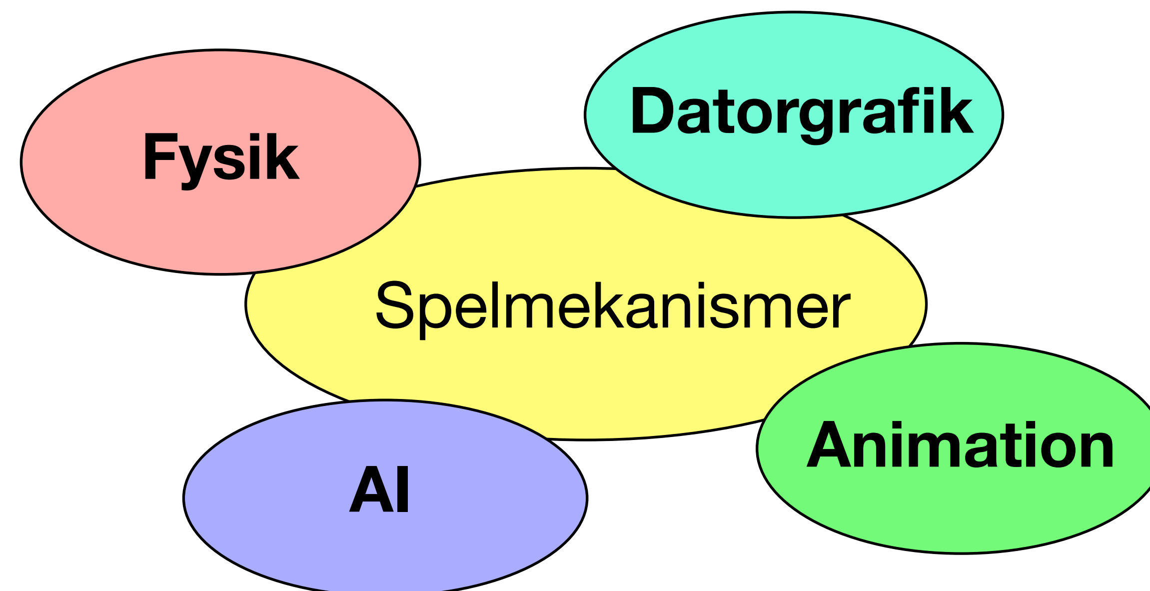
TSBK 03
Teknik för avancerade datorspel

Ingemar Ragnemalm, ISY



Föreläsning 6 (5)

GPU computing

- GPU computing, vad och varför
- Partikelsystem i shaders
- Compute Shaders
- Intro till CUDA

Föreläsningsfrågor

1. När är GPUer mycket snabbare än en CPU?
2. Vad har texturerna för roll för partikelsystemssimuleringen?
3. Vad för slags problem bör köras på GPU?
4. Hur kan partikelsystem interagera med omgivningen?
5. Varför blev transponering snabbare med shared memory?

GPU Computing
Generella beräkningar på GPUs som inte nödvändigtvis är bilder

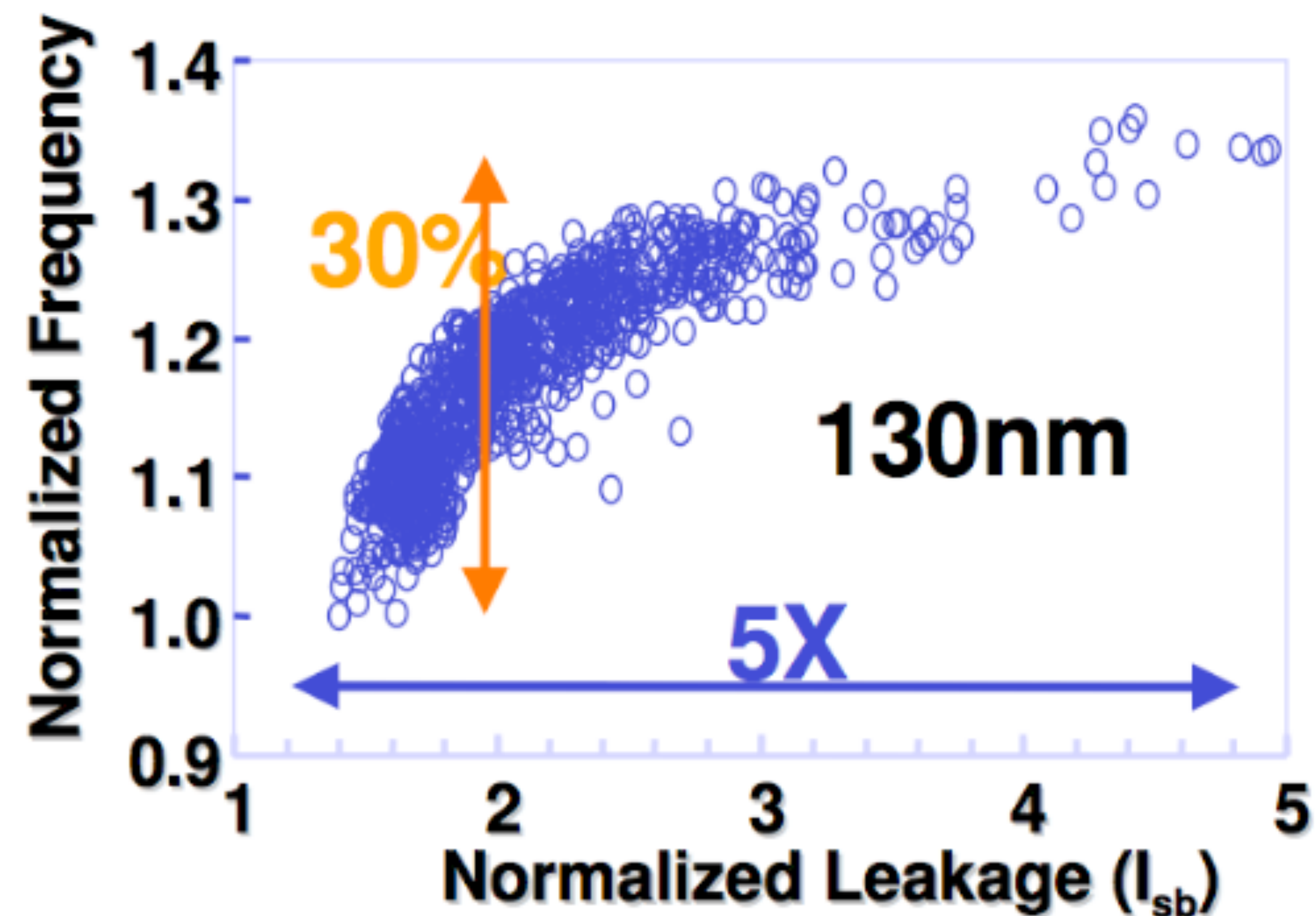
Använd GPU'ns beräkningskraft för andra problem än grafik!

Argument för: GPU'erna uppvisar stor prestandaökning medan CPU'er håller på att avstanna.

Nyckelkomponenter: Shaders, flyttalsbuffrar, parallellism.

Power wall

Vi kan inte längre höja frekvensen. Effekten stiger brant.



Men vi kan öka ANTALET!

Topp-GPUs 163 TFLOPS!

Mer konsumentvänligt: Ca 44 TFLOPS

CPU: 576 GFLOPS

Superdatorn på ditt skrivbord

Hur är detta möjligt?

Area use:

Control	ALU	ALU
	ALU	ALU

SIMD architecture, många små processorer, alla gör samma sak! SIMD = Single Instruction, Multiple Data

Flera möjliga APIer

Klassisk GPU Computing: Fragment shaders

Compute shaders (OpenGL mfl)

CUDA (NVidia)

OpenCL (öppen)

Direct Compute (Microsoft)

Metal (Apple)

Typisk ”klassisk” GPU Computing: som filtreringsdelen i HDR

Samma grundkoncept:

- Rendera till rektangel över hela bilden
- Gärna FBOs
- Flyttalsbuffrar
- Ping-ponging, flera pass med olika shaders
- Fokus på fragment-shaders

Enklaste GPU Computing: Processning av existerande bilder

Skillnad: Vi jobbar med en bild, en rektagulär yta, från början.

Linjära filter precis som i shaders, möjligheter till multipass och kombination av bilder.

GPU Computing-konceptet i fragment shaders

- Array av indata = textur
- Array av utdata = resulterande frame buffer
- Beräkningskärna = shader
- Beräkning = rendering
- Återkoppling = växling av FBO eller kopiering av textur

OBS skillnaden i renderingssituation:

- Extremt enkel geometri. Inget behov av vertexshaders.
- Mycket arbete på pixelnivå. Stora krav på fragmentshaders, höga precisionskrav.

Enkelt exempel: process-array

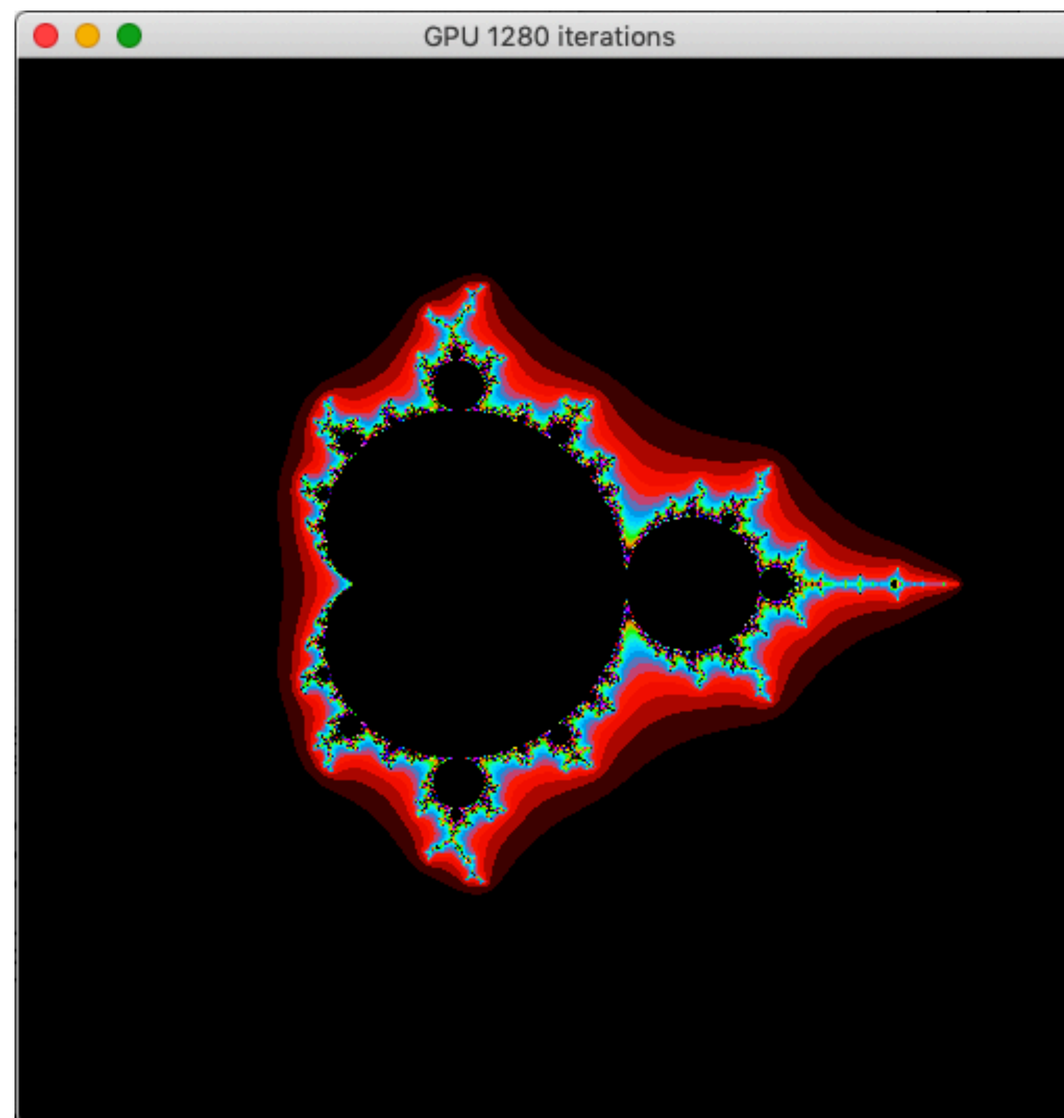
Array av data läggs i textur

```
uniform sampler2D texUnit;
in vec2 texCoord;
out vec4 fragColor;

void main(void)
{
   vec4 texVal  = texture(texUnit, texCoord);
   fragColor = sqrt(texVal);
}
```

Hämtas sedan tillbaka till CPU igen.

Enkelt prestandaexempel: Mandelbrot



Trivial parallelliserbart problem! (Nästan.)

Shader-baserad GPU Computing idag?

Om ditt problem passar i grafikpipelinen!

Utmärkt för problem som lätt mappas på texturdata.

Problem: Data bör mappas på RGBA. Renderingsdelarna gör ditt program mer komplicerat.

Att processa partikelsystem med shaders

Stora partikelsystem bör hanteras på GPU! Detta kan göras med shaders, men även med CUDA/CL/CS.

CPU: Ohanterbart med mer än ganska små system.

Partikelsystem lämpligt för parallella beräkningar.

Shaders nära grafiken, men lite mindre flexibelt än t.ex. CUDA.

Minimalt partikelsystem

Position p
Hastighet v

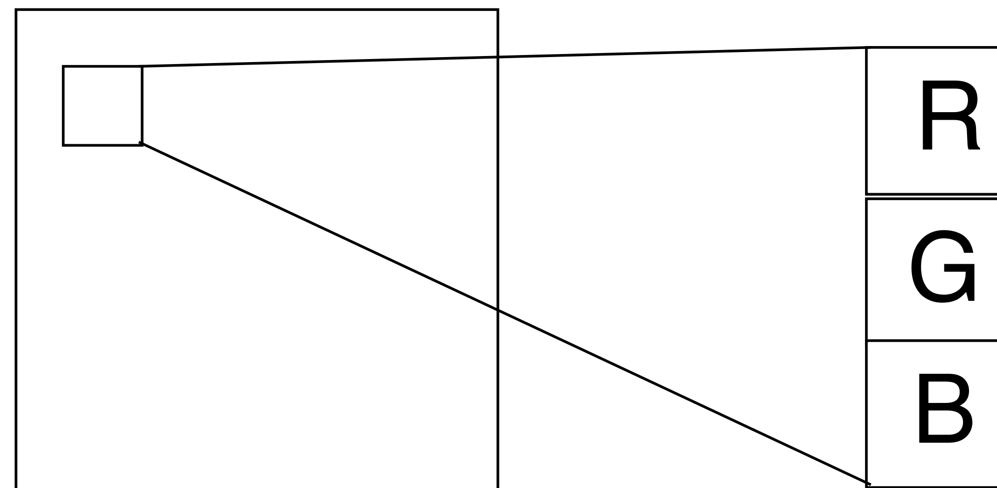
Uppdatering:

Position p = p + v * dt

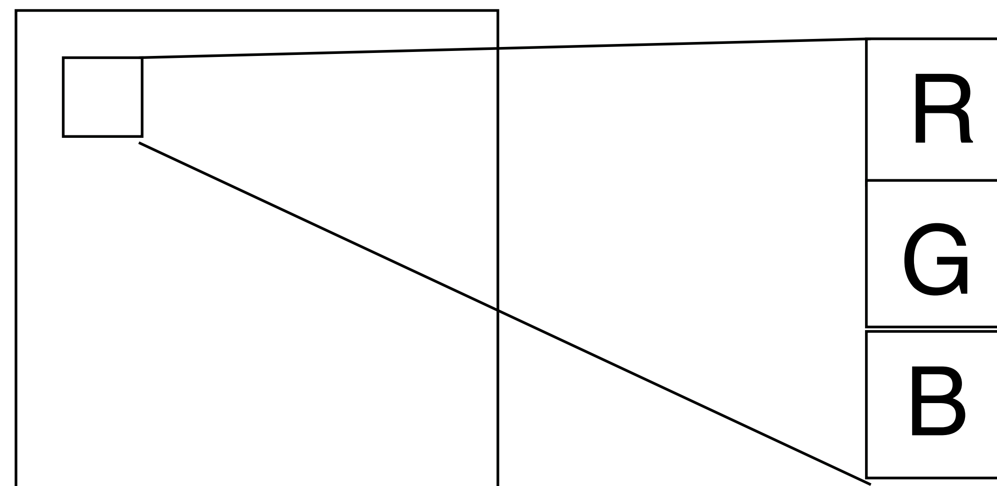
Hastighet v = v + a * dt

Lagra data i texturer!

positionTex



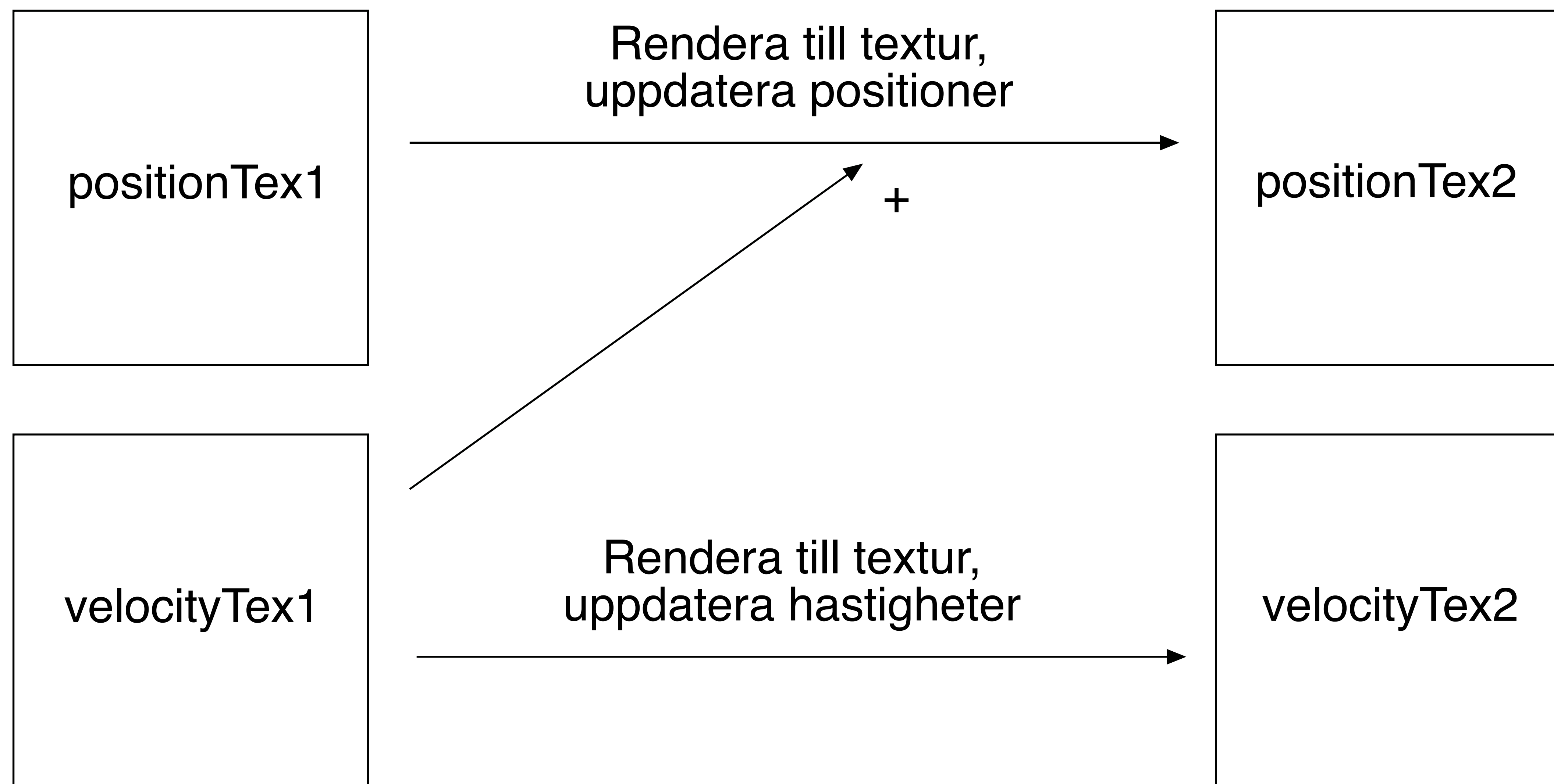
velocityTex



Dubbla texturer för ping-ponging

positionTex1	positionTex2
velocityTex1	velocityTex2

Rendera till FBO för att uppdatera



Rendering av partikelsystem

Görs typiskt med billboarding

Billboard = riktas alltid mot kameran

Texturerad quad med textur med transparens

Enklast: Tag bort rotation

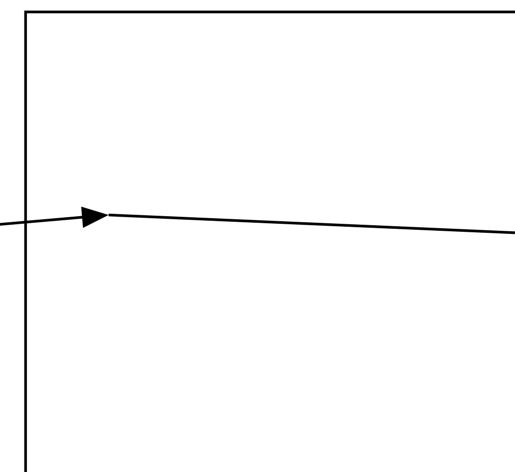
Rendering med instancing:

Intancing = Rendera många gånger med ett anrop.

Beräkna position i textur från instancing-index

Sätt position (vertexshader) på billboard från position läst ur textur

```
int i = gl_InstanceID;
int x = i / texSize;
int z = i % texSize;
vec2 tc = vec2((float(x)) / texSize,
(float(z)) / texSize);
vec4 p = (texture(posTex, tc) -
vec4(0.5)) * 3.5; // Position scaled from
texture value
```



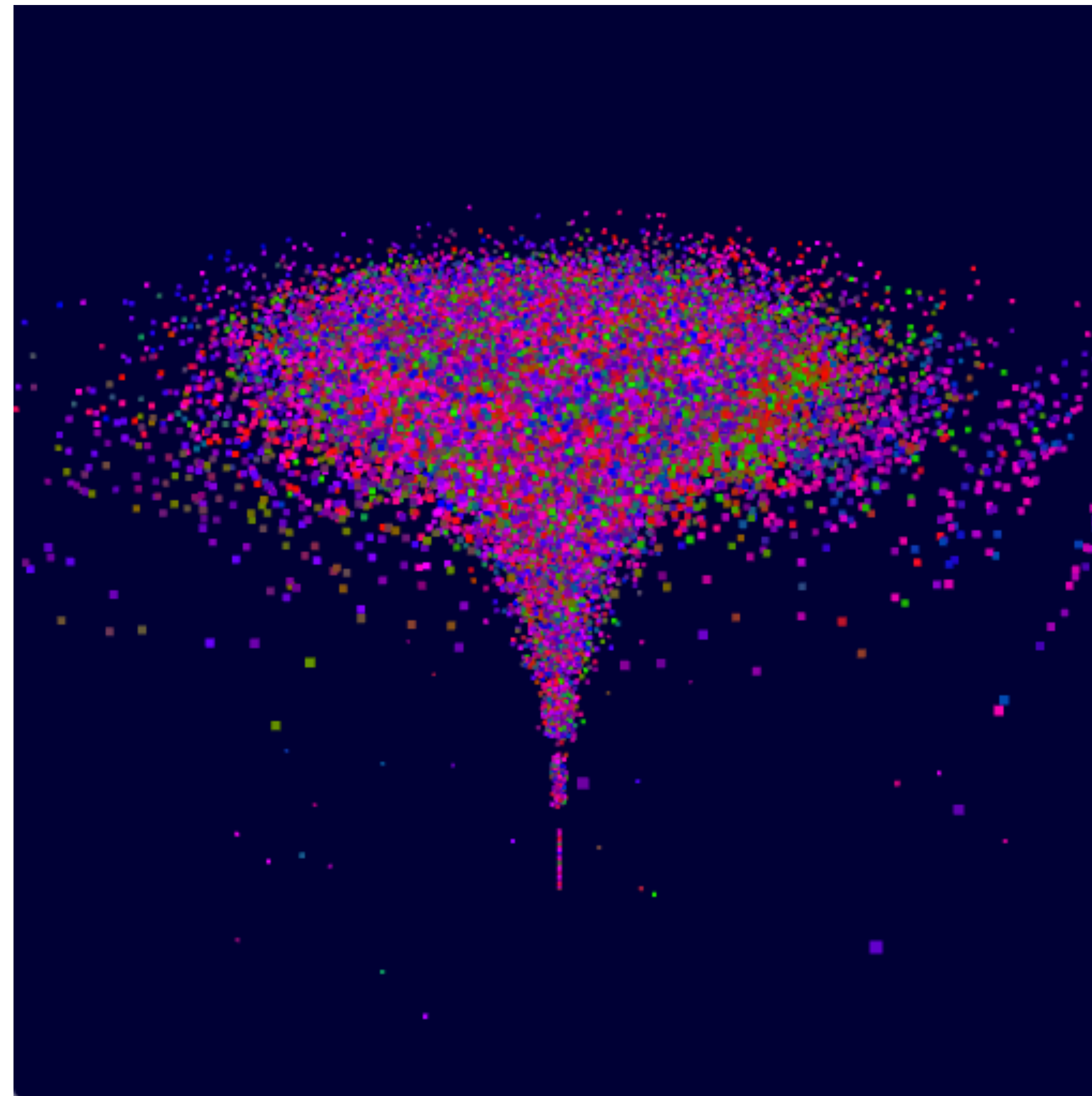
data utläst ur beräknad texel

Exempel 1: 8-bitars partikelsystem

Alla texturer/FBOer är RGBA8888

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

Det ger en upplösning på 256 steg för både position och hastighet!



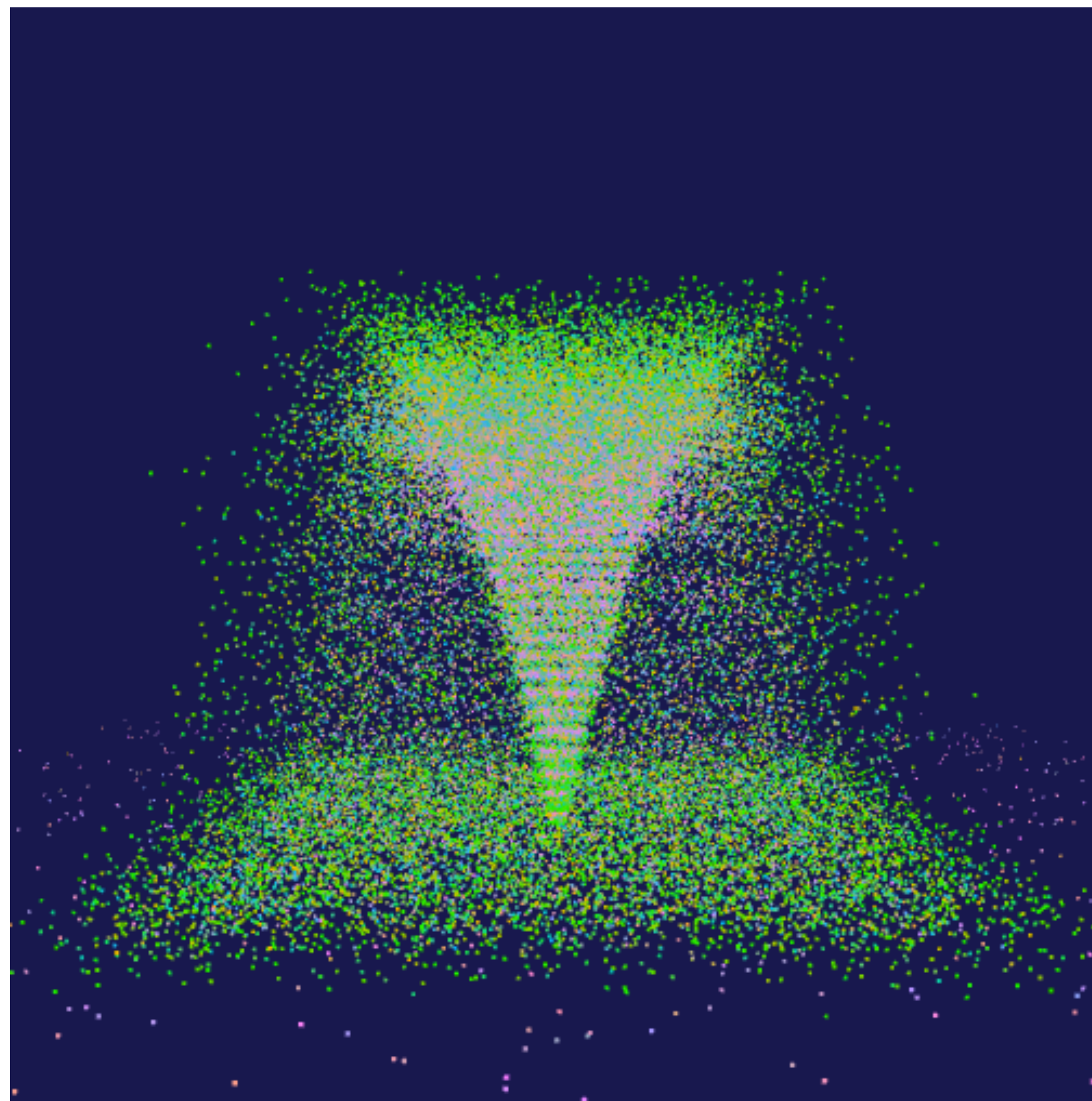
Inte kass. Lite begränsad i storlek och rörelsefrihet.

Bättre: Flyttalstexturer!

Ny tillämpning för flyttalstexturer!

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
NULL);
```

8 bitar byts mot 32 (24 bitar förutom exponenten!)



Mycket bättre. Stor frihet på alla sätt.

Minimala exempel!

Kommer upp på kurssidan.

Avsedda som bas att bygga på.

Klarar inte sortering eller beroenden av andra partiklar eller omgivningen (mer än plat mark)

Tre problem kvar:

Snyggare partiklar

Sortering

Beroenden av andra partiklar och omgivningen

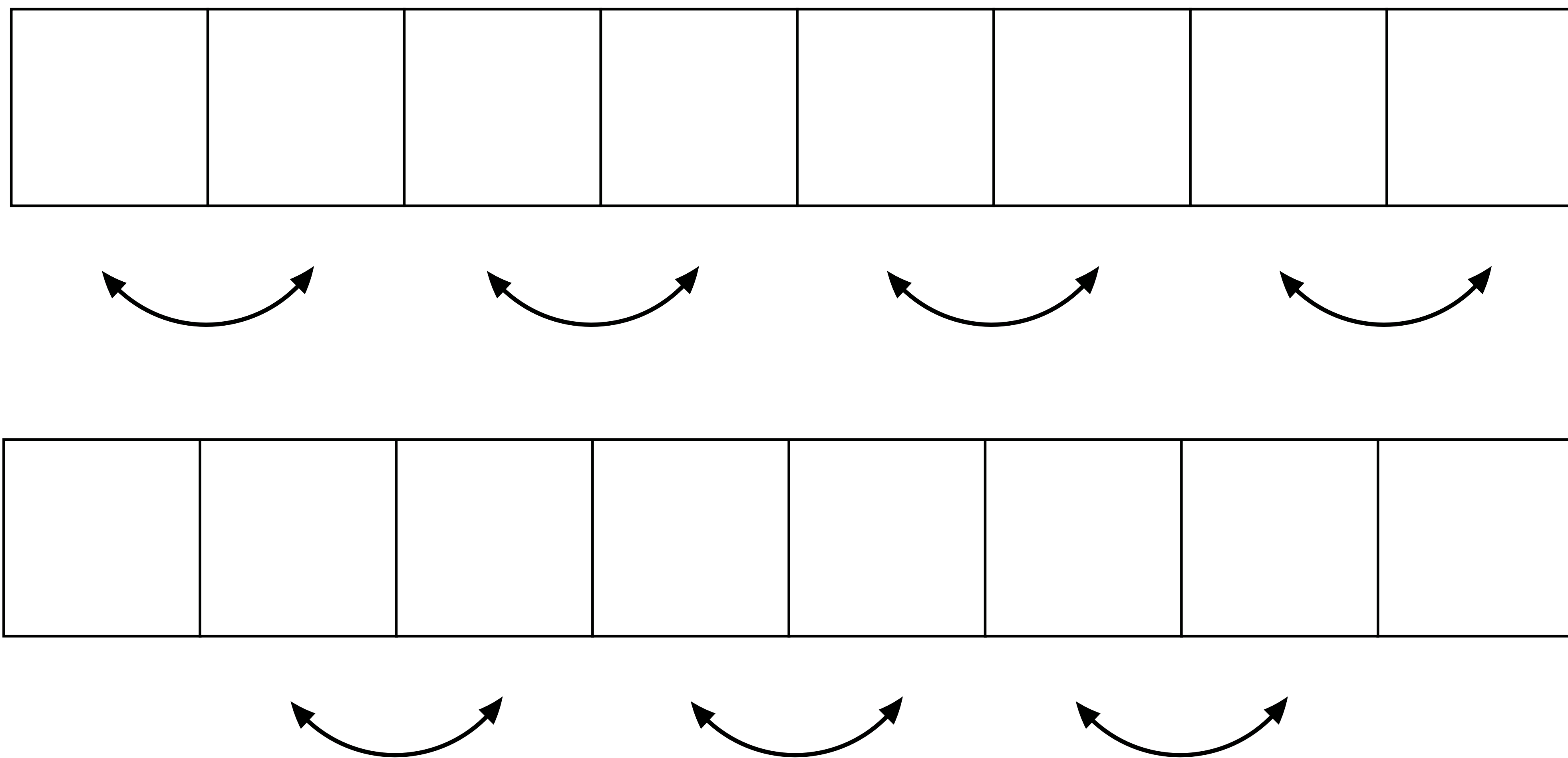
Sortering behövs i många fall!

Partiklar med semitransparens (nära kameran)!

Sortera på avstånd från kameran

- QuickSort olämplig
- Bitonic Merge Sort bättre
- Parallell bubblesort i några steg per frame kan vara ett bättre val!

Parallell bubblesort: Jämför parvis



Trick för att minska problemen

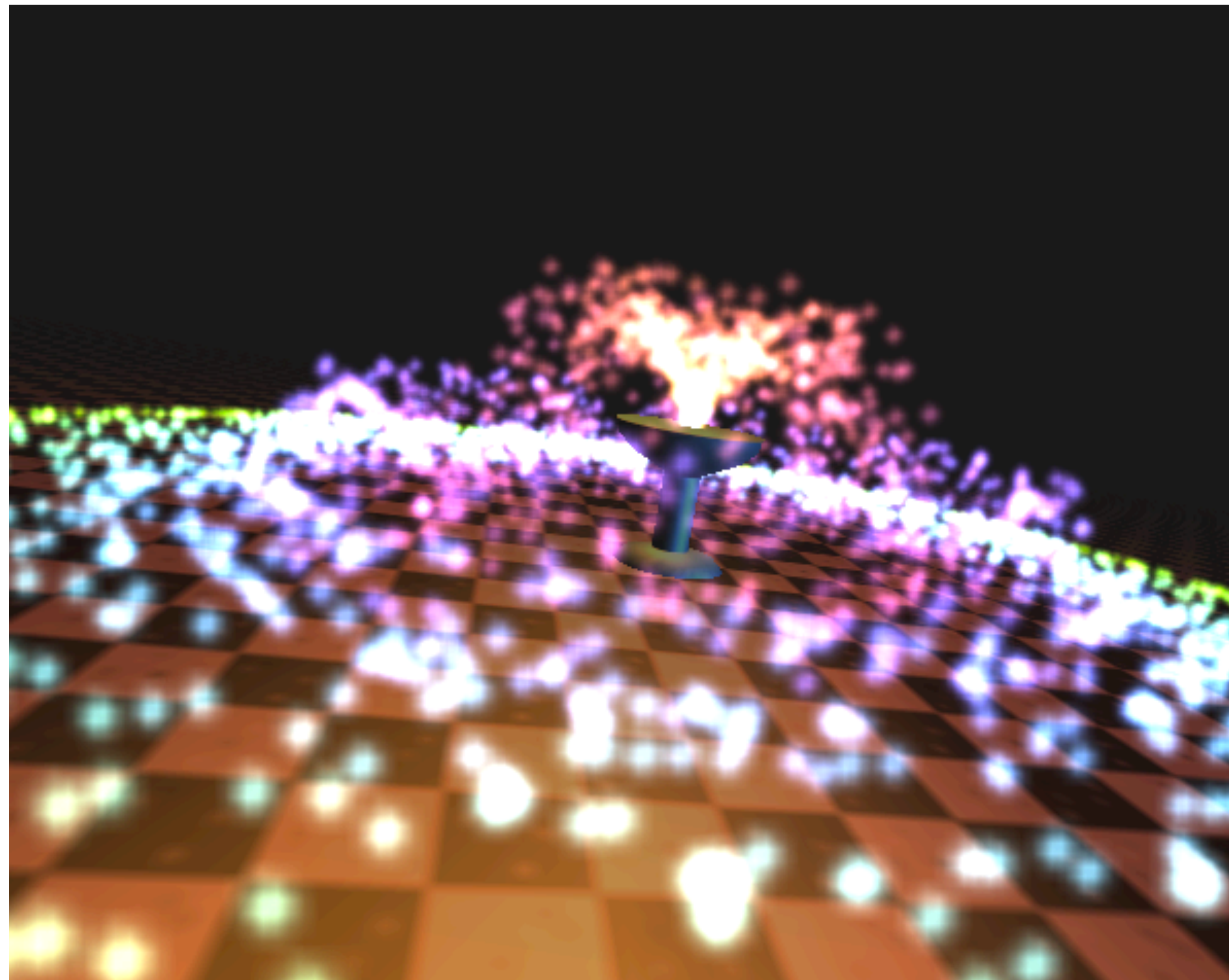
1) discard()

Bäst med skarpa kanter i textur

2) additiv blending

Fungerar för partiklar utan detalj

Gammalt exempel med additiv blendning



Kollisionsdetektering för partikelsystem

Inom systemet: Svårt problem - många till många

- Dela upp i celler! Octrees möjligt, uniform uppdelning ofta effektivare.

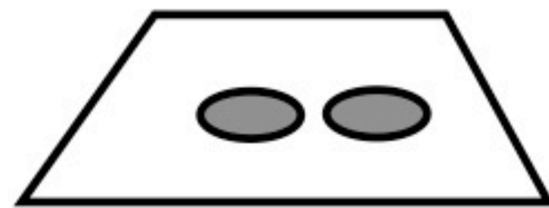
- Uppdatera cellerna när det behövs, när objekt rör sig utanför sin cell.

- Lagring knepigt, löses bäst med *spatial hashing*. för att lagra referens till alla partiklar i en stor array

- Test mot omgivning: Kan förenklas t.ex. med Z-buffer!

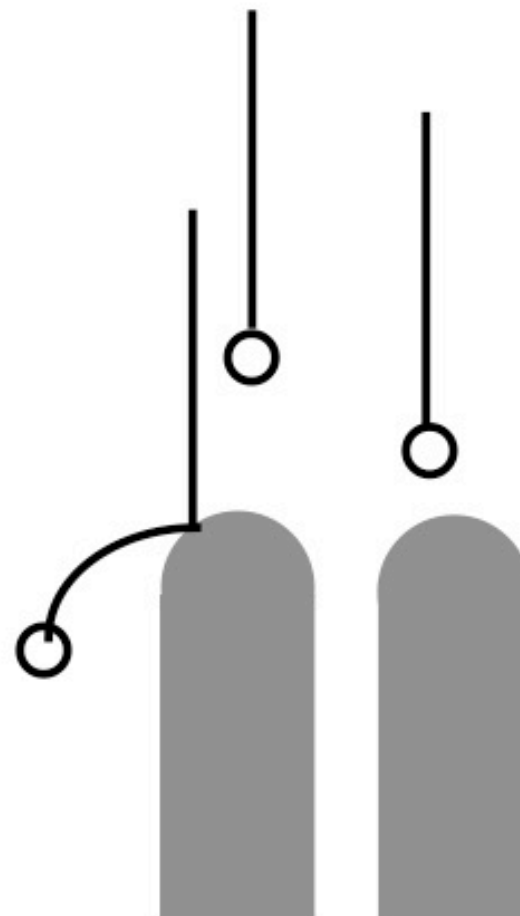
Exempel: Regn/snö mm

Kamera för att rendera Z-buffer uppifrån



Z-buffer

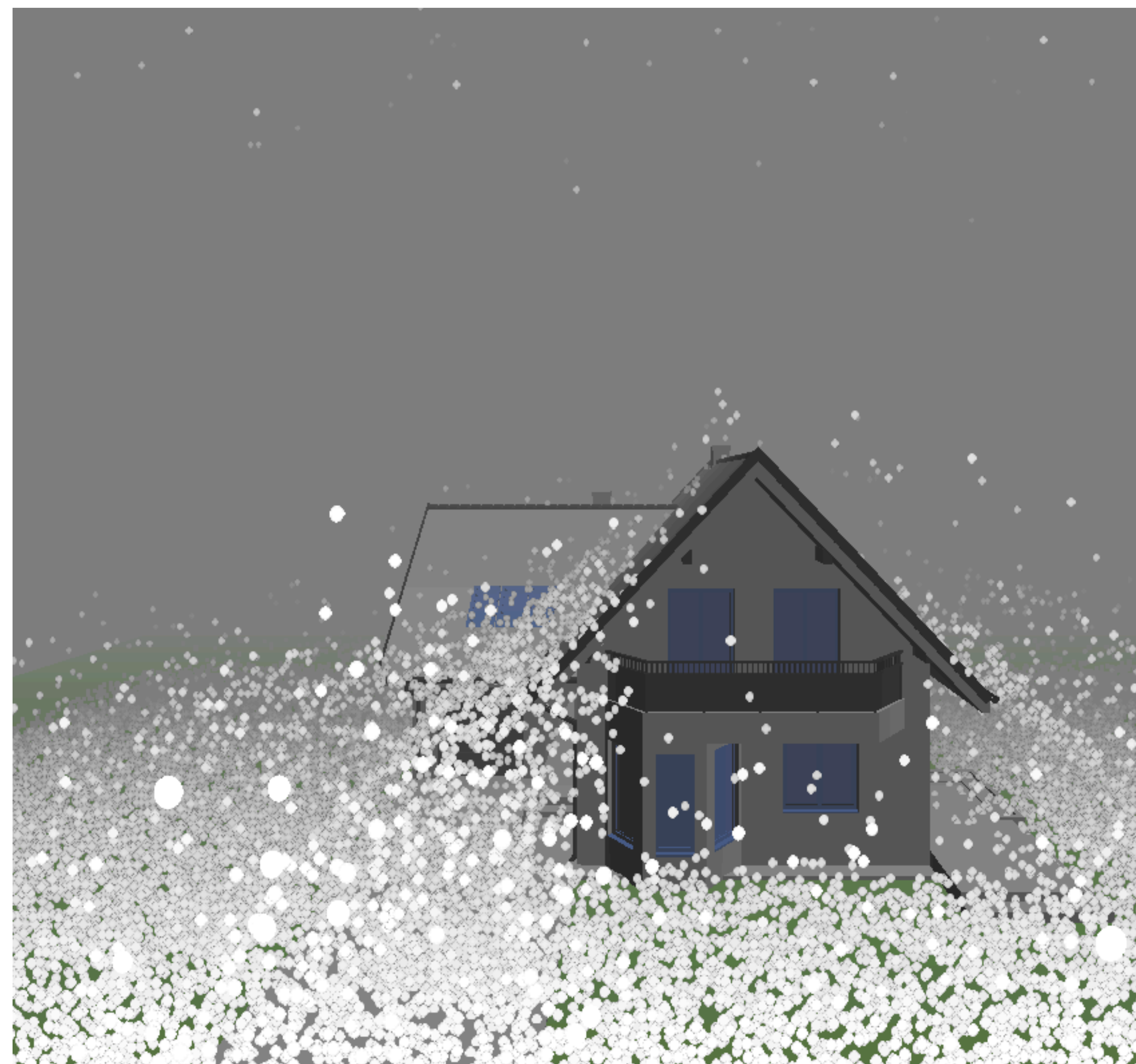
Scen



Partiklar kan nu studsa av objekt enbart genom att inspektera Z-buffern!

Projekt: Hailstorm

Partikelsystem i shaders med instancing



Kollision med omgivning med Z-buffern

Sammanfattning:

- Lagra partikeldata i texturer, en per texel
- Bind texturerna till FBOer
- Uppdatera genom rendering med ping-ponging
- Kollisionsdetektering
- Sortering (om det behövs)
- Rendera partiklar med instancing

Compute shaders

Framtiden för GPU computing eller sen efterapning av Direct Compute?

Tidigare rent Microsoft-koncept, Direct Compute

Senare även i OpenGL, ny shadertyp från OpenGL 4.3

Starkt alternativ

Varför använda det i stället för CUDA eller OpenCL?

+ Bättre integration med OpenGL

+ Ingen extra installation behövs!

+ Enklare att konfigurera än OpenCL

+ Inte NVidia-specifikt som CUDA

+ Om du kan GLSL så är Compute Shaders (ganska) lätt!

Men det är ju inte bara plus...

- En del nya koncept

- Inte del av grafikpipelinen som fragment shaders

- Apple har egen lösning

Compute shaders är ensamma, kompileras inte med några andra.

Parallella trådar

Arkitektur och utvidgning av C för parallellbearbetning.

Skapar ett stort antal trådar som körs parallellt /mer eller mindre).

Mycket är precis som i grafik! Du kan inte anta att alla trådar körs parallellt. De körs ett antal i taget: en warp (varp, syftar på vävning).

Men nu ser det mer ut som ett vanligt C-program. Inget trassel med data som lagras som pixlar, som i GLSL. Vi kan jobba med vanliga arrayer!

Liknar fragment shader-baserade lösningar:

1. Ladda upp data till GPU

2. Exekvera beräkningskärna

3. Ladda ner resultat

OK, hur gör jag?

Kompileras som alla andra shaders!

Modifiera labbkoden från GL_utilities, kompilera (ensam) som GL_COMPUTE_SHADER.

Lätta saker:

- Uniforms fungerar som vanligt
- Texturer fungerar som vanligt

Lite annorlunda än vanliga shaders

Attribut finns inte

Inte en tråd per fragment

Shader Storage Buffer Objects:

Generell buffertyp för godtyckliga data

Kan deklareras så shadern ser det som en array av strukturer

Kan läsas och skrivas fritt av Compute Shaders!

Hur får jag in indata?

Ladda upp till SSBO:

```
glGenBuffers(1, &ssbo);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
glBufferData(GL_SHADER_STORAGE_BUFFER, size, ptr,
GL_STATIC_DRAW);
```

Hur får shadern veta om den?

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, id,
ssbo);
```

```
layout(std430, binding = id, buffer x {type y[];};
```

Hur accessar jag data i shadern?

Bestäm antal trådar per block:

layout(local_size_x = width, local_size_y = height)

Trådnummer:

gl_GlobalInvocation
gl_LocalInvocation

```
void main()
{
	buffer[gl_GlobalInvocation.x] =
		- buffer[gl_GlobalInvocation.x];
}
```

Hur kör jag kärnan?

glUseProgram(program);

glDispatchCompute(sizex, sizey, sizez);

Argumenten till glDispatchProgram anger antalet block / workgroups. Antal trådar (work items) per block anges av shadern.

Hur får jag ut utdata?

```
glBindBuffer(GL_SHADER_STORAGE, ssbo);
ptr = (int *) glMapBuffer(GL_SHADER_STORAGE,
    GL_READ_ONLY);
```

Läs sedan från ptr[i]

```
glUnmapBuffer(GL_SHADER_STORAGE);
```

Komplett huvudprogram:

```
int main(int argc, char **argv)
{
  glutInit (&argc, argv);
  glutCreateWindow("TEST1");

// Load and compile the compute shader
  GLuint p =loadShader("cs.csh");

  GLuint ssbo; //Shader Storage Buffer Object

  // Some data
  int buf[16] = {1, 2, -3, 4, 5, -6, 7, 8, 9,
                10, 11, 12, 13, 14, 15, 16};
  int *ptr;

// Create buffer, upload data
  glGenBuffers(1, &ssbo);
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  glBufferData(GL_SHADER_STORAGE_BUFFER,
      16 * sizeof(int), &buf, GL_STATIC_DRAW);

// Tell it where the input goes!
// "5" matches "layout" in the shader.

  glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
         5, ssbo);

// Get rolling!
   glDispatchCompute(16, 1, 1);

// Get data back!
  glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo);
  ptr = (int *)glMapBuffer(
         GL_SHADER_STORAGE_BUFFER,
         GL_READ_ONLY);
  for (int i=0; i < 16; i++)
  {
    printf("%d\n", ptr[i]);
  }
}
```

Enkel Compute Shader:

OBS: Egentligen alldeles för mycket trådar för data (16*16*16)

```
#version 430
#define width 16
#define height 16

// Compute shader invocations in each work group

layout(std430, binding = 5) buffer bbs {int bs[];};

layout(local_size_x=width, local_size_y=height) in;

//Kernel Program
void main()
{
  int i = int(gl_LocalInvocationID.x * 2);
  bs[gl_LocalInvocationID.x] = -bs[gl_LocalInvocationID.x];
}
```

Kan du köra Compute Shaders?

Krav: OpenGL 4.3 + Kepler!

Inget svårt problem. 600-serien och uppåt.

Men stöds inte av Apples OpenGL. (4.2)

Organisation av beräkningar

1 kernel - 1 grid

1 grid - många block

1 block/work group - 1 streaming multiprocessor (SM)

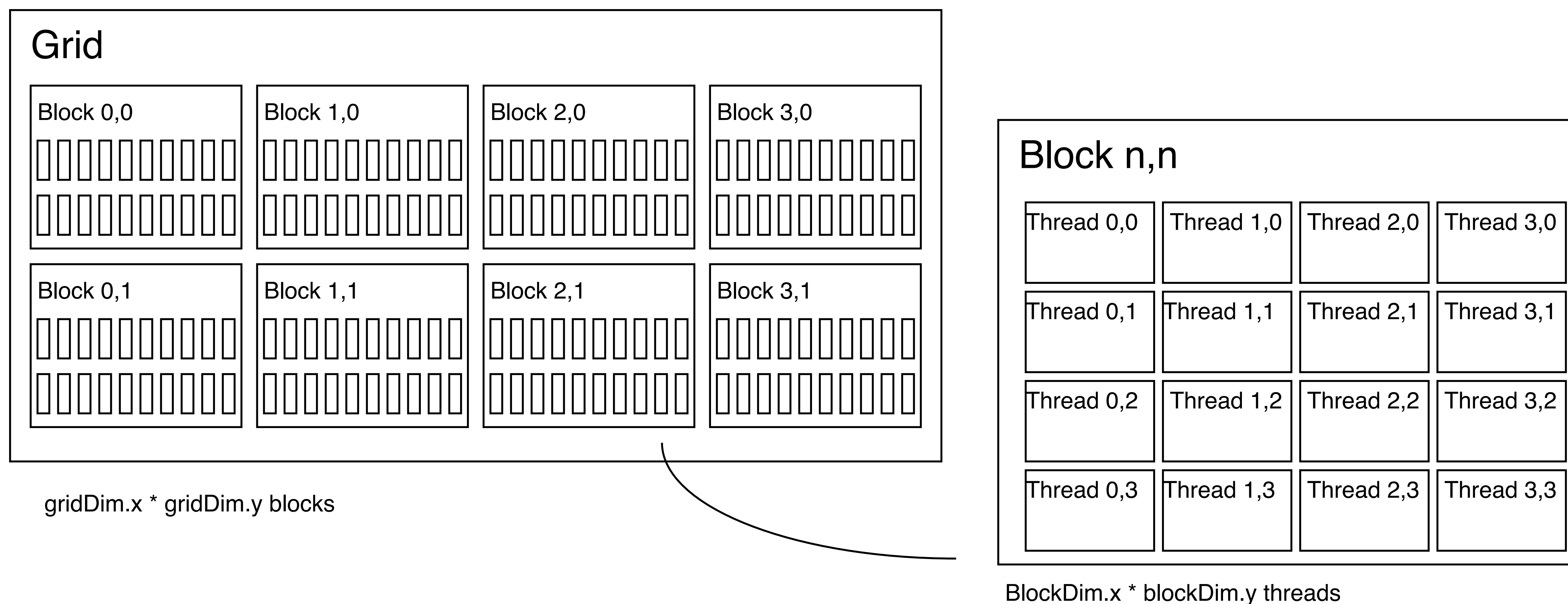
1 block - många trådar

Använd många trådar och många block! > 200 blocks rekommenderas.

Antal trådar bör vara multipel av 32

Fördelning av beräkningar över threads och blocks

Hierarkisk modell



Minnesaccess

Vitalt för prestanda!

Minnestyper

Coalescing

Exampel på hur man kan använda delat minne

Minnestyper

Global

Shared

Constant (read only)

Texture cache (read only)

Local

Registers

Viktiga när man optimerar

Globalt minne

400-600 cycles latency!

Använd lokalt delat minne som snabb mellanlagring -
en manuell cache!

Ordnade minnesaccesser (Coalescing)!

Kontinuerligt
Starta på 2-potens-adress
Addressera enligt trådnumrering

Använd shared memory för att omorganisera data!

Utvidgad beräkningsmodell:

1. Ladda upp data till GPU
2. Ladda upp delar av data tiii shared
3. Exekvera beräkningskärna
4. Synkronisera, upprepa från 2.
5. Ladda ner resultat

Använd shared memory för att minska antalet accesser av globalt minne

Läs block till shared memory

Processa

Skriv tillbaka det som behövs

Shared memory är en ”manuell cache”

Exempel: Matrismultiplikation

Använd shared memory för att förbättra ordningen av accesser i globalt minne

En fråga om cache såväl som *coalescing*.

Coalescing = Accessa minne i ordning!

Låt närliggande trådar accessa närliggande minne!

Shared memory och synkronisering

Shared memory deklareras *shared*.

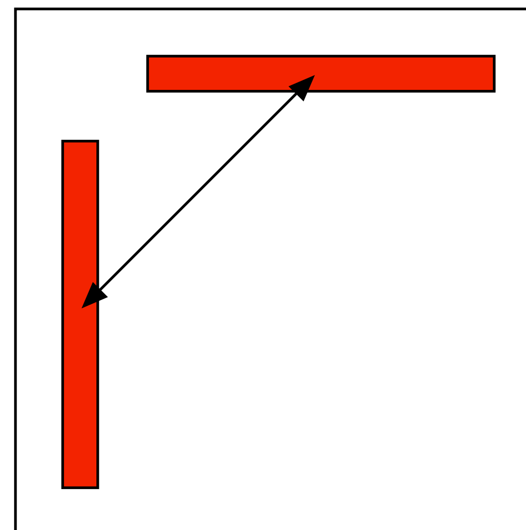
shared float myShared[SIZE];

Synkronisering finns i flera former, kallas barriers.

barrier()
memoryBarrier()
memoryBarrierShared()
groupMemoryBarrier()

Matristransponering

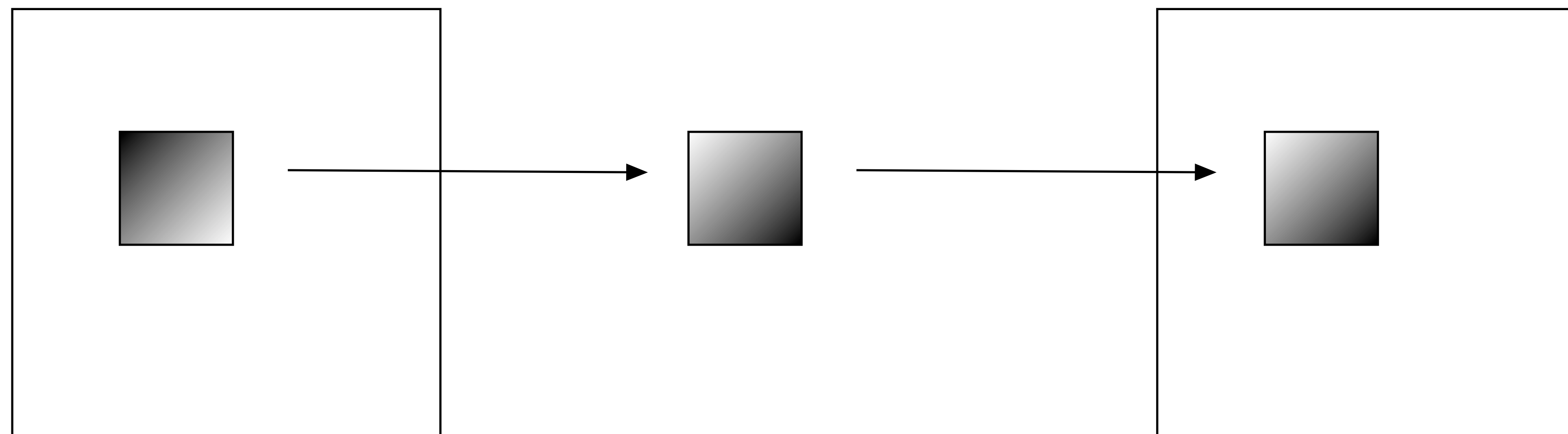
Problem med minnesaccess (coalescing)



Rad-för-rad och kolumn-för-kolumn. Get språng i minnet.

Matristransponering

Lösning med shared memory



Read from global memory to shared memory

In order from global, any order to shared

Write to global memory

In order write to global, any order from shared

Kod: Transponering med shared memory

```
#version 450
#extension GL_ARB_compute_shader : enable
#define width 16
#define height 16

layout(std430, binding = 7) buffer bufc {float c[];};
layout(std430, binding = 5) buffer bufa {float a[];};
layout(local_size_x=width, local_size_y=height) in;

shared float s[width*height];

//Kernel Program
void main()
{
   uint theSizeX = gl_NumWorkGroups.x * gl_WorkGroupSize.x;
   uint theSizeY = gl_NumWorkGroups.y * gl_WorkGroupSize.y;

   int i = int(gl_GlobalInvocationID.y*theSizeX + gl_GlobalInvocationID.x);
   int j = int(gl_GlobalInvocationID.x*theSizeY + gl_GlobalInvocationID.y);
   int li = int(gl_LocalInvocationID.y*width + gl_LocalInvocationID.x);
   int lj = int(gl_LocalInvocationID.x*height + gl_LocalInvocationID.y);
   s[li] = a[i];

   barrier();

   c[i] = s[lj];
}
```

Liveexempel: Naiv matrismultiplikation

Bara naiva varianten.

Accelerering i stil med den jag fick med CUDA (kommer i CUDA-delen).

Compute Shaders, ofta förbisett starkt alternativ

- Portabelt mellan olika grafikkort och OS
- I princip samma funktionalitet som CUDA och OpenCL
- Ingen separat installation

CUDA

En lösning för generella beräkningar.

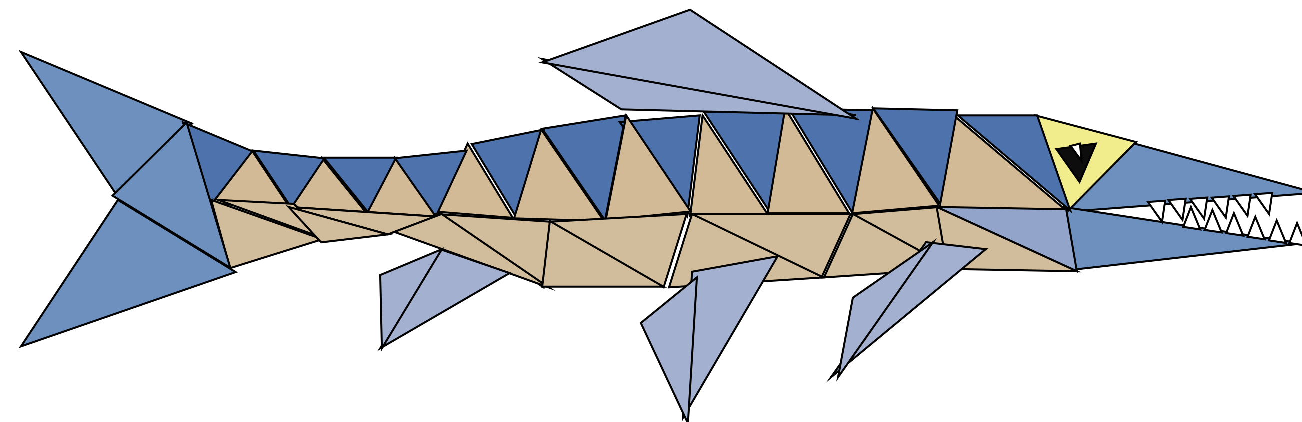
En introduktion:

Programmingsmodell och språk

Exempel

CUDA = Compute Unified Device Architecture

fast egentligen:



Utvecklat av NVidia

Bara på NVidia-kort, G80 eller bättre GPU-arkitektur

Designat för att dölja att hårdvarans egentligen är för grafik, och för att ge mer kontroll och flexibilitet

Integrerad kod

Källkoden till värdprogram och beräkningskärnor kan vara i samma källfil, skrivna som ett enda program!

Betydande skillnad mot shaders där kärnan är separat och explicit laddas och kompileras av värden.

Beräkningskärnor identifieras med speciella modifierare i koden.

Enkelt CUDA-exempel

Ett fungerande, kompilerbart, körbart exempel

```
#include <stdio.h>

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

__global__
void simple(float *c)
{
   c[threadIdx.x] = threadIdx.x;
}

int main()
{
   int i;
   float *c = new float[N];
   float *cd;
   const int size = N*sizeof(float);

   cudaMalloc( (void**)&cd, size );
   dim3 dimBlock( blocksize, 1 );
   dim3 dimGrid( 1, 1 );
   simple<<<dimGrid, dimBlock>>>(cd);
   cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
   cudaFree( cd );

   for (i = 0; i < N; i++)
     printf("%f ", c[i]);
   printf("\n");
   delete[] c;
   printf("done\n");
   return EXIT_SUCCESS;
}
```

Enkelt CUDA-exempel

Ett fungerande, kompilerbart, körbart exempel

```
#include <stdio.h>

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

__global__
void simple(float *c)  Kernel
{
   c[threadIdx.x] = threadIdx.x;
}                 thread identifier

int main()
{
   int i;
   float *c = new float[N];
   float *cd;
   const int size = N*sizeof(float);

   cudaMalloc( (void**)&cd, size );  Allocate GPU memory
   dim3 dimBlock( blocksize, 1 );  1 block, 16 threads
   dim3 dimGrid( 1, 1 );
   simple<<<dimGrid, dimBlock>>>(cd);  Call kernel
   cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
   cudaFree( cd );                    Read back data

   for (i = 0; i < N; i++)
     printf("%f ", c[i]);
   printf("\n");
   delete[] c;
   printf("done\n");
   return EXIT_SUCCESS;
}
```

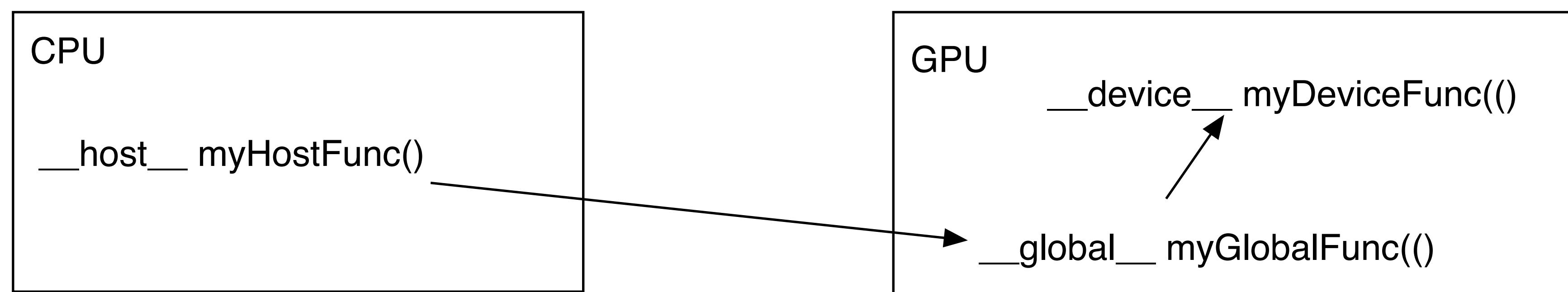
Modifierare för kodtyper

Tre modifierare anger hur kod skall användas:

__global__ körs på GPU, startas av CPU. Detta är beräkningskärnans ingångspunkt.

__device__ är kod som körs på GPU

__host__ är kod som körs på CPU (default).



Minneshantering

cudaMalloc(ptr, datasize)
cudaFree(ptr)

Liknar CPUns minnesallokering, men görs av CPUn för att allokera på GPU

cudaMemCpy(dest, src, datasize, arg)

arg = cudaMemcpyDeviceToHost
or cudaMemcpyHostToDevice

Körning av kärnan

simple<<<griddim, blockdim>>>(…)

(Mycket egendomlig syntax.)

”Grid” är en grid av ”block”. Varje block har nummer inom grid och varje tråd har nummer inom sitt block.

Inbyggda variabler för kärnan:

threadIdx och blockIdx
blockDim och gridDim

(OBS, inget prefix som i GLSL.)

Kompilera Cuda

nvcc

nvcc är nvidias kompilator, /usr/local/cuda/bin/nvcc

Källfiler har suffix .cu

Enklast möjliga kommandorad:

```
nvcc simple.cu -o simple
```

(Fler parametrar finns för bibliotek mm.)

Kan länkas med C++-kod

Indexera data med thread/block-IDs

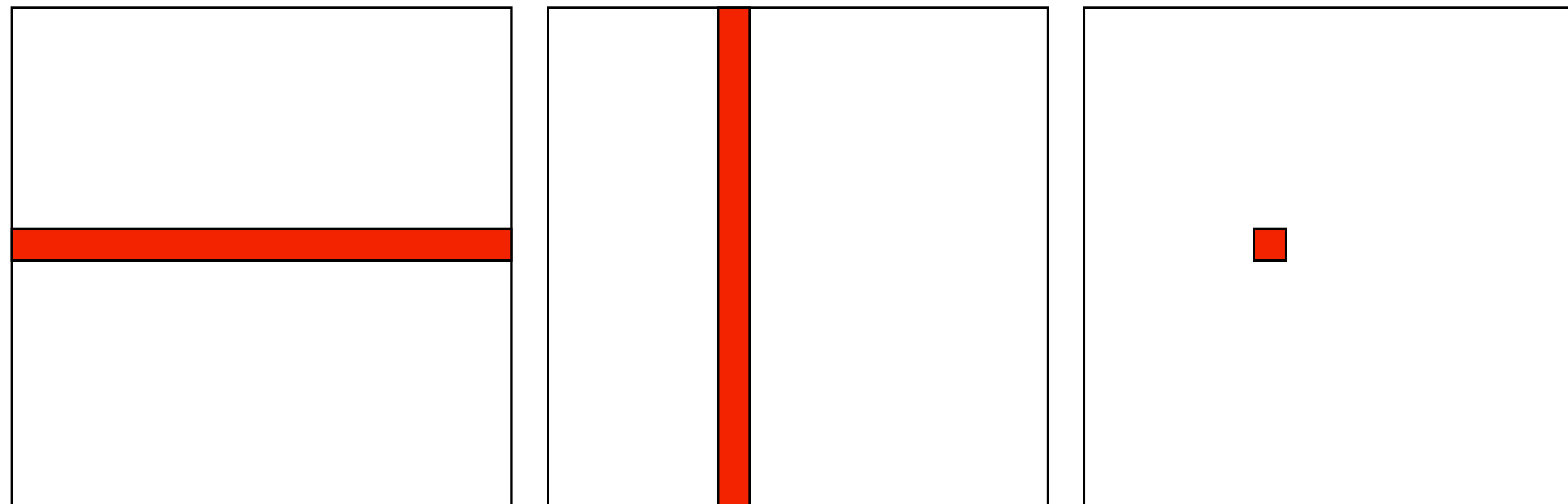
Beräkna index via blockIdx, blockDim, threadIdx

Enkelt exempel, beräkna kvadrat av varje element (enbart kärnan):

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Exempel: Optimerad matrismultiplikation

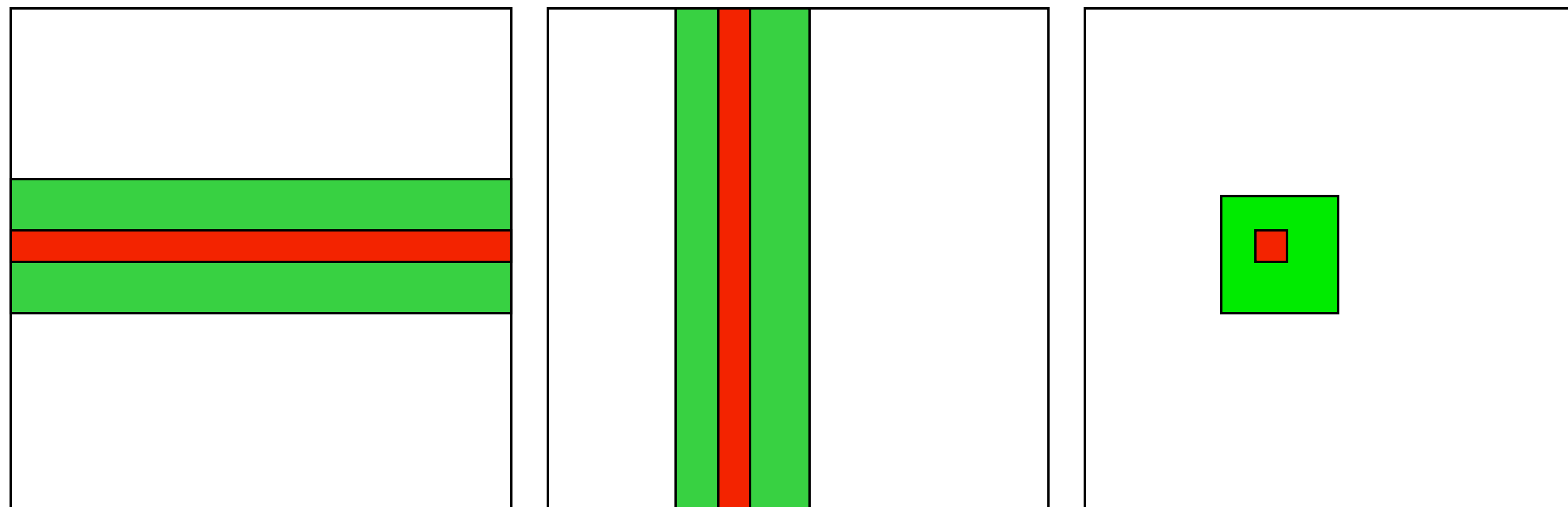
Mer shared memory!



För att multiplicera två N*N-matriser måste varje element accessas N gånger!

Naiv implementation: $2N^3$ globala minnesaccesser!

Dela upp matrismultiplikationen

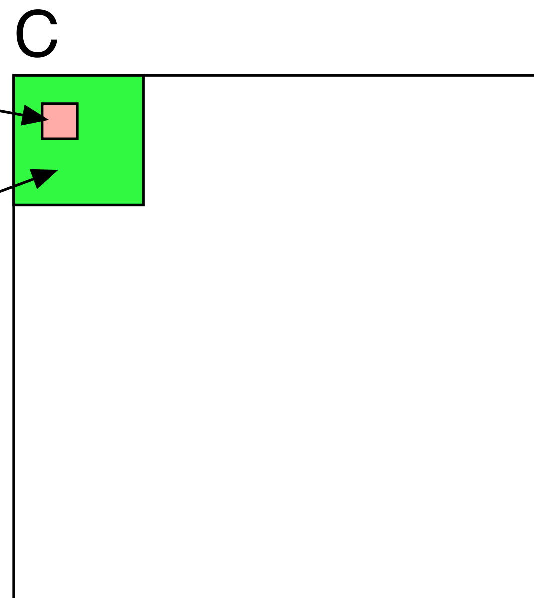


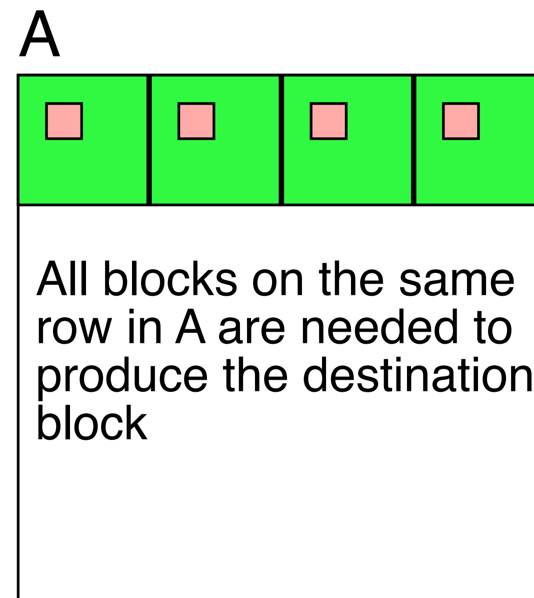
Låt varje block generera en del av utdata

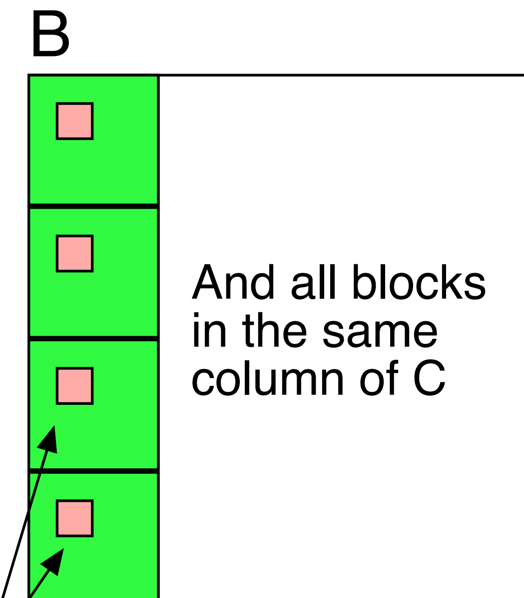
Läs in de delar av matrisen som blocket behöver i shared memory.

Destination element for thread

Destination block for thread

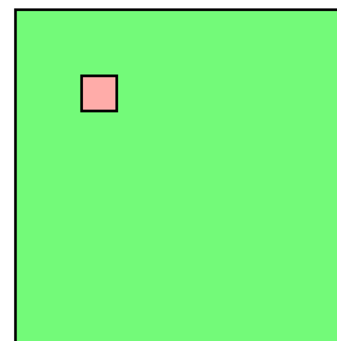


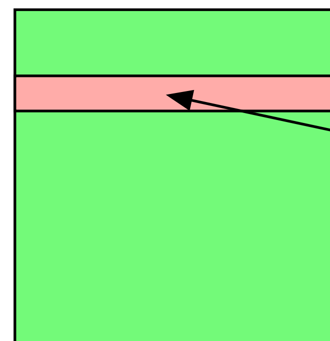


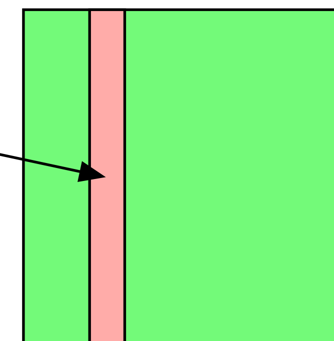


For every block, the thread reads one element matching the destination element

For every block, we loop over the part of one row and column to perform that part of the computation







What one thread reads is used by everybody in the same row (A) or column (B)!

Synkronisering

Så fort du gör något där en del av beräkningen beror av en annan så måste du synkronisera!

__syncthreads()

Typisk implementation:

- Läs till shared memory
- __syncthreads()
- Processa shared memory
- __synchthreads()
- Skriv resultatet till globalt minne.

Var kan jag köra CUDA?

Alla NVidia efter 8800 (≈alla!)

Korsplattform: MS Windows, Linux (MacOSX upp till 10.10).

Olympen + Asgård + Egypten

GLSL, CUDA eller Compute shaders?

- GLSL är överlägset mest portabelt och lättast att installera.

- CUDA är elegant och integrerat men kräver specialinstallation och är kräset på hårdvara.

- Compute shaders är portabelt (förutom Apple), kräver ingen specialinstallation bortom drivrutinerna.

Projekt med CUDA eller Compute Shaders?

Varför inte - om du har ett problem stort nog.

Parallellprogrammering är framtiden!

All shaderprogrammering är parallellprogrammering.

Så gott som all prestandaökning i framtiden kommer från parallelism.

Vi fortsätter i TDDD56.

...och kanske i era projekt i denna kurs?