Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Hausarbeit im Seminar:
Effiziente Programmierung

Performance Modellierung - Roofline Modellierung

Theodor Wulff

6wulff@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 6948352
Fachsemester 5

Betreuer: Jannek Squar

Inhaltsverzeichnis

1 Einleitung

Meine Seminarbeit im Seminar „Effiziente Programmierung“ befasst sich mit dem Thema „Performance Modellierung“ im Hinblick auf den Spezialfall des Roofline Modells.

Die Arbeit beginnt mit einer Einführung zum Thema Modellierung. Dabei wird auf die Unterschiede zwischen verschiedenen Methoden eingegangen, sowie die Vorteile gut gewählter Modellierung kurz genannt.

Im Hauptteil der vorliegenden Arbeit wird das Modell zunächst allgemein erläutert, anschließend werden die einzelnen Facetten noch genauer beleuchtet.

Den Schluss bildet eine praxisnahe Anwendung des Roofline Modells mittels des Intel Advisors. Hier wird anhand von kurzen Codebeispielen gezeigt, welche Optimierungen im Modell erkennbar sind und einen Einfluss auf die Performance haben können.

1.1 Motivation - Modellierung

Im allgemeinen findet Modellierung in allen möglichen Anwendungsbereichen in unterschiedlichen Formen statt. Dabei verändert sich die Modellierungsmethode je nachdem was zu welchem Zweck modelliert wird. Soll zum Beispiel ein Überblick über Zusammenhänge gegeben werden oder sollen die Ergebnisse einer Analyse dargestellt werden?

Die Qualität eines gewählten Modelles zeichnet sich in der Regel durch mehrere Dinge aus. Ein gutes Modell sollte möglichst intuitiv lesbar sein und mit einer gewissen Übersichtlichkeit einherkommen. Viele Modelle erfordern zwar Vorwissen (so auch das Roofline Modell), aber ist dieses Vorwissen einmal vorhanden, so lassen sich ohne großen Zeitaufwand viele Informationen entnehmen. [LWS+14]

In der Performance Modellierung lassen sich die Modellierungsmethoden grob in zwei Arten unterteilen: Statistische und „Bound and Bottleneck“ Analysemethoden.

1.2 Statistische Analysemethoden

Statistische Analysemethoden geben den zu untersuchenden Systemzustand möglichst genau wieder. Ziel ist es durch die Auswertung großer Datensätze Trends im Systemverhalten zu erkennen und darauf aufbauend das System zu optimieren. Statistische Analysen haben den Vorteil, dass das korrekte Erkennen des Ist-Zustands des System zu einer sehr genauen, idealisierten, teilweise problemfreien Modelldarstellung führen kann. Allerdings haben statistische Methoden auch den Nachteil, dass nicht unbedingt ohne viel Vorwissen ablesbar ist, welche Optimierungen eingeführt werden müssen, um die Performance noch weiter zu steigern. Auch ist nicht erkennbar, wie viel Einfluss Optimierungen haben können oder was die einschränkenden Faktoren sind. [sta]

1.3 Bound and Bottleneck Analysemethoden

„Bound and Bottleneck“ Analysemethoden, zu denen auch das Roofline Modell gehört, dienen, wie der Name schon vermuten lässt, der Identifizierung systemkritischer Grenzen, die den erreichbaren Wert einschränken. Im Gegensatz zu statistischen Analysemethoden ist die Ergebnisdarstellung der „Bound and Bottleneck“ Analyse etwas ungenauer. Die genau erreichten Systemwerte sind in den resultierenden Modellen nicht unbedingt erkennbar. Dennoch lassen sich wertvolle Erkenntnisse aus eben diesen Modellen ziehen. Das Erkennen eines einschränkenden Faktors (was anhand bloßer Statistiken nicht immer möglich ist) macht die „Bound and Bottleneck“ Analyse für viele Anwendungen interessant. [WWP09]

Ein etwas bekannteres Beispiel der „Bound and Bottleneck“ Analyse ist das amdahlsche Gesetz.

Das amdahlsche Gesetz besagt, dass die Leistungssteigerung durch das Hinzufügen von n Prozessoren nicht so hoch ist wie die n-Fache Leistung eines einzelnen Prozessors. Zudem hat die Zunahme des n-ten Prozessors einen geringeren Effekt als das Hinzufügen des (n-1)-ten Prozessors. Damit trifft das Gesetz auch eine wichtige Aussage über den Leistungsgewinn parallelen Codes. Je nach parallelisierungsgrad des Programms ändert sich der Leistungsgewinn. Dadurch, dass das Wachstum beschränkt ist, lässt sich die maximale Anzahl Prozessoren bestimmen, die einen lohnenden Leistungszuwachs bringt.

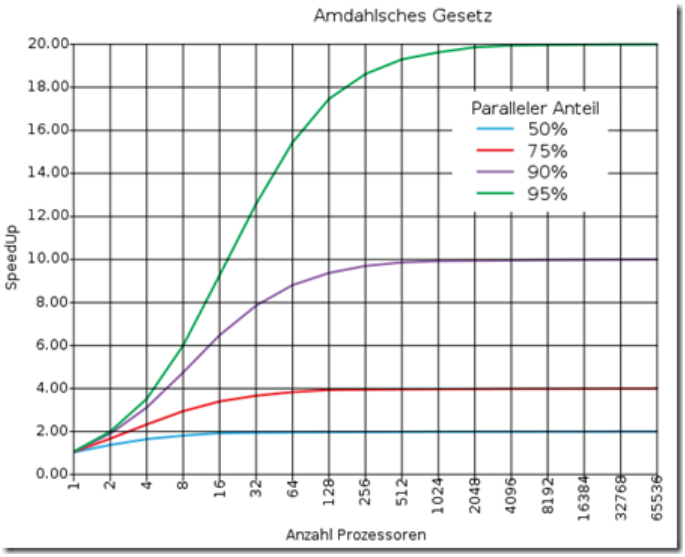

Abbildung 1.1: Laufzeitsteigerungswachstum nach dem Amdahlschen Gesetz [amd]

2 Roofline Modellierung

Grundsätzlich handelt es sich bei der Roofline Modellierung um eine „Bound and Bottleneck“ Analyse. Laut Samuel Williams, Andrew Waterman und David Patterson bietet das Roofline Modell eine leicht verständliche Variante der Performance Modellierung, die Einsichten zur Verbesserung von paralleler Software und Hardware für Floating-Point-Berechnungen bietet [Wil10]. Das lässt die Roofline Modellierung aus anderen Modellierungsverfahren herausstechen.

In den Folgenden Abschnitten werde ich die Funktionsweise der Roofline Modellierung in seiner einfachsten Form erläutern und dann Schritt für Schritt erweitern.

2.1 Messwerte

Zunächst muss definiert werden, welche Eigenschaften eines Systems bei der Analyse betrachtet werden. Das Modell verwendet in der Regel die Eigenschaften Berechnungsarbeit, Kommunikation und „Arithmetic Intensity“.

Die Berechnungsarbeit bezeichnet die gemessenen „Floating-Point-Operationen“ in einer Sekunde (FLOP/s). Da dieser Wert schnell im Milliardenbereich liegt, wird er meistens in GFLOP/s (10^9 FLOP/s) angegeben.

Die Kommunikation bezeichnet die Anzahl der transferierten Bytes (bzw. GB) zwischen Speicher und CPU innerhalb einer Sekunde.

Bei der „Arithmetic Intensity“ handelt es sich um eine zusammengesetzte Eigenschaft, die das Datenaufkommen zwischen CPU und Speicher angibt. Berechnen lässt sich die „Arithmetic Intensity“ durch Division von Berechnungsarbeit und Kommunikation [Wil10]:

$$Arithmetic\ Intensity = \frac{Berechnungsarbeit}{Kommunikation} \tag{2.1}$$

In manchen Arbeiten wird der Wert der „Arithmetic Intensity“ durch „Operational Intensity“ ersetzt. Der Unterschied besteht darin, dass „Operational Intensity“ das Datenaufkommen zwischen Cache und DRAM (anstatt zwischen CPU und Speicher im Allgemeinen) betrachtet. Das führt dazu, dass Speicheroptimierungen eines Computers mit in die Modellierung miteinbezogen werden können. [WWP09]

Im Folgenden werde ich den Term „Arithmetic Intensity“ mit der zuvor gegebenen Definition verwenden.

2.2 Grundlegende Funktionsweise

Der Roofline Modellierung liegt die Annahme zu Grunde, dass die Datenübertragungsrate oftmals die Ressource ist, die zur Einschränkung der Performance führt. Damit hätte man auch bereits die erste Performance Grenze, an die es sich anzunähern gilt, festgelegt.

Als zweite Grenze wird dementsprechend die maximal erreichbare CPU Performance einer Architektur definiert. Die x-Achse bildet die „Arithmetic Intensity“ ab, während auf der y-Achse die erreichbare Performance abgebildet wird. Sowohl x-Achse als auch y-Achse sind logarithmisch skaliert.

Da die maximale CPU Performance nur bei voller Auslastung des Systems erreicht werden kann und sich bei variierender „Arithmetic Intensity“ nicht verändert, lässt sich die Grenze als konstante Funktion in den Roofline Graphen eintragen. Die limitierende Datenübertragungsrate wird mit der „Arithmetic Intensity“ multipliziert, wodurch sich eine linear steigende Funktion ergibt. Daraus folgt für die maximal erreichbare Performance [Wil10]:

$$max.\,erreichbare\,Performance = min \begin{cases} max.\,Floating\,Point\,Performance \\ max.\,Datenübertragungsrate \cdot Arithmetic\,Intensity \end{cases} \tag{2.2}$$

Daraus lassen sich schon die ersten Erkenntnisse über die Limitierungen eines Systems ziehen. Das System heißt „bandwidth bound“, wenn die maximale CPU Performance nicht genutzt werden kann, da die Datenübertragungsrate der limitierende Faktor ist. Andernfalls heißt das System „compute bound“. [WWP09]

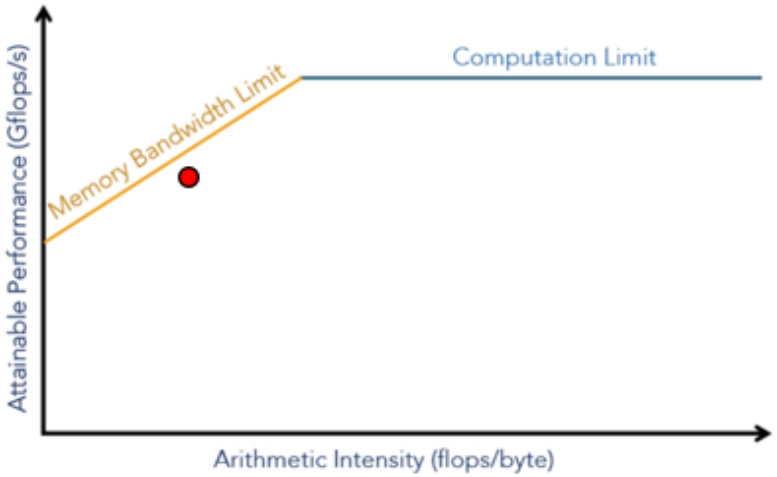

Abbildung 2.1: Einfachster Roofline Graph mit gemessener Leistung eines beliebigen Beispielsystems (rot) [Opt]

Grafisch lassen sich die Eigenschaften eines Systems leicht ablesen (2.1). Liegt die gemessene Leistung links vom Schnittpunkt der limitierenden Geraden, so spricht man von „bandwidth bound“, sonst von „compute bound“. Die ermittelte Leistung des Beispielsystems ist „bandwidth bound“, und hat die Grenze nun schon fast erreicht. Um wieder Platz für mögliche Optimierungen einzuräumen, muss die „Arithmetic Intensity“ erhöht werden. Ist dies für ein System nicht weiter möglich, so markiert dies das Ende der Opti-

mierungen. Gleiches gilt natürlich auch für ein „compute bound“ System, welches seine Grenze schon erreicht hat.

Das setzt das Roofline Modell von anderen Modellierungen ab - es lässt sich absehen wann das Ende der Optimierungen erreicht ist und eventuell mögliche Verbesserungen keinen Einfluss mehr haben.

2.3 Limitierungen der Datenübertragungsrate

Ein wichtiges Konzept der Roofline Modellierung ist das Schichten von unterschiedlichen Begrenzungen in einem Roofline Graphen. Durch die Abstände zwischen dem gemessenen Verhalten und den einzelnen Limitierung lässt sich ablesen, welche Optimierung den gewünschten Effekt haben kann. Im folgenden Abschnitt möchte ich auf einige Möglichkeiten der hinzufügbaren Limitierungen bezüglich der Datenübertragungsrate eingehen.

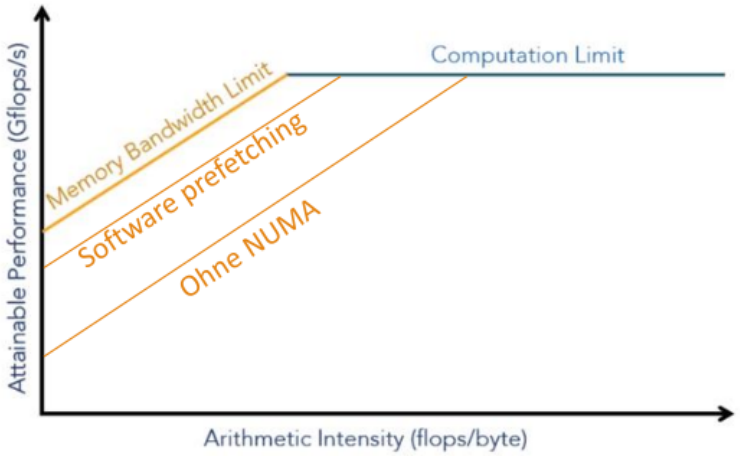

Abbildung 2.2: Roofline Graph mit Limitierungen der Datenübertragungsrate [Opt]

Die Limitierung definieren dabei ein neues Performancemaximum, das nicht überschritten werden kann, wenn die entsprechende Optimierung nicht stattgefunden hat. Daran lässt sich auch erkennen, welche Optimierung am sinnvollsten wäre. Angelehnt an das obige Beispiel hilft es nicht, das „Software Prefetching“ eines Systems zu optimieren, wenn deutlich zu erkennen ist, dass der eigentlich limitierende Faktor an anderer Stelle liegt.

Beim Optimieren der Datenübertragungsrate sind die Ziele die benötigten Daten einerseits möglichst schnell zu laden, aber andererseits das fälschliche Laden von unnötigen Daten zu vermeiden. „Non-Uniform-Memory-Access (NUMA)“ ist eine Methode die Zugriffszeiten auf einzelne Speichereinheiten zu beschleunigen und damit die Performance zu erhöhen. In „UMA-Systemen (Uniform-Memory-Access)“ sind die Zugriffszeiten im Idealfall auf jede Speichereinheit gleich lang, ungeachtet deren Lokalität. Hier besitzt je-

de Prozessoreinheit einen eigenen lokalen Speicher, auf den keine weitere Komponente zugreifen kann.

„NUMA-Systeme“ hingegen besitzen zusätzlich zu den lokalen Speichereinheiten auch einen von mehreren Prozessoreinheiten adressierbaren Speicher. Das führt zu nicht einheitlichen Zugriffszeiten. So hat eine Adressierung im prozessornahen Speicher eine kürzere Zugriffszeit als eine entsprechend weiter entfernte Adressierung. Der so entstehende Performanceunterschied lässt sich dann als Limitierung in den Roofline Graphen einfügen. [numa] [numb]

Speicheradressierung ist gerade in Bereichen wie Hochleistungsrechnen oder der Verarbeitung von großen Datenmengen ein stark einschränkender Faktor. „Prefetching“ ist seit langem eine gängige Lösung um den Performanceverlust zu reduzieren. Dabei wird versucht anhand von erkennbaren Mustern vorherzusagen, welche Daten als nächstes angefragt werden, um diese aus langsameren Speichereinheiten in den deutlich schnelleren Cache zu laden. Dies kann auf Hardwareebene geschehen, z.B. durch das Erkennen von „Strides (Schrittweiten)“. Hier wird nach erfolgreichem Laden von Daten aus einer Adressmenge mit ähnlichem Abstand die möglicherweise nächstangeforderte Adresse berechnet und im Voraus geladen.

Im Falle von „Software Prefetching“ wird das Berechnen der möglicherweise benötigten Daten in die Software oder den Compiler verlegt. Das Analysieren von Schleifen ermöglicht beispielsweise das vorzeitige Laden von wiederholt benötigten Befehlen, die nach einmaligem Laden nicht mehr aus dem Cache entfernt werden. [AJ17]

In den meisten modernen Systemen liegt durch die Verwendung von „Cache Lines“ bereits eine Art „Software Prefetching“ vor. „Cache Lines“ bilden die kleinste, verwaltbare Einheit innerhalb eines Prozessors. Die Länge kann von 64 Bit bis hin zu 128 Bit variieren. Bei einem „Cache Miss“ wird die gesuchte „Cache Line“, falls in einem niedrigeren Level vorhanden, geladen. Man macht sich hier die „Spatial Locality“ eines Systems zu Nutze. Die Grundannahme liegt hier darin, dass im Speicher nah beieinander liegende Adressen eine höhere Chance haben geladen zu werden. Dies ist z.B. bei der Iteration über ein Array der Fall.

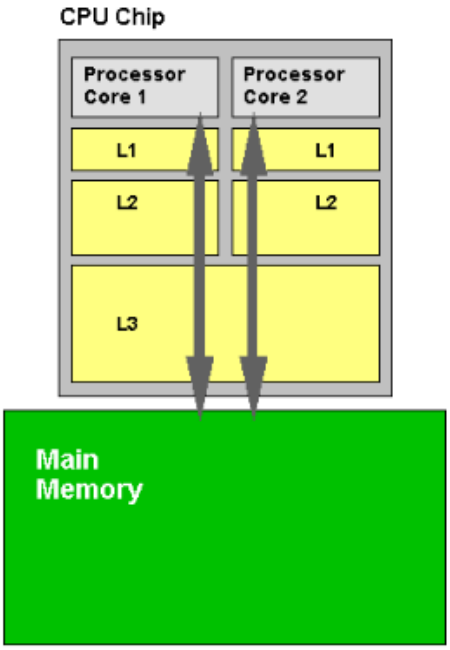

Abbildung 2.3: Modell der drei Cachelevel [caca]

Die Verwendung von „Cache Lines“ bringt allerdings auch Nachteile mit sich. In vielen Fällen wird nicht der gesamte

Datensatz einer „Cache Line“ benötigt - es
werden unnötige Daten geladen, die das Laden wichtiger Ressourcen beeinträchtigen. [cacb]

Parallelisierung ist oft ein effektives Mittel um Systeme zu beschleunigen. Unabhängige Tasks werden gleichzeitig ausgeführt, was die Anzahl der auf Bearbeitung wartenden Tasks dementsprechend verringert. Das Gesetz von Little, die Basis der Warteschleifentheorie, bietet dabei eine Möglichkeit die maximale Anzahl parallel auszuführender Tasks zu bestimmen, sodass noch ein Performancegewinn herausspringen könnte:

$$L = \lambda W \tag{2.3}$$

Das Gesetz besagt, dass die Anzahl der durchschnittlich wartenden Items L (in unserem Fall die Tasks oder Befehle), die sich in der Warteschlange befinden, durch das Produkt aus der durchschnittlichen Wartezeit W und der durchschnittlichen Anzahl der ankommenden Items pro Zeiteinheit λ berechnen lässt. Der optimale Parallelisierungsgrad ergibt sich dann aus der errechneten Anzahl wartender Items. Bei der Optimierung sollte das Gesetz von Little hinzugezogen werden um festzustellen, bis zu welchem Grad sich die Parallelisierung lohnen kann. Schließlich bedeuten mehr parallel ausgeführte Tasks auch eine höhere Belastung der Kommunikation. [al.08]

Im nächsten Abschnitt werde ich noch genauer auf verschiedene Möglichkeiten zur Optimierung durch Parallelisierung eingehen.

2.4 Limitierungen der CPU Performance

Wie bei den Bandbreiten Limitierungen, lässt sich ein Roofline Graph auch durch Begrenzungen parallel zur maximal erreichbaren CPU Performance erweitern. Hier lassen sich alle Funktionen eintragen, die die Anzahl Operationen pro Sekunde direkt beschränken. Insbesondere spielen hier verschiedene Arten von Parallelisierung eine Rolle.

Parallelisierungen lassen sich auf unterschiedlichen Ebenen des Systems anwenden. „Instruction Level Parallelism (ILP)“ bezeichnet die Anzahl der parallel ausführbaren Operationen eines Programms. Dabei wird durch die Berechnung der Abhängigkeiten zwischen den einzelnen Operationen die Anzahl der ausführbaren Operationen pro Zeiteinheit berechnet. Dazu ein Beispiel:

1. e = a + b
2. f = c + d
3. g = e * f

Obiges Beispiel zeigt einen kleinen Beispielalgorithmus [ins]. Die Werte a,b,c und d seien konstante Werte, die nicht zuvor berechnet werden müssen. e und f sind demnach komplett unabhängig voneinander und lassen sich somit parallel berechnen. g hingegen

hängt von den Ergebnissen in e und f ab, muss als sequenziell nach den beiden Additionen berechnet werden. Nimmt man desweiteren an, eine Operation benötigt eine Zeiteinheit zur Berechnung, so werden zwei Zeiteinheiten (eine für die parallele Berechnung von e und f und eine für die Berechnung von g) benötigt um die Operationen des Beispielalgorithmus auszuführen. Damit ergibt sich entsprechend ein ILP-Wert von 3/2 bzw. 1,5.

Im kleinen Rahmen, wie bei dem obigen Beispiel, ist der Performancegewinn an dieser Stelle sicherlich noch nicht besonders groß. Doch man beachte, dass in der Realität solche Berechnung oft in Schleifen stattfinden, sodass der Performancegewinn mit jeder Iteration größer wird. Bei großen Schleifen, ist die Parallelisierung auf „Instruction Level“ nicht zu missachten.

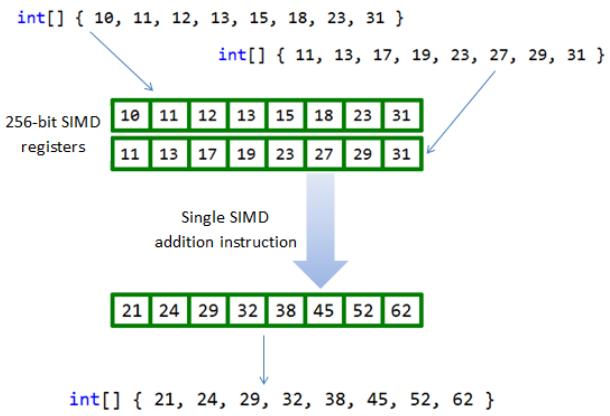

Abbildung 2.4: Single Instruction Multiple Data [sin]

Abbildung 2.4 verdeutlicht die Arbeitsweise von „Single Instruction Multiple Data (SIMD)“. Hier liegt eine Art der Parallelisierung vor, die sich wiederholende Strukturen, auf die dieselbe Operation angewendet werden sollen, zu Nutze macht. So kann beispielsweise bei Iteration über zwei Arrays mit Integern, die paarweise addiert und ausgegeben werden sollen, ein „SIMD-Befehl“ ausreichen um jenes Array zu erzeugen. Man wendet also einen Befehl (Single Instruction) auf einen größeren Datensatz (Multiple Data) an. Das verringert die Anzahl der zu verwendenden Schleifen, was letzten Endes auch wieder die Performance positiv beeinflussen kann. [sin]

Die Anzahl der auszuführenden Befehle lassen sich nicht nur durch das einzelne Anwenden auf ein größeres Datenset verringern, sondern beispielsweise auch durch das Einführen von kombinierten Befehlen, wie „Fused Multiply Add (FMA)“. Eine „FMA-Operation“ führt, wie der Name schon andeutet eine Multiplikation und eine Addition aus. Es hängt vom Hersteller und der Version des „FMA-Befehlssatzes“ ab, nach welchem Schema die einzelnen Werte berechnet werden. [fus]

Das letzte Beispiel, das hier Erwähnung finden soll aber wegen des großen Umfangs nur kurz erläutert wird, ist der „Thread-Level-Parallelism (TLP)“. Ein Thread ist ein Teil eines Programms, der unabhängig vom Hauptprogramm ausgeführt werden kann. Mittels „Multithreading“ lassen sich mehrere Threads gleichzeitig ausführen und somit die gesamte Performance erhöhen. [mul]

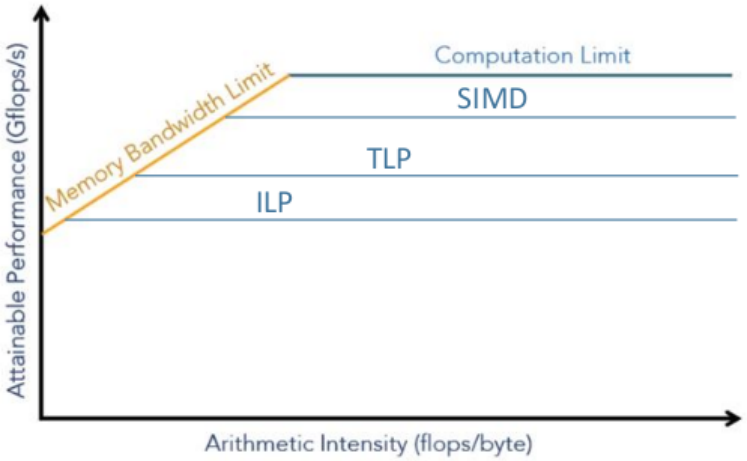

Abbildung 2.5: Roofline Graph mit Limitierungen der CPU Performance [Opt]

2.5 Arithmetic Intensity Walls

In den vorherigen Abschnitten wurde gezeigt, wie sich das Roofline Modell durch Performance- und Bandbreitengrenzen erweitern lässt. In diesem Abschnitt soll noch ein weiterer Faktor hinzugefügt werden, der die Optimierungsmöglichkeiten begrenzt. Allerdings in diesem Fall nicht in der erreichbaren vertikalen Position, sondern in der horizontalen. Die sogenannten „Arithmetic Intensity Walls“ werden durch vertikale Geraden dargestellt. Sie geben an inwiefern sich die „Arithmetic Intensity“ eines Programms noch erweitern lässt. [Wil10]

Per Definition (siehe Abschnitt Messwerte 2.1) hängt die „Arithmetic Intensity“ von der Berechnungsarbeit und der Kommunikation ab. Die Berechnungsarbeit ist in der Regel nur bedingt beeinflussbar, da sich diese aus der Datenmenge und der gewünschten Verarbeitung ergibt. Die Datenmenge zu erhöhen würde zwar die „Arithmetic Intensity“ erhöhen, aber eben mit längerer Laufzeit einhergehen, was nicht wirklich gewünscht ist.

Die nötige Kommunikation zu verringern bringt hier eher die gewünschten Ergebnisse. Da der Cache die schnellste Speichereinheit moderner Rechnerarchitekturen ist, gibt die Cachegröße gleich eine Begrenzung der Kommunikation und damit auch der „Arithmetic Intensity“ an. In der Regel wird allerdings nicht die Cachegröße als „Arithmetic Intensity Wall“ eingeführt, sondern die mit dem Cache verbundenen „Cache Misses“.

Grundsätzlich unterscheidet man drei verschiedene Arten von „Cache Misses“: „Compulsory, Capacity und Conflict“. „Compulsory Misses“ bezeichnen die unausweichlichen Verfehlungen, die beim anfänglichen Aufruf eines leeren Caches auftreten. In einem leeren Cache können die benötigten Daten unmöglich zu finden sein. Einschränken ließe sich dies durch das vorzeitige Laden der benötigten Daten in den Cache. [Dug]

„Capacity Misses“ treten auf, wenn die zu verarbeitende Datenmenge die Cachegröße übersteigt und nicht alle Daten gehalten werden können. Eine Möglichkeit diese einzuschränken, bilden Algorithmen, die gezielt ermitteln welche Daten nicht mehr im Cache

gebraucht werden und entsprechend gelöscht werden können. Die hierzu verwendeten Verfahren wählen die zu löschenden Werte teilweise durch das Analysieren der vorherigen Befehle aus. So lassen sich Schleifen beispielsweise identifizieren, sodass anfänglich bearbeitete Werte entfernt werden können. Teilweise werden die Werte aber auch rein zufällig ausgewählt.

„Conflict Misses“ entstehen dann, wenn ein neu in den Cache geladener Datenblock einen älteren Datenblock oder Teile von diesem überschreibt. Wenn diese überschriebenen Daten nun gefordert werden, kommt es zum „Cache Miss“.

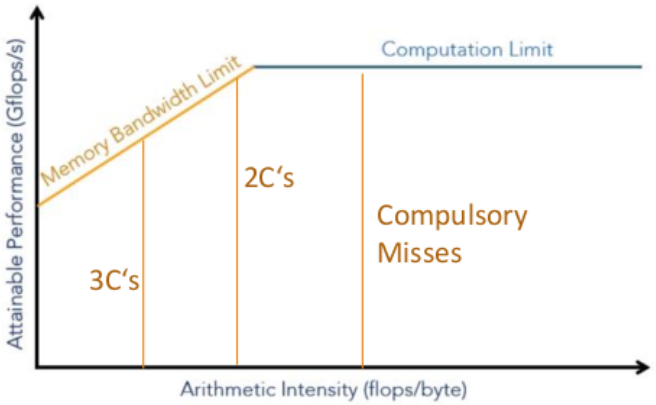

Abbildung 2.6: Roofline Graph mit „Arithmetic Intensity Walls“ [Opt]

Im Roofline Modell werden die drei daraus entstehenden „Arithmetic Intensity Walls“ teilweise mit 3C´s und 2C´s beschriftet, was die Anzahl der möglichen „Cache Misses“ angibt.

3 Roofline Modellierung in der Praxis

Im folgenden Kapitel wird die Anwendung des Roofline Modells mit dem Intel Advisor anhand zweier einfacher Beispielalgorithmen noch genauer erläutert.

3.1 Workflow

In der Praxis ist oft nicht nur die gewählte Art der Modellierung, sondern auch der Zeitpunkt der Modellierung und Optimierung relevant. Zwar sollten sowohl Optimierung als auch Modellierung fester Bestandteil der Softwareentwicklung sein, sie sollten aber nicht zu früh in den Workflow des Projekts integriert werden. Es lohnt sich nicht mit der Optimierung anzufangen bevor nicht die grundlegende Funktionalität implementiert ist. Das betrifft allerdings nur die nicht trivialen Optimierungen, die man als Entwickler treffen kann - offensichtliche Optimierungen sollten natürlich dennoch eingeführt werden.

Intel stellt einen Beispielworkflow zur Entwicklung mit integrierten Optimierungsschritten zur Verfügung, den ich an dieser Stelle vorstellen möchte (3.1). [wor]

Die ersten beiden Stufen (von oben nach unten) stellen das Projektmanagement, sowie das (erste) Deployment dar. Erst nach dem ersten Deployment wird das Verhalten des Projekts analysiert und modelliert. An dieser Stelle kommt das Roofline Modell zum Einsatz. Schleifen sind oft Faktoren, die großen Einfluss auf die Performance haben. Diese werden ebenfalls analysiert. Aus den bereits gewonnenen Erkenntnissen lässt sich oft bereits die Performance der Anwendung verbessern. Optional kann man die Analyse auch noch ein paar Schritte weitertreiben und neben Schleifen auch den Speicherzugriff und Abhängigkeiten in der Software analysieren. Nach vorgenommenen implementierung kann die Anwendung erneut kompiliert werden. Der Zyklus beginnt erneut.

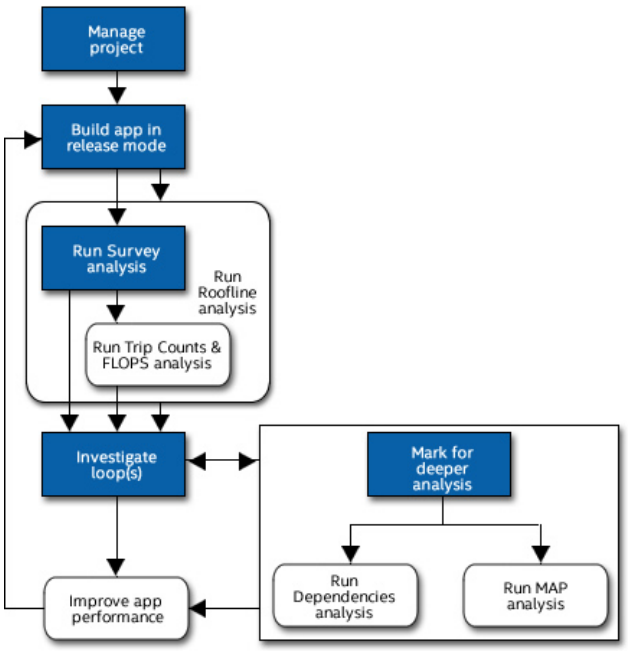

Abbildung 3.1: Beispielworkflow [wor]

3.2 Verwendete Daten

Das Datenset, das verwendet wurde, basiert auf dem Roofline Showcase Programm von Intel [roo] [1]. Die Algorithmen werden in Schleifen mit je 10.000 Iterationen auf Arrays oder Structs der Länge 1328 ausgeführt. Bei den Algorithmen handelt es sich um einfache arithmetische Operationen. Wichtig ist, dass die Zugriffszeiten auf ein Array kürzer sind, als die auf ein Struct.

3.3 Algorithmus mit niedriger Arithmetic Intensity

Der erste Algorithmus mit relativ niedriger „Arithmetic Intensity“ addiert zwei Datensets.

$$X = Ya + Yb \tag{3.1}$$

Im folgenden Roofline Graphen ist die Performance des Algorithmus nach unterschiedlichen Optimierungen dargestellt. Die Optimierungen beinhalten Strukturveränderungen - die Daten liegen als Array, das Structs enthält, vor - sowie die Vektorisierung (und das damit verbundene Nutzen der zugehörigen Befehlssätze) der Daten.

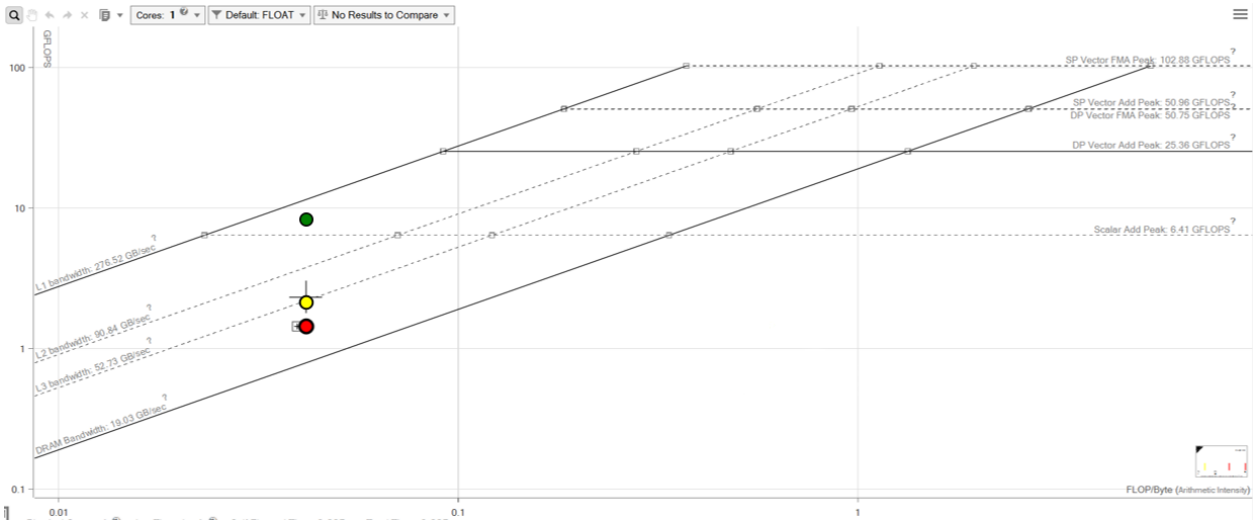

Abbildung 3.2: Ergebnisse der Roofline Analyse mit verschiedenen Optimierungen: AoS (rot), SoA (gelb) und SoA mit Vektorisierung (grün)

Es lässt sich erkennen, dass das Ändern der Struktur von „AoS (Array of Structs)“ zu „SoA (Struct of Arrays)“ bereits einen kleinen Performancegewinn bringt. Die in unmittelbarer Nähe liegenden Grenzen werden aber noch nicht überschritten. Wenn die Datenübertragungsrate an dieser Stelle der limitierende Faktor wäre, so würde eine Optimierung durch Vektorisierung keine weitere Laufzeiterhöhung mehr einbringen. Dem

[1] In dieser Arbeit werden nur zwei der im Showcase beinhalteten Algorithmen besprochen, da diese ausreichen um einen Überblick über die Anwendung des Roofline Modells zu gewinnen.

ist aber nicht so und es gelingt sogar die Limitierung der Skalaraddition zu durchbrechen. [Arr]

An dieser Stelle lässt sich nun feststellen, dass kaum noch Luft für Optimierungen übrig ist. Das Limit der Datenübertragungsrate ist fast erreicht. Um Platz für neue Optimierungen zu schaffen, müsste nun die „Arithmetic Intensity“ erhöht werden. Lässt sich die „Arithmetic Intensity“ nicht erhöhen, könnte man versuchen durch weitere Optimierungen, die Performance bis ganz ans Limit zu treiben. Ansonsten markiert dies das Ende der Optimierungen.

3.4 Algorithmus mit hoher Arithmetic Intensity

Der zweite Algorithmus verwendet wieder dieselben Datensets, addiert diese nun aber mehrfach nach folgendem Schema:

$$X = Ya + Ya + Yb + Yb + Yb \tag{3.2}$$

Die Optimierungen, die hier vorgenommen wurden, beinhalten wieder das Ändern der Struktur wie im vorherigen Abschnitt und die Vektorisierung. Neu hinzu kommt der Einsatz von „Fused Multiply Add Befehlssätzen“.

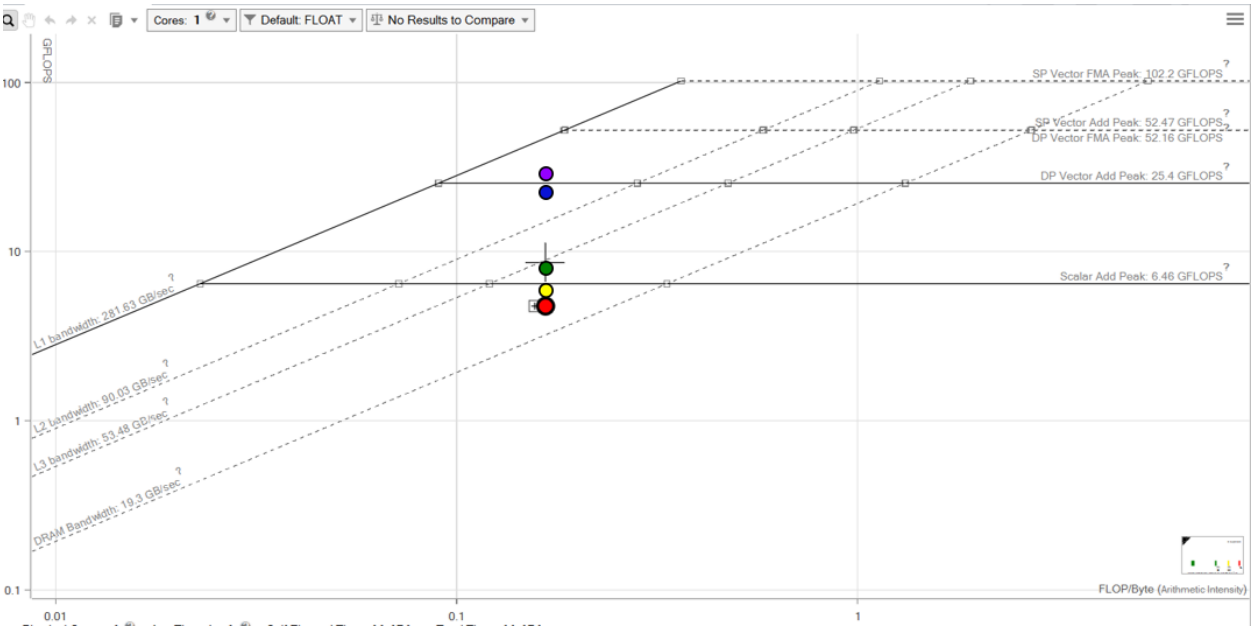

Abbildung 3.3: Ergebnisse der zweiten Roofline Analyse mit verschiedenen Optimierungen: AoS (rot), SoA (gelb) und SoA mit Vektorisierung (grün), AoS mit Vektorisierung (blau), SoA mit Vektorisierung und FMA (lila)

Leicht erkennbar ist, dass die höhere Arithmetische Intensität mehr Spielraum für Optimierungen lässt. Ähnlich wie beim vorherigen Algorithmus bringt auch hier die alleinige Strukturänderung kaum Laufzeitgewinn. Beschränkt wird die Performance deutlich durch die Verwendung von Skalaraddition. Das Vektorisieren bringt einen deutlichen

Performancegewinn, insbesondere bei den Daten, die als „SoA“ vorliegen. Die Performance der als „AoS“ vorliegenden Daten hingegen wird durch die Datenübertragungsrate des dritten Cache Levels begrenzt. Das Hinzufügen von „Fused Multiply Add Befehlssätzen“ bewirkt bei den vektorisierten Daten noch einen weiteren Perfomancezuwachs.

An dieser Stelle ist nun das Limit der Optimierungen fast erreicht, sodass die Optimierung zu einem Abschluss kommen kann oder die „Arithmetic Intensity“ erhöht werden müsste.

4 Abschließend

In den letzten Kapiteln wurden die Grundlagen des Roofline Modells genauestens erläutert. Das Roofline Modell bietet eine gute Möglichkeit um Erkenntnisse über sowohl das Laufzeitverhalten von Programmen zu erlangen als auch mögliche einschränkende Ressourcen zu erkennen.

Die hier vorgestellte Interpretation des Roofline Modells ist nur eine Möglichkeit der Anwendung. Dadurch, dass die Basis der Roofline Modellierung recht simpel ist, lässt sich das Modell an nahezu jede Situation anpassen. So lassen sich beispielsweise jederzeit neue Grenzen auf einer der Achsen hinzufügen oder andere entfernen.

Auch lassen sich die Betrachtungen der einzelnen Parameter anpassen. So lässt sich beispielsweise anstatt der Analyse des Datenaustauschs zwischen CPU und Speicher, gezielt die Kommunikation zwischen Cache und Speicher analysieren.

Das Roofline Modell bietet eine rundum nützliche und flexible Möglichkeit Systeme zu optimieren.

Literaturverzeichnis

[AJ17] AINSWORTH, Sam ; JONES, Timothy M.: Software prefetching for indirect memory accesses. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (2017)

[al.08] *Kapitel* Little's Law. In: AL., John D. C. L.: *Building Intuition*. Springer US, 2008

[amd] *SharePoint Mathematik - Amdahlsches Gesetz*. http://blog.trivadis.com/b/collaboration/archive/2014/07/04/sharepoint-mathematik-amdahlsches-gesetz-performance-performance.aspx, . – Online; Letzter Zugriff: 14.02.2019

[Arr] *Difference between Array and Structure*. https://techdifferences.com/difference-between-array-and-structure.html, . – Online; Letzter Zugriff: 14.02.2019

[caca] *cache*. https://www.pcmag.com/encyclopedia/term/39177/cache, . – Online; Letzter Zugriff: 14.02.2019

[cacb] *Why software developers should care about CPU caches*. https://medium.com/software-design/why-software-developers-should-care-about-cpu-caches-8da04355bb8a, . – Online; Letzter Zugriff: 14.02.2019

[Dug] DUGAN, Ben: *Concerning Caches*. https://courses.cs.washington.edu/courses/cse378/02sp/, . – Online; Letzter Zugriff: 14.02.2019

[fus] *Floating Point: How does Fused Multiply-Add (FMA) work and what is its importance in computing?* https://www.quora.com/Floating-Point-How-does-Fused-Multiply-Add-FMA-work-and-what-is-its- . – Online; Letzter Zugriff: 14.02.2019

[ins] *Instruction Level Parallelism*. https://www.scribd.com/doc/33700101/Instruction-Level-Parallelism#scribd, . – Online; Letzter Zugriff: 14.02.2019

[LWS+14] LO, Yu J. ; WILLIAMS, Samuel ; STRAALEN, Brian V. ; LIGOCKI, Terry J. ; CORDERY, Matthew J. ; WRIGHT, Nicholas J. ; HALL, Mary W. ; OLIKER, Leonid: Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 2014

[mul] *Multithreading*. https://www.itwissen.info/Multithreading-multithreading.html, . – Online; Letzter Zugriff: 14.02.2019

[numa] *NUMA*. https://www.pcmag.com/encyclopedia/term/48164/numa, . – Online; Letzter Zugriff: 14.02.2019

[numb] *NUMA (non-uniform memory access)*. https://whatis.techtarget.com/definition/NUMA-non-uniform-memory-access, . – Online; Letzter Zugriff: 14.02.2019

[Opt] *Optimizing Application Performance with Roofline Analysis*. https://www.codeproject.com/Articles/1191905/Optimizing-Application-Performance-with-Roofline, . – Online; Letzter Zugriff: 14.02.2019

[roo] *Training Sample: Intel Advisor Roofline*. https://software.intel.com/en-us/articles/training-sample-intel-advisor-roofline, . – Online; Letzter Zugriff: 15.02.2019

[sin] *Parallelism on a Single Core - SIMD with C*. https://instil.co/2016/03/21/parallelism-on-a-single-core-simd-with-c/, . – Online; Letzter Zugriff: 14.02.2019

[sta] *statistical analysis*. https://whatis.techtarget.com/definition/statistical-analysis, . – Online; Letzter Zugriff: 14.02.2019

[Wil10] *Kapitel* The Roofline Model. In: WILLIAMS, Samuel W.: *Performance Tuning of Scientific Applications*. CRC Press, 2010

[wor] *Vectorization Workflow Diagram*. https://software.intel.com/en-us/advisor-user-guide-vectorization-workflow-diagram, . – Online; Letzter Zugriff: 14.02.2019

[WWP09] WILLIAMS, Samuel W. ; WATERMAN, Andrew ; PATTERSON, David: Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. In: *Communications of the ACM - A Direct Path to Dependable Software* 52 (2009)