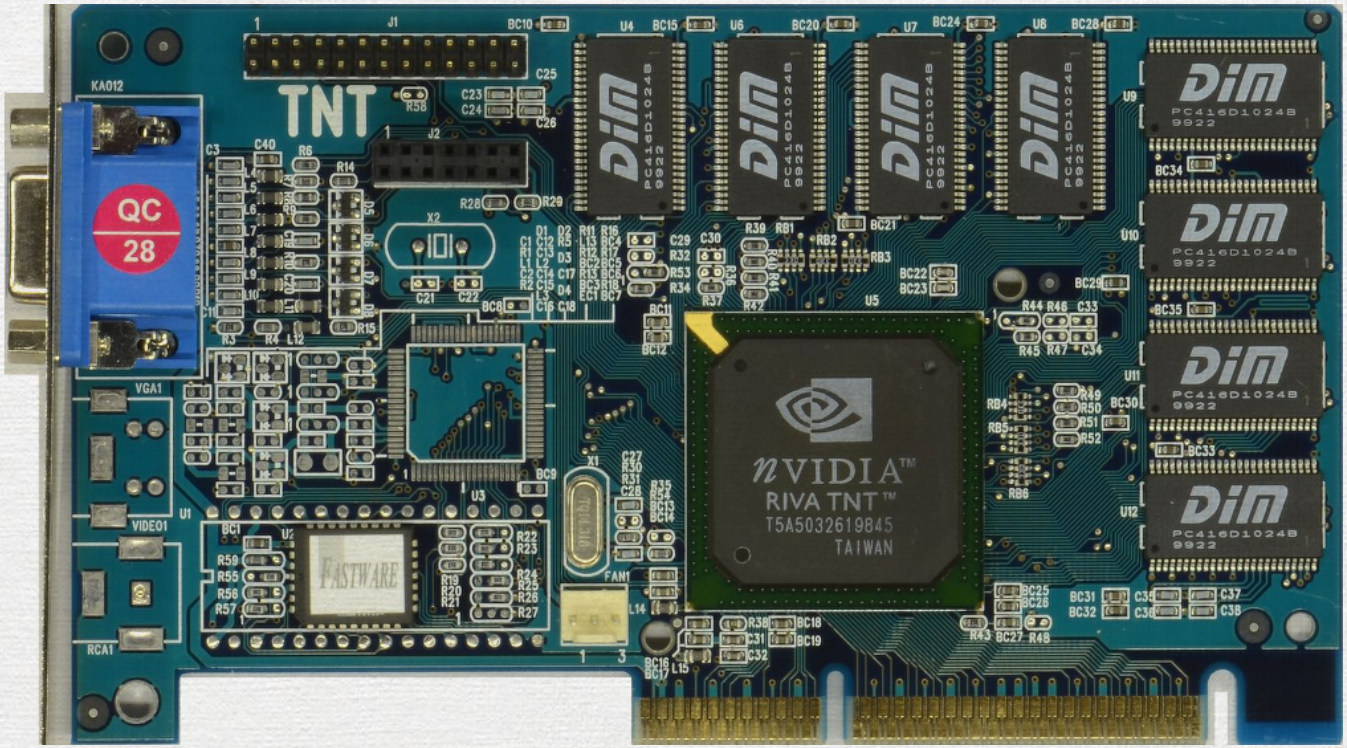


Figure 2: 2000 - RivaTNT 2:2:2 pixel:texture mapping:render output units@90MHz, 16MB RAM @100MHz

GPUs

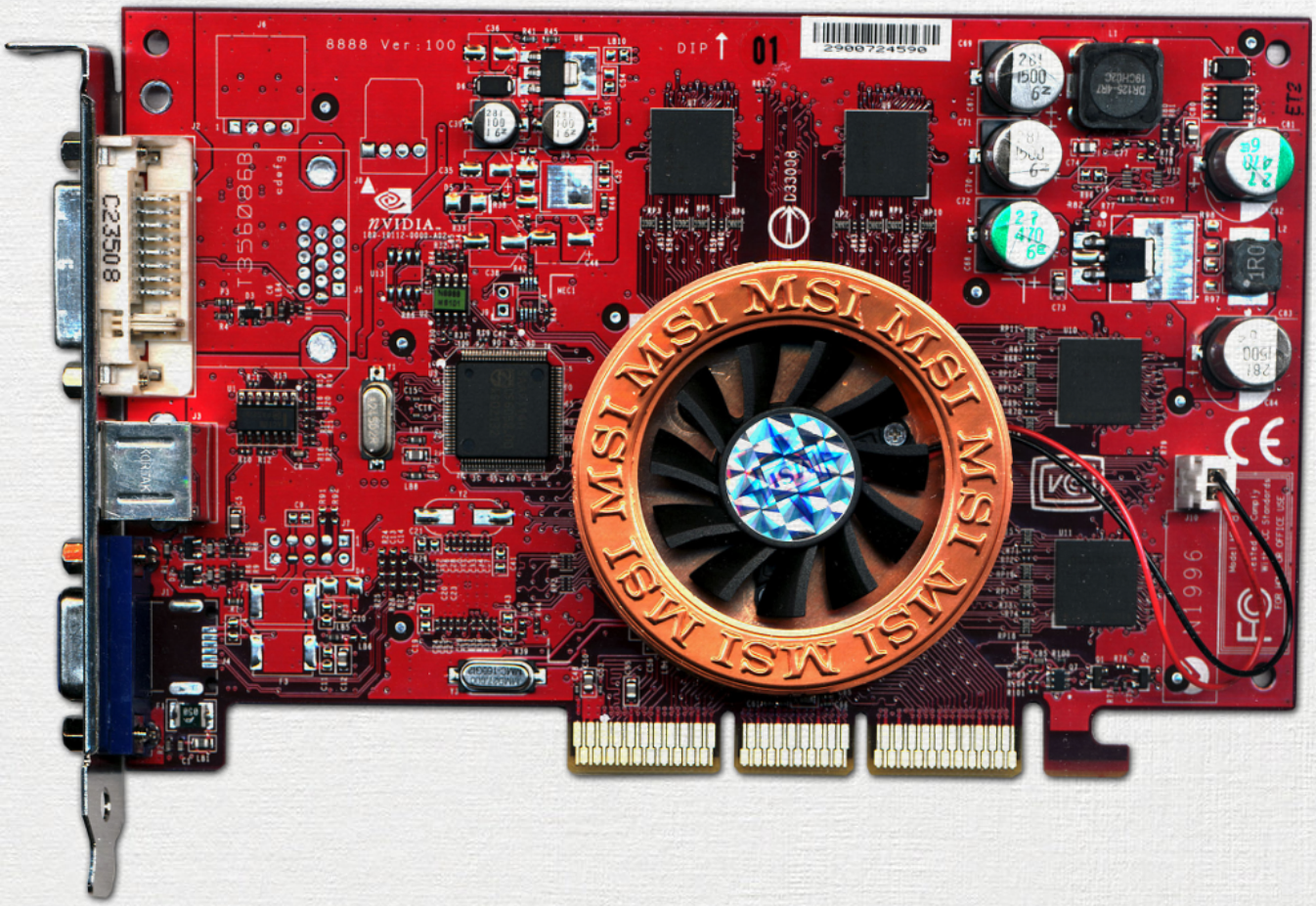


Figure 3: 2001 - GeForce MX440 8x: 2:4:2 @275MHz, 64MB RAM@250MHz

GPUs

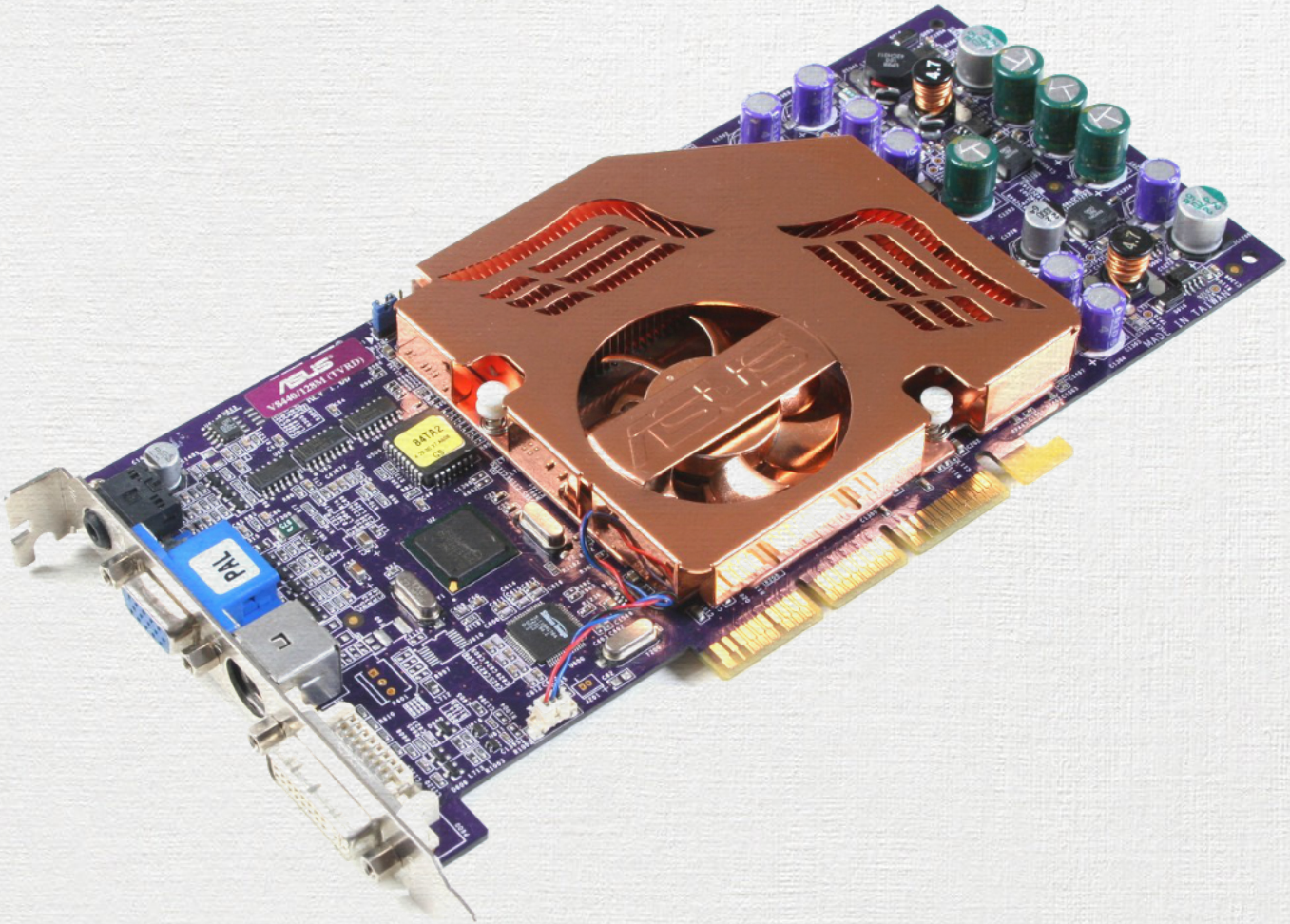


Figure 4: 2002 - GeForce Ti4400: 4:2:8:4 pixel shader:verter shader:texture mapping:render output @275MHz, 128MB RAM@275MHz

GPUs

Figure 5: GeForce GTX 1080: 2560:160:64 shader processor:texture mapping:render output @1733MHz (1800MHz boosted), 8GB RAM@10000MT/s. 8228 GFLOPS.

GPUs

Figure 6: GeForce RTX 2080 Ti: 4352:272:88:544:68 shader processor:texture mapping:render output:tensor cores:raytracing cores @1350MHz (1545MHz boosted), 11GB RAM@14000MT/s. 11750 GFLOPS, 94003 Tensor GFLOPS

GPUs

Figure 7: GeForce RTX 3090: 10496:328:112:328:82 shader processor:texture mapping:render output:tensor cores:raytracing cores @1395MHz (1695MHz boosted), 24GB RAM@19500MT/s. 35580 GFLOPS, 285480 Tensor GFLOPS

GPUs

Figure 8: GTX 1080

GPUs

Figure 9: Tesla V100

GPUs

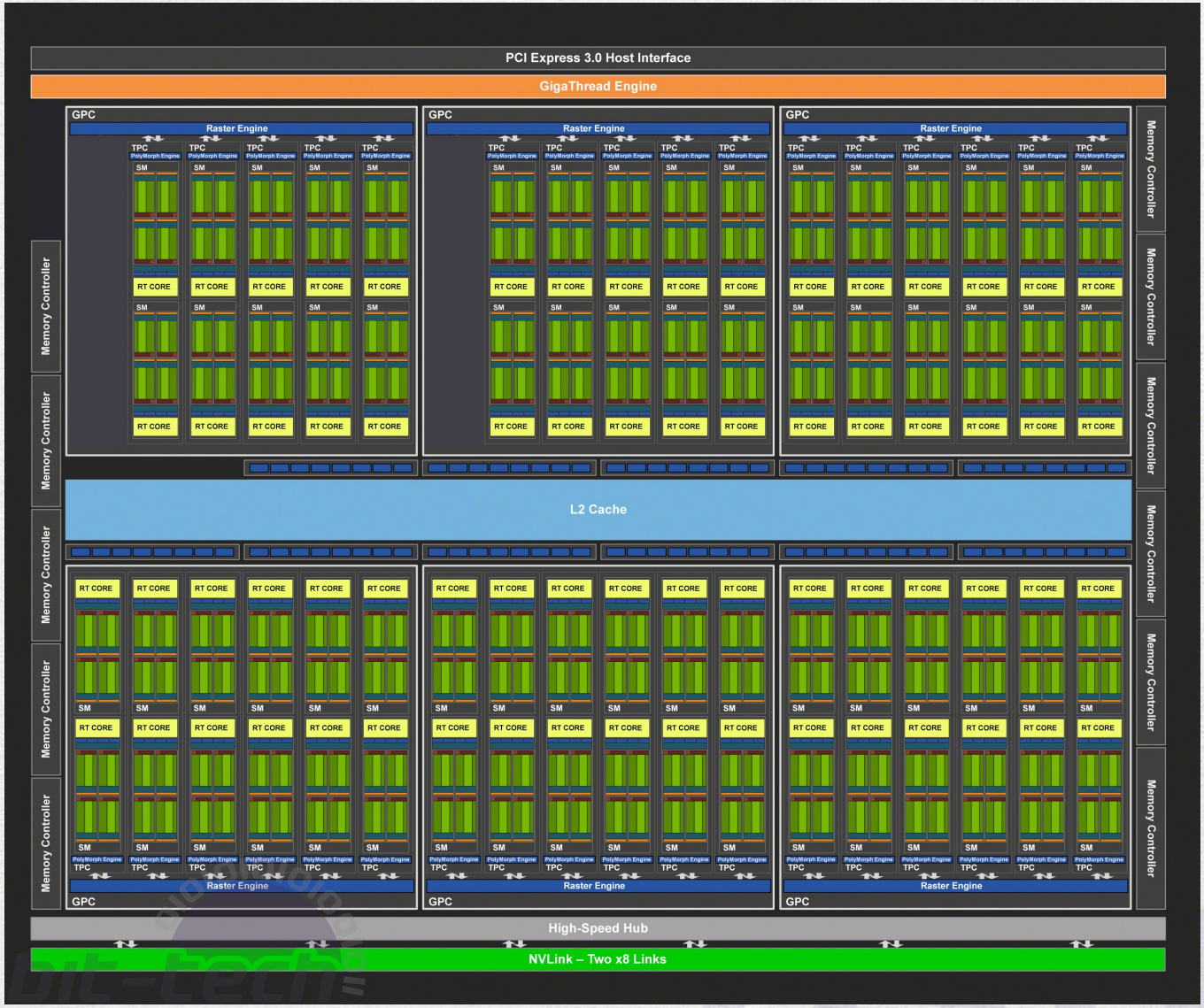
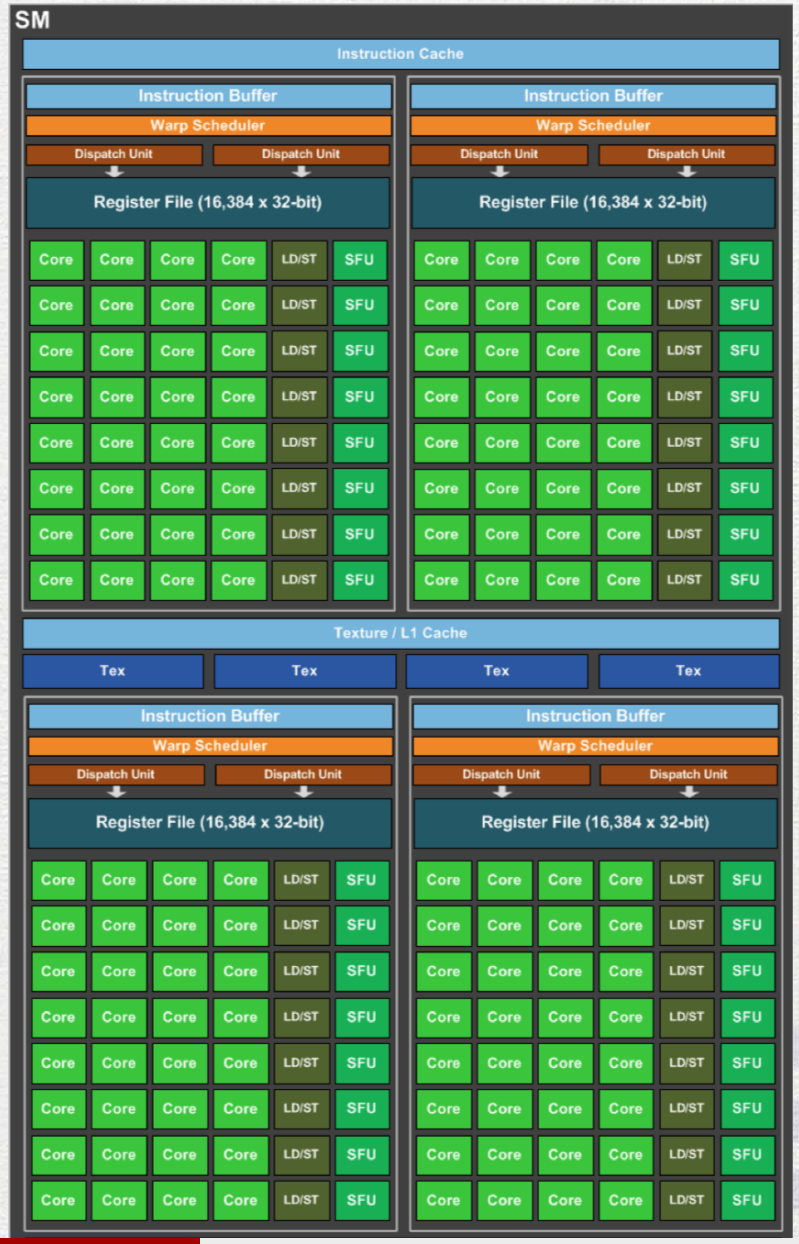


Figure 10: RTX 2080 Ti

GPUs

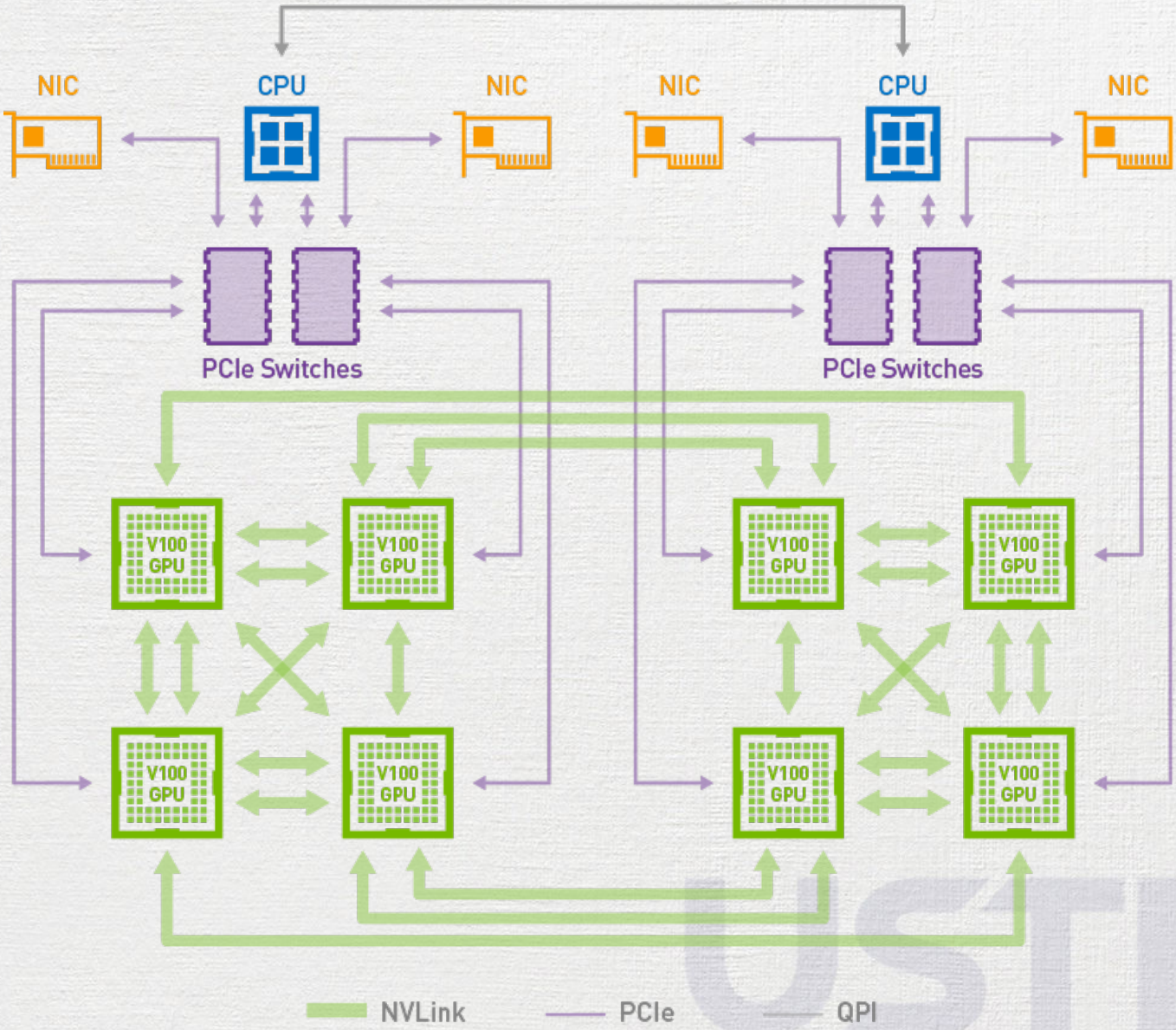

SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core
LD/ST
SFU
Texture / L1 Cache
Tex

GPUs

Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHZ	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS	5	6.8	10.6	15.7
Peak FP64 TFLOPS	1.7	.21	5.3	7.8
Peak Tensor TFLOPS	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12GB	Up to 24GB	16GB	16GB
TDP	235W	250W	300W	300W
Transistors	7.1B	8B	15.3B	21.1B
GPU Die Size	$551mm^2$	$601mm^2$	$610mm^2$	$815mm^2$

src: volta architecture white paper

GPUs


NIC
CPU
NIC
NIC
CPU
NIC
PCIe Switches
PCIe Switches
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
V100 GPU
NVLink
PCIe
QPI

CPU vs GPU

CPU	GPU
Low Latency	High throughput
Complex computations	Simple computation
Flexible	Less flexible
High clock speed	Lower clock speed
Few cores	Mooooooore cores
Like a boss	Like an army of interns

Labwork 2: Get to know your GPU

- Use `numba.cuda` to get to know your GPU
  - Device name
  - Core info: multiprocessor count, core count
  - Memory info: memory size
- Hint
  - [Optional] Use `numba.cuda.detect()` to print GPUs, if you have more than one ☺
  - Use `numba.cuda.select_device()` to select and get a GPU
  - Use its `id`, `name` atributes...

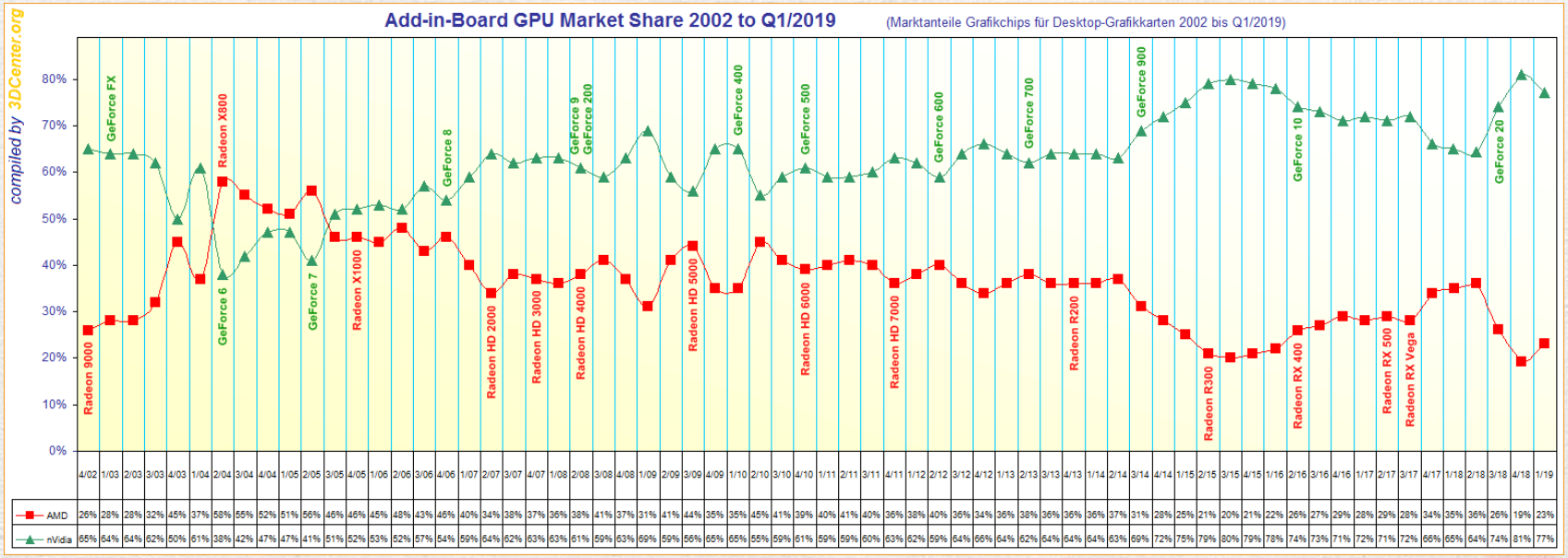
Labwork 2: Get to know your GPU

- Write a report (in LaTeX):
  - Name it « Report.2.Device.Info.tex »
  - Capture output of your labwork regarding GPU information
- Push your code and report to your forked repository

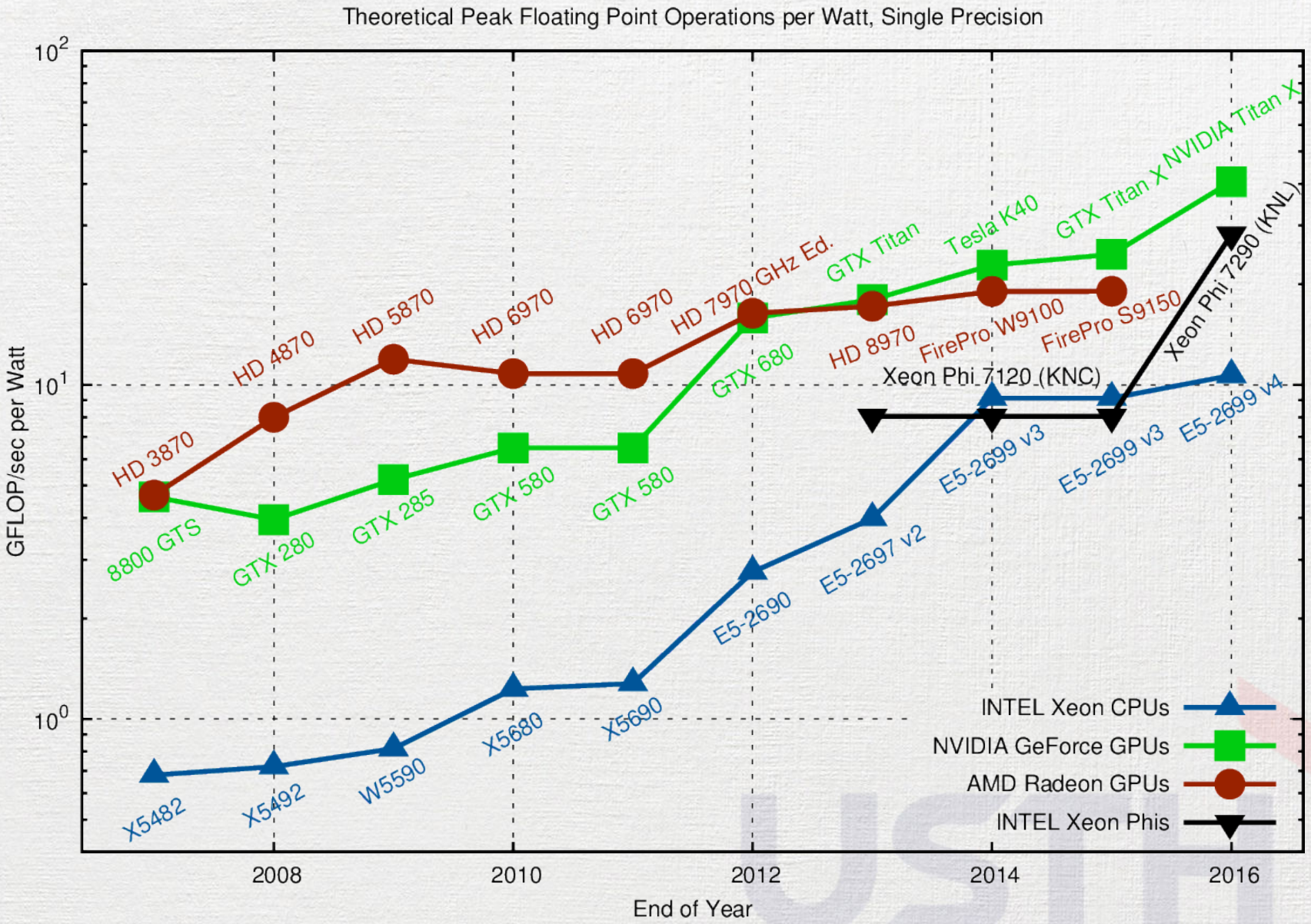
Why?

Availability

- NVIDIA GPUs are everywhere

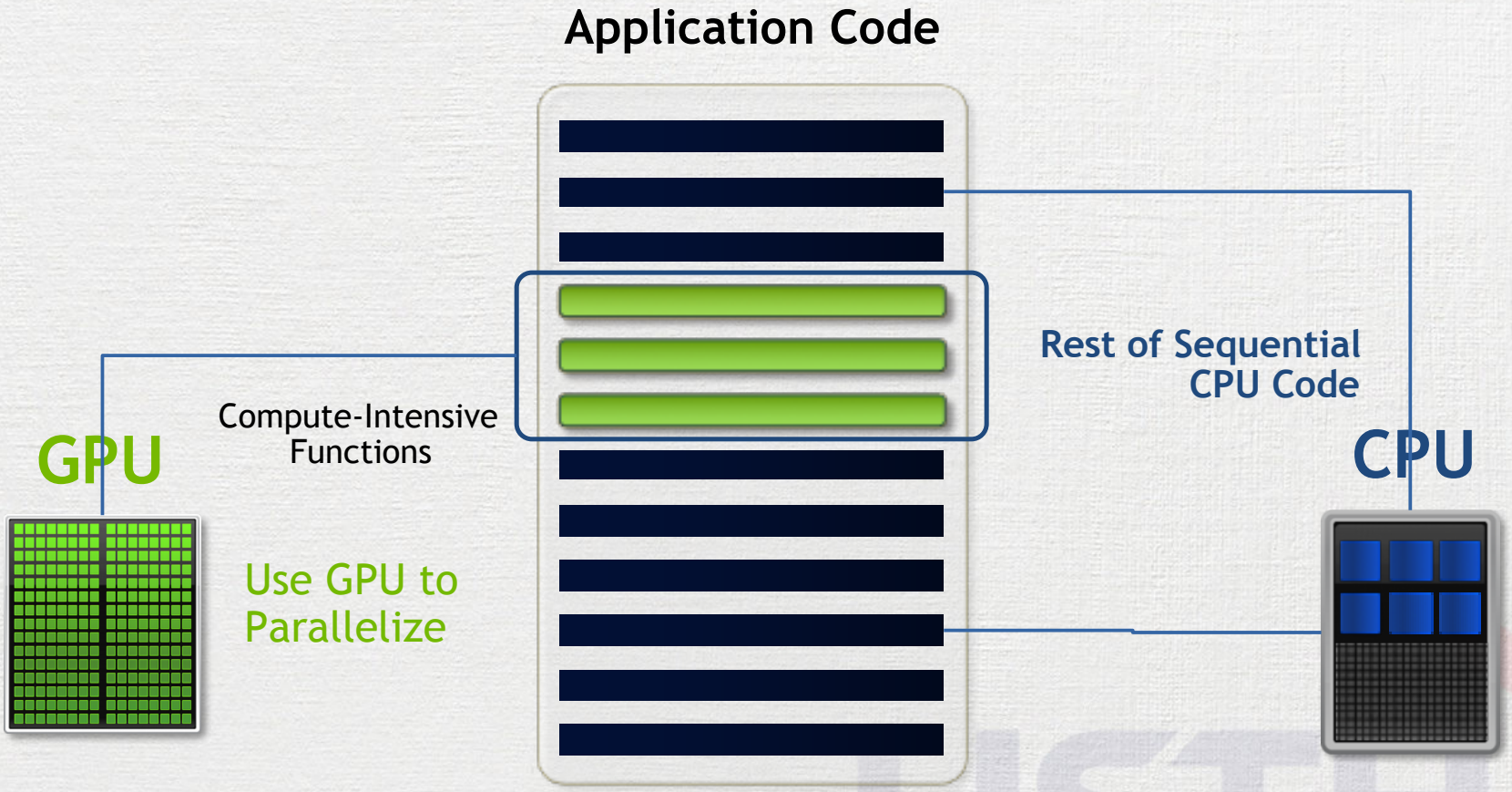


Power Efficiency



Change and speedup

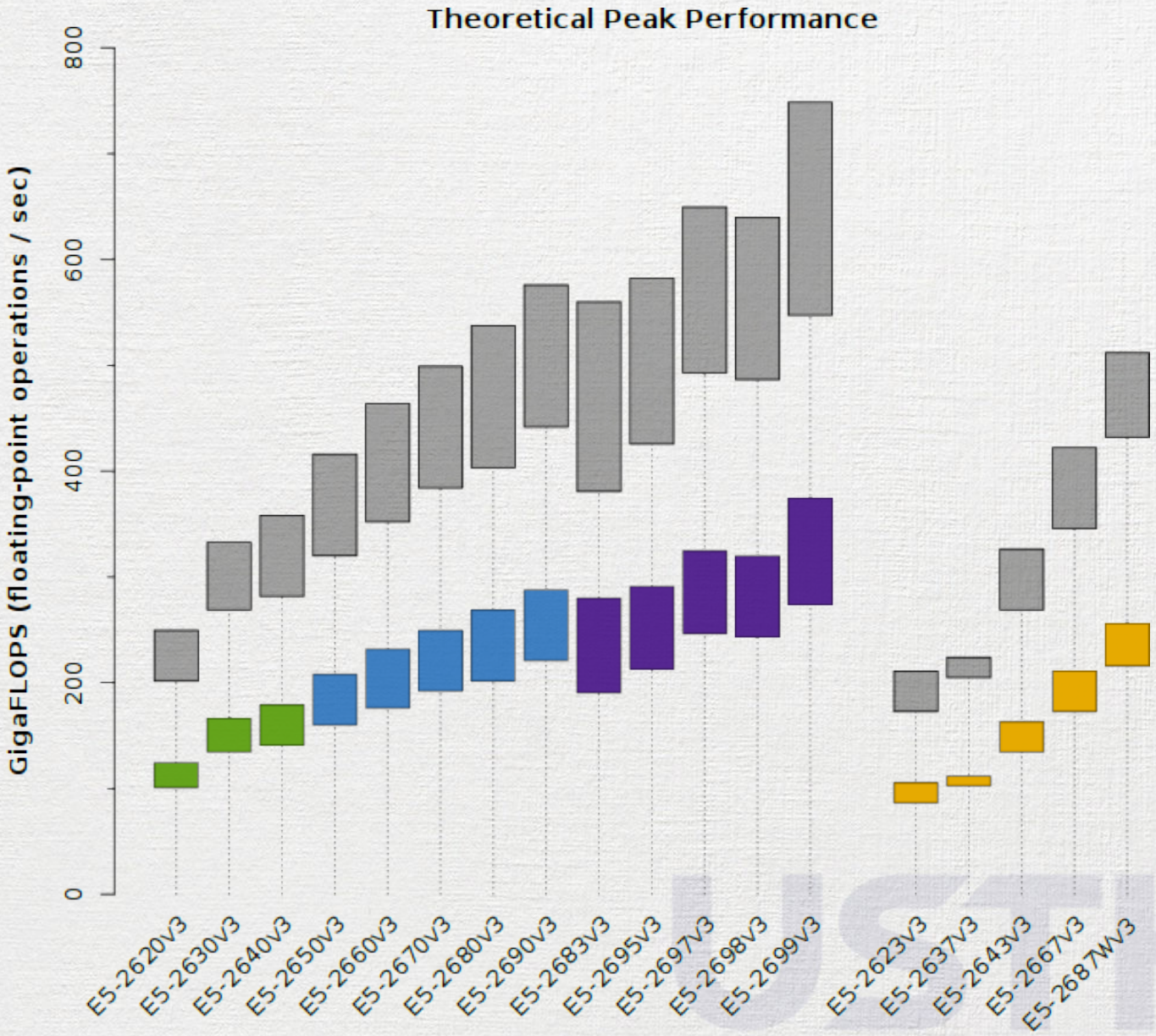
- Small change, big speedup



Performance

- Transparent scalability
  - No code change
  - Runs on better GPUs

Performance



Performance

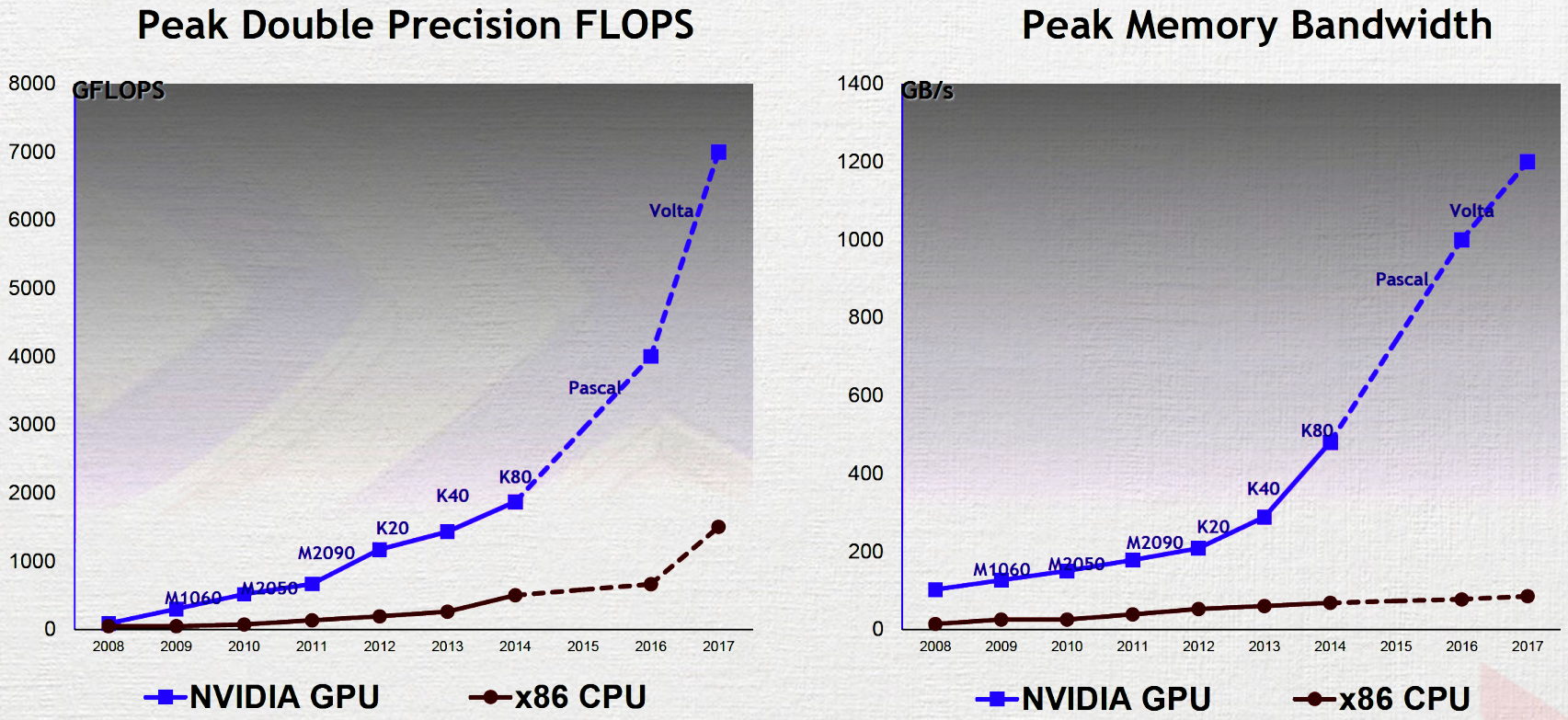


Figure 12: Predicted Pascal/Volta performance (2014)

Compute API

- C extension
- Easier to learn
- No need to map computation to Graphics API (texture, fragments...)

CUDA vs OpenCL

Feature	CUDA	OpenCL
Open	No	Yes
Hardware Support	NVIDIA	NVIDIA, AMD, Intel, Qualcomm...
CPU Support	No	CPU Device
Compilation	Offline	Offline + Online
Extension	Proprietary	`EXTENSION`
Portability	No	Yes
Performance	Similar[2]	Similar
Language Support	C/C++	C/C++/Fortran

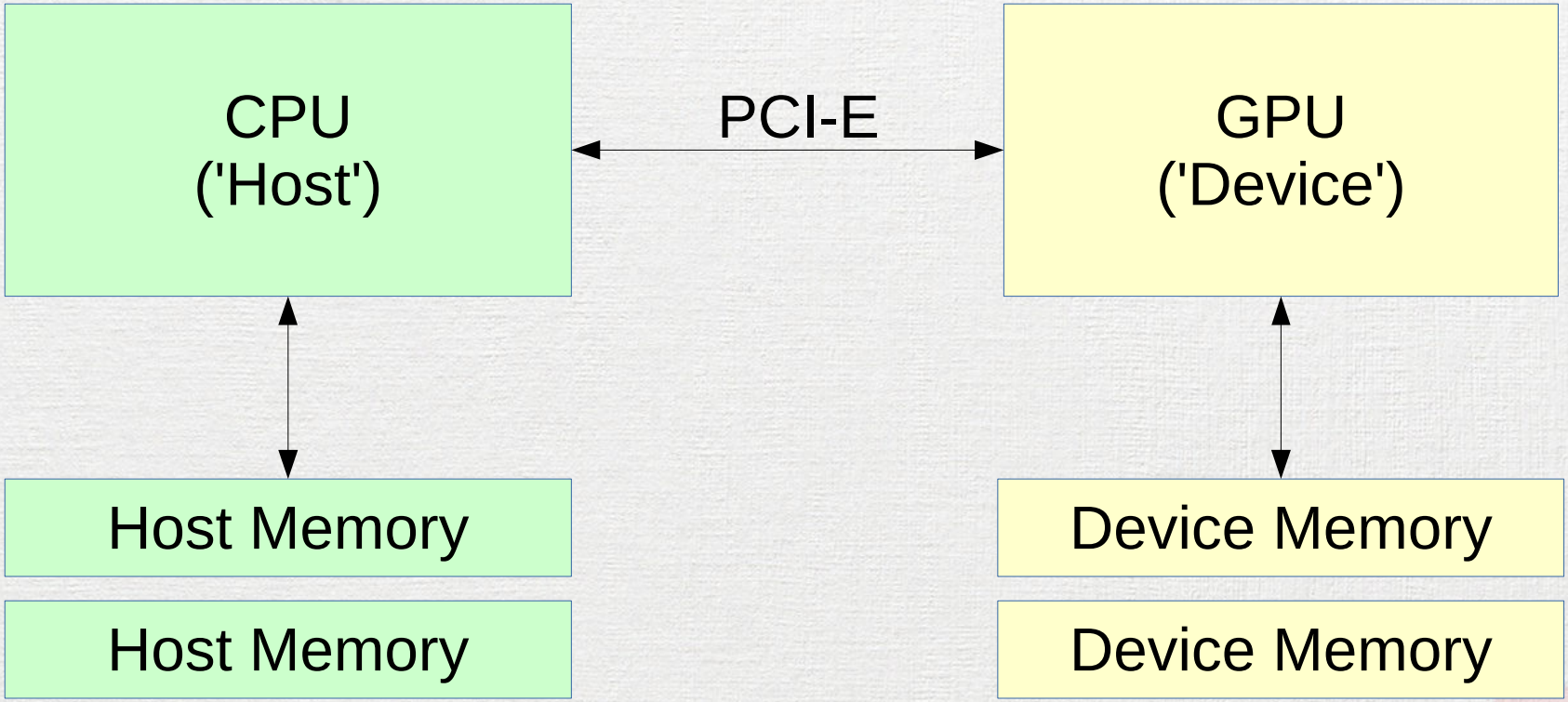
[2] Benchmark CUDA vs OpenCL on NVIDA Hardware

How

Components

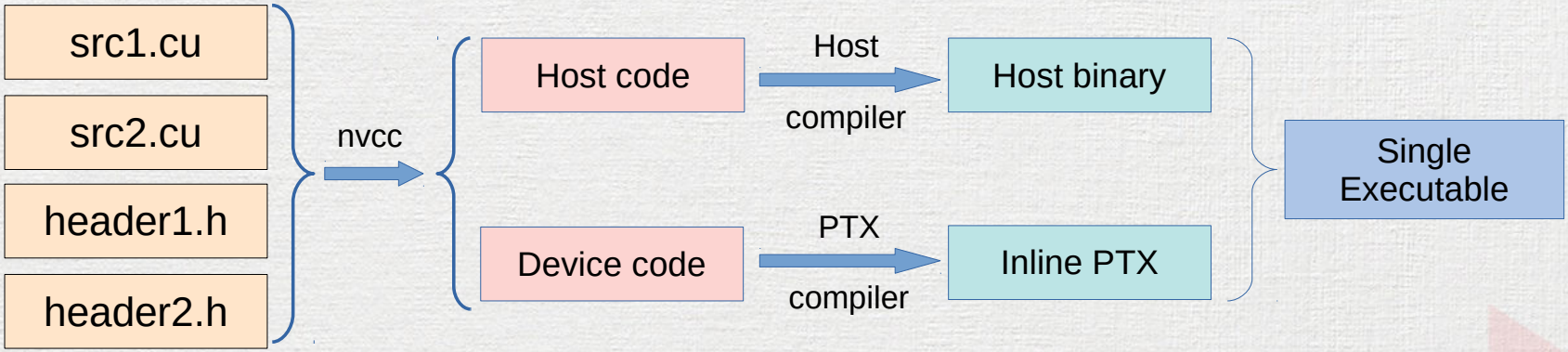
- CPU (host) and system memory
- GPU (device) and video memory

Components


CPU
('Host')
PCI-E
GPU
('Device')
Host Memory
Host Memory
Device Memory
Device Memory

Compilation in C/C++

- Source: `.cu` files, `.h` headers
- nvcc splits to host code, device code
- Compiled and combined



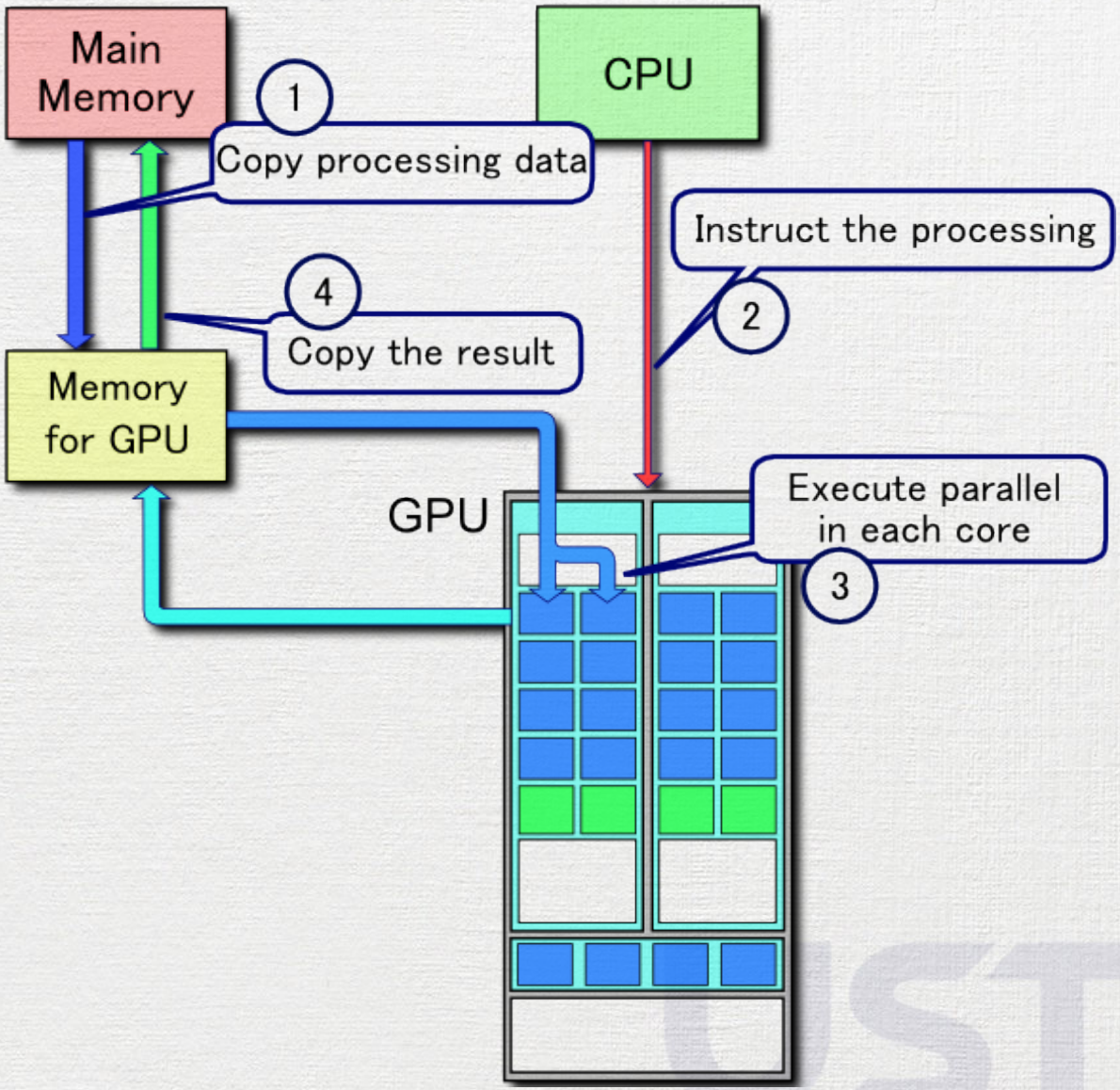
CUDA in Numba

- Source: `py` files
- JIT compilers to compile into PTX

Flow of CUDA program

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

Flow of CUDA program



Feeding data

- Remind : allocate memory on host?

```
data = numpy.zeros(shape, dtype)
```

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Remind : allocate memory on host?

```
data = numpy.zeros(shape, dtype)
```

```
void * malloc(size_t size);
```

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Remind : allocate memory on host?

```
data = numpy.zeros(shape, dtype)
```

```
void * malloc(size_t size);
```

```
new []
```

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Remind : allocate memory on host?

```
data = numpy.zeros(shape, dtype)
```

```
void * malloc(size_t size);
```

```
new []
```

Windows:

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Remind : allocate memory on host?

```
data = numpy.zeros(shape, dtype)
```

```
void * malloc(size_t size);
```

```
new []
```

Windows:

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Example

```
hostInput = numpy.zeros(
            (imageHeight, imageWidth, 3),
            numpy.uint8)
```

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns values

- Allocate memory on the device, mainly for output values

```
devData = cuda.device_array(shape, dtype)
```

`devData`: an object representing a buffer on the device.

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns values

- Allocate memory on the device, mainly for output values

```
devData = cuda.device_array(shape, dtype)
```

`devData`: an object representing a buffer on the device.

- Example

```
devOutput = cuda.device_array(
                                (imageHeight, imageWidth, 3),
                                numpy.uint8)
```

Feeding data

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Or, if data should be allocated AND transfered from host to device

```
devData = cuda.to_device(hostData)
```

Launching kernel

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- "Kernel": function runs on device, annotated with `@cuda.jit`
- Launch the kernel:

```
kernelName[numBlock, blockSize](args...)
```

Launching kernel

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Example

```
pixelCount = imageWidth * imageHeight
blockSize = 64
gridSize = pixelCount / blockSize
grayscale[gridSize, blockSize](devInput, devOutput)
```

- Number of blocks and block size: next session

Running on device

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Define a kernel
- Each kernel is executed in a separated thread
- Very fast thread scheduler
  - No context switch
  - Zero overhead scheduling
- Image
  - Each thread processes one pixel
  - A block of thread processes several “nearby” pixels

Running on device

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- Example

```
@cuda.jit
def grayscale(src, dst):
        # where are we in the input?
    tidx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    g = np.uint8((src[tidx, 0] + src[tidx, 1] + src[tidx, 2]) / 3)
    dst[tidx, 0] = dst[tidx, 1] = dst[tidx, 2] = g
```

- Note: kernel can only access memory allocated on device
  - `src`: device variable
  - `dst`: device variable

Getting back result

1. Host feeds device with data
2. Host asks device to process data
3. Device processes data in parallel
4. Device returns result

- `devData.copy_to_host()`
  - Allocate a new buffer
  - Copy from device to host
- Example

```
hostOutput = devOutput.copy_to_host()
```

Summary: Flow of CUDA program

1. Host feeds device with data: `cuda.device_array()`, `cuda.to_device()`
2. Host asks device to process data: `kernelName[]()`
3. Device processes data in parallel:
   - `@cuda.jit kernelName`
   - `cuda.threadIdx.x`, `cuda.blockIdx.x`, `cuda.blockDim.x`
4. Device returns result: `copy_to_host()`

Extra CUDA vs OpenCL: CUDA image grayscaling

```
devSrc = cuda.to_device(flatSrc)
devDst = cuda.device_array((pixelCount, 3), np.uint8)
grayscale[gridSize, blockSize](devSrc, devDst)
hostDst = devDst.copy_to_host()
```

Extra CUDA vs OpenCL: CUDA image grayscaling

```
cudaMalloc(&devInput, pixelCount * sizeof(uchar3));
cudaMalloc(&devGray, pixelCount * sizeof(float));
cudaMemcpy(devInput, hostInput,
        pixelCount * sizeof(uchar3),
        cudaMemcpyHostToDevice);
rgb2grayCUDA<<<dimGrid, dimBlock>>>(
        devInput, devGray, regionSize);
cudaMemcpy(hostGray, devGray,
        pixelCount * sizeof(float),
        cudaMemcpyDeviceToHost);
cudaFree(devInput);
cudaFree(devGray);
```

Extra CUDA vs OpenCL: OpenCL image grayscaling

```
cl_context context = clCreateContext(0, 1,
        deviceIds, NULL, NULL, &ret);
cl_command_queue queue = clCreateCommandQueue(context,
        deviceIds[0], 0, &ret);
cl_mem devInput = clCreateBuffer(context,
        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
        pixelCount * sizeof(uchar3), hostInput, &ret);
cl_mem devGray = clCreateBuffer(context, CL_MEM_READ_WRITE,
        pixelCount * sizeof(cl_float), NULL, &ret);
cl_program program = clCreateProgramWithSource(context, 1,
        &blurDetectProgram, &programSize , &ret);
ret = clBuildProgram(program, 1, deviceIds, NULL, NULL, NULL);
cl_kernel rgb2gray = clCreateKernel(program, "rgb2grayOCL", &ret);
clSetKernelArg(rgb2gray, 0, sizeof(cl_mem), (void *)&devInput);
clSetKernelArg(rgb2gray, 1, sizeof(cl_mem), (void *)&devGray);
clSetKernelArg(rgb2gray, 2, sizeof(pixelCount),
        (void *)&pixelCount);
ret = clEnqueueNDRangeKernel(queue, rgb2gray, 1, NULL,
        workSize, 0, 0, 0, 0);
ret = clEnqueueReadBuffer(queue, devGray, CL_TRUE, 0,
        pixelCount * sizeof(float), hostGray, 0, NULL, NULL);
```

Labwork 3: Hello, CUDA!

- Make image RGB-to-gray converter using Numba CUDA
  - Load an image from file (`matplotlib`'s `imread`)
  - Flatten image into 1D array of RGB (`reshape(pixelCount, 3)`)
  - Implement grayscale using CPU (`for` range)
  - Implement grayscale using GPU
  - Save/show the image after each grayscale to validate the result visually
- Use `time.time()` to measure speedup

Labwork 3: Hello, CUDA!

- Write a report (in LaTeX)
  - Name it « Report.3.cuda.tex »
  - Explain how you implement the labwork
  - Try experimenting with different block size values
  - Plot a graph of block size vs time
  - Discuss the graph
- Push the report and your code to your forked repository

Extra: Grayscaling

- Input: color RGB image I of size $(h, w, 3)$
- Output: gray image g of size (h, w)
- Simplest conversion:

$$g(y, x) = \frac{1}{3} \sum_{i=0}^{3} I(y, x, i)$$