Programmierkonzepte und Algorithmen

Florian Thom
Phillip Friedel

Struktur

1. Problembeschreibung
2. Software
3. Implementierung
    a. YCbCr
    b. Gaussian-Blur
4. Benchmarks
5. Probleme
6. Verbesserungen

Problembeschreibung

RGB

- Rot, Grün und Blau-Kanäle
- Keine gute Kompressionsmöglichkeiten

YCbCr

- YCbCr
  - Y -> Luma (Helligkeit)
  - Cb ->Blau-Komponente relativ zur Grün-Komponente
  - Cr -> Rot-Komponente relativ zur Grün-Komponente
- Bessere Kompression möglich

RGB zu YCbCr

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

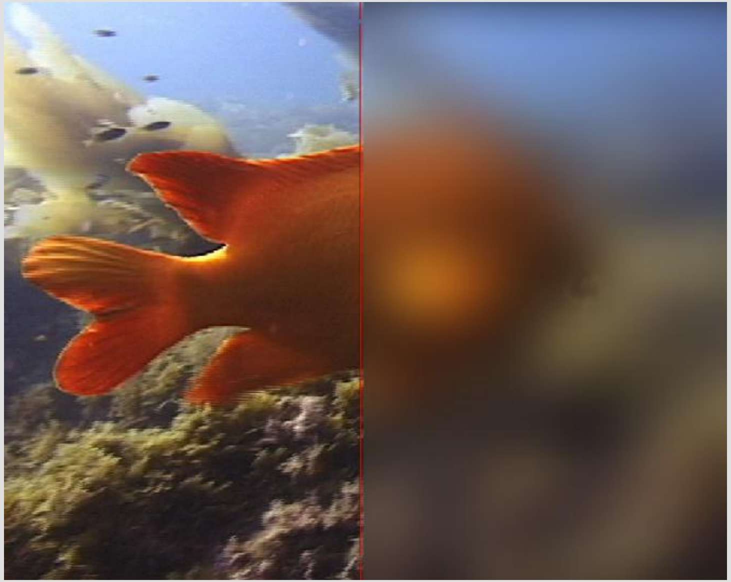
$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.714 \cdot (Cr - delta) - 0.344 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

Gaussian Blur

- Filter der oft genutzt wird um Bilder “weicher” zu machen und Noise zu reduzieren/eliminieren -> Verschwommenes Bild



Gaussian Blur

- Gaussian Kernel -> Matrix die mit der Gauß-Formel gefüllt wird
  - σ -> Grad der Verschwemmung
  - x -> Horizontale Entfernung zum mittleren Pixel
  - y -> Vertikale Entfernung zum mittleren Pixel

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$\frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Software

OpenCV

- "Open source" Bibliothek mit Funktionen für die Bildbearbeitung -> C, Python, Java, C++
- Bilder laden, anzuzeigen und abzuspeichern
- Ergebnisse mit OpenCV vergleichen -> Auswertung bezüglich Korrektheit und Laufzeit.

CUDA

- Nvidia -> Programmier-Technik zum Auslagern von Prozesse auf die Grafikkarte
- GPU erlaubt hohe Parallelisierung von Prozessen -> Schnellere Arbeitszeiten als mit CPU
- Voraussetzung -> Nvidia-Grafikkarte

Implementierung

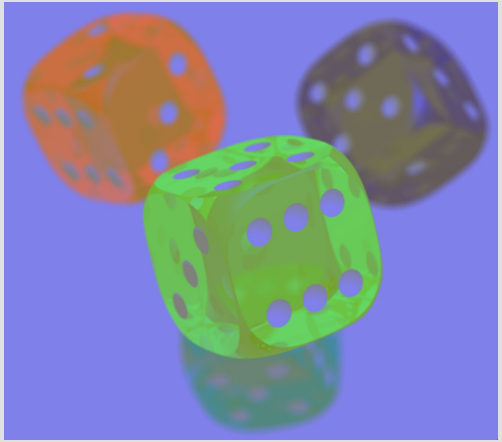
YCbCr

```cpp
cv::Mat colorConversionYCBCR(string image_name)
{
    int channels = 3;
    cv::Mat matBGR = readImageWithName(image_name);
    if (matBGR.channels() == 4)
    {
        matBGR = bgra2bgr(matBGR);
    }
    cv::Mat mat = convertMatBGRToRGB(matBGR);
    tuple<int, int> imageSize = getMatSize(mat);

    // so it DOES includes the channels
    int dataSize = get<0>(imageSize) * get<1>(imageSize) * channels;


    int maxThreadsInBlockX = 1024;
    dim3 blockDims(maxThreadsInBlockX, 1, 1);
    dim3 gridDims(ceil((dataSize / 3) / maxThreadsInBlockX), 1, 1);

    uchar* data = returnMatDataWithCharArray(mat);
    uchar* resultdata = convertRGBToYCBCR(data, dataSize, gridDims, blockDims);

    cv::Mat matResultYCRCB = returnMatFromCharArray(resultdata, imageSize);

    return matResultYCRCB;

}
```

```
unsigned char * convertRGBToYCBCR(unsigned char* data, int dataSize, dim3 gridDims, dim3 blockDims)
{
    unsigned char* dataResult = (unsigned char*) malloc(sizeof(unsigned char) * dataSize);

    unsigned char* dev_data;
    unsigned char* dev_dataResult;
    cudaMalloc(&dev_data, sizeof(unsigned char) * dataSize);
    cudaMalloc(&dev_dataResult, sizeof(unsigned char) * dataSize);

    cudaMemcpy(dev_data, data, sizeof(unsigned char) * dataSize, cudaMemcpyHostToDevice);

    dev_convertColorSpace <<< gridDims, blockDims >>> (dev_data, dev_dataResult, dataSize);

    cudaMemcpy(dataResult, dev_dataResult, sizeof(unsigned char) * dataSize, cudaMemcpyDeviceToHost);

    cudaFree(&dev_data);
    cudaFree(&dev_dataResult);
    cudaDeviceReset();

    return dataResult;
}
```

```
// rgb->Ycbcr
__global__ void dev_convertColorSpace(unsigned char* dev_data, unsigned char* dev_dataResult, int dataSize)
{
    int channels = 3;
    int globalThreadId = (blockIdx.x * channels) * blockDim.x + (threadIdx.x*channels);

    // grid-stride loop
    for(int dataElement = globalThreadId; dataElement < (dataSize-(channels)); dataElement= dataElement+(gridDim.x * blockDim.x)*(channels))
    {
        unsigned char r = dev_data[dataElement + 0];
        unsigned char g = dev_data[dataElement +1];
        unsigned char b = dev_data[dataElement +2];

        dev_dataResult[dataElement + 0] = 16 + (((r << 6) + (r << 1) + (g << 7) + g + (b << 4) + (b << 3) + b) >> 8); // Y
        dev_dataResult[dataElement + 1] = 128 + (((r << 7) - (r << 4) - ((g << 6) + (g << 5) - (g << 1)) - ((b << 4) + (b << 1))) >> 8); // Cb
        dev_dataResult[dataElement + 2] = 128 + ((-((r << 5) + (r << 2) + (r << 1)) - ((g << 6) + (g << 3) + (g << 1)) + (b << 7) - (b << 4)) >> 8); // Cr
    }
}
```

Input



Output

Differenz

Gaussian-Blur

Gaussian-Blur

Funktionsdefinition in “main.cpp”

- Zweck: Main für Gauss in “.cpp” - Umgebung
  - Definition Block- und Grid-Dim
  - Aufruf der Methode “applyGaussianFilter(...)” in “cuda.cu”

```
int maxThreadsInBlockX = sqrt(1024);
int maxThreadsInBlockY = sqrt(1024);
dim3 blockDims = dim3(maxThreadsInBlockX, maxThreadsInBlockY,1);
dim3 gridDims = dim3(ceil((newImageHeight*channels) / (float) maxThreadsInBlockY),
                    ceil((newImageWidth*channels) / (float) maxThreadsInBlockX),
                    1);

uchar* tmp1 = returnMatDataWithCharArray(matBGR);
uchar* resultdata = applyGaussianFilter(tmp1, dataSize, gridDims, blockDims,
                                        channels, get<1>(imageSize), get<0>(imageSize),
                                        filterHeight, sigma);

cv::Mat resultMat = returnMatFromCharArray(resultdata, imageSizeResultImage);
```

Gaussian-Blur

“cuda.cu”: “applyGaussianFilter(...)”

- Zweck: Main für Gauss in “.cu” -Umgebung
  - Filtererstellung
  - Datenübergabe an “gaussianAllChannel(...)” zur weiteren Bearbeitung

```
unsigned char* applyGaussianFilter(unsigned char* data, int dataSize, dim3 gridDims,
                                   dim3 blockDims, const int channelsPara, int imageHeight,
                                   int imageWidth, int filterHeight, double sigma)
{
    double* filter = createGaussianFilter(filterHeight, filterHeight, sigma);
    unsigned char* resultData = gaussianAllChannel(data, dataSize, gridDims, blockDims,
                                                   filter, imageHeight, imageWidth, filterHeight);
    return resultData;
}
```

Gaussian-Blur

“cuda.cu”: “applyGaussianFilter(...)” ➔ “createGaussianFilter(...)”

- Zweck: Filtererstellung
  - Kernel - Erstellung
    - Höhe == Breite
    - Sigma variabel
  - Kernel - Normalisierung
    - Mapping auf W=[0;1]
  - Kernel - Flatten
    - kernel[h][h] ➔ kernel[h*h]

```
for (int x = -height/2; x <= height/2; x++) {
    for (int y = -height/2; y <= height/2; y++) {
        r = sqrt(x * x + y * y);
        kernel[x + height/2][y + height/2] = (exp(-(r * r) / s)) / (PI * s);
        sum += kernel[x + height/2][y + height/2];
    }
}


// normalising the Kernel
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < height; ++j) {
        kernel[i][j] /= sum;
    }
}

double* kernelFlat = (double*)malloc(height * height * sizeof(double));

for (int h = 0; h < height; h++){
    for (int w = 0; w < height; w++){
        kernelFlat[h * height + w] = kernel[h][w]; // y*width+width_pos
    }
}
```

Gaussian-Blur

“cuda.cu”: “createGaussianFilter(...)” ➔ “gaussianAllChannel(...)”

- Zweck: Managen des Kernel-Calls
    - Device Variablen deklarieren
    - Speicher-Allokieren
    - Datenübertragung an Device
    - Kernel-Call

```
dev_applyGaussianALL << < gridDims, blockDims >> > (dev_data, dev_dataResult, dev_filter, dataSize, imageHeight, imageWidth, filterHeight);
```

    - Datenübertragung an Host
    - Speicherfreigabe

Gaussian-Blur

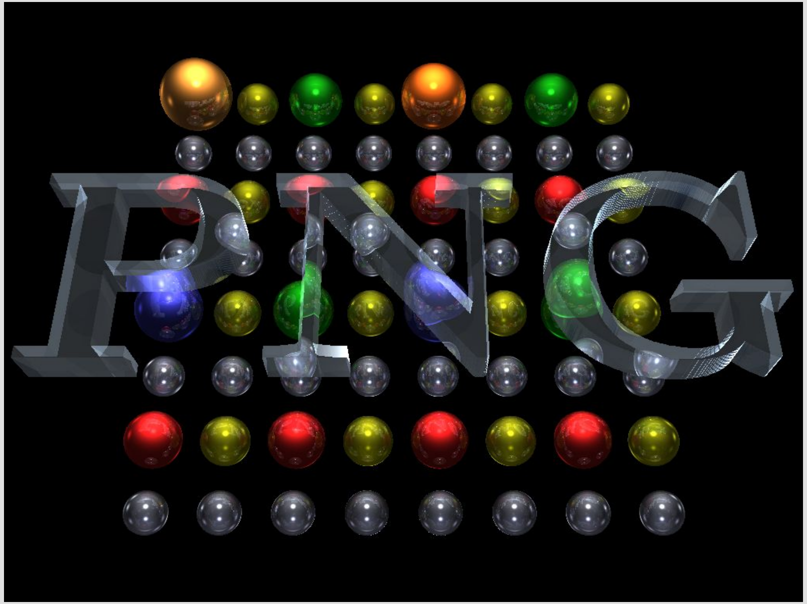
“cuda.cu”: … ➜ “gaussianAllChannel(...)” ➜ “dev_applyGaussianAll(...)”

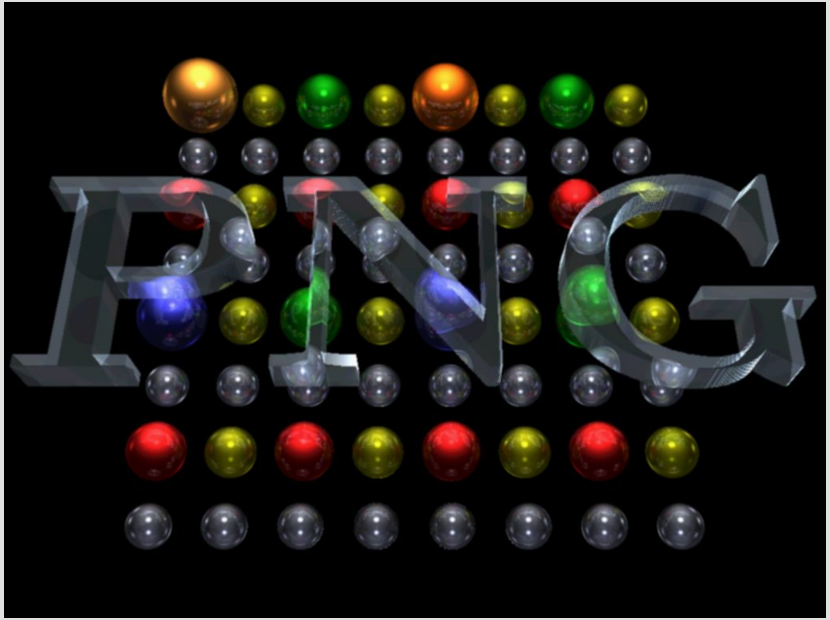
- Zweck: Parallelisierte Convolution des Bildes
  - Berechnung der “Global-ID”
  - Anwendung des Gauss-Algorithmus

```
__global__ void dev_applyGaussianALL(unsigned char* dev_data, unsigned char* dev_dataResult, double* filter, int dataSize, int imageHeight, int imageWidth, int filterHeight)
{
    int channels = 3;

    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;
    int currentIndex = threadId;

    int imageYSource = currentIndex / (channels * imageWidth);
    int imageXSource = currentIndex % (channels * imageWidth);

    int cuttedAway = filterHeight / 2;

    int newImageHeight = imageHeight - filterHeight + 1;
    int newImageWidth = imageWidth - filterHeight + 1;

    int currentChannel = currentIndex % channels;

    if (!(imageYSource < cuttedAway || imageYSource >(imageHeight - 1 - cuttedAway) || imageXSource < cuttedAway * channels || imageXSource >(imageWidth * channels - 1 - cuttedAway * channels))){
        ////height
        for (int h = 0; h < filterHeight; h++){
            //width
            for (int w = 0; w < filterHeight; w++){
                double tmp = filter[h * filterHeight + w] * dev_data[((imageYSource + h) * (channels * imageWidth) + (imageXSource + (channels * w)))];
                dev_dataResult[((imageYSource - cuttedAway) * (channels * newImageWidth) + (imageXSource - channels * cuttedAway))] += tmp;

            }
        }
    }
}
```

Gaussian-Blur

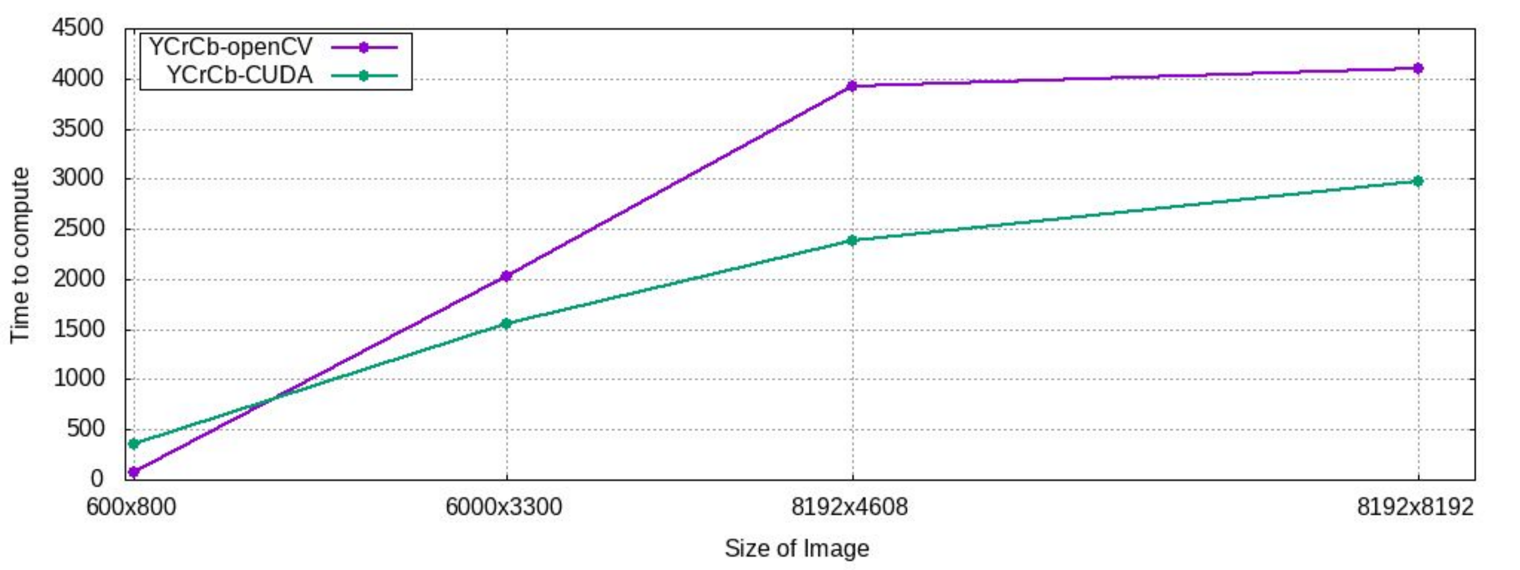
Ergebnis





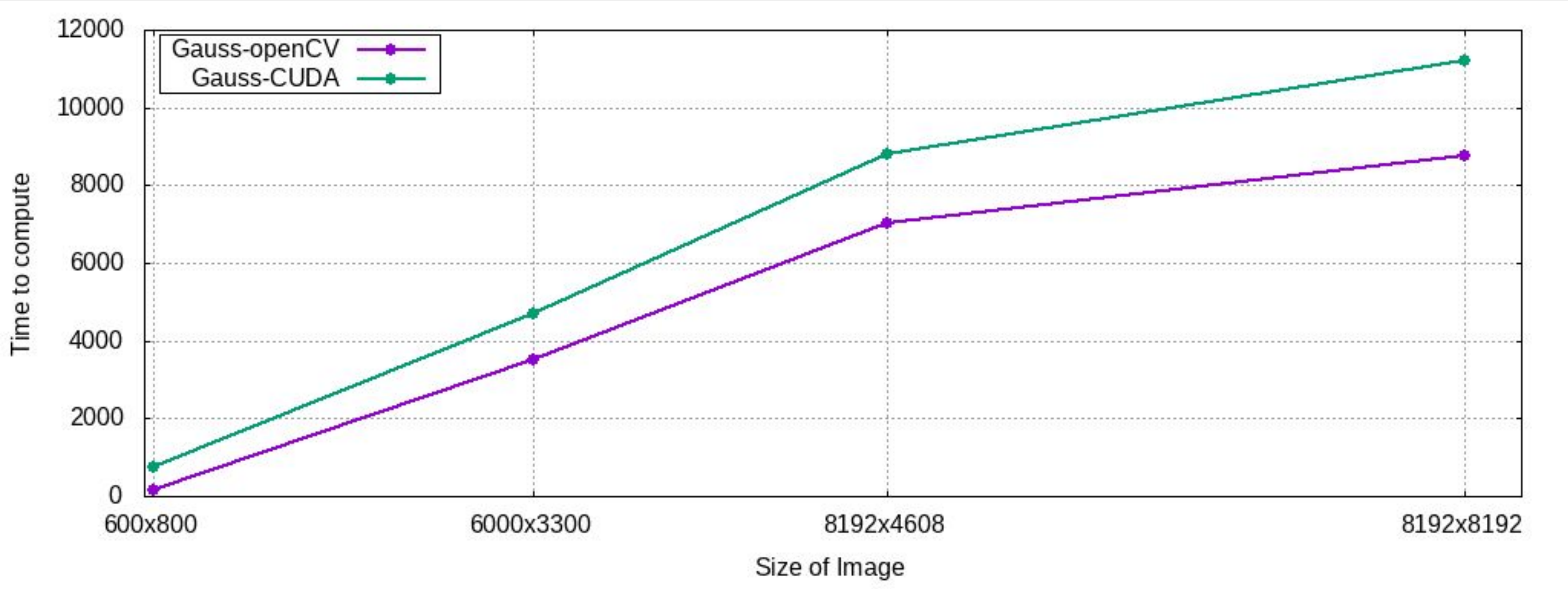
Benchmarks

Benchmark YCbCr



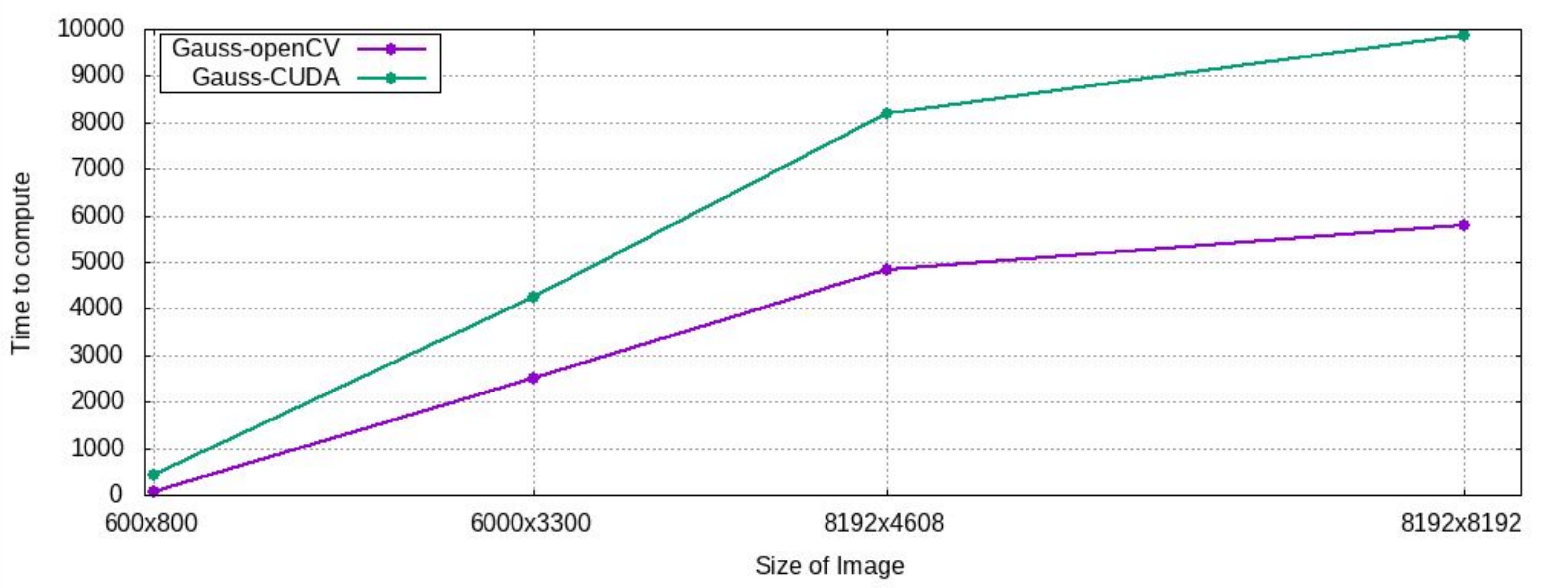
Benchmark Gauss (1)

- Vor Optimierung



Benchmark Gauss (2)

- Nach Optimierung



Probleme

Watch Dog Timer

- Verwendung Grafikkarte ohne Grafik-Output
- Benutzung von Linux

C++

- Beispiel:

```
int arraySize = 10;
int* tmp = (int*) malloc(sizeof(int) * arraySize);
int* tmp = (int) tmp[10]
```

Debugging Cuda

- Nsight Monitor mit der Nsight Visual Studio Edition

Verbesserungen

- Cuda Unified Memory (managed) (Cuda 6+)
- Cuda-Async
- Cuda-Streams (unabhängig von Stream 0)
- Cuda Shared Memory

Vielen Dank für die Aufmerksamkeit