

# Die GPU als Rechenknecht

am Beispiel von

## CUDA

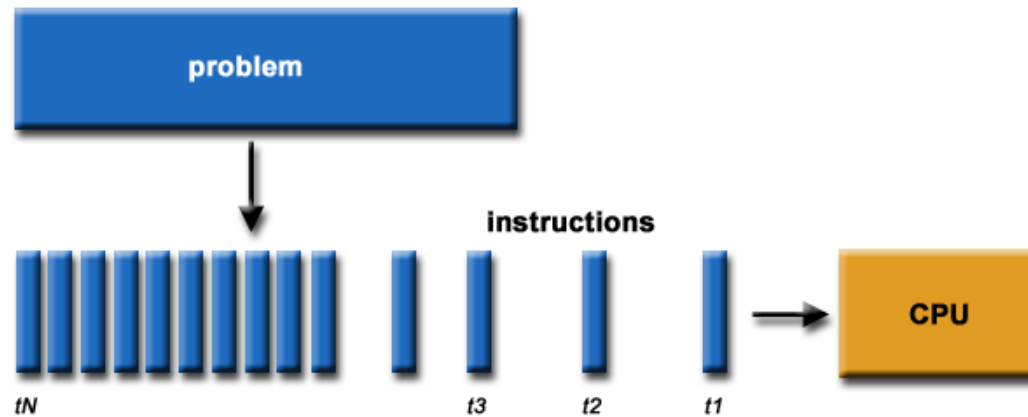
Matthias Schaff  
UnFUG, WS09/10

19.11.09

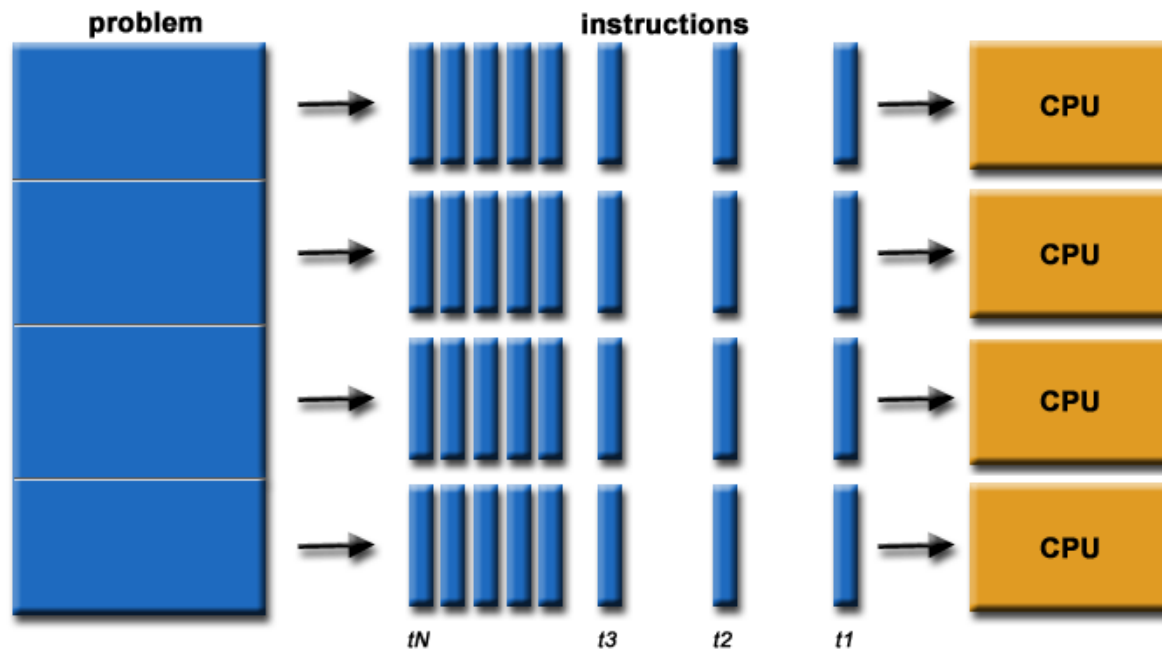
# Was euch erwartet

- Einführung
- CUDA – Programmiermodel
- CUDA – Hardware
- CUDA – Programming
- Beispiele
- Performance

# Parallel Computing

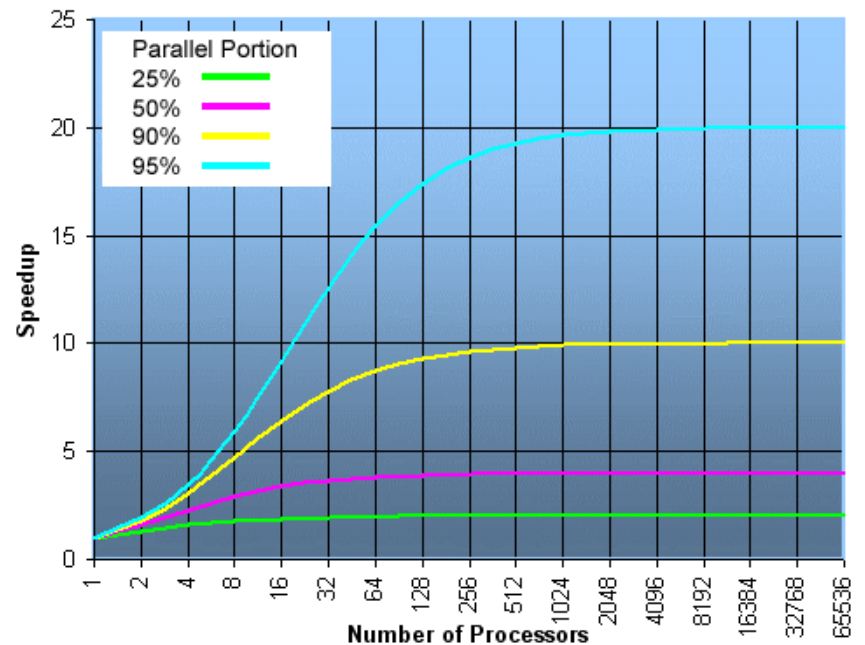
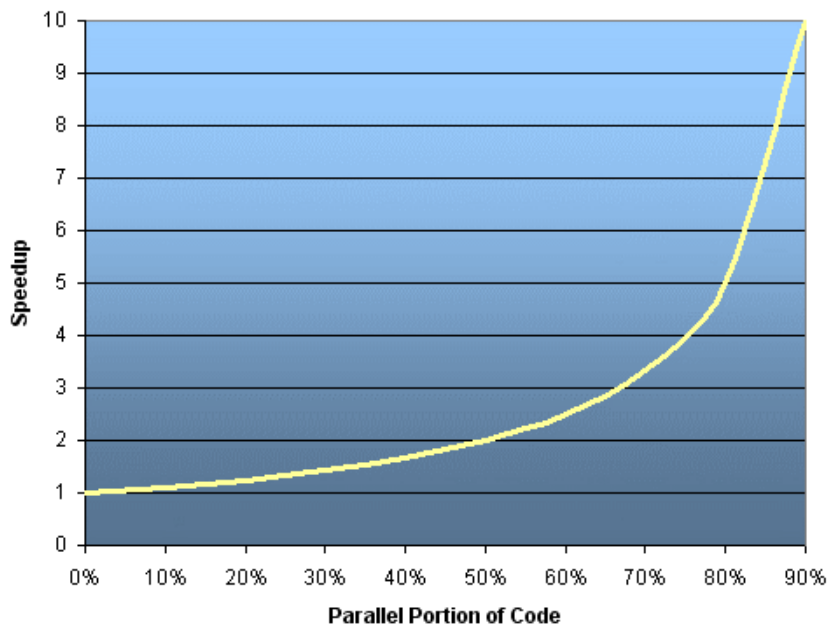


# Parallel Computing



# Parallel Computing

- Zu beachten:
  - Granularität
  - Datenabhängigkeiten
  - Synchronisation
- Amdhal's Law:



# 1996 – First Teraflop Computer

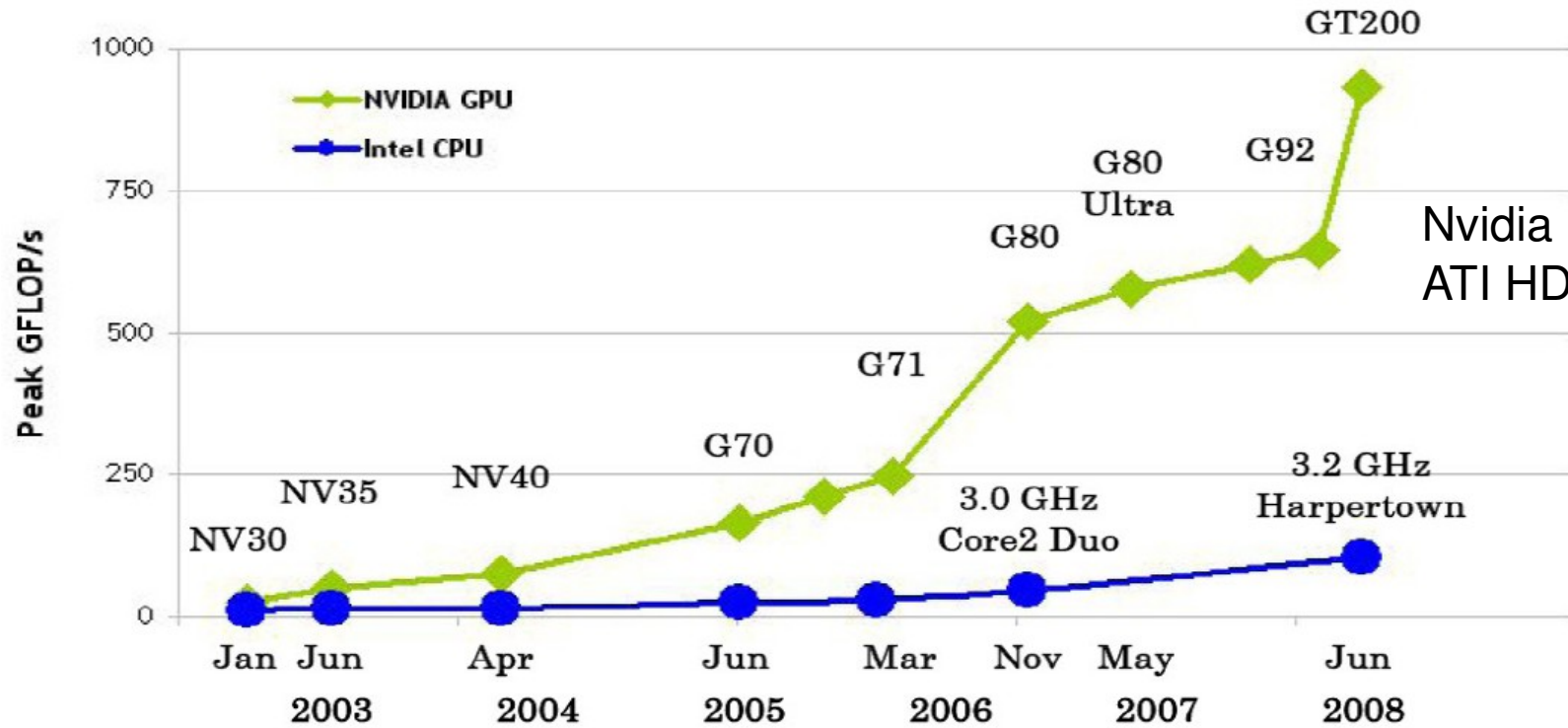
**ASCI Red**, the world's first **TeraFLOP** computer established in 1996, used nearly **10,000** Intel Pentium Pro processors running at 200MHz and consuming **500kW of Power**



It required an additional **500kw** just to keep the room cool!



# Performance Vergleich...



Nvidia Fermi: ca 4 TFLOPS  
 ATI HD 5870: ca 2,6 TFLOPS

GT200 = GeForce GTX 280

G71 = GeForce 7900 GTX

NV35 = GeForce FX 5950 Ultra

G92 = GeForce 9800 GTX

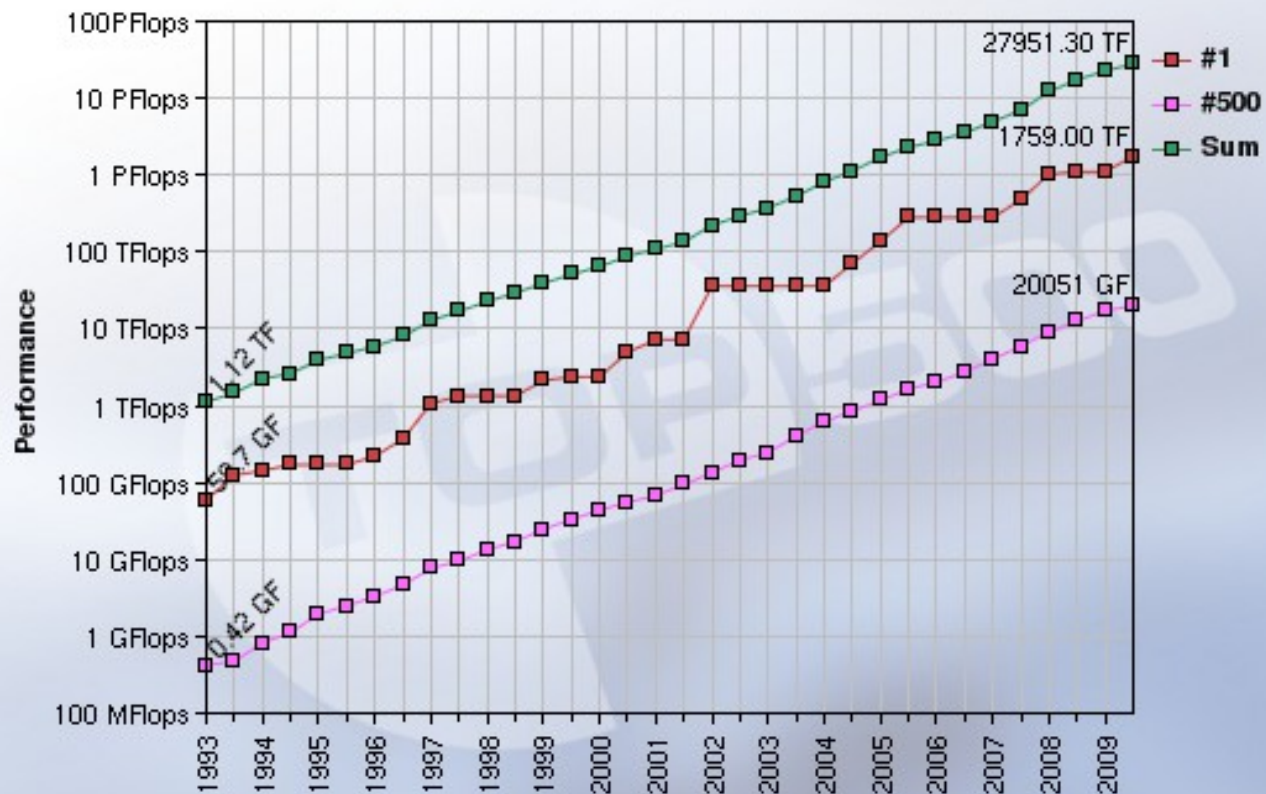
G70 = GeForce 7800 GTX

NV30 = GeForce FX 5800

G80 = GeForce 8800 GTX

NV40 = GeForce 6800 Ultra

## Performance Development



# Graue Theorie

# Was ist CUDA

- Hardware - Software Architektur
- Easy general-purpose computing auf der GPU
- Wie funktioniert das?
  - GPU ist Co-Prozessor
  - Bearbeitet datenparallele, rechenintensive Bereiche
- Vorteil:
  - Frei erhältlich
  - Einfacher Umstieg, da C mit Erweiterungen
  - gute Dokumentation!

# Was braucht CUDA

- GraKa: Nvidia Gforce 8xxxx oder neuer (G80 Chip)
- CUDA – Treiber (ohne X geht nix)
- CUDA SDK
  - NVCC, API, CudaProfiler uvm..
- Ein geeignetes Probelem
- Zeit.....
- Solls schnell sein... noch viel mehr Zeit....

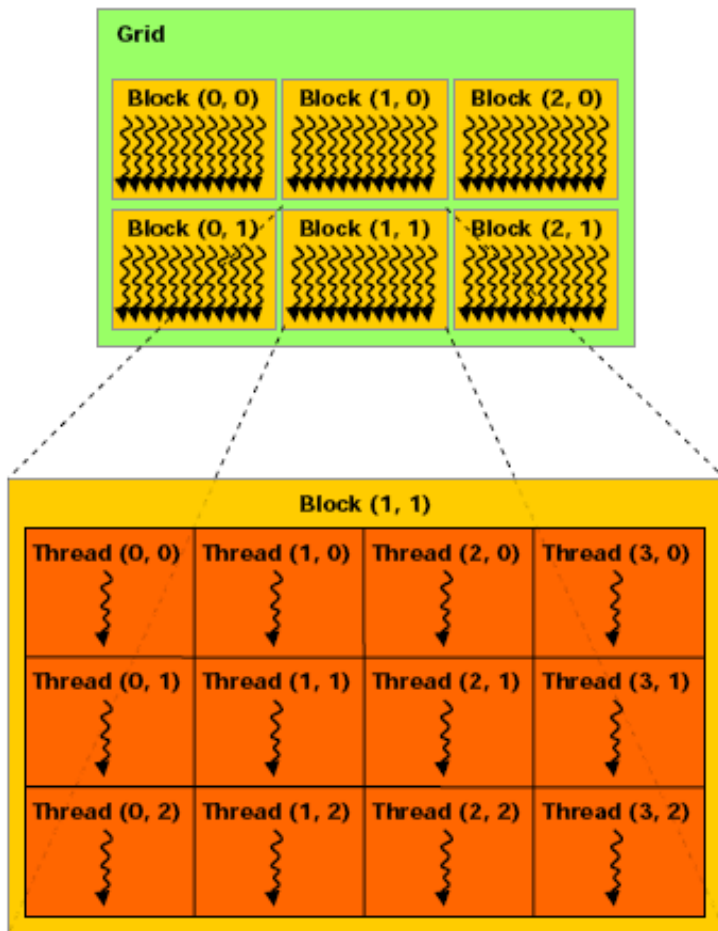
# CUDA Programming Model

- *Kernel*: das Programm auf der Graka, ist eine Funktion:

```
__global__ foo_kernel(int* in, int* out)
```

- Kernel werden in verschiedenen *Threads* gleichzeitig ausgeführt
- getrennte Speicherbereiche → wir müssen Daten kopieren...

# CUDA Programming Model

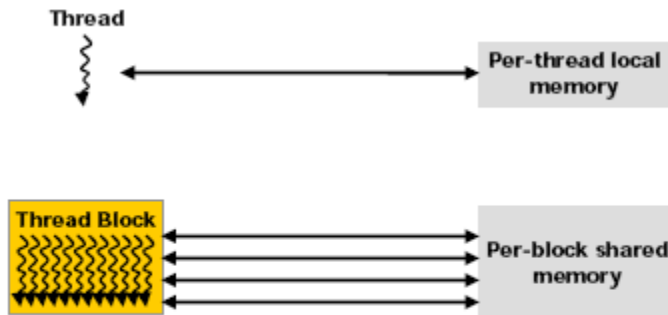


- Wir organisieren unsere Threads in Blocks (3D)
- Wir organisieren unsere Blocks in Grids (2D)
- Jeder Thread kann eindeutig identifiziert werden:

```
gid=blockDim.x*blockId.x  
+threadId.x
```

Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1

# Cuda Memmmory Model



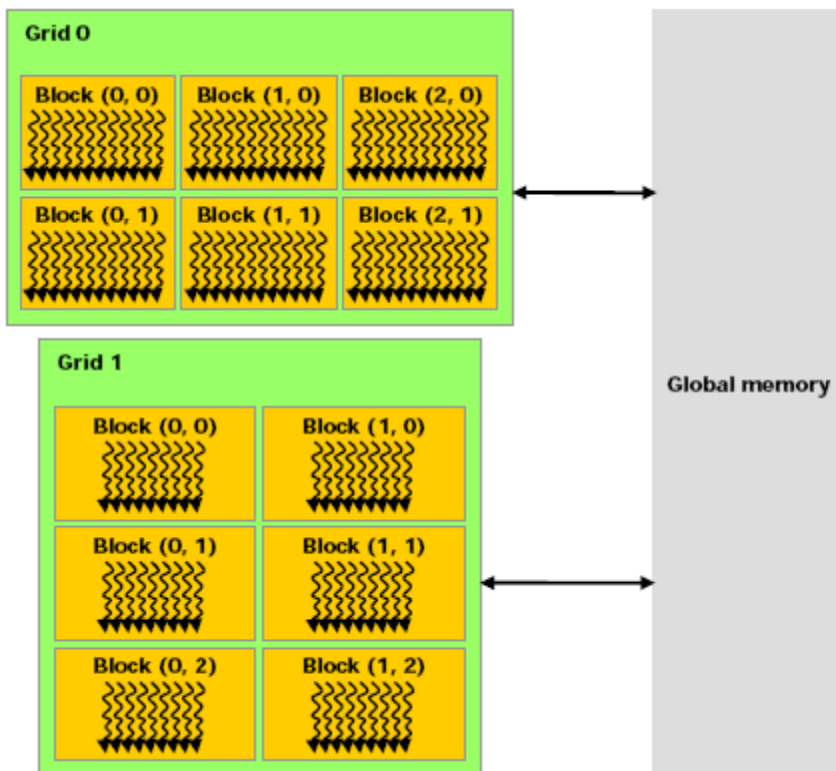
- Thread
  - eigentlich nur Register
  - Local-mem is fake!  
(liegt global, aber nur lokal zugreifbar)

- Shared Mem
  - Cached (1-2 Cycles)
  - 16KiB pro Block (CUDA 1.3)

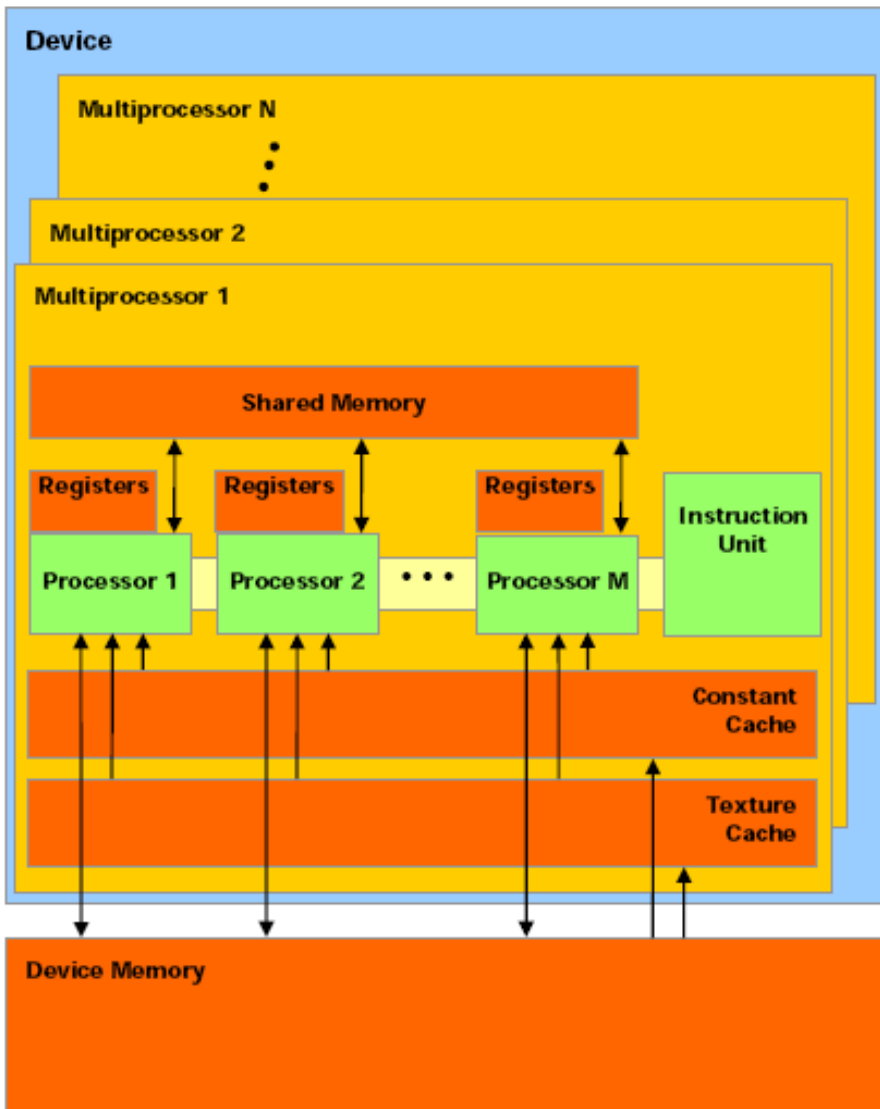
- Global Mem
  - langsam (bis 600 Cycles)
  - 512-..... MiB groß
  - Off-chip

- Trotzdem noch schnell...

Global zu Global ca 93 GB/s!

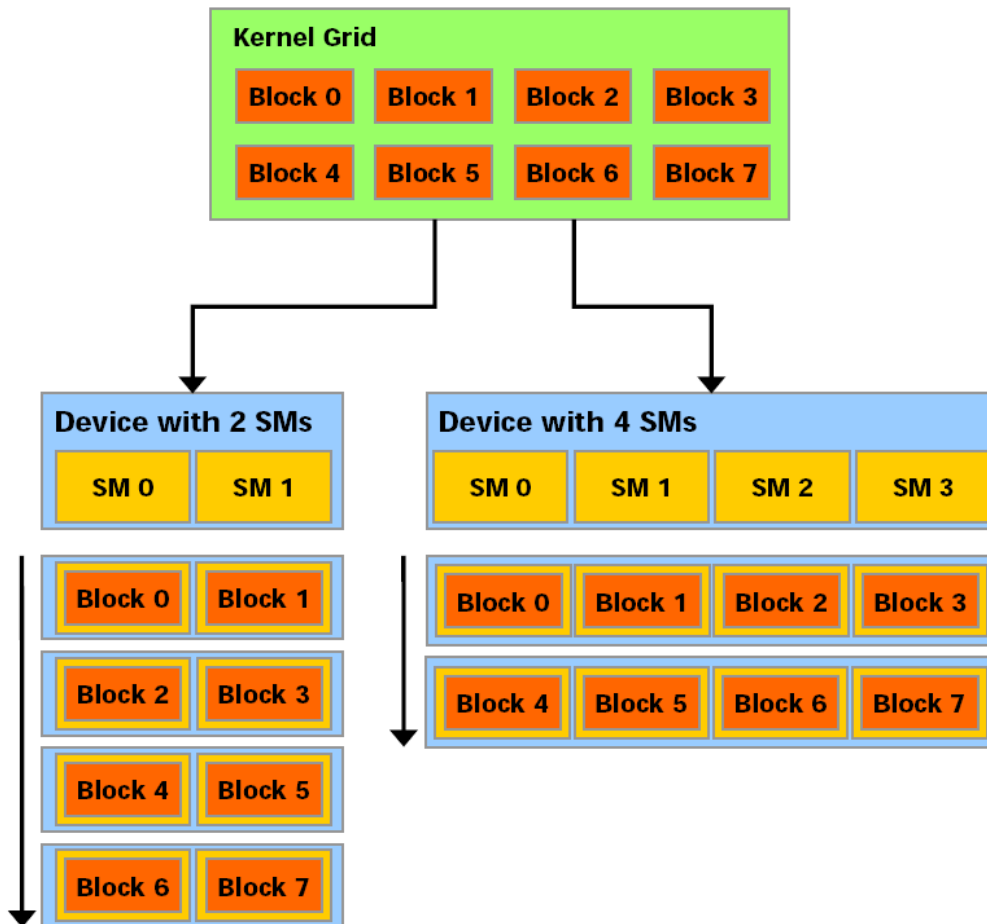


# Cuda Hardware



- Streaming Multiprocessors (SMs)
  - 8 Scalar Processors (SPs)
  - 16 KiB Shared Mem
  - 64KiB Constant Mem
  - 16Ki 32-Bit Register (!!!!!)
  - Scheduling Einheit

# Block/Thread Mapping auf HW



- Blocks werden dynamisch auf die HW verteilt → skaliert!
- Terminiert ein Block, wird der nächste geschedule
- Threads laufen in SPs
- Warps kleinste scheduling Einheit (32 Threads)
- 1 SM verarbeitet 24 Warps (3 pro SP)

→ **768 concurrent Threads pro SM!**

# CUDA Programming

- Wir müssen trennen:
  - Host: alles was auf der CPU läuft
  - Device: alles was in der Graka passiert
- Mehrere Speicherbereiche und Prozessoren forder Deklaration
  - Function type qualifiers:
    - `__device__`, `__global__`, `__host__`
  - Variable type qualifiers:
    - `__device__`, `__constant__`, `__shared__`
- Kernelaufrufe haben besondere Notation (vom Host aus..)
  - `foo<<dimGrid, dimBlock, sharedMem>>(…)`
- Build-in Variabeln für Blockgröße, Block-ID, Gridgröße usw..

# CUDA Programming: Speicher

- Allokieren von globalem Speicher:
  - Host: `cudaMalloc(void** devPtr, size_t count);`
  - Device: `__device__ int a;`
- Shared Mem
  - Device: `__shared__ int data[2];`
  - Kernelparameter
- Kopieren:
  - `cudaMemcpy(void* dst, const void* src, size_t n, cudaMemcpy<von>To<nach>)`
  - `von,nach ∈ {Host, Device}`

# Cuda Programming: Bsp

```
int main(void)
{
    float*a_h, *b_h;    // pointers to host memory
    float*a_d;         // pointer to device memory
    intN = 1000;
    size_t size = N*sizeof(float);

    // allocate arrays on host
    a_h= (float*)malloc(size);
    b_h= (float*)malloc(size);

    // allocate array on device
    cudaMalloc((void*)&a_d, size);

    // initialize data

    // copy data from host to device
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);

    foo_kernel<<<gridDim, blockDim>>>(a_d);

    // retrieve result from device and store it in b_h
    cudaMemcpy(b_h, a_d, size, cudaMemcpyDeviceToHost);

    // cleanup
    free(a_h);
    free(b_h);
    cudaFree(a_d);
}
```

Hands on!



# Bsp: Matrizen elementweise addieren

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

# Bsp: Matrizen elementweise addieren

```
void add_matrix( float* a, float* b, float* dest, int N ) {
    int index;
    for ( int i = 0; i < N; ++i )
        for ( int j = 0; j < N; ++j ) {
            index = i + j*N;
            dest[index] = a[index] + b[index];
        }
}

int main() {...
    add_matrix( a, b, dest, N );
...}
```

**N\*N**  
**Iterationen!**

```
__global__ add_matrix ( float* a, float* b, float* dest, int N ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if ( i < N && j < N )
        dest[index] = a[index] + b[index];
}

int main() {...
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( a, b, dest, N );
...}
```

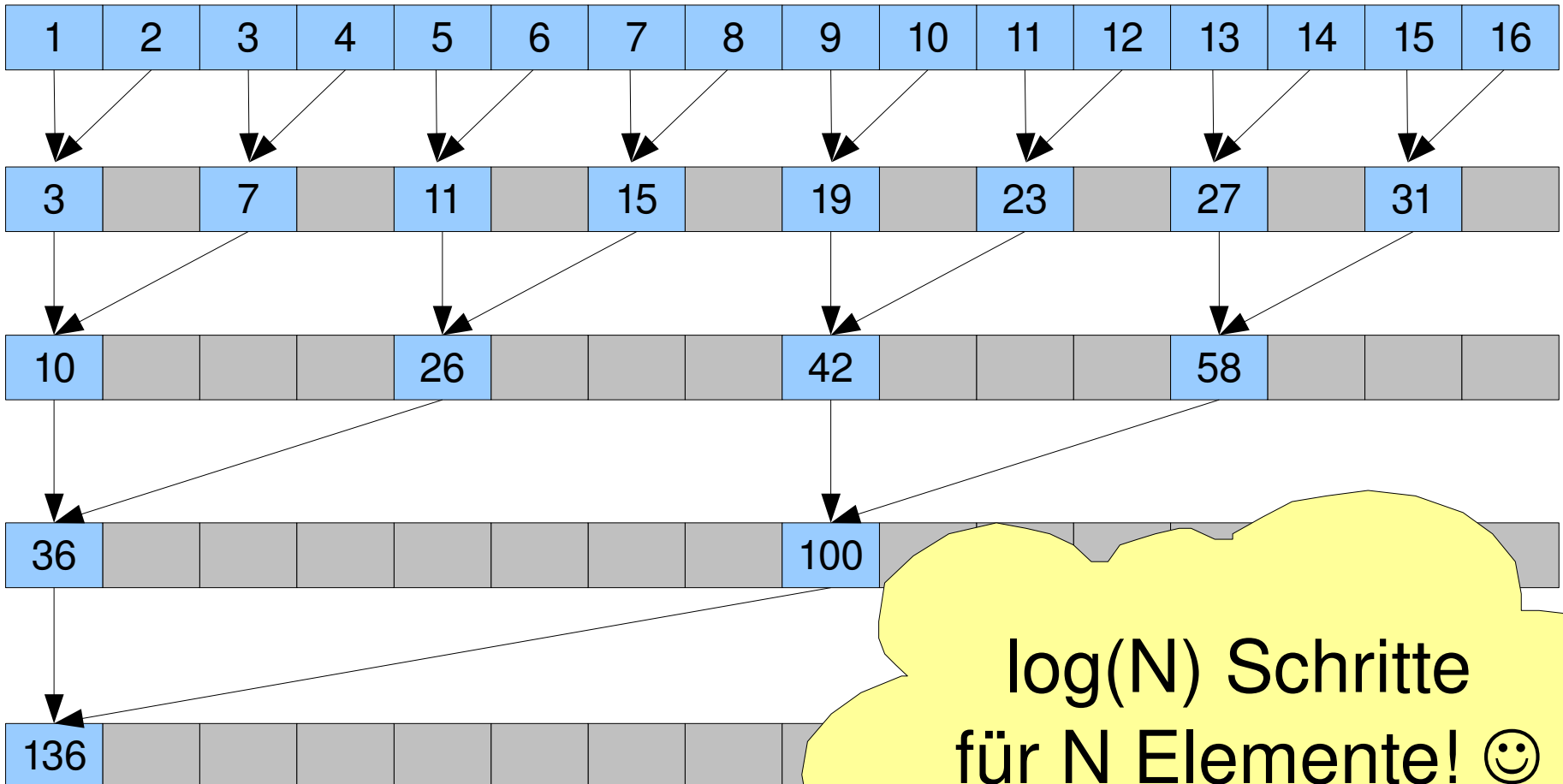
**Iterationen?**

# Summe über Array

- Es soll die Summe über ein N elementiges Array gebildet werden
- In CPU-Code... klar...
  - Schleife über alle Elemente, Summe mitführen
  - N Schritte für N Elemente ☹️
- In GPU...
  - Parallelität nutzen die Idee... aber wie?

**Parallel Sum Reduction!**

# Parallel Sum Reduction



**log(N) Schritte  
für N Elemente! 😊**

20 Schritte = 1048576 Elemente

# Parallel Sum Reduction in CUDA

```
__global__ void
Sum_Kernel( int* g_idata, int* g_odata)
{
    extern __shared__ int data[];
    int tid=threadIdx.x;
    int bdim=blockDim.x;
    int gid=blockIdx.x*bdim+tid;
    data[tid]=g_idata[gid];
    // do reduction in shared mem
    for(unsigned int s=1; s <bdim; s *= 2) {
        if (tid % (2*s) == 0) {
            data[tid] += data[tid + s];
        }
        __syncthreads();
    }

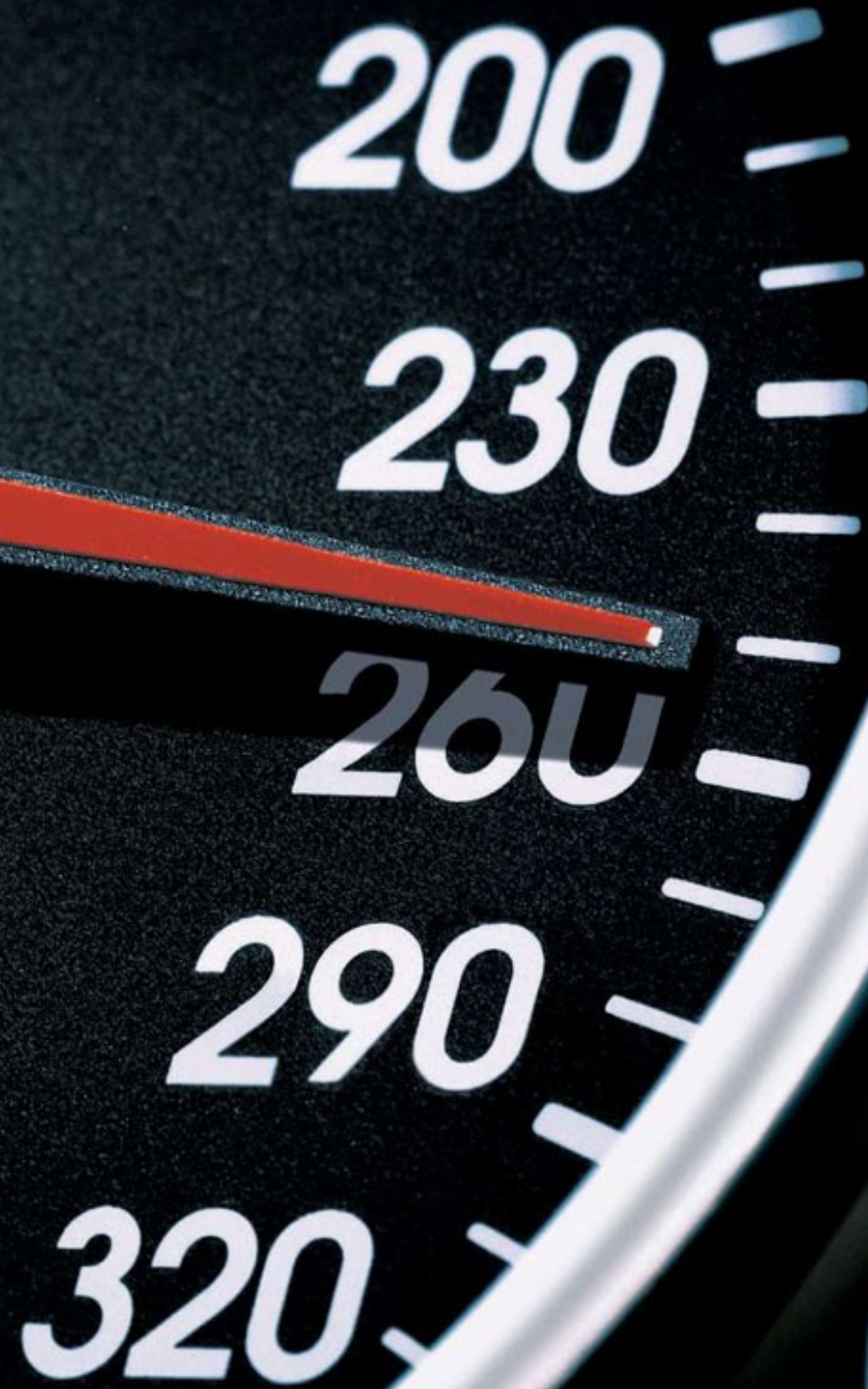
    if(tid==0)
        g_odata[bid]=data[0];
}
```

Jeder Thread kopiert  
sein Element  
in Sharedmem

Fiese Schleife:  
macht die 2er  
Potenzen

Deaktiviert Threads

blockweit synchronisieren  
wegen Sharedmem

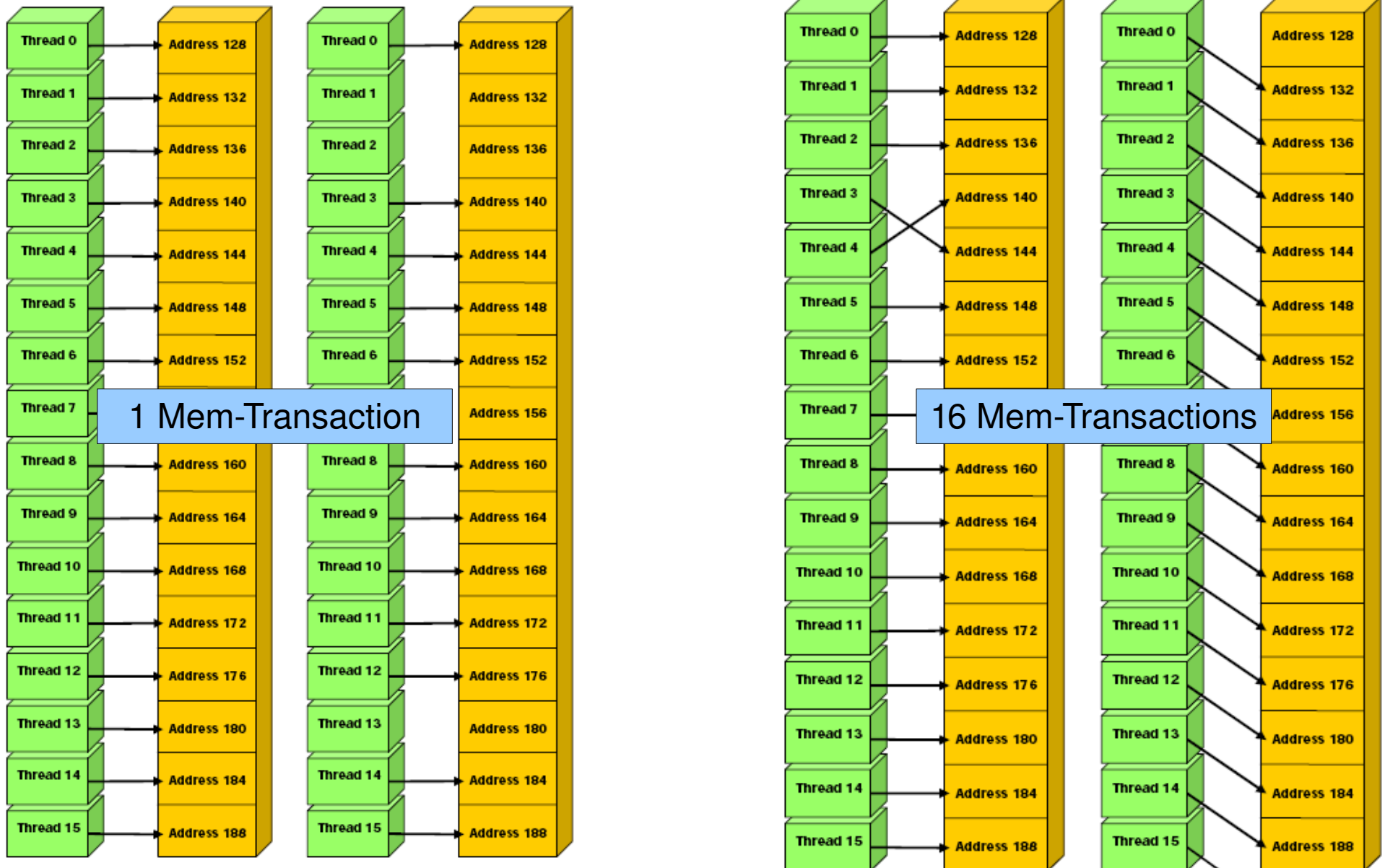


**How to get  
reeeeeeeally  
fast.....**

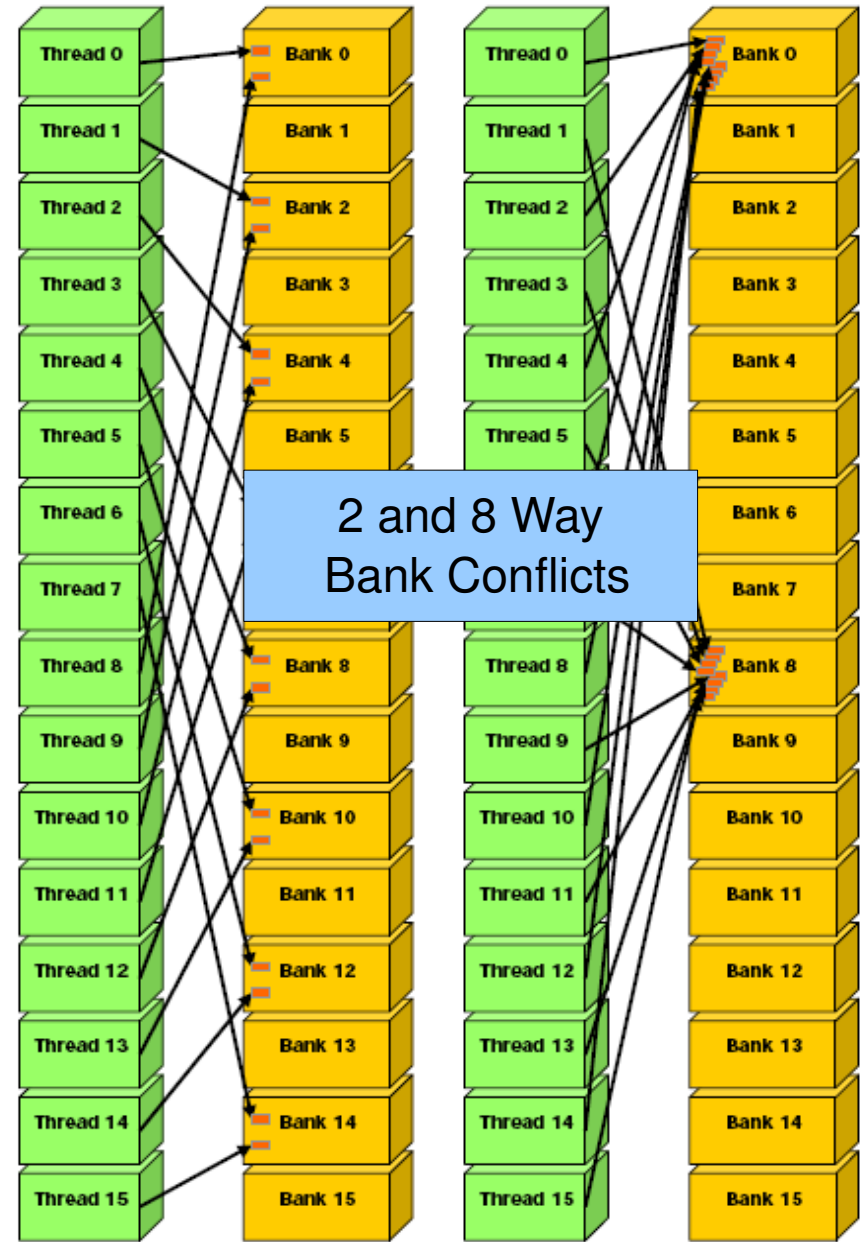
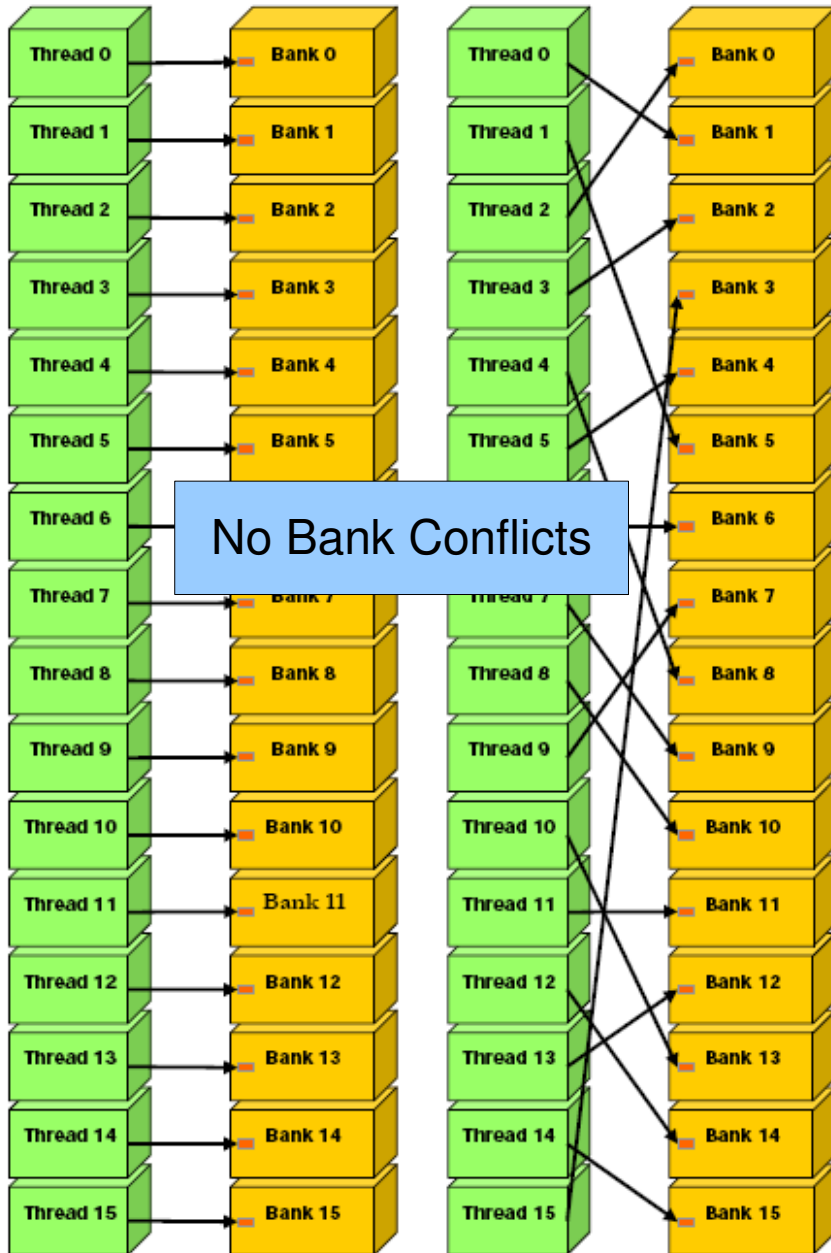
# Performance

- Nur schnell, wenn Architektur bei Algorithmus u. Implementierung beachtet!
  - Warps sollten keine datenabhängigen Programmfluss haben, sonst serialisierte Ausführung!
  - Speicherzugriffe sollten in 32, 64 oder 128bit sein!
  - Warps in Threads sollten *coalesced* (=vereinigt) auf Globalmem zugreifen, sonst serialisiert!
    - Der k. Thread sollte auf der k. Element zugreifen
  - Shared Mem zugriffe nach Memory Banks ausrichten.. sonst... serialisiert!

# Coalesced Global Mem Access

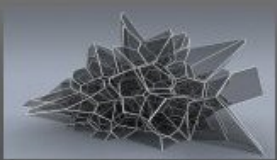

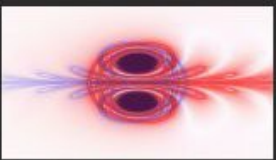
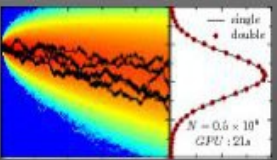
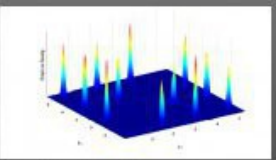


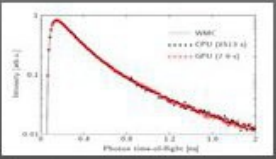
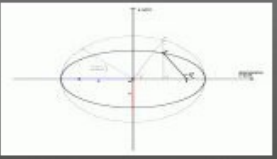



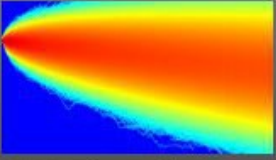
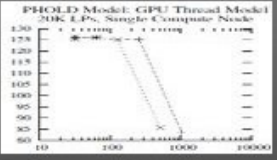
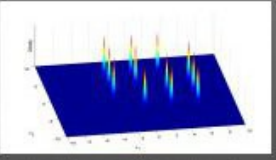


# Bank Conflicts



# Performance

- „Easy to get startet - hard to reach peak Performance!“
- Problem und Implementierung müssen genau passen!
  - Ist das der Fall, dann gehts ab...

 <p>GPU-based Acceleration of the Genetic Algorithm 2600 x</p>	 <p>Computer Generated Hologram on GPU - Simple color electroholography 1500 x</p>	 <p>Optimization of FTLE Calculation 1000 x</p>	 <p>Stochastic Differential Equations with CUDA 675 x</p>	 <p>On the utility of graphics cards to perform massively parallel simulation of 500 x</p>
 <p>Real Time Elimination of Undersampling Artifacts in CE MRA using Variational 2300 x</p>	 <p>Fast Total Variation for Computer Vision 1000 x</p>	 <p>Parallel computing with graphics processing units for high-speed Monte Carlo 1000 x</p>	 <p>Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units: Applica 600 x</p>	 <p>Solving Kinetic Equations on GPUs I: Model Kinetic Equations 500 x</p>
 <p>Implementation of a Lattice-Boltzmann method for numerical fluid mechani 1840 x</p>	 <p>Fast and Exact Solution of Total Variation Models on the GPU 1000 x</p>	 <p>Accelerating numerical solution of Stochastic Differential Equations with 675 x</p>	 <p>AN APPROACH FOR THE EFFECTIVE UTILIZATION OF GP-GPUS IN PARALLEL 539 x</p>	 <p>Massively Parallel Population-Based Monte Carlo Methods 500 x</p>

<EOF>