Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Arya Mazaheri, Marcus Ritter

Systemnahe und Parallele Programmierung (WS 20/21)

Praktikum: CUDA

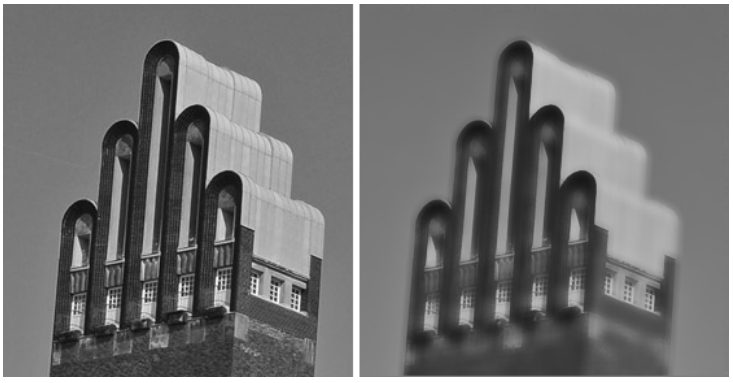
Die Lösungen müssen bis zum 9. Februar 2021, 15:00 Uhr, in Moodle submittiert werden. Anschließend müssen Sie Ihre Lösungen einem Tutor vorführen. Alle Programmieraufgaben müssen mit CUDA gelöst werden, und dann auf dem Lichtenberg Cluster kompilier und ausgeführ sein. Alle Lösungen müssen zusammen in einer Archivdatei eingereicht werden.

In diesem Praktikum geht es um die CUDA Programmierung. Im Folgenden sollen Sie einen Algorithmus implementieren, der ein Bild unter Beibehaltung der Kanten weichzeichnet. Das Problem findet breite Anwendung im Gebiet der Computer Grafik. Die Eingabe für Ihren Algorithmus ist ein Farbbild im RGB Format. Das heißt, die Farbe von jedem Pixel wird durch sein Rot-, Grün- und Blauwert bestimmt. Wir beschreiben nun die einzelnen Schritte Ihres Algorithmus:

1) Farben in Graustufen Konvertieren. Es gibt verschiedene Methoden, die drei RGB Farbintensitäten auf einen einzigen Grauwert abzubilden. Beispiele sind der Durchschnitt der drei Farbwerte, Entsättigung, und Dekomposition. In diesem Praktikum verwenden wir eine Technik basierend auf der Farbmetrik um die ursprünglich wahrgenommene Helligkeit des Farbbildes im Graustufenbild zu erhalten. Der Grauwert eines Pixels wird demnach wie folgt aus den RGB Farbwerten berechnet:

$$Grau = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B \tag{1}$$

2) Weichzeichnen. In diesem Schritt zeichnen wir das Graustufenbild unter Beibehaltung der Kanten weich. Dazu verwenden wir einen bilateralen Filter. Die neue Intensität von jedem Pixel ergibt sich aus dem gewichteten Durchschnitt der Intensitäten benachbarter Pixel. Die Gewichte ergeben sich aus einer Gauß-Verteilung und weiteren Eigenschaften, wie zum Beispiel dem Farbabstand zwischen Nachbarpixel und betrachtetem Pixel. Die folgende Abbildung zeigt auf der rechten Seite das Ergebnis der Anwendung des Filters.



Von den oben beschriebenen Schritten befindet sich eine sequenzielle Implementierung in der Datei `serial.cpp`. Das Hauptprogramm ist in der Datei `main.cpp`. In der Datei `kernel.cu` befinden sich Fragmente einer Parallelisierung mit CUDA. Vervollständigen Sie bitte die Implementierung mit CUDA in der Datei `kernel.cu`. In Ihrer CUDA Implementierung soll in jedem Kernel jeder Thread einen Pixel verarbeiten. Im Folgenden geben wir noch ein paar Hinweise und beschreiben die einzelnen Aufgaben für die Implementierung.

Allgemeine Hinweise: Zum Kompilieren oder Ausführen eines CUDA Programms auf dem Lichtenberg Cluster müssen sie das CUDA Modul laden: `module add cuda`. Kompilieren können sie bereits

auf den Login Knoten. Zum Ausführen des CUDA Progamms muss jedoch ein Batch Job submittiert werden. Nutzen sie dafür die Datei `batch_job.sh` als Beispiel.

Hinweis zu Occupancy in CUDA: Bevor ein CUDA Kernel gestartet wird, muss seine Blockgröße festgelegt werden. Zum Berechnen einer geeigneten Blockgröße, die die Ausführungszeit des Kernels minimiert, eignet sich das Konzept der *Occupancy*. Die Occupancy ist das Verhältnis von der Anzahl aktiver *Warps* auf einem Multiprozessor der GPU, zu der maximal möglichen Anzahl von Warps, die auf einem Multiprozessor aktiv sein können. Beachten Sie, dass die höchste Occupancy nicht notwendigerweise die beste Performance liefert. Nichtsdestotrotz bietet sie eine gute Heuristik für die geeignete Wahl der Ausführungskonfiguration eines Kernels.

Aufgabe 1

(5 Punkte) Berechnen Sie die Dimensionen der Blöcke und des Grids für den Kernel `cuda_grayscale`. Nutzen Sie die vorgegebene Funktion `cudaOccupancyMaxPotentialBlockSize`, um eine möglichst hohe Occupancy zu erreichen.

Aufgabe 2

(5 Punkte) Allokieren Sie zwei Puffer im Speicher auf dem *Device* um Bilder während der Verarbeitung zu speichern. Initialisieren Sie jedes Byte der Puffer mit dem Wert `0` unter Verwendung der Funktion `cudaMemset`. Mehr Informationen über `cudaMemset` finden Sie mit der Suchfunktion auf `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`.

Aufgabe 3

(5 Punkte) Allokieren Sie einen Puffer für das Eingabebild auf dem Device und kopieren Sie das Eingabebild in diesen Puffer.

Aufgabe 4

(15 Punkte) Implementieren Sie den Kernel `cuda_grayscale`, der das farbige Eingabebild in ein Graustufenbild konvertiert. Rufen Sie den Kernel entsprechend im Programm auf.

Aufgabe 5

(3 Punkte) Kopieren Sie das erstelle Graustufenbild zurück in den Hauptspeicher und speichern Sie es in einer Datei, damit Sie die Korrektheit des Kernels `cuda_grayscale` überprüfen können.

Aufgabe 6

(6 Punkte) Berechnen Sie die Dimensionen der Blöcke und des Grids für den `cuda_bilateral_filter` Kernel. Nutzen Sie die vorgegebene Funktion `cudaOccupancyMaxPotentialBlockSize`, um eine möglichst hohe Occupancy zu erreichen.

Aufgabe 7

(6 Punkte) In der Funktion `cuda_updateGaussian` muss am Ende das Array `fGaussian` in ein Array `cGaussian` im Constant Memory der GPU kopiert werden. Allokieren Sie dafür zunächst das Array `cGaussian` im Constant Memory. Danach kopieren Sie das Array `fGaussian` vom Host Memory in das Array `cGaussian` im Constant Memory mit der Funktion `cudaMemcpyToSymbol`. Mit dieser Funktion können wir einfach den Namen des Arrays `cGaussian` als Ziel der Kopieroperation angeben. Alternativ müssten wir für das Kopieren mit `cudaMemcpy` zuvor die Addresse von `cGaussian` im Constant Memory ermitteln. Mehr Informationen über `cudaMemcpyToSymbol` finden Sie mit der Suchfunktion auf `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`.

Aufgabe 8

(4 Punkte) Implementieren Sie die Funktion `cuda_gaussian`, die im Kernel `cuda_bilateral_filter`

gerufen wird.

Aufgabe 9

(15 Punkte) Implementieren Sie den Kernel `cuda_bilateral_filter` und wenden Sie ihn auf das Graustufenbild an.

Aufgabe 10

(3 Punkte) Kopieren Sie das weichgezeichnete Bild vom Device Memory in den Host Memory, sodass es in eine Datei zum späteren Betrachten geschrieben werden kann.

Aufgabe 11

(4 Punkte) Geben Sie alle dynamisch allokierten Puffer im Device Memory wieder frei.

Aufgabe 12

(4 Punkte) Einer der größten Vorteile von CUDAs Speicherverwaltung ist, dass sie zu großen Teilen der von C ähnelt, mit der Programmierer bereits vertraut sind. Der folgende Quellcode zeigt ein Beispiel das ein 1D Array `x` zuweist und dieses als Eingabe für den `sort` Kernel nutzt. Was stimmt nicht mit dem Quellcode? Welche Fehler können sie finden?

```
1 float *x = (float *)malloc(sizeof(float) * ARRAY_SIZE);
2 initializeArray(x, ARRAY_SIZE, ...);
3
4 // Allocate the copy of the array on the GPU
5 cudaMalloc(&x, ARRAY_SIZE * sizeof(float));
6
7 sort<<<grid, block>>>(x, ARRAY_SIZE, ...);
```