Studie zu Cacheblocking als Tilingmethode zur effizienten Bandbreitenausnutzung

Andreas Baier
Universität Passau
Email: baiera@stud.uni-passau.de

***Zusammenfassung*—Diese Studie evaluiert Cache Blocking als Verfahren zur Bandbreitenoptimierung. An einem einfachen Beispiel wird erkundet, welche Rolle diese klassische Codeoptimierung im Vergleich zu den bewährten Optimierungsstufen aktueller Compiler spielt und ob überhaupt ein Performanzgewinn erzielt kann. Abschließend wird in einem Fallbeispiel ein konkreter Anwendungsfall geprüft.**

I. Einleitung

Seit einem knappen Jahrzehnt bewegt sich die Peak-Performanz, also die maximale Rechenleistung, und die maximale Speicherbandbreite von Rechnerarchitekturen kontinuierlich auseinander. Verantwortlich hierfür ist zum einen die weite Verbreitung von Many-Core Prozessoren, aber auch die Komplexität der einzelnen Kerne hat durch die Weiterentwicklung der verwendeten arithmetischen Recheneinheiten, wie *Fused Multply-Add*, die Einführung *logischer Prozessoren*, z.B. beim *Intels Hyper Threading Technologie* sowie die Fortentwicklung der Rechen Pipelines zugenommen, um nur einige Beispiele zu nennen. Die Speicherbandbreite konnte aber nicht im gleichem Maß erhöht werden. Die so entstandene *DRAM-Gap*, also der Latenz- und Bandbreitenunterschied zwischen schnellen Prozessor Caches und dem Hauptspeicher, sorgt bei den gestiegenen Anforderungen an die zu verarbeitenden Datenmengen im Zeitalter des *Big Data* für den größten Flaschenhals. Hier sind zudem wenige Neuerungen zu erwarten. Auf Seiten der CPUs nehmen Caches prozentual betrachtet bereits den größten Teil der Chipfläche ein. Zugriffsoptimierende Technologien, wie *Out-of-Order Execution* und *Prefetching* können nur teilweise die abreißenden Datenströme und hohen Latenzen zu Speichernstrukturen niedrigerer Ordnung kompensieren, so dass Prozessoren sich häufig im *IDLE* Zustand befinden (*memory bound*).

Verfahren, die den CPU Cache effizient nutzen, sind somit von höchstem Interesse, da durch eine Wiederbenutzung von Daten im Cache (*data reuse*) sowie durch intelligente Datentransfers (*Datenlokalität*), die das Laden unnötiger Daten vermeiden, wertvolle Bandbreite gespart werden kann.

In den letzten Jahren gab es einen Sprung in der Compilertechnik mitunter hervorgerufen durch die offene Architektur des Compilerframeworks *LLVM*. Neben den klassischen Optimierungen, die häufig die Peak-Performanz durch Reduzierung ausgeführter Berechnungen sowie Registerallokation im Fokus haben, reihen sich komplexe datendurchsatzerhöhende Optimerungen wie Autovektorisierung und automatisierte Schleifenparallelisierung ein.

Eine recht alte aber im *High Performance Computing* Bereich nach wie vor populäre low level Technik für Cache Optimierung ist das sogenannte *Cache Blocking*, bei dem Datenzugriffe in Blöcken arrangiert werden, die die Größe der CPU Caches der verwendetetn Plattform berücksichtigen. [1] [2]

Diese Studie evaluiert Cache Blocking als Bandbreitenoptimierung. Es soll an einem einfachen Beispiel erkundet werden, ob diese klassische Codeoptimierung gegen die automatisierten Optimierungen aktueller Compiler überhaupt einen Performanzgewinn erzielen kann. Abschließend soll eine kurze Fallstudie klären, ob es reale Anwendungsfälle gibt.

ab
24. September 2014

A. Motivation

Cache Blocking ist prinzipbedingt stark abhängig vom Datenzugriffsmuster und den aktuellen Cachegrößen der Ausführungsplattform. Es ist somit höchst plattformabhängig. Die hohe Anzahl von Prozessorarchitekturen macht daher eine generelle Lösung schwierig, da viele Prozessoren unterschiedliche Cache Parameter besitzen und angebotene Programmbinaries häufig viele Plattformen bedienen müssen. Aus diesem Grund ist Cache Blocking momentan hauptsächlich im HPC Bereich verhaftet, da hier eine Codeoptimierung häufig manuell für hochspezialisierte Plattformen statt findet.

Die Peak-Performanz aktueller Rechner macht es allerdings möglich auch umfangreiche Programme erst zur Laufzeit in Maschinencode zu übersetzen (*Just-in-Time Compiler, JIT*), so dass die spezifischen Plattformparameter und somit die Größe des vorhanden Caches während der Kompilation zur Verfügung stehen. Außerdem stehen durch Fortentwicklungen der Compilertechnik mächtige Frameworks zur Verfügung, die die einfache Entwicklung von JIT Compiler ermöglichen.

Diese Studie dient als vorausgehende Evaluation für eine mögliche Masterarbeit, die sich mit Cacheoptimierungen zur Laufzeit mittels JIT-Technologien auseinandersetzen würde. Im Fokus würde das Cache Blocking-Verfahren stehen.

B. Ziele

Gegenstand dieser Evaluation sind somit folgende Fragen und Ziele:

- Kann durch Cache Blocking überhaupt ein signifikanter Performanzgewinn gegenüber unoptimierten Code und

den üblichen Optimierungsstufen (z.B. Compileroption -O3) aktueller Compiler erzielt werden? Wird automatisiertes Cache Blocking bei architekturspezifischer Kompilation gar bereits umgesetzt?
- Wie verhalten sich unterschiedliche Blockgrößen kleiner als die Cachegröße auf die Performanz? Ist eine optimale Blockgröße deutlich auszumachen?
- Gibt es reale Anwendungsszenarien?
- Bei genug Indizien für einen möglichen Performanzgewinn, soll außerdem die Entwicklung einer Testinfrastruktur angestoßen werden, sodass später das Deployment von Benchmarks auf anderen Architekturen erleichtert wird.

Natürlich dient diese Studie auch zur Entwicklung von Ideen für eine *automatisierte* Cacheoptimierung mittels Cache Blocking. Abschließend werden Möglichkeiten diskutiert, wie eine Umsetzung aussehen könnte.

II. Hintergrund

Bisher wurde Programmcode häufig nach seiner arithmetischen Intensität beurteilt. Diese ergibt sich aus dem Verhältnis von Berechnung (meist angegeben in FLOPS) zu übertragenem Byte (siehe Abb. 1).

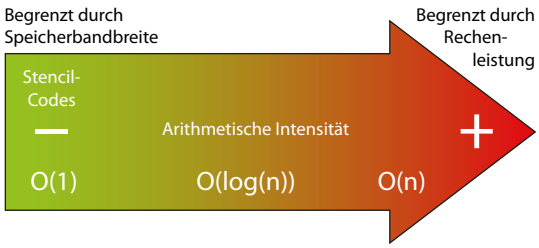

Abbildung 1. Arithmetische Intensität als Verhältnis von Berechnung und übertragenen Daten.

Durch die immer höhere Peak-Performance bei gleichzeitig hohen Datenmengen gerät die Berechnung häufig in den Hintergrund und das Programm wird speichergebunden (*memory bound*).

A. Speicherhierarchie

Der Speicher ist bei modernen Architekturen hierarchisch aufgebaut. An der Spitze stehen die Prozessorregister, dann kommen kleine aber schnelle Prozessorcaches, die im günstigem Fall den Zugriff in wenigen Taktzyklen erlauben (wenige Nanosekunden). Caches sind meist in mehrere in Größe und Latenz aufsteigende Level unterteilt. Aktuelle Prozessoren besitzen typischerweise zwei bis drei Level von wenigen Kilobyte bis Megabyte Größe. Die Prozessoren der *Intel Core-i* Reihe besitzen z.B. 64 KB Level 1 Cache für Daten und Instruktionen und bis zu 8 MB Daten Cache im Level 3. Die nächste Stufe der Speicherhierarchie markiert der Haupt- oder Arbeitsspeicher mit mehreren Gigabyte Größe, dann kommt der Sekundärspeicher in Form von Festplatten, Solid State Disks und optischen Datenträgern. Der Zugriff auf den Hauptspeicher benötigt in der Regel mehrere hundert Taktzyklen, wodurch sich die DRAM-Gap ergibt. Der Zugriff auf die Sekundärspeicher beginnt im Millisekundenbereich.

Werden die für eine Berechnung benötigten Daten im Prozessor Cache vorgefunden, so spricht man von einem *Cache Hit*, im anderem Fall von einem *Cache Miss*. Bei letzterem müssen die Daten aus dem Hauptspeicher geholt werden. Der Zugriff auf den Hauptspeicher erfolgt dabei nicht elementweise. Caches sind in sogenannte *Cache Lines* organisiert, die typischerweise eine Länge von 16 bis 64 Byte haben. Beim Laden von Daten aus dem Speicher werden aufeinanderfolgende Elemente der Länge einer Cache Line in den CPU Cache geladen.

Wenn die für eine Berechnung benötigten Daten so organisiert sind, das sie konsekutiv im Speicher liegen, so spricht man von *spatialer Lokalität*, d.h. mit einer Cache Line können möglichst viele benötigte Daten auf einmal geladen werden.

Bei durchsatzorientierten Programmen, kann man zwei Klassen unterscheiden, die sich an Hand der Wiederbenutzung von bereits im Prozessorcache geladenen Werten unterscheiden. Bei Daten Streams werden große Datenmengen transferiert, bei denen eine geringe Wiederbenutzung während der Verarbeitung vorliegt. Bei diesem Typ ist es wichtig, dass aufeinanderfolgende Daten konsekutiv im Speicher vorliegen (spatiale Lokalität). Die zweite Klasse von Programmen besitzt regelmäßige Zugriffsmuster, die die Wiederbenutzung (*reuse*) bereits geladener Werte erlaubt. Hier spielt zusätzlich die zeitliche Dimension eine Rolle und man spricht von *temporaraler Lokalität*.

Es ist das Ziel von Cache Blocking die temporale und spatiale Lokalität zu erhöhen.

B. Tiling und Cacheblocking

Cache Blocking ist eine Form des *Loop Tilings*. Beim Tiling werden (mehrdimensionale) Schleifen in kleinere meist rechteckige Blöcke, sog. *Tiles* (*Kacheln*), aufgeteilt.

Das folgende Beispiel demonstriert dieses Verfahren: Das erste Listing enthält eine einfache elementweise Zuweisung der Elemente vom zweidimensionalen Feld `B` an die Zellen des Feldes von `A`. Der Iterationsraum wird durch die Laufindices `i` und `j` aufgespannt.

```
for i := 0 to n
  for j := 0 to n
    A[i][j] := B[i][j]
```

Mittels Tiling wird dieser Iterationsraum nun in zweidimensionale quadratische Blöcke der Größe `block` aufgeteilt. Die inneren Schleife über `i` und `j` läuft nun nicht mehr über die Dimension `n` auf einmal sondern in Schritten der Länge `block`.

```
for ii := 0 to n by block
  for jj := 0 to n by block
    for i := ii to min(ii+block-1, n)
      for j := jj to min(jj+block-1, n)
        A[i][j] := B[i][j]
```

Für jede Blockdimension wird folglich eine weitere Schleife eingeführt. Die nötigen Compilermethoden heißen *strip-mining* und *loop interchange*, auf die hier aber nicht näher

eingegangen wird. [1] [3] Sie sind z.B. in der LLVM Erweiterung *Polly* implementiert.

Beim Cache Blocking wird die Blockgröße an den vorhandenen Cache angepasst, mit dem Ziel, dass die im Schleifenrumpf verarbeiteten Daten die Cachegröße nicht überschreiten.

III. Aufbau des Experiments

Für das Experiment wurde ein einfaches Testprogramm geschrieben, dass einen zweidimensionalen Feldzugriff in einer ungeblockten und einer äquivalenten geblockten Variante enthält. Es war möglich die Feldgrößen und die Blockgrößen zu variieren.

A. Variablen und Operationalisierung

Das Experiment unterteilt sich in ein univariates 1-faktorielles Experiment zur Bestimmung geeigneter Blockgrößen und in ein univariates 4-faktorielles Experiment zur Performanzmessung.

Beim ersten Suchlauf zur Bestimmung der Blockgröße diente als unabhängige Variable die Blockgröße. Diese wurde zwischen 2 und 1600 in 4er-Schritten erhöht. Die Zielvariable war die ermittelte Performanz (s.u. für die verwendete Metrik).

Das Hauptexperiment besaß als unabhängige Variablen die Testparameter Programmcode, Feldbreite, Compilerversion und ausgeführte Compiler Optimierung. Die abhängige Variable war ebenfalls die erzielte Performanz.

Die Variable Programmcode ist zweistufig: Für das Experiment wurde ein einfaches Testprogramm geschrieben, dass einen zweidimensionalen Feldzugriff in einer ungeblockten und einer äquivalenten geblockten Variante enthält.

Die Felddimension wurde im Intervall 2^2-$2^16 = 65536$ in 2er Potenzen variiert, sie ist also 15-stufig.

Die Variable Compiler ist zweistufig. Das Testprogramm wurde mit dem *GNU Compiler gcc* (Version 4.2.8) sowie mit dem *LLVM Compiler Clang* (Version 3.4) kompiliert.

Im ersten Durchlauf wurde das Programm ohne Optimierungen kompiliert (Compilerschalter `-O0`). Für den zweiten Durchlauf wurden die Compileroptimierungen aus *Optimierungsstufe* `-O3` in Kombination mit den konkreten Plattformparametern (`-march=native`) verwendet. Unter gcc wurde die *Vektorisierung* und die *SSE*-Optimierungen eingeschaltet (`-ftree-vectorize -msse2`). Clang benutzt seinen *Autovektorisierer* in der installierten Version im Optimierungslevel `O3` automatisch.

Die Zielvariable ist die erzielte Performanz. Hierfür wurde die *C-Header Datei cycle.h* vom *MIT* verwendet, die auch im HPC Verwendung findet. Diese benutzt nach Möglichkeit Hardware Zähler um die Zuverlässigkeit der Messungen zu erhöhen.

Als Metrik wurde die Anzahl der Prozessorticks pro Schleifenrumpf verwendet (siehe Kap. III-B). Zwar wurde parallel auch die Laufzeit gemessen, jedoch erwies sich diese als ungenauer Indikator für die genutzte Speicherbandbreite.

B. Störfaktoren

Bei Performancemessungen dieser Art gibt es zahlreiche systembedingte Störfaktoren, die die Laufzeit des Testprogramms beeinflussen. Folgende Maßnahmen wurden daher getroffen:

Um spätere Vergleichstests zu vereinheitlichen wurde ein Linux Mint 17 (Qiana) in der Standardinstallation installiert. Es basiert auf Ubuntu 14.04 LTS, verwendet aber nicht den Unity-Desktop, der in ersten Tests einen hohen Overhead verursachte.

Alle Performancetests wurden ohne Grafische Oberfläche auf der Konsole ausgeführt, um Interrupts anderer Anwendungen zu minimieren. Alle im Hintergrund laufenden Netzwerkdienste wurden abgeschaltet.

Um Nebeneffekte des Frequenzscalings zu vermeiden und die Präzision der gemessenen Ticks zu erhöhen, wurde die Frequenz auf 2,292 GZ festgesetzt und der *Intel-Turbo-Mode*, der ein kurzzeitiges Overclocken einzelner Kerne erlaubt, abgeschaltet. Darüber hinaus wurde *Intels Hyperthreading* abgeschaltet, um zu verhindern, dass das Testprogramm sich mit einem anderen Programm einen Kern teilt. Vor allen Tests wurden die Caches geflusht.

Um Interaktionseffekte zwischen den einzelnen Testläufen zu vermeiden, wurden die verwendeten Felder vor jedem Durchlauf neu erstellt und initialisiert und nach dem Durchlauf freigegeben.

C. Material

Das Testprogramm besteht aus einer quadratischen zweidimensionalen Schleife der variablen Größe $n \times n$. Im Schleifenrumpf ist ein Lese- und ein Schreibzugriff enthalten. Die arithmetische Intensität ist vernachlässigbar, so dass dieses Programmstück für große n speichergebunden ist.

```
...
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    B[j][i] = A[i][j] + 1;
...
```

Die Besonderheit des obigen Zugriffsmusters besteht im Schreibzugriff auf die Zellen von `B`, da hier mit dem inneren Laufindex (`j`) über die erste Dimension iteriert wird. Stellt man sich die Felder `A` und `B` als Quadrate vor, so hieße dies, dass in jeder neuen Iteration der Schleifen bei `A` die Zelle in der nächsten Spalte der aktuelle Zeile gelesen wird. Bei `B` wird hingegen die Zelle der nächsten Reihe der aktuellen Spalte gelesen. Der Feldzugriff wandert somit für `A` horizontal durch das Feld, für `B` vertikal.

Da Arrays in der Sprache C entlang der innersten Feldindices (horizontal) im Speicher abgelegt werden, heißt dies, dass beim Laden von Daten aus `A` in den CPU Cache sich in der Cache Line bereits zukünftig benötigte Werte befinden. Dies trifft bei den Zugriffen auf B nicht zu. Hier befinden sich Daten der nächsten Spalten in einer Cache Line, die aber erst zu einem viel späteren Zeitpunkt geschrieben werden.

Hinzu kommt das Schreibzugriffe in der Regel aufwändiger sind als Lesezugriffe. Sollte sich eine zu schreibende Speicherstelle nicht bereits im Cache befinden, so wird diese auf

vielen Systemen erst aus dem Speicher in den Cache geladen, um erst dann entsprechend modifiziert zu werden (Stichwort *write allocate*).

Alles in allem ist so gesichert, dass der Programmcode stark speichergebunden ist.

Da es sich bei der Zuweisung um eine Transposition einer Matrix handelt, ist dieser Programmcode weit verbreitet.

Das nächste Listing zeigt den C Code für die geblockte Variante. Der Einfachheit halber sind die Blöcke quadratisch mit der Seitenlänge `block`. Der Code wurde analog nach dem Muster aus Kapitel II-B erstellt.

```
...
for (int ii = 0; ii < n; ii += block)
  for (int jj = 0; jj < n; jj += block)
    for (int i = ii; i < MIN(ii+block, n); ++i)
      for (int j = jj; j < MIN(jj+block, n); ++j)
        B[j][i] = A[i][j] + 1;
...
```

D. Testplattform

Als Testplattform wurde eine Lenovo Thinkpad Workstation W520 mit einem Intel Core i7-2820QM Prozessor mit 16 GB RAM verwendet. Der Prozessor läuft mit einer Taktfrequenz von 2.30GHz und besitzt 8192 KB L3 Cache. Weitere vorbereitete Testplattformen konnten aus Zeitgründen nicht mit in die Testergebnisse aufgenommen werden.

IV. Hypothesen

Die folgenden Annahmen dienten als Grundlage für die Erstellung von Hypothesen im Vorfeld dieses Experiments.

Cache Blocking setzt eine gewisse tiefere Kenntnis der Zugriffsmuster innerhalb des Quelltextes voraus, da häufig nicht einwandfrei ermittelt werden kann, wieviel Speicher innerhalb eines Schleifenrumpfs tatsächlich angefordert wird.

Da zudem Compiler im Allgemeinen benutzt werden um Programme für *Familien* von Architekturen zu erstellen (z.B. *i686*, *x86_64*) können nur wenige Annahmen über spezielle Eigenschaften der Prozessoren getroffen werden. Insbesondere sind die Cache Eigenschaften der Ausführungsplattform zum Compilationszeitpunkt meistens unbekannt, so dass Cache Optimierung bisher nicht im Fokus stand. Dies spiegelt aktuelle Entwicklungen im HPC wieder, wo sehr viel Aufwand im Vorfeld eines Programms getrieben wird, um aktuelle Plattformparameter gezielt in den Code einfließen zu lassen (siehe Auto-Tuning-Ansätze bei PATUS [4]).

Als weitere grobe Heuristik dient zudem eine Suche nach den Begriffen *strip mining* und *loop interchange* im Programmcode von LLVM und GCC, die nur zu wenigen Treffern führte. Ein Umsetzung dieser Quelltexttransformationen scheint bisher vorwiegend im Projekt *Polly* verwirklicht zu sein, das allerdings eine automatisierte Schleifenparallelisierung im Fokus hat.

Somit kämen wir zu folgenden Hypothesen (die Bezeichnung *aktuelle Compiler* wird im Folgenden synonym für die zwei bekanntesten offenen Repräsentanten GCC und Clang benutzt):

1. Hypothese: Aktuelle Compiler sind nicht in der Lage optimalen Code zur Bandbreitenmaximierung zu generieren.

2.a) Hypothese: Aktuelle Compiler sind auch bei Kenntnis von Cache-Parametern nicht in der Lage optimalen Code zur Bandbreitenmaximierung zu generieren.

2.b) Hypothese: Aktuelle Compiler generieren von sich aus kein Cache Blocking.

3. Hypothese Es ist möglich mit Cache Blocking einen signifikanten Performanzgewinn gegenüber automatisch optimiertem Programcode zu erzielen.

Als weitere Forschungsfrage soll in dieser Arbeit zudem die Praxisrelevanz behandelt werden: Wenn sich die Hypothesen bestätigen sollten, gibt es konkrete Anwendungsfälle in der Praxis?

V. Durchführung

Das Experiment lief in drei Phasen ab:

1) Zunächst wurde eine geeignete Blockgröße für Cache Blocking ermitteln bei fester (großer) Feldbreite von $n = 65536$ und unoptimiertem Programmcode.
2) Es wird ein Performanzvergleich von ungeblockter und geblockter Variante bei durch den Compiler unoptimiertem Code und variabler Feldbreite durchgeführt.
3) Es wird ein Performanzvergleich von ungeblockter und geblockter Variante bei durch den Compiler optimiertem Code und variabler Feldbreite durchgeführt.

Phase 2 und 3 wurde sowohl mit dem *GNU Compiler gcc* als auch mit dem *LLVM Compiler Clang* durchgeführt.

Jeder Test wurde dreimal ausgeführt. Als Ergebnis wurde das arithmetische Mittel der drei Testläufe ausgegeben.

VI. Auswertung

Abbildung 2 zeigt die Ergebnisse des Suchdurchlaufs für die Blockgrößen. Die höchste Performance konnte für Blöcke der Größe 8 ermittelt werden, wofür im Schnitt 24.37 Ticks/Schleifenrumpf benötigt wurden. Die maximal gemessene Anzahl Ticks pro Schleifenrumpf lag bei 61.65591 bei einer Blockgröße von 1600, jedoch war bereits bei einer Blockgröße 256 die Anzahl Ticks/pro Rumpf doppelt so groß wie das Minimum.

In Abbildung 3 sind die Ergebnisse der Performanztests für den GCC abgebildet. Abbildung 4 enthält die entsprechenden Ergebnisse für Clang. Die x-Achse enthält die getesteten Feldbreiten, auf der y-Achse ist die erzielte Performanz in Ticks / Schleifenrumpf abgebildet. Kleinere Zahlen bedeuten daher eine höher Performanz.

Es ist zu sehen, dass die ungeblockte und unoptimierte Ausführung eine obere Schranke für alle weiteren Tests darstellt. Für kleine n bis 512 Feldbreite verläuft dieser Testlauf auf ähnlich schnellem Niveau, wie die anderen Testfälle. Ab einer Feldbreite von 512 fällt die Performanz deutlich ab. Die geblockte Variante zeigt in der unoptimierten wie auch in der optimierten Fassung einen nahezu gleichen Verlauf. Die optimierte Variante ist dabei geringfügig schneller. Bei der ungeblockten aber optimierten Variante ist die Performanz zwischen $n = 8$ und 512 genauso schnell wie bei den

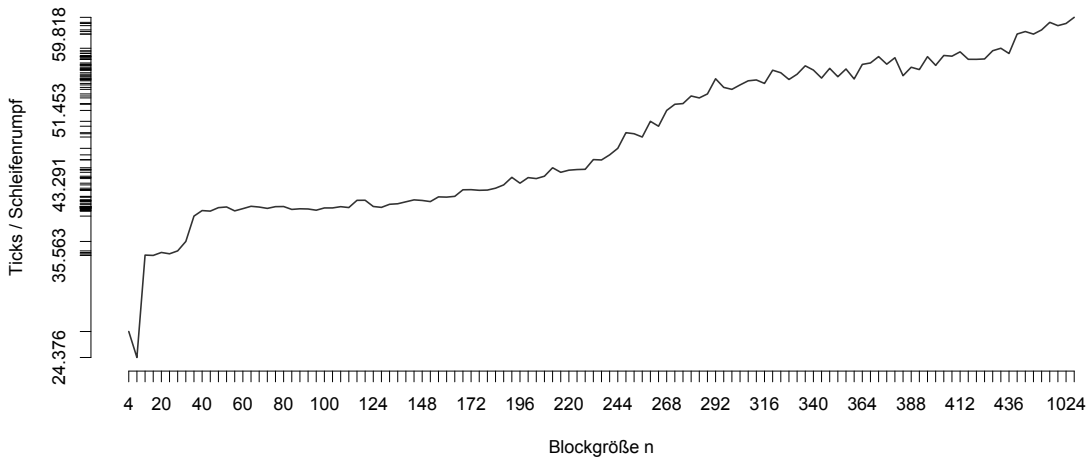

Abbildung 2. Blockgrößen und die entsprechende Performanceresultate in Ticks/Schleifenrumpf.

geblockten Varianten, ab $n = 512$ fällt sie aber auf die Kennlinie der unoptimierten Fassung zurück.

Die Kennlinien der Testergebnisse von Clang zeigen die gleichen Charakteristika wie die für GCC. Allerdings zeigte die geblockte und unoptimierte Fassung (rote Kennlinie) eine andere Anfangscharakteristik im Bereich von $n = 4$ bis 8. Sie besaß deutlich höhere Werte als alle anderen Kennlinien.

VII. Diskussion und Interpretation

Unter Berücksichtigung der Verläufe der Kennlinien lässt sich festhalten, dass durch ein gezieltes Blocking der Iterationsräume tatsächlich ein Performanzgewinn feststellen ließ. Dieser wurde ab Feldbreiten von mehr als 512 umso deutlicher sichtbar. Unter Clang konnte bei einer Feldgröße von 16384×16384 ein maximaler Speedup von 5.97 erreicht werden, bei gcc lag bei gleicher Feldbreite der maximale Speedup bei 9.64.

Eine Sichtung der Einzeldaten der Kennlinie von Clang im unoptimierten aber geblockten Testlauf zeigte, dass die Testläufe bei jeweils einem von drei Messwerten deutliche Abweichungen enthielten und somit das arithmetische Mittel nach oben rissen. Die anderen beiden Werte lagen knapp unter dem Niveau der grauen Kennlinie. Es kann daher davon ausgegangen werden, dass es sich hier um Ausreißer handelt, die auf eine systembedingte Störgröße zurückzuführen sind.

Somit lässt sich feststellen, dass die Hypothesen 1-3 nicht widerlegt werden konnten: Das Experiment zeigt, dass die Compiler nicht in der Lage waren Maschinencode zu generieren, der die Speicherbandbreite effizienter nutzt als das mittels Cache Blocking manuell optimierte Programm. Darüber hinaus war es möglich mittels Cache Blocking für große Felder einen deutlichen Performanzzuwachs zu erzielen.

Überraschend ist, dass nur sehr wenige sehr kleine Blockgrößen zu guten Ergebnissen führten (siehe Abb. 2). Offen bleibt, ob die Größe für ein optimales Blocking nur experimentell bestimmt werden kann oder ob man in der Lage ist die Blockgröße als Funktionswert einer Funktion des Schleifenrumpfes darzustellen.

Es wäre denkbar, dass man zusätzliche Annotationen einführt, die den Compiler über Optimierungsmöglichkeiten informieren, z.B. indem obere Schranken für eine Einschränkung des Suchraums der Blockgrößen angegeben werden.

Es wäre zudem wünschenswert mehr Informationen über konkrete Cache Hit/Miss Raten zu erfahren. Hier würde es sich anbieten mittels Profilingbibliotheken, z.B. mit *likwid*, die Programmausführung zu analysieren.

A. Diskussion und Einschränkungen der Validität

Das Experiment konnte eine hohe interne Validität aufweisen. Das Testprogramm erfüllt die Vorausgaben einer speichergebundenen Anwendung. Die theoretische Vermutungen konnten in den Testläufen bestätigt werden. Die verwendeten Compileroptimierungen sind allgemein üblich und spiegeln den State-of-the-Art des Compilerbaus wieder.

Das Experiment besitzt eine hohe Konstruktvalidität: Die gewählte Blockgröße bzgl. der Größe des Prozessorcache ist ausreichend gering, so dass die Datenlokalität für große Feldgrößen maximiert werden konnte.

Dennoch konnte nur eine geringe externe Validität erreicht werden. Zum einen fanden alle Tests auf der gleichen Plattform statt, zum anderen war das Zugriffsmuster und die arithmetische Komplexität für reale Szenarien zu einfach aufgebaut. Hinzu kommt, dass die Konfiguration des Testsystems aufgrund des Experimentsaufbaus nicht realen Szenarien entspricht.

Eine Einschränkung der Validität ergibt sich aus einer fehlenden Varianzanalyse der Testergebnisse. Diese muss aus Zeitgründen für eine spätere Analyse vorbehalten bleiben. Das

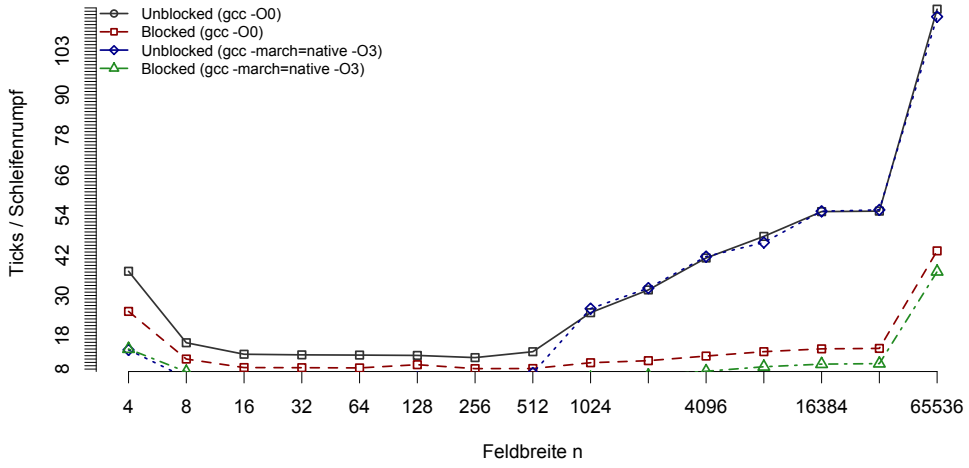

Abbildung 3. Ergebnisse der Testläufe unter GCC.

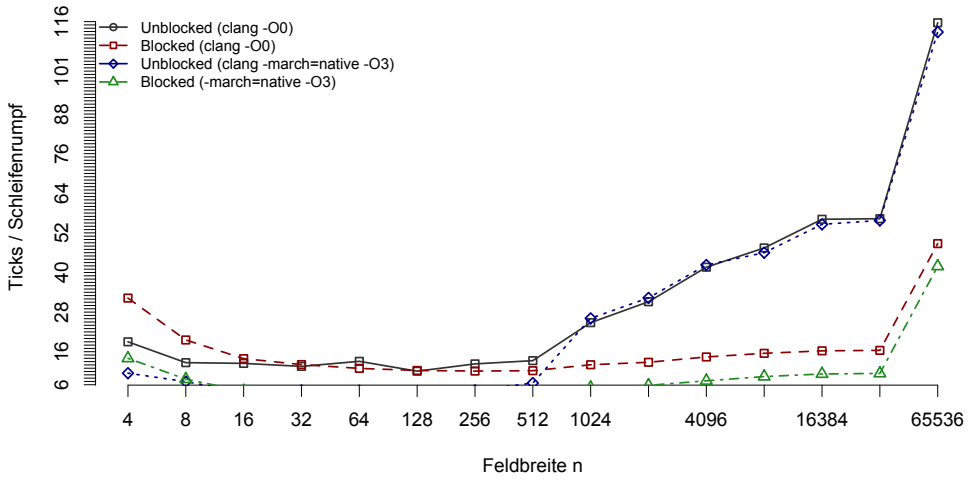

Abbildung 4. Ergebnisse der Testläufe unter CLang.

arithmetisches Mittel der Einzelwerte wurde intern vorberechnet, so dass eine Analyse der Streuung der Einzelergebnisse noch aussteht. Bei einer ersten visuellen Sichtung konnten allerdings außer bei den Ausreißern im geblockten Durchlauf mit Clang keine groben Abweichungen festgestellt werden. Für die Aussagekraft der Werte spricht zudem die extreme Ähnlichkeit der charakteristischen Eigenschaften der Kennlinien.

VIII. Fallbeispiel

Um der Frage nach der Praxisrelevanz der bisherigen Testergebnisse nach zu gehen, wurde ein Fallbeispiel entwickelt.

Für die Vorauswahl geeigneter Kandidaten wurden folgende Kriterien bestimmt: Das Programm muss speichergebunden sein und es sollte eine hohe Relevanz besitzen, d.h. es sollte entweder weit verbreitet oder aufgrund seiner Funktionalität populär sein. Hierfür wurde mit dem Adminstrationtool *debtree* ein Abhängigkeitsbaum der installierten Pakete erstellt. Als geeignete Kandidaten wurden die *zlib* und *libpng* ausgesucht, da sie zentrale Knoten im Abhängigkeitsbaum darstellen. Bzgl. dem Popularitätskriterium fiel die Wahl auf den Gaußschen Weichzeichner innerhalb der Grafikanwendung Gimp 2.9. Der Gauß-Filter dient zum Glätten und Weichzeichnen von Bildinhalten und nimmt eine zentrale Bedeutung in

der Bildverarbeitung ein. Ausgeklammert wurden durchsatzorientierte Programme, die auf OpenCL basieren.

Ein rasches Codereview ergab, dass alle drei Kandidaten klassische Feldzugriffsmuster aufweisen und keine speziell auf Cachezugriff optimierten Tilingmuster enthalten.

Aus Interesse des Autors entfiel die Entscheidung zu Gunsten des Gaußschen Weichzeichners.

Die Modifikation beinhaltete eine Anpassung des Programmcodes von `blur-gauss-selective.c`. In der zentralen Funktion `matrixmult_mmx` wurde ein parametrisierbares Blocking eingefügt.

Diese Datei liegt dabei bereits in einer optimierten plattformabhängigen Variante vor. Zentrale Berechnungen wurde bereits mittels Inline Assembler vektorisiert.

A. Performanztest

Als Testobjekt wurde eine große Grafik mit 5796×3857 Pixeln und 63,8 MB Größe ausgewählt. Die Berechnungszeit wurde mittels der C Headerdatei *cycle.h* berechnet (siehe Kap. III-A). Als Parameter dienten die Blockinggrößen $4 - 256$ im Abstand von Zweipotenzen. Für den Performanztest wurden jeweils vier Durchläufe gestartet. Der erste Durchlauf diente als als *Warm up* und wurde nicht eingerechnet, um Effekte der Gimp-eigenen Cacheverwaltung zu vermeiden. Von den folgenden drei Durchläufen wurde das arithmetisches Mittel als Ergebnis ausgegeben.

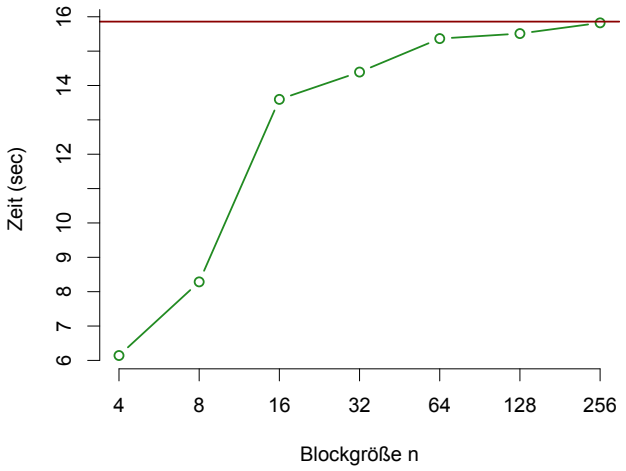

Abbildung 5. Ergebnisse des Fallbeispiels: Blocking des Gaußschen Weichzeichner unter Gimp 2.9.

Abbildung 5 zeigt das Ergebnis der Messungen. Der rote Balken stellt die Referenzmessung ohne Blocking da. Die grüne Kennlinie zeigt die Messungen unter den verschiedenen Blockgrößen.

B. Diskussion

Bei den Tests konnte mit Blockinggrößen unter 256 ein deutlicher Performanzgewinn verzeichnet werden.

Einschränkend muss erwähnt werden, dass nur eine Bilddatei getestet wurde, sowie nur eine Architektur. Außerdem wurden die gleichen Einschränkungen bei der Prozessorkonfiguarion gemacht, die bereits beim Experiment erwähnt wurden. Die Rechnerkonfiguration entsprach somit keinem Produktivsystem.

IX. Related Work

Loop Tiling ist eine gut erforschte Quellcodetransformation und es existiert ausführliche Literatur. Eine gute Anlaufstelle besitzen die Artikel [3] und [1]. Laut [2] wird Cache Blocking bisher hauptsächlich im HPC Bereich eingesetzt. Eine ausführliche Auseinandersetzung mit diesem Thema bietet die Doktorarbeit von Matthias Christen [4].

Einen alternativen und plattformunabhängigen Ansatz verfolgen *Cache-oblivious Verfahren* [5]. Diese versuchen ähnlich zum *Divide-and-Conquer* Verfahren die zu bearbeitenden Datenbereiche so zu verkleinern, dass sie auf möglichst vielen Systemen in den Prozessorcache passen.

X. Zusammenfassung

Diese Studie evaluierte Cache Blocking als durchsatzoptimierende Variante des loop Tilings mit Hilfe eines einfachen speichergebundenen Beispielprogramms.

Es wurde gezeigt, dass durch die manuelle Anwendung einer einfachen und grundlegenden Schleifentransformation eine Programmoptimierung durchgeführt werden kann, die durch die standardmäßigen Compileroptimierungen nicht abgedeckt ist.

Es war möglich, aufzuzeigen, dass für große Datenmengen ein deutlicher Performanzgewinn erzielbar ist.

Außerdem konnte anhand des Fallbeispiels eines Gaußschen Weichzeichners ein konkreter wenn auch plattformabhängiger Anwendungsfall demonstriert werden.

Abschließend lässt sich hinzufügen, dass sich zusätzliche Messungen auf anderen Plattformen anbieten würden, um die Testergebnisse auf eine belastbarere Basis zu stellen.

Literatur

[1] M. Wolfe, “More iteration space tiling,” in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’89. New York, NY, USA: ACM, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[2] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2010.

[3] F. Irigoin, “Tiling,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, pp. 2040–2049.

[4] M.-M. Christen, “Generating and auto-tuning parallel stencil codes,” Ph.D. dissertation, Philosophisch-Naturwissenschaftlichen Fakultät der Universität Basel, Basel, 2011.

[5] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 1999, pp. 285–297.

[6] (2013, 07) Cache blocking techniques. [Online]. Available: https://software.intel.com/en-us/articles/cache-blocking-techniques

[7] M. D. Lam, E. E. Rothberg, and M. E. Wolf, “The cache performance and optimizations of blocked algorithms,” *SIGPLAN Not.*, vol. 26, no. 4, pp. 63–74, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/106973.106981