

## 2.6 Graphikprozessoren

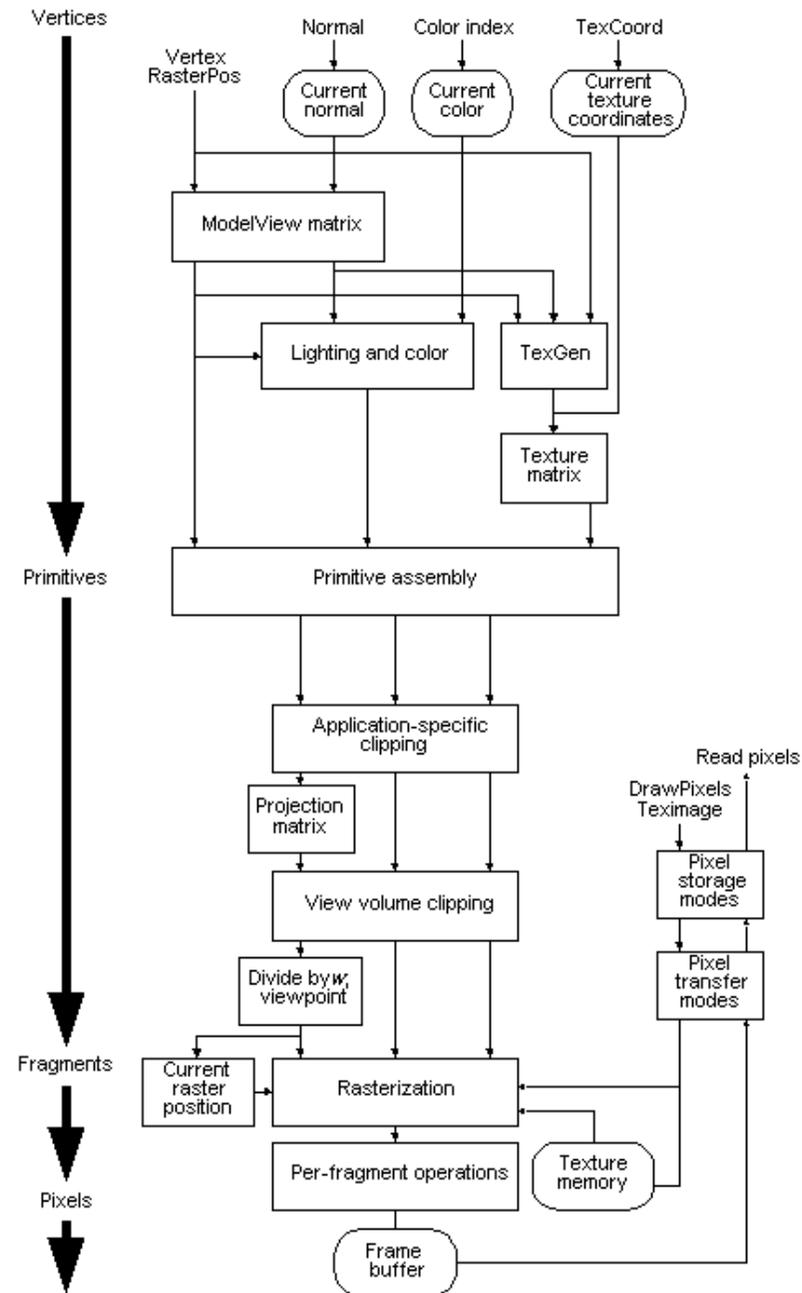
Peter Marwedel  
Informatik 12  
TU Dortmund

2012/04/16

# 3D Grafikpipeline (OpenGL)

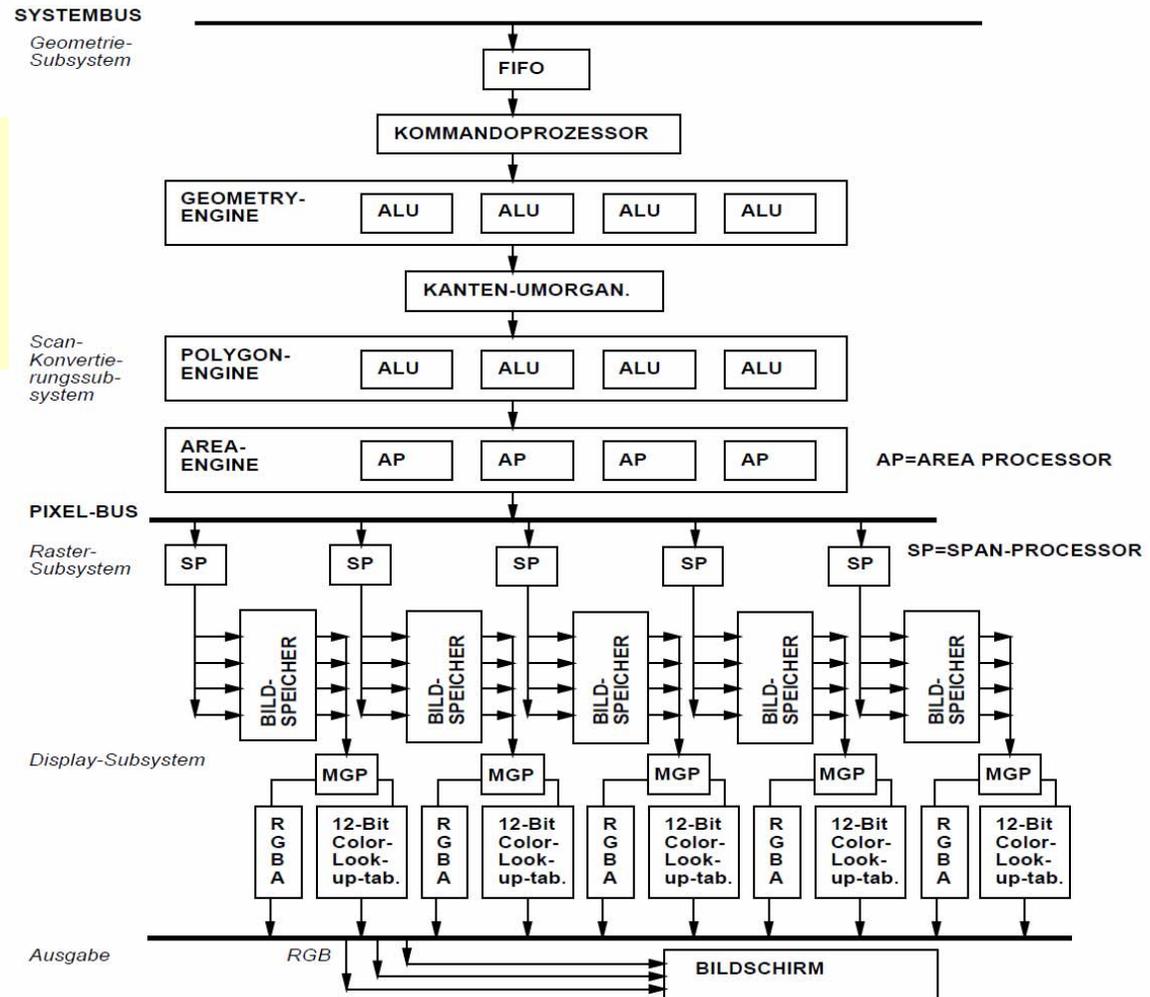
Erzeugung von Bildern  
üblicherweise Fließband-  
artig organisiert.  
Beispiel: OpenGL

☞ Siehe Vorlesungen des LS7 zu  
den einzelnen Graphikoperationen



# Struktur früher Graphikbeschleuniger

[Müller-Schloer, Schmitter:  
RISC-Workstation-  
Architekturen, Springer-  
Verlag, 1991]



Neuere  
Graphikbeschleuniger  
stärker programmierbar

Abbildung 5.80: Graphikbeschleuniger der Serie *Silicon Graphics GTX*

---

# General Purpose GPU (GPGPU) Programming

---

Kann die Programmierbarkeit von GPUs genutzt werden, um auch außerhalb der Grafikprogrammierung Anwendungen zu beschleunigen?

Prinzipien (NVIDIA):

- Entwicklung eines (portablen) C-Dialects für GPUs, **CUDA** (*Compute Unified Device Architecture*)
- Einheit der Parallelverarbeitung ist ein *thread*
- Gruppen von 32 *threads* heißen *thread block (warp)*
- *Thread blocks* werden auf *multithreaded SIMD* Prozessoren ausgeführt

# Konventionen

- Für Funktionen: `_device_`, `_global_`: GPU, `_host_`: CPU
- Für Variablen: `_device_`, `_global_`: GPU-Speicher zugeordnet, von allen SIMD-Prozessoren zu nutzen.
- Erweiterter Funktionsaufruf  
`name<<<dimGrid,dimblock>>>(…Parameter-Liste…)` mit  
dimGrid = Elemente im Grid (in Blöcken) und  
dimBlock = Elemente im Block (in *threads*)
- Spezielle Variablennamen:  
**blockIdx** = aktueller Block  
**threadIdx** = aktueller *thread*  
**blockDim** = Anzahl *threads* im Block



# DAXPY in C und CUDA

```
//Invoke DAXPY
daxpy(n, 2.0, x, y)
//DAXPY in C
void daxpy(int n, double a, double*x,double *y)
{ for (int i=0;i<n;++i) y[i]=a*x[i]+y[i]; }
```

```
//Invoke DAXPY with 256 threads per block
_host_
int nblocks = (n+ 255)/256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y)
//DAXPY in CUDA
_device_
void daxpy(int n, double a, double *x, double*y)
{ int i = blockIdx*blockDim.x + threadIdx.x;
  if (i<n) y[i] = a* x[i] +y[i];
}
```

Keine Abhängigkeit  
zwischen  
Schleifeniterationen  
(*no loop-carried  
dependences*)

Über *thread*-Grenzen  
hinaus benachbarte  
Speicheradressen  
erforderlich, um gute  
Speicherperformanz zu  
erreichen

# Vergleich von Bezeichnungen

Treffende Bezeichn.	Außerhalb GPUs	CUDA/NVIDIA	Bedeutung
<i>Vectorizable Loop</i>	<i>Vectorizable Loop</i>	<i>Grid</i>	Vektorisierbare Schleife
<i>Body of vectorizable loop</i>	<i>Body of (strip-mined) vectorizable loop</i>	<i>Thread block</i>	
<i>Sequence of SIMD Lane Operations</i>	<i>One iteration of a loop</i>	<i>CUDA Thread</i>	
<i>Thread of SIMD instructions</i>	<i>Thread of vector instructions</i>	<i>Warp</i>	32 threads
<i>SIMD instruction</i>	<i>Vector instruction</i>	<i>PTX instruction</i>	PTX = Hardwarebef.
<i>Multithreaded SIMD processor</i>	<i>(Multithreaded) vector processor</i>	<i>Streaming multiprocessor</i>	
<i>Thread block scheduler</i>	<i>Scalar processor</i>	<i>Giga Thread Engine</i>	
<i>SIMD thread scheduler</i>	<i>Thread scheduler in multithreaded CPU</i>	<i>WARP scheduler</i>	
<i>SIMD Lane</i>	<i>Vector Lane</i>	<i>Thread processor</i>	

---

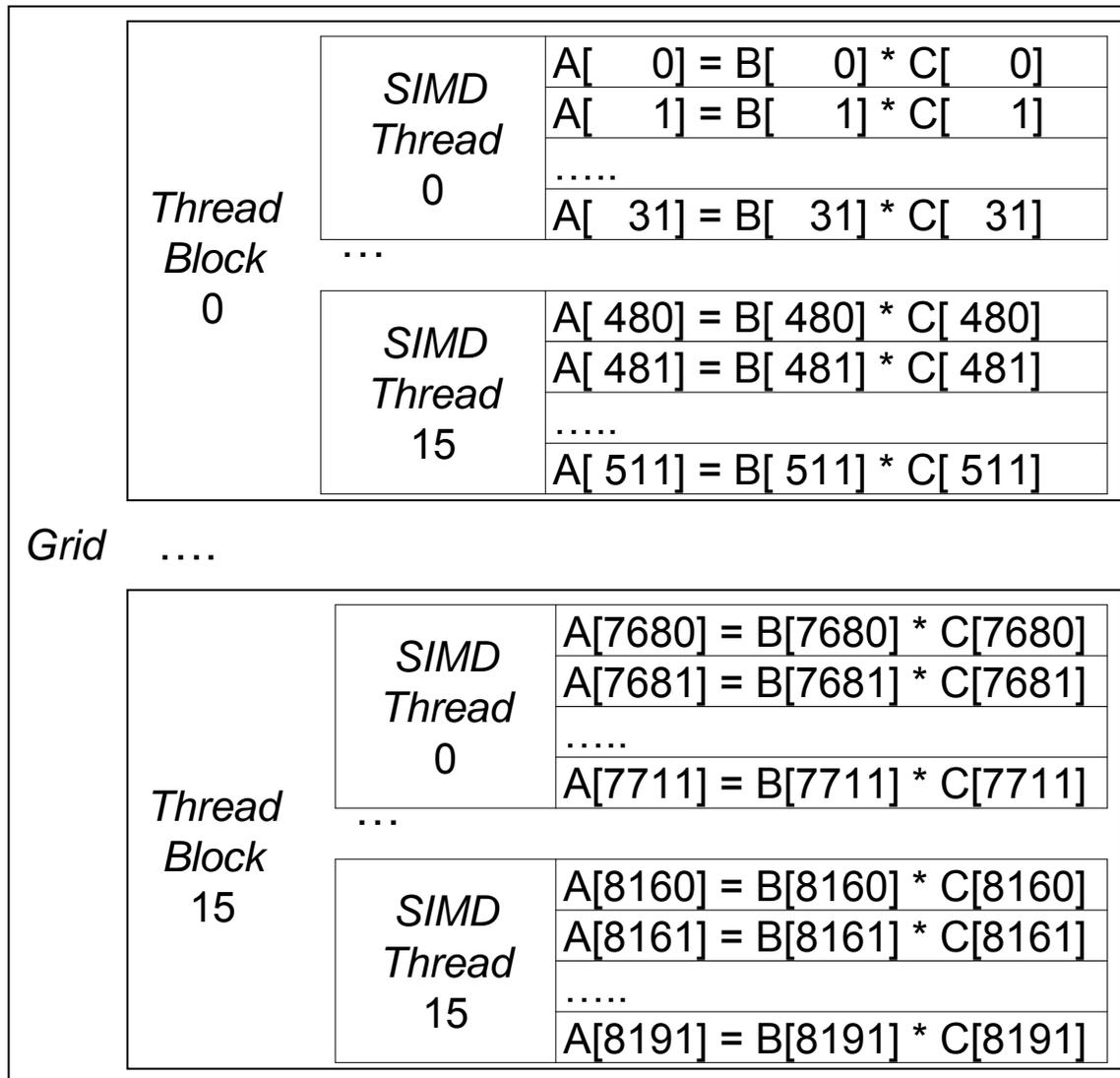
# Beispiel

---

Multiplikation zweier Vektoren der Länge 8192

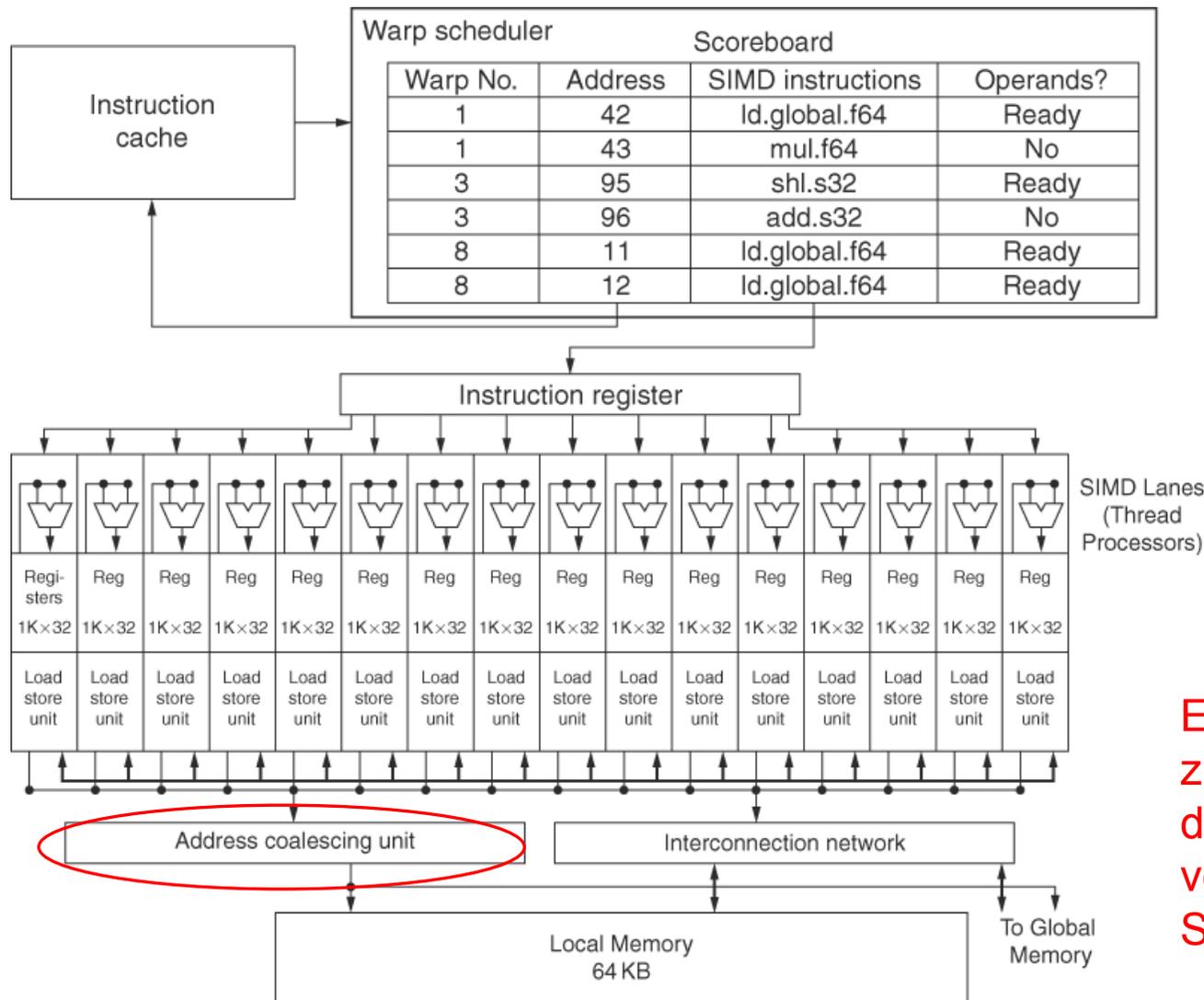
- Code, der alle Elemente erfasst, heißt *grid*
- **SIMD-Befehl** erfasst 32 Elemente (*threads*)
- Ein *thread block* umfasst 512 Elemente
- Das *Grid* enthält mithin 16 Blöcke
- Durch den *thread block scheduler* wird jedem Block ein *multithreaded SIMD* Prozessor zugewiesen
- Fermi-GPUs haben bis zu 16 *multithreaded SIMD* Prozessoren

# Beispiel: Abbildung einer Vektormultiplikation



- Verteilung einer Multiplikation von Vektoren mit 8192 Elementen.
- Der *thread block scheduler* weist *thread blocks* multithreaded SIMD Prozessoren zu.
- Der Hardware *thread scheduler* wählt aus, welcher *thread* als nächstes ausgeführt wird.
- Max. 16 *threads/block* bei Tesla und max. 32 bei Fermi-Prozessoren

# Vereinfachtes Block Diagram eines *Multithreaded SIMD*-Prozessors



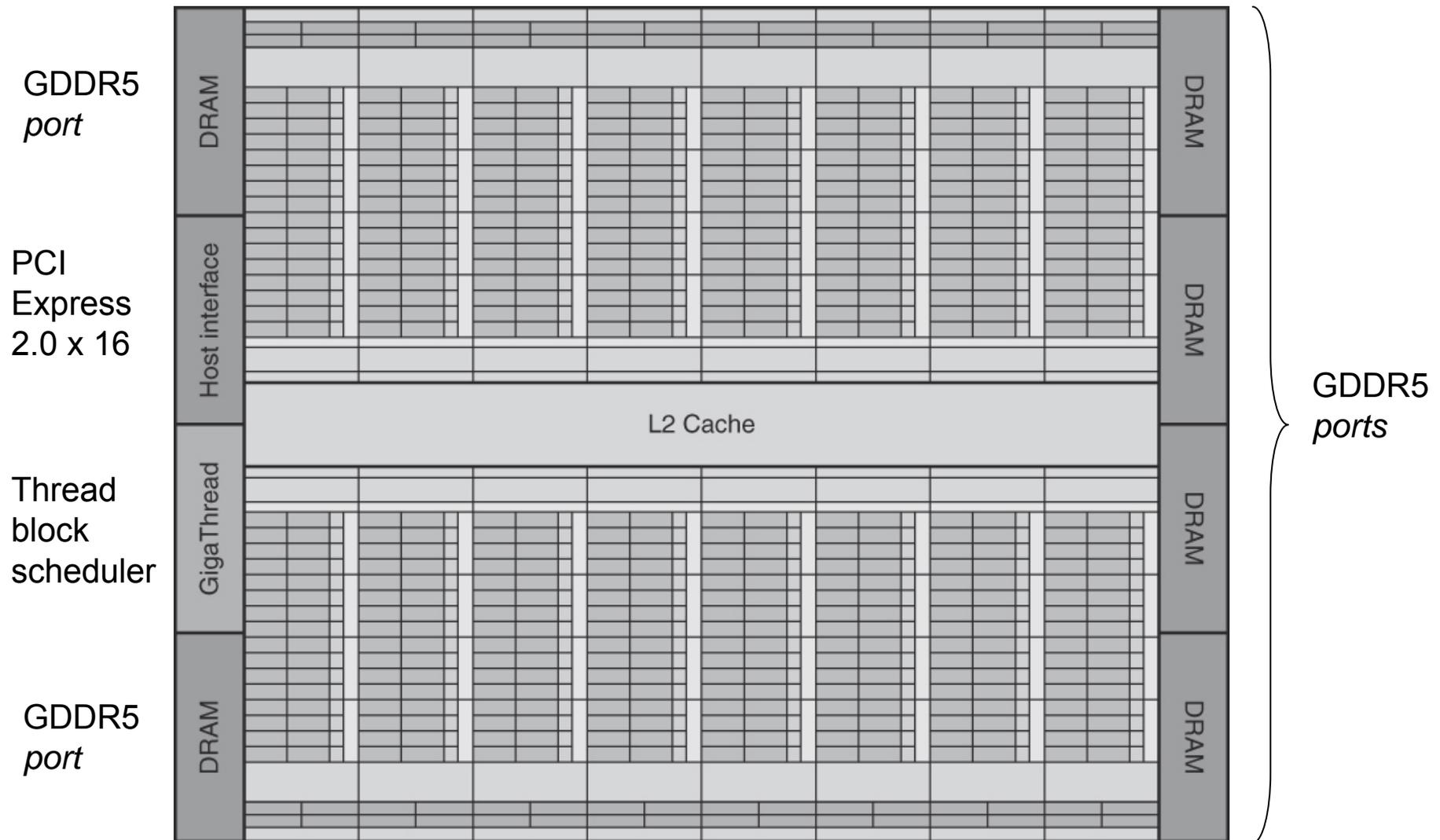
*thread scheduling* kann unter 48 unabhängigen *threads* einen ausführbereiten auswählen

Max. 16 *threads* für Tesla- und 32 *threads* für Fermi-Prozessoren

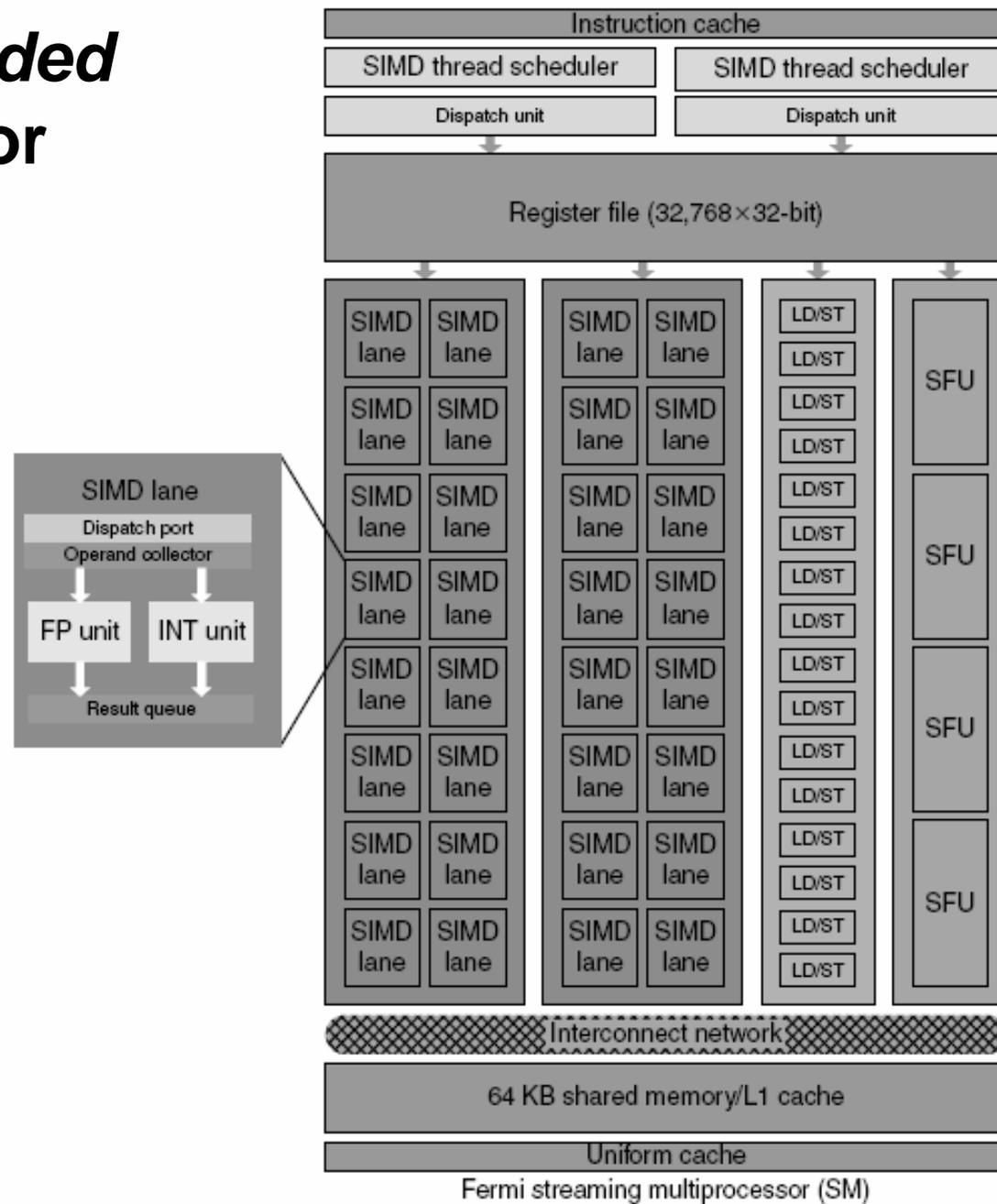
Effiziente Speicherzugriffe entstehen erst durch Zusammenfassen von benachbarten Speicherzugriffen

# Layout der Fermi GTX 480 GPU

## 16 *Multithreaded SIMD*-Prozessoren



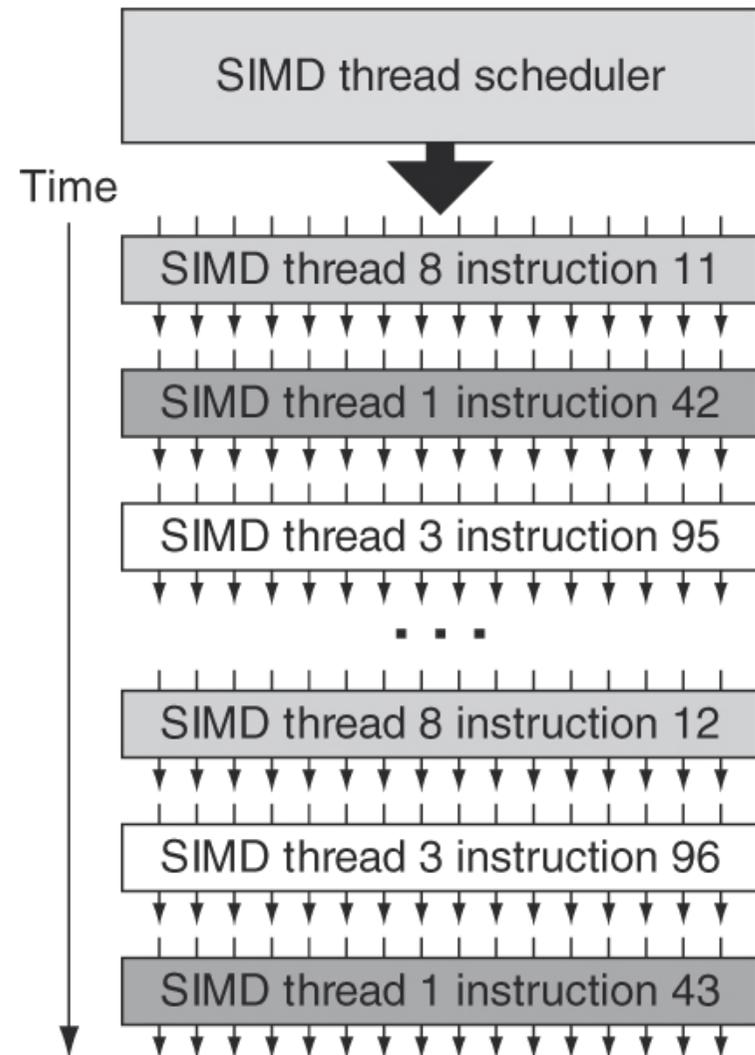
# Fermi *Multithreaded* SIMD Processor



# Auswahl von ausgeführten *threads* durch den SIMD *thread scheduler*

Beispiel einer Ausführung von *threads*:

Annahme:  
durch *scheduling* innerhalb der *multithreaded SIMD*-Prozessoren lässt sich die Speicherlatenzzeit verstecken, ausreichend viele vorhandene *threads* vorausgesetzt.



# NVIDIA PTX GPU Befehle, Auswahl (1)

Gruppe	Befehl	Bedeutung	Kommentar
Arithmetik	type $\in \{.s32, .u32, .f32, .s64, .u64, .f64\}$		Datentypen
	add.type d,a,b	$d=a+b$	
	mad.type d,a,b,c	$d=a*b+c$	
	min.type d,a,b	$d=(a<b)?a:b$	
	selp.type d,a,b	$d=p? a:b$	<i>select with predicate</i>
Spezial- befehle	type $\in \{.f32, .f64 \text{ (teilweise)}\}$		Datentypen
	sqrt.type d,a	$d=\text{sqrt}(a)$	
	rsqrt.type d,a	$d=1/\text{sqrt}(a)$	
	sin.type d,a	$d=\text{sin}(a)$	
	lg2.type d,a	$d=\log_2(a)$	
	ex2 d,a	$d=2^a$	
Logik	type= $\{.pred, .b32, .b64\}$		Datentypen
	and.type	$d= a \wedge b$	
	or.type	$d= a \vee b$	
	shl.type	$d=a \ll b$	

## NVIDIA PTX GPU Befehle, Auswahl (2)

Gruppe	Befehl	Bedeutung	Kommentar
Speicher	space $\in$ { .global, .shared, .local, .const }		Datentypen
	ld.space.type d[a+offset]	d=Mem[a+offset]	
	st.space.type d[a+offset]	d=Mem[a+offset]	
	tex.nd.dtyp,btyp	d=text2d(a,b)	
	atom.spc.op.type	d=Mem[a], Mem[a]=d op b	Atomare Operation
Kontrolle	branch t	if (p) goto t	Bedingter Sprung
	call	Funktionsaufruf	
	ret	Rückkehr vom Funktionsaufruf	
	bar.sync	Warte auf <i>threads</i>	<i>Barrier</i> Synchronisation
	exit	d=2 <sup>a</sup>	Beende <i>thread</i>

- PTX=*Parallel Thread Execution*
- PTX-ISA ist eine Abstraktion des Hardwarebefehlssatzes
- Übersetzung in Hardware-ISA erfolgt durch Software
- PTX benutzt virtuelle Register

---

# Beispiel: DAXPY in PTX

---

```
shl.s32      R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.s32      R8, R8, threadIdx ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]      ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4          ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2          ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0      ; Y[i] = sum (X[i]*a + Y[i])
```

---

# Realisierung von IF-Statements

---

GPUs benutzen Masken zur Realisierung von IF-Statements.

Masken:

- Einträge enthalten Masken für jede *SIMD lane*
- Legen fest, welche *threads* Ergebnisse abspeichern  
(alle *threads* werden ausgeführt)

Pro *thread-lane* 1-bit *predicate register*, durch Programmierer angegeben.

# Beispiel

```
if (X[i] != 0)
    X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

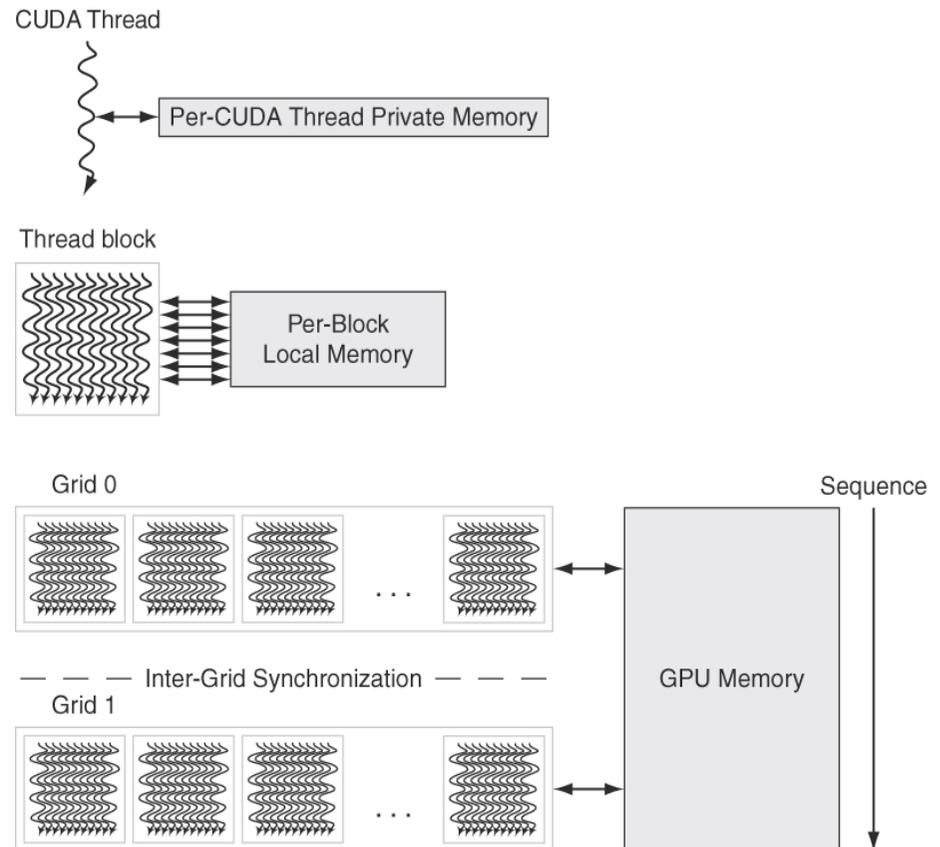
```
ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0 ; P1 is predicate register 1
@!P1, bra ELSE1, *Push ; Push old mask, set new mask bits
; if P1 false, go to ELSE1
```

```
ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2 ; Difference in RD0
st.global.f64 [X+R8], RD0 ; X[i] = RD0
@P1, bra ENDIF1, *Comp ; complement mask bits
; if P1 true, go to ENDIF1
```

```
ELSE1: ld.global.f64 RD0, [Z+R8] ; RD0 = Z[i]
st.global.f64 [X+R8], RD0 ; X[i] = RD0
ENDIF1: <next instruction>, *Pop ; pop to restore old mask
```

# NVIDIA GPU Speicherstrukturen

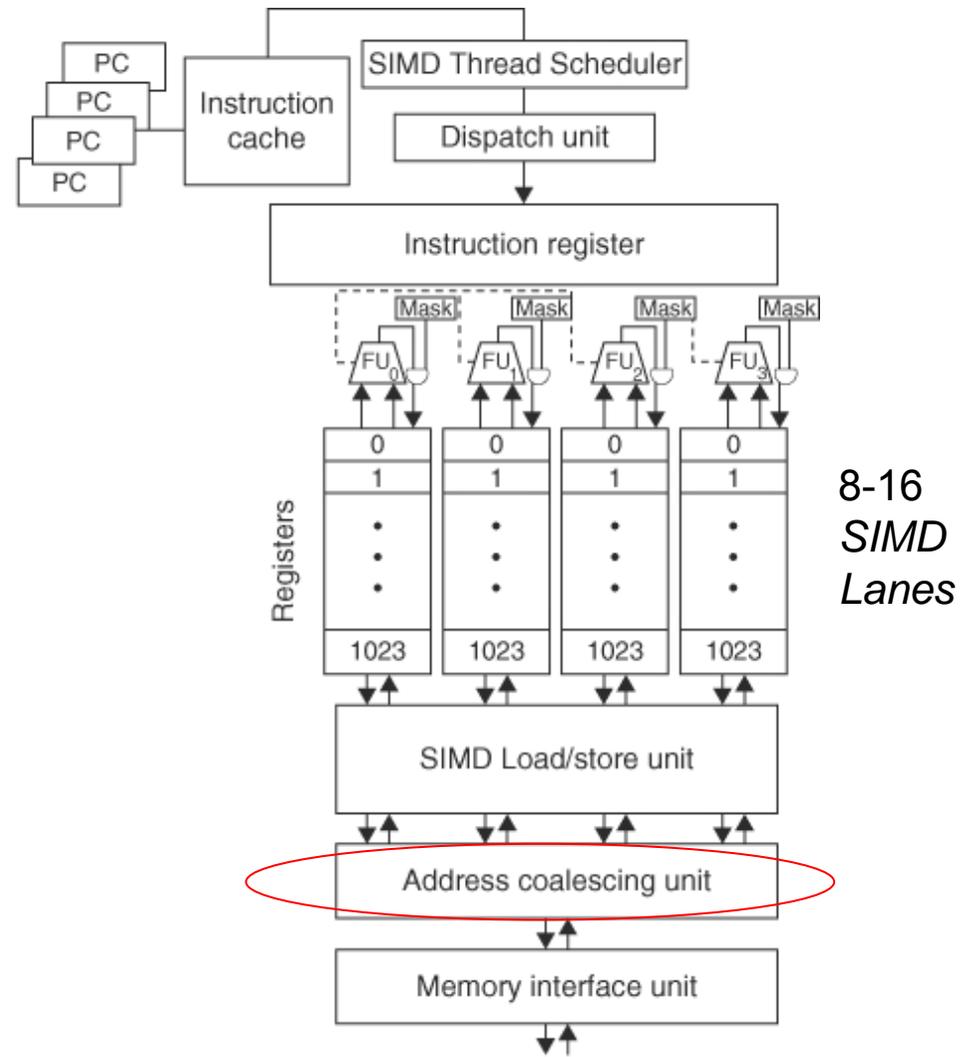
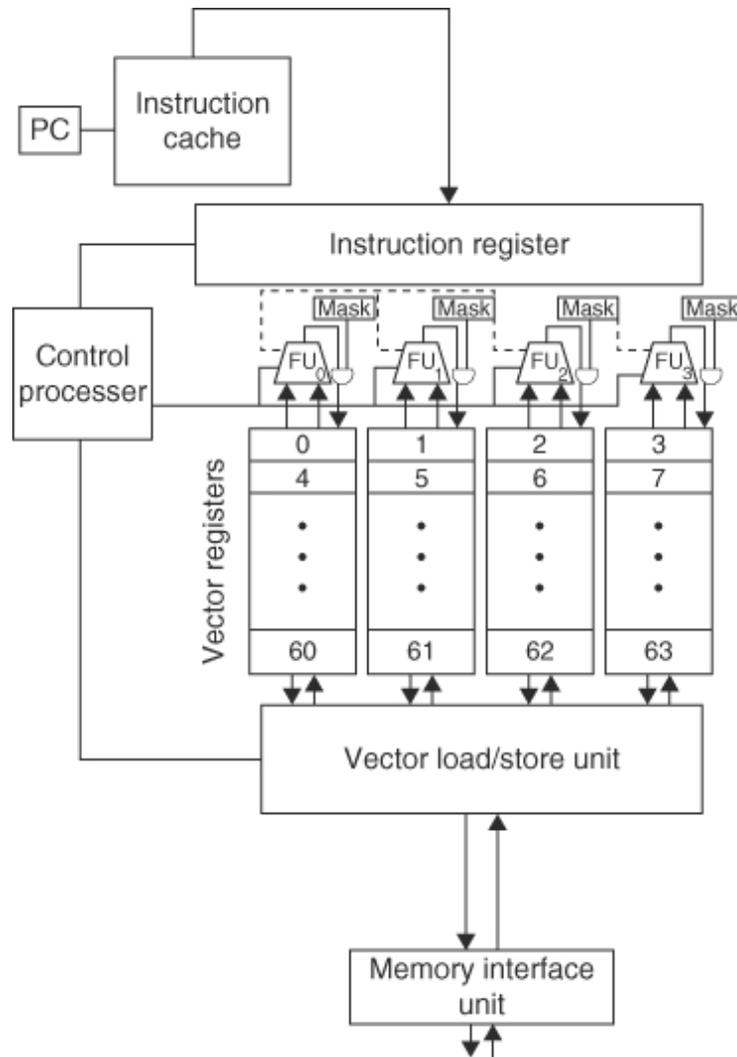
- 1. Private memory:** privater Abschnitt im *off-chip*-DRAM-Speicher. Für:
  - Stack
  - ausgelagerte Register
  - private Variablen
- 2. Local Memory:** von *threads* innerhalb eines *multithreaded SIMD*-Prozessors gemeinsam nutzbarer Speicher
- 3. GPU-Memory:** *Off-Chip*-Speicher, der von allen *thread*-Blöcken und auch von der CPU genutzt werden kann.



## Vergleich von Bezeichnungen (2)

Treffende Bezeichnung	Außerhalb GPUs	CUDA/NVIDIA	Bedeutung
<i>GPU memory</i>	<i>Main memory</i>	<i>Global memory</i>	(Teilweise o. Cache)
<i>Private memory</i>	<i>Thread or stack local storage (OS)</i>	<i>Local memory</i>	Im Hauptspeicher (!), nur innerhalb eines <i>threads</i> nutzbar
<i>Local memory</i>	<i>Local memory</i>	<i>Shared Memory</i>	Schneller lokaler Speicher für einen SIMD-Prozessor, Zur Kommunikation innerhalb von <i>blocks</i>
		<i>Texture memory</i>	Gecachter CPU-Hauptspeicher
<i>CPU memory</i>		<i>Durch CPU zum GPU-Speicher kopierbar, durch GPU zum CPU-Speicher kopierbar.</i>	
<i>SIMD Lane registers</i>	<i>Vector lane registers</i>	<i>Thread processor registers</i>	

# Ähnlichkeiten zwischen Vektor- und Grafikprozessoren



---

# NVIDIA GPU Architektur

---

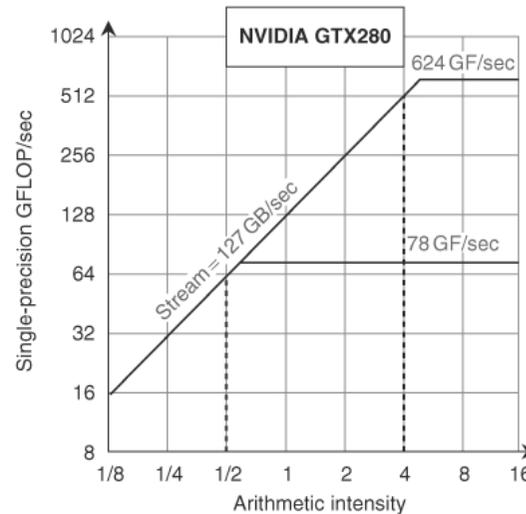
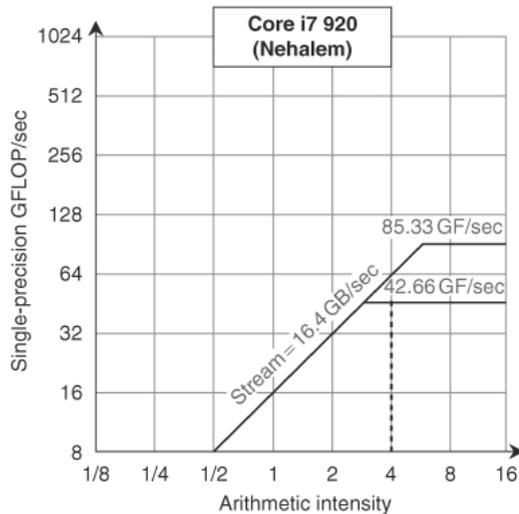
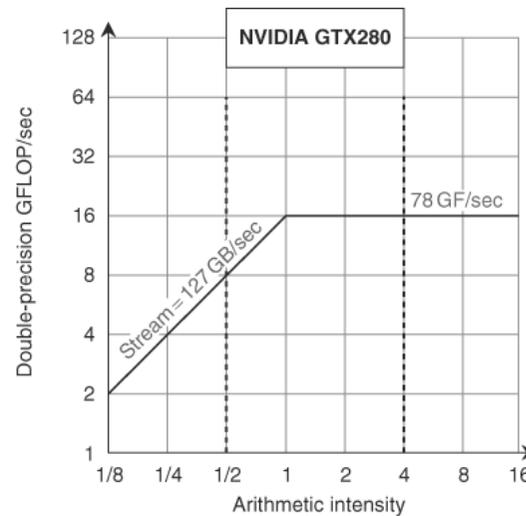
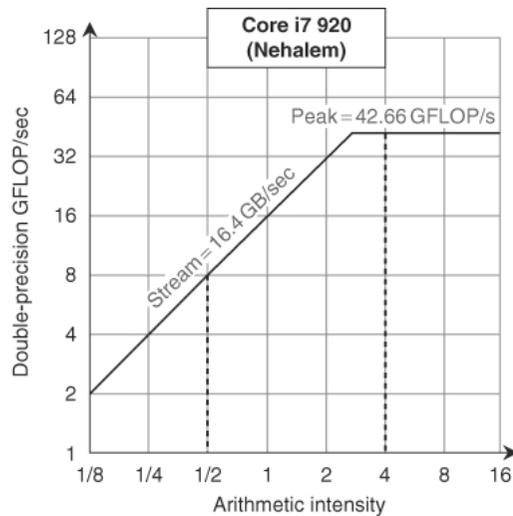
Ähnlichkeiten zu Vektormaschinen:

- Beide sind geeignet für datenparallele Anwendungen
- Beide enthalten *gather/scatter*-Unterstützung
- Beide enthalten Maskenregister
- Beide haben große Registerfiles

Unterschiede:

- Kein Skalarprozessor bei GPUs
- GPU benutzt Multithreading zum Verstecken von Speicherlatenz
- GPU hat viele funktionelle Einheiten, statt wenigen mit vielen Fließbandstufen

# Vergleich Grafikprozessor/CPU mit dem Roofline-Modell



- To hit the highest computation rate on the Core i7 you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds.
- For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors.
- The DP FP performance ceiling is also in the bottom row to give perspective.

---

# Zusammenfassung

---

- Stärkere Programmierbarkeit von Grafikprozessoren erlaubt deren Einsatz für allgemeines Rechnen, v.a. für Vektor-orientiertes Rechnen
- NVIDIA setzt mit Grafikprozessoren und CUDA Standards.
- Einheit der Parallelausführung ist ein *thread*
- *threads* werden in Blöcke zusammengefasst
- Die Blöcke machen eine parallelisierbare Schleife aus
- *thread* Blöcke werden den *multithreaded SIMD*-Prozessoren zugewiesen
- Innerhalb der Prozessoren werden die auszuführenden *threads* dynamisch per Hardware ausgewählt
- Lokalität zwischen *threads* wg. Speicherzugriffen wichtig