Architektur und Programmierung von Grafik- und Koprozessoren

General Purpose Programmierung auf Grafikprozessoren

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

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 ⇒ 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 ⇒ Nutzung des parallelen Host Interface.

Host Interface

CUDA Streams

Kernel in Default Stream submittieren (explizit):

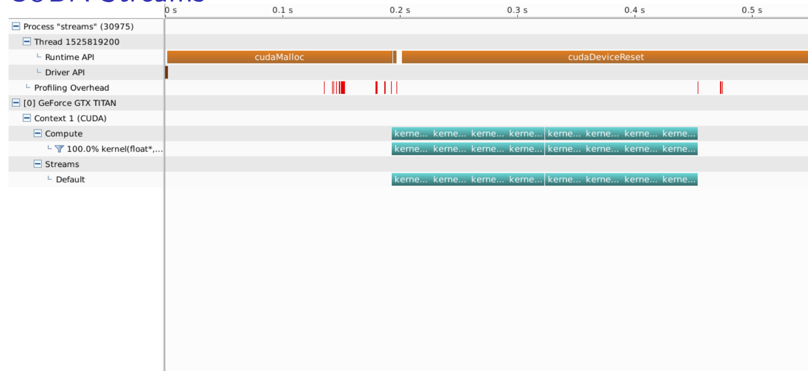
```
kernel<<<num_blocks,
         threads_per_block,
         shared_memory_size,
         0 // default queue/stream: 0
         >>>();
```

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

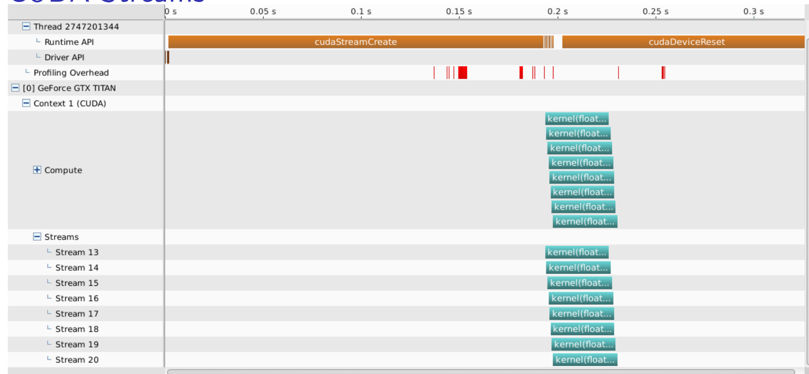
CUDA Streams




```
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.

Host Interface

Ausführungszeit Messen

- Problem: Kernel Aufrufe sind asynchron / CPU Ausführung kehrt sofort zurück ⇒ kein Profiling mit konventionellen C/C++ Mitteln.
- Einfache Lösung: `cudaDeviceSynchronize()` nach Kernelaufruf.
  - Das *stalled* die ganze Pipeline dieser GPU (meistens unerwünscht).
- Besser: CUDA Event API.

Host Interface

Ausführungszeit Messen

Mit CUDA Event API können Events (ähnlich wie Tokens, vgl. Vorlesungsteil zu GPU Architektur) in den Command Stream eingefügt werden.

Registriere erstes Event *vor* Kernel Aufruf, registriere zweites Event *nach* Kernel Aufruf. Synchronisation nur bzgl. dieses Events, messe dann Zeit zwischen den Events mit `cudaEventElapsedTime()`.

Host Interface

Ausführungszeit Messen

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<..,..>>>();
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float elapsed = 0.0f;
cudaEventElapsedTime(&elapsed, start, stop);

cudaEventDestroy(stop);
cudaEventDestroy(start);
```

Host Interface

Ausführungszeit Messen

Mit CUDA Event API können auch andere asynchrone CUDA Funktionen (z. B. `cudaMemcpyAsync()`) geprofiled werden.

Für Profiling *zur Entwicklungszeit* empfehlen sich die CUDA Profiling Tools (Visual Compute Profiler & `nvprof` Command Line Profiler).

Interoperabilität mit Grafik APIs

- Manchmal will man das Ergebnis einer Simulation o. ä. auf der gleichen GPU visualisieren, auf der auch gerechnet wurde.
- Wenn Daten ohnehin in GPU Speicher liegen, sollte nicht extra auf CPU kopiert werden, um mit OpenGL oder D3D anzuzeigen.
- CUDA und Grafik APIs können sich Buffer Objekte (Vertex Buffer, Pixel Buffer..) teilen.
- Dafür muss jeweilige Ressource explizit mit dem einen API gemapped und dem anderen unmapped werden.

Interoperabilität mit Grafik APIs

Beispiel: geteilter Zugriff auf PBO

```
// Init
cudaGraphicsGLRegisterImage(...);

// Use with CUDA
cudaGraphicsMapResources(...);
cudaGraphicsResourceGetMappedPointer(&pixel_pointer, ...);
// Here we can fill the PBO with CUDA
cudaGraphicsUnmapResources(...);

glBindBuffer(GL_PIXEL_UNPACK_BUFFER, ...);
// Use PBO from OpenGL
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
```

Thrust Template Library

CUDA Installation bringt Template Library für Speicherallokation und Speichertransfer, sowie GPU Datenstrukturen und Algorithmen mit.

```
void host_function()
{
    // Host vector is basically a std::vector<>
    thrust::host_vector<int> h_numbers = generateNumbers();

    // This wraps cudaMalloc() and cudaMemcpy()
    thrust::device_vector<int> d_numbers(h_numbers);

    // Thrust has, amongst other algorithms,
    // a number of fast sorting primitives
    thrust::sort(d_numbers.begin(), d_numbers.end());

    // Copy back to CPU
    thrust::copy(d_numbers.begin(), d_numbers.end(),
                 h_numbers.begin());
}
```

Thrust Template Library

Saxpy mit Thrust (Variante 1)

vgl.: https://github.com/thrust/thrust/wiki/Quick-Start-Guide

```
void saxpy1(thrust::device_vector<float>& y,
            thrust::device_vector<float>& a,
            thrust::device_vector<float>& x)
{
    // a *= x
    thrust::transform(x.begin(),x.end(),a.begin(),a.end(),
                      thrust::multiplies<float>());

    // y += a
    thrust::transform(a.begin(),a.end(),y.begin(),y.end(),
                       thrust::plus<float>());
}
```

Thrust Template Library

Saxpy mit Thrust (Variante 2)

```
struct saxpy_functor {
    template <typename Tuple>
    __host__ __device__
    void operator()(Tuple t) {
        using thrust::get;
        // y = a * x + y
        get<0>(t) = get<1>(t) * get<2>(t) + get<0>(t);
    }
};
void saxpy2(thrust::device_vector<float>& y,
            thrust::device_vector<float>& a,
            thrust::device_vector<float>& x) {
    using namespace thrust;
    for_each(make_zip_iterator(make_tuple(
                 y.begin(), a.begin(), x.begin())),
             make_zip_iterator(make_tuple(
                 y.end(), a.end(), x.end())),
             saxpy_functor());
}
```

Thrust Template Library

- Thrust nützlich als generischer Wrapper für CUDA Runtime API Funktionen.
- Thrust Datenstrukturen können nicht direkt in Kernels verwendet werden, dazu muss man (raw) Device Pointer aus den Datenstrukturen extrahieren:

```
thrust::device_vector<int> d_numbers(...);
kernel<<<..,..>>>(
    thrust::raw_pointer_cast(d_numbers.data()));
```

Tools

- Nvidia `nvcc` Compiler.
- Nvidia Occupancy Calculator.
- CUDA Debugger.
  - Debuggen mit GPUs umständlich, man braucht zwei GPUs.
  - Seit Compute Capability 2.0 zumindest `printf()` in Kernels.
- Nvidia Visual Compute Profiler.
  - Profiling auf Runtime API Ebene, keine Profiling Informationen für Instruktionen in Kernels.
  - Kennziffern aus Cubin (Registerzahl, Speicherbedarf etc. geben Aufschluss über Kernel Performance).
- Nvidia Nsight: Integration mit Entwicklungsumgebungen (Eclipse, Visual Studio).

CUDA Multi-GPU Programmierung

Multi-GPU Programmierung

- Mit CUDA kann man aus einem Programm Code auf mehrere GPUs verteilen ("Single Host, Multiple GPUs").
- Alle CUDA Aufrufe / Kernel Launches etc. beziehen sich auf die *gerade aktive* GPU.
- Mit `cudaSetDevice()` wird aktive GPU gesetzt.
- *cudaSetDevice()* kann auch asynchron zu Kernel Launches etc. aufgerufen werden.
- Nicht ungebräuchlich: Multi-Threading Applikation $\Rightarrow$ pro GPU ein Thread.

Multi-GPU Programmierung

Folgender Code ist valide, das Device darf gewechselt werden, während der erste Kernel noch rechnet $\Rightarrow$ Asynchronizität.

```
cudaSetDevice(0);
kernel<<<..,..>>>();
cudaSetDevice(1);
kernel<<<..,..>>>();
```

Multi-GPU Programmierung

Nicht-triviale Multi-GPU Programme werden Daten zwischen Devices austauschen:

```
cudaMemcpyPeerAsync(dst_on_GPU1, 1 /*dst ID*/,
                    src_on_GPU0, 0 /*src ID*/,
                    size, streams[0]);
```

Manche GPUs unterstützen Peer-Access $\Rightarrow$ dann wird direkt per DMA über PCIe kopiert, andernfalls Umweg über CPU Speicher.

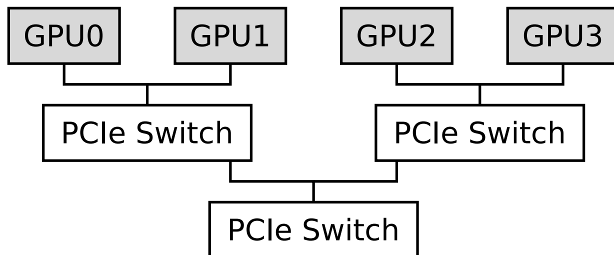
Multi-GPU Programmierung

cudaMemcpyPeerAsync

- Duplex Modus: bis 12 GB/s
- GPU $\Leftrightarrow$ GPU Transfer asynchron, blockiert die Host Ausführung nicht.
- Selbst ohne DMA: einfach zu benutzendes API, es müssen nicht explizit Buffer auf Host verwaltet werden.

Multi-GPU Programmierung

cudaMemcpyPeerAsync



PCIe *kein* Bus System ⇒ Latenz unterschiedlich, je nachdem, welche PCIe Lanes miteinander kommunizieren.

Andererseits Contention, wenn GPUs sich Kommunikationspfad teilen.

⇒ Datentransfer von Multi-GPU Programmen muss für PCIe optimiert werden.

Dynamischer Parallelismus

Dynamic Parallelism

Seit Compute Capability 3.5 können Compute Kernels *ohne Zutun der CPU* (ggf. rekursiv) Compute Kernels starten.

Vermeide GPU $\Rightarrow$ CPU $\Rightarrow$ GPU round-trip.

Ähnliche Funktionalität gibt es durch Extensions auch in den Grafik APIs Vulkan und OpenGL ("Indirect Rendering", `glDrawArraysIndirect()`).

Praktisch für Algorithmen wie *Adaptive Grid Refinement* (teile Grid dynamisch in feinere Zellen auf, falls z. B. lokal hohe Frequenzen in Daten).

Dynamic Parallelism Beispiel

Kernel ruft sich rekursiv selbst auf:

```
__global__ void kernel()
{
    if (threadIdx.x == 0 && threadIdx.y == 0)
    {
        if (condition)
        {
            dim3 blocks  = ...;
            dim3 threads = ...;
            kernel<<<blocks, threads>>>();
        }
    }
}

void func()
{
    kernel<<<..,..>>>();
}
```

Dynamic Parallelism

- Dynamic Parallelism nicht beschränkt auf Rekursion (kann auch andere Kernel aufrufen).
- Kann dann Performance Vorteile mit sich bringen, wenn Subdivision Kommunikation mit Host nicht rechtfertigt.

Recap

- CUDA herstellerspezifisches API für GPGPU. Viel weniger komplexes API als Vulkan.
- CUDA exponiert *Speichermodell* moderner GPUs. Programmierparadigma basiert darauf, Berechnungen durch massiven Parallelismus hinter Speicherzugriffslatenzen zu verstecken.
- CUDA Device Code integriert sich in Host Programm und kann auch CPU Routinen verwenden (diese müssen jedoch auch für das Device kompiliert werden).
- CUDA hat nicht unwesentlich zu Nvidias Popularität im HPC Umfeld beigetragen.

Literaturempfehlungen

- Jason Sanders und Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming (2010).
- Nvidia: CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2018).