Die Vektoreinheit - dein Freund und Helfer

Mehr Performance zum Nulltarif?

Dr. Andreas Ehmanns, MBDA Deutschland GmbH

Vektoreinheiten sind seit vielen Jahren in den gängigen Prozessorfamilien Standard und dennoch werden sie - auch im embedded Bereich - häufig von den Softwareentwicklern nicht verwendet. Gerade bei ARM-Prozessoren ist die Entwicklung in diesem Bereich in den letzten Jahren deutlich voran geschritten und eröffnet dem (embedded) Entwickler neue Leistungsbereiche. Von diesen sehr leistungsfähigen Einheiten Gebrauch zu machen und zu entscheiden, wann deren Nutzen brauchbare Vorteile bringt, erscheint häufig wesentlich schwieriger, als es wirklich ist.

Beim Namen Vektoreinheit denken Software-Entwickler häufig, dass dies irgendetwas mit Super-Computing zu tun hätte und sind sich gar nicht bewusst, das aber auch nahezu jeder PC, die meisten Tablets und Smartphones und auch viele embedded Geräte SIMD-Einheiten verbaut haben. SIMD (Single Instruction, Multiple Data) bezeichnet die Ausführung einer Instruktion auf mehreren Datenelementen. Je nach Prozessor-Familie werden unterschiedliche Namen für die verbauten SIMD-Einheiten verwendet. Bekannte Vertreter dieser Gruppe sind:

- Intel x86: MMX, SSE/AVX
- AMD x86: 3DNow!
- Power-Familie: Altivec
- ARM: NEON

Historie

Obwohl es die ersten Versionen dieser Einheiten schon Ende der 90er Jahre gab, fanden sie nur zögerlich Unterstützung durch Software. Da die SIMD-Einheiten durch „normal“ geschrieben Programme nicht genutzt werden, müssen sie explizit durch die Software programmiert werden. Erst die Einführung der „Autovektorisierung“, bei der der Compiler versucht, parallel ausführbare Rechenaufgaben zu erkennen und die Vektoreinheit mit zu benutzen, verbesserte die Situation, allerdings erreichen auch aktuelle Compiler (Stand Herbst 2018) noch nicht die Performance, die mit einer direkten Programmierung der SIMD-Einheit möglich ist. Im Bereich Grafik und Multimedia erkannte man zuerst die Möglichkeiten der Leistungssteigerung durch die SIMD-Einheit, da hier häufig gleiche Operationen auf mehrfachen Daten (z.B. Pixeln) auszuführen sind. Dies mag auch ein Grund dafür sein, warum einige Software-Entwickler die SIMD-Einheiten auch heute - rund 20 Jahre später – immer noch nur mit Grafik und Multimedia in Verbindung bringen. Abgesehen von spezielle Datentypen (wie z.B. „Pixel“ bei Altivec) sind diese Einheiten jedoch nicht auf einen bestimmten Einsatzzweck beschränkt, sondern lassen sich für allerlei Rechnungen hernehmen, sofern die Rechenaufgabe parallelisierbar ist.

Auch wenn die Nutzung der SIMD-Einheiten heute wesentlich mehr verbreitet ist, so hält sich bei einigen Software-Entwicklern immer noch hartnäckig das Gerücht, dass es sehr kompliziert sei, die Vektoreinheiten zu programmieren und dass man ein Hardware- und/oder Assembler-Guru sein müsse. Compiler, wie z.B. der gcc, bieten

sogenannte Builtins, d.h. die Benutzung der Befehle zur Programmierung der Vektoreinheit können ganz einfach durch Parameter eingeschaltet werden, z.B. durch `-maltivec -mabi=altivec` beim PowerPC, oder durch `-mavx2` für x86 AVX2. Die dann vorhandenen sogenannten Intrinsics sind eine Art C-API und mappen in den meisten Fällen 1-1 auf die entsprechenden Assembler-Befehle. Umfangreiche Dokumentationen zu den einzelnen Funktionen, den Parametern und Varianten sind im Internet zu finden.

Voraussetzungen
Bevor es an die Nutzung der SIMD-Einheit geht, stellt sich zunächst die Frage, ob der zu optimierende Code parallelisierbar ist, oder ob einzelne Rechenschritte voneinander abhängen. Klassische Beispiele für gut parallelisierbare Rechenaufgaben sind Schleifen, bei denen die Berechnung der einzelnen Durchläufe nicht von den Ergebnissen der vorherigen abhängen. Also z.B. Vektoraddition, -Multiplikation, das Rechnen mit Matrizen und vieles mehr.

Weiterhin muss vorher geklärt werden, ob die Hardware eine SIMD-Unterstützung hat und in welchem Umfang. Gerade bei ARM gibt es auch durchaus den Fall, dass die SIMD-Einheit komplett fehlt. Und umgekehrt bei x86 ist die Fülle der Erweiterungen, die Intel regelmäßig eingeführt hat, sehr groß und auf den ersten Blick unübersichtlich. Angefangen von MMX über SSE, SSE2, … AVX, AVX-256, AVX-512, …
Ja nach Erweiterung, sind dann bestimmte Befehle verfügbar, oder nicht. Genauere Informationen dazu sind z.B. in den Referenzdokumenten im Internet zu finden.

Umsetzung
Ist die Software-Aufgabe parallelisierbar und ist die Unterstützung der Hardware vorhanden, so steht der Nutzung der SIMD-Einheit nichts mehr im Weg. Aktuelle Compiler unterstützen problemlos die gängigen SIMD-Einheiten und ihre Varianten.

Ein grundsätzliches Thema bei der Programmierung von SIMD-Einheiten ist das Format der Daten. Die meisten aktuellen Einheiten haben eine Breite von 128 Bit (Ausnahmen: AVX-256: 256 Bit, AVX-512: 512 Bit), d.h. es passen z.B. vier 32-Bit Float oder vier 32-Bit breite Integer hinein. Dafür gibt es eigene Datenformate, was wiederum bedeutet, dass Daten, die in einem nicht-Vektorformat vorliegen, erst in diese kopiert werden müssen und nach der Berechnung eventuell auch wieder zurück gewandelt werden. Dazu gibt es spezielle Funktionen, die diese Aufgabe elegant übernehmen.
Beispiel: Ein 4-fach Vektor für 32-Bit Float heißt bei NEON: `float32x4_t`

Vorsicht: Es gibt viele Beispiele im Internet, bei denen ein `union` verwendet wird, um von der Vektoreinheit aus auf die Daten zuzugreifen und sich den Kopiervorgang zu sparen. Dies setzt allerdings voraus, dass die Originaldaten im Speicher 16 Byte aligned sind, also genau die Breite der Vektoreinheit. Das ist in der Regel zwar der Fall, aber durch den Compiler nicht immer garantiert. Ist dies nicht der Fall, so gibt es keine Fehlermeldung oder Exception, sondern die unteren vier Bits der Adresse (bei Altivec) werden einfach ignoriert und die verwendeten Daten sind korrupt. Wer auf Nummer sicher gehen will, sollte die vorhanden Funktionen zum Konvertieren verwenden.

Sind die Daten im richtigen Format, kann es mit der Programmierung losgehen. Hier empfiehlt es sich, die Referenz-Handbücher ein wenig zu studieren, um zu sehen, welche Funktionen überhaupt angeboten werden. Das sind zum einen mathematische Funktionen (im Wesentlichen die Grundrechenarten), aber auch Kopier-Funktionen, Bit-Manipulations-Funktionen und einige mehr. Wer mehr als die Grundrechenarten braucht (z.B einen Sinus, oder die Exponential-Funktion), wird jedoch schnell an die Grenzen stoßen. Allerdings gibt es im Internet sehr viele Bibliotheken, die genau diese fehlenden Funktionen anbieten. Selten wird es vorkommen, dass eine Funktion selbst implementiert werden muss. Dann empfiehlt es sich, auf die mathlib zurückzugreifen und die benötigte Funktion in der gewünschten SIMD-Syntax nachzubilden.

Ist eine bereits klassisch implementierte Funktion (oder ein Teil davon) auf die Verwendung der SIMD-Einheit umgestellt, so sollte als erstes überprüft werden, ob die Funktion korrekt funktioniert. Dazu empfiehlt es sich, Referenz-Eingangs und –Ausgangs Daten zu erzeugen und die Korrektheit der Implementierung damit zu überprüfen. Dabei ist darauf zu achten, dass sämtliche Verzweigungen im Programm durchlaufen werden, um nicht Fehler in ungetesteten Zweigen zu übersehen.

Funktioniert das Programm korrekt, so können die Früchte der Arbeit geerntet werden. Häufig ist die Erwartung, dass der optimierte Teil des Programmes um einen Faktor vier schneller wird, wenn man z.B. mit Int32 arbeitet, da mit der SIMD-Einheit vier Berechnungen gleichzeitig ausgeführt werden können. Diese Annahme berücksichtigt aber nicht, dass die SIMD-Einheit je nach Architektur und Design eine komplett eigene Anbindung an die Peripherie und den Speicher hat. Nicht selten sind Geschwindigkeitssteigerungen von deutlich mehr als einem Faktor vier möglich. Allerdings kann auch der umgekehrte Fall eintreten. Insbesondere dann, wenn wenig gerechnet werden muss, wenn große Datenmengen zwischen SIMD- und nicht-SIMD Datentypen hin- und her-kopiert werden müssen, oder die Hardware (z.B. die Speicheranbindung) den Engpass darstellt.

Eine zuverlässige Aussage, welcher Performance-Gewinn bei der Umstellung auf die SIMD-Einheit zu erwarten ist, kann in aller Regel nicht getroffen werden, da diese von vielen Faktoren abhängt. Prinzipiell empfiehl es sich zunächst einmal anzuschauen, an welchen Stellen im (nicht-SIMD) Code die meiste Zeit verbraucht wird (Profiling). Diese Codestellen oder Funktionen empfehlen sich dann dafür, umgeschrieben zu werden. Erst so bekommt der Entwickler in Kombination mit dem Compiler und der Ziel-Hardware eine verlässliche Aussage über die Performance. Da sie stark von den verwendeten Instruktionen abhängt, lässt sie sich nur bedingt auf andere Codeteile übertragen.

Trotz der vielen „aber“, wird der Software-Entwickler am Ende des Tages entlohnt, wenn er sieht, welcher Performance-Gewinn mit den SIMD-Einheiten möglich ist. Verschiedene Algorithmen haben gezeigt, dass unter optimalen Voraussetzungen der Parallelisierbarkeit Geschwindigkeitssteigerung von Faktor 10-15 möglich sind.

Autovektorisierung

Eine recht einfache Möglichkeit, die Leistung der SIMD-Einheit zumindest teilweise nutzen zu können, stellt die Autovektorisierung dar. Die meisten gängigen Compiler unterstützen diese und gerade in den letzten Jahren, sind die Compiler in diesem

Bereich erheblich besser geworden. Bei der Autovektorisierung versucht der Compiler selbständig, geeignete Codeteile in SIMD-Form zu bringen und damit die Vektoreinheit zu nutzen, ohne dass der Programmierer explizit entsprechende Intrinsics bei der Erstellung seines Programmes verwenden muss. Wieviel Performance-Gewinn dadurch erzielbar ist, hängt zum einen davon ab, wie das Programm aufgebaut ist und welche Rechenoperationen verwendet werden und welcher Compiler in welcher Version verwendet wird. Existiert bereits ein Programm, so lohnt es sich auf jeden Fall, die Autovektorisierung auszutesten. Allerdings sollte auch hier sicherheitshalber überprüft werden, dass das Programm mit dieser Option die richtigen Ergebnisse liefert.

Beim gcc lässt sich die Autovektorisierung mit dem Switch `-ftree-vectorize` einschalten. Der gcc schaltet diese Option erst bei `–O3` automatisch ein, allerdings gibt es auch Compiler und/oder Wrapper, die diese Option schon bei `-O2` aktivieren.

Um ein Bild zu bekommen, was der Compiler bei der Autovektorierung macht, hilf die `-ftree-vectorizer-verbose=N Option.`
Ist hingegen die Autovektorisierung nicht erwünscht, oder soll sie zur Ausführung von Vergleichsmessungen explizit ausgeschaltet werden, kann dies mit Hilfe der Option `–fno-tree-vectorize` geschehen.

Auch wenn die Compiler in den letzten Jahren immer besser geworden sind, so ist mit einer direkten Programmierung der SIMD-Einheit in aller Regel deutlich mehr an Performance zu gewinnen, als mit der Autovektorisierung. Wie groß der Gewinn sein wird, ist auch hier sehr schwer abschätzbar und nur durch eine reale Messung bestimmbar.

Ein Nachteil der SIMD-Einheiten soll hier nicht verschwiegen werden. Dadurch dass die Intrinsics spezifisch für eine Prozessorfamilie sind, leidet die Portierbarkeit des Codes. Die Prozessorhersteller achten zwar darauf, dass neuere Versionen einer Vektoreinheit auch alle alten Befehle unterstützen, bei Wechsel der Prozessorfamilie jedoch ist eine Portierung des SIMD-Codes erforderlich. Generell empfiehlt es sich, die Codeteile, die SIMD-Befehle benutzen, in einem Wrapper oder ähnlichem Konstrukt zu kapseln, soweit dies das Design der Applikation(en) ermöglicht.

Zusammenfassung
Ist eine Performancesteigerung eines Programmes gewünscht oder erforderlich, so lohnt es sich in vielen Fällen, die Vektoreinheit der Hardware (soweit vorhanden) zu nutzen. Autovektorisierung durch den Compiler kann in einfacheren Fällen schon eine Steigerung der Rechenleistung bringen, eine direkte Programmierung der SIMD-Einheit bietet jedoch bei parallelisierbarem Code meist einen wesentlich höheren Gewinn, der unter optimalen Bedingungen sogar Faktoren betragen kann. Gerade im embedded Bereich, wo die Rechenleistung häufig deutlich limitierter ist, kann damit aus der vorhandenen Hardware entscheidend mehr Leistung herausgeholt werden.

Literaturverzeichnis

[1] Altivec Technology Programming Interface Manual
www.nxp.com/docs/en/reference-manual/ALTIVECPIM.pdf

[2] NEON Intrinsics Reference
developer.arm.com/technologies/neon/intrinsics

[3] Intel Intrinsics Guide
software.intel.com/sites/landingpage/IntrinsicsGuide/

Autor

Andreas Ehmanns beschäftigt sich seit mehr als 20 Jahren mit eingebetteten Systemen und den Herausforderungen für weiche und harte Echtzeit-Systeme. Schon Ende der 90er Jahre begann er damit, Linux sowohl auf etablierten als auch auf neuen Systemen mit Echtzeitanforderungen einzusetzen. Er arbeitet als 'Technischer Berater für Embedded Software Systeme' und untersucht unter anderem die Eignung verschiedener Prozessorarchitekturen für den Einsatz im embedded Bereich.

Kontakt

universeii@gmx.de