GPUs in Linux Systemen

Graphic Processing Unit (GPU) ist ein auf die Berechnung von Grafiken spezialisierter und optimierter Prozessor für Computer, Spielkonsolen und Smartphones. Sie besitzen einige Tausend Compute Einheiten (skalierbares Array von multicore and multithreaded Streaming Multiprocessors, SM), die ein ein hohes Maß an Parallelisierung erlauben und dadurch CPUs in ihrer theoretischen Rechenleistung deutlich überlegen sind, so erreicht eine NVIDIA Tesla P100 (2017) 11 TFLOPS und eine moderne CPU 2 TFLOPS. Dies ist aber nur für spezielle vektorisierbare Algorithmen wirklich von Vorteil (z.B. Lösung von Linearen Gleichungssystemen → Neuronale Netze). Beim GPU Computing wird der Grafikprocessor gemeinsam mit der CPU zur Beschleunigung von wissenschaftlich technischen Berechnungen verwendet (GPGPU computing). Moderne GPUs rechnen inzwischen mit double precision. Es existieren 2 Plattformen zur Integration von GPUs zum wissenschaftlichen Rechnen

- NVIDIA‘s proprietäres Programmierinterface für GPUs CUDA
- Offene Programmierplatform OpenCLfür uneinheitliche Prozessoren (CPU, GPU und FPGA)

Sowohl CUDA als auch OpenCL sind auf Linux Systemen unterstützt.

General Purpose GPUs

NVIDIA Pascal Architektur. „Preiswerte“ (< 200 Euro) Grafikkarte für das GPGPU Computing mit einer Leistung von 10 TFLOPS.

NVIDIA, GTX1060

NVIDIA, Tesla V100.

Modell	Mikroarchitektur	GPU	Anzahl SMs	CUDA[3] Cores	Speicher	GFLOPS 32FP[4]
Tesla P4	Pascal	GP104	20	2560	8 GB GDDR5	5500
Tesla P40	Pascal	GP102	30	3840	24 GB GDDR5X	11800
Tesla P100	Pascal	GP100	56	3584	16 GB HBM2[5]	10600
Tesla V100	Volta	GV100	80	5120	16 GB HBM2	15700

Aufbau einer NVIDIA GPU

Eine GPU enthält mehrere Streaming Multiprocessors (SMs), bei der GeForce GTX 1060 mit 6 GB sind es beispielsweise 10 SMs, die Tesla GV100 enthält 84 SMs. Ähnlich wie CPU-Cores können diese mehrere Threads parallel ausführen, jeweils 32 Threads sind zu einem so genannten Warp gruppiert. Ein Kernel muss mindestens einem Warp zur parallelen Ausführung zugewiesen werden. Eine SM einer Volta-GPU kann bis zu 64 Warps parallel ausführen. Mit 84 SMs ergeben sich also 5.376 Warps. Das entspricht genau der Anzahl der so genannten CUDA-Cores.

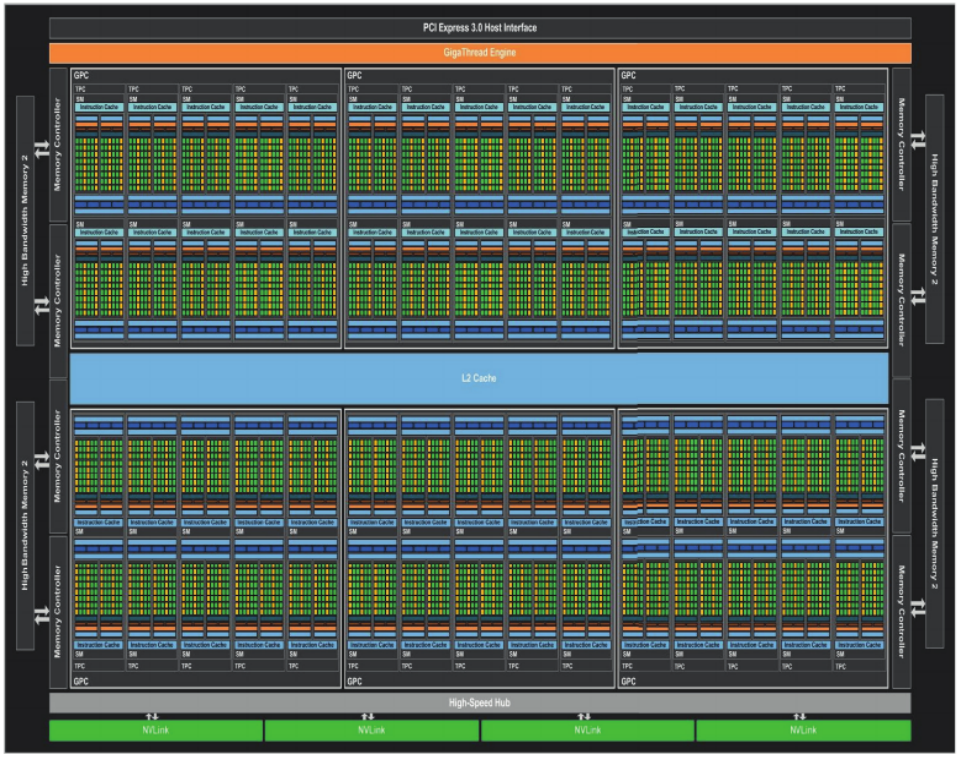
Streaming Multiprozessor SM:

Aufbau eines einzelnen SM mit Pascal-Architektur Der SM ist in diesem Fall in 2 identische Processing Units geteilt, von denen jeder 32 CUDA-Cores besitzt.

- Double Precision Units (DP)
- Special Function Units (SFU) sin, cos, ….
- Texture Mapping Units (TMUs)
- Load/Store-Units (LD/ST)

Aufbau einer NVIDIA GPU

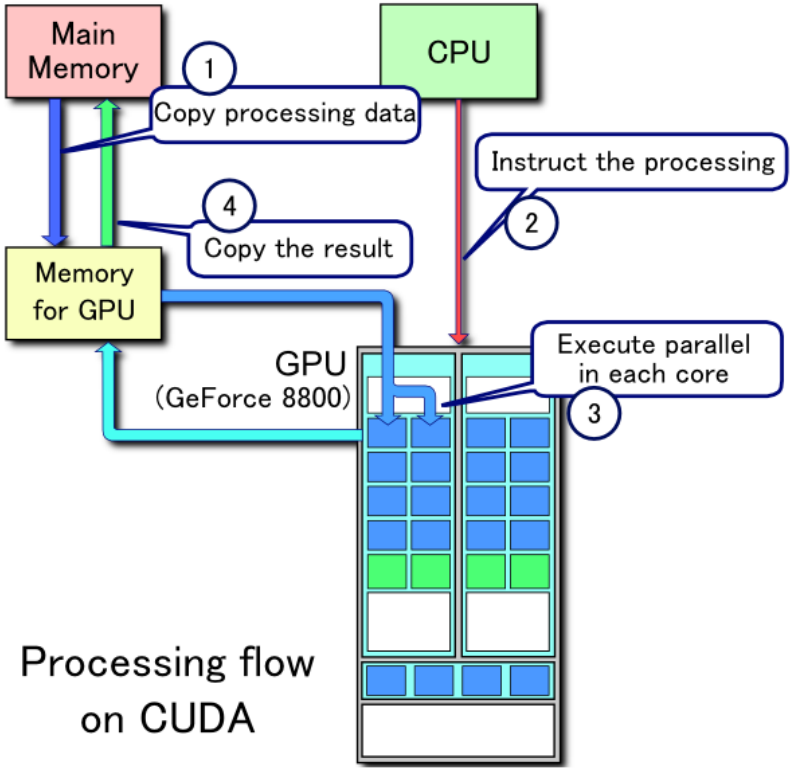
Eine GPU enthält mehrere Streaming Multiprocessors (SMs), bei der GeForce GTX 1060 mit 6 GB sind es beispielsweise 10 SMs, die Tesla GV100 enthält 84 SMs. Ähnlich wie CPU-Cores können diese mehrere Threads parallel ausführen, jeweils 32 Threads sind zu einem so genannten Warp gruppiert. Ein Kernel muss mindestens einem Warp zur parallelen Ausführung zugewiesen werden. Eine SM einer Volta-GPU kann bis zu 64 Warps parallel ausführen. Mit 84 SMs ergeben sich also 5.376 Warps. Das entspricht genau der Anzahl der so genannten CUDA-Cores.



Aufbau einer NVIDIA-GPU:
Alle SMs haben einen identischen Aufbau und einen L2-Cache, Memory Controller, Bus-Interfaces mit der so genannten GigaThread Engine, die ankommende Aufgaben auf die einzelnen SMs verteilt.

CUDA

CUDA ist eine NVIDIA Architektur für parallele Berechnungen auf Grafik-prozessoren. CUDA umfasst C und C++ Erweiterungen um Parallelität von Daten und Berechnungen zu formulieren (10^3 Cuda Cores). Vorausetzungen zur Benutzung sind eine NVIDIA GPU und eine Installation des CUDA Toolkits. Die Erweiterungen erlauben die Verwendungen von weiteren C++ Keywords, die mit dem `nvcc` Compiler code erzeugen, der auf host (CPU) und auf device (GPU) ausgeführt werden kann.



Das Toolkit enthält neben dem NVCC-Compiler, Treibern und der Runtime-Unterstützung diverse Debugging- und Profiling-Tools, sowie spezielle für die GPU beschleunigte Bibliotheken aus den Bereichen Deep Learning, Signal- Bild- und Videoverarbeitung, Lineare Algebra / Mathematik und Parallele Algorithmen.

CUDA

CUDA Bibliotheken:

Die zum Teil CPU-orientierte Bibliotheken fast ohne Code-Änderungen ersetzen können.

Bibliothek	Bereich	Zweck	im Deep Learning SDK	Anmerkung
cuDNN	Deep Learning	Routinen für neuronale Netze	x	
TensorRT	Deep Learning	Neural Network Inference	x	
DeepStreamSDK	Deep Learning	Video- und Bildanalyse	x	
cuBLAS	Lin. Algebra Math	Standard-Lib Basic Lineare Algebra (BLAS)	x	
cuSPARSE	Lin. Algebra Math	Matrizen-Operationen	x	
cuRAND	Lin. Algebra/Math	Zufallszahl-Generatoren		
cuSolver	Lin. Algebra/Math	Solver für lineare Gleichungssysteme		
AmgX	Lin. Algebra/Math	Erweiterte Solver für Hightech-Industrie und Forschung		Registrierung nötig
cuFFT	Signal/Bild/Video	klassische Fast Fourier Transformationen		
NVIDIA Codec SDK	Signal/Bild/Video	Video Encoder/Decoder		Registrierung nötig
NCCL	Parall. Algorithmen	Unterstützung für Multi-GPU- und Multi-Node-Systeme	x	
nvGRAPH	Parall. Algorithmen	Graphen- und Cluster-Algorithmen		
Thrust	Parall. Algorithmen	High Level Interface zur Parallelisierung von Grund-Algorithmen		zum Einstieg in GPU Computing geeignet

CUDA Installation

- CUDA toolkit 10.0 Installation für OpenSuSE 15

Bei der Version 10.0 handelt es sich um eine von uns mit OpenSuSE 15 getesteten Version. Das lokal auszuführende rpm File kann über folgenden link heruntergeladen werden.

cuda-repo-opensuse15-10-0-local-10.0.130-410.48-1.0-1.x86_64.rpm

Die Installation erfolgt mit

```
rpm -i cuda-repo-opensuse15-10-0-local-10.0.130-410.48-1.0-1.x86_64.rpm
zypper refresh
zypper install cuda
```

Der CUDA code befindet sich in `/usr/local/cuda-10.0`

Weiterhin wollen wir die Bibliothek cuDNN, Version 7.5 für CUDA 10.0 installieren. Das tar File finden wir unter „`cuDNN für Linux`“

cudnn-10.0-linux-x64-v7.5.0.56.tgz

```
cp ~root/Downloads/cudnn-10.0-linux-x64-v7.5.0.56.tgz /usr/local/src/.
tar -zxvf cudnn-10.0-linux-x64-v7.5.0.56.tgz
cp -a /usr/local/src/cudnn/cuda/include/* /usr/local/cuda/include
cp -a /usr/local/src/cudnn/cuda/lib64/*  /usr/local/cuda/lib64
```

CUDA Installation

- CUDA toolkit 10.0 Installation für OpenSuSE 15

  Weiterhin wollen wir die Bibliothek NCCL, Version 2.4.2 für CUDA 10.0 installieren.

  Das File wird unter „O/S agnostic local installer“ heruntergeladen.

  nccl_2.4.2-1+cuda10.0_x86_64.txz

```
cp ~root/Downloads/nccl_2.4.2-1+cuda10.0_x86_64.txz /usr/local/src/.
tar -zxvf nccl_2.4.2-1+cuda10.0_x86_64.txz
cp -a /usr/local/src/nccl_2.3.7-1+cuda10.0_x86_64/include/*   \
 /usr/local/cuda/include
cp -a /usr/local/nccl_2.3.7-1+cuda10.0_x86_64/lib/* \
 /usr/local/cuda/lib64
```

- Um CUDA Programme auszuführen, müssen in .bashrc folgende Variable für die user gesetzt werden und alle user müssen in der group video sein.

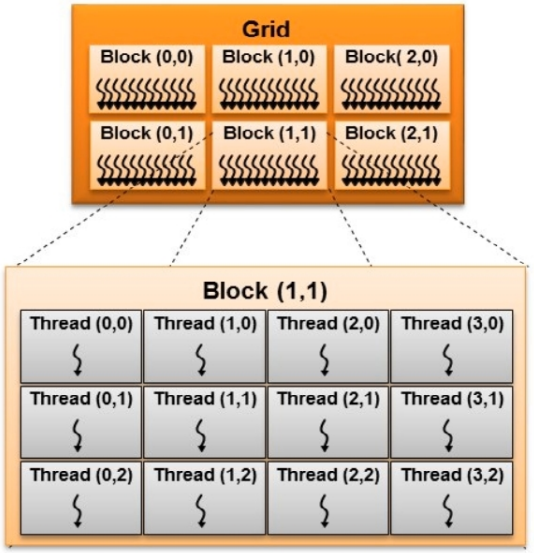
```
# CUDA
export MYCUDA=/usr/local/cuda/bin
# add local directory
export PATH=.:$MYCUDA:$PATH

export LD_LIBRARY_PATH=.:$MYCUDA/lib64:$LD_LIBRARY_PATH
```

CUDA Programming

- CUDA ist eine heterogene Compute Umgebung und erlaubt sowohl den host (CPU) als auch das über PCIe angebundene device (GPU) anzusprechen.

Operationen auf Daten in der GPU (device) werden parallel in Form von Kernels ausgeführt, ein Kernel kann von vielen threads ausgeführt werden. Threads sind Aufgaben, die parallel auf einem subset der Daten abgearbeitet werden. Threads sind in thread blocks organisiert und die wiederum in ein Grid. Sie können unabhängig voneinander 1D, 2D oder 3D sein. Der host startet die Kernel mit einer gewählten Dimensionierung.

CUDA Kernel werden mit `__global__` spezifiziert
- werden vom host gerufen und auf dem device ausgeführt
- kein Zugang zum host memory und host functions
- müssen void zurückgeben
- statische Variable sind nicht erlaubt




```
__global__ void myKernel(int * a);
int main(){
dim_3 dGrid, dBlock;
myKernel <<< dim_3 dGrid, dim_3 dBlock >>> ( variables )
……...
__global__ void myKernel(int * a){
……...
```

CUDA Syntax

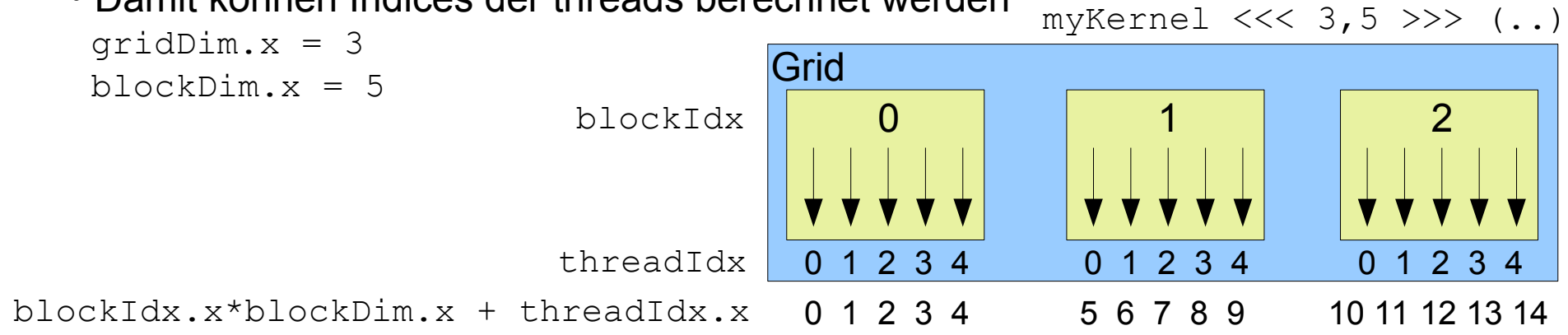
- Kernel
  - Dem Kernel können wie in C/C++ Funktionen Argumente übergeben werden.
    - Pointer zum device memory
    - Parameter mit call by value

```
__global__ void myKernel(int * a , int b){
  a[0] = b;
}
```

  - Die Kernel besitzen builtin read only Variable
    - `gridDim` : Dimension des Grids
    - `blockIdx` : Index eines Blocks im Grid
    - `blockDim` : Dimension des Blocks
    - `threadIdx` : Index des threads im thread Block

    nach dem Kernel call kann die Dimensionierung nicht geändert werden
  - Damit können Indices der threads berechnet werden

```
gridDim.x = 3
blockDim.x = 5
```



CUDA Syntax

- Kernel
  - C/C++ support
    - alle C Operatoren +, -, >, …
    - Funktionen der Standard math Bibliothek sinf(), cosf(), ….
    - if ( ) , for ( ) , while() , ...
    - Klassen, Templates, C++11
    - Nicht implementiert sind: Standard C++ Library, Run time type information, Expception handling, ….
  - Es gibt device Funktionen

```
__device__ int myDeviceFunction(int i){
……
}

__global__ void myKernel(int * a){
   idx = blockIdx.x * blockDim.x + threadIdx.x;
   a[idx] = myDeviceFunction(idx);
}
```

host und device haben verschiedene Addressräume

  - Der host verwaltet den device Speicher

```
cudaMalloc(void ** pointer , size_t nBytes);
cudaMemset(void * pointer ,int value ,size_t count);
cudaFree(void * pointer );
cudaMemcpy(void* dest, void* src, size_t nBytes, direction)
```

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyHostToHost
cudaMemcpyDeviceToDevice

CUDA Beispiele

- Beispiel Programm

myFirstCuda.cu

```
include <stdio.h>
__global__ void myFirstCuda(int *y){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  y[i] = threadIdx.x;
}

int main(void){
   int *y , *d_y;
   y = (int*)malloc(12*sizeof(int));
   cudaMalloc(&d_y, 12*sizeof(int));
   myFirstCuda<<<3,4>>>(d_y);
   cudaMemcpy(y, d_y, 12*sizeof(int), cudaMemcpyDeviceToHost);
   for (int i = 0; i < 12; i++) printf("%i",y[i]);
}
```

- Compile und execute

```
$> nvcc myFirstCuda.cu -o myFirstCuda
$> ./myFirstCuda
```

- Das folgende Beispiel skaliert einen Vektor mit 1 Million Elementen und addiert einen weiteren Vektor.

saxpy.cu

Wir wollen nun die CUDA Beispiele genauer betrachten. Wie bereits diskutiert gibt es interne Kernel Variablen, die Zugang zum GridIdx und BlockIdX erlauben.

Arbeitsvorschlag:

- Setzen Sie im Kernel im Programm `myFirstCuda.cu` die array Variable konstant, auf den blockIdx, auf den threadIdx und auf i . Interpretieren Sie jeweils den Output.

- `saxpy.cu` enthält ausführliche Kommentare. Sehen Sie sich den Code an und versuchen Sie den Programmfluss zu verstehen.

- Kompilieren Sie `saxpy.cu` und führen Sie das Programm aus.

- Führen Sie nun das ausführbare Programm mit `nvprof` aus. Hier sehen Sie welche Zeiten in welchen Programmschritt verbraucht werden.