CUDA

En lösning för generella beräkningar.

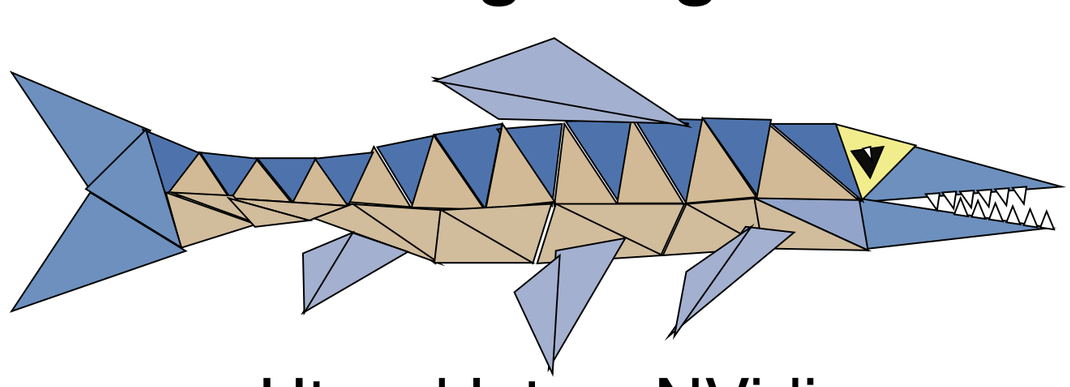
En introduktion:

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)
{
   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

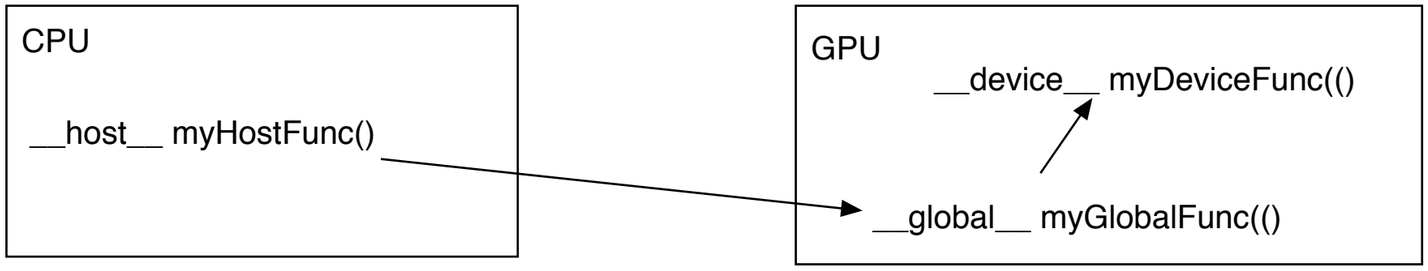
Modifierare för kodtyper

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

Indexera data med thread/block-IDs

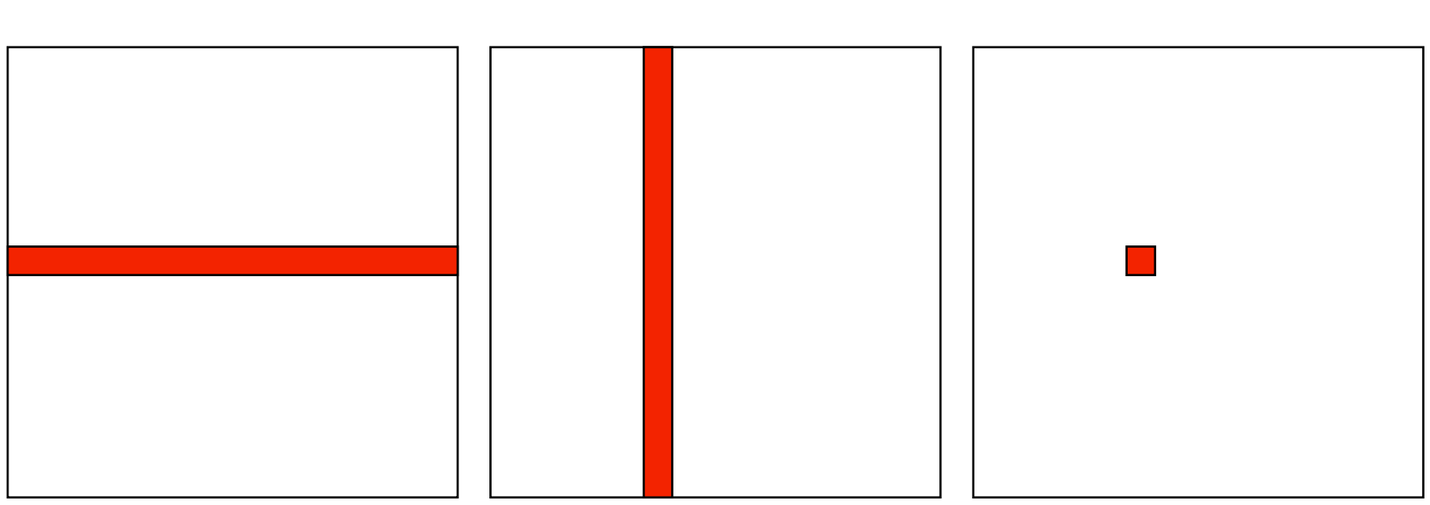
Beräkna index via blockIdx, blockDim, threadIdx

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

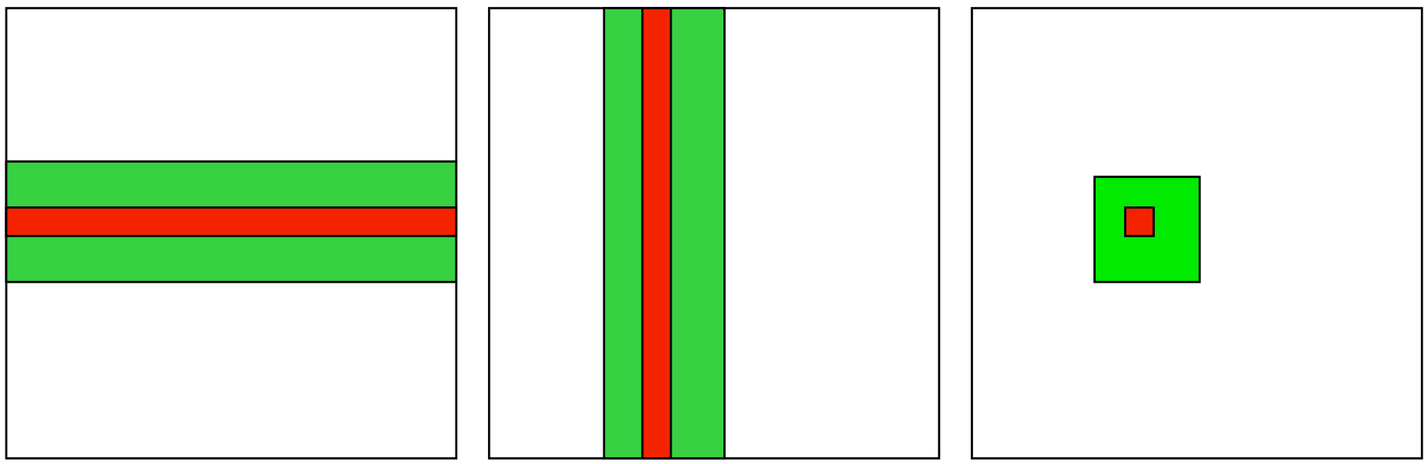
Mer shared memory!



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

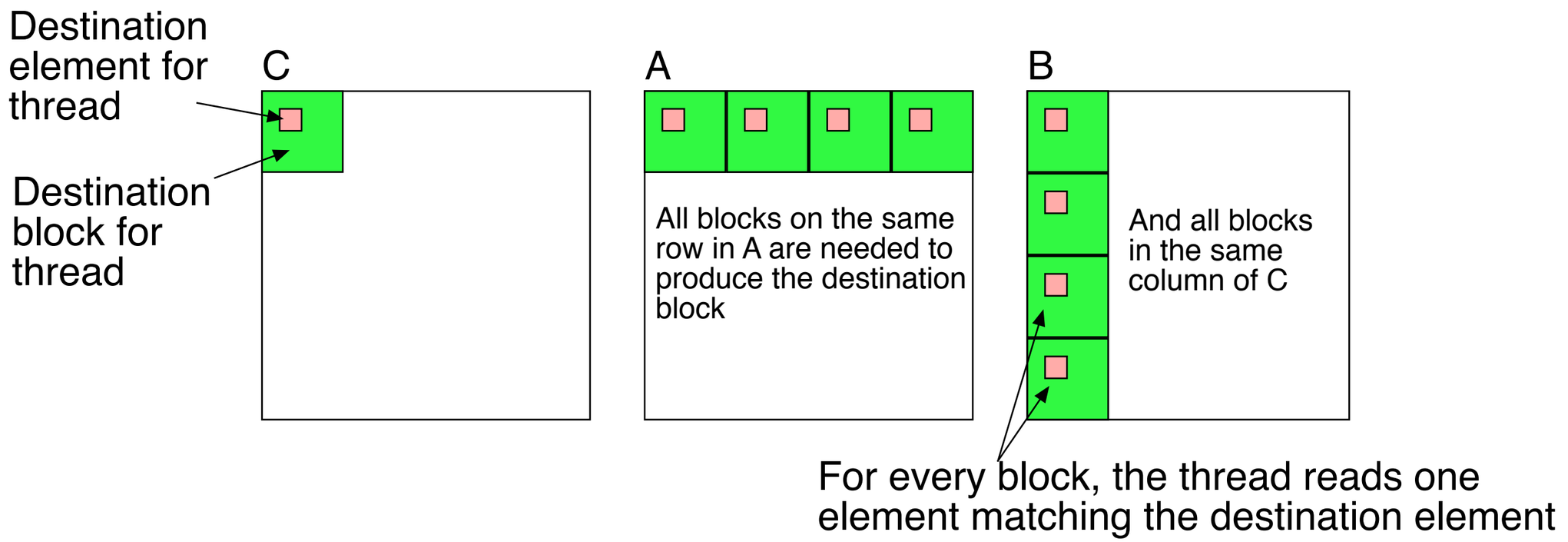
Naiv implementation: $2N^3$ globala minnesaccesser!

Dela upp matrismultiplikationen

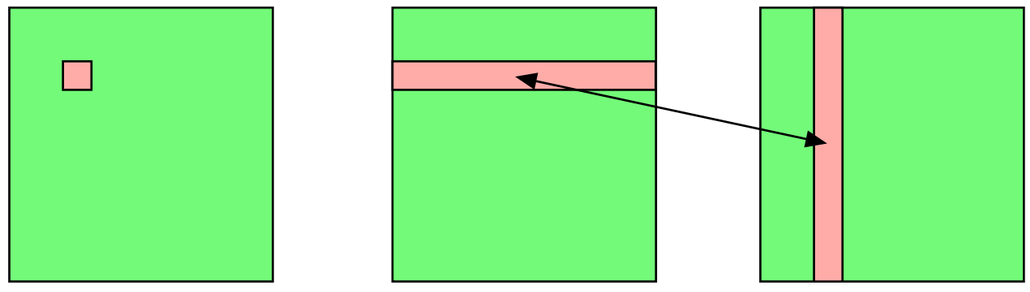


Låt varje block generera en del av utdata

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



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?