Versuch S1

Programmierung mit dem Befehlssatz Streaming SIMD Extensions (SSE)

Dipl.-Ing.(FH) Daniel Walther

Friedrich-Schiller-Universität Jena
Institut für Informatik
Lehrstuhl für Advanced Computing

20. März 2018

Inhaltsverzeichnis

Computer arbeiten klassischerweise seriell, stets wird nur eine einzelne Anweisung ausgeführt. Möchte man die Berechnungen beschleunigen, hat sich seit Jahren das Ausnutzen von Parallelität als Lösung etabliert. Multicore-Prozessoren gehören inzwischen zum Alltag, und die Anzahl der Kerne auf einem Chip wird zukünftig weiter steigen. Diese Kerne arbeiten weitestgehend autark. D.h., sie führen alle voneinander unabhängige Befehlsströme aus, die auf unterschiedlichen Daten operieren. Dieses Prinzip wird als *Multiple Instruction, Multiple Data* (MIMD) bezeichnet. Für Multimedia-Anwendungen und andere Probleme, in denen Daten überwiegend in einer vektoriellen Arbeitsweise verwendet werden, hat sich eine andere Form der Parallelverarbeitung bewährt. Hier wird nun ein Befehl gleichzeitig auf mehreren Daten ausgeführt, so dass statt eines Einzelergebnisses ein ganzer Ergebnisvektor entsteht. Dieses Prinzip ist unter dem Begriff *Single Instruction, Multiple Data* (SIMD) bekannt und ist schematisch in Abb. 1 illustriert. Zur SIMD-Berechnung sind in modernen Prozessoren sogenannte Vektor-Einheiten vorhanden, die oftmals bis zu 32 Werte gleichzeitig berechnen können.

Gegenstand dieses Versuchs ist es, eine kleine Videoschnitt-Software in der Programmiersprache C zu implementieren, die mit Hilfe von SIMD-Befehlen parallel folgende Aufgabe bewältigt: Beginnend mit einem Schwarzbild wird ein Video A eingeblendet, dieses zwei Sekunden lang gezeigt, danach in ein Video B übergeblendet, um nach weiteren zwei Sekunden nach schwarz auszublenden.

Die Eingangsvideos A und B werden im Versuch bereitgestellt. Außerdem steht eine Vielzahl an Hilfsfunktionen (siehe Abschnitt 2.3) zur Verfügung, die die Programmierung erleichtern. Ihre Aufgabe wird es sein, die Ein-, Aus- und Überblendungen unter Verwendung von Instruktionen aus dem Befehlssatz *Streaming SIMD Extensions* (SSE) zu implementieren.

1 Vorbereitung und Voraussetzungen

Hinweis: Dieser Abschnitt soll nicht protokolliert werden, er dient lediglich der Vorbereitung und dem Kolloquium.

Informieren Sie sich zu folgenden Fragen, z.B. durch Lesen von http://arstechnica.com/old/content/2000/03/simd.ars

- Was beschreibt die Klassifikation von Flynn?
- Was ist Altivec?
- Was ist SSE, was hingegen SSE2?
- Welche Erweiterunge brachte SSE4 und AVX?
- Was versteht man unter Intrinsics?
- Was ist SIMD und wo findet es Verwendung?
- Was versteht man in der Bildverarbeitung unter RGBA?
- Wieviele Farben hat der sRGB-Farbraum?

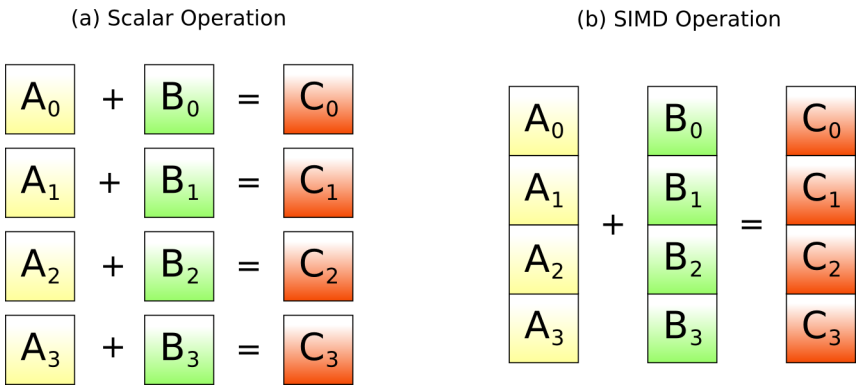

Abbildung 1: a) Skalare Operationen verknüpfen nur jeweils ein skalares Datum mit einem anderen. Eine Folge von skalaren Operationen wird daher durch Hintereinanderausführung solcher skalaren Operationen ausgewertet. b) Dagegen verarbeiten SIMD-Operationen mehrere skalare Daten mit einem einzigen (Vektor-)Befehl und sind somit für die Berechnung der gleichen Datenmenge schneller. Quelle: www.kernel.org

1.1 Grundlagen der Bildverarbeitung

Bilder werden im Computer durch eine Vielzahl einzelner Bildpunkte dargestellt, den sogenannten Pixeln. Je nach Einsatzzweck wird ein einzelner Punkt durch einen unterschiedlich großen Farbraum repräsentiert. So könnte ein Pixel im einfachsten Fall beispielsweise durch ein einzelnes Bit beschrieben werden, das die Farben Schwarz (0) und Weiß (1) kodiert. Dabei ist die Interpretation, ob der Wert 0 die Farbe Schwarz oder die Farbe Weiß repräsentiert, frei wählbar. Möchte man feinere Abstufungen zulassen, können auch 8 Bit pro Bildpunkt verwendet werden und liefern somit einen „Farbraum“ von $2^8 = 256$ Graustufen. Es ist natürlich möglich, diese 256 diskreten Werte als feststehende Farbpalette zu interpretieren, wie es beispielsweise bei GIF der Fall ist, so dass mit 8 Bit Farb- statt Grautöne dargestellt werden können. Für echte Farbbilder verwendet man hingegen oftmals den sogenannten sRGB-Farbraum. Dabei werden pro Pixel drei Farbkanäle gespeichert: 8 Bit für Rot, 8 Bit für Grün und 8 Bit für Blau. Pro Farbkanal sind wieder je 256 Abstufungen möglich, durch Mischen dieser Werte mit den anderen Kanälen entstehen somit ca. 16 Millionen Farben.

Bei drei Farbkanälen mit je 8 Bit ergeben sich pro Bildpunkt 24 Bit, die es zu speichern gilt. Da der Zugriff auf 24 Bit auf den meisten Maschinen deutlich langsamer als der zusammenhängende Zugriff auf 32 Bit ist[1], repräsentiert oftmals ein vierter Farbkanal, der sogenannte Alpha-Kanal, die fehlenden 8 Bit. Der Alpha-Kanal wird dabei entweder ignoriert oder entspricht dem Grad der Transparenz, gibt also an, wie stark die RGB-Farbwerte darunterliegende Bilder überlagern (Deckkraft). Der Alpha-Kanal in diesem RGBA-Farbraum wird im Rahmen dieses Versuchs ignoriert und daher nicht näher betrachtet, liegt aber mit im Speicher und muss dadurch zumindest beachtet werden.

Die Farbwerte eines einzelnen Bildpunkts liegen üblicherweise direkt hintereinander im

[1] Die Stichworte *Word Size* und *Alignment* seien hier genannt.

Speicher, gefolgt von den Farbwerten des nächsten Bildpunkts. Beginnend mit dem ersten Pixel entspricht ein Bild somit einem großen Array von einzelnen Farbwerten. Dieses Array ist meist eindimensional, in dem alle Zeilen gedanklich zu einer einzigen langen Zeile verbunden werden. Diese unterschiedlichen Repräsentationen sind in Abb. 2 dargestellt.

Speichert man eine Folge von Bildern und spielt sie dann schnell hintereinander ab, entsteht durch die Trägheit des Auges der Eindruck eines bewegten Films, man erhält ein Video. Jedes Video besteht aus einer Vielzahl von Einzelbildern (Frames), die mit einer fest vorgegebenen Geschwindigkeit wiedergegeben werden. Im europäischen Raum haben sich dabei 25 Bilder pro Sekunde etabliert[2], da die europäischen Stromnetze mit 50Hz Wechselspannung arbeiten und durch Halbierung dieser Frequenz ein stabiler Bildwiederholtakt abgeleitet werden kann.

1.2 Speicherlayout im Versuch

Im Abschnitt 1.1 wurde gezeigt, dass ein Bildpunkt im sRGB-Farbraum durch 24 Bit bzw. im RGBA-Farbraum durch 32 Bit repräsentiert werden kann. In diesem Versuch soll aber ein-, aus- und überblendet werden. Multipliziert man alle Farbkanäle mit 0, so erhält man konventionsgemäß ein schwarzes Bild, multipliziert man sie hingegen mit 1, erhält man das Originalbild. Wir betrachten hier exemplarisch die Änderung des Farbwertes für den roten Kanal. Es ist offensichtlich, dass eine Einblendung für diesen Farbkanal mathematisch wie folgt beschrieben werden kann:

$$r_{\text{neu}} = \beta \cdot r_{\text{alt}}, \qquad \beta \in [0, 1].$$

Dabei beginnt man mit einem Skalierungswert $\beta = 0$ und steigert diesen Wert über die Dauer der Einblendung schrittweise bis auf $\beta = 1$. Damit ist der Skalierungswert β keine Ganzzahl, so dass praktikablerweise Fließkomma-Arithmetik Verwendung findet. Der kleinste IEEE-Fließkommatyp in der Sprache C ist `float`. Darunter versteht man eine Zahl vom Typ *Single-Precision Floating-Point* (SPFP), deren Länge 32 Bit beträgt.

Da das Ergebnis einer Multiplikation mit einer Fließkommazahl ohnehin wieder eine Fließkommazahl ist, sind in diesem Versuch sowohl Ein- als auch Ausgabebilder ebenfalls schon im SPFP-Format. Ein einzelner Pixel wird somit durch vier 32bit-Fließkommazahlen repräsentiert, je eine für den roten, grünen und blauen Kanal sowie eine Fließkommazahl für den Alpha-Kanal, die allerdings ignoriert wird.

Die Befehle aus dem Befehlssatz *Streaming SIMD Extensions* (SSE) arbeiten auf Vektoren der Länge 128 Bit, die beispielsweise vier SPFPs aufnehmen und gemeinsam verarbeiten können. Seien etwa vier unterschiedliche Skalierungswerte β_r, β_g, β_b und β_a für die vier Farbkanäle gegeben. Statt für jeden Farbkanal die Überblendungen nacheinander einzeln zu berechnen, was durch die Gleichungen

$$\begin{aligned} r_{\text{neu}} &= \beta_r \cdot r_{\text{alt}}, \\ g_{\text{neu}} &= \beta_g \cdot g_{\text{alt}}, \\ b_{\text{neu}} &= \beta_b \cdot b_{\text{alt}}, \\ a_{\text{neu}} &= \beta_a \cdot a_{\text{alt}} \end{aligned}$$

[2] vgl. PAL und NTSC, http://de.wikipedia.org/wiki/NTSC#Bildwiederholrate

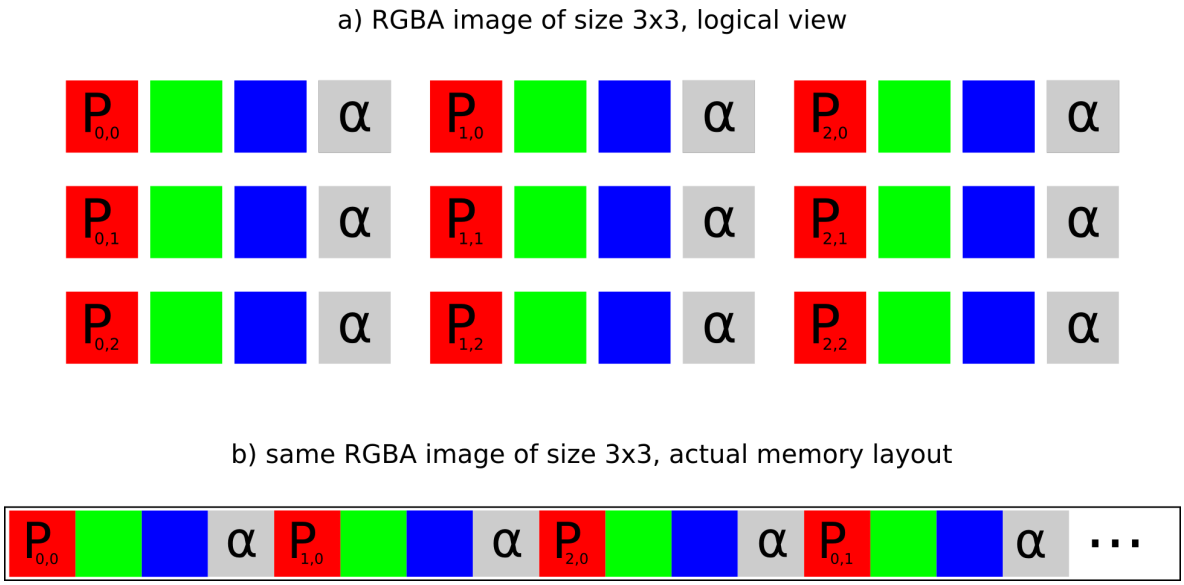

Abbildung 2: Ein zweidimensionale Farbbild besteht aus einzelnen Bildpunkten, die jeweils durch die vier Farbwerte Rot, Grün, Blau und Alpha beschrieben werden. Auch wenn das Bild wie unter (a) dargestellt in zwei Dimensionen interpretiert wird, so liegt es doch physikalisch zusammenhängend in einer eindimensionalen Anordnung im Speicher (b).

angezeigt wird, werden in diesem Versuch die Überblendungen in allen vier Farbkanäle gleichzeitig in einer Vektoroperation durchgeführt:

$$\begin{bmatrix} r_{\text{neu}} \\ g_{\text{neu}} \\ b_{\text{neu}} \\ a_{\text{neu}} \end{bmatrix} = \begin{bmatrix} \beta_r \\ \beta_g \\ \beta_b \\ \beta_a \end{bmatrix} \cdot \begin{bmatrix} r_{\text{alt}} \\ g_{\text{alt}} \\ b_{\text{alt}} \\ a_{\text{alt}} \end{bmatrix}.$$

Der C-Datentyp für SSE-Vektoren der Länge 128 Bit heißt `__m128` und entspricht einem prozessorinternen Register, das mit Hilfe der in Abschnitt 2.1 gezeigten Funktionen gesetzt, geladen, gespeichert und berechnet werden kann. Die Funktion `__mm_load_ps` lädt beispielsweise vier SPFP-Werte aus dem Hauptspeicher in das Prozessorregister, die danach durch `__mm_mul_ps` mit einem zweiten Vektor in einem anderen Register multipliziert werden. Das Ergebnis der Berechnung wird prozessorintern in einem dritten `__m128`-Register gehalten und kann mittels `__mm_store_ps` in den Hauptspeicher zurückgeschrieben werden.

2 Funktionsreferenz

2.1 SSE-Schnellübersicht

- `__m128 _mm_mul_ps(__m128 a , __m128 b );`

  Multiplies the four single-precision, floating-point values of a and b.

- `__m128 _mm_add_ps(__m128 a , __m128 b );`

  Adds the four single-precision, floating-point values of a and b.

- `__m128 _mm_load_ps(float * p );`

  Loads four single-precision, floating-point values.

- `void _mm_store_ps(float *p, __m128 a );`

  Stores four single-precision, floating-point values.

- `__m128 _mm_set_ps(float z , float y , float x , float w );`

  Sets the four single-precision, floating-point values to the four inputs.

- `__m128 _mm_set1_ps(float w );`

  Sets the four single-precision, floating-point values to w.

2.2 Weiterführende Informationen zu SSE

- Intel Intrinsics Guide: http://software.intel.com/sites/landingpage/IntrinsicsGuide
- MUL: http://msdn.microsoft.com/en-us/library/22kbk6t9(VS.80).aspx
- ADD: http://msdn.microsoft.com/en-us/library/c9848chc(VS.80).aspx
- LOAD: http://msdn.microsoft.com/en-us/library/zzd50xxt(VS.80).aspx
- STORE: http://msdn.microsoft.com/en-us/library/s3h4ay6y(VS.80).aspx
- SET: http://msdn.microsoft.com/en-us/library/afh0zf75(VS.80).aspx
- SET1: http://msdn.microsoft.com/en-us/library/2x1se8ha(VS.80).aspx

2.3 Hilfsfunktionen

Die folgenden Hilfsfunktionen werden zur Verfügung gestellt und können in der Implementierung verwendet werden.

- `void black_image (float *image)`

  Erzeugt ein schwarzes Bild und legt es in der Variable `image` ab.

- `void load_next_frame_from_videoA (float *image)`

  Lädt das nächste Bild von Video A in die Variable `image`.

- `void load_next_frame_from_videoB (float *image)`

  Lädt das nächste Bild von Video B in die Variable `image`.

- `void save_frame (*image)`

  Speichert das Bild `image` in den Output-Datenstrom.

- `void save_for_seconds (float *image, unsigned int duration)`

  Speichert das Bild `image` für die Dauer von `duration` Sekunden in den Output-Datenstrom.

Außerdem ist die Konstante `IMAGE_SIZE` definiert. Sie enthält die Anzahl der Bildpunkte des Bildes und kann als Obergrenze für Schleifen verwendet werden.

2.4 Variablen und Datentypen

Ihnen stehen die bereits vordefinierten drei Bildspeicher zur Verfügung: `inputframeA`, `inputframeB` und `outputframe`. Verwenden Sie

- `inputframeA`, um das aktuelle Bild des ersten Videos im Speicher zu halten,
- `inputframeB` für das aktuelle Bild des zweiten Input-Videos und
- `outputframe` für das Bild, wie es in den Ausgabe-Datenstrom geschrieben werden soll.

Ein Beispiel illustriert die Verwendung der Hilfsfunktionen aus Abschnitt 2.3. Es sollen abwechselnd je ein Bild aus Video A und Video B geladen und in den Ausgabestrom gespeichert werden, gefolgt von einem vollständig schwarzen Bild:

```
black_image (outputframe);
while (1) {
   load_next_frame_from_videoA (inputstreamA);
   load_next_frame_from_videoB (inputstreamB);
   save_frame (inputstreamA); /* filled with video frame A */
   save_frame (inputstreamB); /* filled with video frame B */
   save_frame (outputframe);  /* filled with black by black_image */
}
```

3 Aufgaben

3.1 Anzeige eines schwarzen Bildes für zwei Sekunden

Das fertige Video soll in den ersten zwei Sekunden mit einem schwarzen Bild beginnen. Implementieren Sie diese Funktionalität. Aktivieren Sie den ersten Aufgabenblock im Programm durch Ändern der Zeichenkette `#if 0` in die Zeichenkette `#if 1`, kompilieren und starten Sie das Programm mittels

```
$ make run
```

und prüfen Sie eventuell entstandene Dateien im output-Verzeichnis. Mit Hilfe von

```
$ make view
```

können Sie sich bereits die ersten zwei Sekunden Ihres Videoclips anschauen.

3.2 Einblenden des Videos A für zwei Sekunden

Programmieren Sie mit Hilfe der SSE-Befehle eine Einblendung in das Video A über zwei Sekunden Länge. Implementieren Sie also

$$\texttt{outputframe} = \beta \cdot \texttt{inputframeA},$$

wobei $\beta \in [0, 1]$ über den Zeitraum von zwei Sekunden kontinuierlich wächst. Kompilieren und testen Sie das Ergebnis wie oben. Erklären Sie später im Protokoll jede Zeile dieser Einblend-Funktion.

3.3 Anzeige des Videos A für zwei Sekunden

Nachdem das Einblenden abgeschlossen ist, soll nun Video A zwei Sekunden lang angezeigt werden. Implementieren Sie diese Funktionalität und testen Sie das Ergebnis wie oben. Präsentieren Sie (ohne Erklärung) den Code im Protokoll.

3.4 Überblendung des Videos A in Video B über vier Sekunden

Dies dürfte der schwierigste Teil des Versuchs sein. Video A soll dabei über einen Zeitraum von vier Sekunden zu Video B überblendet werden (crossfade). Beachten Sie, dass sich beide Videos bewegen. Folglich implementieren Sie

$$\texttt{outputframe} = \beta \cdot \texttt{inputframeA} + (1 - \beta) \cdot \texttt{inputframeB},$$

wobei $\beta \in [0, 1]$.

Im Versuch ist ein Großteil des dafür benötigten Codes bereits vorgegeben. Erarbeiten Sie daher in der Vorbereitung nur, welche SSE-Befehle Sie benötigen, wo Teilergebnisse auftreten und wie diese miteinander verknüpft werden.

Testen Sie das Ergebnis auf Korrektheit und protokollieren Sie inklusive Erläuterungen Ihren Algorithmus (Code + Erklärung).

3.5 Anzeige des Videos B für zwei Sekunden

Analog zu Abschnitt 3.3 zeigen Sie nun Video B für zwei Sekunden.

3.6 Ausblendung des Videos B für zwei Sekunden nach Grün

Analog zu Abschnitt 3.2 blenden Sie nun Video B über zwei Sekunden lang aus. Diesmal aber nicht zur Fabe schwarz, sondern zur Farbe grün.

3.7 Anzeige eines grünen Bildes für eine Sekunde

Lassen Sie Ihren Film mit einer Sekunde grünem Bild enden. Funktioniert alles zufriedenstellend, informieren Sie Ihren Betreuer und präsentieren Sie ihm Ihr Endergebnis.

3.8 Assembler-Output

Da nun alles funktioniert, lohnt sich ein Blick in den Assemblercode des Programms. Führen Sie dazu

```
$ make asm
```

aus. Dies erzeugt eine Datei `simd.s`. Betrachten Sie die Datei. Suchen Sie dabei nach Vorkommen Ihrer SSE-Befehle (Hinweis: Das Präfix `__mm_` entfällt). Sichern Sie die Datei unter einem anderen Namen. Markieren Sie im Protokoll die SSE-Befehle im Assembler-Output.

Bearbeiten Sie nun erneut den C-Quelltext (`simd.c`) des Programms und ersetzen Sie alle arithmetischen SSE-Befehle (`__mm_mul_ps`, `__mm_add_ps`) durch die gewohnten binären Operatoren '*' und '+'.

Rufen Sie

```
$ make asm
```

auf. Was bemerken Sie? (Hinweis: Der Unix-Befehl `diff -u Datei1 Datei2` hilft Ihnen, Unterschiede zwischen zwei Dateien zu finden).