Vektorisierung in OpenMP: SIMD-Parallelität

Seminararbeit
Dlyaver Djebarov

Lehrstuhl für Hochleistungsrechnen, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Betreuer: Dipl.-Inf. Christian Terboven

Um bei der Softwareentwicklung oder beim Hochleistungsrechnen eine maximale Leistung zu erzielen, ist es üblich, mit einer großen Datenmenge zu arbeiten. Die serielle Verarbeitung solcher Daten kann jedoch beträchtliche Zeit in Anspruch nehmen, weshalb eine Vektorisierung und Parallelisierung der Berechnungen erforderlich ist. Für eine effizientere Vektorisierung enthält OpenMP eine spezielle SIMD-Direktive, deren Funktionsweise in dieser Arbeit beschrieben wird.

Keywords: OpenMP, SIMD, Vektorisierung

1 Einleitung

Die Rechenleistung der Welt nimmt täglich zu und um mit dem Fortschritt Schritt zu halten, müssen die Menschen in der Lage sein, relativ schnell Rechnungen durchführen zu können. Prozessoren spielen dabei sicherlich eine wichtige Rolle, aber die Erhöhung der Anzahl der Transistoren bei gleichzeitiger Verringerung ihrer Größe scheint bereits eine technische Herausforderung zu sein, und so haben sich Ingenieure dafür entschieden, Multicore-Prozessoren zu entwickeln, anstatt die Frequenz eines einzelnen Kerns weiter zu erhöhen. Aus diesem Grund ist die Parallelisierung einer der Schlüssel zum Hochleistungscode. Auf diese Weise kann der Programmierer die Möglichkeiten des Computersystems in vollerem Umfang nutzen.

Gleichzeitig nimmt die Menge der zu verarbeitenden Daten ständig zu. Dabei handelt es sich sowohl um alltägliche Aufgaben wie die Kodierung/Dekodierung von Audio- und Videodaten als auch um komplexe wissenschaftliche Berechnungen. SISD-Systeme („Single Instruction, Single Data“ oder „Einzelne Anweisung, Einzelne Daten“) schneiden unter diesen Bedingungen schlecht ab, da sie grundsätzlich nicht in der Lage sind, eine Datenmenge mit einem einzigen Prozessorbefehl zu verarbeiten.

2 SIMD

Nach der flynnsche Klassifikation [1] steht SIMD für „Single Instruction, Multiple Data“ oder „Einzelne Anweisung, Mehrere Daten“. Wie der Name schon sagt, bedeutet dieses Rechenmodell, dass mehrere Daten mit einer einzigen Anweisung verarbeitet werden können. „Das SIMD-Modell ist für die potenzielle Leistung des Prozessors so wichtig, dass bei Nichtverwendung von SIMD weniger als die Hälfte des Prozessors tatsächlich genutzt wird.“ [2]

Berechnungen mit SIMD-Anweisungen werden als vektorbasiert bezeichnet und auf speziellen Vektorprozessoren durchgeführt. Der Unterschied zwischen Vektorprozessoren und Skalarprozessoren besteht darin, dass Vektorprozessoren sogenannte Vektorregister (SIMD-Register) verwenden, die mehreren Skalarregistern entsprechen - die Anzahl der Skalare, die gleichzeitig verarbeitet werden können, wird als Vektorlänge bezeichnet. Ein Vektorregister kann man sich als ein eindimensionales Array mit einer festen Größe in Bytes vorstellen (siehe Abbildung 1) [2]. Da ein Vektorregister eine Reihe von Daten speichern kann, ist es möglich durch den Betrieb von Vektorregistern, mehrere Daten parallel so schnell wie skalare Befehle zu verarbeiten, aber das Verschieben von Daten in und aus Vektorregistern nimmt einige Zeit in Anspruch, was manchmal die Leistung beeinträchtigen kann. Die logische Schlussfolgerung ist, dass je größer die Länge des Vektorregisters ist, desto mehr Daten können parallel verarbeitet werden und desto größer ist die Codeleistung.

Compiler sind aktiv an der Code-Autovektorisierung beteiligt, liefern aber nicht immer effektive Ergebnisse, da ihre Möglichkeiten begrenzt sind - ist es manchmal unmöglich, genügend Informationen aus dem Quellcode

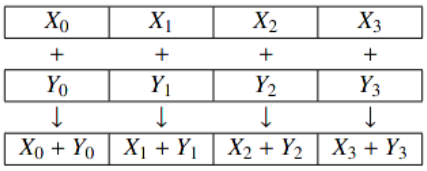

Abbildung 1: Vertikale Additionsoperation bei Vektorregistern mit vier Skalaren

zu extrahieren, um eine effektive und sichere Optimierung durchzuführen. (cf. Rice's Theorem [3])

2.1 SIMD-Unterstützung durch Compiler und Bibliotheken

Damit die Vektorisierung und die Autovektorisierung funktionieren, ist es oft notwendig, mit den Fähigkeiten eines bestimmten Compilers vertraut zu sein und die notwendigen Flags zu kennen, um den Quellcode korrekt zu kompilieren. Für den GNU GCC 11.1.0 Compiler wäre der erforderliche Satz von Flags für eine effiziente Kompilierung zum Beispiel `-ftree-vectorize -lmvec -ffast-math -march=native -O3`. Für den Intel 18.0.2 Compiler ist der folgende Satz von Flags zu bevorzugen: `-xHost -fp-model=fast=2 -O3`.

Das Thema der vektorisierten mathematischen Bibliotheken verdient eine besondere Aufmerksamkeit. Ihr Hauptunterschied zu herkömmlichen (skalaren) Bibliotheken besteht darin, dass jede Funktion in ihnen zeitgetestet und unter Verwendung von SIMD-Befehlen neu geschrieben wurde, was die Verwendung mit SIMD-Vektoren ermöglicht. Beispiele für solche Bibliotheken sind `libmvec` für GCC-Compiler und `SVML` für Intel-Compiler.

2.2 SIMD-Befehle

Die Vektorregister werden über einen erweiterten Befehlssatz gesteuert. Genau so wie reguläre (skalare) Anweisungen können diese grundlegende Berechnungen wie Addition, Multiplikation, Subtraktion, bitweise Disjunktion, Konjunktion, Negation sowie bitweise Verschiebungen usw. durchführen. Beim Umgang mit reellen Zahlen werden sie durch die Operationen der Quadratwurzelung, der reellen Division und andere ergänzt. Es ist wichtig, daran zu denken, dass Operationen mit reellen Zahlen vom Prozessor 3-4 mal langsamer ausgeführt werden können als ihre ganzzahligen Entsprechungen, daher ist es besser, wenn möglich ganzzahlige Typen zu verwenden. [2]

Vektorregisteroperationen können in zwei Arten unterteilt werden: vertikal und horizontal. Die vertikale Datenverarbeitung kann so aussehen, dass zwei Vektorregister genau übereinander gelegt werden, so dass ihre Skalarpositionen übereinstimmen, und dann eine Berechnung mit jedem Skalarpaar durchgeführt wird. Horizontale Operationen hingegen werden mit nur einem Vektorregister und den darin befindlichen Skalaren durchgeführt. Ein gutes Beispiel ist die Berechnung der Summe der Elemente eines Vektorregisters. Die Skalare werden dann sozusagen "horizontal in eins gepackt".

2.3 Speicherausrichtung

Der Prozessor liest Daten nicht Byte für Byte aus dem Speicher, sondern in Blöcken, deren Größe Zweierpotenzen sind und sich je nach Prozessor unterscheiden. Solche Blöcke werden als Cache-Zeilen bezeichnet und haben einen festen Anfang und ein festes Ende im RAM. Durch diesen Mechanismus können mehr lokale Daten im Cache des Prozessors gespeichert werden, was die Laufzeit von Programmen verringert.

Bei der Arbeit mit Datenvektoren, die sich im Speicher befinden, besteht immer die Gefahr, dass sich ein Vektor gleichzeitig teilweise in mehreren Cache-Zeilen befindet, was dazu führt, dass zusätzliche Daten in den Cache des Prozessors geladen werden und verlangsamt das Programm um ein Vielfaches.

Es ist daher notwendig, die Anordnung der Daten im Speicher im Auge zu behalten und sie an den Cache-Zeilen auszurichten. Wenn die Daten in einem Stack gespeichert sind (in C und C++ sind dies lokale Funktionsvariablen), richten die Compiler sie in der Regel automatisch aus, aber wenn der Speicher dynamisch zugewiesen wird, gibt es keine Garantie, dass ein Zeiger auf die ausgerichtete Speicherstelle zurückgegeben wird. Um den bereits ausgerichteten Speicher zuzuweisen, können Funktionen aus Standardbibliotheken oder Betriebssystemen wie `posix_memalign()` [4] oder `aligned_alloc()` [5] verwendet werden. Manchmal werden auch verschiedene Hacks verwendet, um Daten im zugewiesenen von `malloc()` Speicher auszurichten.

Ein weiteres Problem ist, dass einige der Befehle im erweiterten Befehlssatz nur mit ausgerichteten Daten arbeiten und einen Segmentierungsfehler verursachen können, wenn das Argument falsch ist. Da die Startadresse des dynamisch zugewiesenen Speichers fast zufällig ist, tritt der Fehler möglicherweise nicht jedes Mal auf, was die Suche erheblich erschwert und zusätzliche menschliche Ressourcen erfordert.

3 SIMD in OpenMP

Um effektiven vektorisierten Code zu schreiben, ist es natürlich wichtig, viel über die Architektur der Maschine zu wissen, auf der der Code ausgeführt werden soll. Programmierende werden unweigerlich mit der Notwendigkeit konfrontiert, den Speicher besonders sorgfältig zu überwachen. Wie bereits erwähnt, führt die Vektorisierung nicht immer zu einer höheren Codeleistung. So sollten Programmierende in der Lage sein, die Kosten für die Übertragung von Daten in den Vektorregister und aus dem Vektorregister abzuschätzen. Die Übertragbarkeit des Codes wird sehr schwierig sein, da nicht klar ist, welche Vektorerweiterung zu verwenden ist. Dadurch wird Code in mehreren Versionen erzeugt, was eine schlechte Praxis ist.

Eine Lösung für diese Probleme könnte ein Werkzeug sein, das es dem Programmierer ermöglicht, Code auf einer abstrakteren Ebene zu vektorisieren, indem er „syntaktischen Zucker“ verwendet, d.h. spezielle Direktiven, die nur angeben, welche Aktionen ausgeführt und welche Datenströme parallelisiert werden sollen. Ein solches Tool wurde auf dem Eighth International Workshop on OpenMP in Rom, Italien, im Juni 2012 in einem Forschungspapier mit dem Titel „Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures“ [6] vorgeschlagen.

Die Idee war, zu OpenMP-Direktiven eine weitere für die Vektorisierung des Codes zuständige Direktive hinzuzufügen. Eine solche Anweisung, kombiniert mit inhärenten Klauseln, würde festlegen, welche Daten in vektorisiertem Code privat sein sollten (siehe Abschnitt 3.1), welche Vektorlänge akzeptabel ist und wie die Einrückung für vektorisierte Daten aussieht. In demselben Papier wurde vorgeschlagen, auch Funktionen zu vektorisieren, so dass man mit Vektoren statt mit Skalaren arbeiten kann. Ein wichtiger Faktor war, dass der Programmierer keinen Low-Level-Code mehr schreiben oder auf eine konkrete Architektur abstimmen musste - die gesamte Arbeit wurde von OpenMP übernommen.

Die Idee wurde verfeinert und in den neuen Standard OpenMP 4.0 aufgenommen. Im Allgemeinen können alle SIMD-Direktiven entsprechend ihrer Funktionalität in drei Gruppen unterteilt werden: Schleifen-Direktiven, Block-Direktiven und Direktiven zur Steuerung der Funktionsvektorisierung.

3.1 Schleifen-Direktiven

Um die Vektorisierung mit Daten in for-Schleifen zu verwenden wird die folgende Anweisung verwendet:

```
#pragma omp simd [Klausel]
```

Die Direktive wird vor Beginn der Schleife geschrieben und kann mit verschiedenen Klauseln erweitert werden, die die Art der Daten und Operationen in der Schleife für den Compiler spezifizieren. Es ist wichtig, daran zu denken, dass eine Schleife nicht die Schlüsselwörter `break`, `continue` oder `return` enthalten darf, da sie sonst nicht mit SIMD-Anweisungen kompiliert werden kann.

Wenn das obengenannte SIMD-Konstrukt auf eine Schleife angewandt wird, wird die Schleife in Teile aufgeteilt, die in einem einzigen Thread konkurrierend ausgeführt werden, wobei jede Iteration bereits von einer SIMD-Lane ausgeführt wird und die Anzahl dieser Iterationen der Vektorlänge entspricht.

Das Speichermodell in OpenMP ist so angelegt, dass jede konkurrierende Instanz über einen gemeinsamen und einen separaten (privaten) Speicher verfügt, um Konflikte zu vermeiden. Im Zusammenhang mit der Parallelisierung von Threads bedeutet dies, dass jeder Thread seine eigenen Variablen und den von allen Threads gemeinsam genutzten Speicher hat. Bei der Vektorisierung benutzen private Variablen jedoch separate Vektorregister, wodurch es möglich ist, für jeden Skalar (für jede konkurrierende Instanz) eine Kopie der Variablen zu haben. Die folgende Direktive wird verwendet, um private Variablen kennzuzeichnen:

```
private([variable], ...)
```

Kopien aller in der Klausel aufgeführten Variablen existieren nur innerhalb eines vektorisierbaren Codefragments. Schleifenvariablen und alle in einer Schleife initialisierten Variablen sind automatisch privat [7]. Sobald der Schleifenblock beendet ist, sind die Kopien der Variablen nicht mehr verfügbar. Wenn private Variablen am Ende einer vektorisierten Schleife erhalten bleiben müssen, verwenden man

```
lastprivate([variable], ...)
```

Die Direktive stellt sicher, dass die betreffende Variable den Wert der privaten Variable aus der letzten Schleifeniteration enthält.

In einem umstrittenen Fall, wenn eine Variable als privat deklariert ist, aber im Körper der vektorisierten Schleife geändert wird, ist der Wert der Variablen nach der Ausführung der Schleife derselbe wie der vor der Schleife zugewiesene Wert.

In einer Schleife ist es oft notwendig, ein Array von Werten auf ein einzelnes Element zu reduzieren, z. B. um die Summe der Arrayelemente oder ihr Produkt zu ermitteln. Dies wird durch horizontale Registervektoroperationen erreicht. Um solche Operationen zu bezeichnen, wird die folgende Klausel verwendet:

```
reduction([operation]: [variable], ...)
```

wobei `operation` die Operation ist, die durch Reduction angewendet wird, und `variable` die Variable ist, die das Ergebnis der Reduction aufnimmt. Eine Schleife, die diese Anweisung enthält, wird wie folgt kompiliert: Die zu aggregierenden Werte werden in einen Registervektor gestellt, dann wird eine horizontale Operation auf sie angewendet, deren Ergebnis in die Reduktionsvariable gestellt wird.

```
simdlen([value])
```

Die `simdlen`-Direktive akzeptiert als Parameter eine ganze Zahl, die die Anzahl der parallel laufenden Iterationen in der SIMD-Schleife angibt. Aber der Compiler hat die Möglichkeit, diesen Wert zu ignorieren, wenn seine Verwendung als ineffizient erachtet wird. Die Verwendung des richtigen Parameters kann zu einer Leistungssteigerung führen, weil der Compiler aggressivere Vektorisierungsstrategien anwenden kann, ohne befürchten zu müssen, dass Datenabhängigkeiten zerstört werden [2], aber ein unvorsichtiger Gebrauch dieser Funktion kann den Code verlangsamen.

Wenn Berechnungen im Schleifenkörper auf Werten basieren, die in früheren Iterationen der Schleife berechnet wurden, (loop-carried dependence) oder auf solchen Werten, dass die Schleifenvariante geändert werden muss, um auf sie zuzugreifen, kann der Compiler keinen Code erzeugen, wenn eine solche Situation nicht explizit angegeben ist. Das folgende Codefragment ist ein Beispiel für eine Datenabhängigkeit in Berechnungen:

```
for (i = 0; i < (N-4); i++) {
    a[i] = a[i+4] + b[i] * c[i];
}
```

Um den Compiler davor zu warnen, soll die Klausel

```
safelen([value])
```

angeben werden, wobei `value` die maximale Anzahl von Skalaren ist, die während einer Schleifeniteration im generierten Code parallel verarbeitet werden (mit anderen Worten, dies ist die Obergrenze der Vektorlänge). Der Standardwert für `value` ist die Anzahl der Schleifeniterationen. Es ist wichtig, den Unterschied zwischen den Direktiven `simdlen` und `safelen` zu verstehen: Die Erste dient nur als Empfehlung für die Wahl der Vektorlänge, während die Letzte explizite Grenzen setzt.

Bei verschachtelten Schleifen ist es manchmal effizienter, sie zu einer großen Schleife zu erweitern und deren Iterationen auf konkurrierende Instanzen zu verteilen. Dieser Ansatz reduziert den Effekt des Loop-Peelings, das die Menge der nichtsequentiellen Daten erhöht. Dazu existiert eine SIMD-Direktive

```
collapse([value])
```

wobei `value` die Anzahl der zu kollabierenden Schleifen angibt. Bei der Verwendung dieser Direktive ist es ratsam, das Ergebnis der Vektorisierung zu überprüfen, da sie die Geschwindigkeit des Codes negativ beeinträchtigen kanm. Der Grund dafür ist, dass einige verschachtelte Schleifen Ganzzahldivision und Division mit Rest verwenden, die relativ langsam sind. Wenn die Speicherzugriffe im Schleifenkörper nicht sequentiell sind, müssen Sie außerdem Gather und Scatter Operationen verwenden, was ebenfalls Zeit kostet.

Oft müssen Vektoren im Programmcode mit Werten gefüllt werden, die linear im RAM liegen. In diesem Fall kann ein Vektor nach diesem Muster konstruiert werden. Die Operationen Gather und Scatter dienen diesem Zweck. Die Erste setzt Skalare in den Vektor, die distant zueinander sind. Die Zweite „entpackt“ das Vektorregister zurück in den Arbeitsspeicher und sorgt für einen gewissen Abstand zwischen allen Skalaren. Einige Architekturen unterstützen diese Operationen beim Laden und Speichern von Daten. [7]

Bestenfalls gibt es keinen Abstand zwischen Skalaren, und dann wird ein ganzes Speicherfragment in das Vektorregister geladen. Dasselbe gilt für das Speichern von Vektorregistern in den RAM. Im ungünstigsten Fall befinden sich Skalare nicht linear im Speicher, und um sie zu laden/speichern, muss jeder Skalar einzeln bedient werden, was die Programmlaufzeit erhöht. Zur Veranschaulichung der linearen Abhängigkeit bei der Anordnung von Skalaren im Speicher wird die folgende Direktive verwendet:

```
linear([variable]: [step], ...)
```

Der Parameter `variable` kann eine Ganzzahlvariable oder ein Zeiger sein. Der Wert von `step` gibt an, wie stark sich die Variable in jeder Iteration der Schleife ändert. Auf diese Weise wird Code erzeugt, der zwei Vektoren enthält: einen, der aus den Werten der Schleifenvariante in den ersten n Iterationen besteht, und den anderen, der aus Werten besteht, mit denen der erste Vektor addiert werden kann, um in den nächsten n Iterationen einen invarianten Vektor zu erzeugen, wobei n die Länge des Vektorregisters ist.

Hinweis: Die in der linearen Klausel aufgeführten Variablen sind privat. Die Verwendung dieser Direktive in Schleifen ist optional, da Compiler, anders

als bei Funktionen (siehe Abschnitt 3.3), das Verhalten einer Variablen im Schleifenkörper automatisch bestimmen können. [7]

Wie in Abschnitt 2.3 erwähnt, ist der Align für die Programmleistung sehr wichtig. Um dem Compiler mitzuteilen, wie ein Datenarray ausgerichtet ist, wird die Klausel

```
aligned([ptr]: [alignment], ...)
```

verwendet, wobei `ptr` ein Zeiger auf das Datenarray ist. `alignment` ist so eine positive ganze Zahl, dass bei jedem Programmstart die Adresse des `ptr` Datenarrays garantiert ohne Rest durch diese Zahl geteilt wird.

Diese Information ist für den Compiler nützlich, da die ausgerichteten Schreib- und Speicheranweisungen bei der Codegenerierung verwendet werden, was das Programm sicherlich beschleunigt. Im Idealfall sollten für x86-Architektur die Daten an den Werten in der Tabelle 1 ausgerichtet sein. [2]

Data Type	**Alignment**
char	1
short	2
int	4
long int	8
float	4
double	8
long double	16
XMM register	16
YMM register	32
ZMM register	64

Tabelle 1: Speicherausrichtung für jeden Datentyp in C und C++

3.2 Block-Level-Direktiven

Im OpenMP 4.5-Standard gibt es nur eine Direktive auf Blockebene - die Direktive, die in einer Schleifendirektive verschachtelt werden kann. Manchmal kann ein Teil einer Schleife nicht parallelisiert werden, da er die Berechnungsergebnisse beeinflussen kann. In diesem Fall soll die folgende Klausel verwendet werden:

```
#pragma omp simd ordered
```

Die Verwendung dieser Klausel garantiert, dass alle Operationen des Blocks skalar sind und sequentiell gemäß der ursprünglichen Iterationsreihenfolge ausgeführt werden. Garantiert wird dies dadurch, dass immer nur eine SIMD lane auf den geschützten Bereich zugreifen kann. Es ist jedoch wichtig zu wissen, dass die Verwendung der Klausel `ordered` den Code stark verlangsamen kann. Außerdem lässt die die Ausführung von `break`, `continue` und `return` innerhalb eines Blocks nicht zu.

3.3 Vektorisierende Funktionen

Die Vektorisierung von Schleifen bedeutet auch, dass jede Anweisung im Schleifenkörper vektorisiert wird, aber wenn diese Anweisung ein Funktionsaufruf ist, kann der Aufruf einer Funktion mit einem skalaren Argument für jedes Element des Vektorregisters zu langsam sein. Um Leistungsprobleme in solchen Situationen zu vermeiden, können in OpenMP Vektorregisterfunktionen aus skalaren Funktionen erzeugt werden. Zu diesem Zweck wird die `declare`-Direktive verwendet:

```
#pragma omp declare simd [Klause]
```

Wenn der Compiler auf eine solche Anweisung stößt, kompiliert er mehrere Kopien der Funktion - die erste Kopie ist die Originalfunktion, die mit Skalaren arbeitet und außerhalb des Vektorblocks verwendet wird. Die zweite Kopie der Funktion wird bereits mit Vektorregistern arbeiten, so dass die Funktion auf mehrere Werte parallel angewendet werden kann. Der dritte Typ der Funktion hat ein zusätzliches Argument - eine Bitmaske, die festlegt, auf welche Zellen des Vektorregisters die Funktion angewendet wird und auf welche nicht. Dies ist notwendig, damit die Bedingungen im Körper der vektorisierten Schleife korrekt funktionieren. Wenn die Funktion jedoch Seiteneffekte hat, kann dies zu undefiniertem Verhalten führen. Außerdem darf eine vektorisierte Funktion keine Ausnahmen ausführen.

Die Klausel

```
simdlen([value])
```

legt die maximale Länge eines Vektorregisters fest und verhindert, dass mehr als `value` Skalarwerte auf einmal verarbeitet werden. Wie bei den Schleifen ist dies notwendig, um Fehler bei der Vektorisierung abhängiger Berechnungen (loop-carried dependence) zu vermeiden.

Wenn sich ein Argument einer Vektorfunktion für jedes Element des Vektorregisters linear ändert, sollte dem Compiler das Vorhandensein einer solchen Variablen auch durch die Klausel

```
linear([variable]: [linearstep], ...)
```

mitgeteilt werden, wobei `argument` der Name der Variablen und `linearstep` der Schritt ist, mit dem

sich die Variable für den nächsten Vektorregisterwert ändert. Die Klause `linear` kann auch einen von zwei qualifizierenden Modifikatoren enthalten. Die Syntax sieht folgendermaßen aus:

```
linear([modifier]([variable]): [step])
```

Der Wert `modifier` akzeptiert folgende Modifikatoren:

- `ref` - wird verwendet, wenn der Zeiger in Klammern linear ist.
- `uval` - wird verwendet, wenn die Adresse der Variablen bei jedem konkurrierenden Aufruf dieselbe ist und der Wert der Variablen linear ist.

Wenn die Funktion ein Argument enthält, das für alle konkurrierenden Aufrufe unveränderlich ist, sollte der Compiler auf dieses Argument aufmerksam gemacht werden, damit er einen effizienteren Code erzeugen kann. Dies geschieht durch die Verwendung der Klausel

```
uniform([variable], ...)
```

die als `variable` den Namen des unveränderlichen Arguments nimmt. Die kombinierte Verwendung von Direktiven `linear` und uniform ermöglicht es dem Compiler, schnelle unit-stride Anweisungen anstelle von Gahter und Scatter zu verwenden [2]. Dies ist möglich, weil der Compiler weiß, wie die konkurrierenden Speicheranforderungen aussehen werden.

Wie bereits erwähnt, erstellt der Compiler drei Kopien einer Funktion - eine originale oder skalare, eine zweite vektorisierte und eine dritte vektorisierte, aber mit einer Maske als Argument und unter Beteiligung an Schleifenbedingungen. Oft ist eine dritte Kopie unnötig, weil die Funktion nur außerhalb des Bedingungskörpers aufgerufen wird. Um die Größe der ausführbaren Datei zu verringern, kann dem Compiler mitgeteilt werden, dass eine dritte Kopie nicht erforderlich ist. Dies kann mit der notinbranch Klausel geschehen.

```
notinbrach
```

Wenn die Ausrichtung von Zeigern auf Datenfelder bekannt ist, die als Parameter an eine vektorisierte Funktion übergeben werden, können Sie diese in der speziellen Klausel

```
aligned([ptr]: [alignment])
```

angeben, wobei `ptr` ein Zeiger auf die Daten ist und `alignment` immer eine positive Anzahl von Bytes ist. Die Bedeutung von abgeglichenen Daten wurde in Abschnitt 2.3 erörtert. Wenn aber Programmierende nicht sicher sind, ob die Daten immer genau so ausgerichtet sind, wie in der `aligned`-Klausel angegeben, ist es am besten, sie nicht zu verwenden, da dies zu Fragmentierungsfehlern führen kann.

3.4 Herstellerspezifische Direktiven

Compiler erzeugen den Code so, dass die resultierende Binärdatei auf verschiedenen Prozessoren mit derselben Architektur ausgeführt werden kann, weshalb viele prozessorspezifische Optimierungen nicht anwendbar sind. Das Flag `-m` für GNU GCC und `-x` für den Intell-Compiler helfen, dieses Problem zu lösen, aber selbst das führt nicht zu SIMD-spezifischen Optimierungen, da der kleinste 128-Bit-XMM-Vektor oft standardmäßig verwendet wird. Um dem Compiler mitzuteilen, welche Registergröße er verwenden soll, gibt es in OpenMP die Klausel

```
processor(string)
```

wobei `string` die so genannte Prozessor-ID ist. Mögliche Prozessor-IDs und ihre zugehörigen Erweiterungen des Befehlssatzes sind in Tabelle 2 aufgeführt [2]. In dieser Tabelle sind auch die entsprechenden Register für jede Erweiterung vorgestellt. Der Unterschied besteht darin, dass jedes Vektorregister eine andere Größe hat: `ZMM` - 512 Bit, `YMM` - 256 Bit und `XMM` - 128 Bit.

Processor ID	**ISA**	**Registers**
pentium_4	SSE2	XMM
pentium_4_sse3	SSE3	XMM
core_2_duo_ssse3	SSSE3	XMM
core_2_duo_sse4_1	SSE4.1	XMM
core_i7_sse4_2	SSE4.2	XMM
core_2nd_gen_avx	AVX	YMM
core_3rd_gen_avx	AVX	YMM
core_4th_gen_avx	AVX2	YMM
mic	KNC	ZMM
mic_avx512	AVX512	ZMM

Tabelle 2: Gültige Prozessor-IDs

3.5 Vektorisierung und Parallelisierung

Vor der vierten Version von OpenMP, als dem Standard eine Erweiterung für die Vektorisierung von Code fehlte, verwendeten die Programmierer umgebungsspezifische Vektorisierungsmethoden, die stark

an die Umgebung gebunden waren. Natürlich war dieser Code schlecht portierbar und enthielt oft Seiteneffekte. Heutzutage wird die Vektorisierung und die Parallelisierung in OpenMP leicht kombiniert. Deshalb steigern sie spürbar die Leistung des Codes. Zu diesem Zweck wird die folgende Syntax verwendet:

```
#pragma omp for simd [Klausel]
```

Wenn dieses Konstrukt verwendet wird, werden die durchzuführenden Berechnungen zuerst zwischen Threads und dann zwischen Vektorregistern geteilt. Deshalb ist es wichtig, daran zu denken, dass die Anzahl der Threads die Codeleistung beeinflussen kann: Wenn die Schleife von einer großen Anzahl von Threads gemeinsam genutzt wird, könnte jeder Thread ein Speicherfragment erhalten, das für die Vektorisierung zu klein ist, was die Verwendung von Vektorregistern nicht kompensiert und darüber hinaus die Leistung verringert. Eine einfache Lösung dieses Problems kann darin bestehen, eine Chunk-Größe bei der Aufteilung der Arbeit zwischen den Threads zu verwenden, die ein Vielfaches der Vektorlänge in Bytes ist. Das `simd:static`-Argument der Schedule-Klausel, das die Chunk-Größe anhand der Formel

ceil(`chunk_size` / `simd_len`) * `simd_len`

berechnet, bietet eine solche Lösung.

Die private-Direktive macht Variablen für jede einzelne SIMD-Lane privat.

3.6 OpenMP-Unterstützung durch Compiler

Um die Vorteile von OpenMP nutzen zu können, muss dem Compiler ausdrücklich mitgeteilt werden, dass der zu kompilierende Code OpenMP-Direktiven enthält. Dies kann mit den Flags `-fopenmp` und `-openmp` für GCC- bzw. Intel-Compiler erreicht werden. Wenn der Quellcode auch SIMD-Direktiven enthält, müssen Sie die Kompilierungsflags `-fopenmp-simd` und `-openmp-simd` hinzufügen.

Die meisten modernen Compiler sind in der Lage, Fehler, Warnungen und Hinweise im Zusammenhang mit der Vektorisierung in OpenMP auszugeben. In GCC beispielsweise warnt der Compiler bei aktiviertem Flag `-Wopenmp-simd`, wenn er eine benutzerdefinierte Optimierung für nachteilig hält. Der Intel-Compiler verwendet ein Standardmittel zur Aufnahme von Meldungen in die Compilerausgabe - die Flags `-diag-enable`, `-diag-disable`, `-diag-warning`, `-diag-error` und `-diag-remark`. Um Nachrichten auf der richtigen Ebene auszugeben, muss das Schlüsselwort `openmp` nach dem Flag angegeben werden, z.B.: `-diag-enable=openmp`.

4 Benchmark & Evaluation

Um die Effizienz der Vektorisierung in OpenMP zu demonstrieren, habe ich zwei C++ Programme aus dem öffentlichen Quellen optimiert und mit dem GNU GCC Version 11.1.0 Compiler mit verschiedenen Kombinationen von Flags getestet, um die Wirksamkeit der Parallelisierung zusammen mit der Vektorisierung und der Vektorisierung für verschiedene Befehlssystemerweiterungen zu bewerten. Jeder Test entspricht der folgenden Reihe von Kennzeichnungen:

native&parallel - Diese Tests enthielten nicht nur die Direktiven `omp simd`, sondern auch `omp parallel for` und deren Kombination `omp parallel for simd`. Beim Testen wurden solche Flaggen verwendet: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -march=native -mfma -O3`

native - Bei diesen Tests gab es keinerlei Thread-Parallelisierung des Codes und es wurden nur die Vektorisierungsfähigkeiten von OpenMP genutzt. Kompilationsfahnen: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -march=native -mfma -O3`

avx512f - Um den grundlegenden Unterschied bei der Codebeschleunigung für Prozessoren mit verschiedenen erweiterten Befehlssätzen herauszufinden, wird der Code mit 256-Bit-Vektorregistern kompiliert. Kompilationsfahnen: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -mmavx512f -mfma -O3`

avx2 - In diesem Test wird der Code mit 256-Bit-Vektorregistern kompiliert. Kompilationsfahnen: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -mavx2 -mfma -O3`

sse4.2 - Diese Tests verwenden den erweiterten sse4.2-Befehlssatz, der mit Vektorregistern von 128 Byte Länge arbeitet. Kompilationsfahnen: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -msse4.2 -mfma -O3`

mmx - Solche Tests werden mit 64-Bit-Vektorregistern - den kleinsten Vektorregistern - kompiliert. Kompilationsfahnen: `-fopenmp -fopenmp-simd -ftree-vectorize -ffast-math -lmvec -mmmx -mfma -O3`

autovec - Tests dieser Art sind notwendig, um festzustellen, wie gut Compiler heute bei der automatischen Vektorisierung von Code sind. Kompilationsfahnen: `-ftree-vectorize -ffast-math -lmvec -mfma -O3`

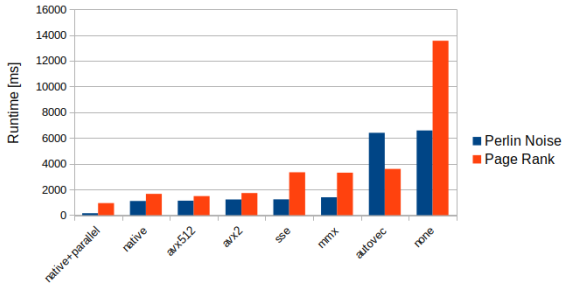

Abbildung 2: Absolute Testlaufzeit

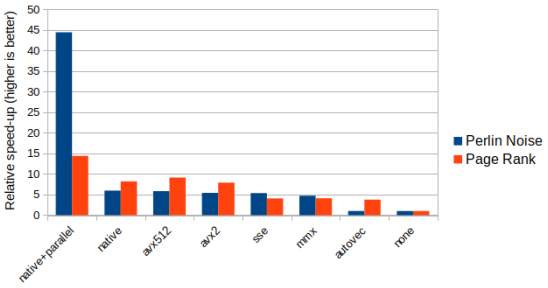

Abbildung 3: Relative Testlaufzeiten

none - Kontrolltests, die keine Optimierungen im Zusammenhang mit Vektorisierung oder Parallelisierung durchführen. Kompilationsfahne: `-O2`

Der Testrechner läuft unter dem Betriebssystem Ubuntu 20.04 (Linux-Kernel: 5.10.0-1052-oem) auf einem Intel® Core® i5-11400H Prozessor (getaktet mit 4,5 GHz, Unterstützung des erweiterten Befehlssatzes AVX-512). Jedes Programm wurde 30-mal gemessen, um die durchschnittliche Ausführungszeit genauer abschätzen zu können. Bei allen Tests wird der Code mit denselben Eingabedaten ausgeführt. Dem Repository mit den zu testenden Programmen und dem Benchmarking-Skript wird folgender Digitaler Objektbezeichner (DOI) zugewiesen: `10.5281/zenodo.5854407`

4.1 Perlin-Noise

Dieser Benchmark erzeugt eine Pseudo-Zufallszahlenmatrix mit 3000x3000 Elementen unter Verwendung des Perlin Classical Noise Algorithmus, der in der Praxis häufig im Spielmodus verwendet wird, um plausible Zufallsstrukturen wie Wolken, Nebel, Landschaften usw. zu erzeugen.

Die Testergebnisse (Abbildungen 2 und 3) zeigen, dass durch die Kombination von Threaded-Parallel- und Vektorprogrammierung eine Leistungssteigerung um das 44-fache erreicht werden konnte. Wenn wir die Parallelisierung durch Threads ablehnen, beträgt die Leistungssteigerung „nur“ das sechsfache. Dieses Ergebnis ist darauf zurückzuführen, dass Perlins Lärmberechnungen im zweidimensionalen Raum für jeden Punkt unabhängig durchgeführt werden können, was bedeutet, dass die Leistung umso höher ist, je mehr konkurrierende Instanzen an den Berechnungen beteiligt sind.

Interessant ist auch, dass der Compiler die Autovektorisierung des Codes eher schlecht handhabte - eine 1,02-fache Leistungssteigerung gegenüber der nicht vektorisierten Version kann als Messfehler angesehen werden. Dies bezieht sich auf die These, dass Compiler ihre Arbeit nicht überall gut machen.

4.2 PageRank

Der Pagerank-Algorithmus hat sich seit langem als ein guter Algorithmus für das Ranking von Websites etabliert. Für den Test wird ein Graph verwendet, das ein Netz von 600 Websites emuliert.

Diesmal sind die Testergebnisse nicht ganz so beeindruckend, aber die kombinierte Verwendung von Threading und Vektorparallelität hat dazu beigetragen, den Code im Vergleich zur nicht optimierten Version um den Faktor 14,4 zu beschleunigen. Die Verwendung verschiedener Befehlssatzerweiterungen führte erwartungsgemäß auch zu unterschiedlichen Leistungswerten - je größer die Länge der Vektorregister und deren Anzahl, desto größer der Leistungsgewinn.

Dieses Beispiel zeigt, im Gegensatz zum vorherigen, die Effizienz der Autovektorisierung. Er verliert nicht mehr als 2 Mal gegenüber der manuellen Vektorisierung mit OpenMP.

5 Fazit

In diesem Beitrag wurde eine Möglichkeit zur Vektorisierung von Code durch OpenMP-Funktionen vorgestellt. Dieser Ansatz ist leichter zu erlernen als Low-Level-Programmierung.

Es garantiert auch eine gute Portabilität des Quellcodes, was für moderne Aufgaben entscheidend ist. Wie die Tests gezeigt haben, ist die von Compilern unterstützte Autovektorisierung nicht immer in der Lage, die Aufgabe zu bewältigen und alle Abhängigkeiten im Quellcode zu finden. Programmierende hingegen haben ein umfassenderes Verständnis des Problems und können daher dem Compiler korrekt mitteilen, wie er den Code am besten erzeugt.

Es ist wichtig zu bedenken, dass die Vektorisierung kein Allheilmittel ist, auch wenn es so scheint. Das Schreiben von Code, der leicht vektorisiert werden kann, erfordert oft zusätzliche Entwicklungszeit, einen kreativen Ansatz und ein Verständnis für SIMD-Parallelität. Ein weiterer Nachteil kann die

Verschlechterung der Codequalität im Hinblick auf die Lesbarkeit sein, was die Teamarbeit erschwert.

Literatur

[1] Michael J Flynn. „Some computer organizations and their effectiveness“. In: *IEEE transactions on computers* 100.9 (1972), S. 948–960.

[2] Joseph N Huber, Oscar R Hernandez und Matthew Graham Lopez. „Effective vectorization with OpenMP 4.5“. In: *ORNL/TM-2016/391. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF)* (2017).

[3] Henry Gordon Rice. „Classes of recursively enumerable sets and their decision problems“. In: *Transactions of the American Mathematical Society* 74.2 (1953), S. 358–366.

[4] *posix_memalign(3) — Linux manual page*. 2021. URL: `https://man7.org/linux/man-pages/man3/posix_memalign.3.html` (besucht am 30. 01. 2022).

[5] *C++ documentation for std::aligned_alloc*. 2018. URL: `https://en.cppreference.com/w/cpp/memory/c/aligned_alloc` (besucht am 30. 01. 2022).

[6] Michael Klemm u. a. „Extending OpenMP* with vector constructs for modern multicore SIMD architectures“. In: *International Workshop on OpenMP*. Springer. 2012, S. 59–72.

[7] Ruud Van der Pas, Eric Stotzer und Christian Terboven. *Using OpenMP# The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT press, 2017.