Programmierung von Graphikkarten

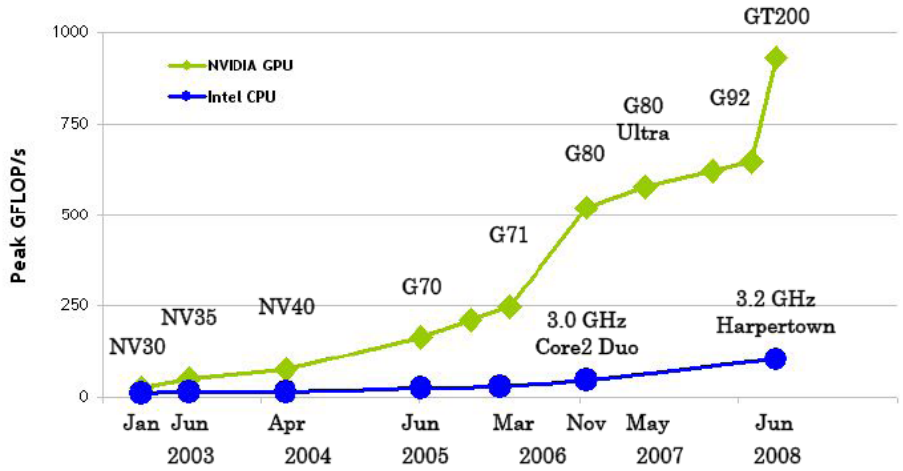
Stefan Lang

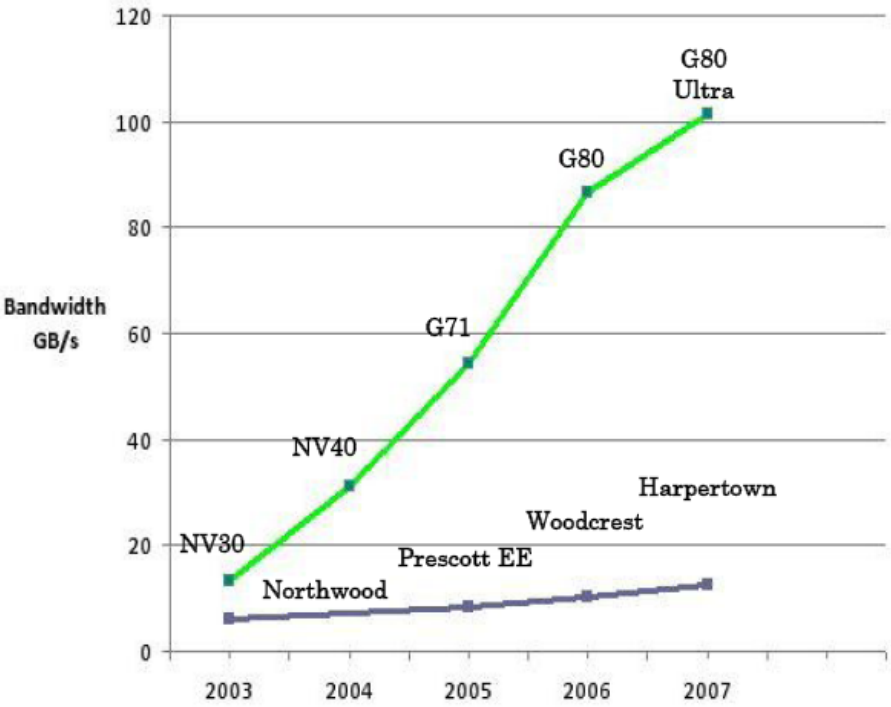
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 13/14

Motivation

- Entwicklung von Graphikprozessoren (GPU) ist dramatisch:





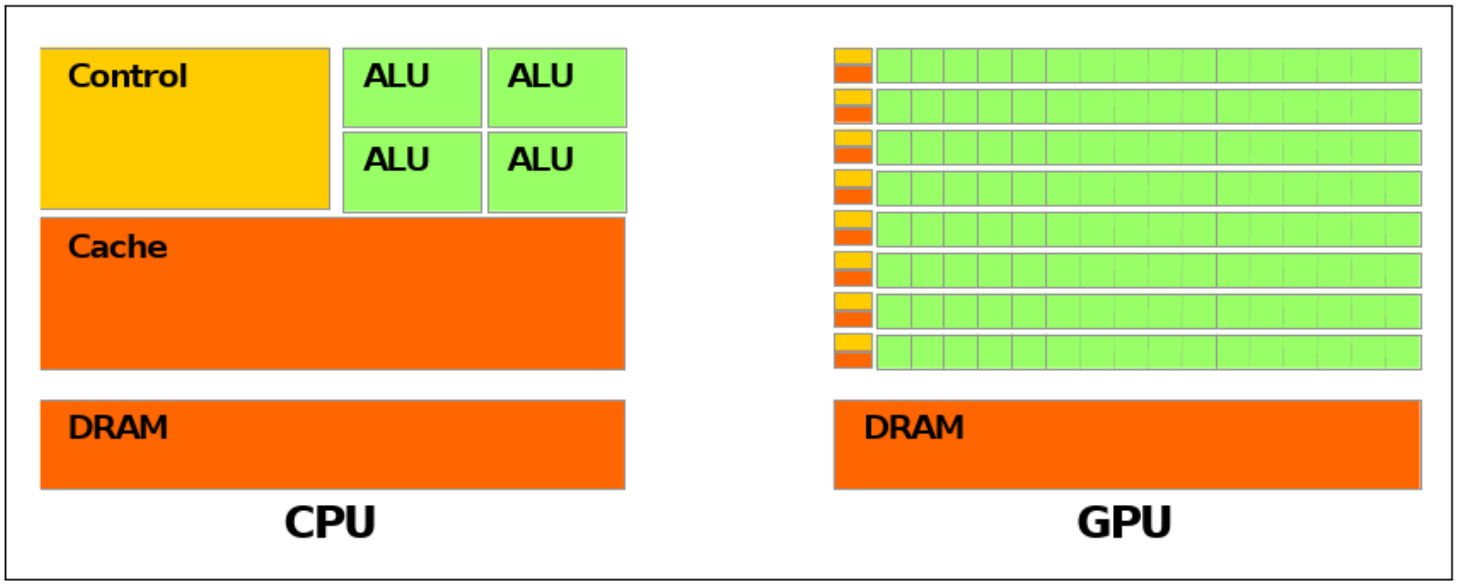
- GPUs sind hochparallele Prozessoren!
- **GPGPU computing**: Verwende GPUs für paralleles Rechnen.

GPU - CPU Vergleich

	Intel QX 9770	NVIDIA 9800 GTX
Since	Q1/2008	Q1/2008
Cores	4	16×8
Transistors	820 Mio	754 Mio
Clock	3200 MHz	1688 MHz
Cache	4×6 MB	16×16 KB
Peak	102 GFlop/s	648 GFlop/s
Bandwith	128 GB/s	70.4 GB/s
Price	1200 $	150 $

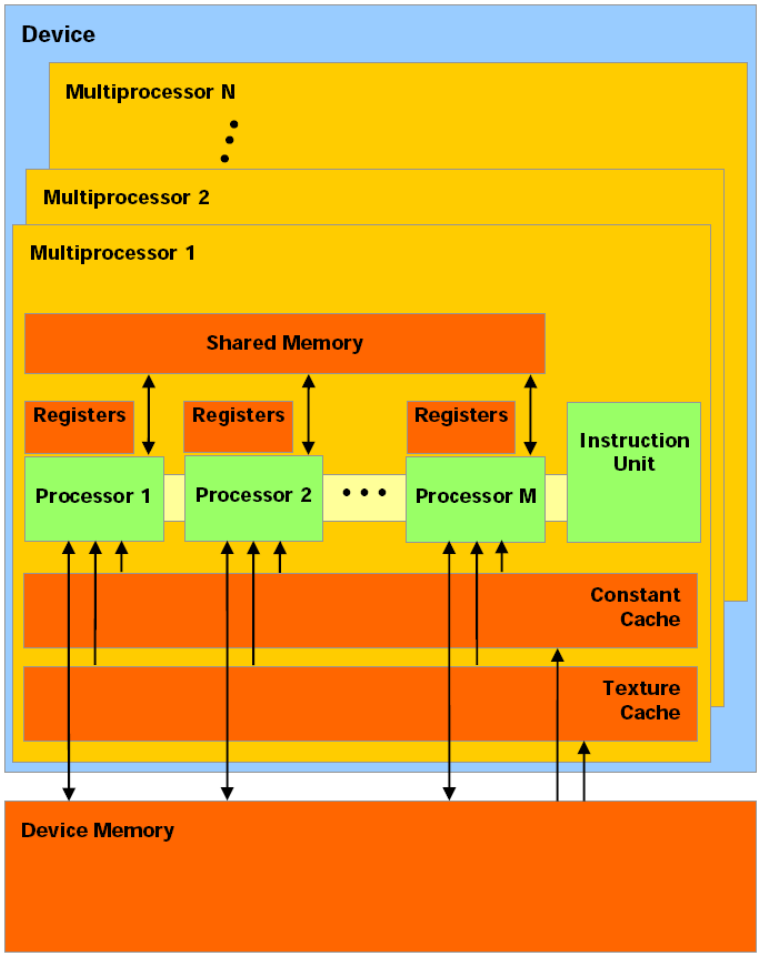
Letztes Modell GTX 280 hat 30×8 cores und eine Peakleistung von 1 TFlop/s.

Chiparchitektur: CPU vs. GPU



GPU weitaus mehr Transistoren für Datenverarbeitung, dafür weniger Transistoren für Cache

Hardware auf einen Blick

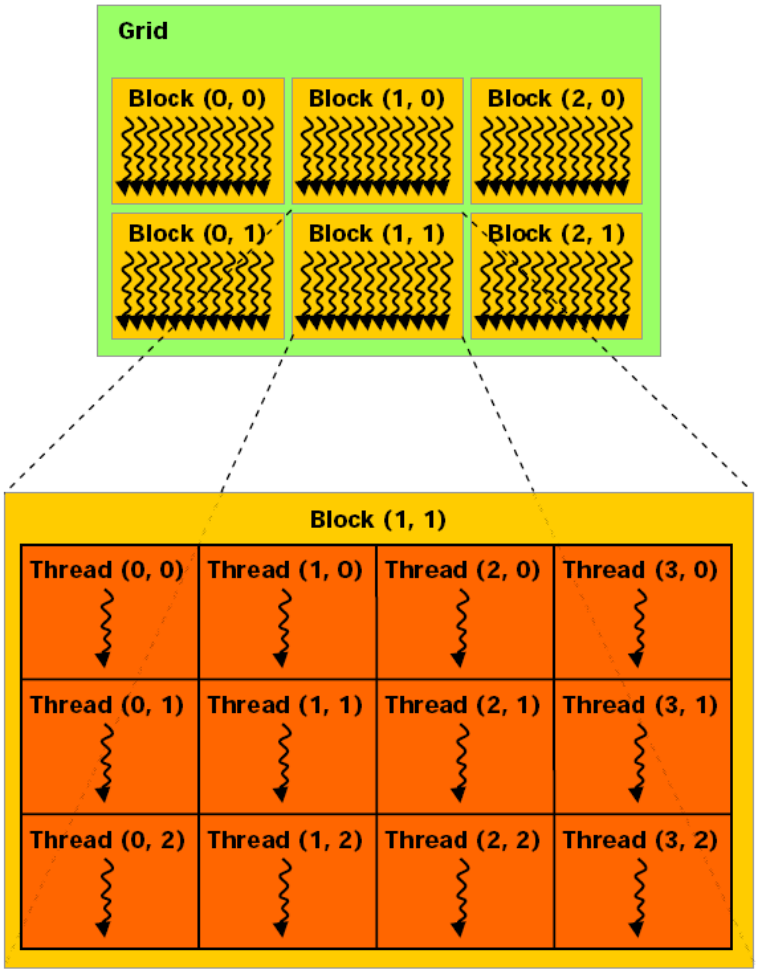


- Ein Multiprozessor (MP) besteht aus $M = 8$ “Prozessoren”.
- MP hat eine Instruktionseinheit und 8 ALUs. Threads welche verschiedene Instruktionen ausführen werden serialisiert!
- 8192 Register pro MP, werden auf Threads zur Kompilerzeit aufgeteilt.
- 16 KB shared memory pro MP, organisiert in 16 Bänken.
- Bis zu 4GB global memory, Latenzzeit 600 Taktzyklen, Bandbreite bis zu 80 GB/s .
- Konstant- und Texturspeicher wird gecached und ist read-only.
- Graphikkarten liefern hohe Leistung bei Arithmetik mit einfacher Genauigkeit, doppelte Genauigkeit geringere Leistung.

CUDA

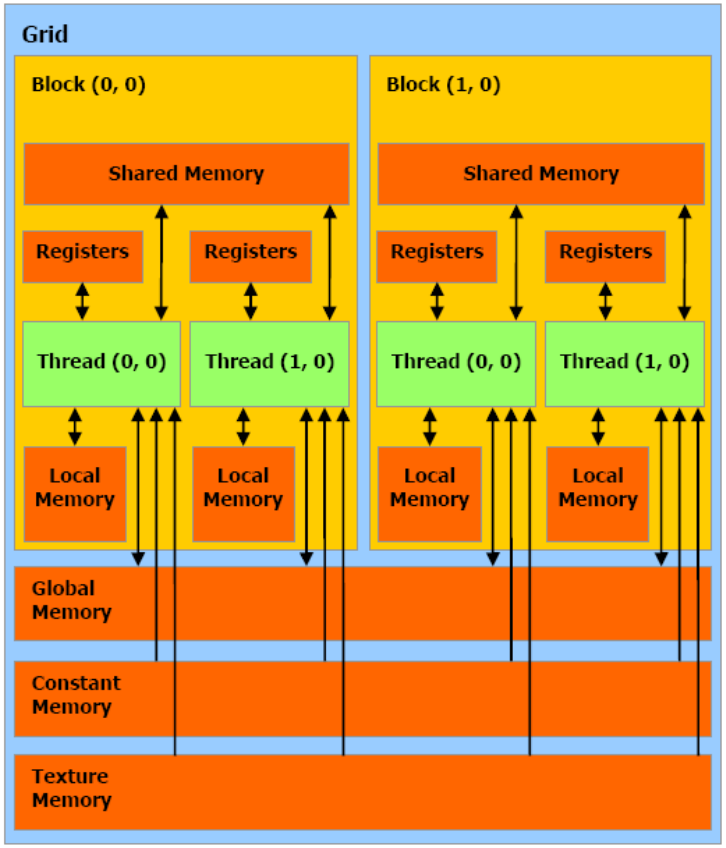
- Steht für **Compute Unified Device Architecture**
- Skalierbares Hardwaremodell mit z.B. 4×8 Prozessoren in einem Notebook und 30×8 Prozessoren auf einer High-End Karte.
- C/C++ Programmierumgebung mit Spracherweiterungen. Spezieller Compiler `nvcc`.
- Der auf der GPU ausführbare Code kann nur in C geschrieben sein.
- Laufzeitumgebung und verschiedene Anwendungsbibliotheken (BLAS, FFT).
- Extensive Menge von Beispielen.
- Koprozessor Architektur:
  - Einige Codeteile laufen auf der CPU die dann Code auf der GPU anstösst.
  - Daten müssen explizit zwischen CPU und GPU Speicher kopiert werden (kein direkter Zugriff).

Programmiermodell auf einen Blick



- Parallele Threads kooperieren über gemeinsame Variablen.
- Threads sind in Blöcken von einer “wählbaren” Größe.
- Blöcke können 1-, 2- oder 3-dimensional sein.
- Blöcke sind in einem Gitter mit variable Größe organisiert.
- Gitter können 1- or 2-dimensional sein.
- # Threads ist typisch größer # cores (“hyperthreading”).
- Blockgröße wird bestimmt durch HW/Problem, Gittergröße bestimmt von Problemgröße.
- Kein Overhead durch Kontextwechsel.

Speicherhierarchie



- pro Thread
  - Register
  - Local memory (ungecacht)
- pro Block
  - Shared memory
- pro Grid
  - Global memory (ungecacht)
  - Constant memory (read-only, gecacht)
  - Texturspeicher (read-only,gecacht)

Beispiel eines Kernel

```
1 __global__ void scale_kernel (float *x, float a)
  {
3   int index = blockIdx.x*blockDim.x + threadIdx.x;
    x[index] *= a;
5 }
```

- `__global__` Funktionstypbezeichner qualifiziert diese Funktion zur Ausführung auf dem Device und kann nur vom host (“kernel”) aufgerufen werden.
- Built-in Variable `threadIdx` enthält Position des Threads innerhalb des Blocks.
- Built-in Variable `blockIdx` speichert Position des Blocks innerhalb des Gitters.
- Built-in variable `blockDim` liefert die Größe des Blocks.
- Built-in Variable `gridDim` enthält Dimension des Gitters
- In obigem Beispiel ist jeder Thread verantwortlich um ein Element des Vektors zu skalieren.
- Die Gesamtzahl eines Threads muß and die Größe des Vektors angepaßt werden.

Ausführung und Leistungsaspekte

- Divergenz: Volle Leistung kann nur erreicht werden wenn alle Threads eines warps eine identische Anweisung ausführen.
- Threads werden in *warps* von 32 Threads geschedult.
- Hyperthreading: Ein MP sollte mehr als 8 Threads zu einer Zeit (empfohlene Blockgröße ist 64) ausführen, um Latenzzeit zu verdecken.
- Shared memory Zugriff benötigt 2 Taktzyklen.
- Schnellste Instruktionen sind 4 Zyklen (e.g. single precision multiply-add).
- Zugriff zum shared memory ist nur schnell falls jeder Thread auf eine andere Bank zugreift, ansonsten werden die Bankzugriffe serialisiert.
- Zugriff zum global memory kann durch Zusammenfassung des Zugriffs zu alignierten Speicherstellen beschleunigt werden. Benötigt spezielle Datentypen, z.B. `float4`.

Synchronisation / Branching

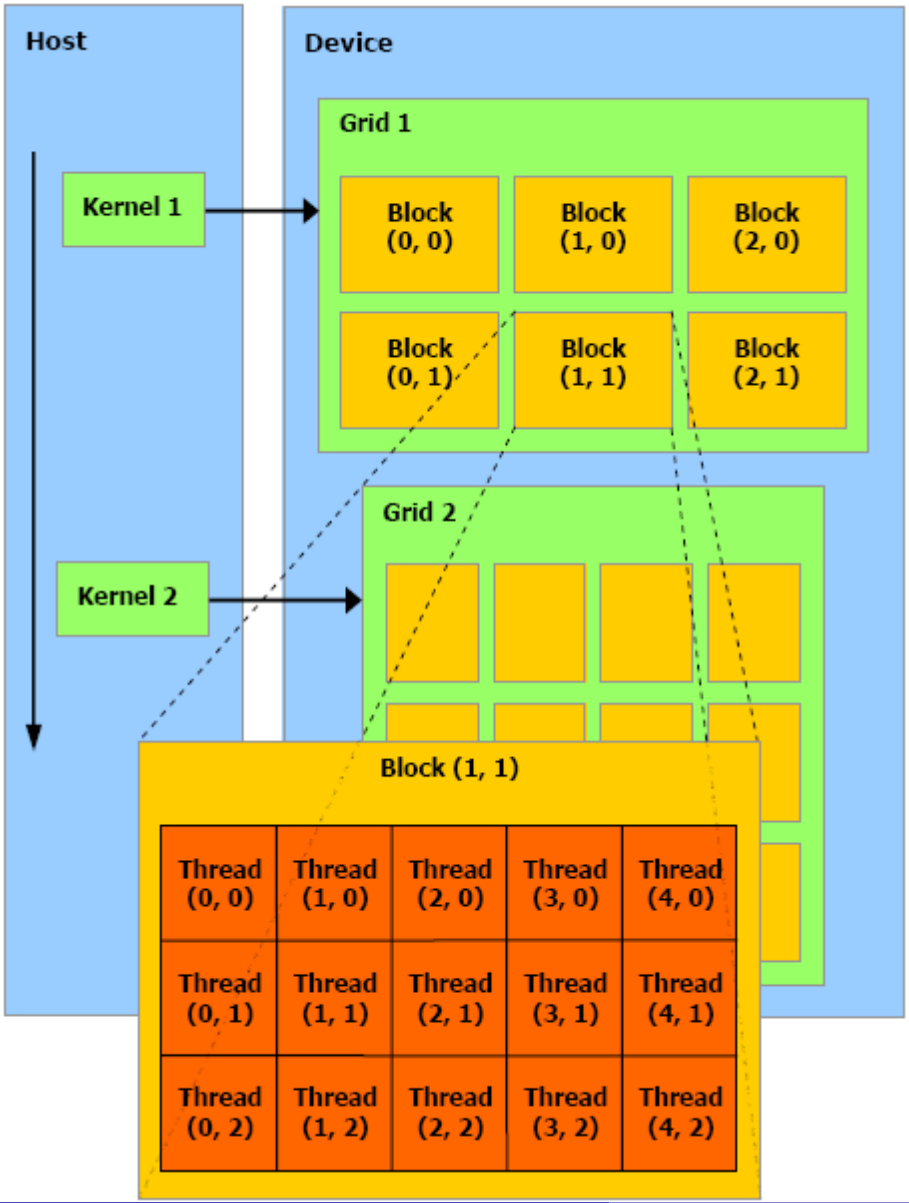
Synchronisation

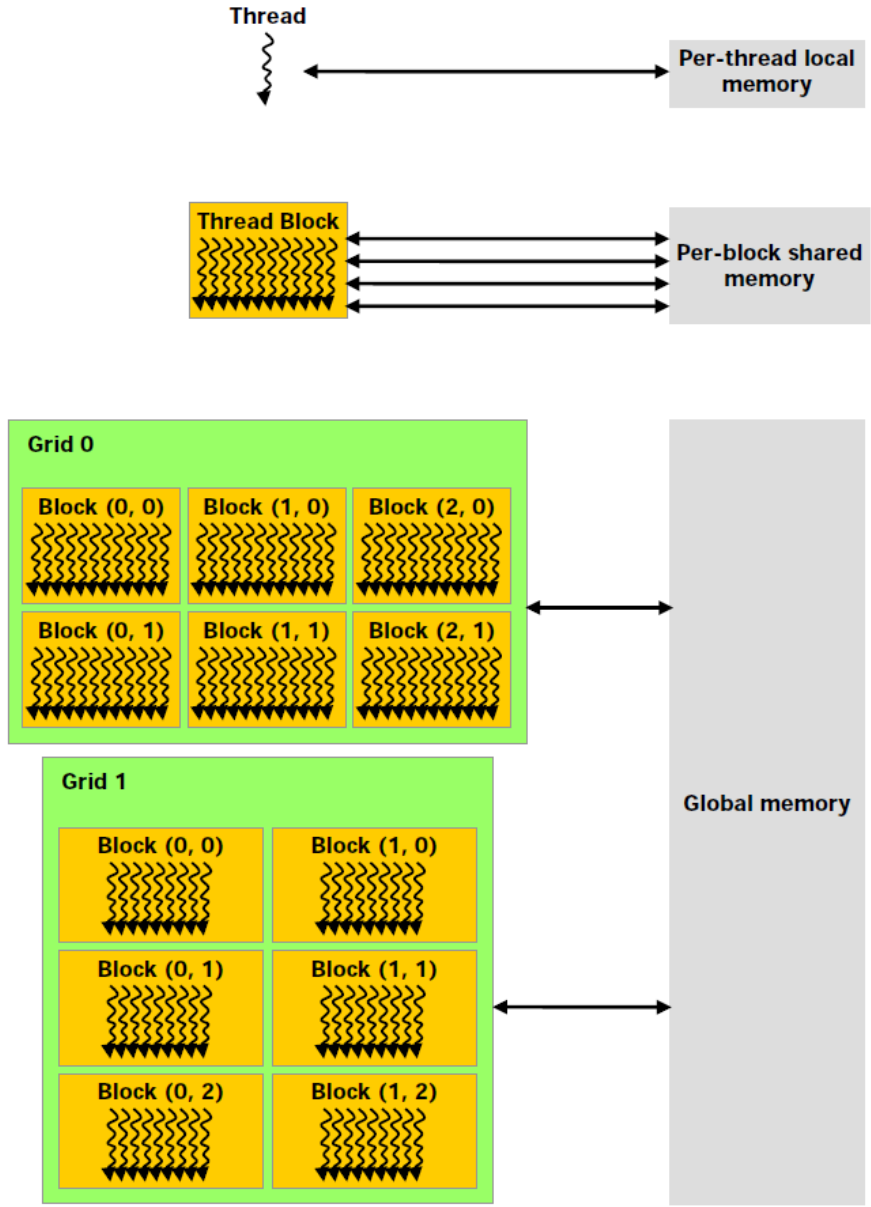
- Synchronisation mit Barriere auf Blockebene.
- keine Synchronisationsmechanismen zwischen Blöcken.
- Aber: Kernelaufrufe sind billig, können zur Synchronisation zwischen Blöcken verwendet werden.
- Atomare Operationen (nicht alle Modelle ab Compute Capability 1.1).

Branching

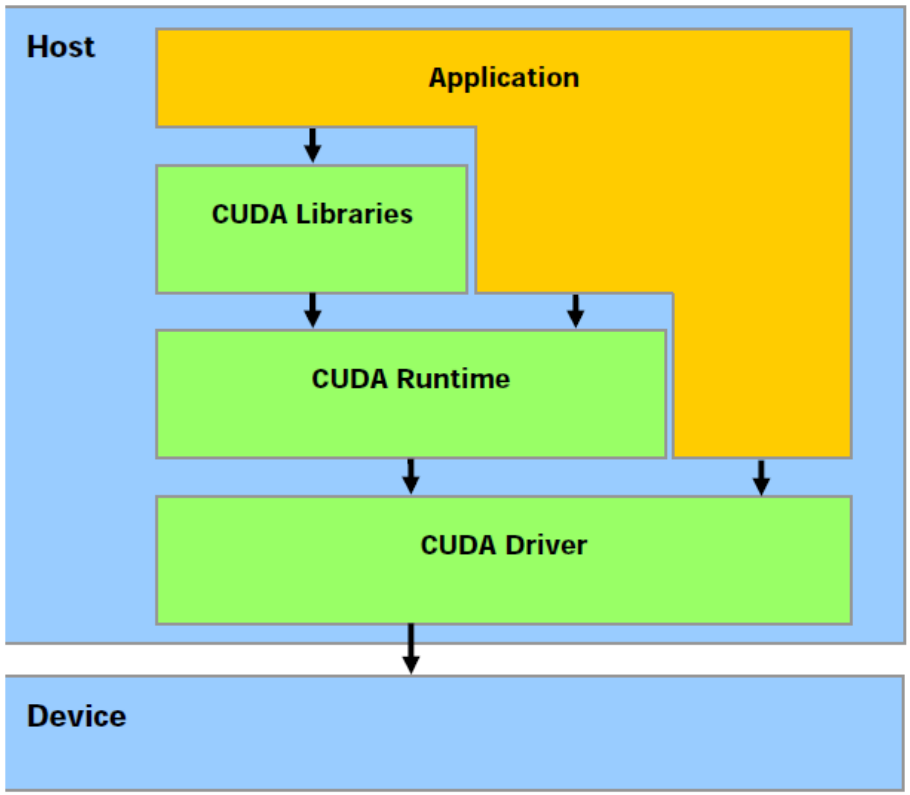
- Jeder Streamprozessor hat seinen eigenen Programmzähler und kann individuell verzweigen.
- Aber: Verzweigungsdivergenz innerhalb eines warps (32 Threads) ist teuer, abweichende Threads werden seriell ausgeführt.
- keine Rekursion

Ausführungsmodell





CUDA API



- Erweiterungen zu Standard C/C++
- Laufzeitumgebung: Common, Komponenten
- Software Development Kit (CUDA SDK) mit vielen Beispielen
- CUFFT und CUBLAS Bibliotheken
- Unterstützung für Windows, Linux und Mac OS X

CUDA Spracherweiterungen

- Funktionstyp Bezeichner
  - `__device__` auf device, auffrufbar vom device.
  - `__global__` auf device, auffrufbar vom host.
  - `__host__` auf host, auffrufbar vom host (default).
- Variablentyp Bezeichner
  - `__device__` in global memory, Gültigkeit von App.
  - `__constant__` in constant memory, Gültigkeit von App.
  - `__shared__` in shared memory, Gültigkeit von Block.
- Direktiven zum Kernel Aufruf (siehe unten).
- Built-in variables `__gridDim__`, `__blockIdx__`, `__blockDim__`, `__threadIdx__`, `__warpSize__` .

CUDA Ausführungskonfiguration

- Kernel instantiation:
  kernelFunc «<Dg, Db, Ns»> (arguments)
- dim3 Dg: Größe des Gitters
- Dg.x * Dg.y = Anzahl der Blöcke
- dim3 Db: Größe jedes Blocks
- Db.x * Db.y * Db.z = Anzahl der Threads pro Block
- Ns: Byteanzahl des dynamisch allokierten shared memory pro Block

Hallo CUDA I

```
// scalar product using CUDA
// compile with: nvcc hello.cu -o hello

// includes, system
#include<stdlib.h>
#include<stdio.h>

// kernel for the scale function to be executed on device
__global__ void scale_kernel (float *x, float a)
{
  int index = blockIdx.x*blockDim.x + threadIdx.x;
  x[index] *= a;
}

// wrapper executed on host that calls scale on device
// n must be a multiple of 32 !
void scale (int n, float *x, float a)
{
  // copy x to global memory on the device
  float *xd;
  cudaMalloc( (void**) &xd, n*sizeof(float) ); // allocate memory on device
  cudaMemcpy(xd,x,n*sizeof(float),cudaMemcpyHostToDevice); // copy x to device

  // determine block and grid size
  dim3 dimBlock(32);   // use BLOCKSIZE threads in one block
  dim3 dimGrid(n/32);  // n must be a multiple of BLOCKSIZE!

  // call function on the device
  scale_kernel<<<dimGrid,dimBlock>>>(xd,a);

  // wait for device to finish
  cudaThreadSynchronize();

  // read result
  cudaMemcpy(x,xd,n*sizeof(float),cudaMemcpyDeviceToHost);

```

Hallo CUDA II

```
    // free memory on device
38  cudaFree(xd);
}
40
int main( int argc, char** argv)
42 {
  const int N=1024;
44  float sum=0.0;
  float x[N];
46  for (int i=0; i<N; i++) x[i] = 1.0*i;
  scale(N,x,3.14);
48  for (int i=0; i<N; i++) sum += (x[i]-3.14*i)*(x[i]-3.14*i);
  printf("%g\n",sum);
50  return 0;
}
```

Skalarprodukt I

```
1  // scalar product using CUDA
   // compile with: nvcc scalarproduct.cu -o scalarproduct -arch sm_11
3
   // includes, system
5  #include<stdlib.h>
   #include<stdio.h>
7  #include<math.h>
   #include<sm_11_atomic_functions.h>
9
   #define PROBLEMSIZE 1024
11 #define BLOCKSIZE 32

13 // integer in global device memory
   __device__ int lock=0;
15
   // kernel for the scalar product to be executed on device
17 __global__ void scalar_product_kernel (float *x, float *y, float *s)
   {
19   extern __shared__ float ss[]; // memory allocated per block in kernel launch
     int block = blockIdx.x;
21   int tid = threadIdx.x;
     int index = block*BLOCKSIZE+tid;
23
     // one thread computes one index
25   ss[tid] = x[index]*y[index];
     __syncthreads();
27
     // reduction for all threads in this block
29   for (unsigned int d=1; d<BLOCKSIZE; d*=2)
       {
31       if (tid%(2*d)==0) {
           ss[tid] += ss[tid+d];
33       }
         __syncthreads();
35   }
```

Skalarprodukt II

```
37    // combine results of all blocks
      if (tid==0)
39      {
          while (atomicExch(&lock,1)==1) ;
41        *s += ss[0];
          atomicExch(&lock,0);
43      }
    }
45
    // wrapper executed on host that uses scalar product on device
47  float scalar_product (int n, float *x, float *y)
    {
49    int size = n*sizeof(float);

51    // allocate x in global memory on the device
      float *xd;
53    cudaMalloc( (void**) &xd, size ); // allocate memory on device
      cudaMemcpy(xd,x,size,cudaMemcpyHostToDevice); // copy x to device
55    if( cudaGetLastError() != cudaSuccess)
        {
57        fprintf(stderr,"error␣in␣memcpy\n");
          exit(-1);
59      }

61    // allocate y in global memory on the device
      float *yd;
63    cudaMalloc( (void**) &yd, size ); // allocate memory on device
      cudaMemcpy(yd,y,size,cudaMemcpyHostToDevice); // copy y to device
65    if( cudaGetLastError() != cudaSuccess)
        {
67        fprintf(stderr,"error␣in␣memcpy\n");
          exit(-1);
69      }

71    // allocate s (the result) in global memory on the device
      float *sd;
73    cudaMalloc( (void**) &sd, sizeof(float) ); // allocate memory on device
```

Skalarprodukt III

```
    float s=0.0f;
75  cudaMemcpy(sd,&s,sizeof(float),cudaMemcpyHostToDevice); // initialize sum on device
    if( cudaGetLastError() != cudaSuccess)
77    {
        fprintf(stderr,"error in memcpy\n");
79      exit(-1);
      }
81
    // determine block and grid size
83  dim3 dimBlock(BLOCKSIZE);   // use BLOCKSIZE threads in one block
    dim3 dimGrid(n/BLOCKSIZE);  // n is a multiple of BLOCKSIZE
85
    // call function on the device
87  scalar_product_kernel<<<dimGrid,dimBlock,BLOCKSIZE*sizeof(float)>>>(xd,yd,sd);

89  // wait for device to finish
    cudaThreadSynchronize();
91  if( cudaGetLastError() != cudaSuccess)
      {
93      fprintf(stderr,"error in kernel execution\n");
        exit(-1);
95    }

97  // read result
    cudaMemcpy(&s,sd,sizeof(float),cudaMemcpyDeviceToHost);
99  if( cudaGetLastError() != cudaSuccess)
      {
01      fprintf(stderr,"error in memcpy\n");
        exit(-1);
03    }

05  // free memory on device
    cudaFree(xd);
07  cudaFree(yd);
    cudaFree(sd);
09
    // return result
```

Skalarprodukt IV

```
11    return s;
  }
13
  int main( int argc, char** argv)
15 {
    float x[PROBLEMSIZE], y[PROBLEMSIZE];
17  float s;
    for (int i=0; i<PROBLEMSIZE; i++) x[i] = y[i] = sqrt(2.0f);
19  s = scalar_product(PROBLEMSIZE,x,y);
    printf("result␣of␣scalar␣product␣is␣%f\n",s);
21  return 0;
  }
```

Bemerkung: Dies ist nicht die effizienteste Version. Siehe das CUDA Tutorial für eine Version welche die volle Speicherbandbreite nutzt