UiO : University of Oslo

INF3320: Computer Graphics and Discrete Geometry

Advanced topics

Martin Reimers, André R. Brodtkorb

November 18, 2014

Advanced topics

Akenine-Möller, Haines and Hoffman:

- BRDF theory (Chapter 7.5)
- Global illumination (Chapter 9.1-9.8)

Bi-directional Reflectance Distribution Functions

Bi-directional Reflectance Distribution Functions

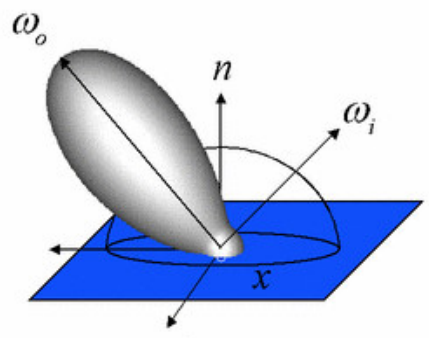
Recall: Phong shading equation - light/surface interaction

$$\mathbf{L}_o(\mathbf{v}) = \mathbf{L}_{ambient} + \mathbf{L}_{diffuse}(\mathbf{l}) + \mathbf{L}_{specular}(\mathbf{l}, \mathbf{v})$$

More general material-light interaction involves BRDF

$$\mathbf{f}(\mathbf{l}, \mathbf{v}) := \frac{dL_o(\mathbf{v})}{dE(\mathbf{l})}$$

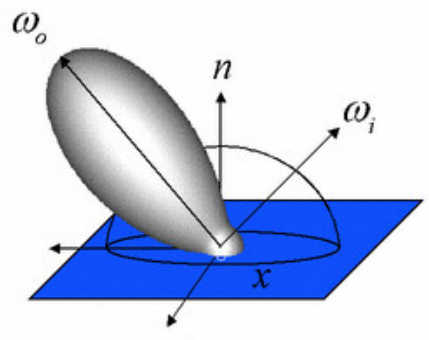
Ratio: outgoing/incoming light in direction $\mathbf{v}$ from direction $\mathbf{l}$



Bi-directional Reflectance Distribution Functions

In the special case of point/directional light source

$$\mathbf{f}(\mathbf{l}, \mathbf{v}) = \frac{\mathbf{L}_o(\mathbf{v})}{\mathbf{E}_L \cos\theta}$$

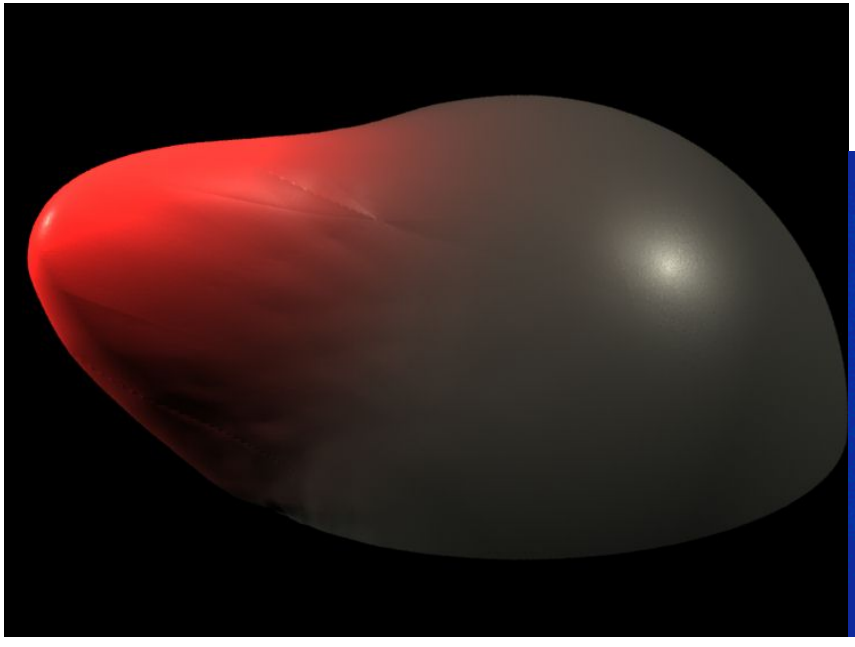


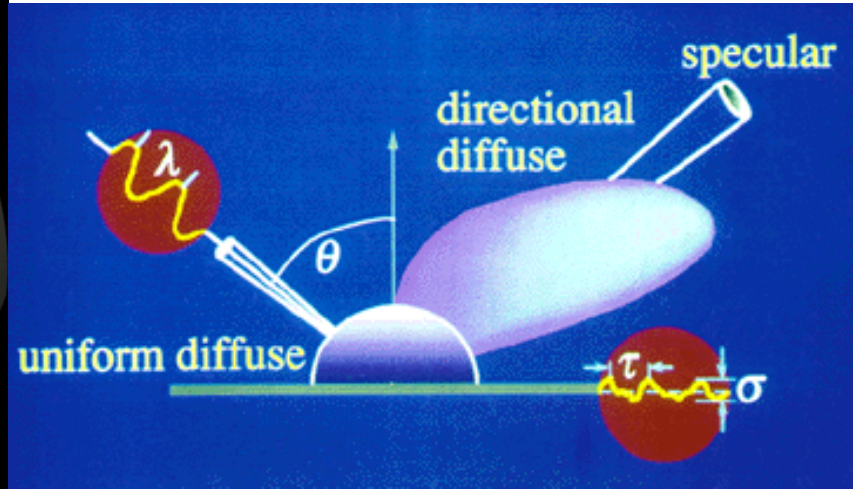
Can add up components: $\mathbf{L}_o(\mathbf{v}) = \sum_k \mathbf{f}(\mathbf{l}_k, \mathbf{v}) E_{l_k} \cos\theta_k$

Like in Phongs model: $\mathbf{L}_o(\mathbf{v}) = \mathbf{L}_{ambient} + \mathbf{L}_{diffuse}(\mathbf{l}) + \mathbf{L}_{specular}(\mathbf{l}, \mathbf{v})$

BRDF

Examples



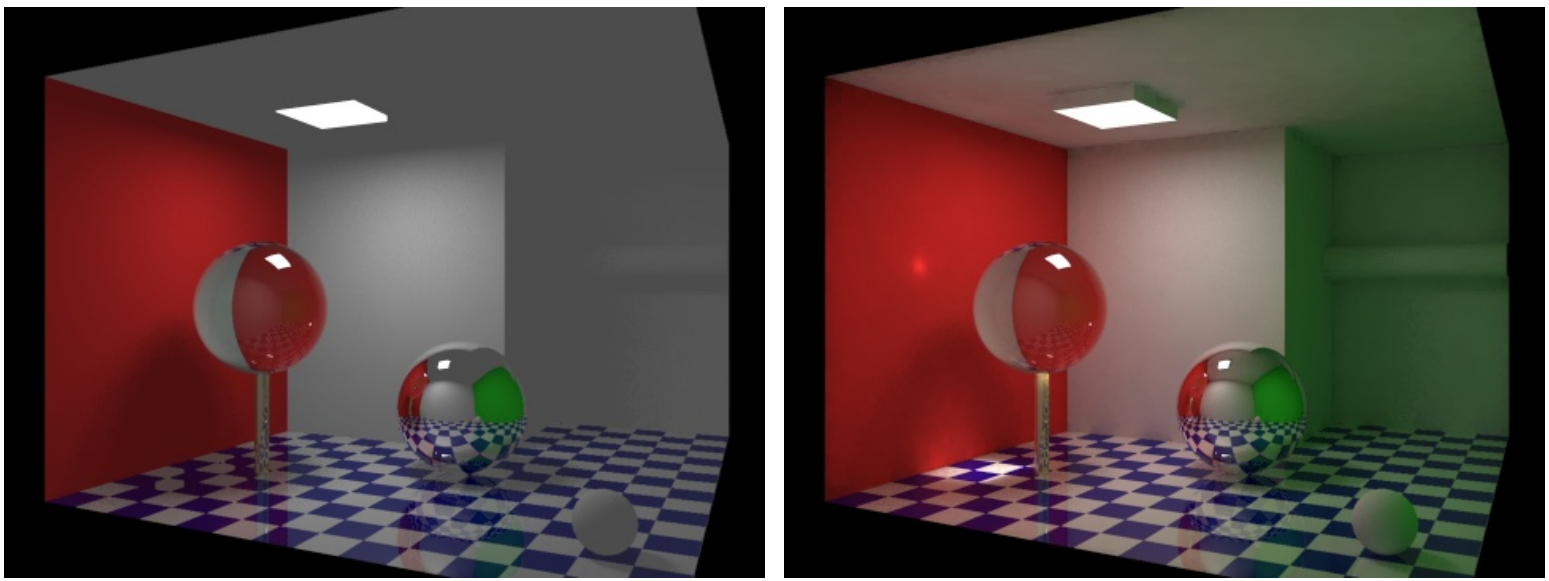


Building, storing, evaluating BRDFs is a big field of research

Global Illumination

Indirect illumination

Direct illumination models are very simplistic
... but renders in real time



Indirect (global) illumination looks a lot more realistic
... but requires a lot more effort

Global Illumination

The ligthing at a particular point depends on other objects:

- Reflection
- Shadows
- Refraction
- ...

Diffuse inter-reflection is perhaps the most important factor - light bounces off a number of objects before it hits the eye

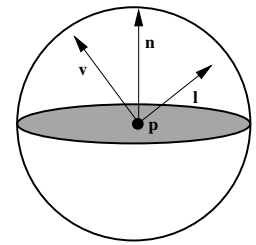
Techniques:

- Ambient occlusion
- Radiosity
- Ray-tracing
- Pre-computed lighting

The rendering equation

The outgoing radiance from surface point **p** in direction **v** is

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_\Omega \mathbf{f}(\mathbf{l}, \mathbf{v}) \otimes L_i(\mathbf{p}, \mathbf{l}) \cos\theta_i d\omega$$



- $\theta_i = \angle(\mathbf{l}, \mathbf{n})$
- $\mathbf{f}(\mathbf{l}, \mathbf{v})$ is the BRDF
- $\mathbf{L}_i(\mathbf{p}, \mathbf{l})$ is the incoming radiance from $d\omega_i \subset \Omega$
- $\mathbf{L}_e(\mathbf{p}, \mathbf{l})$ is the emitted radiance

Lighting models amount to approximating such equations

Simulate light transport, i.e. how photons bounce around

Global illumination: simulate LE as a global system (interactions)

Global illumination algorithms

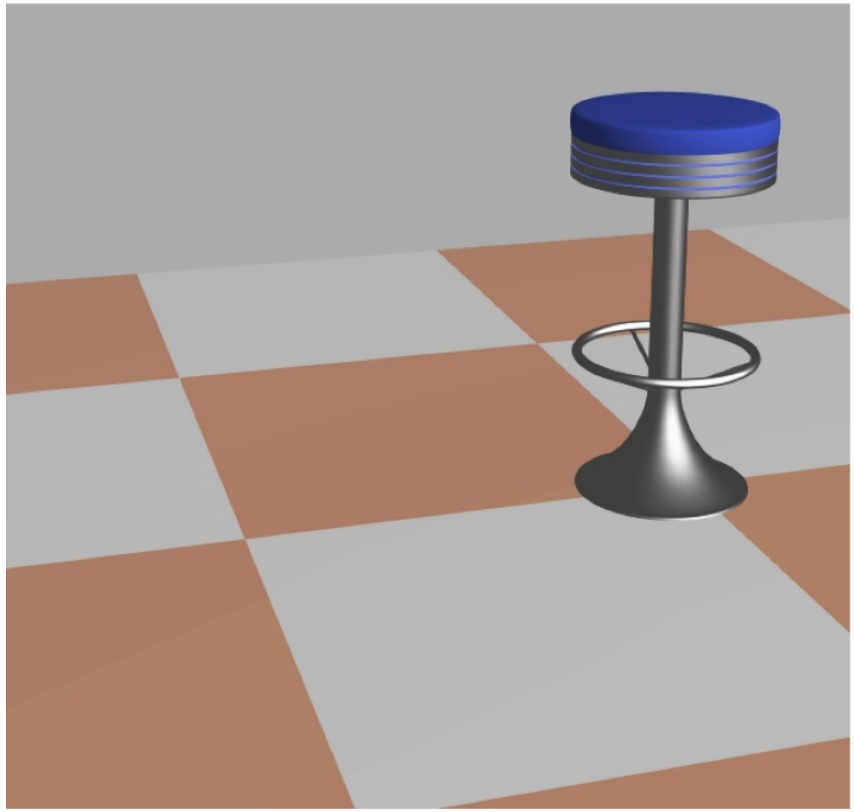
Elements:

- Shadows and ambient occlusion (does light hit surface?)
- Reflections, transmittance, refraction (light/material interaction)
- Full Global Illumination (putting it all together)
  - Radiosity
  - Raytracing
  - Photon mapping

Common to most algorithms:

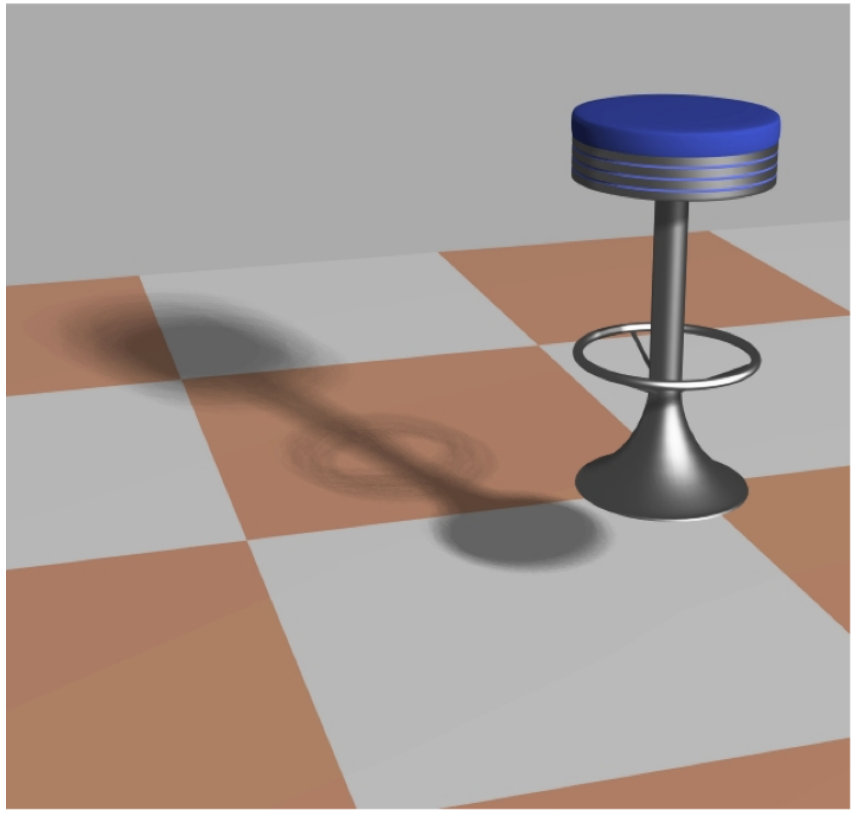
- heavy computations
- approximations for real time
- require pre-computations

Shadows



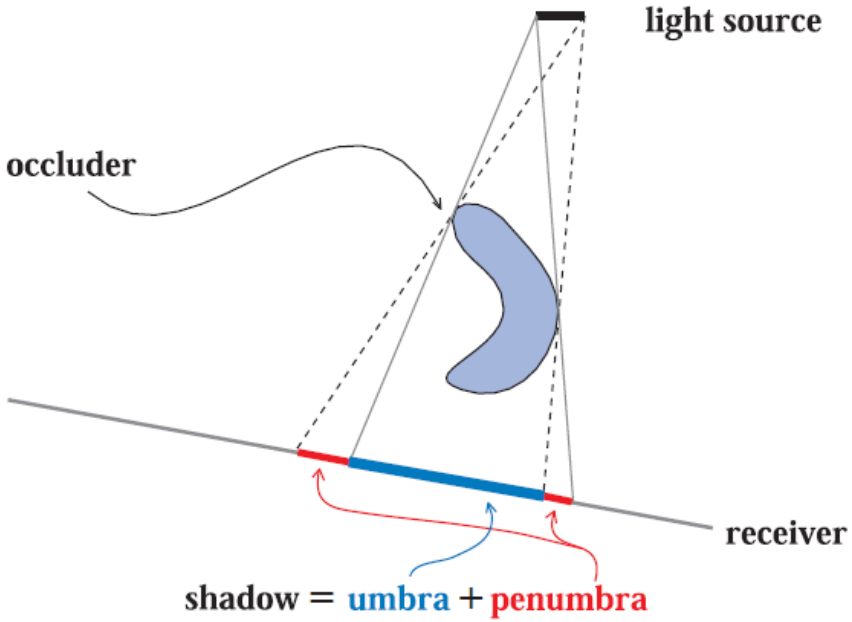
Shadows yields realism, and give clues about spatial relationships

Shadows

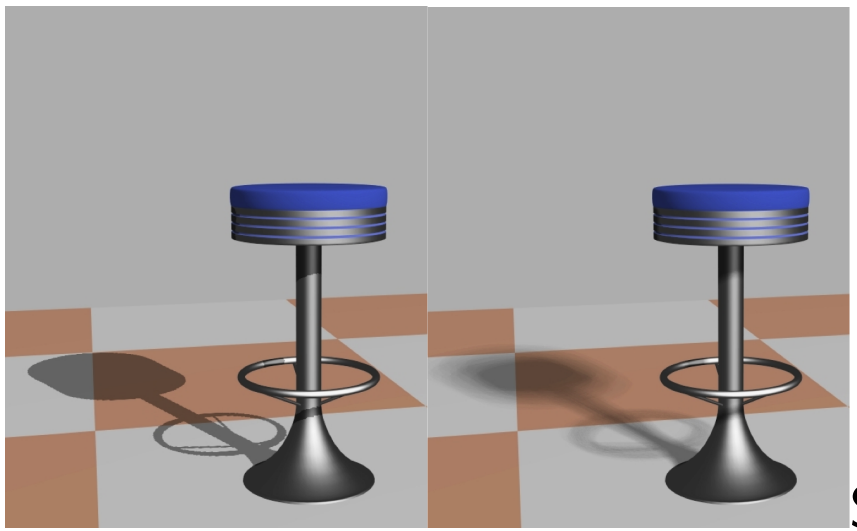


Shadows yields realism, and give clues about spatial relationships

Shadows and light sources



Hard (pointlight)



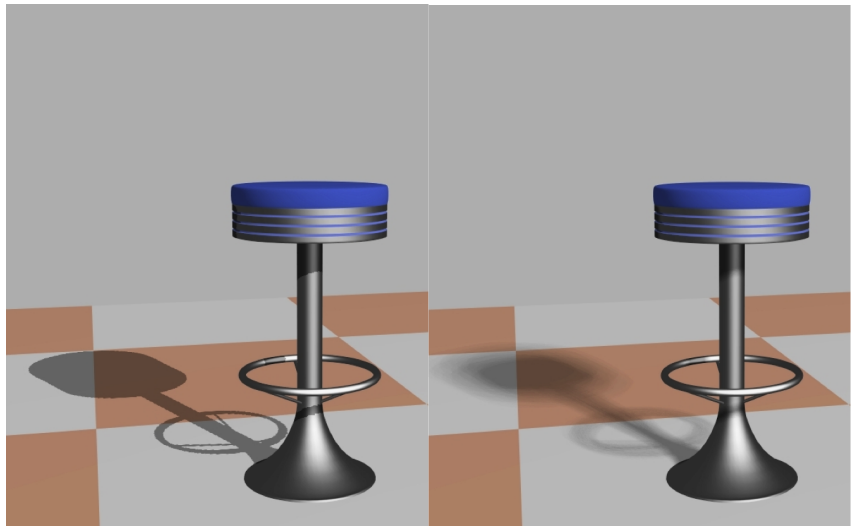
Soft (source with area)

Shadow algorithms

Most Famous algorithms:

1. Shadow maps
2. Shadow volumes

Both produce hard shadows



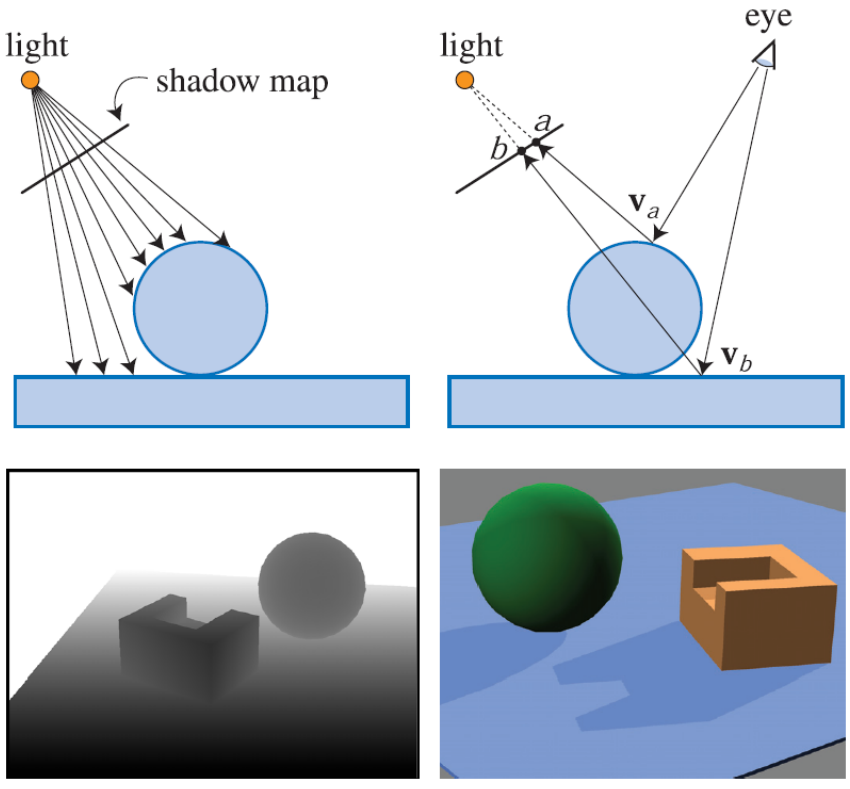
... but can be used to produce soft shadows

Shadow map algorithm

Popular, real-time method [Williams 1978], Idea: two render-passes

Shadow pass

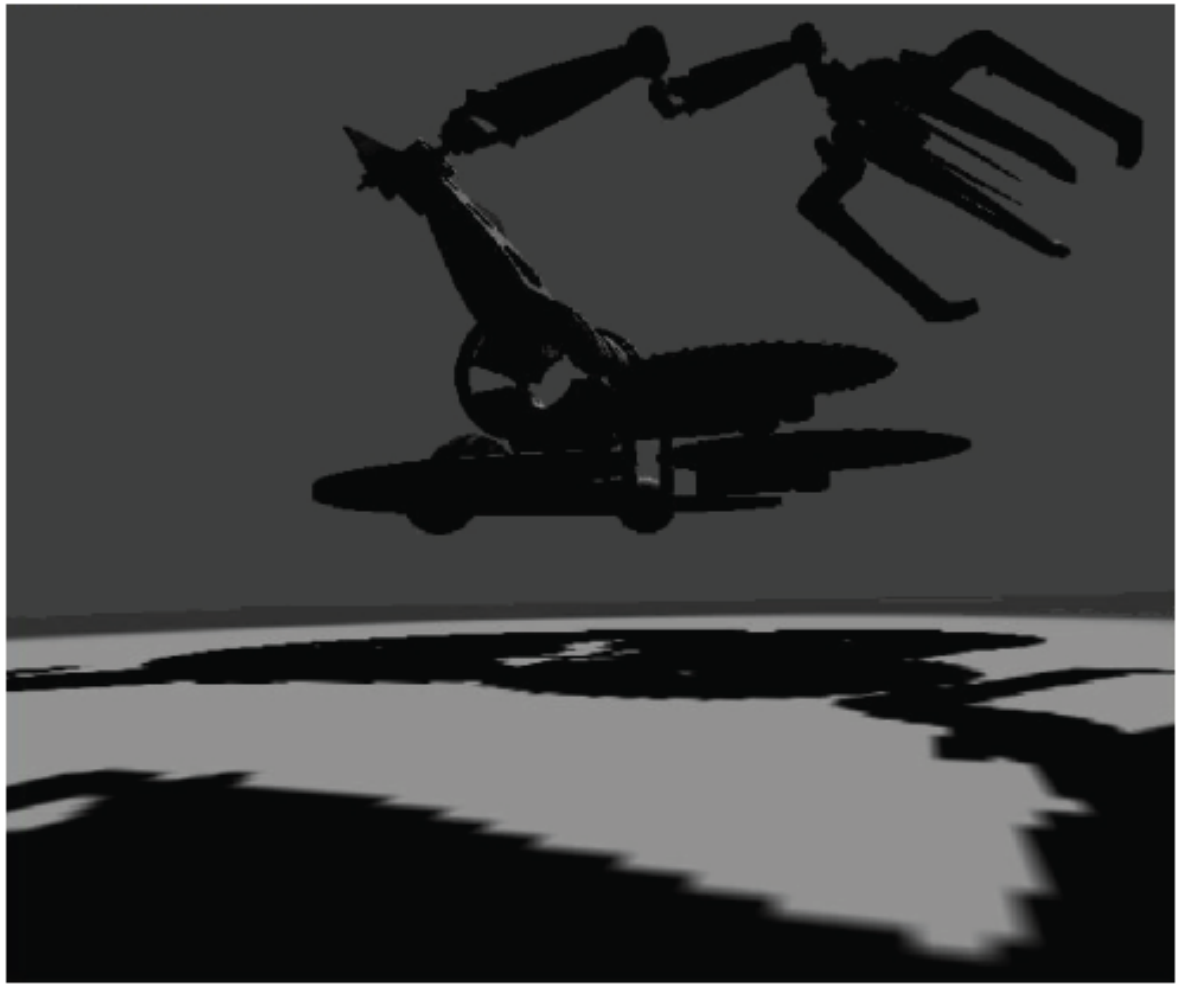
- Camera in lightpos
- Depth test
- Shadow map = occluder depth from light



Render pass

- Camera in eyepos
- Lighting: check fragments depth from light

Shadows maps

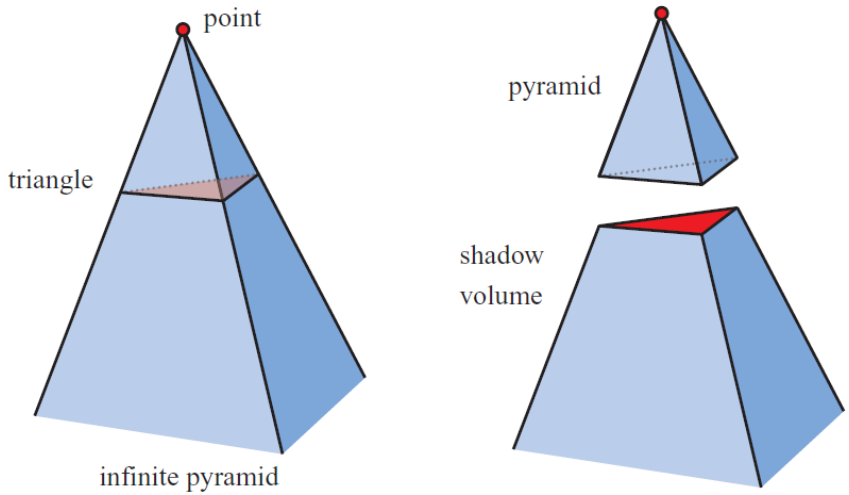


- Fast and simple method
- Image based, sampling artefacts

Shadow volumes algorithm

Idea [Heidmann 1991]:

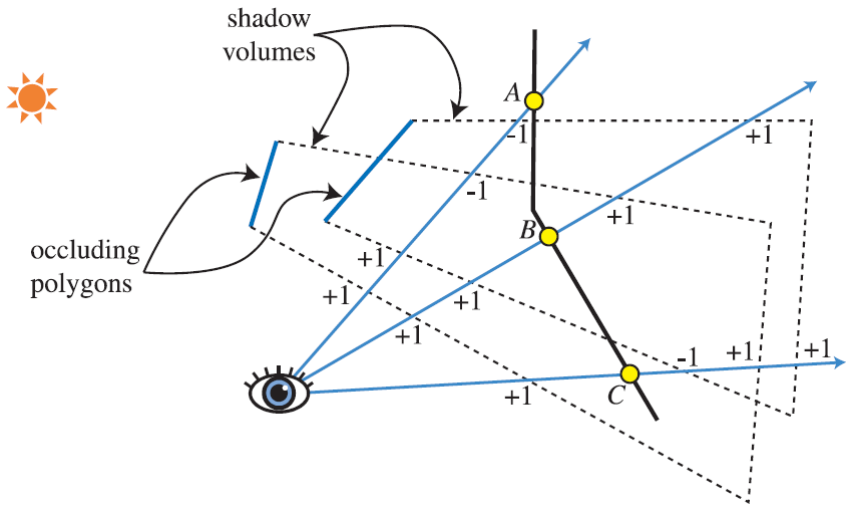
- Bound the shadow-volume by polygons (quads)
- A fragment is in shadow if there is an odd number of quads between it and the eye



Implementation:

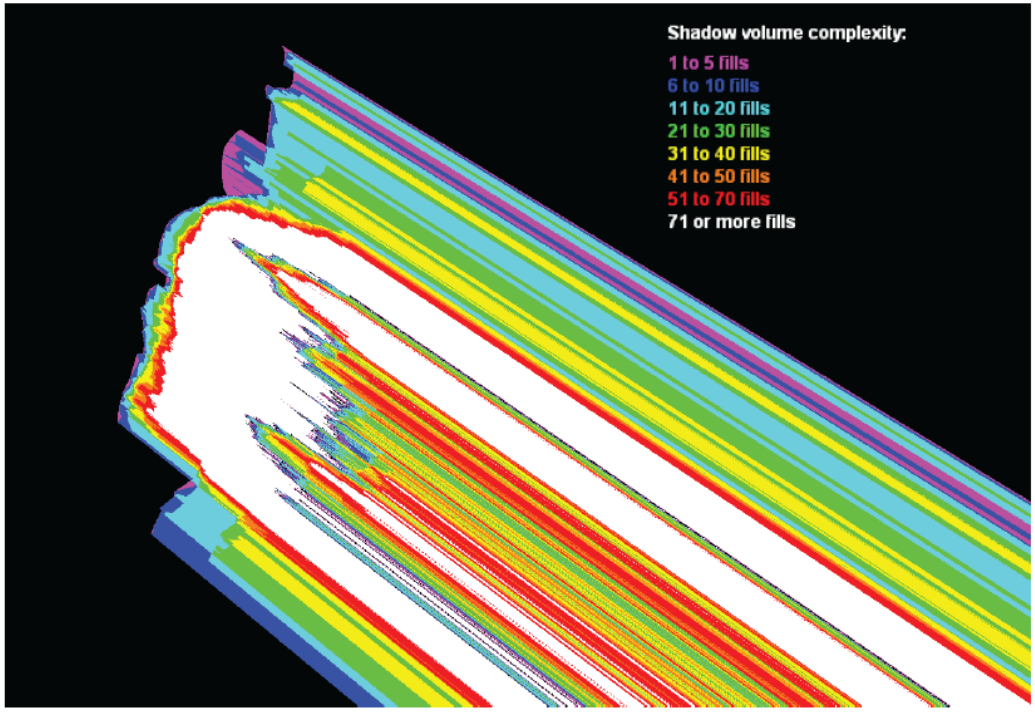
1. Render geometry with depth-test, ambient light
2. Render front facing polygons, incrementing stencile buffer
3. Render back facing polygons, decrementing stencile buffer
4. Render diffuse and specular where stencile buffer is 0

Shadow volumes algorithm



- Shadow volumes generated per triangle (many quads)
- .. or per silhouette edge (more complex, simpler rendering)
- use vertex/geometry shader
- Efficient real-time algorithm
- Not image based $\Longrightarrow$ no sampling problems

Shadows volumes



Shadow volumes in Doom3

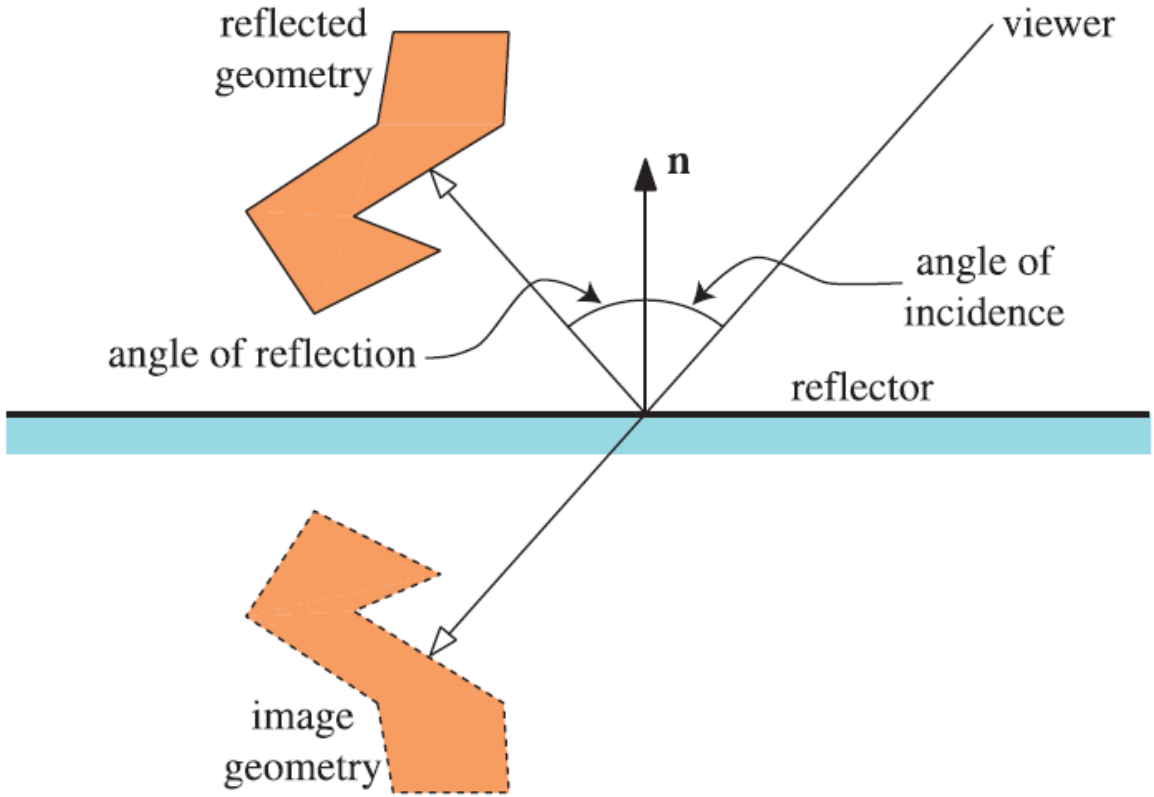
Fast but hard shadows (hard game ...)

Soft Shadows

... hard algorithms, more work

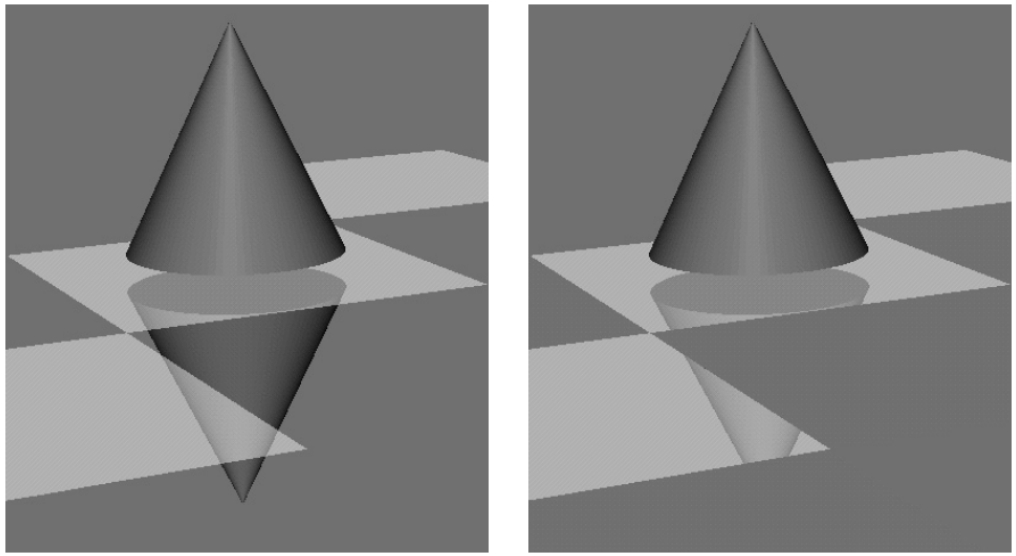
- Simulate penumbra from area light source (penumbra wedges)
- Smoothen hard shadow edges, e.g. by filtering techniques
- Ray-tracing

Reflections



Can achieve planar reflection by `glScale(1,1,-1)`

Planar reflections

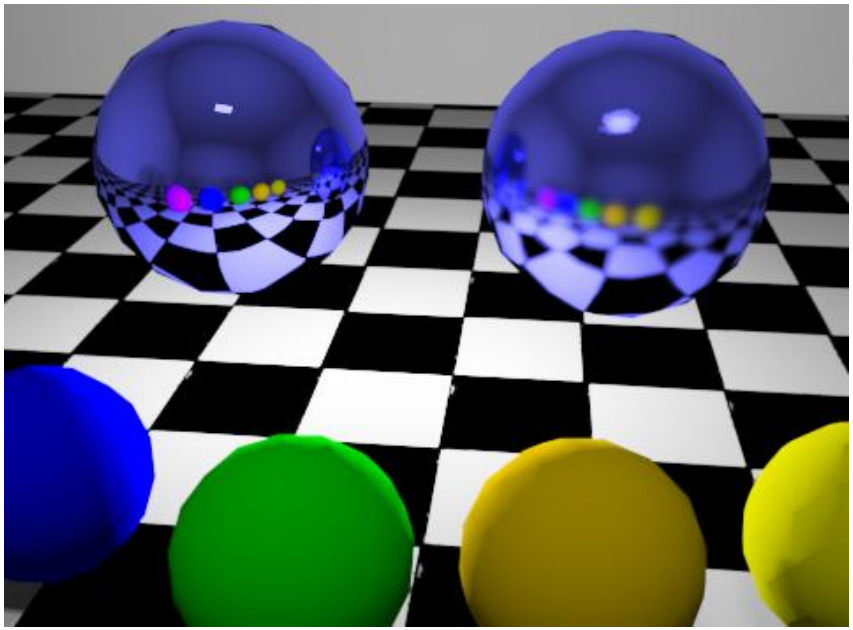


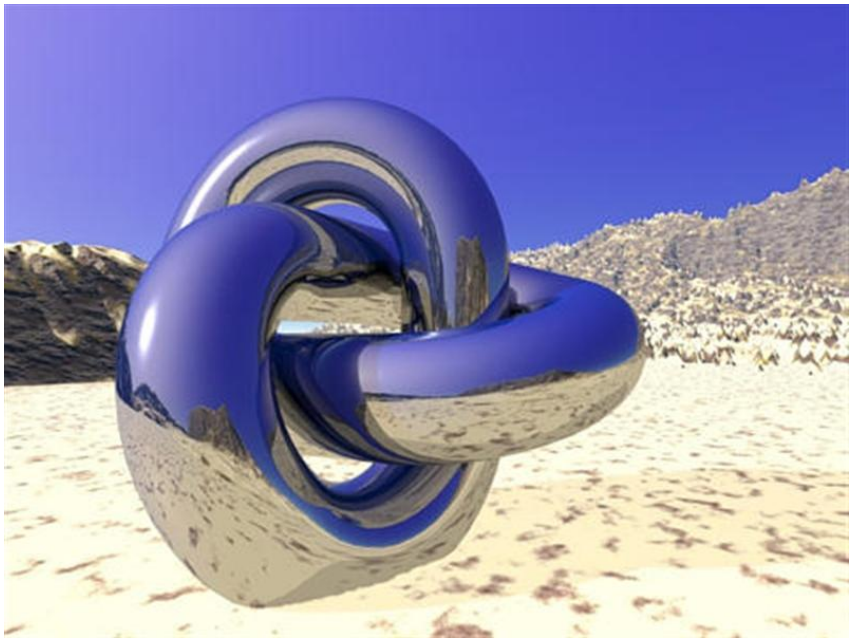
Algorithm:

1. Render ground plane to stencil buffer
2. render glScale'd geometry with stencil masking
3. render ground plane semi transparent
4. render unscaled model

Reflections

Real life reflections can be a lot more complex

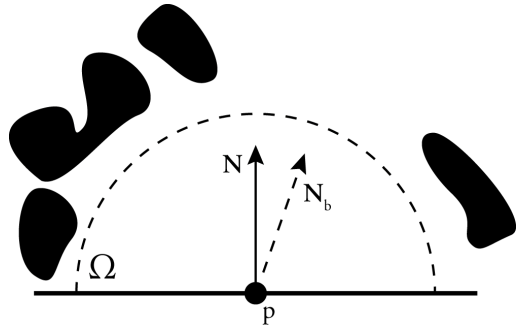




We have seen one simple solution before: environment mapping

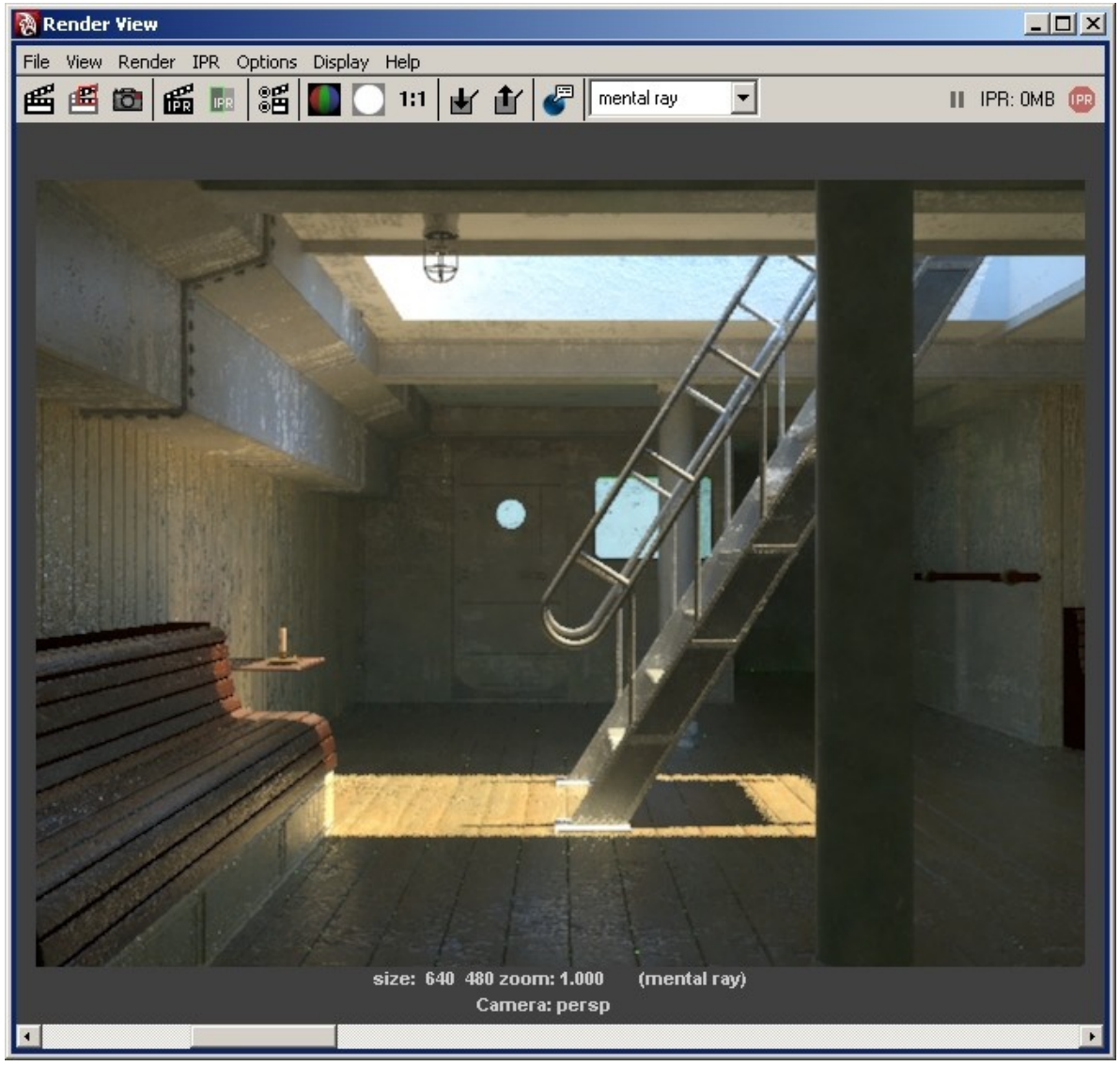
Ambient occlusion

How much ambient light hits a point? How much sky is seen?

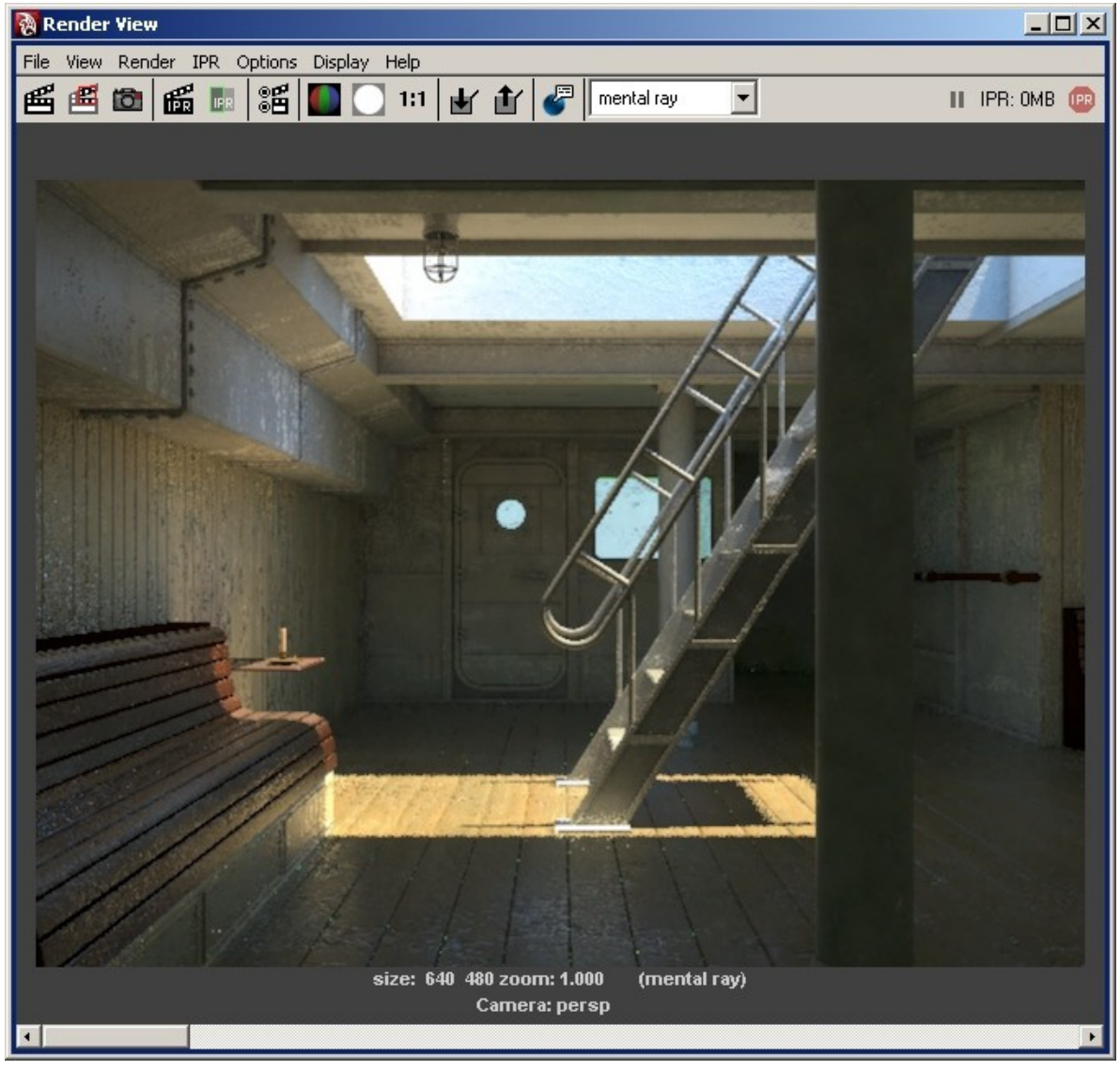


Both self occlusion and other occluders

Ambient occlusion

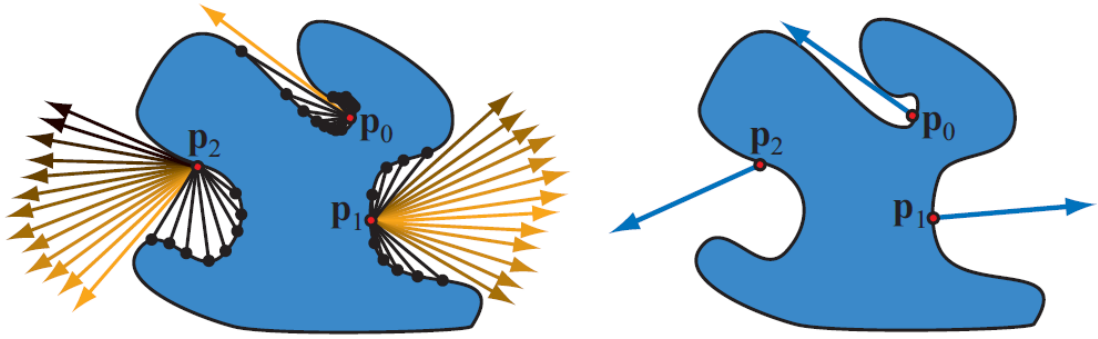


Ambient occlusion



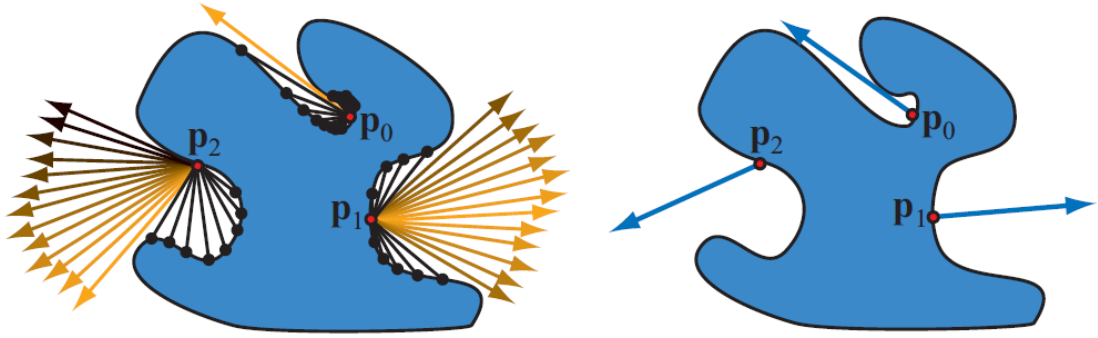
Ambient occlusion

Slow Algorithm: send rays in many directions

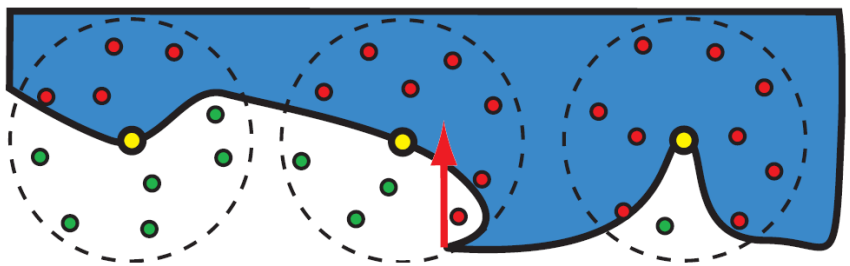


Ambient occlusion

Slow Algorithm: send rays in many directions



Fast alg.: Screen space AO: GPU technique from 2007 (CryTek)



Test a number of points in a sphere against the z-buffer:
Many points fail $\Longrightarrow$ point occluded

Transmission, refraction, reflection

Transparent materials: α-based over-blending is non-physical, must take e.g. thickness and absorption into account, Snells law

From Crysis (Crytek)

Caustics

Reflection or refraction of curved surface

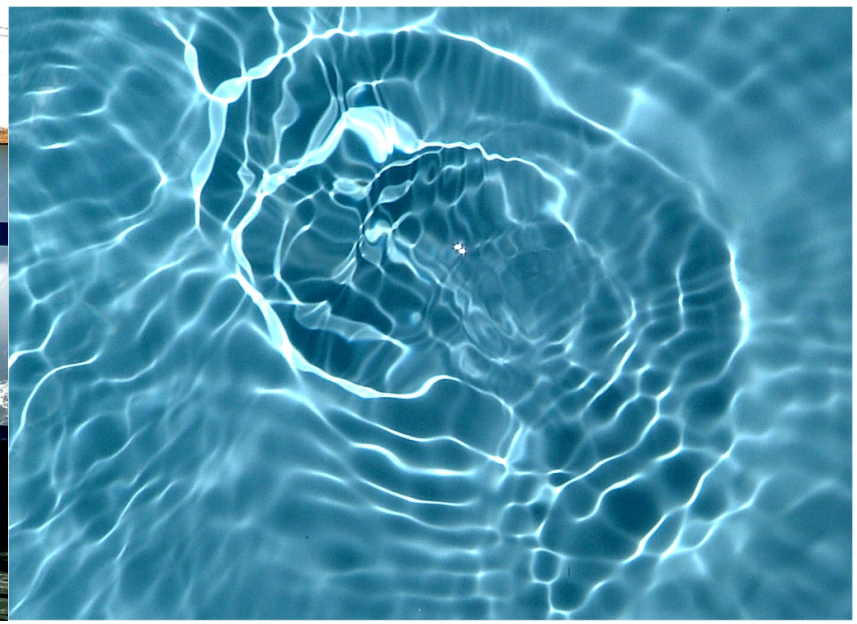
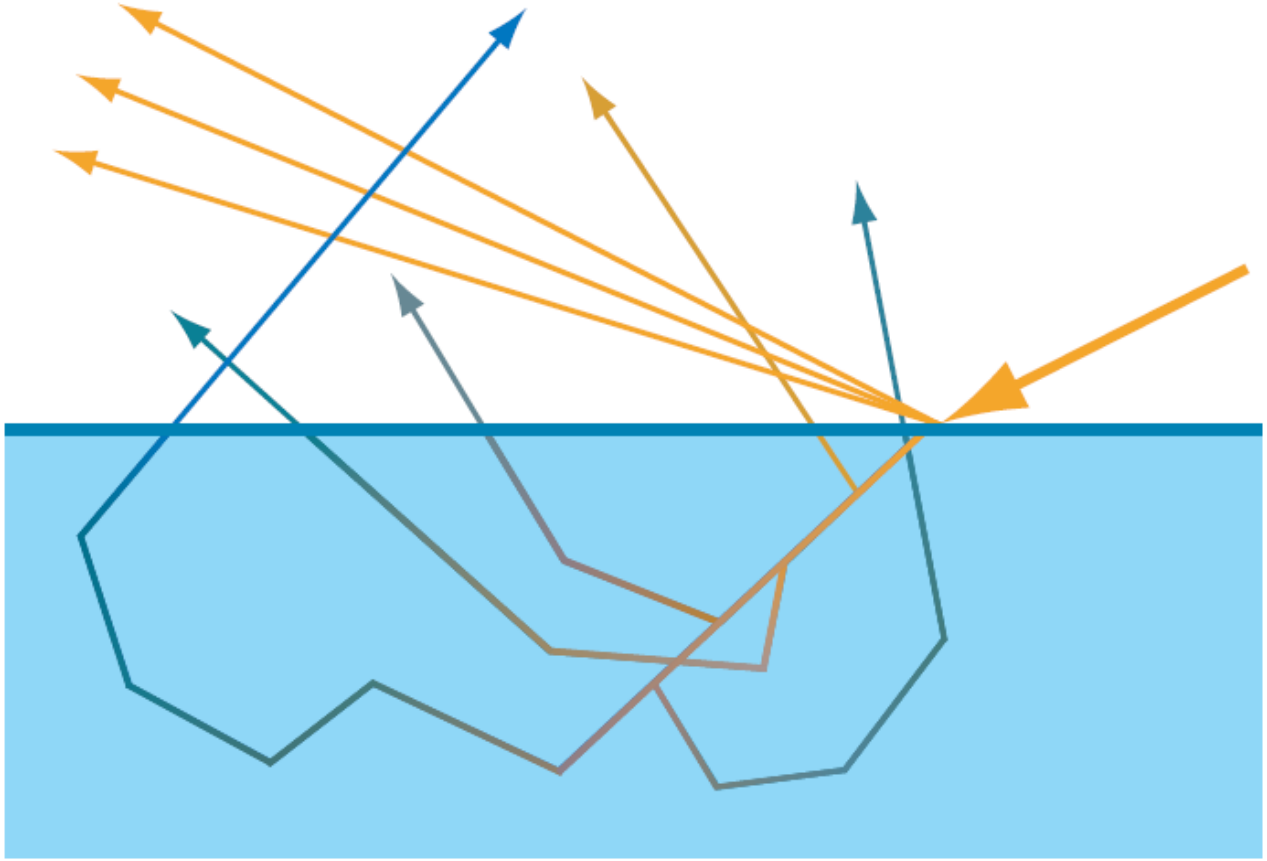


Image space and object space techniques (multipass)

Subsurface scattering

Light scattered inside a substance and re-emitted, e.g. skin, marble



Subsurface scattering

Techniques: BSSRDF, wrap lighting (color shift), texture diffusion, depth maps, ..

Full Global Illumination

All the techniques discussed so far are local and specialized solutions of the rendering equation

Full Global Illumination amounts to simulating light transport in the whole scene, i.e. solve rendering equation globally.

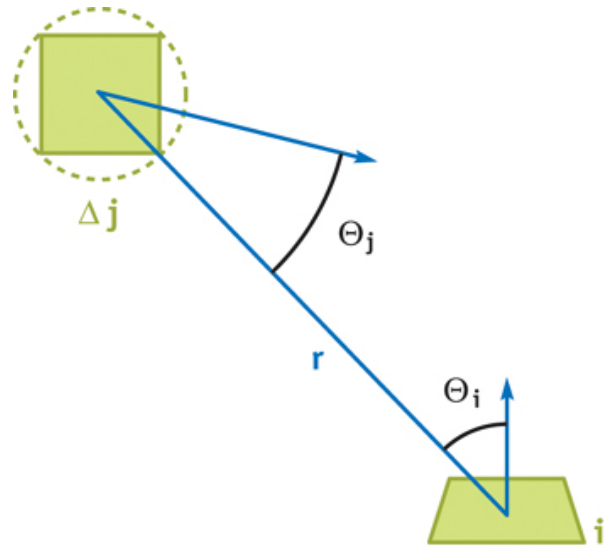
Radiosity

Finite element type method for diffuse scenes and indirect lighting

Radiosity II

Based on *form factors* that say how much one surface point depend on the lighting in another point

$$df_{ij} = \frac{\cos\theta_i \cos\theta_j}{\pi d^2} h_{ij} da_j$$



Simulation approach: The scene is discretized into patches with inter-dependent lighting. Solve for equilibrium, given light sources and patch reflectance

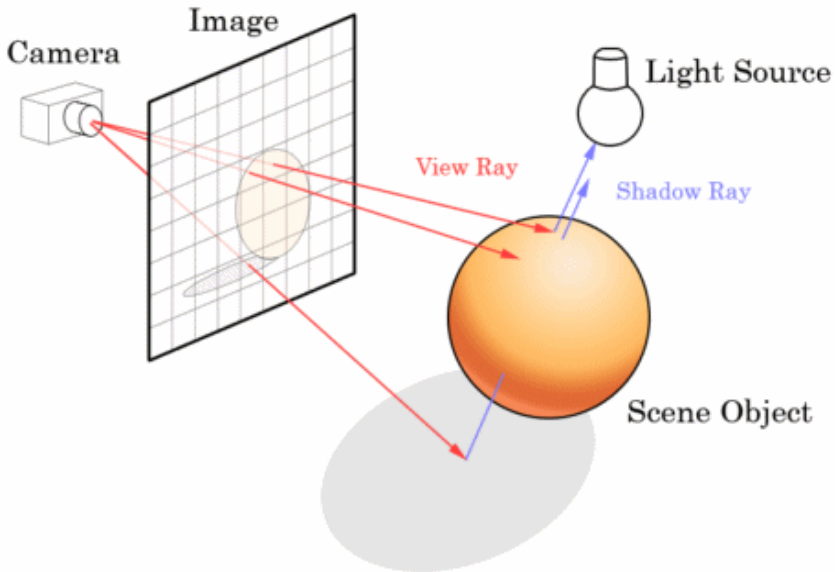
$$A\mathbf{x} = \mathbf{b}$$

Radiosity III

Raytracing

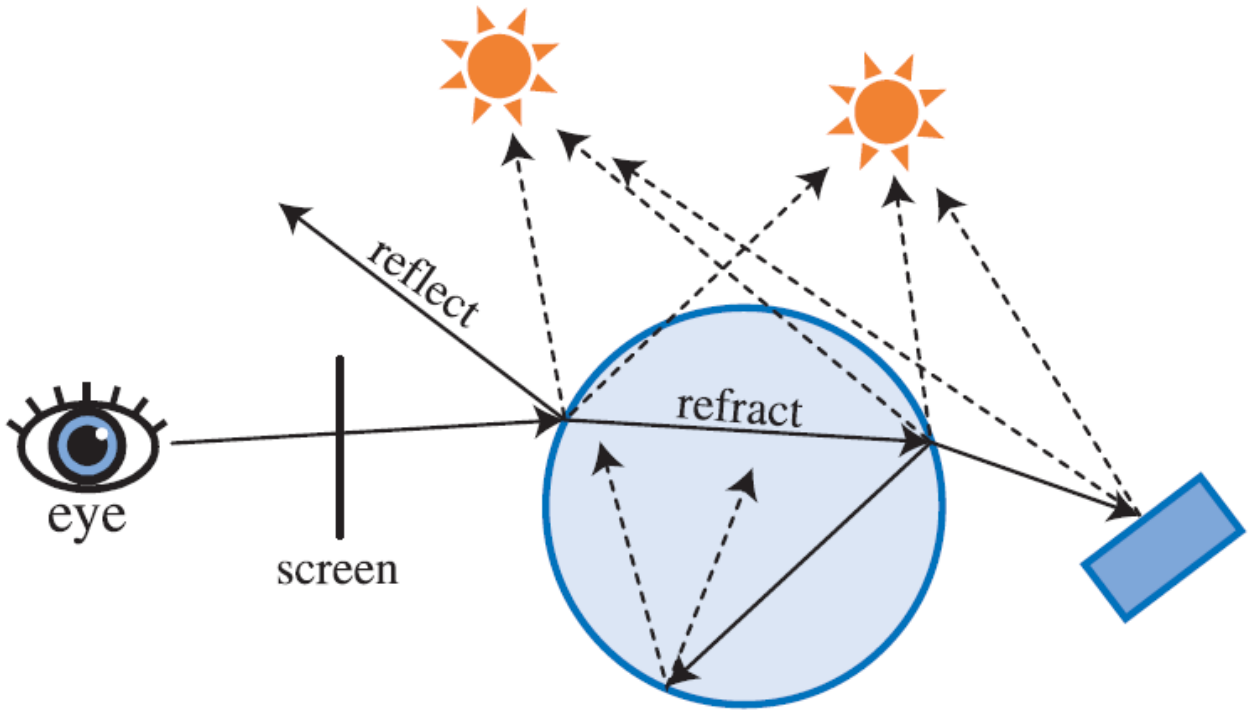
Basic Raytracing algorithm:

1. Emit one ray through each pixel
2. Repeat
   - 2.1 For each ray (until diffuse surface hit, or max-depth)
     - 2.1.1 intersect scene
     - 2.1.2 compute surface interaction/reflection, lighting
     - 2.1.3 spawn new rays when needed



Advanced versions use Monte Carlo Methods (sampling)

Raytracing



Raytracing II

UiO : Department of Informatics
University of Oslo

INF3320: Computer Graphics and Discrete Geometry

Beyond Graphics: GPUs for Computing

André R. Brodtkorb

November 19, 2014

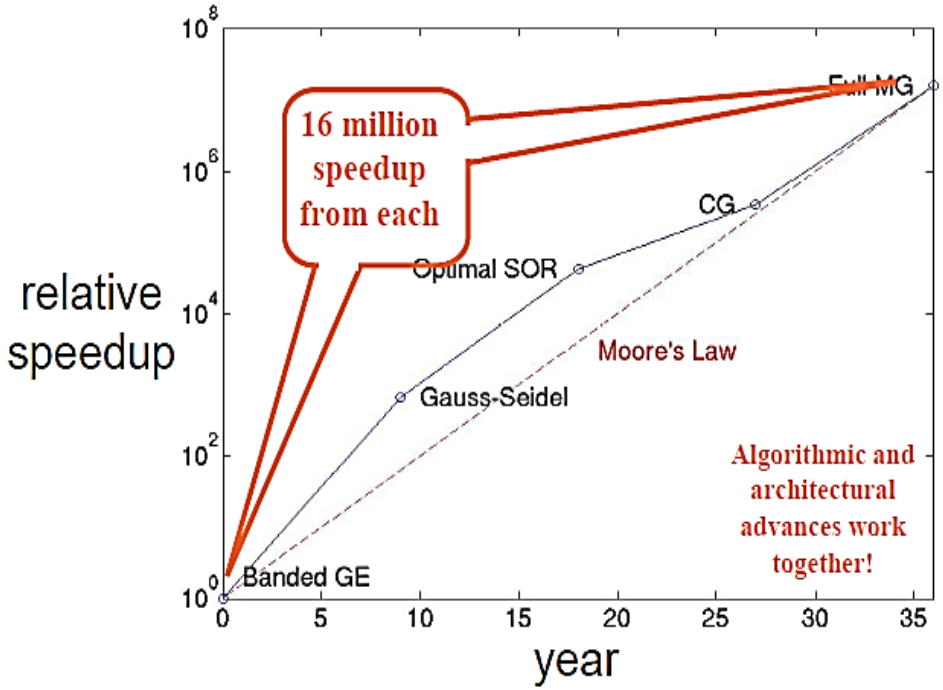
Outline

- Motivation for going parallel
- Multi- and many-core architectures
- Example: Computing π on the GPU
- Optimizing memory access

Motivation for going parallel

Why care about computer hardware?

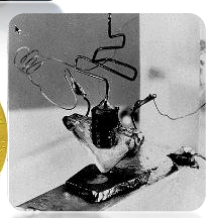
- The key to increasing performance, is to consider the full algorithm and architecture interaction.
- A good knowledge of both the algorithm and the computer architecture is required.
- Our aim is to equip you with some key insights on how to design algorithms for todays and tomorrows parallel architectures.



Graph from David Keyes, Scientific Discovery through Advanced Computing, Geilo Winter School, 2008

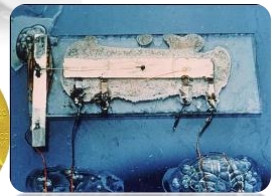
History lesson 1/2

1942: Digital Electric Computer
(Atanasoff and Berry)



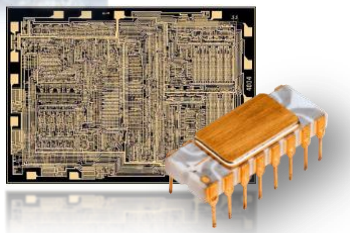
1947: Transistor
(Shockley, Bardeen, and Brattain)

1956

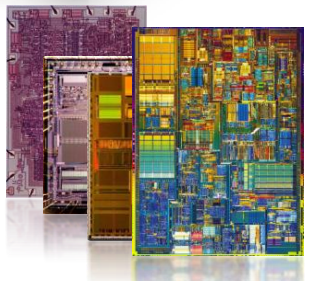


1958: Integrated Circui
(Kilby)

2000



1971: Microprocessor
(Hoff, Faggin, Mazor)



1971- Exponential growth
(Moore, 1965)

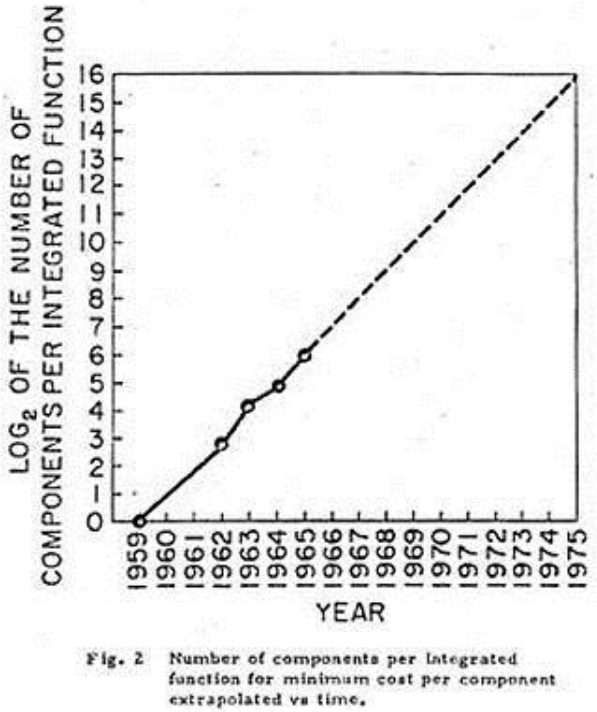
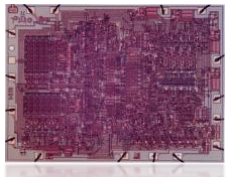


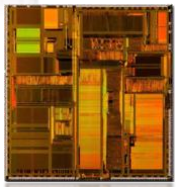
Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

History lesson 2/2

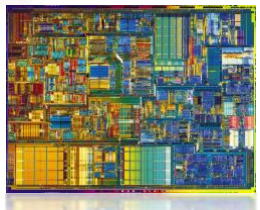


1971: 4004,
2300 trans, 740 KHz

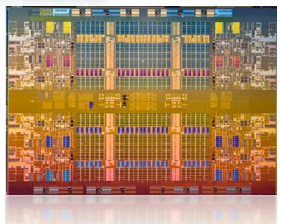
1982: 80286,
134 thousand trans, 8 MHz



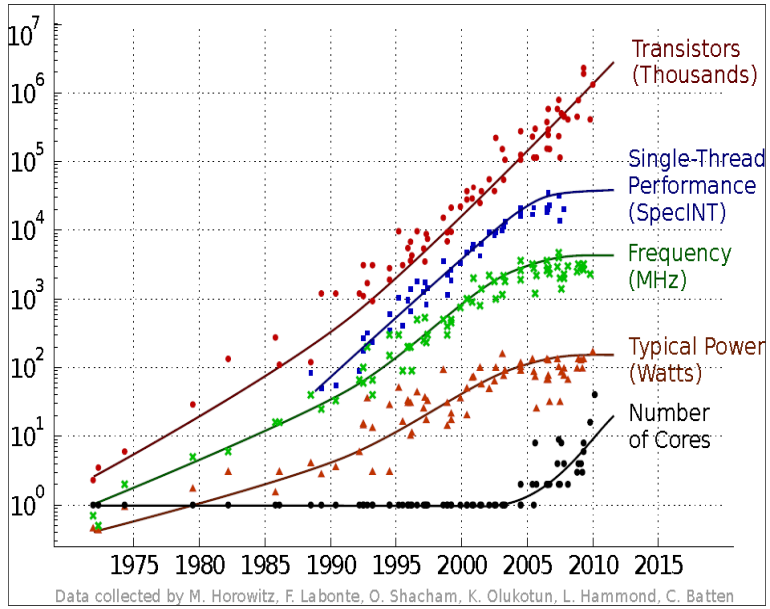
1993: Pentium P5,
1.18 mill. trans, 66 MHz



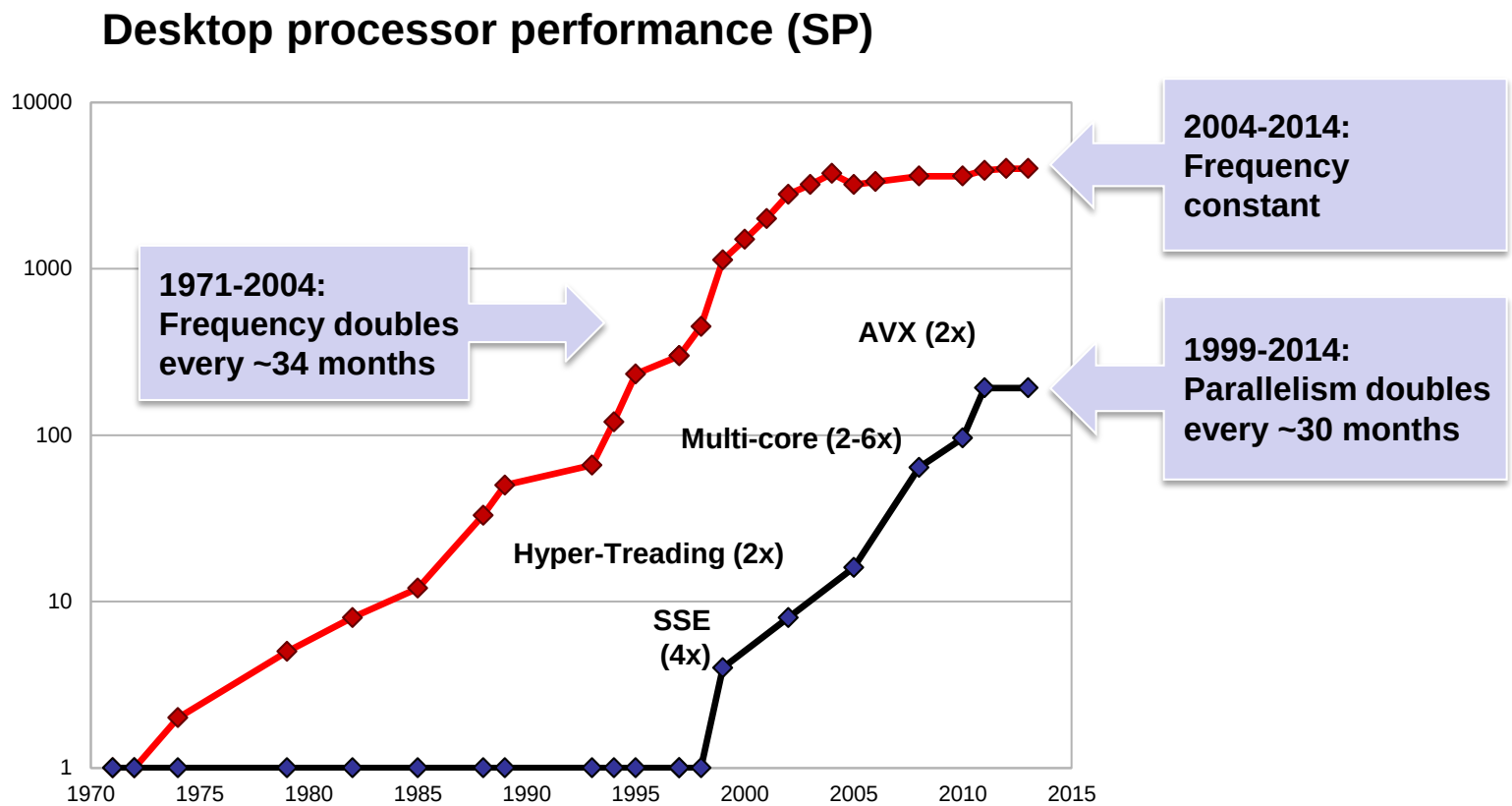
2000: Pentium 4,
42 mill. trans, 1.5 GHz



2010: Nehalem
2.3 bill. Trans, 8 cores, 2.66 GHz



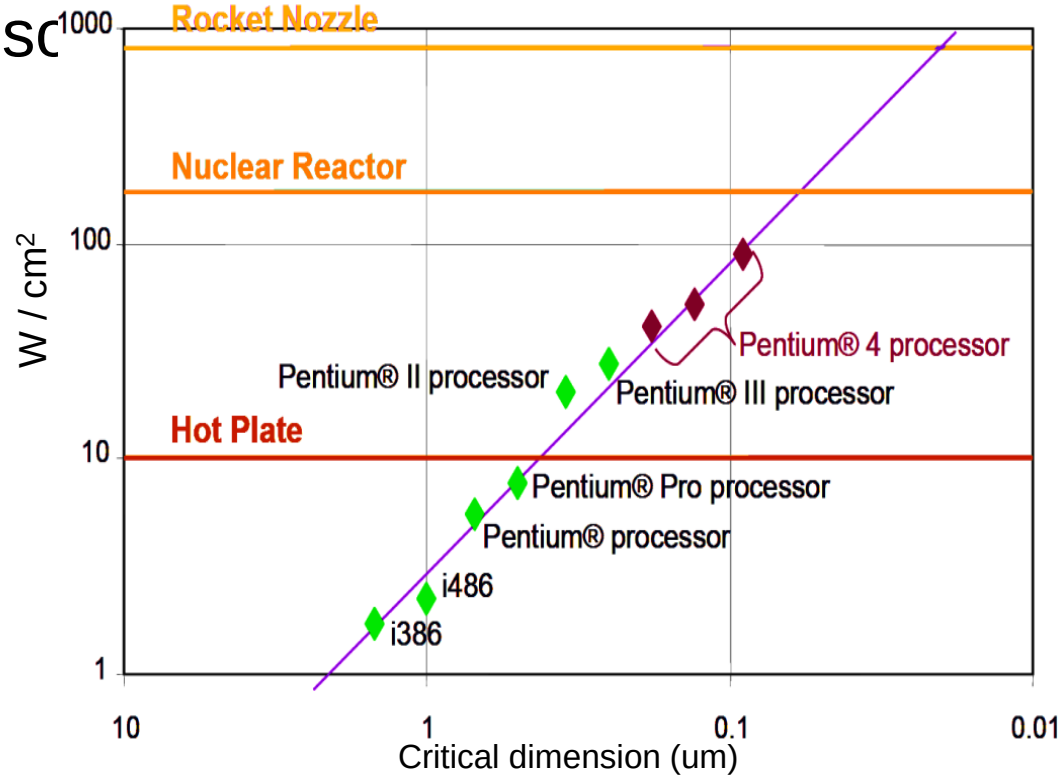
End of frequency scaling



- 1970-2004: Frequency doubles every 34 months (Moore's law for performance)
- 1999-2014: Parallelism doubles every 30 months

What happened in 2004?

- Heat density approaching that of nuclear reactor core: Power wall
  - Traditional cooling solutions (heat sink + fan) insufficient
- Industry sc

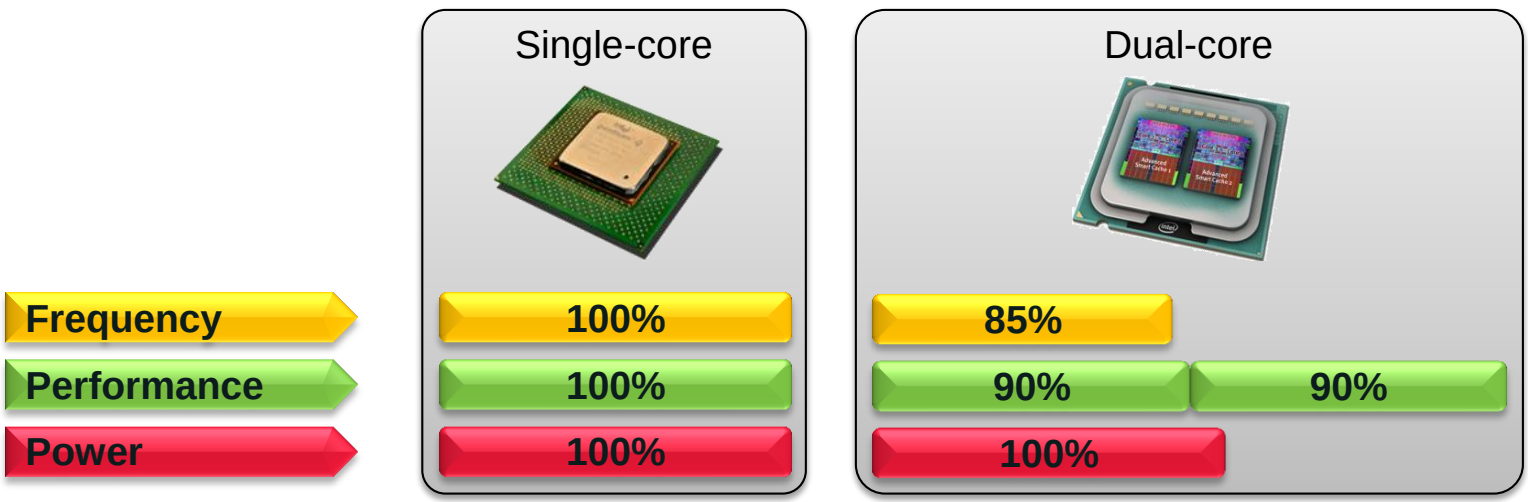


Original graph by G. Taylor, “Energy Efficient Circuit Design and the Future of Power Delivery” EPEPS’09

Why Parallelism?

The power density of microprocessors is proportional to the clock frequency cubed:[1]

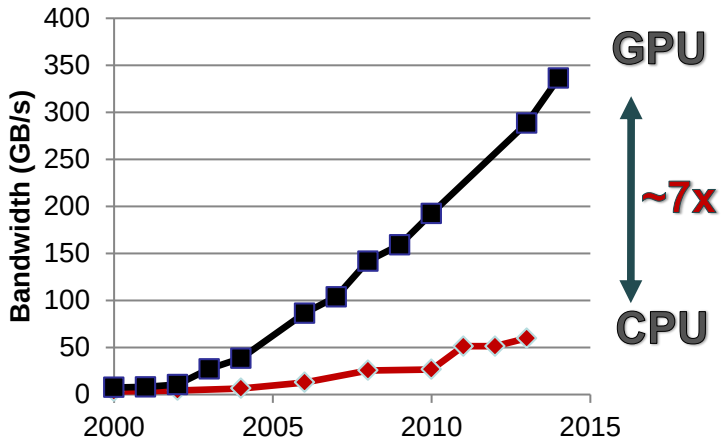
$$P_d \propto f^3$$

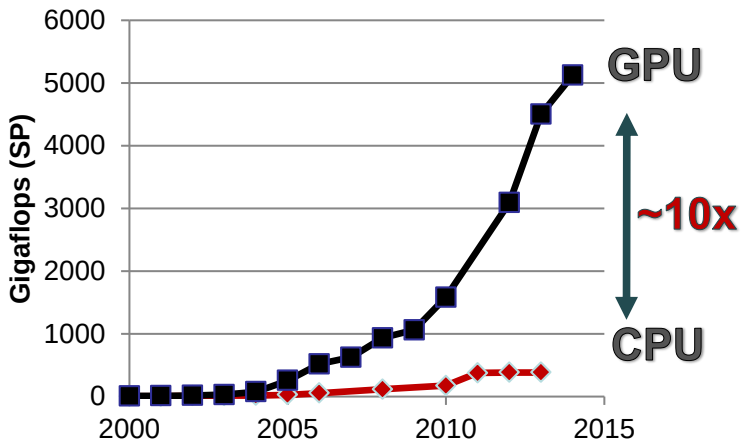


[1] Brodtkorb et al. State-of-the-art in heterogeneous computing, 2010

Massive Parallelism: The GPU

- Up-to 5760 floating point operations in parallel!
- 5-10 times as power efficient as CPUs!





Multi- and many-core architectures

A taxonomy of parallel architectures

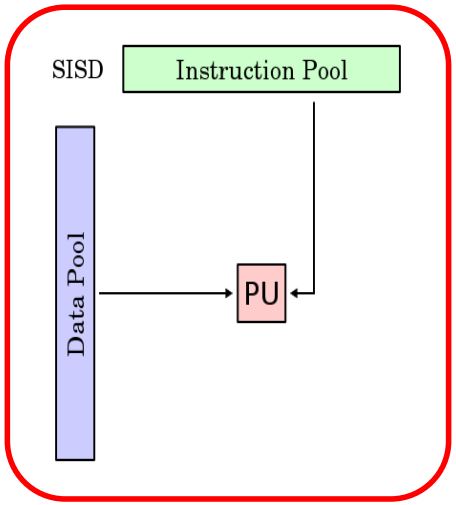
- A taxonomy of different parallelism is useful for discussing parallel architectures
  - 1966 paper by M. J. Flynn: Some Computer Organizations and Their Effectiveness
  - Each class has its own benefits and uses

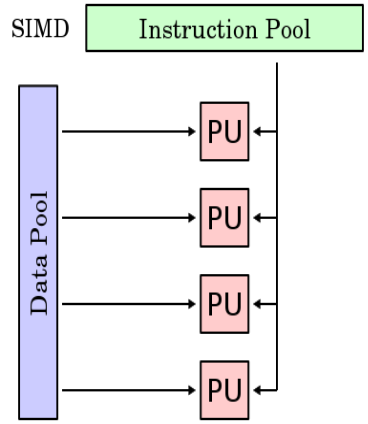
	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

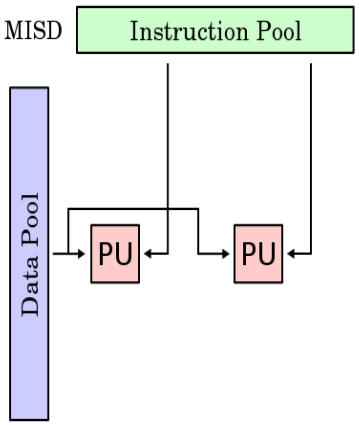
M. J. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., 1966

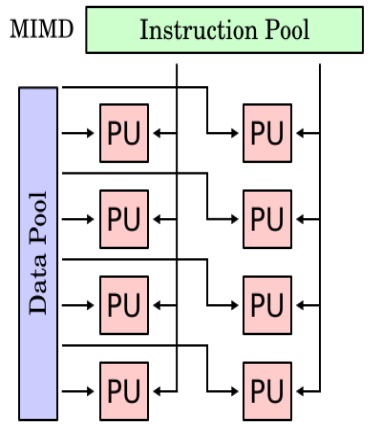
Single instruction, single data

- Traditional serial mindset:
  - Each instruction is executed after the other
  - One instruction operates on a single element
  - The typical way we write C / C++ computer programs
- Example:
  - c = a + b







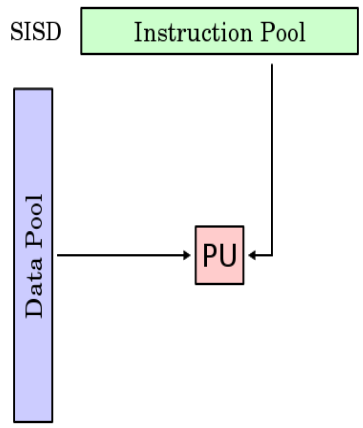


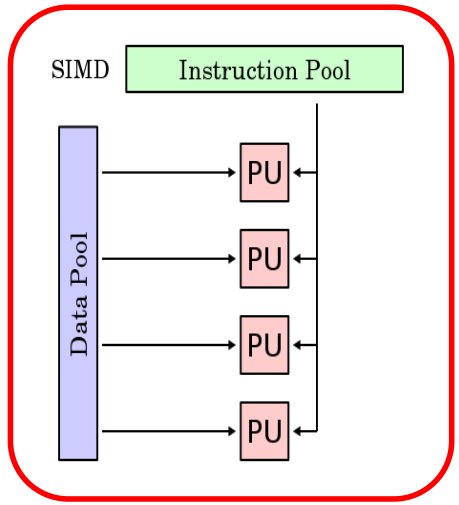
Images from Wikipedia, user Cburn

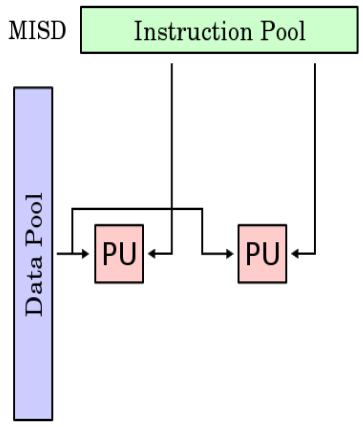
SA 3.0

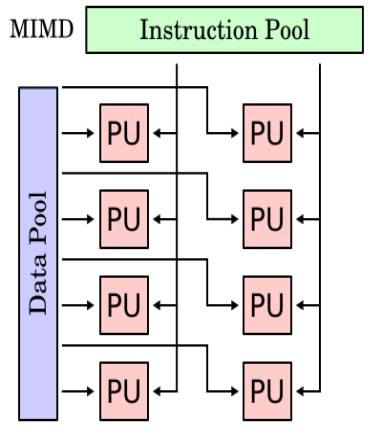
Single instruction, multiple data

- Traditional vector mindset:
  - Each instruction is executed after the other
  - Each instruction operates on multiple data elements simultaneously
  - The way vectorized MATLAB programs often are written
- Example:
  - c[i] = a[i] + b[i] i=0…N
  - a, b, and c are vectors of fixed length (typically 2, 4, 8, 16, or 32)







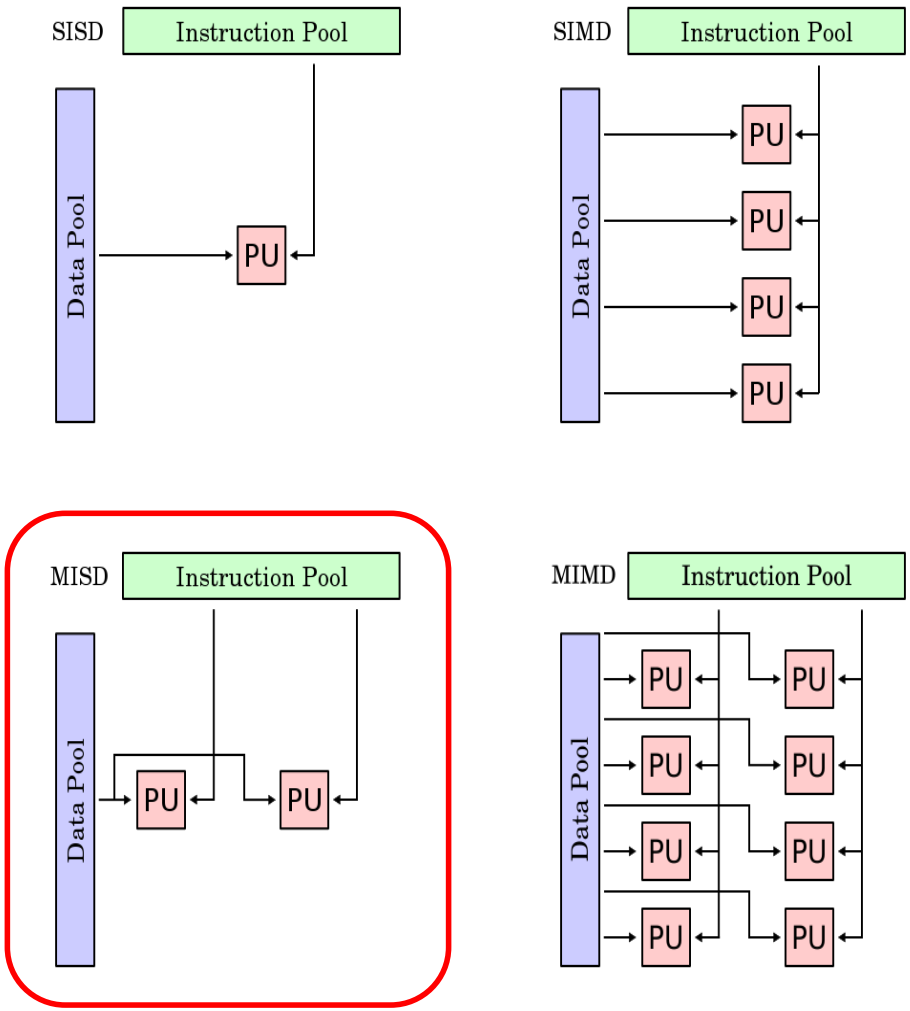


Images from Wikipedia, user Cburn

SA 3.0

Multiple instruction, single data

- Only for special cases:
  - Multiple instructions are executed simultaneously
  - Each instruction operates on a single data element
  - Used e.g., for fault tolerance, or pipelined algorithms implemented on FPGAs
- Example (naive detection of catastrophic cancellation):
  - PU1: z1 = x*x – y*y
    PU2: z2 = (x-y) * (x+y)
    if (z1 – z2 > eps) { ... }

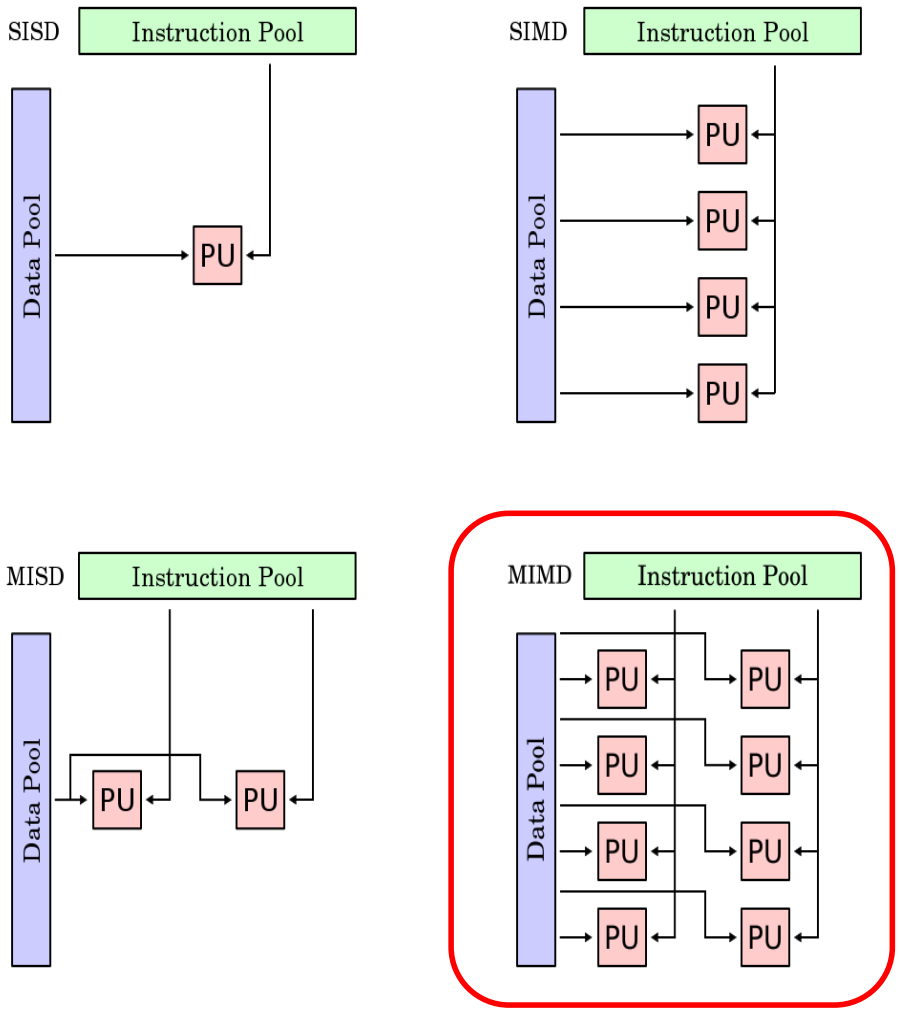


Images from Wikipedia, user Cburn

SA 3.0

Multiple instruction, multiple data

- Traditional cluster computer
  - Multiple instructions are executed simultaneously
  - Each instruction operates on multiple data elements simultaneously
  - Typical execution pattern used in task-parallel computing
- Example:
  - PU1: c = a + b
    PU2: z = (x-y) * (x+y)
    variables can also vectors of fixed length (se SIMD)

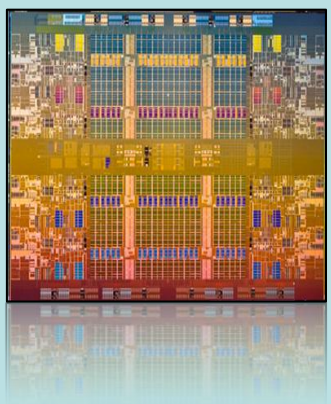


Images from Wikipedia, user Cburn

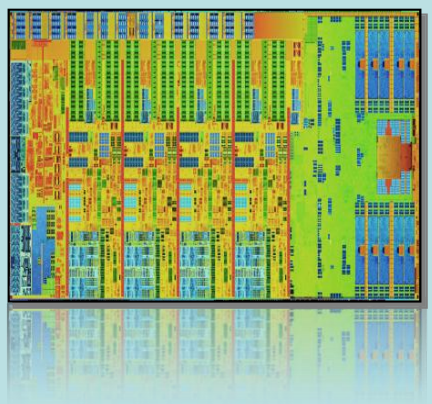
SA 3.0

Multi- and many-core processor designs

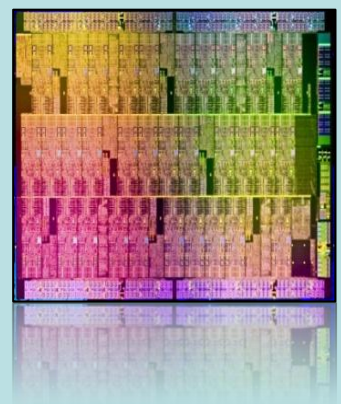
- 6-60 processors per chip
- 8 to 32-wide SIMD instructions
- Combines both SISD, SIMD, and MIMD on a single chip
- Heterogeneous cores (e.g., CPU+GPU on single chip)



Multi-core CPUs:
x86, SPARC, Power 7



Heterogeneous chips:
Intel Haswell, AMD APU

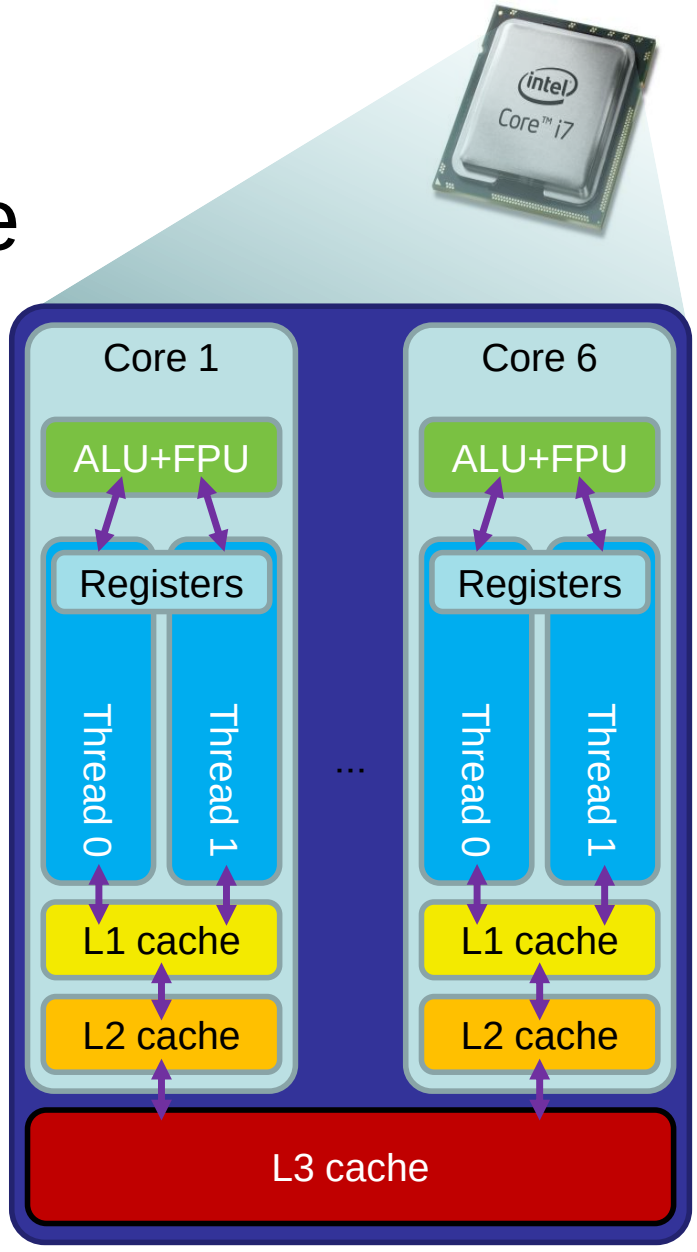


Accelerators:
GPUs, Xeon Phi

Multi-core CPU architecture

- A single core
  - L1 and L2 caches
  - 8-wide SIMD units (AVX, single precision)
  - 2-way Hyper-threading (hardware threads)
    When thread 0 is waiting for data,
    thread 1 is given access to SIMD units
  - Most transistors used for cache and logic

- Optimal number of FLOPS per clock cycle:
  - 8x: 8-way SIMD
  - 6x: 6 cores
  - 2x: Dual issue (fused mul-add / two ports)
  - Sum: 96!

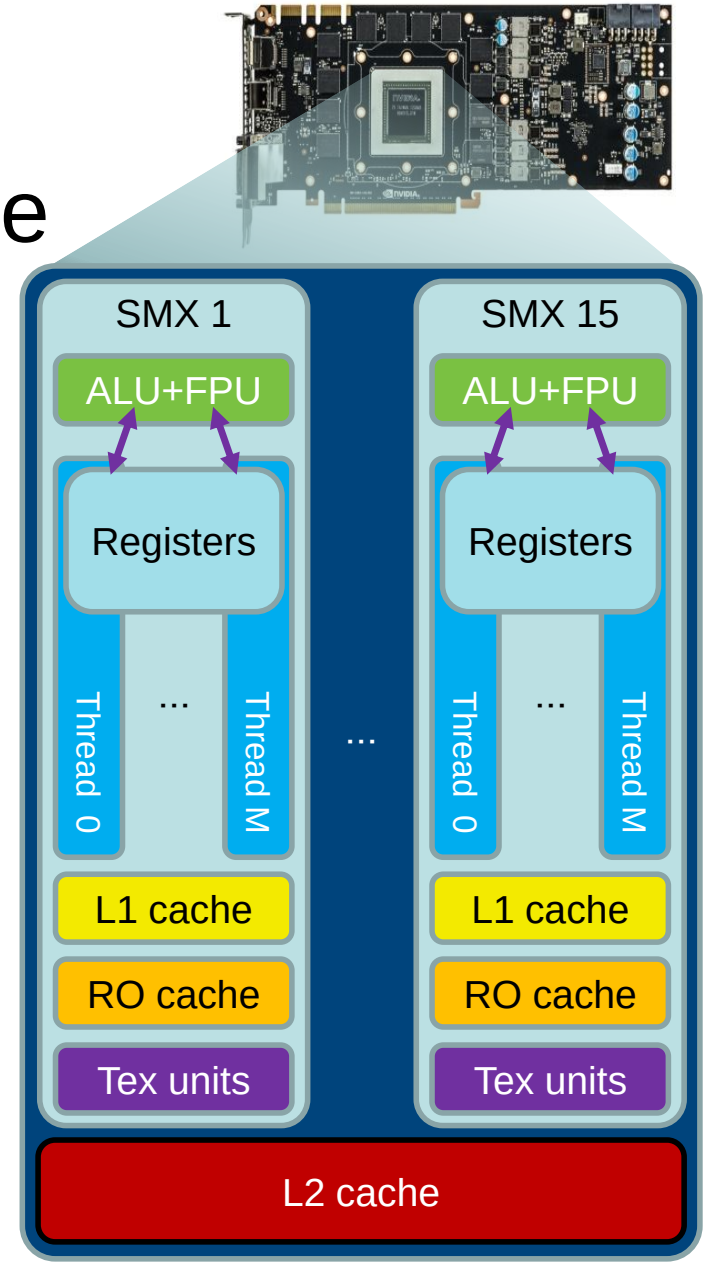


Simplified schematic of CPU design

Many-core GPU architecture

- A single core (Called streaming multiprocessor)
  - L1 cache, Read only cache, texture units
  - Six 32-wide SIMD units (192 total, single precision)
  - Up-to 64 warps simultaneously (hardware warps)
    Like hyper-threading, but a warp is 32-wide SIMD
  - Most transistors used for floating point

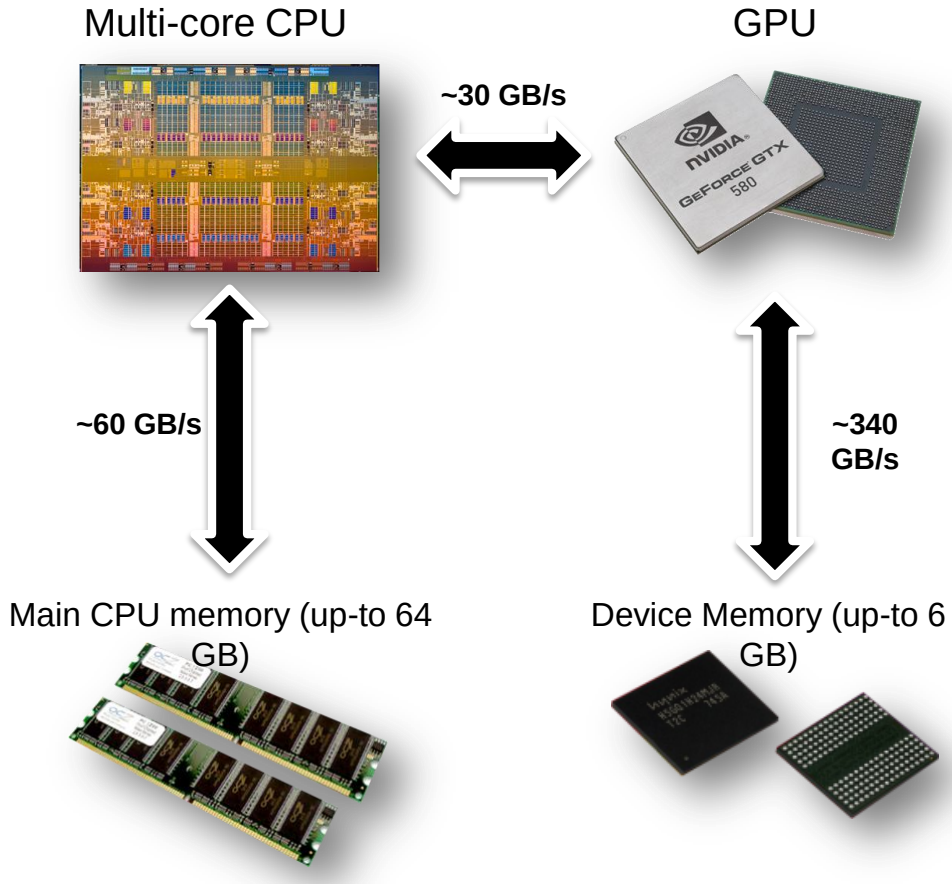
- Optimal number of FLOPS per clock cycle:
  - 32x: 32-way SIMD
  - 2x: Fused multiply add
  - 6x: Six SIMD units per core
  - 15x: 15 cores
  - Sum: 5760!



Simplified schematic of GPU design

Heterogeneous Architectures

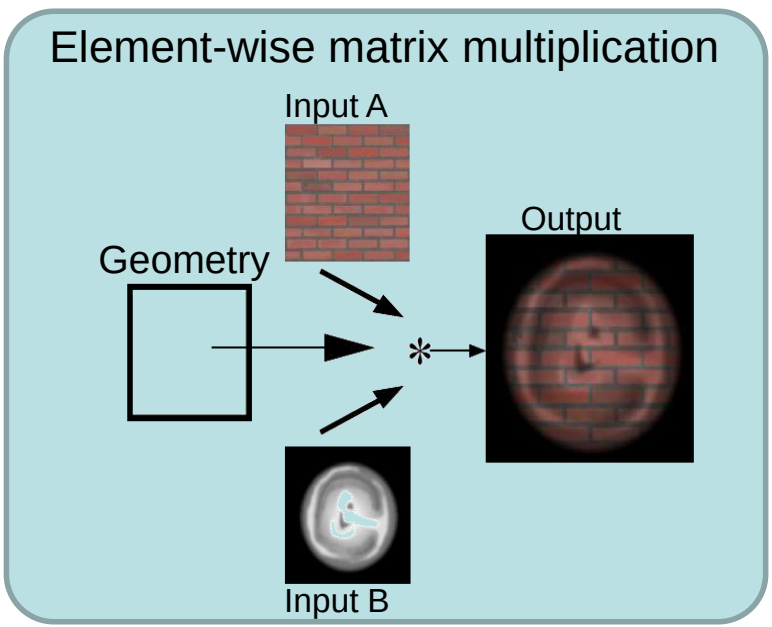
- Discrete GPUs are connected to the CPU via the PCI-express bus
  - Slow: 15.75 GB/s each direction
  - On-chip GPUs use main memory as graphics memory
- Device memory is limited but fast
  - Typically up-to 6 GB
  - Up-to 340 GB/s!
  - Fixed size, and cannot be expanded with new dimm's (like CPUs)

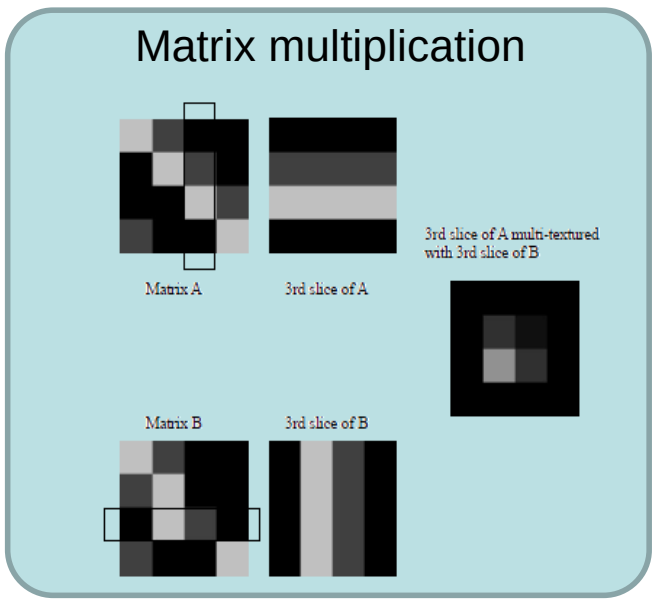


Programming GPUs

Early Programming of GPUs

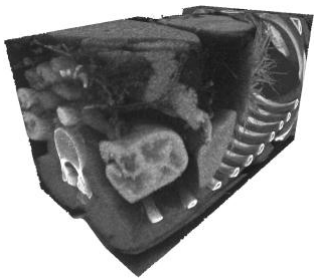
- GPUs were first programmed using OpenGL and other graphics languages
  - Mathematics were written as operations on graphical primitives
  - Extremely cumbersome and error prone
  - Showed that the GPU was capable of outperforming the CPU



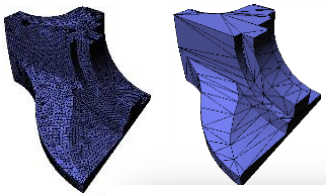


[1] Fast matrix multiplies using graphics hardware, Larsen and McAllister, 2001

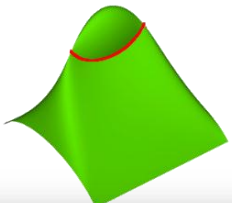
Early GPU Research at SINTEF

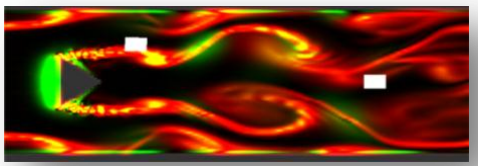


Registration of medical data (~20x)

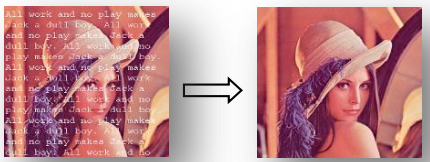


Prepa

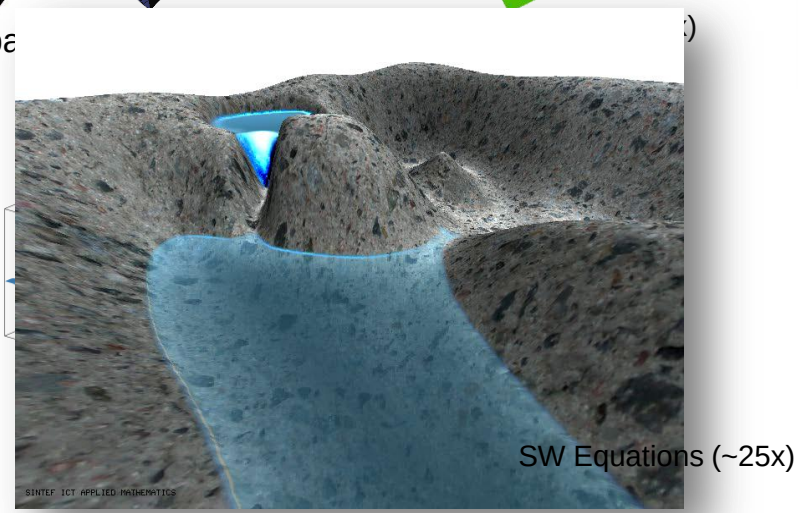




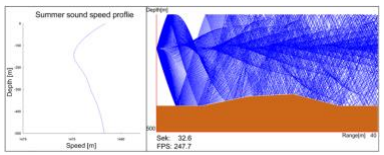
Fluid dynamics and FSI (Navier-Stokes)



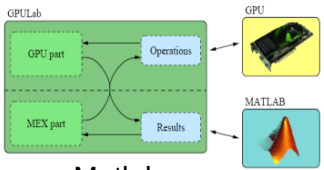
Inpainting (~400x matlab code)



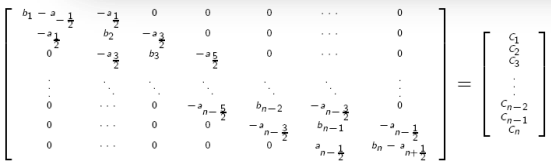
SW Equations (~25x)



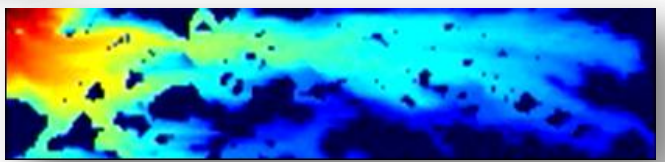
Marine aqoustics (~20x)



Matlab Interface

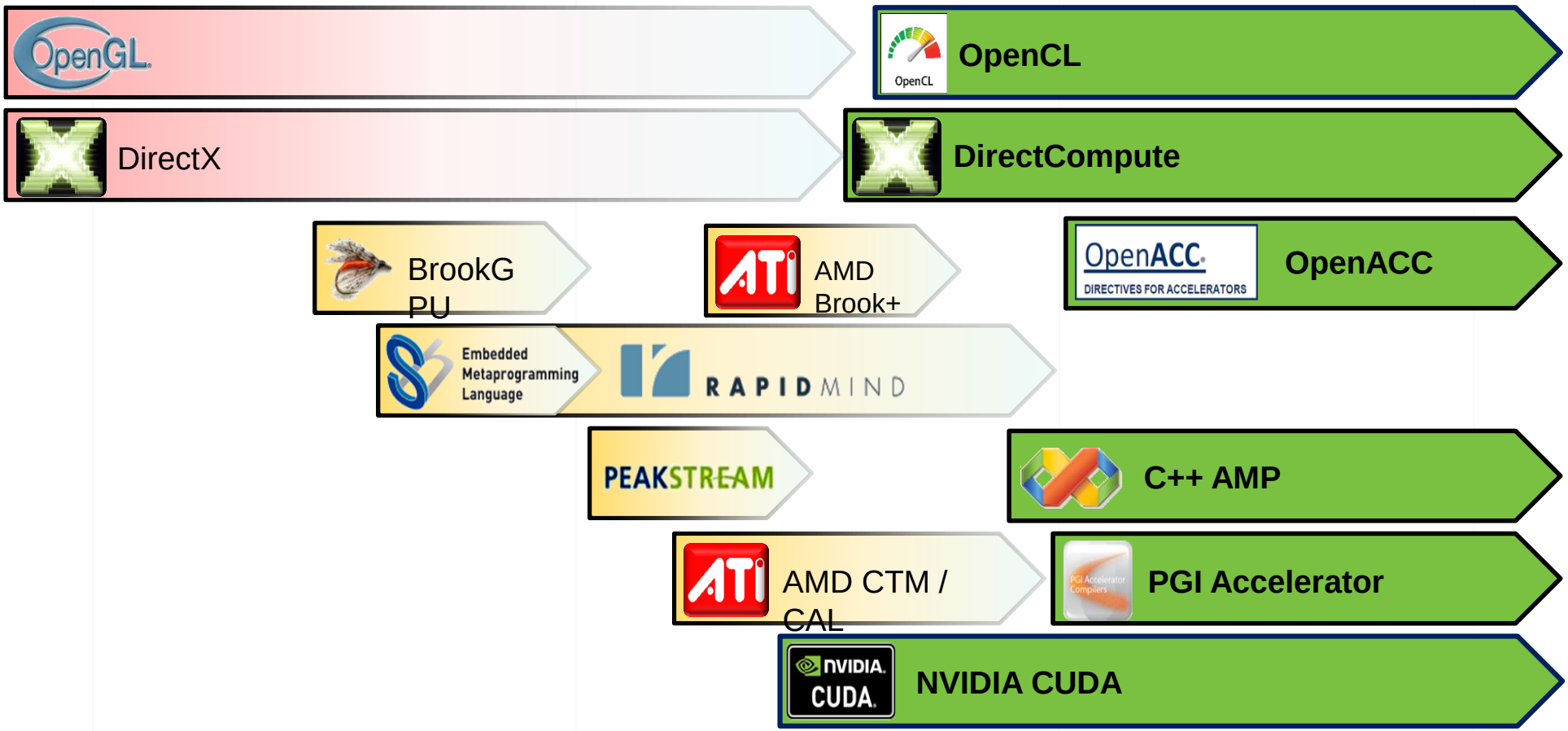


Linear algebra



Water injection in a fluvial reservoir (20x)

GPU Programming Languages



Computing with CUDA

- We will focus on CUDA, as it has the most mature development ecosystem
  - Released by NVIDIA in 2007
  - Enables programming GPUs using a C-like language
  - Essentially C / C++ with some additional syntax for executing a function in parallel on the GPU

- OpenCL is a very good alternative that also runs on non-NVIDIA hardware (Intel Xeon Phi, AMD GPUs, CPUs)
  - Equivalent to CUDA, but slightly more cumbersome.

OpenCL

- For high-level development, languages like OpenACC (pragma based) or C++ AMP (extension to C++) exist
  - Typicall works well for toy problems, and not so well for complex algorithms

Adding two matrices in CUDA 1/2

- We want to add two matrices, a and b, and store the result in c.

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

- For best performance, loop through one row at a time (sequential memory access pattern)

C++ on CPU

```
void addFunctionCPU(float* c, float* a, float* b,
                    unsigned int cols, unsigned int rows) {
     for (unsigned int j=0; j<rows; ++j) {
          for (unsigned int i=0; i<cols; ++i) {
               unsigned int k = j*cols + i;
               c[k] = a[k] + b[k];
          }
     }
}
```

Matrix from Wikipedia: Matrix addition

Adding two matrices in CUDA 2/2

```
__global__ void addMatricesKernel(float* c, float* a, float* b,
                                  unsigned int cols, unsigned int rows) {
    //Indexing calculations
    unsigned int global_x = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int global_y = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int k = global_y*cols + global_x;

    //Actual addition
    c[k] = a[k] + b[k];
}

void addFunctionGPU(float* c, float* a, float* b,
                    unsigned int cols, unsigned int rows) {
    dim3 block(8, 8);
    dim3 grid(cols/8, rows/8);
    ... //More code here: Allocate data on GPU, copy CPU data to GPU
    addMatricesKernel<<<grid, block>>>(gpu_c, gpu_a, gpu_b, cols, rows);
    ...  //More code here: Download result from GPU to CPU
}
```

GPU function

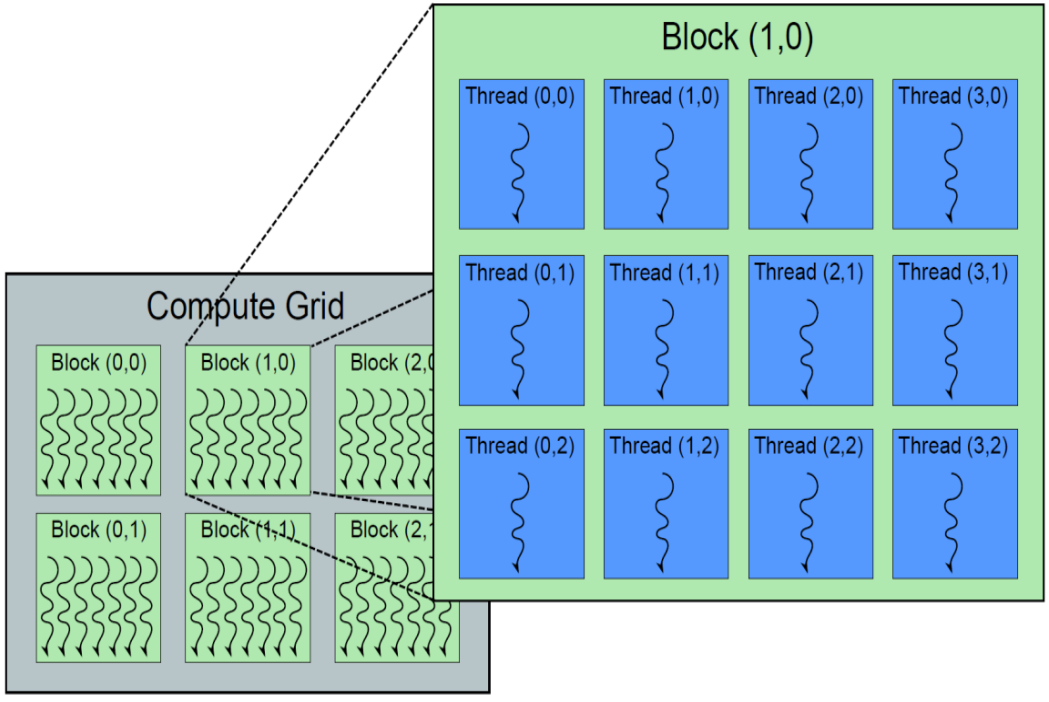
Indices

Run on GPU

Implicit double for loop
for (int blockIdx.x = 0;
blockIdx.x < grid.x;
blockIdx.x) { …

Grids and blocks in CUDA

- Two-layered parallelism
  - A block consists of threads: Threads within the same block can cooperate and communicate
  - A grid consists of blocks: All blocks run independently.
  - Blocks and grid can be 1D, 2D, and 3D

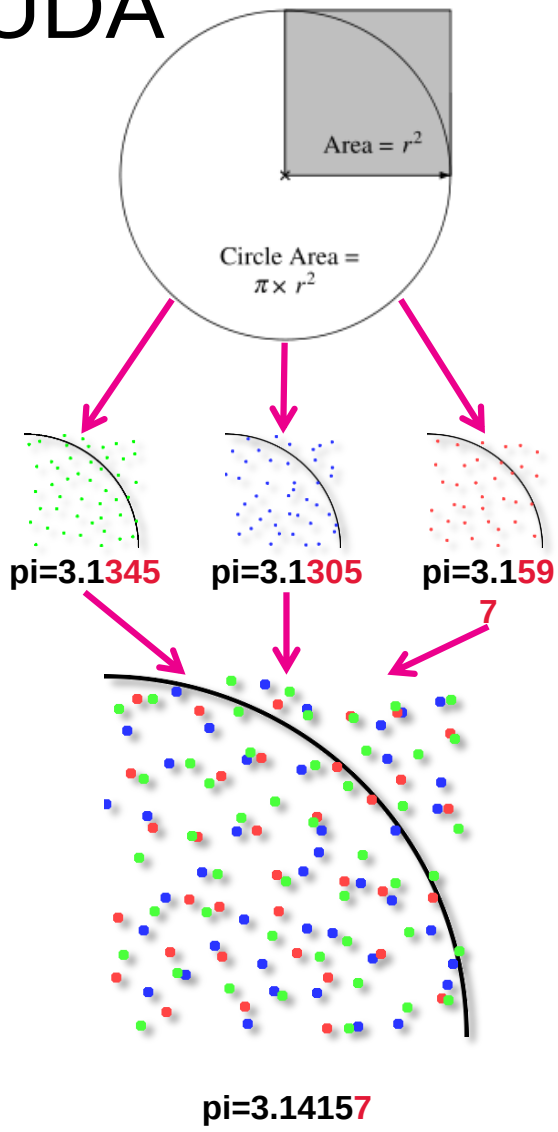


- Global synchronization and communication is only possible between kernel launches
  - Really expensive, and should be avoided if possible

Example: Computing π with CUDA

- Algorithm:
  1. Sample random points within a quadrant
  2. Compute distance from point to origin
  3. If distance less than r, point is inside circle
  4. Estimate π as 4 #points inside / #points outside

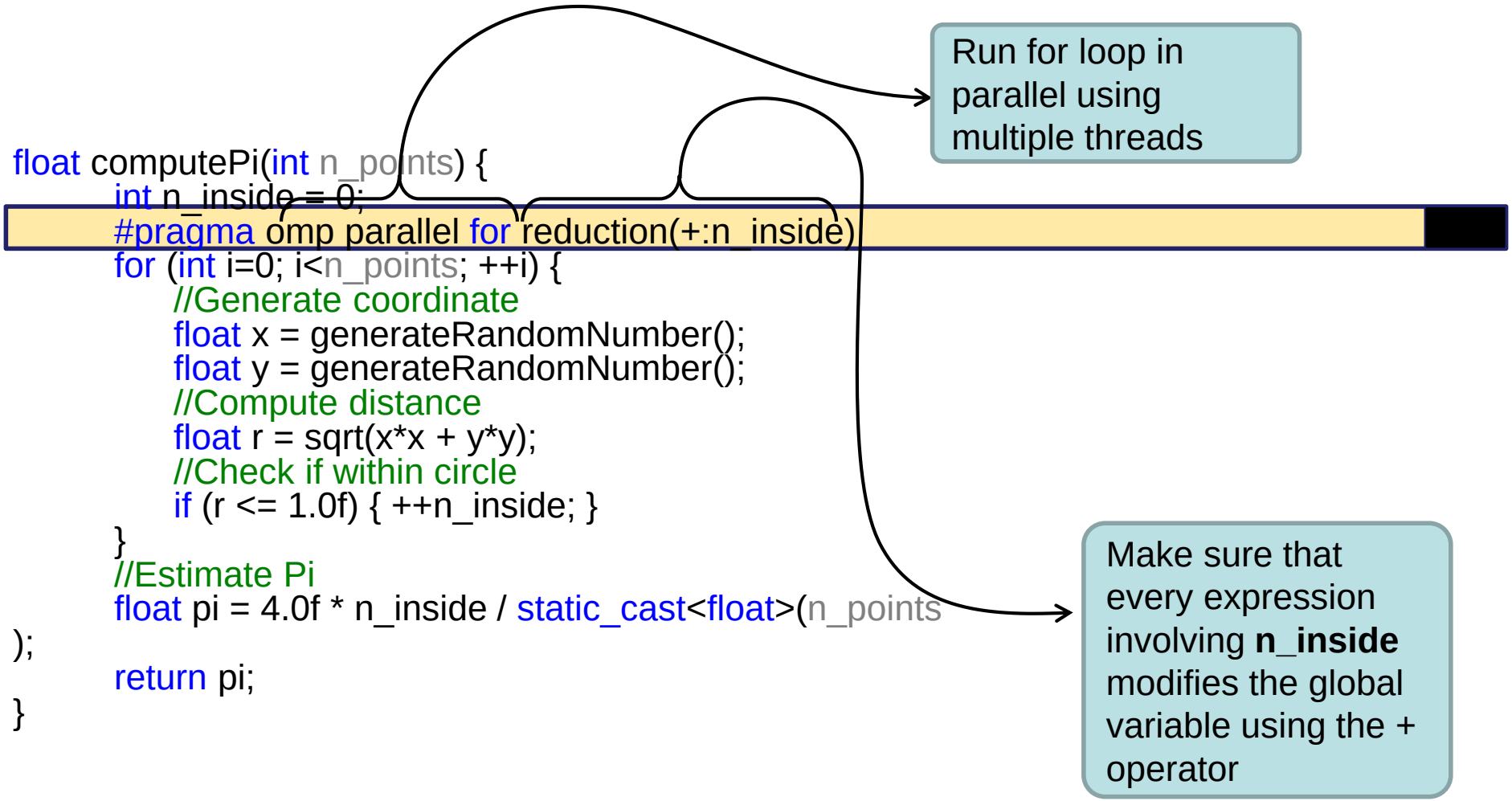
- Remember: The algorithms serves as an example:
  it's far more efficient to estimate π as 22/7, or 355/113☺



Serial CPU code (C/C++)

```
float computePi(int n_points) {
    int n_inside = 0;
    for (int i=0; i<n_points; ++i) {
        //Generate coordinate                                   1
        float x = generateRandomNumber();
        float y = generateRandomNumber();
        //Compute distance                                      2&3
        float r = sqrt(x*x + y*y);
        //Check if within circle
        if (r < 1.0f) { ++n_inside; }
    }
    //Estimate Pi
    float pi = 4.0f * n_inside / static_cast<float>(n_points);  4
    return pi;
}
```

Parallel CPU code (C/C++ with OpenMP)




```
float computePi(int n_points) {
        int n_inside = 0;
        #pragma omp parallel for reduction(+:n_inside)
        for (int i=0; i<n_points; ++i) {
                //Generate coordinate
                float x = generateRandomNumber();
                float y = generateRandomNumber();
                //Compute distance
                float r = sqrt(x*x + y*y);
                //Check if within circle
                if (r <= 1.0f) { ++n_inside; }
        }
        //Estimate Pi
        float pi = 4.0f * n_inside / static_cast<float>(n_points
);
        return pi;
}
```

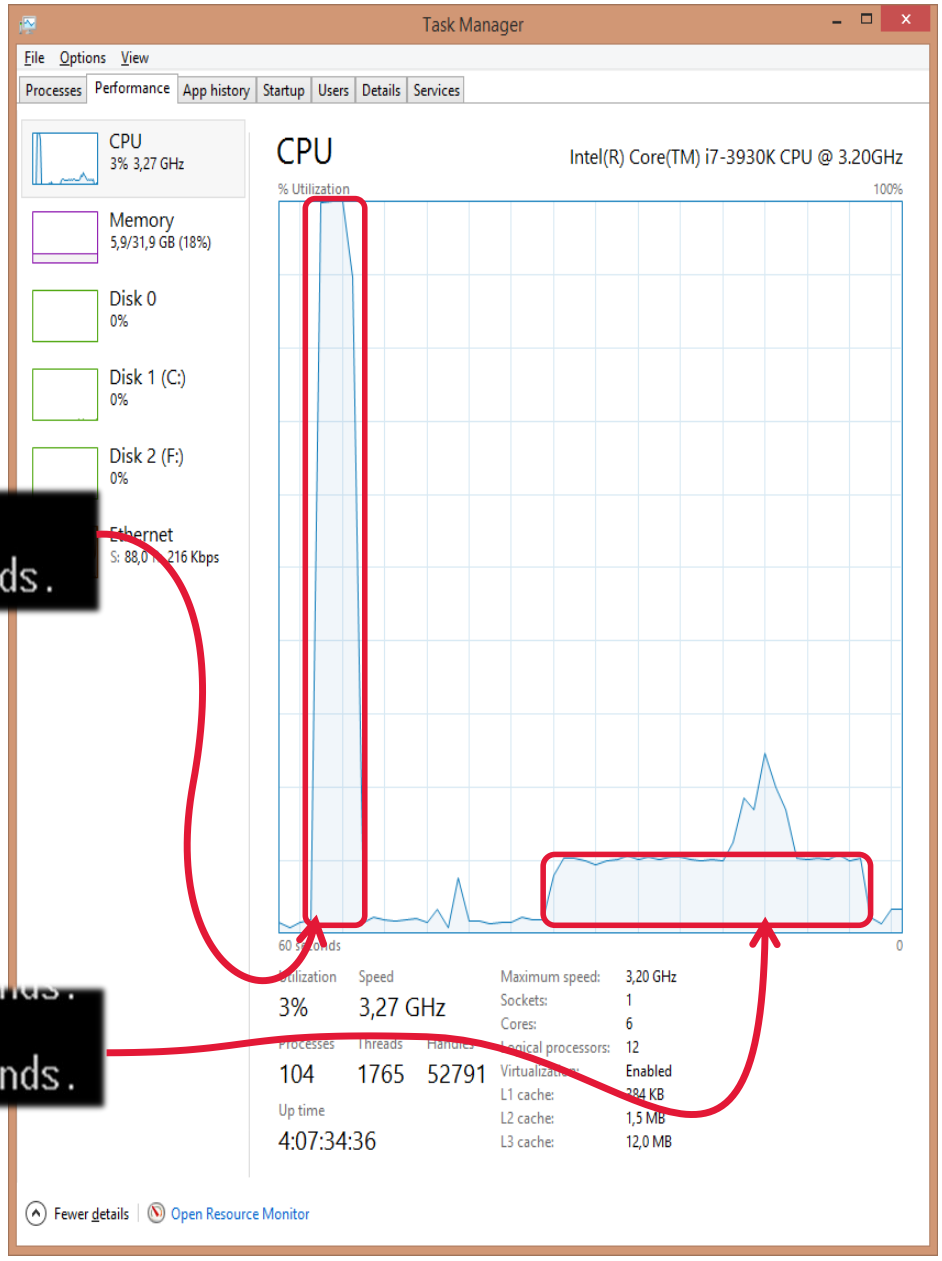
Performance

- Parallel: 3.8 seconds @ 100% CPU

```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141476 in 3.799772 seconds.
```

- Serial: 30 seconds @ 10% CPU

```
Please enter number of iterations: 1000000000
Estimated Pi to be: 3.141495 in 29.883573 seconds.
```



Parallel GPU version 1 (CUDA) 1/3

GPU function

```
__global__ void computePiKernel1(unsigned int* output) {
        //Generate coordinate
        float x = generateRandomNumber();
        float y = generateRandomNumber();

        //Compute radius
        float r = sqrt(x*x + y*y);

        //Check if within circle
        if (r <= 1.0f) {
                output[blockIdx.x] = 1;
        } else {
                output[blockIdx.x] = 0;
        }
}
```

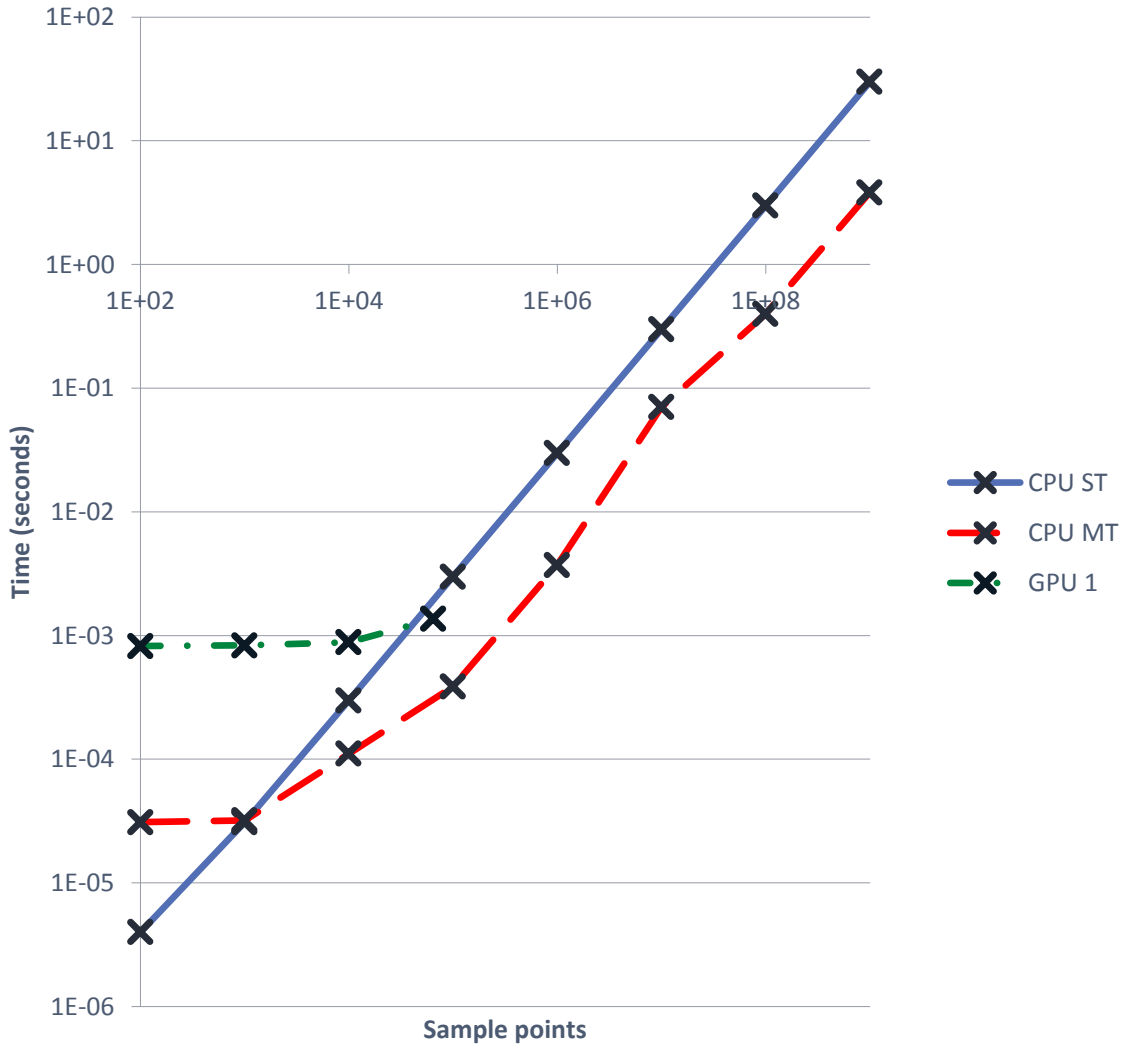
*Random numbers on GPUs can be a slightly tricky, see cuRAND for more information

Parallel GPU version 1 (CUDA) 2/3

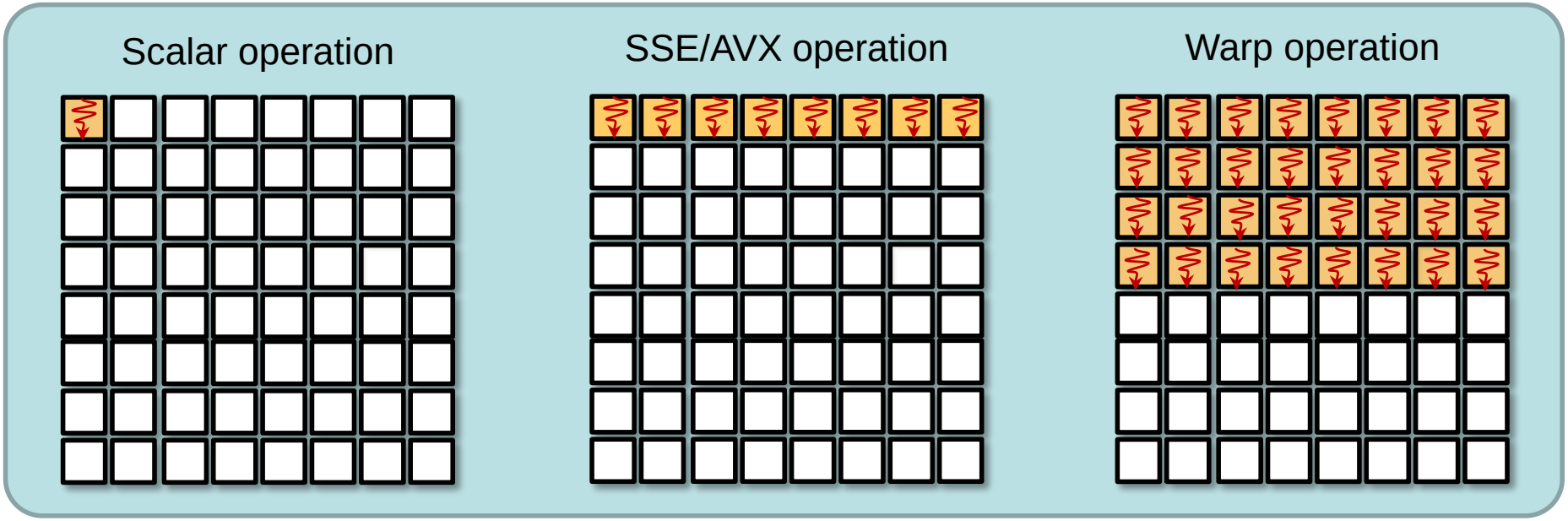
```
float computePi(int n_points) {
      dim3 grid = dim3(n_points, 1, 1);
      dim3 block = dim3(1, 1, 1);

      //Allocate data on graphics card for output
      cudaMalloc((void**)&gpu_data, gpu_data_size);

      //Execute function on GPU (“lauch the kernel”)
      computePiKernel1<<<grid, block>>>(gpu_data);

      //Copy results from GPU to CPU
      cudaMemcpy(&cpu_data[0], gpu_data, gpu_data_size, cudaMemcpyDeviceToHost);

      //Estimate Pi
      for (int i=0; i<cpu_data.size(); ++i) {
            n_inside += cpu_data[i];
      }
      return pi = 4.0f * n_inside / n_points;
}
```

Parallel GPU version 1 (CUDA) 3/3

- Unable to run more than 65535 sample points
- Barely faster than single threaded CPU version for largest size!
- Kernel launch overhead appears to dominate runtime
- The fit between algorithm and architecture is poor:
  - 1 thread per block: Utilizes at most 1/32 of computational power.

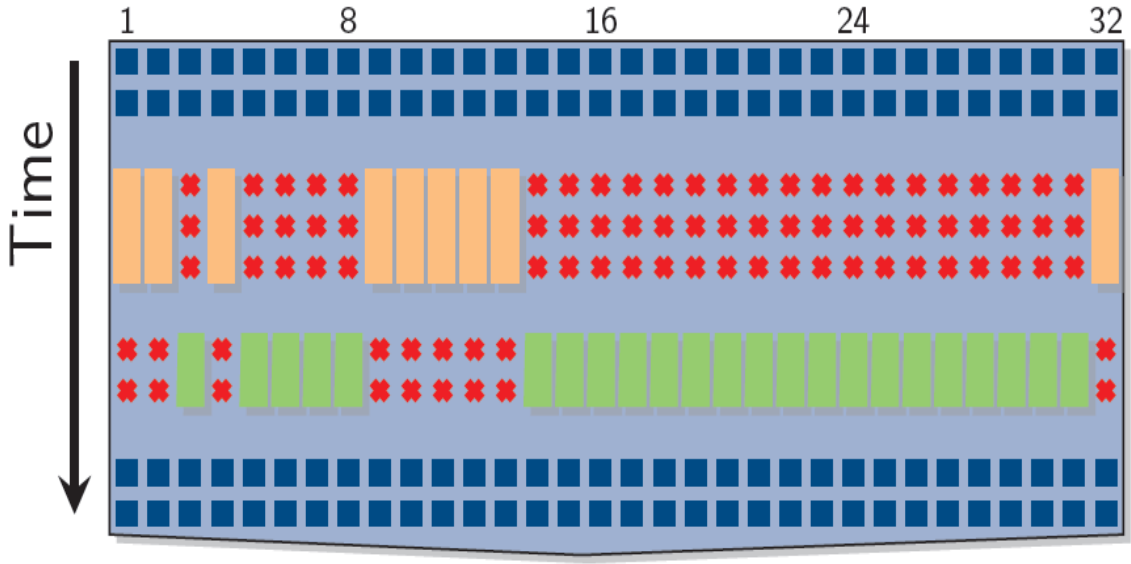


GPU Vector Execution Model



- **CPU scalar:** 1 thread, 1 operand on 1 data element
- **CPU SSE/AVX:** 1 thread, 1 operand on 2-8 data elements
- **GPU Warp:** 32 threads, 32 operands on 32 data elements
  - Exposed as **individual threads**
  - Actually runs the **same instruction**
  - Divergence implies **serialization and masking**

Serialization and masking




```
// Non-divergent code
if( x > 0 ) {
    y = pow(x, exp);
    y *= Ks;
    z = y + Ka;
} else {
    x = 0;
    z = Ka;
}
// Non-divergent code
```

Hardware automatically serializes and masks divergent code flow:

- Execution time is the sum of all branches taken
- Programmer is relieved of fiddling with element masks (which is necessary for SSE/AVX)
- Worst case 1/32 performance
- Important to **minimize divergent code flow <u>within warps</u>**!
  - Move conditionals into data, use min, max, conditional moves

Parallel GPU version 2 (CUDA) 1/2

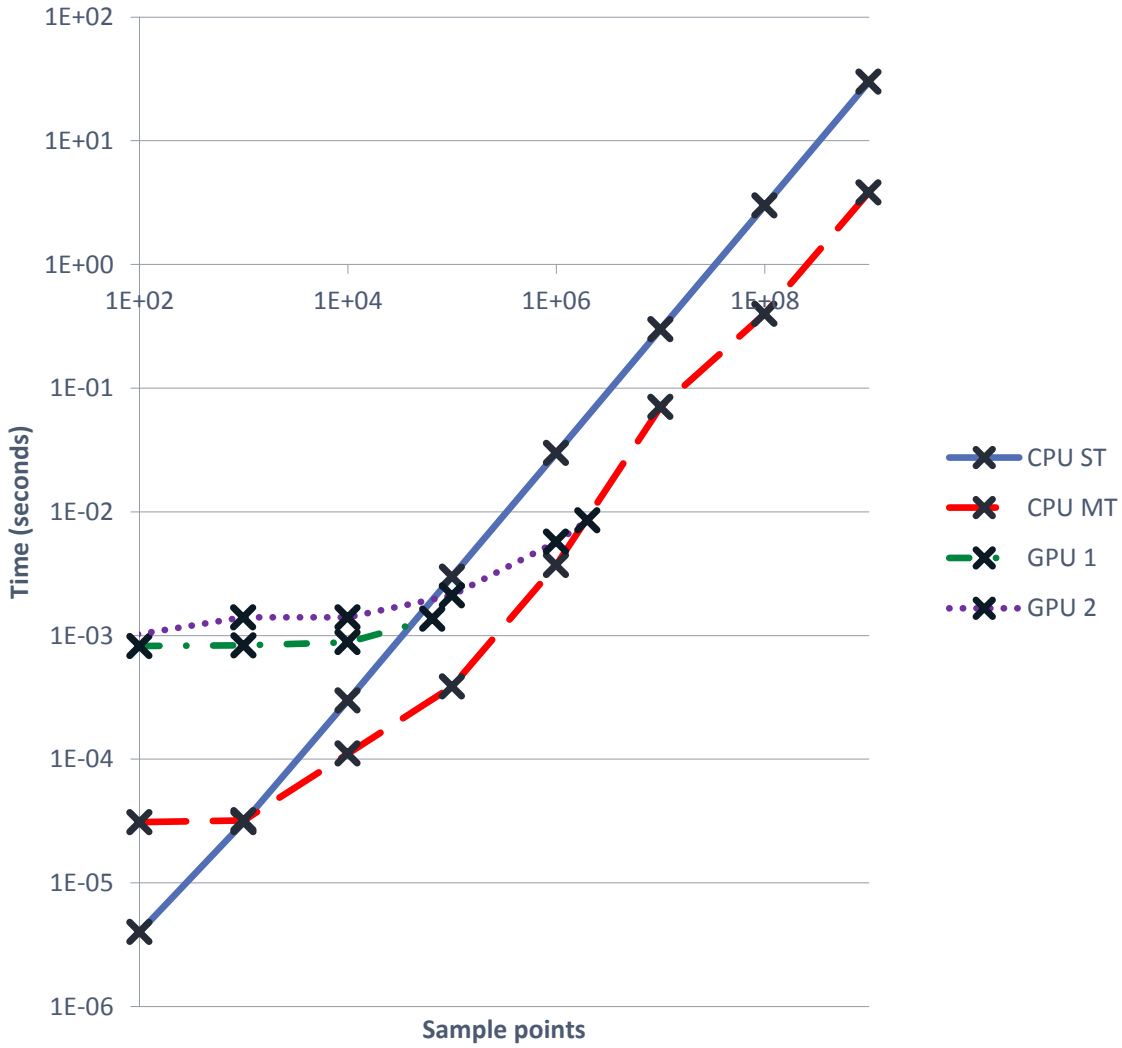
```
__global__ void computePiKernel2(unsigned int* output) {
        //Generate coordinate
        float x = generateRandomNumber();
        float y = generateRandomNumber();

        //Compute radius
        float r = sqrt(x*x + y*y);

        //Check if within circle
        if (r <= 1.0f) {
                output[blockIdx.x*blockDim.x + threadIdx.x] = 1;
        } else {
                output[blockIdx.x*blockDim.x + threadIdx.x] = 0;
        }
}

float computePi(int n_points) {
        dim3 grid = dim3(n_points/32, 1, 1);
        dim3 block = dim3(32, 1, 1);
        …
        //Execute function on GPU ("lauch the kernel")
        computePiKernel1<<<grid, block>>>(gpu_data);
        …
}
```

New indexing

32 threads per block

Parallel GPU version 2 (CUDA) 2/2

- Unable to run more than 32*65535 sample points
- Works well with 32-wide SIMD
- Able to keep up with multi-threaded version at maximum size!
- We perform roughly 16 operations per 4 bytes written (1 int): memory bound kernel!
  Optimal is 60 operations!



Parallel GPU version 3 (CUDA) 1/4

```
__global__ void computePiKernel3(unsigned int* output, unsigned int seed) {
__shared__ int inside[32];

//Generate coordinate
//Compute radius
…

//Check if within circle
if (r <= 1.0f) {
        inside[threadIdx.x] = 1;
} else {
        inside[threadIdx.x] = 0;
}

… //Use shared memory reduction to find number of inside per block
```

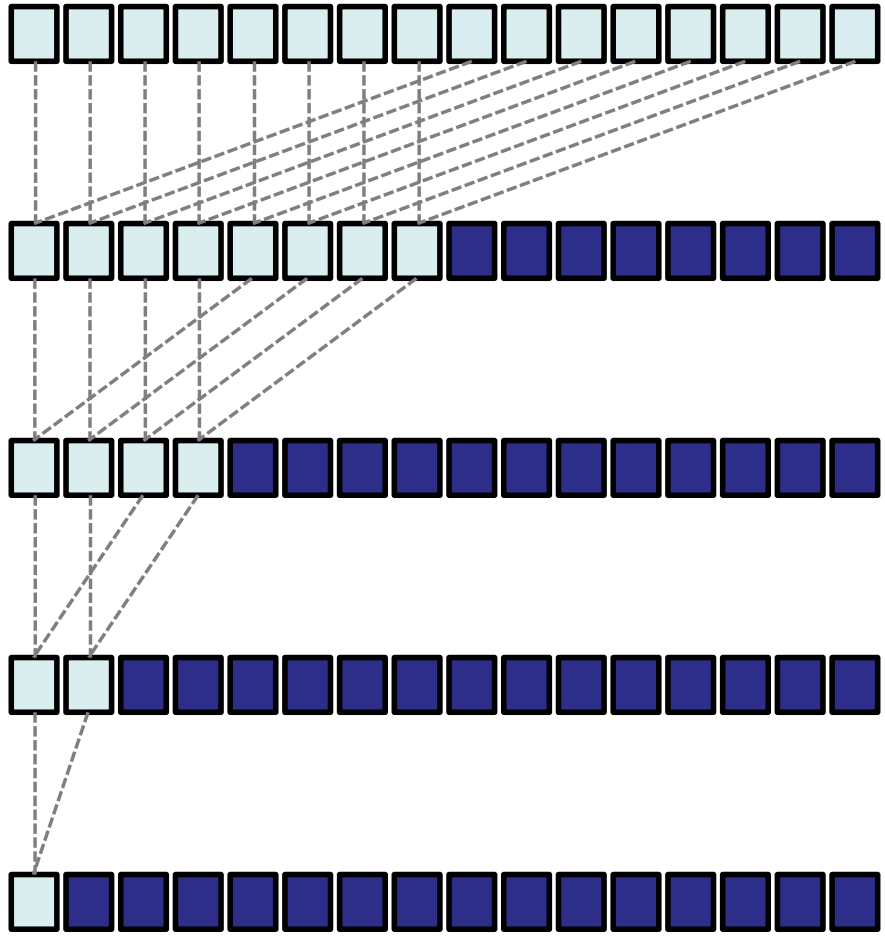
Shared memory: a kind of “programmable cache”
We have 32 threads: One entry per thread

Parallel GPU version 3 (CUDA) 2/4

```
… //Continued from previous slide

//Use shared memory reduction to find number of inside per block
//Remember: 32 threads is one warp, which execute synchronously
if (threadIdx.x < 16) {
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+16];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+8];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+4];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+2];
        p[threadIdx.x] = p[threadIdx.x] + p[threadIdx.x+1];
}

if (threadIdx.x == 0) {
        output[blockIdx.x] = inside[threadIdx.x];
}
}
```

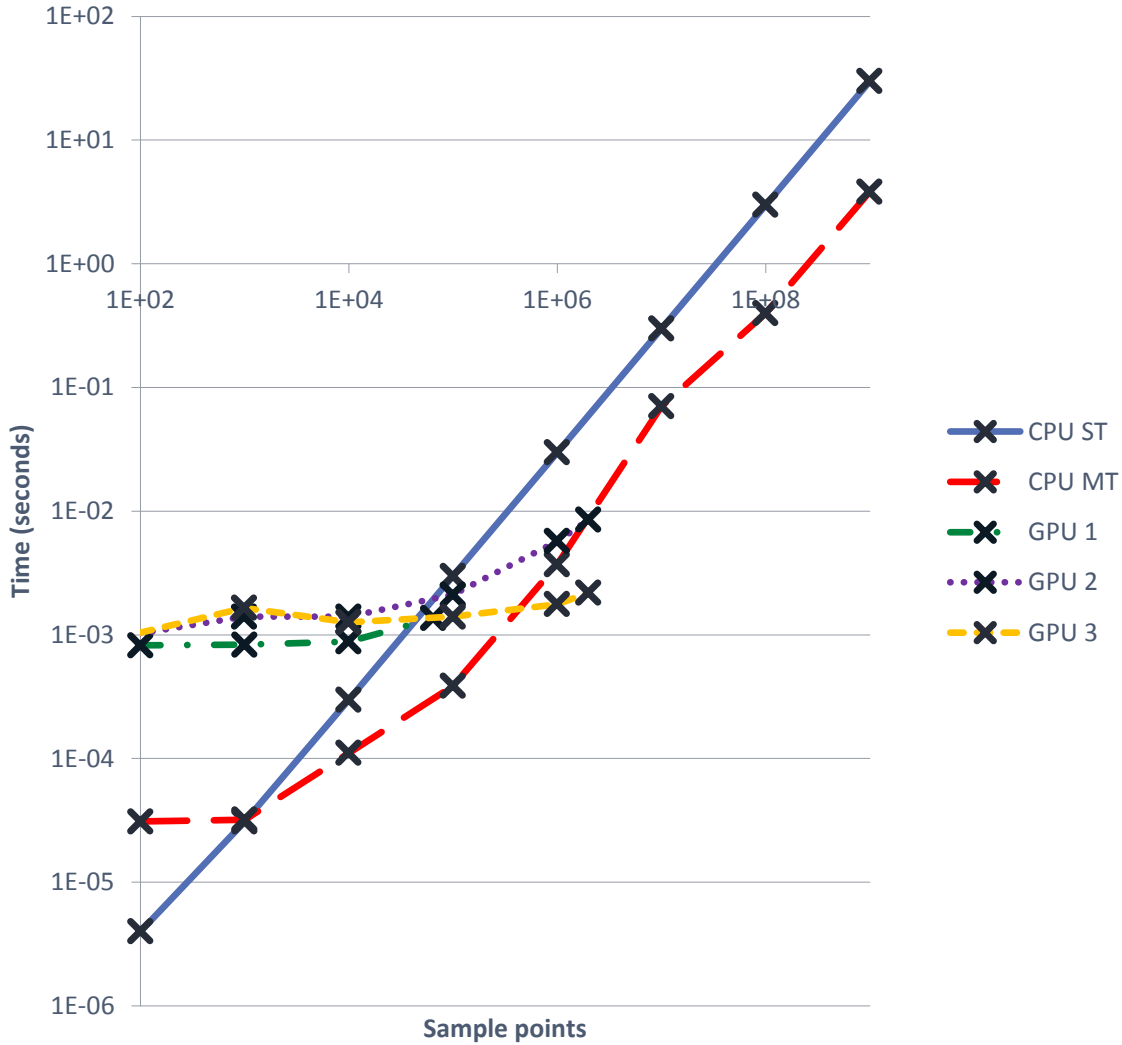
Parallel GPU version 3 (CUDA) 3/4

- Shared memory is a kind of programmable cache
  - Fast to access (just slightly slower than registers)
  - Programmers responsibility to move data into shared memory
  - All threads in one block can see the same shared memory
  - Often used for communication between threads
- Sum all elements in shared memory using shared memory reduction



Parallel GPU version 3 (CUDA) 4/4

- Memory bandwidth use reduced by factor 32!
- Good speed-up over multithreaded CPU!
- Maximum size is still limited to 65535*32.
- Two ways of increasing size:
  - Increase number of threads
  - Make each thread do more work

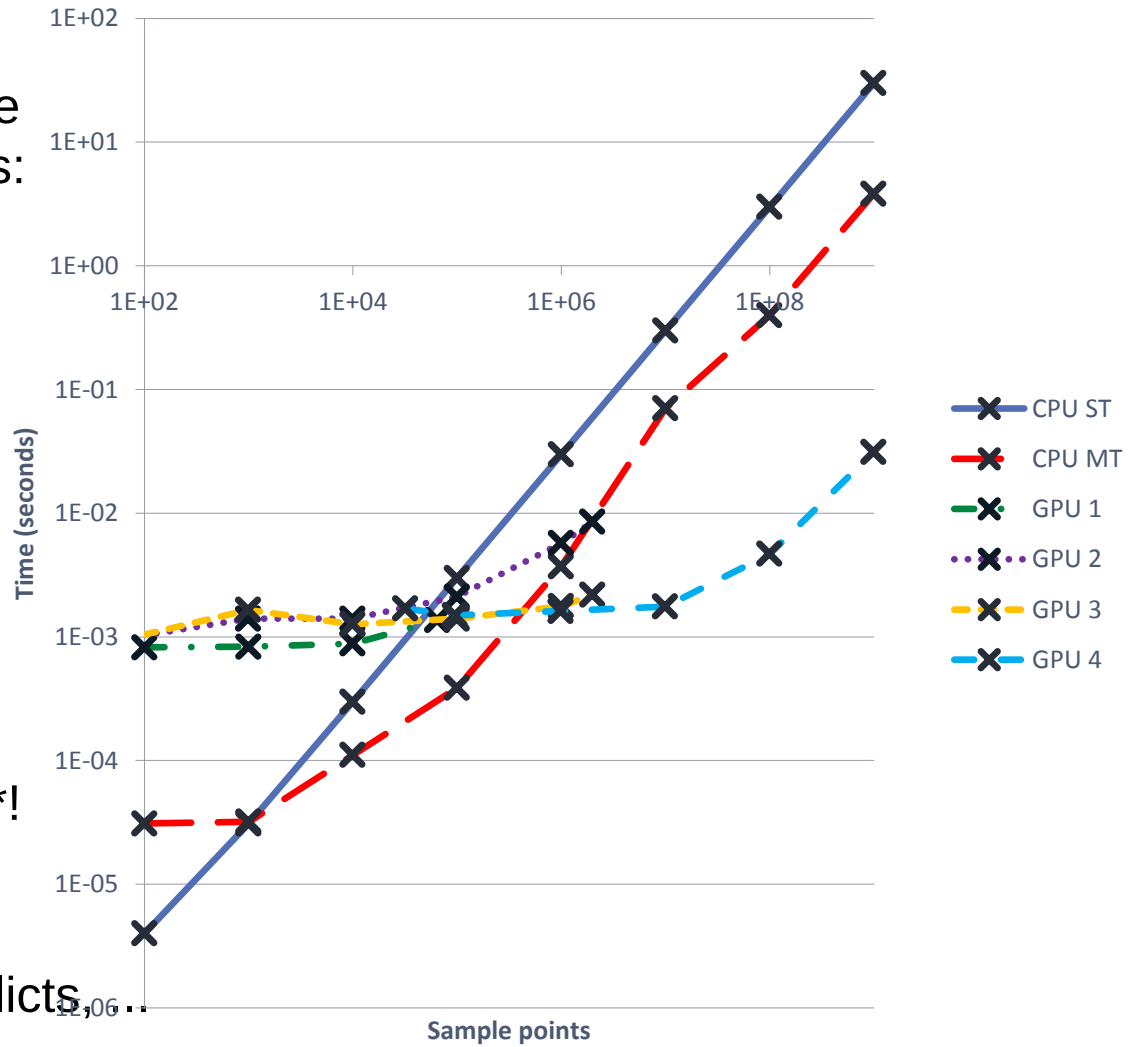


Parallel GPU version 4 (CUDA) 1/2

```
__global__ void computePiKernel4(unsigned int* output) {
        int n_inside = 0;

         //Shared memory: All threads can access this
        __shared__ int inside[32];
        inside[threadIdx.x] = 0;

        for (unsigned int i=0; i<iters_per_thread; ++i) {
                //Generate coordinate
                //Compute radius
                //Check if within circle
                if (r <= 1.0f) { ++inside[threadIdx.x]; }
        }

        //Communicate with other threads to find sum per block
        //Write out to main GPU memory
}
```

Parallel GPU version 4 (CUDA) 2/2

- Overheads appears to dominate runtime up-to 10.000.000 points:
  - Memory allocation
  - Kernel launch
  - Memory copy
- Estimated GFLOPS: ~450
  Thoretical peak: ~4000
- Things to investigate further:
  - Profile-driven development*!
  - Check number of threads, memory access patterns, instruction stalls, bank conflicts, …



*See e.g., Brodtkorb, Sætra, Hagen,
GPU Programming Strategies and Trends in GPU Computing, JPDC, 2013

Comparing performance

- Previous slide indicates speedup of
  - 100x versus OpenMP version
  - 1000x versus single threaded version
  - Theoretical performance gap is 10x: why so fast?

- Reasons why the comparison is fair:
  - Same generation CPU (Core i7 3930K) and GPU (GTX 780)
  - Code available on Github: you can test it yourself!

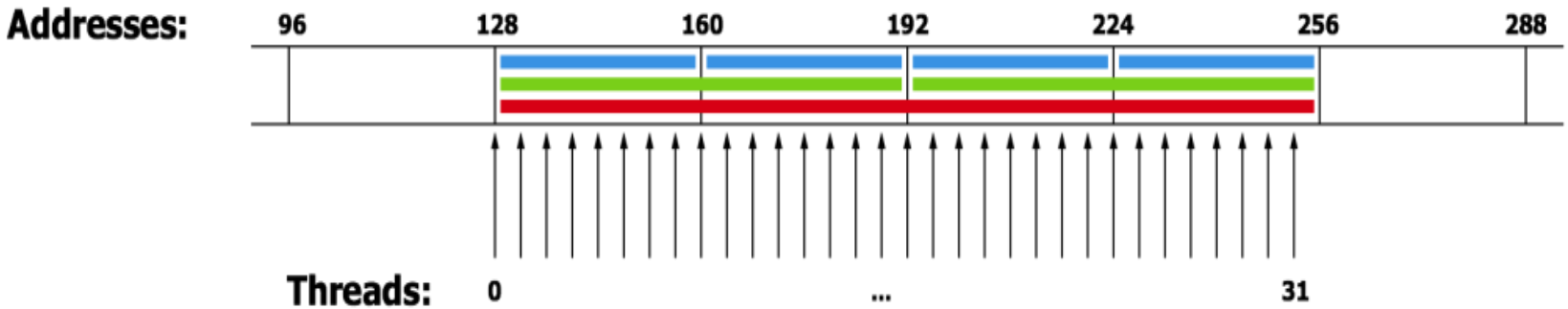
- Reasons why the comparison is unfair:
  - Optimized GPU code, unoptimized CPU code.
  - I do not show how much of CPU/GPU resources I actually use (profiling)
  - I cheat with the random function (I use a simple linear congruential generator).

Optimizing Memory Access

Memory access 1/2

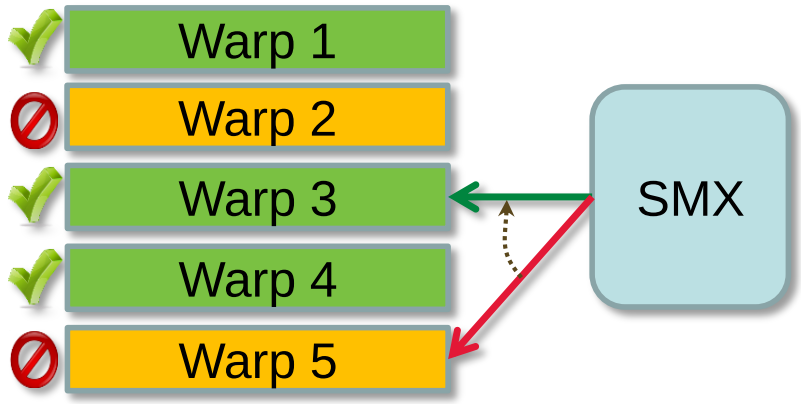
- Accessing a single memory address triggers transfer of a full *cache line* (128 bytes)
  - The smallest unit transferrable over the memory bus
  - Identical to how CPUs transfer data

- For peak performance, 32 threads should use 32 consecutive integers/floats
  - This is referred to as coalesced reads



- On modern GPUs: Possible to transfer 32 byte segments: Better fit for random access!
- Slightly more complex in reality: see CUDA Programming Guide for full set of rules

Memory access 2/2



- GPUs have high bandwidth, and high latency
  - Latencies are on the order of hundreds to thousands of clock cycles
- Massive multithreading hides latencies
  - When one warp stalls on memory request, another warp steps in and uses execution units
- Effect: Latencies are completely hidden as long as you have enough memory parallelism:
  - More than 100 simultaneous requests for full cache lines per SM (Kepler).
  - Far more for random access!

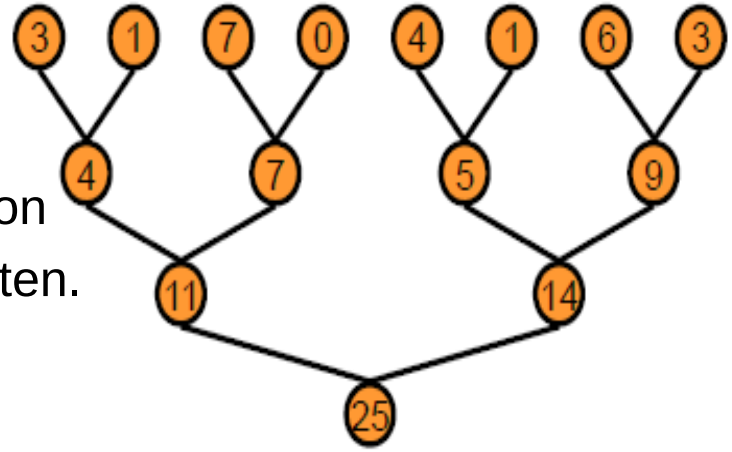
For more details, see Paulius Micikevicius, GPU Performance Analysis and Optimization, 2013

Example: Parallel reduction

- Reduction is the operation of finding a single number from a series of numbers
  - Frequently used parallel building block in parallel computing
  - We've already used it to compute π
- Examples:
  - Find minimum, maximum, average, sum
  - In general: Perform a binary operation on a set data
- CPU example:

```
//Initialize to first element
T result = data[0];

//Loop through the rest of the elements
for (int i=1; i<data.size(); ++i) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

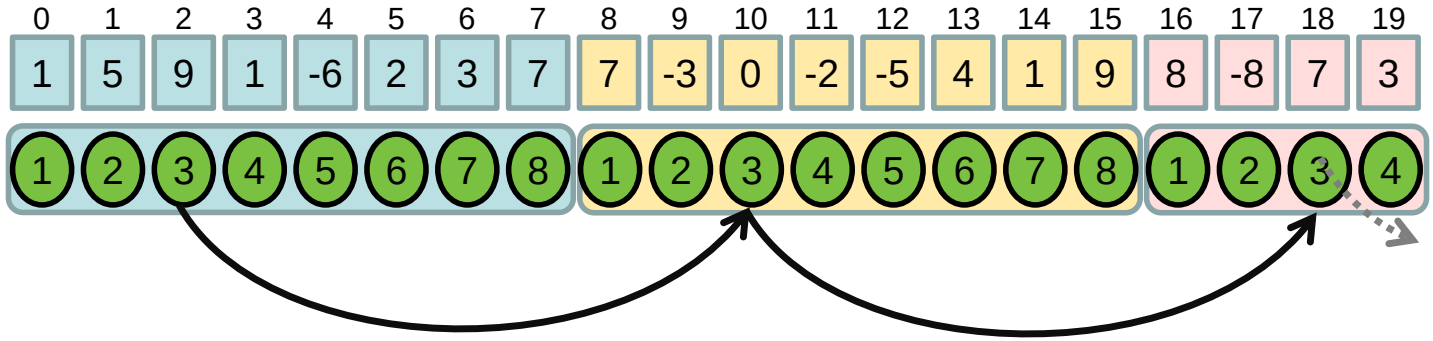
Parallel considerations



- This is a completely memory bound application
  - O(1) operation per element read and written.
  - Need to optimize for memory access!
- Classical approach: represent as a binary tree
  - log2(n) passes required to reduce n elements
  - Example: 10 passes to find maximum of 1024 elements
- General idea:
  - Use few blocks with maximum number of threads (i.e., 512 in this example)
  - *Stride* through memory until all items are read
  - Perform shared memory reduction to find single largest

Example based on Mark Harris, Optimizing parallel reduction in CUDA

Striding through data




```
for (int i=threadIdx.x; i<size; i += blockDim.x) {
        //Perform binary operator (e.g., op(a, b) = max(a, b))
        result = op(result, data[i]);
}
```

- Striding ensures perfect coalesced memory reads
- Thread 2 operates on elements 2, 10, 18, etc. for a block size of 8
- We have block size of 512: Thread 2 operates on elements 2, 514, 1026, …
- Perform "two-in-one" or "three-in-one" strides for more parallel memory requests

Shared memory reduction 1/2

- By striding through data, we efficiently reduce N/num_blocks elements to 512.
- Now the problem becomes reducing 512 elements to 1:
  lets continue the striding, but now in shared memory
- Start by reducing from 512 to 64 (notice use of __syncthreads()):

```
__syncthreads(); // Ensure all threads have reached this point

// Reduce from 512 to 256
if(tid < 256) { sdata[tid] = sdata[tid] + sdata[tid + 256]; }
__syncthreads();

// Reduce from 256 to 128
if(tid < 128) { sdata[tid] = sdata[tid] + sdata[tid + 128]; }
__syncthreads();

// Reduce from 128 to 64
if(tid < 64) { sdata[tid] = sdata[tid] + sdata[tid +  64]; }
__syncthreads();
```

Shared memory reduction 2/2

- When we have 64 elements, we can use 32 threads to perform the final reductions
- Remember that 32 threads is one warp, and execute instructions in SIMD fashion
- This means we do not need the syncthreads:

```
if (tid < 32) {
        volatile T *smem = sdata;
        smem[tid] = smem[tid] + smem[tid + 32];
        smem[tid] = smem[tid] + smem[tid + 16];
        smem[tid] = smem[tid] + smem[tid + 8];
        smem[tid] = smem[tid] + smem[tid + 4];
        smem[tid] = smem[tid] + smem[tid + 2];
        smem[tid] = smem[tid] + smem[tid + 1];
}

if (tid == 0) {
        global_data[blockIdx.x] = sdata[0];
}
```

- Volatile basically tells the optimizer "off-limits!"
- Enables us to safely skip __syncthreads()

Summary

- All current processors are parallel:
  - You cannot ignore parallelization and expect high performance
  - Serial programs utilize 1% of potential!
- Getting started coding for GPUs has never been easier:
  - Nvidia CUDA tightly integrated into Visual Studio
  - Excellent profiling tools available with toolkit
- Low hanging fruit has been picked:
  - The challenge now is to devise new intelligent algorithms that take the architecture into consideration

Some references

- Code examples available online: http://github.com/babrodtk
- NVIDIA CUDA website: https://developer.nvidia.com/cuda-zone
- Brodtkorb, Hagen, Schulz and Hasle, **GPU Computing in Discrete Optimization Part I: Introduction to the GPU,** EURO Journal on Transportation and Logistics, 2013.
- Schulz, Hasle, Brodtkorb, and Hagen, **GPU Computing in Discrete Optimization Part II: Survey Focused on Routing Problems,** EURO Journal on Transportation and Logistics, 2013.
- Brodtkorb, Sætra and Hagen, **GPU Programming Strategies and Trends in GPU Computing**, Journal of Parallel and Distributed Computing, 2013.