LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 6
30.05.2022 - 05.06.2022

T6.1 Konvertierung von Strings zu Zahlen

Die Programmargumente in `argv` sind stets Strings, oftmals möchte man diese aber auch als numerische Werte betrachten. Im Folgenden wollen wir uns deshalb mit Funktionen beschäftigen, die eine entsprechende Konvertierung ermöglichen.

Vorlage: `https://gra.caps.in.tum.de/m/stringtonum.tar`

1. Die Funktionen `atol`, bzw. `atof` um Strings in `long`, bzw. `double` Werte zu konvertieren kennen Sie bereits. Warum ist deren Verwendung oft problematisch? Sehen Sie sich hierzu die relevanten `man` pages an.
2. Basierend auf den eben betrachteten `man` pages: Welche weiteren Funktionen bieten sich für die Konvertierung von Strings in `long`, bzw. `double` Werte womöglich an?
3. Finden Sie die Bedeutung der Parameter `const char* nptr` und `char** endptr` der `strtol` und `strtod` Funktionen heraus.
4. Konvertieren Sie nun mit `strtod` die Nutzereingabe in einen `double` Wert. Wie können Sie folgende Fehler abfangen?
   - Die Nutzereingabe repräsentiert keine gültige Fließkommazahl.
   - Die übergebene Fließkommazahl passt nicht in den Wertebereich eines `double`.

S6.1 Grundlagen SIMD

Im bisherigen Verlauf des Praktikums haben wir mit *skalaren* Instruktionen gearbeitet, bei denen in jeder Instruktion genau ein Wert verarbeitet wurde. In vielen Fällen wird jedoch dieselbe Operation auf mehrere Daten angewendet, beispielsweise bei Matrix-Operationen oder Bild-Verarbeitung. Hierfür stellen verschiedene Prozessoren sog. *Vektoreinheiten* zur Verfügung, welche nach dem *SIMD*[2]-Prinzip arbeiten, also die gleiche Instruktion auf mehreren Daten gleichzeitig ausführen.

1. Betrachten Sie folgende Instruktion:

```
addps xmm0,xmm1
```

[2]Single Instruction, Multiple Data

Um welche Register handelt es sich? Was bedeutet das Suffix `ps`? Welche Operation führt diese Instruktion aus?

2. Worin besteht der Vorteil gegenüber der Verwendung von mehreren, skalaren Additionen?

3. Wie unterscheidet sich folgende Instruktion von der obigen?

```
paddd xmm0,xmm1
```

4. Worin besteht der Unterschied zwischen den Instruktionen `movaps` und `movups`? Welche Anforderungen gibt es bezüglich des *Alignment*, wenn ein Speicheroperand bei der Instruktion `addps` verwendet wird? Ziehen Sie auch die Intel-Dokumentation heran.

P6.1 Nutzung von SIMD in Assembler: Saxpy [3 Pkt.]

Im Folgenden werden wir die SSE-Erweiterungen für die Optimierung einer häufig verwendeten Operation der linearen Algebra[4] nutzen. Die Funktion `saxpy` mit folgender Signatur führt folgende Operation durch:

```
void saxpy(size_t n, float alpha, const float x[n], float y[restrict n])
```

$$\vec{y} \leftarrow \alpha \cdot \vec{x} + \vec{y} \qquad \vec{x}, \vec{y} \in \mathbb{R}^n, \alpha \in \mathbb{R}$$

Vorlage: `https://gra.caps.in.tum.de/m/blas.tar`

1. Machen Sie sich klar, in welchen Registern die einzelnen Argumente beim Aufruf der Funktion gemäß der Calling Convention stehen.

2. Sorgen Sie dafür, dass der skalare Wert `alpha` in alle Elemente eines Vektorregisters verteilt wird.

3. Erstellen Sie am Anfang der Funktion eine Schleife, die über die Vektoren x und y in Blöcken von 4 Elementen iteriert und abbricht, sobald der nächste Schleifendurchlauf weniger als 4 Elemente zu verarbeiten hätte.

4. Laden Sie nun innerhalb der Schleife jeweils 4 benachbarte Elemente aus den Vektoren x und y in `xmm`-Register und führen Sie die eigentliche mathematische Operation mit den Instruktionen `mulps` und `addps` aus. Beachten Sie, dass für die Parameter kein *Alignment* spezifiziert ist.

5. Vergewissern Sie sich durch geeignete Aufrufe des Rahmenprogramms, dass Ihre Implementierung (mit Ausnahme der letzten Elemente, sollte die Länge kein Vielfaches von 4 sein) korrekt funktioniert.

[4] Es handelt sich um Funktionen der *Basic Linear Algebra Subprograms (BLAS)*.

6. Vervollständigen Sie Ihre Implementierung, indem Sie am Ende der Funktion die Berechnung mit skalaren Operationen für die verbleibenden Elemente durchführen.

P6.2 Nutzung von SIMD in C: Sdot [3 Pkt.]

Implementieren Sie die Funktion `sdot` unter Nutzung von SIMD-Intrinsics in C, welche das Skalarprodukt von zwei Vektoren berechnet. Die Funktion hat folgende Signatur und führt die untenstehende Berechnung durch:

```
float sdot(size_t n, const float x[n], const float y[n])
```

$$dot = \vec{x}^T \cdot \vec{y} \qquad \vec{x}, \vec{y} \in \mathbb{R}^n$$

Empfehlung: Akkumulieren Sie jeweils das Produkt von vier benachbarten Werten in einem Vektorregister, dessen Elemente Sie nach Ende Ihrer Block-Schleife aufsummieren.

Vorlage: `https://gra.caps.in.tum.de/m/blas.tar`

1. Finden Sie mithilfe des *Intel Intrinsics Guide*[5] eine Funktion, mit der sich ein `__m128` auf den Wert 0 setzen lässt. Nutzen Sie diese, um Ihren akkumulierenden Vektor zu initialisieren.

   Hinweis: Es empfiehlt sich, die *Technologies* auf SSE und SSE2 zu beschränken.

2. Die generelle Struktur der eigentlichen Verarbeitung ist wie oben: zunächst werden 4 Elemente gleichzeitig verarbeitet und am Ende werden die verbleibenden Elemente behandelt. Implementieren Sie entsprechende Schleifen.

3. Laden Sie mittels geeigneter Intrinsics in der SIMD-Schleife jeweils einen Vektor aus `x` und `y` – achten Sie dabei auf den Datentypen und das (nicht garantierte) Alignment. Multiplizieren Sie diese Werte und addieren Sie das Produkt auf den akkumulierenden Vektor, ebenfalls mit geeigneten Intrinsics.

4. Addieren Sie im Anschluss an die SIMD-Schleife alle Elemente des akkumulierenden Vektors. Hierzu können Sie beispielsweise auch geeignete *Move-* oder *Shuffle-*Intrinsics verwenden.

5. Implementieren Sie die skalare Schleife für die verbleibenden Operationen mittels gewohnter Operatoren für Speicherzugriff und Arithmetik.

Q6.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)

[5] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE,SSE2`