TECHNISCHE UNIVERSITÄT DRESDEN

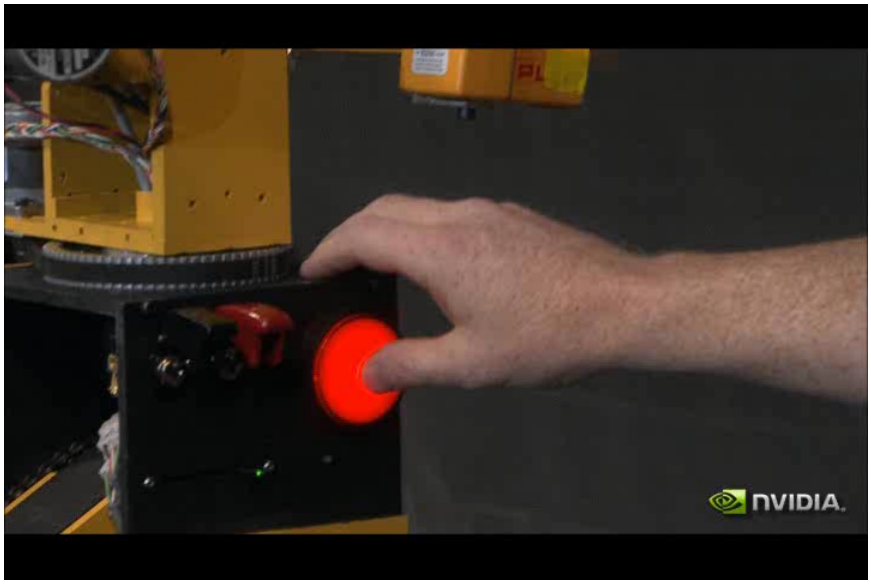
Einführung in die Technische Informatik

GPU Programmierung

Wolfgang E. Nagel (wolfgang.nagel@tu-dresden.de)
Guido Juckeland (guido.juckeland@tu-dresden.de)

WARUM PARALLEL RECHNEN?

Eine Recheneinheit – Vielseitig, aber alleine

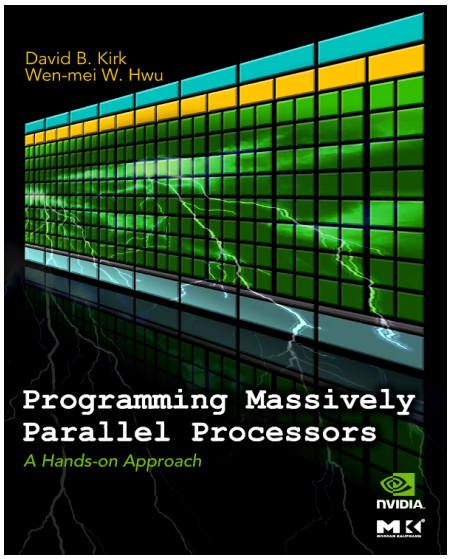


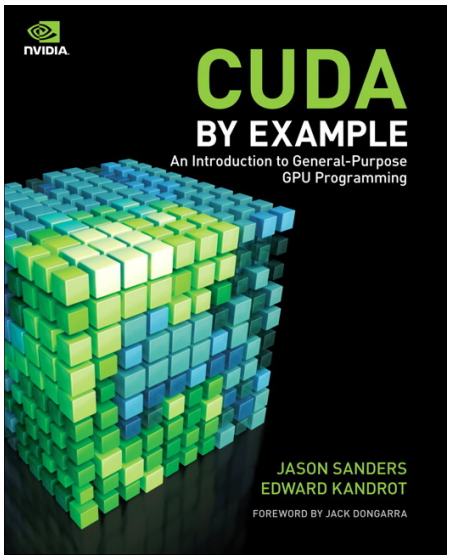
www.nvidia.com/content/nvision2008/art_science/index.html

Viele Recheneinheiten – Dumm, aber stark im Team

www.nvidia.com/content/nvision2008/art_science/index.html

Literatur

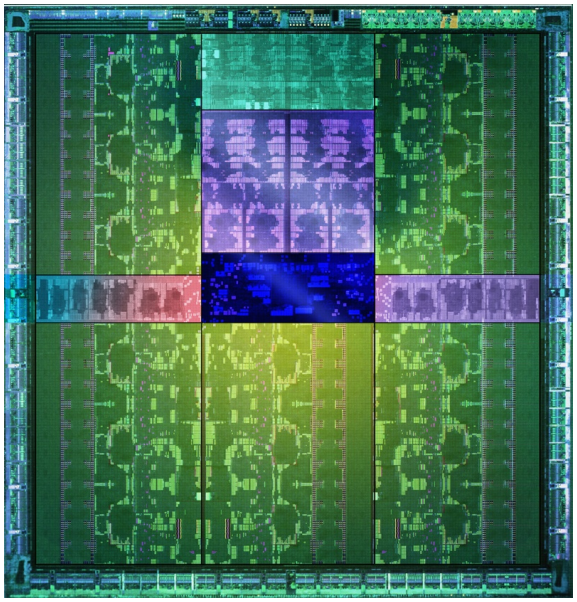




Beide Bücher liegen in der SLUB in ausreichender Zahl zur Ausleihe bereit

GRAFIKKARTEN ALS VIELKERNPROZESSOREN

NVIDIA Kepler GK110 Chip



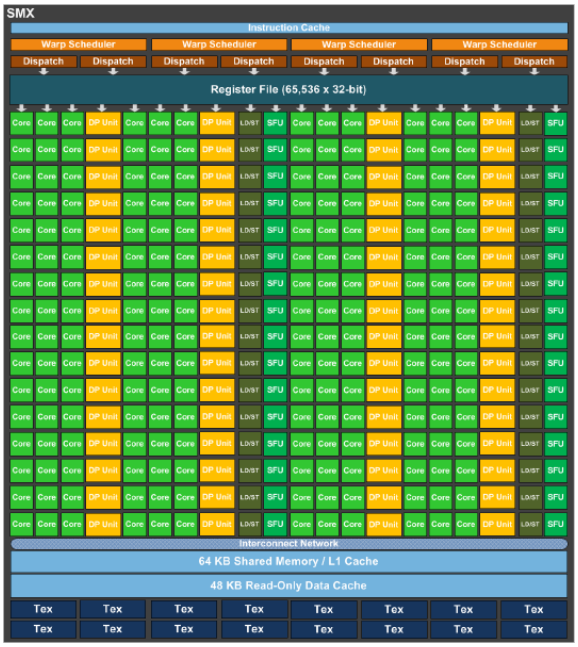
- 2880 CUDA Kerne
- 15 SMX Mehrkerneinheiten
- 384-bit Memory Controller
- Bis zu 24GB of GDDR5 Speicher
- Zweite Generation ECC
- Mindestens 1.5 TFLOPS DP FP64
- Ziel: 250 GB/s Bandbreite

vr-zone.com/articles/nvidia-s-monster-gpu-for-tesla-k20-2013-geforce-and-quadro-cards/15884.html

NVIDIA Kepler GK110 Aufbau

www.anandtech.com/show/5840/gtc-2012-part-1-nvidia-announces-gk104-based-tesla-k10-gk110-based-tesla-k20/2

NVIDIA Kepler GK110 Mehrkerneinheit

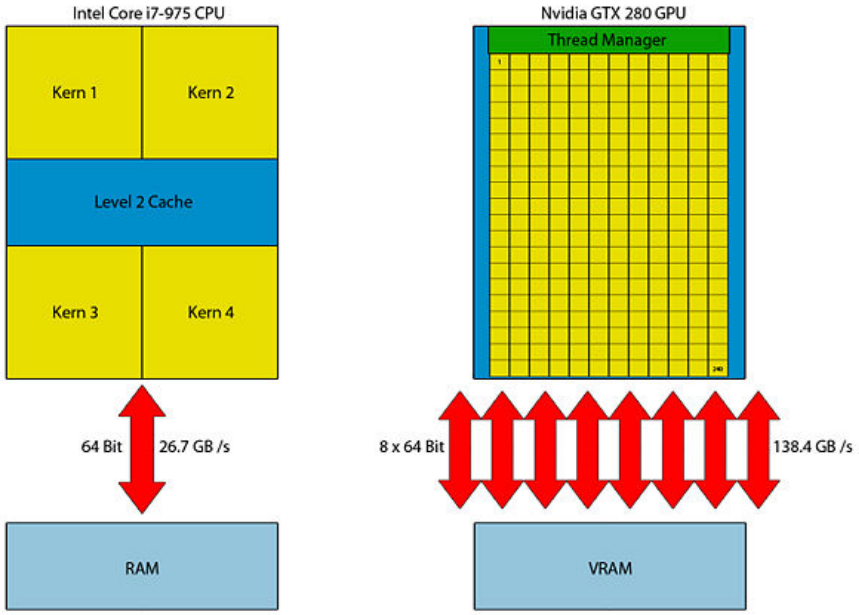


SMX

„Streaming Multiprocessor“

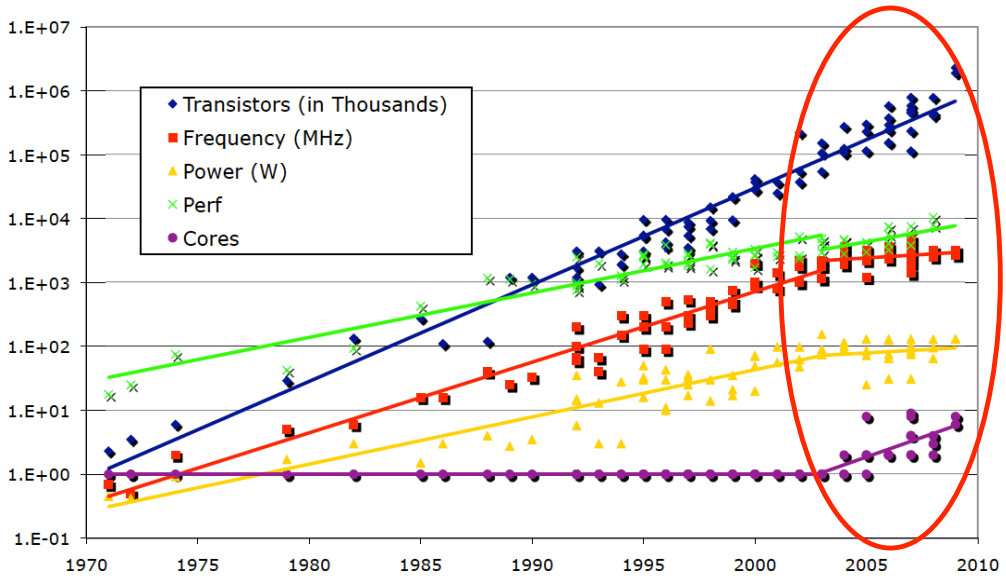
www.anandtech.com/show/5840/gtc-2012-part-1-nvidia-announces-gk104-based-tesla-k10-gk110-based-tesla-k20/2

Der kleine Unterschied



winfwiki.wi-fom.de/index.php/Bild:GPGPU_CPU_vs_GPU.jpg

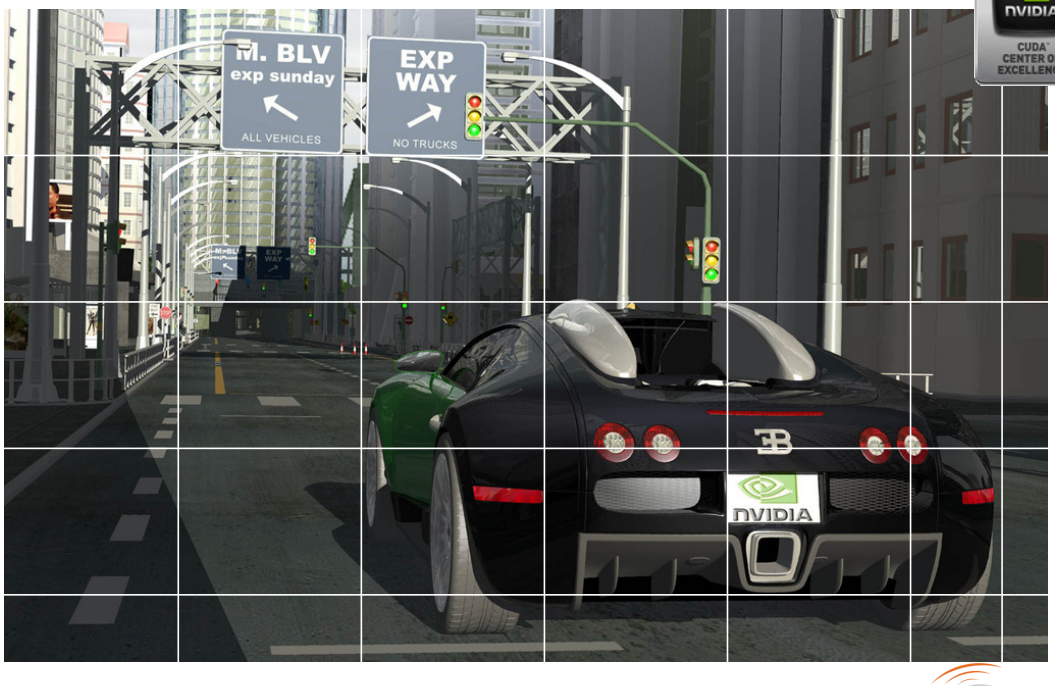
Nur mit mehr Rechenkernen gibt's mehr Leistung



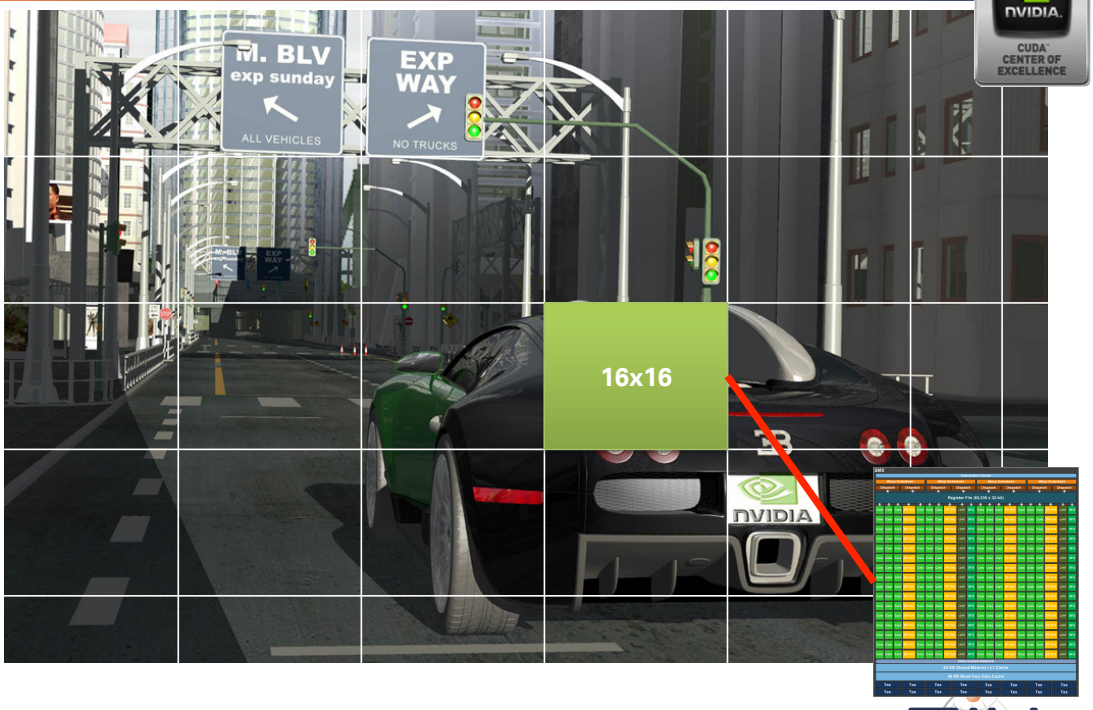
Kathy Yelick: „Ten Ways to Waste a Parallel Computer"
Keynote *ISCA 2009*. **The 36th International Symposium on Computer Architecture**
(mit Daten von Kunle Olukotun, Lance Hammond, Herb Sutter,Burton Smith, Chris Batten und Krste Asanovic)

GRAFIKKARTEN ZUM RECHNEN NUTZEN

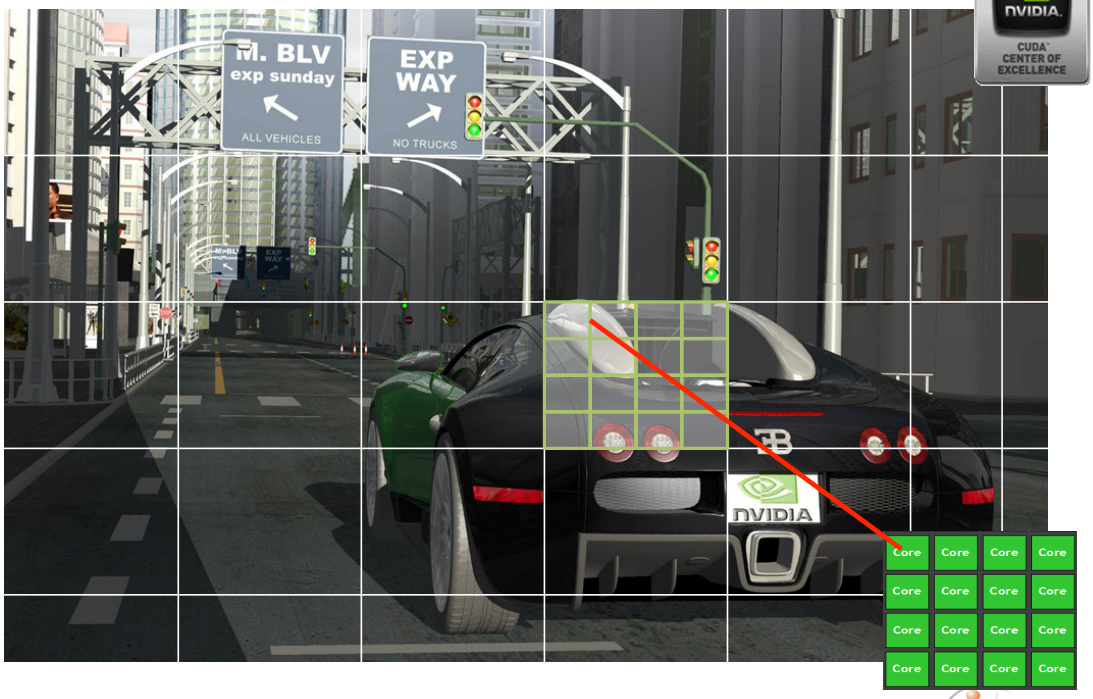
Zerlegung eines Bildes in Gitterblöcke



Block wird SMX zugewiesen



Untergitter (Pixel) werden Cores zugewiesen



Verschiedene Universen (Beispiele)

Spracherweiterungen

- CUDA
- OpenCL

Verschiedene Universen (Beispiele)

Bibliotheken
- cuBLAS
- cuFFT
- Thrust

Mathematikprogramme
- MATLAB
- Mathematica
- IDL

PRAGMA-Ansätze
- OpenACC
- OpenHMPP
- OpenMP

Grafik
- OpenGL
- DirectX

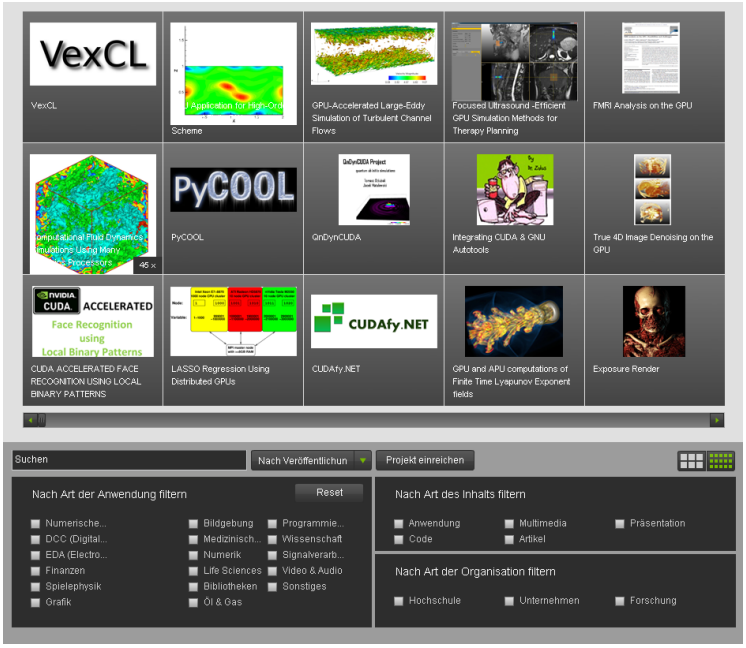
Spracherweiterungen
- CUDA
- OpenCL
- DirectCompute

Interpretersprachen
- Python: PyCUDA
- Perl: KappaCUDA

Prozessmodelierung
- LabView
- Simulink

Hochsprachen
- C/C++
- FORTRAN
- JAVA (RootBeer, JCUDA)
- Erlang

CUDA Community Showcase

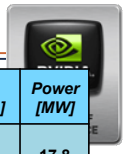


www.nvidia.de/object/cuda_apps_flash_de.html#state=home

Supercomputer TOP 10

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
2	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	16324.8	20132.7	7890
3	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
4	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8162.4	10066.3	3945
5	Forschungszentrum Juelich (FZJ) Germany	JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393216	4141.2	5033.2	1970
6	Leibniz Rechenzentrum Germany	SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM	147456	2897.0	3185.1	3423
7	Texas Advanced Computing Center/Univ. of Texas United States	Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi Dell	204900	2660.3	3959.0	
8	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT	186368	2566.0	4701.0	4040
9	CINECA Italy	Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	163840	1725.5	2097.2	822
10	IBM Development Engineering United States	DARPA Trial Subset - Power 775, POWER7 8C 3.836GHz, Custom Interconnect IBM	63360	1515.0	1944.4	3576

Top500 11/13 - TOP 10



Rank	*Installation Site*	*Manufacturer*	*Computer*	*Country*	*Cores*	*Rmax [Tflops]*	*Power [MW]*
1	National Super Computer Center in Guangzhou	NUDT	Tianhe-2 (MilkyWay-2) TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, Intel Xeon Phi	China	3120000	33863	17.8
2	DOE/SC/Oak Ridge National Laboratory	Cray Inc.	Titan Cray XK7 , Opteron 6274 16C 2.200GHz, NVIDIA K20x	USA	560640	17590	8.2
3	DOE/NNSA/LLNL	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.60 GHz	USA	1572864	17173	7.9
4	RIKEN Advanced Institute for Computational Science (AICS)	Fujitsu	K computer SPARC64 VIIIfx 2.0GHz	Japan	705024	10510	12.7
5	DOE/SC/Argonne National Laboratory	IBM	Mira BlueGene/Q, Power BQC 16C 1.60GHz	USA	786432	8587	3.9
6	Swiss National Supercomputing Centre (CSCS)	Cray Inc.	Piz Daint Cray XC30, Xeon E5-2670 8C 2.600GHz, NVIDIA K20x	Switzerland	115984	6271	2.3
7	Texas Advanced Computing Center/Univ. of Texas	Dell	Stampede PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Intel Xeon Phi	USA	462462	5168	4.5
8	Forschungszentrum Juelich (FZJ)	IBM	JUQUEEN BlueGene/Q, Power BQC 16C 1.600GHz	Germany	393216	5009	2.3
9	DOE/NNSA/LLNL	IBM	Vulcan BlueGene/Q, Power BQC 16C 1.600GHz	USA	393216	4293	2.0
10	Leibniz Rechenzentrum	IBM	SuperMUC iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz	Germany	147456	2897	3.4

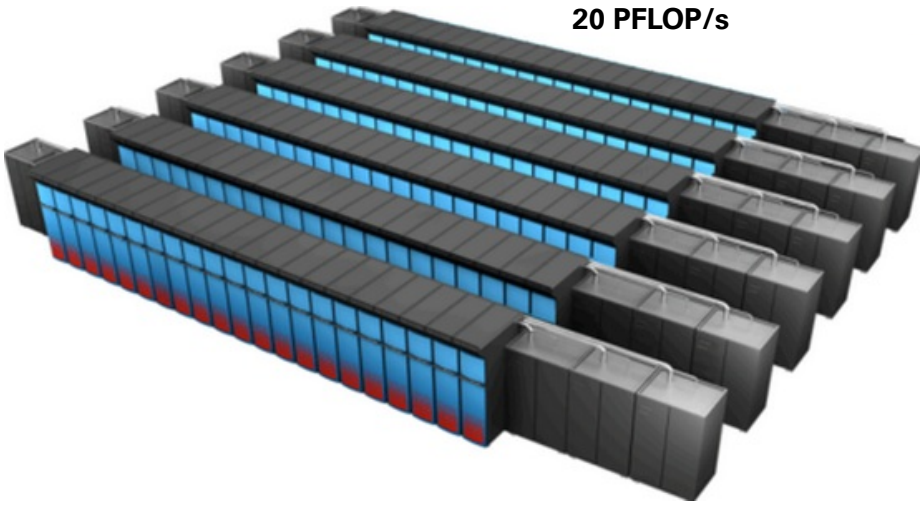
Green500 11/13 - TOP 10

Rank	Installation Site	Manu-facturer	Computer	Country	Cores	Rmax [Gflops]	MFLOPS/W	Top500 rank
1	GSIC Center, Tokyo Institute of Technology	NEC	TSUBAME-KFC LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C, 2.100GHz, NVIDIA K20x	Japan	2720	125	4503	311
2	Cambridge University	DELL	Wilkes Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, NVIDIA K20	UK	5120	191	3632	166
3	Center for Computational Sciences, University of Tsukuba	Cray	HA-PACS TCA Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, NVIDIA K20x	Japan	4864	277	3518	134
4	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC30, Xeon E5-2670 8C 2.600GHz, NVIDIA K20x	Switzerland	115984	5587	3186	6
5	ROMEO HPC Center - Champagne-Ardenne	Bull SA	romeo Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, NVIDIA K20x	France	5720	255	3131	151
6	GSIC Center, Tokyo Institute of Technology	NEC/HP	TSUBAME 2.5 Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, NVIDIA K20x	Japan	74358	2831	3069	11
7	University of Arizona	IBM	iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, NVIDIA K20x	USA	3080	145	2702	336
8	Max-Planck-Gesellschaft MPI/IPP	IBM	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, NVIDIA K20x	Germany	15840	710	2629	49
9	Financial Institution	IBM	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, NVIDIA K20x	USA	3264	146	2629	328
10	CSIRO	Xenon Systems	CSIRO GPU Cluster Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Nvidia K20m	Australia	4620	168	2359	260

TITAN at Oak Ridge National Lab

18.000 KEPLER GPUs

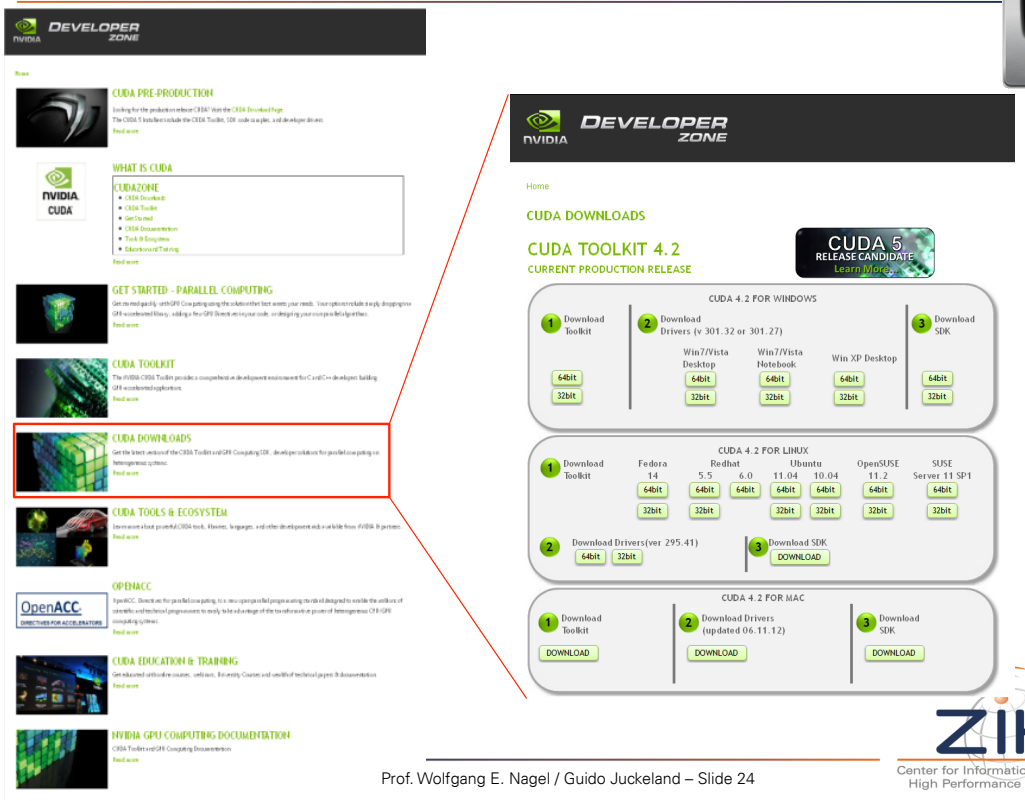
20 PFLOP/s



perilsofparallel.blogspot.de/2012/01/20-pflops-vs-10s-of-mloc-oak-ridge.html

ERSTE SCHRITTE
CUDA INSTALLIEREN

developer.nvidia.com/category/zone/cuda-zone



Getting Started (Guides + Beispiele)

developer.nvidia.com/cuda/nvidia-gpu-computing-documentation

NVIDIA DEVELOPER ZONE

Home

NVIDIA GPU COMPUTING DOCUMENTATION

Find all the latest documentation for the CUDA Toolkit, Libraries, Tools and SDK's below.

Individual SDK code samples can be found here.

Documentation of legacy versions of the CUDA Toolkit and SDKs are including in the installable legacy downloads available on the CUDA Toolkit Archive Page.

We recommend that all developers upgrade to the latest version , download the Latest Version of the CUDA Toolkit today.

CUDA Getting Started Guide (Windows) — Download

This guide will show you how to install and check the correct operation of the CUDA development tools in Windows.

CUDA Getting Started Guide (Linux) — Download

This guide will show you how to install and check the correct operation of the CUDA development tools in Linux.

CUDA Getting Started Guide (Mac OS X) — Download

This guide will show you how to install and check the correct operation of the CUDA development tools in Mac OS X.

Getting Started with CUDA SDK samples — Download

This guide covers the introductary CUDA SDK samples beginning CUDA developers should review before developing your own projects.

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- **Habe ich Zugang zu CUDA-fähiger Hardware?**

developer.nvidia.com/cuda/cuda-gpus

NVIDIA DEVELOPER ZONE

Home

CUDA GPUS

NVIDIA GPUs power millions of desktops, notebooks, workstations and supercomputers around the world, accelerating computationally-intensive tasks for consumers, professionals, scientists, and researchers.

Find out all about CUDA and GPU Computing by attending our GPU Computing Webinars and joining our free-to-join CUDA Registered developer Program.

- Learn about Tesla for technical and scientific computing
- Learn about Quadro for professional visualization

GeForce Desktop Products

GPU	Compute Capability
GeForce GTX 690	3.0
GeForce GTX 680	3.0
GeForce GTX 670	3.0
GeForce GTX 660 Ti	3.0
GeForce GTX 660	3.0
GeForce GTX 650	3.0
GeForce GTX 560 Ti	2.1
GeForce GTX 550 Ti	2.1
GeForce GTX 460	2.1
GeForce GTS 450	2.1
GeForce GTS 450*	2.1
GeForce GTX 590	2.0
GeForce GTX 580	2.0
GeForce GTX 570	2.0

CUDA-Enabled Tesla GPU Computing Products

Tesla Workstation Products		Tesla Data Center Products	
GPU	Compute Capability	GPU	Compute Capability
Tesla C2075	2.0	Tesla K20	3.5
Tesla C2050/C2070	2.0	Tesla K10	3.0
Tesla C1060	1.3	Tesla M2050/M2070/M2075/M2090	2.0
Tesla C870	1.0	Tesla S1070	1.3
Tesla D870	1.0	Tesla M1060	1.3
		Tesla S870	1.0

CUDA-Enabled Quadro Products

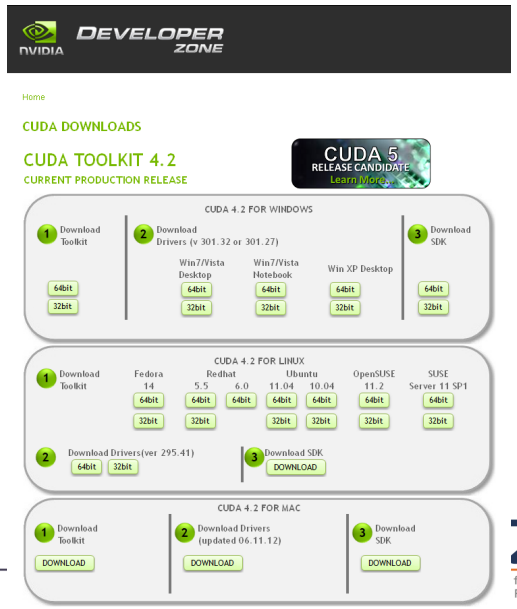
Quadro Desktop Products		Quadro Mobile Products	
GPU	Compute Capability	GPU	Compute Capability
Quadro K5000	3.0	Quadro K500M	3.0
Quadro 6000	2.0	Quadro 5010M	2.0
Quadro 5000	2.0	Quadro 5000M	2.0
Quadro 4000	2.0	Quadro 4000M	2.1
Quadro 4000 for Mac	2.0	Quadro 3000M	2.1

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen

developer.nvidia.com/cuda/cuda-downloads



TECHNISCHE UNIVERSITÄT DRESDEN

ZIH for Information Services & Performance Computing

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen

- Prüfen, ob das System CUDA-Grafikkartentreiber unterstützt (vor allem bei LINUX wichtig!)

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen
- Prüfen, ob das System CUDA-Grafikkartentreiber unterstützt
  (vor allem bei LINUX wichtig!)
- Prüfen, ob ein Compiler installiert ist, der CUDA unterstützt
  (LINUX: gcc, Windows: MS Visual Studio mit C++, Mac OS X: gcc)

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen
- Prüfen, ob das System CUDA-Grafikkartentreiber unterstützt
  (vor allem bei LINUX wichtig!)
- Prüfen, ob ein Compiler installiert ist, der CUDA unterstützt
  (LINUX: gcc, Windows: MS Visual Studio mit C++, Mac OS X: gcc)
- CUDA-fähigen Grafikkartentreiber installieren

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen
- Prüfen, ob das System CUDA-Grafikkartentreiber unterstützt
  (vor allem bei LINUX wichtig!)
- Prüfen, ob ein Compiler installiert ist, der CUDA unterstützt
  (LINUX: gcc, Windows: MS Visual Studio mit C++, Mac OS X: gcc)
- CUDA-fähigen Grafikkartentreiber installieren
- CUDA Toolkit herunterladen und installieren

Schrittweise zur erfolgreichen CUDA-Installation

Installationsschritte

- Habe ich Zugang zu CUDA-fähiger Hardware?
- Grafikkartentreiber mit CUDA-Fähigkeiten herunterladen
- Prüfen, ob das System CUDA-Grafikkartentreiber unterstützt
  (vor allem bei LINUX wichtig!)
- Prüfen, ob ein Compiler installiert ist, der CUDA unterstützt
  (LINUX: gcc, Windows: MS Visual Studio mit C++, Mac OS X: gcc)
- CUDA-fähigen Grafikkartentreiber installieren
- CUDA Toolkit herunterladen und installieren
- Zusätzlich kann man den GPU Computing SDK herunterladen und installieren und sich Beispielprogramme anschauen

ERSTE SCHRITTE CUDA NUTZEN

Mein erstes CUDA-Programm!

vec2zero.cu

```
#define NELEMENTS 16;              //16 Elemente
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS];           //Deklaration Vektor veca
  int i;                           //Laufvariable für Schleife

  for ( i=0; i<NELEMENTS; i++ ) {  //Schleife über Elemente
    veca[i] = 0.0f;                //Zuweisung der Elemente
  }

  return 0;
}
```

Compilieren (LINUX)


```
bussmann@gpunode> nvcc vec2zero.cu –o vec2zero
bussmann@gpunode> ./vec2zero
```

Das funktioniert nur bei korrekter Softwareinstallation und wenn der Code auf einem CUDA-fähigen Device ausgeführt werden kann.

Ohne GPU erhält man den Fehler:

```
NVIDIA: could not open the device file /dev/nvidiactl (No such device or address).
```


Die GPU wirklich nutzen: Einen „Kernel“ schreiben


```
#define NELEMENTS 16;
__global__ void kernel( void ) {    //ein Kernel, der nix macht
}
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS];
  int i;

  for ( i=0; i<NELEMENTS; i++ ) {
    veca[i] = 0.0f;
  }

  return 0;
}
```

Die GPU wirklich nutzen: Einen „Kernel“ aufrufen


```
#define NELEMENTS 16;
__global__ void kernel( void ) {    //ein Kernel, der nix macht
}
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS];
  int i;

  kernel<<<1,1>>>();                //Tut nix, will nur spielen
  for ( i=0; i<NELEMENTS; i++ ) {
    veca[i] = 0.0f;
  }

  return 0;
}
```


Zwischenspiel: HOST und DEVICE

„HOST“

„DEVICE“

Speicher auf GPU reservieren und freigeben

```
int main( int argc, char *argv[] ) {

  float *devicepointer;                          //Zeiger auf float

  //Die folgende Zeile reserviert Speicher für 1 float auf der GPU
  //und setzt den Pointer veca_elem_dev auf diesen Speicherbereich
  cudaMalloc( (void**)&devicepointer, sizeof(float) );
  //Die folgende Zeile gibt den GPU-Speicher, auf den veca_elem_dev
  //zeigt auf der GPU frei.
  cudaFree( devicepointer );

  return 0;
}
```


Datentransfer: cudaMemcpy

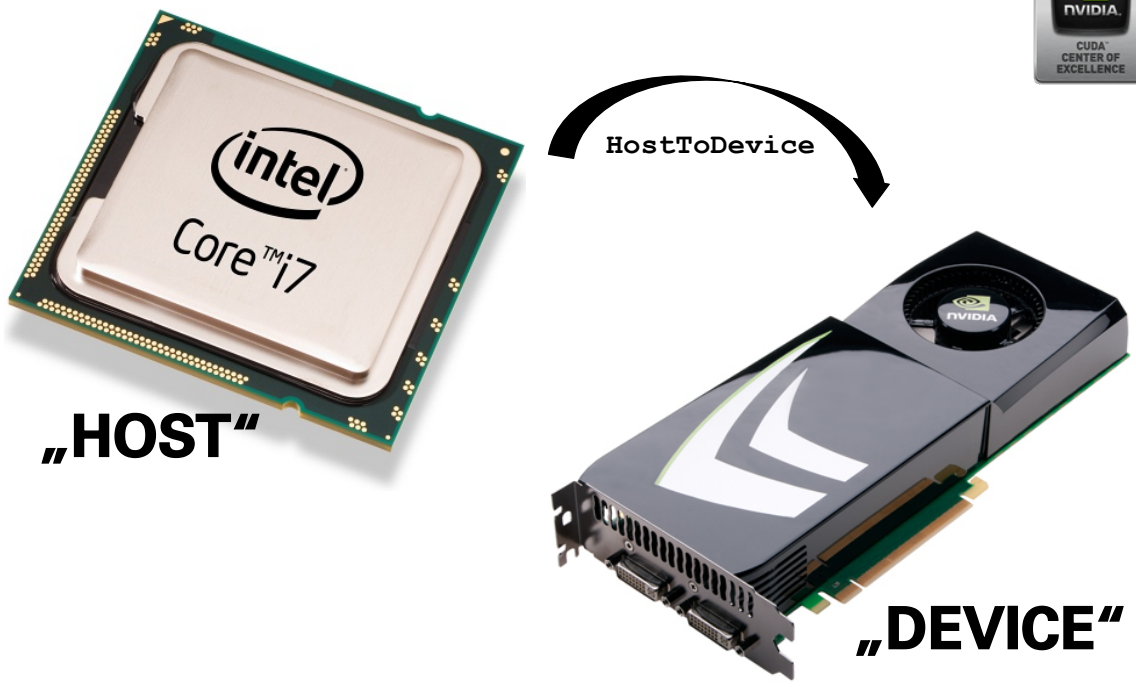
„HOST“

„DEVICE“

Speicher von DEVICE nach HOST kopieren

```
#define NELEMENTS 16;
int main( int argc, char *argv[] ) {

  float hostvariable;      //float-Variable auf HOST
  float *devicepointer;    //Zeiger auf Speicher auf DEVICE

  cudaMemcpy( &hostvariable,          //Zeiger auf HOST-Variable
              devicepointer,          //Zeiger auf DEVICE-Speicher
              sizeof(float),          //Größe des Speichers: 1 float
              cudaMemcpyDeviceToHost );
  return 0;
}
```


Datentransfer: cudaMemcpy



Speicher von HOST nach DEVICE kopieren


```
#define NELEMENTS 16;
int main( int argc, char *argv[] ) {

  float hostvariable;      //float-Variable auf HOST
  float *devicepointer;    //Zeiger auf Speicher auf DEVICE

  cudaMemcpy( devicepointer,          //Zeiger auf DEVICE-Speicher
              &hostvariable,          //Zeiger auf HOST-Variable
              sizeof(float),          //Größe des Speichers: 1 float
              cudaMemcpyHostToDevice );
  return 0;
}
```


Ein kleiner Blick in die Familie cudaMemcpy


```
cudaMemcpy
cudaMemcpy2D
cudaMemcpy2DArrayToArray
cudaMemcpy2DAsync
cudaMemcpy2DFromArray
cudaMemcpy2DFromArrayAsync
cudaMemcpy2DToArray
cudaMemcpy2DToArrayAsync
cudaMemcpy3D
cudaMemcpy3DAsync
cudaMemcpy3DPeer
cudaMemcpy3DPeerAsync
cudaMemcpyArrayToArray
cudaMemcpyAsync
cudaMemcpyFromArray
cudaMemcpyFromArrayAsync
cudaMemcpyFromSymbol
cudaMemcpyFromSymbolAsync
cudaMemcpyPeer
cudaMemcpyPeerAsync
cudaMemcpyToArray
cudaMemcpyToArrayAsync
cudaMemcpyToSymbol
cudaMemcpyToSymbolAsync
```

Fehler abfangen mit cudaError_t

CUDA-Funktionen geben einen Wert vom Typ cudaError_t zurück:

```
typedef enum cudaError cudaError_t
```

Dieser kann verschiedene Werte annehmen:

`cudaSuccess`	**The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see `cudaEventQuery()` and `cudaStreamQuery()`).**
`cudaErrorMissingConfiguration`	**The device function being invoked (usually via `cudaLaunch()`) was not previously configured via the `cudaConfigureCall()` function.**
`cudaErrorMemoryAllocation`	**The API call failed because it was unable to allocate enough memory to perform the requested operation.**
`cudaErrorInitializationError`	**The API call failed because the CUDA driver and runtime could not be initialized.**
`cudaErrorLaunchFailure`	**An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until `cudaThreadExit()` is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.**

...

Den Fehler interpretieren

Zur Interpretation von cudaError_t kann man die Funktion

```
const char* cudaGetErrorString( cudaError_t error)
```

nutzen:

```
cudaError_t error = cudaGetLastError();
printf( "CUDA error: %s\n", cudaGetErrorString( error ) );
```

Fehler abfangen mit cutilXXXX

Der CUDA-SDK bietet einige Funktionen zum Fehlerabfangen

```
cutilSafeCall( ... );       //prüft Funktionsrückgabe
                            //für CUDA-Funktionen
cutilSafeCall( cudaMalloc(...) );

cutilCheckMsg( ... );       //prüft kernel-Aufrufe
kernel<<<1,1>>>();
cutilCheckMsg("kernel failed\n");

cutilCheckError( ... );     //prüft SDK-Aufrufe
```


Fehler abfangen mit HandleError (CUDA by Example)


```
static void HandleError( cudaError_t err, const char *file, int line ) {

  if (err != cudaSuccess) {
    printf( "%s in %s at line %d\n", cudaGetErrorString( err ),  file, line );
    exit( EXIT_FAILURE );
  }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))
```

Ein richtiges CUDA-Programm


```
#define NELEMENTS 16;
__global__ void settozero( float *elem ) {
  *elem = 0.0f;
}
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS];
  float *devicemem;
  int i;

  cudaMalloc( (void**)&devicemem, sizeof(float) );
  for ( i=0; i<NELEMENTS; i++ ) {
    settozero<<<1,1>>>( devicemem );
    cudaMemcpy( &veca[i], devicemem, sizeof(float), cudaMemcpyDeviceToHost );
  }

  cudaFree( devicemem );
  return 0;
}
```


ZUSAMMENFASSUNG

Kernel und Speicherverwaltung

Kernel werden auf der GPU (dem Device) aufgerufen. Sie können nur auf Speicher auf der GPU zugreifen. Sie verhalten sich wie C-Funktionen.

```
__global__ returntype kernelname( paramdec1, paramdec2, …) { … }
kernelname<<<dim3 grid, dim3 block>>>( param1, param2, …);
```

Speicherverwaltung:

```
cudaError_t cudaMalloc( void **devPtr, size_t size )
cudaError_t cudaFree( void *devPtr)
cudaError_t cudaMemcpy ( void *destination,              //Ziel
                         const void *source,             //Quelle
                         size_t count,                   //Größe
                         enum cudaMemcpyKind direction ) //Richtung
enum cudaMemcpyKind: cudaMemcpyHostToDevice,
                     cudaMemcpyDeviceToHost, …
```


Fehlerbehandlung

Fehler haben den Typ

```
cudaError_t
```

Sie werden entweder direkt von einer Funktion zurückgegeben oder können mit Hilfe der Funktion

```
cudaError_t cudaGetLastError( void )
```

bestimmt werden.

Der zugehörige Wert kann mit Hilfe der Funktion

```
const char* cudaGetErrorString( cudaError_t error )
```

in einen Fehler-String konvertiert werden.

KERNEL UND THREADS

Kernel-Aufrufe

Ein *Kernel* ist ein Stück Code, das für die Ausführung auf der GPU übersetzt wird. Kernel können auf allen CUDA-fähigen GPUs einzeln und hintereinander vom Host aufgerufen werden:

```
int main( int argc, char *argv[] ) {


   kernel1<<<…,…>>>(…);
   kernel2<<<…,…>>>(…);
   kernel3<<<…,…>>>(…);
   return 0;
}
```

Dabei wird gewartet, bis jeder Kernelaufruf abgearbeitet wurde, bevor der nächste Kernel aufgerufen wird. Die Abfolge der Kernelaufrufe bleibt somit strikt seriell.

Wie kann ich dann aber parallel arbeiten?

CUDA ermöglicht die parallele Ausführung von Kernel-Code. Dabei wird der Code eines Kernels parallel auf vielen Prozessoren gleichzeitig ausgeführt.

Jede Instanz eines solchen ausgeführten Kernels wird *Thread* genannt

```
int main( int argc, char *argv[] ) {

  kernel1<<<…,…>>>(…); //erzeugt viele, parallele Threads
  kernel2<<<…,…>>>(…); //erzeugt viele, parallele Threads
  kernel3<<<…,…>>>(…); //erzeugt viele, parallele Threads
  return 0;
}
```

Threads können, in Abhängigkeit der Daten, unterschiedliche Anweisungen ausführen. Um jedoch die maximale Leistung aus der GPU zu holen, sollte man immer im *SIMD-Modell* denken.

Exkurs: SIMD

SIMD steht für *S*ingle *I*nstruction, *M*ultiple *D*ata. Man kann dies so zusammenfassen, dass unabhängig von den Daten jeder Thread, d.h. jeder parallele Kernel-Aufruf, dieselben Instruktionen mit jeweils anderen Daten ausführt:

```
__global__ void settozero( float *elem ) {
  *elem = 0.0f;
}
…
  settozero<<<…,…>>>( devicemem );
```

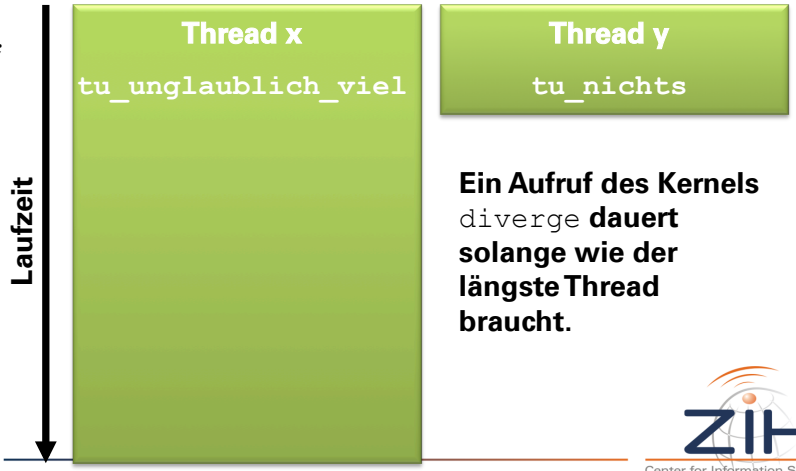
Thread 1 `*elem = 0.0f;`	**Thread 2** `*elem = 0.0f;`	**Thread 3** `*elem = 0.0f;`	**Thread 4** `*elem = 0.0f;`
Thread 5 `*elem = 0.0f;`	**Thread 6** `*elem = 0.0f;`	**Thread …** `*elem = 0.0f;`	**Thread ∞** `*elem = 0.0f;`

http://s09.idav.ucdavis.edu/talks/02_kayvonf_gpuArchTalk09.pdf

Exkurs: Thread-Divergenz

Werden verschiedene Teile des Kernel-Codes je nach Datenlage ausgeführt, so spricht man von Thread-Divergenz, da nun unterschiedliche Threads mit unterschiedlichen Daten unterschiedliche Dinge tun

```
__global__ void diverge( void *data ) {
  if ( data[mythread] > zufallszahl )
    tu_unglaublich_viel;
  else
    tu__nichts;
}
```



Ein Aufruf des Kernels `diverge` **dauert solange wie der längste Thread braucht.**

Exkurs: Kernel-Aufrufe

Ab Compute Capability 2.x gibt es die Möglichkeit, mehrere Kernel parallel auszuführen. Diese werden in so genannten *Streams* organisiert.

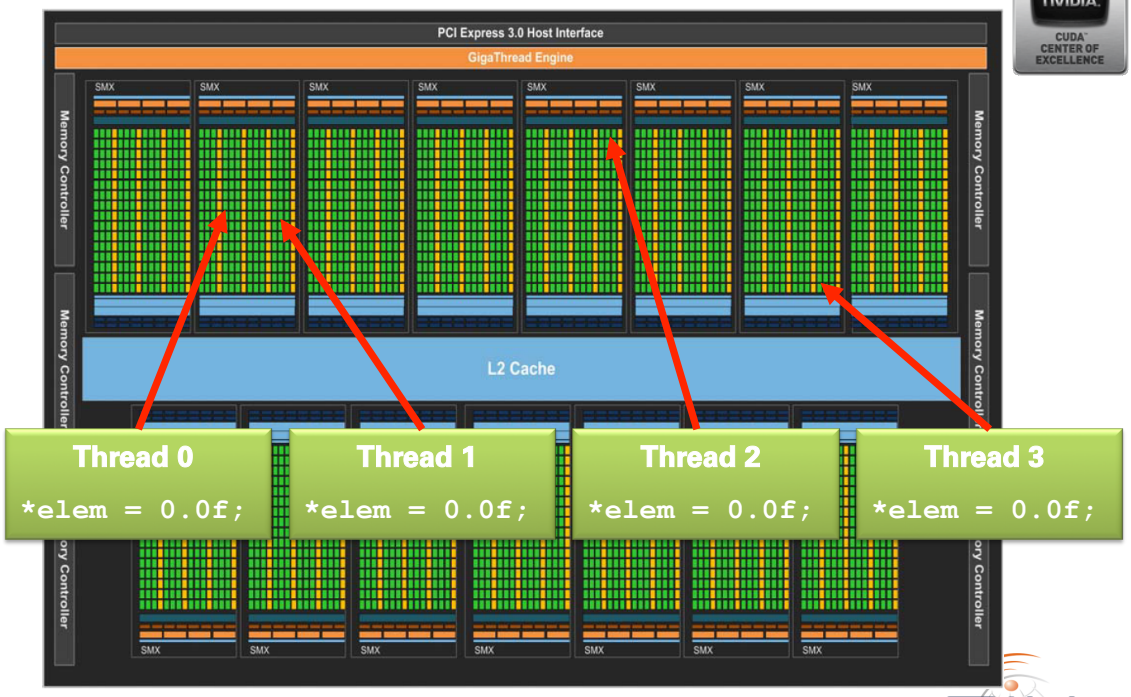
Diese Möglichkeit der Parallelisierung wird hier nicht besprochen, soll aber erwähnt werden.

Ab CUDA 5.0 und Kepler-Architektur gibt es zusätzlich die Möglichkeit, dass Kernel selbst mehrere Kernel ausführen. Sogar Rekursion ist möglich!

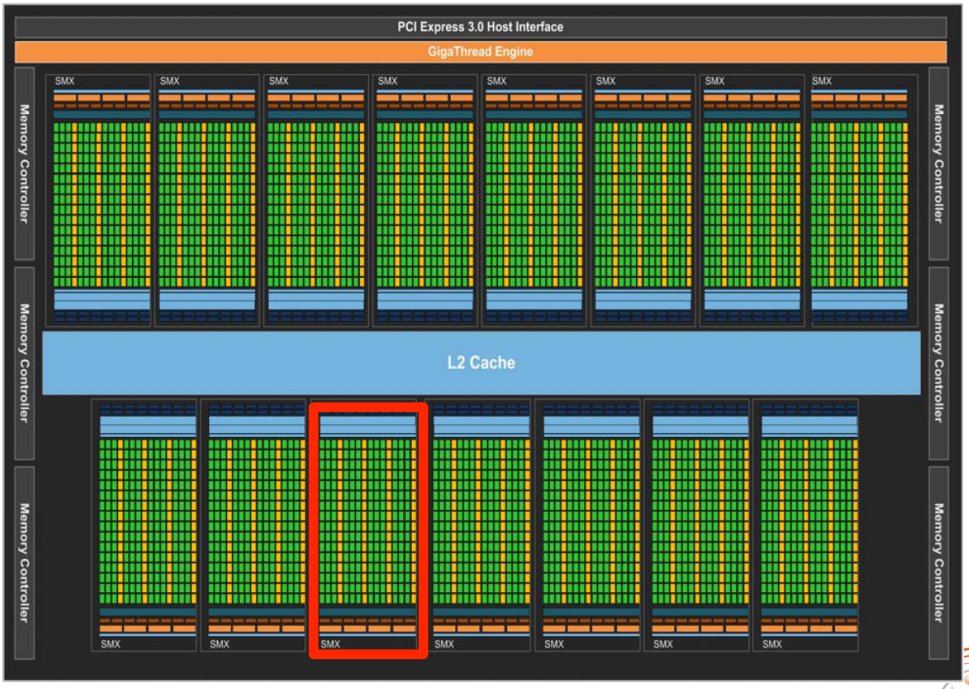
```
__global__ childkernel( void *data ) {…}

__global__ parentkernel( void *data ) {
  childkernel<<<…,…>>>( data );  //Dynamic Parallelism! YEAH!
}
…
parentkernel<<<…,…>>>( data );
```

EIN AUSFLUG IN DIE ARCHITEKTUR VON GPUS

Kepler GK110 Architektur



Kepler GK110 Architektur

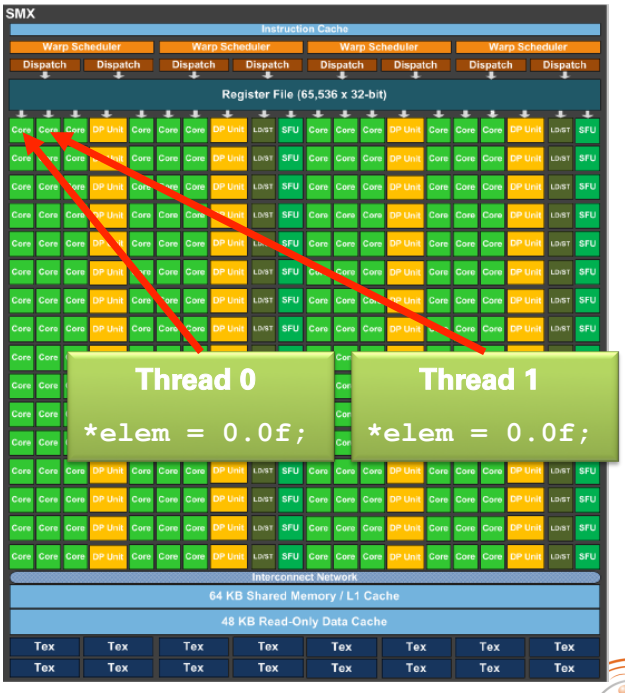


Streaming Multiprocessor

Threads, die parallel auf einem SMX ausgeführt werden sollen, sollten in so genannten *Blöcken* ausgeführt werden.

Sie können sich Speicher gemeinsam teilen und zusammen darauf zugreifen.

(hierzu später mehr)



WIE ERZEUGEN UND STEUERN WIR THREADS?

THREAD-BLOCKS

Thread-Blocks (Blöcke)

Ein *Thread-Block* ist eine Einheit von Threads, die gemeinsam auf einem Streaming Multiprocessor (SMX) parallel ausgeführt werden.

Die Reihenfolge, in der die Threads ausgeführt werden, nicht festgelegt, d.h. ein beliebiger Thread auf einem Core kann vor oder nach einem Thread auf einem anderen Core gestartet bzw. beendet werden.

Threads in einem Block können miteinander über gemeinsamen, schnellen Speicher (shared memory) Daten austauschen.

```
int nx; int ny; int nz; …
dim3 block(nx, ny, nz); //nx, ny, nz = Größe des Blocks in 3D
kernel<<<1,block>>>(…); //Ruft nx*ny*nz threads auf, die in
                        //einem gemeinsamen Block laufen
```


Die Struktur `dim3`


```
struct dim3
{
    unsigned int x, y, z;
};
```

Thread-Blocks und Anzahl der Threads

Die Anzahl der Threads in einem Thread-Block wird durch das Produkt der Dimensionen des Thread-Blocks bestimmt:

```
dim3 block512(16,8,4) //Anzahl Threads ist 16*8*4 = 512
```

Die Anzahl der Threads in einem Block ist begrenzt. Ebenso ist die Anzahl der Threads in den einzelnen Dimensionen begrenzt.

Compute Capability	1.0	1.1	1.2	1.3	2.x	3.0
Max. Blockgröße in x,y	512				1024	
Max. Blockgröße in z	64					
Max. Threads pro Block	512				1024	

Was kann meine Hardware?

Wie bereits erwähnt, erhält man mit

```
cudaError_t cudaGetDeviceProperties( struct cudaDeviceProp *prop,
                                     int    device );
```

Informationen zu seiner GPU. Diese enthalten auch die Informationen zu Thread-Blöcken:

```
struct cudaDeviceProp {
  …
  int maxThreadsPerBlock; //Maximale Zahl der Threads pro Block
  int maxThreadsDim[3];   //Maximale Größe eines Blocks in x,y,z
  …
}
```

GRIDS OF BLOCKS

Kann ich mehr als 1024 Threads ausführen?

Es ist möglich, deutlich mehr Threads auszuführen. Dazu werden mehrere Thread-Blöcke parallel gestartet.

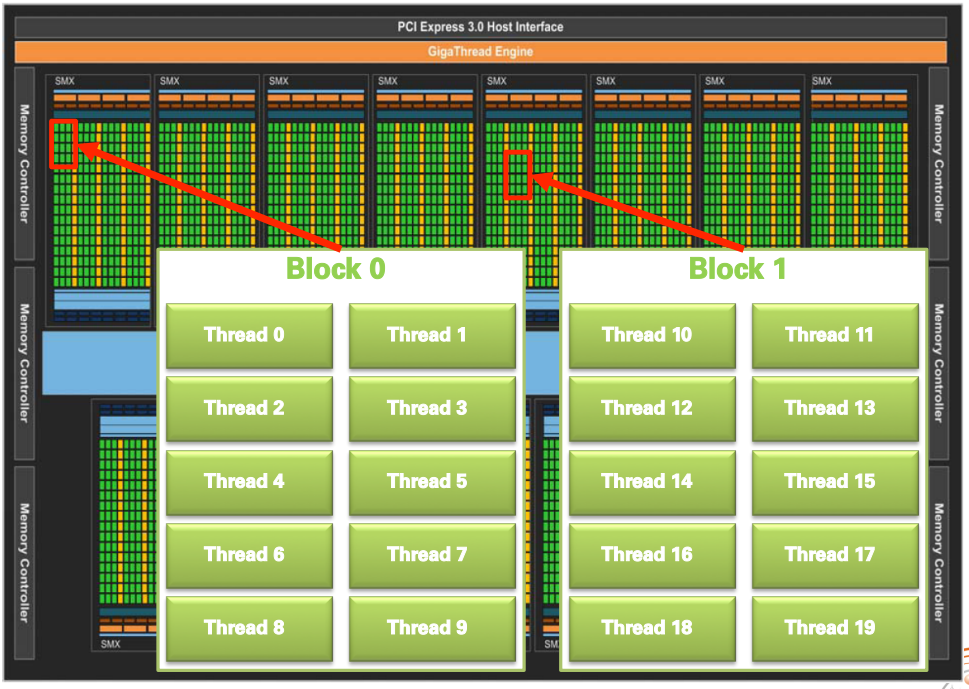
Hierzu erzeugt man Gitter (Grids) von Blöcken (Blocks).

Threads in unterschiedlichen Blöcken besitzen keinen gemeinsamen (shared) schnellen Speicher, über den sie Daten austauschen können.

Wie bei Blöcken ist die Abarbeitungsreihenfolge zufällig. Jeder Block im Gitter kann auf einem beliebigen (freien) Multiprozessor gestartet werden.

```
int nx; int ny; int nz; …
dim3 grid(nx, ny, nz); //nx, ny, nz = Größe des Grids in 3D
kernel<<<grid,1>>>(…); //Ruft nx*ny*nz threads auf, jeden in
                       //einem eigenen, getrennt laufenden Block
```

Verteilung von Gittern



Gitter und Anzahl der Thread-Blöcke

Die Anzahl der Blöcke in einem Gitter wird durch das Produkt der Dimensionen des Gitters bestimmt:

```
dim3 grid512(16,8,4) //Anzahl Blöcke ist 16*8*4 = 512, CC ab 2.x
```

Die Anzahl der Blöcke in einem Gitter ist begrenzt. Ebenso ist die Anzahl der Blöcke in den einzelnen Dimensionen begrenzt.

Compute Capability	1.0	1.1	1.2	1.3	2.x	3.0
Dimension des Grids	2D (grid(nx,ny,1))				3D (grid(nx,ny,nz))	
Max. Gridgröße in x,y,z	65535					2^{31}-1
Max. Gridgröße	65535					2^{31}-1

Was kann meine Hardware?

Wiederum erhält man mit

```
cudaError_t cudaGetDeviceProperties( struct cudaDeviceProp *prop,
                                     int    device );
```

Informationen zu seiner GPU. Diese enthalten Daten zu Gittern:

```
struct cudaDeviceProp {
  …
  int maxGridSize[3]; //Maximale Größe eines Gitters in x,y,z
  …
}
```


Zusammenfassung Gitter und Blöcke

Bei einem Kernelaufruf

```
dim3 block(bnx,bny,bnz);
dim3 grid(gnx,gny,gnz);
kernel<<<grid,block>>>(…);
```

werden `bnx*bny*bnz` **Threads in** `gnx*gny*gnz` **Blöcken gestartet, d.h. es werden insgesamt** `bnx*bny*bnz*gnx*gny*gnz` **Threads aufgerufen (aber oft nicht gleichzeitig ausgeführt!).**

Man kann einen Kernelaufruf daher auch verstehen als

```
kernel<<<Anzahl unabhängiger Thread-Blöcke,Threads pro Block>>>
```

THREADS IM BLOCK STARTEN

Threads in einem Block aufrufen

```
dim3 block1(8, 1, 1);
kernel<<<1,block1>>>(…); //8 Threads in einem Block, alle in x
```

oder auch einfach:

```
kernel<<<1,8>>>(…);      //8 Threads in einem Block
```

Threads in einem Block werden linear durchnummeriert.

Jeder Thread erhält eine eindeutige ID:

```
dim3 threadIdx
```

Ein Beispiel: Paralleles `settozero` im Block


```
#define NELEMENTS 512 //maximale Anzahl Threads pro Block
__global__ void settozero( float *elem ) {

  int tid   = threadIdx.x; //Wir haben einen Vektor, d.h. eine
                           //eindimensionale Liste!
  elem[tid] = 0.0f;        //Jeder Thread setzt ein Element auf 0
}
```

Achtung! In diesem Beispiel nehmen wir an, dass unser Thread-Block mit

`dim3 block1d(NELEMENTS,1,1)`

`settozero<<<1,block1d>>>(…)`

aufgerufen wird. Dies ist equivalent zum Aufruf

`settozero<<<1,NELEMENTS>>>(…)`

Ein Beispiel: Paralleles `settozero` im Block

Um den gesamten Vektor parallel bearbeiten zu können, müssen wir Speicherplatz für den gesamten Vektor im GPU-Speicher reservieren

```
#define NELEMENTS 512   //maximale Anzahl Threads pro Block
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS]; //Vektor im Host-Speicher
  float *devicemem;      //Pointer auf Vektor im GPU-Speicher

  //Jetzt reserviere Speicherplatz für den gesamten Vektor
  //auf der GPU, d.h. für NELEMENTS float-Variablen.
  cudaMalloc( (void**)&devicemem, NELEMENTS*sizeof(float) );
  …
  cudaFree(devicemem);
  …
}
```

Ein Beispiel: Paralleles `settozero` im Block

Nun müssen wir den Thread-Block definieren und aufrufen

```
#define NELEMENTS 512 //maximale Anzahl Threads pro Block
int main( int argc, char *argv[] ) {

  …
  dim3  block1d(NELEMENTS,1,1); //1D-Block aus NELEMENTS Threads
  …
  settozero<<<1,block1d>>>( devicemem ); //übergibt ein Array
  …
}
```


Paralleles `settozero` im Block im Ganzen


```
#define NELEMENTS 512
__global__ void settozero( float *elem ) {
  int tid   = threadIdx.x;
  elem[tid] = 0.0f;
}
int main( int argc, char *argv[] ) {

  float veca[NELEMENTS];
  float *devicemem;
  dim3  block1d(NELEMENTS,1,1);

  cudaMalloc( (void**)&devicemem, NELEMENTS*sizeof(float) );
  settozero<<<1,block1d>>>( devicemem );
  cudaMemcpy( veca, devicemem, NELEMENTS*sizeof(float), cudaMemcpyDeviceToHost );
  cudaFree( devicemem );

  return 0;
}
```

Threads in einem Block aufrufen


```
dim3 block1(8, 1, 1);
kernel<<<1,block1>>>(…); //8 Threads in einem Block, alle in x
```

oder auch einfach:

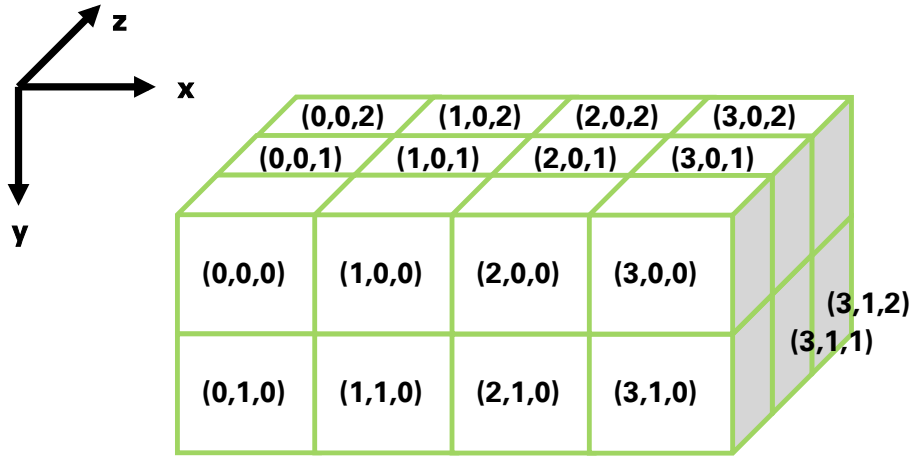
```
kernel<<<1,8>>>(…);      //8 Threads in einem Block
```

Andere Varianten:

```
dim3 block2(1, 8, 1);
kernel<<<1,block2>>>(…); //8 Threads in einem Block, alle in y
dim3 block3(1, 1, 8);
kernel<<<1,block3>>>(…); //8 Threads in einem Block, alle in z
dim3 block4(2, 1, 4);
kernel<<<1,block4>>>(…); //8 Threads, 2 in x, 1 in y, 4 in z
dim3 block5(2, 2, 2);
kernel<<<1,block5>>>(…); //8 Threads, je 2 in x, y und z
```


Verteilung der Threads über den Block


```
dim3 block(4,2,3);
kernel<<<1,block>>>(…);
```



New Threads on the Block


```
__global__ void kernel( void *data ) {
  int tidx = threadIdx.x; //Position des Threads im Block in x
  int tidy = threadIdx.y; //Position des Threads im Block in y
  int tidz = threadIdx.z; //Position des Threads im Block in z
}
```

```
dim3 block(4,2,3);
kernel<<<1,block>>>(data);
```

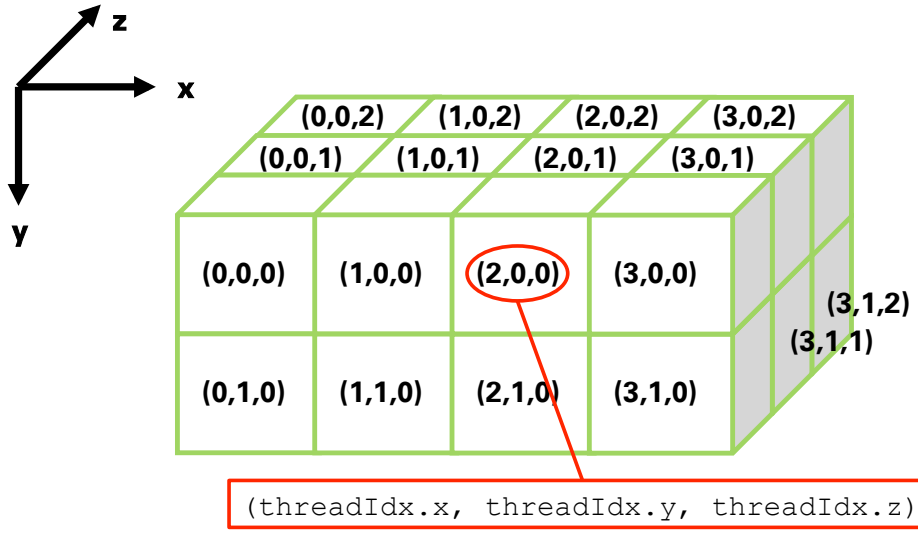
ruft dann 24 = 4*2*3 Kernel auf mit

(threadIdx.x, threadIdx.y, threadIdx.z):

(0,0,0),(1,0,0),(2,0,0),(3,0,0),(0,1,0),(1,1,0),(2,1,0),(3,1,0),
(0,0,1),(1,0,1),(2,0,1),(3,0,1),(0,1,1),(1,1,1),(2,1,1),(3,1,1),
(0,0,2),(1,0,2),(2,0,2),(3,0,2),(0,1,2),(1,1,2),(2,1,2),(3,1,2)

Verteilung der Threads über den Block


```
dim3 block(4,2,3);
kernel<<<1,block>>>(…);
```



Noch ein Beispiel: Matrix auf Null

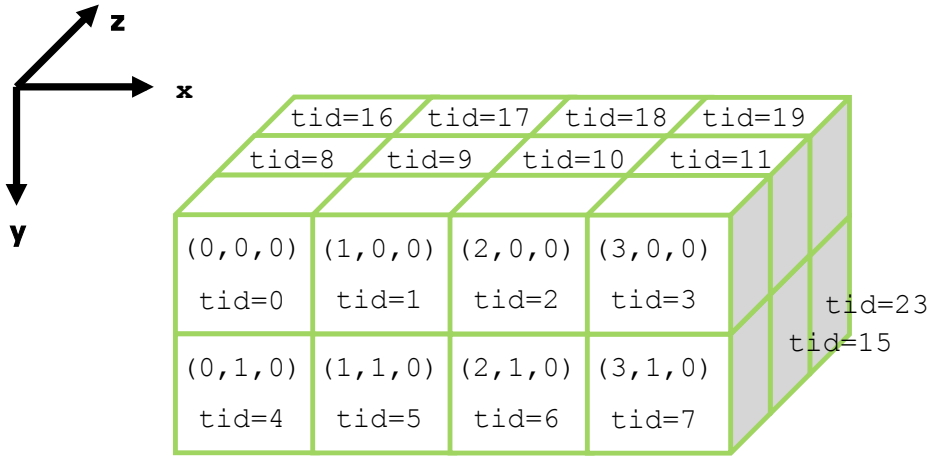

```
#define DIMMAT 16
__global__ void settozero( float *elem ) {
  int tidx         = threadIdx.x;
  int tidy         = threadIdx.y;
  elem[tidx][tidy] = 0.0f;
}
int main( int argc, char *argv[] ) {

  float matrixa[DIMMAT][DIMMAT];
  float *devicemem;
  dim3  block2d(DIMMAT,DIMMAT,1);

  cudaMalloc( (void**)&devicemem, DIMMAT*DIMMAT*sizeof(float) );
  settozero<<<1,block2d>>>( devicemem );
  cudaMemcpy( veca, devicemem, DIMMAT*DIMMAT*sizeof(float), cudaMemcpyDeviceToHost );
  cudaFree( devicemem );
  return 0;
}
```


Alles auf Null — Linearisieren der Thread-ID


```
#define DIMX 4
#define DIMY 2
#define DIMZ 3
__global__ void settozero( float *elem ) {
  int tid = threadIdx.x + threadIdx.y * DIMX + threadIdx.z * DIMX * DIMY;
  elem[tid] = 0.0f;
}
int main( int argc, char *argv[] ) {

  float 3dstuff[DIMX*DIMY*DIMZ];
  float *devicemem;
  dim3  block3d(DIMX,DIMY,DIMZ);

  cudaMalloc( (void**)&devicemem, DIMX*DIMY*DIMZ*sizeof(float) );
  settozero<<<1,block3d>>>( devicemem );
  cudaMemcpy( veca, devicemem, DIMX*DIMY*DIMZ*sizeof(float), cudaMemcpyDeviceToHost );
  cudaFree( devicemem );
  return 0;
}
```

Thread-IDs im Block

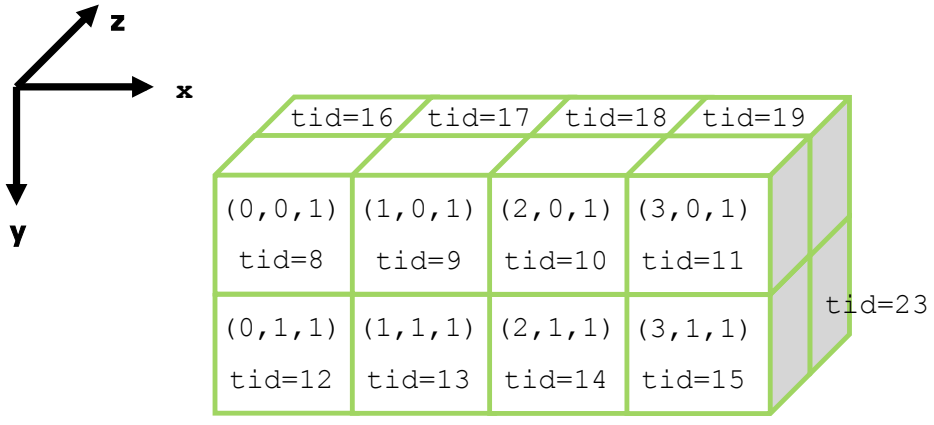

```
dim3 block3d(4,2,3);
settozero<<<1,block3d>>>(…);
```




```
int tid = threadIdx.x + threadIdx.y * DIMX + threadIdx.z * DIMX * DIMY;
```


Thread-IDs im Block

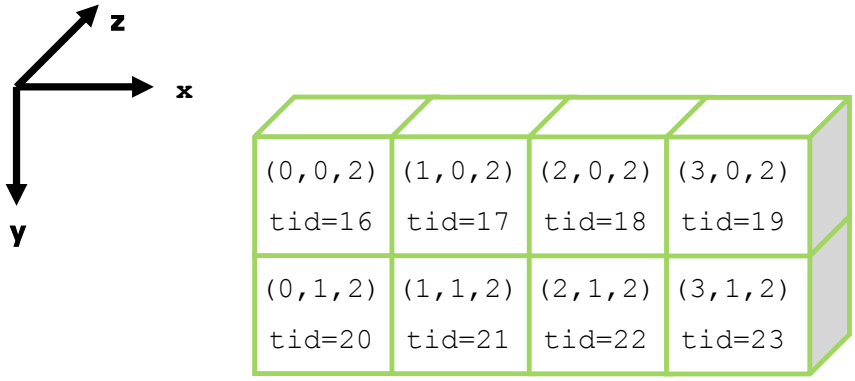

```
dim3 block3d(4,2,3);
settozero<<<1,block3d>>>(…);
```




```
int tid = threadIdx.x + threadIdx.y * DIMX + threadIdx.z * DIMX * DIMY;
```

Thread-IDs im Block

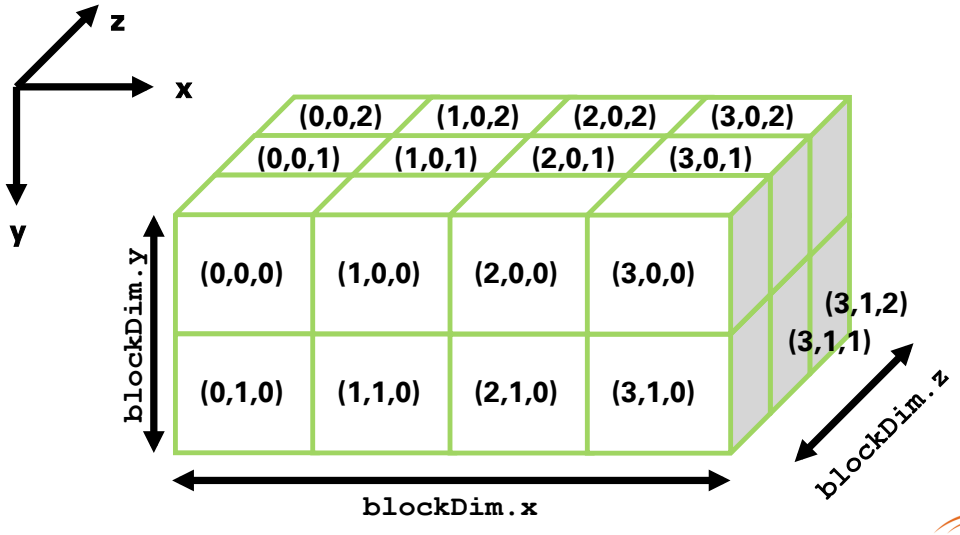

```
dim3 block3d(4,2,3);
settozero<<<1,block3d>>>(…);
```




```
int tid = threadIdx.x + threadIdx.y * DIMX + threadIdx.z * DIMX * DIMY;
```


CUDA hilft mit `blockDim`


```
dim3 block(4,2,3);
kernel<<<1,block>>>(…);
```



CUDA hilft mit `blockDim`


```
#define DIMX 4
#define DIMY 2
#define DIMZ 3
__global__ void settozero( float *elem ) {
  int tid = threadIdx.x              +
            threadIdx.y * blockDim.x +
            threadIdx.z * blockDim.x * blockDim.y;
  elem[tid] = 0.0f;
}

int main( int argc, char *argv[] ) {
…
  dim3 block3d(DIMX,DIMY,DIMZ);
…
}
```


1-THREAD-BLÖCKE IM GITTER STARTEN

Threads (1-Thread-Blöcke) im Gitter starten


```
dim3 grid1(8, 1, 1);
kernel<<<grid1,1>>>(…); //8 1-Thread-Blöcke, alle in x
```

oder auch einfach:

```
kernel<<<8,1>>>(…);     //8 1-Thread-Blöcke
```


Ein Beispiel: Paralleles `settozero` im Gitter


```
#define NELEMENTS 65535 //maximale Anzahl Blöcke im Gitter
__global__ void settozero( float *elem ) {

  int tid   = blockIdx.x;  //Warum steht hier nicht threadIdx.x?
  elem[tid] = 0.0f;        //Jeder Thread setzt ein Element auf 0
}
```

Achtung! In diesem Beispiel nehmen wir an, dass unser Thread-Block mit

```
dim3 grid1d(NELEMENTS,1,1)
settozero<<<grid1d,1>>>(…)
```

aufgerufen wird. Dies ist equivalent zum Aufruf

```
settozero<<<NELEMENTS,1>>>(…)
```

Wir starten damit 65535 1-Thread-Blöcke! Jeder Thread hat in seinem Block ID 0!

Threads (1-Thread-Blöcke) im Gitter starten


```
dim3 grid1(8, 1, 1);
kernel<<<grid1,1>>>(…); //8 1-Thread-Blöcke, alle in x
```

oder auch einfach:

```
kernel<<<8,1>>>(…);     //8 1-Thread-Blöcke
```

Andere Varianten:

```
dim3 grid2(1, 8, 1);
kernel<<<grid2,1>>>(…); //8 1-Thread-Blöcke, alle in y
dim3 grid3(1, 1, 8);
kernel<<<grid3,1>>>(…); //8 1-Thread-Blöcke, alle in z
dim3 grid4(2, 1, 4);
kernel<<<grid4,1>>>(…); //2 1-Thread-Blöcke in x, 1 in y, 4 in z
dim3 grid5(2, 2, 2);
kernel<<<grid5,1>>>(…); //Je 2 1-Thread-Blöcke in x, y und z
```


Block-ID und Gittergröße

Wie bei `threadIdx` **gibt es auch eine ID für Blocks:**

```
blockIdx.x //Block-ID im Gitter in x
blockIdx.y //Block-ID im Gitter in y
blockIdx.z //Block-ID im Gitter in z
```

Analog zur Blockgröße `blockDim` **gibt es auch eine Gittergröße**

```
gridDim.x //Größe des Gitters in x
gridDim.y //Größe des Gitters in y
gridDim.z //Größe des Gitters in z
```

Wiederholung: Matrix auf Null mit Block-Gitter


```
#define DIMMAT 255
__global__ void settozero( float *elem ) {
  int tidx         = blockIdx.x;
  int tidy         = blockIdx.y;
  elem[tidx][tidy] = 0.0f;
}
int main( int argc, char *argv[] ) {

  float matrixa[DIMMAT][DIMMAT];
  float *devicemem;
  dim3  grid2d(DIMMAT,DIMMAT,1);

  cudaMalloc( (void**)&devicemem, DIMMAT*DIMMAT*sizeof(float) );
  settozero<<<grid2d,1>>>( devicemem ); //startet DIMMAT*DIMMAT 1-Thread-Blöcke
  cudaMemcpy( veca, devicemem, DIMMAT*DIMMAT*sizeof(float), cudaMemcpyDeviceToHost );
  cudaFree( devicemem );
  return 0;
}
```

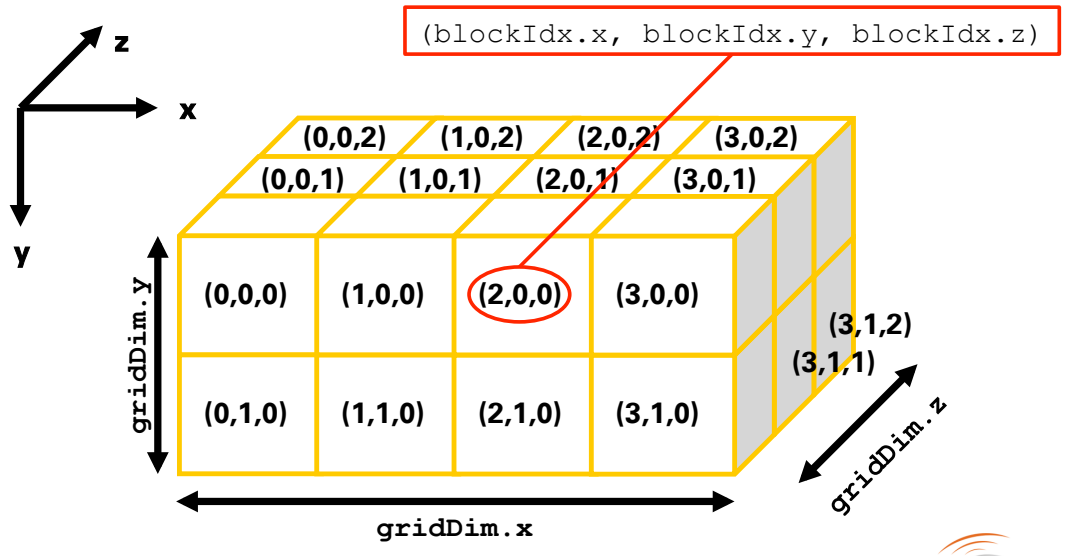

Threads in Blöcken vs. 1-Thread-Blöcke in Gittern


```
#define DIMX 4
#define DIMY 2
#define DIMZ 3
__global__ void settozero( float *elem ) {
  int tid = threadIdx.x      +
    threadIdx.y * blockDim.x +
    threadIdx.z * blockDim.x * blockDim.y;

  elem[tid] = 0.0f;
}

int main( int argc, char *argv[] ) {
…
  dim3 block3d(DIMX,DIMY,DIMZ);
…
  settozero<<<1,block3d>>>( devicemem )
…
}
```

```
int tid = blockIdx.x     +
  blockIdx.y * gridDim.x +
  blockIdx.z * gridDim.x + gridDim.y;

dim 3 grid3d(DIMX,DIMY,DIMZ);

settozero<<<grid3d,1>>>( devicemem )
```

Blockverteilung im Gitter

```
dim3 grid(4,2,3);
kernel<<<grid,1>>>(…);
```



BLÖCKE IM GITTER STARTEN

Noch einmal: Paralleles `settozero`

Das Problem mit unserem parallelen `settozero` ist, dass wir im Block auf `maxThreadsPerBlock` (512 oder 1024) Threads beschränkt sind.

Das heißt, unser Code kann nur Vektoren (Arrays) mit maximal so vielen Elementen wie Threads in einen Block passen auf Null setzen.

Mit Gittern können wir immerhin schon `maxGridSize[1]` 1-Thread-Blöcke starten, also insgesamt 65535 oder 2^{31}-1 Threads., je nach Compute Capability.

Jetzt wollen wir Gitter aus Blöcken bauen, bei denen jeder Block mehrere Threads enthält. So können wir zunächst einmal bis zu

`maxThreadsPerBlock * maxGridSize[1]` $\leq 1024*(2^{31}-1)$

Kernelaufrufe starten.

Paralleles `settozero` im 1D Blockgitter


```
#define NTHREADS 512   //maximale Anzahl Threads pro Block
#define NBLOCKS  65535 //maximale Anzhal von Blöcken im Gitter
__global__ void settozero( float *elem ) {

  int tid   = threadIdx.x + blockIdx.x * blockDim.x;
  elem[tid] = 0.0f;
}

dim3 block1d(NTHREADS,1,1);
dim3 grid1d(NBLOCKS,1,1);
settozero<<<grid1d,block1d>>>(…)
```

Dies ist equivalent zum Aufruf

```
settozero<<<NBLOCKS,NTHREADS>>>(…)
```

Paralleles `settozero` im 1D Blockgitter

Gitter	Blöcke				
blockIdx.x 0	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 1	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 2	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
...	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 65534	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511

Paralleles `settozero` im 1D Blockgitter

Gitter	Blöcke				
	threadIdx.x 0	threadIdx.x 1	threadIdx.x 2	...	threadIdx.x 511
blockIdx.x 0	tid=0	tid=1	tid=2	...	tid=511
blockIdx.x 1	tid=512	tid=513	tid=514	...	tid=1023
blockIdx.x 2	tid=1024	tid=1025	tid=1026	...	tid=1535
blockIdx.x 3	tid=1536	tid=1537	tid=1538	...	tid=2047

```
tid = threadIdx.x + blockIdx.x * blockDim.x;
```

Paralleles `settozero` im 1D Blockgitter


```
int main( int argc, char *argv[] ) {

  float veca[NTHREADS*NBLOCKS];
  float *devicemem;
  dim3  block1d(NTHREADS,1,1);
  dim3  grid1d(NBLOCKS,1,1);

  cudaMalloc( (void**)&devicemem, NTHREADS*NBLOCKS*sizeof(float) );
  settozero<<<grid1d,block1d>>>( devicemem ); //übergibt ein Array
  cudaMemcpy( veca, devicemem, NTHREADS*NBLOCKS*sizeof(float),
              cudaMemcpyDeviceToHost );  //kopiert Array
  cudaFree(devicemem);
  return 0;
}
```


Tabelle zu IDs und Größen

Typ	ID	Größe
Thread	`threadIdx`	`1`
Block	`blockIdx`	`blockDim`
Grid	`-`	`gridDim`

Blöcke im Gitter starten


```
dim3 grid1(8, 1, 1);
dim3 block(nx,ny,nz);
kernel<<<grid1,block>>>(…); //8 Blöcke, alle in x
```

oder auch einfach:

```
kernel<<<8,block>>>(…);     //8 Blöcke
```


Blöcke im Gitter starten


```
dim3 grid1(8, 1, 1);
dim3 block(nx,ny,nz);
kernel<<<grid1,block>>>(…); //8 Blöcke, alle in x
```

oder auch einfach:

```
kernel<<<8,block>>>(…);     //8 Blöcke
```

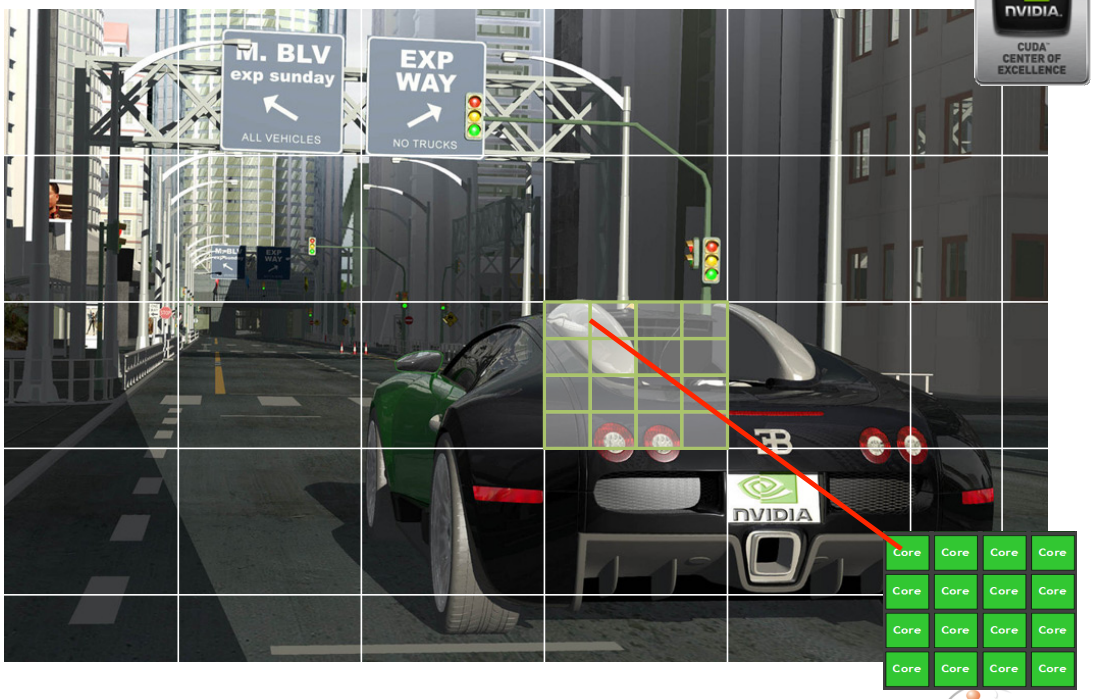
Andere Varianten:

```
dim3 grid2(1, 8, 1);
kernel<<<grid2,block>>>(…); //8 Blöcke, alle in y
dim3 grid3(1, 1, 8);
kernel<<<grid3,block>>>(…); //8 Blöcke, alle in z
dim3 grid4(2, 1, 4);
kernel<<<grid4,block>>>(…); //2 Blöcke in x, 1 in y, 4 in z
dim3 grid5(2, 2, 2);
kernel<<<grid5,block>>>(…); //Je 2 Blöcke in x, y und z
```

Threads in Blöcken in Gittern

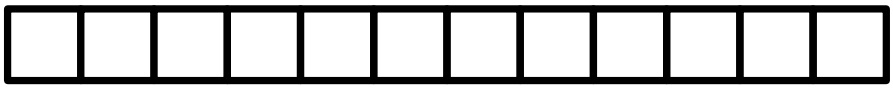
PARALLELE TECHNIKEN: DOMAIN DECOMPOSITION

Gebietszerlegung (Domain Decomposition)

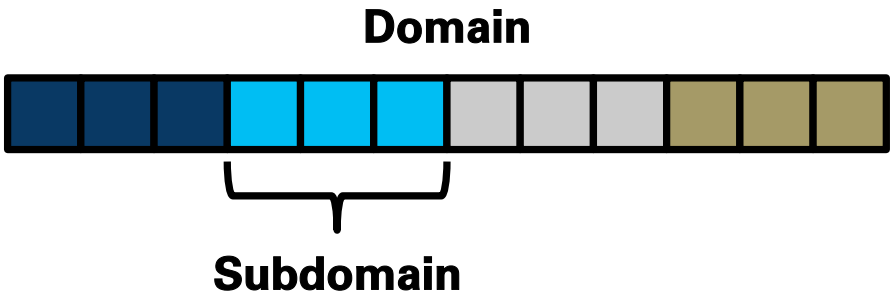


Domain Decomposition in 1D

Domain



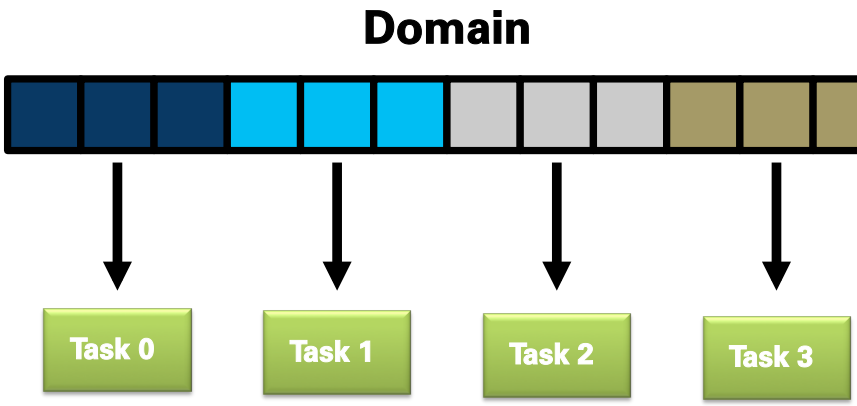
Domain Decomposition in 1D



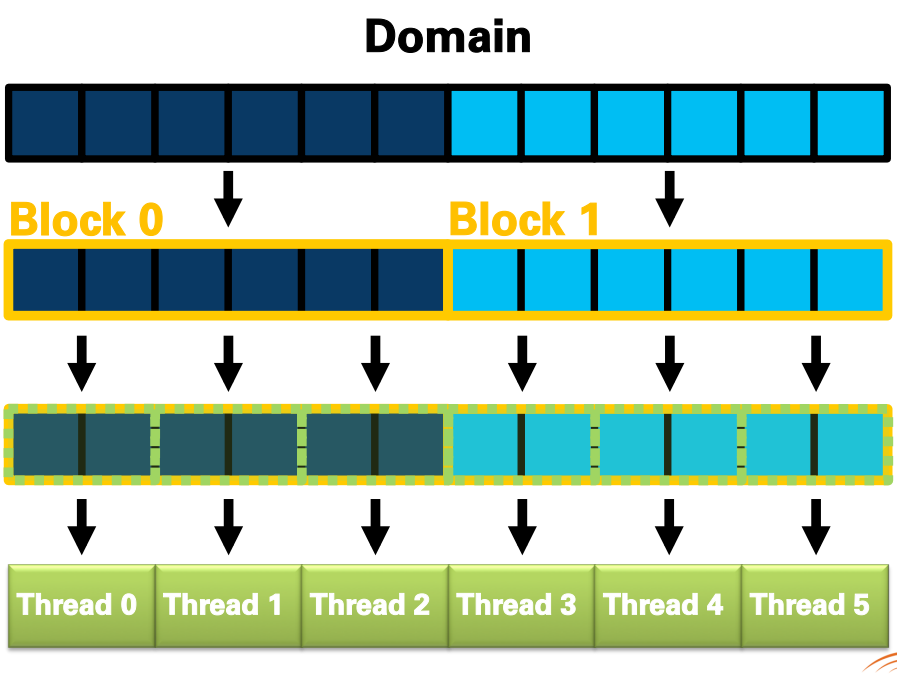
„Partitioning“

„Subdivision“

Domain Decomposition in 1D



Domain Decomposition in 1D in CUDA



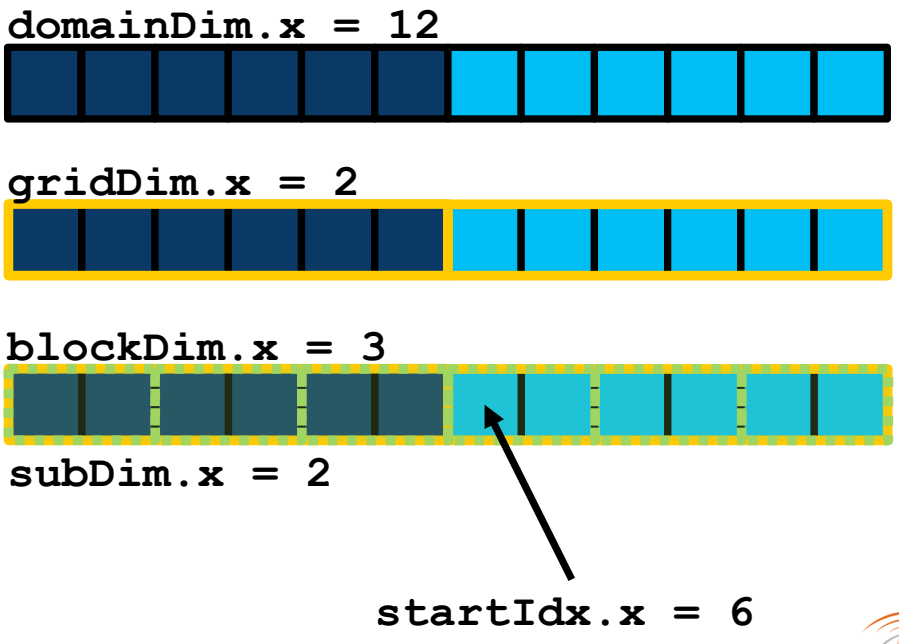
Domain Decomposition in 1D in CUDA

```
__global__ void settozero( float *elem, int NELEMENTS ) {

  int subDim_x    = NELEMENTS / ( blockDim.x * gridDim.x ); //Anz. Elemente in Subarray
  int tid         = threadIdx.x + blockIdx.x * blockDim.x;  //Thread-ID
  int subStartIdx = tid * subDim_X                          //Startindex im Array
  int i;                                                    //Laufindex im Array

  for ( i=0; i++; i<subDimx ) elem[i+subStartIdx] = 0.0f;
}

int main( int argc, char *argv[] ) {

  const int NELEMENTS = 128000;
  float veca[NELEMENTS];
  …
  settozero<<<10,128>>>( devicemem, NELEMENTS ); //subDim_x = 100
  …
}
```

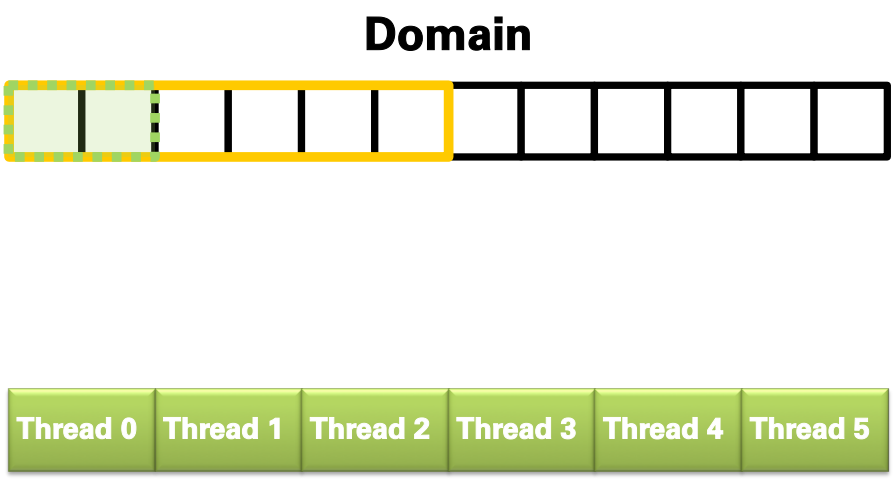
Die Welt in 3D


```
__global__ void WorkOnSubdomain( data *domain, dim3 domainDim ) {

  dim3 subDim;      //Größe der Subdomain in x,y,z
  dim3 tid;         //Thread-ID in x,y,z
  dim3 subStartIdx; //Anfangsposition der Daten der Subdomain in der Domain

  subDim.x      = domainDim.x / ( blockDim.x * gridDim.x ); //Achtung, Integerdivision
  subDim.y      = domainDim.y / ( blockDim.y * gridDim.y ); //Achtung, Integerdivision
  subDim.z      = domainDim.z / ( blockDim.z * gridDim.z ); //Achtung, Integerdivision
  tid.x         = threadIdx.x + blockIdx.x * blockDim.x;
  tid.y         = threadIdx.y + blockIdx.y * blockDim.y;
  tid.z         = threadIdx.z + blockIdx.z * blockDim.z;
  subStartIdx.x = tid.x * subDim.x;
  subStartIdx.y = tid.y * subDim.y;
  subStartIdx.z = tid.z * subDim.z;

  //Schleifen in x,y,z oder ähnliches…
}
```

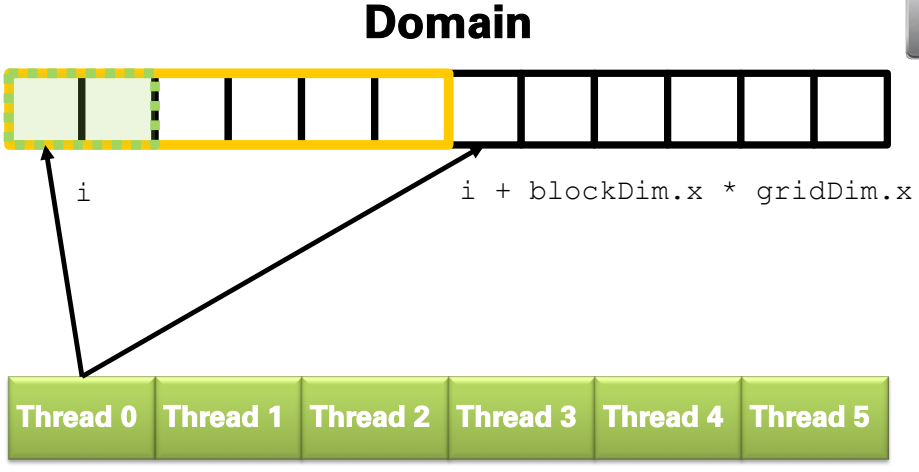

Domain Decomposition in 1D in CUDA



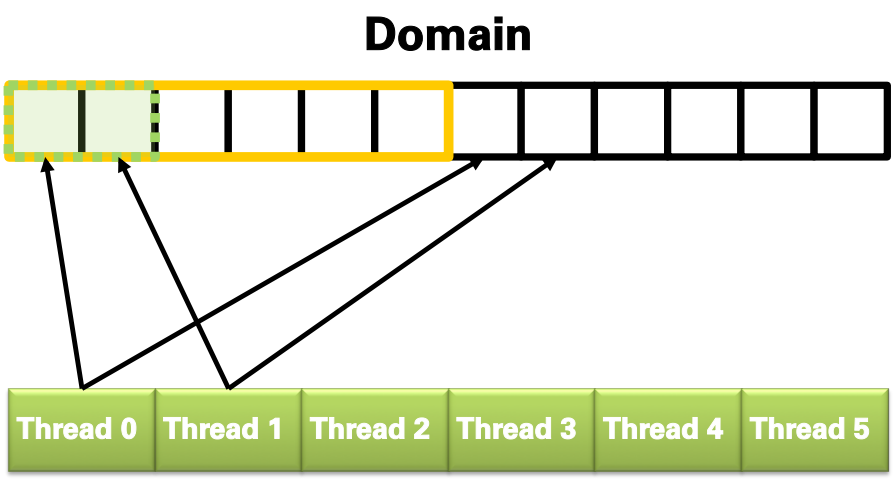
Die „CUDA by Example“-Variante



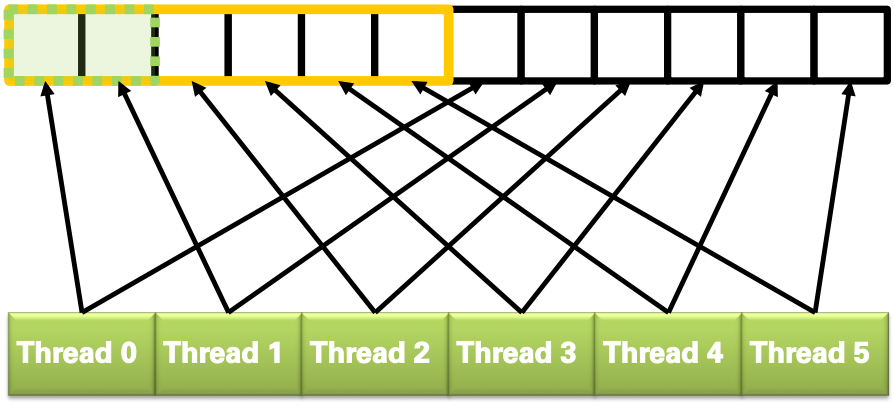
Die „CUDA by Example“-Variante



Die „CUDA by Example“-Variante



Die „CUDA by Example“-Variante



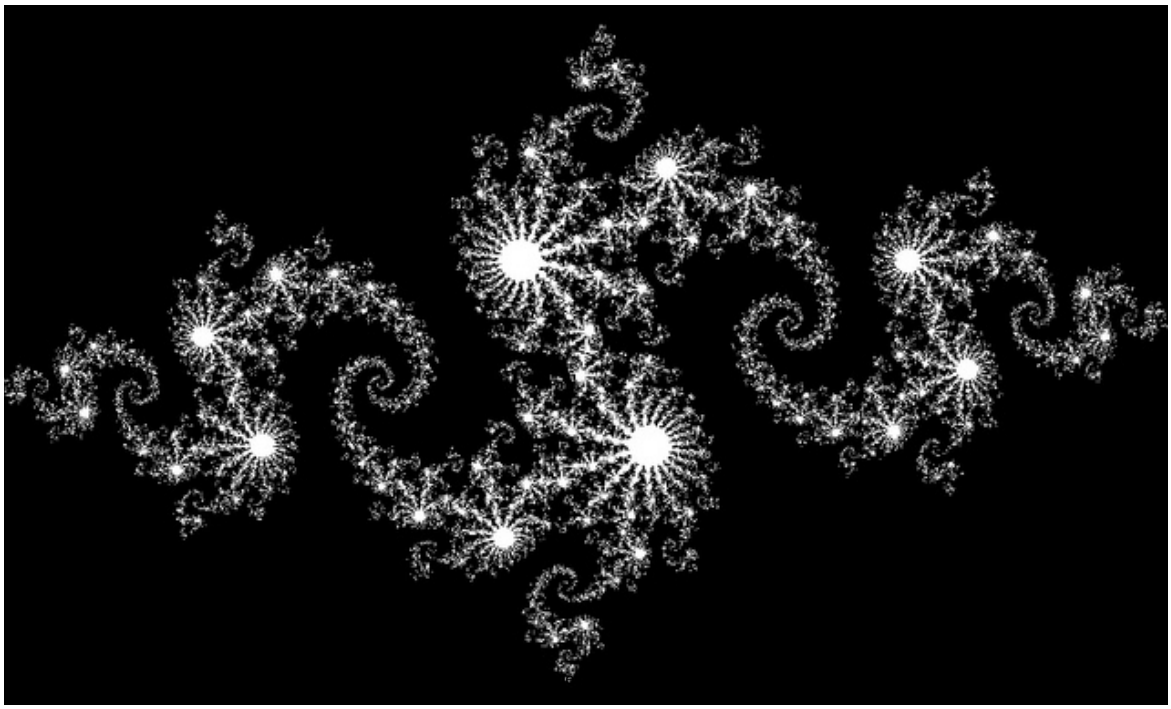
Domain Decomposition in 1D „CUDA by Example“


```
__global__ void settozero( float *elem, int NELEMENTS ) {

  int tid    = threadIdx.x + blockIdx.x * blockDim.x;  //Thread-ID
  int idxinc = blockDim.x * gridDim.x;                 //Index-Inkrement im Array
  int i      = tid;                                    //Laufindex im Array

  while ( i<NELEMENTS ) {
    elem[i]  = 0.0f;
    i       += idxinc;
  }
}
```


BEISPIELE FÜR DOMAIN DECOMPOSITION

„CUDA by Example“

Julia-Menge



Paralleles Suchen

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

--

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Paralleles Suchen

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

--

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Paralleles Suchen

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

--

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Paralleles Suchen

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

--

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Paralleles Suchen

Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

--

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

Paralleles Suchen

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

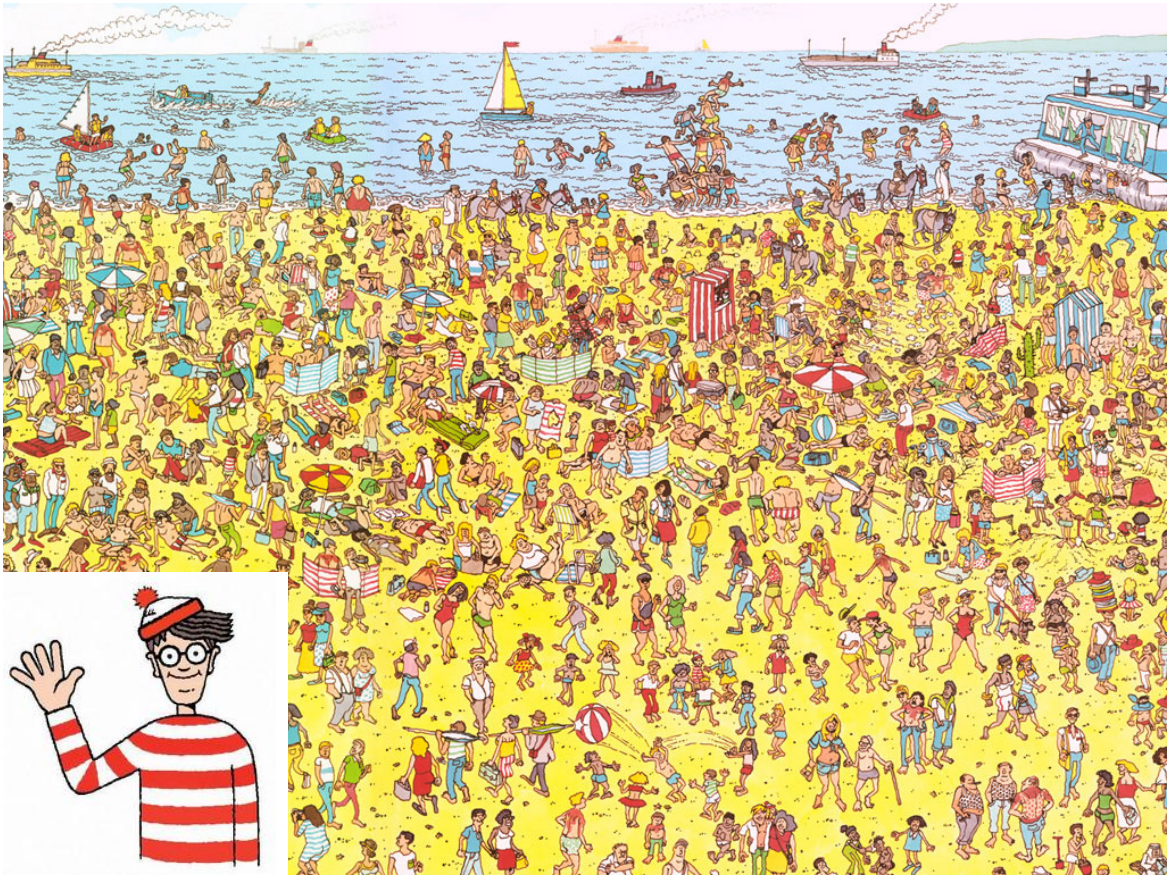
--

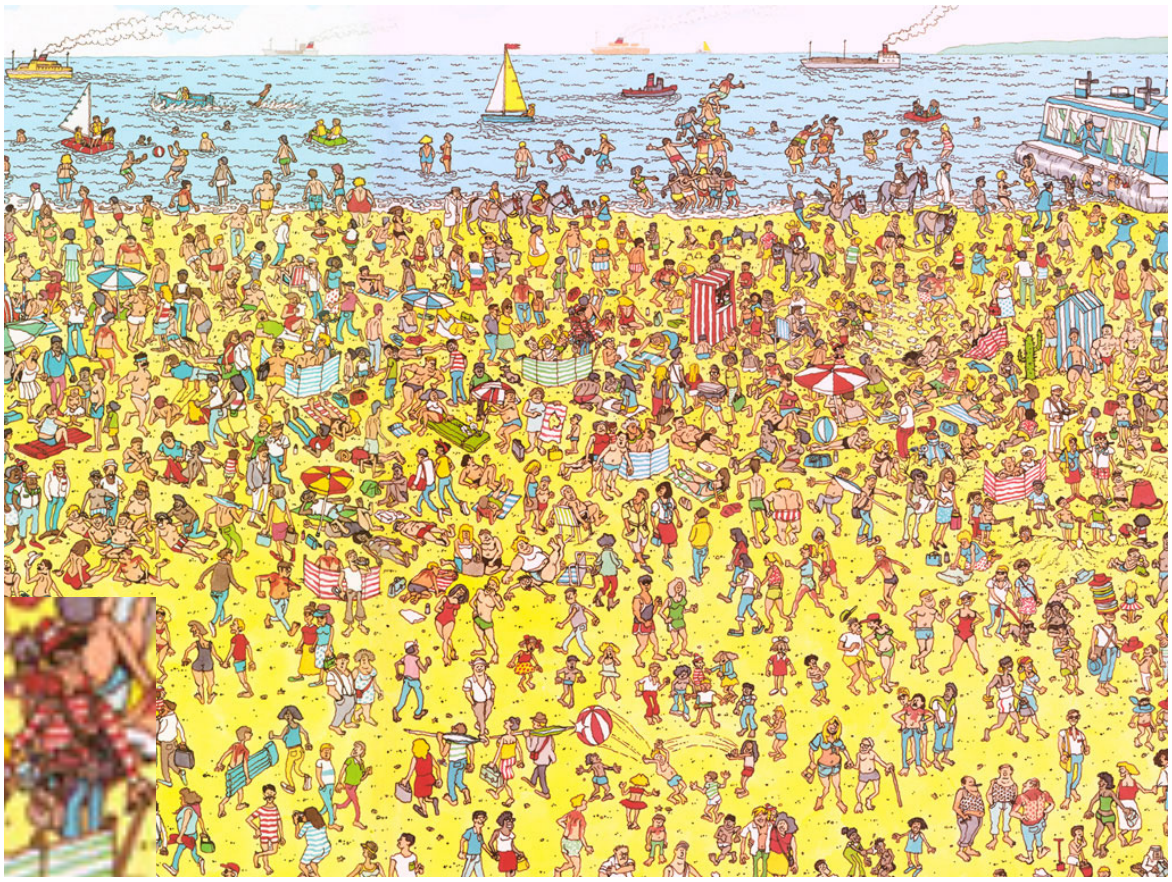
Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

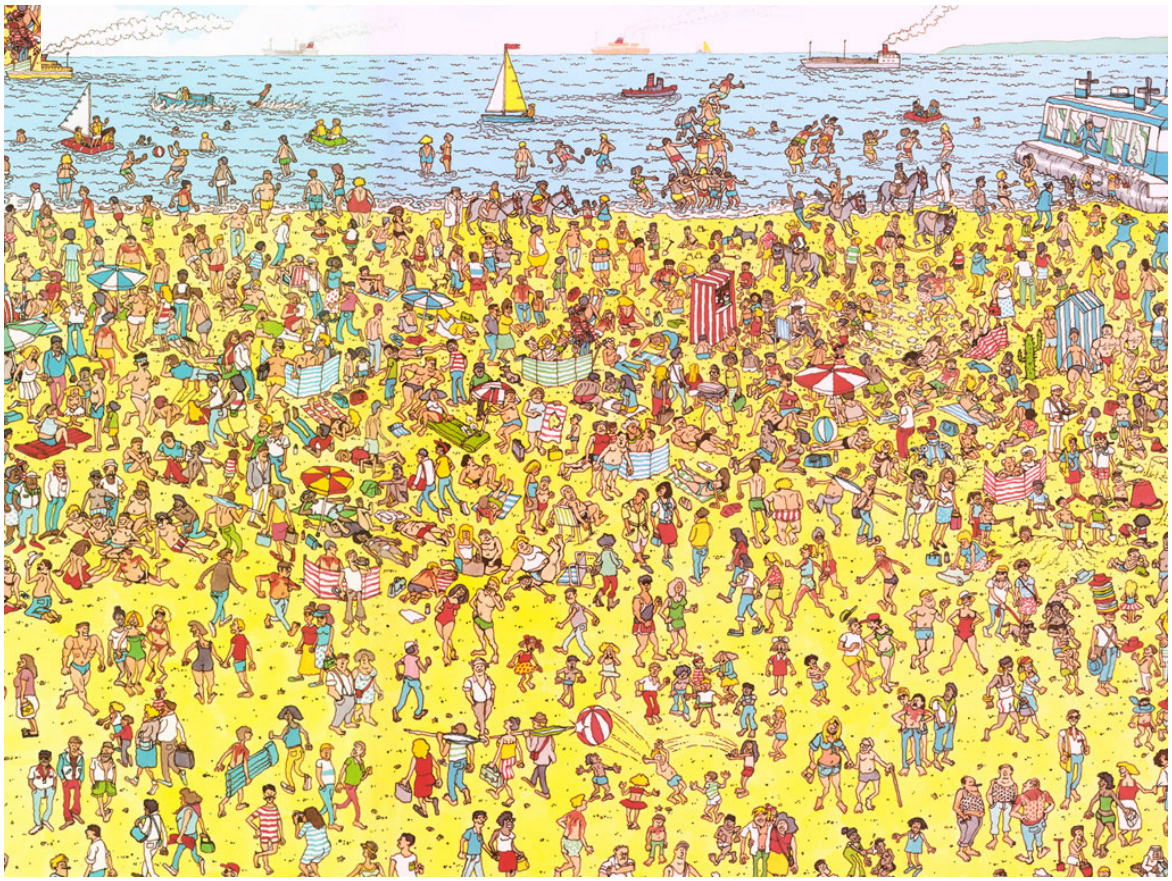
Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

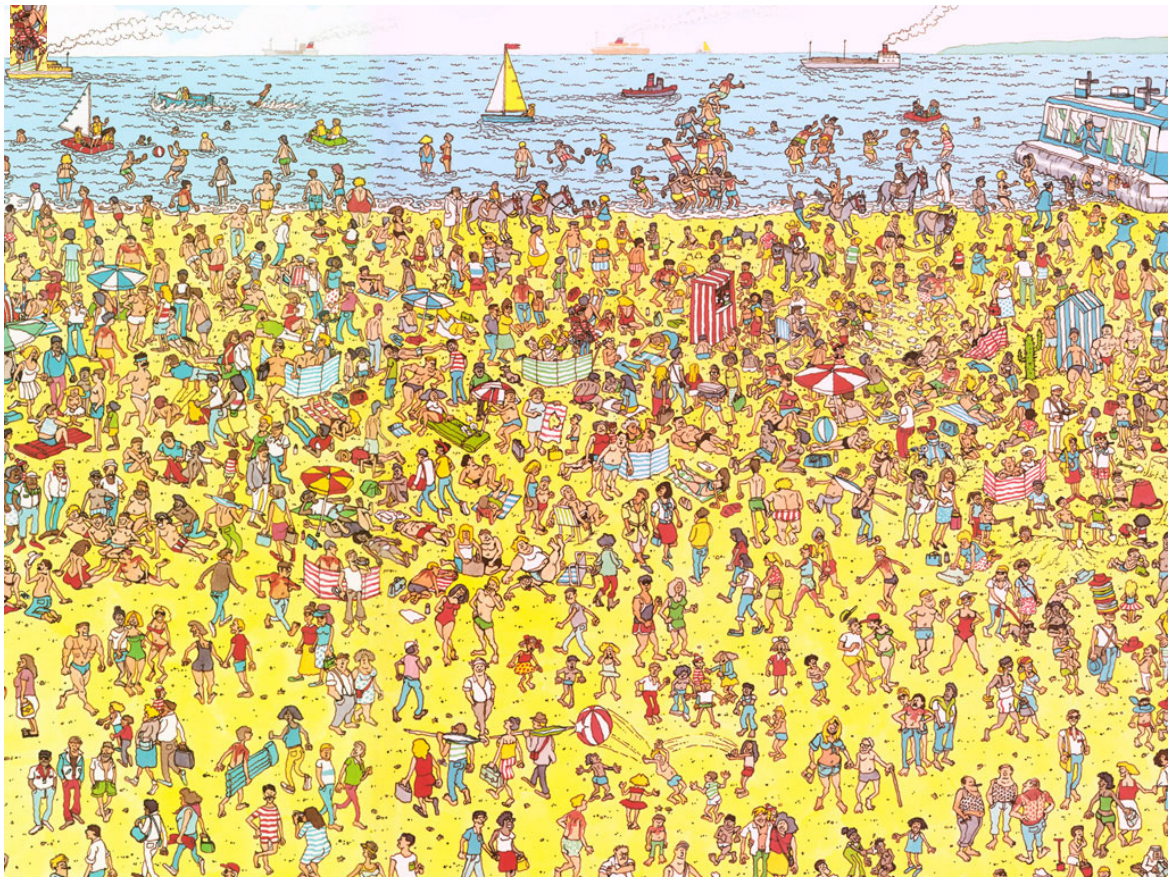
Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

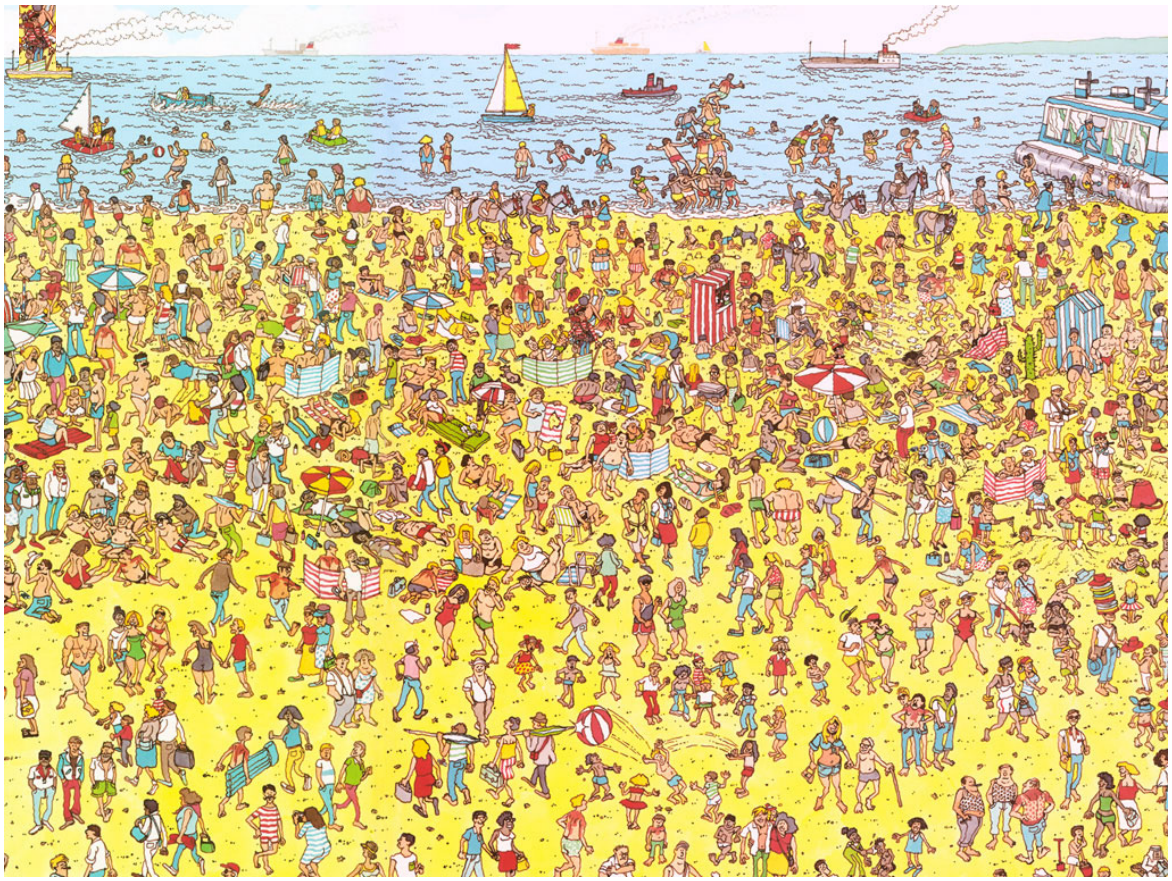
Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

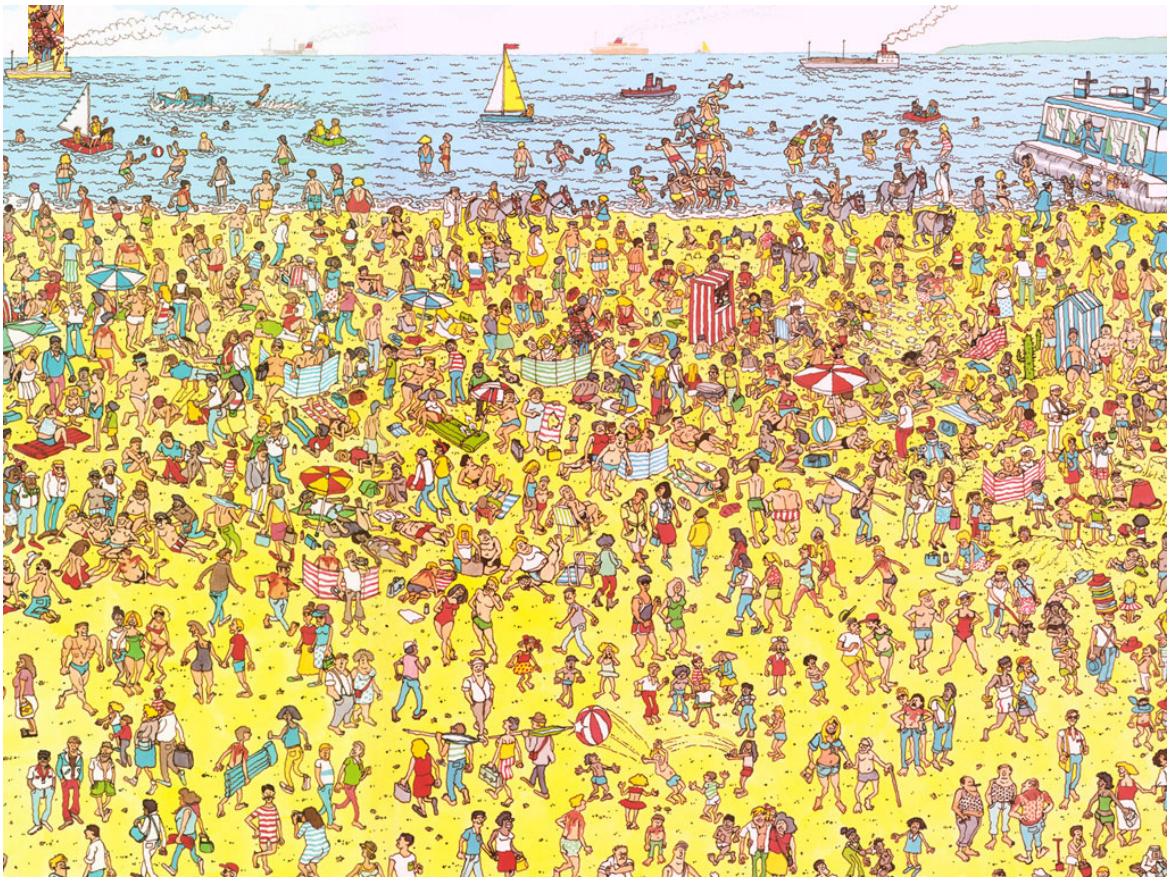






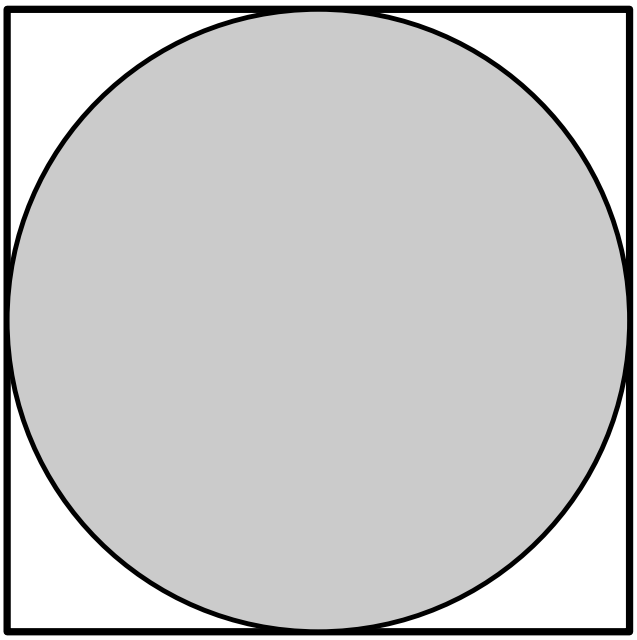




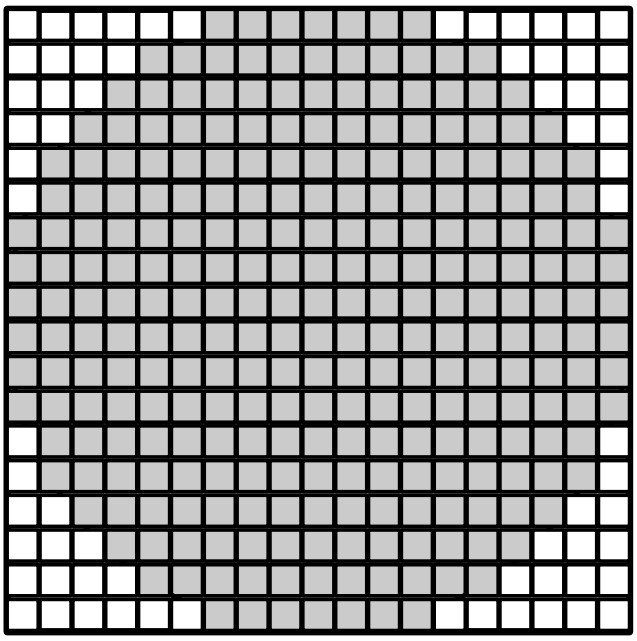


EIN BEISPIEL: π PARALLEL BERECHNEN

Fläche = πr^2



Zerlegen der Fläche in Pixel



Muss das Pixel grau oder weiss sein?

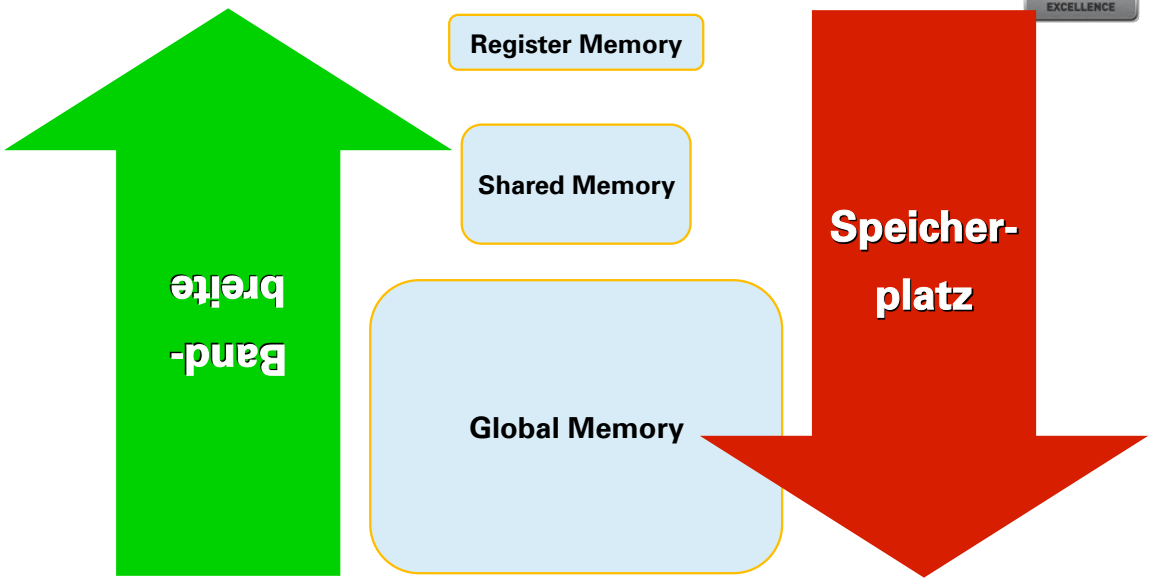
```
#define NBLOCKS  = 4096
#define NTHREADS = 128
__global__ void setgreypixels(byte *pixels) {
  int    tidx  = threadIdx.x + blockIdx.x * blockDim.x;          //Thread-ID in
  x
  int    tidy  = threadIdx.y + blockIdx.y * blockDim.y;          //Thread-ID in
  y
  float  x;    = float(tidx) / ( blockDim.x * gridDim.x ) - 0.5; //x-Pos Pixel
  float  y;    = float(tidy) / ( blockDim.y * gridDim.y ) - 0.5; //y-Pos Pixel
  int    pidx  = tidx * blockDim.x * gridDim.x + tidy;           //Pixel-ID

  pixels[pidx] = byte( ( x*x + y*y ) <= ( 0.25 ) ); //Schlecht: C-Konvention!
}
…
dim3 blocksquare(NTHREADS, NTHREADS);
dim3 gridsquare(NBLOCKS, NBLOCKS);
…
setgreypixels<<<gridsquare,blocksquare>>>(devPixels);
```

Pixel zählen und Fläche normieren

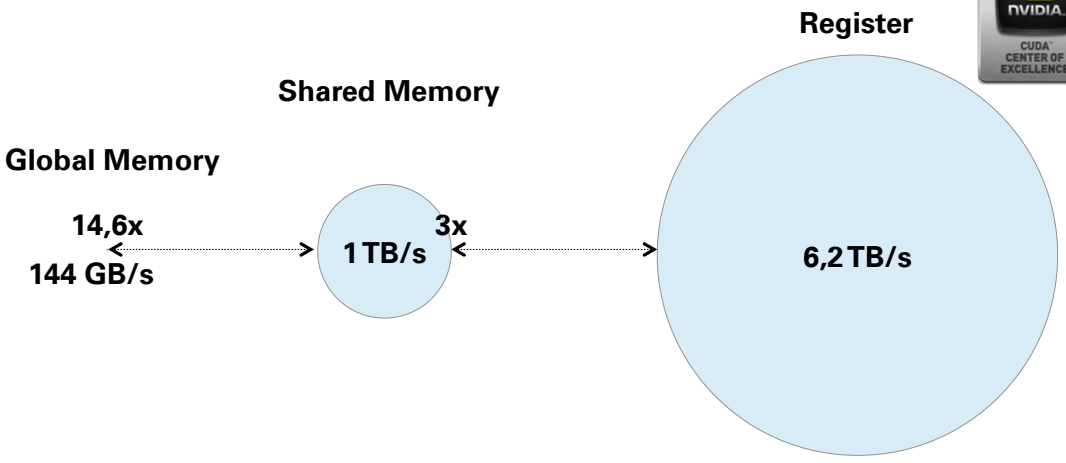

```
int main( int argc, char *argv[] ) {

  …
  dim3  blocksquare(NTHREADS, NTHREADS);              //quadratischer Block
  dim3  gridsquare(NBLOCKS, NBLOCKS);                 //quadratisches Grid
  int   pidx;
  int   npixels = 0;
  float pi;
  …                                                    //malloc host/device
  setgreypixels<<<gridsquare,blocksquare>>>(devPixels);
  …                                                    //cudaMemcpy
  for ( pidx=0; pidx++; pidx<=NTHREADS*NBLOCKS ) npixels += hostPixels[pidx];
  pi = float( NTHREADS * NBLOCKS ) / float( npixels );
  …                                                    //printf, free, etc.
}
```


SPEICHERZUGRIFFE

Speicherhierarchie



Speicherbandbreite



Tesla C2050 (FERMI)

Shared memory: 4 Byte * 32 Banks * 14 SM * 0,5 * 1,15GHz = 1,03 Tbyte/s

Registers (a*b+c): 3 * 4 Byte * 14 SM * 32 Threads * 1.15 GHz = 6,18 Tbyte/s

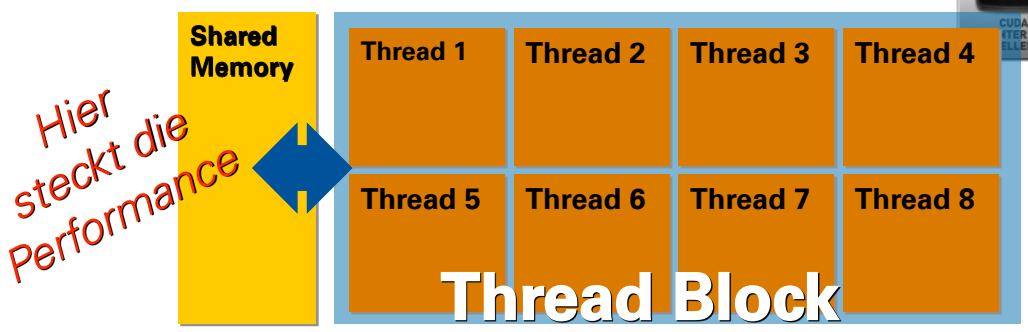
Register-Speicher

Anzahl von 32 bit Registern pro Multiprozessor

Compute Capability 1.0-1.1:	8k
Compute Capability 1.2-1.3:	16k
Compute Capability 2.x:	32k
Compute Capability 3.0, 3.5:	64k

Was?	Registerspeicher ist der lokale Speicher für eine ALU, on-chip
Wie?	Einfach im Kernel eine Variable deklarieren, z.B. `int counter;`
Wer?	Genau einem Thread zugewiesen, nur dieser kann darauf zugreifen
Lebensdauer?	Während der Laufzeit des Threads
Zugriff?	Lesen und Schreiben

Shared Memory



Maximaler Shared Memory pro Multiprozessor

Compute Capability 1.0-1.3:	16k
Compute Capability 2.x, 3.0, 3.5:	48k

Was?	Speicher zum **schnellen** Datenaustausch für Threads, on-chip
Wie?	`__shared__ float commondata[threadsPerBlock];`
Wer?	**Alle Threads in einem Block**
Lebensdauer?	Während der Laufzeit des Thread Blocks
Zugriff?	Lesen und Schreiben

Shared Memory: Feste Größe


```
__global__ void foo(){
    __shared__ int a[256];
    …
}
```

Erzeugt ein Integer Array „a“ mit 256 Elementen im Shared Memory für jeden Threadblock

Achtung!
Größe des Arrays im Kernel festgelegt!

Shared Memory: Dynamische Größe zur Laufzeit


```
__global__ void foo(){
    extern __shared__ int a[];
    …
}

int main(…){
    …
    foo<<< NBLOCKS, NTHREADS,
    NTHREADS*sizeof(int)>>>();
    …
}
```

Erzeugt ein Integer Array „a“ mit NTHREADS Elementen im Shared Memory für jeden Threadblock

Shared Memory: Füllen

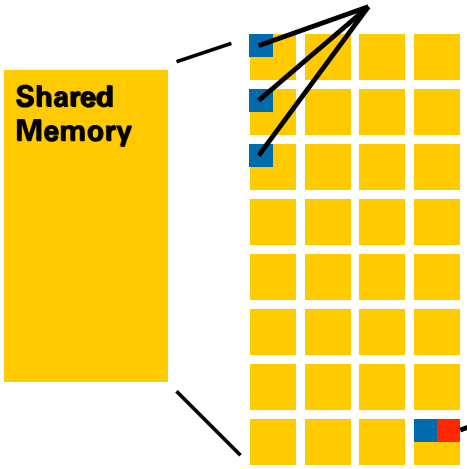

```
__global__ void foo(int *b_gpu){
    __shared__ int a[256];
    idx=… //get global index
    …
    // every thread copies one b to a
    a[threadIdx.x]=b[idx];

    // wait for all threads in a block
    __syncthreads;
    …
}
```


Shared Memory: Typische Verwendung


```
__global__ void foo(int *b_gpu){
    __shared__ int a[256];
    idx=… //get global index
    …
    // copy reused date to shared memory
    // sync
    // compute
    // sync
    // write back to global memory
    // evtl. sync
}
```

Struktur des Shared Memory: Banks

Hintereinanderliegende 32 bit Worte werden in hintereinanderliegende banks abgelegt (*interleaving*)

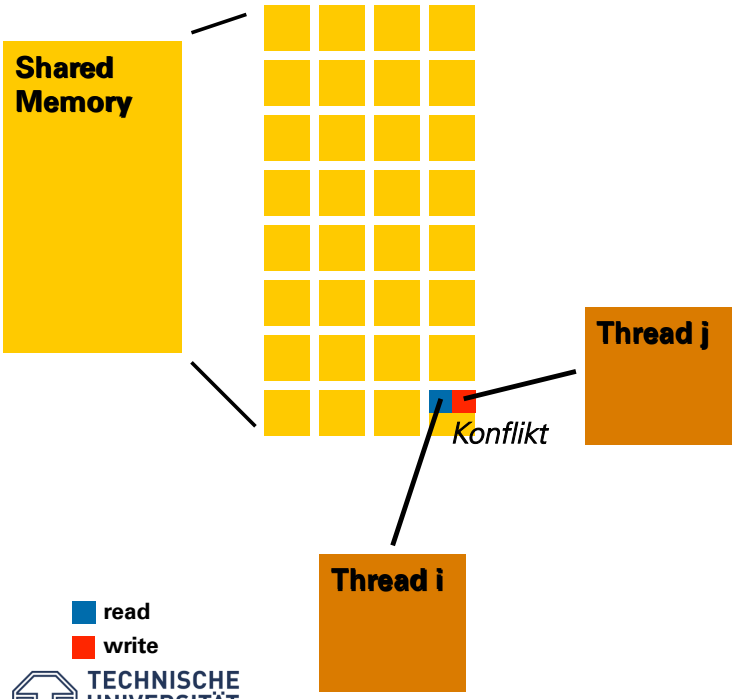


Anzahl der Shared Memory Banks

Compute Capability 1.0-1.3: 16
Compute Capability 2.x, 3.0, 3.5: 32

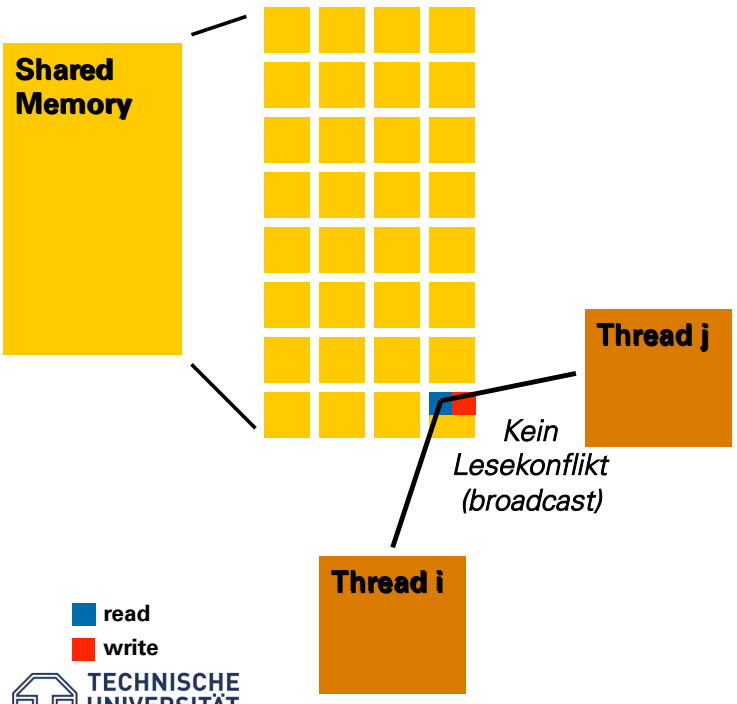
read
write

Shared Memory: Bank-Konflikte

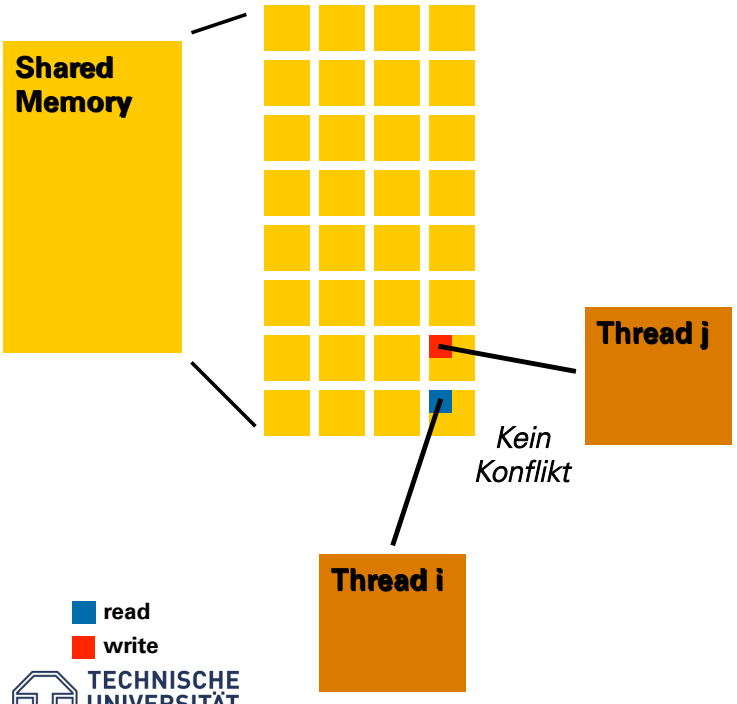


read
write

Shared Memory: Bank-Konflikte



Shared Memory: Bank-Konflikte



Strided Zugriff

32 Bit

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

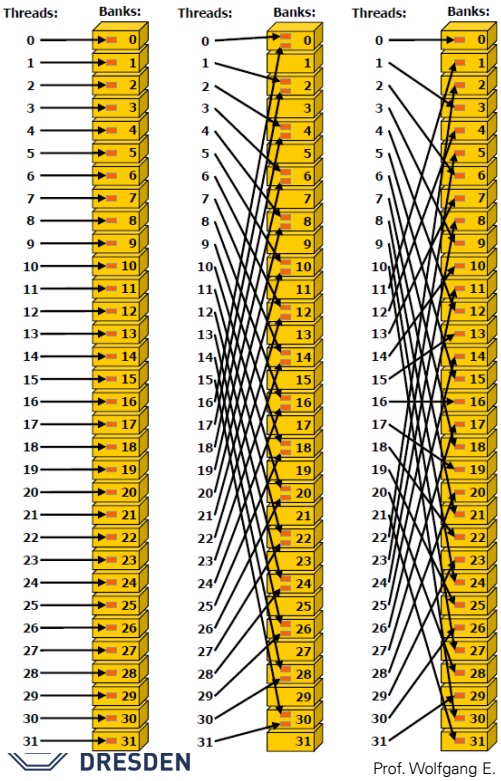
`tid` und `tid+n` greifen auf dieselbe Bank zu, falls `s*n`=Anzahl Banks

>32 Bit

```
struct type {
  float x, y, z;
};

__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

3 separate 32-Bit-Lesezugriffe ohne Bank-Konflikte,
da auf `X,Y,Z` mit einem Stride von 32 Bit zugegriffen wird

Strided Zugriff und Bank-Konflikte (SM 2.x)



Links:

Lineare Adressierung mit 1x32 Bit Stride
Kein Bank-Konflikt

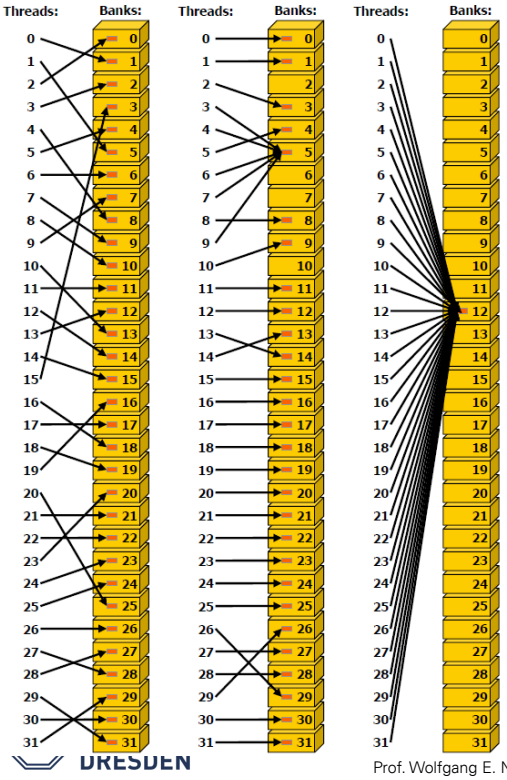
Mitte:

Lineare Adressierung mit 2x32 Bit Stride
2-Wege Bank-Konflikt

Rechts:

Lineare Adressierung mit 3x32 Bit Stride
Kein Bank-Konflikt

Strided Zugriff und Bank-Konflikte (SM 2.x)



Links:

Zufällige Adressierung
Kein Bank-Konflikt

Mitte:

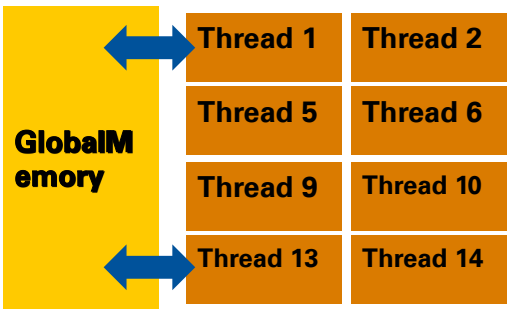
Gemischte Adressierung
Kein Bank-Konflikt, da Threads 3,4,6,7,9 auf dasselbe 32 Bit-Wort zugreifen

Rechts:

„Broadcast“
Kein Bank-Konflikt

Global Memory: Viel Platz, aber laaaaangsaaaaam

Das kann man maximal auf eine GPU packen



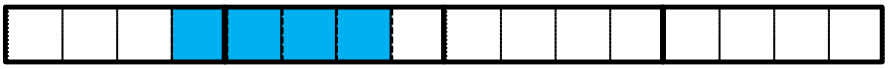
Wenn man nur Global Memory nutzt, sollte man sich lieber CPUs kaufen

Derzeit ca. 4 bis 6 GB

Was?	**Speicher zum langsamen Datenaustausch zwischen Blocks**
Wie?	`__global__ float commondata[maxMem];`
Wer?	**Alle Threads auf dem Device, Host**
Lebensdauer?	**Laufzeit des Programms**
Zugriff?	**Lesen und Schreiben**

Nicht ausgerichtet gespeicherte Daten (global mem)

Speicher (4 x 128 Byte)



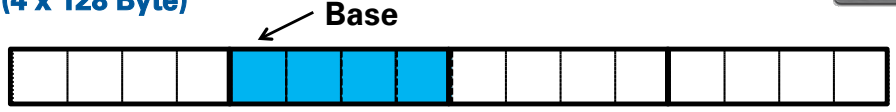
Angeforderter Speicher (1 x 128 Byte not aligned)

Übertragener Speicher (2 x 128 Byte)

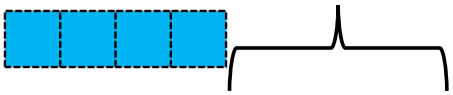
50% verschenkte Speicherbandbreite

Ausgerichtet gespeicherte Daten (global memory)

Speicher (4 x 128 Byte)



Angeforderter Speicher (1 x 128 Byte aligned)



Übertragener Speicher (1 x 128 Byte)

2D Speicherzugriff

```
T* elem = (T*)((char*)Base + Row * pitch) + Column
```

Daten ausgerichtet speichern

Speicher wird ausgerichtet erstellt (auf der Host Seite)

1D Array

- `cudaMalloc()` — Startadresse ist ausgerichtet

2D Array

- `cudaMallocPitch()` — Startadresse jeder Zeile ist ausgerichtet

Weitere Mallocs für Device Speicher

- `cudaMalloc3D()`, `cudaMallocArray()`, `cudaMalloc3DArray()`

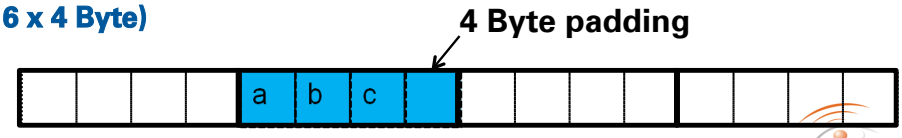
Beispiel: Ausrichtung von Strukturen und Klassen


```
struct Foo {
       float a;
       float b;
       float c; }
// sizeof(Foo) == 12
```

Speicher (16 x 4 Byte)


```
struct __align__(16) Bar {
       float a;
       float b;
       float c; }
// sizeof(Bar) == 16
```

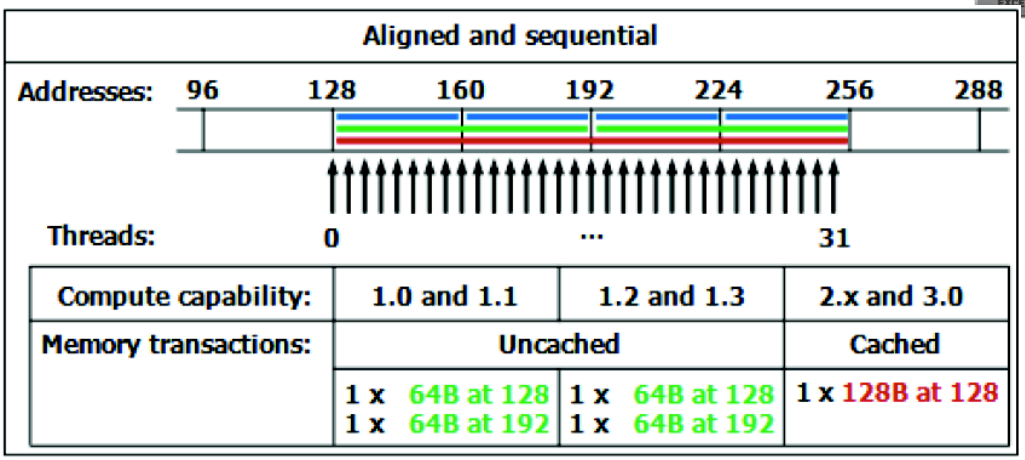
Speicher (16 x 4 Byte)



Caching für Global Memory (ab CC 2.x)

Wie?	**Compiler-Flag `-dlcm`** **`-Xptxas -dlcm=ca` (L1 und L2 Cache genutzt)** **`-Xptxas -dlcm=cg` (nur L2 Cache genutzt)**
Was passiert?	**Es werden immer 128 Bytes gecached (CC 2.x)** **Diese entsprechen ausgerichteten 128 Byte-Segmenten** **Mit L1+L2 Cache werden einmal 128 Byte transferiert** **Mit L2 Cache alleine werden 4x32 Byte transferiert**
Und dann?	**Wird mehr als 128 Byte pro Thread benötigt, so wird eine Speicheranfrage pro Warp in separate 128-Byte-Anfragen gesplittet**
Parallelität?	**128-Byte-Anfragen werden unabhängig voneinander bearbeitet** **Ist das Datum im Cache, geschieht dies mit der Cache-Geschwindigkeit, sonst mit der Geschwindigkeit des Global Memory**

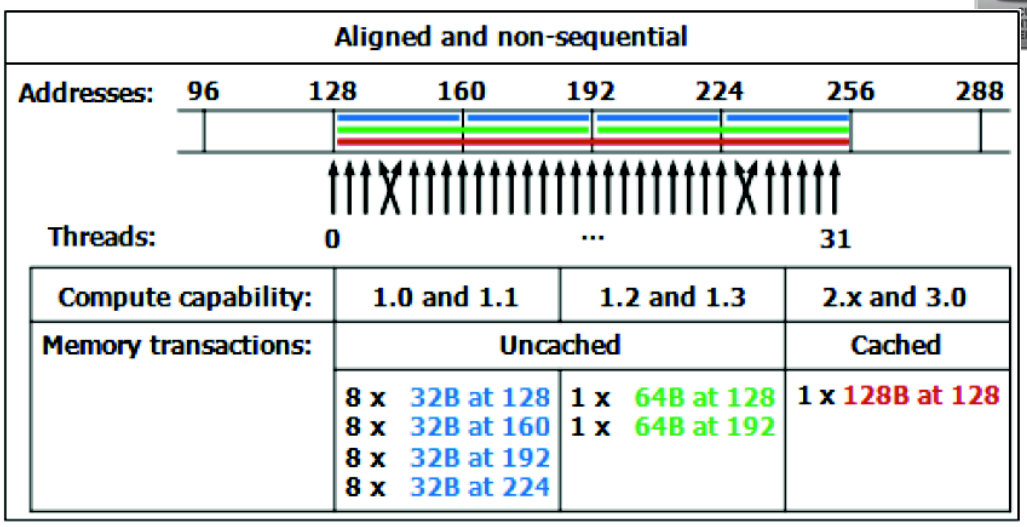
Ausgerichteter, sequentieller Zugriff auf Global Mem.



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Warp-Zugriff, ein 4 Byte Wort pro Thread

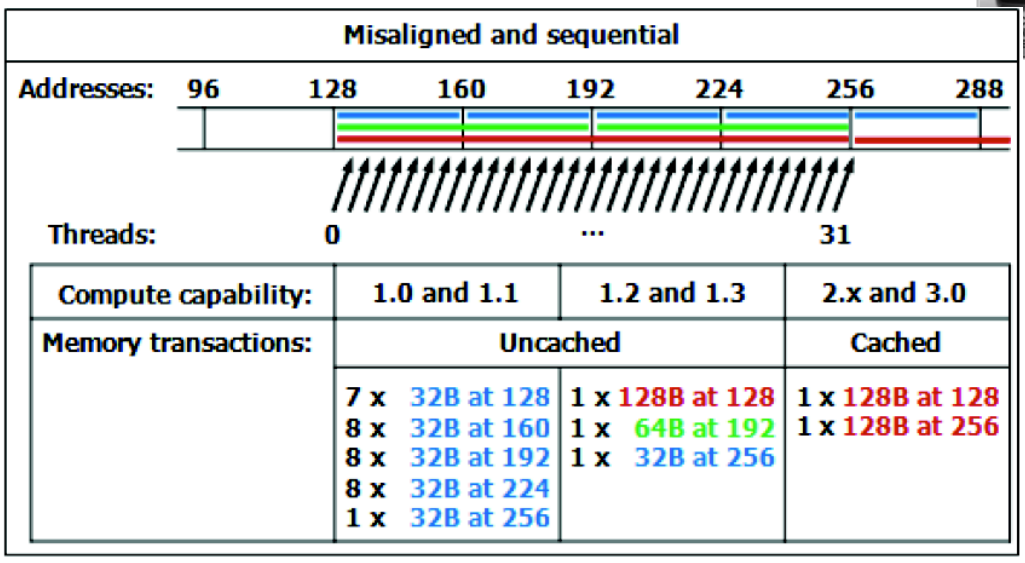
Ausgerichteter, nicht sequentieller Zugriff



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Warp-Zugriff, ein 4 Byte Wort pro Thread

Nicht ausgerichteter, sequentieller Zugriff



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Warp-Zugriff, ein 4 Byte Wort pro Thread

Vor- und Nachteile ausgerichteter Speicherung

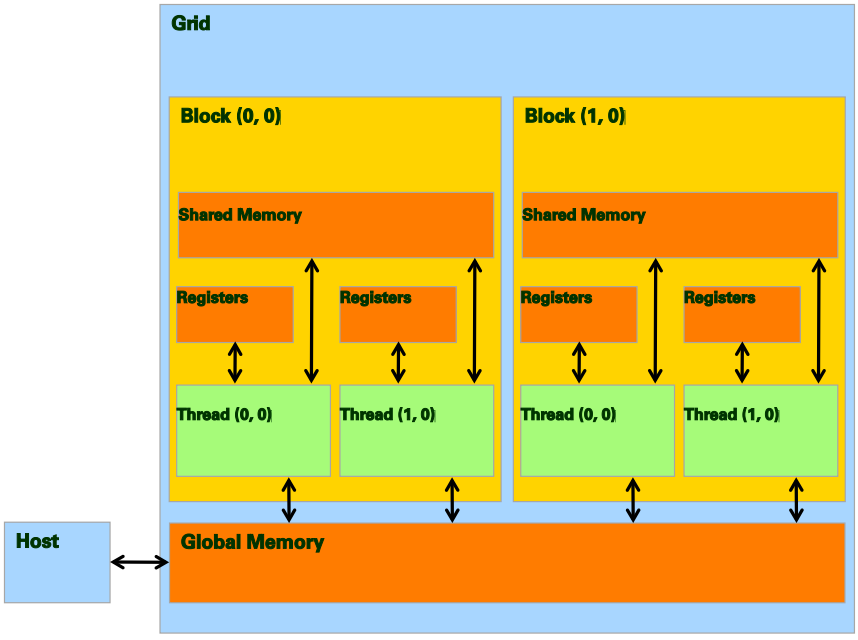
Vorteile

- Ausnutzung theoretischer Speicherbandbreite
- Bessere Ausnutzung L1/L2 Cache
- Weniger Kollisionen beim Speicherzugriff
- Vermeidet Nutzung von Local Memory durch `nvcc`

Nachteile

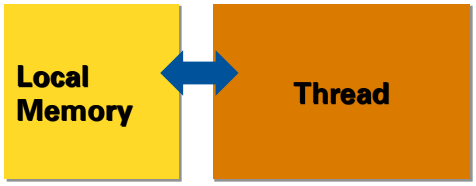
- Umständliche Speicherzugriffe
- Eventuell langsame 2D/3D `cudaMemcopy` host2device/ device2host

Übersicht der Hardware-Speicher inklusive Zugriff



Local Memory: Getarnter Global Memory

Kann Concurrency verhindern! Do not use!



Alles, was nicht in Register passt oder keine definierte Größe hat

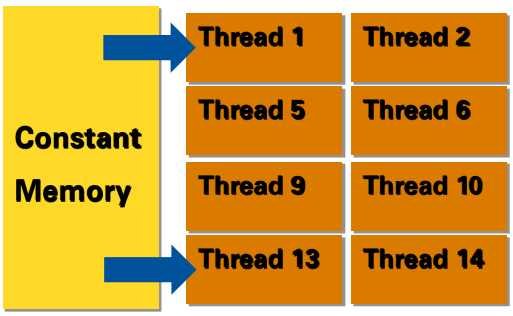
Größe des Local Memory pro Thread

Compute Capability 1.0-1.3: 16kB
Compute Capability 2.x: 512kB

Was?	**Globaler Memory, Pro 32 bit Wort genau ein Thread, coalesced**
Wie?	`__local__ int counter;`
Wer?	**Genau einem Thread zugewiesen, nur dieser kann darauf zugreifen**
Lebensdauer?	**Während der Laufzeit des Threads**
Zugriff?	**Lesen und Schreiben**

Constant Memory: Read-Only Device Memory

Kann Bandbreite sparen



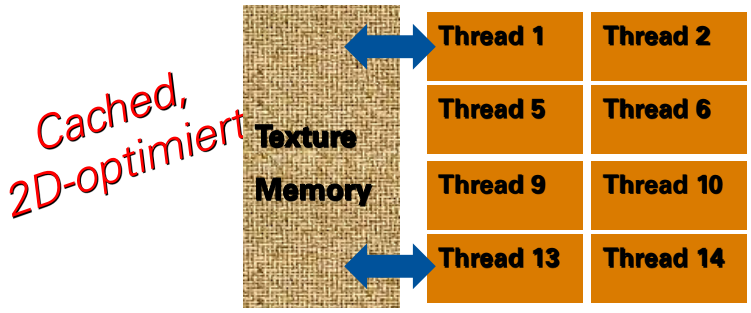
Wenn Du es nicht ändern mußt, geht's schneller.

Größe des Constant Memory

Compute Capability 1.0-2.x: 64kB

Was?	**Device Memory, gecached (8kB), Broadcast (spart Lesezugriffe)**
Wie?	`__constant__ float ReadOnlyData [N];`
Wer?	**Alle Threads auf dem Device, Host**
Lebensdauer?	**Laufzeit des Programms**
Zugriff?	**Nur Lesen**

Texture Memory: Grafischer Speicher für Nostalgiker



Ist fast wie OpenGL oder Direct3D

Derzeit ca. 4 bis 6 GB

Was?	**Im Grunde Global Memory, zugriffsoptimiert, OpenGL/Direct3D**
Wie?	`texture<DataType, TextureType, ReadMode> tex;` `cudaBindTextureToArray(tex, cudaArray);`
Wer?	**Alle Threads auf dem Device, Host**
Lebensdauer?	**Laufzeit des Programms**
Zugriff?	**Nur Lesen**