Stereo Vision

Max Noppel

07.April 2018, vspace.one

Hintergrund

- Praktikum Multikernprogrammierung am KIT
- OpenMP, LockFree, Go, SIMD (SSE(2),AVX(2)), Native C++, Tools, CUDA

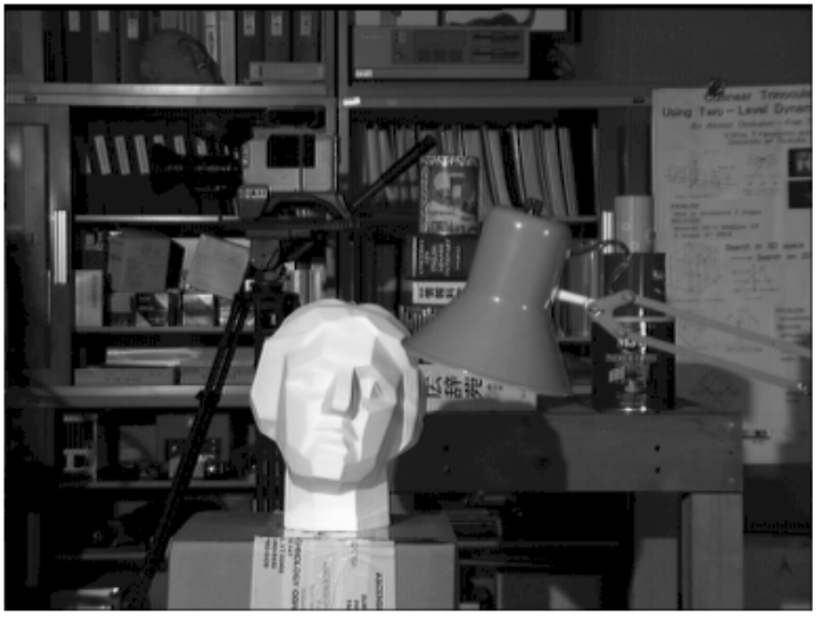
Einführung

- Stereoskopie
- Pipeline
- Filter
  - Diff of Gauss
  - Diff of Bilateral
- Kameramodell
- Stereo Matching
  - Sum of Squared Differences
  - Zero-Normalized Cross-Correlation

(Stereoskopie)

Figure 1: Stereoskopie

(Stereoskopie) Gewünschtes Ergebnis



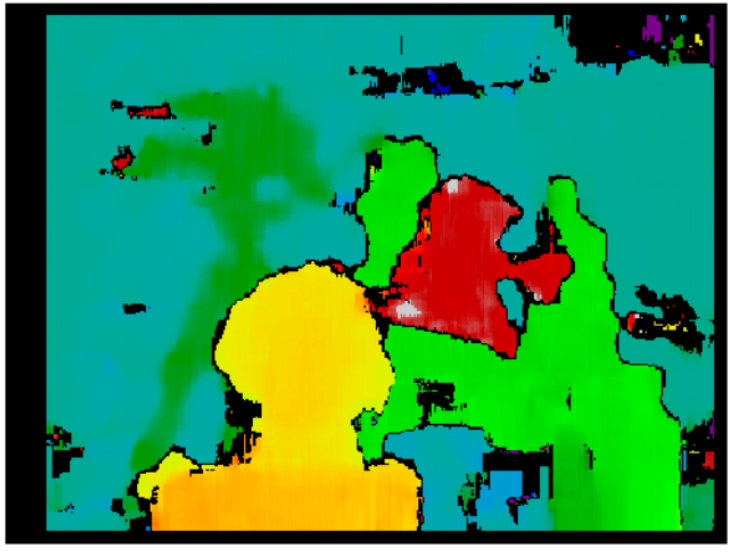


Figure 2: Gewünschtes Ergebnis

(Pipeline)

- Bilder laden (schauen wir nicht an)
- Rectification/Undistortion (schauen wir nicht an)
- Preprocess
- StereoMatching
- Visualisierung (schauen wir nicht an)

(Preprocessing) Bilateral

$$I_G[\vec{x}] = \frac{1}{W(\vec{x})} \sum_{[\vec{x}] \in S} G_{\sigma_S}(||\vec{x} - \vec{y}||) G_{\sigma_I}(||I[\vec{y}] - I[\vec{x}]||) I[\vec{y}]$$

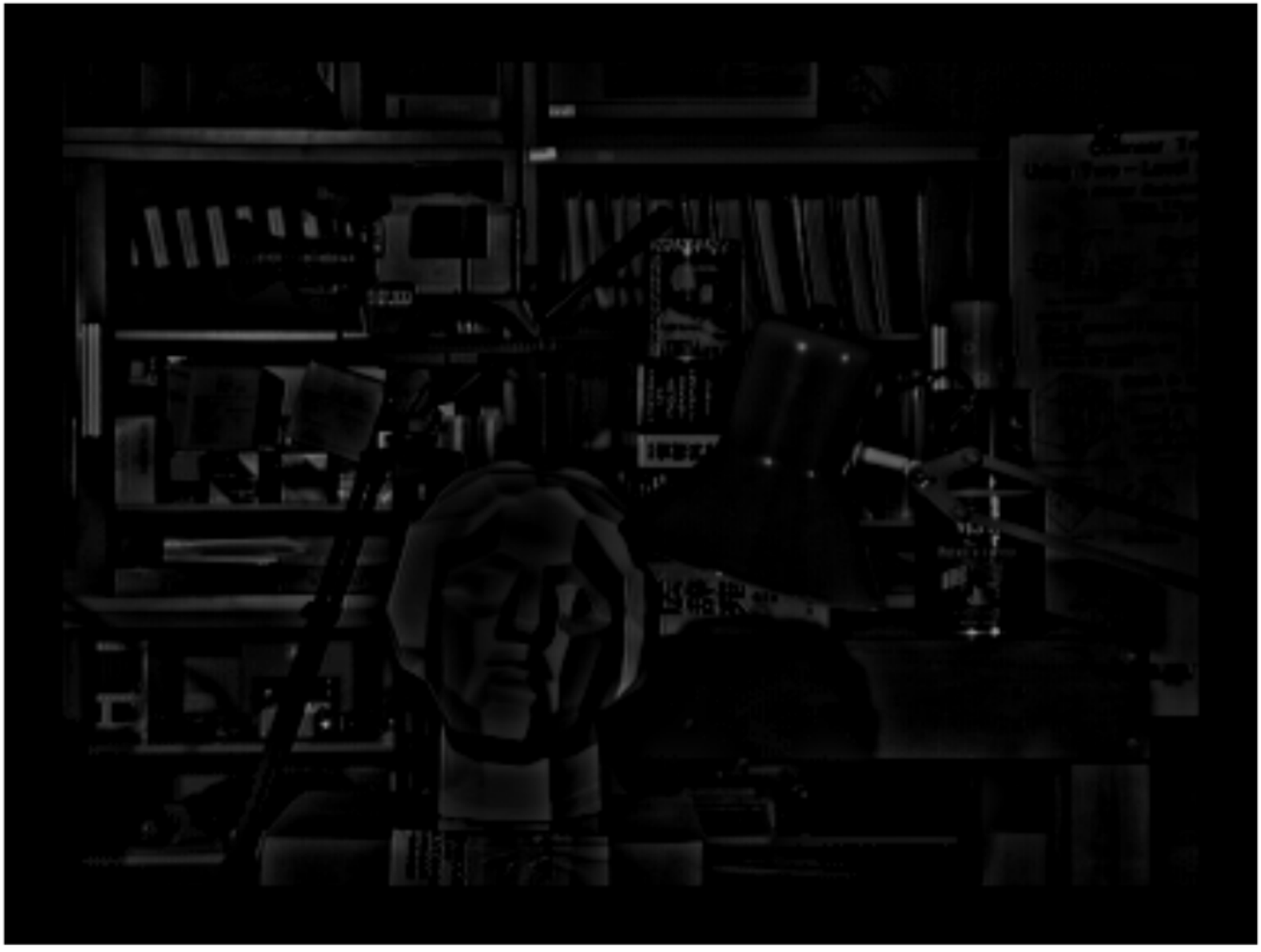
(Preprocessing) Bilateral

Figure 3: Bilateraler Filter Beispiel[1]

[1] Bilateral Filtering with CUDA, Lasse Kløjgaard Staal, 2007

(Preprocessing) Diff Of Bilateral

$$I_{DoB}[\vec{x}] = I[\vec{x}] - I_B[\vec{x}]$$



(Preprocessing) Gaussglättung

$$I_G[\vec{x}] = \sum_{[\vec{y}] \in S} G_\sigma(||\vec{x} - \vec{y}||) I[\vec{y}]$$

(Preprocessing) Diff Of Gauss

$$I_{DoG}[\vec{x}] = I[\vec{x}] - I_G[\vec{x}]$$

(Preprocessing - Gaußfilter) Kernel berechnen

- Mit der Formel den Kernel berechnen
- Mit $\sigma = 0.798$ bekommen wir folgenden Matrix $\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$
- Anwenden für jeden Bildpunkt

(Preprocessing - Gaußfilter) Kernel anwenden

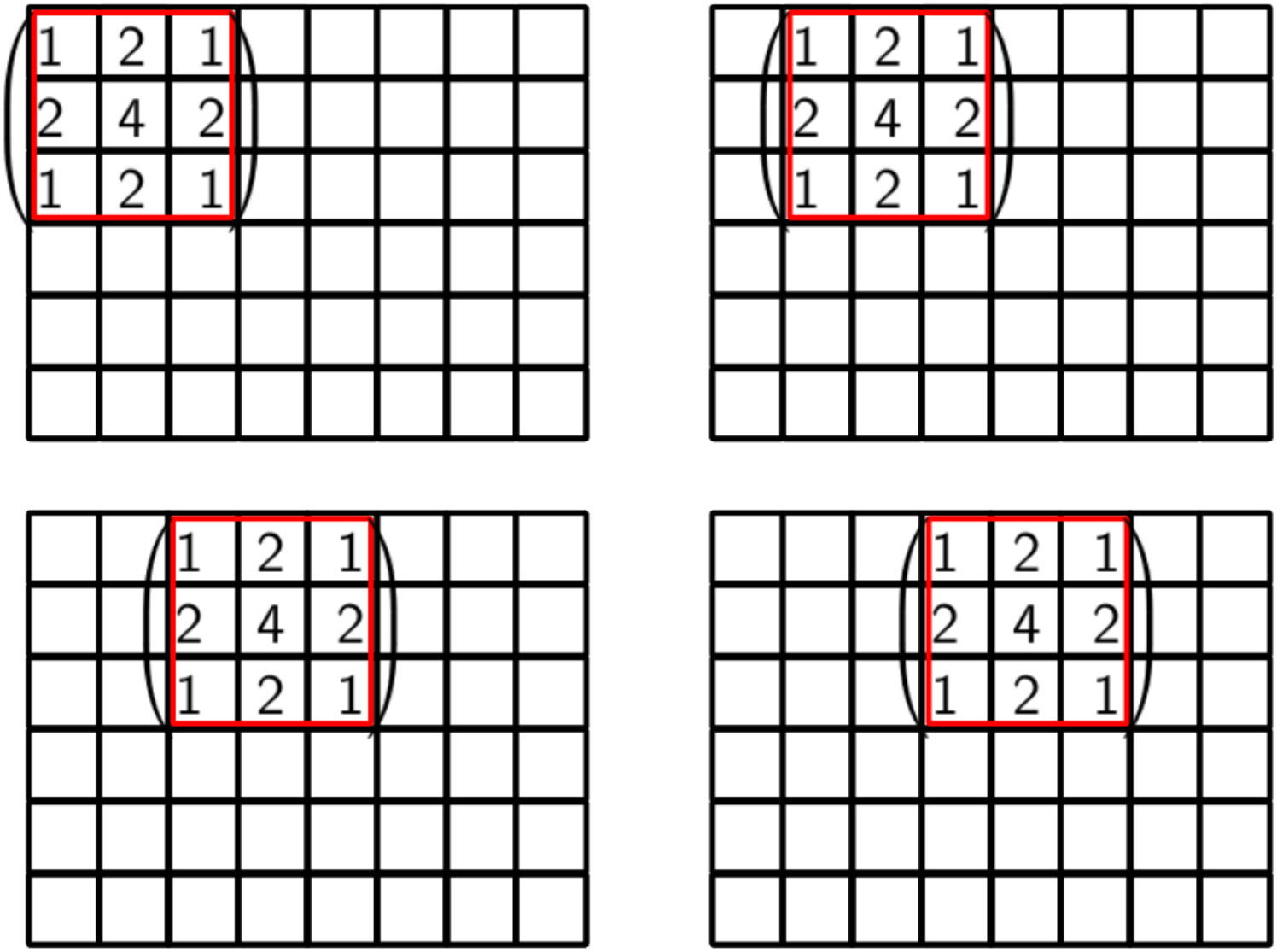


Figure 4: Gaußkernel anwenden

(Preprocessing - Gaußfilter) Separierbarkeit

- $\frac{1}{4}\begin{pmatrix}1\\2\\1\end{pmatrix} * \frac{1}{4}\begin{pmatrix}1 & 2 & 1\end{pmatrix} = \frac{1}{16}\begin{pmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{pmatrix}$
- Anwendbar für separierbare Matrizen
- Beispielsweise auch möglich für Sobel- und Perwitt-Operator.
- Bilateraler Filter nicht separierbar $\rightarrow$ trotzdem gemacht (gut genug)

(Preprocessing - Gaußfilter) Cachelines

- In Cache-Richtung arbeiten bzw. Bild transponieren

(Preprocessing - Gaußfilter) Pixelgrößen

- GPU arbeiten mit Speicherblöcken aus 4 Bytes
- RGB braucht aber nur 24 Bit Pixel, $\rightarrow$ aufblasen auf 4 Byte pro Px

(Preprocessing - Gaußfilter) Performanz

- (v1): Sequenziell
- (v2): Naive Cuda Implementierung
- (v3): Separiert

(Preprocessing - Gaußfilter) Performanz

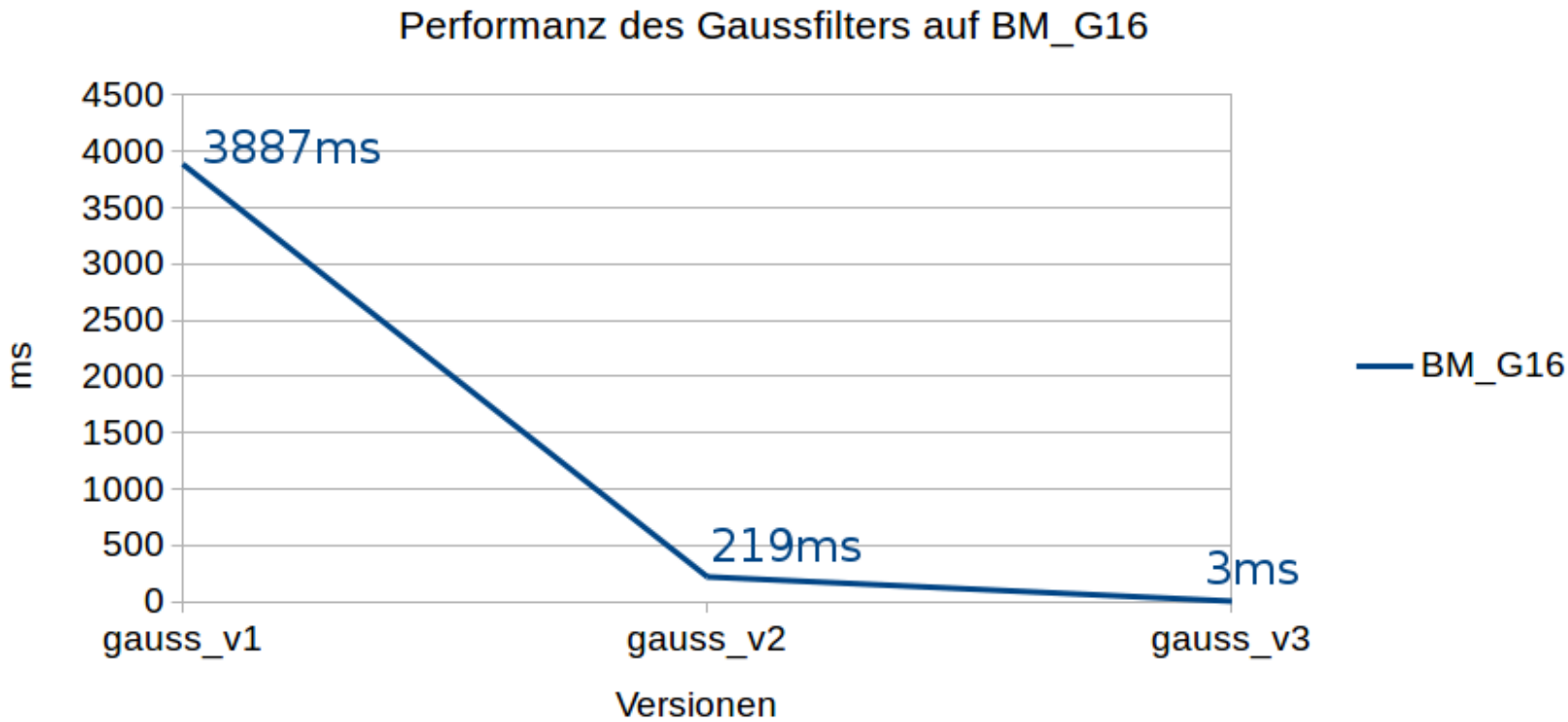


Figure 5: Performanz DoG[2]

[2] $\sigma = 16$, 3σ, 512x353, Intel Xeon CPU E5-1620 (8x3,7 GHz), GeForce GTX 550 Ti, Intel Xeon Phi (KNC), CentOS 6.9

(Kameramodell)

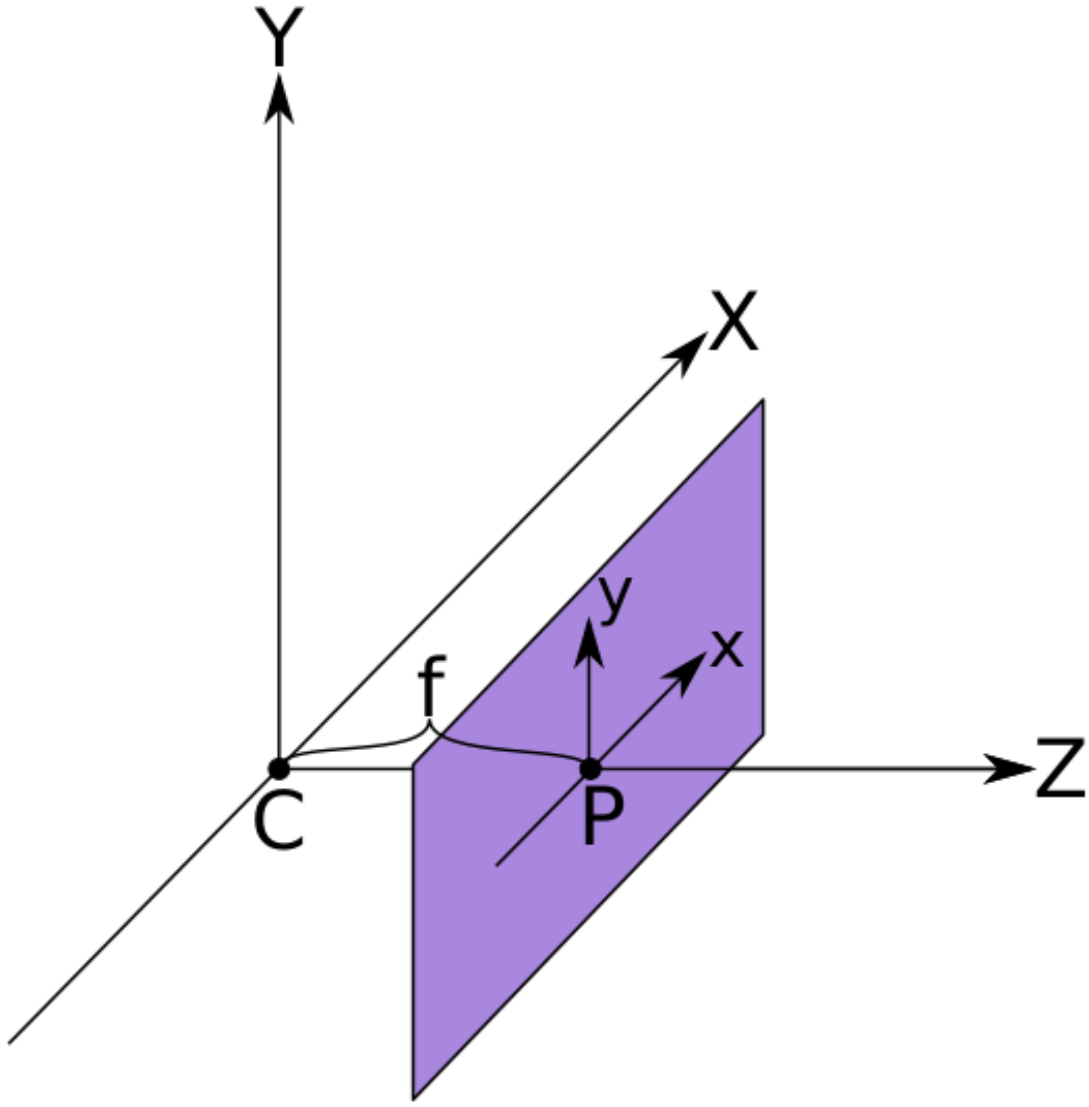


Figure 6: Lochkameramodell

(StereoMatching) Stereorekonstruktion

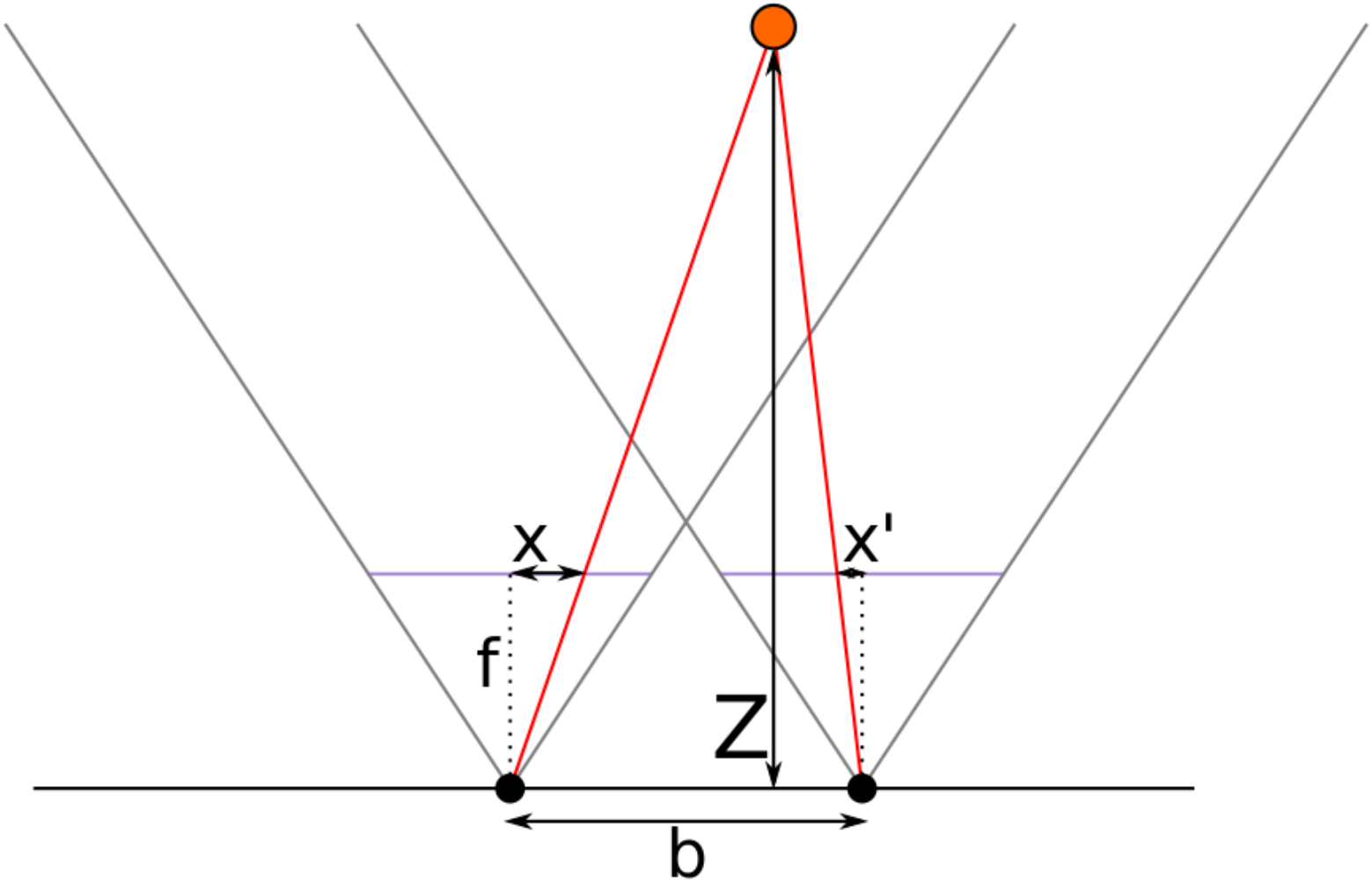


Figure 7: Stereorekonstruktion

(Stereo Matching)

- Wir suchen x und x'
- Wir suchen das selbe Pixel im anderen Bild
- Dank Epipolargeometrie müssen wir nur in der selben Zeile suchen
- Und nur eine Richtung

(Stereo Matching) BlockMatching

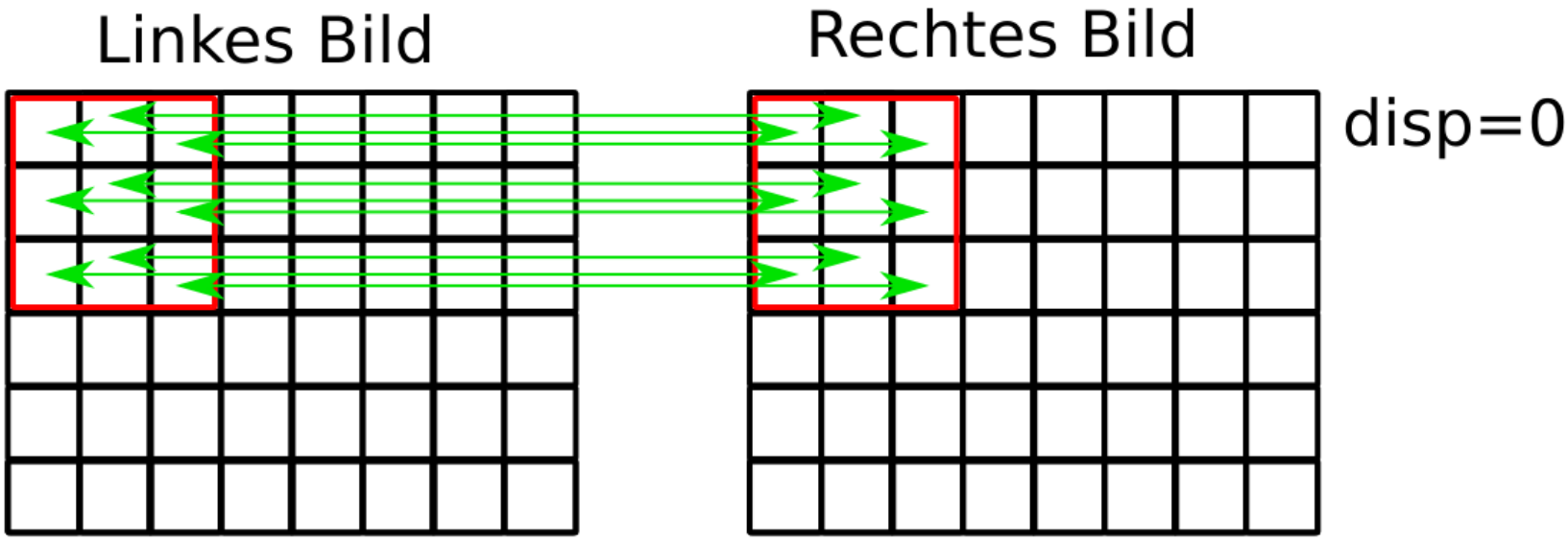


Figure 8: BlockMatching 0

(Stereo Matching) BlockMatching

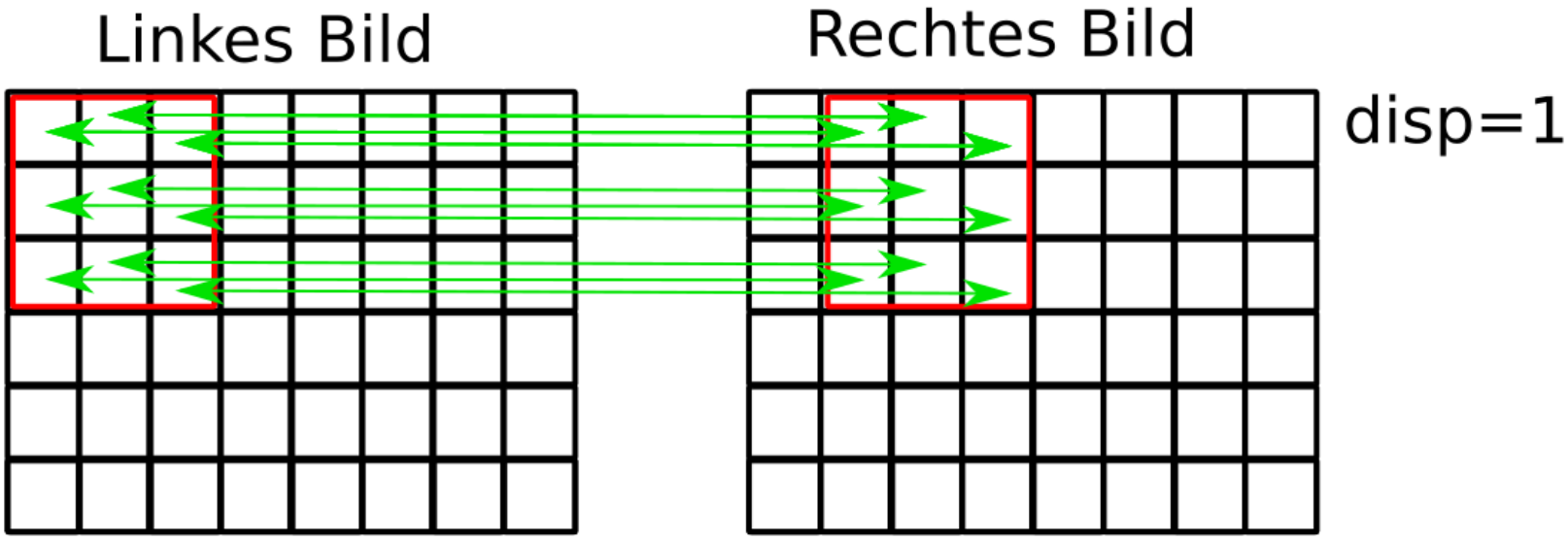


Figure 9: BlockMatching 1

(Stereo Matching) BlockMatching

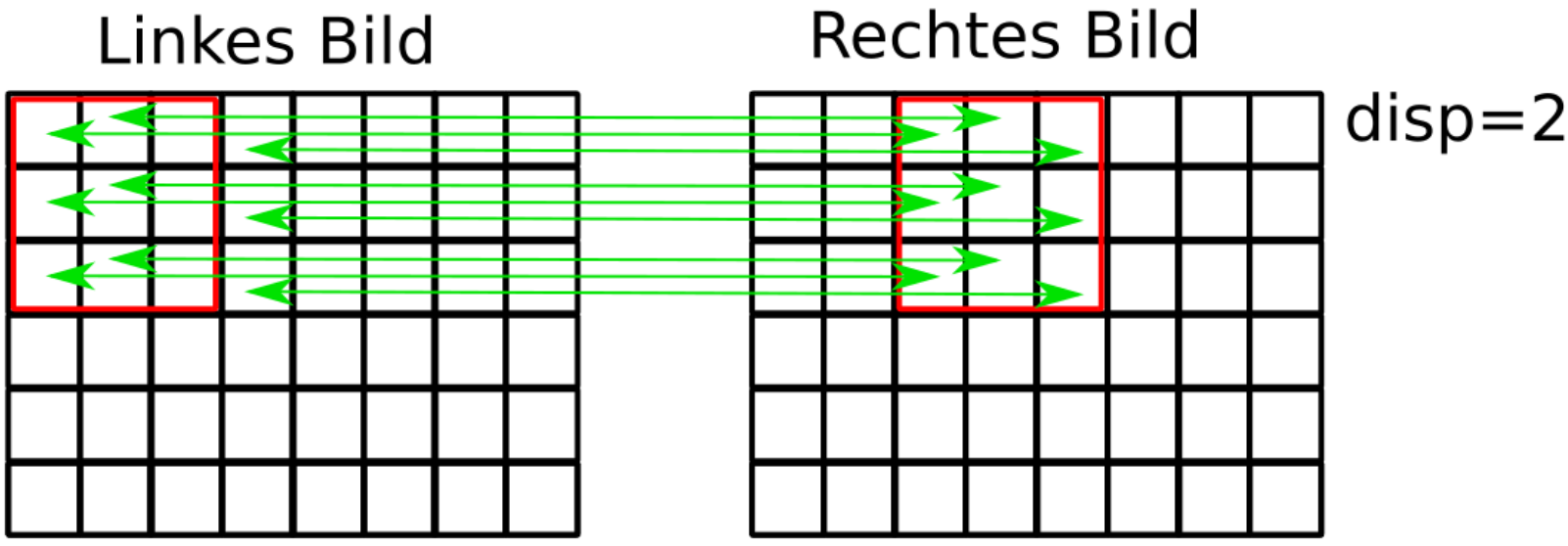


Figure 10: BlockMatching 2

Aber

Wie erkennen wir den Pixel wieder?

(Stereo Matching) Zero-Normalized Cross-Correlation

- $\frac{1}{n} \sum_{x,y} \frac{1}{\sigma_f \sigma_t} (f(x,y) - \mu_f)(t(x,y) - \mu_t)$
- Schauen wir nicht weiter an!

(Stereo Matching) Sum of Squared Differences

- $\frac{1}{n} \sum_{x,y} (f(x,y) - t(x,y))^2$

(Stereo Matching) Sum of Squared Differences

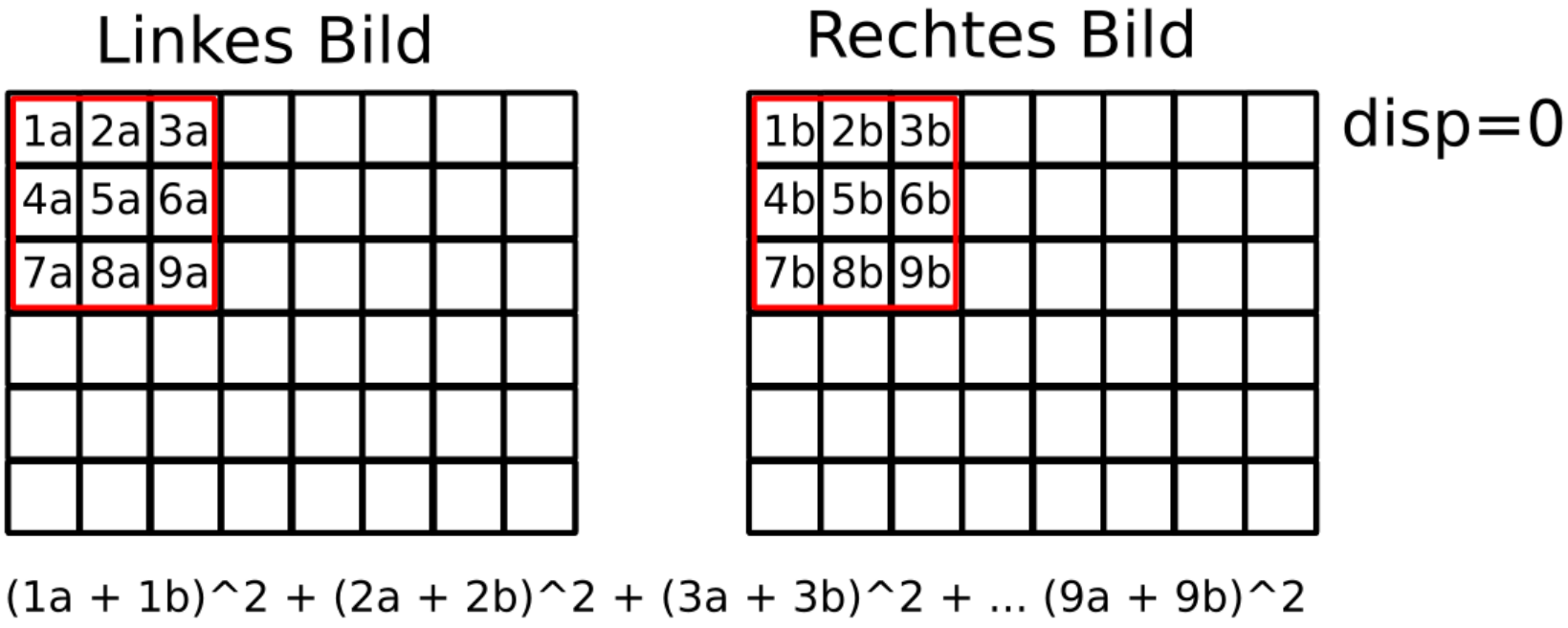
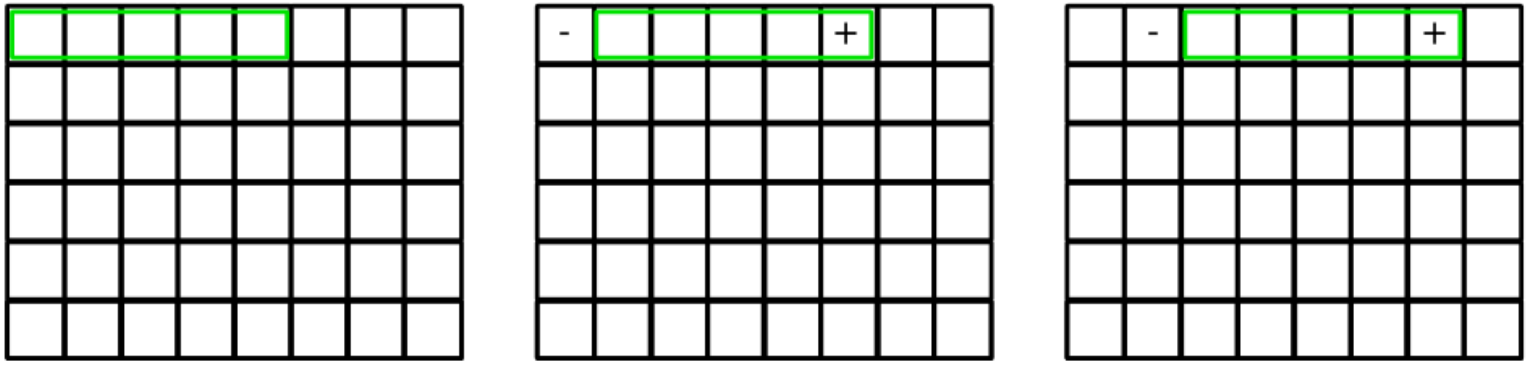


Figure 11: SSD Blockmatching

(Stereo Matching - SSD) Rolling Window

- $2 \times kernelRadius + 1)^2 \times width \times height \times ndisp$
- wird zu
- $(2 \times (2 \times kernelRadius + 1) \times width + 2 \times width \times (height - 1)) \times ndisp$



(Stereo Matching - SSD)

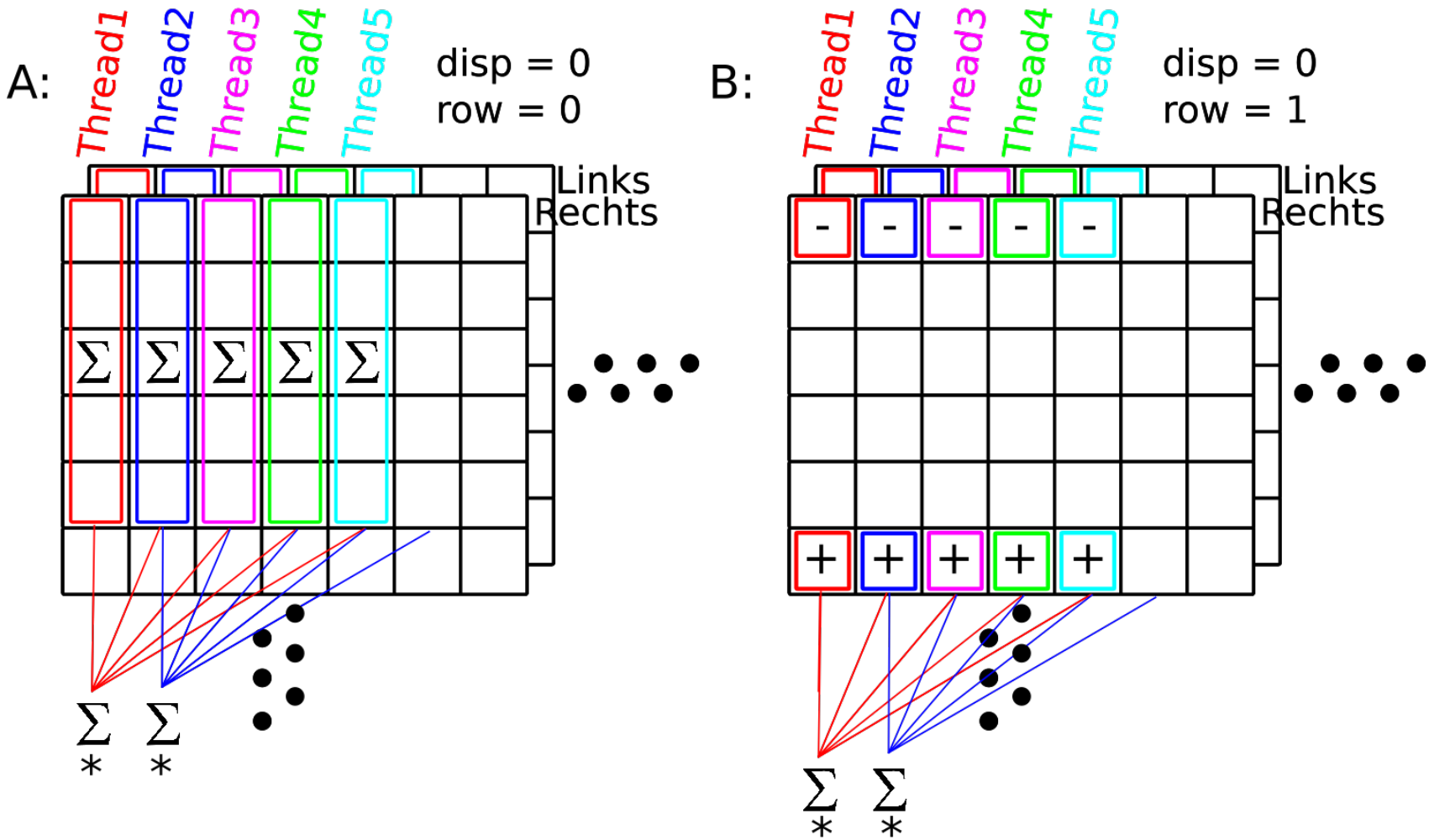


Figure 12: Algorithmus[3]

[3] Idee von "Stereo Imaging with Cuda", Joe Stam, Nvidia, 2008

(Stereo Matching - SSD)

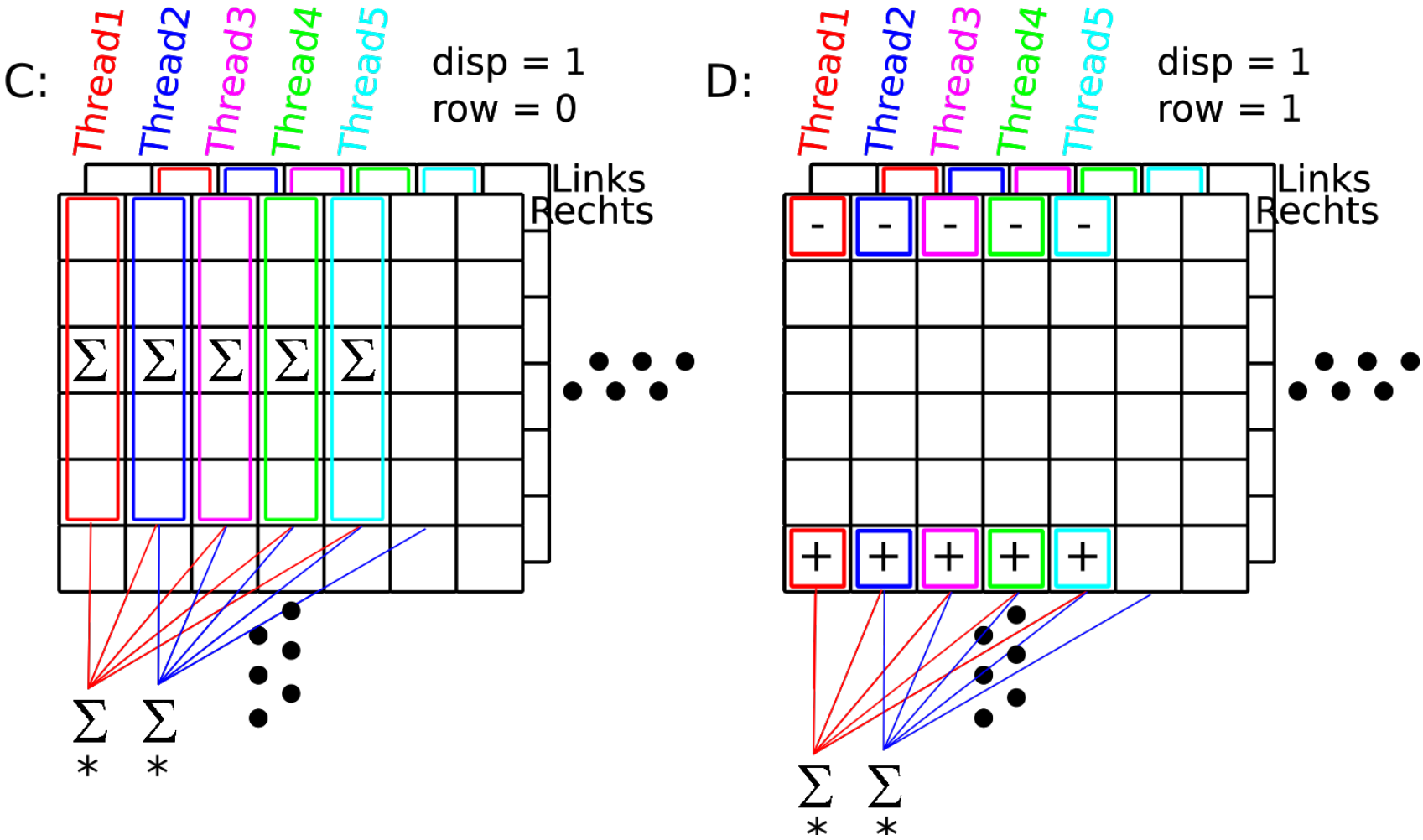


Figure 13: Algorithmus[4]

[4] Idee von "Stereo Imaging with Cuda", Joe Stam, Nvidia, 2008

(Stereo Matching - SSD) Performanz

- (cuda_v2): Naive Cuda Implementierung
- (cuda_v3): Berechne nur Spalten und summiere diese
- (cuda_v4): Ränder nicht separat betrachten
- (cuda_v5): Rolling Window

(Stereo Matching - SSD) Performanz

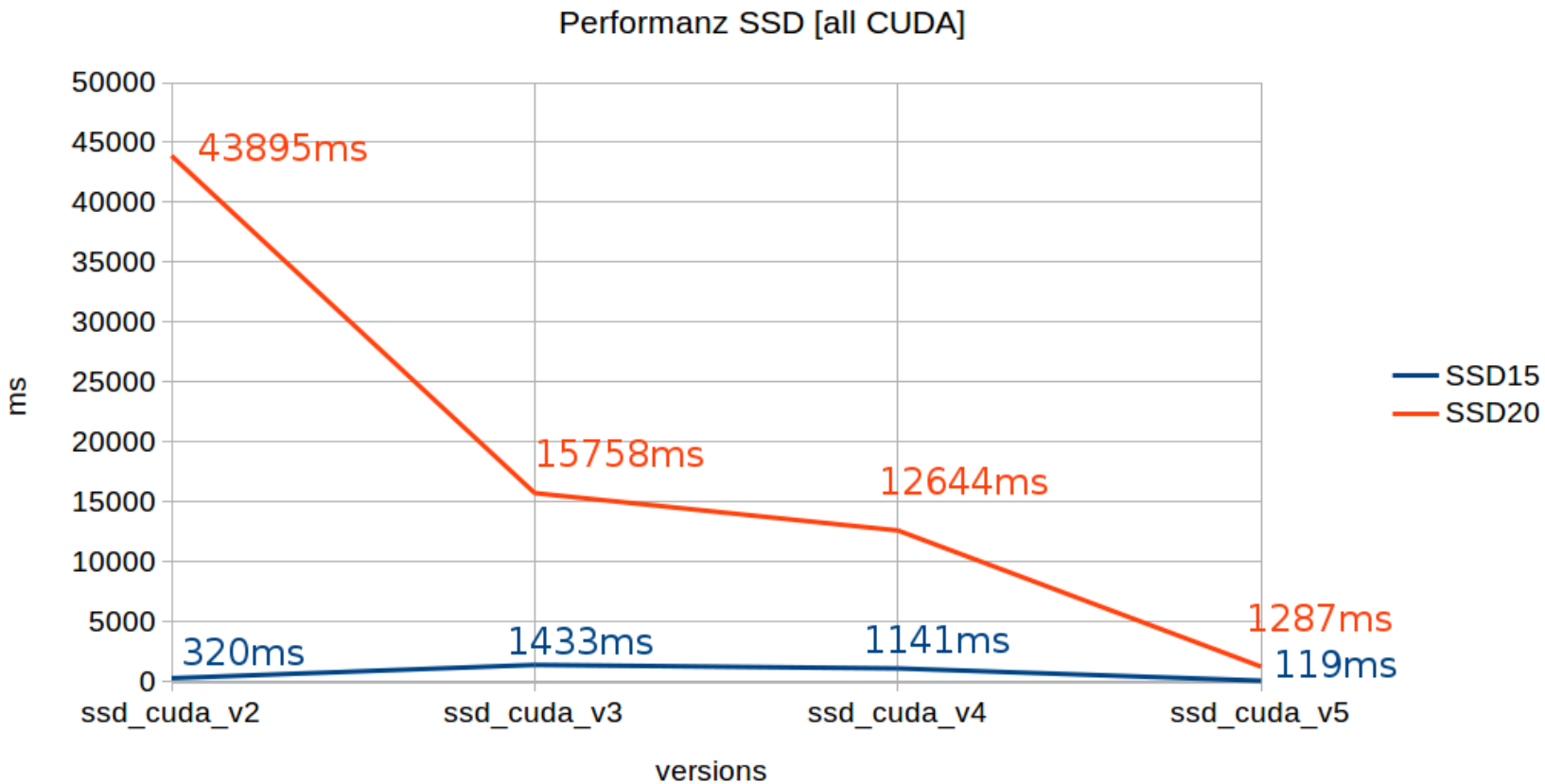


Figure 14: Performanz SSD[5]

[5]SSD15: 400x270, ndisp=100, window=21x21; SSD20: 2964x2000 ndisp=250, window=21x21; Intel Xeon CPU E5-1620 (8x3,7 GHz), GeForce GTX 550 Ti, Intel Xeon Phi (KNC), CentOS 6.9

Nochmehr Optimierungen

- Präfixsummen
- SharedMemory
- Texturspeicher
- CPU und GPU parallel
- Gauss mit Fouriertransformation in Frequenzbereich transformieren

Literatur/Quellen

- Stereo Imaging with Cuda, Joa Stam, Nvidia, 2008
- Bilateral Filtering with CUDA, Lasse Kløjgaard Staal, 2007
- ZNCC-based template matching using bounded partial correlation, Luigi Di Stefano, Stefano Mattoccia, Federico Tombari, 2005
- Effiziente kantenerhaltende Glättung und ihre Anwendung in der Praxis, Georg Braun (TU Wien), 2011
- Stereo Vision, Lukas Frank, Maximilian Noppel, Tobias Viehmann, 2018

Exkurs: GPUs

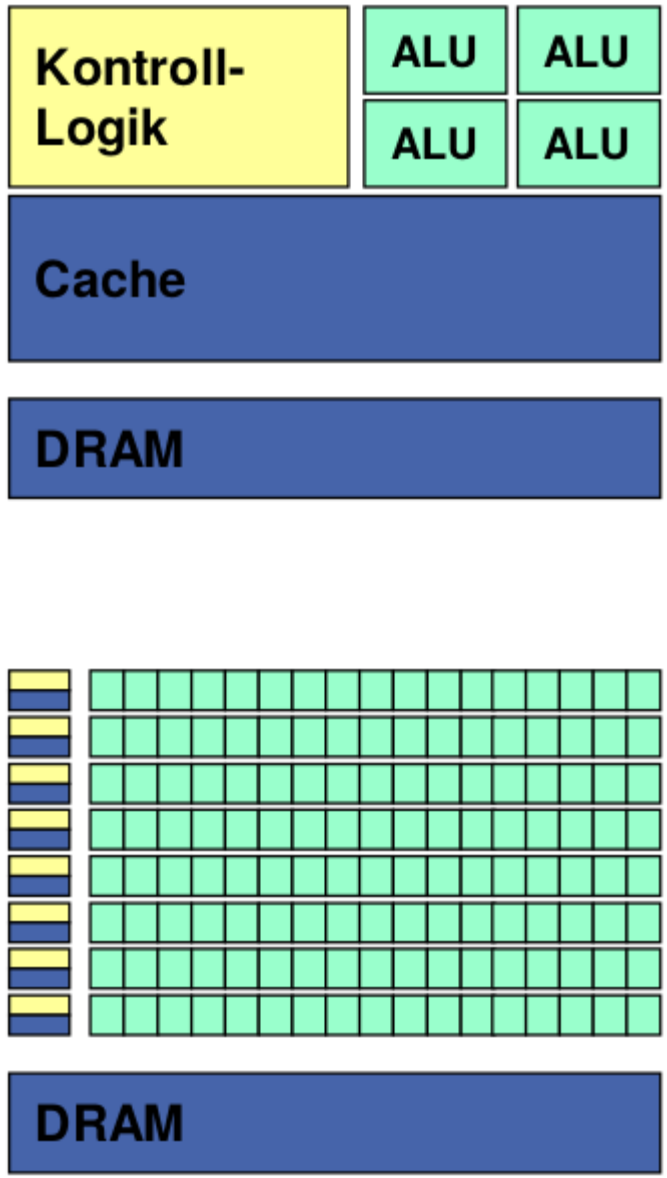


Figure 15: GPU versus CPU

Exkurs: GPUs

- Geeignet für spezielle Anwendungen
- mit großen Datenmengen, Datenparallelität, SIMD-Arbeitsweise
- Große Latenzen zum Hauptspeicher
- Weniger komplexe Kontrollflusslogik

Exkurs: GPUs

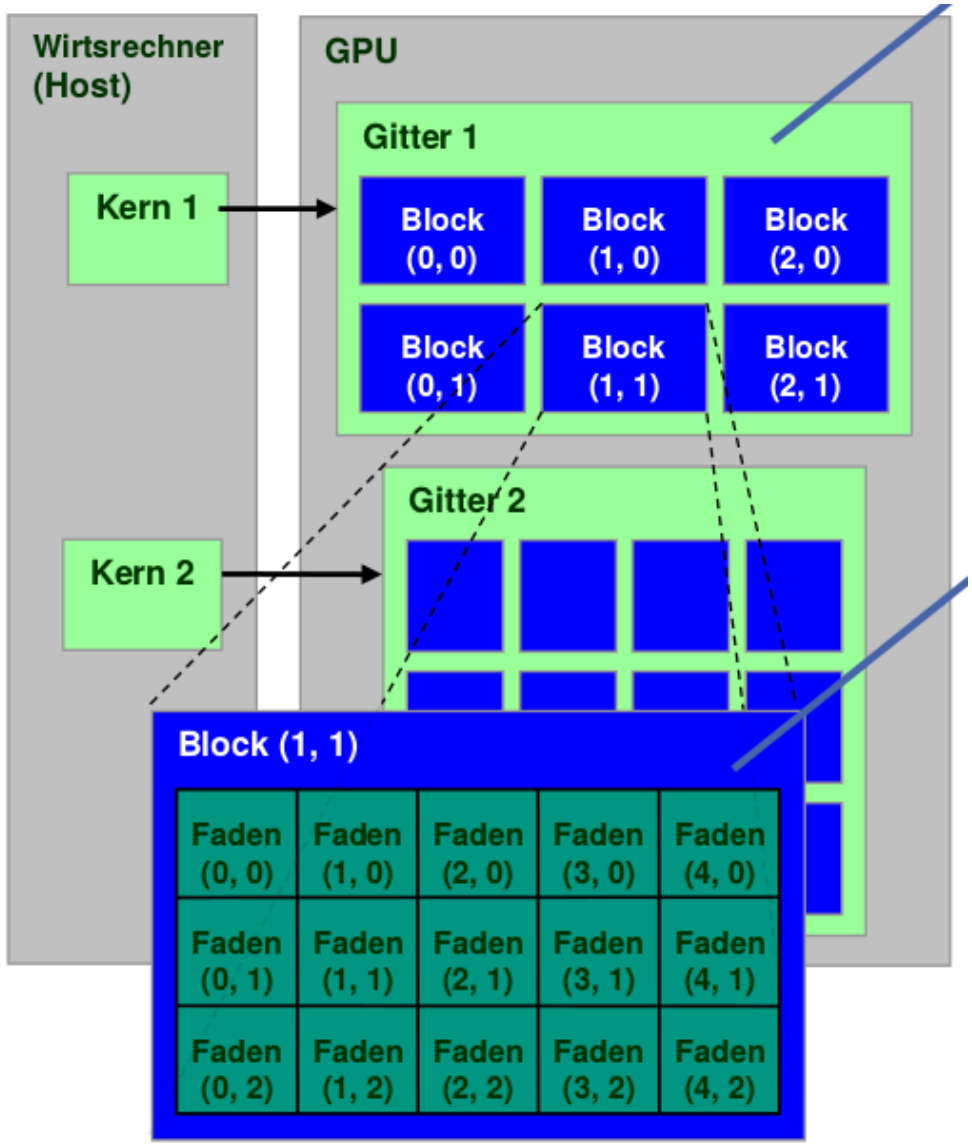


Figure 16: Aufbau einer Grafikkarte

Exkurs: GPUs

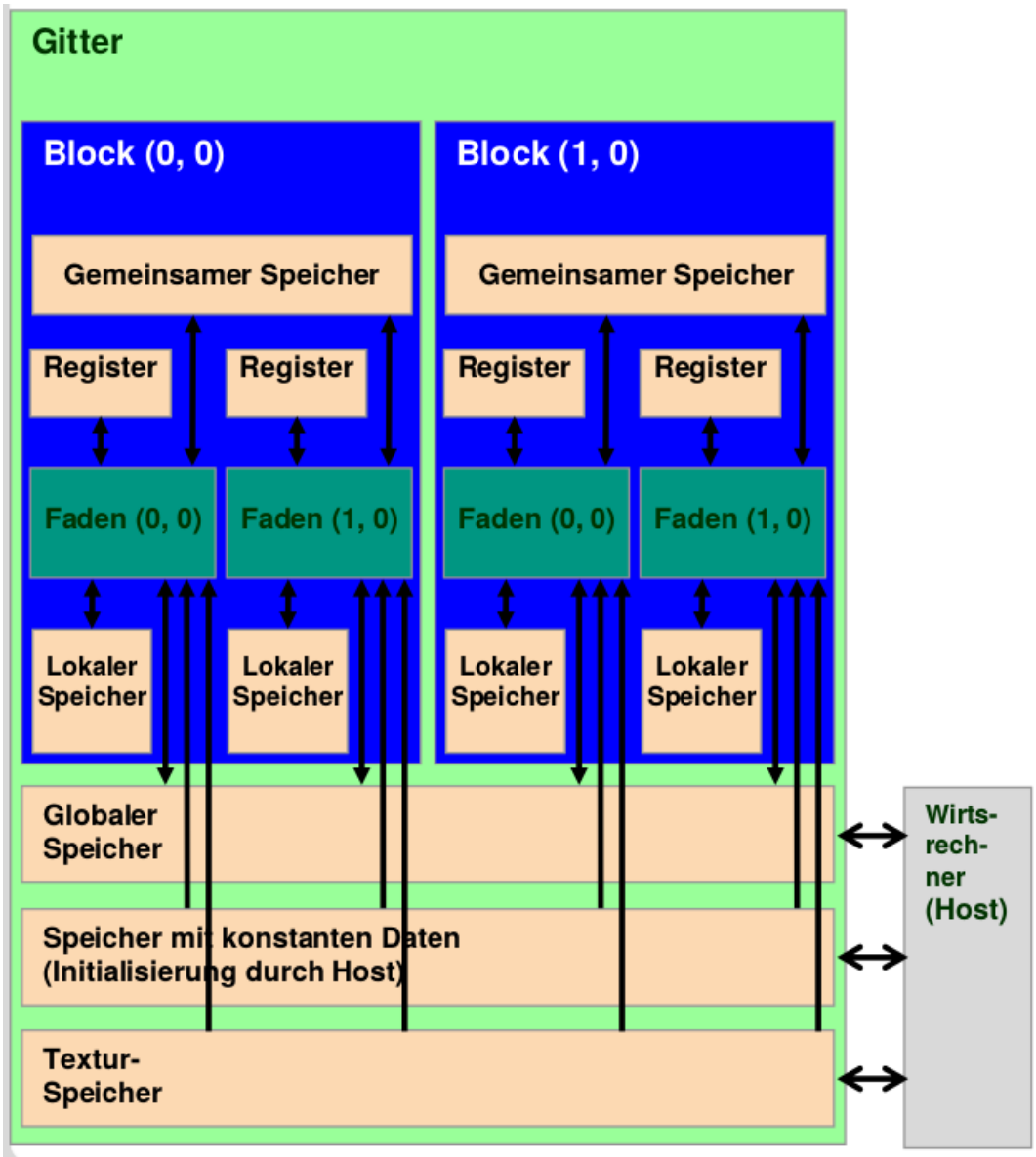


Figure 17: Speichermodell einer Grafikkarte

Exkurs: CUDA

- Compiler: nvcc
- ähnlich der Programmiersprache C
- CUDA Parts werden durch Macros markiert im Code
- Tausende Threads verfügbar
- Führt Kernels auf der GPU aus
- Ein Kernel wird auf einem Gitter ausgeführt mit mehreren Blöcken und Threads

Exkurs: CUDA

- cudaMalloc(): Alloziert Speicher auf der GPU im globalen Speicherbereich
- cudaFree(): Gibt Speicher wieder frei
- cudaMemcpy(): Kopiert Daten über PCI Bus (Host nach Host, Host nach GPU, GPU nach Host oder GPU nach GPU)

Exkurs: CUDA Variablen

Erweiterungen für Variablendeklarationen	Ablageort	Sichtbarkeit	Lebensdauer
`__device__ __shared__ int sharedVar;`	Gemeinsamer Speicher	Block	Block
`__device__ int globVar;`	Globaler Speicher	Gitter	Applikation
`__device__ __constant__ int constVar;`	Speicher mit konstanten Daten	Gitter	Applikation
`__managed__ int managedVar;`	Automatisch verwalteter Speicher	Gitter & Host	Applikation

Figure 18: Variablen

Exkurs: CUDA Funktionen

Erweiterungen für Funktionsdeklarationen	Nur ausführbar auf	Nur aufrufbar auf
`__device__ float gpuFunction()`	GPU	GPU
`__global__ void kernelFunction()`	GPU	Wirtsrechner
`__host__ float hostFunction()`	Wirtsrechner	Wirtsrechner

Figure 19: Funktionen

Keine Rekursion, keine variable Anzahl an Args, keine statische Variablen

Exkurs: CUDA Kernel aufrufen

```
__global__ void kernelFunc(params);

kernelFunc<<<GitterDim,BlockDim, BytesSharedMem>>>(params);
```

Exkurs: CUDA Welcher Thread bin ich?

- `threadIdx`: Index des Threads innerhalb des Blocks
- `blockIdx`: Index des Blocks im Gatter
- `blockDim`: Dimension des Blocks

Exkurs: CUDA Vektoraddition (Beispiel)

```
__global__ void vecAdd(float* A, float* B, float* C){
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Exkurs: CUDA Vektoraddition (Beispiel)

```
float *h_A = ...; *h_B = ...;

float *d_A, *d_B, *d_C;

cudaMalloc(&d_A, N*sizeof(float));
cudaMalloc(&d_B, N*sizeof(float));
cudaMalloc(&d_C, N*sizeof(float));

cudeMemCpy( d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevi
cudeMemCpy( d_B, h_B, N*sizeof(float), cudaMemcpyHostToDevi

vecAdd<<<N/256,256>>>(d_A,d_B,d_C);

cudeMemCpy( h_C, d_C, N*sizeof(float), cudaMemcpyDeviceToHo
```

Exkurs: CUDA Syncs

- `void __synchThreads()` synchronisiert Threads in einem Block
- Seit CUDA 9.0: `thread_group` und `thread_block`: Sync über mehrere Blöcke

Exkurs: CUDA Programmiermodell

- Aus GPU Quelltext wird PTX Code erzeugt
- C/C++ Code bleibt übrig
- PTX Code wird JIT übersetzt
- Übersetzung durch Gerätetreiber vor dem ersten Kernelausruf