Aufgabe 06 -

# CUDA/MPI - Grayscaling und Embossing

Ilona Eisenbraun (s0561762)
Thomas Diewald (s0554334)

Fachbereich 4: Angewandte Informatik (M)
Programmierkonzepte und Algorithmen

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

# Agenda

1. Einführung
2. Lösungsansatz
3. Ergebnisse
4. Performance-Betrachtung
5. Fazit



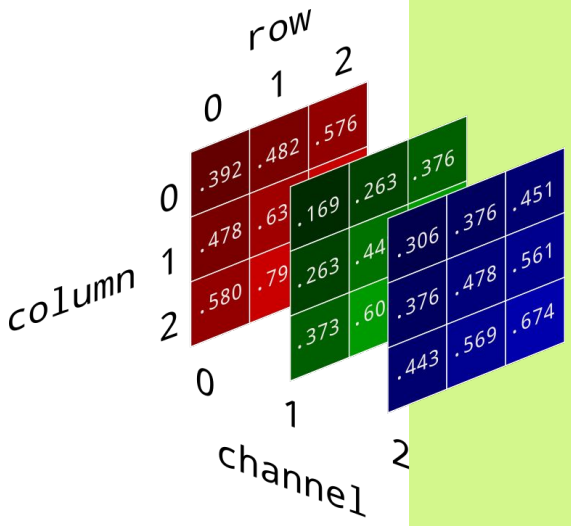
[2,3]

# Einführung

„Verteilung eines Bildes auf mehrere Prozesse durch MPI, Implementation der Farbraumkonvertierung von RGB in Graustufen und Anwenden des Emboss-Algorithmus mit CUDA“

# Grayscaling

- Prozess zur erstellung eines Grauwertbildes
- Formel zur Bestimmung des Grautons = Rot*0,21+Grün*0,72+Blau *0,07




[4]

# Embossing

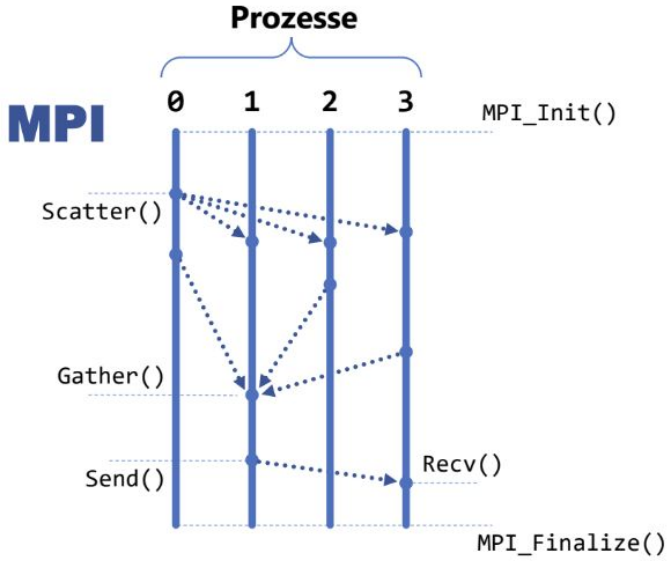
- Erkennung von Kanten anhand von Nachbarschaftsintensitäten
- stark kernelabhängig

[5]

# Lösungsansatz

# MPI: Message-Passing-Interface

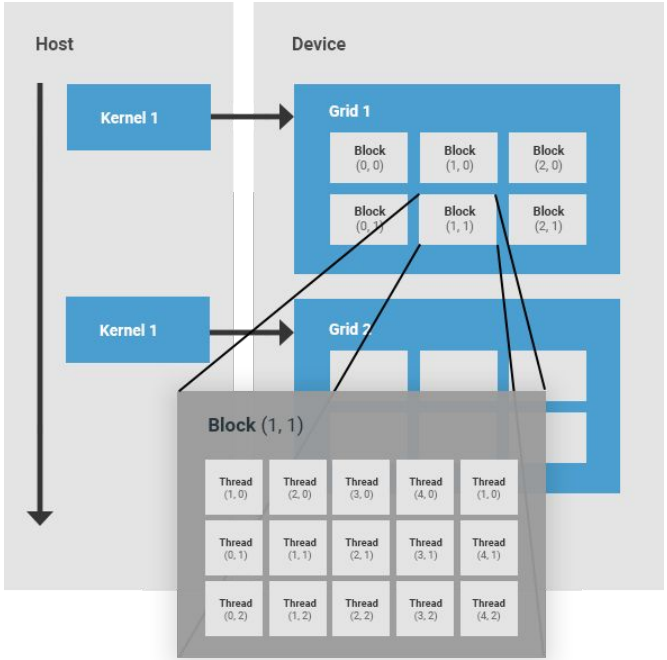
- Datenaustausch in Systemen mit verteiltem Speicher
- Anders als OpenMP: Ausführen mehrerer parallelisierter Prozesse
- Point-to-Point und Broadcasting
- Geeignet für Cluster-Berechnungen




[6]

# CUDA:
# Compute Unified Device Architecture

- Plattform für Parallelisierung von GPU-Berechnungskerneln
- Aufteilung der Threads eines Kernels in Blöcke
- Gruppierung der Blöcke im Grid




[7]

# Parallelisierung mit MPI

distributed memory system

- Initialisierung
- Rang des Prozesses
- Anzahl Prozesse im Kommunikator
- Barrier zur Synchronisation

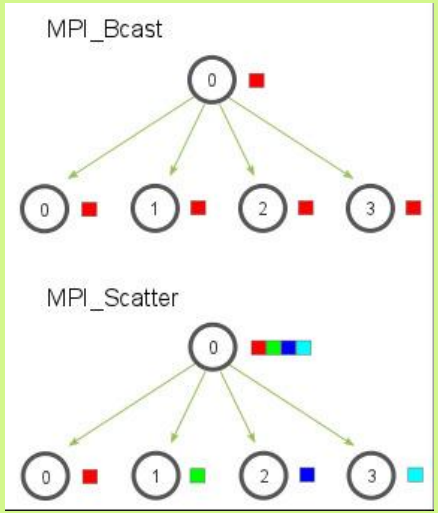
```
int main(int argc, char **argv)
{
        int size, rank;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Barrier(MPI_COMM_WORLD);
        int root = 0;
        cv::Mat blanksliceemboss;
        cv::Mat sendslice;
        cv::Mat blankslicegrey;
        int rowstep;
        int colstep;
        int fullnewcol = 0;
        int fullnewrow = 0;
```

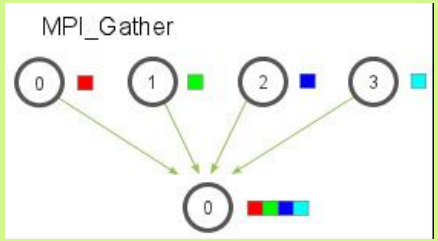
# Parallelisierung mit MPI

**In unseren Programmen wird ausschließlich globale Kommunikation verwendet.**

**Dies bedeutet:**
**der Prozess beginnt mit der Bearbeitung sobald er alle benötigten Daten hat.**

**Der Prozess endet nachdem er seinen Teil abgearbeitet hat.**




[8]

# Parallelisierung mit MPI

Broadcast benötigter Variablen an alle Prozesse

Verteilung auf die Prozesse

Bündeln der Berechnungsergebnisse aller Prozesse

```
//VERTEILEN DER PARAMETER WITH BROADCAST TO ALL PROCESSES
MPI_Bcast(&rowstep, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&colstep, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&fullnewcol, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&fullnewrow, 1, MPI_INT, root, MPI_COMM_WORLD);
cv::Mat mat = cv::Mat(fullnewcol, rowstep, CV_8UC4);
MPI_Scatter(sendslice.data,
            fullnewrow * colstep * 4,
            MPI_BYTE,
            mat.data,
            fullnewrow * colstep * 4,
            MPI_BYTE,
            root,
            MPI_COMM_WORLD);
cv::Mat gray(fullnewcol, rowstep, CV_8UC4);
cv::Mat emboss(fullnewcol, rowstep, CV_8UC4);
convert(mat, gray, true);
//Hier soll das Grayscale Bild eingesammelt werdedn
MPI_Gather(gray.data,
           fullnewrow * colstep * 4,
           MPI_BYTE,
           blankslicegrey.data,
           fullnewrow * colstep * 4,
           MPI_BYTE,
           root,
           MPI_COMM_WORLD);
```

# GPU-Parallelisierung mit CUDA

```
void convert(const cv::Mat& input, cv::Mat& output,bool flag) {
    // Calculate total number of bytes of input and output image
    const int colorBytes = input.step * input.rows;
    const int grayBytes = output.step * output.rows;
    unsigned int* d_kernel;
    unsigned char* d_input, * d_output;
    // Allocate device memory
    SAFE_CALL(cudaMalloc<unsigned char>(&d_input, colorBytes), "CUDA Malloc Failed");
    SAFE_CALL(cudaMalloc<unsigned char>(&d_output, grayBytes), "CUDA Malloc Failed");
    // Copy data from OpenCV input image to device memory
    SAFE_CALL(cudaMemcpy(d_input, input.ptr(), colorBytes, cudaMemcpyHostToDevice), "CUDA Memcpy Host To Device Failed");
    // Threads per Block
    const dim3 block(32, 32);
    // Calculate grid size to cover the whole image
    const dim3 grid((input.cols + block.x - 1) / block.x, (input.rows + block.y - 1) / block.y);
    // Launch the color conversion kernel
    if(flag ==true){
    grayscale_kernel << <grid, block >> > (d_input, d_output, input.cols, input.rows, input.step, output.step);
    }else {
    emboss_kernel << <grid, block >> > (d_input, d_output, input.cols, input.rows, input.step, output.step);
    }
    // Synchronize to check for any kernel launch errors
    SAFE_CALL(cudaDeviceSynchronize(), "Kernel Launch Failed");

    // Copy back data from destination device meory to OpenCV output image
    SAFE_CALL(cudaMemcpy(output.ptr(), d_output, grayBytes, cudaMemcpyDeviceToHost), "CUDA Memcpy Host To Device Failed");

    // Free the device memory
    SAFE_CALL(cudaFree(d_input), "CUDA Free Failed");
    SAFE_CALL(cudaFree(d_output), "CUDA Free Failed");
}
```

Speicherzuweisung

Auslagerung auf GPU

Aufteilung auf Berechnungen

Synchronisation

Kopieren auf Host

Speicherfreigabe

# GPU-Parallelisierung mit CUDA

Funktion global setzen

Konvertierung in Graustufen

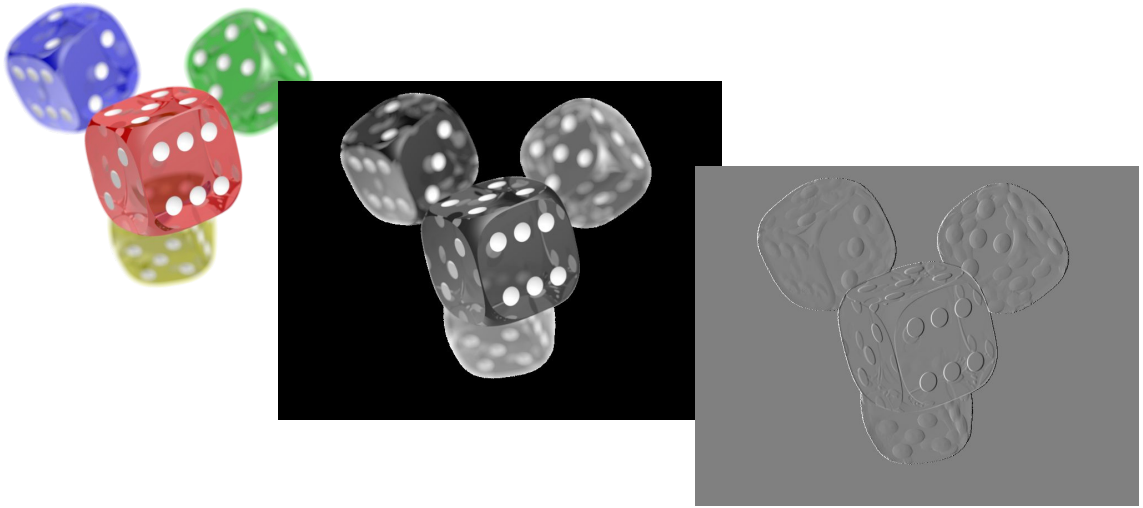
```
__global__ void grayscale_kernel(unsigned char* input, unsigned char* output,
    int width, int height, int colorWidthStep, int grayWidthStep) {
    {
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;
        if ((x < width) && (y < height))
        {
            //Loc base Image
            const int color_tid = y * colorWidthStep + (4 * x);

            //Loc in Grayscale
            const int gray_tid = y * colorWidthStep + (4 * x);

            const unsigned char blue = input[color_tid];
            const unsigned char green = input[color_tid + 1];
            const unsigned char red = input[color_tid + 2];
            const unsigned char alpha = input[color_tid + 3];
            const float gray = red * 0.21f + green * 0.72 + blue * 0.07f;

            output[gray_tid] = static_cast<unsigned char>(gray);
            output[gray_tid+1] = static_cast<unsigned char>(gray);
            output[gray_tid+2] = static_cast<unsigned char>(gray);
            output[gray_tid+3] = static_cast<unsigned char>(alpha);
        }
    }
}
```

# Ergebnisse



[9]

# Performance

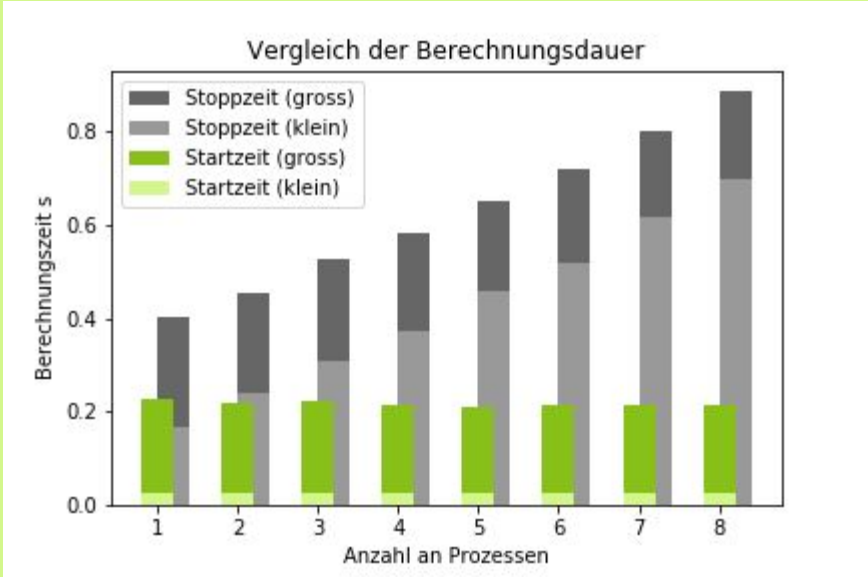
gemessen an:
**dice_micro** ( 439 x 389 px )
**dice_large** ( 1754 x 1554 px )

**Maschine Ilona:**

**8-Core FX-9590**

**32GB DDR 3 RAM 2209.7MHZ**

**Nvidia GTX-1060**



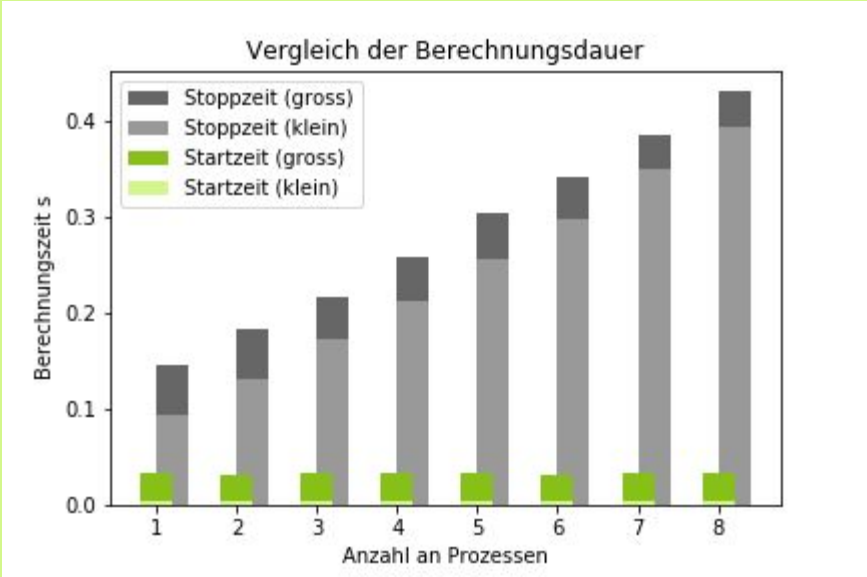
gemessen an:
**dice_micro** ( 439 x 389 px )
**dice_large** ( 1754 x 1554 px )

**Maschine Thomas:**

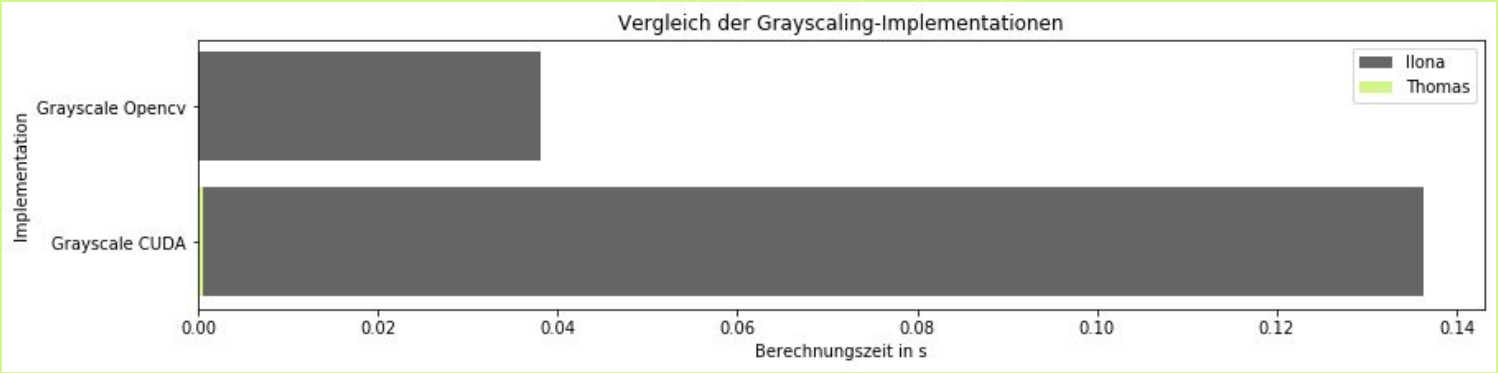
**Core I7 9700K**

**32GB DDR 4 RAM 3200MHZ**

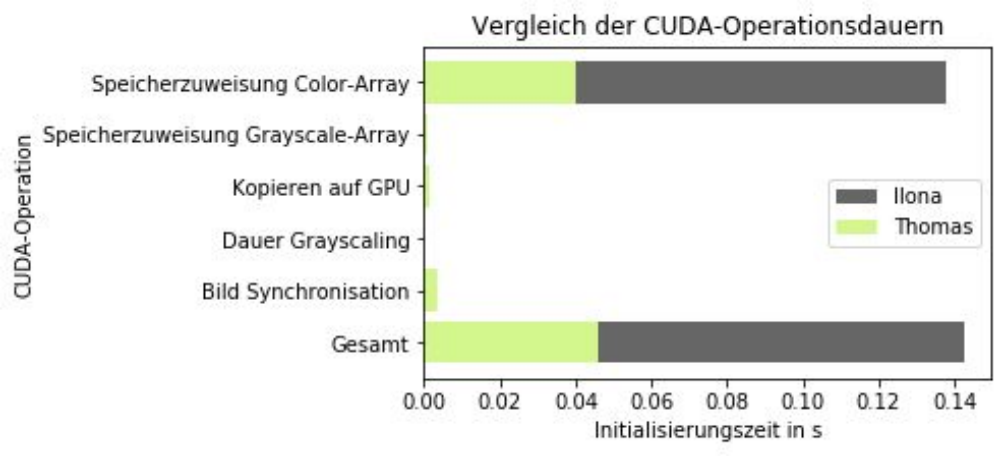
**Nvidia RTX 2080Ti**



gemessen an:
**dice_large** ( 1754 x 1554 px )



gemessen an:
**dice_large** ( 1754 x 1554 px )



# Fazit

> Your problem is that CUDA needs to initialize! It will always initialize for the first image and generally takes between 1-10 seconds, depending on the alignment of Jupiter and Mars. Now try this. Do the computation twice and then time them both. You will probably see in this case that the speeds are within the same order of magnitude, not 20.000x, that's ridiculous. Can you do something about this initialization? Nope, not that I know of. It's a snag.

# Was konnte man beobachten?

- Laufzeiten haben fast linear mit der Anzahl an Prozessen zugenommen
- Initialisierungen für Buffer nehmen gleich viel Zeit in Anspruch

- Die CUDA-Initialisierungen sind aufwendig und daher plain OpenCV unterlegen
- **ABER: für größere Operationen wäre CUDA von Vorteil**

# Vielen Dank!

## Fragen?

# Quellen


[1] - HTW Logo : https://corporatedesign.htw-berlin.de/logos/logo-htw-berlin/
[2] - CUDA Logo: https://upload.wikimedia.org/wikipedia/en/thumb/b/b9/Nvidia_CUDA_Logo.jpg/300px-Nvidia_CUDA_Logo.jpg
[3] - MPI Logo: https://repository-images.githubusercontent.com/181981725/1af35680-6a25-11e9-985e-a483461309fe
[4] - RGB Decomposition: https://e2eml.school/images/image_processing/three_d_array.png
[5] - Image Emboss: https://upload.wikimedia.org/wikipedia/commons/8/84/Emboss_example.jpg
[6] - MPI Dokumentation: https://www.dropbox.com/s/e8ekrdktgx8fl1t/ProgKo_02_MPI.pdf?dl=0
[7] - CUDA Execution Model: https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/execution-model
[8] - MPI Execution Scatter Gather: https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
[8] -MPI Execution Broadcast: https://nyu-cds.github.io/python-mpi/fig/04-broadcast.png
[9] - Dice Image: https://www.dropbox.com/s/5uhqu8m8a7entcd/images.zip?dl=0
[10]- Zitat: https://stackoverflow.com/questions/12074281/why-opencv-gpu-code-is-slower-than-cpu/16038287