Untersuchung zur Parallelisierbarkeit der gängigen Implementierungen der Intervallarithmetik

Wissenschaftliches Rechnen, Prof. Auer

Stefan Kapsreiter
E-Mail: s.kapsreiter@hs-wismar.de
www.hs-wismar.de

HOCHSCHULE WISMAR
UNIVERSITY OF APPLIED SCIENCES TECHNOLOGY, BUSINESS AND DESIGN

Zusammenfassung

Diese Arbeit untersucht die Threat-Sicherheit von Intervallarithmetik und die daraus folgenden Auswirkungen auf Effizienz und Laufzeit. Die Problematik wird anhand von Unterteilungsverfahren dargestellt, auf seine atomaren Bestandteile heruntergebrochen und technisch erklärt. Dies wird durch programmierte Beispiele belegt. Zuletzt werden zu beachtende Punkte hervorgehoben.

Inhalt

Unterteilungsstrategien

Grundlagen

Vorausgesetzt:

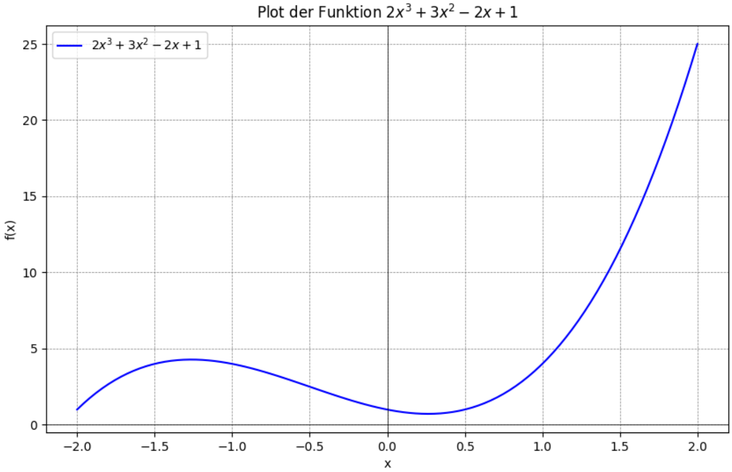
Wissen über Intervalle und dazugehörige Arithmetik

Außerdem:

Intervallauswertung und Überschätzung:

Wiederholung: Bei zusammengesetzten Funktionen wird der Wertebereich bei direkter Auswertung einer Funktion oft überschätzt
-> Abhängigkeitsproblem
-> Einhüllungseffekt

Ein Beispiel: $f(x) = 2x^3 + 3x^2 - 2x + 1$



Beispiel:

$f(x) = 2x^3 + 3x^2 - 2x + 1$ mit $A = [-1, 1]$

Nach $f'(x)$:
y_{min} für $x \in A = 0.7178872292$
y_{max} für $x \in A = 4$

Intervallauswertung:
$f(A) = [-3, 8]$

Beispiel:

Aufteilen des Intervalls:
$A = [-1, 1]- > B = [-1, 0] \cap C = [0, 1]$

$f(B) = [-1, 2]$
$f(C) = [-1, 6]$

$f(B) \cap f(C) = [-1, 6]! = f(A)$

Beispiel:

Weiteres Aufteilen:

Definiere:

Für ein Interval X und eine minimale Intervallgröße w:

$$M_{sub}(X, w) = \{[\underline{X} + k * w, \underline{X} + (k+1) * w] | n = \frac{width(X)}{w}, 0 <= k < n\}$$

und

$$f(M_{sub}) = \text{Vereinigung aller Ergebnisse}$$

Beispiel:

Aus dem Computer mit boost::interval berechnet (abgeschnitten, nicht gerundet):

$$
\begin{aligned}
f(A_{sub}([-1,1],0.5)) &= [-0.5, 5.75] \\
f(A_{sub}([-1,1],0.2)) &= [0.33599\ldots, 4.97599\ldots] \\
f(A_{sub}([-1,1],0.1)) &= [0.52399\ldots, 4.54200\ldots] \\
\ldots \quad &\quad \ldots \\
f(A_{sub}([-1,1],0.0005)) &= [0.71688\ldots, 4.00299\ldots]
\end{aligned}
$$

Erinnerung:

y_{min} für $x \in A = 0.7178872292$

y_{max} für $x \in A = 4$

Floating Point Operations auf der CPU

Problem!

$\frac{2}{0.0005} = 4000$ zu berechnende Intervalle
Gegebene Funktion: 9 Intervall-Operationen -> 3 Multiplikation, 2 Potenzen, 2 Addition, 1 Subtraktion
Multiplikation = 8 Reguläre Arithmetische Operationen
Potenz = in unseren Fällen 4 und 2
Addition = 2
Subtraktion = 2

$$4000 * (3 * 8 + 4 + 2 + 2 * 2 + 2) = 144.000$$

$\rightarrow$ Sehr viele Operationen für genaue Ergebnisse, selbst bei „kurzen” Funktionen.

Das größere Problem:

Dezimalzahlen zu Binärzahlen:

Stelle	Binär	Dezimal
2^3	00001000	8
2^2	00000100	4
2^1	00000010	2
2^0	00000001	1
2^{-1}	0.1	0.5
2^{-2}	0.01	0.25
2^{-3}	0.001	0.125
2^{-4}	0.0001	0.0625

Tabelle 1: Wertigkeiten der Stellen im Binärsystem

Das größere Problem:

0.2 in Binär:

0.2 * 2 = 0.4 -> 0.0...
0.4 * 2 = 0.8 -> 0.00...
0.8 * 2 = 1.6 -> 0.001...
0.6 * 2 = 1.2 -> 0.0011...
0.2 -> Wiederholend

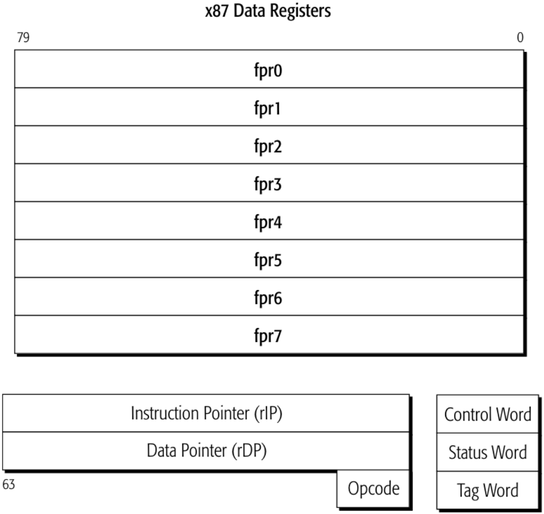
Das größere Problem:

Inklusionseigenschaft:

Bei Maschinenintervallen wird nach außen gerundet. D.h., dass die untere Grenze nach unten und die obere Grenze nach oben gerundet wird.

Boost::interval hat dies implementiert und rundet für lower und upper (zumindest bei float) unterschiedlich. Rundungen werden auf der CPU in der FPU (Floating Point Unit) verarbeitet. Der Interessante Teil ist hierbei das x87 Register, welches nach IEEE 754 standardisiert ist.

Aufbau des x87 Registers



Aufbau des x87 Registers

- FPU hat 8 Register in denen parallel Operationen ausgeführt werden können
- Control World kontrolliert aktuellen Rundungsmodus: to-nearest, toward-zero, downward, upward
- Rundungsmodus gilt für alle 8 Operationen
- Intervallarithmetik benötigt immer auf und abrunden

Theorie 1:

Da ein einziges Flag das gesamte Register kontrolliert, sollten beim schnellen Wechseln von Rundungsmodus Fehler passieren.

Da die genaue Arbeitsweise des Registers nicht nur von der CPU Architektur, sondern auch von Betriebssystem und verwendetem Compiler abhängt, ist es schwer, eine Aussage über die Verifizierbarkeit zu treffen. Es soll so zunächst geprüft werden, ob Fehler messbar sind.

Theorie 1: Test mit 2/10

```
__m128 xmm1 = _mm_loadu_ps(valA);
__m128 xmm2 = _mm_loadu_ps(valB);

for (int i = 0; i < MAX_IT; i++) {
    int mode = rand() % 4;

    switch (mode) {
     case 0: _MM_SET_ROUNDING_MODE(_MM_ROUND_DOWN);
    break;
     case 1: _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);
    break;
     case 2: _MM_SET_ROUNDING_MODE(_MM_ROUND_NEAREST);
    break;
     case 3: _MM_SET_ROUNDING_MODE(_MM_ROUND_TOWARD_ZERO
```

Theorie 1: Test mit 2/10

```
float result[1];

_mm_storeu_ps(result, xmm_result);

if (mode == 0 || mode == 3) {
    if (result[0] > 0.2) {
        std::cout << "Rounding␣error" << std::endl;
    }
}

if (mode == 1 || mode == 2) {
    if (result[0] < 0.2) {
        std::cout << "Rounding␣error" << std::endl;
    }
}
```

Theorie 1: Test mit 2/10

Mit 10.000 Threads, welche diese Berechnung je 10.000 mal ausführen: Kein Fehler!
Es wurden 2 Prozessoren (AMD Ryzen 9 5950X und Intel i7 10500u), Windows 10 und 11, Ubuntu 22.04.2 je mit den neusten GNU Compilern getestet.

Folgerung: Sicherheitsmechanismus, welcher Verifizierbarkeit sicherstellt.

Theorie 2:

FPU wartet, bis der richtige Rundungsmodus eingestellt ist. Dies sollte einen messbaren Unterschied in der Laufzeit bei Multithreading verursachen.

Da ein Fehler trotz expliziten Versuchen nicht aufgetreten ist, soll nun die Laufzeit untersucht werden. Hierfür wird eine zweite Funktion erstellt, welche die selbe Berechnung jedoch mit festgelegtem Rundungsmodus berechnet.

Theorie 2: rounding_random.cpp

```
for (int i = 0; i < MAX_IT; i++) {
    _MM_SET_ROUNDING_MODE(_MM_ROUND_DOWN);
    __m128 xmm_result = _mm_div_ps(xmm1, xmm2);
    float result[1];
    _mm_storeu_ps(result, xmm_result);

    _MM_SET_ROUNDING_MODE(_MM_ROUND_UP);
    xmm_result = _mm_div_ps(xmm1, xmm2);
    _mm_storeu_ps(result, xmm_result);
...
```

Theorie 2: rounding_fixed.cpp

```
for (int i = 0; i < MAX_IT; i++) {
    __m128 xmm_result = _mm_div_ps(xmm1, xmm2);
    float result[1];
    _mm_storeu_ps(result, xmm_result);

    xmm_result = _mm_div_ps(xmm1, xmm2);
    _mm_storeu_ps(result, xmm_result);
...
```

Theorie 1: Test mit 2/10

Mit 10.000 Threads, welche diese Berechnungen je 10.000 mal ausführen:
Auf Ryzen 9 5950x, Ubuntu 22.04.2 (WSL), g++ 11.4.0:

Execution time with random rounding: 327 ms
Average time per calculation: 3 ns
———
Execution time with fixed rounding: 234 ms
Average time per calculation: 2 ns
[1]

[1] Durchschnittlich liegt der Unterschied bei etwa 80 ms

Folgerung:

Das Rechnen mit schnell wechselnden Rundungsmodi ist auf der CPU sicher, kostet jedoch zwischen 20 und 30 % zusätzliche Rechenzeit.
Viele Intervalloperationen sind somit sehr teuer.
Außerdem sehr viele kleine Berechnungen, also eher eine Aufgabe für die GPU!

Codebeispiele:

Unterteilen von Intervallen:
split_intervals.cpp

Runden von Intervallen:
rounding_test.cpp

Prüfung auf Fehler bei Multithreading:
manual_rounding_modes.cpp

Laufzeittest von Rundungsmodi:
rounding_fixed.cpp
rounding_random.cpp

GPU

Runden auf der GPU

- Berechnen von unterteilten Intervallen erfordert hohe Mengen an (oft) wenig komplexen Operationen
- GPU eignet sich für solche Aufgaben
- Hohe Parallelisierbarkeit verringert Laufzeit signifikant
- Keine untersuchbaren Implementierungen da es keine Bibliotheken gibt
- Nvidia stellt beispielhaften Intervall-Header bereit, welcher boost::interval nachvollzogen ist

Runden auf der GPU

4.5. Differences from x86

NVIDIA GPUs differ from the x86 architecture in that rounding modes are encoded within each floating point instruction instead of dynamically using a floating point control word. Trap handlers for floating point exceptions are not supported. On the GPU there is no status flag to indicate when calculations have overflowed, underflowed, or have involved inexact arithmetic. Like *SSE*, the precision of each GPU operation is encoded in the instruction (for x87 the precision is controlled dynamically by the floating point control word).

Beispiel auf der GPU

Selbes Programm aber in CUDA:

Execution time with random rounding: 35 ms
Average time per calculation: 0 ns
———
Execution time with fixed rounding: 34 ms
Average time per calculation: 0 ns
———

Woher der Unterschied?

- Compiler nimmt automatisch Optimierungen vor
- Lineare und „unnötige” Programmverläufe werden gekürzt
- Kaum messbare Unterschiede in Laufzeit
- Keine Rundungsfehler

Codebeispiele:

Effizienzprüfung von Rundungen:
rounding_cuda.cu

Intervallunterteilungen:
intervals_cuda.cu

Auswirkungen auf Intervallarithmetik

Fazit

- Wechseln von Rundungsmodus kostet Zeit auf CPU
- Laufzeit von CPU allgemein länger als auf GPU
- GPU bietet sichere Ergebnisse zu Bruchteil der Zeit
- Beide rechnen verifiziert und erfüllen Einschlusskriterium

Was kann getan werden?

Option 1:
Auf der GPU arbeiten!

Option 2:
Rundungen auf der CPU minimieren. Allerdings wird dies in gängigen Implementierungen nicht getan bzw. unterstützt. Optimierungen durch Aufteilung nach Rundungsmodi wäre denkbar, jedoch müssen hierfür alle Intervalle bekannt sein. D.h. Unterteilungsstrategien wie rekursives teilen in der Mitte bis Mindestbreite erreicht wird, nicht möglich.

Literatur

AMD Programmer Guides:

https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf

https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/26569.pdf

Allgemeine Chip Infos:

https://de.wikipedia.org/wiki/X87

https://en.wikichip.org/wiki/amd/ryzen_9/5950x

https://www.website.masmforum.com/tutorials/fptute/fpuchap1.htm

https://www.gnu.org/software/libc/manual/html_node/Rounding.html

https://maxwelllefevre.wordpress.com/2015/02/19/fpu-rounding-in-different-architectures-presentation/

Literatur

Nvidia CUDA Floating Points:
https://docs.nvidia.com/cuda/floating-point/#differences-from-x86
https://docs.nvidia.com/cuda/cuda-c-programming-guide/