

Architektur und Programmierung von Grafik- und Koprozessoren

General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

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* × *#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 * x[i] + y[i];
}

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

CUDA Kernels

2D Grid, 1 Block, $N \times N$ 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, b, c);
    ...
}
```

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 Bloেকে 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_Occupancy_Calculator.xls (Read-Only)

Click Here for detailed instructions on how to use this occupancy calculator.
 For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>

Just follow steps 1, 2, and 3 below (or click here for help)

1) Select Compute Capability (click): 7.0 (280)
 1a) Select Shared Memory Size (Config Bytes): 32768

2) Enter your resource usage:
 Threads Per Block: 128 (280)
 Registers Per Thread: 32
 Shared Memory Per Block (bytes): 4096

(Don't edit anything below this line)

3) GPU Occupancy Data is displayed here and in the graphs:
 Active Threads per Multiprocessor: 1634 (280)
 Active Warps per Multiprocessor: 32
 Active Thread Blocks per Multiprocessor: 8
 Occupancy of each Multiprocessor: 50%

Physical Limits for GPU Compute Capability: 7.0

25	Threads per Warp	32
26	Max Warps per Multiprocessor	64
27	Max Thread Blocks per Multiprocessor	32
28	Max Threads per Multiprocessor	2048
29	Maximum Thread Block Size	1024
30	Registers per Multiprocessor	80036
31	Max Registers per Thread Block	80036
32	Max Registers per Thread	2015
33	Shared Memory per Multiprocessor (bytes)	32768
34	Max Shared Memory per Block	32768
35	Register allocation unit size	256
36	Register allocation granularity	warps
37	Shared Memory allocation unit size	256
38	Warp allocation granularity	4
39		

40 Allocated Resources Per Block Limit Per SM = Allocatable Blocks Per SM

41	Warps (Threads Per Block / Threads Per Warp)	4	64	38
42	Registers (Warp limit per SM due to per-warp reg count)	4	32	8
43	Shared Memory (Bytes)	4096	32768	8

44 Note: SM is an abbreviation for Streaming Multiprocessor

45 Maximum Thread Blocks Per Multiprocessor Blocks/SM * Warps/Block = Warps/SM

47	Limited by Max Warps of SM (Max Blocks per Multiprocessor)	8	4	32
48	Limited by Registers per Multiprocessor	8	4	32
49	Limited by Shared Memory per Multiprocessor	8	4	32

50 Note: Occupancy better to show in orange

Physical Max Warps/SM = 64
 Occupancy = 32 / 64 = 50%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Impact of Varying Block Size

Impact of Varying Shared Memory

Impact of Varying Register Count Per Thread

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, **Berechnungen 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 * x[i] + y[i];
}
```

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

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

Speichertransfers

CUDA Saxpy

(2/2)

...

```
cudaMemcpy(d_x, h_x, S, cudaMemcpyHostToDevice);  
cudaMemcpy(d_y, h_y, S, cudaMemcpyHostToDevice);
```

```
saxpy<<<1, N>>>(a, d_x, d_y);
```

```
// read back result
```

```
cudaMemcpy(h_y, d_y, S, cudaMemcpyDeviceToHost);
```

```
cudaFree(d_x); cudaFree(d_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);
}
```

Shared Memory

- ▶ Shared Memory lohnt sich, wenn Threads aus einer Warp häufig auf Daten der anderen Threads in der Warp zugreifen.
- ▶ Je nach Architektur globaler Speicher gecached \Rightarrow für trivialparallele Applikationen mit einfachen Speicherzugriffsmustern lohnt sich Shared Memory u. Umst. nicht.
- ▶ Matrix Operationen (z. B. DGEMM) profitieren eher von Shared Memory
 - ▶ Kopiere *Block* in Shared Memory und bearbeite lokal.

Shared Memory

Matrix Multiplikation mit Shared Memory

$C = A \times B$ (vgl. NVIDIA Programming Guide.)

```
int row = threadIdx.y, col = threadIdx.x;
float cv = 0.0f;
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    Matrix Asub = GetSubMatrix(A, blockIdx.x, m);
    Matrix Bsub = GetSubMatrix(B, m, blockIdx.y);

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[row][col] = Asub[row][col];
    Bs[row][col] = Bsub[row][col];
    __syncthreads();

    for (int e = 0; e < BLOCK_SIZE; ++e)
        cv += As[row][e] * Bs[e][col];
    __syncthreads();
}
C[row][col] = cv;
```

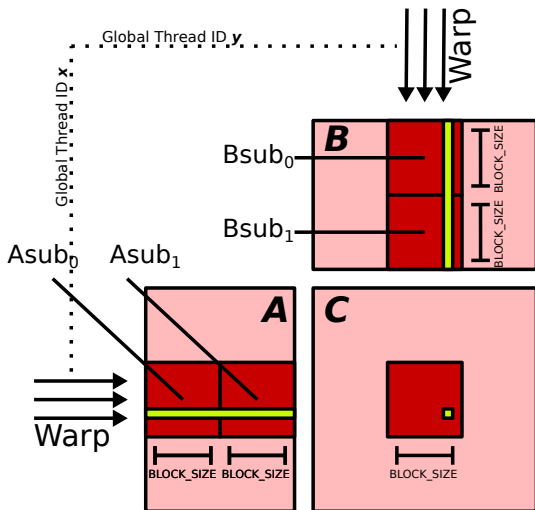
Shared Memory

Matrix Multiplikation mit Shared Memory

- ▶ 2D Grid, $\text{Idx}.x$ für Zeile in Matrix A, $\text{Idx}.y$ für Spalte in Matrix B.
- ▶ Iteriere über *Blockgröße*, kopiere Submatrizen für ganzen Block in Shared Memory.
- ▶ Skalarproduktoperation wird von jedem Thread in Shared Memory durchgeführt.
- ▶ Vor und nach Skalarprodukt müssen Threads synchronisiert werden, damit nicht schon andere Threads aus Warp anderen Block in Shared Memory laden.

Shared Memory

Matrix Multiplikation mit Shared Memory



Texturspeicher

- ▶ Read-Only Speicherbereich.
- ▶ Dedizierter Cache nur für Texturen.
- ▶ Dedizierte Hardware für Filtering (lineare Interpolation).
- ▶ Texturspeicher nicht direkt adressierbar, Verwaltung über spezielle Texturobjekte.

Texturspeicher

- ▶ Seit Nvidia Kepler Architektur (2012): Bindless Texturen und Texture Objects
- ▶ Texturen können relativ flexibel erzeugt werden und müssen nicht mehr explizit an Texture Units gebunden werden.
 - ▶ Vorher: fixe Anzahl an Texturen im Programm.
- ▶ Texturspeicher ist nicht direkt adressierbar (keine Zeiger), stattdessen Zugriffsobjekt.
 - ▶ IHVs halten Organisation des Texturspeichers geheim.

Texturspeicher

Texture Objects

Erzeuge Texture Object in Host Code:

```
cudaTextureObject_t obj;  
cudaCreateTextureObject(&obj, ...);
```

Dedizierte Texturzugriffsfunktionen in Kernel:

```
__global__ void kernel(cudaTextureObject_t obj1D,  
                       cudaTextureObject_t obj2D,  
                       cudaTextureObject_t obj3D)  
{  
    float t1 = tex1D(obj1D, 0.5f);  
    float t2 = tex2D(obj2D, make_float2(0.3f, 0.4f));  
    float t3 = tex3D(obj3D,  
                    make_float3(0.3f, 0.4f, 0.5f));  
}
```

⇒ man kann Texturen nicht direkt adressieren.

Texturspeicher

Konfiguration von Texture Objects

Beim Erzeugen von Texture Objects legt man u. a. fest,

- ▶ welche Dimensionalität die Textur hat (1D, 2D, 3D),
- ▶ welchen Datentyp die gespeicherten Texel haben,
- ▶ wie interpoliert wird (nächster Nachbar; linear),
- ▶ ob Koordinaten normalisiert sind $[0..1)$ oder nicht $[0..Width)$,
- ▶ sowie den Wrap Modus:
 - ▶ Clamp
 - ▶ Wrap
 - ▶ Mirror
 - ▶ Border