Web Based Graphics & Virtual Reality Systems

Programmable Shaders

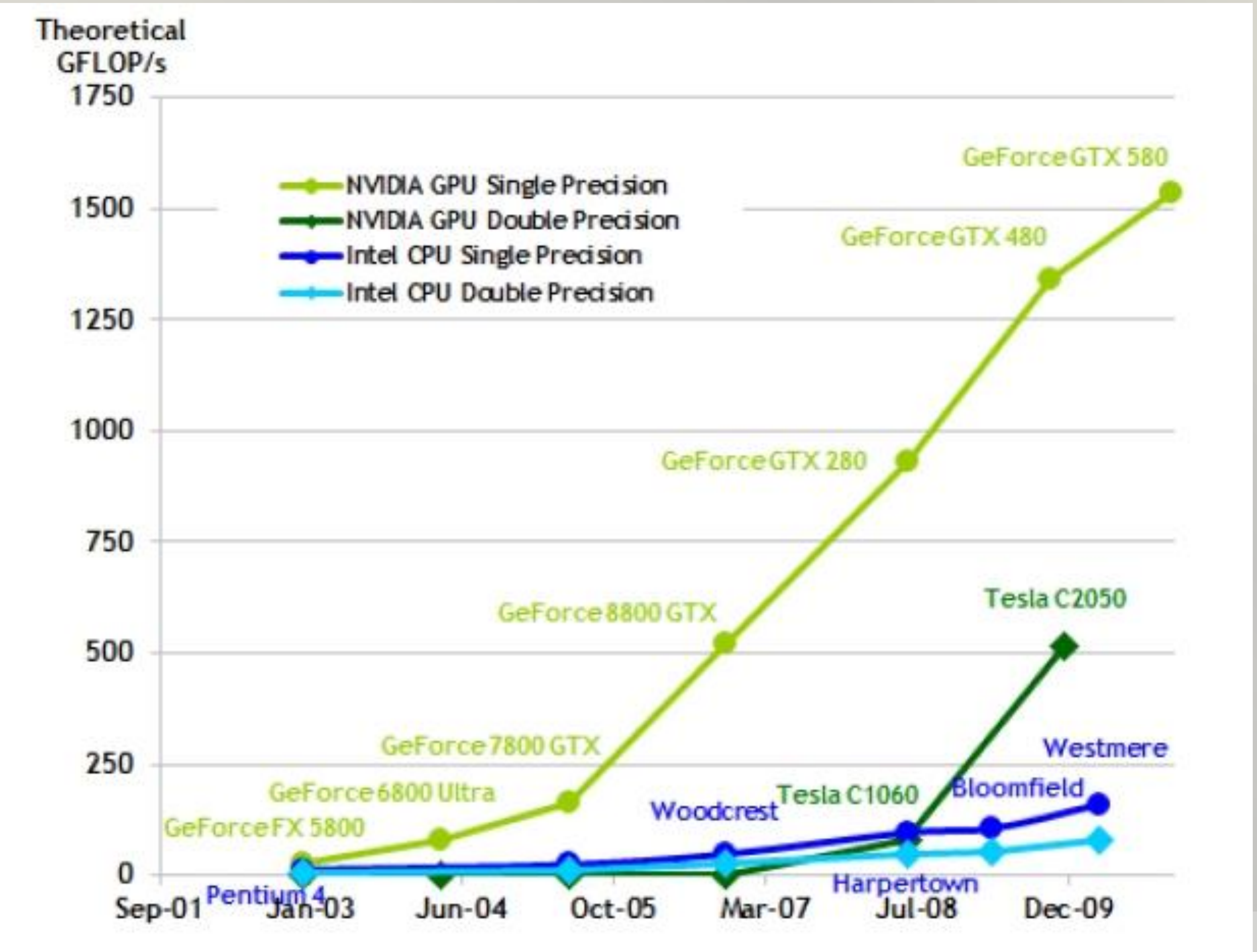
CPU and GPU

- The concept of GPU starts from early 2000
- Specially tailored for processing rendering tasks of graphics
- GPU become programmable in its rendering process
  - Shader will be introduced later in this lecture

CPU and GPU

- Basic difference between CPU and GPU
  - Large number of parallel units in the pipeline
- This design is especially useful for graphics applications
  - Large number of vertices undergo similar or same transformation and operations
- Major GPU manufacturers
  - Nvidia : Geforce Series
  - AMD / ATI : Radeon Series
  - Intel

Computing Power of CPU and GPU



GFLOP :

FLoating-point OPerations per Second

Computing Power of CPU and GPU

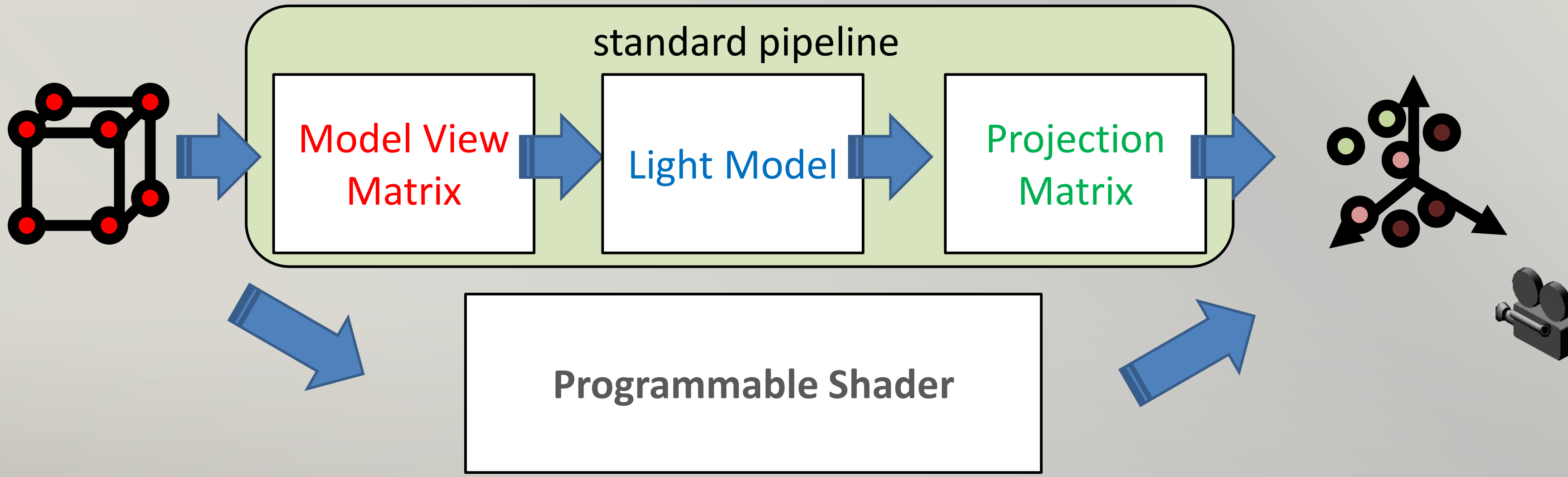
- Compute
  - Intel Core i7 – 4 cores – 100 GFLOP
  - NVIDIA GTX280 – 240 cores – 1 TFLOP
- Memory Bandwidth
  - System Memory – 60 GB/s
  - NVIDIA GTX680 – 200 GB/s
  - PCI-E Gen3 – 8 GB/s
- 2012 – GPU peak FLOP about 10x CPU

Programmable Shader

- The standard rendering pipeline are being hard-wired in graphics hardware for years
- Until around 10 years before, the graphics hardware becomes programmable
- The small programs which can customize certain parts of the rendering pipeline are called “Shader”

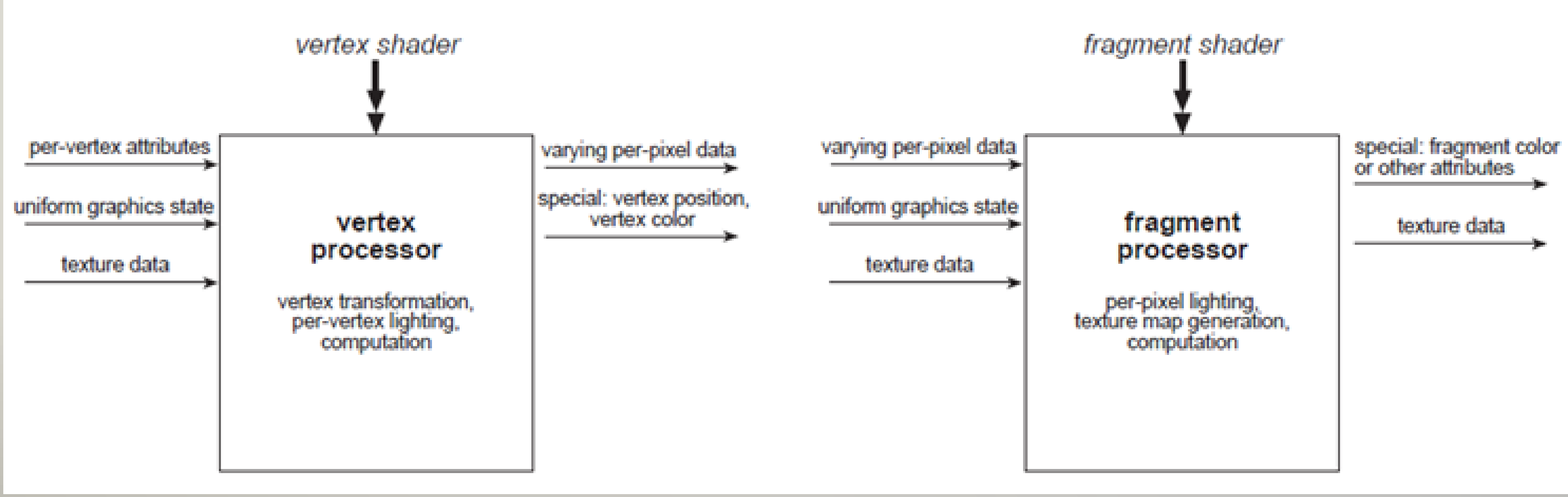
Programmable Shader

- Instead of going through standard pipeline
- We can now write own programs to perform other processing steps wanted
- E.g. for vertex processing :



Programmable Shader

- Two major kinds of shaders are available:
  - Vertex shader (for vertex processing)
  - Fragment shader (for fragment processing)

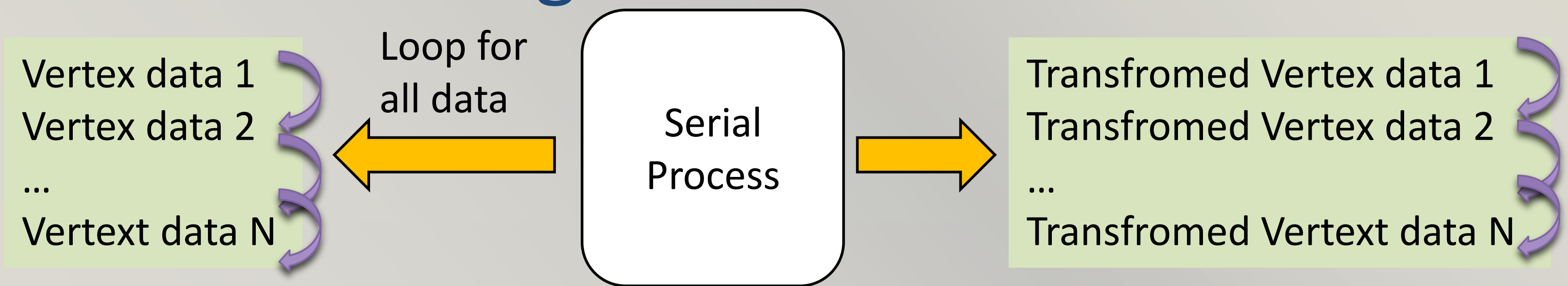


Programmable Shader

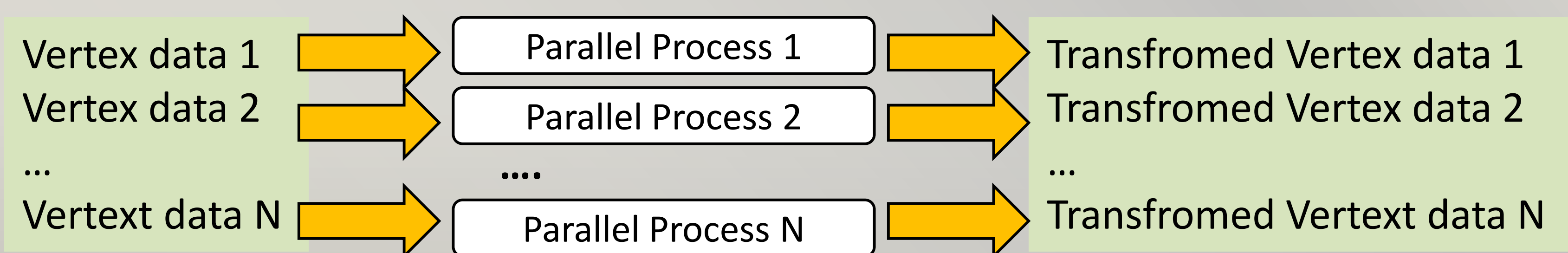
- GPU hardwares are designed to process a large number of vertices and fragments **at the same time**
  - The major difference of GPU shaders to normal programs is that they are being processed in a **parallel manner**
  - Serial program process all data one by one
  - Parallel programs processes all data at the same time

Programmable Shader

- Serial Processing

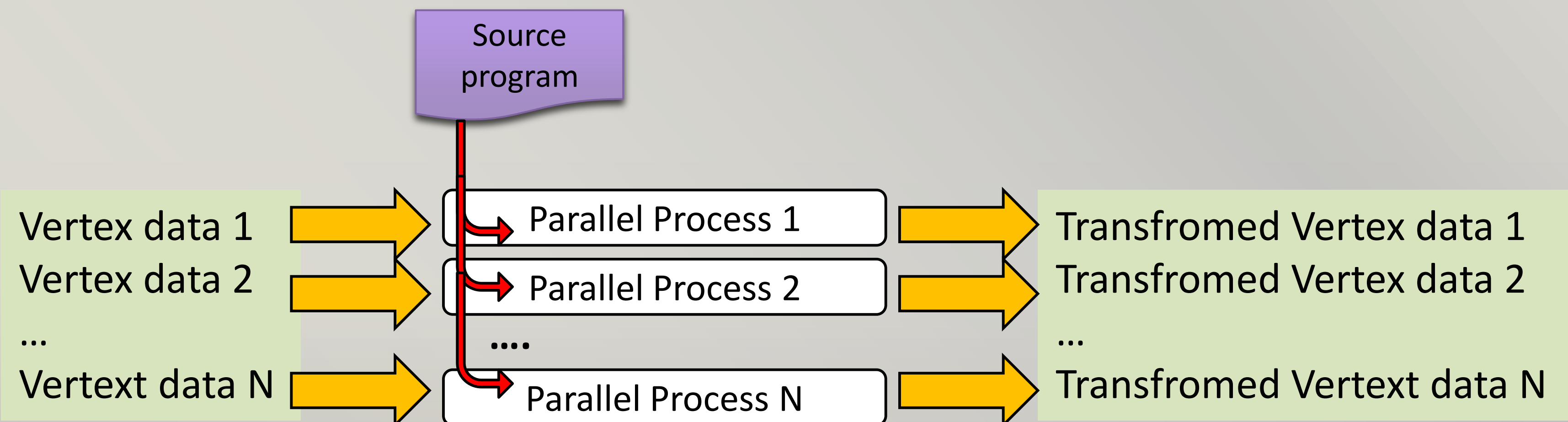


- Parallel Processing



SIMD

- The parallel processing model in GPU is called SIMD (Single Instruction Multiple Data)
- SIMD architecture
  - Incoming data are different , but all program source are the same



Programmable Shader

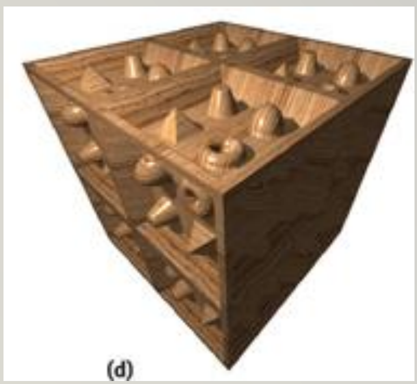
- Although shader is run inside GPU, several languages are available for writing it
  - Cg (Nvidia)
  - GLSL (Open standard)
  - CUDA (Nvidia)
  - OpenCL (Open standard)

Shaders

- Various kinds of shaders are available
  - Vertex Shader
  - Fragment Shader
  - Geometry Shader
- Responsible for handling different elements in corresponding part of the rendering pipeline

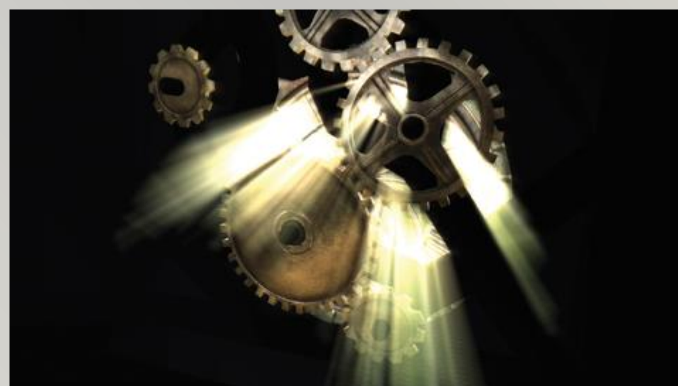
Applications of Shaders

- Special and advanced real-time effects can be done with the use of shaders



Bump mapping

Volumetric lighting

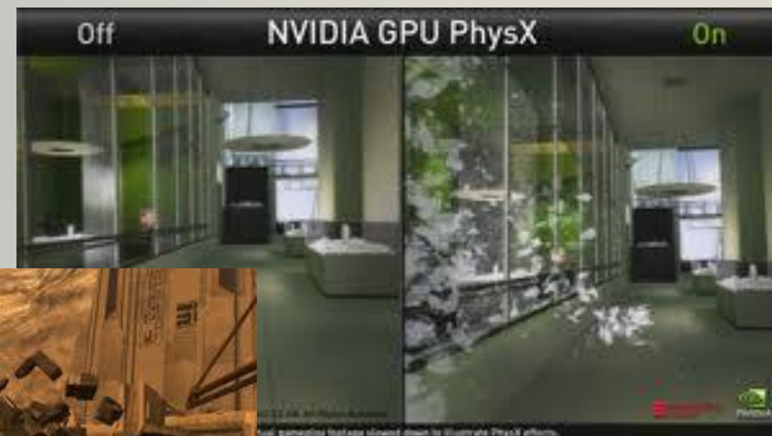


Shadow mapping

Skin rendering

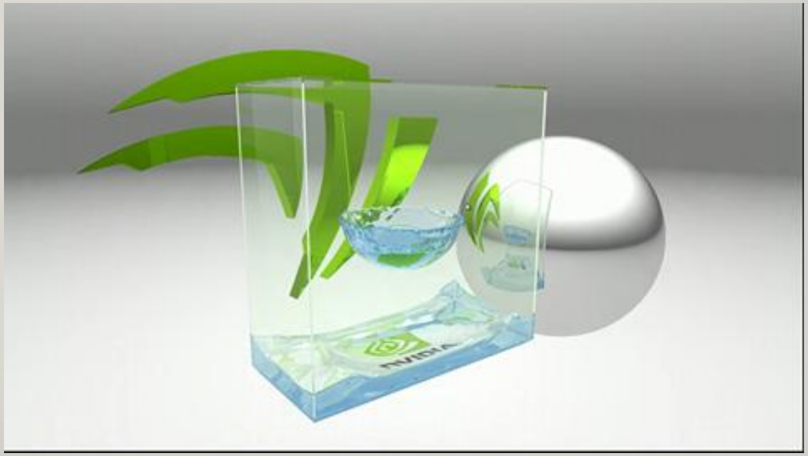
Application: Physics Simulation

- PhysX engine (Nvidia)
- Rigid motions
- Cloth simulation
- Fluid dynamics



Application on Rendering Approaches

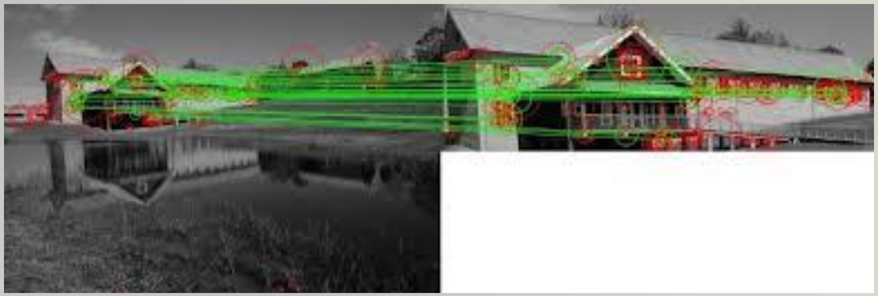
- Real-time non-photorealistic rendering
- Real-time Raytracing



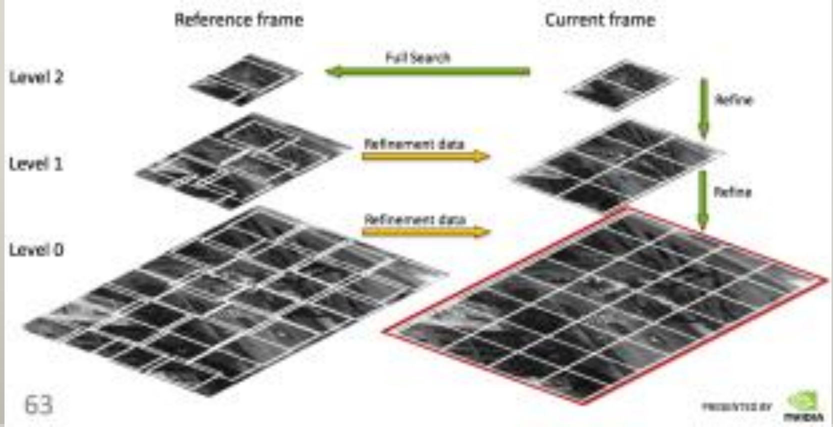
Nvdia GPU Raytracing

Toon Shaded Game

Application: Image and Video Processing



- Image Registration
- Motion Estimation
- Video Encoding and Compression
- Mainly to accelerate computational intensive tasks



GPU Architecture: the basics

- CPU runs machine or assembly code of your compiled program
- GPU runs also assembly code, but from your compiled shader

Shader

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

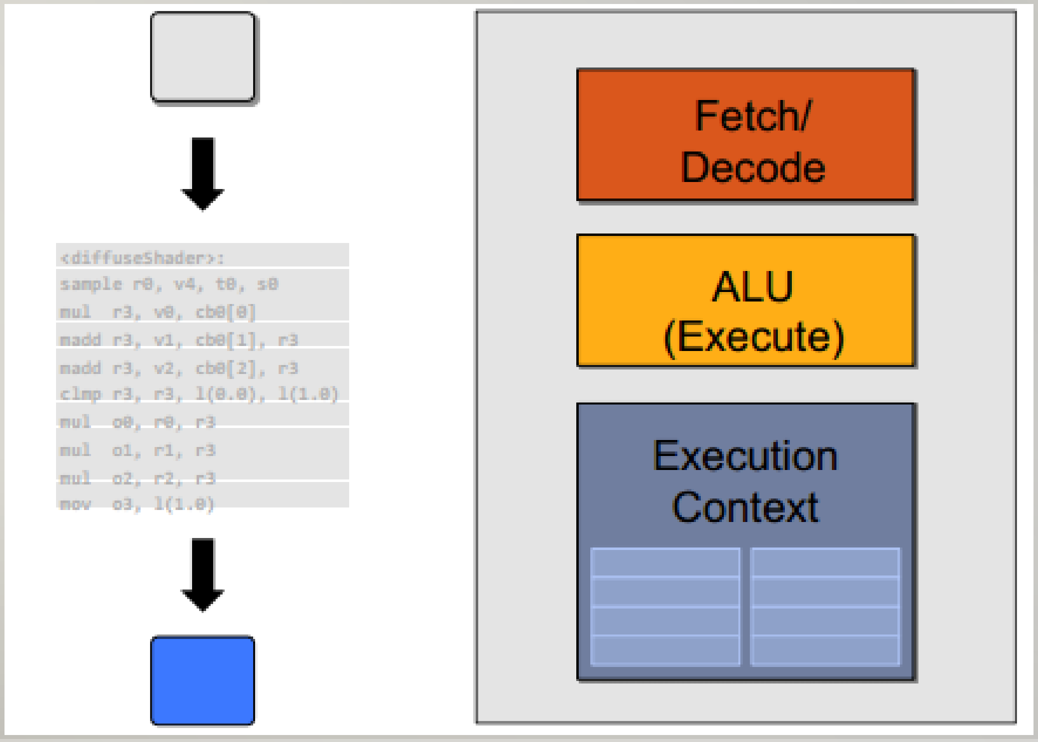
Compile

Assembly code

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

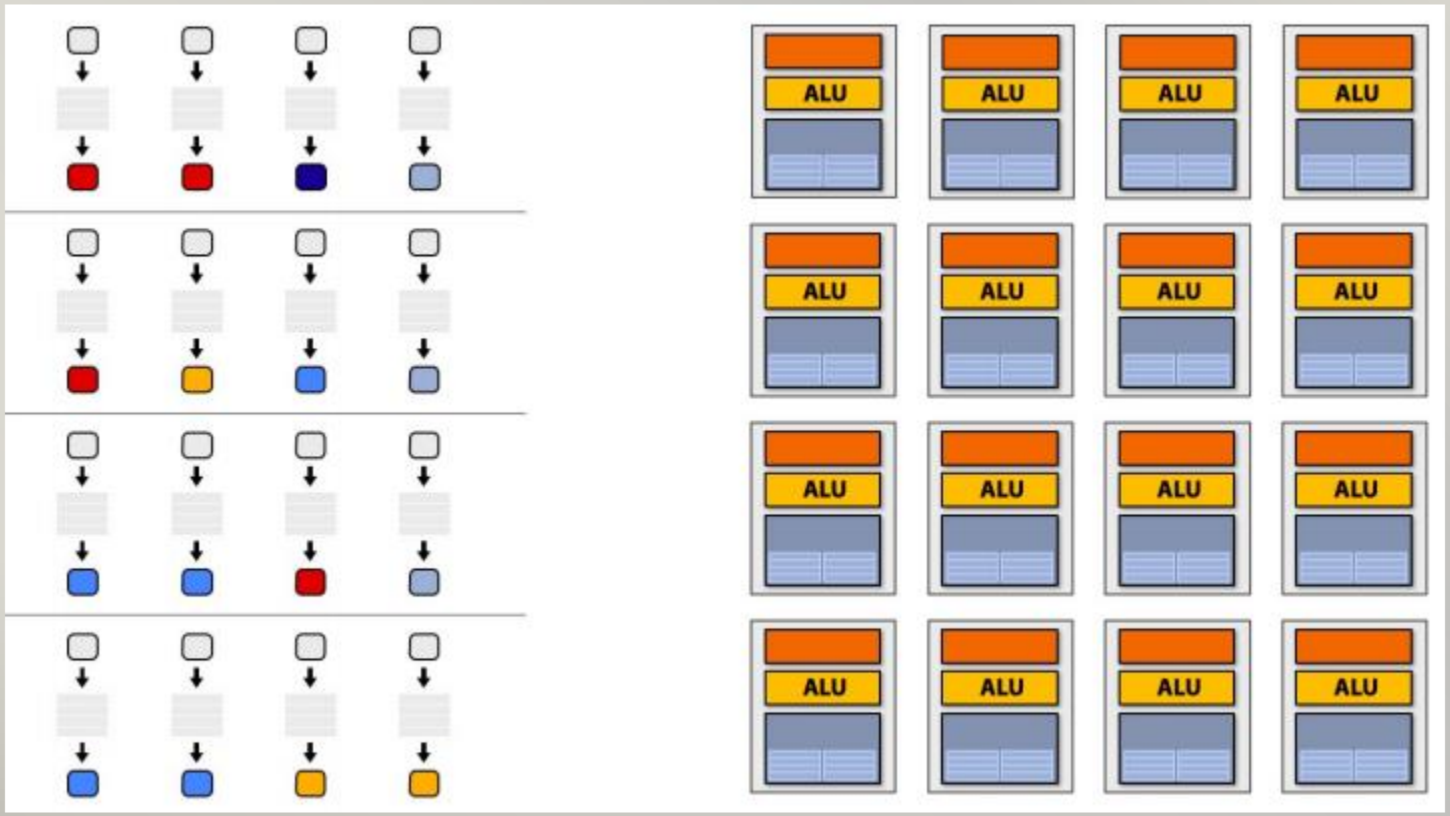
Cores in GPU

- Each GPU core runs the shader for a particular element (e.g fragment)



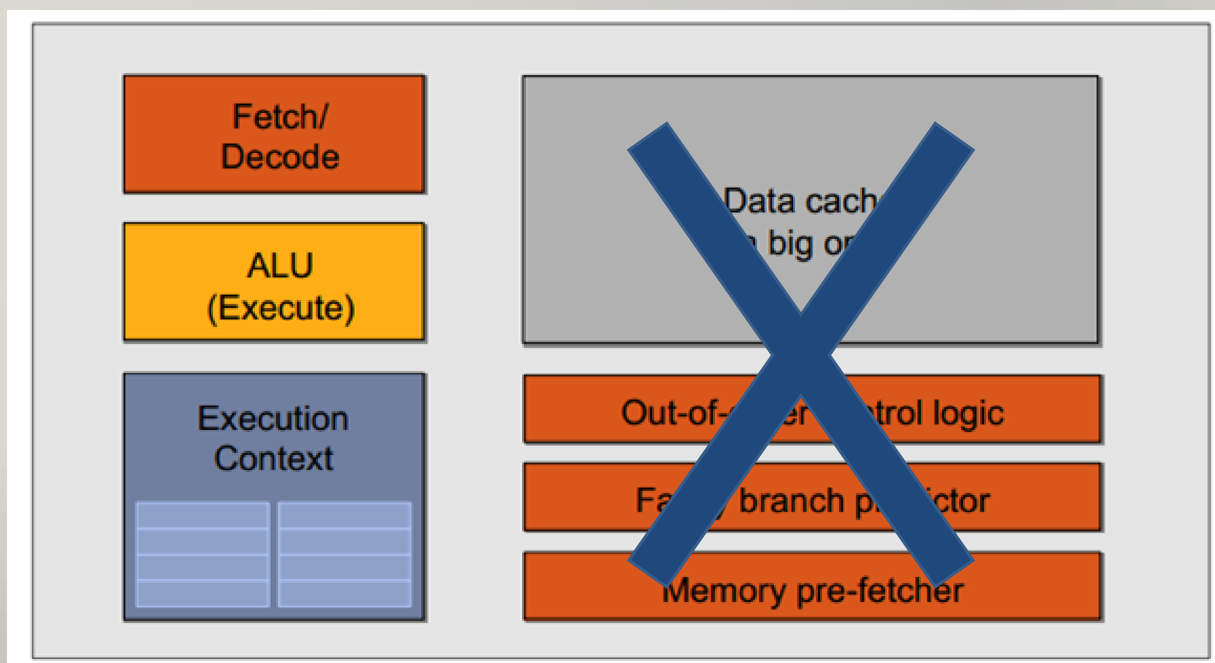
Cores in GPU

- Multiple cores running in parallels



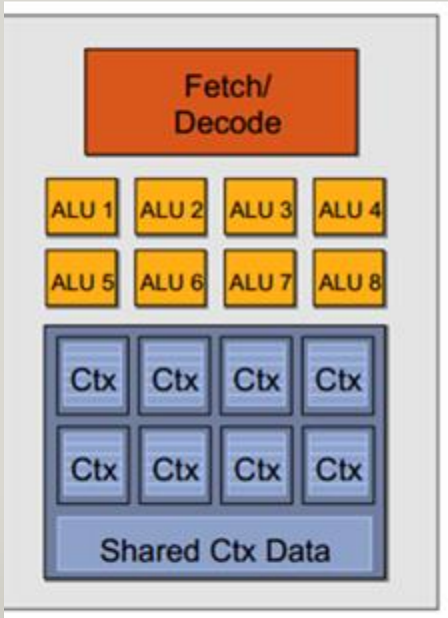
Cores in GPU

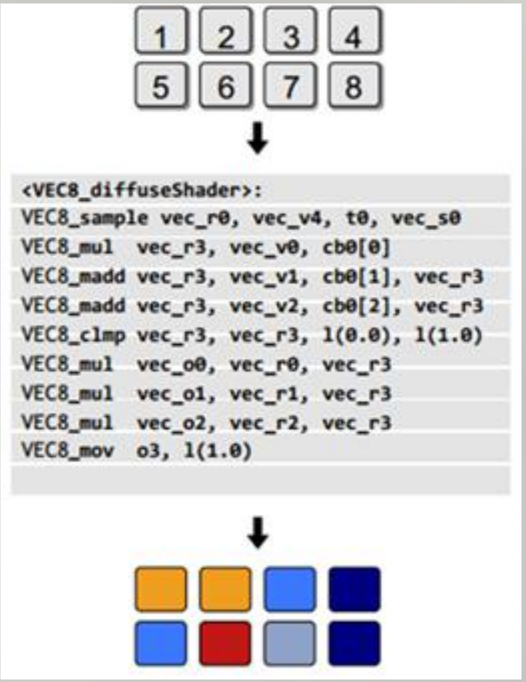
- Compare to a CPU-style core, GPU keeps a single core as simple as possible
  - Referred as “Slimmed down core”



Cores in GPU

- Tailor for SIMD processing
  - Combine fetch unit of several elements in one core

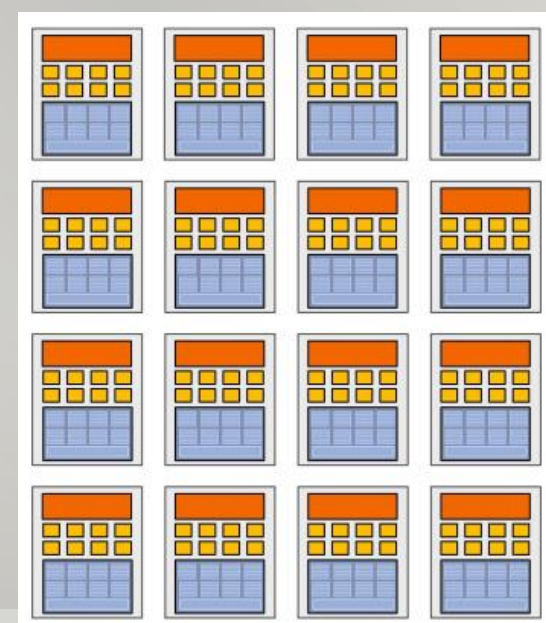




Cores in GPU

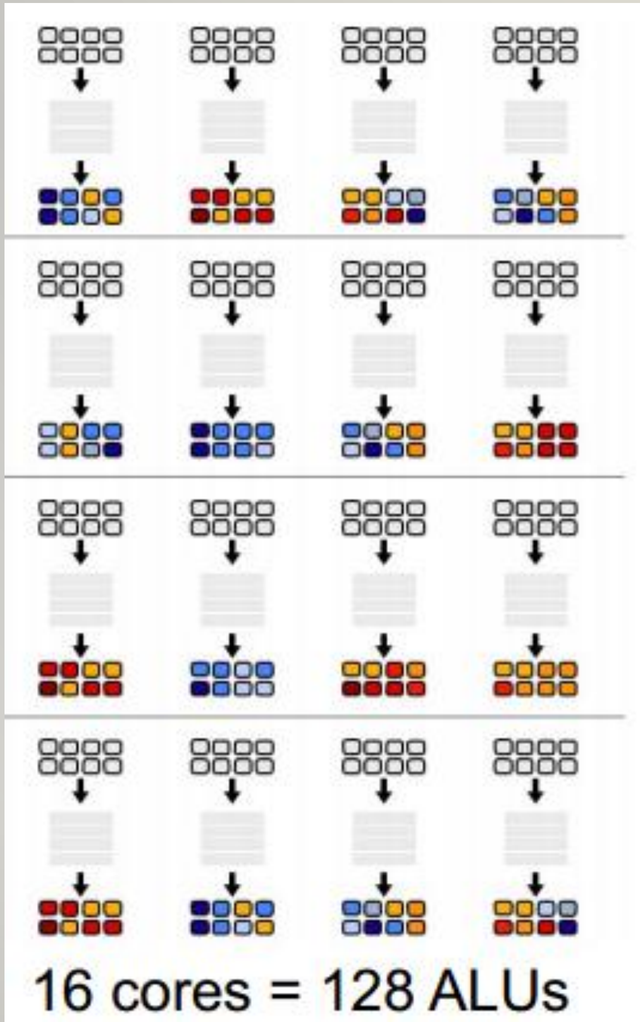
- In practice, 16 to 64 fragments share an instruction stream

- Multiple cores runs in a GPU
  - Each of them can run same or different shader program



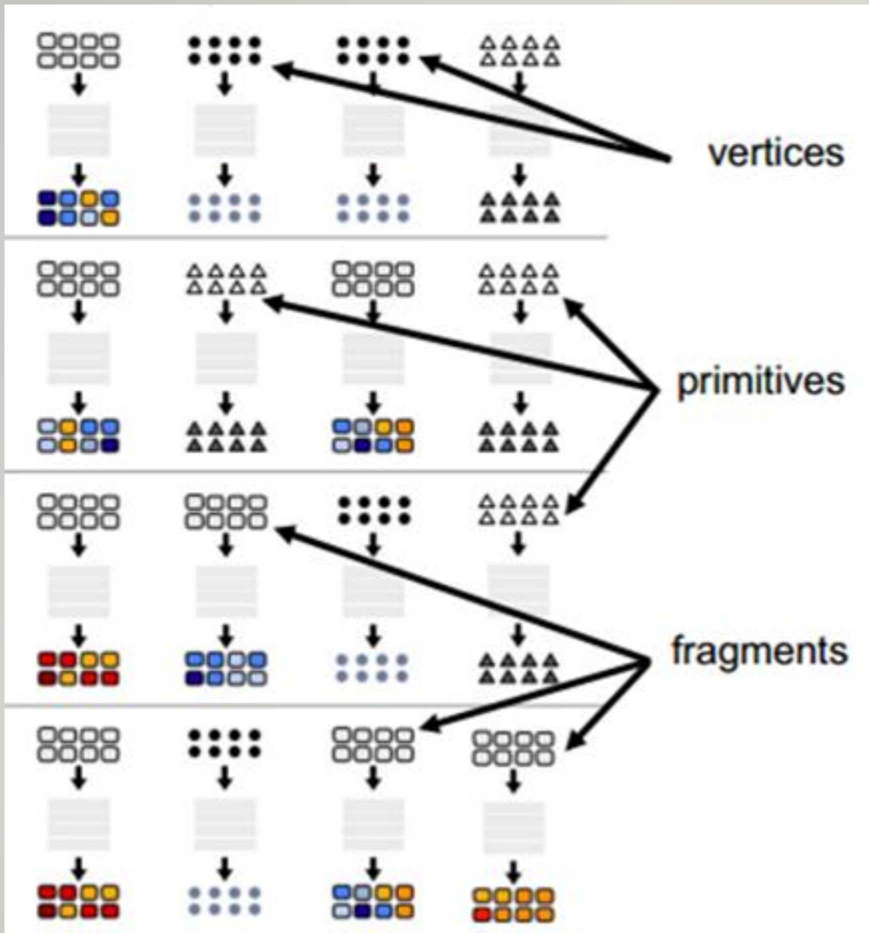
Cores in GPU

- Each core processes several elements simultaneously
  - Several ALU in one core



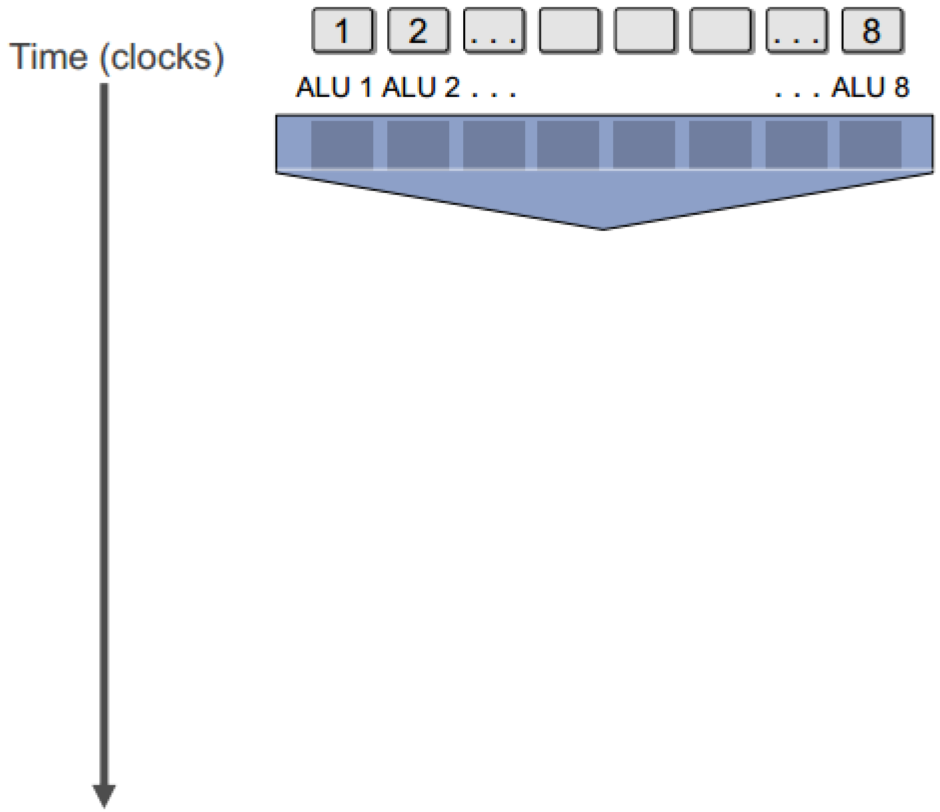
Cores in GPU

- Each core can process different elements including
  - Vertices
  - Triangles (primitives)
  - Fragments



Problem of Branching in Core

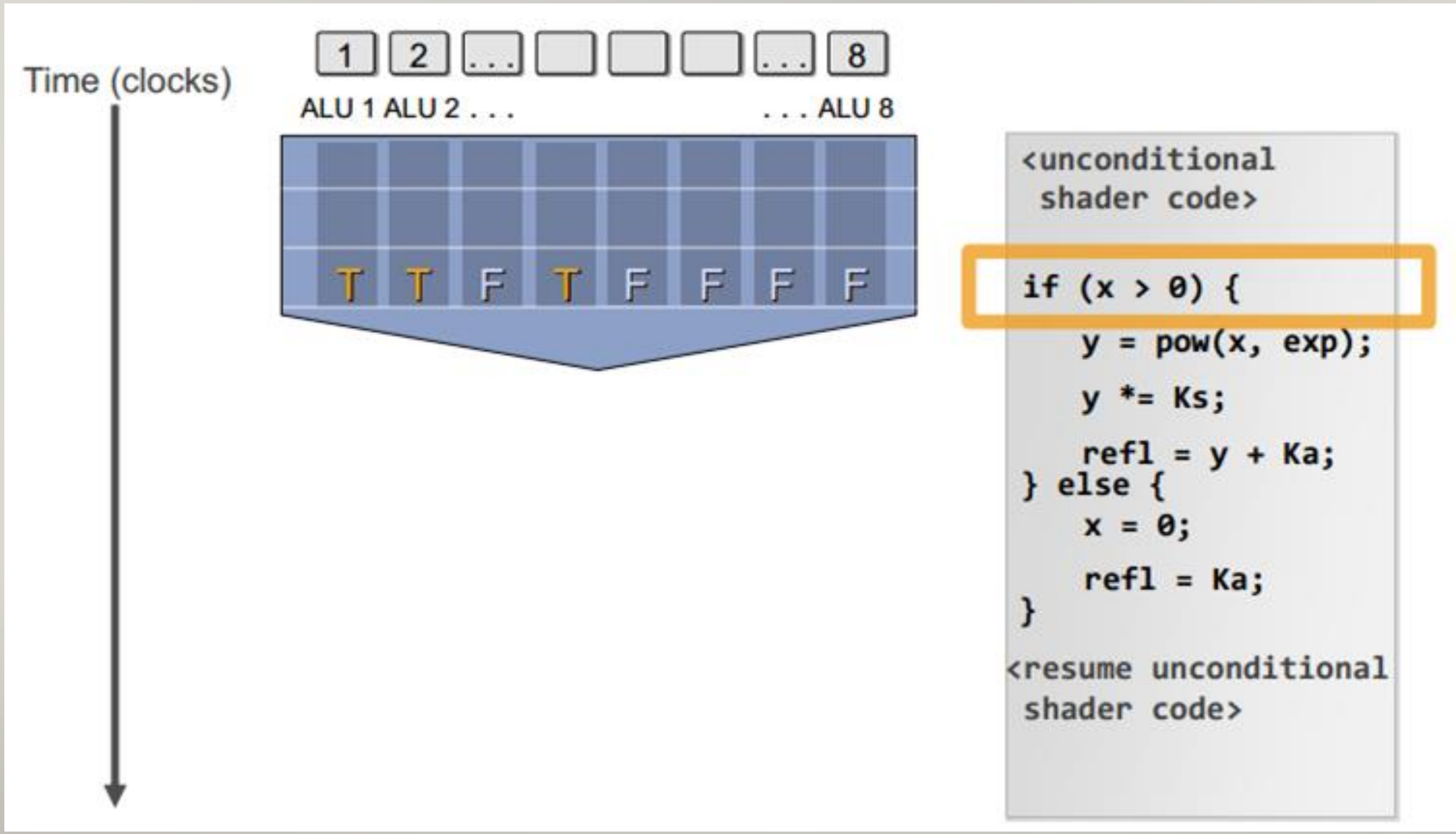
- However, not all element runs the same




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional
 shader code>
```

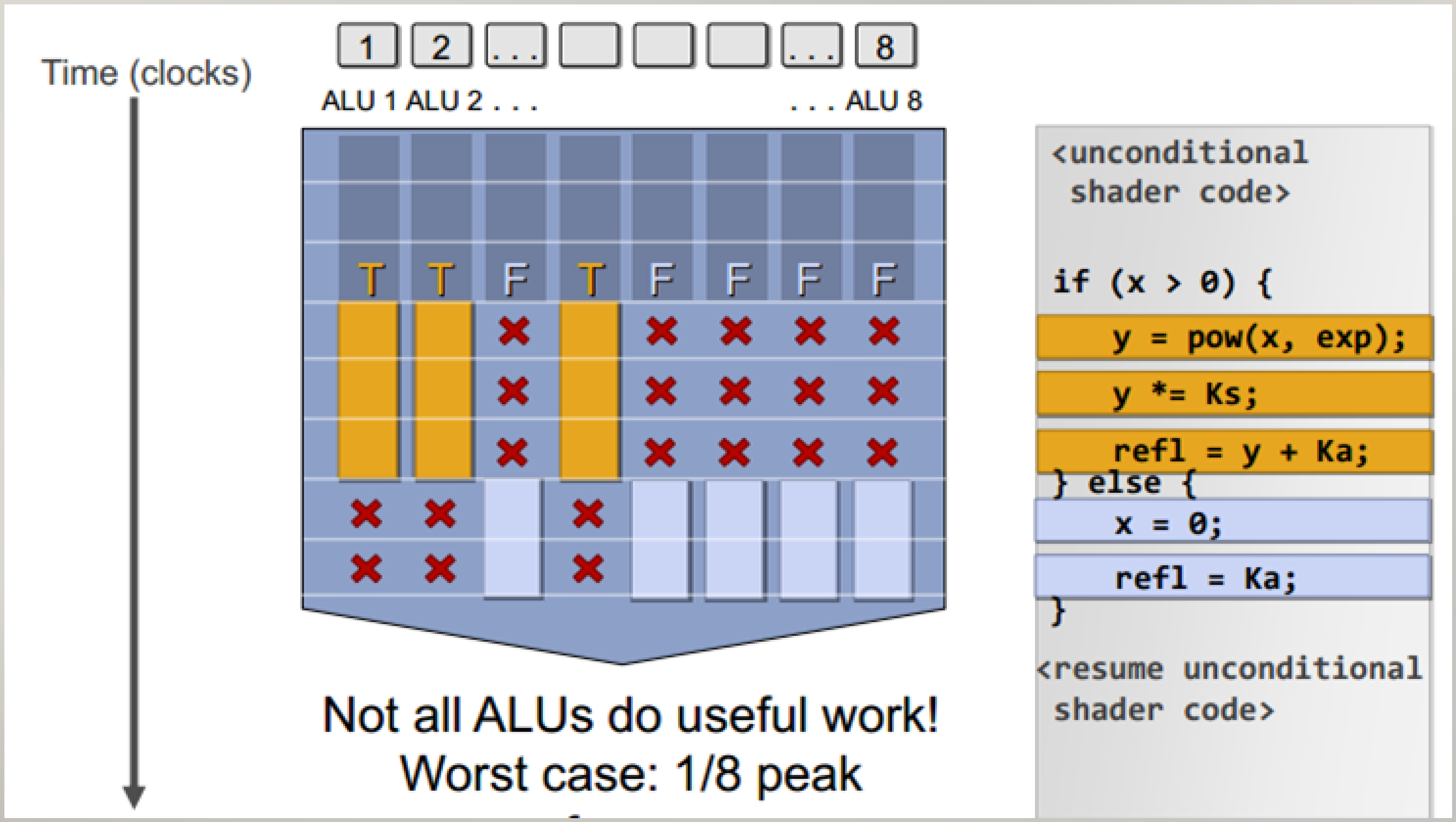
Problem of Branching in Core

- An if-else will divide elements in 2 groups



Problem of Branching in Core

- Cross represents ALU idle time



Solution of Branching in Core

- Increase throughput with enlarged context
- Avoid latency stalls by interleaving execution of many groups of fragments
  - When one group stalls, work on another group

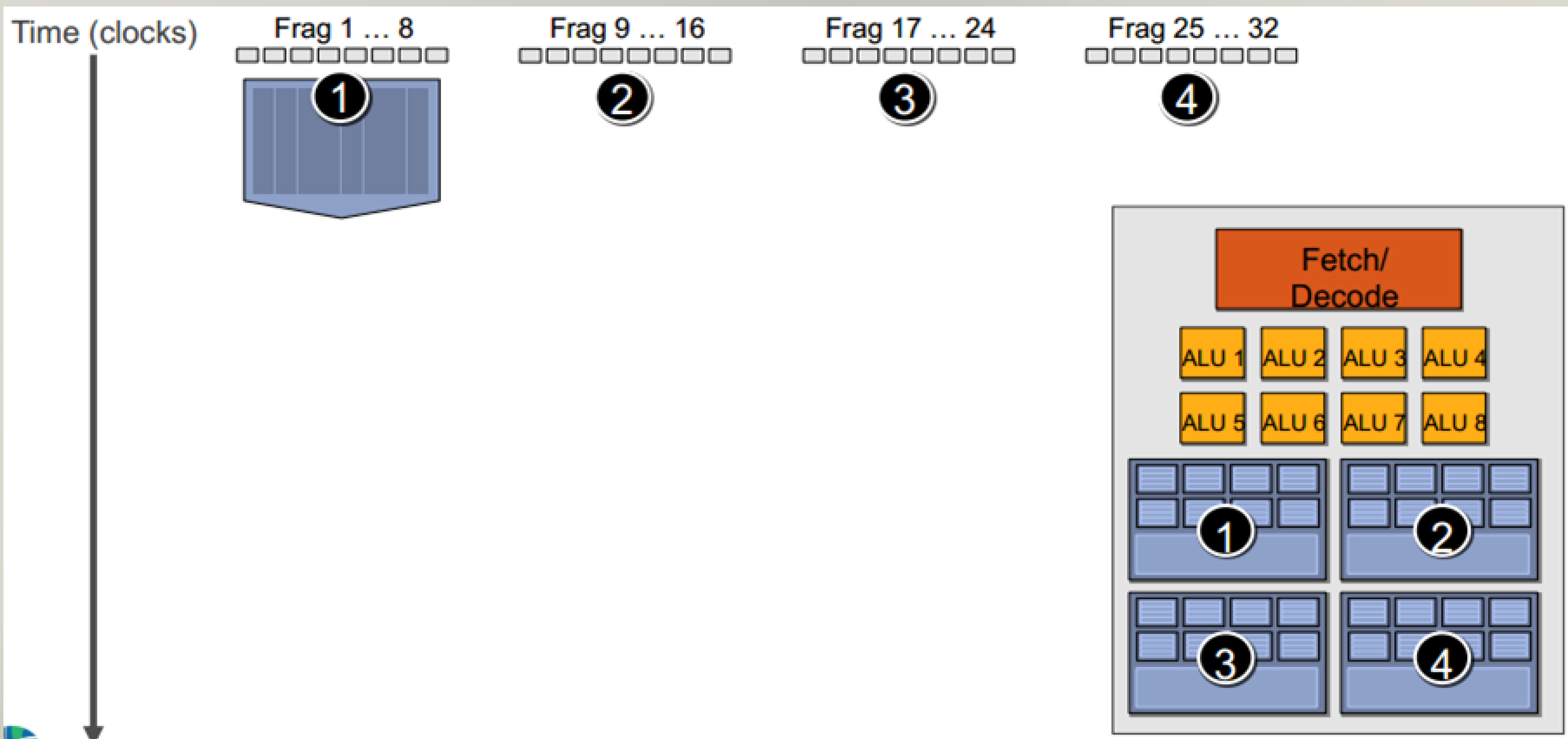
Summary: three key ideas

1. Use many “slimmed down cores” to run in parallel

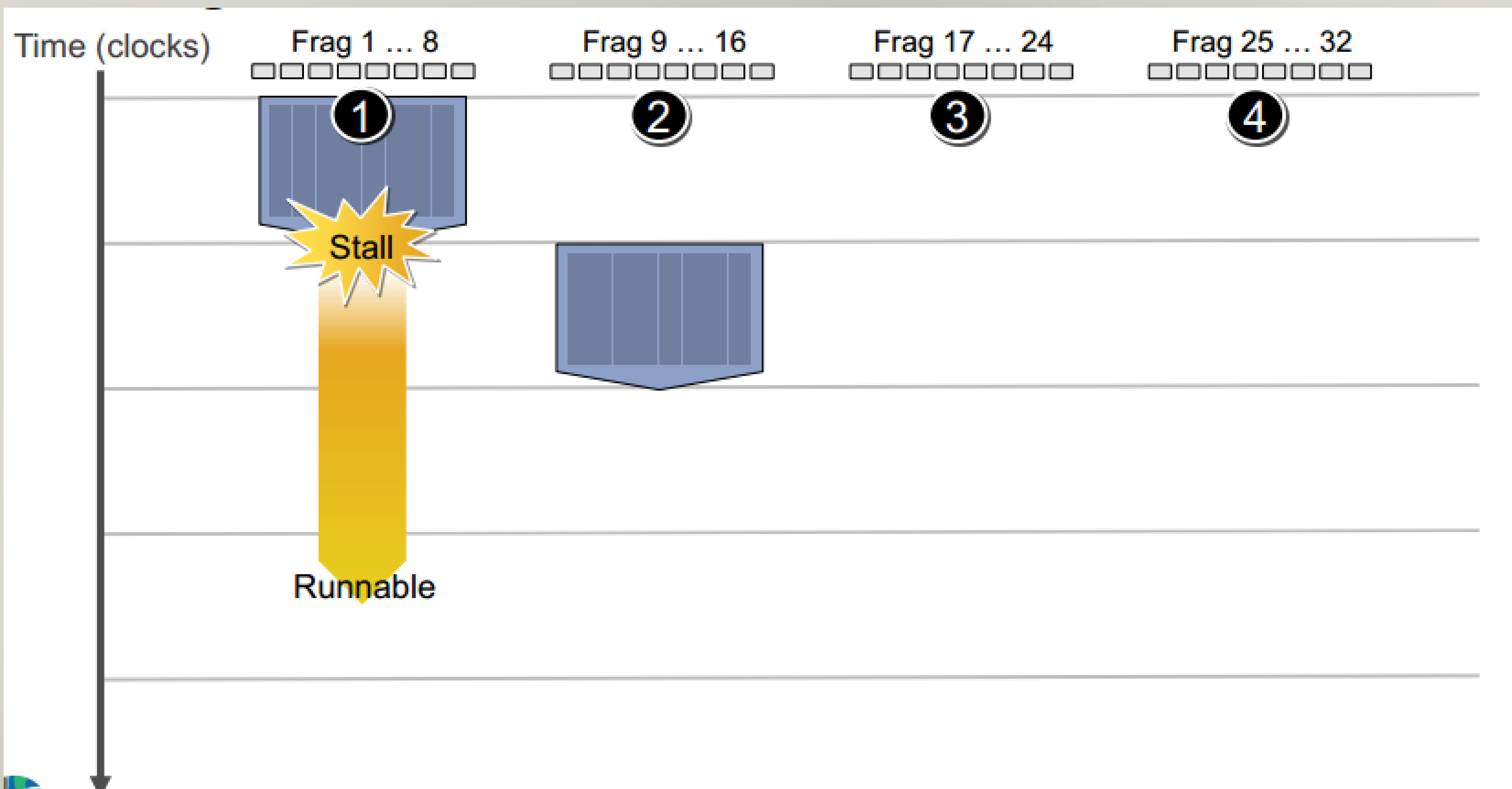
2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   – Option 1: Explicit SIMD vector instructions
   – Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   – When one group stalls, work on another group

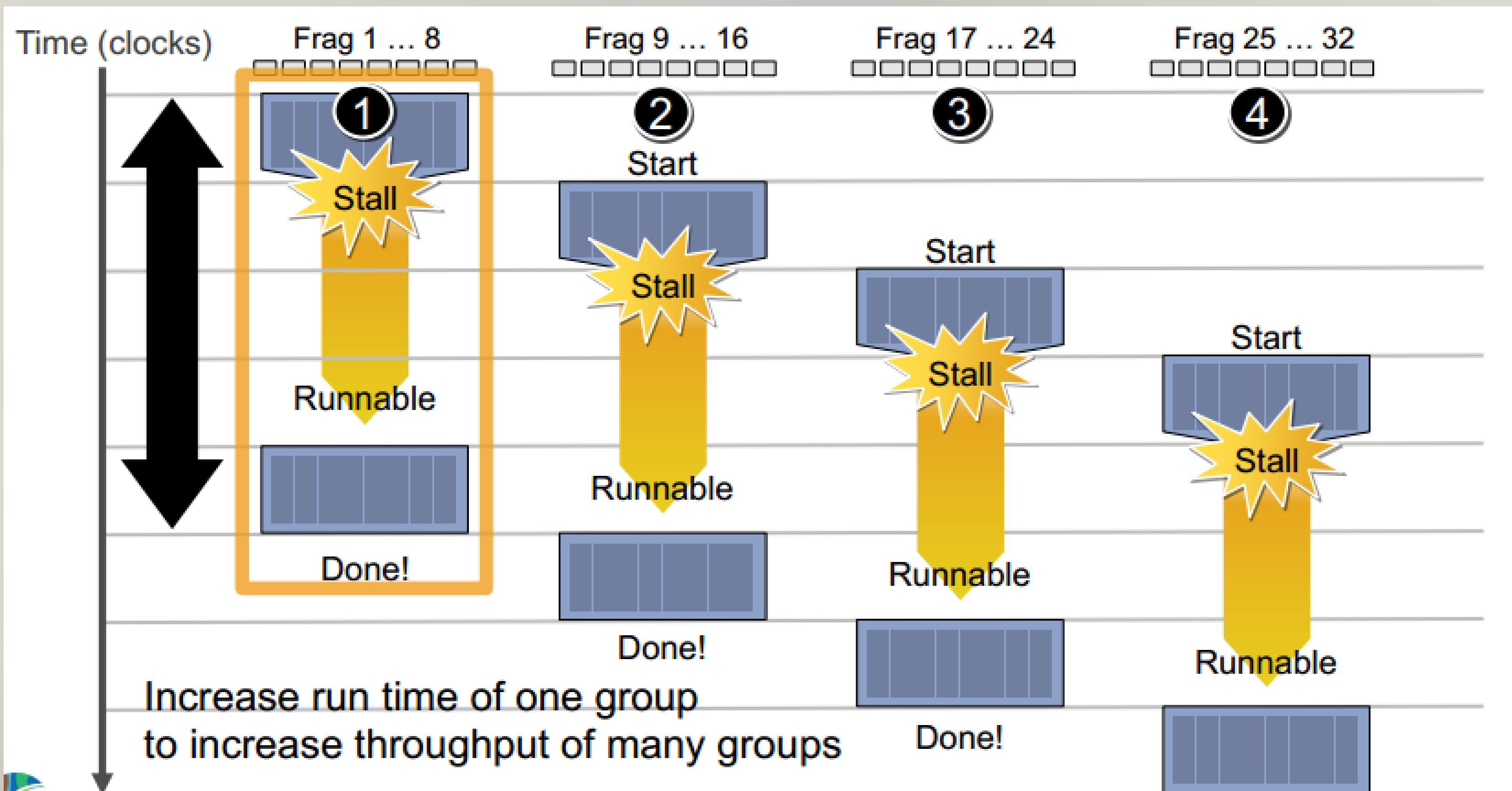
Solution of Branching in Core



Solution of Branching in Core

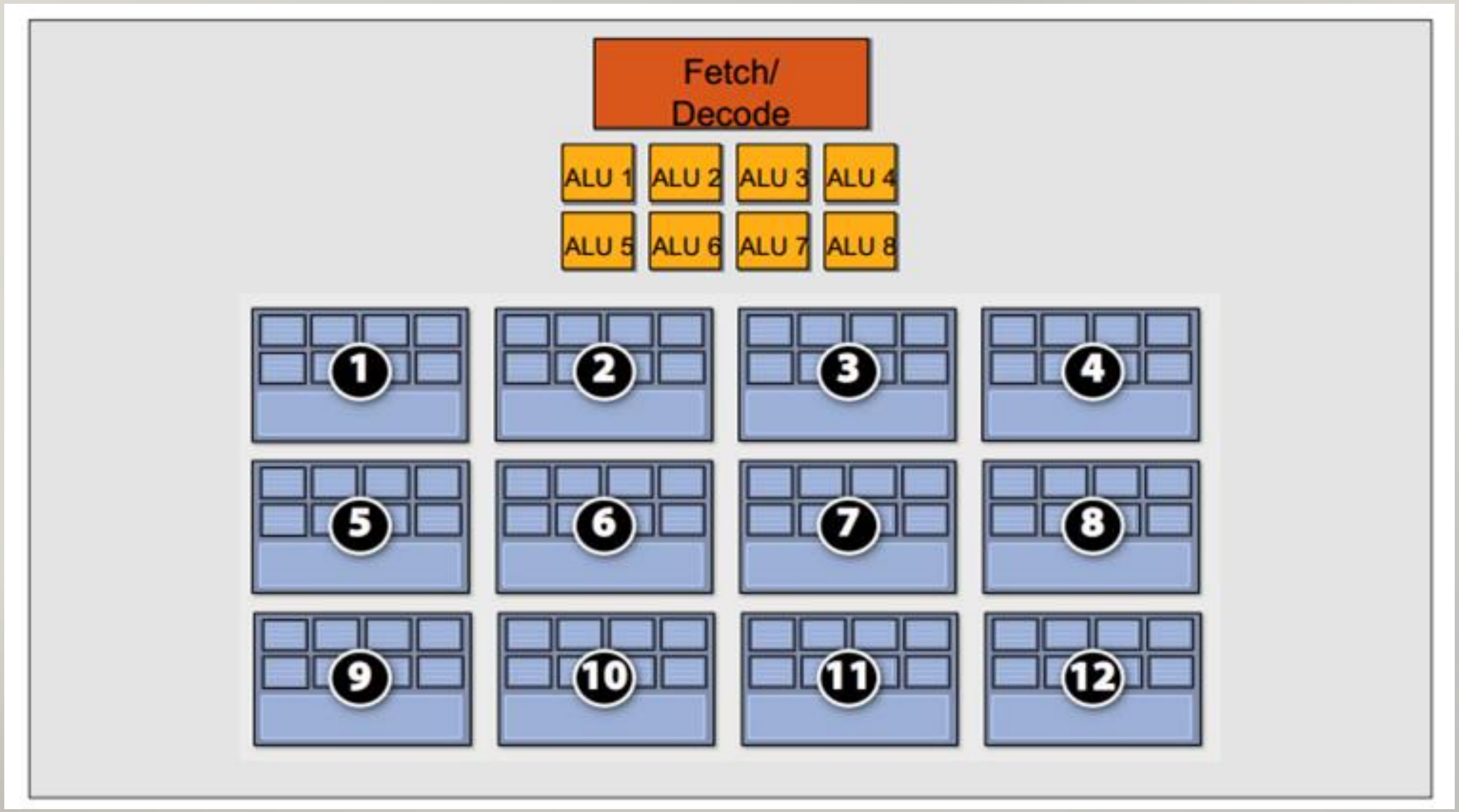


Solution of Branching in Core



Solution of Branching in Core

- Storing context



An Example Chip

16 cores

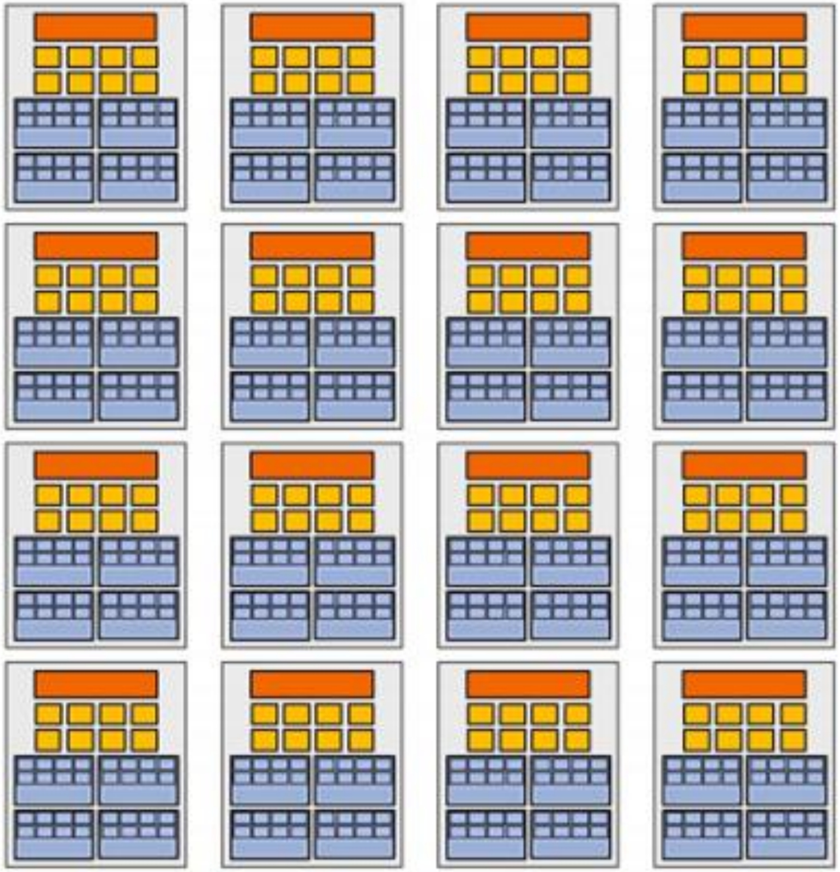
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



Shader Programming

- Setup of environment in the rendering engine (e.g. OpenGL)
  - Loading geometry, texture and other graphics elements into the engine (i.e. memory on GPU)
  - Loading shader program
  - Bind the shader program in rendering loop
- Shader program runs in parallel for all elements automatically during rendering
- The typical output is to frame buffer, but for GPGPU, we may have to dump output back to main memory

Shaders

- GLSL is used as example in the follow
- *Shader object*: an individual vertex, fragment, etc. shader
  - Are provided shader source code as a string
  - Are compiled
- Shader program: Multiple shader objects linked together
- Uniform variables: for passing in parameters to the shader program

Shader Objects

- Compile a shader object:

```
const char *source = // ...
GLint sourceLength = // ...

GLuint v = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(v, 1, &source, &sourceLength);

glCompileShader(v);

// ...
glDeleteShader(v);
```

Create a Shader Object and return its handle

Provide the shader's source code as a string

Compile the shader

Shader Programs

- Link a shader program:

```
GLuint v = glCreateShader(GL_VERTEX_SHADER);
GLuint f = glCreateShader(GL_FRAGMENT_SHADER);
// ...

GLuint p = glCreateProgram();
glAttachShader(p, v);
glAttachShader(p, f);

glLinkProgram(p);
// ...
glDeleteProgram(v);
```

A program needs a vertex and fragment shader

Link the shaders to form a shader program

Notice that the program had not yet applied to any of your rendered objects !!!

Using Shader Programs

Apply the Shader Program before any draw commands of OpenGL.

The same program can be applied on different drawing targets

```
GLuint p = glCreateProgram();
// ...

glUseProgram(p);

glDraw*(); // * because there are lots of draw functions
// ...
```

Uniforms

Each *active* uniform variable has an integer index location (or handle).

```
GLuint p = glCreateProgram();
// ...
glLinkProgram(p);
```

```
GLuint m = glGetUniformLocation(p, "u_modelViewMatrix");
GLuint l = glGetUniformLocation(p, "u_lightMap");
```

```
// ...

glUseProgram(p);
mat4 matrix = // ...
```

`glUniform*` for all sorts of datatypes

```
glUniformMatrix4fv(m, 1, GL_FALSE, &matrix[0][0]);
glUniform1i(l, 0);
```

Uniforms can be changed as often as needed before applying to drawing commands, but are constant during a draw call !!

Shader Programming

A Simple Example

- Input: a teapot model
- Vertex Shader: Manipulate the vertices of the model so as to make it flatten
- Fragment Shader: a pass through or assigned certain color, e.g.

```
void main() {
  gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

The Flatten Shader

http://www.lighthouse3d.com/

- The Vertex Shader makes all input vertices with z = 0 (keeps x and y)
- Actually making a cross-section at z=0

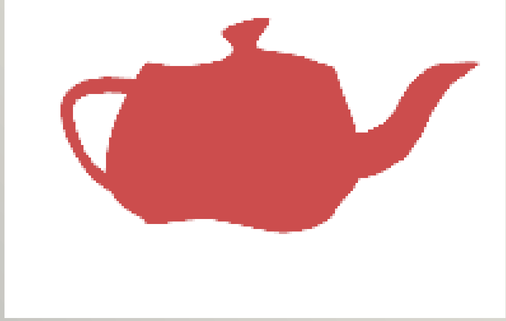
```
void main(void) {

vec4 v = vec4(gl_Vertex);
v.z = 0.0;

gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Shader predefined values are started with “gl_”

Turn every coordinate with z =0

The Flatten Shader

- Another alternative is to assign z value with a sine function of coordinate x
- It will produce a wavy flat teapot



```
void main(void) {

   vec4 v = vec4(gl_Vertex);
   v.z = sin(5.0*v.x )*0.25;

   gl_Position = gl_ModelViewProjectionMatrix * v;

}
```

Simple Toon Shading

- Input: a teapot model
- Vertex Shader: Multiply vertex with modelview and projection matrices, and pass normal to fragment shader
- Fragment Shader: Apply a stepping function to map the renderred color

Simple Toon Shading

- The Vertex Shader
  - Passing normal vector to fragment shader
  - Values will be interpolated on each fragment

```
varying vec3 normal;
void main() {

  vec4 v = vec4(gl_Vertex);
  gl_Position = gl_ModelViewProjectionMatrix * v;

  normal = gl_Normal;
}
```

Simple Toon Shading

- Based on per-fragment normal to compute shading
- Then, use shading intensity to map final color

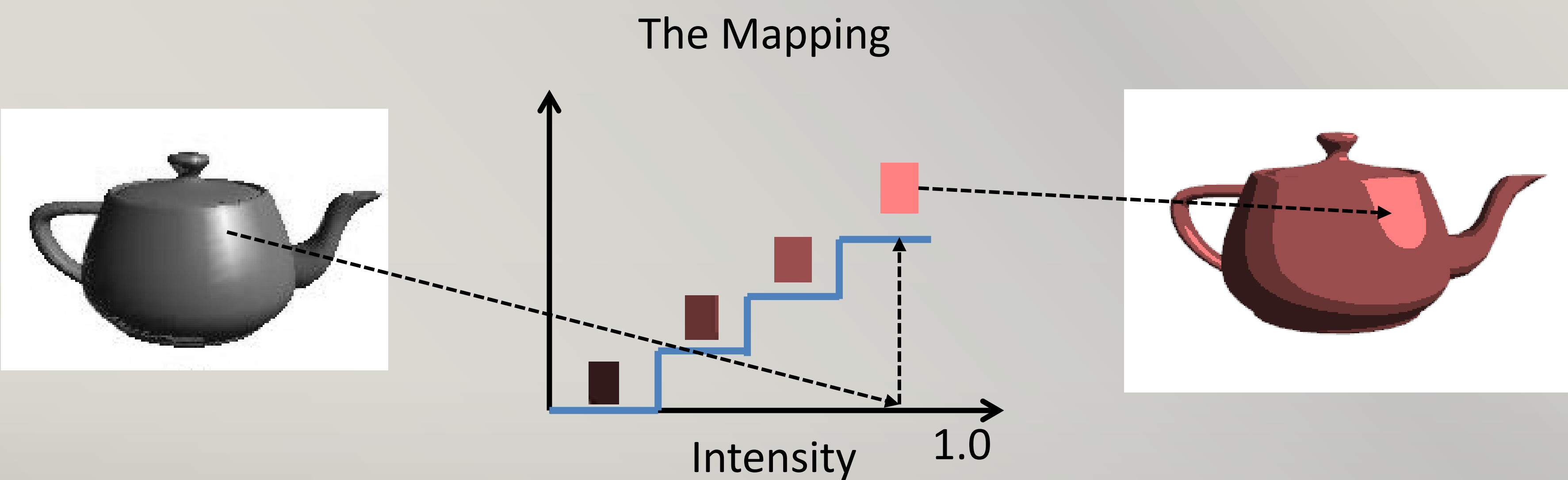
```
uniform vec3 lightDir;
varying vec3 normal;
void main() {
  float intensity;
  vec4 color;

  intensity = dot(lightDir,normalize(normal));
  if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
  else
  if (intensity > 0.5)  color = vec4(0.6,0.3,0.3,1.0);
  else
  if (intensity > 0.25) color = vec4(0.4,0.2,0.2,1.0);
  else
    color = vec4(0.2,0.1,0.1,1.0);
  gl_FragColor = color; }
```

Compute shading using Phong model

Smoothly changing intensity is mapped to 4 different colors

Simple Toon Shading

- The mapping is like a **quantization** to the shading intensity

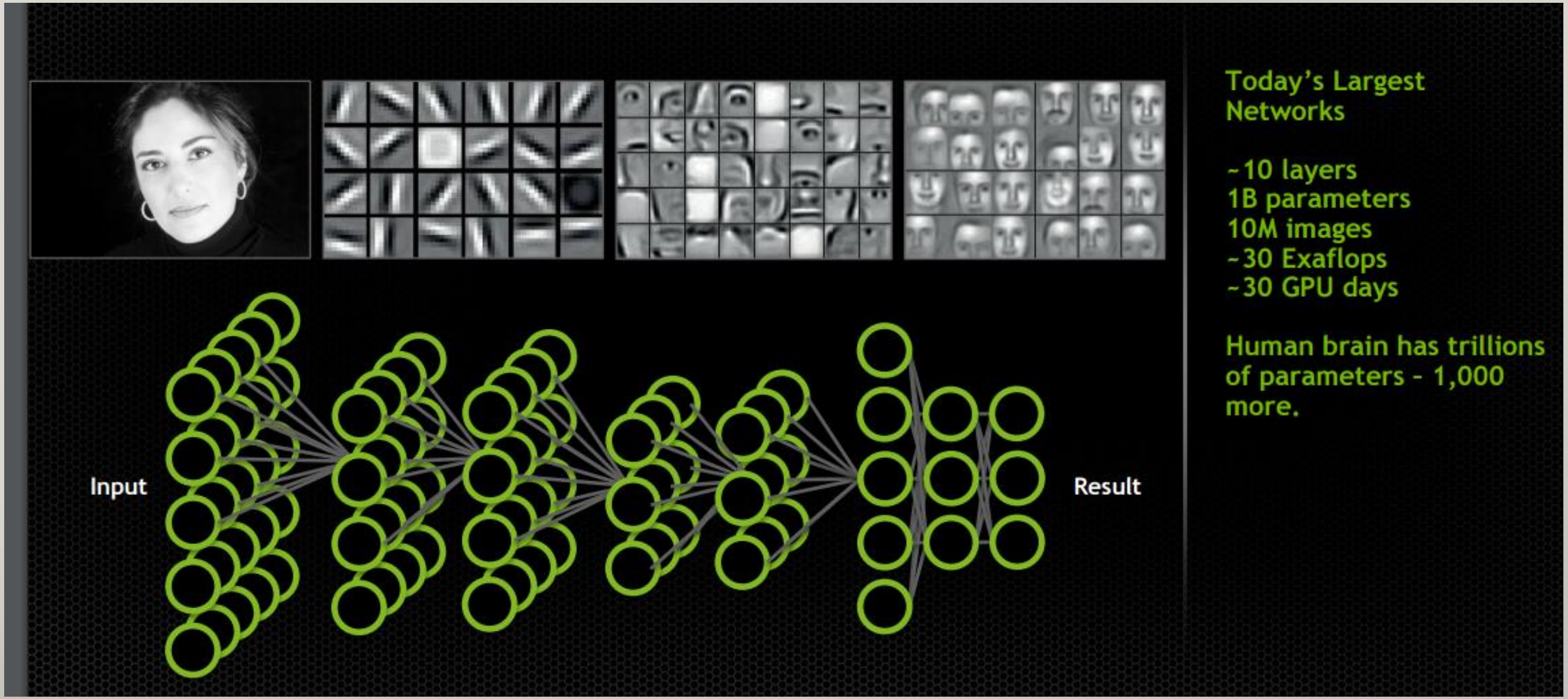


GPGPU

- General Purpose GPU
  - Not only for graphics related applications
  - Physics based simulation can also be regarded as a kind of GPGPU
- Recent examples: Bitcoin Miner using GPU

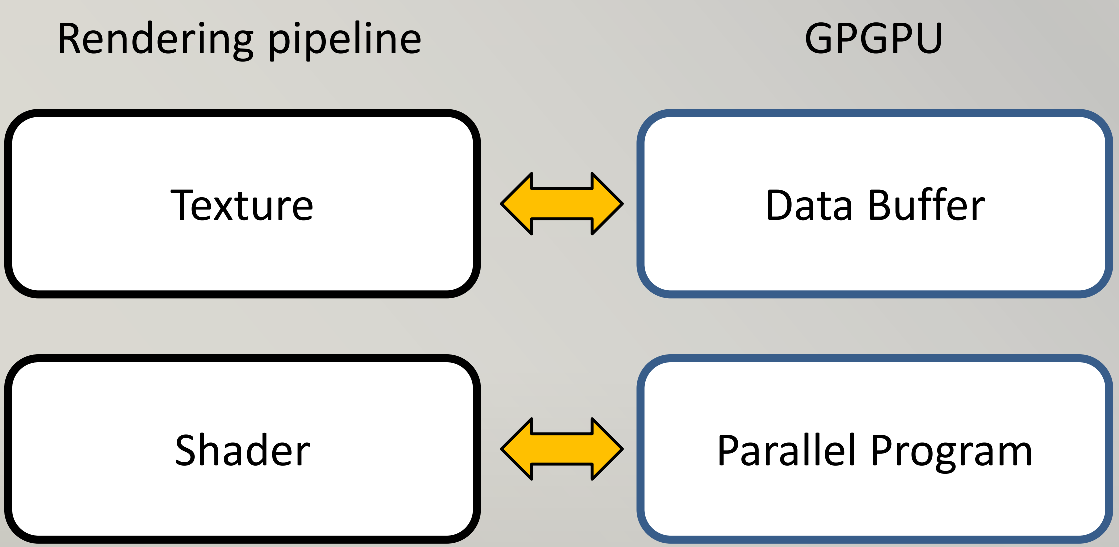
GPGPU

- Recent examples: Deep learning



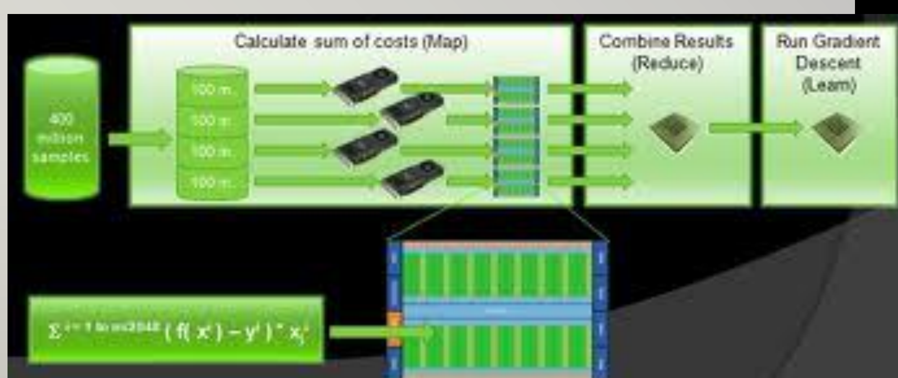
GPGPU

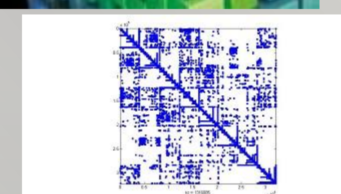
- We use GPU as a parallel processor
- A mapping between rendering pipeline and general purpose computation
  - Use Texture as Data Buffer, Shader as Parallel Program



Application of GPGPU

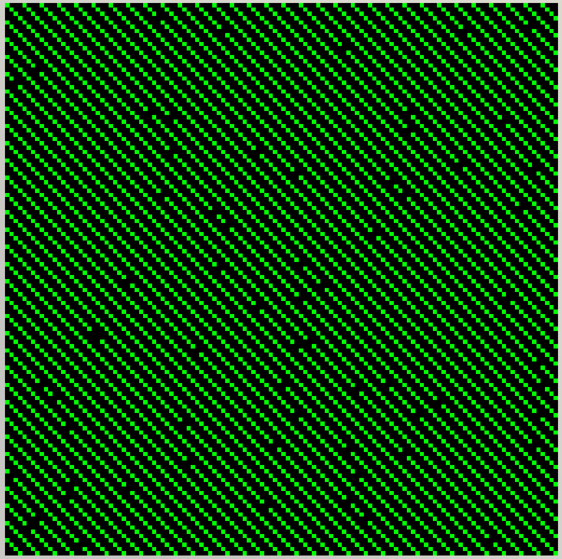
- Accelerated Linear Algebra Computation
  - E.g. CUBLAS, cuSPARSE
- Massive Random Number Generation
- Machine Learning Algorithms
- Sorting



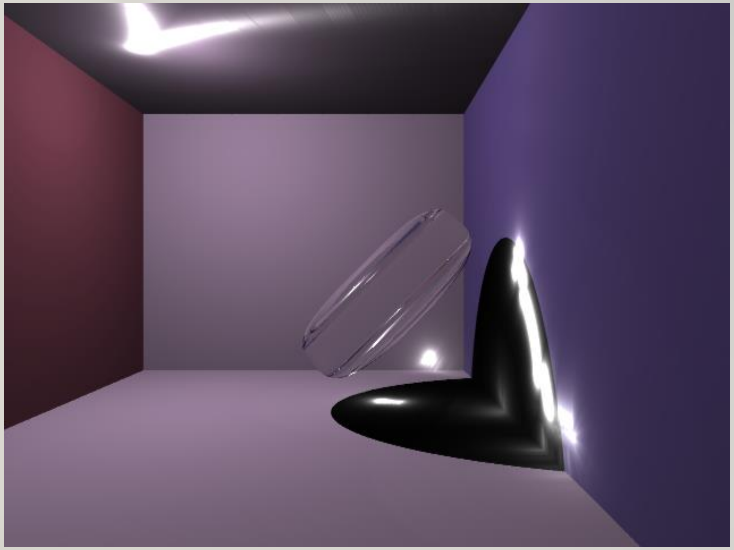


Pseudo-random number generator (PRNG)

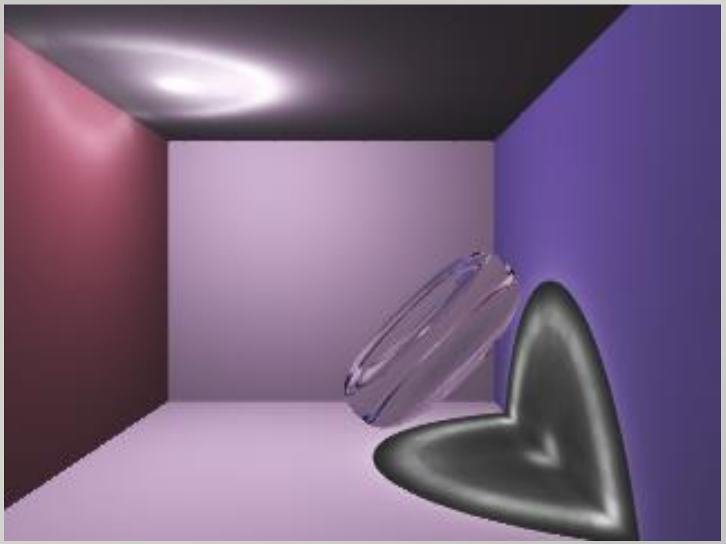
- Provide uniform random numbers
- Example : rand() in C
- Stochastic rendering algorithms
  - Photon-mapping
  - Distribution ray-tracing
- Poor randomness -> slow convergence



Pseudo-random number generator (PRNG)



Results from an LCG
10,000 photons



Control Image
10,000 photons

Some common PRNG

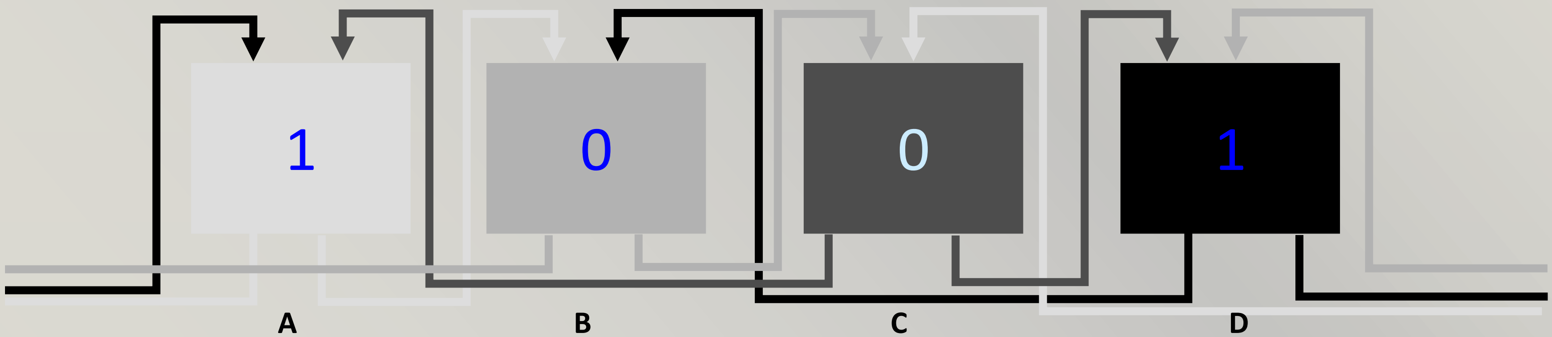
- linear congruential generator (LCG)
  - $R_{n+1} = aR_n + b \pmod{m}$
- lagged Fibonacci generator
  - $R_n = R_{n-j} \# R_{n+k} \pmod{m}$ (where # is a binary operator)
- High precision integer arithmetics
- Can't fit in the current GPU !
- GPU implementation is not available

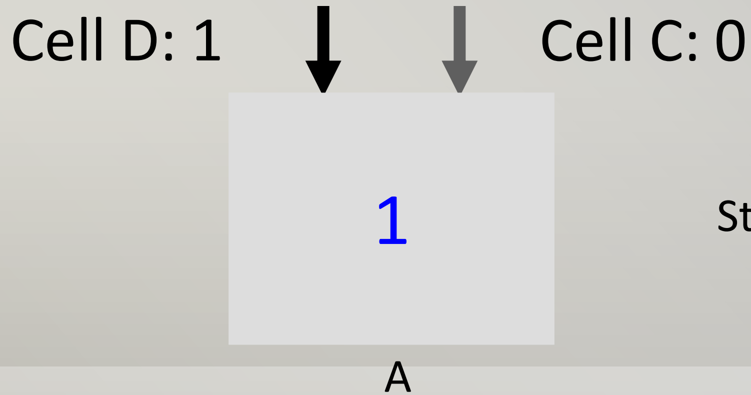
PRNG on GPU

- CA-based PRNG
- No high precision integer arithmetics
- Adoptable PRNG for different GPU
- PRNG Performance on different GPU
  - 7800 GTX and FX5900
- Why not find optimal PRNG automatically ?

GPU Accelerated PRNG

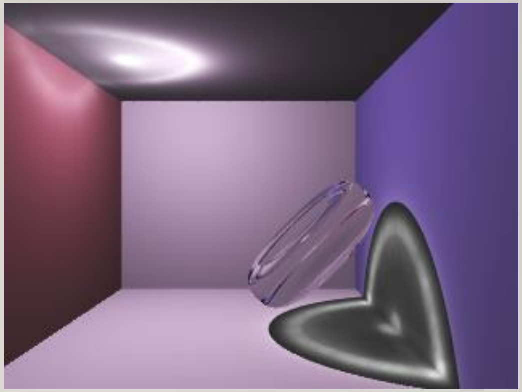
- GPU : NO High precision integer arithmetic
- Most PRNG cannot fit in all GPU



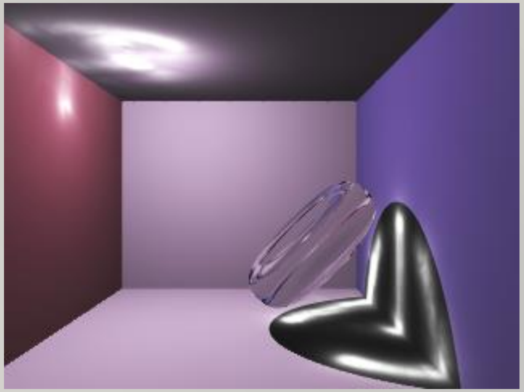


Step(1, 3- 1 – 2*0)

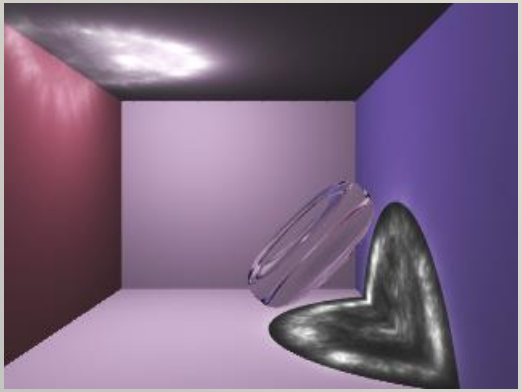
Convergence of Optimization



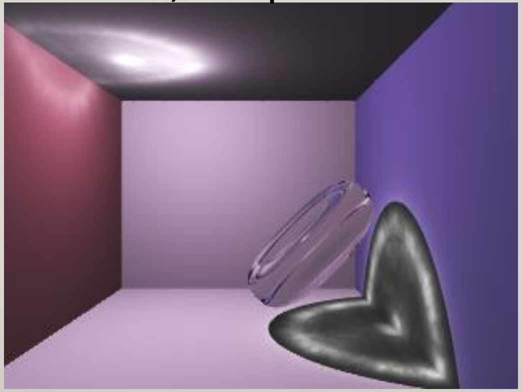
Control
10,000 photons



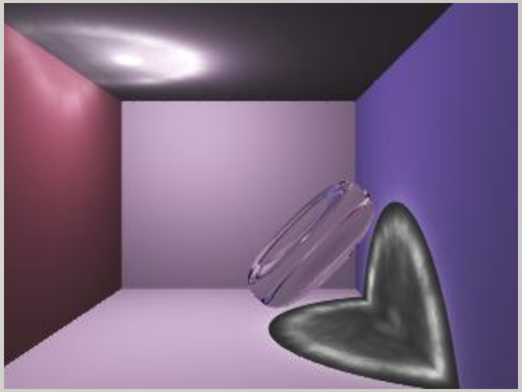
Generation 1



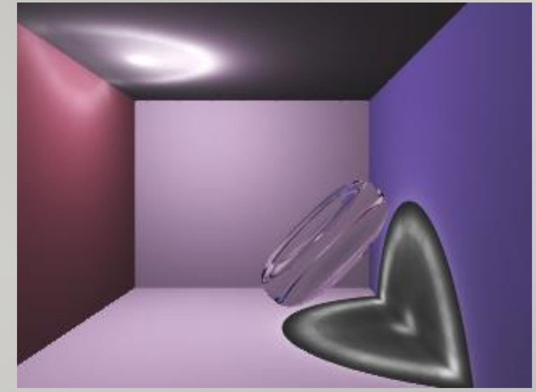
Generation 2



Generation 4



Generation 8



Generation 11

Performance

- Performance compare with CPU
- 1,000 Parallel PRNG
- 13 times faster

Random numbers generated	GPU CA-PRNG	Software CA-PRNG
10,000	0.004s	0.043s
100,000	0.031s	0.425s
1,000,000	0.31s	4.274s
10,000,000	3.098s	43.003s
100,000,000	31.875s	430s

"Implementing High-Quality PRNG on GPU,"
W. M. Pang, T. T. Wong and P. A. Heng,
Shader X5: Advanced Rendering Techniques, *Edited by W. Engel, Charles River Media, 2007, pp. 579-590.*

Quicksort algorithm

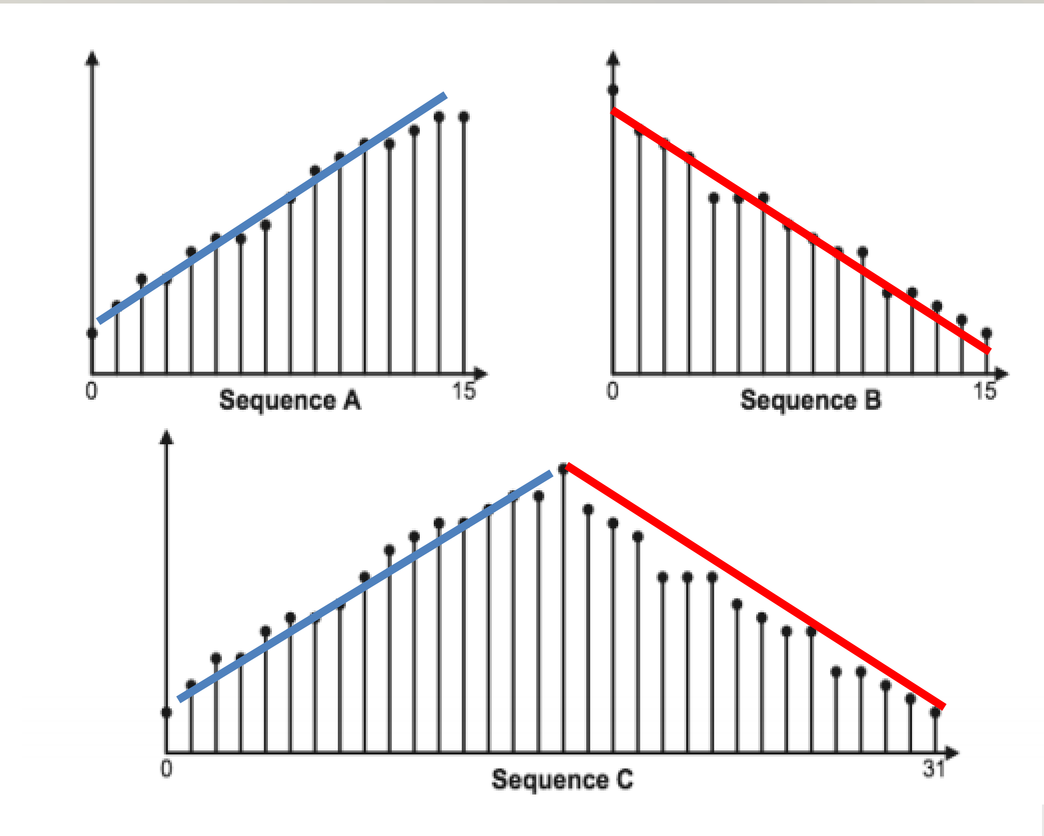
- Divide and conquer approach (recursive)
  - Recursively divide the sequence into two based on a selected pivot value
- For n items, it has O(n log(n)) complexity on average and $O(n^2)$ in the worst case.
- But it is not very well fit for parallel processing
  - The two sequence are not balanced
  - Operation depends on the previous operations

Bitonic Sort

- Bitonic sort is a sort algorithm that works only with bitonic sequences of values
- A bitonic sequence is
  - A sequence a0, a1, ..., an is bitonic if and only if
  - There is an i, such that a0, ... ai is monotonically increasing and
  - ai, ... an is monotonically decreasing
- There is a cyclic shift of the sequence which makes the previous condition true

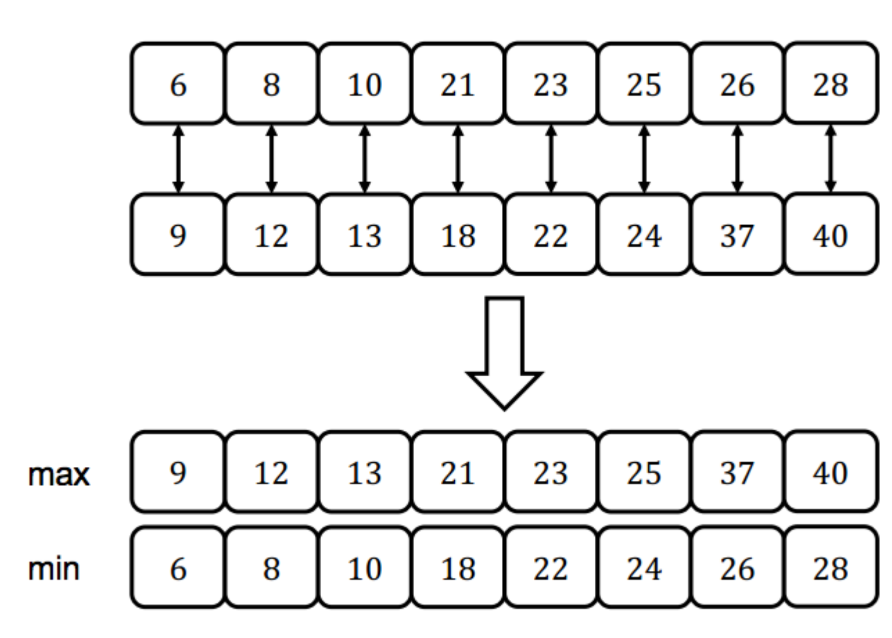
Bitonic Sort

- Bitonic sequences



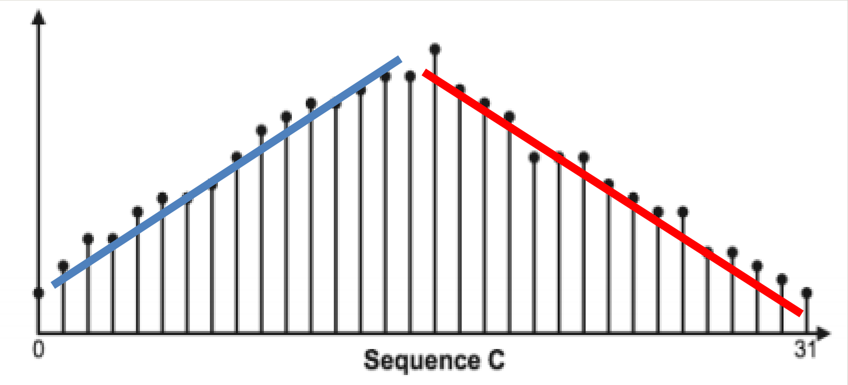
Bitonic Split

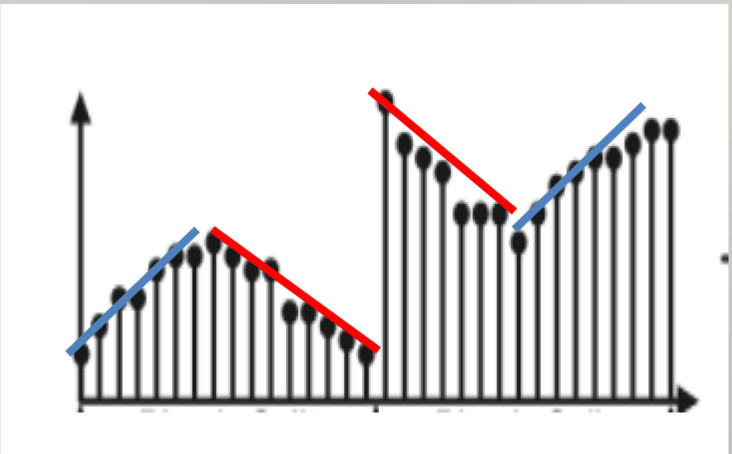
- This operation contains 2 Steps
  - Compare each element in the lower half of the sequence (0 - i) with the corresponding element in the upper half (i+ N/2)
  - If the element in the lower half is greater than the element in the upper half, swap the two elements



Bitonic Split

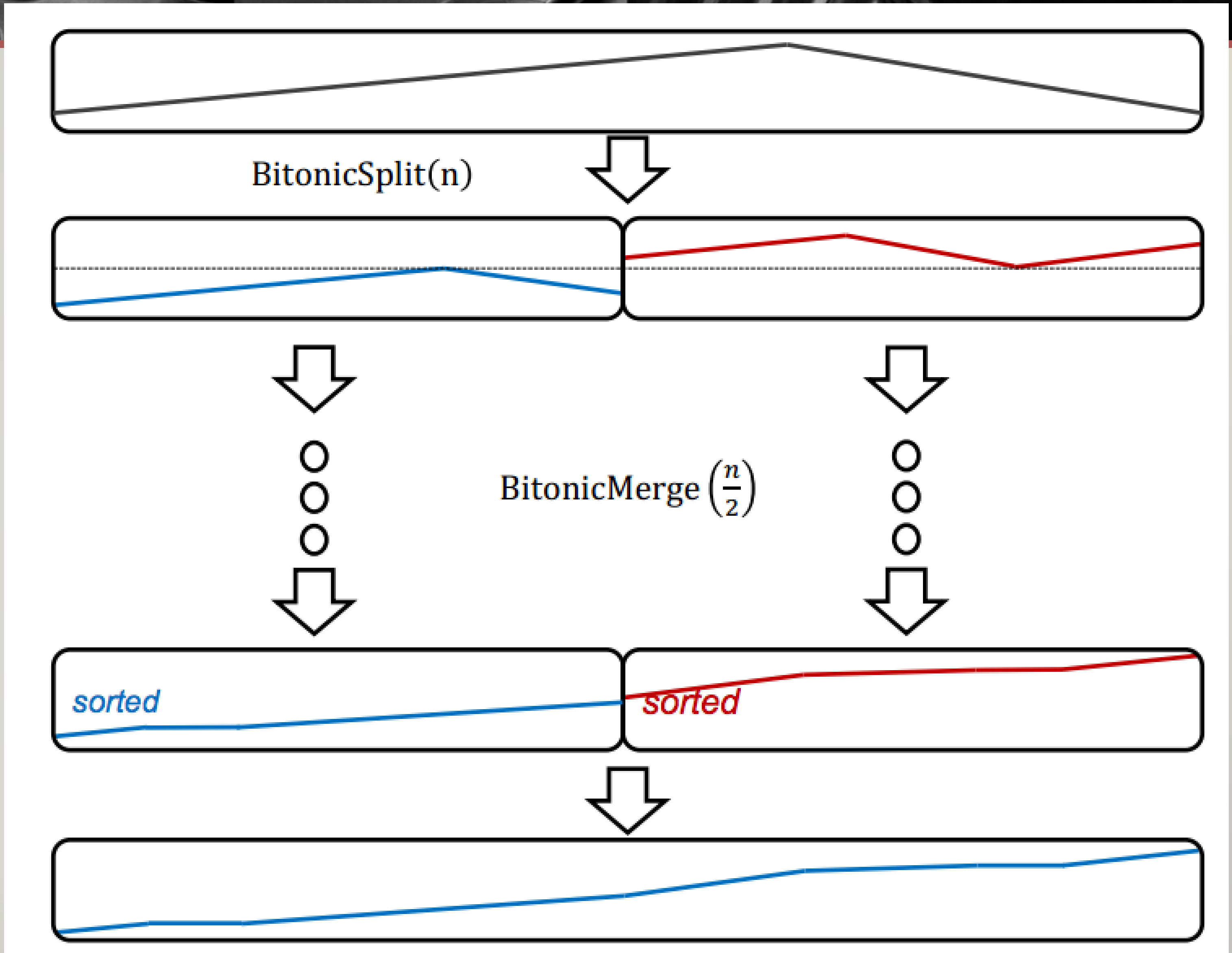
- If we apply the bitonic split to the sequence C , we obtain a sequence which is not bitonic
- But the two sequences are bitonic themselves





Bitonic Merge

- The bitonic merge consists in applying the bitonic split iteratively
- If a sequence contains N elements N=2k, the bitonic merge requires k steps
  - Each step is a bitonic split
  - Step i, splits N/i elements



The whole sequence will be sorted after the bitonic merge operation !!

Bitonic Merge

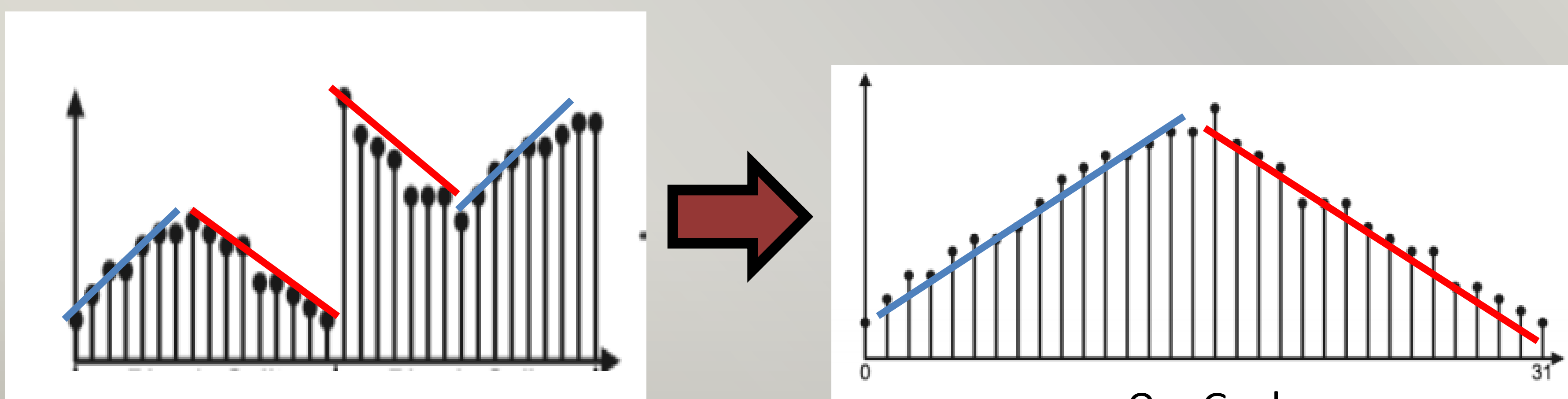
- If the sequence is bigger than N=2k but smaller than N=2k+1
  - We sort N=2k+1 elements, filling the extra values with the maximum possible value
- It is obvious that the algorithm can be parallelized easily
  - Each comparison in a pair of numbers within bitonic split is independent to each other

Bitonic Sort

- But what if our sequence is not bitonic at the beginning? It is very likely to be…
- How about we first transform the sequence into a bitonic one?
- Split the sequence into two half each time
- Apply recursively on each half to construct a bitonic sequence

Bitonic Sort

- Ultimately, we will split into a sequence of only two elements
  - Note that all two-elements sequence is already bitonic
- So, in each step, our objective is to
  - Combine two sequences with the first half being increasing and the upper half decreasing



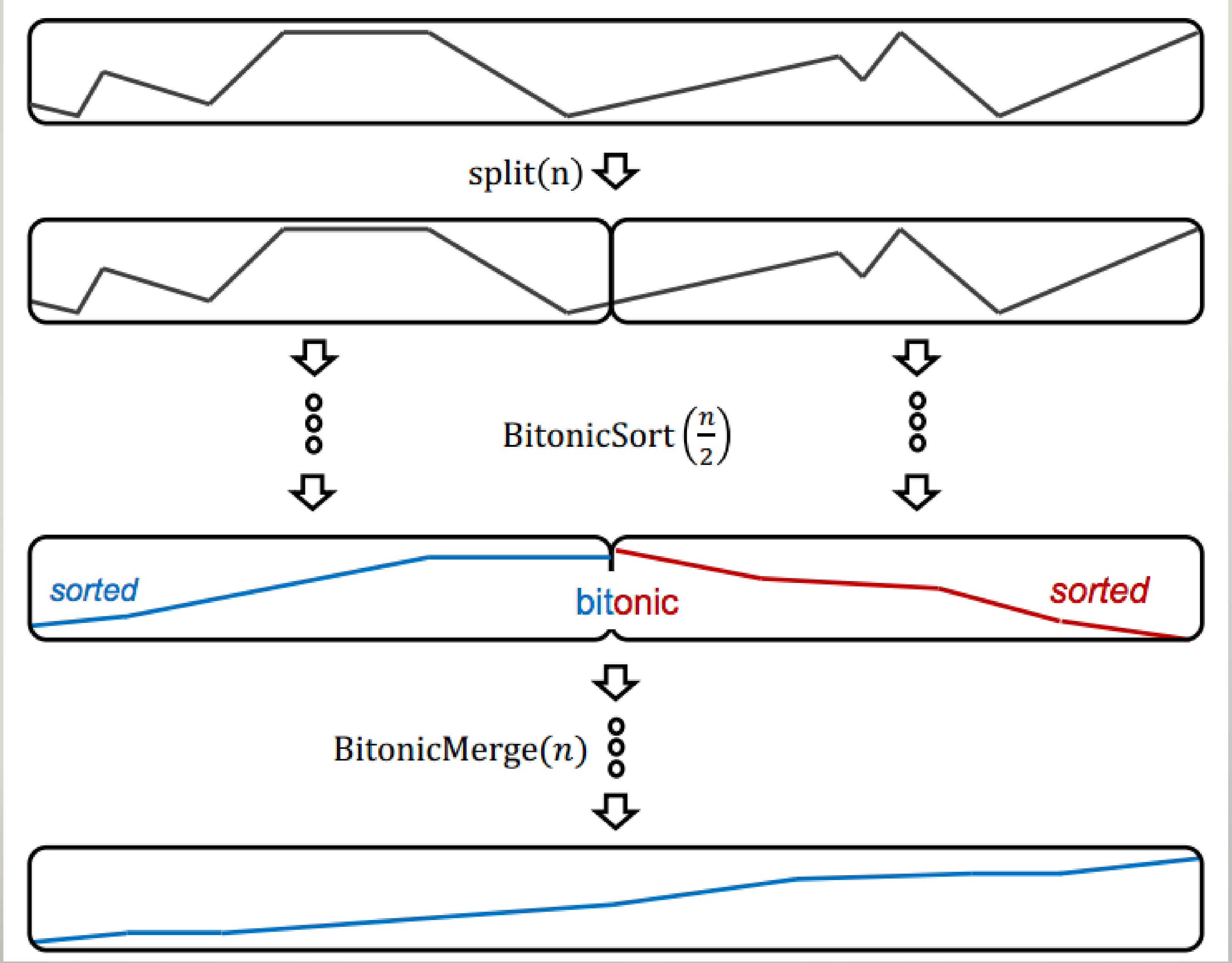
Our Goal

Bitonic Sort

- In other words, we are doing two sortings

1. First half : sort in ascending order
2. Second half: sort in descending order

- So, we can apply a Bitonic sort (or bitonic merge) !
  - As each half now is bitonic itself



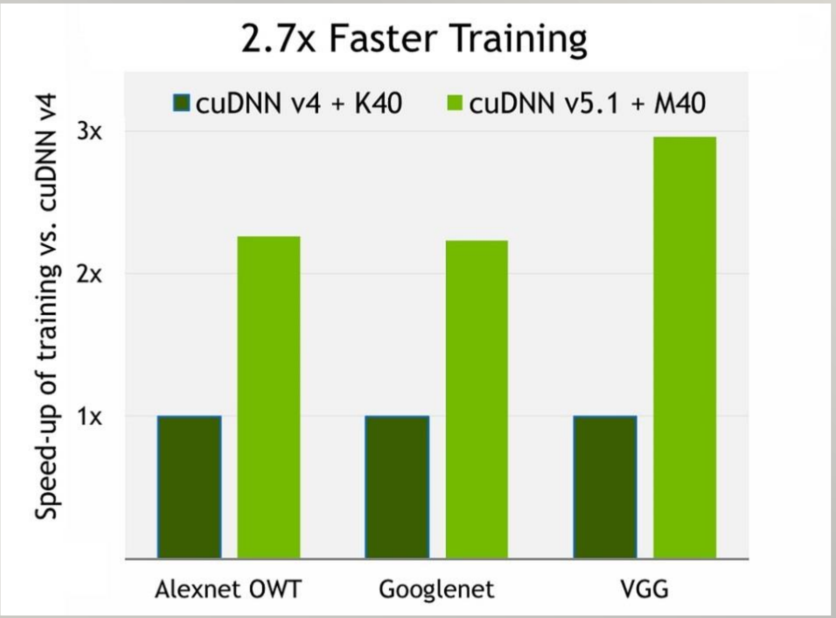
split(n)
BitonicSort$\left(\frac{n}{2}\right)$
sorted
bitonic
sorted
BitonicMerge(n)

Bitonic Sort

- Creating a bitonic sequence is actually sorting the data
  - Following a special order
- It is data independent
  - We can make a sorting network!
- So, very suitable for parallel processing

Deep Learning with GPU

- cuDNN
  - https://developer.nvidia.com/cudnn
- Many deep learning libraries support GPU
  - Caffe, Torch, Chainer
- DIGITS
  - For visualization
- (refers to notes from nVidia)



Summary

- We have discussed different topics related to GPU, e.g.
  - Difference between CPU and GPU
  - Strength and weakness of GPU
  - Graphics applications of GPU
  - Various types of Shaders
  - General Purpose GPU