Architektur und Programmierung von Grafik- und Koprozessoren

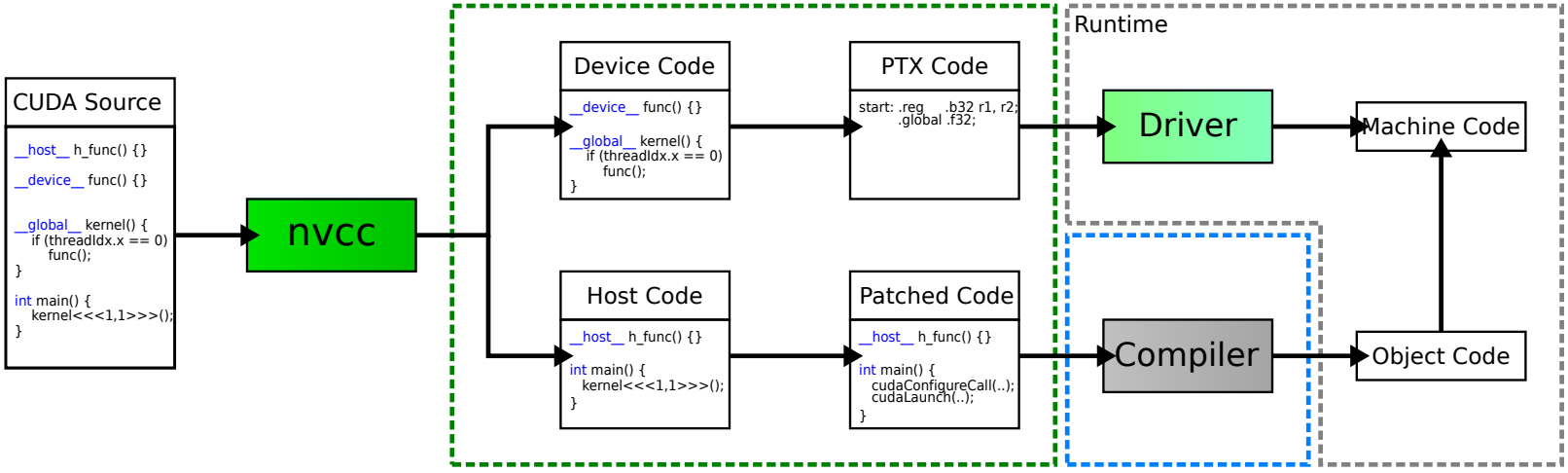
General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Kompilieren mit nvcc und CUDA Runtime API



1. nvcc verarbeitet `.cu` Dateien mit Host- und Device Code
   - Device Code $\Rightarrow$ PTX (Nvidias GPU ISA).
   - Host Code $\Rightarrow$ gepatchter Host Code, Routinen zum Laden von PTX und Aufruf von GPU Kernels.
2. Runtime: Treiber kompiliert PTX in Chip-spezifischen Maschinencode.
3. Host Programm ruft GPU Maschinencode auf.

Kompilieren mit nvcc und CUDA Runtime API

- Device Compiler Teil von nvcc übersetzt nach PTX (und alternativ nach cubin).
- Treiber übersetzt *zur Laufzeit* PTX Code in Maschinencode (just-in-time (JIT) compilation).
  - Maschinencode wird im User-Verzeichnis gecached, JIT nur bei erstem Programmstart nach Rekompilieren.
  - JIT Compiler kann dediziert für Target-Platform / GPU Architektur optimieren.
- Übersetzen nach cubin $\Rightarrow$ PTX Code editierbar, kann nach nvcc Lauf handoptimiert werden.
- PTX unterstützt 64-bit. Entweder gesamte Toolchain (Host & Device) 32-bit oder 64-bit.

CUDA Programmiermodell

Single Instruction Multiple Thread (SIMT). Ähnlich wie SIMD, jedoch nicht *explizit*.

Keine expliziten SIMD Instruktionen. Thread Funktionen (Kernels) exponieren Instruktionsfluss eines einzelnen Threads.

Nvidia PTX ISA: keine SIMD opcodes (anders als AMD GPUs).

Implizit: alle Threads in Warp führen die gleichen Instruktionen aus. Betritt ein Thread aus der Warp einen Branch (`if..else`), warten alle anderen Threads inaktiv.

CUDA Programmiermodell

Auf einem SM laufen i. d. R. mehrere Warps gleichzeitig. Alle Threads/Warps auf SM haben geteilten (on-chip) Speicher (*shared memory*) (siehe CUDA Speichermodell) $\Rightarrow$ Barrier Synchronisation einzelner Warps.

Kernels werden von CPU aus angestoßen. CPU initiiert auch Speichertransfers. Kernels und andere Operationen asynchron.

CUDA Kernels

CUDA Kernels

Spracherweiterung / Aufrufsyntax Kernels

CUDA Compiler erweitert C++ um spezielle Syntax, um Compute Kernels aufzurufen.

```
__global__ void kernel()
{
}

int main()
{
    kernel<<<num_blocks, num_threads>>>();
}
```

Kernel ist Einstiegspunkt des GPU Programms (ähnlich wie `main()` für Host).

Programmausführung kann von dort weiter verzweigen.

Globale Funktion `kernel()` wird von (*#Blöcke* $\times$ *#Threads pro Block*) Threads ausgeführt.

CUDA Kernels

1D Grid, 1 Block, N Threads

```
__global__ void saxpy(float* a, float* x, float* y)
{
    int i = threadIdx.x;
    y[i] = a[i] * x[i] + y[i];
}

int main()
{
    ...
    saxpy<<<1, N>>>(a, x, y);
    ...
}
```

CUDA Kernels

2D Grid, 1 Block, N × N Threads

```
__global__ void matrix_add(float** a, float** b, float** c)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    c[i][j] = a[i][j] + b[i][j];
}

int main()
{
    ...
    dim3 num_threads(N, N);
    matrix_add<<<1, num_threads>>>(a, x, y);
    ...
}
```

CUDA Kernels

2D Grid unterteilt in Blöcke

```
__global__ void matrix_add(float** a, float** b, float** c)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    c[i][j] = a[i][j] + b[i][j];
}

int main()
{
    ...
    // N/32 Bloecke in x-Richtung, M/32 in y-Richtung
    dim3 threads_per_block(32, 32);
    dim3 blocks(round_up(N / threads_per_block.x),
                round_up(M / threads_per_block.y));
    matrix_add<<<blocks, threads_per_block>(a, x, y);
    ...
}
```

CUDA Kernels

Optimale Blockgröße

Das ermitteln der optimalen Blockgröße hängt von Registerzahl ab, die Kernel benötigt.

Übersetze Kernel mit `nvcc` Option `-Xptxas -v`:

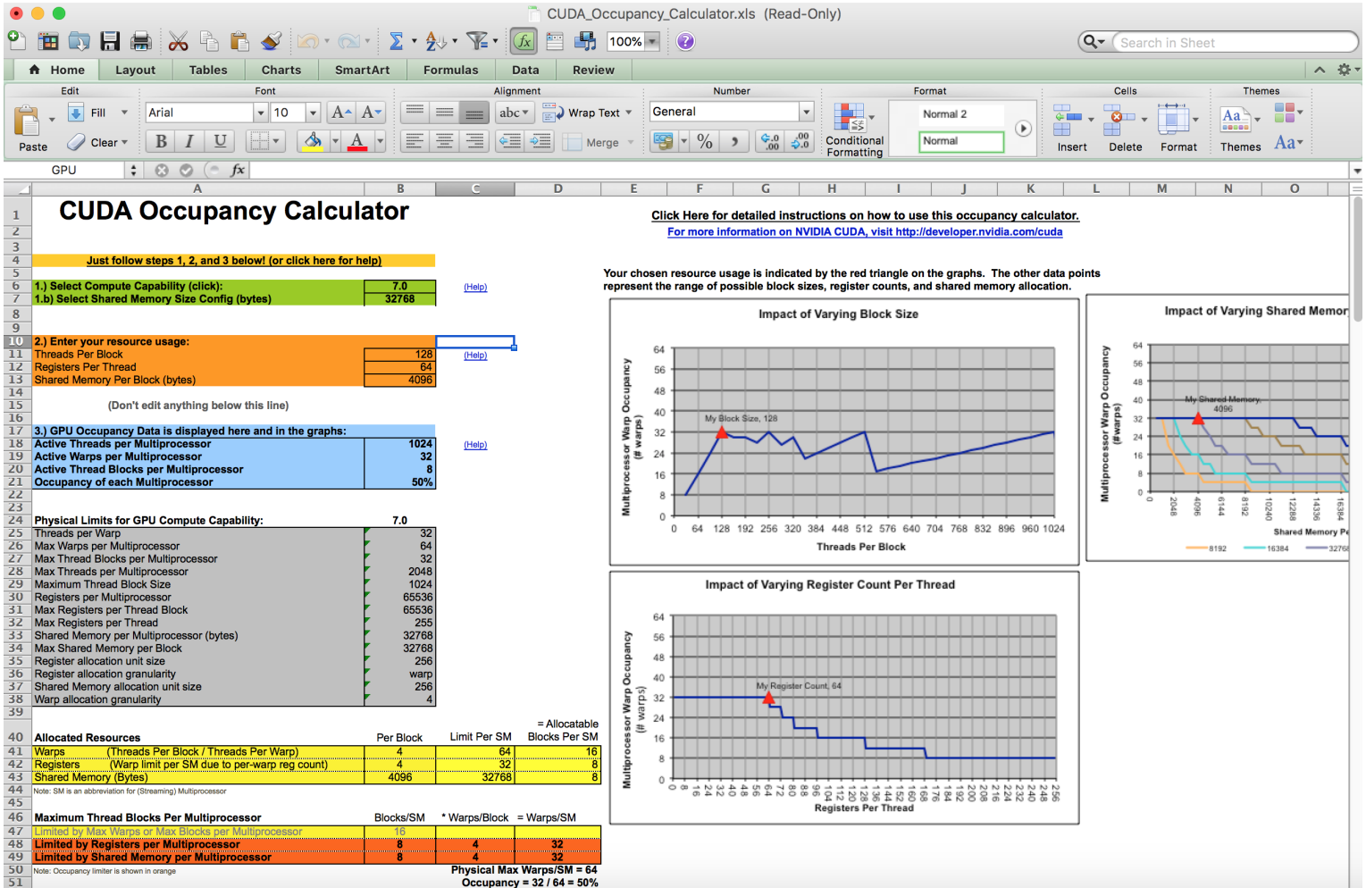
```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6kernelPfi' for 'sm_20'
ptxas info    : Function properties for _Z6kernelPfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 20 registers, 44 bytes cmem[0], 36 bytes cmem[16]
```

Kepler GK110, Maxwell, Pascal: 64k 32-bit Register File, max. 255 Register pro Thread.

CUDA Kernels

Optimale Blockgröße

Bestimme Blockgröße mit CUDA_Occupancy_Calculator.xls



CUDA Kernels

Threads identifizieren

In CUDA Kernels stehen eingebaute Variablen

`gridDim.{x|y|z}`
`blockIdx.{x|y|z}`
`blockDim.{x|y|z}`
`threadIdx.{x|y|z}`

zur Verfügung. `gridDim` gibt Anzahl der Blöcke an. Über `blockIdx` wird Block identifiziert, über `blockDim` dessen Dimensionen und über `threadIdx` der *lokale* Thread Index im Block.

Der globale Thread Index kann mittels

```
unsigned i = blockIdx.x * blockDim.x + threadIdx.x;
unsigned j = blockIdx.y * blockDim.y + threadIdx.y;
unsigned k = blockIdx.z * blockDim.z + threadIdx.z;
```

ausgerechnet werden.

CUDA Kernels

- Single Source Modell: Host Code und Kernel Code können in der gleichen Compilation Unit aufgeführt werden, teilen sich `#include` Direktiven etc.
  - vgl. “Kompilieren mit nvcc”, Code wird später von CUDA Compiler aufgeteilt.
- Reduzierter C++ Sprachumfang in Kernels (Speicherallokation, Rekursion etc. nicht oder nur eingeschränkt unterstützt).
- Funktionen können “wiederverwendet” werden: `__host__` & `__device__` Funktionen.

CUDA Kernels

Funktionsannotation

```
// Function can only be used on CPU
float dot(vec3 u, vec3 v) {
    return u.x * v.x + u.y * v.y + u.z * v.z;
}

// Function can only be used on CPU
__host__ float dot(vec3 u, vec3 v) {
    return u.x * v.x + u.y * v.y + u.z * v.z;
}

// Function can only be used on GPU
__device__ float dot(vec3 u, vec3 v) {
    return u.x * v.x + u.y * v.y + u.z * v.z;
}

// Function can be used on CPU and GPU
__host__ __device__ float dot(vec3 u, vec3 v) {
    return u.x * v.x + u.y * v.y + u.z * v.z;
}
```

CUDA Kernels

Inlining

Herkömmlicher Funktionsaufruf (z. B. CPU) $\Rightarrow$ Funktions-Stack, Argumente etc. werden in speziellen Registern gespeichert, werden, wenn Funktion zurückkehrt, wieder freigegeben.

GPU und notorischer Registermangel $\Rightarrow$ Funktionen fast immer inline.

Daher schlechter bis kein Support für Rekursion auf GPUs (würde zu unkontrollierbarer Rekursionstiefe führen).

CUDA Kernels

Generelle Empfehlungen

1.) Vermeide Divergenz, da Threads in Warp bei dynamischem Branching aufeinander warten.

2.) Koaleszierende Speicherzugriffe: Threads sollten immer von alignierten Speicheradressen lesen.

3.) Wegen Speicherzugriffslatenz: oft lohnt es sich, **Berechungen immer wieder** zu **wiederholen**, **anstatt** Ergebnisse **zwischenzuspeichern**.

4.) Einmal auf der GPU, vermeide Kommunikation mit Host und führe **möglichst viele Berechnungen in Kernel** aus.

CUDA Runtime API

CUDA Runtime Funktionen

Neben Kernels, die auf der GPU ausgeführt werden, steht Runtime Bibliothek zur Verfügung, die das Interface zwischen Host und Device steuert.

CUDA Runtime Funktionen haben Präfix `cuda`, z. B. `cudaMalloc()`, `cudaMemcpy()` etc.

CUDA Runtime Funktionen

Runtime Initialisierung

Runtime wird *implizit* initialisiert. Erster CUDA Funktionsaufruf initialisiert Runtime.

- Runtime erstellt *CUDA Kontext* für jede installierte, CUDA-kompatible GPU.
- JIT Compilation und Laden von Device Code in GPU Speicher bei Kontexterzeugung.
- Kontext wird von allen CPU-Threads geteilt!
- `cudaDeviceReset()` zerstört aktuellen Kontext, nächster Runtime Funktionsaufruf erstellt neuen Kontext.

Fehlerbehandlung

Die meisten CUDA Runtime Funktionen geben Fehlercode zurück:

```
cudaError_t err = cudaGetDeviceCount(...);
if (err != cudaSuccess) {
    ...
}
```

`cudaGetLastError()` und `cudaPeekLastError()`: prüfe ob Kernel Fehler ausgelöst hat.

```
// Get last error, reset to cudaSuccess
cudaError_t err = cudaGetLastError();

// Peek last error w/o reset
cudaError_t err = cudaPeekLastError();
```

Fehlerbehandlung

Achtung, Kernels werden *asynchron* ausgeführt. `LastError` bezieht sich womöglich nicht auf den richtigen Kernel $\Rightarrow$ füge Synchronisation / Barrier ein:

```
// Call kernel
kernel<<<1,N>>>(params);

// Errors due to invalid kernel configuration (<<<...>>>)
cudaError_t err1 = cudaGetLastError();

// Kernel errors (e.g. out-of-bounds memory access)
cudaError_t err = cudaDeviceSynchronize();
```

Derartig synchronisierte Kernels laufen nicht mehr asynchron. Fehlerbehandlung sollte nur im Debug Mode durchgeführt werden.

`cudaDeviceSynchronize()` ist generelles Synchronisationsprimitiv (falls Applikation Synchronisation erfordert).

Globaler Speicher

cudaMalloc und cudaFree

- Wie in C/C++ muss DDR Speicher reserviert und später wieder freigegeben werden.
- *Device Zeiger*: deklariere C-style raw pointer, weise mit cudaMalloc Adresse aus GPU Adressraum zu. Dieser Pointer steht auf dem Host zur Verfügung, kann aber nur auf dem Device dereferenziert werden.

```
int* d_pointer;
constexpr int N = 32;
cudaMalloc(&d_pointer, sizeof(int) * N);
...
cudaFree(d_pointer);
```

Speichertransfers

- Bidirektionale Speichertransfers zwischen Host / Device (VRAM) und Speichertransfers zwischen VRAM Speicherbereichen: `cudaMemcpy()`.
- Aufruf Semantik wie ANSI-C `memcpy`, mit viertem Parameter, der Richtung angibt.
  - `enum cudaMemcpyKind`: `cudaMemcpyHostToHost` (wie `memcpy()`), `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`.
- VRAM wird mit Device Zeiger referenziert.

```
int h_array = { 0, 1, 2, 3, 4 };
int* d_array;
cudaMalloc(&d_array, sizeof(int) * 5);
cudaMemcpy(d_array, h_array, sizeof(int) * 5,
           cudaMemcpyHostToDevice);
```

Speichertransfers

CUDA Saxpy

(1/2)

```
__global__ void saxpy(float* a, float* x, float* y)
{
    int i = threadIdx.x;
    y[i] = a[i] * x[i] + y[i];
}

int main()
{
    constexpr int N = ..;
    float *h_a, *h_x, *h_y;
    // ... init h_a, h_x, and h_y

    float *d_a, *d_x, *d_y;
    size_t S = N * sizeof(float);
    cudaMalloc(&d_a, S);
    cudaMalloc(&d_x, S);
    cudaMalloc(&d_y, S);
```

Speichertransfers

CUDA Saxpy

(2/2)

```
    ...

    cudaMemcpy(d_a, h_a, S, cudaMemcpyHostToDevice);
    cudaMemcpy(d_x, h_x, S, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, S, cudaMemcpyHostToDevice);

    saxpy<<<1, N>>>(a, x, y);

    cudaMemcpy(h_a, d_a, S, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_x, d_x, S, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_y, d_y, S, cudaMemcpyDeviceToHost);

    cudaFree(h_a); cudaFree(h_x); cudaFree(h_y);

    ...
}
```

Shared Memory

- Steht limitiert (16 kb, 64 kb etc.) allen gemeinsam ausgeführten Threads auf SM zur Verfügung.
- Keine direkte Adressierung via Zeiger, sondern spezielle Syntax in Kernel (CUDA Keyword `__shared__`).
- Zwei Arten von Allokation: *statisch* und *dynamisch*.

Shared Memory

Statische Allokation

```
__global__ void kernel(int* data)
{
    // Static size shared memory
    __shared__ int shared_ints[64];

    // Access with local thread ID
    shared_ints[threadIdx.x]
        = data[blockIdx.x * blockDim.x + threadIdx.x];

    // Access to shared memory must be synchronized
    __syncthreads();

    // Now access low-latency memory
    ...
}
```

Shared Memory

Dynamische Allokation

```
__global__ void kernel(int* data)
{
    // Shared memory, don't specify size
    __shared__ int shared_ints[];
}

void call_kernel() {
    // Specify variable shared memory size
    // when calling kernel (must still adhere
    // to platform limits!)
    kernel<<<
        num_blocks,
        threads_per_blocks,
        shared_memory_size // <<<
        >>>(data);
}
```