CUDA

En lösning för generella beräkningar.

En introduktion:

Programmingsmodell och språk

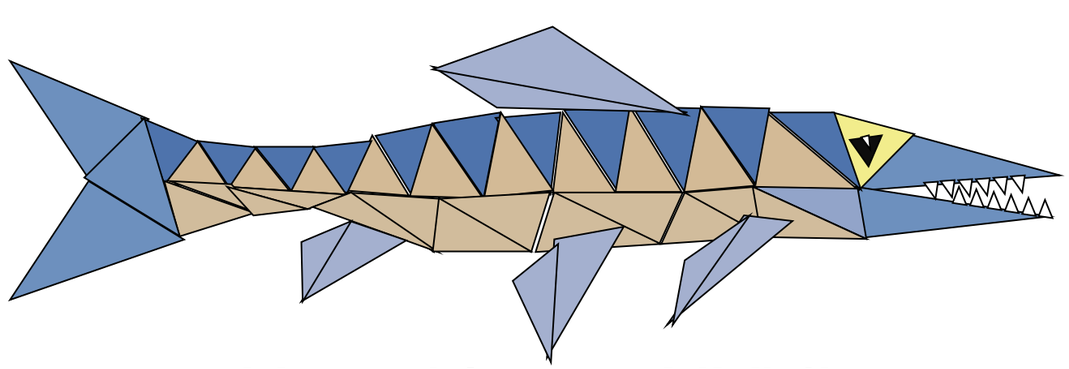
Minnesareor och minnesaccess

Delat minne

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

Liknar shader-baserade lösningar:

1. Ladda upp data till GPU

2. Exekvera beräkningskärna

3. Ladda ner resultat

Integrerad kod

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

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

Beräkningskärnor identifieras med speciella modifierare i koden.

CUDA

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 parallelt. 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!

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)
{
    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;
}
```

Kernel

thread identifier

Allocate GPU memory

1 block, 16 threads

Call kernel

Read back data

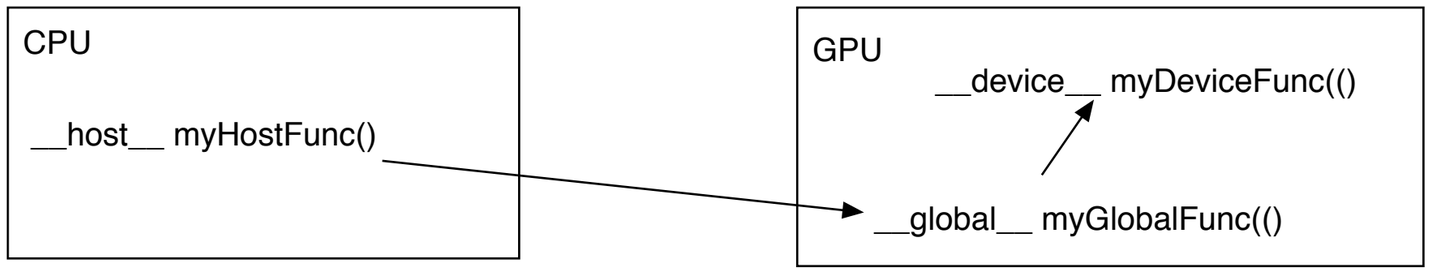
Modifierare för kodtyper

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

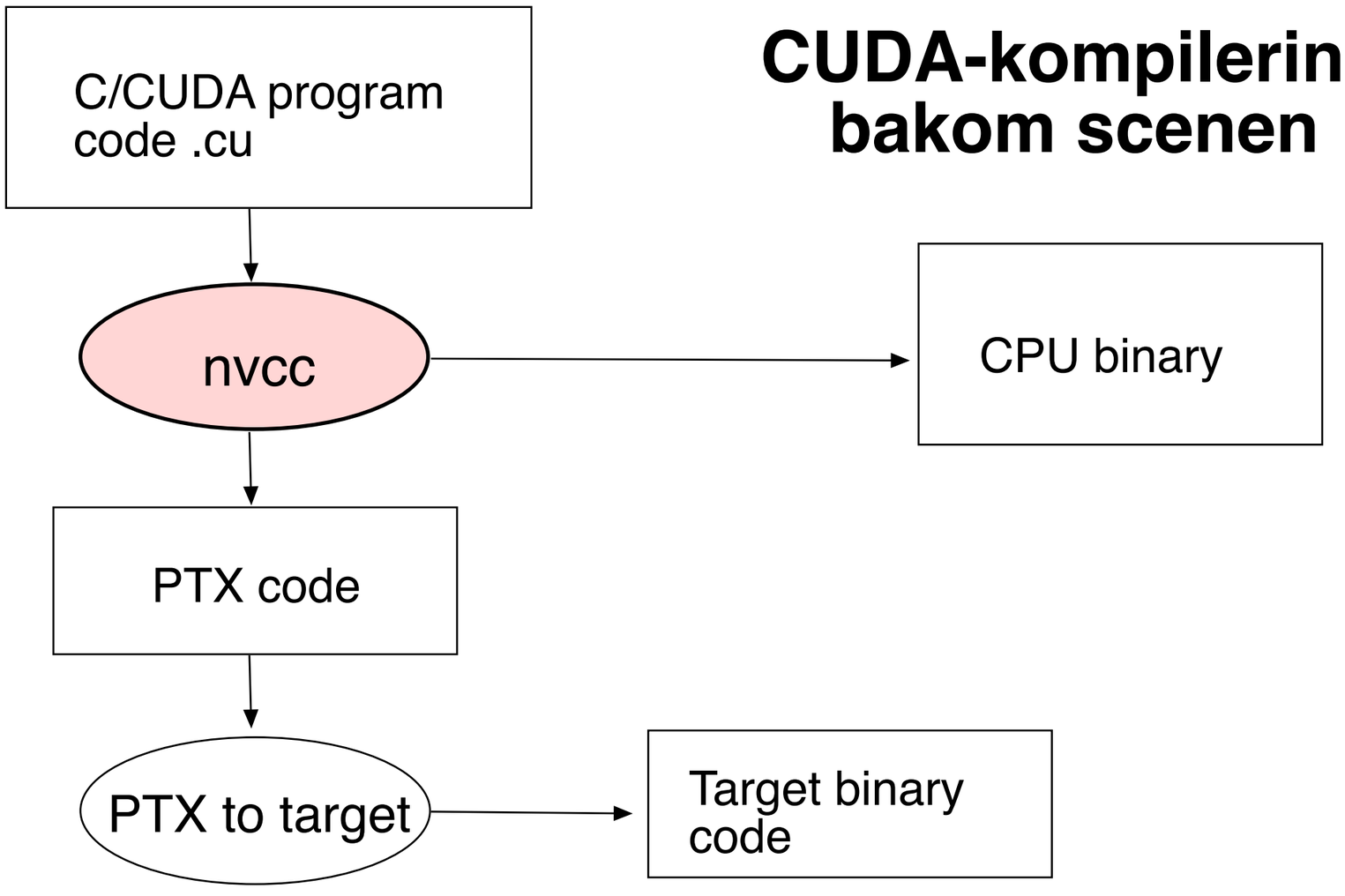
Enklast möjliga kommandorad:

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

(Fler parametrar finns för bibliotek mm.)

Kan länkas med C++-kod

CUDA-kompilering bakom scenen



Beräkningar med CUDA

Organisation och åtkomst

Blocks, threads...

Warps

En warp är det minsta antal trådar som processas parallellt i en CUDA-kapabel enhet. Detta antal är satt till 32.

Vi behöver normalt inte tänka så mycket på warps utan i första hand tänka på block och threads.

Processing organization

1 warp = 32 threads

1 kernel - 1 grid

1 grid - many blocks

1 block - 1 streaming multiprocessor (SM)

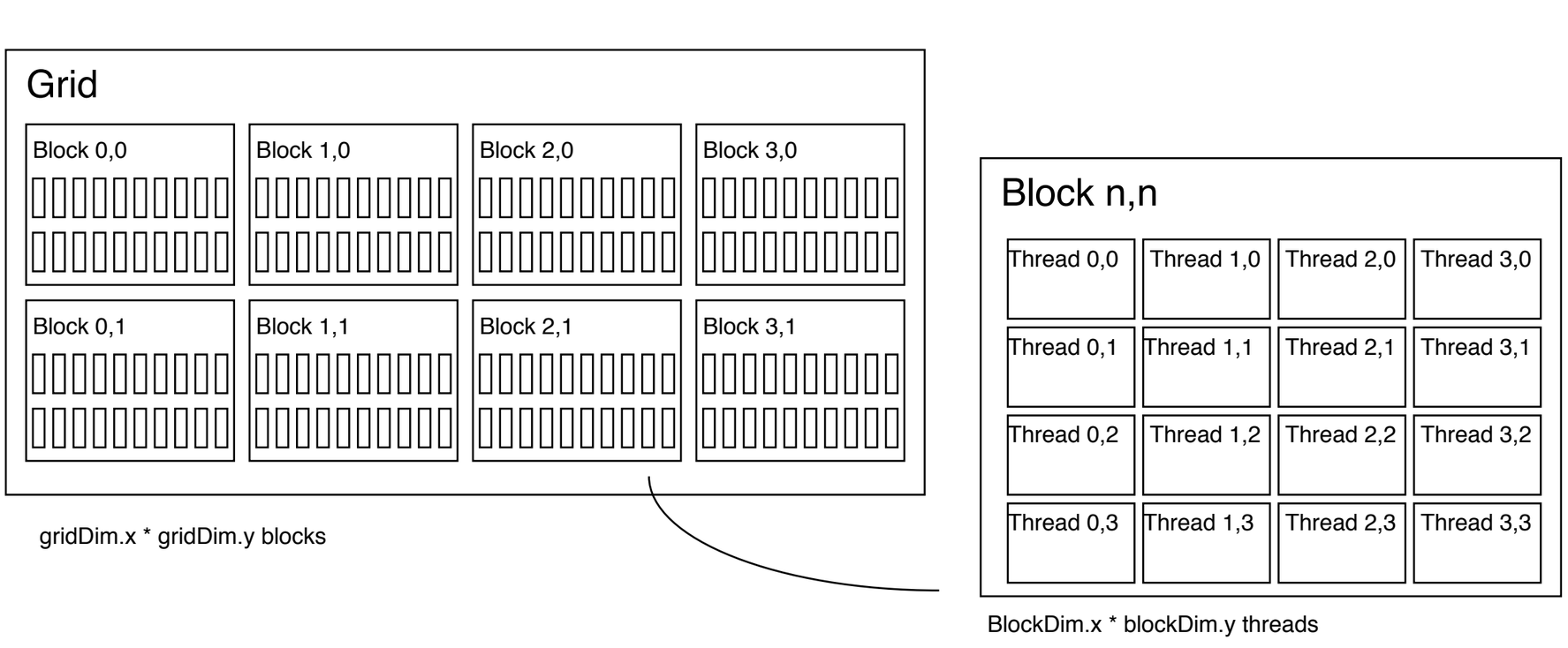
1 block - many threads

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



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];
}
```

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

Ordnade minnesaccesser (Coalescing)!

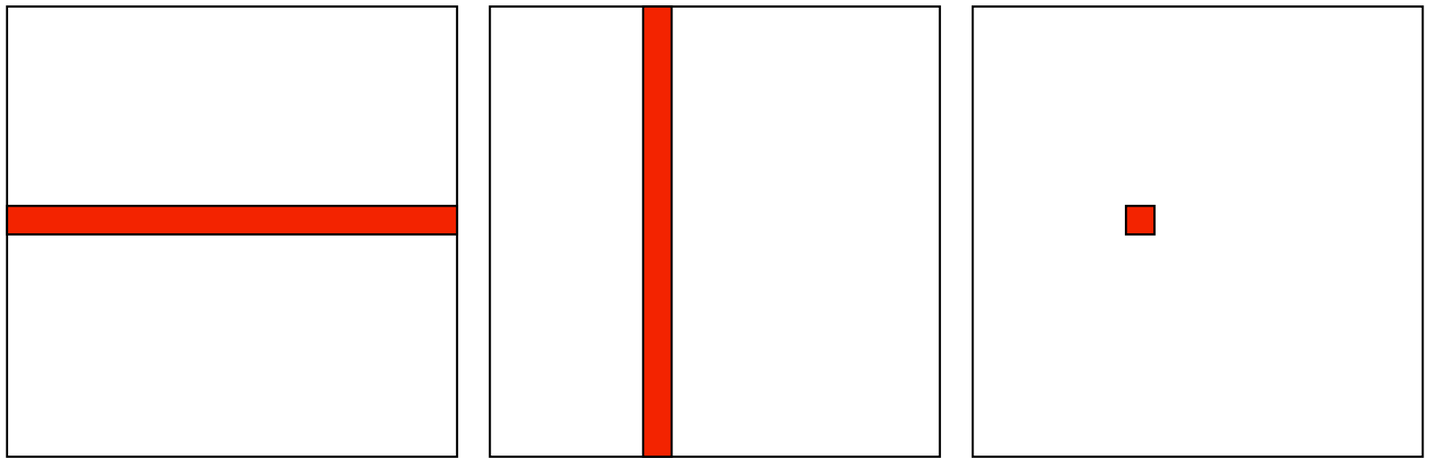
Kontinuerligt
Starta på 2-potens-adress
Addressera enligt trådnumrering

Använd shared memory för att omorganisera data!

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

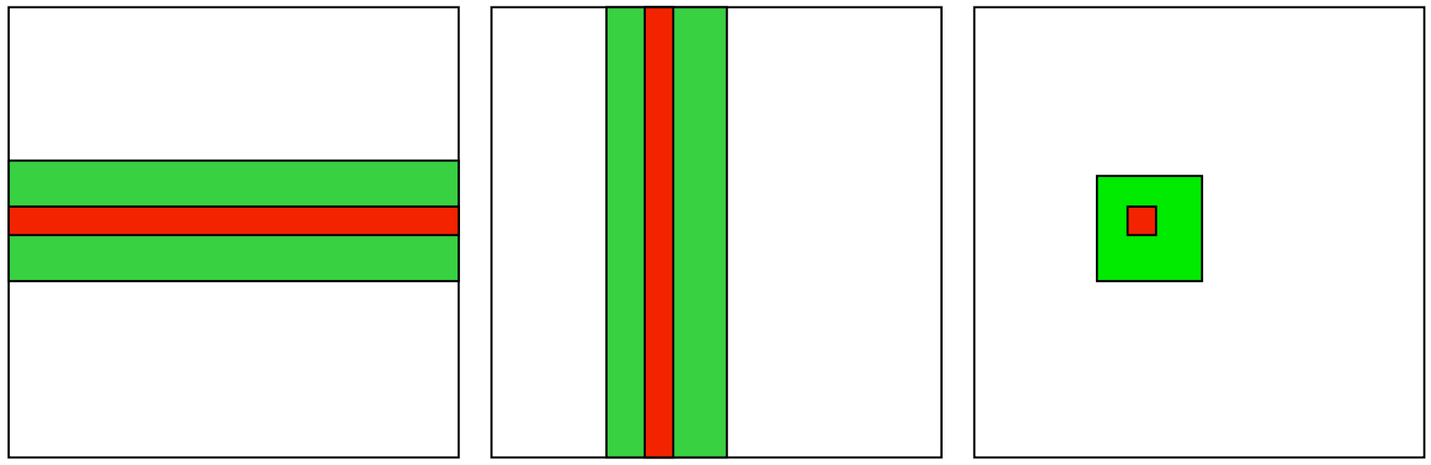
Matrismultiplikation



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

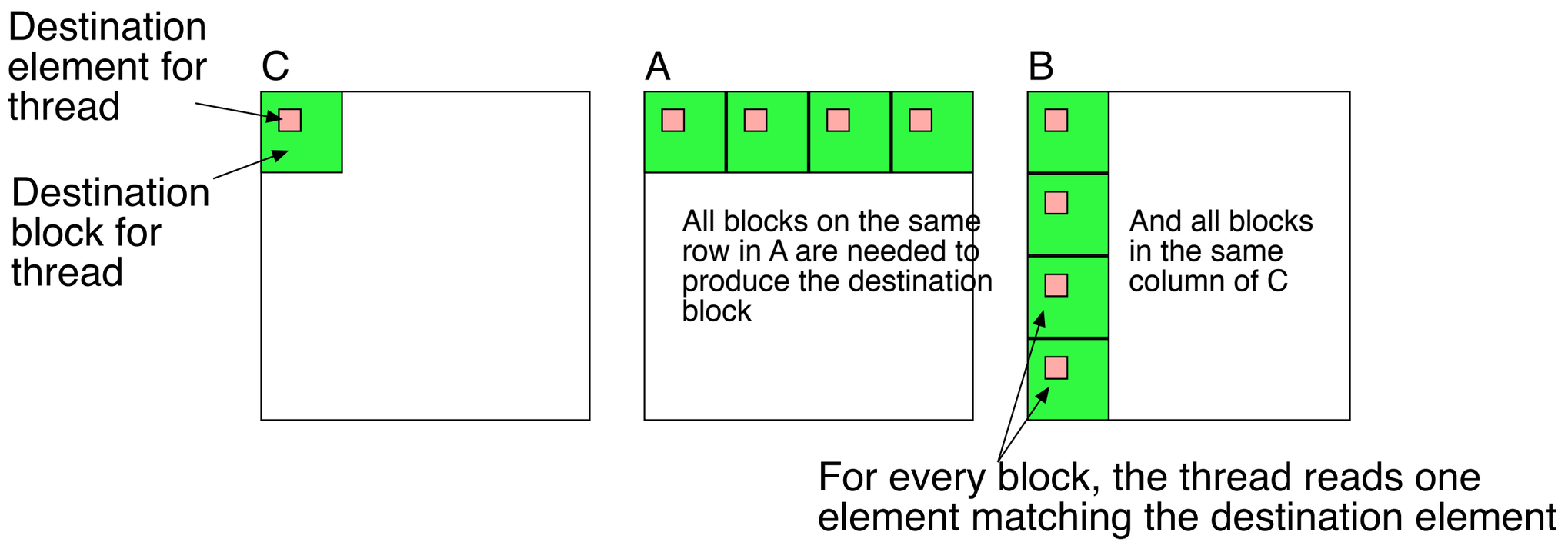
Naiv implementation: $2N^3$ globala minnesaccesser!

Matrismultiplikation

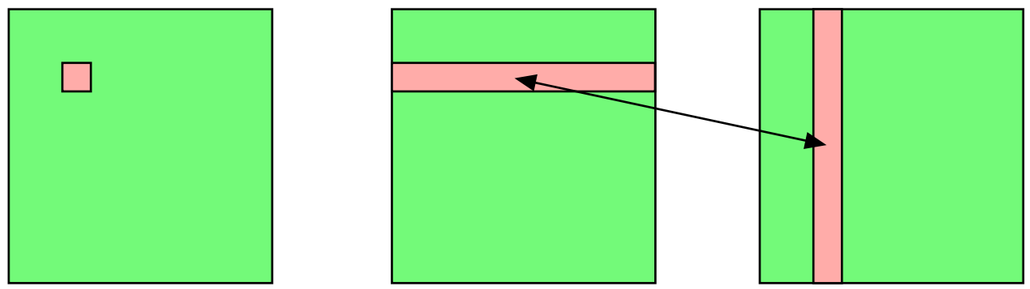


Låt varje block generera en del av utdata

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



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)!

Modifierad beräkningsmodell:

Ladda upp data till globalt GPU-minne

För valda delar:

Ladda upp delar av data till shared memory

Processa partiella data

Skriv partiella data till globalt minne

Läs ner resultatet till värden

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.

Accelering med coalescing

Minnesöverföringar kan vara 10x snabbare om man kan addressera enligt coalescing-reglerna!

Exempel: Matristransponering

Inga beräkingar!

Enbart minnesaccesser.

Matristransponeing

Naiv implementation

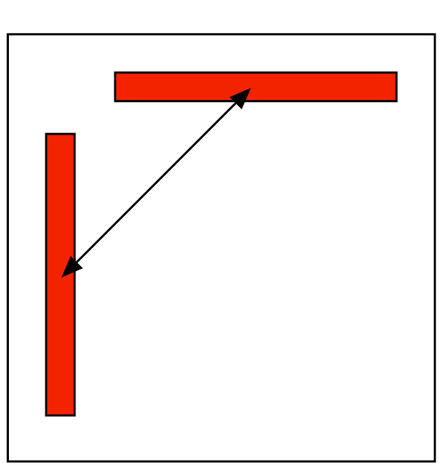
```
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height)
   {
       unsigned int index_in  = xIndex + width * yIndex;
       unsigned int index_out = yIndex + height * xIndex;
       odata[index_out] = idata[index_in];
   }
}
```

Hur kan detta vara dåligt?

Matristransponeing

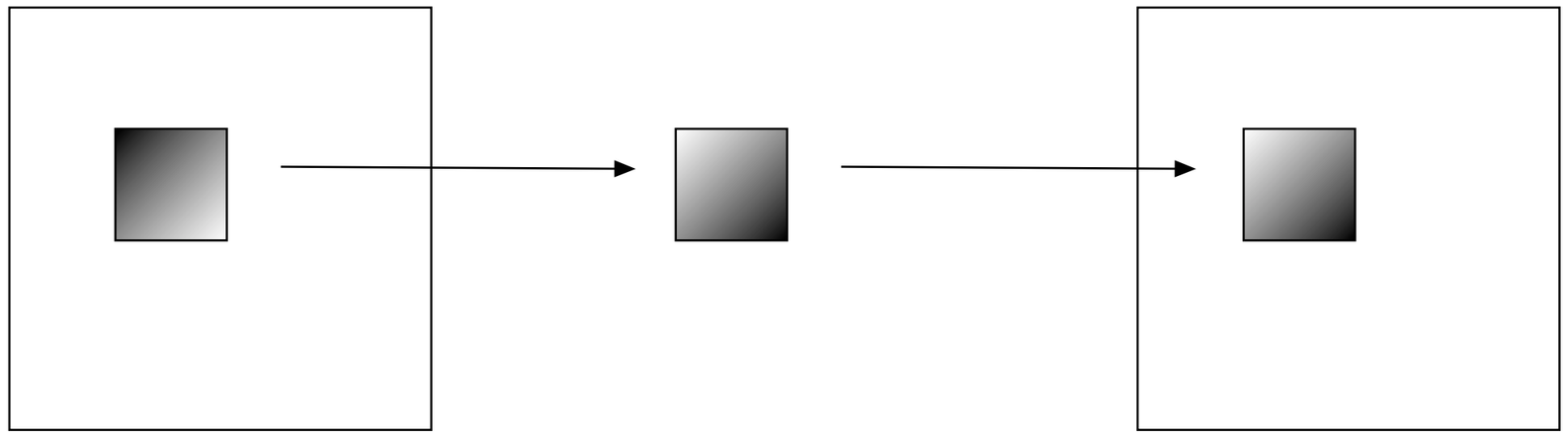
Coalescing-problem



Rad för rad och kolum för kolumn.
Kolumnvis access är non-coalesced!

Matristransponering

Lösning med coalescing



Läs från globalt minne till shared memory

Läs i ordning från globala minnet, godtycklig ordning till shared

Skriv till globalt minne

Skriv i ordning till globalt minne, godtycklig ordning från shadred

Bättre matristransponering med CUDA

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {
        unsigned int index_in = yIndex * width + xIndex;
        block[threadIdx.y][threadIdx.x] = idata[index_in];
    }

    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width))
    {
        unsigned int index_out = yIndex * height + xIndex;
        odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Shared memory for temporary storage

Read data to temporary buffer

Write data to tglobal memory

Tumregler för coalescing

- Starta på multipel av 64
- Addressera i ordning enligt trådnummer
- Man hoppa över vissa om man vill.
- Data bör vara i block om 4, 8 eller 16 bytes

OpenCL

OpenCL är en nyare (2009) lösning för GPU Computing

Motdrag mot CUDA

Uppbackat av Apple - då

Gjort för att ge större frihet i hårdvara

Inbyggt i MacOSX sedan 10.6

Numera deprecated av Apple

Har inte CUDAs eleganta integration

Var kan jag köra CUDA?

Alla NVidia efter 8800 (≈alla!)

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

En hyfsat modern Mac är perfekt för OpenCL. (Förinstallerat.)
Dock viss varning för att Apple fasar ut det.

Olympen + Asgård

GLSL, CUDA eller OpenCL?

- GLSL är överlägset mest portabelt och lättast att installera.
- CUDA är elegant och integrerat men kräver specialinstallation och är kräset på hårdvara.
- OpenCL är portabelt, kräver specialinstallation överallt utom på OSX (hittills).

=> räkna inte ut GPGPU på shaders än!

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?