1 Einführung und Allgemeines

In diesem Praktikum sollten verschiedene parallele Zugriffsmuster auf Grafikarten gemessen und verglichen werden. Dabei verwendet wurde CUDA [1], eine von Nvidia entwickelte C++-Erweiterung, die Grafikkarten dieser Firma für parallelisierbare Berechnungen abseits von klassischem Rendering nutzbar macht. In diesem Fall verwendet wurde eine GTX 1070. Einige relevanten Spezifikationen dieser GPU sind:

- 1920 Cores, wovon ein Warp jeweils 32 Threads umfasst [2]
- Maximale Taktung laut „nvidia-smi“: 1708 MHz [2]
- Speicherkonfiguration: 8 GB GDDR5 RAM [2]
- Herstellerangabe für den Durchsatz: 256 GB/s [2]
- PCIe 3.0 fähig, in der Messung wurde allerdings nur PCIe 2.0x16 verwendet. Die maximale Kopierrate von PCIe 2.0x16 entspricht 8.0GB/s [3]

1.1 Grundsätzliche Beobachtungen zur Kopierrate

Eine Grundlage der Programmierung mit der CUDA ist das Management der Daten, welche auf der CPU (genannt „Host“) oder der GPU (genannt „Client“) liegen, da CPU-Kerne nur bedingt auf den GPU-RAM zugreifen können und umgekehrt. Mithin ist die mit Standard-Kopieroperatoren erzielbare Kopierrate interessant, die gleichzeitigen einen guten Vergleichsmaßstab für die Effizienz der Verschiedenen Kernel darstellt. Abbildung 1 zeigt die für `cudaMemcpy()` gemessenen Kopierraten und die vom Hersteller angegebenen Maximalwerte sowie die Spezifikation für den verwendeten PCIe.

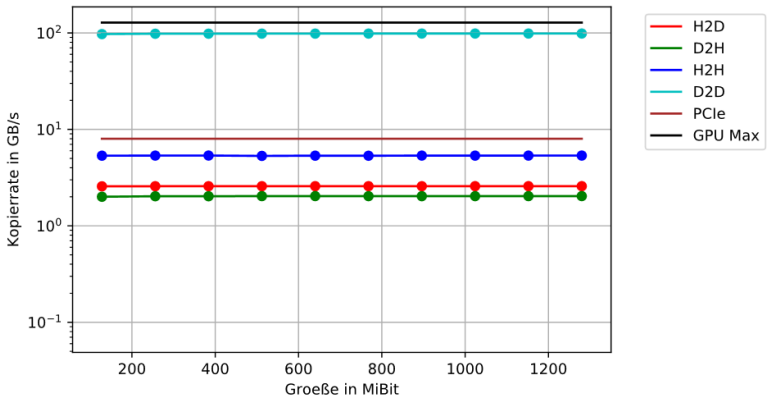

Abbildung 1: Kopierrate von `cudaMemcpy()`

Bei der tatsächlichen Durchführungen der Messungen ist es außerdem wichtig, zu beachten, dass jeweils der erste Kernel Aufruf eines CUDA Programms eine wesentlich längere Zeit benötigt als darauf folgende Aufrufe des selben Kernels. Abbildung 2 zeigt

eine Beispielmessung für einen Empty Kernel, dh. einen Kernel, der zwar aufgerufen wird, dann auf der GPU allerdings keine Arbeit verrichtet. Um dieser Verzerrung entgegenzuwirken, muss vor jeder Laufzeitmessung für einen Kernel dieser einmal als „Warmup“ aufgerufen, aber dann aus der eigentlichen Messung herausgerechnet werden.

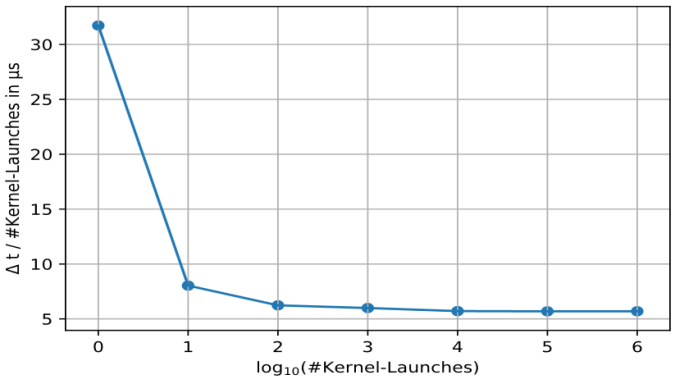

Abbildung 2: Startup Cost eines Empty Kerneles

2 Copy Kernel

Der „Copy Kernel“, der jeweils ein Element eines Buffers in die entsprechende Position eines anderen Buffers kopiert, lässt sich durch folgenden CUDA-Quellcode beschreiben:

```
template<typename T>
__global__
void copyKernel(T* out, T* in) {
        unsigned id = threadIdx.x + blockIdx.x * blockDim.x;

        out[id] = in[id];
}
```

Obwohl die Funktion als Input nur Zeiger auf ein Quell-und ein Ziel-Buffer nimmt, gibt es doch drei implizite Parameter. Diese treten auch bei allen folgenden Messungen auf:

1. Anzahl der Blöcke, in die die einzelnen Threads von der GPU gruppiert werden
2. Die Größe dieser Blöcke
3. Zugriffstyp `T` (zB. `char`, `int`)

Abbildung 3 und Abbildung 4 zeigen die Auswirkungen von Variationen über die drei Parameter auf die durch den Copy Kernel erzielte Kopierrate. Dabei zeigt sich, dass

sowohl für die Anzahl der Blöcke, für die Blockgröße als auch für `sizeof(T)` ein jeweils höherer Parameter zu einer besseren Kopierrate führt. Für hinreichend hohe Werte wird eine Sättigung nahe des Ergebnisses aus Abbildung 1 erzielt. Interessant ist außerdem, dass die Abbildungen sich mit steigendem `sizeof(T)` je linearer verhalten, je geringer Blockgröße bzw. Blockanzahl sind.

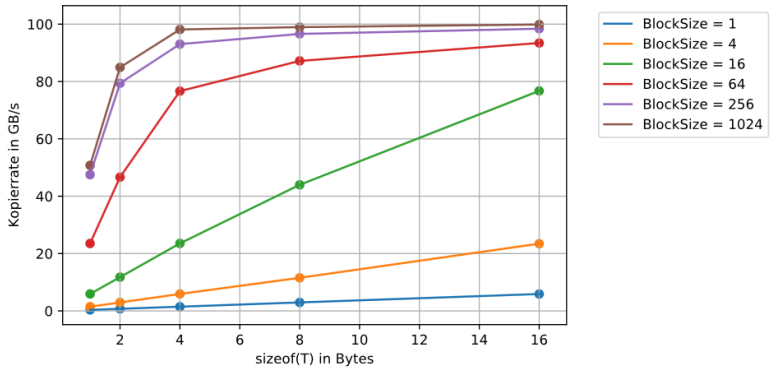

Abbildung 3: Variation über `BlockSize` und `sizeof(T)`, `numBlocks` = 16384

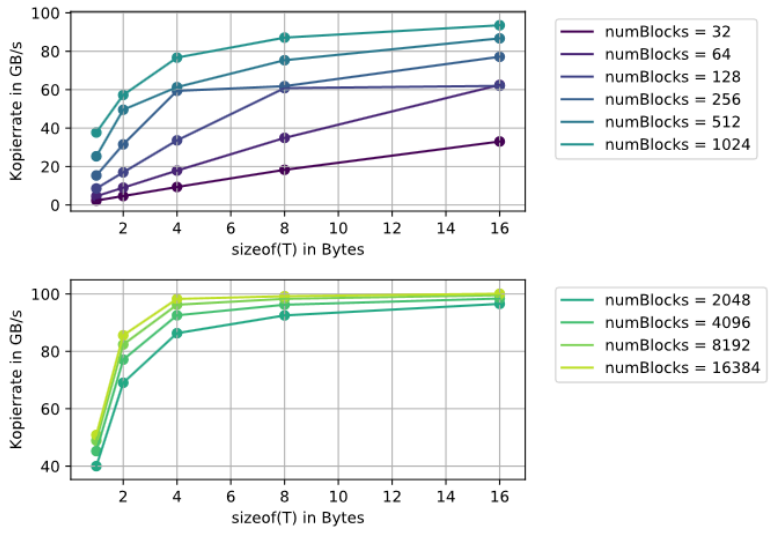

Abbildung 4: Variation über `numBlocks` und `sizeof(T)`, `BlockSize` = 1024

3 Strided Access

Der Strided Access ist wie folgt definiert:

template<**typename** T>

```
__global__
void copyKernel(T* out, T* in, int stride) {
        unsigned id = threadIdx.x + blockIdx.x * blockDim.x;

        out[id*stride] = in[id*stride];
}
```

Das heißt, der Zugriff erfolgt nicht mehr auf jedes Element konsekutiv hintereinander, sondern nur auf jedes N-te Element. Zusätzlicher Parameter ist dann der `stride`, der festlegt, wie viele Buffer-Einträge übersprungen werden. Bei elementarem Zugriff, wie er zum Beispiel in einer Turingmaschine definiert ist, sollte sich keine Änderung der Kopierrate ergeben, sofern die nicht bearbeiteten Elemente nicht mitgerechnet werden. Beim der Messung zeigt sich aber, dass deutliche Cashe-Effekte auftreten. Bei Messungen zeigt sich ein nahezu exponentieller Abfall der Kopierrate mit steigendem `stride`, für den die beiden Messungen Abbildung 5 und Abbildung 6 beispielhaft stehen. Bei `sizeof(T)` $\cdot$ `stride` $= 32$ und `sizeof(T)` $\cdot$ `stride` $= 64$ ergeben sich jeweils zwei fast gleiche Werte, und ab `sizeof(T)` $\cdot$ `stride` $= 2048$ fällt die Kopierrate wenn überhaupt nur noch vernachlässigbar ab (bei diesen beiden Messungen wie auch bei allen anderen).

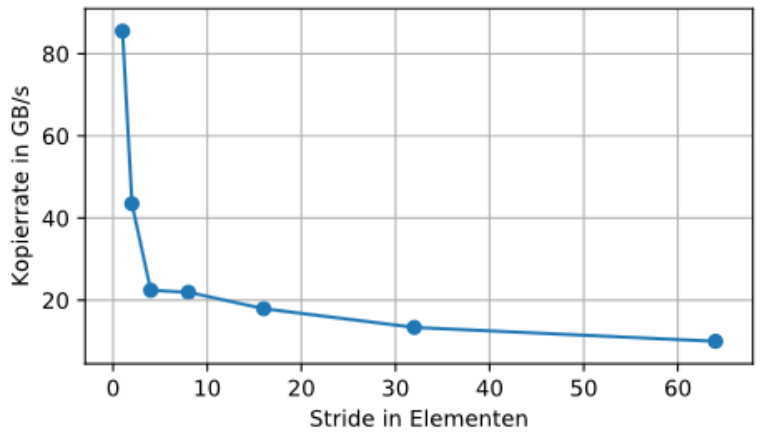

Abbildung 5: Strided Access: Variation über `stride`, `BlockSize` $= 256$, `numBlocks` $=$ 4096, `T`: `int2` (`sizeof(int2)` $= 8$)

4 Offset Access

Der Offset Access ist ein etwas komplexerer Kernel. Sein Quellcode lautet wie folgt:

```
template<typename T>
__global__
void oftKernel (T* out,
                T* in,
```

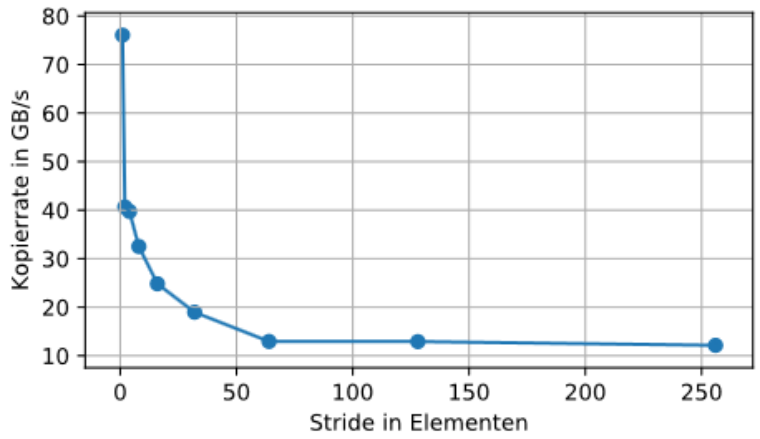

Abbildung 6: Strided Access: Variation über `stride`, `BlockSize` = 128, `numBlocks` = 2048, `T`: `int4` (`sizeof(int4)` = 16)

```
                const unsigned int sd_size,
                const unsigned int block_size,
                const unsigned int I,
                const unsigned int L)
{
        const unsigned int sd_id = static_cast<int>(threadIdx.x / L);
        const unsigned int id = threadIdx.x - sd_id * L;
        const unsigned int sd_start = blockIdx.x *
                        blockDim.x * I + sd_id * L * I;

        for (unsigned int i = 0; i < I; i++)
        {
                const unsigned el_id = sd_start + i * L + id;
                ((T*) out)[el_id] = ((T*) in)[el_id];
        }
}
```

Abbildung 7 zeigt das Zugriffsmuster, das aus diesem Kernel resultiert. Ein „oftKernel“ greift nicht konsekutiv mit jedem Thread auf ein Element der Buffer zu. Er teilt die Buffer zwar in gleich große Abschnitte auf, die jeweils einem Block zugeordnet werden, unterteilt die Blöcke allerdings erneut in Subdomains. Diese Subdomains bestehen dann aus einem gewissen Anteil Threads, die jeweils eine gewisse Anzahl an Iterationen durchführen müssen, um alle Kopiereinträge abzuarbeiten. Neben den bekannten Parametern `T` und `BlockSize` ergeben sich dann noch zusätzlich die Anzahl der Iterationen `I` und die Anzahl der Threads pro Subdomain `L`. Die Anzahl der Blöcke ergibt sich dann aus diesen Parametern und der Größe der Subdomains auf den Buffern `sd_size`.

Die Abbildungen 8, 9 und 10 zeigen verschiedene Variationen über die Parameter

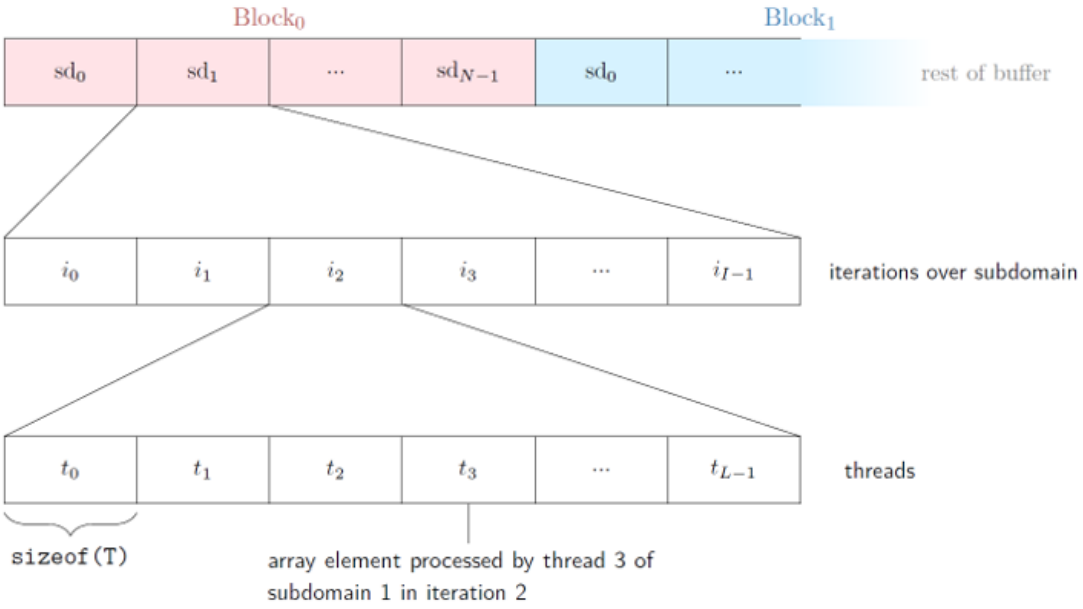

Abbildung 7: Zugriffsmuster des Offset Access

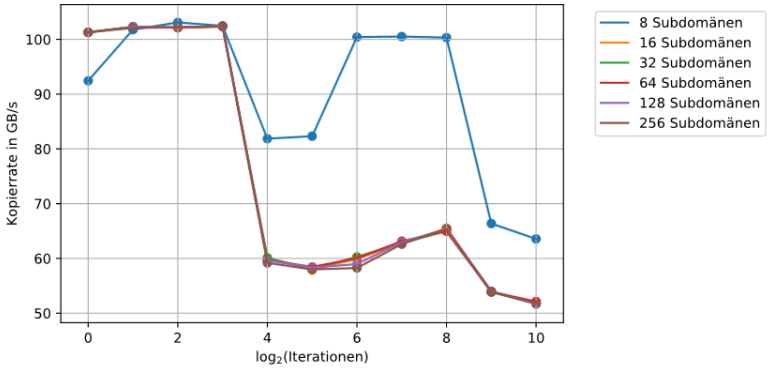

Abbildung 8:

des Kernels. Es zeigt sich, dass die Zusammenhänge im Detail komplex sind, und ohne eine tiefere Analyse der verschiedenen Caches der verwendeten GPU (isb. Gesamt-und Zugriffsgröße) wohl nicht vollständig zu verstehen sind. Dennoch lässt sich festhalten, dass eine Sättigung vergleichbar mit vorherigen Kopieroperationen erreicht werden kann, wenn die Parameter richtig gewählt werden, obwohl der Warp, d.h. die jeweils 32 immer gleich ausgeführten Threads, nicht auf nacheinander liegende Elemente zugreifen muss (siehe isb. Abbildung 10). Gleichzeitig zeigt sich ein deutlicher Zusammenhang mit der `L2`-Cache Line-Size von 128 Byte - ist die Zugriffsgröße einer Iteration gleich 128 Byte oder ein ganzzahliger Teiler davon, ist die Kopierrate wesentlich höher.

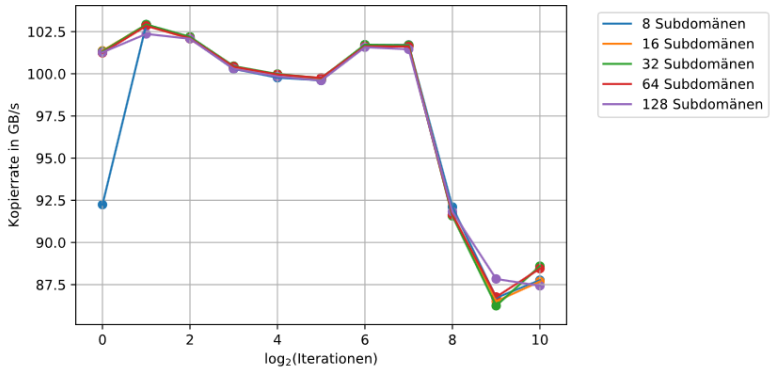

Abbildung 9:

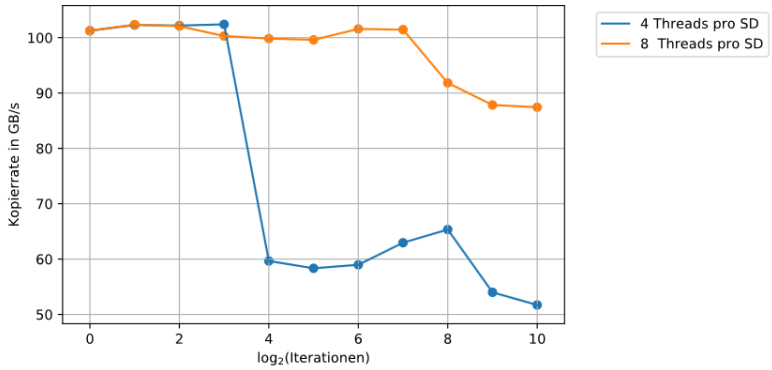

Abbildung 10:

5 Allokation: Standard und UnifiedMemory

Neben der Standard-Allokation über die C++ Funktion `malloc()` kann CUDA auch mit `cudaMallocHost()` einen sogenannten Pinned Memory Buffer auf dem Host-RAM allokieren. Dieser erlaubt schnelleres Kopieren und direkte Zugriffe der Kernel auf den

Host-RAM, ohne vorher im Code händisch eine Kopieroperation auf den GPU-Speicher vornehmen zu müssen. Mit einem Copy Kernel wie dem aus Kapitel 2 kann dann wieder eine Kopierraten-Messung vorgenommen werden (Abbildung 11). Einen Unterschied bezüglich der Anzahl Blöcke scheint es nicht zu geben, die Kurven liegen perfekt aufeinander. Die Kopierrate scheint linear zu steigen, bis eine Sättigung erreicht wird, die etwas unter dem Niveau des Geschwindigkeit des verwendeten PCIe's liegt. Diese Sättigung wird beim Kopieren vom Device zum Host bereits mit weniger Threads erreicht, und das erreichte Niveau ist ein wenig höher.

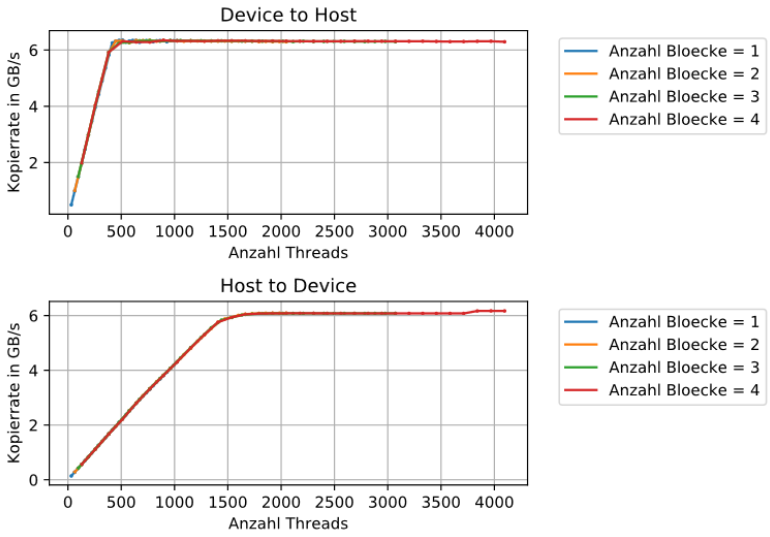

Abbildung 11: Copy Kernel Kopierrate auf Pinned Memory, `T`: `int`

Zusätzlich können Buffer in CUDA auch mithilfe von `CudaMallocManaged()` allokiert werden. Dabei wird vom Programmierer überhaupt keine Angabe getroffen, ob der Buffer auf dem CPU- oder GPU-Ram liegt, sondern CUDA wickelt eventuell benötigte Kopiervorgänge automatisch ab. Abbildung 12 zeigt eine Kopierraten-Messung des Copy-Kernels auf Unified Memory. Dabei wurde durch Cs `memset()` und CUDAs `cudaMemset()` sichergestellt, dass die Input- und Output-Buffer jeweils vollständig auf den Host bzw. dem Device RAM kopiert wurden, bevor die Messung begann. Es zeigt sich ein ähnlicher Verlauf wie in der vorherigen Messung, allerdings ist die Form der Kurven viel ungleichmäßiger (möglicherweise ist der Messfehler größer), die Anzahl der Blöcke macht einen deutlicheren Unterschied und die Sättigung wird auch später erreicht.

6 Kernel im Producer-Consumer-Verhältnis

Für diese Messungen wurden zwei identische Grafikkarten benötigt .Aufgrund der Komplexität des verwendeten Codes sei dessen Gestalt hier nur kurz umrissen: Zwei Kernel, die (in eine Richtung) miteinander kommunizieren müssen, d.h. der eine Kernel benötigt

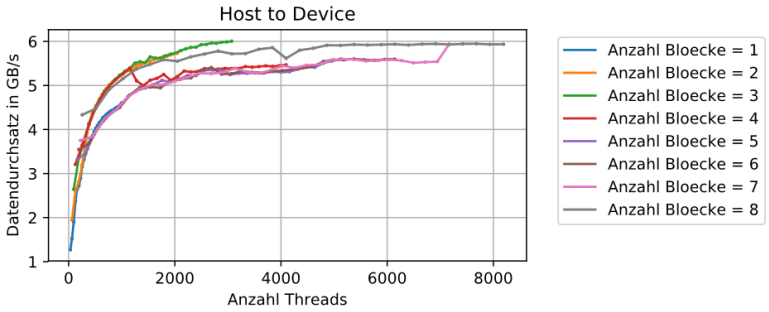

Abbildung 12: Copy Kernel Kopierrate auf Unified Memory, `T`: `int`

für die Erfüllung seiner Aufgabe die Ergebnisse des anderen aber nicht umgekehrt, stehen in einem Producer-Consumer-Verhältnis. Im vorliegenden Fall errechnet dabei der Producer den Durchschnitt eines Teils des Input-Buffers, während der Consumer diesen berechneten Durchschnitt ausliest und ihn vielfach in den entsprechenden Teil gleicher Größe des Output-Buffers dupliziert. Dabei wartet der Consumer jeweils nur auf einen Teil des Buffers, bevor er diesen dupliziert. Eine Grafikkarte führt dann jeweils einen Kernel aus. Eine Zeitmessung für das Producer-Consumer-System bei Ignorieren der Kommunikation findet sich in Abbildung 13: Die benötigte Zeit steigt also nahezu linear mit den Ergebnissen pro Block, was auch zu erwarten ist, da mit diesen auch die Menge der kopierten Daten steigt.

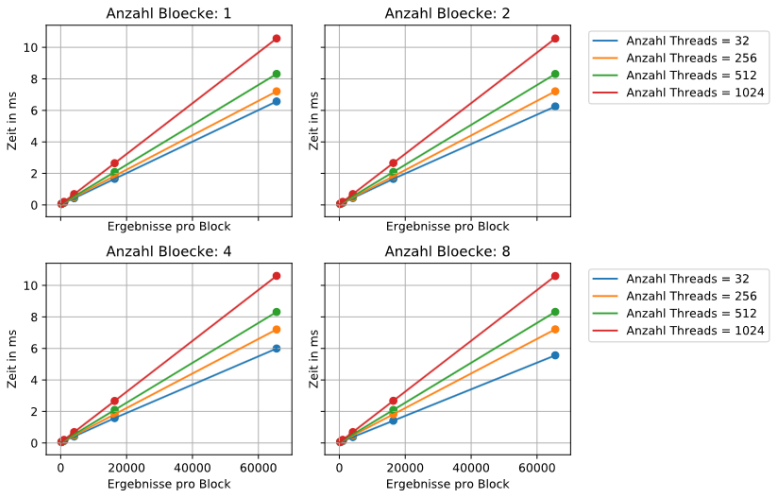

Abbildung 13: Laufzeit Producer-Consumer-Verhältnis ohne Kommunikation

Die Graphen mit aktivierter Kommunikation verlaufen ähnlich, allerdings auf einem wesentlich höheren Niveau. Auch die Berechnung des Anteils der Kommunikation am Gesamtergebnis führt demzufolge zu einem mehr oder weniger konstanten Verlauf der Kurven (Abbildung 14). Lediglich bei einer sehr geringen Anzahl von Ergebnissen pro

Block zeigt sich eine gewisse Abweichung. Mit einem Anteil über 80%, teilweise sogar über 90% liegt der Kommunikationsanteil jedenfalls sehr hoch, und er scheint mit steigender Anzahl von Blöcken auch leicht zu steigen. Umgekehrt handelt es sich bei dem Ausschlag der Kurve für 8 Blöcke und 1024 Threads beim letzten Wert vermutlich um einen Messfehler.

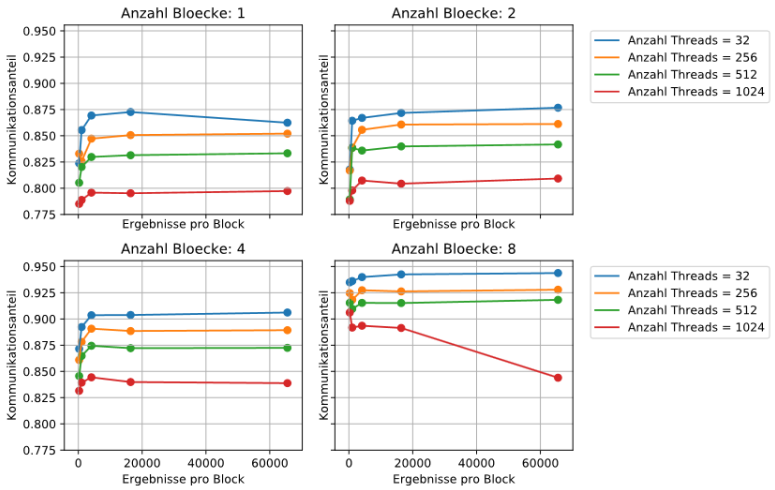

Abbildung 14: Anteil der durch Kommunikation verlorenen Zeit im Producer-Consumer-Verhältnis

Literatur

[1] Nvidia Website: “CUDA Zone”, developer.nvidia.com/cuda-zone, aufgerufen am 6.2.2019

[2] Nvidia Website: “GTX 1070”, nvidia.com/de-de/geforce/products/10series/geforce-gtx-1070, aufgerufen am 6.2.2019

[3] Wikipedia: “PCI Express”, en.wikipedia.org/wiki/PCI_Express, aufgerufen am 6.2.2019