

# Leistungsfähigkeit von CUDA-Programmen steigern

## Teil 1

Peter Kulczycki  
[peter.kulczycki<AT>fh-hagenberg.at](mailto:peter.kulczycki@fh-hagenberg.at)

Department of Software Engineering  
University of Applied Sciences Upper Austria  
Softwarepark 11, 4232 Hagenberg, Austria

Version 1.00.30331 – 5. Jänner 2016

# Testumgebung

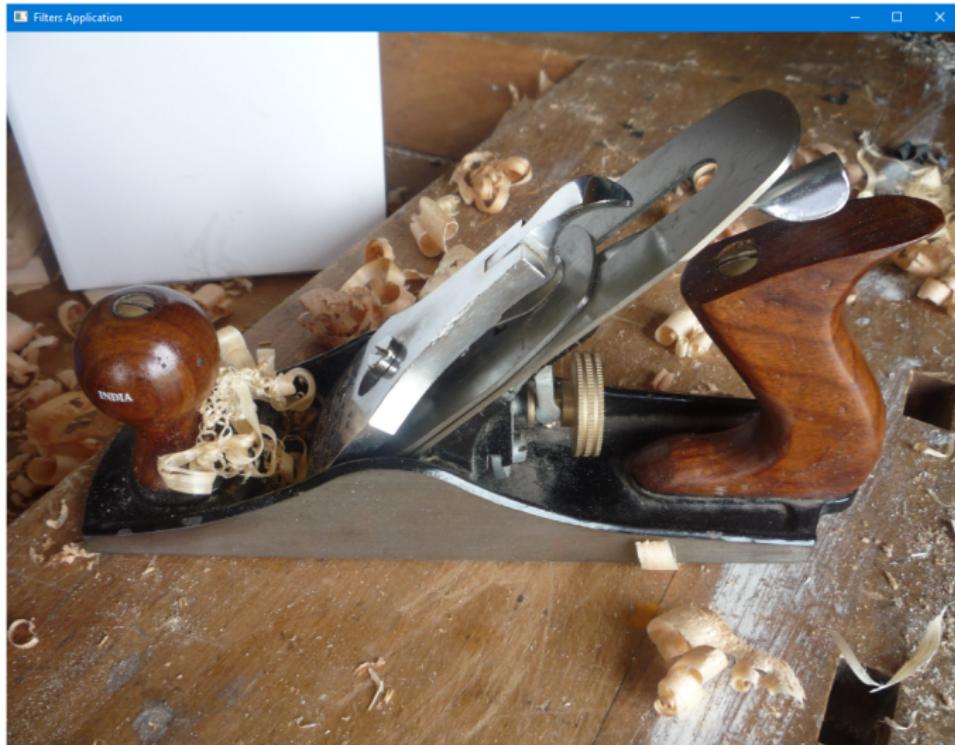
---

Host:	Intel Core i7-4810MQ, 2,8 GHz, 16 GB
Device:	Nvidia Quadro K2100M, GK106 Kepler, CC3.0, 2 GB
OS:	Windows 10
IDE:	Visual Studio 2013 Update 5
Debugger	Nvidia Nsight Visual Studio Edition 5.0

---

*Anmerkung:* Die hier im Weiteren vorgestellten Maßnahmen zur Steigerung der Leistungsfähigkeit von CUDA-Programmen können sich bei anderer als der hier angegebenen Hard- und/oder Software unterschiedlich (auch gar nicht oder sogar leistungsmindernd) auswirken.

# Demoprogramm



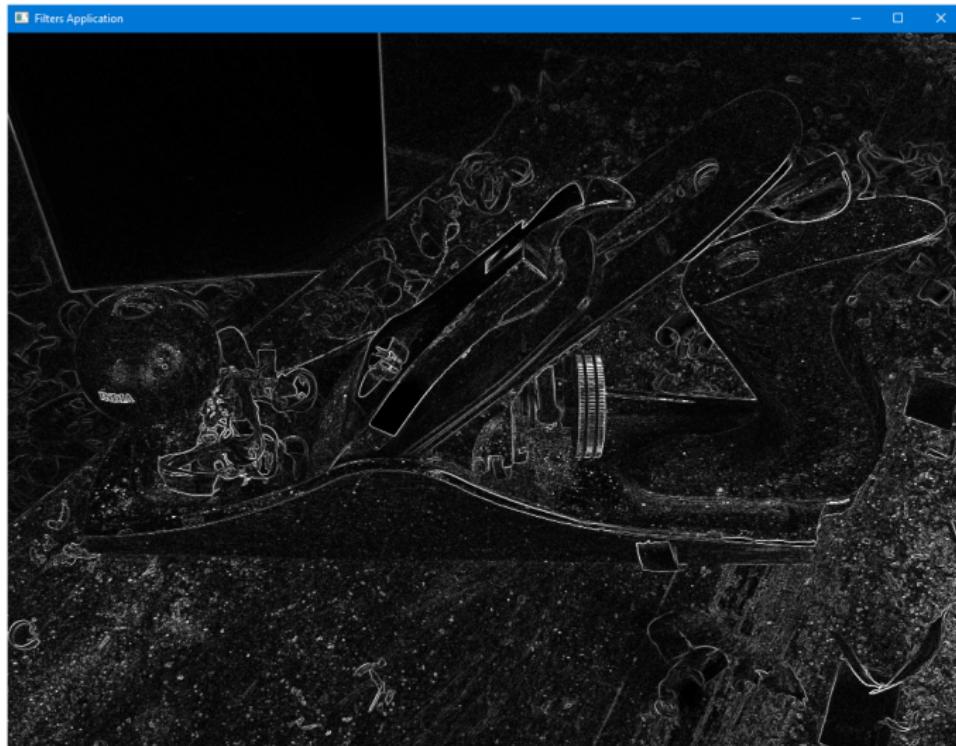
Das Originalbild (24 Bit,  $3.648 \times 2.736$  pels) ...

# Demoprogramm



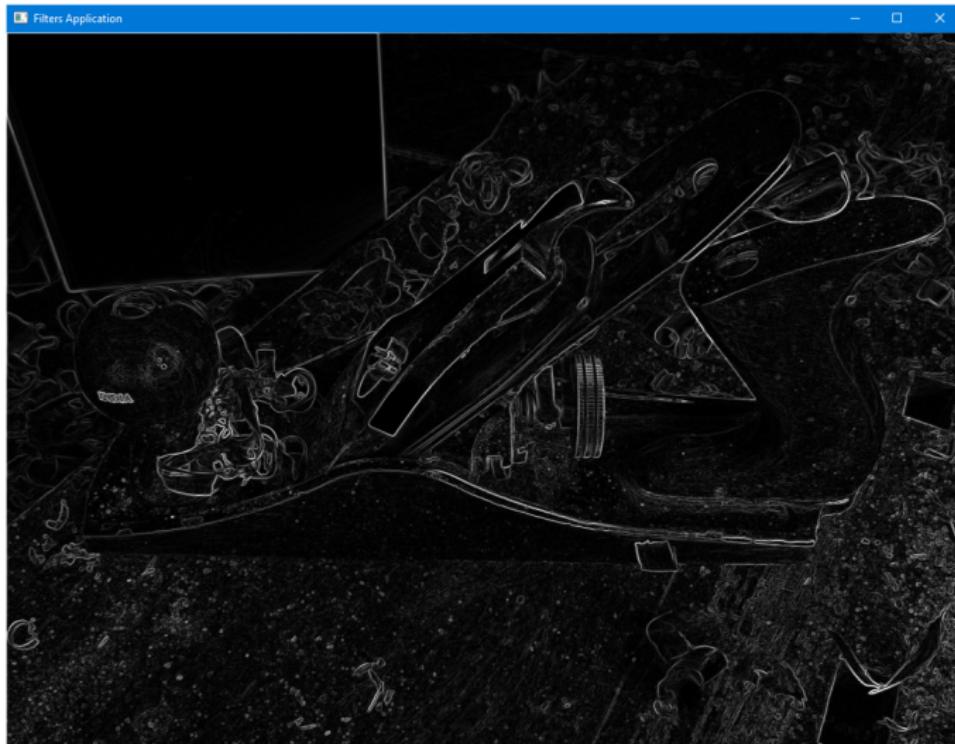
... nach der Umwandlung in Graustufen ...

# Demoprogramm



... und nach der Kantendetektion (ohne Weichzeichnen).

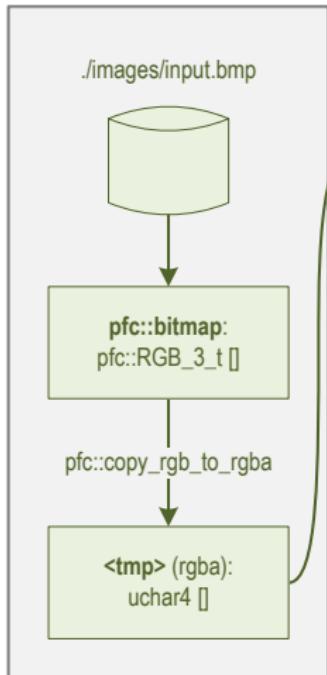
# Demoprogramm



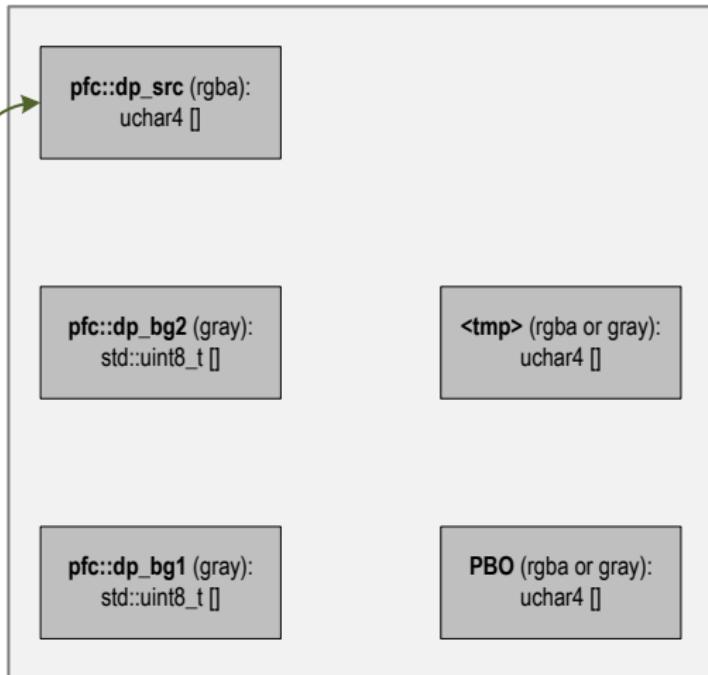
... und nach der Kantendetektion (mit vorhergehendem Weichzeichnen).

# Bufferhandling

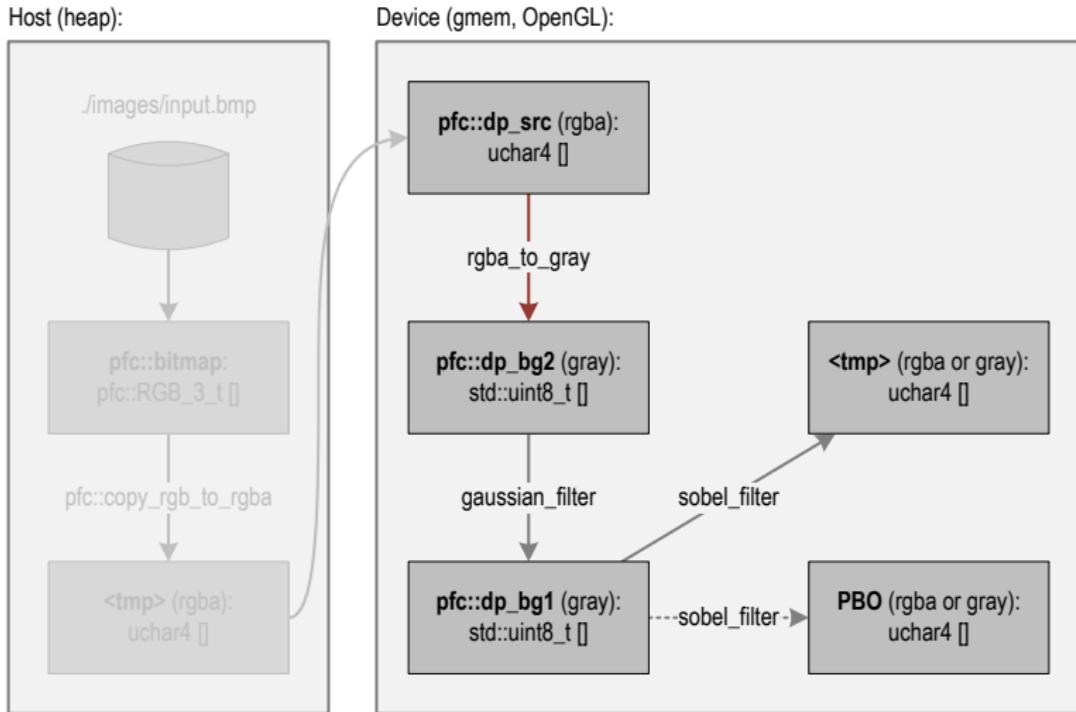
Host (heap):



Device (gmem, OpenGL):



# Bufferhandling



# Algorithmische Grundlagen

Die Umwandlung des Farbbilds in eine Graustufenbild erfolgt nach diesem Schema:

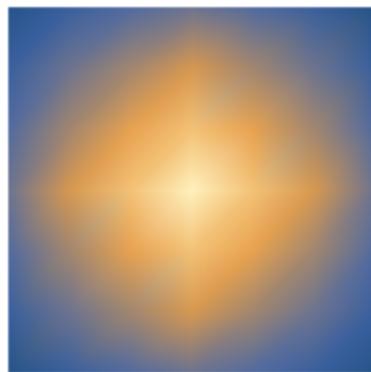
```
1  namespace pfc {
2
3      using byte_t = std::uint8_t;
4
5      template <typename T> T & min (T & a, T & b) {
6          return (a < b) ? a : b;
7      }
8
9      pfc::byte_t gray_value (float r, float g, float b) {
10         return pfc::byte_t (pfc::min (
11             255,
12             int (.299f * r + .587f * g + .114f * b)
13         ));
14     }
15
16 } // namespace pfc
```

Anmerkung: Beide Funktionen sind mit den CUDA Attributen `__host__`, `__device__` und `__forceinline__` versehen.

# Algorithmische Grundlagen

Das Weichzeichnen wird mithilfe eines  $7 \times 7$  Gaussfilters (siehe z. B. [https://en.wikipedia.org/wiki/Gaussian\\_blur](https://en.wikipedia.org/wiki/Gaussian_blur)) durchgeführt:

1	2	3	4	3	2	1
2	4	6	8	6	4	2
3	6	9	12	9	6	3
4	8	12	16	12	8	4
3	6	9	12	9	6	3
2	4	6	8	6	4	2
1	2	3	4	3	2	1



```
1 function gauss()
2     for each pixel p do
3         p ← weighted sum of p and its 48 neighbors
4     end for
5 end function
```

# Algorithmische Grundlagen

Die Kantendetektion wird mithilfe eines  $3 \times 3$  Sobeloperators (siehe z. B. [https://en.wikipedia.org/wiki/Sobel\\_operator](https://en.wikipedia.org/wiki/Sobel_operator)) durchgeführt:

$$\begin{array}{c} s_x: \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \quad s_y: \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array} \end{array}$$

```
1 function sobel ()  
2   for each pixel p do  
3      $g_x \leftarrow$  weighted sum of p and its eight neighbors (using matrix  $s_x$ )  
4      $g_y \leftarrow$  weighted sum of p and its eight neighbors (using matrix  $s_y$ )  
5  
6      $p \leftarrow \sqrt{g_x^2 + g_y^2}$   
7   end for  
8 end function
```

# Programmversionen

V	Feature
0	initiale Version
1	Blockgröße $32 \times 2$ : verbessertes Coalescing ( <i>coalescing</i> , Verbindung)
2	Blockgröße $8 \times 16$ : verbesserte Occupancy ( <i>occupancy</i> , Belegung)
3	Blockgröße $32 \times 4$ : verbesserte Occupancy
4	verwendet <code>__launch_bounds__</code> : reduzierte Belastung der Register ( <i>register pressure</i> ) und verbesserte Occupancy
5	verbesserter Speicherzugriff über Shared-Memory ( <i>smem</i> ), reduzierter Druck auf die Load-Store-Unit
6	verbesserte Faltungsmatrix ( <i>convolution kernel/filter</i> )
7	reduzierter Berechnungsaufwand durch separierbare Filter
8	Verarbeitung von zwei Pixels pro Thread: verbesserte Speichereffizienz und Instruction-Level-Parallelism (IPL)

# Programmversionen

V	Feature
9	verbesserter Zugriff auf den Shared Memory: weniger Konflikte auf den Speicherbänken des <i>smems</i>
10	Verwendung von <code>floats</code> : reduzierte Belastung der arithmetischen Pipeline
11	Verwendung von intrinsischen Mathematikfunktionen ( <i>math intrinsics</i> )

*Anmerkung:* Die auszuführende Programmversion muss auf der Befehlszeile angegeben werden (z. B.: `filters 5`). Die Verwendung von OpenGL muss bereits beim Kompilieren durch Definition des Makros `WITH_OPENGL` spezifiziert werden.

# Programmversionen

V	gaussian						sobel		rgba	
	V	smb	pref	lb	res	Block	V	Block	V	Block
0	1	def.	L1\$			8 × 8	1	32 × 8	1	32 × 8
1	1	def.	L1\$			32 × 2	1	32 × 8	1	32 × 8
2	1	def.	L1\$			8 × 16	1	32 × 8	1	32 × 8
3	1	def.	L1\$			32 × 4	1	32 × 8	1	32 × 8
4	2	def.	L1\$	x		32 × 4	1	32 × 8	1	32 × 8
5	3	def.	smem			32 × 4	1	32 × 8	1	32 × 8
6	4	def.	smem		x	32 × $\frac{8}{2}$	1	32 × 8	1	32 × 8
7	5	def.	smem		x	32 × 8	1	32 × 8	1	32 × 8
8	6	def.	smem		x	32 × 8	1	32 × 8	1	32 × 8
9	6	8	smem		x	32 × $\frac{8}{2}$	1	32 × 8	1	32 × 8
10	7	8	smem		x	32 × $\frac{8}{2}$	1	32 × 8	1	32 × 8
11	7	8	smem		x	32 × $\frac{8}{2}$	2	32 × 8	2	32 × 8

# Messergebnisse

Version	0	1	2	3	4	5	6	7	8
gray	12	12	12	12	12	12	12	12	12
blur	524	506	293	283	283	197	170	69	107
edge	67	67	67	67	67	67	67	66	67
total	614	595	380	317	371	284	259	157	196

Version	9	10	11
gray	12	12	30
blur	107	106	108
edge	67	67	76
total	195	193	224

Anmerkung: Alle Messwerte sind in ms angegeben und beziehen sich auf die weiter oben gegebene Hard- und Software. Kompiliert wurde eine 32 Bit-Applikation im DEBUG-Modus und ohne OpenGL.

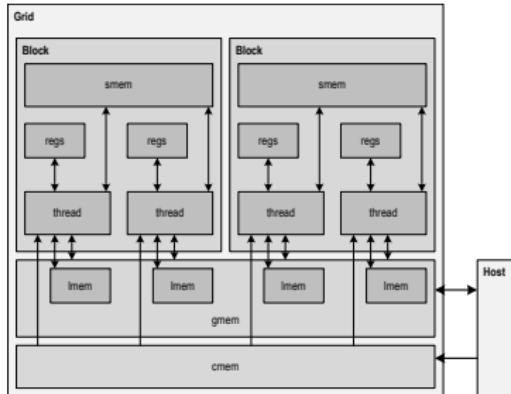
# GPU-Grundbegriffe

Grundbegriffe des CUDA-Ausführungsmodells (alphabetisch):

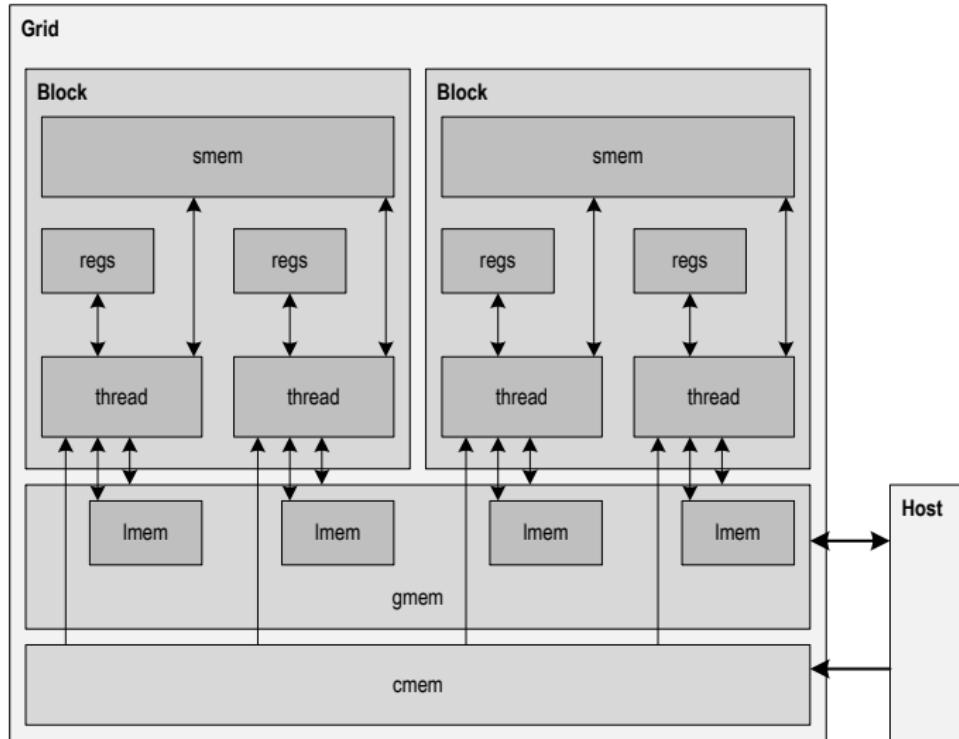
- 1 Bank
- 2 Block (1D, 2D und 3D)
- 3 Cache (L1\$, L2\$) und Cache-Line
- 4 Compute-Capability
- 5 Device, Grid (1D, 2D und 3D)
- 6 Host
- 7 Kernel (Shader)
- 8 Scheduler (SIMT-Architectur)
- 9 Streaming-Multiprozessor (SM, SMX, SMM)
- 10 Streaming-Processor (Core, CUDA-Core, Shader-Unit)
- 11 Thread
- 12 Warp

# GPU-Speicherhierarchie

Speicherart	Geschwindigkeit	Lifetime	Scope
global ( <i>gmem</i> )	langsam ( <i>uncached off-chip</i> )	Appl.	Grid
konstant ( <i>cmem</i> )	schnell ( <i>cached off-chip</i> )	Appl.	Grid
shared ( <i>smem</i> )	schnell ( <i>on-chip</i> )	Block	Block
lokal	langsam ( <i>uncached off-chip</i> )	Thread	Thread
Register ( <i>regs</i> )	sehr schnell ( <i>on-chip</i> )	Thread	Thread



# GPU-Speicherhierarchie



Anmerkung: Die Caches L1\$ und L2\$ sind nicht eingezeichnet.

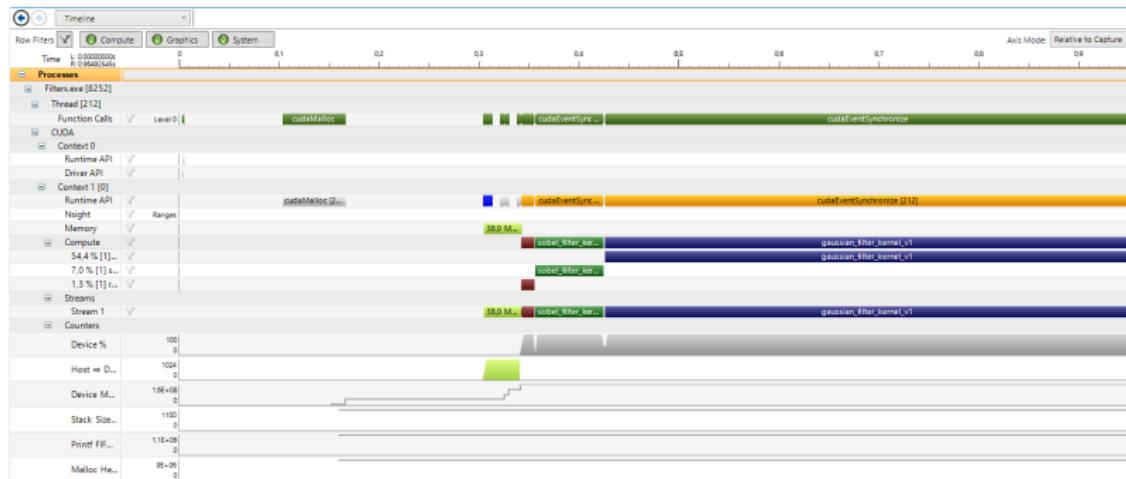
# Debugger

Siehe dazu [NVIDIA Corporation, 2015b, Kapitel „CUDA Debugger“] unter [http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/nvidia\\_nsight.htm](http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/nvidia_nsight.htm).

- 1 Control Execution
- 2 Set GPU-Breakpoints
- 3 Global Freeze
- 4 Specify Debugger-Context
- 5 View Memory und View Variables
- 6 View CUDA-Information
- 7 CUDA Warp-Watch

# Version 0: Hotspots

- 1 Demoprogramm „Filters“ in der **Version 0** und ohne OpenGL kompilieren (WIN32, Debug, Makro `WITH_OPENGL` ist nicht definiert)
- 2 Kommando „NSIGHT“/„Start Performance-Analysis“ ausführen. „Trace Application“ und „CUDA“ auswählen. Kommando „Launch“ ausführen. Auswahl „Timeline“ treffen.



# Version 0: Hotspots

- 3 Auswahl „CUDA Summary“ treffen.
- 4 Im Punkt „Top Device-Functions by total Time“ den Kernel `gaussian_filter_kernel` als Hotspot identifizieren. Mit 524,6 ms nimmt er 54,4 % der Gesamtlaufzeit ein.

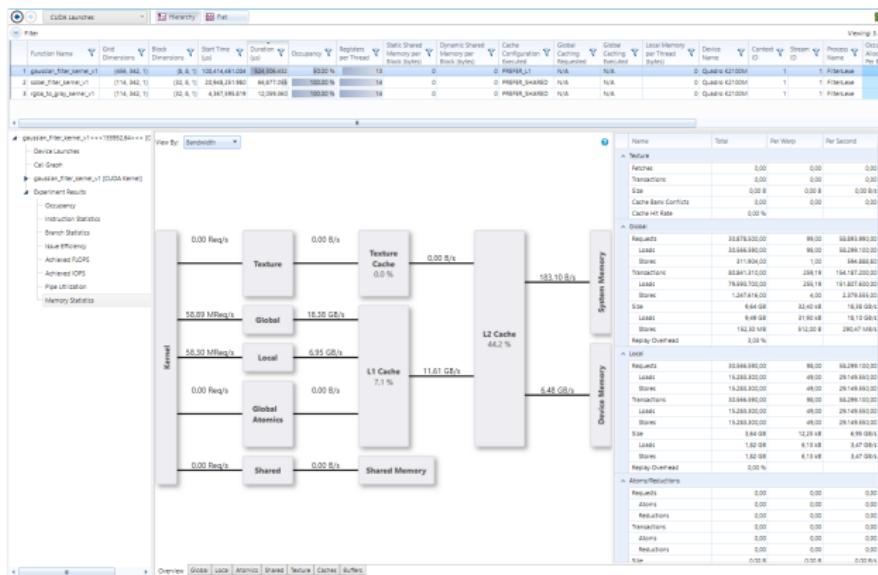
**Top Device Functions By Total Time** [Summary](#) | [All](#)

	Name	Launches	Device %	Total (μs)	Min (μs)	Avg (μs)	Max (μs)
1	<code>gaussian_filter_kernel_v1</code>	1	54.37	524,637.526	524,637.526	524,637.526	524,637.526
2	<code>sobel_filter_kernel_v1</code>	1	6.97	67,249.208	67,249.208	67,249.208	67,249.208
3	<code>rgba_to_gray_kernel_v1</code>	1	1.28	12,338.162	12,338.162	12,338.162	12,338.162

*Vermutung:* Der Kernel `gaussian_filter_kernel` ist der Hotspot des Demoprogramms. Die weiteren Analysen konzentrieren sich auf diesen Kernel.

# Gründe identifizieren

- 1 Kommando „NSIGHT“/„Start Performance-Analysis“ ausführen.  
„Profile CUDA-Application“ auswählen. Alle Experimente („All (Kernel-Level)“) auswählen. Kommando „Launch“ ausführen.  
Auswahl „CUDA-Launches“ und „Memory-Statistics“ treffen.



# Gründe identifizieren

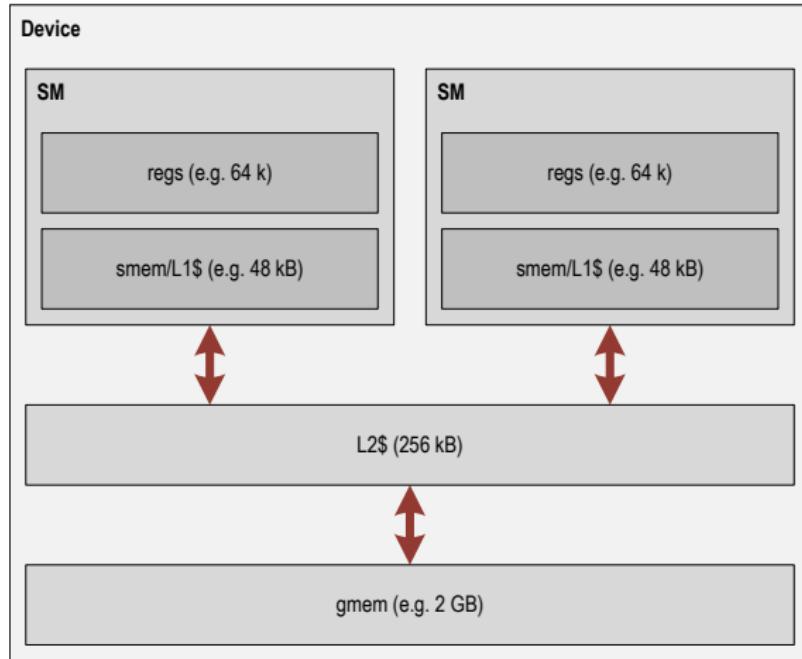
Es können zwei Hauptursachen für die Existenz eines Hotspots verantwortlich sein:

- 1 Die Recheneinheiten stoßen bei der Ausführung des entsprechenden Kernels an ihre Leistungsgrenzen (ab ca. 60 % Auslastung).
- 2 Die Speicherchips stoßen bei der Ausführung des entsprechenden Kernels an ihre Leistungsgrenzen (ab ca. 60 % Auslastung).

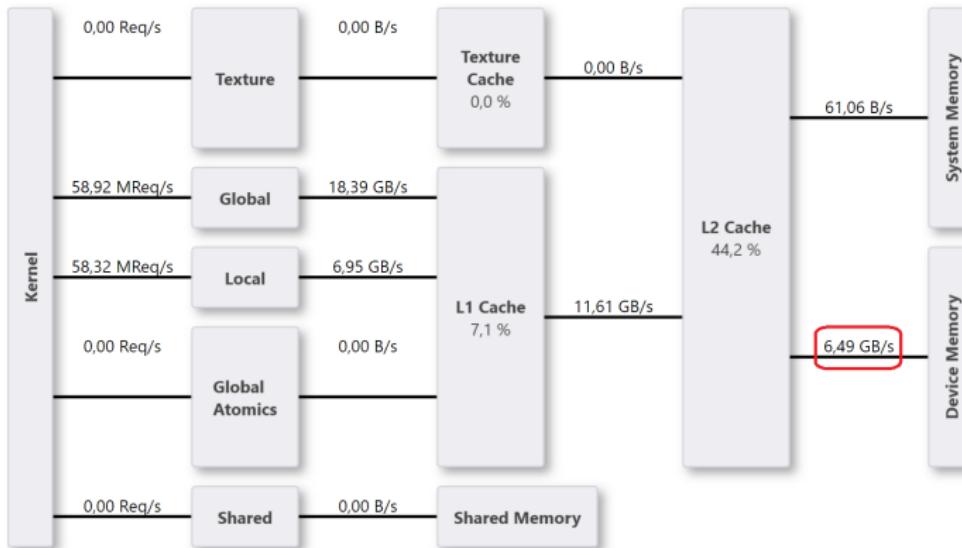
Vier Kombinationen dieser beiden Ursachen sind möglich:

<b><i>compute</i></b>	<b><i>memory</i></b>	<b>Grund für Hotspot</b>
> 60 %		<i>compute bound</i>
	> 60 %	<i>bandwidth bound</i>
> 60 %	> 60 %	<i>compute und bandwidth bound</i> <i>latency bound</i>

# Bandbreite

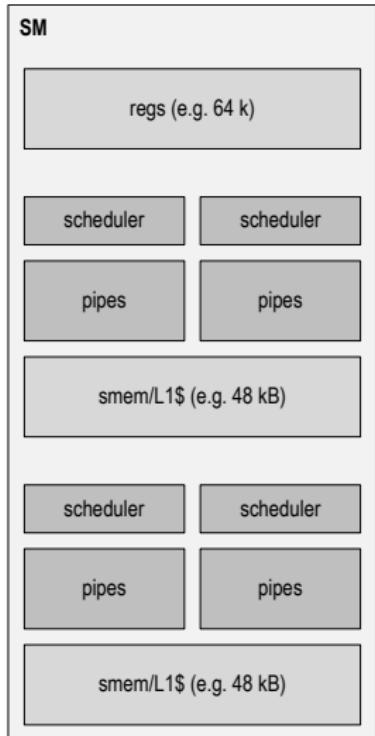


# Bandbreite



*Vermutung:* Die Bandbreite zwischen Device-Memory (*gmem*) und dem L2\$ liegt meist jenseits der 100 GB/s. Der Kernel *gaussian\_filter\_kernel* lastet diese Verbindung mit nur 6,5 GB/s aus. Dieser Kernel wird also nicht durch zu geringe Bandbreite eingebremst.

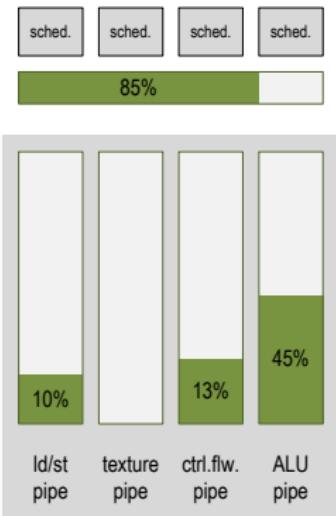
# Befehlsdurchsatz



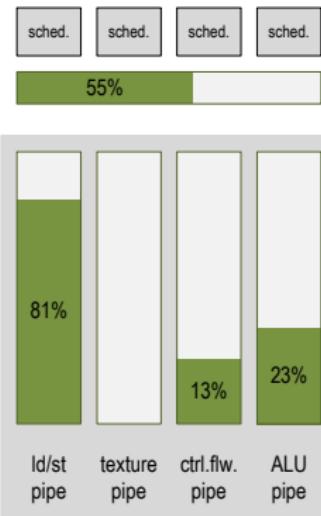
- 1 Jeder Streaming-Multiprozessor besitzt vier Scheduler (bei Kepler- und Maxwell).
- 2 Ein Scheduler emittiert Befehle (*issues instructions*) in die Pipes.
- 3 Jeder Scheduler emittiert (bei Fermi-, Kepler- und Maxwell) bis zu zwei Befehle pro Zyklus (*cycle*).
- 4 Ein Scheduler emittiert Befehle aus einem einzelnen Warp.
- 5 Ist eine Pipe voll, so kann der betreffende Scheduler keine weiteren Befehle mehr emittieren.

# Befehlsdurchsatz

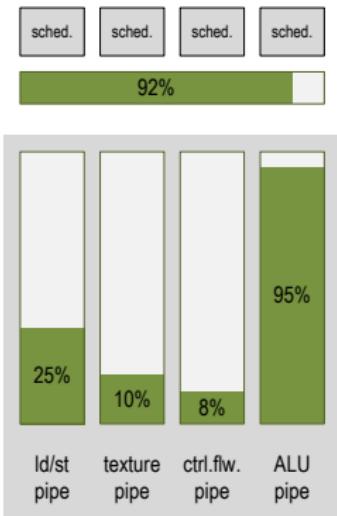
schedulers saturated



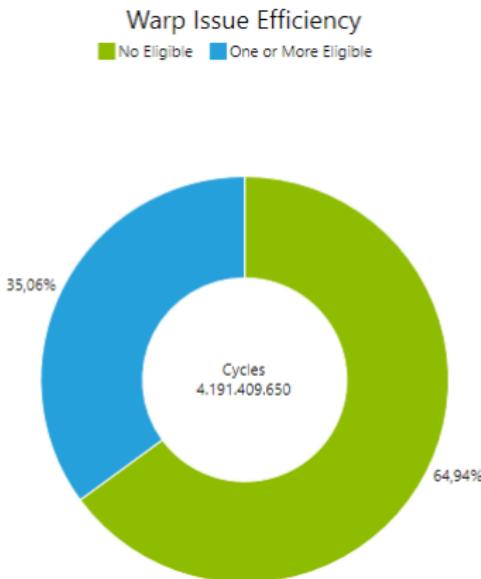
pipe saturated



schedulers and pipes  
saturated



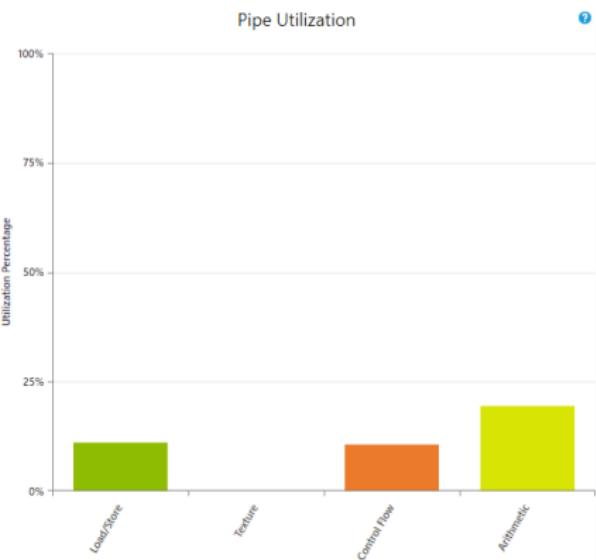
# Warp-Issue-Efficiency



- Der blaue Sektor bezeichnet jenen Anteil der 4,2 Gigazyklen, bei dem ein Scheduler mindestens einen Warp zur Auswahl hatte, um den Kernel `gaussian_filter_kernel` auszuführen. Mit 1,5 Gigazyklen (oder 35,1 %) sind die Scheduler der GPU nicht sehr stark belastet.
- Der grüne Sektor bezeichnet jenen Anteil der Zyklen, zu dem kein Warp selektiert und daher auch kein Befehl emittiert werden konnte (*not eligible*). Je niedriger dieser Wert ist, umso besser.

**Vermutung:** Die Scheduler der GPU sind, da sie zu 64,9 % nichts zu tun haben, kein Problem bei der Ausführung des Kernels `gaussian_filter_kernel`.

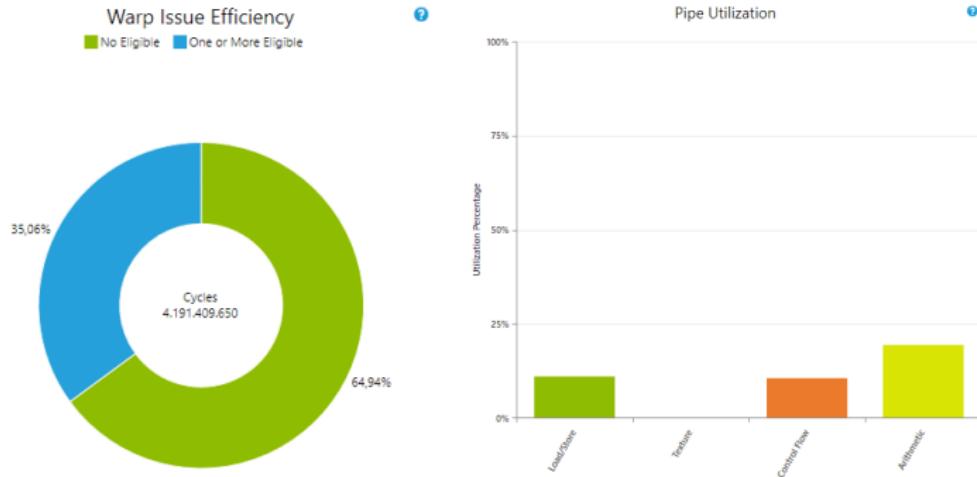
# Pipe-Utilization



- 1 Zeigt die Auslastung der vier wichtigsten Pipelines (*major logical pipelines*) der GPU.
- 2 „Load/Store“ umfasst alle Lese- und Schreiboperationen in den *gmem*, *lmem* und *smem*.
- 3 „Control Flow“ umfasst Befehle wie z. B. *branches*, *jumps*, *function calls*, *loops*, *returns*, *syncs*.
- 4 „Arithmetic“ umfasst Befehle wie z. B. *floating point ops.*, *integer ops.*, *conversion ops.*, *movement ops.*

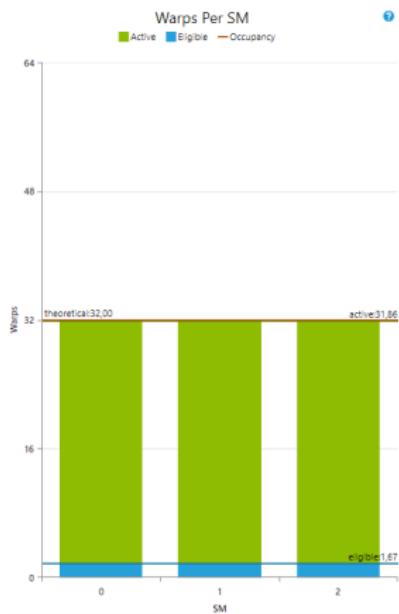
**Vermutung:** Keine der vier Pipelines ist ausgelastet. Daher können sie kein Problem bei der Ausführung des Kernels `gaussian_filter_kernel` sein.

# Befehlsdurchsatz



**Vermutung:** Weder die Scheduler noch die Pipelines der GPU sind ein Flaschenhals bei der Ausführung des Kernels. Der Kernel gaussian\_filter\_kernel kann daher nur *latency bound* sein.

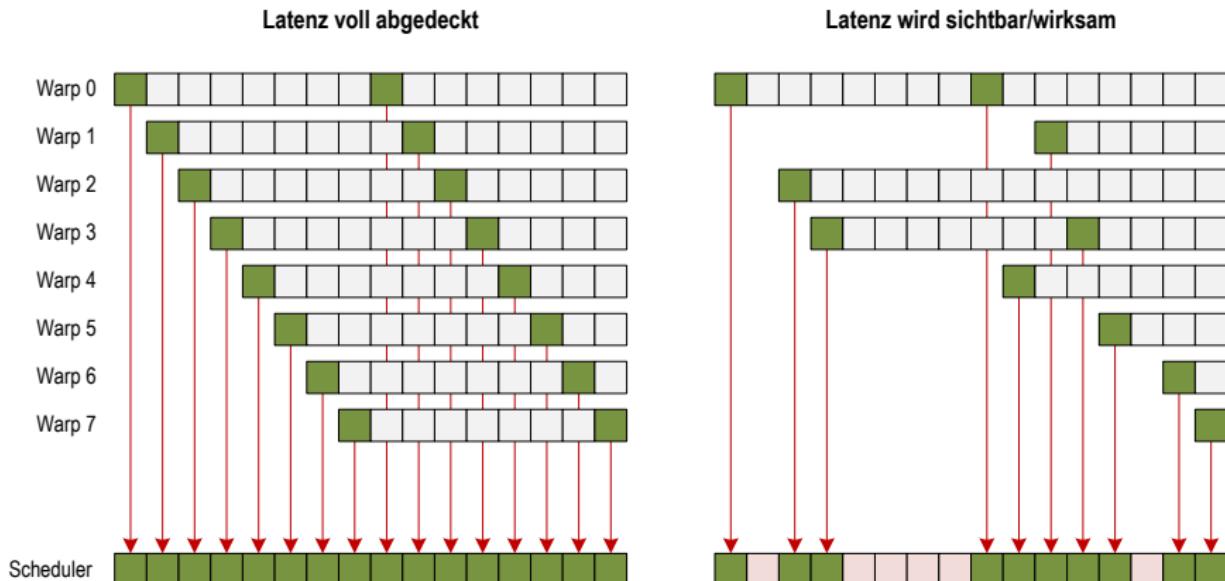
# Occupancy



- 1 Jeder Streaming-Multiprozessor hat begrenzte Ressourcen (bei Kepler z. B. 64 k Register à 32 Bit (ein 4 Byte-Word), 48 kB *smem*, 16 aktive Blöcke, 64 aktive Warps und 2.048 aktive Threads (*full occupancy*)).
- 2 Mit einer – gemäß der aktuellen Konfiguration des Kernels – theoretisch möglichen maximalen Occupancy von 32 Warps, mit 31,9 aktiven Warps pro Zyklus und 1,7 freien Warps pro Zyklus wird die maximal mögliche Occupancy von 64 Warps bei weitem nicht erreicht.

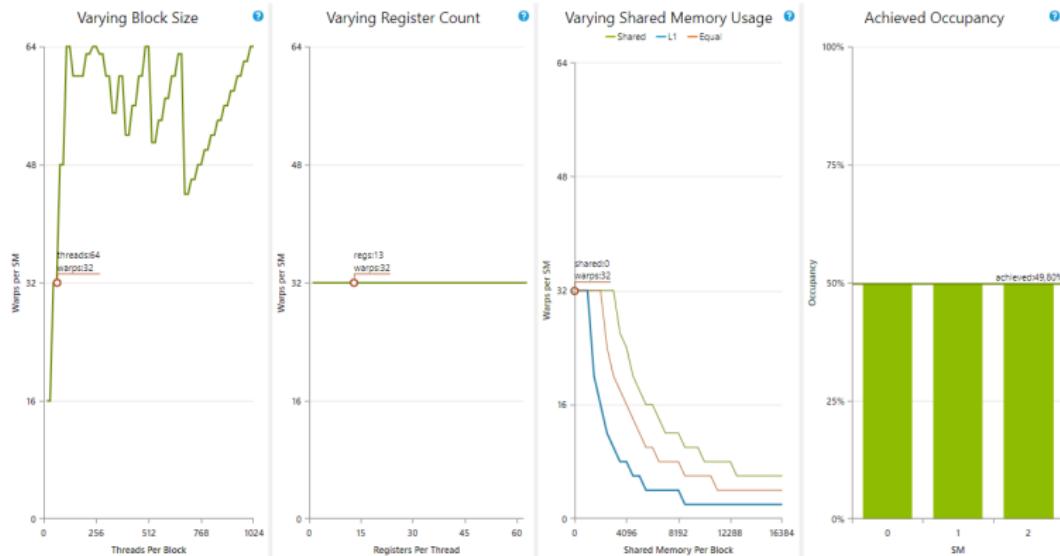
**Vermutung:** Die verfügbaren Ressourcen der Streaming-Multiprozessoren der GPU sind bei einer Occupancy von nur 50 % mit der Ausführung des Kernels `gaussian_filter_kernel` nur **halb ausgelastet**. Die Occupancy muss gesteigert werden.

# Latenz/zu geringe Occupancy



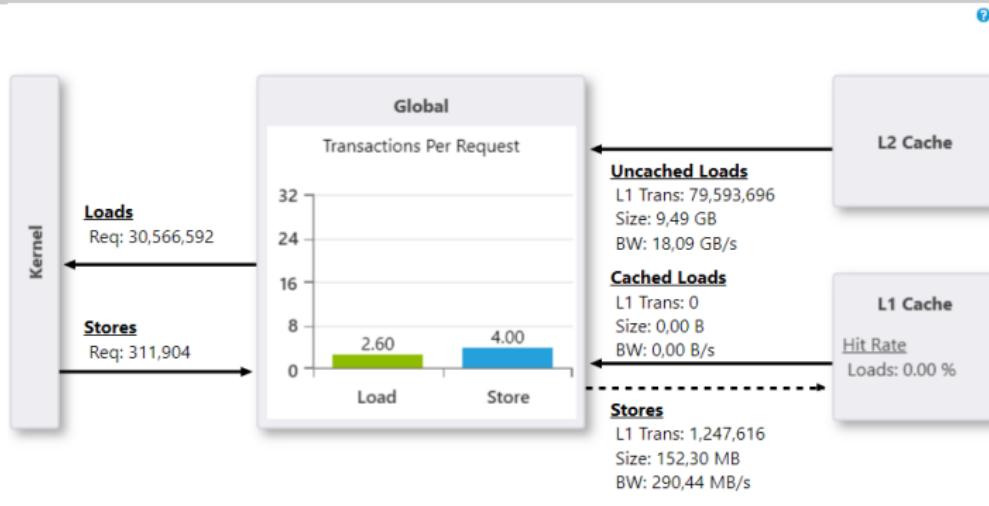
- 1 GPUs versuchen Latenzen zu vermeiden, indem sie viele Tasks gleichzeitig in Arbeit halten („*a lot of work in flight*“).
- 2 Im Bild rechts finden die Scheduler nicht genügend Warps, die ausführungsbereit wären (grüne Quadrate). Es gibt zu wenig aktive Warps.

# Weitere Gründe



**Vermutung:** Die **Blockgröße** von  $8 \times 8$  scheint nicht zu passen. Die Anzahl der Register pro Thread ist aber in Ordnung. *smem* wird vom Kernel **gaussian\_filter\_kernel** (noch) nicht genutzt. Siehe dazu auch den [CUDA Occupancy-Calculator](#).

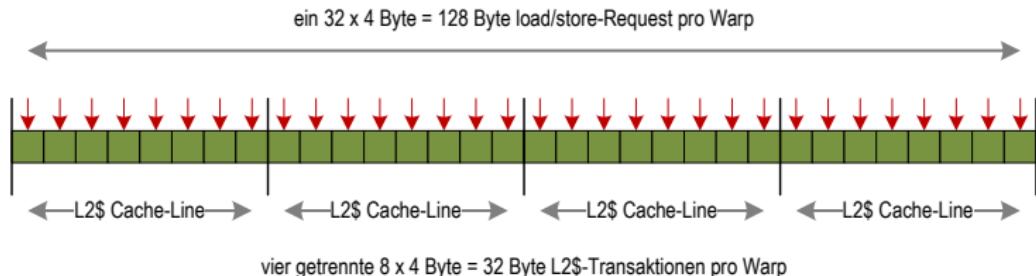
# Weitere Gründe



- 1 Obiges Experiment zeigt, dass der Zugriff vom Kernel aus auf den *gmem* sehr ineffizient erfolgt.
- 2 Pro Memory-Request, den der Kernel `gaussian_filter_kernel` durchführt, erfolgen bis zu vier L2\$ Speichertransfers (gewichtet sind es 2,6 Transaktionen pro Request).
- 3 Die niedrige L1\$-Hit-Rate von 0 % zeigt an, dass alle Daten vom bzw. über den L2\$ kommen.

# Speichertransfers

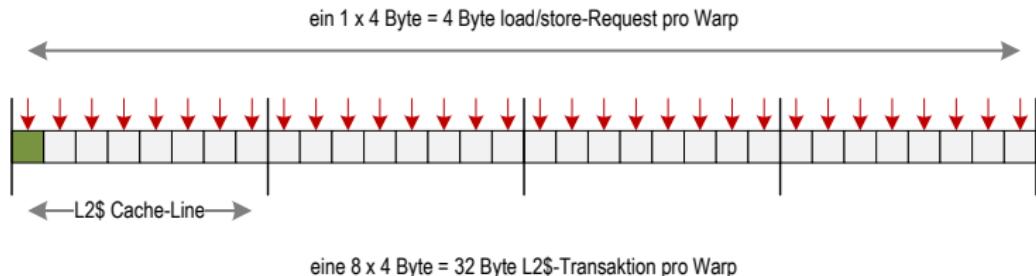
Annahme: Ein Warp führt 32 load/store-Requests à 4 Byte (ein Word) durch. Die angeforderten Daten sind ausgerichtet (*aligned*) und liegen direkt hintereinander (*consecutive*).



- 1 Die 32 Threads dieses Warps greifen auf verschiedene Elemente desselben 128 Byte-Segments zu.
- 2 Dieser Request wird vom L2\$ in vier getrennten 32 Byte-Transaktionen durchgeführt.
- 3 Es werden 128 Byte angefordert und 128 Byte übertragen.

# Speichertransfers

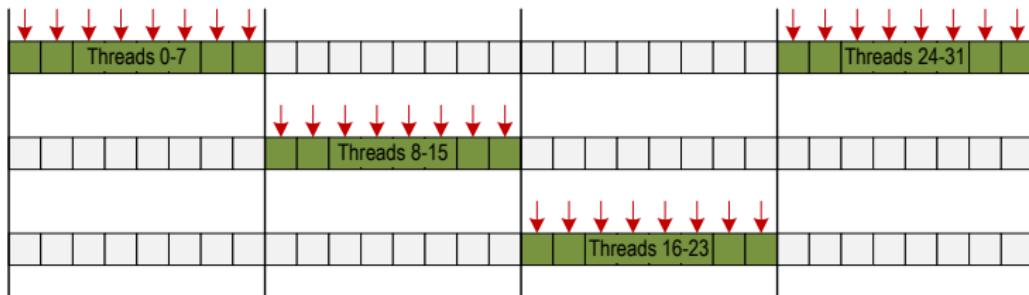
Annahme: Ein Warp führt 32 load/store-Requests à 4 Byte (ein Word) durch. Die angeforderten Daten liegen jedoch jeweils 128 Byte auseinander (*stride*).



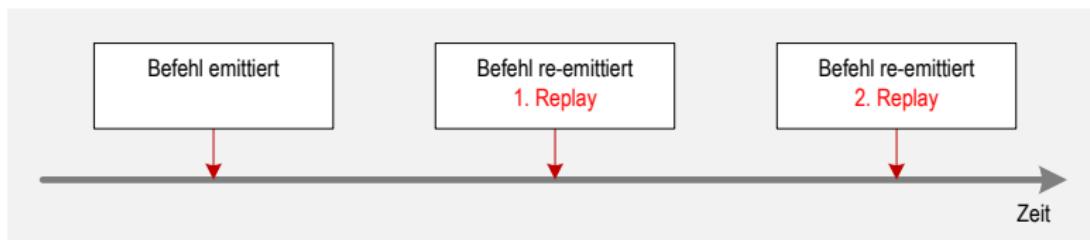
- 1 Jeder Thread liest nur die ersten 4 Byte eines 128 Byte-Segments.
- 2 Dieser Request wird vom L2\$ in 32 getrennten 32 Byte-Transaktionen durchgeführt.
- 3 Es werden 128 Byte angefordert aber 1.024 Byte übertragen.

# Transaktionen und Replays

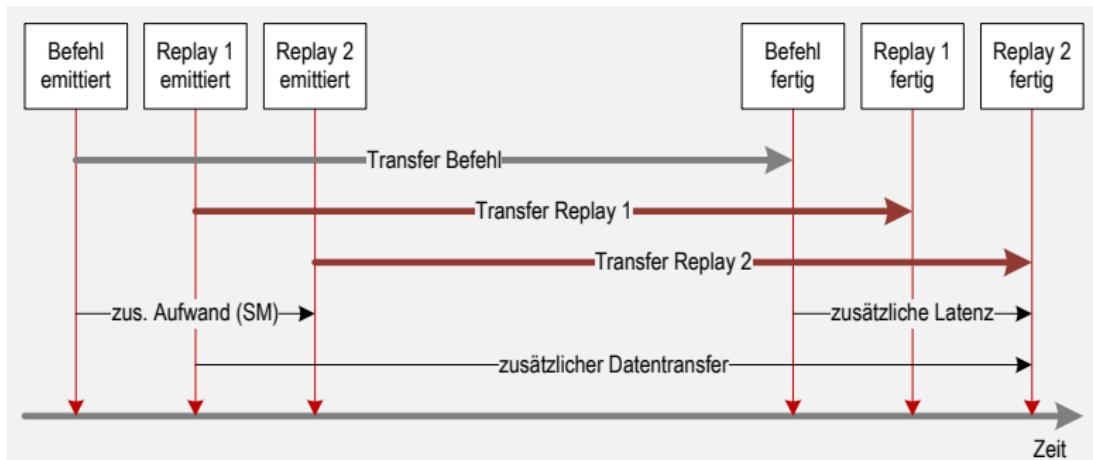
Annahme: Ein Warp liest Daten aus einem Adressbereich, der drei Zeilen à 128 Byte umfasst:



Dieses Zugriffsmuster hat zur Folge, dass der emittierte Lesebefehl zwei Mal wiederholt werden muss (*replay*) und damit aus einem Request drei Transaktionen werden:



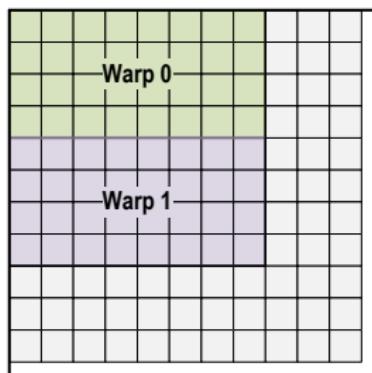
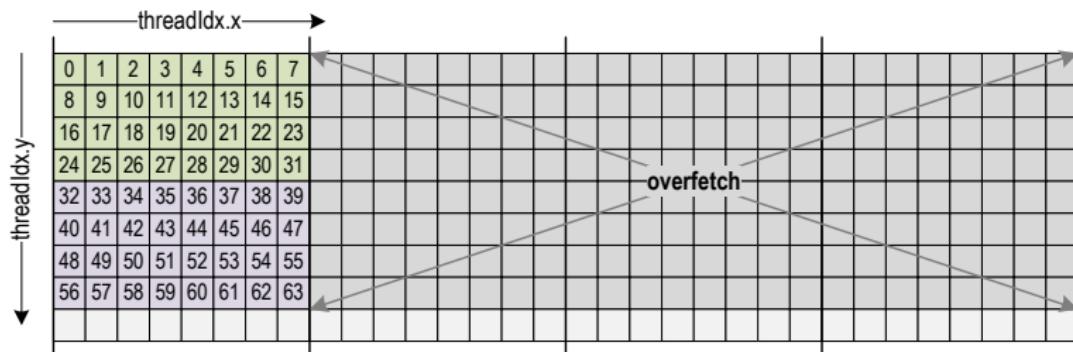
# Transaktionen und Replays



Treten Replays auf, so dauern Requests länger und binden mehr Ressourcen:

- 1 mehr emittierte Befehle (zusätzlicher Aufwand für einen SM)
- 2 mehr Datentransfers
- 3 längere Ausführungsduer (zusätzliche Latenz)

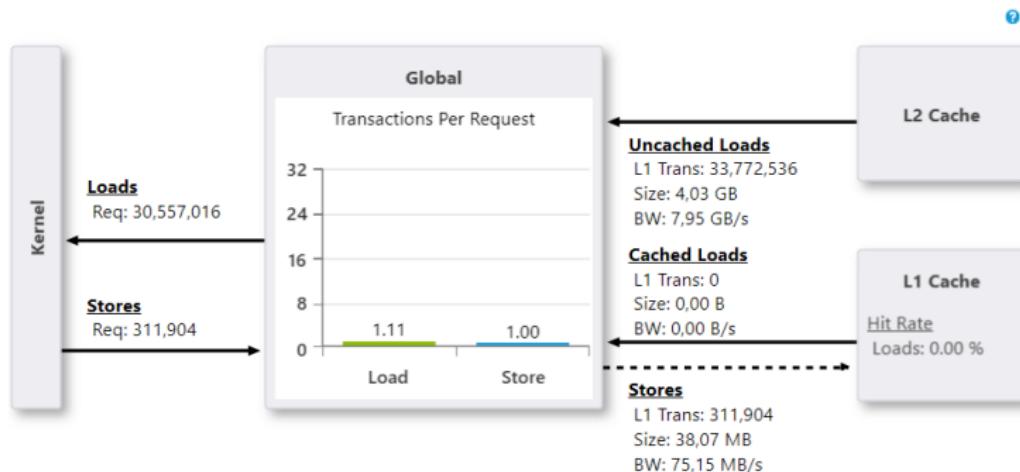
# Blockkonfiguration



- 1 Die Blockgröße, mit der der Kernel konfiguriert wird, beträgt  $8 \times 8$  Threads (zwei Warps pro Block).
- 2 Daraus resultiert ein sehr ungünstiges Speicherzugriffsmuster. Es werden 300 % mehr Daten übertragen, als notwendig (*overfetch*).
- 3 Eine Blockgröße von  $32 \times 2$  Threads wäre vermutlich besser (Version 1 des Demoprogramms).

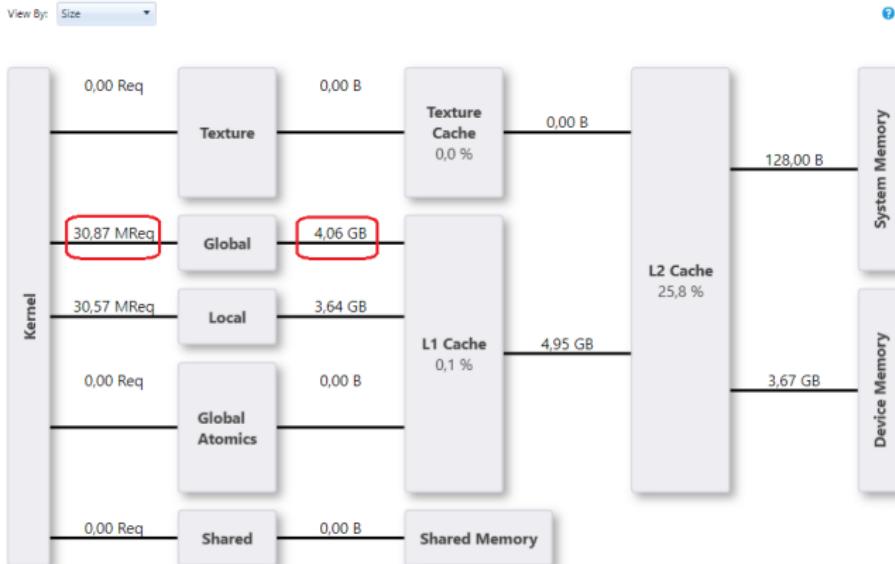
# Version 1: Ergebnisse

Durch die Änderung der Blockgröße von  $8 \times 8$  Threads auf  $32 \times 2$  Threads haben sich die Anzahl der Transaktionen pro Memory-Request von (gewichtet) 2,6 auf 1,1 um 57,6 % reduziert.



Die anderen der bisher analysierten Leistungsparameter (Bandbreite, Warp-Issue-Efficiency, Pipe-Utilization, Warps per SM etc.) haben sich nicht verändert.

# Version 5: Shared-Memory

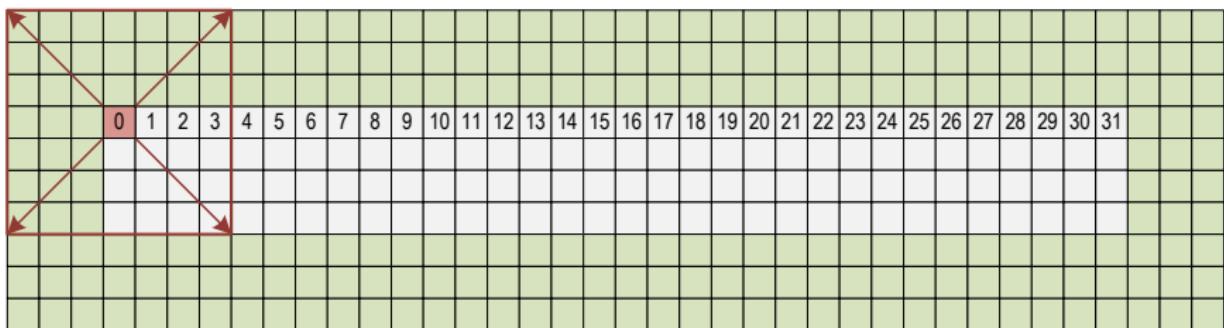


Der Kernel `gaussian_filter_kernel` transferiert die Bilddaten (immerhin 4,1 GB) mit einer Bandbreite von 14,3 GB/s über den entfernten *gmem*.

*Vermutung:* Die Daten müssen näher zum Kernel gebracht werden.

# Version 5: Shared-Memory

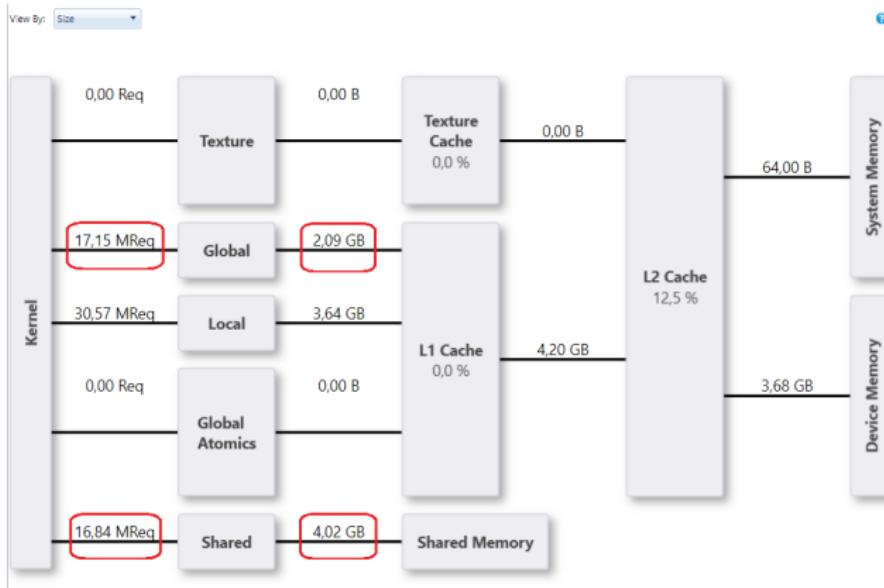
Benachbarte Pixel verwenden für „ihren“ Gaussfilter fast dieselben Pixel:



Verwenden mehrere Threads (eines Blocks bzw. SMs) dieselben Daten, dann können sie sich diese Daten via *smem* teilen:

```
1  __shared__ pfc::byte_t adj [10][64]; // a local variable
2
3  // load adjacent data into shared memory
4
5  __syncthreads(); // wait for all threads to finish loading
6
7  // use data
```

# Version 5: Shared-Memory



Diese Maßnahme hat die Anzahl der Requests an den *gmem* um immerhin 44,4 % reduziert. Die Bandbreite stieg auf 20,4 GB/s.

Alle bisherigen Maßnahmen zusammengenommen erzielen einen Speedup (Version 1 zu Version 5) von 2,7 .

# Quick-Reference

---

## Category: Latency Bound / Occupancy

---

Problem: latency is exposed due to low occupancy

Goal: hide latency behind more parallel work

Indicators:

- occupancy low (< 60 %)
- execution-dependency high

Strategy: increase occupancy by:

- varying block size
  - varying shared-memory usage
  - varying register count
-

# Quick-Reference

---

## Category: Latency Bound / Coalescing

---

Problem: memory is accessed inefficiently (high latency)

Goal: reduce number of transactions per request

Indicators:

- low global load/store efficiency
- high number of transactions per request

Strategy: improve memory coalescing by:

- cooperative loading inside a block
- change block layout
- aligning data
- changing data layout to improve locality

---

# Quick-Reference

---

## Category: Bandwidth Bound / Coalescing

---

Problem: too much unused data clogging memory system

Goal: reduce traffic, move more *useful* data per request

Indicators:

- low global load/store efficiency
- high number of transactions per request

Strategy:

- improve memory coalescing by:
  - cooperative loading inside a block
  - change block layout
  - aligning data
  - changing data layout to improve locality

---

# Quick-Reference

---

## Category: Latency Bound / Shared Memory

---

Problem: long memory latencies are harder to hide

Goal: reduce latency, move data to faster memory

Indicators:

- *smem* not occupancy limiter
- high L2\$-hit rate
- data reuse between threads and smallish working set

Strategy: cooperatively move data closer to SM:

- *smem*
  - registers
  - constant memory
  - texture cache
-

# Quick-Reference

---

## Category: Memory Bound / Shared Memory

---

Problem: too much data movement

Goal: reduce amount of data traffic to/from *gmem*

Indicators:

- higher than expected memory traffic to/from *gmem*
- low arithmetic intensity of the kernel

Strategy: cooperatively move data closer to SM:

- *smem*
  - registers
  - constant memory
  - texture cache
-

# Quick-Reference

---

## Category: Compute Bound / Branch Divergence

---

Problem: diverging threads

Goal: reduce divergence *within* warps

Indicators:

- low warp execution efficiency
- high control flow utilization

Strategy:

- refactor code to avoid intra-warp divergence
- registers
- restructure data to avoid data-dependent branch divergence

---

# Quick-Reference

---

**Category:** Latency Bound / Instruction-Level Parallelism

---

Problem: not enough independent work per thread

Goal: do more parallel work inside single threads

Indicators: high execution dependency (increasing occupancy has little positive effect, still registers available)

Strategy:

- unroll loops (`#pragma unroll`)
- refactor threads to compute  $n$  output values at the same time (code duplication)

---

# Quick-Reference

---

## Category: Compute Bound / Algorithmic Changes

---

Problem: GPU is computing as fast as possible

Goal: reduce computation (if possible)

Indicators: compute bound problem (speedup only with less computation)

Strategy:

- pre-compute or store (intermediate) results
- trade memory for compute time
- use a computationally less expensive algorithm

---

# Literatur

-  NVIDIA Corporation.  
CUDA Toolkit Documentation, 2015.  
Verfügbar im Internet unter <http://docs.nvidia.com/cuda>.
-  NVIDIA Corporation.  
NVIDIA Nsight Visual Studio Edition, 2015.  
Verfügbar im Internet unter [http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/nvidia\\_nsight.htm](http://docs.nvidia.com/gameworks/index.html#developertools/desktop/nsight/nvidia_nsight.htm).