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.

   Bei beiden Funktionen ist es nicht möglich, eventuelle Fehler bei der Konvertierung abzufangen. Der relevante Ausschnitt aus der `atol` `man` page hierzu lautet beispielsweise:

   ```
   The atoi() function converts the initial portion of the string
   pointed to by nptr to int. The behavior is the same as

       strtol(nptr, NULL, 10);

   except that atoi() does not detect errors.
   ```

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?

   Zum Konvertieren von Strings in `long` Werte gibt es die Funktion `strtol`, zum Konvertieren von Strings in `double` Werte die Funktion `strtod`.

3. Finden Sie die Bedeutung der Parameter `const char* nptr` und `char** endptr` der `strtol` und `strtod` Funktionen heraus.

   Der Parameter `nptr` gibt den String an, der in eine Zahl konvertiert werden soll. `endptr` fungiert als eine Art zweiter Rückgabewert und ist insbesondere zur Fehlerbehandlung nützlich. `endptr` ist ein Pointer auf einen `char`-Pointer, was der Funktion ermöglicht, den Wert des `char`-Pointers selbst zu verändern. Wird also für `endptr` nicht `NULL` übergeben, so wird der von `endptr` referenzierte Pointer von `strtol`/`strtod` auf den ersten Character gesetzt, der nicht konvertiert werden konnte. Im folgenden Codebeispiel würde `endptr` nach dem Funktionsaufruf also auf den Buchstaben `'x'` zeigen:

```c
char* endptr;
long result = strtol("485X9", &endptr, 10);
```

Bei erfolgreicher Konvertierung des gesamten Strings, zeigt der Pointer auf dessen Nullterminal. `endptr` muss also insbesondere kein Pointer auf einen gültigen String sein, sondern es reicht aus, mithilfe des `&`-Operators z.B. die Adresse eines uninitialisierten `char`-Pointers zu übergeben.

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`.

   Die Fehler lassen sich wie folgt abfangen:
   - ***Hierfür können wir den `endptr` Parameter nutzen. Zunächst prüfen wir, ob `endptr` auf den Anfang des zu konvertierenden Strings zeigt. Ist dies der Fall, so konnte kein einziger Character konvertiert werden. Andernfalls überprüfen wir, ob der erste Character, der nicht konvertiert werden konnte, das terminierende `NULL`-Byte des Strings ist. Falls dem nicht so ist, konnte nicht der gesamte String konvertiert werden und wir brechen auch hier das Programm ab. Eine weniger strikte Fehlerbehandlung ist je nach konkretem Anwendungsfall aber natürlich auch denkbar.***
   - ***Im Fall, dass die übergebene Fließkommazahl nicht in den Wertebereich eines `double` passt, wird die `errno`[1] auf den Wert `ERANGE` gesetzt.* Vor *der Konvertierung kann diese also auf den Wert 0 gesetzt werden und* nach *der Konvertierung mit dem Wert `ERANGE` verglichen werden.***

   Die Umwandlung von `argv[1]` in einen `double` Wert kann also folgendermaßen implementiert werden:

```c
...
errno = 0;
char* endptr;

double x = strtod(argv[1], &endptr);

if (endptr == argv[1] || *endptr != '\0') {
    fprintf(stderr, "%s could not be converted to double\n",
            argv[1]);
    return EXIT_FAILURE;
} else if (errno == ERANGE) {
    fprintf(stderr, "%s over- or underflows double\n", argv[1]);
    return EXIT_FAILURE;
}
...
```

[1] `errno` ist im Header `errno.h` definiert

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.

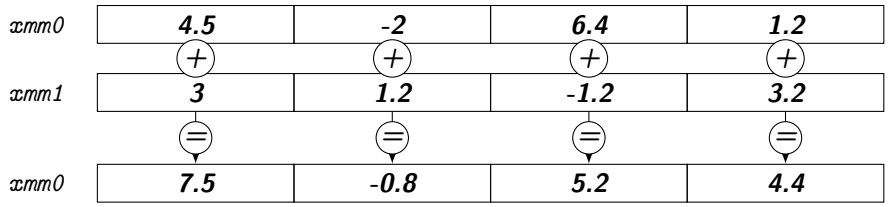
1. Betrachten Sie folgende Instruktion:

   ```
   addps xmm0,xmm1
   ```

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

   Es handelt sich um die SSE-Register. Das Suffix `ps` steht für "Packed Single" und gibt an, dass eine Vektoroperation auf Single Precision Floating-Point-Zahlen durchgeführt werden soll.

   Gemäß dem SIMD-Prinzip führt diese Instruktion insgesamt 4 Floating-Point-Additionen aus, siehe folgende Grafik.

   Analog führt die Instruktion `addpd` 2 Double Precision Floating-Point-Additionen aus.

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

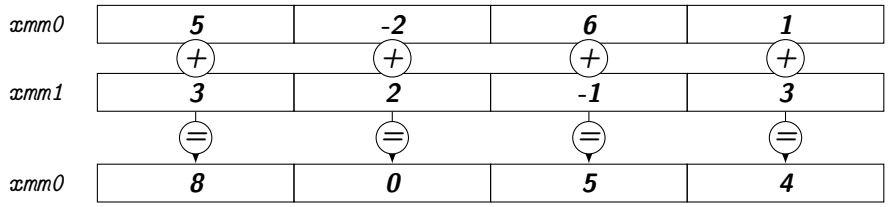
   Kürzere Ausführungszeit: 4 skalare Operationen benötigen das 2–4-fache an Berechnungszeit.

3. Wie unterscheidet sich folgende Instruktion von der obigen?

   ```
   paddd xmm0,xmm1
   ```

[2] Single Instruction, Multiple Data

Es wird mit 32-Bit Integern ("d" für dword) statt Floating-Point-Werten gerechnet.



Analog führen die Instruktionen `paddb`/`paddw`/`paddq` 16/8/2 8/16/64 Bit Integer-Additionen aus.

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.

   Bei `movaps` muss der Speicherbereich 16-Byte-aligned sein (d.h. die unteren vier Bits der Adresse sind 0). Bei `movups` ist dies nicht erforderlich, benötigt dafür aber länger in der Ausführung.

   Wenn bei arithmetischen Instruktionen eine Speicheradresse verwendet wird, muss auch diese immer entsprechend ausgerichtet sein. Dies ergibt sich aus dem Intel SDM, Vol. 2A Sec. 2.4 (AVX and SSE Exception Specification).

   Bei falschem Alignment gibt es seitens des Prozessors die Exception `#GP`[3], welche vom Betriebssystem als* Segmentation Fault *weitergegeben wird.

   Bei AVX-Instruktionen ist Alignment nur noch bei den Instruktionen mit explizitem Alignment (z.B. `movaps`) erforderlich, nicht jedoch aber bei arithmetischen Instruktionen.

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])
```

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

[3] General Protection Fault

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

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.
   - *n* ***in*** `rdi`
   - `alpha` ***in*** `xmm0`
   - *x* ***in*** `rsi`
   - *y* ***in*** `rdx`
2. Sorgen Sie dafür, dass der skalare Wert `alpha` in alle Elemente eines Vektorregisters verteilt wird.

   Hierfür gibt es mehrere Möglichkeiten:
   - `pshufd xmm0, xmm0, 0x00` ***— Der dritte Operand ist eine 8-Bit Konstante. Jeweils zwei Bit geben für ein Element des Zieloperanden (`xmm0`) an, welcher der vier Werte des Quelloperands (`xmm0`) genommen werden soll. 0 bedeutet dementsprend, dass für alle vier Elemente das unterste Element genommen werden soll.***
   - `unpcklps xmm0, xmm0; movlhps xmm0, xmm0` ***— Hier wird erst das unterste Element in die unteren beiden Elemente dupliziert. Danach werden die unteren 64 Bit in die oberen 64 Bit kopiert.***
   - ***Eine Kombination aus*** `movd` ***und*** `pinsrd` ***sollte auch gehen. Da*** `pinsrd` ***allerdings aus einem General-Purpose-Register kopiert, ist diese Variant deutlich aufwendiger und nicht zu empfehlen.***
   - ***Auf Systemen mit AVX2 gibt es*** `vbroadcastss`***.***
   - ***. . . Füge weitere Varianten hier ein*** ☺ ***. . .***
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.

```
1  saxpy:
2    pshufd xmm0, xmm0, 0x00
3    jmp .Lsimd_loop_check
4  .Lsimd_loop:
5    // ...
6    sub rdi, 4
7    add rsi, 16   // shift x pointer
8    add rdx, 16   // shift y pointer
9  .Lsimd_loop_check:
10   cmp rdi, 4
11   jge .Lsimd_loop
```

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.

```
  // ...
  movups xmm1, [rsi]
  movups xmm2, [rdx]
  mulps xmm1, xmm0
  addps xmm2, xmm1
  movups [rdx], xmm2
  // ...
```

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.

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

```
  // ...
  test rdi, rdi
  jz .Lret      // return if n == 0
.Lscalar_loop:
  movss xmm1, [rsi]
  mulss xmm1, xmm0
  addss xmm1, [rdx]
  movss [rdx], xmm1
  add rsi, 4
  add rdx, 4
  sub rdi, 1
  jnz .Lscalar_loop
.Lret:
  ret
```

Gesamtlösung:

```
saxpy:
  // distribute alpha to all elements
  pshufd xmm0, xmm0, 0x00
  jmp .Lsimd_loop_check
.Lsimd_loop:
  movups xmm1, [rsi]
  movups xmm2, [rdx]
  mulps xmm1, xmm0
  addps xmm2, xmm1
  movups [rdx], xmm2
  sub rdi, 4
  add rsi, 16   // shift x pointer
  add rdx, 16   // shift y pointer
```

```
14 .Lsimd_loop_check:
15   cmp rdi, 4
16   jge .Lsimd_loop
17
18   test rdi, rdi
19   jz .Lret       // return if n == 0
20 .Lscalar_loop:
21   movss xmm1, [rsi]
22   mulss xmm1, xmm0
23   addss xmm1, [rdx]
24   movss [rdx], xmm1
25   add rsi, 4
26   add rdx, 4
27   sub rdi, 1
28   jnz .Lscalar_loop
29 .Lret:
30   ret
```

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.

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

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.

Referenzlösung (GCC):

```
float sdot(size_t n, const float x[n], const float y[n]) {
    size_t i = 0;
    __m128 sum = _mm_set_ps1(0);
    for (i = 0; i < (n & ~3ul); i += 4)
        sum += _mm_loadu_ps(&x[i]) * _mm_loadu_ps(&y[i]);
    float res = sum[0] + sum[1] + sum[2] + sum[3];
    for (; i < n; i++)
        res += x[i] * y[i];
    return res;
}
```

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