Marco Fratarcangeli
http://www.dis.uniroma1.it/~frat/
frat@dis.uniroma1.it

General-purpose computing on graphics processing units (GPGPU)

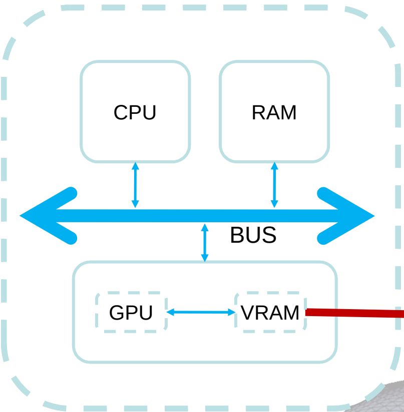
Sapienza University of Rome
Dept. of Computer and Systems Science (DIS)

Linköping University
Dept. of Electrical Engineering (ISY)

Outline

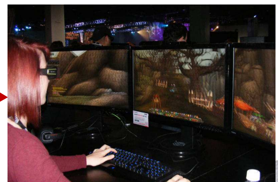
- Definitions & motivation
- GPU architecture
- GPGPU and CUDA
- Mapping data structures to the GPU
- Examples

What is a graphics card



Everything is local

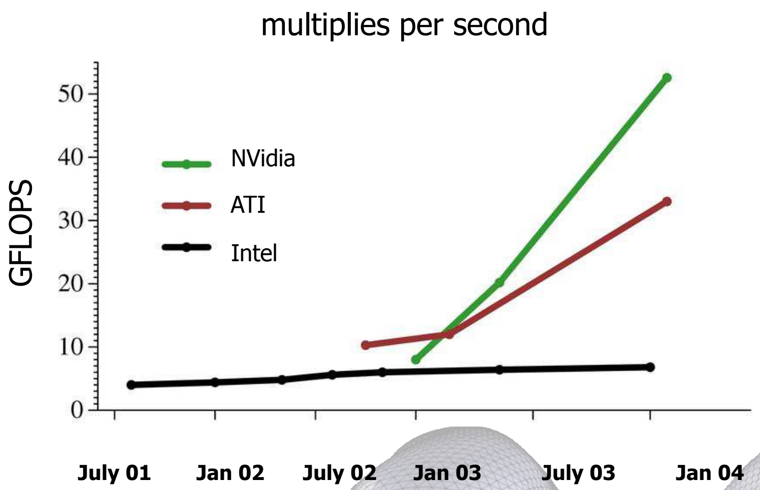

Main manufacterer:
Nvidia and ATI



Screen

- Graphics cards have evolved into a flexible and powerful processor
  - Programmability
    - » Shaders, CUDA, OpenCL
  - Precision
    - » Float 32
  - Power
    - » Hundreds of cores, GBs of ram
  - Cheap
    - » From hundreds to few thousand euros

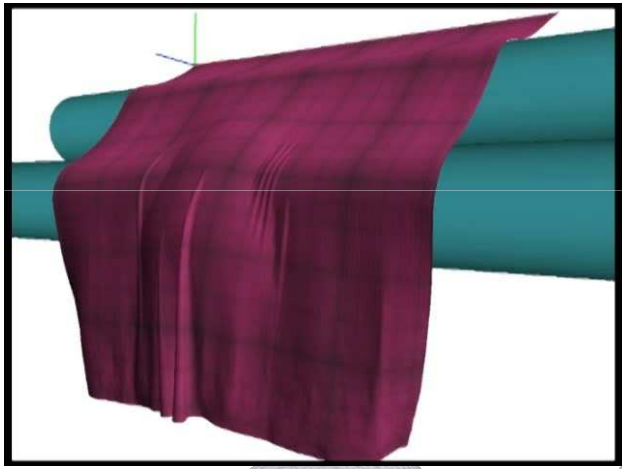
Computational Power



[Ian Buck]

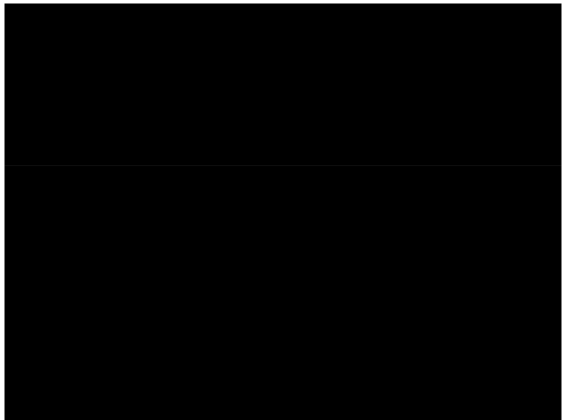
FLOPS (or **flops** or **flop/s**, for **floating-point operations per second**)

Example: cloth simulation



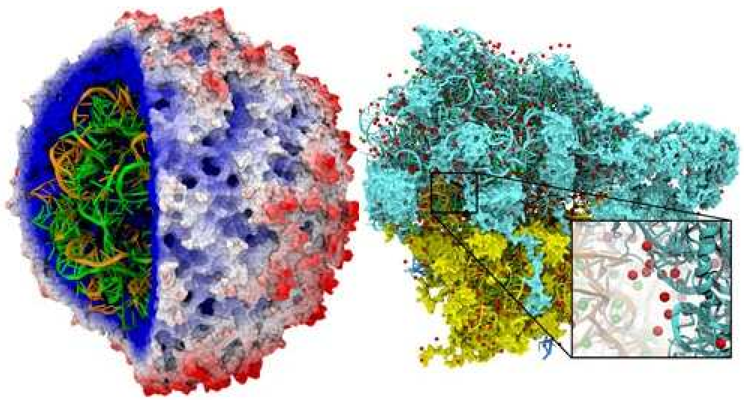
[M. Fratarcangeli]

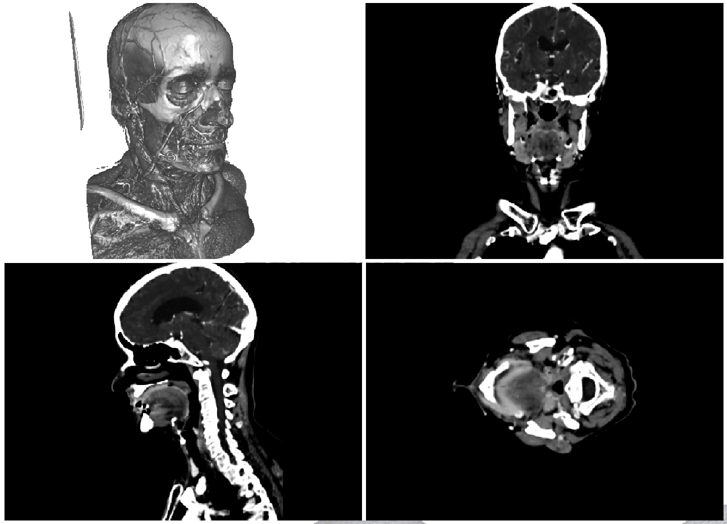
Example: N-body simulation



- 33.5 M particles
- 32 GPUs
- O(n log(n))

[T. Hamada & K. Nitadori]





Applications

- Large matrix/vector operations (BLAS)
- Protein Folding (Molecular Dynamics)
- Finance modeling
- FFT (SETI, signal processing)
- Raytracing
- Physics Simulation [cloth, fluid, collision,…]
- Sequence Matching (Hidden Markov Models)
- Speech/Image Recognition (Hidden Markov Models, Neural nets)
- Databases
- Sort/Search
- Medical Imaging (image segmentation, processing)
- And many, many, many more…
- See **GPGPU.org** for more examples

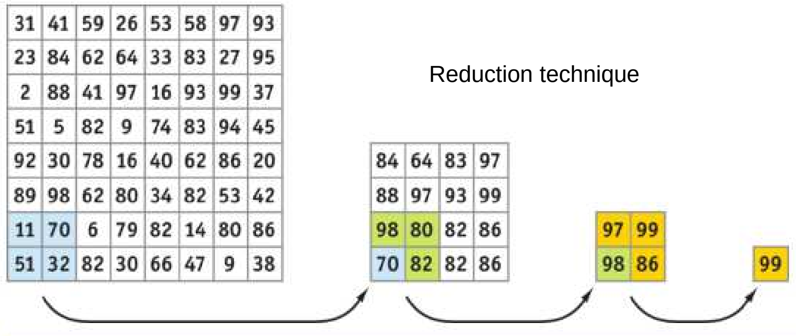
Room occupied by various circuits

[nVidia]

Example: Finding the max

Used to store 4 float (or a vec4)

Reduction technique



[GPU gems 2]

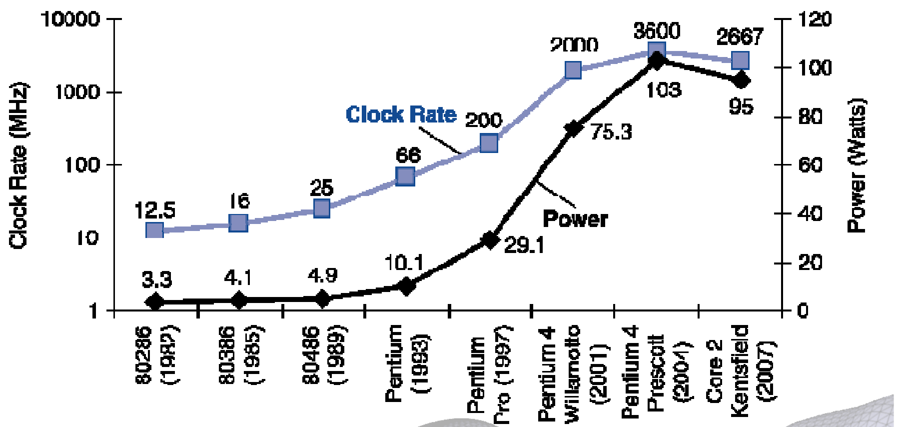
Problem: not so easy to use

- Can't simply "port" code written for the CPU!
- GPUs designed for and driven by video games
  - Programming model is unusual & tied to computer graphics
  - Programming environment is tightly constrained
- Underlying architectures are:
  - **Inherently parallel**
  - Rapidly evolving (even in basic feature set!)
  - Largely secret

Decline of the CPU evolution

- Power wall: clock frequency can no longer go up
- Memory wall: the memory architecture is insufficient
- ILP wall: attempts to parallelize are failed

Power wall



[Patterson & Hennessy]

Power wall

- The design goal for the late 1990's and early 2000's was to drive the clock rate up. This was done by adding more transistors to a smaller chip.
- Unfortunately, this increased the power dissipation of the CPU chip beyond the capacity of inexpensive cooling techniques.

Power wall

- 13% higher frequency => 73% more power
- Easy solution: Lower frequency a little, win much power
- Replace one high-frequency CPU with two slightly slower – for the same cost
  - Works nicely for two CPUs
- Increasing the number of cores brings to the *memory wall*

Memory wall

- The memory is slower than the CPU
- With more and more CPUs fighting for accessing the same RAM and caches, efficiency degrades
- Memory and bus bandwidth help (if available)

ILP wall

- ILP: Instruction Level Parallelism
- It takes an effort to program parallel systems.
- Programs must be rewritten to fit. The programs must be parallelized.
- Another problem: ***availability***
  - Machines were there but booking was required
  - bad development tools (e.g., debugger)
  - unavailable experts – nobody could help

History - CPU

- 80’s: CPU and system at the same speed.
  - Zero wait states
- 1993: CPUS faster than the rest of the system. Rapid raise of frequency.
- Late 90’s to present: Multi-CPU systems, multi-core CPUs.
- CPUs are still improving, but going for higher frequency is not as obvious as before.

History - GPU

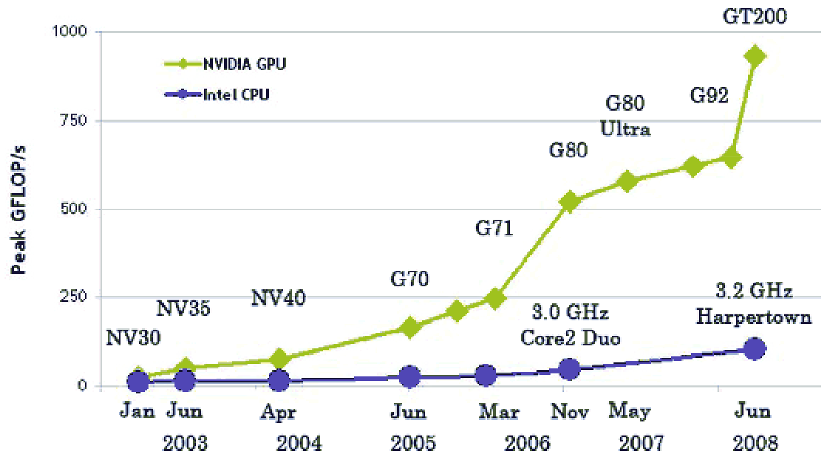
- 80’s: hardware sprites - push pixels with low-level code.
- 1993: Textured 3D games - Wolfenstein3D, Doom
- Early 90’s: professional 3D boards
- 1996: 3dfx Voodoo1
- 2001: programmable shaders
- 2006: G80, unified architecture. CUDA
- 2009: OpenCL
- 2010: Fermi architecture

Quantitative evolution

	1995	2005	
CPU Frequency (GHz)	.1	3.2	32x
Memory Frequency (GHz)	.03	1.2	40x
Bus Bandwidth (GB/sec)	.1	4	40x
Hard Disk Size (GB)	.5	200	400x
Pixel Fill Rate (GPixels/sec)	.0004	3.3	8250x
Vertex Rate (Gverts/sec)	.0005	.35	700x
Graphics flops (Gflops/sec)	.001	40	40000x
Graphics Bandwidth (GB/sec)	.3	19	63x
Frame Buffer Size (MB)	.2	256	128x

FLOPS (or **flops** or **flop/s**, for **floating-point operations per second**)

Quantitative comparison (GFlops)



Why so fast?

- Designed to handle large amounts of data:
  - Complex geometries (vertices + triangles)
  - Millions of output pixels
- Graphics pipeline is parallel
  - Parallelism hides memory latency
- Multibillion game industry pushes for horse power
  - Graphics card is a key component
  - New games need new impressive features

Hiding memory latency

- Each core has a small amount of local/private memory
- With many tasks per core:
  - When a task waits for a memory access, run another

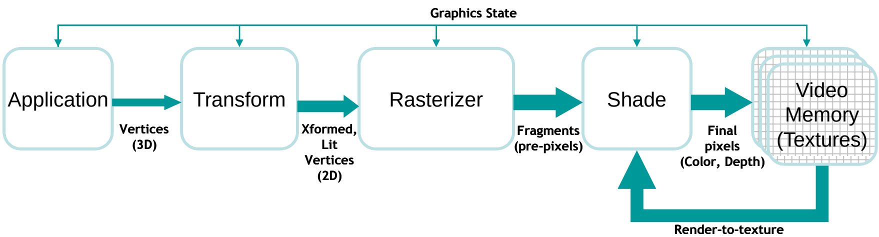
GPU capabilities

- **Must process pixels fast**
  - Early GPUs could draw textured, shaded triangles much faster than the CPU

- **Must do multiplication and divisions fast**
  - Must transform vertices and normalize vectors
- **Must be programmable**
  - E.g. Phong shading, Bump mapping

- **Must work in floating point**
  - E.g. High Dynamic Range

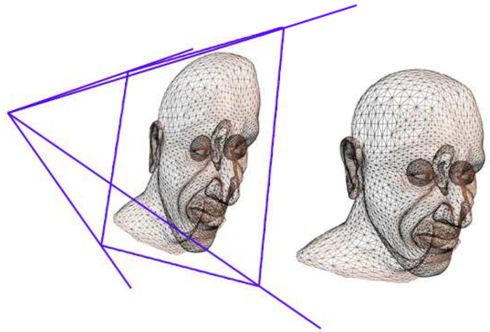
The Graphics Pipeline



- A simplified graphics pipeline
  – Note that pipe widths vary
  – Many caches, FIFOs, and so on not shown

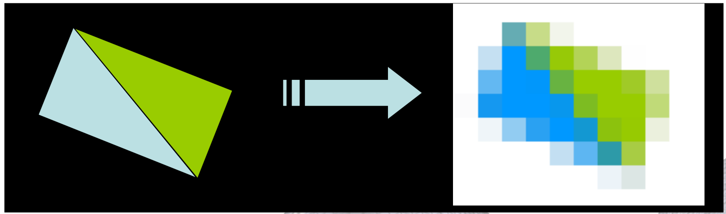
GPU Pipeline: Transform

- Vertex Processor (multiple operate in parallel)
  – Transform from "world space" to "image space"
  – Compute per-vertex lighting



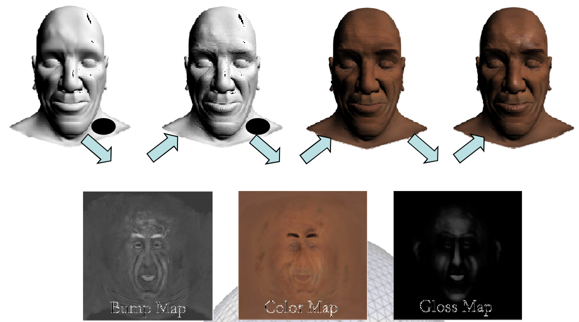
GPU Pipeline: Rasterizer

- Rasterizer
  - Convert geometric rep. (vertex) to image rep. (fragment)
    - Fragment = image fragment
      - Pixel + associated data: color, depth, stencil, etc.
  - Interpolate per-vertex quantities across pixels

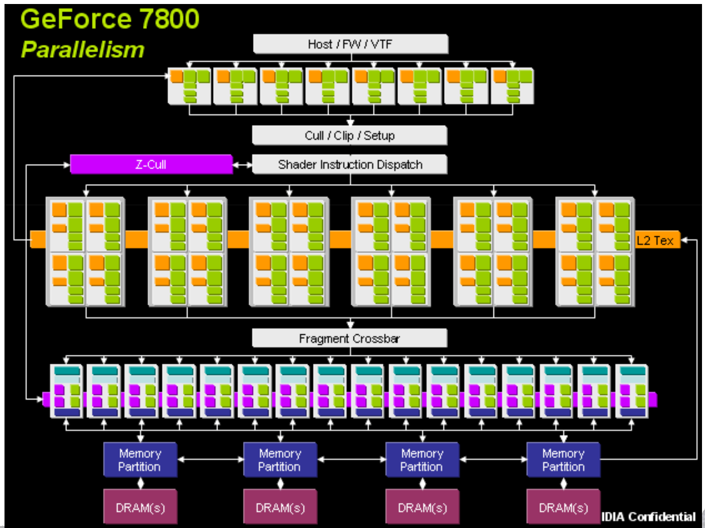


GPU Pipeline: Shade

- Fragment Processors (multiple in parallel)
  - Compute a color for each pixel
  - Optionally read colors from textures (images)



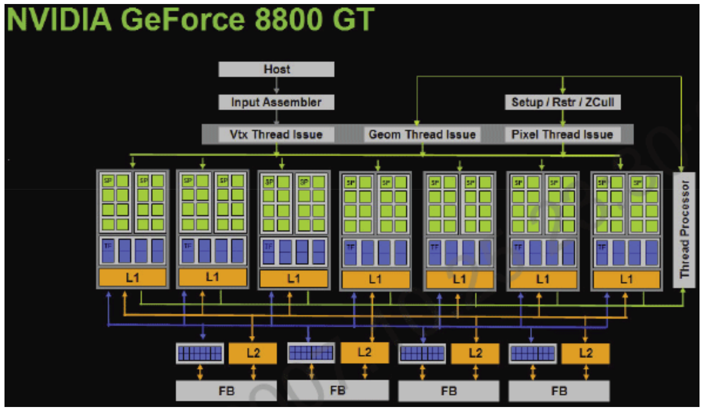
Overview of GeForce 7800 GTX - 2005



Vertex processors

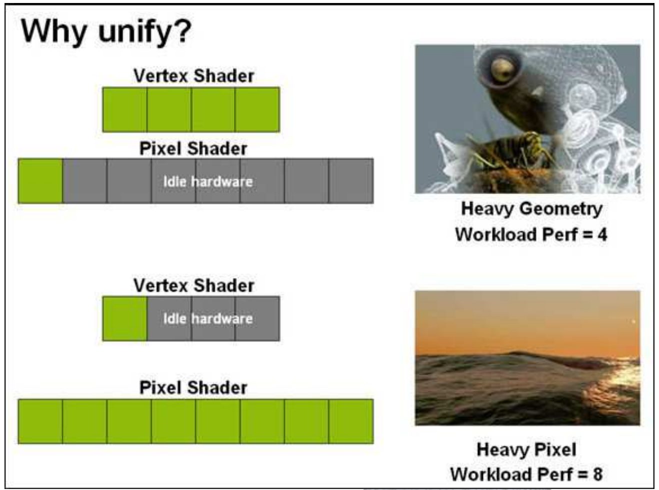
Fragment processors

Overview of GeForce 8800 GTX - 2006



Unified processors

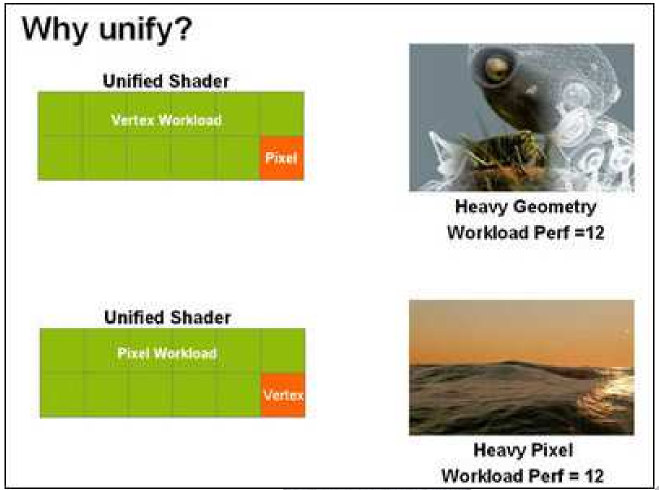
Why unify? - Load Balance



Vertex problem
e.g. Complex geometry

Fragment problem
e.g. Advanced rendering

Why unify? - Load Balance



2012: current architecture for gaming platforms

- GTX 550 Ti (~120€)
  - 192 cores
  - 1 GB ram
  - Width of the memory interface: 192 bit
- GTX 570 (~300€)
  - 480 cores
  - 1280 MB ram
  - Width of the memory interface: 320 bit

GPGPU

- General Purpose computation on Graphics Processing Units
- Perform demanding calculations on the GPU instead of the CPU
- Initially a wild idea, now very serious
- High processing power in parallel

Programming model: Streaming

- Problemi con il branching
- Use branching for early exit

CUDA-based GPGPU

- Compute Unified Device Architecture
- Integration of CPU and GPU code in the same program
- Hides graphics legacy
- Only works on NVidia hardware
- Requires extra-software – not very elegant
- Cross-platform: *nix/Mac/Win, mobile (Tegra3)
- Excellent results,
  - 100x speedups are common, before optimizing

high-level computing model

- 1. upload data to GPU
- 2. execute kernel
- 3 download the result to the CPU

Integrated source

- The source code of host (CPU) and kernel (GPU) can be in the same source file, written as one and the same program.
- Kernel code is identified by special modifiers

CUDA

- An architecture AND a C extension
- Spawn a large number of threads, to be run virtually in parallel
- All computations can't be executed in parallel. Instead, they are executed a bunch at a time – a ***warp***

CUDA – hello world

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

//kernel
__global__ void hello(char *a, int *b)
{
   a[threadIdx.x] += b[threadIdx.x];
}


int main()
{
   char a[N] = "Hello \0\0\0\0\0\0";
   int b[N] = {15, 10, 6, 0, -11, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0};

   char *ad;
   int *bd;

   const int csize = N*sizeof(char);
   const int isize = N*sizeof(int);
   printf("%s", a);

// allocate GPU memory
cudaMalloc((void**)&ad, csize);
cudaMalloc((void**)&bd, isize);

// upload to GPU memory
cudaMemcpy(ad, a, csize, cudaMemcpyHostToDevice);
cudaMemcpy(bd, b, isize, cudaMemcpyHostToDevice);

// 1 block, 16 threads
dim3 dimBlock(blocksize, 1);
dim3 dimGrid(1, 1);

// call kernel
hello<<<dimGrid, dimBlock>>>(ad, bd);

// download to CPU memory
cudaMemcpy(a, ad, csize, cudaMemcpyDeviceToHost);
cudaFree(ad);

printf("%s\n", a);
return EXIT_SUCCESS;
}
```

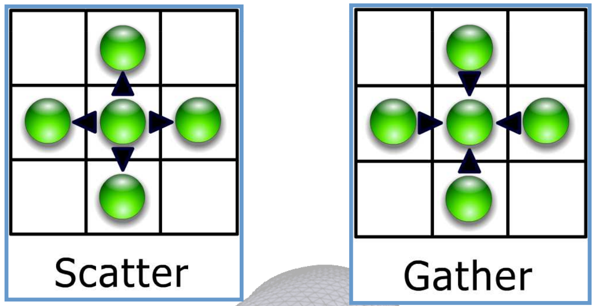
Example: N-Body symulation

- Typical "grid" computation
- N = 8192 bodies
- N^2 gravity computations
- 64M force computations / frame
- ~25 flops per force
- 44 frames per second
- GTS 250 (80€)
- 25 * 64M * 44 = ~70.4 GFlops

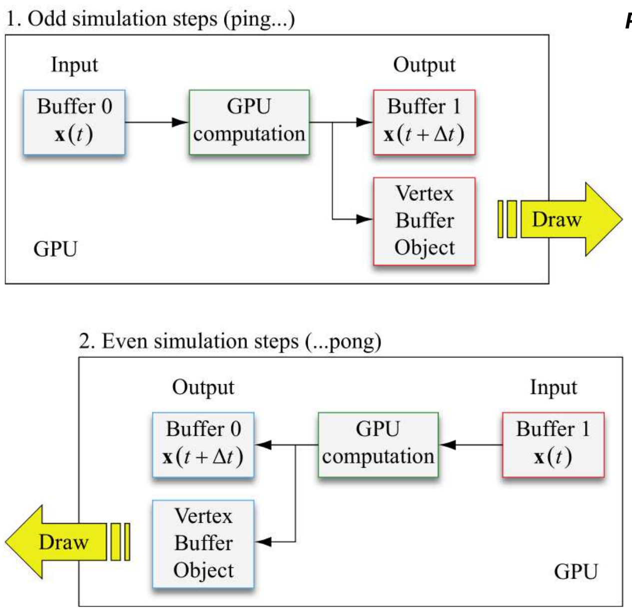
[Nyland, Harris, Prins]

Scatter vs Gather

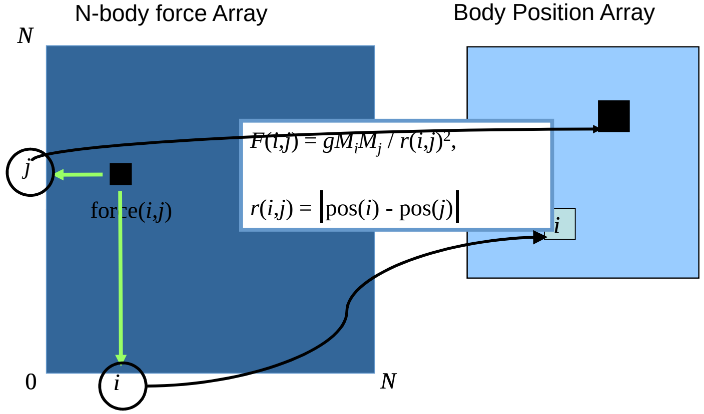
- Grid communication
  - Grid cells (cores) share information



Ping-pong technique



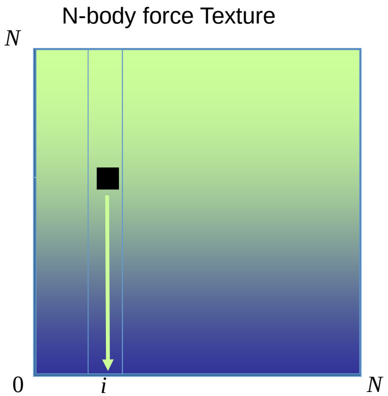
Computing Gravitational Forces



Force is proportional to the inverse square of the distance between bodies

Computing Total Force

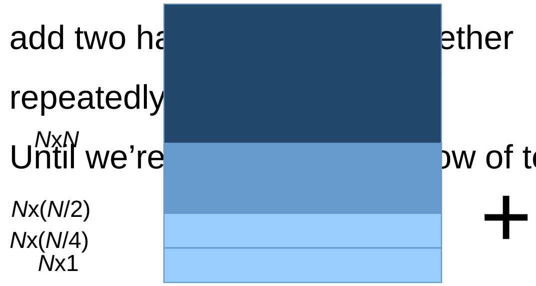
- Have: array of (i,j) forces
- Need: total force on each particle i
  - Sum of each column of the force array
- Can do all N columns in parallel



This is called a *Parallel Reduction*

Parallel Reductions

- **1D parallel reduction:**
  - sum N columns or rows in parallel
  - add two ha … ether
  - repeatedly
  - Until we're … w of texels



Requires $\log_2 N$ steps

Update positions and velocities

- We obtain a 1D array of total forces
  - One per body
- Update velocity
  - $\mathbf{v}(i, t+dt) = \mathbf{v}(i, t) + \mathbf{F}_{tot}(i)*dt$
  - A simple kernel reads previous velocity and force arrays and creates a new velocity texture
- Update position
  - $\mathbf{x}(i, t+dt) = \mathbf{x}(i, t) + \mathbf{v}(i, t)*dt$
  - Simple kernel reads previous position and velocity array, creates new position array

links

- http://www.gpgpu.org/developer
- http://developer.nvidia.com/cuda-downloads
- http://developer.nvidia.com/gpu-computing-sdk

Example: cloth simulation

- See the paper

overview

- Programming model and language
- Memory spaces and memory access
- Shared memory
- Example