/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2016 - Lecture 10: “GPGPU (1)”

Welcome!

Global Agenda:

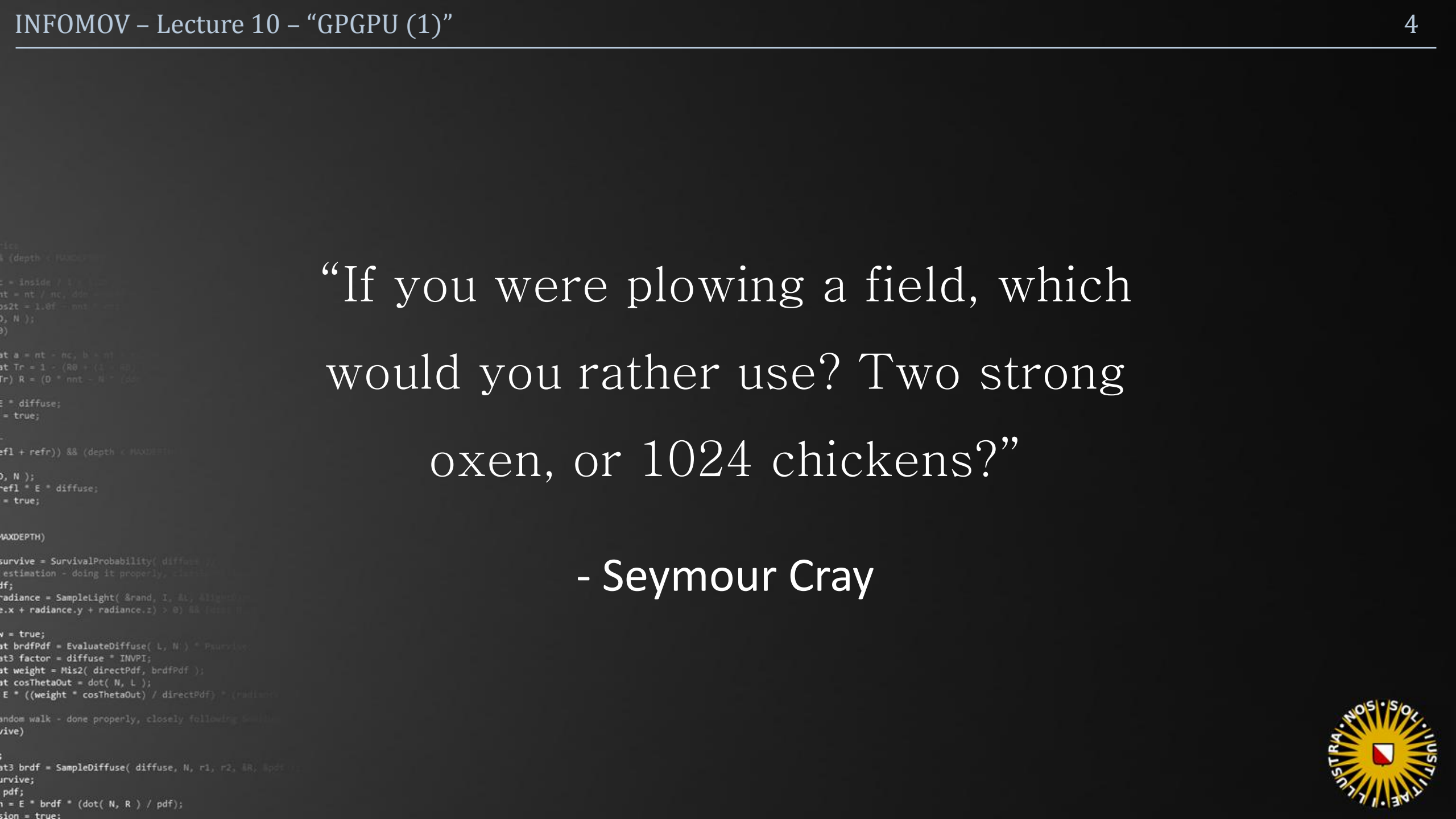
1. GPGPU (1) : Introduction, architecture, concepts
2. GPGPU (2) : Practical Code using GPGPU
3. GPGPU (3) : Parallel Algorithms, Optimizing for GPU

Today’s Agenda:

- Introduction to GPGPU
- Example: Voronoi Noise
- GPGPU Programming Model
- OpenCL Template

“If you were plowing a field, which would you rather use? Two strong oxen, or 1024 chickens?”

- Seymour Cray



Introduction

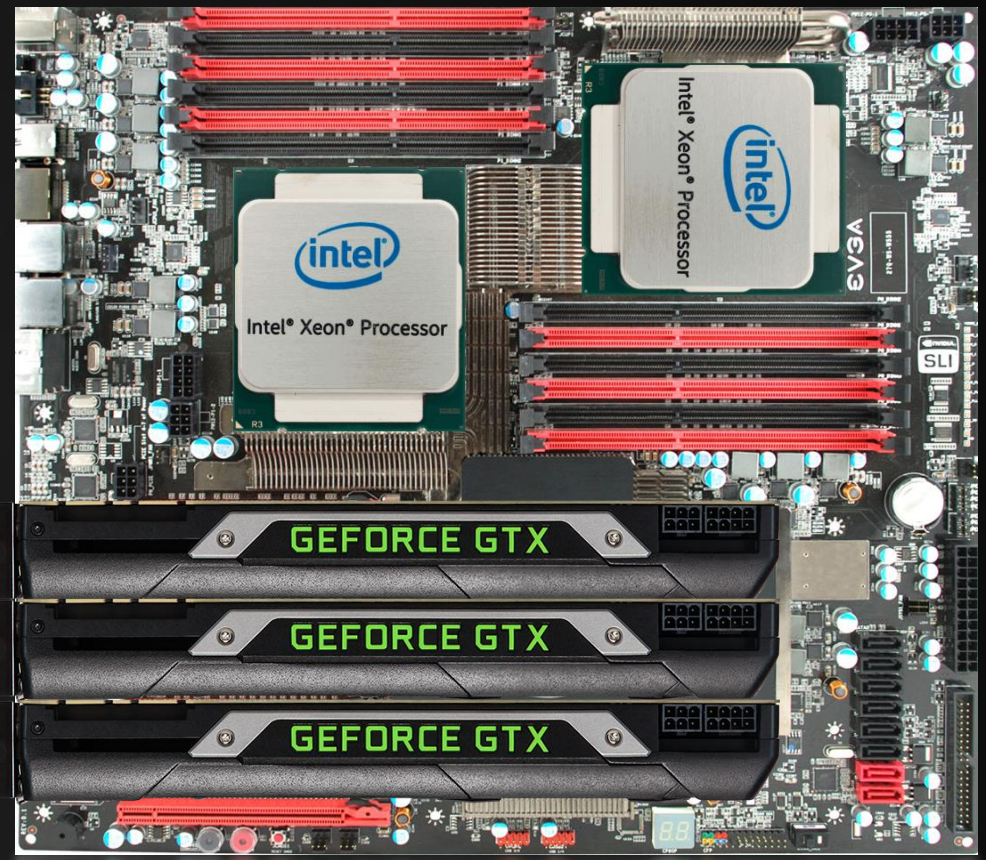
Heterogeneous Processing

The average computer contains:

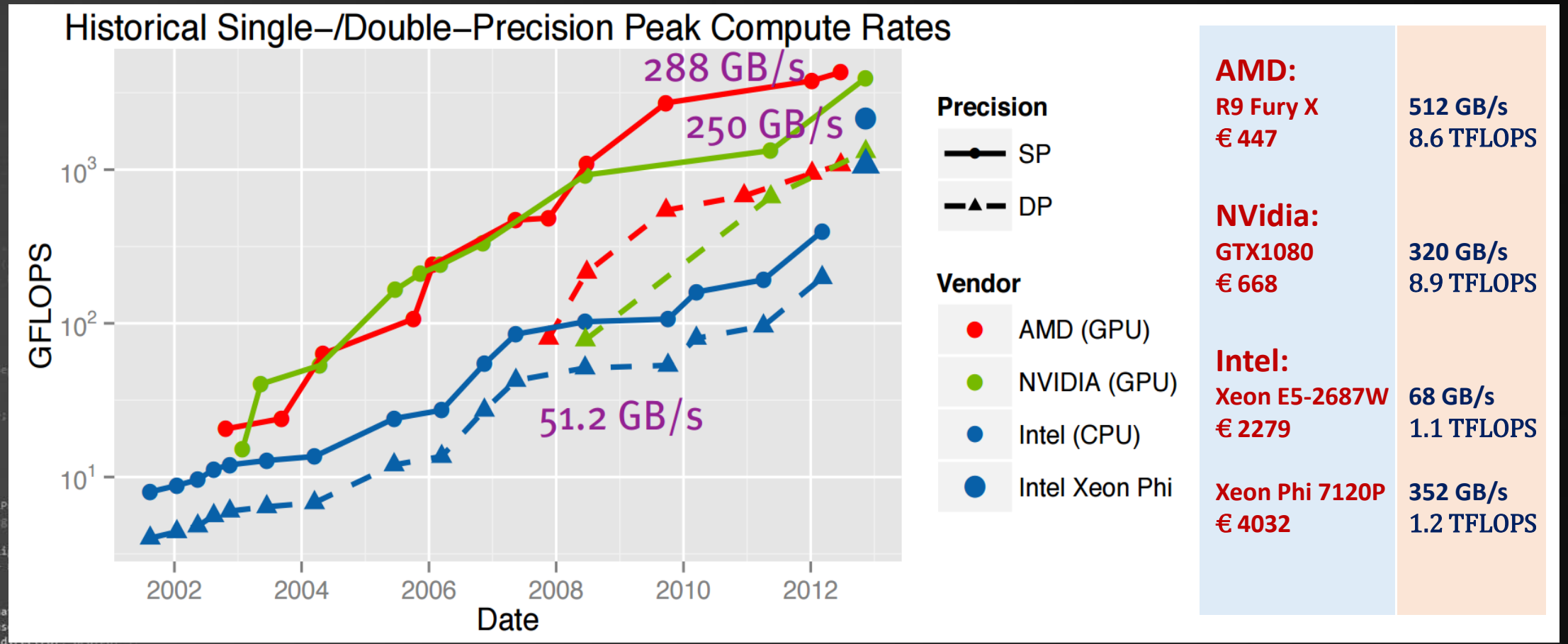
- 1 or more CPUs;
- 1 or more GPUs.

We have been optimizing CPU code.
A vast source of compute power has remained unused:

The Graphics Processing Unit.



Introduction



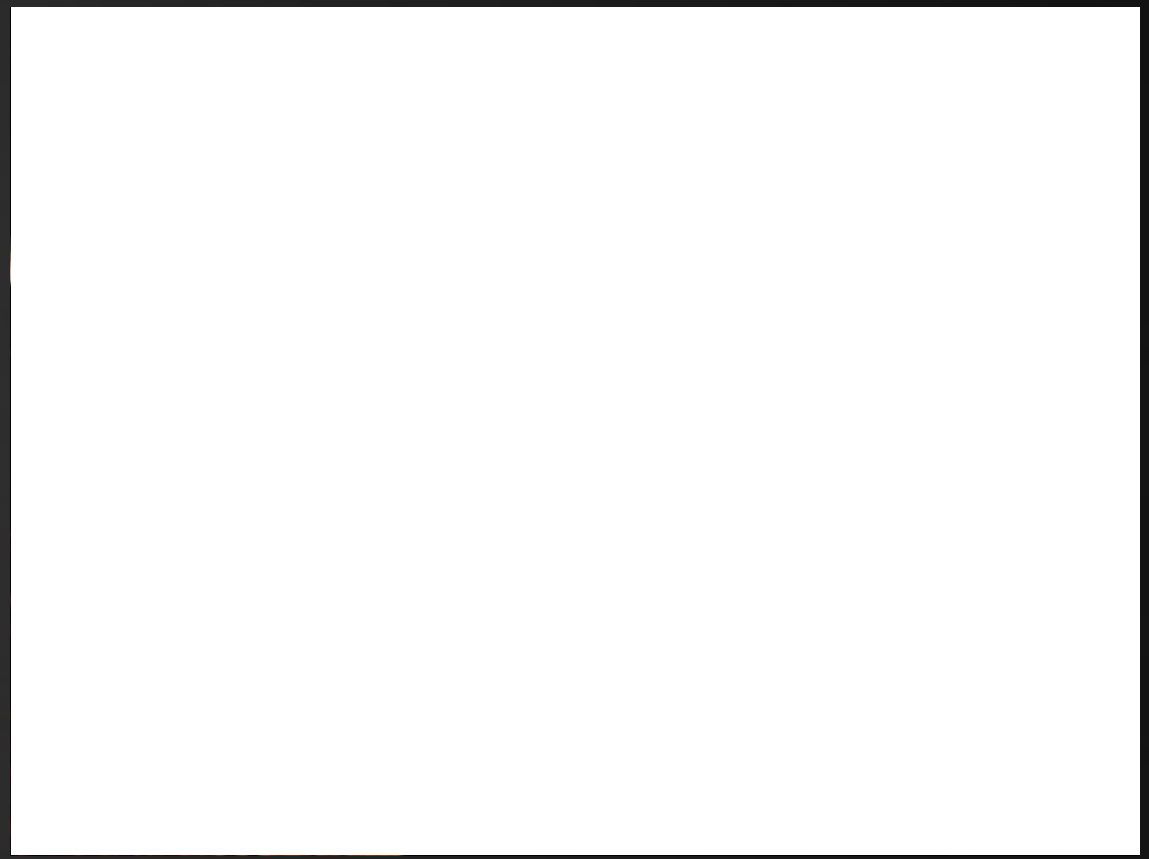
AMD:	
R9 Fury X € 447	512 GB/s 8.6 TFLOPS
NVidia:	
GTX1080 € 668	320 GB/s 8.9 TFLOPS
Intel:	
Xeon E5-2687W € 2279	68 GB/s 1.1 TFLOPS
Xeon Phi 7120P € 4032	352 GB/s 1.2 TFLOPS

Introduction

A Brief History of GPGPU

Introduction

A Brief History of GPGPU



Introduction

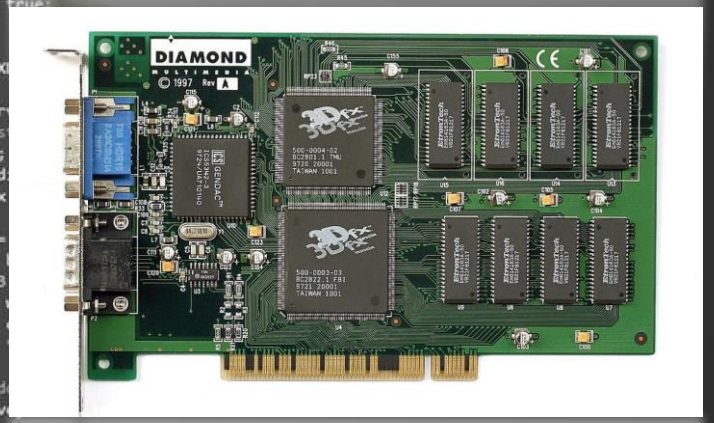
A Brief History of GPGPU

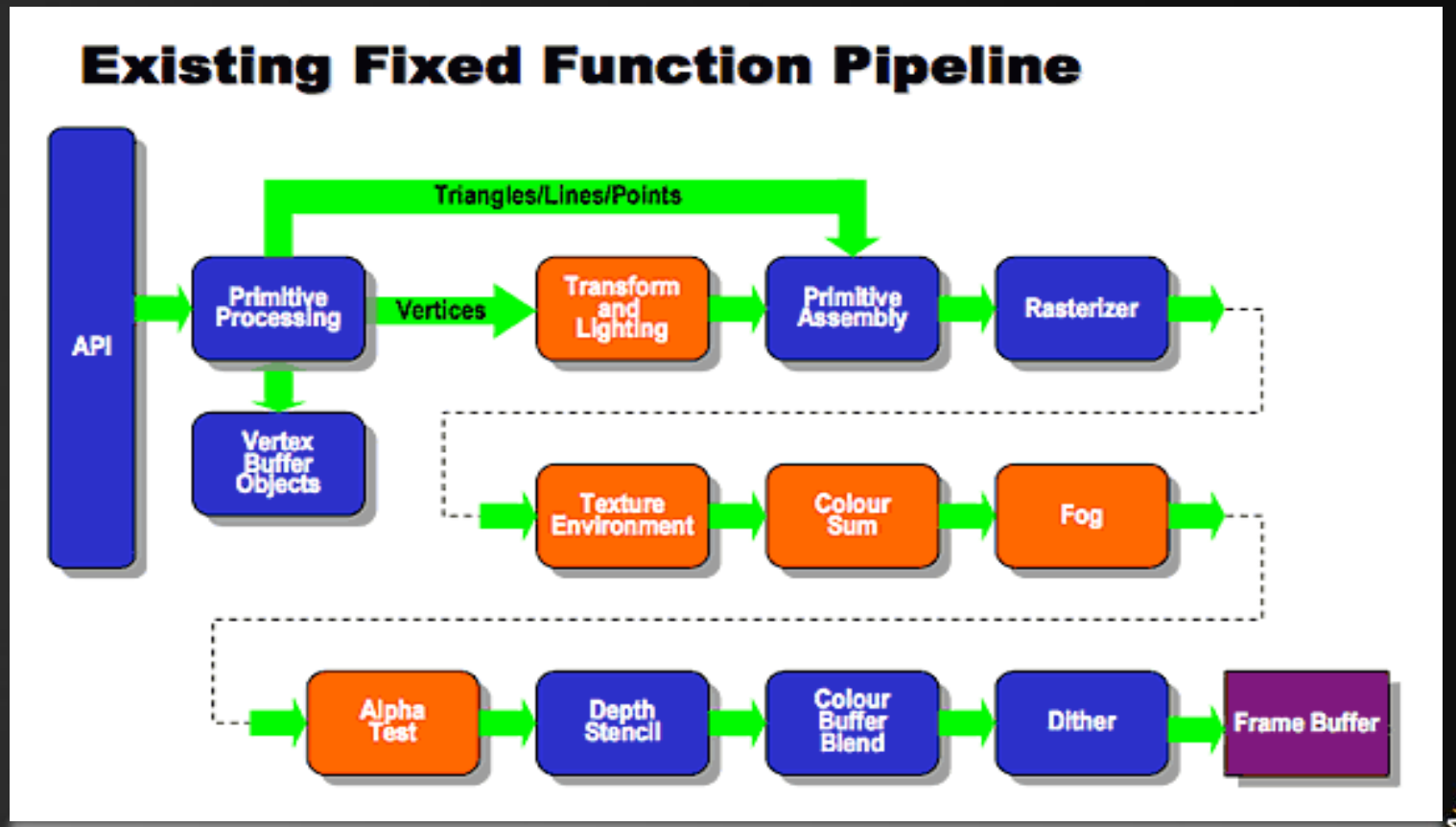
NVidia NV-1
(Diamond Edge 3D)
1995

3Dfx –
Diamond Monster 3D
1996

Introduction

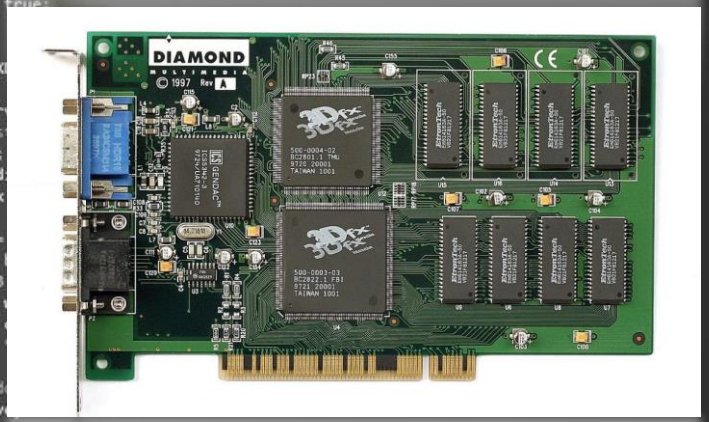
A Brief History of GPGPU

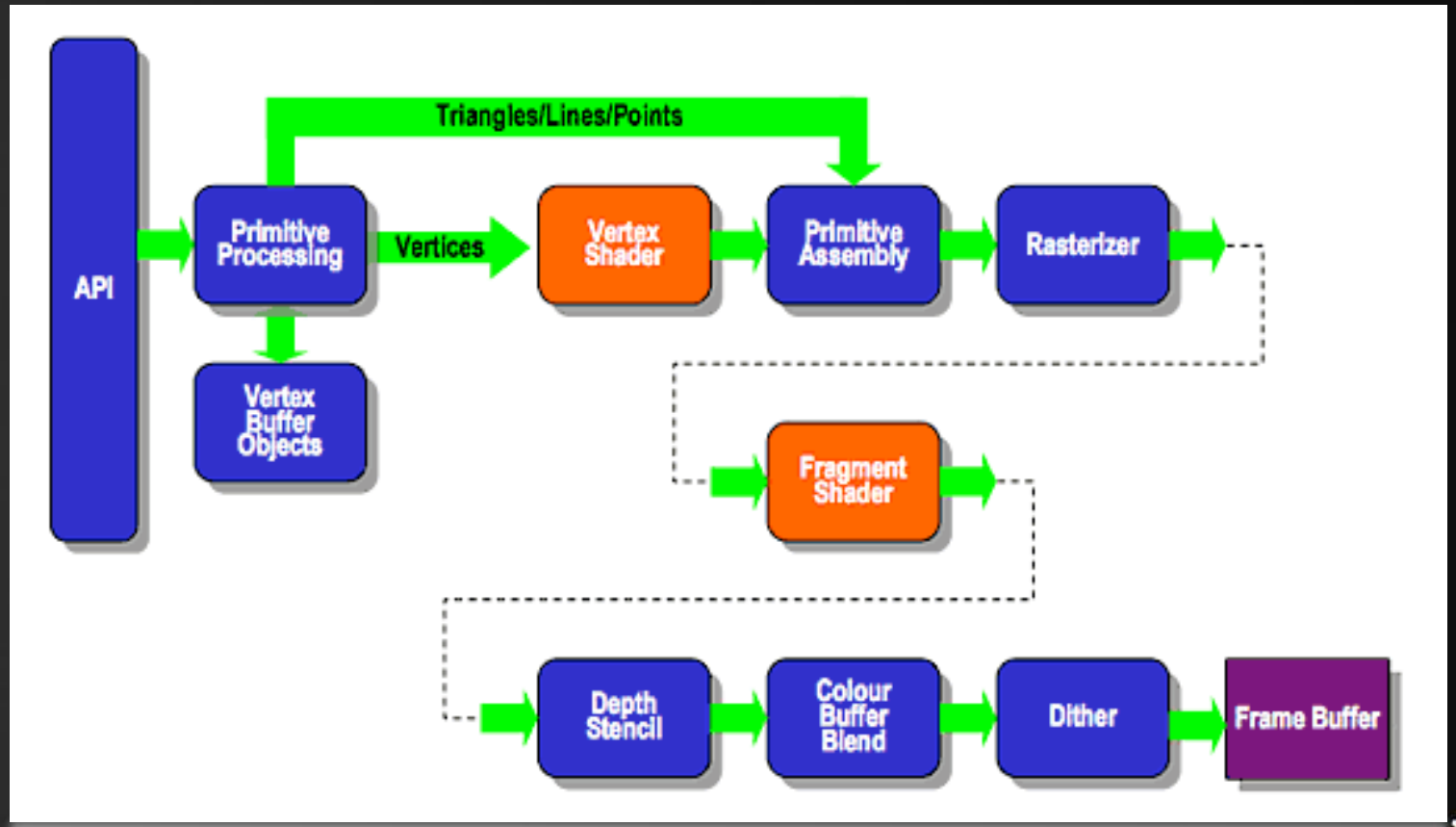




Introduction

A Brief History of GPGPU





Introduction

A Brief History of GPGPU

```
Ok
load"cas:GUSANO"
Skip :RANA
Found:GUSANO
Ok
list 1-100
65 OPEN "GRP:" FOR OUTPUT AS#1
70  STOP ON:ON STOP GOSUB 4000
71 COLOR15,15,15:SCREEN2
72 FORF=1TO8
73 READA:S$=S$+CHR$(A)
74 NEXTF
75 SPRITE$(1)=S$
76 COLOR15,4,13:S$=""
80 SOUND 7,255:SOUND8,15:RR=25
90 SCREEN2,2:PRESET(130,5):PRINT #1,"
MAX.:";PEEK(-1200)
100 DIMX3(500),Y3(500):PSET(35,5),4:P
RINT#1,"PUNTOS:  0":PSET(200,5),4:PRIN
T#1,"@":PSET(212,5),4:PRINT#1,"@":XB=
200:Q=1:RR=25
Ok
color  auto  goto  list  run
```

POTATO SALAD

Some good cooks sprinkle grated pimiento cheese on this

- 4 cups diced cooked potatoes
- 1 cup sliced celery
- 3 hard-cooked eggs, cut up
- ½ cup finely cut onion or sliced green onions
- ¼ cup sliced radishes
- 1 cup mayonnaise
- 1 tablespoon vinegar
- 1 teaspoon prepared mustard
- 1½ to 2 teaspoons salt
- ⅛ teaspoon pepper
- Lettuce

Mix all the ingredients in a bowl. Cover and refrigerate several hours so flavors can blend. Serve on crisp lettuce. Makes 6 servings.

Introduction

A Brief History of GPGPU

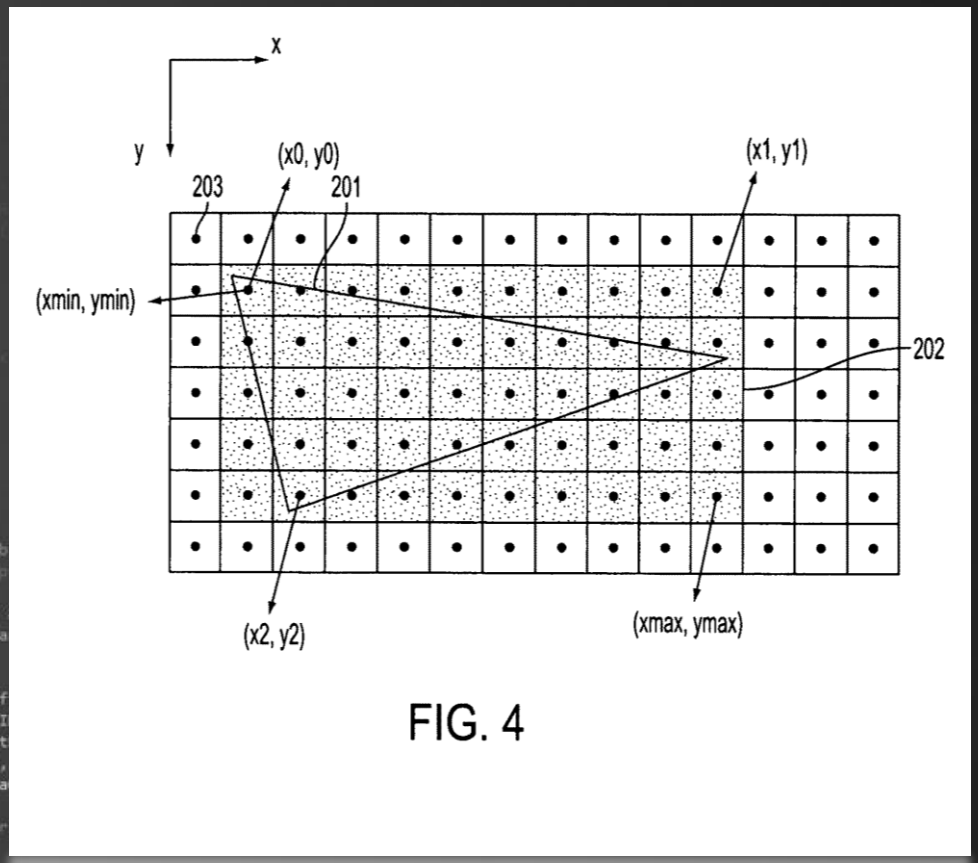


FIG. 4

GPU - conveyor belt:

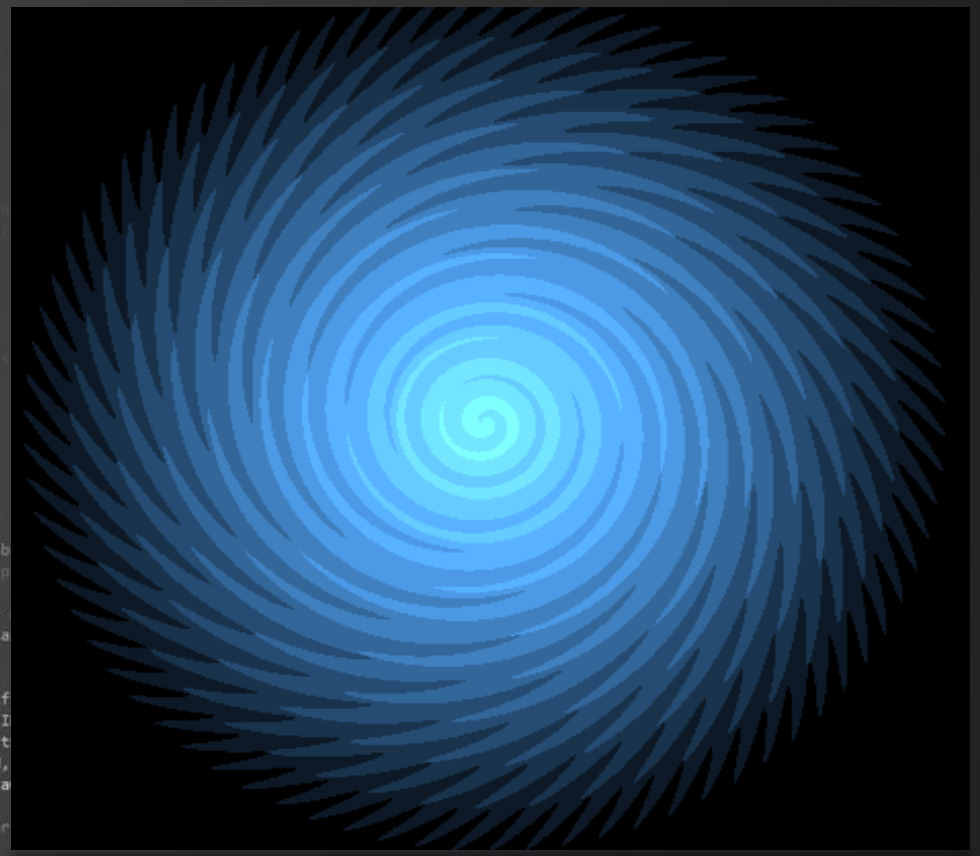
input = vertices + connectivity

step 1: transform
step 2: rasterize
step 3: shade
step 4: z-test

output = pixels

Introduction

A Brief History of GPGPU



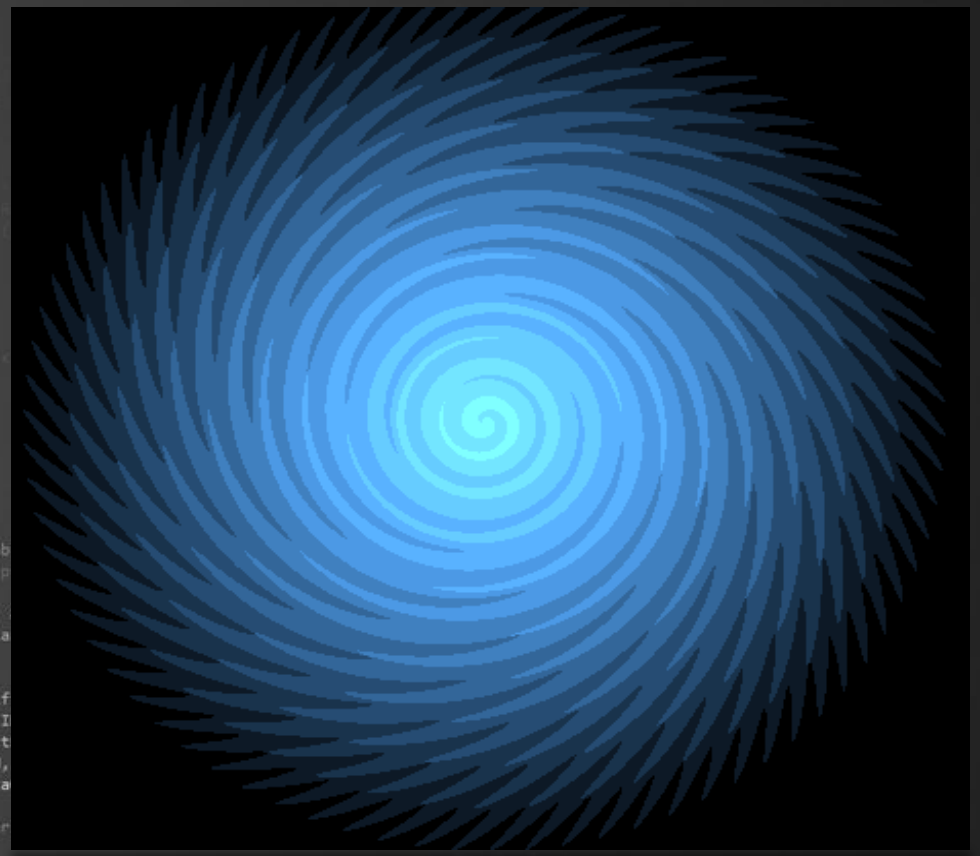
https://www.shadertoy.com/view/4sjSRt

```
void main(void)
{
    float t = iGlobalTime;
    vec2 uv = gl_FragCoord.xy / iResolution.y;
    float r = length(uv), a = atan(uv.y,uv.x);
    float i = floor(r*10);
    a *= floor(pow(128,i/10));
    a += 20.*sin(0.5*t)+123.34*i-100.*
         (r*i/10)*cos(0.5*t);
    r += (0.5+0.5*cos(a)) / 10;
    r = floor(N*r)/10;
    gl_FragColor = (1-r)*vec4(0.5,1,1.5,1);
}
```

GLSL ES code

Introduction

A Brief History of GPGPU



GPUs perform well because they have a constrained execution model, based on massive parallelism.

CPU: Designed to run one thread as fast as possible.

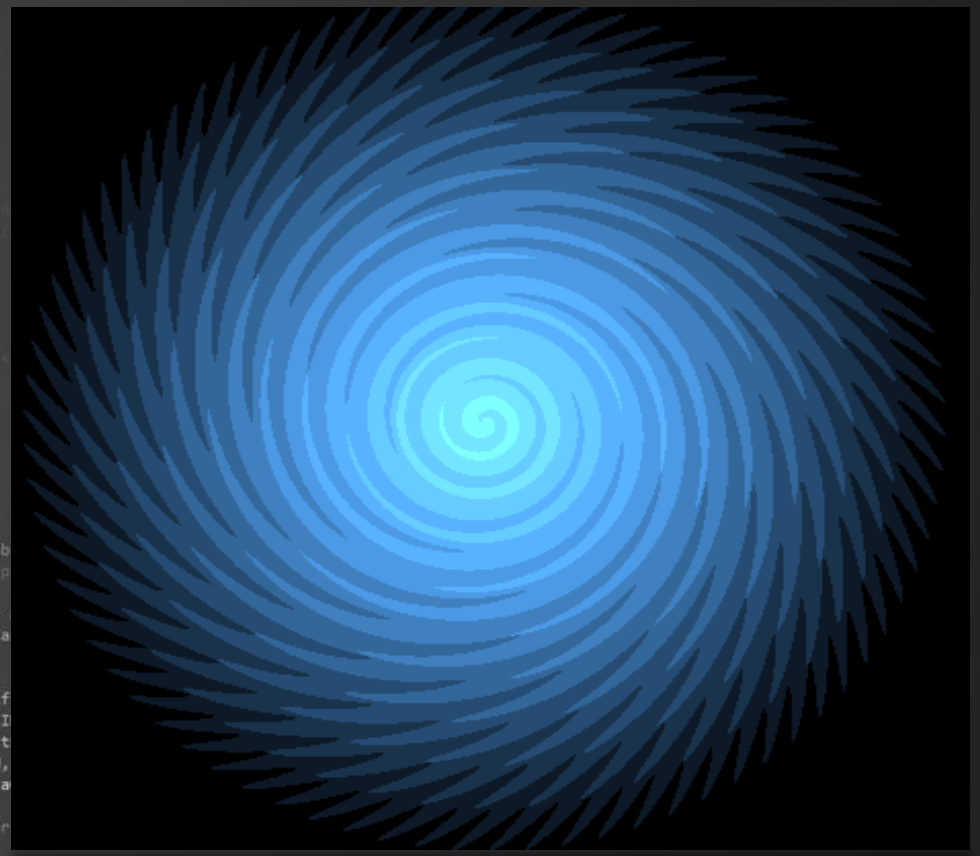
- Use caches to minimize memory latency
- Use pipelines and branch prediction
- Multi-core processing: *task parallelism*

Tricks:

- SIMD
- “Hyperthreading”

Introduction

A Brief History of GPGPU



GPUs perform well because they have a constrained execution model, based on massive parallelism.

GPU: Designed to combat latency using many threads.

- Hide latency by computation
- Maximize parallelism
- Streaming processing ➔ Data parallelism ➔ SIMT

Tricks:

- Use typical GPU hardware (filtering etc.)
- Cache anyway

Introduction

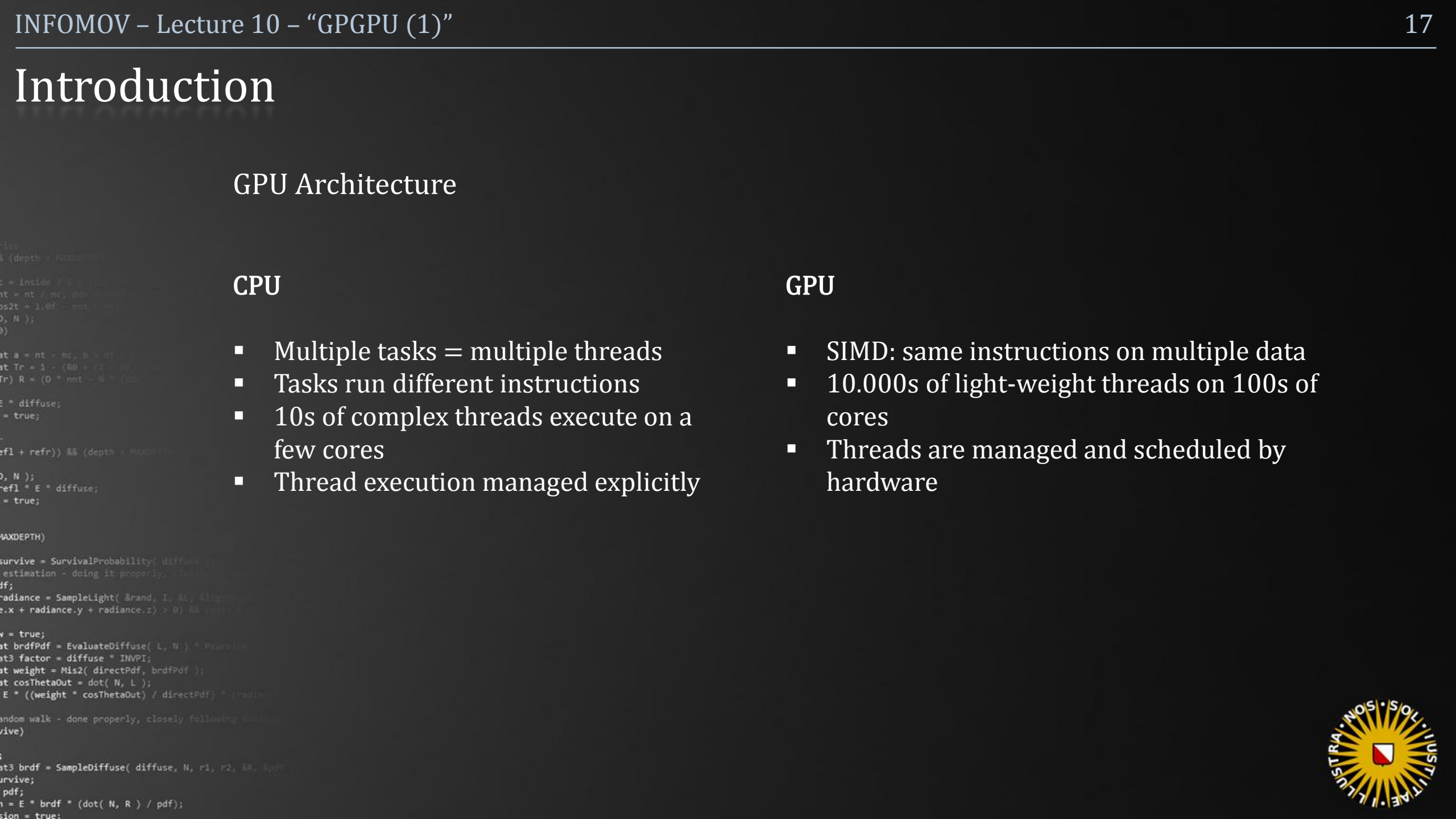
GPU Architecture

CPU

- Multiple tasks = multiple threads
- Tasks run different instructions
- 10s of complex threads execute on a few cores
- Thread execution managed explicitly

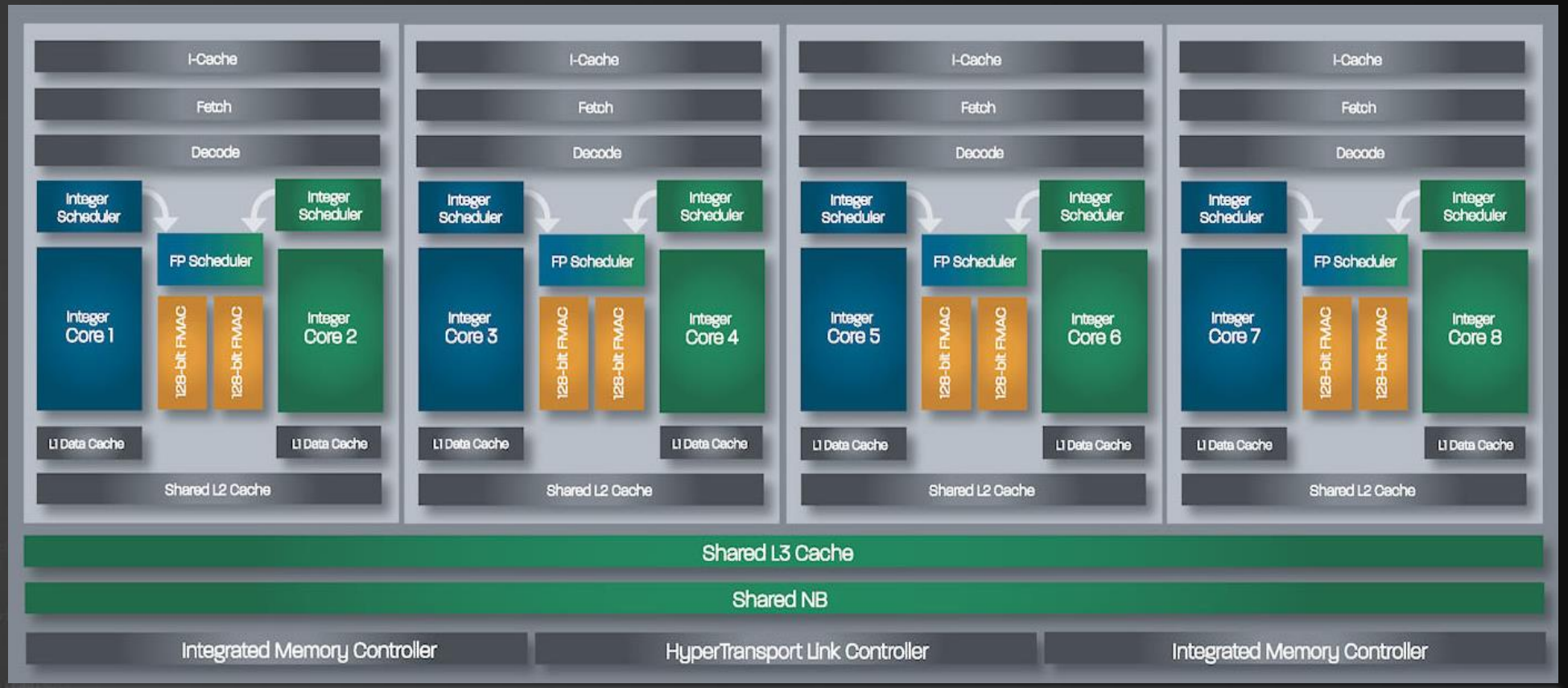
GPU

- SIMD: same instructions on multiple data
- 10.000s of light-weight threads on 100s of cores
- Threads are managed and scheduled by hardware



Introduction

CPU Architecture

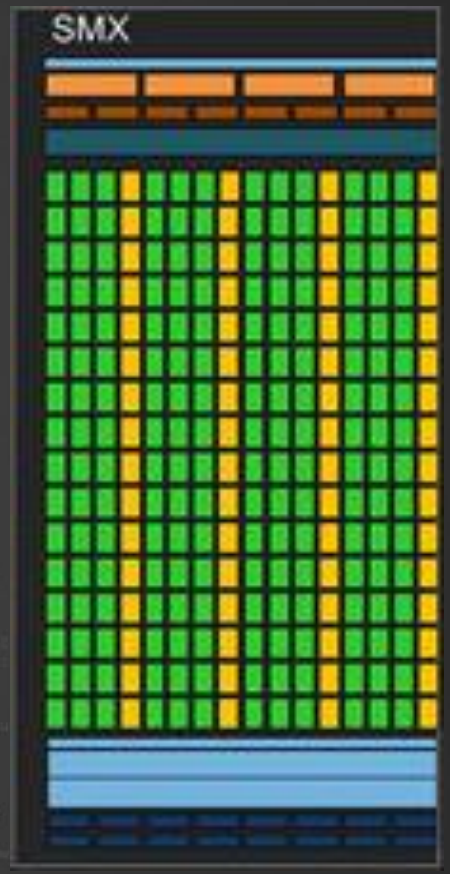


Introduction

GPU Architecture

Introduction

GPU Architecture

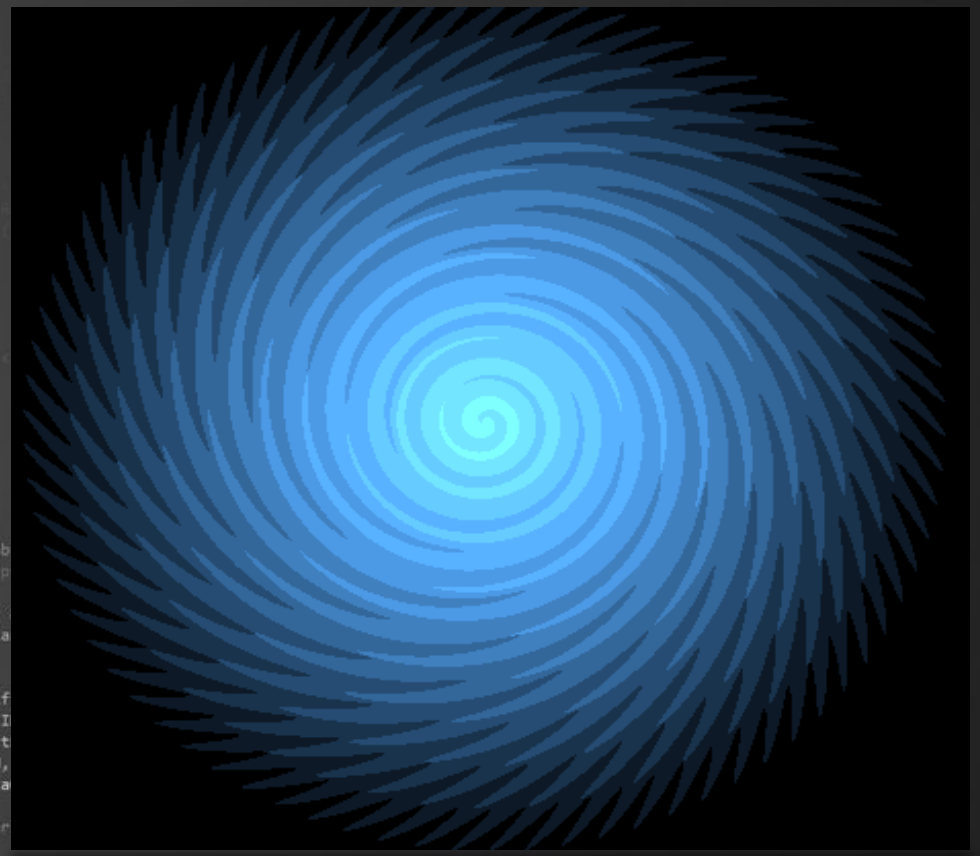


SIMT Thread execution:

- Group 32 threads (vertices, pixels, primitives) into warps
- Each warp executes the same instruction
- In case of latency, switch to different warp (thus: switch out 32 threads for 32 different threads)
- Flow control: …

Introduction

GPGPU Programming

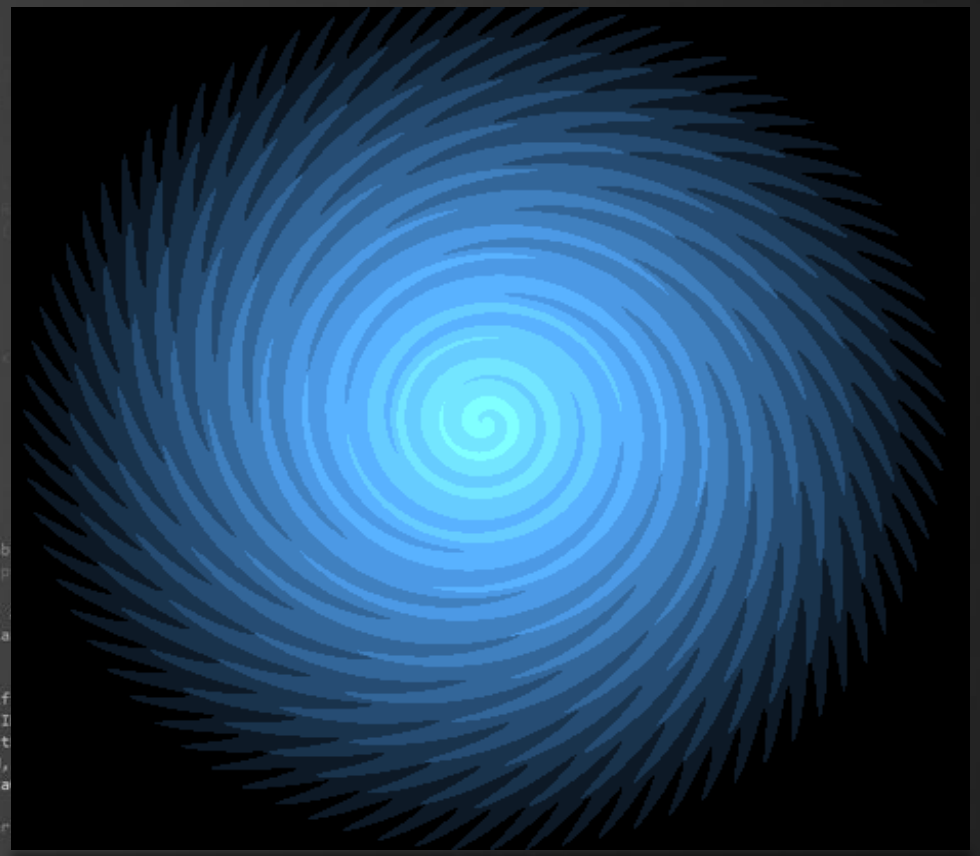


https://www.shadertoy.com/view/4sjSRt

```
void main(void)
{
    float t = iGlobalTime;
    vec2 uv = gl_FragCoord.xy / iResolution.y;
    float r = length(uv), a = atan(uv.y,uv.x);
    float i = floor(r*10);
    a *= floor(pow(128,i/10));
    a += 20.*sin(0.5*t)+123.34*i-100.*
         (r*i/10)*cos(0.5*t);
    r += (0.5+0.5*cos(a)) / 10;
    r = floor(N*r)/10;
    gl_FragColor = (1-r)*vec4(0.5,1,1.5,1);
}
```

Introduction

GPGPU Programming



Easy to port to GPU:

- Image postprocessing
- Particle effects
- Ray tracing
- …

Today’s Agenda:

- Introduction to GPGPU
- Example: Voronoi Noise
- GPGPU Programming Model
- OpenCL Template

Example

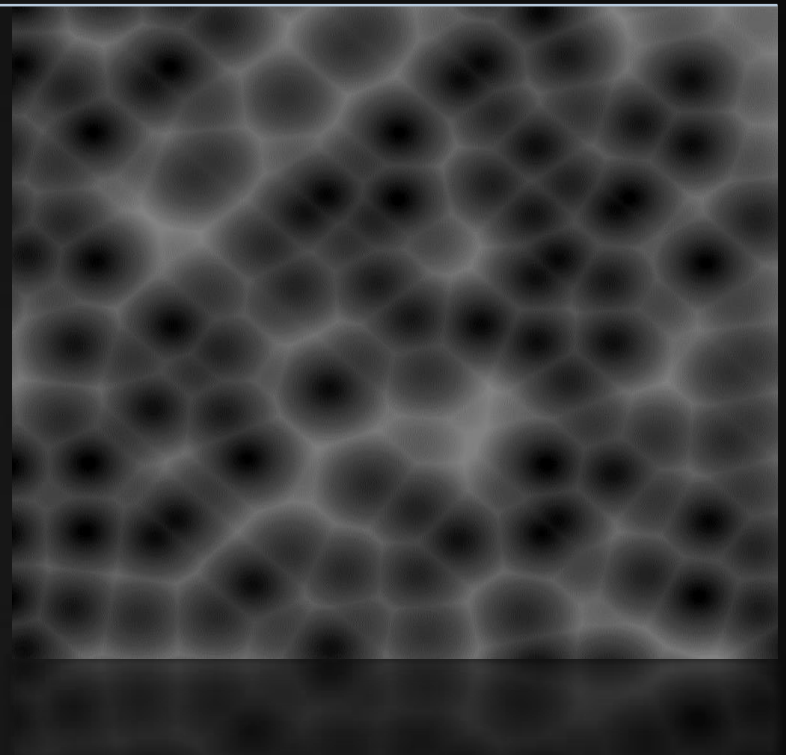
Voronoi Noise / Worley Noise*

Given a set of points, and a position x in $\mathbb{R}^2$,
$F_1(x)$ = distance of x to closest point.

For Worley noise, we use a Poisson distribution for the points.
In a lattice, we can generate this as follows:

1. The expected number of points in a region is constant (Poisson);
2. The probability of each point count in a region is computed using the discrete Poisson distribution function;
3. The point count and coordinates of each point can be determined using a random seed based on the coordinates of the region in the lattice.

*A Cellular Texture Basis Function, Worley, 1996



Example

Voronoi Noise / Worley Noise*

```
vec2 Hash2( vec2 p, float t )
{
   float r = 523.0f * sinf( dot( p, vec2(53.3158f, 43.6143f)
   return vec2( frac( 15.32354f * r + t ), frac( 17.25865f *
}

float Noise( vec2 p, float t )
{
   p *= 16;
   float d = 1.0e10;
   vec2 fp = floor( p );
   for( int xo = -1; xo <= 1; xo++ ) for (int yo = -1; yo <= 1; yo++)
   {
      vec2 tp = fp + vec2(xo, yo);
      tp = p - tp - Hash2( vec2( fmod( tp.x, 16.0f ),
         fmod( tp.y, 16.0f ) ), t ), d = min( d, dot( tp, tp ) );
   }
   return sqrtf( d );
}
```

* https://www.shadertoy.com/view/4djGRh

Characteristics of this code:

- Pixels are independent, and can be calculated in arbitrary order;
- No access to data (other than function arguments and local variables);
- Very compute-intensive;
- Very little input data required.

Example

Voronoi Noise / Worley Noise*

Timing of the Voronoi code in C++:

~750ms per image (800 x 512 pixels).

Executing the same code in OpenCL (GPU: GTX480):

~12ms (62x faster).

Example

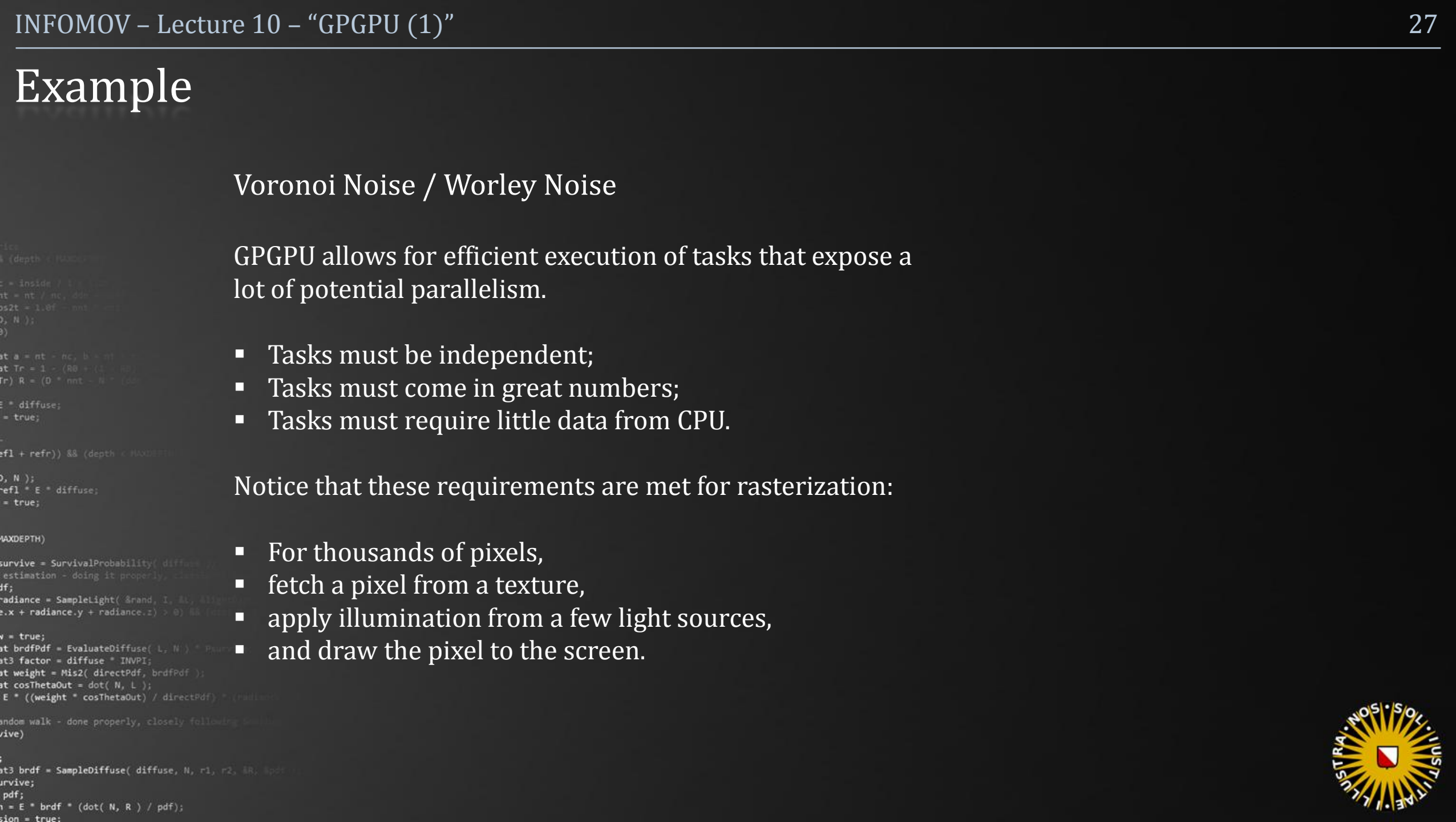
Voronoi Noise / Worley Noise

GPGPU allows for efficient execution of tasks that expose a lot of potential parallelism.

- Tasks must be independent;
- Tasks must come in great numbers;
- Tasks must require little data from CPU.

Notice that these requirements are met for rasterization:

- For thousands of pixels,
- fetch a pixel from a texture,
- apply illumination from a few light sources,
- and draw the pixel to the screen.



Today’s Agenda:

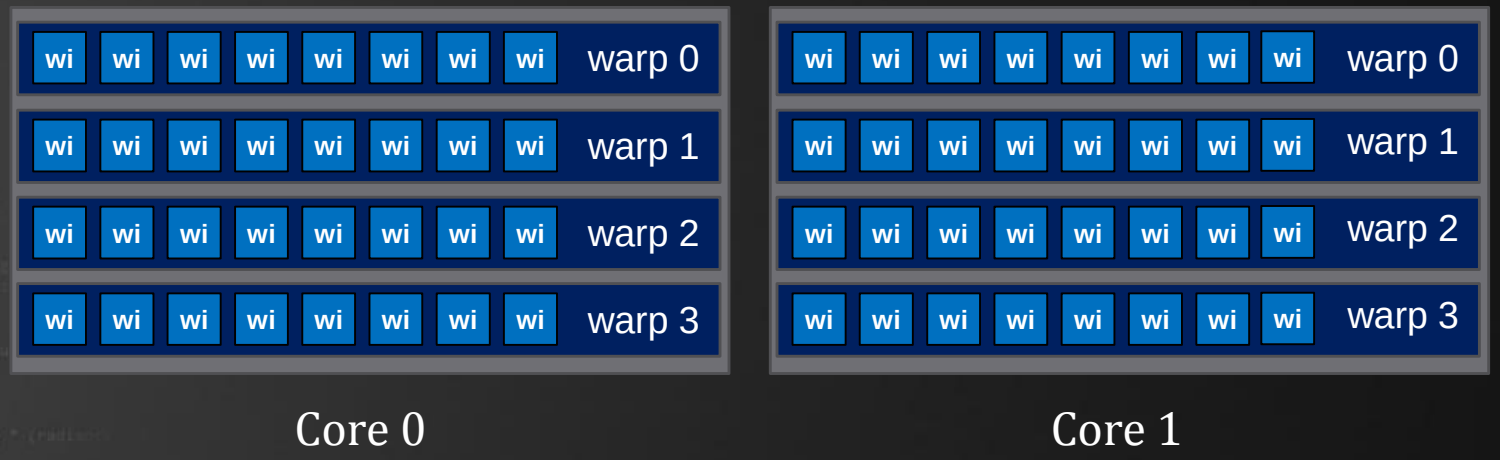
- Introduction to GPGPU
- Example: Voronoi Noise
- GPGPU Programming Model
- OpenCL Template

Programming Model

GPU Architecture

A typical GPU:

- Has a small number of ‘shading multiprocessors’ (comparable to CPU cores);
- Each core runs a small number of ‘warps’ (comparable to hyperthreading);
- Each warp consists of 32 ‘threads’ that run in lockstep (comparable to SIMD).



Programming Model

GPU Architecture

Multiple warps on a core:

The core will switch between warps whenever there is a stall in the warp (e.g., the warp is waiting for memory). Latencies are thus hidden by having many tasks.
This is only possible if you feed the GPU enough tasks: $cores \times warps \times 32$.

Core 0

Core 1

Programming Model

GPU Architecture

Threads in a warp running in lockstep:

At each cycle, all ‘threads’ in a warp must execute the same instruction. Conditional code is handled by temporarily disabling threads for which the condition is not true. If-then-else is handled by sequentially executing the ‘if’ and ‘else’ branches. Conditional code thus reduces the number of active threads (occupancy). Note the similarity to SIMD code!

Core 0

Core 1

Programming Model

SIMT

The GPU execution model is referred to as SIMT: Single Instruction, Multiple Threads.

A GPU is therefore a very wide vector processor.

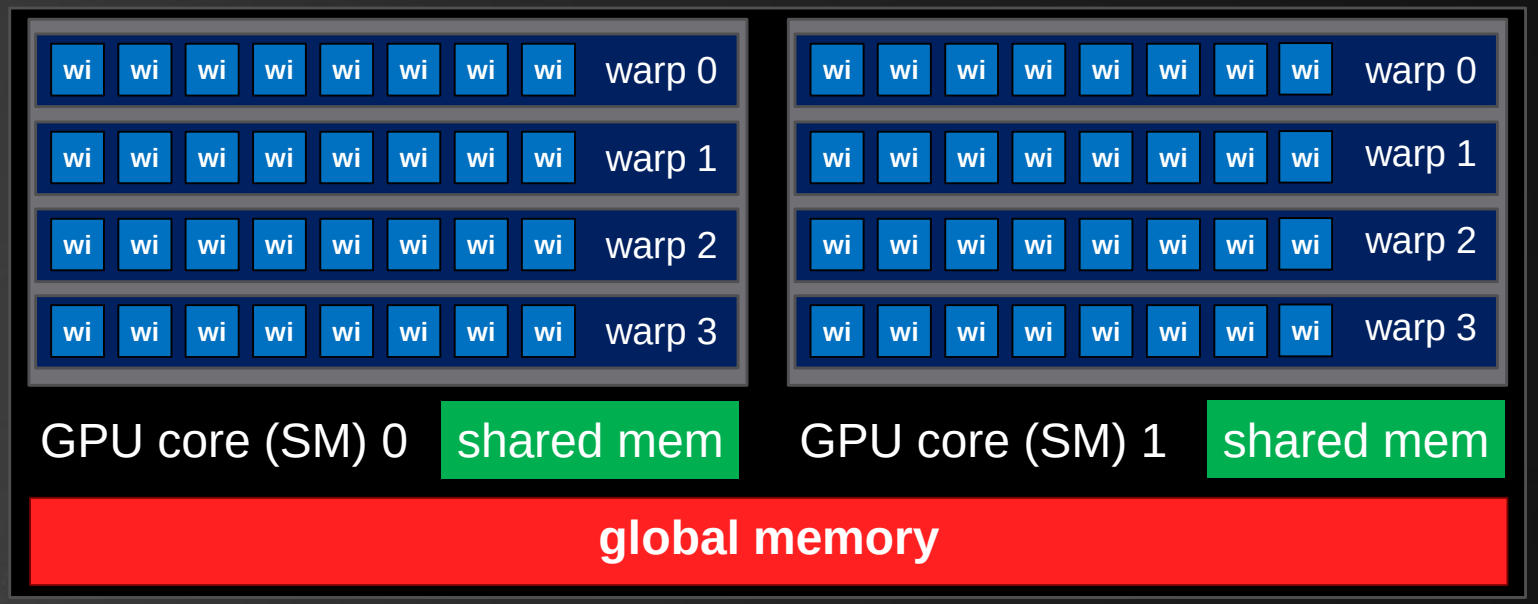
Converting code to GPGPU is similar to vectorizing code on the CPU.

Core 0

Core 1

Programming Model

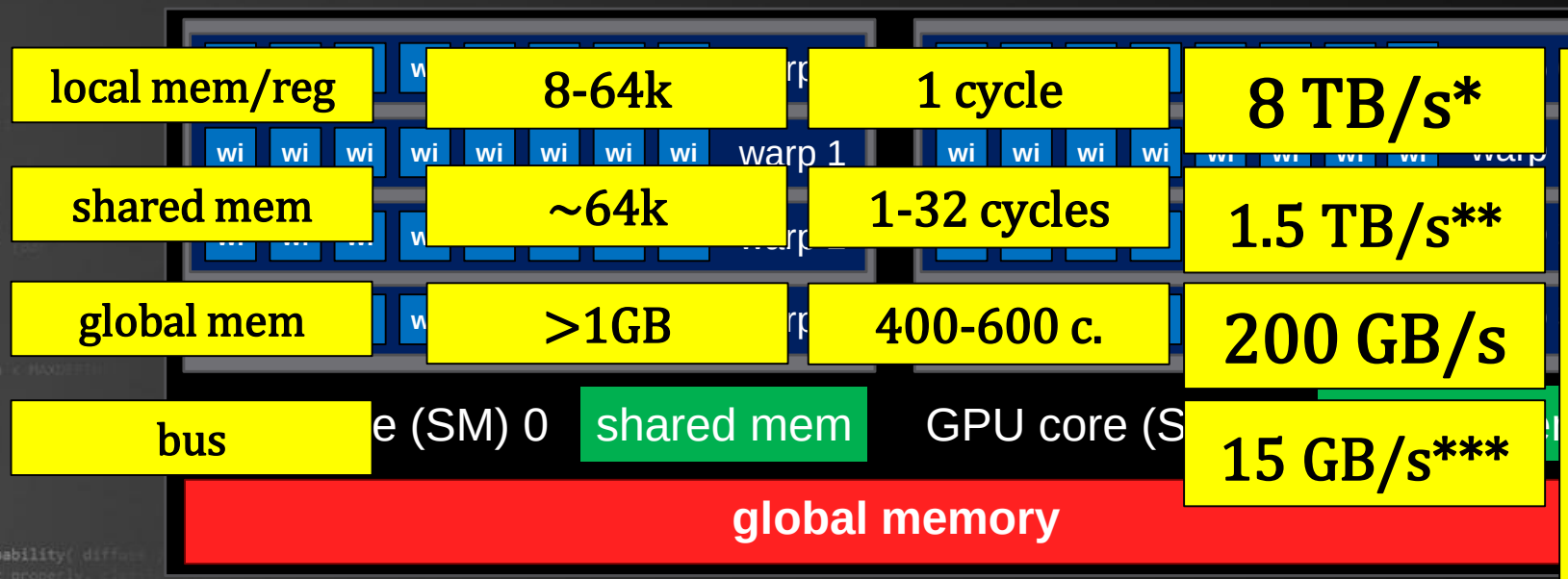
GPU Memory Model



- Each SM has a large number of registers, which is shared between the warps.
- Each SM has shared memory, comparable to L1 cache on a CPU.
- The GPU has global memory, comparable to CPU RAM.
- The GPU communicates with the ‘host’ over a bus.

Programming Model

GPU Memory Model



For reference, Core i7-3960X:

- RAM bandwidth for quad-channel DDR3-1866 memory: **18.1GB/s**
- L2 bandwidth: 46.8GB/s*

*: Molka et al., Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. 2014.

** Values for NVidia G80 (Tesla)*
*** Fermi uses L1 cache*
**** PCIe 3.0*

Programming Model

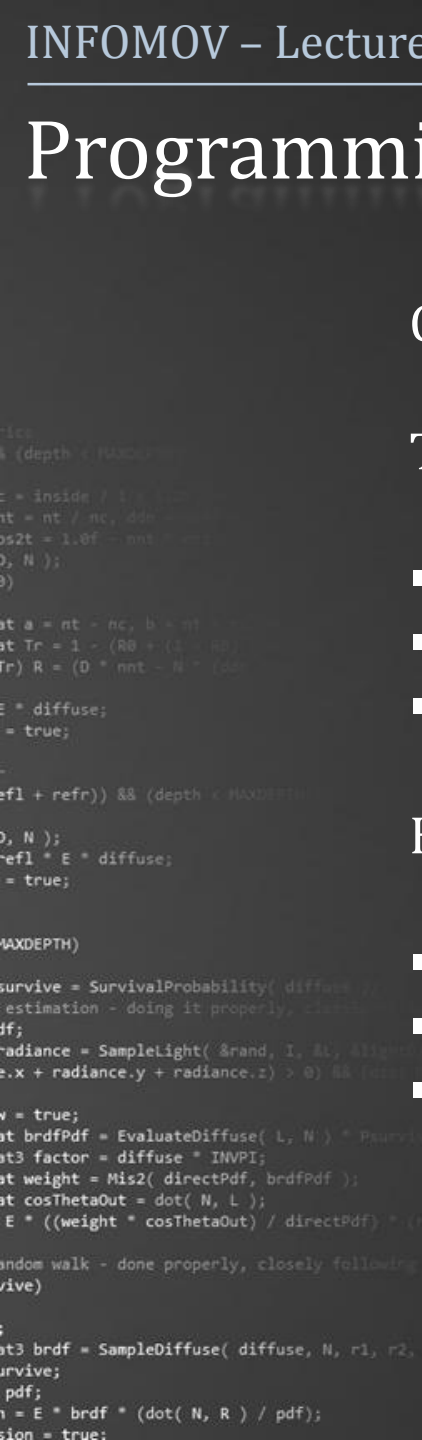
GPU Memory Model

There appear to be many similarities between a CPU and a GPU:

- Cores, with hyperthreading
- A memory hierarchy
- SIMD

However, there are fundamental differences in each of these.

- One GPU core will execute 4-8 warps (instead of 2 on the CPU);
- The memory hierarchy is explicit on the GPU, rather than implicit on the CPU;
- GPU SIMD on the other hand is implicit (SIMT model).



Programming Model

GPGPU Programming Model

A number of APIs is available to run general purpose GPU code:

Pixel shaders:
- Executed as part of the rendering pipeline
- The number of tasks is equal to the number of pixels

Graphics-centric work: Shading, postprocessing (using a full-screen quad)

Compute shaders:
- Executed as part of the rendering pipeline
- More control over the number of tasks

Graphics-centric work: Preparing data, output to textures / vertex buffers / …

OpenCL / CUDA:
- Executed independent of rendering pipeline
- Full control over memory hierarchy and division of tasks over hardware

General Purpose

Programming Model

GPGPU Programming Model

APIs like CUDA and OpenCL may look like C, but are in fact heavily influenced by the underlying hardware model.

```
__kernel void task( write_only image2d_t outimg, __global uint* logBuffer )
{
   float t = 1;
   int column = get_global_id( 0 );
   int line = get_global_id( 1 );
   float c = Cells( (float2)((float)column / 500, (float)line / 500), t );
   write_imagef( outimg, (int2)(column, line), c );
}
```

- Kernel: one task (of which we need thousands to run efficiently);
- get_global(0,1): identifies a single task from a 2D array of tasks.

Many threads will execute the same kernel. We can not execute different code in parallel.

Programming Model

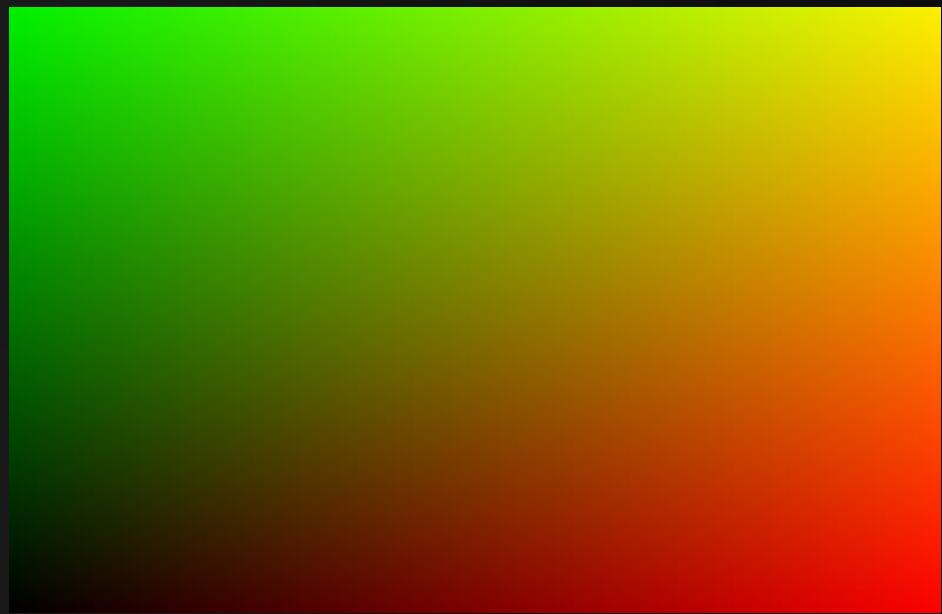
GPGPU Programming Model

Kernels are invoked from the host:

```
size_t workSize[2] = { SCRWIDTH, SCRHEIGHT };
void Kernel::Run( cl_mem* buffers, int count )
{
    …
    clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, NULL, 0, 0, 0 );
    …
}
```

Device code:

```
__kernel void main( write_only image2d_t outimg )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    float red = column / 800.;
    float green = line / 480.;
    float4 color = { red, green, 0, 1 };
    write_imagef( outimg, (int2)(column, line), color );
}
```



Programming Model

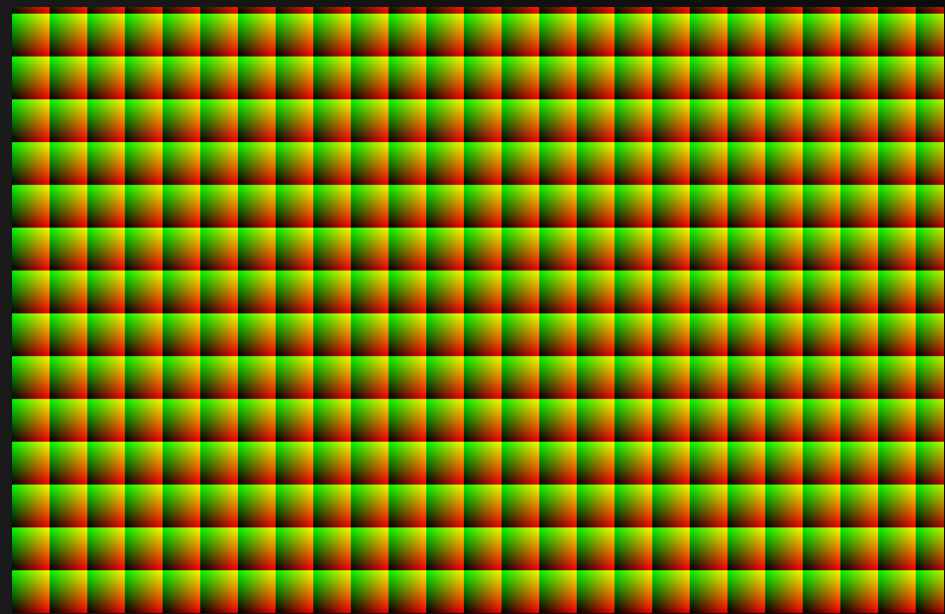
GPGPU Programming Model

Kernels are invoked from the host:

```
size_t workSize[2] = { SCRWIDTH, SCRHEIGHT }, localSize[2] = { 32, 32 };
void Kernel::Run( cl_mem* buffers, int count )
{
    …
    clEnqueueNDRangeKernel( queue, kernel, 2, 0, workSize, localSize, 0, 0, 0 );
    …
}
```

Device code:

```
__kernel void main( write_only image2d_t outimg )
{
    int column = get_global_id( 0 );
    int line = get_global_id( 1 );
    float red = get_local_id( 0 ) / 32.;
    float green = get_local_id( 1 ) / 32.;
    float4 color = { red, green, 0, 1 };
    write_imagef( outimg, (int2)(column, line), color );
}
```



Today’s Agenda:

- Introduction to GPGPU
- Example: Voronoi Noise
- GPGPU Programming Model
- OpenCL Template

Template

OCL_Lab: The Familiar Template

The OpenCL template is a basic experimentation framework for OpenCL.
Game::Tick implements the following functionality:

1. Set arguments for the OpenCL kernel;
2. Execute the OpenCL kernel (which stores output in an OpenGL texture);
3. Draw a full-screen quad using a shader.

You can find the OpenCL code in program.cl;
The shader is defined in vignette.frag.

/INFOMOV/

END of “GPGPU (1)”

next lecture: “GPGPU (2)”