

Anpassung von Stencil-Codes zur Laufzeit mit Hilfe von Umschreiben auf Binärebene für dynamisch bestimmtes Speicherlayout

Konrad M. Pröll¹

Abstract: Die Optimierung von Stencil-Codes ist eine zentrale Herausforderung im Bereich des Hochleistungsrechnens. Die meisten Ansätze fokussieren sich entweder darauf, diese zur möglichst effizienten Berechnung auf großen Parallelrechenstrukturen zu parallelisieren oder die möglichst effiziente Ausnutzung des Caches, um die Leistung zu steigern. Viele Stencil-Codes nutzen einfache Arrays zur Speicherung der Matrix. Komplexere Datenstrukturen erhöhen die Rechenzeit des Stencil-Codes dadurch, dass die Berechnung der Speicheradresse einer Zelle deutlich komplizierter wird und sogar bedingte Sprünge enthält. In dieser Arbeit wird ein Ansatz vorgeschlagen, wie aus bereits kompilierten Stencil-Codes zur Laufzeit das Speicherlayout analysiert werden kann und das Programm durch partielle Evaluation optimiert wird. Im Gegensatz zu konventioneller partieller Evaluation wird hierbei nicht für konstante Argumente, sondern für Wertebereiche, in denen sich ein Argument befindet, spezialisiert. Durch diese Methode können die Leistungseinbußen merklich reduziert werden.

Keywords: Programmoptimierung; Partielle Evaluation; Stencil-Codes; High Performance Computing; Umschreiben auf Binärebene

1 Einleitung

Stencil-Codes sind eine Art von Programmen, die häufig im Bereich des wissenschaftlichen Rechnens zum Einsatz kommen. Hierbei wird der Wert sämtlicher Zellen einer Matrix unter Berücksichtigung der Werte der Nachbarzellen in jeder Iteration neu berechnet, indem ein gewichteter Durchschnitt dieser gebildet wird. Die Form des Stencil bestimmt, welche Nachbarzellen bei der Neuberechnung berücksichtigt werden. Stencil-Codes können für alle naturwissenschaftlichen Probleme, die auf das Lösen von Differenzialgleichungen reduzierbar sind, verwendet werden. Diese Gruppe von Problemen beinhaltet physikalische Simulationen (Strömungsdynamik, Ausbreitung von Wärme, Auswirkungen von Erdbeben), aber auch Bereiche der Quantenmechanik oder der Bildverarbeitung [Da09; RYQ11]. Sie werden genutzt, um kontinuierliche Differenzialgleichungen zu diskretisieren, sodass diese überhaupt erst berechenbar werden.

Häufig werden Stencil-Codes für sehr große Matrizen über eine große Anzahl an Iterationen verwendet, was zu einer hohen Rechenzeit führt, weshalb die Optimierung dieser Programme

¹ Technische Universität München, Fakultät für Informatik, konrad.proell@tum.de

ein wichtiges Forschungsgebiet im Bereich des Hochleistungsrechnen (HPC) ist. Die meisten Optimierungen fokussieren sich auf das Anpassen des Programms zur Ausführung auf Parallelarchitekturen, z. B. über Message Passing Interface (MPI), nicht zuletzt weil die einzelnen Neuberechnungen innerhalb eines Iterationsschrittes meist nur von den Werten des vorhergehenden Schrittes abhängt, sodass das Parallelisieren grundsätzlich ohne größere Schwierigkeiten möglich ist, sowie zur effektiven Ausnutzung des Caches, da einfache Stencil-Codes meist durch die Speicherbandbreite der Maschine limitiert werden [Kr07].

Gewöhnlich werden die Matrizen für Stencil-Codes in normalen Arrays gespeichert, sodass die Speicheradressen der Zellen mittels Offset-Berechnung ermittelt werden können. Unter Umständen kann es vorteilhaft sein, ein dynamisch bestimmtes Speicherlayout zu verwenden und verschiedene Teile der Matrix an verschiedenen Adressen abzulegen. Solche dynamisch bestimmte Speicherlayouts besitzen den Nachteil, dass die Berechnung der Speicheradresse einer Zelle deutlich komplizierter ist und sogar bedingte Sprünge enthält.

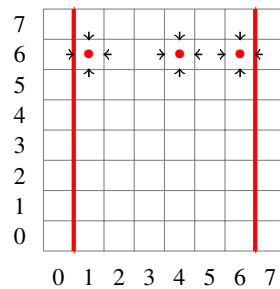


Abb. 1: Beispiel für ein Speicherlayout. Die linke und rechte äußere Spalte werden an anderen Orten als der Rest der Matrix gespeichert, etwa um die “Halo”-Zellen bei Parallelisierung dichter abzuspeichern.

In Abb. 1 ist eine Matrix zu sehen, die in drei verschiedenen Arrays abgelegt ist: Eines für die äußere linke Spalte, eines für die äußere rechte sowie eines für den Rest. Ein solches Speicherlayout kann z. B. sinnvoll sein, um die “Halo”-Zellen, die bei einer Parallelisierung auf Architekturen mit verteiltem Speicher genutzt werden [KS10], an einer anderen Stelle im Speicher abzulegen, sodass sie dichter im Speicher liegen. Im Falle eines solchen Speicherlayouts muss bei jeder Ausführung des Stencil-Kernels (d. h. bei jeder Neuberechnung eines Wertes) für jeden Punkt des Stencils geprüft werden, in welchem Array er liegt. Ein gewöhnlicher 4-Punkt-Stencil enthält also bei dem dargestellten Layout 4 - 8 bedingte Sprünge, die die Rechnerzeit eines Stencil-Codes erheblich erhöhen.

In diesem Paper wird eine Methode vorgestellt, um zuerst aus dem kompilierten Stencil-Kernel das Speicherlayout in einer Analysephase zu rekonstruieren (die möglichen Varianten des Kernels ermitteln), und den Kernel mittels partieller Evaluation für dieses Speicherlayout zu optimieren. Die weitere Arbeit ist folgendermaßen aufgebaut: Zuerst werden die bearbeitete Problemstellung sowie verwandte Arbeiten vorgestellt. Daraufhin soll das Projekt, in das die Methode eingebettet wird, knapp vorgestellt werden, bevor die konkrete

Vorgehensweise präsentiert wird. Anschließend wird die Implementierung in Bezug auf die Verbesserung der Laufzeit evaluiert, bevor die Ergebnisse beurteilt werden.

Die Forschungsbeiträge dieser Arbeit sind:

- Identifizieren des Speicherlayouts aus kompiliertem Stencil-Kernel
- Optimieren von Stencil-Kernels für dynamisch bestimmtes Speicherlayout mittels partieller Evaluation für Wertebereiche

2 Verwandte Arbeiten

In diesem Abschnitt sollen verwandte Arbeiten vorgestellt und diskutiert werden.

Spezialisierung bzw. partielle Evaluation ist eine Programmtransformationstechnik, bei der vor der eigentlichen Programmausführung Teile bereits ausgewertet werden, die von Variablen abhängig sind, die bei jedem Aufruf gleich sind. Hierbei wird das Programm auf jene Variablen “spezialisiert”. Spezialisierung wird vor allem aus zwei Gründen eingesetzt: Zum einem zur Generierung effizienter Compiler²[JGS93, S. 13 ff.], und zum anderen zur Optimierung von Programmen.

Srinivasan und Reps [SR15] stellen ein Konzept zur Spezialisierung von Maschinencode vor. Dabei wird der Maschinencode in einen Syntaxbaum umgewandelt, in dem sowohl die Abhängigkeiten der einzelnen Prozeduren untereinander als auch die Abhängigkeiten innerhalb dieser analysiert werden. Im Anschluss können mittels “Slicing” die Abhängigkeiten der einzelnen Befehle untereinander analysiert werden. Auf dieser Ebene kann dann der Programmcode für verschiedene konstante Parameter spezialisiert werden. In [SR16] stellen die Autoren eine verbesserte Methode des Slicing vor.

Köster et al. [Kö14] verwenden Spezialisierung für Stencil-Codes, um Programme, die für ein variables Stencil-Layout entwickelt wurden, auf ein konkretes zu spezialisieren. Im Rahmen ihrer Arbeit haben sie eine Zwischenrepräsentation entwickelt, die Programme in einer Baumstruktur speichert und auf jegliche Blockstrukturen verzichtet. Jeder Knoten des Baums stellt eine Definition dar, während Kanten für direkte Abhängigkeiten zwischen Definitionen stehen. Auf dieser Ebene wird dann der Programmcode für bestimmte Variablen spezialisiert und alle von diesem Parameter abhängige Ausdrücke ausgewertet [LKH15]. Die Autoren messen bei der Verwendung eines Jacobi-Kernel spürbar bessere Leistungen für die spezialisierte Variante gegenüber der variablen bei einer Beschleunigung um einen Faktor zwischen 2 und 3.

Diese Ansätze teilen mit anderen geläufigen Konzepten der Spezialisierung, dass hierbei der Programmcode für *konstante* Parameter spezialisiert wird. Im Fall eines dynamisch

² Durch Spezialisierung eines Spezialisierers für einen Interpreter wird ein Compiler generiert

bestimmten Speicherlayouts soll nicht für einen konstanten Wert, sondern einen Wertebereich spezialisiert werden. Daher wird ein neuer Ansatz benötigt, in dem Funktionen nicht nur für konstante Werte, sondern ganze Wertebereiche spezialisiert werden.

3 Hintergrund

Am Lehrstuhl für Rechnerarchitektur & Parallele Systeme der Technischen Universität München³ wird seit 2015 im Projekt **DBrew**⁴ eine C-Bibliothek entwickelt, die es ermöglicht bereits kompilierten Code mittels Umschreiben auf Binärebene zu optimieren. Hierbei wird der Binärcode in Maschinencode, strukturiert in Basic Blocks, dekodiert und nach der Optimierung wieder in Binärcode konvertiert.

Im ersten Schritt kann der dekodierte Programmcode spezialisiert werden. Hierbei kann der Programmierer diejenigen Parameter, die spezialisiert werden, als “statisch” setzen. Anschließend wird ein Funktionsaufruf emuliert. Hierbei wird für alle Werte gespeichert, ob sie statisch sind. Operationen, deren Ergebnisse statisch sind, werden hierbei bereits während der Emulation ausgeführt. Basic Blocks werden während der Emulation in einem Stack-Speicher verwaltet: Zu Beginn wird derjenige Block, der die Sprungmarke der jeweiligen Funktion enthält, auf dem Stack abgelegt und bearbeitet, während in den weiteren Schritten immer derjenige Block auf dem Stack abgelegt wird, der auf den gerade bearbeiteten Block folgt. Falls ein Block mit einem bedingten Sprung endet, sodass mehrere verschiedene Blöcke auf ihn folgen können, wird zuerst geprüft, ob die Sprungbedingung aufgrund der Parameter, für die spezialisiert wird, ausgewertet werden kann. Sollte dies der Fall sein, wird der Sprungbefehl durch einen unbedingten Sprung ersetzt, ansonsten werden beide möglichen Blocks auf dem Stack abgelegt [WB16].

Im Anschluss an die Emulation kann der Maschinencode mittels LLVM weiter optimiert werden [EW17]. Abschließend wird ein Pointer zurückgegeben, über den die Funktion aufgerufen werden kann.

4 Vorgehensweise

Im Folgenden soll die konkrete Vorgehensweise vorgestellt werden.

4.1 Metainformationen

Um Wertebereiche für Eingabeparameter festzustellen, für die Sprungbedingungen andere Werte ergeben, wird zuerst für jeden Wert (d. h. Register, Flag und Stack) ein **symbolischer**

³ <https://www.lrr.in.tum.de/startseite/>

⁴ Dynamic Binary rewriting

Ausdruck gespeichert, der die Abhängigkeit des Wertes von den Parametern der Funktion beschreibt. Symbolische Ausdrücke werden in einem Ausdrucksbaum gespeichert: Die Blätter beinhalten entweder einen konstanten Wert, falls der jeweilige Wert von keinem Parameter abhängt, oder den Parameter, von dem der jeweilige Wert abhängt. Die Knoten im Baum werden genutzt, um mathematische Operationen zu speichern. In Abb. 3 ist ein solcher Ausdrucksbaum abgebildet, der die Operation $(x + 5) * 2$ speichert, wobei x ein Parameter der Funktion ist.

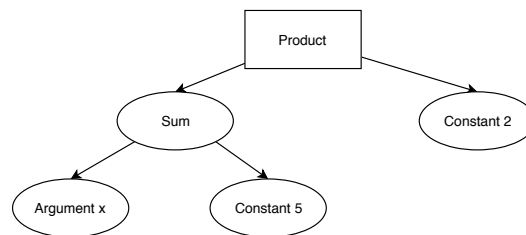


Abb. 2: Aufbau des symbolischen Ausdrucks für $(x + 5) * 2$

Im nächsten Schritt wird für jeden Wert ein **Wertebereich** eingeführt, der jeweils den minimalen und maximalen Wert, den ein Ergebnis annehmen kann, beschreibt. Eine Variable kann auch mehrere disjunkte Wertebereiche haben. Anfangs setzt der Nutzer den Wertebereich der Parameter der Funktion. Anschließend wird bei jeder Operation der Wertebereich des jeweiligen Ergebnisses berechnet.

Abschließend wird in einem **Kontrollflussgraph** die Struktur des Programmes festgehalten. In jedem Knoten wird hierbei ein Basic Block gespeichert, während Kanten zeigen, welche Knoten von einem Basic Block aus erreicht werden können. Zusätzlich wird die jeweils letzte Instruktion des Blocks gespeichert, da diese den Kontrollfluss betrifft, sowie etwaige Sprungbedingungen in Form eines symbolischen Ausdrucks und die Wertebereiche aller Funktionsargumente, für die der jeweilige Knoten erreicht werden kann.

4.2 Analyse

Nachdem die Metainformationen, die während eines Durchlaufs zusätzlich betrachtet werden, vorgestellt wurden, soll nun gezeigt werden, wie mit diesen mögliche Spezialisierungen identifiziert werden können.

Anfangs setzt der Nutzer die Wertebereiche der Funktionsparameter auf die zu erwartbaren Werte. Im Anschluss wird das ein Funktionsaufruf wie in Abschnitt 3 beschrieben emuliert. Hierbei wird denjenigen Registern, in denen die Funktionsargumente stehen, symbolische Ausdrücke angehängt, dass das Parameter der Funktion sind.

Im weiteren Verlauf der Emulation wird zwischen Instruktionen, die den Kontrollfluss des Programms nicht beeinflussen (einfache Instruktionen), und solchen, die den Kontrollfluss beeinflussen unterschieden.

Einfache Instruktionen werden folgendermaßen bearbeitet:

1. Alle Operanden der Instruktion sind konstant: In diesem Fall hängt das Ergebnis nicht (mehr) von Argumenten der Funktion ab. Deshalb kann eine solche Instruktion einfach emuliert werden, der symbolische Ausdruck des Ergebnisses wird auf eine Konstante gesetzt.
2. Es gibt nicht-konstante Argumente, aber für jeden dynamischen Operanden liegt ein symbolischer Ausdruck vor: Die Instruktion wird symbolisch ausgeführt, sodass ein neuer Ausdruck entsteht, der das Ergebnis der Operation als Ausdrucksbaum enthält. Anschließend wird der Wertebereich des Ergebnisses ermittelt. Sollte der Wertebereich nur einen Wert umfassen, ist das Ergebnis konstant, sodass der ermittelte Ausdruck verworfen und wiederum durch einen symbolischen Ausdruck für eine Konstante ersetzt werden kann, andernfalls sind der zuvor ermittelte Ausdruck und Wertebereich das Ergebnis dieser Operation.
3. Es gibt nicht-konstante Operanden, für die kein symbolischer Ausdruck vorliegt⁵: In diesem Fall können Abhängigkeiten nicht beurteilt werden, weshalb auch für das Ergebnis kein Ausdruck ermittelt werden kann. Wenn dieses Ergebnis in weiteren Instruktionen referenziert wird, kann auch über deren Ergebnis keine Aussage getroffen werden.

Instruktionen, die den **Kontrollfluss** beeinflussen, können zu unterschiedlichen Spezialisierungen führen, weswegen sie gesondert behandelt werden müssen. Solche Instruktionen können, müssen aber nicht zwingend bedingte Sprünge sein. Beispiele für weitere solche Instruktionen sind *ret* oder *jmp*. Sie werden folgendermaßen bearbeitet:

1. Der Sprung wird ohne Bedingung ausgeführt oder die Sprungbedingung ist konstant: Diese Art von Instruktion beeinflusst die möglichen Spezialisierungen nicht. Im Fall von Sprungbefehlen wird der Kontrollflussgraph nicht aktualisiert und der nächste Block wird zum aktuellen Knoten hinzugefügt. Im Falle eines *ret* wird diese Instruktion im Knoten gespeichert, da die Funktion an dieser Stelle des Kontrollflussgraphen beendet ist.
2. Teile der Sprungbedingungen sind dynamisch, und es liegen für alle dynamischen Sprungbedingungen symbolische Ausdrücke vor: Dies sind diejenigen bedingten Sprünge, für die das Programm anschließend spezialisiert werden soll. Die Sprungbedingung wird symbolisch ausgewertet, sodass ein neuer symbolischer Ausdruck entsteht, der beschreibt, wann die Bedingung erfüllt ist. Sollte der Ausdruck nur eine Variable (d. h. Funktionsparameter) enthalten, wird er danach aufgelöst⁶. Das Ergebnis

⁵ z. B. weil die Verarbeitung der Instruktion noch nicht implementiert wurde oder die Variable global außerhalb der Funktion definiert wurde

⁶ Das Auflösen komplexerer Ausdrücke führt zu sehr feingranularen Spezialisierungen. Funktionen für Wertebereiche zu spezialisieren, die ohnehin kaum genutzt werden, verspricht wenig Leistungsverbesserung und blockiert unverhältnismäßig Speicher.

dieser Operation wird neben dem genauen Sprungbefehl im aktuellen Knoten gespeichert. Anschließend werden an diesen für die beiden möglichen Ausführungspfade zwei Kinder angehängt. In beide Kinder wird auf Basis der Sprungbedingung und der Wertebereiche im Elternknoten derjenige Wertebereich der Funktionsparameter berechnet.

3. Dynamische Sprungbedingungen, für die keine symbolischen Ausdrücke vorliegen: Diese Sprungbedingungen können nicht analysiert werden. An den Knoten des aktuellen Blocks werden zwei Kinder angehängt, in die die Wertebereiche der Funktionsargumente des aktuellen Knotens kopiert wird. Dadurch können trotzdem noch folgende Sprünge analysiert werden, sollten sie verwertbare Sprungbedingungen haben.

Am Ende des Analyselaufs können die Ergebnisse dem Kontrollflussgraphen entnommen werden. Wie in Abb. 3 zu sehen ist, enthalten sämtliche Knoten jeweils, für welche Wertebereiche der Funktionsparameter diese erreicht werden kann, die letzte Instruktion und, falls es sich hierbei um einen bedingten Sprung handelt, die Sprungbedingung. In der Beispielfunktion, die einen Parameter hat, wurde der Wertebereich des Arguments auf $[0, 9]$ festgesetzt. In der Folge trifft die Programmausführung auf einen bedingten Sprungbefehl *JL*⁷, wobei die Sprungbedingung ein Vergleich des Funktionsarguments mit 5 ist. Die linke Hälfte des Baums wird deshalb erreicht, wenn das Argument < 5 ist, weswegen im entsprechenden Knoten als Intervall $[0, 4]$ gespeichert ist. Am Ende können die Spezialisierungen den Blättern entnommen werden: Das linke Blatt kann nicht erreicht werden, weswegen kein Wertebereich darin gespeichert ist. Das zweite Blatt wird für $X \in [0, 4]$ erreicht, das dritte für $X = 5$ und das letzte für $X \in [6, 9]$. Deshalb können durch Spezialisierung dieser Varianten alle bedingten Sprünge entfernt werden.

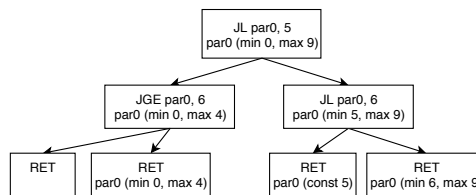


Abb. 3: Beispiel eines Kontrollflussgraphen am Ende eines Analyselaufs. Es wurden drei mögliche Spezialisierungen festgestellt, die den Blättern des Baumes entnommen werden: $X \in [0, 4]$, $X = 5$ sowie $X \in [6, 9]$.

4.3 Spezialisierung

Nachdem die gegebene Funktion analysiert worden ist und dadurch diejenigen Wertebereiche identifiziert wurden, für die die Sprungbedingungen der bedingten Sprünge zum gleichen

⁷ Jump if Less, wird ausgeführt, wenn der erste Operand kleiner als der zweite ist (d. h. $SF \neq OF$ [In16])

Ergebnis führen, kann die Funktion durch weitere Läufe von DBrew für die einzelnen Bereiche spezialisiert werden. Hierbei werden zu Beginn die Wertebereiche der Parameter auf die in der Analyse festgestellten gesetzt und eine weitere Emulation gestartet. Die Spezialisierung wird hierbei dadurch erwirkt, dass durch die engeren Wertebereiche Sprungbedingungen, die während der Analyse noch dynamisch waren, konstant werden und ausgewertet werden können, sodass die bedingten Sprungbefehle durch unbedingte Sprünge ersetzt werden.

5 Evaluation und Diskussion

In diesem Abschnitt soll die zuvor vorgestellte Methode zur Spezialisierung evaluiert werden. Hierbei wird zuerst das benutzte System vorgestellt, bevor die erzielten Geschwindigkeitszunahmen in verschiedenen Testfällen gezeigt werden. Abschließend werden die Ergebnisse diskutiert.

5.1 Testaufbau

Als Testsystem wurde ein Intel®Core™i3-2310m mit 2.1 GHz mit dem Betriebssystem Ubuntu 17.10 verwendet. Als Compiler wurde *gcc 7.2.0* mit den Optionen *-O2 -mavx* genutzt. Die genutzte Kernelversion ist *4.13.0-36-generic*. In jedem Durchlauf wurde der Stencil-Code über 50 Iterationen bei einer zweidimensionalen Matrix mit der Auflösung 2000*2000 ausgeführt. Jede dieser Messungen wurde 200 mal wiederholt. Sämtliche Messungen wurden seriell durchgeführt.

5.2 Messergebnisse

Im Folgenden soll anhand zweier Testfälle der Funktionsumfang der Analyse- und Spezialisierungsphase anhand eines Jacobi-Kernels, der zur Approximation der Wärmeleitungsgleichung verwendet wird, gezeigt werden. Die drei Testfälle sind in Abb. 4 dargestellt. Laufzeitmessungen werden für fünf Konfigurationen vorgenommen:

- Standard: In dieser Konfiguration wird die Domäne entlang der Zeilen durchlaufen.
- Sortiert: In dieser Konfiguration wird die Ausführreihenfolge so angepasst, dass innerhalb einer Iteration alle Aufrufe, die zu einem Speicherlayout gehören, durchlaufen werden, bevor die Ausführung zum nächsten Layout übergeht. Außerdem werden innerhalb eines Bereichs die Schleifen so angepasst, dass der Cache möglichst gut ausgenutzt wird. Hierdurch soll außerdem getestet werden, ob ein moderner Branch-Predictor einen ähnlichen Effekt wie die Spezialisierung erzielen kann.

- **Matrix:** In dieser Konfiguration wird DBrew genutzt, um den Stencil-Kernel mittels gewöhnlicher Spezialisierung für die Größe der Matrix spezialisiert.
- **Matrix, sortiert:** In dieser Konfiguration wird zuerst der Kernel wie in der vorherigen Variante spezialisiert. Anschließend werden die Schleifen wie in der Variante zuvor sortiert.
- **Spezialisiert:** In dieser Variante wird zuerst der Stencil-Code wie in dieser Arbeit vorgeschlagen spezialisiert. Die Ausführungsreihenfolge ist identisch zur Option "sortiert".

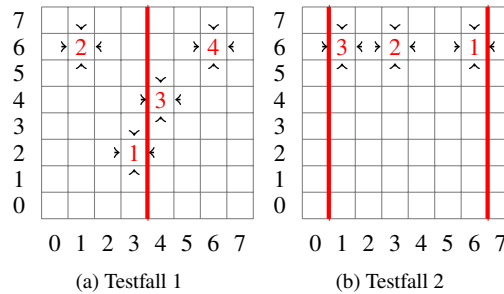


Abb. 4: Behandelte Testfälle. In Abb. 5a wird die Matrix in der Mitte vertikal in zwei Hälften geteilt, weswegen vier verschiedene Spezialisierungen möglich sind. In Abb. 5b werden die linke und rechte Spalte der Matrix an einer anderen Adresse gespeichert, etwa um *Halo*-Zellen bei einer Parallelisierung dichter abzulegen. Die nummerierten Felder stellen die unterschiedlichen Spezialisierungen dar.

Testfall 1

In Testfall 1 wird die Matrix in der Mitte vertikal in zwei Hälften geteilt. Die Analysephase erkennt korrekt die vier möglichen Spezialisierungen $X = 3$, $X \in [1, 2]$, $X = 4$ und $X \in [5, 6]$. Hierbei wird die linke Hälfte zeilenorientiert ("row-major") und die rechte Hälfte spaltenorientiert ("column-major") gespeichert.

Wie in Abb. 5 zu sehen ist, wird durch das Umsortieren ein erheblicher Geschwindigkeitsvorteil erzielt, da dann auch bei der spaltenorientierten rechten Hälfte der Cache gut ausgenutzt wird. Durch das Spezialisieren für die einzelnen Speicherbereiche wird ein Geschwindigkeitszuwachs von etwa 8% im Vergleich zur gewöhnlichen Spezialisierung erzielt.

Testfall 2

In Testfall 2 wird die Matrix in drei Teile geteilt, wobei jeweils die äußerste linke und rechte Spalte an einer anderen Adresse gespeichert werden. Dies kann z. B. dazu genutzt werden,

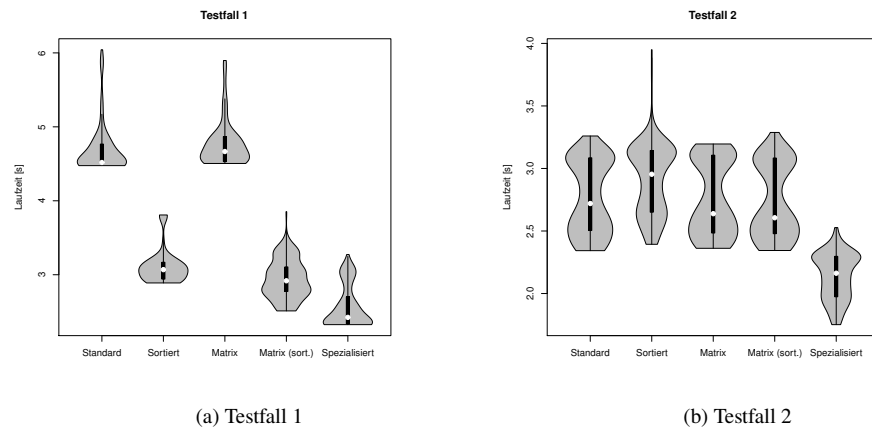


Abb. 5: Ergebnisse der Laufzeitmessungen. In beiden Testfällen kann durch Nutzung spezialisierter Kernel für die jeweiligen Ausführungspfade die Laufzeit verringert werden.

“Halo”-Zellen beim Parallelisieren für eine Architektur mit verteiltem Speicher an einer anderen Stelle im Speicher dichter abzulegen. Die Analysephase erkennt korrekt die drei möglichen Spezialisierungen $X = 6$, $X \in [2, 5]$ und $X = 1$.

Wie Abb. 5 zu entnehmen ist, ist in diesem Fall die Spezialisierung etwa 10% schneller. Anders als im vorherigen Fall führt das Umsortieren der Schleifen hier zu einer Verlangsamung, weil dadurch das Cache-Verhalten nicht verbessert, sondern verschlechtert wird. Der höhere Speedup der Spezialisierung als im vorherigen Fall erklärt sich dadurch, dass in diesem Fall für jeden Punkt zwei Bedingungen statt nur einer geprüft werden müssen.

5.3 Diskussion

Der vorgestellte Ansatz führte zu einer merklichen Leistungsverbesserung: Speedups im Bereich von 8% - 10% sind zwar im Vergleich zum Potential in Optimierungen zur

besseren Cachenutzung recht gering, sind aber unter Berücksichtigung dessen, dass einfache Stencil-Codes stark durch die Speicherbandbreite limitiert werden, durchaus bemerkbar.

Ein Nachteil dieses Ansatzes, der bei den untersuchten Speicherlayouts nicht auftritt, ist, dass unter Umständen toter Code generiert wird: Wenn Sprungbedingungen von komplexen Berechnungen abhängen, und die Sprungbedingung in der Spezialisierung konstant wird, trifft dies nicht zwingend auf die vorherigen Berechnungen zu. Deshalb werden dann bei jedem Durchlauf diese Berechnungen durchgeführt, aber das Ergebnis nicht weiter genutzt. Durch die anschließende Transformation des spezialisierten Codes in eine Zwischenrepräsentation wie LLVM kann dieses Problem behoben werden.

Eine weitere Hürde ist, dass komplexere Speicherlayouts zu sehr vielen Spezialisierungen führen kann. In diesem Fall könnte z. B. über eine Analyse festgestellt werden, welche Varianten besonders oft ausgeführt werden, und dann nur diese spezialisieren und für die restliche Ausführung die generische Funktion nutzen.

6 Fazit

Im Rahmen dieser Arbeit wurde vorgeschlagen, wie Stencil-Kernel für dynamisch bestimmte Speicherlayouts mittels Spezialisierung für Wertebereiche optimiert werden. Zuerst wurde gezeigt, wie im Rahmen einer Analysephase das Layout so identifiziert werden kann, dass die Wertebereiche der Eingabeparameter, für die die einzelnen Punkte des Stencils auf den gleichen Speicherbereich zugreifen, festgestellt werden. Anschließend wurde der Kernel so spezialisiert, dass die Teile des Programms, die für alle Werte in diesem Bereich konstant sind, bereits ausgewertet wurden, während diejenigen, die vom tatsächlichen Wert abhängen, erst zur Laufzeit berechnet werden. Es wurde gezeigt, dass dadurch Beschleunigungen im Bereich von 10% erzielt werden können.

Danksagung

Diese Arbeit ist im Rahmen meiner Masterarbeit am Lehrstuhl für Rechnerarchitektur & Parallele Systeme entstanden. Mein Dank gilt Josef Weidendorfer von Alexis Engelke für die Betreuung dieser Arbeit.

Literatur

- [Da09] Datta, K.: Auto-tuning Stencil Codes for Cache-based Multicore Platforms, AAI3411221, Diss., Berkeley, CA, USA, 2009, ISBN: 978-1-124-03708-0.
- [EW17] Engelke, A.; Weidendorfer, J.: Using LLVM for Optimized Lightweight Binary Re-Writing at Runtime. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). S. 785–794, Mai 2017.

- [In16] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-L, 253666-060US, Intel Corporation, Sep. 2016.
- [JGS93] Jones, N. D.; Gomard, C. K.; Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
- [Kö14] Köster, M.; Leiða, R.; Hack, S.; Membarth, R.; Slusallek, P.: Platform-Specific Optimization and Mapping of Stencil Codes through Refinement. In: Proceedings of the First International Workshop on High-Performance Stencil Computations (HiStencils). Vienna, Austria, S. 1–6, 21. Jan. 2014.
- [Kr07] Krishnamoorthy, S.; Baskaran, M.; Bondhugula, U.; Ramanujam, J.; Rountev, A.; Sadayappan, P.: Effective Automatic Parallelization of Stencil Computations. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '07, ACM, San Diego, California, USA, S. 235–244, 2007, ISBN: 978-1-59593-633-2.
- [KS10] Kjolstad, F. B.; Snir, M.: Ghost Cell Pattern. In: Proceedings of the 2010 Workshop on Parallel Programming Patterns. ParaPLoP '10, ACM, Carefree, Arizona, USA, 4:1–4:9, 2010, ISBN: 978-1-4503-0127-5.
- [LKH15] Leiða, R.; Köster, M.; Hack, S.: A Graph-based Higher-order Intermediate Representation. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO '15, IEEE Computer Society, San Francisco, California, S. 202–212, 2015, ISBN: 978-1-4799-8161-8.
- [RYQ11] Rahman, S. M. F.; Yi, Q.; Qasem, A.: Understanding Stencil Code Performance on Multicore Architectures. In: Proceedings of the 8th ACM International Conference on Computing Frontiers. CF '11, ACM, Ischia, Italy, 30:1–30:10, 2011, ISBN: 978-1-4503-0698-0.
- [SR15] Srinivasan, V.; Reps, T.: Partial Evaluation of Machine Code. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, ACM, Pittsburgh, PA, USA, S. 860–879, 2015, ISBN: 978-1-4503-3689-5.
- [SR16] Srinivasan, V.; Reps, T.: An Improved Algorithm for Slicing Machine Code. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016, ACM, Amsterdam, Netherlands, S. 378–393, 2016, ISBN: 978-1-4503-4444-9.
- [WB16] Weidendorfer, J.; Breitbart, J.: The Case for Binary Rewriting at Runtime for Efficient Implementation of High-Level Programming Models in HPC. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). S. 376–385, Mai 2016.