Architektur und Programmierung von Grafik- und Koprozessoren

General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

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;
```

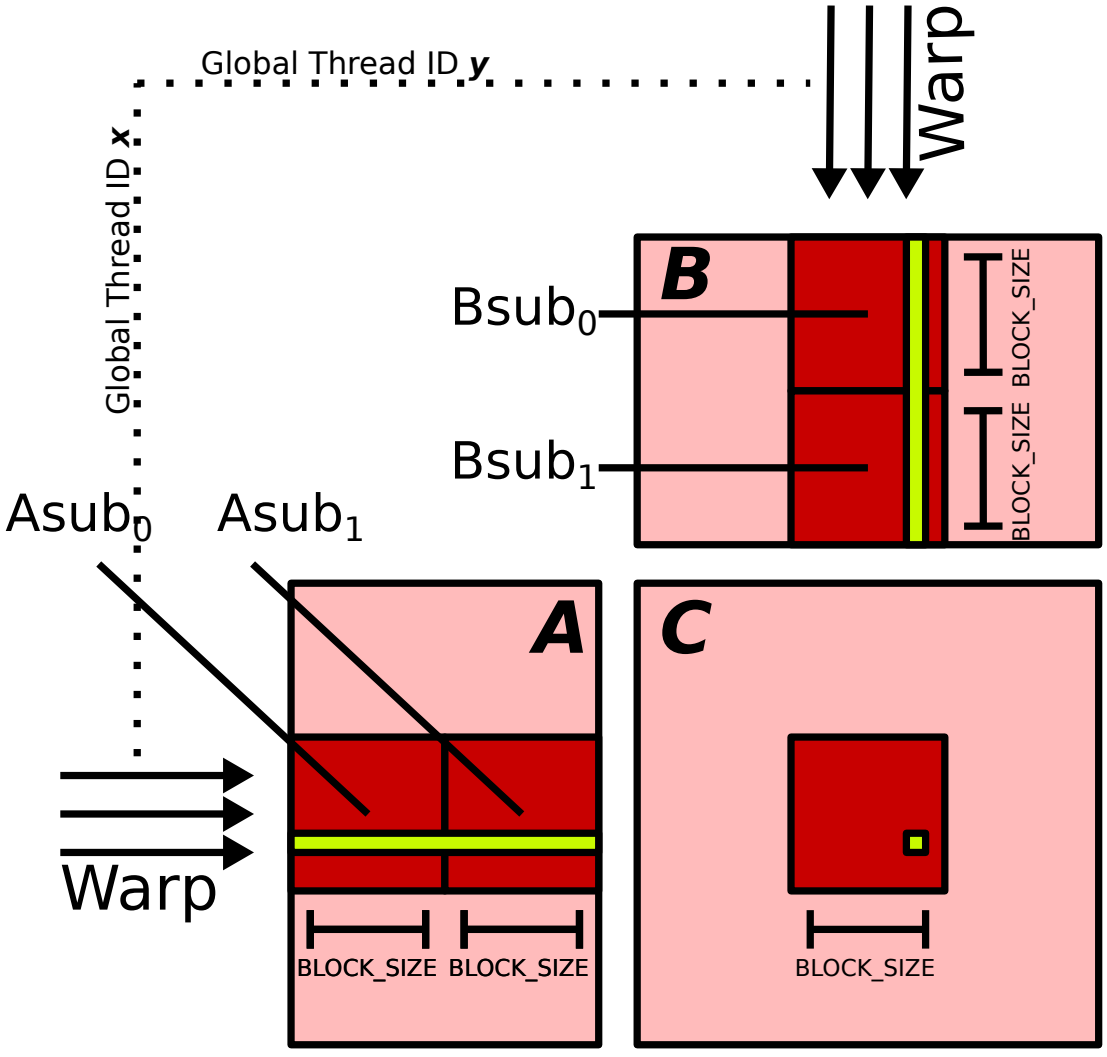
Shared Memory

Matrix Multiplikation mit Shared Memory

- 2D Grid, `Idx.x` für Zeile in Matrix `A`, `Idx.y` für Spalte in Matrix `B`.
- Iteriere über *Blockgröße*, kopiere Submatrizen für ganzen Block in Shared Memory.
- Skalarproduktionoperation 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));
}
```

$\Rightarrow$ 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

Host Interface

Default Queue

- vgl. Vorlesungseinheit 3: GPUs haben paralleles Host Interface mit mehreren Command Queues - häufig mehrere dedizierte Compute Queues.
- Mit Vulkan werden Queues explizit verwaltet und Command Buffer explizit in bestimmte Queue submittiert.
- Mit CUDA werden keine Command Buffer recorded, stattdessen werden Kernels in die *Default Queue* submittiert.

Host Interface

CUDA Streams

- Asynchrone Ausführung von Kernels $\Rightarrow$ CPU Ausführung kehrt zurück / CPU blockiert nicht.
- Kernel werden im Default aber nicht (immer) parallel ausgeführt.
  - teils abhängig von CUDA Version (z. B. seit CUDA 7 “Default Stream per Thread” etc.)
- Explizites submittieren in unterschiedliche Queues mit CUDA Stream Objekten $\Rightarrow$ Nutzung des parallelen Host Interface.

Host Interface

CUDA Streams

Kernel in Default Stream submittieren (explizit):

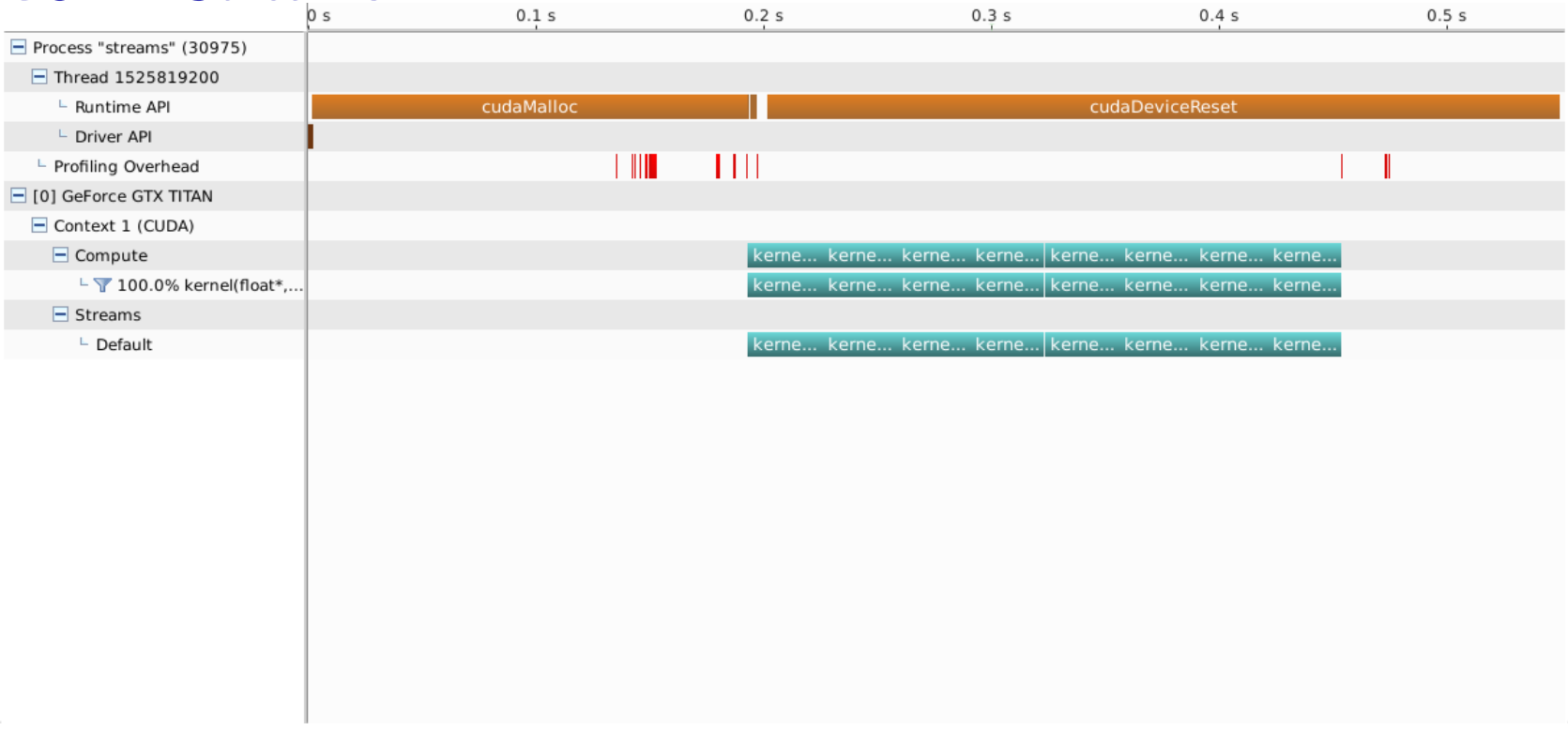
```
kernel<<<num_blocks,
         threads_per_block,
         shared_memory_size,
         0 // default queue/stream: 0
         >>>();
```

Mehrere Kernel in unterschiedliche Streams submittieren:

```
cudaStream_t streams[num_streams];
for (int i = 0; i < num_streams; ++i)
{
    cudaStreamCreate(&streams[i]);
    kernel<<<num_blocks,
             threads_per_block,
             shared_memory_size,
             streams[i]
             >>>();
}
```

Host Interface

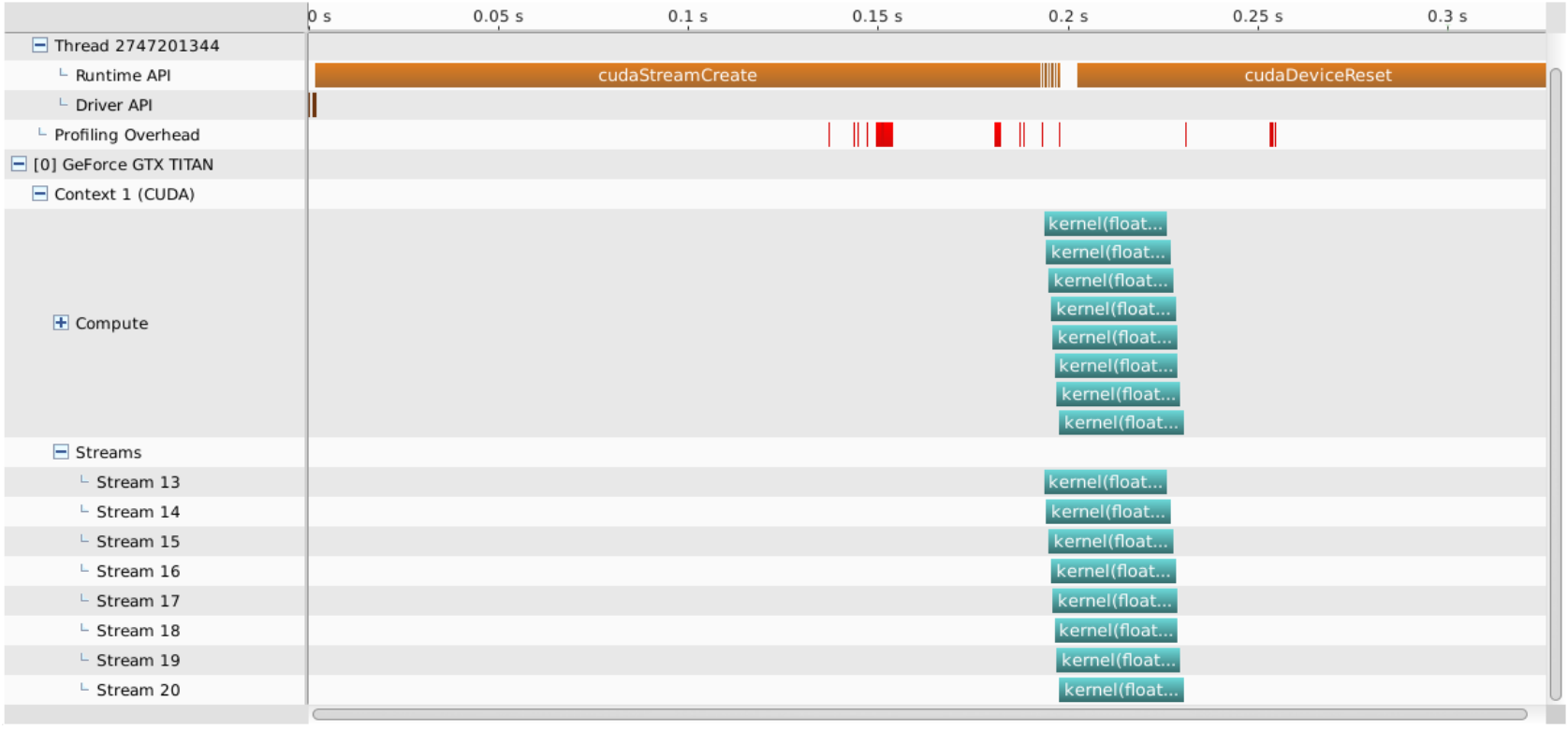
CUDA Streams




```
for (int i = 0; i < 8; ++i) {
    kernel<<<..,..,..,
             0 // default queue/stream: 0
             >>>();
}
```

Host Interface

CUDA Streams




```
for (int i = 0; i < 8; ++i) {
    cudaStreamCreate(&streams[i]);
    kernel<<<..,..,..
             streams[i]
             >>>();
}
```

Host Interface

Synchronisation

Kernels: asynchron, Nebenläufigkeit durch Streams.

`cudaMemcpy()`: blockierend, wartet, bis alle vorherigen CUDA & Kernel Aufrufe zurückgekehrt sind.

`cudaMemcpyAsync()`: kehrt sofort zurück und blockiert die CPU nicht.

`cudaDeviceSynchronize()`: explizite Device Synchronisation.