Numerical computing in Julia

GPU programming

Tim Besard (UGent, Belgium)
Valentin Churavy (OIST, Japan)

What are GPUs?

What GPUs were meant to do

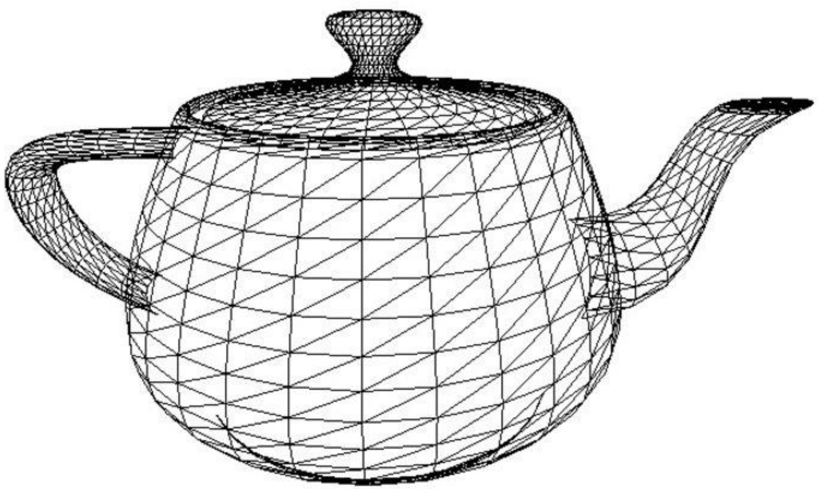
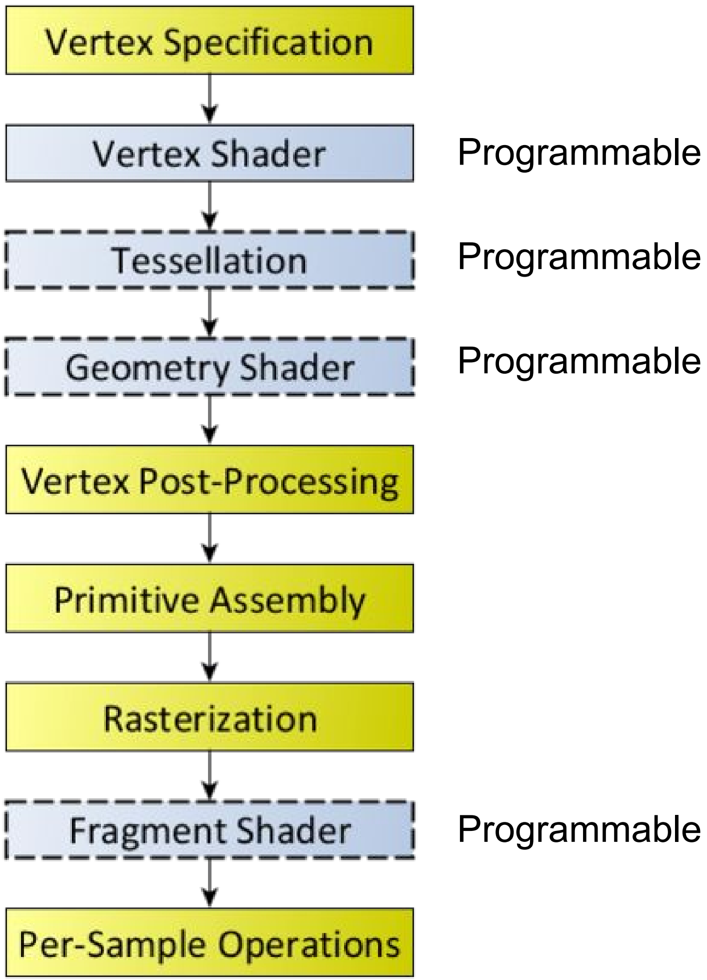


Image credit: Henrik Wann Jensen

Input: description of a scene:
Surface geometry + surface materials, lights, camera, etc

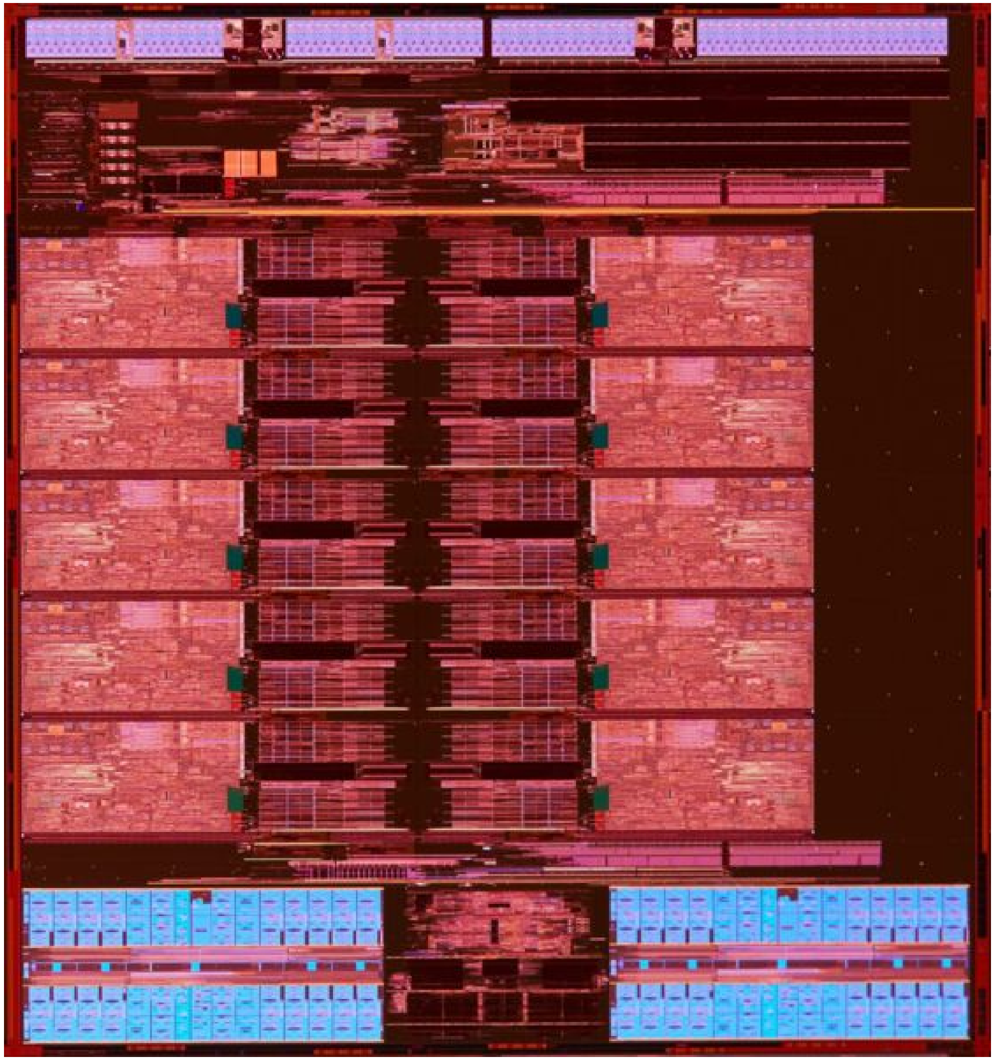
GPU → specialised co-processor for graphics
Why?

Graphics shader pipeline



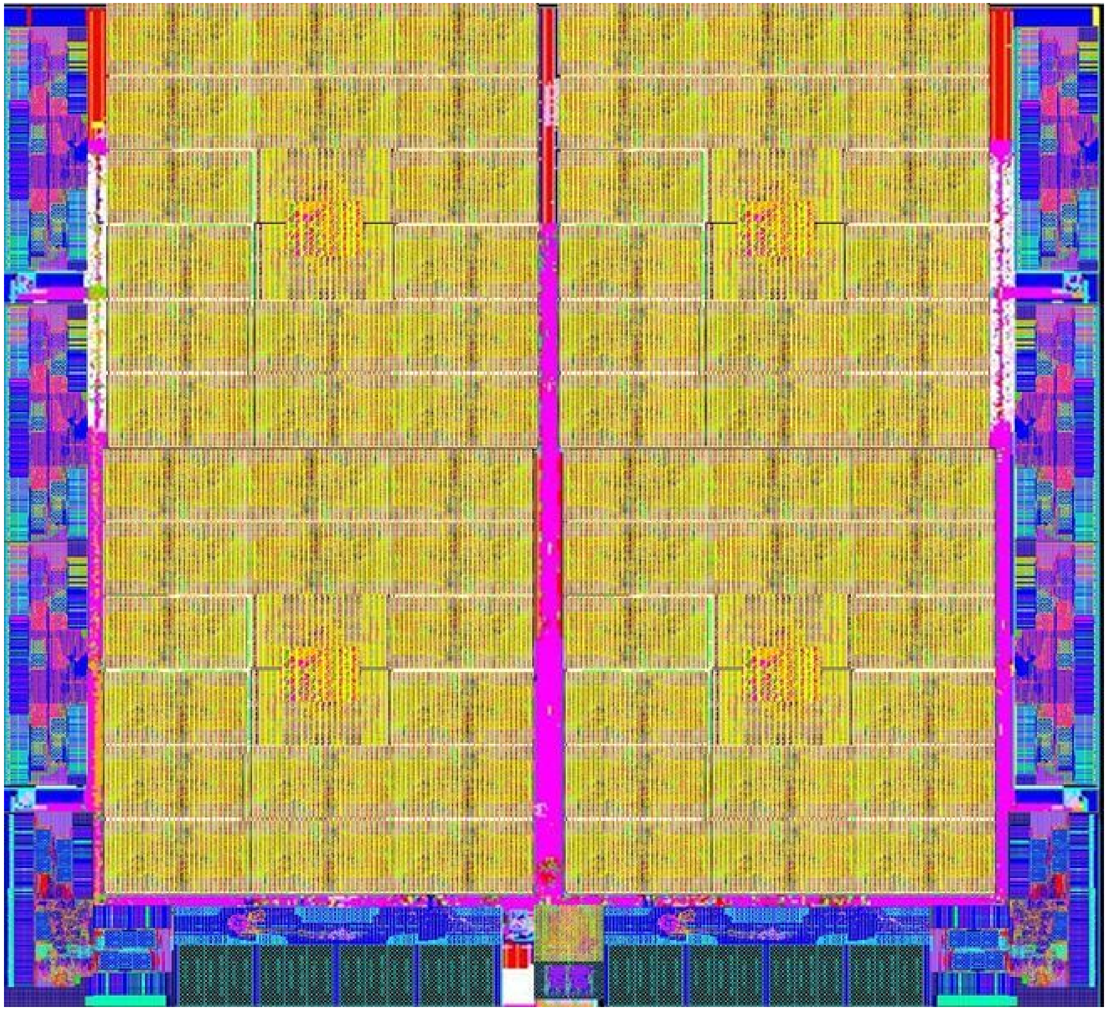
opengl.org

Heavyweight core architecture



https://en.wikichip.org/wiki/intel/microarchitectures/broadwell

Lightweight cores architecture



https://en.wikichip.org/wiki/pezy/pezy-sc

Why use GPUs?

A small supercomputer in your desktop/laptop/phone

- HPC:
    - NVIDIA Tesla P100 | 21 TFLOPS FP16 | 10 TFLOPS FP32 | 5 TFLOPS FP64
    - NVIDIA Tesla K80 | 8.93 TFLOPS FP32 | 2.91 TFLOPS FP64
    - AMD FirePro S9100 | 4.22 TFLOPS FP32 | 2.11 TFLOPS FP64
- Desktop
    - NVIDIA Geforce 1080 | 8.9 TFLOPS FP32
    - NVIDIA Titan X | 11 TFLOPS FP32
    - AMD Radeon R9 Fury X | 8.6 TFLOPS FP32
- Phone
    - NVIDIA Tegra X1 | 512 GFLOPS FP32
    - PowerVR GT7XT | 115 GFLOPS FP32

In comparison Xeon E5-2600v3 | 500 GFLOPS FP64

Energy efficiency: https://www.top500.org/green500/lists/2016/06/

Why use GPUs

Problems that map well onto the GPU:

- Dense Linear Algebra
- Computer Vision
- Physics simulations
    - Fluids
    - Weather
- Deep Learning (and other Machine Learning paradigms)

Problems that do not map as well (but you will still see performance gains):

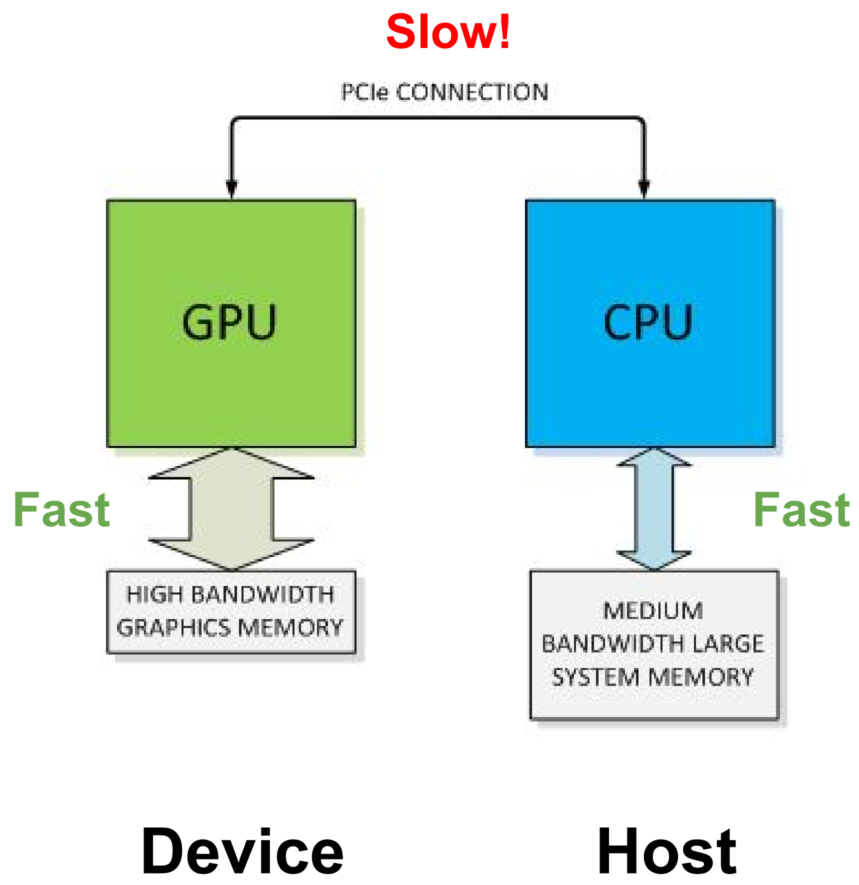
- Graph processing
- Sparse Linear Algebra

A short history of GPGPU

- 2002/2003: Using shaders for scientific computation on a GPU.
  - Render 2 triangles that exactly cover the screen
  - One shader operation per pixel = one shader computation per output element
  - GPUs can be used for data-parallel programming on arbitrary data.
  - Examples:
    - [Harris2002] (http://dl.acm.org/citation.cfm?id=569061)
    - [Bolz2003] (http://dl.acm.org/citation.cfm?id=882364)
- 2004: Brook stream programming language out of Stanford [Buck2004]
  - Brook compiler converted generic stream program into OpenGL commands and shader programs
  - C-like language
- 2007 NVIDIA Tesla architecture offered the first non-graphics-specific interface.
  - Introduction of CUDA (C-like language) with the design goal of maintaining a low abstraction distance.
- 2009 OpenCL was introduced as an open standard.

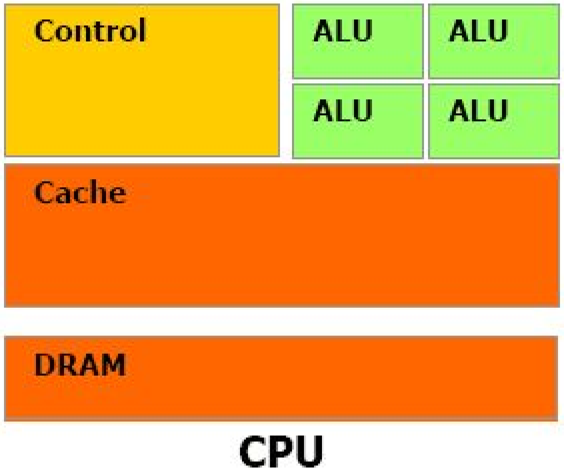
How do GPUs work?

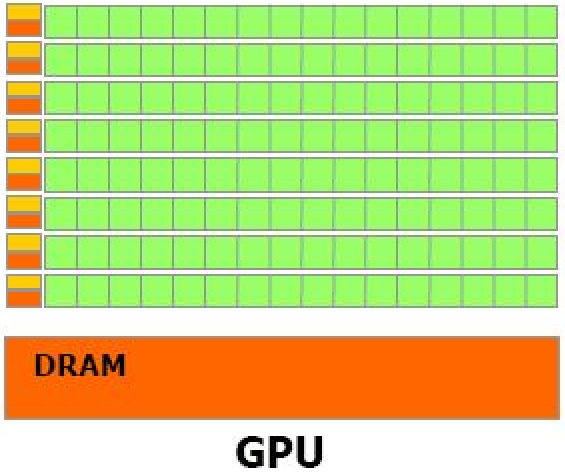
How do GPUs work?



Source: NVIDIA

How do GPUs work?





Heavyweight cores
Limited parallelism

Advanced control circuitry
- Out-of-order execution
- Branch prediction

Lightweight cores → higher latency
Massive parallelism

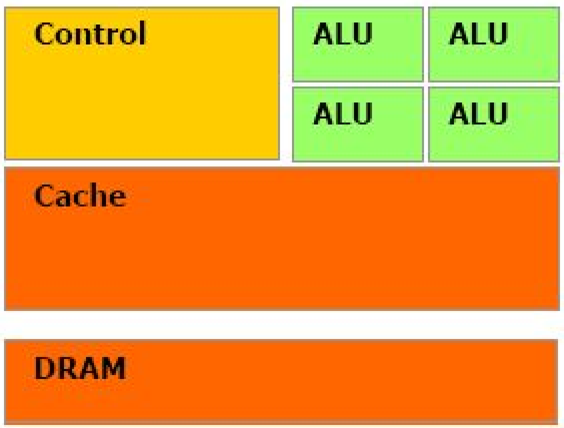
Source: AllegroViva

How do GPUs work: latency hiding

Requirement: enough work to schedule (*occupancy*)

Source: TU/e

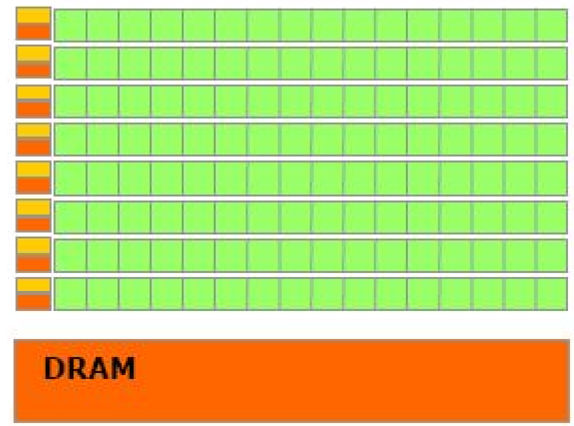
How do GPUs work?



CPU

Heavyweight cores
Limited parallelism

Advanced control circuitry
- Out-of-order execution
- Branch prediction



GPU

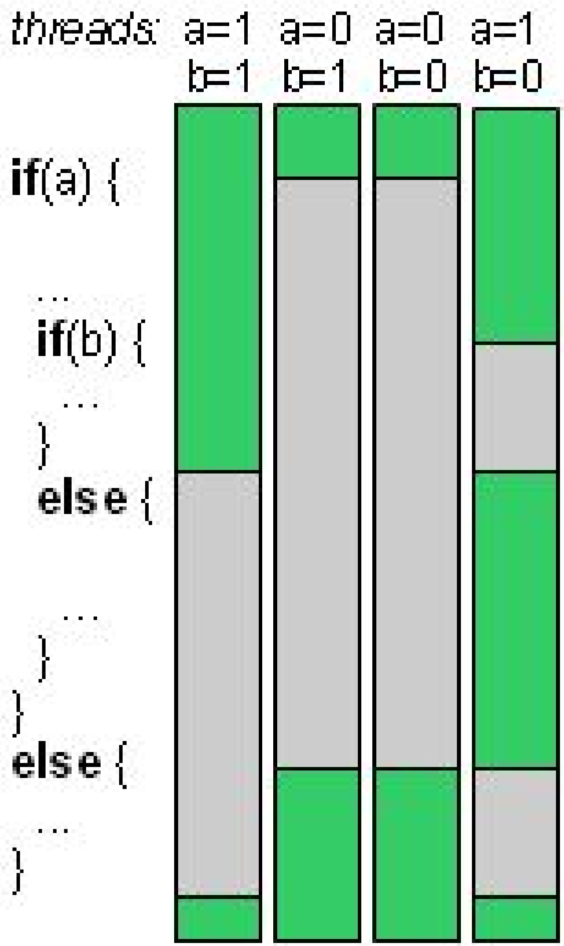
Lightweight cores

Massive parallelism

Simple, shared control circuitry
- Lockstep execution

Source: AllegroViva

How do GPUs work: lockstep execution



Source: Y. Kreinin

How do GPUs work?

✓ Massive parallelism

✗ Lightweight cores

- High latency
- Latency hiding, given sufficient occupancy

✗ Lockstep execution

- Avoid branch divergence

✗ Memory transfer cost

How to program GPUs?

How to program GPUs: libraries

Mostly array-based abstractions:

- cuBLAS → CUBLAS.jl
- Thrust
- ArrayFire → ArrayFire.jl
- GPUArrays.jl
- …

```
using ArrayFire

a = rand(AFArray{Float64}, 100)
b = …

product = a * b
c = a .> b
fast_fourier = fft(a)
```

How to program GPUs: low-level toolkits

Shallow wrappers around the hardware:

- OpenCL
- CUDA → CUDAnative.jl

How to program GPUs: vector addition

```
function cpu_vadd(a, b, c)
  for i = 1:length(a)
    c[i] = a[i] + b[i]
  end
end

len = 16
a = rand(len)
b = rand(len)
c = similar(a)




cpu_vadd(a, b, c)
```

```
function gpu_vadd(a, b, c)
  i = get_my_index()
  c[i] = a[i] + b[i]
end
```

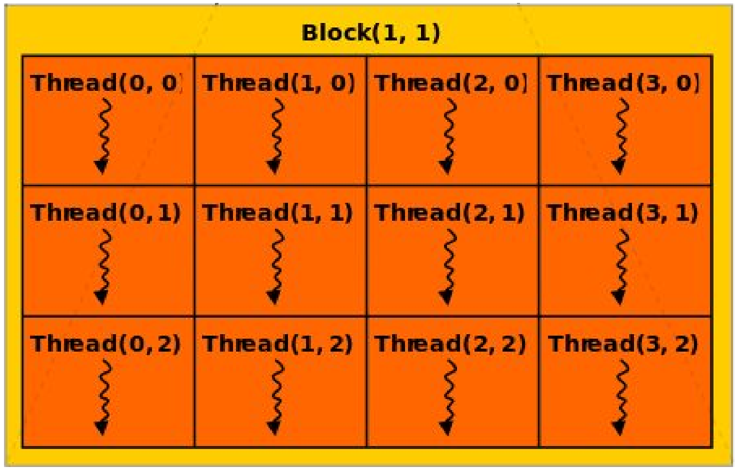
Executed by each thread

```
len = 16
a = rand(len)
b = rand(len)

gpu_a = CuArray(a)
gpu_b = CuArray(b)
gpu_c = CuArray(Float64, len)

@cuda (...) gpu_vadd(gpu_a, gpu_b, gpu_c)

c = Array(d_c)
```

How to program GPUs: CUDA indexing

`get_my_index()` ?

- `threadIdx()` & `blockDim()`
- `@cuda (1, threads) kernel(...)`

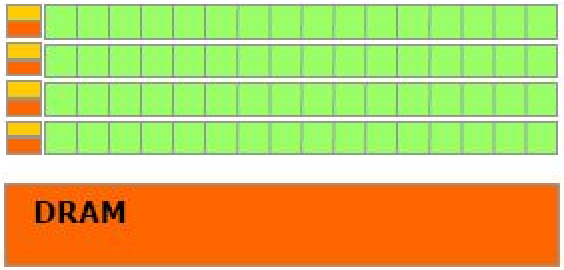



```
function gpu_vadd(a, b, c)
  i, j = threadIdx().x, threadIdx().y
  c[i,j] = a[i,j] + b[i,j]
end

threads = (size(a,1), size(a,2))
@cuda (1,threads) gpu_vadd(...)
```

Source: NVIDIA

How to program GPUs: CUDA indexing

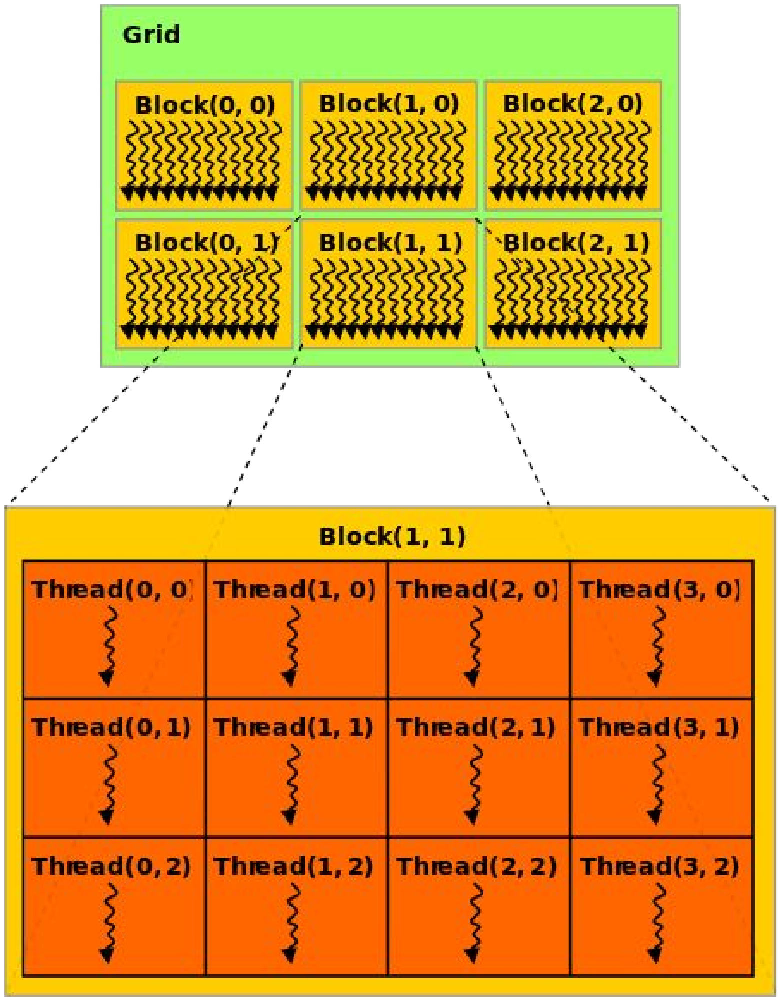
- `threadIdx(), blockDim()`
- `blockIdx(), gridDim()`
- `@cuda (blocks, threads) kernel(...)`



Threads are executed together!

✓ Cheaper communication

✗ Lockstep execution



Source: NVIDIA, AllegroViva

How to program GPUs: prefix sum

```
function cpu_scan{T}(data)
    cols = size(data,2)
    for col in 1:cols
        accum = zero(T)

        rows = size(data,1)
        for row in 1:rows



            accum += data[row,col]



            data[row,col] = accum
        end
    end
end
```

```
4×4 Array{Int64,2}:
  1   5   9  13
  2   6  10  14
  3   7  11  15
  4   8  12  16
```

↓

```
4×4 Array{Int64,2}:
  1   5   9  13
  3  11  19  27
  6  18  30  42
 10  26  42  58
```

How to program GPUs: prefix sum

```
function cpu_scan{T}(data)
    cols = size(data,2)
    for col in 1:cols
        accum = zero(T)

        rows = size(data,1)
        for row in 1:rows



            accum += data[row,col]



            data[row,col] = accum
        end
    end
end
```

```
@target ptx function gpu_scan{T}(data)
    cols = gridDim().x
    col = blockIdx().x


    rows = blockDim().x
    row = threadIdx().x

    accum = zero(T)
    for i in 1:row
        accum += data[i,col]
    end



    data[row,col] = accum
end

@cuda (size(data,2), size(data,1))
gpu_scan(...)
```

How to program GPUs: prefix sum

4×4 Array{Int64,2}:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

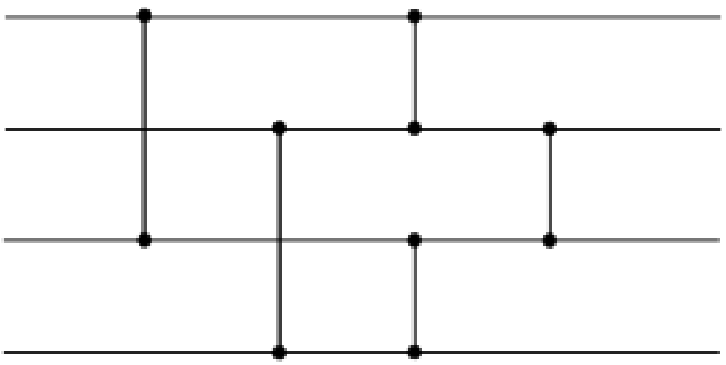
4×4 Array{Int64,2}:

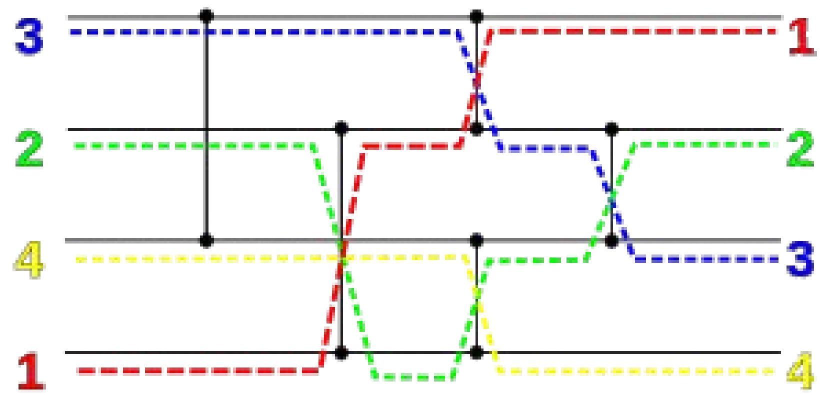
1	5	9	13
2	6	10	14
3	7	30	15
4	8	12	16

```
@target ptx function gpu_scan{T}(data)
    cols = gridDim().x
    col = blockIdx().x


    rows = blockDim().x
    row = threadIdx().x

    accum = zero(T)
    for i in 1:row
        accum += data[i,col]
    end
    sync_threads()

    data[row,col] = accum
end

@cuda (size(data,2), size(data,1))
gpu_scan(...)
```

Branch divergence

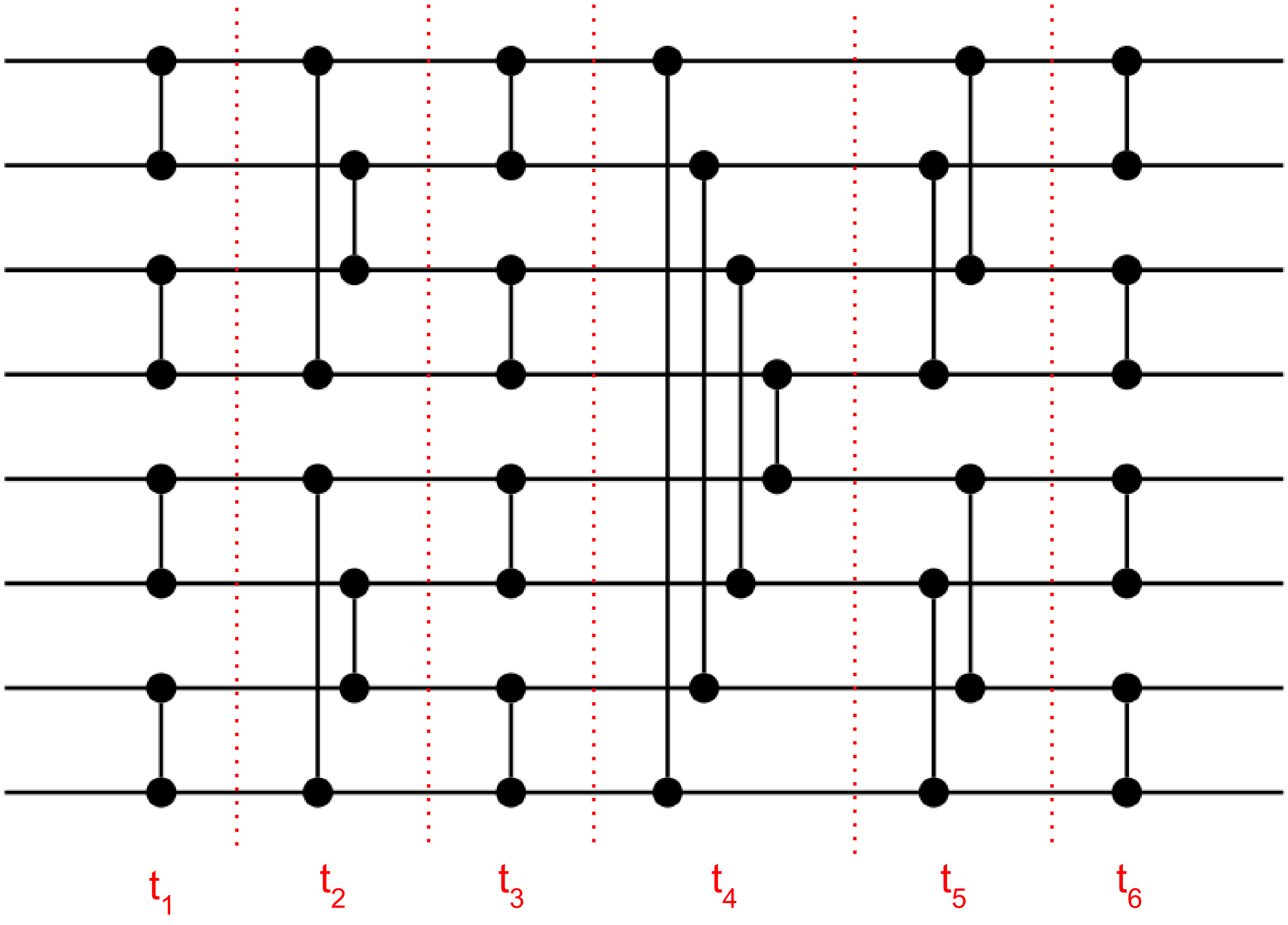
How to program GPUs: sorting





Source: Wikipedia

How to program GPUs: sorting



Source: Wikipedia