CUDA

Hello CUDA

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

/**
 * KernelFunktion (Ausführung auf der GPU)
 */
__global__ void hello_cuda()
{
        printf("Hello CUDA!\n");
}

/**
 * Host Code (Ausführung auf der CPU)
 */

int main()
{
        hello_cuda <<<1,1>>> (); // Kernel Launch, was ist '<< <1,1> >>' ?
        cudaDeviceSynchronize();
        cudaDeviceReset();
        return 0;
}
```

Modifiers

Kernelfunktion

```
__global__ void called_from_host(int* some_data, int size) { ... }
```

Devicefunktion

```
__device__ int called_from_device(int* some_data, int size) { ... }
```

Modifiers

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <string>
#include <stdio.h>

using namespace std;

__device__ const char* part_two() {
    return "CUDA";
}

__device__ const char* part_one() {
    return "Hello";
}

__device__ void result(char* buffer, size_t bufferSize) {
    const char* p1 = part_one();
    const char* p2 = part_two();
    const char* to_append = "!!!";

    snprintf(buffer, bufferSize, "%s %s%s", p1, p2, to_append);
}

__global__ void i_am_api()
{
    char buffer[50];
    result(buffer, sizeof(buffer));
    printf("%s\n", buffer);
}

// Host Code wie zuvor
```

Funktionsumfang in CUDA

GPUs sind primär Rechenmaschinen ihre Stärke liegt in der hohen parallelisierbarkeit einfacher Rechenoperationen

- Eingeschränkte Unterstützung der C/C++ Standard Libraries

Stattdessen spezialisierte Libraries wie:

- cuFFT - CUDA Fast Fourier Transform
- cuDNN - CUDA Deep Neural Network
- Thrust - Parallele Algorithmen Bibliothek
- DALI - NVIDIA Data Loading Library

Übersicht wichtiger Funktionen der CUDA Runtime API

```
...
cudaDeviceSynchronize();
cudaDeviceReset();
...
```

- `cudaDeviceSynchronize()` vgl. join()
- `cudaDeviceReset()`

Kernel Launch

```
hello_cuda <<<a,b>>>();
```

```
dim3 grid, block;
block = dim3(32); // dim3(32,1,1) == 32
grid = dim3(48); // dim3(48,1,1) == 48

hello_cuda <<<grid, block>>>();
```

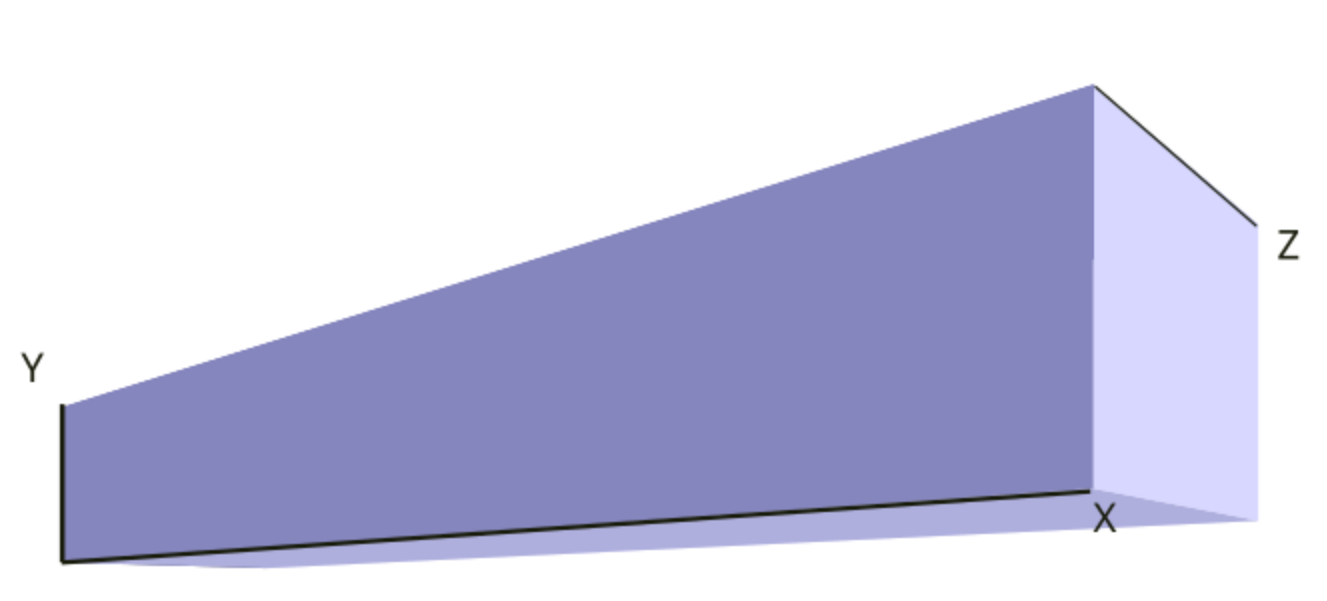
Ausgabe: 32 * 48 = 1536 mal Hello CUDA

Thread Organization

- Thread Blöcke
- Grids
- Threads

Thread Block

Threads werden in **Blöcken** organisiert



Für jeden Block gilt: $X \times Y \times Z \leq 1024$ (bzw. 512 für CC $\geq$)

Thread Block

- Jeder Streaming Multiprozessor (SM) führt mindestens einen Block aus.
- Hat eine BlockID (blockIdx.x, blockIdx.y, blockIdx.z)
- Wird in **Warps** unterteilt

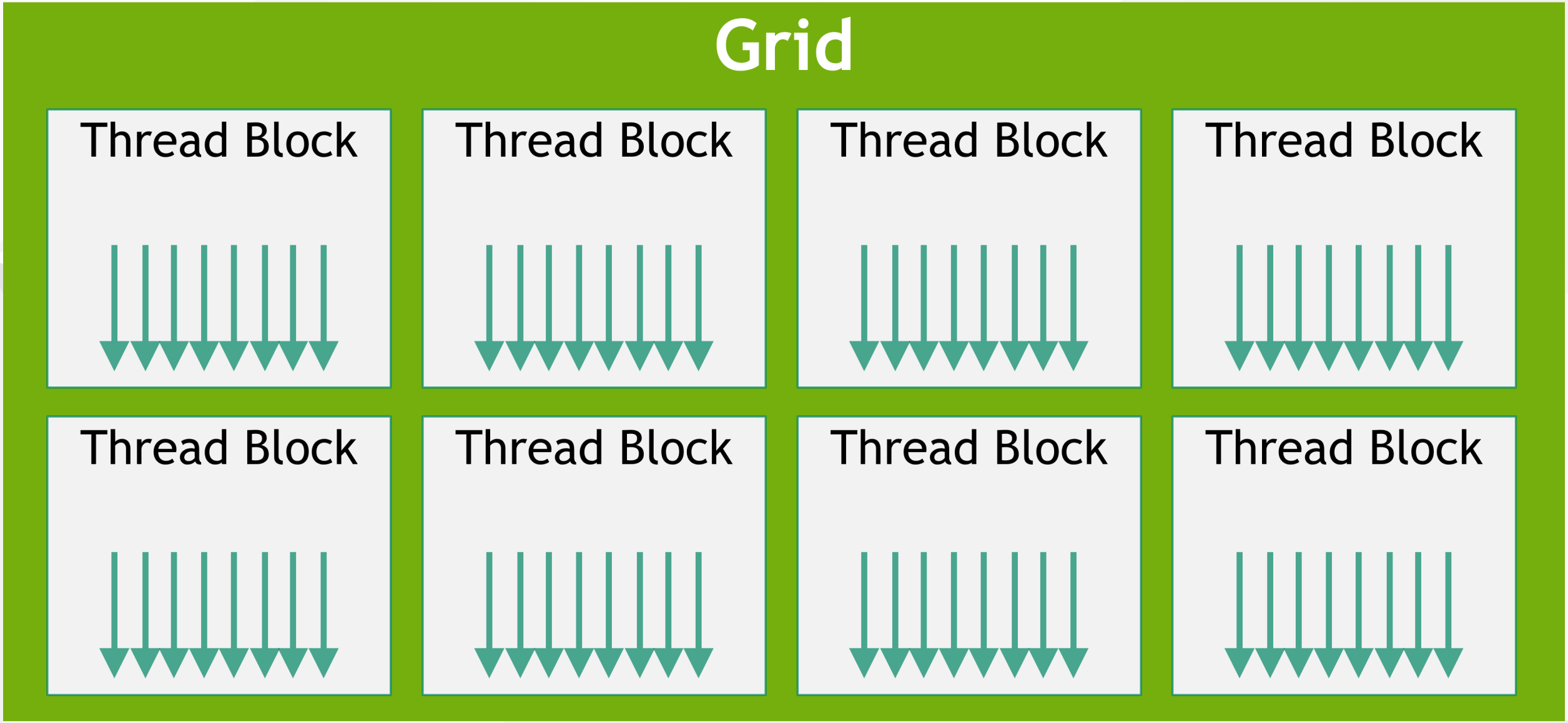
Grid

Blöcke werden in **Grids** organisiert

- Jedes Grid kann $(2^{31} - 1) * 65535 * 65535$ Blöcke beinhalten

$$\#threads = 1024 * (2^{31} - 1) * 65535 * 65535 = 2.305.843.008.139.952.128$$

(theoretisches Maximum CC $\geq$ 7.5)


Grid
Thread Block
Thread Block
Thread Block
Thread Block
Thread Block
Thread Block
Thread Block
Thread Block

Thread

threadId ist nur innerhalb eines blocks eindeutig!

```
// Launch Konfiguration grid = dim3(48, 1, 1) & block = dim(256, 1, 1)
int gid = blockIdx.x + threadIdx.x
```

```
// Launch Konfiguration grid = dim3(48, 48, 1) & block = dim(256, 1, 1)
int tid = threadIdx.x;
int block_offset = blockIdx.x * blockDim.x;
int row_offset = gridDim.x * blockDim.x * blockIdx.y;
int gid = row_offset + block_offset + tid;
```

Abhängig vom Problem benötigen wir keine **gid**

```
// Launch Konfiguration grid = dim(48, 48, 1) & block = dim(128, 128, 1)
// Für Matrizenberechnung
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Matrixmultiplaktion

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "time.h"

__device__ float multiply(float a, float b) {
    return a * b;
}

__global__ void mat_mul() {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Hier Matrixmutiplikation ausführen

}

void fillMatrix(float *matrix, int width, int height) {
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            matrix[i * width + j] = (float)rand() / (float)(RAND_MAX / 10);
        }
    }
}
```

```
int main(void) {
    int matSize = 1000;
    int matSizeBytes = matSize * matSize * sizeof(float);
    srand(time(NULL));

    float *matrixA, *matrixB, *result;
    *matrixA = (float *)malloc(matSizeBytes);
    *matrixB = (float *)malloc(matSizeBytes);
    *result = (float *)malloc(matSizeBytes);

    if (matrixA == NULL || matrixB == NULL || result == NULL) {
        fprintf(stderr, "Speicherzuweisung fehlgeschlagen.\n");
        return 1;
    }
    fillMatrix(matrixA, matSize, matSize);
    fillMatrix(matrixA, matSize, matSize);

    dim3 blockSize(10,10);
    dim3 gridSize((matSize + blockSize.x-1)/blockSize.x, (matSize + blockSize.y-1)/blockSize.y);

    // Wie übergeben wir die Matrix bzw. allgemein Werte an die GPU?
    mat_mul <<<gridSize, blockSize>>>();

    cudaDeviceSynchronize();
    cudaDeviceReset();
}
```

Übersicht wichtiger Funktionen

- `cudaMalloc(void **devPtr, size_t size)` vgl. malloc()
- `cudaMemcpy(void *dest, void *scr, size_t size, cudaMemcpyKind m)`
- `cudaFree()` vgl. free()

cudaMemcpyKind

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Matrixmultiplaktion: Mit Datenübertragung

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "time.h"

__device__ float multiply(float a, float b) { return a * b; }

__global__ void mat_mul(float *matA, float *matB, float *res, int size) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
        float sum = 0.0;
        for (int k = 0; k < size; k++) {
            sum += multiply(matA[row * size + k], matB[k * size + col]);
        }
        res[row * size + col] = sum;
    }
}

// fill Matrix unverändert
void fillMatrix(float *matrix, int width, int height) {...}
```

Matrixmultiplaktion: Mit Datenübertragung

```
int main(void) {
    int matSize = 1000;
    int matSizeBytes = 1000 * 1000 * sizeof(float);

    srand(time(NULL));

    float *h_matA, *h_matB, *h_res, *d_matA, *d_matB, *d_res;

    h_matA = (float *)malloc(matSizeBytes);
    h_matB = (float *)malloc(matSizeBytes);
    h_res = (float *)malloc(matSizeBytes);

    if (h_matA == NULL || h_matB == NULL || h_res == NULL) { fprintf(stderr, "Speicherzuweisung fehlgeschlagen.\n");
        return 1;
    }
    fillMatrix(h_matA, matSize, matSize);
    fillMatrix(h_matB, matSize, matSize);

    cudaMalloc((void**)&d_matA, matSizeBytes); // Speicherallokation auf dem Device
    cudaMalloc((void**)&d_matB, matSizeBytes);
    cudaMalloc((void**)&d_res, matSizeBytes);
    ...
```

Matrixmultiplikation: mit Datenübertragung

```
    ...
    cudaMemcpy(d_matA, h_matA, matSizeByte, cudaMemcpyHostToDevice); // Von Host zu Device
    cudaMemcpy(d_matB, h_matB, matSizeByte, cudaMemcpyHostToDevice);

    dim3 blockSize(10,10);
    dim3 gridSize((matSize + blockSize.x-1)/blockSize.x, (matSize + blockSize.y-1)/blockSize.y);

    mat_mul <<<gridSize, blockSize>>>(d_matA, d_matB, d_res, matSize);

    cudaDeviceSynchronize();
    cudaMemcpy(h_res, d_res, matSizeByte, cudaMemcpyDeviceToHost); // Von Device zu Host
    cudaFree(d_matA); cudaFree(d_matB); cudaFree(d_res); // Speicherfreigabe Device
    free(h_matA); free(h_matB); free(h_res); // Speicherfreigabe Host
    cudaDeviceReset();
}
```

Error Handling

Returntype `cudaError`

```
cudaError_t status = cudaMalloc((void**)&devicePtr, size);
if (status != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed: %s\n", cudaGetErrorString(status));
    // Fehlerbehandlung...
}
```

Error Handling

Was wenn der Funktionsaufruf keinen Wert zurückgibt?

`cudaGetLastError()`

```
a_kernel_function<<<grid, blocks>>>(...);

cudaError_t status = cudaGetLastError();
if (status != cudaSuccess) {
    fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(status));
    // Fehlerbehandlung
}
```

Abfragen von Hardwareinformationen über cudaDeviceProp

`cudaDeviceProp properties` &
`cudaGetDeviceProperties(&properties, deviceNumber)`

Informationen:

- Gerätename
- #Multiprozessoren
- Warpsize
- usw.

Beispiel

```
void query_device() {
        int devNo = 0;
        cudaDeviceProp iProp;
        cudaGetDeviceProperties(&iProp, devNo);
        printf("Anzahl der MP: %d\n", iProp.multiProcessorCount);
        printf("Max Anzahl von Threads pro MP: %d\n", iProp.maxThreadsPerMultiProcessor);
        printf("Warp-Größe: %d\n", iProp.warpSize);
        printf("Warps pro MP: %d\n", iProp.maxThreadsPerMultiProcessor / iProp.warpSize);
}
```

```
Ausgabe:
Anzahl der Multiprozessoren: 48
Maximale Anzahl von Threads pro Multiprozessor: 1536
Warp-Größe: 32
Maximale Anzahl von Warps pro Multiprozessor: 48
```

Das CUDA Execution Model

- Warps
- SIMT
- Warp Divergence
- Warp Synchronisierung
- Schlussfolgerungen

Die kleinste schedulbare Einheit - Warp

- Jeder Block wird in **Warps** zu je 32 Threads unterteilt.
- Jeder **Warp** hat eine eindeutige Warp ID
- Warps werden nach **SIMT** ausgeführt
- Warps teilen sich **gemeinsamen Speicher** innerhalb eines SM

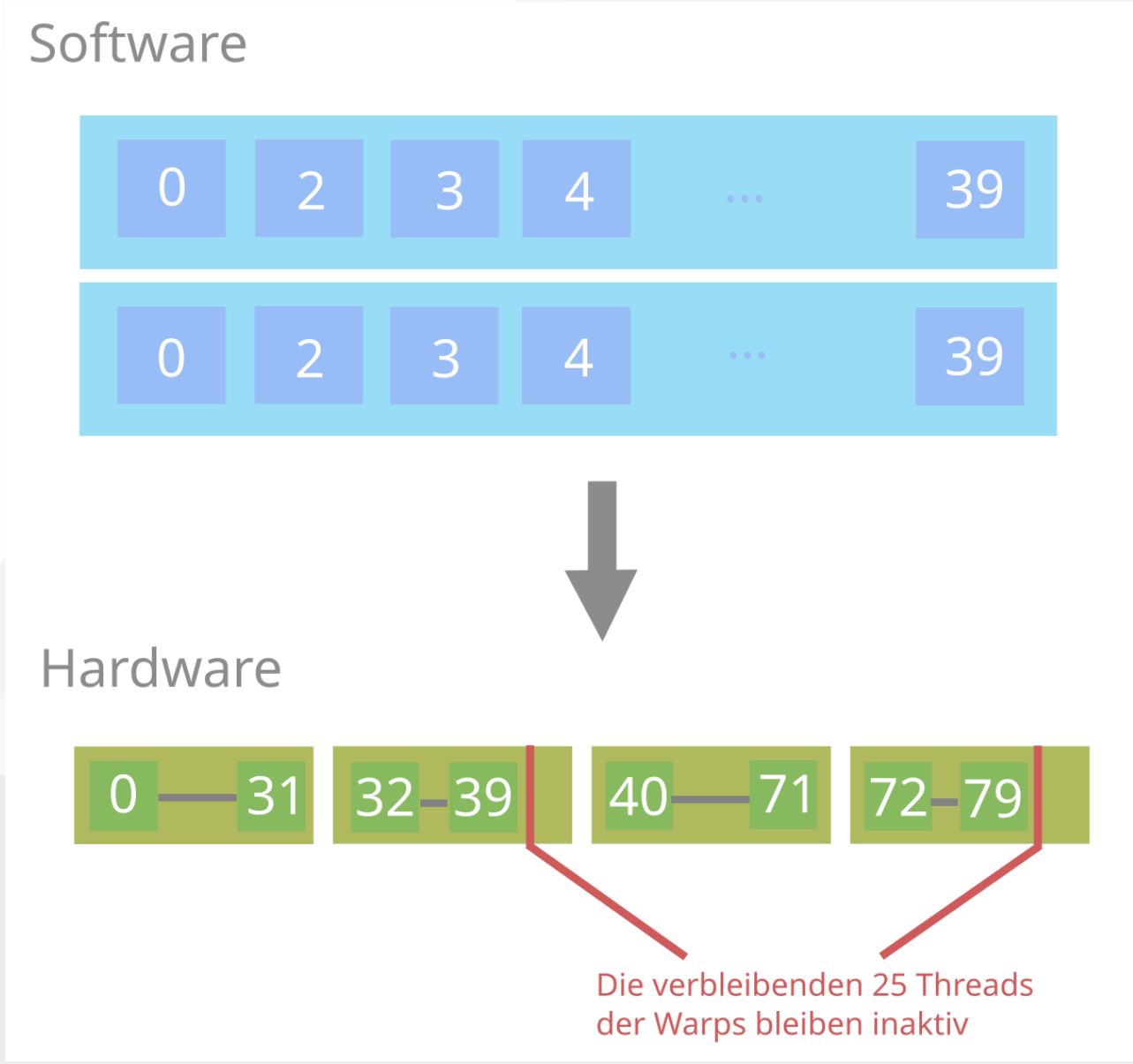
Können einen der folgenden Zustände annehmen:

- *selected* (vgl. running)
- *eligible* (vgl. ready)
- *stalled* (vgl. blocked)

Warp

Ungünstige Blockgröße:

- verschwendetes Parallelisierungspotential
- worst case: 2304 Threads
- best case: 73.728 Threads



Ausführungsmodell SIMT

Single Instruction Multiple Threads (spezialfall von SIMD)

- Alle Threads innerhalb eines Warps werden synchron ausgeführt

SIMT vs SIMD

SIMT bietet höhere Flexibilität auf Kosten der Performance

- **Single instruction, multiple register sets**
- **Single instruction, multiple addresses**
- **Single instruction, multiple flow paths**

Warp Divergence

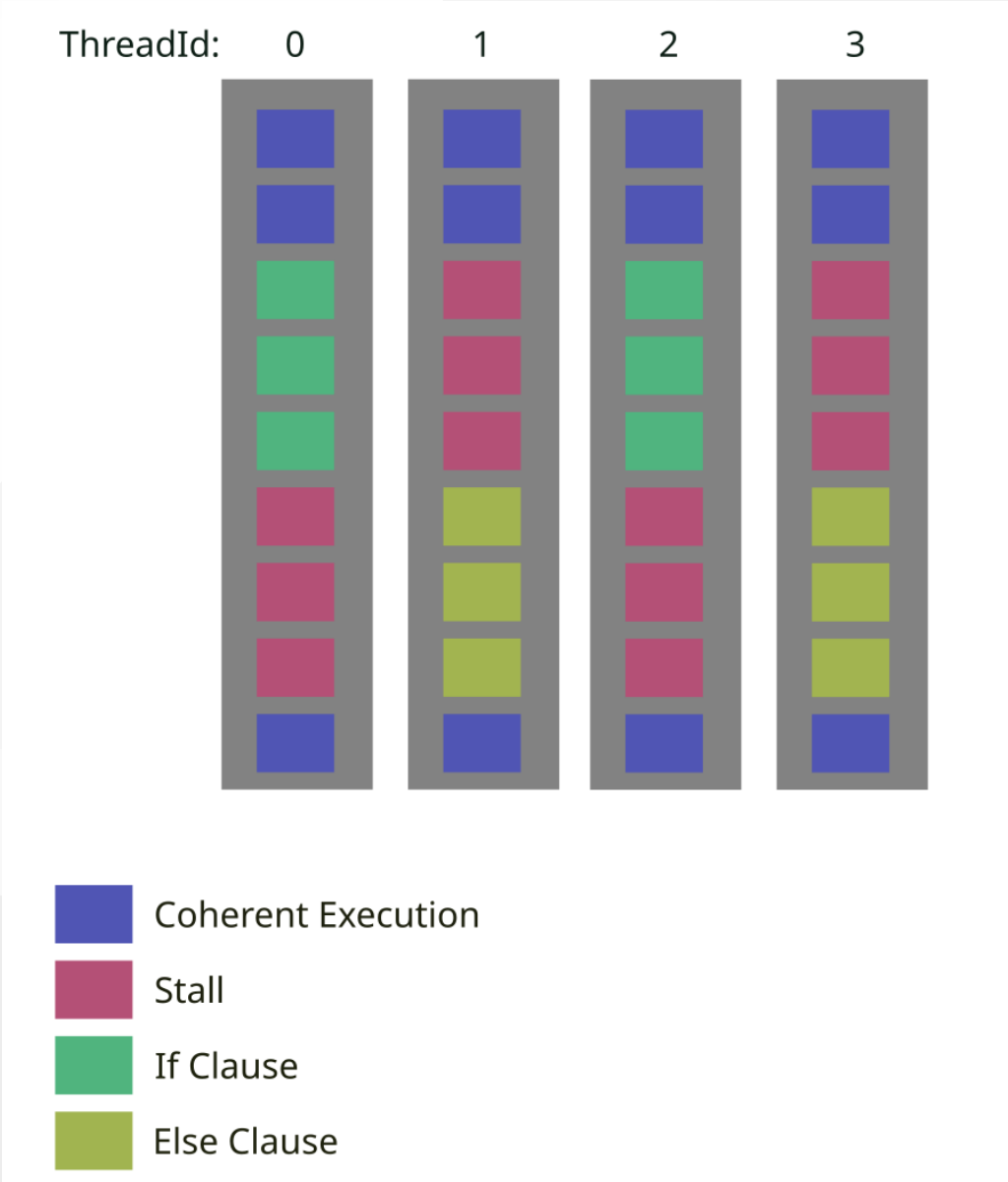
Control flow statements mit divergierenden Pfaden (innerhalb eines Warps).

```
__global__ void this_causes_warp_divergence() {
        int tid = threadIdx.x;
        if (tid % 2 == 0) {
                // do something
        } else {
                // do something else
        }
}
```

Warp Divergence

Divergierende Pfade werden seriell ausgeführt.

Performanceverlust: 50%



Warp Divergence

Metrik für Warp Divergence ist die **branch efficiency**

$$\text{Branch Efficiency} = 100\% \times \frac{\#\text{Branches} - \#\text{Divergent Branches}}{\#\text{Branches}}$$

Warp Divergence

```
__global__ void this_does_not_cause_warp_divergence() {
    // Jeder Threadblock hat 64 Threads
    int tid = blockIdx.x * threadIdx.x;

    if (tid / 32 < 1) {
        // do something
    } else {
        // do something else
    }
}
```

Warp Synchronisierung

Warps innerhalb eines Blocks nicht notwendigerweise synchron!

Synchronisierung erfolgt mittels:
`__syncthreads()` (vgl. pthread_barrier_wait(...))

```
__global__ void modifyArray(int *data, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    data[index] += 1;
    __syncthreads();
    if (index % 2 == 0) {
        data[index] *= 2;
    }
}
```

Schlussfolgerungen

- Warps entsprechen Threads in klassischer Programmierung
- Blocksize = $X \times Warpsize$ vermeidet nutzlose Threads
- Control Flow Statements - Warp Divergence vermeiden
- Warps innerhalb eines Blocks keine synchrone Ausführung

Profiling Tools

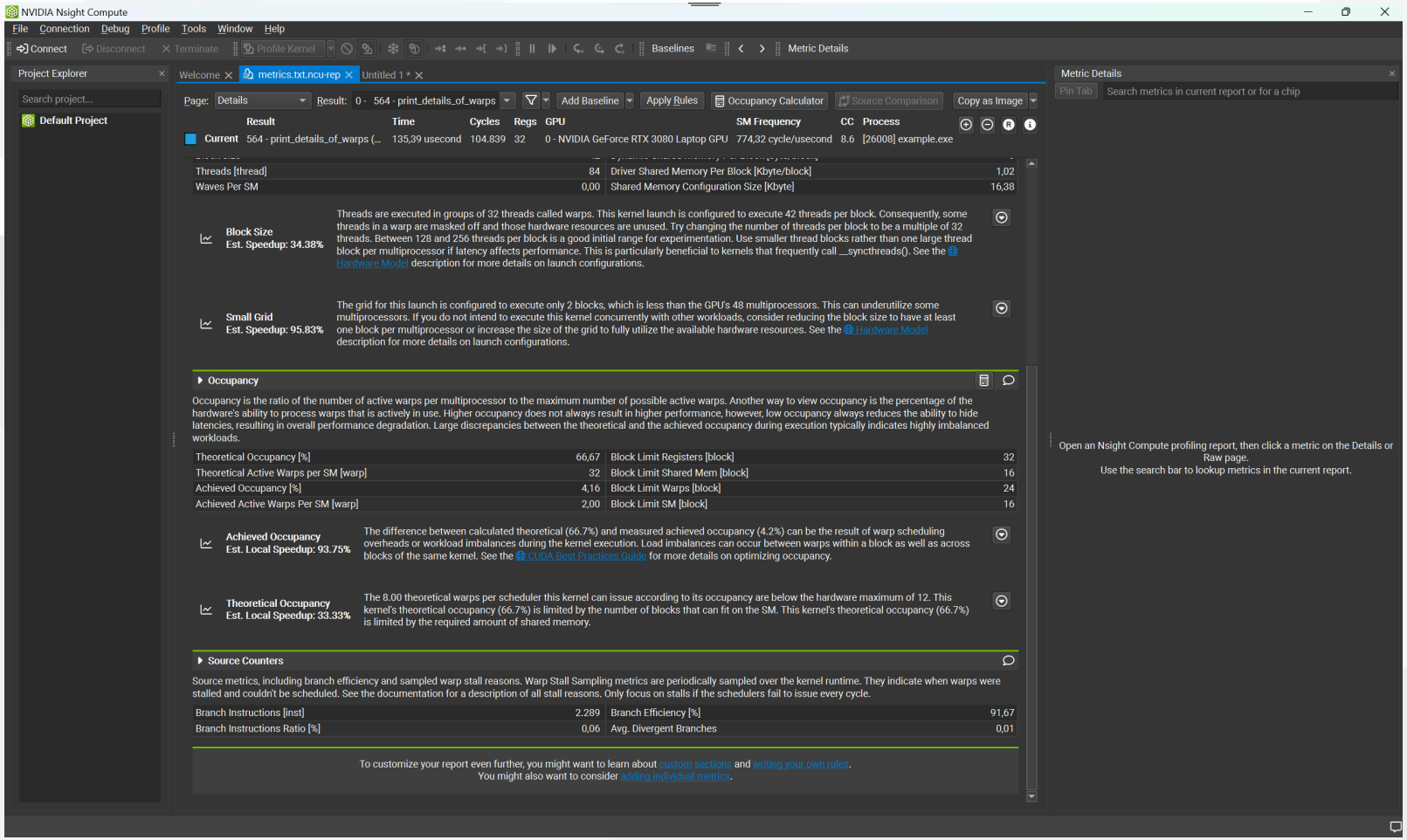
Nsight Compute

- UI
- integrierter Occupancy Calculator
- verschiedene Metriken
- viele weitere Funktionen

nvprof

- Commandline Tool für Cuda Capability < 7.5

Nvidia Nsight Compute



Nvidia nvprof

CUDA_Freshman — tony@tony-Lenovo: ~/Project/CUDA_Freshman/build/4_sum_arrays_timer — ssh tony@192.168.3.19 — 123×28

```
/usr/local/cuda/bin/nvprof
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/4_sum_arrays_timer$ sudo /usr/local/cuda/bin/nvprof ./sum_arrays_timer
==25442== NVPROF is profiling process 25442, command: ./sum_arrays_timer
Using device 0: GeForce GTX 1050 Ti
Vector size:16777216
Execution configuration<<<16384,1024>>> Time elapsed 0.002269 sec
Check result success!
==25442== Profiling application: ./sum_arrays_timer
==25442== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   61.43%  26.535ms         2  13.268ms  13.181ms  13.354ms  [CUDA memcpy HtoD]
                   33.70%  14.557ms         1  14.557ms  14.557ms  14.557ms  [CUDA memcpy DtoH]
                    4.86%  2.1011ms         1  2.1011ms  2.1011ms  2.1011ms  sumArraysGPU(float*, float*, float*, int)
      API calls:   72.81%  131.50ms         3  43.832ms  293.94us  130.89ms  cudaMalloc
                   22.99%  41.510ms         3  13.837ms  13.326ms  14.742ms  cudaMemcpy
                    2.49%  4.4985ms         3  1.4995ms  237.61us  2.1331ms  cudaFree
                    1.24%  2.2402ms         1  2.2402ms  2.2402ms  2.2402ms  cudaDeviceSynchronize
                    0.20%  365.47us        94  3.8870us     125ns  160.97us  cuDeviceGetAttribute
                    0.19%  352.14us         1  352.14us  352.14us  352.14us  cudaGetDeviceProperties
                    0.03%  55.473us         1  55.473us  55.473us  55.473us  cuDeviceTotalMem
                    0.02%  39.777us         1  39.777us  39.777us  39.777us  cuDeviceGetName
                    0.01%  23.903us         1  23.903us  23.903us  23.903us  cudaLaunch
                    0.00%  4.5850us         1  4.5850us  4.5850us  4.5850us  cudaSetDevice
                    0.00%  1.5450us         3     515ns     135ns  1.0970us  cuDeviceGetCount
                    0.00%  1.2670us         4     316ns     131ns     657ns  cudaSetupArgument
                    0.00%  1.0350us         1  1.0350us  1.0350us  1.0350us  cudaConfigureCall
                    0.00%     718ns         2     359ns     182ns     536ns  cuDeviceGet
tony@tony-Lenovo:~/Project/CUDA_Freshman/build/4_sum_arrays_timer$
```

Matrixmultiplikation: Anpassung der Blockgröße

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "time.h"

#define TILE_DIM 16; // Tile Dimension (16 * 16 = 256) Optimale Blockgröße

__device__ float multiply(float a, float b) { return a * b; }

__global__ void mat_mul(float *matA, float *matB, float *res, int size) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
         float sum = 0.0;
        for (int i = 0; i < TILE_DIM; i++) {
            sum += multiply(matA[threadIdx.y][i], matB[i][threadIdx.x]);
        }

        res[row * N + col] = sum;
    }
}

// fill Matrix unverändert
void fillMatrix(float *matrix, int width, int height) {...}
```

Matrixmultiplikation: Anpassung der Blockgröße

```
int main(void) {
    int matSize = 1024; // Neue größe Teilbar durch 32 - vermeidung von überprüfung im Kernel
    int matSizeBytes = matSize * matSize * sizeof(float);

    srand(time(NULL));

    float *h_matA, *h_matB, *h_res, *d_matA, *d_matB, *d_res;

    h_matA = (float *)malloc(matSizeBytes);
    h_matB = (float *)malloc(matSizeBytes);
    h_res = (float *)malloc(matSizeBytes);

    if (h_matA == NULL || h_matB == NULL || h_res == NULL) { fprintf(stderr, "Speicherzuweisung fehlgeschlagen.\n");
        return 1;
    }
    fillMatrix(h_matA, matSize, matSize);
    fillMatrix(h_matB, matSize, matSize);

    cudaMalloc((void**)&d_matA, matSizeBytes); // Speicherallokation auf dem Device
    cudaMalloc((void**)&d_matB, matSizeBytes);
    cudaMalloc((void**)&d_res, matSizeBytes);
    ...
```

Matrixmultiplikation: mit Tiling

```
    ...
    cudaMemcpy(d_matA, h_matA, matSizeByte, cudaMemcpyHostToDevice); // Von Host zu Device
    cudaMemcpy(d_matB, h_matB, matSizeByte, cudaMemcpyHostToDevice);

    dim3 blockSize(TILE_DIM,TILE_DIM);
    dim3 gridSize((matSize + blockSize.x-1)/blockSize.x, (matSize + blockSize.y-1)/blockSize.y);

    mat_mul <<<gridSize, blockSize>>>(d_matA, d_matB, d_res, matSize);

    cudaDeviceSynchronize();
    cudaMemcpy(h_res, d_res, matSizeByte, cudaMemcpyDeviceToHost); // Von Device zu Host
    cudaFree(d_matA); cudaFree(d_matB); cudaFree(d_res); // Speicherfreigabe Device
    free(h_matA); free(h_matB); free(h_res); // Speicherfreigabe Host
    cudaDeviceReset();
}
```

Das CUDA Memory Model

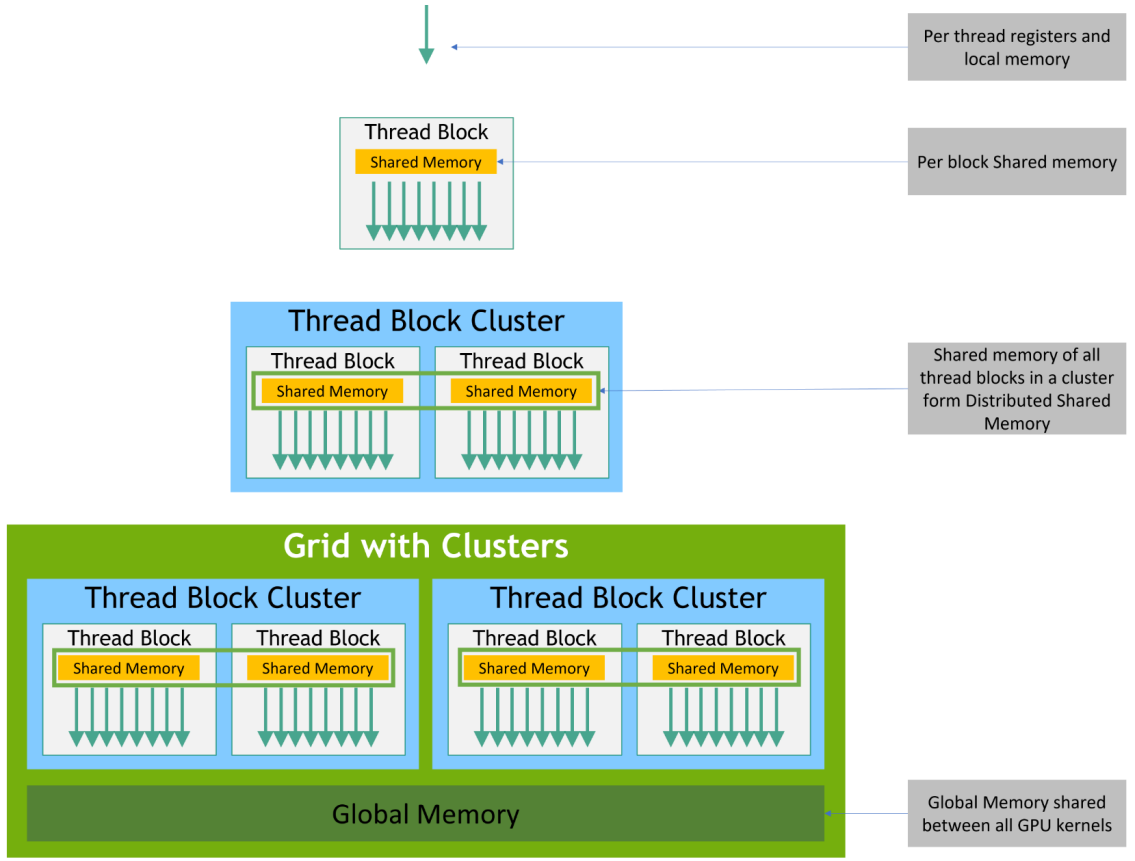
Speicherhierarchien:

- **local memory & registers (thread)**
- **shared memory (thread block)**
- distributed shared memory (thread block clusters) (NVIDIA Hopper Architektur)
- **global memory (shared between all gpu kernels)**

NVIDIA Hopper Architecture

Thread Block Clusters als neue Abstraktionsebene

- Einführung von DSMEM innerhalb von Thread Block Clustern



Local- vs Shared- vs Global-Memory

Memory Type	Location	Cached	Access	Scope	Lifetime
Register	On-chip	n/a	R/W	1 thread	Thread
Local	Off-chip	Yes*	R/W	1 thread	Thread
Shared	On-chip	n/a	R/W	All threads in block	Block
Global	Off-chip	*	R/W	All threads + host	Host allocation

- Implementierung Abhängig von CC

Local- vs Shared- vs Global-Memory

Speicherzugriffszeit:

- registers (wenige Taktzyklen)
- shared memory (einige dutzend Taktzyklen)
- global memory (einige hundert Taktzyklen, bei Cache Miss)
- local memory (einige hundert Taktzyklen, bei Cache Miss)

Besondere Speicherbereiche

- Zero Copy Memory
- Constant Memory
- Texture Memory

Zero Copy Memory

- Liegt im Host Speicher
- Ist sowohl von GPU als auch CPU addressierbar
- Vermeidung von Overhead durch das kopieren vieler (kleiner) Daten

Zugriff langsamer als Device Speicher (über PCIe für externe GPU)

```
cudaHostAlloc((void**)&hostPtr, size, cudaHostAllocMapped);
cudaHostGetDevicePointer(&devicePtr, hostPtr, 0);

myKernel<<<blocks, threads>>>(devicePtr);
```

Constant Memory

- Liegt im globalen Speicher
- Zugriffe werden durch Constant-Cache beschleunigt

Speicher für konst. Daten welche von vielen Threads gelesen werden.

```
__constant__ float constData[256];

__global__ void myKernel(float *data) {
    int i = threadIdx.x;
    float val = constData[i];
}
// Kopieren von Daten in den Constant Memory
cudaMemcpyToSymbol(constData, hostData, sizeof(float) * 256);
```

Texture Memory

- Nutzt Texture Cache, optimiert für räumlich lokale Speicherzugriffe
- Hardware unterstützte Interpolation

Häufig eingesetzt in der Bildverarbeitung

```
texture<float, cudaTextureType1D, cudaReadModeElementType> texRef;

__global__ void myKernel(float *output, int size) {
    int i = threadIdx.x;
            output[i] = tex1Dfetch(texRef, i);
}
// Binden des globalen Speichers an den Texture Reference
cudaBindTexture(NULL, texRef, deviceData, sizeof(float) * size);
```

Tiling

Prinzip aus der Computergrafik

- Aufteilung größerer Datenmenge in kleinere unabhängig zu bearbeitende Datensätze

In unserem Fall:

- Aufteilung größerer Datenmengen in Tiles
- Effiziente Nutzung des Shared Memory Tiles werden in Shared Memory geladen
- Verringerung der Zugriffe auf den global memory

Performance Optimierung mit Tiling

- Im Folgenden Laden der Daten aus dem globalen Speicher in den shared memory
- Synchronisierung aller threads mittels `__syncthreads()` um Fehlerhafte berechnungen zu vermeiden
- Verwendung des Shared Memory anstelle des globalen Speichers

Matrixmultiplikation: Mit Tiling

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "time.h"

#define TILE_DIM 16; // Tile Dimension (16 * 16 = 256) Optimale Blockgröße

__device__ float multiply(float a, float b) { return a * b; }

__global__ void mat_mul(float *matA, float *matB, float *res, int size) {
    __shared__ float aTile[TILE_DIM][TILE_DIM], bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
         float sum = 0.0;
        aTile[threadIdx.y][threadIdx.x] = matA[row*TILE_DIM+threadIdx.x]; // Laden der globalen Daten in die Tiles
        bTile[threadIdx.y][threadIdx.x] = matB[threadIdx.y*N+col];
        __syncthreads(); // Wichtig! Synchronisierung nach Speicheroperationen!

        for (int i = 0; i < TILE_DIM; i++) {
            sum += multiply(aTile[threadIdx.y][i], bTile[i][threadIdx.x]);
        }

        res[row * N + col] = sum;
    }
}

// fill Matrix unverändert
void fillMatrix(float *matrix, int width, int height) {...}
```

Matrixmultiplaktion: Mit Tiling

```
int main(void) {
    int matSize = 1024; // Neue größe Teilbar durch 32 - vermeidung von überprüfung im Kernel
    int matSizeBytes = matSize * matSize * sizeof(float);

    srand(time(NULL));

    float *h_matA, *h_matB, *h_res, *d_matA, *d_matB, *d_res;

    h_matA = (float *)malloc(matSizeBytes);
    h_matB = (float *)malloc(matSizeBytes);
    h_res = (float *)malloc(matSizeBytes);

    if (h_matA == NULL || h_matB == NULL || h_res == NULL) { fprintf(stderr, "Speicherzuweisung fehlgeschlagen.\n");
        return 1;
    }
    fillMatrix(h_matA, matSize, matSize);
    fillMatrix(h_matB, matSize, matSize);

    cudaMalloc((void**)&d_matA, matSizeBytes); // Speicherallokation auf dem Device
    cudaMalloc((void**)&d_matB, matSizeBytes);
    cudaMalloc((void**)&d_res, matSizeBytes);
    ...
```

Matrixmultiplikation: Mit Tiling

```
    ...
    cudaMemcpy(d_matA, h_matA, matSizeByte, cudaMemcpyHostToDevice); // Von Host zu Device
    cudaMemcpy(d_matB, h_matB, matSizeByte, cudaMemcpyHostToDevice);

    dim3 blockSize(TILE_DIM,TILE_DIM);
    dim3 gridSize((matSize + blockSize.x-1)/blockSize.x, (matSize + blockSize.y-1)/blockSize.y);

    mat_mul <<<gridSize, blockSize>>>(d_matA, d_matB, d_res, matSize);

    cuda_t status = cudaGetLastError();

    if (status != cudaSuccess) {
        return fprintf(stderr, "Kernel Launch failed: %s\n", cudaGetErrorString(status));
    }

    cudaDeviceSynchronize();
    cudaMemcpy(h_res, d_res, matSizeByte, cudaMemcpyDeviceToHost); // Von Device zu Host
    cudaFree(d_matA); cudaFree(d_matB); cudaFree(d_res); // Speicherfreigabe Device
    free(h_matA); free(h_matB); free(h_res); // Speicherfreigabe Host
    cudaDeviceReset();
}
```

Best Practices

1. Vermeidung von **Warp Divergenz** ***
2. **Profiling** der Anwendung zum Aufspüren von Bottlenecks & Hotspots ***
3. Auslagern schwer zu parallelisierenden Codes auf den Host ***
4. Vermeidung von unnötigen **Datentransfers** zwischen Host und Device ***
5. Verwendung von **Shared Memory** um unnötige Zugriffe auf Global Memory zu vermeiden**

Performancevergleich CPU vs. GPU

Bilderverzeichnis

Grid
https://docs.nvidia.com/cuda/cuda-c-programming-guide/_images/grid-of-thread-blocks.png
nvprov
https://face2ai.com/CUDA-F-2-2-核函数计时/
Speicherhierarchie
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy
Best Practices

Quellenverzeichnis

Threadorganization
https://docs.nvidia.com/cuda/cuda-c-programming-guide/
SIMT vs SIMD
https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html
Speicherhierarchie
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
Tiling
https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Quellenverzeichnis