Computing Tools for HPDA : a Cache-Oblivious and SIMD Approach

Kenny Peou

université PARIS-SACLAY

Computing Tools for HPDA - A Cache-Oblivious and SIMD Approach

Outils de Calculs Pour le HPDA: Approche Cache-Oblivious et SIMD

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Graduate School : Informatique et sciences du numérique, Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire interdisciplinaire des sciences du numérique** (Université Paris-Saclay, CNRS), sous la direction de **Joel FALCOU**, Maître de Conférences HDR, Université Paris-Saclay, et la co-supervision de **Guillaume QUINTIN**, CEO Agenium Scale

Thèse soutenue à Paris-Saclay, le 06 Décembre 2021, par

Kenny PEOU

Composition du jury

Sylvain CONCHON Professeur, Université Paris-Saclay (Toccata)	Président
Lionel LACASSAGNE Professeur, Sorbonne Université (LIP6, ALSOC)	Rapporteur & Examinateur
Claude TADONKI Chercheur (HDR), Ecole des Mines de Paris (CRI)	Rapporteur & Examinateur
Sébastien LIMET Professeur, Université d'Orleans (LIFO)	Examinateur
Fabienne JÉZÉQUEL Maitre de conférences, Sorbonne (LIP6, PEQUAN)	Examinatrice
Joel FALCOU Maitre de conférences HDR, Université Paris-Saclay	Directeur de thèse

THESE DE DOCTORAT
NNT : 2021UPASG105

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Outils de Calculs Pour le HPDA: Approche Cache-Oblivious et SIMD
Mots-clés: Precision Numerique, Vectorisation, Calcul Haute Performance

Résumé: Ce travail présente trois contributions aux domaines de la vectorisation des CPU et de l'apprentissage automatique.

La première contribution est un algorithme pour calculer une moyenne avec des valeurs en virgule flottante de demi-précision. Dans ce travail réalisé avec un support matériel de demi-précision limité, nous utilisons une bibliothèque logicielle existante pour émuler le calcul de demi-précision. Cela nous permet de comparer la précision numérique de notre algorithme à celle de divers algorithmes couramment utilisés. Enfin, nous effectuons des tests de performance d'exécution en utilisant des valeurs à virgule flottante simples et doubles afin d'anticiper les gains potentiels de l'application de la vectorisation du CPU aux valeurs de demi-précision. Dans l'ensemble, nous constatons que notre algorithme présente des performances numériques légèrement inférieures dans le meilleur des cas en échange de performances numériques nettement supérieures dans le pire des cas, tout en offrant des performances d'exécution similaires à celles d'autres algorithmes.

La deuxième contribution est une bibliothèque de calcul en virgule fixe conçue spécifiquement pour la vectorisation du CPU. Les bibliothèques existantes ne reposent pas sur l'auto-vectorisation du compilateur, qui ne parvient pas à vectoriser les opérations arithmétiques de multiplication et de division. De plus, ces deux opérations nécessitent des opérations de cast qui réduisent la vectorisabilité et ont un réel coût de calcul. Pour remédier à ce problème, nous présentons un format de stockage de données en virgule fixe qui ne nécessite aucune opération de cast pour effectuer des opérations arithmétiques. De plus, nous présentons un certain nombre de benchmarks comparant notre implémentation aux bibliothèques existantes et nous présentons la vitesse de vectorisation du CPU sur un certain nombre d'architectures. Dans l'ensemble, nous constatons que notre format en virgule fixe permet des performances d'exécution égales ou supérieures à toutes les bibliothèques comparées.

La dernière contribution est un moteur d'inférence de réseau neuronal conçu pour réaliser des expériences en variant les types de données numériques utilisées dans le calcul d'inférence. Ce moteur d'inférence permet un contrôle spécifique à la couche des types de données utilisés pour effectuer l'inférence. Nous utilisons ce niveau de contrôle pour réaliser des expériences visant à déterminer l'agressivité avec laquelle il est possible de réduire la précision numérique utilisée dans l'inférence du réseau neuronal PVANet. Au final, nous déterminons qu'une combinaison des types de données standardisés float16 et bfloat16 est suffisante pour l'ensemble de l'inférence.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Computing Tools for HPDA - A Cache-Oblivious and SIMD Approach

Keywords: Numerical Precision, Vectorization, High Performance Computing

Abstract: This work presents three contributions to the fields of CPU vectorization and machine learning.

The first contribution is an algorithm for computing an average with half precision floating point values. In this work performed with limited half precision hardware support, we use an existing software library to emulate half precision computation. This allows us to compare the numerical precision of our algorithm to various commonly used algorithms. Finally, we perform runtime performance benchmarks using single and double floating point values in order to anticipate the potential gains from applying CPU vectorization to half precision values. Overall, we find that our algorithm has slightly worse best-case numerical performance in exchange for significantly better worst-case numerical performance, all while providing similar runtime performance to other algorithms.

The second contribution is a fixed-point computational library designed specifically for CPU vectorization. Existing libraries fail rely on compiler auto-vectorization, which fail to vectorize arithmetic multiplication and division operations. In addition, these two operations require cast operations which reduce vectorizability and have a real computational cost. To allevieate this, we present a fixed-point data storage format that does not require any cast operations to perform arithmetic operations. In addition, we present a number of benchmarks comparing our implementation to existing libraries and present the CPU vectorization speedup on a number of architectures. Overall, we find that our fixed-point format allows runtime performance equal to or better than all compared libraries.

The final contribution is a neural network inference engine designed to perform experiments varying the numerical datatypes used in the inference computation. This inference engine allows layer-specific control of which data types are used to perform inference. We use this level of control to perform experiments to determine how aggressively it is possible to reduce the numerical precision used in inferring the PVANet neural network. In the end, we determine that a combination of the standardized float16 and bfloat16 data types is sufficient for the entire inference.

Sommaire en Francais

Contexte global Les progrès actuels de l'apprentissage automatique stimulent simultanément le développement des logiciels et du matériel. Les exigences de calcul intenses d'applications telles que les réseaux neuronaux artificiels nécessitent des logiciels efficaces qui utilisent pleinement un matériel performant. Ce besoin est si fort que les fabricants de matériel conçoivent de nouvelles machines spécifiquement pour ces applications. Ce faisant, les fabricants de matériel et les chercheurs en apprentissage automatique explorent également les limites des représentations numériques acceptables.

Les réseaux neuronaux artificiels (ANN) constituent une catégorie d'algorithmes d'apprentissage automatique faisant l'objet de recherches actives. Ils sont utilisés pour des applications allant des jeux de hasard[10][101], de la détection et de la reconnaissance d'objets[46], du traitement du langage naturel[16], au diagnostic médical[74] et plus encore.

Le domaine de l'apprentissage automatique évolue, tout comme les logiciels correspondants. Toutefois, il ne suffit pas que les bibliothèques logicielles d'apprentissage automatique intègrent de nouvelles méthodes et techniques ; elles cherchent également à accélérer continuellement leurs performances.

Les algorithmes d'apprentissage automatique sont très exigeants en termes de calcul. Ils ont tendance à impliquer le traitement répété de grands ensembles de données. Par exemple, l'apprentissage de l'ANN PVANet[46] présenté au chapitre 4 implique d'effectuer 170 000 itérations sur un ensemble de données de 450 Mo. Par conséquent, les logiciels d'apprentissage automatique doivent tirer parti de tous les accélérateurs matériels disponibles, tels que la vectorisation du CPU, le traitement multicœur et les unités de traitement graphique (GPU).

Les exigences élevées en matière de calcul des logiciels d'apprentissage automatique entraînent même des développements au niveau du matériel. Outre les accélérateurs standard, des accélérateurs d'IA dédiés sont en cours de développement pour les applications d'apprentissage automatique. Les unités de traitement Tensor (TPU) de Google et les unités de traitement visuel (VPU) Movidius d'Intel, développées spécifiquement pour les applications d'apprentissage automatique, en sont deux exemples notoires. Même avec ce matériel spécialisé, les applications d'apprentissage automatique sont souvent gourmandes en ressources de calcul et donc inadaptées à une utilisation de masse. C'est pourquoi des efforts sont déployés pour réduire la charge de calcul des applications d'apprentissage automatique.

Un moyen de réduire le coût des calculs d'apprentissage automatique est de réduire la précision numérique utilisée pendant les calculs. En termes simples, cela signifie utiliser des nombres plus petits pour effectuer les mêmes calculs. La réduction de la précision présente deux avantages principaux : elle réduit la mémoire nécessaire pour effectuer le calcul et permet d'effectuer davantage de calculs en parallèle via la vectorisation. Ces deux avantages se combinent en une vitesse d'exécution plus rapide. Comme effet secondaire, la réduction de la précision d'une application réduit également la consommation d'énergie, car moins de calculs sont physiquement effectués par le matériel. La réduction de la précision comporte un risque de dégradation des performances. Cependant, il existe des méthodes pour atténuer ce risque[40][110]. En outre, les algorithmes d'apprentissage automatique, notamment les ANN, sont capables de produire des modèles robustes tolérant les imprécisions numériques.

Le matériel des CPU et des GPU commence à prendre en charge de nouveaux formats numériques en raison des progrès de l'apprentissage automatique. Les CPU et les GPU ont commencé à prendre en charge le format de précision demi-IEEEStandard2008[51][79][61]. En outre, il a été annoncé que le format bfloat16[99] serait bientôt pris en charge[27][78][98]. Si ces deux formats sont parmi les plus simples à expérimenter et à mettre en œuvre, ils ne sont pas les seuls à mériter d'être examinés. Certains travaux étudient la viabilité d'autres formats, tels que le format à virgule fixe (fixed point), le format binaire (CourbariauxB16) et le format ternaire (NNTernary). Ces travaux permettront même d'examiner la viabilité des formats à virgule flottante non standard.

Contenu Ce manuscrit sera divisé en deux parties. La première partie, intitulée État de l'art, présente les informations générales de base nécessaires à la compréhension des contributions présentes dans la deuxième partie. Cette deuxième partie, intitulée Contributions, présente les travaux réalisés au cours de cette thèse.

Le chapitre 1 présente les bases de l'architecture des ordinateurs nécessaires à la compréhension des travaux présentés dans les chapitres 5 et 6. Nous commencerons par décrire brièvement comment les nombres sont physiquement représentés à l'intérieur d'un ordinateur. Ensuite, nous montrons le coût

d'exécution de diverses opérations sur ces nombres, puis certaines méthodes qui atténuent ce coût. Nous présentons ensuite l'importance et les vitesses des différents niveaux de mémoire. Enfin, nous décrivons les notions de base du parallélisme et les moyens physiques les plus courants de réaliser le parallélisme.

Le chapitre 2 se concentre sur certaines des bases de l'écriture et de l'optimisation des logiciels. Tout d'abord, nous discutons de la manière de chronométrer les logiciels afin de pouvoir analyser les résultats.

Le chapitre 3 explique en détail comment les nombres sont représentés dans un ordinateur. Nous commençons par la base binaire utilisée pour représenter toutes les valeurs numériques. Nous décrivons ensuite les nombres entiers, les valeurs à virgule fixe et les valeurs à virgule flottante, ainsi que leurs limites. Enfin, nous décrivons les bases de l'approximation numérique des fonctions mathématiques.

Le chapitre 4 présente les réseaux neuronaux artificiels. Nous commençons par une brève description de ce que sont les ANN. Nous distinguons ensuite les étapes de formation et d'inférence de l'utilisation des ANN, en précisant que nous nous concentrons spécifiquement sur l'inférence dans ce travail. Ensuite, nous décrivons les calculs les plus exigeants en termes de puissance de calcul effectués par la plupart des ANN et certaines méthodes permettant d'atténuer cette intensité. Nous décrivons ensuite un moyen de mesurer la précision d'un ANN. Enfin, nous décrivons le réseau ANN de PVANet qui sera examiné au chapitre 7.

Le chapitre 5 propose un algorithme pour calculer la moyenne d'un ensemble de nombres en utilisant le format de demi-précision. Ce travail a été publié précédemment dans [83]. Nous examinons d'abord les algorithmes établis pour le calcul de la moyenne. Au fur et à mesure que nous les examinons, nous notons les insuffisances de chaque algorithme lorsqu'il calcule avec des nombres de demi-précision. Nous proposons ensuite un algorithme qui ne subit pas de perte de précision dramatique avec des nombres de demi-précision. Enfin, nous comparons à la fois la précision et la vitesse d'exécution optimisée des algorithmes examinés.

Le chapitre 6 présente les travaux réalisés pour développer une extension en virgule fixe de la bibliothèque logicielle nsimd SIMD. Nous décrivons d'abord certaines des spécificités du format numérique en virgule fixe. Ensuite, nous comparons quelques bibliothèques logicielles à virgule fixe existantes dans le contexte de notre cas d'utilisation souhaité (les expériences présentées dans le chapitre sec:neural). Nous fournissons une description de la bibliothèque logicielle nsimd dans laquelle ce travail a été intégré. Enfin, nous décrivons le type numérique à virgule fixe implémenté, présentons les algorithmes utilisés pour l'implémenter, et montrons des benchmarks de ses performances.

Le chapitre 7 présente un cadre pour réaliser des expériences avec la précision numérique ANN ainsi que les résultats d'une première expérience utilisant ce cadre. Pour ce faire, nous décrivons nos besoins et examinons quelques bibliothèques logicielles ANN couramment utilisées. Ensuite, nous présentons le moteur d'inférence personnalisé développé dans le but d'effectuer nos expériences. Enfin, nous présentons les résultats d'expériences utilisant ce moteur d'inférence pour faire varier la précision numérique utilisée lors de l'inférence de l'ANN PVANet.

Dans ce résumé, nous décrirons les travaux présentés dans les chapitres 5, 6, et 7.

Moyenne demi-précision Le chapitre 5 propose un algorithme pour calculer la moyenne d'un ensemble de nombres en utilisant le format de demi-précision, puis compare ses performances numériques et d'exécution aux algorithmes existants.

Il existe un certain nombre d'approches pour calculer la moyenne d'un ensemble de nombres. Nous observons d'abord les propriétés numériques d'un certain nombre d'approches, que nous appelons Naive, Kahan et Iterative. En observant la façon dont chacune de ces approches est calculée, nous constatons qu'elles sont sensibles à des problèmes numériques tels que le débordement et les erreurs d'arrondi. Ces problèmes numériques ne sont généralement pas significatifs lorsqu'on utilise des valeurs à virgule flottante de 32 bits, mais ils deviennent beaucoup plus visibles avec des valeurs à virgule flottante de 16 bits. En réponse à cela, nous proposons un nouvel algorithme, appelé moyenne en cascade, qui évite les erreurs de débordement et d'arrondi pour un faible coût de calcul.

En observant expérimentalement les performances numériques (Tableau 5.2), nous confirmons que les approches existantes peuvent échouer pour des tailles d'entrée aussi petites que 100 éléments pour les raisons décrites ci-dessus. En examinant la moyenne de Cascading, nous confirmons également qu'elle ne souffre pas d'erreurs significatives dues au débordement ou aux erreurs d'arrondi. Ses performances numériques peuvent être légèrement moins précises que celles des autres approches pour les petites entrées, mais elles deviennent beaucoup plus précises pour les grandes entrées.

En observant expérimentalement les performances d'exécution (Figure 5.2 et Figure 5.3), nous constatons que les approches Naive et Upcast sont systématiquement les plus rapides car elles effectuent le moins de

calculs. La moyenne en cascade est généralement la deuxième plus rapide, s'exécutant au moins aussi vite que la moyenne de Kahan et toujours plus vite que la moyenne itérative.

Bibliothèque en virgule fixe Le chapitre 6 présente les travaux réalisés pour développer une extension en virgule fixe de la bibliothèque logicielle SIMD nsimd.

Nous définissons un format à virgule fixe en utilisant la notation Qa.b où a est le nombre de bits entiers et b le nombre de bits décimaux. Cela nous permet de définir comment les opérations arithmétiques seront effectuées.

Ensuite, nous expliquons l'utilisation de base de la bibliothèque nsimd et présentons quatre fonctions qui ont dû être ajoutées à celle-ci afin d'implémenter l'extension de précision fixe. Ces fonctions sont : `clz` (compter les zéros de tête), `shlv` (décalage variable à gauche), `shrv` (décalage variable à droite), et `div` (division entière). Chacune de ces fonctions est présentée avec un tableau indiquant quelles architectures de CPU supportent les opérations nécessaires et des benchmarks d'exécution qui confirment les performances en fonction du support matériel.

Nous présentons ensuite l'API, les fonctions fournies par, et le format de données de l'extension de précision fixe. Le plus important de ces éléments est le format des données : au lieu d'utiliser les traditionnels $a+b$ bits pour stocker un nombre Qa.b, nous choisissons d'utiliser $a+2b$ bits. Cela nous permet d'effectuer toutes les opérations intermédiaires en utilisant le même type de données et d'éviter les opérations de cast. Ce format de données offre des avantages significatifs en termes d'accélération SIMD potentielle et de facilité de développement logiciel.

Enfin, nous présentons un certain nombre de fonctions implémentées et comparons leurs performances d'exécution à celles d'autres bibliothèques à précision fixe. Les fonctions présentées sont l'addition, la multiplication, la division, le sinus, le cosinus, la tangente, la racine carrée et la réciproque. Dans tous les cas examinés, sauf un, nos fonctions sont plus rapides que les bibliothèques comparées.

Moteur d'inférence à précision arbitraire Le chapitre 7 présente un cadre permettant de réaliser des expériences avec une précision numérique ANN ainsi que les résultats d'une première expérience utilisant ce cadre.

Tout d'abord, nous présentons notre moteur d'inférence personnalisé. Notre moteur offre deux avantages majeurs par rapport aux autres : la possibilité de supporter des types de données C++ personnalisés et la possibilité de contrôler le type de données utilisé lors du calcul de toutes les couches d'inférence. En raison du nombre de couches présentes dans de nombreux réseaux neuronaux, ce contrôle total peut devenir compliqué à utiliser. C'est pourquoi nous avons également développé des outils de niveau supérieur permettant de définir plus facilement les règles d'utilisation des différents types de données.

Ensuite, nous réalisons des expériences sur le réseau neuronal PVANet. Dans ces expériences, nous choisissons un type de couche et faisons varier son type de données d'inférence tout en maintenant les calculs en virgule flottante 32 bits pour tous les autres types de couches. Nous testons avec tous les formats de virgule flottante inférieurs à 32 bits et avec une variété de formats à précision fixe. Le tableau 7.2 résume les résultats de ces expériences. Nous constatons que l'inférence de PVANet peut être entièrement réduite aux formats à virgule flottante de 16 bits, alors que son inférence ne peut être entièrement réalisée en utilisant notre format à précision fixe.

Contents

Introduction

Global Context Current advancements in machine learning are simultaneously driving development in both software and hardware. The intense computational demands of applications like artificial neural networks drive a need for efficient software that fully utilizes performant hardware. This need is so strong that hardware manufacturers are designing new machines specifically for such applications. In doing so, hardware manufacturers and machine learning researchers also explore limits of acceptable numerical representations.

Machine learning consists of algorithms that allow computers to find patterns in data. One type of machine learning method is cluster analysis, which groups data points into related clusters. For example, Figure 1 shows the application of a meanshift clustering algorithm to distinguish between rows of grains.

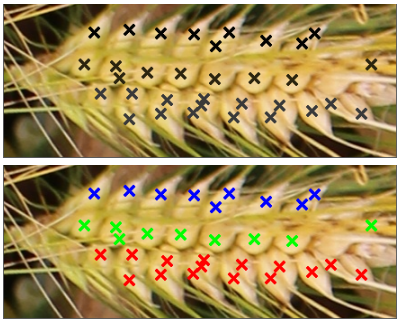

Figure 1: Using meanshift to distinguish between rows of grains

Artificial Neural Networks (ANNs) are one category of machine learning algorithms being actively researched. They are used for applications varying from playing games[10][101], object detection and recognition[46], natural language processing[16], to medical diagnosis[74] and more.

As the field of machine learning evolves, so must the corresponding software. However, it is not enough for machine learning software libraries to only integrate new methods and techniques – they also seek to continuously accelerate their performance.

Machine learning algorithms are very computationally demanding. They tend to involve repeatedly processing large datasets. For example, training the PVANet[46] ANN presented in Chapter 4 involves performing 170,000 iterations over a 450MB dataset. As a result, machine learning software needs to take advantage of any available hardware accelerators, such as CPU vectorization, multicore processing, and Graphical Processing Units (GPUs).

The high computational demands of machine learning software are even driving developments in hardware. In addition to the standard accelerators, there are also dedicated AI accelerators in development for use by ANN applications. Two high-profile examples are Google's Tensor Processing Units (TPUs)[57] and Intel's Movidius Visual Processing Units (VPUs)[55] developed specifically for use by machine learning applications. Even with this specialized hardware, machine learning applications are often computationally intensive and thus unsuitable for mass use. As a result, efforts are being made in order to reduce the computational load of

machine learning applications.

One means of reducing the cost of machine learning computations is to reduce the numerical precision used during the computations. Put simply, this means using smaller numbers to perform the same computations. Reduced precision provides two main benefits – it reduces the memory needed to perform the computation and it allows more computations to be performed in parallel via vectorization. Both of these benefits combine into a faster execution speed. As a side effect, reducing the precision of an application also reduces the energy usage, as less computations are physically performed by the hardware. There is a risk of degraded performance when reducing precision. However, there are methods to mitigate this risk[40][110]. In addition, machine learning algorithms, especially ANNs, are capable of producing robust models tolerant to numerical imprecisions.

CPU and GPU hardware are beginning to support new numerical formats because of advances in machine learning. Both CPUs and GPUs have begun supporting the half[47] precision format[51][79][61]. In addition, there are announcements that there will soon be support for the bfloat16[99] format[27][78][98]. While these two formats are some of the simplest to experiment with and implement, they are not the only formats worth examining. There are works researching the viability of other formats, such as fixed point[66], binary[17], and ternary[69]. This work will even examine the viability of nonstandard floating point formats.

Industrial Context The works presented here were performed while working as a software engineer at Agenium Scale, a small company that specializes in high performance computing, vectorization (both CPU and GPU), and software optimization. It uses this expertise to develop the open-source nsimd software library, which facilitates the writing of vectorized code. nsimd already supports the latest Intel and ARM architectures and is nearly ready to release CUDA support. Agenium Scale leverages its relationships with hardware manufacturers to add support for cutting edge architectures as they are being developed. This nsimd software library provides the basis and the motivation for the fixed point computational library that will be presented in Chapter 6.

Agenium Scale applies its core specialties via consulting to a number of domains including machine learning, aerospace, and agriculture. The works presented in Chapter 7 were motivated and financed by one such consulting contract to research methods for accelerating neural network inference.

The author's responsibilities within Agenium Scale during the duration of this PhD include developing image processing software for the aerospace and agricultural domains, as well as development and maintenance of the fixed point library presented in Chapter 6 and the custom inference engine described in Chapter 7.

Content This manuscript will be divided into two parts. The first part, labeled State of the Art, presents the general background information required to understand

the contributions present in the second part. This second part, labeled Contributions, presents the works done during this PhD.

Chapter 1 will present the fundamentals of Computer Architecture required to understand the works presented in Chapters 5 and 6. It begins by describing briefly how numbers are physically represented inside of a computer. Next, we show the cost of performing various operations on these numbers, followed by some methods that mitigate this cost. The importance and speeds of various memory levels are then presented. Finally, we describe the basic notions of parallelism and the most common physical means of achieving parallelism.

Chapter 2 focuses on some of the basics of writing and optimizing software. First, we discuss how to time software in order to be able to analyze its speed. Then we present how to take advantage of CPU vectorization. This is presented in the context of vectorization software libraries due to the limitations of compiler optimization.

Chapter 3 goes into detail about how numbers are represented within a computer. We begin with the binary base used to represent all numerical values. Next we describe integers, fixed point values, and floating point values, along with their limitations. Finally, we describe the basics of how mathematical functions are approximated numerically.

Chapter 4 presents Artificial Neural Networks. We begin with a brief description of what ANNs are. We then distinguish between the training and inference stages of ANN usage, noting that we focus specifically on inference in this work. Next, we describe the most computationally intensive computations performed by most ANNs and some methods to alleviate this intensiveness. Then we describe one means of measuring the accuracy of an ANN. Finally, we describe the PVANet ANN that will be examined in Chapter 7.

Chapter 5 proposes an algorithm for computing the average of a set of numbers using the half precision format. This work was previously published in [83]. We first examine established algorithms for computing the average. As we examine them, we note each algorithms inadequacies when computing with half precision numbers. We then propose an algorithm that does not suffer dramatic loss of accuracy with half precision numbers. Finally, we compare both the accuracy and the optimized execution speed of the examined algorithms.

Chapter 6 presents the works performed in developing a fixed point extension the the nsimd SIMD software library. We first describe some of the specificities of the fixed point numerical format. Then we compare a few existing fixed point software libraries in the context of our desired use case (the experiments presented in Chapter 7). We provide a description of the nsimd software library within which this work was integrated. Finally, we describe the implemented fixed point numerical type, present the algorithms used to implement it, and show benchmarks of its performance.

Chapter 7 presents a framework for performing experiments with ANN numerical

precision as well as the results of a first experiment using the framework. In order to do so, we describe our needs and examine some commonly used ANN software libraries. Next, we present the custom inference engine developed for the purpose of performing our experiments. Finally, we present the results of experiments using this inference engine to vary the numerical precision used when performing inference of the PVANet ANN.

Part I

State of the Art

1 - Computer Architecture

Modern computer architectures have a number of properties and features which should be considered when designing algorithms. The properties and features of interest for the purposes of this work are instruction latency, instruction ports, pipelining, branch prediction, memory hierarchies, and parallelism.

1.1 . Data Types

Computing hardware supports a certain number of native data types and operations that can be performed on each type. The natively supported data types can be broken into three major categories – binary, integers, and floating point.

Binary types (bits) are the most simple in principle – a binary value is simply a 0 or a 1. In practice, binary values are not stored and treated in isolation, but in groups of at least 8 binary values (1 byte). There are, however, a number of binary operations that can be performed on bytes of binary values, such as AND, NOT, and XOR operations.

Integer types represent integer, or whole, numbers with no decimal or fractional portions. Most programming languages treat 32-bit integers as the default integer type, but hardware usually also supports 8, 16, 32, and 64 -bit integers, each with signed and unsigned support. Integers require at least basic arithmetic operations (addition, subtraction, multiplication, division), but most CPUs also provide some additional complex operations.

Floating point types represent floating point values, as described in the IEEE754 standard[47]. The default floating point type is usually 32-bits, but hardware usually also supports 64-bit (double) floating point values and in recent years support for 16-bit (half) floating point values has grown[79]. Some computer architectures even implement extended precision floating point processing units as described in[47], allowing for greater than 64-bit intermediate computations. Floating point types also require basic arithmetic operations and most CPUs also contain even more complex operations than for integers.

All of these data types will be described in much more detail in Chapter 3.

Some algorithms require data types that are not natively supported by the hardware used to run them. When this happens, the data type must be *emulated* through software, using the native data types in such a way that they emulate the behavior of the target data type. The cost of the extra operations that perform the emulation usually cause the emulated data type to compute more slowly than any native data type.

1.2 . Instruction Sets

CPUs implement operations on data by implementing Instruction Set Architectures (ISA). An ISA defines many important features, such as the supported data types, number and types of registers, and supported operations. Section 1.1 described the data types, Section 1.6 will describe the registers. This section will focus on the types of operations, or instructions.

The most basic instructions include memory usage, arithmetic and logic (bitwise) operations, and control flow. Memory usage instructions allow the ISA to load and store values to and from memory. Arithmetic and logic operations allow the ISA to perform basic computations. Control flow contains such concepts as function calls and branching which help structure the computations. It is also possible for an ISA to provide more complex instructions. These more complex instructions can provide features such as mathematical functions, SIMD instructions, and larger scale memory transfers.

The two major classifications of ISAs are known as Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). RISC ISAs primarily implement the basic instructions and avoid implementing the more complex instructions. They generally require more individual instructions to perform computations. As a result, they tend to rely on compilers to make software development manageable. CISC ISAs implement more than just the basic instructions – they implement some amount of the more complicated instructions. These complex instructions often have higher instruction latency than the more basic instructions. However, they simplify software development by providing a greater range of available instructions. Of the most commonly available ISAs, ARM's are considered to be RISC while Intel's are considered to be CISC.

1.3 . Instruction Latency

CPUs contain a number of basic operations, which are then used together to perform computations. Different CPU operations can require different amounts of clock cycles to execute because the base operations are physically implemented in the circuitry using different algorithms of various complexity. Some algorithms, such as addition/subtraction, are relatively simple and can be calculated in a single CPU cycle. Others, however, can be much more complex and require many more CPU cycles to calculate. Table 1.1 shows the minimum number of CPU cycles needed to perform various basic arithmetic operations[29]. The differences are especially pronounced when performing integer arithmetic – division takes 22x as many cycles as addition – but still significant for floating point arithmetic where a division operation takes slightly more than 3x as many cycles to calculate as addition. Because of this discrepancy in arithmetic calculation time, it is generally faster to multiply than to divide and generally faster to add than to multiply. As will be shown in section 5.3, the effect of arithmetic instruction latency can have

Operation	Integer	Float	SIMD Float
Addition	1	3	3
Subtraction	1	3	3
Multiplication	3	5	5
Division	22	10	10

Table 1.1: Minimum CPU cycles required to perform basic arithmetic operations[29] on an Intel Haswell architecture for 32 bit integers and 32 bit floats.

a noticeable effect on computation time.

Throughput is a concept strongly related to latency. It measures the number of operations that can be performed in a given time. Simplistically, it can be considered the inverse of latency – an arithmetic computational unit with a latency of 0.1 seconds would have a throughput of 10 arithmetic operations per second. However, in practice it this is not always the case. For example, if a processor has three of the above computational units, it actually has a throughput of 30 arithmetic operations per second. This is indeed often the case with CPU ports.

It is also possible to invert the throughput to obtain the concept of inverse throughput, which is usually measured in cycles per instruction. Inverse throughput differs from latency in that it measures the number of operations that can be completed per unit of time, rather than the length of each operation. Continuing the above example with a latency of 0.1 seconds per operation and throughput of 30 operations per second, the inverse throughput would be 0.03 seconds per operation. One example using this type of measure is [49].

1.4 . Ports, Pipelining

Modern CPUs contain a number of computational *ports*[48] capable of performing specific tasks, although most tasks can be performed by multiple ports per CPU. For example, one port may be capable of adding, subtracting and multiplying, while another port may be capable of loading and storing data. Each of these ports is capable of operating independently of the others and passing its output to another port. This allows the CPU to organize a series of instructions such that the data flows through the ports as a *pipeline*, performing multiple instructions per clock cycle[48][93]. Figure 1.2 shows the execution of pipelined instructions. While there is a latency of 5 cycles between the start and end of a block of instructions, the throughput (once the pipeline has filled) is 1 output per cycle.

Pipelining is most useful when performing the same instructions repeatedly on independent data. In this case, it is possible to organize the instructions so that there is a continuous flow of multiple instructions being executed simultaneously. The resulting process calculates much faster than if only a single instruction were

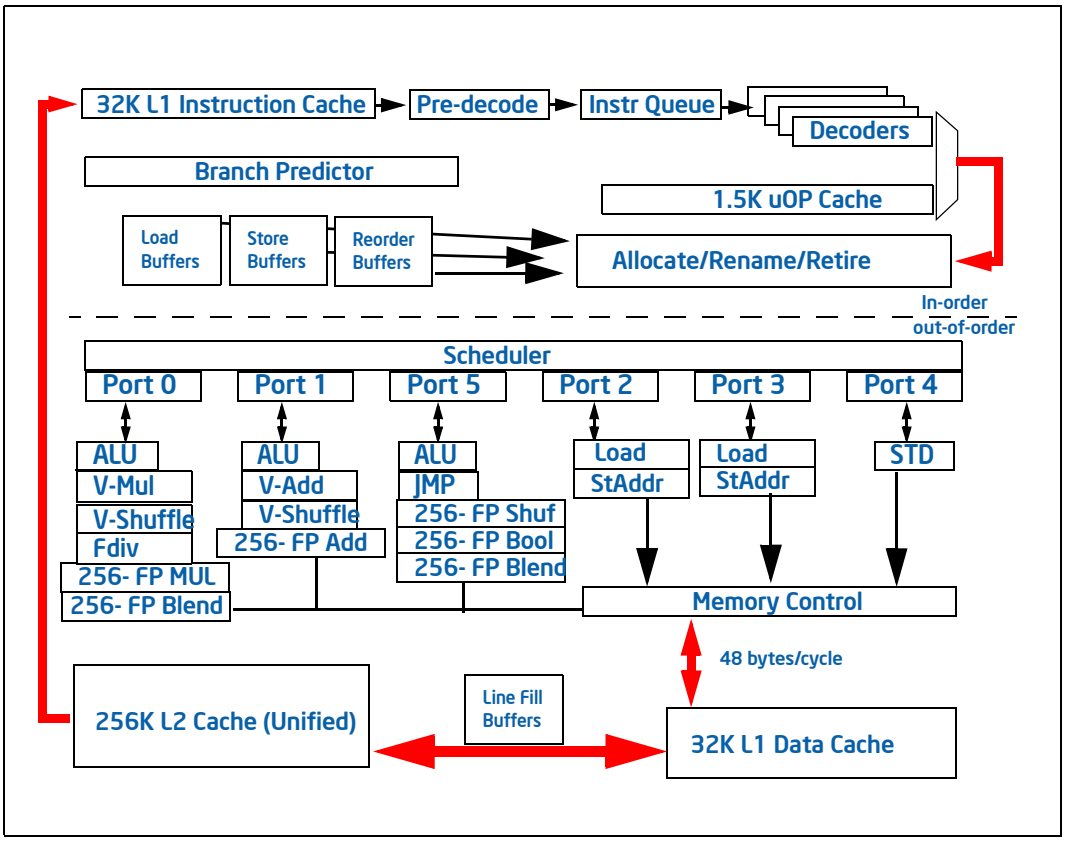

Figure 1.1: Intel Sandy Bridge micro architecture, including ports[48]. While the architecture is not modern, it illustrates well the variety and duplication of operations within CPU ports.

Clock Cycle

Instruction Number	1	2	3	4	5	6	7	8	9	10	11
1	Read	Add	Sub	FMA	Write						
2		Read	Add	Sub	FMA	Write					
3			Read	Add	Sub	FMA	Write				
4				Read	Add	Sub	FMA	Write			
5					Read	Add	Sub	FMA	Write		
6						Read	Add	Sub	FMA	Write	
7							Read	Add	Sub	FMA	Write

Figure 1.2: Example of a simple pipeline in operation. Highlighted section shows full pipeline.

performed per clock cycle. Due to the dependency between steps of the pipeline, it takes time to initially fill the pipeline before the increased throughput can be observed. The same effect can be observed at the end of the pipeline, when there are no more iterations to perform but the last instructions have not yet finished executing[93]. However, these effects grow less influential as the number of repeated iterations grows.

In order to maximize the effectiveness of CPU pipelines, CPUs are capable of reordering operations to perform *out of order* execution[48]. The operations are reordered in order to maximize port usage and length of the pipeline without affecting the computational result.

1.5 . Branch Prediction

Branches are a common code feature where future instructions depend on variable conditions. They occur when using statements such as `if`, `for`, and `while`. When using branches on a pipelining processor, this can create an instruction bottleneck, where we must wait for the condition to evaluate before continuing to execute instructions. The solution to this problem is branch prediction, also known as speculative execution. Rather than wait for the calculation to finish evaluating before proceeding with the next instructions, the processor may choose a branch to execute[48]. If the wrong branch is chosen, the pipeline is flushed out in order to discard the proceeding results, then the correct branch is followed. This costs the same amount of time as if the processor had simply waited for the condition calculation before continuing to execute instructions. This act of waiting for the condition to evaluate is known as a type of pipeline stall. Both pipeline stalls and pipeline flushes lower the processing throughput. If the correct branch is chosen, then the calculation proceeds normally.

Processing units can implement branch prediction in the form of *finite-state machines* that take past branch results as input[93]. A finite-state machine is a mechanism that has a well defined set of possible states, along with well defined transitions between each state. Each transition is a reaction to a form of input. In the case of branch prediction, this input is the result of the most recent branch condition evaluation. The state of the finite-state machine is then used to predict the likely result of the next branch condition evaluation. A simple

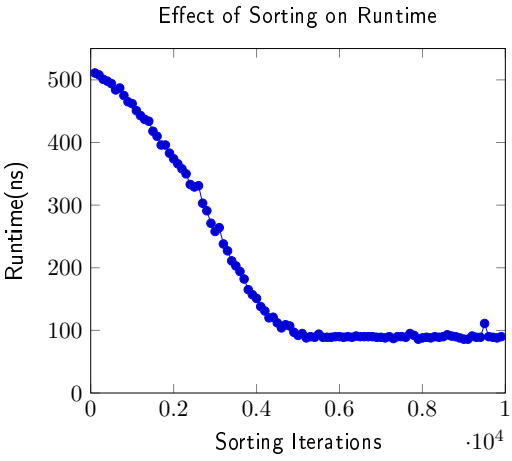

Figure 1.3: Runtime of Algo. 1 depends on how sorted the input is.

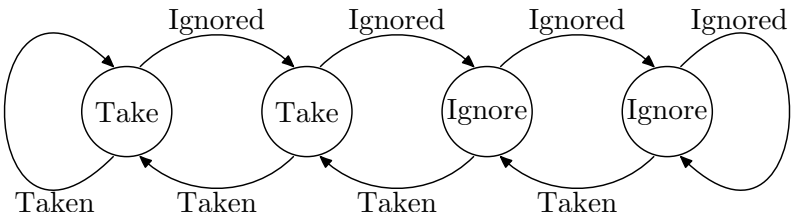

Figure 1.4: A simple finite-state machine for branch prediction.

branch predicting finite-state machine
deciding whether to take or ignore branches is shown in Fig. 1.4.

For code that behaves predictably, branch prediction works very well. One such example is a `for` loop – the loop repeats itself every time it reaches the end of the body, except for the very last iteration. On the other hand, when code behaves unpredictably depending on the data, branch prediction works poorly. One case that exemplifies the extremes is the threshold calculation shown in Algo. 1. This

Algorithm 1 Sample thresholding algorithm.

```
1  for ( int i = 0 ; i < A.size() ; ++i ) {
2    if ( A[i] < 0.5 ) A[i] = 0;
3  }
```

calculation performs differently depending on the nature of the input. If the input is sorted, the calculation is much faster because the results are much more predictable – each input before a certain point is below the threshold and all subsequent points are above it. A simple benchmark of Algo. 1 shows a 7x speedup on a sorted input compared to unsorted.

1.6 . Memory Hierarchy

Algorithm 2 Loop that is likely to cause many cache misses.

```
1 void indirection( int *A, int *B, int *C ) {
2   for ( int i = 0 ; i < A.size() ; ++i ) {
3     A[i] = B[C[i]];
4   }
5 }
```

Memory access is a major limiting factor in computational performance, especially because memory bandwidth does scale up as well as computing power. The solution offered by hardware manufacturers is the presence of multiple levels of memory

Algorithm 3 Cache unfriendly convolution algorithm.

```
void Convolution_unfriendly( int *A, int width,
    int height ) {
  for ( int x = 0; x < width; ++x ) {
    for ( int y = 0; y < height; ++y ) {
      int i = y * width + x;
      A[i] = 10 * A[i];
    }
  }
}
```

Algorithm 4 Cache friendly convolution algorithm – 7x speedup compared to Algo. 3.

```
void Convolution_friendly( int *A, int width, int
    height ) {
  for ( int y = 0; y < height; ++y ) {
    for ( int x = 0; x < width; ++x ) {
      int i = y * width + x;
      A[i] = 10 * A[i];
    }
  }
}
```

Processor	L1	L2	L3
ARM Cortex-A32	8-64KB	128KB-1MB	n/a
ARM Cortex-A55	16-64KB	64-256KB	512KB-4MB
Intel Xeon Phi	32KB	512KB	8MB
AMD Threadripper 3990X	96KB	512KB	128MB

Table 1.2: Cache sizes of some example processors.

hierarchy (Fig. 1.5), with different sizes and access speeds. These levels range from slow hard drives to the relatively fast RAM to the very fast levels of cache. As the memory space of each level diminishes, so does the time to access data in that memory level[48]. In most cases, each of these memory levels draws data from the level directly above it. Generally, the simplest way to accommodate this limitation is to, if possible, design an algorithm in such a way that the input data is a continuous block of data. This is because each level pulls a larger than necessary block of data at each time, in case the rest of the block will also be useful.

If the memory access is regular, then the CPU can easily predict which data blocks will be needed and preemptively pull them from higher memory levels[64][48]. This is known as prefetching. As a result, regular memory accesses are much faster than irregular memory accesses. Conversely, if the memory access is irregular (see Algo. 2), then the CPU will no be able to predict which data block need to be pulled. When the CPU seeks data which is not in the cache, a *cache miss* occurs. Upon a cache miss, the CPU must load the requested data into the lowest level cache, halting progress until the data finishes loading. The effects of cache misses are very noticeable – there is a 7x speedup when using the cache friendly convolution in Algo. 4 instead of the cache unfriendly version in Algo. 3.

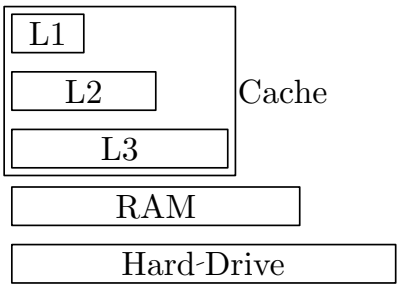

Figure 1.5: Memory hierarchy levels

Registers can be considered as a level of memory hierarchy above the smallest cache. A CPU contains a limited number of registers that it uses for active calculations. This makes registers the fastest memory available to the processing unit. However, the number of registers available is limited. If an algorithm performs calculations that require more immediate values than there are registers, some values will need to be moved to and from the stack, slowing down the calculation speed, especially if this causes a cache misses. There are generally, but not always, enough registers on modern processors to perform most straightforward algorithms. In practice, SIMD registers tend to be more available than scalar registers. This is because, given a same number of each type of registers, some of the scalar

registers will be reserved for certain uses, like function arguments. The amount of registers available, both scalar and SIMD, depends on the CPU implementation. In general, RISC architectures provide more registers than CISC architectures.

1.7 . Parallelism

One way to conceptualize a simple calculation program is to consider it as a single stream of instructions (SI) being performed on a single stream of data (SD). However, it is also possible to perform multiple streams of instructions (MI) or to use multiple streams of data (MD). This lays the foundation of Flynn's taxonomy of computer architectures[28], as shown in Figure 1.6. Multiple streams of instructions operating on a single stream of data (MISD) can allow one to perform multiple unrelated calculations using the same data. Some MISD applications are fault tolerant systems, where the same computation is performed multiple times in order to detect errors, and systolic arrays, which flow a single input through a collection of specialized processors. If we split the input data into multiple streams, we can perform a single stream of instructions on these multiple streams of data (SIMD). This allows us to gain a greater calculation throughput if this single stream of instructions can operate on the multiple streams of data simultaneously. Figures 1.7 and 1.8 showcase the advantage of SIMD computation over SISD. Multiple technologies exist for this purpose – most commonly GPUs, multithreading, and SIMD CPU instruction sets.

	SI	MI
SD	Uniprocessor	Fault Tolerance
MD	Vector CPUs GPUs	Multiprocessing

Figure 1.6: Flynn's taxonomy of computer architectures.

1.7.1 . CPU SIMD

The first attempts at SIMD CPU architectures came in the form of vector computers[73][102]. These supercomputers implemented vector processing instructions, which operate a single instruction sequentially upon a large block (vector) of data. While the operations are not performed in parallel, this architecture can still provide performance gains by eliminating the need to process and decode instructions while processing vectors.

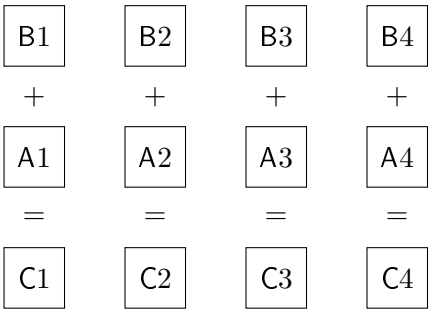

Figure 1.7: Scalar computations.

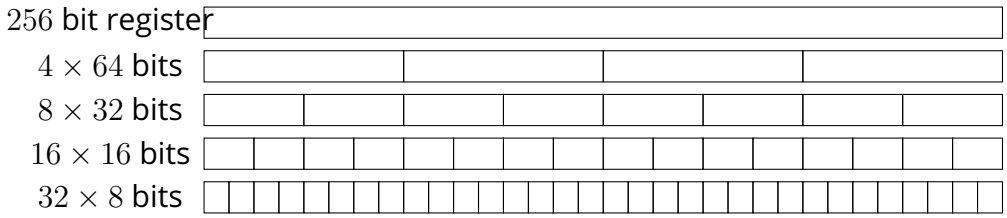

Figure 1.9: Various data types fitting into a 256 bit register.

Another early attempt at SIMD CPU architectures came in the form of parallel processing computers[8]. These computers provided a large number of very simple processors that could execute instructions simultaneously. While this approach failed to take hold at the time, the ideas behind it have returned in both CPU and GPU SIMD architectures.

The current widespread approach to CPU SIMD is to implement large registers along with SIMD instructions that operate on all elements of these registers simultaneously. These SIMD instructions allow the processor to perform operations on all elements within a SIMD register, rather than single pieces of data. These large registers have fixed sizes, regardless of the data type loaded into them. As a result, the amount of values that can be loaded into such a register depends on the data type being processed. Fig. 1.9 shows what types of parallel computation can be performed using a 256 bit SIMD register. For example, a 256 bit register can hold 16 half precision values, 8 single precision values, or 4 double precision values. SIMD instructions act upon all values in the register, so smaller data types are better accelerated by CPU SIMD. Because this acceleration is performed within the CPU, there is no memory transfer to diminish the potential speed gains.

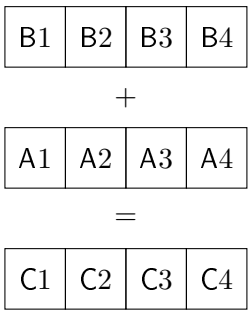

Figure 1.8: SIMD computation.

The act of accelerating a program via SIMD instructions is referred to as both SIMDization and vectorization. If interpreted strictly, vectorization would be limited to the usage of vector computers. However, vectorization has been used with the same meaning as SIMDization for a number of years now[23][20][68].

Each CPU designer provides a different set of instructions to use to take advantage of vectorization[49][6]. This complicates the task of writing code which takes advantage of such instructions, as code written for one CPU may not be portable to another. Fortunately, there are multiple efforts to provide unified APIs which allow developers to write portable code. These efforts will be described in Section 2.2.2, along with more detail about the difference between SIMD instruction

sets offered by different CPU designers.

1.7.2 . Graphical Processing Units

Graphical Processing Units (GPUs) are another means for accelerating computations via SIMD. A GPU consists of a collection of *warps*, which function in a manner analogous to large CPU SIMD registers. All threads within a single warp perform the same instructions simultaneously. Unlike the other approaches mentioned in this section, GPU acceleration is not accomplished as part of a standard CPU – the GPU is physically separate hardware used to add upon the CPU's capabilities. A discrete GPU has its own physical memory and contains hundreds to thousands of simple processing units specialized to perform parallel computations[80], as shown by Figure 1.10. Because the GPU and CPU are separate components with separate memory spaces, any calculation using the GPU requires that the input data be transferred from the CPU to the GPU. For large amounts of data, this transfer can take a significant amount of time.

To allow for the large quantity of processing units, each individual processing unit is less powerful than a CPU core. First, the GPU runs at a slower speed than the CPU – each core performs fewer operations per second. In addition, the GPU cores are not fully independent. They are grouped into clusters such that each cluster of GPU cores performs the same instructions at a given time. This can lead to inefficiencies when using conditional statements, such as `if/else` statements. When a GPU encounters an `if/else`, each branch is entered sequentially. First, the cores that qualify for the `if` perform the `if` branch instructions. Once they have finished, the cores that did not qualify for the `if` perform the `else` instructions. During both of these steps, some of the cores are inactive because the instructions are not intended for them. As a result, GPUs are well suited for larger scale purely SIMD computations. If a computation requires interleaved SISD and SIMD segments, it may not be advantageous to use a GPU.

1.7.3 . Multithreading

Multithreading is the use of multiple "threads" of instructions at once to perform separate tasks. This can be used to perform either SIMD or MIMD computations, and can serve as an extra layer of parallelism on top of vectorization or GPU usage. If each separate task is defined differently, then the program necessarily performs MIMD computations. However, if we define the separate tasks as performing the same instructions on different sections of the same input data, then the resulting program can perform SIMD computations. In this context, it is important to avoid performing the same instructions on overlapping sections of the input data in order to avoid memory conflicts. Depending on the type of memory conflict, it is possible to obtain slower performance or false results.

There are two ways to implement multithreading – using shared memory or distributed memory. Shared memory multithreading requires each thread to have access to the same physical memory. This is generally the case when using multiple

Figure 1.10: Example of an NVIDIA Turing GPU architecture.[80]. Green blocks represent Streaming Multiprocessors (SM) and yellow blocks represent RayTracing cores. Each SM contains 64 CUDA cores for a total of 4606 cores.

threads on a single CPU. In this case, it is possible for each thread passively share data with the others. Distributed memory multithreading has no shared access requirement. It is best suited for cases where the computations are divided between multiple physical processors. In this case, any data sharing between threads requires explicit memory transfers.

Conclusion

This section has described the details of computer hardware necessary to understand the works that will be presented in Section II. We have seen how data types can affect instruction latency and how ports and instruction pipelining can mitigate such latencies. We have also seen the effects of performing computations using various levels of the memory hierarchy and some of the technologies that allow developers to parallelize software. The following section will describe the concepts involved in designing software that takes full advantage of the underlying hardware.

2 - Software & Optimization

This section describes the tools and techniques involved in developing efficient software, such as the works presented in Chapter 6 and Chapter 7.

2.1 . Timing

In order to discus the features that accelerate computation, it is necessary to discuss how benchmark, or measure, the efficiency of the computations. Depending on the complexity of the computation being measured, and the desired level of precision, there are different ways to measure the time taken by a computation.

The most basic unit of time in a processing unit is the clock cycle, which corresponds to a single pulse of electricity emitted by the computer's clock generator. This pulse coordinates the components of the circuit, assuring that they are synchronized. The time taken to perform any operation can thus be measured in clock cycles. A processor running at 3GHz will perform $3 \cdot 10^9$ cycles per second, where each cycle takes $3.33 \cdot 10^{-10}$ seconds. Clock cycles are a natural unit of measure for basic low level operations and can also be useful when performing comparisons between different processors.

More complex computations may take long enough to compute that it makes more sense to measure the time in real world time (seconds, minutes, hours). For example, some mathematical functions such as trigonometric functions, exponents, and square roots can be measured in nanoseconds. More computationally intensive mathematical functions, such as matrix multiplication and image filters are usually measured in milliseconds or seconds. When performing physical simulations, the time to compute tends to range from minutes to hours to days.

There are a few CPU features that complexify the task of benchmarking a computation. The first is the fact that any computation is a task to be performed by an operating system. Unless explicitly configured, the operating system will periodically interrupt the computation in order to perform its base functions, artificially increasing the measured time[100]. Another factor is the presence of CPU C-states[48], which are used to adjust CPU power consumption depending on activity. Depending on which C-state the CPU is in, the CPU may be at reduced voltage (running more slowly). Waking a CPU from a low-voltage C-state to a full-power C-state could also artificially increase the measured time. Fortunately, these perturbations generate a predictable distribution of timings (Fig. 2.1, which can be used to obtain a descriptive benchmark value. Fig. 2.1 clearly shows one main grouping of timings followed by a small after-tail of benchmarks that were slowed down by an OS interruption. In practice, the peak of the largest curve is close to the average of the distribution, so the average of a number of runs can safely be used for simplicity[100].

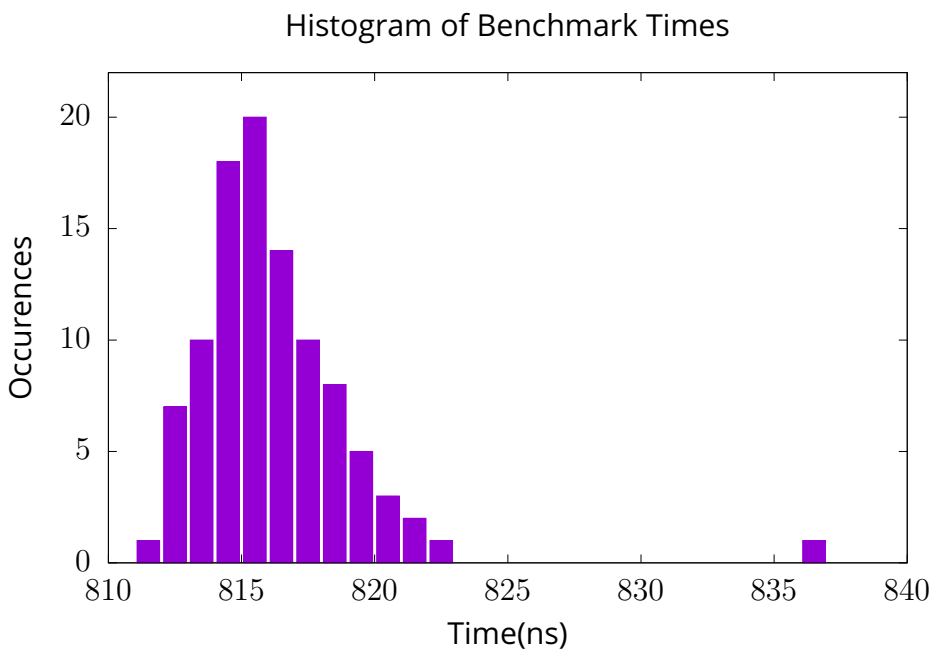

Figure 2.1: Histogram of measured benchmark times(ns) over 100 runs

When optimizing algorithms, the notion of *speedup* is important. The speedup is a metric describing how much an algorithm has been accelerated, as described by

$$\text{speedup} = \frac{t_{\text{old}}}{t_{\text{optimized}}}$$

A larger speedup corresponds to a better optimization. For example, a speedup of two means that an algorithm has been accelerated to run in half the time as it did before.

Benchmark Tools There are a number of tools of varying complexity available for benchmarking programs. If the section of code being benchmarked is well known and well defined, a simple timer (such as those defined in `time.h`, `std::chrono`, or `google-benchmark`[37]) may suffice. More complex tools (such as `perf` and `callgrind`[76]) are needed if one wishes to benchmark within the context of a larger software program. These more complex tools are capable of measure both the absolute time spent in various functions and the proportional time in various functions compared to the rest of the execution time.

Now that we have an idea of how to measure the speeds of algorithms, we can discuss different ways to accelerate them up.

2.2 . SIMD

In very simple cases, modern compilers are capable of automatically vectorizing simple calculation loops to take advantage of CPU SIMD units. These simple cases must be recognized by the compiler to match pre-identified patterns. However, in almost every practical case, vectorized code must be hand-written. Otherwise the vectorization may not be non-portable due to its compiler dependence. This is generally a non-trivial effort for a few reasons. First, the SIMD instructions need to be written directly in *assembly*, or machine language. Because they must be written using CPU intrinsic functions and each CPU architecture provides different SIMD instructions, these vectorized sections of code are not portable. Finally, on some CPUs (x86 architectures) vectorized code operates best on aligned input data, which either complicates the memory allocation process or requires breaking the data into an unaligned head, aligned body, and unaligned tail as shown in Algo. 5.

Algorithm 5 Scalar and vectorized versions of multiplication, including head and tail management.

```
// Scalar
for (int i = 0 ;i < A.size(); ++i) {
  A[i] = B[i] * C[i];
}

// SIMD (Intel)
int align_beg = A.data() + (A.data() % 16);
int align_end = A.data() + A.size()
            - (( A.data() + A.size() ) % 16);
for (int i = 0; i < align_beg; ++i)
  A[i] = B[i] * C[i];
for (int i = align_beg; i<align_end; i+=4)
  __m256 b = _mm256_load_ps( &(B[i]) );
  __m256 c = _mm256_load_ps( &(C[i]) );
  __m256 a = b * c;
  _mm256_store_ps( &(A[i]) , a );
for (int i = align_end; i < A.size(); ++i)
  A[i] = B[i] * C[i];
```

Alignment Some SIMD architectures exhibit different behavior depending on whether the data they operate on is aligned[21]. In the worst case scenario, attempting to load unaligned data into an SIMD register can cause a program to crash. In a less severe scenario, using unaligned data can degrade the performance of a computation. However, these alignment concerns are generally less problematic on modern processors than previous generations.

2.2.1 . Different Architectures

Different CPU architectures can have different SIMD instructions (Algo. 6), even coming from the same manufacturer. Notable among these are Intel's SSE and AVX, ARM's NEON and SVE, and IBM's AltiVec. Each architecture has its own registers, corresponding data types, instructions, and corresponding functions. The result is that SIMD code written for a specific architecture may not work on other architectures. CPUs exhibit backwards compatibility – AVX capable processors also have SSE technology and SVE processors must be NEON compatible – but there is no compatibility between manufacturers. As such, we can consider that the various architectures are organized into families where each member of a family inherits the instructions of its ancestors. For example, in the Intel family of instructions, the AVX512 architecture inherits the instructions from the AVX2 architecture, which inherits instructions from the AVX architecture, and so forth.

Algorithm 6 SIMD addition using different architectures.

```
// AVX
__m256 a, b;
__m256 c = _mm256_add_ps( a , b );

// NEON
float32x4_t a, b;
float32x4_t c = vaddq_f32( a , b );

// VMX
__vector float a, b;
__vector float c = vec_add( a , b );
```

2.2.2 . SIMD Software Libraries

There exist a number of software libraries that facilitate the development of SIMD software. This section will present a number of them, with a focus on CPU SIMD. Table 2.1 summarizes the differentiating features of all presented libraries.

Comparison Criteria

This section explains the criteria being compared in Table 2.1.

General Information The most important column of the general features section is the license. Which license a software library uses affects how it can be included in various projects and what restrictions it imposes on its usage. For example, bsimd has a non-free software license because it is the only paid product

General Information			Instruction Set				Data Type								Features		
Name	Ref	License	AVX-512 512-bit	NEON 128-bit	SVE variable∗	Altivec 128-bit	Float 64	Float 32	Float 16	Integer 64	Integer 32	Integer 16	Integer 8	Fixed Point	Math Func.	C++ Technique	Register Size
OpenMP	[18]	†	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Compiler	Implicit
Eigen	[39]	MPL2	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Op. Overload	Explicit
MIPP	[13]	MIT	Y	Y	N	N	Y	Y	N	Y	Y	Y	Y	N	Y	Op. Overload	Implicit
VCL	[30]	Apache-2	Y	N	N	N	Y	Y	N	Y	Y	Y	Y	N	Y	Op. Overload	Explicit
simdpp	[60]	Boost Software	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	N	N	Exp. Template	Explicit
T-SIMD	[72]	Open-Source	N	Y	N	N	N	Y	N	N	Y	Y	Y	N	N	Op. Overload	Explicit
Vc	[63]	BSD-3-Clause	N	N	N	N	Y	Y	N	Y	Y	Y	N	N	Y	Op. Overload	Implicit
boost.SIMD	[22]	Boost Software	P	N	N	N	Y	Y	N	Y	Y	Y	Y	N	Y	Op. Overload	Implicit
bsimd	[11]	Non-free	P	Y	N	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Exp. Template	Implicit
xsimd	[86]	BSD-3-Clause	Y	Y	N	N	Y	Y	N	Y	Y	N	N	N	Y	Op. Overload	Implicit
nsimd	[91]	MIT	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Op. Overload	Implicit

∗ 128-bit multiples up to 2048
† Implementation dependent

Table 2.1: Comparison of various SIMD wrappers. SSE is not listed as a distinguishing feature as is is supported by all wrappers.

listed. The MIT license is fairly non-restrictive, as it only requires the project integrating it to preserve the library's copyright and license notices. The Apache License 2.0 is similar to the MIT license in terms of restrictions, as it also preserves the library's copyright and license notices. The Boost Software License is also similar to the MIT license in terms of restrictions, as it also preserves the library's copyright and license notices. The BSD 3-Clause is also similar to the MIT license in terms of restrictions, as it also preserves the library's copyright and license notices. t-SIMD's open source software license restricts the software's usage to only scientific or academic purposes.

Instruction Set SIMD instruction sets can be classified first by designer, then by register size.

Intel and AMD design and manufacture x86 CPUs, which implement SSE (128-bit), AVX (256-bit), and AVX-512 (512-bit, Intel only) instructions. Each successive generation also implements the previous ones for backwards compatibility. SSE and AVX are not listed in Table 2.1 because they are supported by all listed software libraries.

ARM proposes the NEON (128-bit) and SVE (variable-bit) instruction sets. SVE is a special case in that it presents a Variable Length Array (VLA) paradigm. SVE registers must be of a size multiple of 128-bits with a maximum of 2048-bits, but the exact multiple is determined by the CPU manufacturer.

IBM designs and manufactures POWER CPUs, which implement AltiVec/VMX/VSX (128-bit) instructions.

Data Type As described in Section 2.4.3, the choice of data type can be an important consideration. Most of the libraries compared in Table 2.1 support the most common data types - 32 and 64 bit float and 8, 16, 32, and 64 bit integers. However, most do not support half precision(16-bit float), which are beginning to be implemented on many CPUs. Currently, x86 CPUs have started implementing

half precision loading and storage of data and conversions between 16-bit and 32-bit floating point types. ARM CPUs have specifications for half precision loading, storage, and arithmetic. Whether these features are actually implemented depends on the manufacturer. Finally, nsimd is the only library to support fixed point data types and computations, which are of interest for this work.

Features For general usage, it is useful for a SIMD wrapper to also contain some mathematical functions. Otherwise the user must implement them using basic arithmetic functions, which can be a difficult task.

The C++ technique column represents one of the technical choices made during design. The compiler directives from OpenMP are the simplest to use and give the most straightforward messages when used, but are not guaranteed to work. The expression template method can be fairly simple to use, but gives the most complicated error messages when failing and can significantly increase compilation time. Lastly, the operator overload method can also be simple enough to use while giving reasonable error messages when failing and does not significantly affect compilation time.

Finally, the Register Size column represents at which point the register sizes are determined. Some wrappers implicitly determine the register size according to the architecture target, while others require the user to explicitly specify the register size when writing the software.

Eigen is a C++ computational library for linear algebra functions. In order to fully take advantage of SIMD optimizations, the library implements and uses an interface that abstracts away the underlying CPU architecture. It is possible to use this interface as a SIMD library, although the user must specify the register sizes that they wish to use. Eigen is one of the few libraries to support half precision via emulation.

OpenMP is an API specification for parallel computation via compiler directives. As such, it is included with most compilers and often requires no extra installation. While OpenMP's focus is on shared-memory multithreading, it also has directives for SIMD computation. The programmer writes a normal computational loop and signals to OpenMP via the `#omp simd` directive that the loop is vectorizable. At compilation, OpenMP will attempt to help the compiler vectorize the loop. However, there is no guarantee of success. In addition, OpenMP is capable of supporting half precision on some architectures that implement it.

MIPP is a C++ 11 library that wraps a portable interface around SIMD intrinsics using operator overloads. It implements the SSE, AVX, AVX-512, and NEON instruction sets for native data types, excluding 16-bit floats. Its API allows the

register sizes to be determined and accounted for at compile time rather than when writing the code.

VCL is a C++ library that wraps a portable interface around SIMD intrinsics using operator overloads. It implements the SSE, AVX, and AVX-512 instruction sets for native data types, excluding 16-bit floats. Its API requires register sizes to be determined and accounted for when writing the code.

simdpp is a C++ library that wraps a portable interface around SIMD intrinsics using template metaprogramming. It implements the SSE, AVX, AVX-512, NEON, and Altivec instruction sets for native data types, excluding 16-bit floats. It supports most instruction sets except for SVE and most data types except for 16-bit floats. Despite using templates, simdpp requires the register sizes to be known when writing the code. In addition, register sizes must be powers of two, which renders SVE support incompatible with the current model.

T-SIMD is a C++ library that wraps a portable interface around SIMD intrinsics using operator overloads. It implements the SSE, AVX, AVX-512, and NEON instruction sets for 32-bit floats and integers 32 bits and smaller. It does not support 64-bit floats, 64-bit integers, or 16-bit floats. In addition, T-SIMD has a license that primarily allows academic uses. This greatly limits its applicability. T-SIMD also requires register sizes to be known when writing the code using it.

Vc is a C++ library that wraps a portable interface around SIMD intrinsics using operator overloads. It implements the SSE and AVX instruction sets for most native data types, excluding 16-bit floats and 8-bit integers. Its API allows the register sizes to be determined and accounted for at compile time rather than when writing the code.

boost.SIMD and bSIMD are intertwined, as bSIMD is the paid extension of the open-source boost.SIMD. They are both C++ libraries that wraps a portable interface around SIMD intrinsics using template metaprogramming. While boost.SIMD only supports SSE, AVX, and (partially) AVX-512, bSIMD has support for SSE, AVX, (partially) AVX-512, NEON, and Altivec. The partial AVX-512 support is limited instructions available to Intel KNL processors. They both support all native data types except for 16-bit floats. Like simdpp, the design of the library requires register sizes to be powers of two, rendering SVE support incompatible with the current model. Their API allows the register sizes to be determined and accounted for at compile time rather than when writing the code.

xsimd is a C++ library that wraps a portable interface around SIMD intrinsics using operator overloads. The mathematical functions provided by xsimd are reimplementations of the algorithms provided by boost.SIMD. xsimd implements the SSE, AVX, AVX-512, and NEON instruction sets for most native data types, excluding 16-bit floats. Its API allows the register sizes to be determined and accounted for at compile time rather than when writing the code.

nsimd is a C++ library that wraps a portable interface around SIMD intrinsics using operator overloads. It implements the SSE, AVX, AVX-512, NEON, SVE, and Altivec instruction sets for all native data types in addition to fixed-point precision. Its API allows the register sizes to be determined and accounted for at compile time rather than when writing the code. In addition to a modern C++ API, nsimd also has C++98 and C APIs.

As can be inferred from the structure of this section and of Table 2.1, nsimd is used as the SIMD wrapper for the work presented in Part II. It will be described in much more detail in Section 6 where it plays an important role.

2.3 . Compilers

Modern compilers are capable of significantly optimizing the codes that they compile. They are capable of optimizations such as reordering code instructions, removing unused code paths, and - most relevantly to this work - auto-vectorizing simple loops. While this auto-vectorization is an impressive accomplishment, compiler auto-vectorization capabilities are frequently overstated. Most compilers are capable of automatically vectorizing simple loops, as show in Algo. 7. However, some studies[58] find that compiler automatic vectorization often achieves results "far from the architectural peak performance"[96] in realistic scenarios. It is relatively simple to design loops that are not overly complicated, but compilers fail to vectorize, such as Algo. 8, which one common compiler – msvc – fails to vectorize, or Algo. 9, which all tested compilers failed to vectorize (as shown in Figs. A.1 - A.8 using [35]). As a result, when vectorizing code it is always

Algorithm 7 Loop that compilers can auto-vectorize.

```
1: function Sum(A, B, C, length)
2:     for i in (0, length) do
3:         C[i] = A[i] + B[i]
4:     end for
5: end function
```

necessary to check if the compiler has successfully auto-vectorized the code and frequently necessary to vectorize by hand when the compiler fails at this task.

Algorithm 8 Simple loop that compilers sometimes fail to auto-vectorize.

```
1: function MulShift(A, B, C, D[i], length)
2:     for i in (0, length) do
3:         C[i] = ( A[i] * B[i] ) » D[i]
4:     end for
5: end function
```

Algorithm 9 Loop that compilers usually fail to auto-vectorize.

```
1: function CheckVals(A, length)
2:     for i in (0, length), i += 2 do
3:         if a[i] * a[i+1] < 10 then
4:             return true
5:         end if
6:     end for
7: end function
```

2.4 . Algorithm Designs

This section lists the most important factors to consider when designing algorithms.

2.4.1 . Complexity

The most important part to optimize is the algorithm being implemented. A well suited unoptimized algorithm can easily perform better than a finely tuned, poorly chosen algorithm. For example, modifying an algorithm that relies on a matrix-matrix multiplication (O(N^3) complexity) to obtain the same results with a matrix-vector multiplication (O(N^2) complexity) can achieve gains greater than any of the methods described in the proceeding sections.

Another algorithmic consideration is the number of operations required to perform the computation. According to Table 1.1, an algorithm that performs 20 integer addition and subtraction operations can still be faster than an algorithm that performs a single integer division operation. Section 6 involves a number of algorithms which attempt to minimize the number of operations involved.

2.4.2 . Memory

Another important consideration is memory usage and memory availability. One of the most important memory limitations is the size of the RAM on the machine. If a computation requires more memory than is available in the RAM, the operating system must begin using a swap partition of the physical storage memory. In the most generous comparison – slow RAM with a fast solid state drive – the RAM is still 10x faster than the SSD. Further, it is can also be useful to consider the size of the various cache levels available on the CPU. Due to the

difference in latencies to access data from various cache levels (Section 1.6), it is often faster to break the data into blocks and perform repeated operations on each block rather than perform repeated operations on the full data[19].

A number of common computations are limited by the available memory bandwidth. The scalar product of two vectors (Algo. 10) is one such example. The computation

Algorithm 10 Scalar product

```
1  float sum = 0;
2  for ( int i = 0 ; i < len ; ++i ) {
3    sum += a[i] * b[i];
4  }
```

Algorithm 11 Matrix multiplication of MxO and OxN matrices.

```
1  for ( int i = 0 ; i < M ; ++i ) {
2    for ( int j = 0 ; j < N ; ++j ) {
3      sum = 0;
4      for ( int k = 0 ; k < O ; ++k ) {
5        sum += a[i][k] * b[k][j];
6      }
7      c[i][j] = sum;
8    }
9  }
```

required to compute the scalar product is very limited compared to the memory usage – it can be reduced to one fused multiply-add (fma) operation for every two load operations. This means that the computation is *memory bound*. Conversely, if a computation is limited by the speed of actual computation, it is *compute bound*. Matrix multiplication (Algo 11) and the Black & Scholes model[77] are popular examples of compute bound computations.

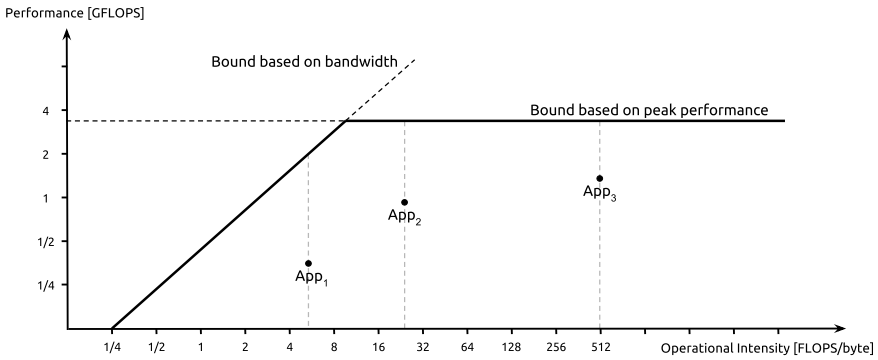

Figure 2.2: Example of a roofline model graph.[34]

This distinction between compute bound and memory bound computations is one of the core notions of the Roofline Model[107] used to predict peak performances of computations on specific machines. In order to perform this predition it is first necessary to compute the arithmetic intensity of the computation. This arithmetic intensity is a ratio of the computational load, referred to as work, to the amount of memory transfers, referred to as memory traffic. Figure 2.2 provides a simple example of a roofline model. If the arithmetic intensity of a computation falls under the sloped portion of the graph, it is a memory bound computation. For example, the scalar product described in Algo. 10 is memory bound because it performs two memory transfers and two arithmetic operations per loop cycle. If the arithmetic intensity of a computation falls under the flat portion of the graph, it is a compute bound computation. The matrix multiplication described in Algo. 11 is compute bound because it can be written to perform many more arithmetic operations than memory transfers, as the data used in the computation is frequently reused.

2.4.3 . Data Types

The choice of data types can also have a strong effect on the performance of a program. It is imperative to choose a data type that has enough numerical precision to calculate correct results. However, it is possible to choose a data type that is too large and slows down the calculations. The size of the data type chosen affects the memory usage of the computations. A program using 16-bit floating point precision will require quadruple the memory of a program using 64-bit floating point precision. This influences the amount of data that can fit into the caches and the RAM, which can affect the speed of the computations. Due to the fixed size of SIMD registers, smaller data types can obtain greater speedups. It is possible to get a theoretical gain of 4x by using half precision instead of double precision for a well vectorized computation. Finally, if using a form of parallelism that requires memory transfers, the reduced data size allows for faster memory transfers – it takes about half the time to transfer half as many bits. The following

section will provide much more detail about the different data types available.

3 - Numerical Representations

This section details the ways in which numbers can be represented on a computer. Different formats will be discussed, along with their respective strengths and weaknesses. Chapters 5, 6, and 7 will present works exploring the effects of various numerical formats applied to certain computations.

3.0.1 . Binary

Most numbers present in everyday life are presented in base 10. This means that we have ten distinct numbers - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 - that we count through before repeating (and incrementing the digit to the left of it). For example, counting past 9 gets 10, counting past 29 gets 30, and counting past 99 gets 100.

However, it is possible to count using any base we want. When counting time, we use base 60 when keeping track of hours (1 hour = 60 minutes) and minutes (1 minute = 60 seconds) and base 24 when keeping track of days (1 day = 24 hours). Here the idea is the same; counting past 59 seconds gets 1 minute, counting past 5 minutes 59 seconds gets 6 minutes, and counting past 59 minutes 59 seconds gets 1 hour.

The last and most important base to present here is base 2. Here we only have two distinct numbers - 0 and 1. Counting from 0 to 9 in binary goes as follows - 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001.

In order to understand how computers handle numbers, it is not base 10 that interests us, nor base 60. It is base 2 that computers use to store numbers. This is because the information is physically stored by transistors, which only have two states - on or off. Therefore, a single transistor is not capable of storing a base 10 digit, as that would require it to have 10 possible states. However, a transistor is capable of storing a digit in base 2, since base 2 only has two numbers available - 0 and 1. This means that it is also possible to arrange a series of transistors so that each transistor in the series represents a base 2 digit of the full number.

In essence, a bit is a single digit of a base 2 number. Since the ideas discussed in the proceeding sections are applicable to both base 2 and base 10, they will be mostly described in base 10.

3.0.2 . Integers

Integers represent whole numbers. They cannot directly represent any decimal values. Numerically, an integer is represented by a set number of digits. This means that there is a set maximum and minimum number. For example, a four digit decimal number can range from 0000 to 9999 and a four bit binary number can range from 0000 (0) to 1111 (15).

In order to be able represent negative numbers in addition to positive numbers, we must devote one digit to the sign. In this case, a four digit decimal number can

represent -999 to +999 and our four bit binary number can represent 1000 (-8) to 0111 (7). In contrast to decimal, binary negative numbers are not simply stored as a negative sign followed by the absolute value. Instead, processors store negative values as the *two's complement* of their corresponding positive values. This two's complement can be obtained by inverting each bit of the initial value, then adding one. As a result, binary numbers retain the property that the bitwise sum of a number and its negative equal zero.

The limited number of bits in an integer result in a potential problem known as *overflow*. When an integer contains the maximum allowable value, adding to this integer causes it to overflow to the minimum value. For example, the maximum four bit unsigned integer is 1111 (15). When adding one the result should be 10000 (16), but because there are only four bits available, the result is truncated to 0000 (0). In the signed integer case, adding one to 0111 (7) would overflow to 1000 (-7).

Due to the indivisibility of integers, division operations can be inexact and vulnerable to truncation errors. The decimal portion of a division result will be truncated, which can lead to significant losses. For example, using integer math the operation $49 \div 10$ returns 4. This is nearly 20% error relative to the mathematical result. However, for operations that do not involve division, the only risk of inaccurate results is from overflow.

3.0.3 . Fixed Point

Next, we want to be able to represent numbers with decimal points, rather than limit ourselves to whole numbers. The simplest way to do this is to set a fixed number of digits before the decimal point and a fixed number of digits after the decimal point. This notation is intuitively called a fixed point representation. Fixed point numbers have many similarities to integers, as integers can be considered a special case of fixed point representation where there are no decimal points. As such, they share the same susceptibility to overflow when exceeding the maximum values. Fixed point values are limited to a predefined number of decimal points, so they are also susceptible to the same types of rounding errors when dividing. Fixed point implementations may perform multiplication and division by computing with more digits than required, then rounding the result. However, this feature is implementation-dependent. Due to the presence of the decimals, there is also an added risk of rounding errors when multiplying. One major drawback to fixed point representation is that it is usually not supported by hardware – floating point

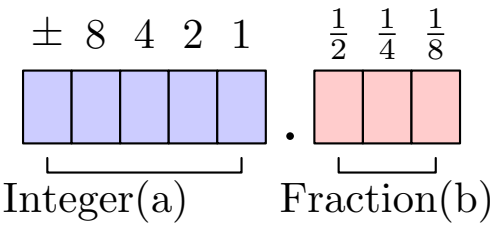

Figure 3.1: Approximate breakdown of a Q5.3 fixed point number.

representation is used instead.

Qa.b Representation This work will describe fixed point formats using the Qa.b representation. In this representation, the 'a' represents the number of bits in the integer portion of the fixed point number. The 'b' represents the number of bits in the decimal portion of the number. For example, the Q5.3 value represented in Figure 3.1 contains 5 integer bits and 3 decimal bits.

Another common representation for fixed point formats is the Qf format, where f is the number of fractional bits. This representation is less precise, as the number of integer bits are not specified.

3.1 . Floating Point Representation

The most common format used for handling numbers containing decimals is the floating point format. This representation is essentially the binary form of normalized scientific notation. This normalized scientific notation consists of a single digit followed by a decimal point and the decimal portion of the value followed by an exponent. For example, 123.456 in normalized scientific notation becomes $1.23456 \cdot 10^2$. In binary, 1101.1 (13.5) becomes $1.1011 \cdot 2^3$. The presence of the exponent allows for a dynamic range of possible values, compared to fixed point representation.

Normalization provides a few benefits. First, it limits each possible value to a single representation. In other words, there is only one way to represent any given value. This simplifies comparisons between floating point values. In addition, normalization maximizes the amount of available digits, which is a desirable property with a limited number of digits. For example, if eight total bits are available in the mantissa, the value 0.0011011 wastes the three bits of leading zeroes. Normalizing the value to 1.1011000 allows room for greater precision. Finally, normalization allows for a memory optimization specific to binary. Binary only allows for values of 0 or 1 and normalization forces the first bit to be nonzero. It is thus possible to make the 1 before the decimal point implicit and avoid storing it.

Floating point representation has more complexities than integer and fixed point representations. The corresponding arithmetic is also more complex, as previously shown in Table 1.1. For example, floating point addition and subtraction take three cycles to evaluate, compared to one cycle for integers. One major reason for this is that floating point values can have different exponent values. The two operands must therefore be aligned in terms of exponents before performing the addition or subtraction. This necessitates shifting (which in binary is equivalent multiplying by a power of two) the lesser value until the exponents are equal.

The IEEE standard[47] details different possible levels of precision for floating point numbers. The most commonly used formats are float (binary32) and double (binary64). These formats use 32 and 64 bits of memory, allocated in the manner

Precision	Sign	Exponent	Mantissa
Half	1	5	10
Single	1	8	23
Double	1	11	52
Quad	1	15	112
bfloat16	1	8	8

Table 3.1: Specifications of IEEE 754-2008 floating point data types and the non-IEEE bfloat16 data type.

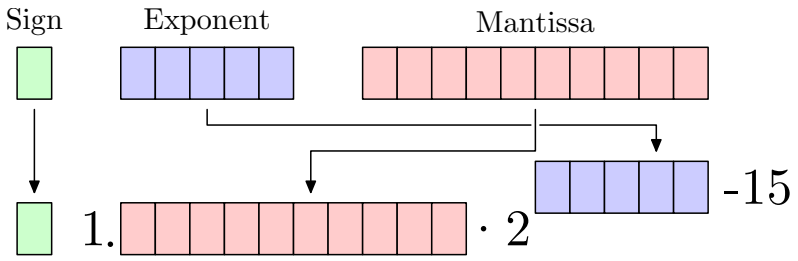

Figure 3.2: Converting a binary16 format floating point value into a numerical value.

indicated in Table 3.1. When performance is valued over precision, there is also the more recent half (binary16) format available which only requires 16 bits of space. However, the limitations of floating point formats, such as overflow and representability, are greatly exaggerated when using half precision.

Floating Point Format A floating point value consists of the three parts shown in Fig. 3.2 – a sign bit, an exponent, and a mantissa. The sign bit represents whether the value is positive or negative. The mantissa represents the decimal portion of the number. IEEE floating point numbers are normalized, meaning that there is an implicit 1 before the decimal. This effectively allows for one more bit in the mantissa. The exponent is related to the power of 2 with which the mantissa will be multiplied to obtain the actual value.

IEEE Standard The Institute of Electrical and Electronics Engineers (IEEE) has released successive versions of their standard IEEE 754, which details numerical formats for computer processors. This standard defines the ways that computers should perform arithmetic operations and store numbers, resulting in a universal standard. As a result, code written for one IEEE compliant processor will give the same mathematical results on any IEEE (same/previous revision) compliant processor.

bfloat16 The Brain Floating Point (bfloat16) an alternative 16-bit floating point format to the IEEE-754 half format. It is also beginning to be supported by hardware. One of the primary interests of the bfloat16 format is the ease of conversion between the bfloat16 and the IEEE-754 single precision floating point format, as the bfloat16 is a truncation of the single precision format.

Overflow

Overflow happens when the values being stored or calculated exceed the limits of what the format can store. For example, half precision floating point numbers can only store values up to 65504. Values greater than 65504 cannot be represented, so care must be taken to avoid calculating or storing any such values in a half precision format, otherwise they will be rounded to infinity.

Underflow

On the other hand, underflow occurs when a value is too small to be properly represented, or *subnormal*. For half precision, this is any value smaller than 2^{-14}. When a calculation results in a subnormal number, a number of bits of information are lost equivalent to the number of leading zeroes required to store the value. Further, if a calculated value is smaller than 2^{-24}, then it will truncate to zero.

Unit of Least Precision

Not every number can be exactly represented in floating point formats – there is a lower limit to their numerical precisions. This notion is represented by the *Unit of Least Precision (ULP)*[36], which corresponds to the least possible difference between two values for a given exponent. As shown in Table 3.2, the value of one ULP is not fixed for each type of precision, but depends on the value of the exponent. For an extreme example, the ULP for half precision floating point values at the top of their range - between 32769 and 65504 - is 32, meaning that only multiples of 32 (2^5) can be represented. Any calculated result is thus rounded to the nearest ULP less than the mathematical result. This ULP is the natural unit to use when discussing error rates of floating point algorithms thanks to its scale invariance.

One way to calculate the ULP is described in Algo. 12. This method is used to calculate the errors presented in Section 5.3. It is also possible to compute a ULP by starting with a known ULP and scaling its exponent up or down appropriately to match the exponent of the values of interest. For example, the C language defines `FLT_EPSILON` as one ULP at a value of 1 in the `float.h` header. One ULP at a value of 4 could thus be computed as `4 * FLT_EPSILON`.

Base Value	Half ULP	Float ULP	Double ULP
2^{-10}	2^{-20}	2^{-33}	2^{-63}
1	2^{-10}	2^{-23}	2^{-53}
2^{10}	1	2^{-13}	2^{-43}

Table 3.2: Value of 1 ULP at different base values for half, single, and double precisions.

Algorithm 12 Pseudo-algorithm for calculating a ULP. The increment/decrement functions increment and decrement the weakest bit of the mantissa while preserving the exponent and sign, causing them to each be one ULP away from x's value.

```
half ulp( half x ) {
  half x_inc = increment_mantissa( x );
  half x_dec = decrement_mantissa( x );
  half ulp1 = x_inc - x;
  half ulp2 = x - x_dec;
  return min( ulp1 , ulp2 );
}
```

Exponent Alignment

The dynamic range of floating point numbers renders arithmetic more complicated – if the two values have different exponents, then the exponents must be aligned before performing any arithmetic. When aligning the exponents, the smaller value's exponent is increased to match that of the larger value. As a result, a number of bits equal to the difference in the exponents are truncated from the end of the smaller value. When this difference in exponents is large or an algorithm involves many misaligned exponents, then this truncation can lead to significant errors. *Absorption* happens if the smaller value is so much smaller that it is completely negated when matching exponents. For all experimental cases presented in section 5.3, this misalignment is the dominant source of error.

Some systems alleviate the errors introduced by misaligned exponents through the use of *extended precision* floating point formats. However, SIMD instructions do not have any such extended precision built in.

Cancellation

It is important to remember that in many cases, the values being used to perform computational arithmetic are approximations. This can lead to results significantly with larger error than expected. One such scenario, known as *cancellation*, is the

subtraction of two nearly equal numbers such that almost all of the digits cancel out. The result of this subtraction is much more sensitive to the imprecision of the original approximations than other operations, such as addition and multiplication.

For example, we may consider a subtraction between the numbers 99.99 and 100.51. The real value of $100.51 - 99.99$ is 0.52. If we use a Q31.1 format, these values become 99.5 and 100.5, each of which is approximately 99% correct as an approximation. However, the result of the subtraction $100.5 - 99.5$ is 1, rather than 0.52. The result of the subtraction contains more than 90% error.

3.2 . Numerical Approximations

Due to the limitations in how numerical values are represented within a computer, it is necessary to approximate many mathematical operations that a user may want to compute. Some operations can be exact or nearly exact, such as integer addition and bitwise operations. However, most operations require some level of approximation. It should be noted that it is possible to perform exact computations using symbolic mathematical functions, but these functions tend to take much longer to compute.

There are many methods to approximate mathematical functions. Every method has its own advantages and disadvantages, making it more or less suitable for certain cases. Below, we describe some of the more simple and widely applicable methods that will be used in section 6.

3.2.1 . Taylor Series Expansion

The Taylor series of a function, described by eq. 3.1 is an infinite sum of functions that approximate the original function, centered around a certain value.

$$f(x) = \sum_{n=0}^{\infty} \frac{f'(a)}{n!} \cdot (x-a)^n \tag{3.1}$$

This infinite sum can be cut off at a chosen n in order to provide an nth order approximation. The error of this nth order approximation is equal to the sum of the remaining cut off terms. The balance between computational simplicity and permissible error thus lies in the choice of n. A small n will allow for more rapid computation at the cost of accuracy, while a large n takes longer to compute, but provides a more accurate result.

The further an input is from the center of the Taylor series (a), the less well the approximation performs. As a result, Taylor series approximations are best suited to repeating functions where it is possible to adjust the input to never be far from a, the center of the approximation. One such example, presented in Section 6.5.3, is the sine function. This function repeats itself completely within a range of 2π, meaning that it is possible to adjust the input to never be further than π from the center, improving the quality of the approximation. Taylor series approximations can also be suitable when the input domain is bound within a well defined range.

Remez Algorithm The result of a Taylor series approximation is a polynomial function. However, there exist other methods for generating polynomial approximations, such as the Remez algorithm[88]. While it can be more difficult to compute the polynomial coefficients without specialized tools such as [15], the resulting approximation is often, but not always, superior to the Taylor approximation.

3.2.2 . Newton-Raphson

The Newton-Raphson method is an iterative algorithm for finding the roots of a function (points where $f(x) = 0$) via eq. 3.2.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{3.2}$$

The function $f(x)$ is defined in such a way that the final x_i is the approximation of a function $g(y)$ at a point of interest a.

As opposed to the Taylor method described above, the Newton-Raphson method is iterative. This means that it performs repeated iterations in order to approach a solution. As such, it can be said to converge upon the solution. Various iterative algorithms display different rates of convergence. A higher order rate of convergence means that an algorithm converges towards its solution more rapidly (per iteration). The Newton-Raphson method described here generally exhibits quadratic (second-order) convergence.

It can be noted that eq. 3.2 does not define x_0 as a starting point. This leaves a degree of freedom in the choice of this x_0 starting seed. For some functions $f(x)$, this seed can be chosen intelligently in such a way that the algorithm converges more quickly.

The choice of x_0 must respect certain conditions for the algorithm to guarantee quadratic convergence. First, $f'(x)$ must be nonzero within a region defined by the distance between x_0 and a. Otherwise the algorithm can stall if it reaches a point where $f'(x) = 0$. Secondly, $f''(x)$ must be continuous within the same region. Finally, x_0 must satisfy certain conditions of sufficient closeness to a. If all of these conditions are not satisfied, then the algorithm may exhibit less than quadratic convergence – if it converges at all.

In order to use the Newton-Raphson method to approximate an operation $g(a)$, $f(x)$ must be defined in such a way that the roots of $f(x)$ are equivalent to the result of the operation we wish to perform. In general, the simplest $f(x)$ for an operation $g(a)$ can be defined as $f(x) = g^{-1}(a) - a$ where $g^{-1}(a)$ is the inverse of the operation $g(a)$. A few examples are provided in table 3.3.

In order to use the Newton-Raphson method, it must be possible to compute $f'(x)$, preferably in a computationally efficient manner. One famous application of this algorithm is computation of a square root. As shown in table 3.3, the operations needed to update x_i are computationally simple.

The choice of the starting seed, x_0, is very important. If it is poorly chosen, the algorithm may take significantly longer to converge, if it does converge on the

$g(a)$	$g^{-1}(x)$	$f(x)$	$f'(x)$	$x_i - \frac{f(x_i)}{f'(x_i)}$
$\frac{1}{a}$	$\frac{1}{x}$	$a - \frac{1}{x}$	$\frac{1}{x^2}$	$x_i \cdot (2 - a \cdot x_i)$
$\sqrt{a}$	x^2	$a - x^2$	$2 \cdot x$	$\frac{1}{2} \cdot (x_i + \frac{a}{x_i})$
$\sin(a)$	$\arcsin(x)$	$a - \arcsin(x)$	$\frac{1}{\sqrt{1-x^2}}$	$x_i - \arcsin(x) \cdot \sqrt{1 - x^2}$

Table 3.3: Newton-Raphson applied to a few example cases.

correct solution. The division example shown in table 3.3 is one such example. In this example, if x_0 is too small, the algorithm will perform a large number of small iterative steps as it accelerates towards the solution. However, if x_0 is too large, the algorithm will converge on $x_i = 0$, rather than $x_i = \frac{1}{a}$. Section 6 will show that in some cases, it is possible to choose an efficient x_0 that avoids both of these possible problems.

There also exist other "Newton-type" methods capable of faster convergence than the classic Newton-Raphson method. For example, the methods presented in [3], [103], and [32] exhibit cubic rates of convergence rather than the classic method's quadratic convergence. While these methods converge more quickly, they also all require more computations per iteration. As a result, they are not automatically superior when considering runtime performance.

It is also possible to use other methods – such as ISA instructions or faster algorithms – to provide a rough approximation of an operation, then use a Netwon-type method to refine the rough approximation into a better approximation. This technique is applied to the reciprocal square root compution in [108] and [67], as well as the reciprocal computation in Section 6.5.4.

Conclusion

Chapter 5 will explore the effects of substituting half precision in the place of floating point precision. Chapter 6 will apply many of the concepts presented here in order to construct a software library to efficiently emulate fixed-point arithmetic. Finally, chapter 7 will combine the concepts presented here with an artificial neural network presented in the following chapter in order to perform a number of experiments.

4 - Artificial Neural Networks

Artificial neural networks (ANNs) are computational tools used to predict outputs from inputs. Popular applications of ANNs tend to be problems that are difficult to generalize clearly and logically, such as object detection, image classification, and translation. These problems also tend to have large, messy inputs which are difficult to handle programmatically. This messiness and complexity is modeled in the general architecture of ANNs, which could be described as a tangled web of interrelated connections.

The basic building block of an ANN is the perceptron. A perceptron is a simple function that takes a number of inputs and converts them into a single output[89]. One simple example of such a function would be a perceptron that outputs the sum of all inputs. In human terms, this could be considered similar to a person looking at data and coming to a conclusion based on the data. It is then possible to set up a number of perceptrons into a network where the output of certain perceptrons is used as the input to other perceptrons. These perceptrons can be arranged into layers such that each layer of perceptrons generates the inputs for the next layer. In practice, neural networks are constructed of series of such layers feeding into other layers in a mostly linear fashion. If a neural network consists exclusively of layers of perceptrons, it can be referred to as a multilayer perceptron.

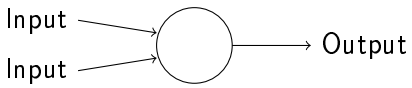

Figure 4.1: Example of a simple perceptron

Rather than manually setting the weights of each perceptron, it is possible to *train* a neural network using a database of inputs and corresponding desired outputs[104]. The most important requirement of this training is the set of example inputs – they should be representative of the target use-case and numerous enough to avoid *overfitting* the problem. Overfitting occurs when a trained network models the training images specifically at the cost of its ability to solve a problem. A simple example of overfitting (Fig. 4.2) would be fitting a five degree polynomial to four data points generated by a second degree polynomial.

4.1 . Training vs Inference

There are two phases in the usage of an ANN – training and inference. The training phase consists of "teaching" the ANN to link inputs to known outputs, while the inference phase uses the trained links to predict the outputs corresponding to inputs that the ANN may or may not have seen before.

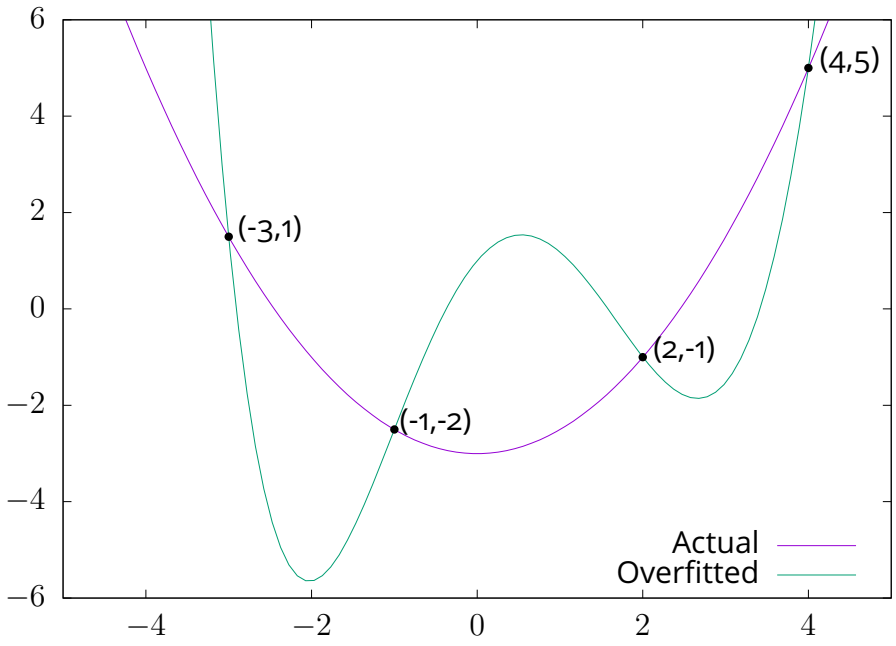

Figure 4.2: Overfitting a complex function to points generated by a simple function

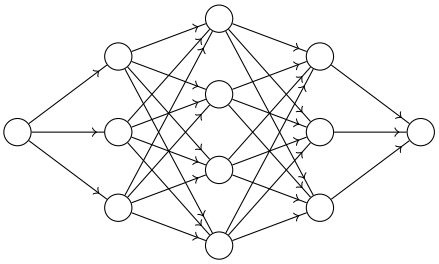

Figure 4.3: Example of a small fully connected neural network

The purpose of the training phase is to adapt the weights of the perceptrons of each layer in order to allow the entire network to predict outputs accurately. This phase requires three components – a set of sample inputs, a set of outputs corresponding to the inputs, and an *objective function*. The inputs should ideally be representative of the inputs the neural network will receive when deployed while also being varied and numerous enough to avoid overfitting issues. The outputs should be comprehensive and verified to be correct. "Garbage in, garbage out" applies very well to the data that neural networks are trained on. Finally, the objective function is a function that quantifies in some way the accuracy of a prediction or set of predictions. This provides a function that can be minimized or maximized over the set of input/output pairs using techniques such as backpropagation[90] or reinforcement learning[95]. These techniques tend to be very computationally intensive due to the large amount of inputs and iterations required. While entire works can be written about neural network training, this work focuses entirely on what happens after the training phase.

Once a neural network is trained, it can be used to perform *inference* on new

inputs. Inference requires a neural network model (or architecture), trained weights for the model, and new inputs. This stage is much less computationally intensive than training, to the point where some models can be computed in real-time. For the models cannot perform inference as quickly as the user would like, there are a number of techniques that can be used to speed up inference (Sec. 4.3).

4.2 . Layers & Computation

There are two types of layers in particular that tend to consume heavy computational resources: fully connected layers and convolutional layers. For example, the PVANet network spends approximately 36% of its inference time inside these two layers.

4.2.1 . Fully Connected Layer

Fully connected layers contain the majority of the trained parameters in a neural network. The number of parameters associated with a fully connected layer are proportional to the number of perceptrons in the input layer times the number of perceptrons in the output layer.

$$p_{out}(j) = b(j) + \sum_{i=0}^{N_{in}} w(i,j)p_{in}(i) \tag{4.1}$$

Computationally, the output is obtained via a matrix-vector multiplication, as shown in Eq. 4.1. This is a well known type of operation, often referred to as a GEMV (GEneral Matrix-Vector) or blas2 computation. Most neural network implementations take advantage of linear algebra software libraries, such as BLAS[4], ATLAS[105], or MKL[50]. Chapter 7 will explain the limitations of the above libraries when performing certain inference optimizations.

4.2.2 . Convolutional Layer

Convolutional layers require a significant amount of computation with a small amount of repeated parameters. In the case of a convolutional layer, the trained parameters represent *convolutional matrices*, also referred to as kernels. Each of these kernels represents a transformation of the input image, as shown in Fig. 4.4.

In order to discuss convolutional layers, it is helpful to understand standalone image convolutions.

Image Convolution

An image convolution is a type of image transformation. This convolution is a simple operation that is applied to every pixel of the input image. The basis of a convolution is the *kernel* – a matrix that describes how to transform each pixel. Kernel sizes represent the size of the region of interest. For example, a kernel of size 1x1 looks only at the input pixel while a kernel of size 3x3 looks at the 3x3 pixel region centered on the input pixel (1 pixel away in each direction, including

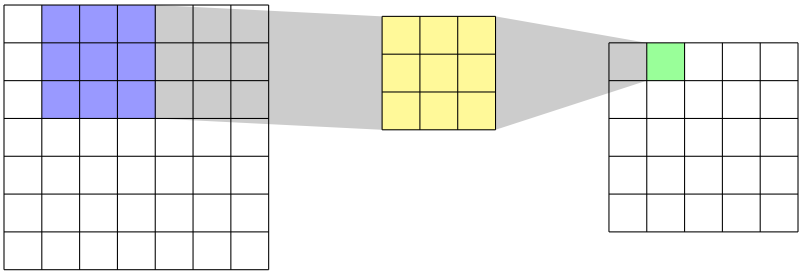

Figure 4.4: Example of a step in an image convolution.

diagonals). Another way to view the kernel is as a sliding window that passes over the input image, pixel by pixel. At each pixel stop, the kernel is applied to the input image by multiplying the value of a kernel entry by the value of the input pixel underneath it. Then, each of these products is added together to obtain the result of the convolution. This process is repeated for each pixel of the input.

What happens when the kernel cannot be overlaid entirely on top of the input image? This is an important question because it applies to all four sides of the image. There are a few main solutions to this problem. First, it is possible to simply ignore the edges and only operate on the inner region of the image. However, this means that the output image will be smaller than the input. Sometimes this is an acceptable tradeoff, sometimes other solutions are needed. Another solution is to wrap around to the opposite edge. For example, a kernel applied to the top left pixel of an image will also include some pixels from the bottom right region of the image. However, if the images contain important information near the edges, this can cause the edges to contaminate each other. A third solution is to *pad*, or extend the sides of, the image with 0-value pixels. This combines with the first solution (ignoring the edges) to output an image of the same size as the input without any contamination between the edges. There exist other solutions, but these are the main ones applied in the context of this work.

Up until here, we have been describing convolutions of single image (1-channel) inputs. However, most images are stored in RGB (3-channel) format with one channel for red colors, one channel for green, and one channel for blue. Each of these channels contains a different type of information about the picture. We could combine the three channels into one before performing a convolution, but we would lose important information about the picture. Instead, we can simply add an extra dimension to our kernel – a 5x5 kernel would become a 5x5x3 kernel. This allows our convolution output to incorporate information from each input channel.

In the context of a convolutional neural network layer, we often have more than 3 input channels and more than one output channel. As a result, the convolutional layer contains one kernel for each output channel and each kernel has a number of channels equal to the number of input channels.

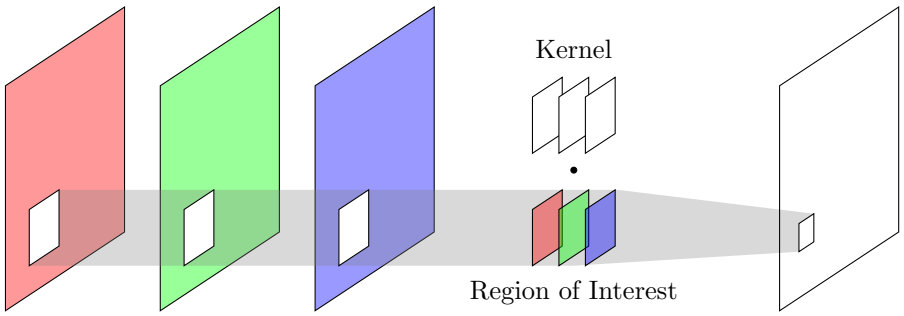

Figure 4.5: Example of a 3-channel convolution

4.3 . Optimizing Inference

While inference is less computationally intensive than training, it is still intensive enough to be worth optimizing. The fully connected layers contribute a large number of weights, which must be stored in memory. For example, PVANet contains 369MB of weights. In addition, the matrix-matrix multiplication required with fully connected layers are memory bound, giving a strong incentive to reduce the memory requirements. The convolution layers contribute relatively few weights but are also memory bound.

As a result, optimization tends to result in reduced memory demands. This can take the form of *pruning* connections to reduce the required computations and remove the associated weights or of reducing precision in order to increase the data flow and parallelizability.

4.3.1 . Pruning

One method of optimizing trained neural networks is to prune away unnecessary connections. Of the vast number of connections in a neural network, a certain amount can be removed with minimal or no effect on the results. This can range from removing a handful of connections within a fully connected layer, to removing a channel of an image convolution, to the removal of an entire layer.

4.3.2 . Reduced Precision

The method of optimizing trained neural networks focused on in this work is the reduction of precision used during inference. This provides a number of advantages. First, the memory requirements can be reduced. Moving from double precision to half precision would cut memory requirements to a quarter of the initial requirements. Second, smaller data types can be moved more quickly. This accelerates memory-bound computations. Thirdly, changing between data types can accelerate arithmetic computations. As shown in Table 1.1, integer addition, subtraction, and multiplication are faster than their floating point counterparts. Finally, reduced precision allows better parallelizability. Smaller data types are better accelerated via SIMD than larger data types.

4.4 . Validation

When experimenting with neural networks, it is important to have a means of validating and quantifying the results. In order to determine if an individual result is correct or not, a database of known inputs and outputs is required. For the purpose of this work, we use the data from the Pascal VOC Challenges[26] to validate results by calculating the mean average precision (mAP) of inference on a set of images. In order to describe the mAP computation, we will first explain the format of the outputs and some of the concepts involved in computing the mAP.

4.4.1 . PASCAL VOC Dataset

The PASCAL VOC (visual object challenge) datasets are a series of image recognition datasets provided for a image recognition/detection challenge. There is one dataset per year for each year from 2005 until 2012, although this work primarily used the 2007 and 2012 datasets for validation. Each dataset consists of several thousand images with corresponding annotated objects. The annotations each contain an object category (20 total categories starting from 2007), a bounding box, and a flag marking if an object is difficult and should be excluded during mAP computation. The mAP computation relies on the concepts of precision and recall and of the intersection over union.

4.4.2 . Precision and Recall

The mAP computation relies on a pair of corresponding principles called *precision* and *recall*.

Precision is a measure of how accurate the results of an inference are. More specifically, it is the number of object detections which were correct (true positives) divided by the total number of object detections (true positives plus false positives). For example, if an inference provides 4 objects and 3 of them match their annotations, we obtain a precision of $3 \div 4 = 0.75$.

Recall is a measure of how well an inference finds objects in an image. More specifically, it is the number of correct object detections (true positives) divided by the number of objects that should have been detected (true positives plus false negatives). For example, if an inference provides 3 objects that match their annotations for an image that contains 5 annotated objects, we obtain a recall of $3 \div 5 = 0.6$.

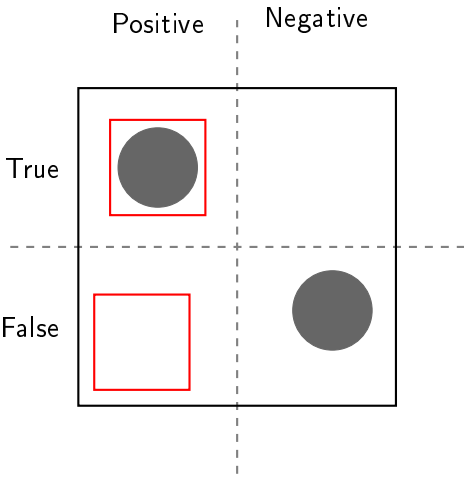

Figure 4.6: Visual example of True/False Positive/Negative classifications.

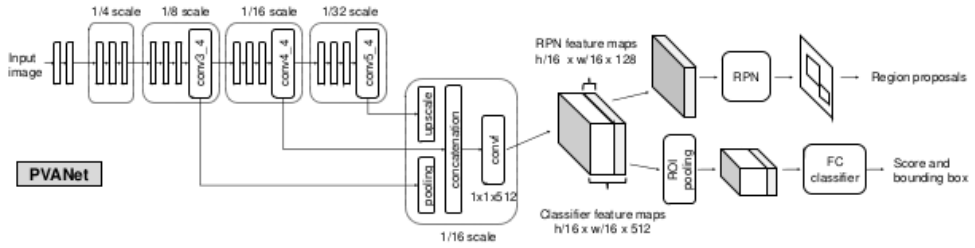

Figure 4.7: Structure of the PVANet neural network[46].

4.4.3 . Intersection over Union

In practice, the results of inferences do not perfectly match the annotated bounding boxes provided in the VOC dataset. This does not mean that the result are invalid. However, it does mean that it is necessary to define a measure that determines if an inference result matches an annotation. For this, we use the area of intersection the annotated bounding box and the inferred bounding box divided by the area of union of the two boxes. An IoU of 1 represents a perfect match, while an IoU of 0 represents a complete mismatch. A value in between can be chosen as the threshold to determine what can be considered a match.

4.4.4 . mAP

The mean average precision is the mean of the average precisions of all classes considered.

In order to compute the average precision, a precision/recall curve $p(r)$ of all results must be constructed. This curve is constructed by first choosing a number of confidence thresholds. Secondly, the corresponding precision and recall are computed considering all inferred objects above the confidence threshold. Thirdly, the precision/recall curve is smoothed out by setting each point equal to the maximum of all points with a greater recall ($p(r_i) = \max(p(r_j | r_j > r_i))$). Finally, the average precision is the area under this final $p(r)$.

VOC mAP It should be noted that the VOC challenge provides two means of approximating the above integral – the 2007 version and the 2012 version. The 2007 mAP computation approximates the integral using 11 points – $0.0, 0.1, 0.2, ..., 0.9, 1.0$. Meanwhile, the 2012 version approximates the integral using all available points along the curve. As a results, the 2012 mAP will generally be lower than the 2007 mAP.

4.5 . The PVANet Neural Network

The experiments that will be presented in Chapter 7 will use the PVANet neural network[46] published by Hong et al. in 2016. There are a few reasons for choosing the PVANet network for this work.

Reproducibility Most importantly, the published results are reproducible – the authors provide a github repository[45] with the code needed to reproduce their results. These reproducible results take the form of an mAP computed from the results of the VOC2007 and VOC2012 challenges.

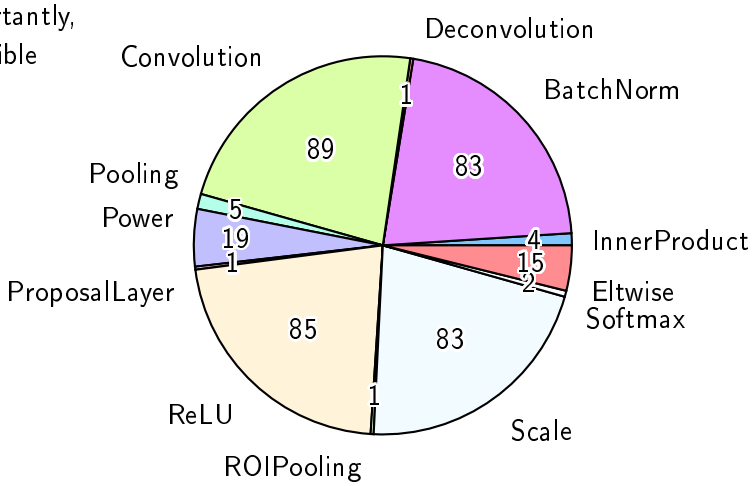

Figure 4.8: Distribution of layers in PVANet.

Accuracy The open-source PVANet achieves an mAP of 84.9% on the VOC2007 challenge and 89.8% on the VOC2012 challenge when computing VOC2007 mAP. This leaves room for the experiments to degrade the output while still maintaining a coherent, usable output.

Speed When computing using only the CPU, PVANet inference takes less than 2 seconds on an Intel i7-4790S at 3.20GHz CPU. This makes it an excellent target for acceleration, as it is not far from lower speed real-time processing. Even a speedup of 2 would allow for inference in less than 1 second.

Variety PVANet contains a variety of layers. Figure 4.8 shows a distribution of computational layer types present in PVANet. Section 4.5 will describe all layer types present in PVANet. Notably, the PVANet classifications rely on fully connected layers in order to "provide possibility to balance between computational cost and accuracy of a network"[46].

Layers Present in PVANet

This section provides brief descriptions of both the purpose and the computational intensity (during inference) of each layer type present in PVANet. Figure 4.9 shows the relative times spent in each layer type during an inference using Caffe. It should be noted that, except for the Input layer, each layer's input is the output of a preceding layer.

BatchNorm The BatchNorm layer performs Batch Normalization of its input. This Batch Normalization serves to normalize its input values, allowing faster network training and increased stability[54]. Computationally, the BatchNorm performs one subtraction and one division operation for each element of its input.

Concat The Concat layer concatenates multiple inputs into a single layer. Computationally, this consists of a simple memory copy from multiple input buffers into a single output buffer.

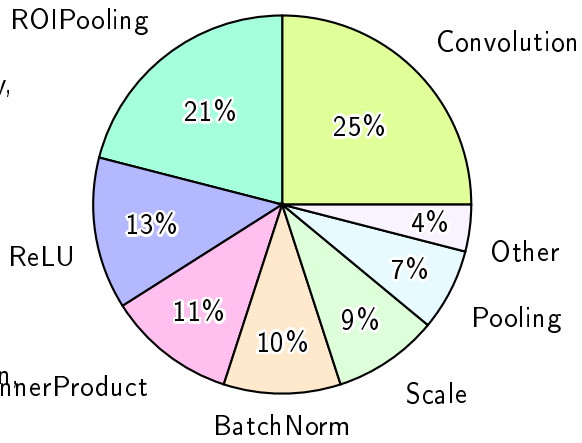

Figure 4.9: Time spent in each PVANet layer type during inference with Caffe. Benchmarked using `callgrind`[76].

Convolution The Convolution layer performs an image convolution as described in Section 4.2.2. Its purpose is to attempt to extract visual information from an input image. The size of the convolutional kernel varies between the different Convolution layers present in PVANet, ranging from 7x7x16 to 1x1x32. Computationally, the intensity depends on the size of the kernel. However, the Convolution tends to be one of the more computationally intensive layers in general.

Deconvolution The Deconvolution layer is similar to the Convolution layer. For the purposes of this work, it suffices to say that the Deconvolution layer is similar in computational intensity to the Convolution layer.

Dropout The Dropout layer serves only in the training phase. It removes weights randomly in order to avoid overfitting[44]. During inference, the Dropout layer performs no computation. It is of no interest for the purposes of the works presented here.

Eltwise The Eltwise layer takes two inputs and performs a single elementwise operation between them. This combines the information present in its two inputs. These operations generally take the form of a multiplication, an addition, or a max. Computationally, the Eltwise layer is relatively simple.

InnerProduct The InnerProduct layer is a fully connected layer which performs the GEMV operation described in Section 4.2.1. This layer is simultaneously the most important and the most difficult to describe the purpose of. It contains the hidden, unintuitive complexity that makes most neural networks work. Computationally, the InnerProduct layer is heavy. It represents only less than 1% of the layers (4 of 455), yet requires 11% of the computation time during inference.

Input The Input layer, true to its name, is for inputs. It provides a buffer into which the input image can be copied for use by other layers. As such, it performs no computation itself.

Pooling The Pooling layer serves to smooth out an image input. It looks at a 2D input region and outputs either the maximum value or the average value in the region. The computational intensity depends on the size of the region being examined, but lies somewhere between the Eltwise layer and the Convolution layer in intensity.

ProposalLayer The ProposalLayer is a layer custom designed for PVANet. Its purpose is to transform the output information into a usable format. Computationally, it is relatively lightweight to its small input size.

ReLU The ReLU layer frequently serves to accelerate training. It sets all negative values of its input to 0 while leaving the positive values of the input intact. Computationally, it is not particularly intensive.

Reshape The Reshape layer reshapes its input. It may be necessary to put a Reshape layer between layers whose dimensions do not exactly match. Computationally, the Reshape consists entirely of memory copies. If the reshape is not cache-friendly, it may be slow.

ROIPooling The ROIPooling is a layer custom designed for PVANet. For each region of interest (ROI) found by previous layers, it generates an output that can be classified by the following layer. Computationally, it can be heavy, depending on the number of ROIs. For each ROI input, the ROIPooling must generate a 6x6x512 output.

Scale The Scale layer scales its inputs. This can serve to normalize the input to some degree, or to allow further layers to behave better numerically. Computationally, it is relatively lightweight – each output is the result of a multiply-add operation on an input ($out_i = a * in_i + b$).

Split The Split layer represents a branch in the neural network architecture. As such, it is implicitly placed after any layer whose output is used by more than one other layer. Computationally, it consists of a single continuous memory copy.

Softmax The Softmax layer serves to transform its inputs into probabilities. The sum of the outputs of a Softmax layer is equal to 1. Computationally, it is moderately intensive, as it requires multiple loops over the input.

Conclusion

Chapter 7 will fully utilize the concepts presented in this chapter. It presents the development of a custom neural network inference engine designed for arbitrary precisions. Then it presents the results of performing experiments varying the arithmetic used during inference of the PVANet neural network.

Part II

Contributions

5 - Half Precision Average

This chapter describes research performed in order to improve the computation of an average of a large set of numbers using the half precision numerical format. We will compare the numerical and runtime performances of several algorithms (including one that we propose). Numerically, we compare the relative errors and failure modes of each algorithm. For runtime performance, we compare SIMD runtime performance of each algorithm using various CPU SIMD architectures.

This section specifically focuses on the half-precision floating point data type initially defined by the Cg programming language in 2002[81], then integrated into the IEEE standard in 2008[47]. While other 16-bit floating point formats[92][53] have existed prior to this one, the IEEE normalized format is the most prominent format. For a number of years, these half-precision types were only implemented on certain GPUs. Nonetheless, some works experimented with emulating half-precision via 32-bit floating point CPU SIMD instruction sets, such as those performed by Etiemble et al[25] and Lacassagne et al[65]. While these works showed limited speedups due to the emulation constraints, they did show some of the potential speed gains. In parallel, some studies implemented half-precision arithmetic on FPGAs – such as the Shirazi et al's scalar[94] and Etiemble et al's SIMD[24] implementations. Some works – such as Piskorski et al[85] – also implemented half-precision arithmetic on configurable CPUs. These works all proved to show some of the potential gains from physically supporting this data type. All of these initial studies also served to explore for which types of applications half-precision types can be sufficient. In these early experiments, they proved more than sufficient for a number of media processing applications.

More recently, industrial use cases of half-precision algorithms, such as in the field of machine learning, have renewed interest in the field of half-precision computation. As a result, hardware manufacturers have begun supporting half-precision as a native datatype (both scalar[51] and SIMD[61][79]). This increased support in turn allows for a stronger focus on half-precision computation in other areas, such as computer vision[85][84]. While this half-precision hardware support is not yet ubiquitous, it is growing rapidly.

5.1 . Computing the Average

Mathematically, the formula for calculating the average of a set A of N elements $a_1, ..., a_N$ is shown in eq. 5.1.

$$\text{Avg} = \frac{1}{N} \sum_{i=1}^{N} a_i \tag{5.1}$$

There are a few existing methods for performing this calculation computationally. The choice of method depends on the contextual limitations under which the average is being calculated. This section will present a few methods for calculating the average.

5.1.1 . Sum Then Divide

The simplest algorithm to implement consists of naively summing all elements of the set before dividing by the number of elements in the set (Algo. 13). This **naive** method will be tested in section 5.3. This method has two potential

Algorithm 13 Naive algorithm for average calculation.

```
1: function Naive Average(Array)
2:     sum = 0
3:     for a in Array do
4:         sum += a
5:     end for
6:     avg = sum / length(Array)
7: end function
```

weaknesses - rounding errors and overflow. Both of these limitations are amplified and more strongly tested in a low-precision context. Rounding errors can be compensated through the use of compensated arithmetic methods such as TwoSum[75], Kahan Sum[59](Algo. 14), and Pairwise Summation[109]. However, the only way to reduce overflow errors is to increase the precision of the sum accumulator. This strategy of increasing the precision will be referred to as the **upcast** method and tested in section 5.3 along with the **Kahan** method.

Algorithm 14 Kahan variation for average calculation.

```
1: function Kahan Average(Array)
2:     sum = 0
3:     rem = 0
4:     for a in Array do
5:         y = a - rem
6:         t = sum + y
7:         rem = ( t - sum ) - y
8:         sum = t
9:     end for
10:    avg = sum / length(Array)
11: end function
```

If one is willing to pay a higher performance cost, there is another technique available to mitigate rounding errors – sorting the input data prior to summation. Rounding errors are minimized when the input data is sorted from low to high,

as this minimizes the order of magnitude differences between the summation accumulator and the inputs being added to it. This sorting approach has a large performance cost because sorting is generally more computationally expensive operation than any of the summation algorithms and cannot be fused into the computational loop for the summation or averaging operation.

5.1.2 . Divide Then Sum

Conversely, one may wish to avoid overflow by dividing each input element by the total number of elements, as seen in Algo. 15. This reduces the risk of overflow and remains vulnerable to rounding errors, but introduces new possible errors (underflow, subnormal values) via the division operation. For a sufficiently large array, this exchanges the overflow risk for an underflow risk that the result of the division calculation is less than one ULP, resulting in zero. While the rounding error is equivalent to the Naive method (the order of magnitude difference is the same), the risk of subnormal and zero results of the division can increase the overall error. This method is not significantly different from the naive approach, so it

Algorithm 15 Divide then sum for average calculation.

```
1: function Divide First Average(Array)
2:     avg = 0
3:     for a in Array do
4:         avg += a / length(Array)
5:     end for
6: end function
```

will not be directly tested in section 5.3.

5.1.3 . Iterative Average

Another method for calculating an average is the **iterative** average. This method iteratively calculates the average according to eq. 5.2, which leads to Algo. 16.

$$\text{Avg}_{i+1} = \text{Avg}_i + \frac{(x_i - \text{Avg}_i)}{i} \tag{5.2}$$

Using this method entirely circumvents potential overflow problems. However, it

Algorithm 16 Iterative algorithm for average calculation.

```
1: function Iterative Average(Array)
2:     avg = 0
3:     for i in 1 → length(Array) do
4:         avg += (avg - Array[i]) / i
5:     end for
6: end function
```

still has limitations due to rounding errors. The larger the set of numbers being averaged, the greater the effect of rounding errors arising from the division.

$$\lim_{i \to \infty} \frac{(x_i - \text{Avg}_i)}{i} = 0 \tag{5.3}$$

These rounding errors allow the algorithm to reach the limit shown in eq. 5.3 easily.

5.2 . Cascading Average

The method that we propose in order to combat rounding errors and overflows in a low-precision environment will be referred to as the *Cascading Average*. This Cascading Average consists of recursively splitting the set of numbers into two sets and calculating the weighted average of the two sets (Algo. 17, Fig. 5.1).

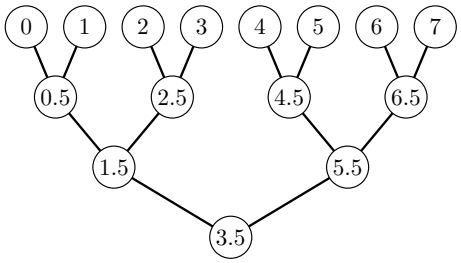

Figure 5.1: Example of the calculations performed by the Cascading Average algorithm.

This construction, consisting of a series of weighted averages between two numbers, offers a few benefits. First, it avoids accumulation into a monolithic sum, minimizing the risk of over/underflow or absorption. Additionally, having each calculation be an average of two numbers within the original input range reduces the severity of truncation errors due to mismatched exponents. There are more opportunities for arithmetic rounding errors to arise due to the increased number of operations, but these are mitigated by the general exponent alignment. These benefits allow for greater robustness over large sets of numbers without greatly elevating the level of error accumulation.

In practice, the recursive implementation of the cascading average quickly results in a stack overflow. It is thus necessary to use a sequential implementation (Algo. 18) in order to obtain good results. The sequential algorithm presented in Algo. 18 consists of a primary computation phase, which iterates over the full array, followed by a secondary phase which accumulates the result into a single value. The primary phase simulates traversal of a binary tree in the same order as the recursive version. In the case of an input that is not a power of two, the cleanup phase then consolidates any remaining nodes of the tree into the result. Because we traverse the input as if it were a binary tree, all divisions prior to the cleanup phase are divisions by 2. This means that they can be implemented as a shift operation or a multiplication by 0.5 rather than a more computationally expensive integer division operation.

Algorithm 17 Simplified cascading algorithm for average calculation. A sequential version of this algorithm was implemented for testing purposes.

```
1: function Cascading Average(Array)
2:     n = length(Array)
3:     if n == 1 then
4:         return Array[0]
5:     else
6:         return ( Cascading Average( Array[0:n/2] ) + Cascading
   Average( Array[n/2:n] ) ) / 2
7:     end if
8: end function
```

5.3 . Results

This section will discuss the runtime and numerical performance of the various average calculation algorithms for various use cases.

5.3.1 . Performance

All performance benchmarks were run using stretches of a same input image in order to obtain different sized input arrays. We perform all speedup benchmarks using IEEE-754 32-bit floating point values, as SIMD 16-bit floating point values are not yet fully supported. While this does not reflect the eventual performance that 16-bit floating point algorithms when they will be fully supported, it does provide a baseline expectation of how each algorithm will perform.

Experimental Setup

Intel performance benchmarks were performed using an Intel Core™ i7-4790S CPU @ 3.20GHz CPU with 16GB of RAM, compiled using GCC 6.3.1. PowerPC performance benchmarks were performed using a Power8 8348-21C @ 2.061GHz 64GB of RAM, compiled using GCC 6.3.0. NEON performance benchmarks were performed using an ARM Cortex-A57r1 @ 1.91GHz 4GB of RAM, compiled using GCC 5.4.1. Each algorithm was benchmarked using scalar instructions, SSE SIMD, and AVX SIMD in order to examine how each algorithm scales with SIMD technology. Scalar benchmarks were complied with options `-O3 -std=c++0x`. SSE benchmarks were complied with options `-O3 -std=c++0x -msse4.2`. AVX benchmarks were complied with options `-O3 -std=c++0x -mavx2`. NEON benchmarks were complied with options `-O3 -std=c++0x -march=armv8-a+simd`. Altivec benchmarks were complied with options `-O3 -std=c++0x -maltivec`. There was no need to disable compiler automatic vectorization via the `-fno-tree-vectorize` flag, as it did not vectorize any of the examples. The SSE benchmarks were written using **nsimd** [11] – an **Agenium Scale** library that provides a portable high level C++ interface for

Algorithm 18 Sequential cascading pseudocode for average calculation. Consists of primary computation phase, followed by a cleanup phase.

```
1:  function Cascading Average(Array)
2:      for i = 0 → length(Array) do
3:          tmp = Array[i]
4:          for j = 0 → ⌈log2(length(Array))⌉ do
5:              if !in_use[j] then
6:                  tmp_avg[j] = tmp
7:                  in_use[j] = true
8:                  break
9:              end if
10:             tmp = ( tmp_avg[j] + tmp ) / 2
11:             in_use[j] = false
12:         end for
13:     end for
14:     worth = 1
15:     accumulated = 0
16:     final_avg = 0
17:     for j = 0 → ⌈log2(length(Array))⌉ do
18:         if in_use[j] then
19:             mul1 = acc_worth / ( worth + acc_worth );
20:             mul2 = worth / ( worth + acc_worth );
21:             final_avg += ( mul1 * final_avg ) + ( mul2 * tmp_avg[j] )
22:             accumulated += worth
23:         end if
24:         worth = worth * 1
25:     end for
26: end function
```

writing vectorial code. Benchmarks were run for the following algorithms – Naive, Kahan summation, Iterative Average, and Cascading Average. All algorithms were vectorized using Intel SSE4.2, Intel AVX2, ARM NEON, and IBM Altivec SIMD instructions. At the time of these experiments, we did not yet have access to an AVX512 machine.

All performance benchmarks were run using stretches of a same input image in order to obtain different sized input arrays.

Results

Table 5.1 provides a reference of the number of addition and division operations required to compute each algorithm. As shown in Figure 5.2, the naive method is

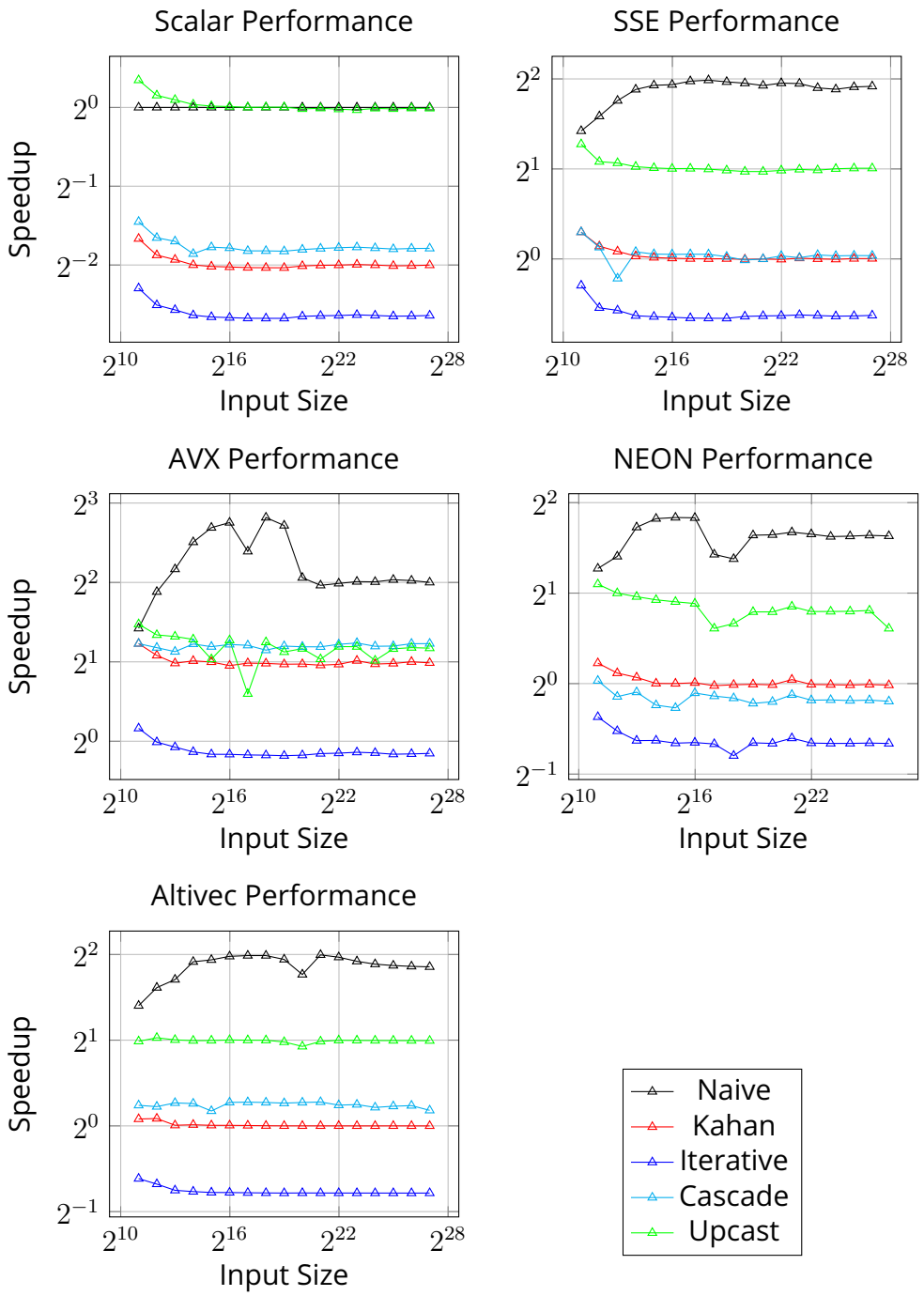

Figure 5.2: Speedup for different average calculation algorithms relative to naive scalar implementation for various input sizes. Bigger is better. Benchmarks performed using 32-bit floats to take advantage of SIMD acceleration.

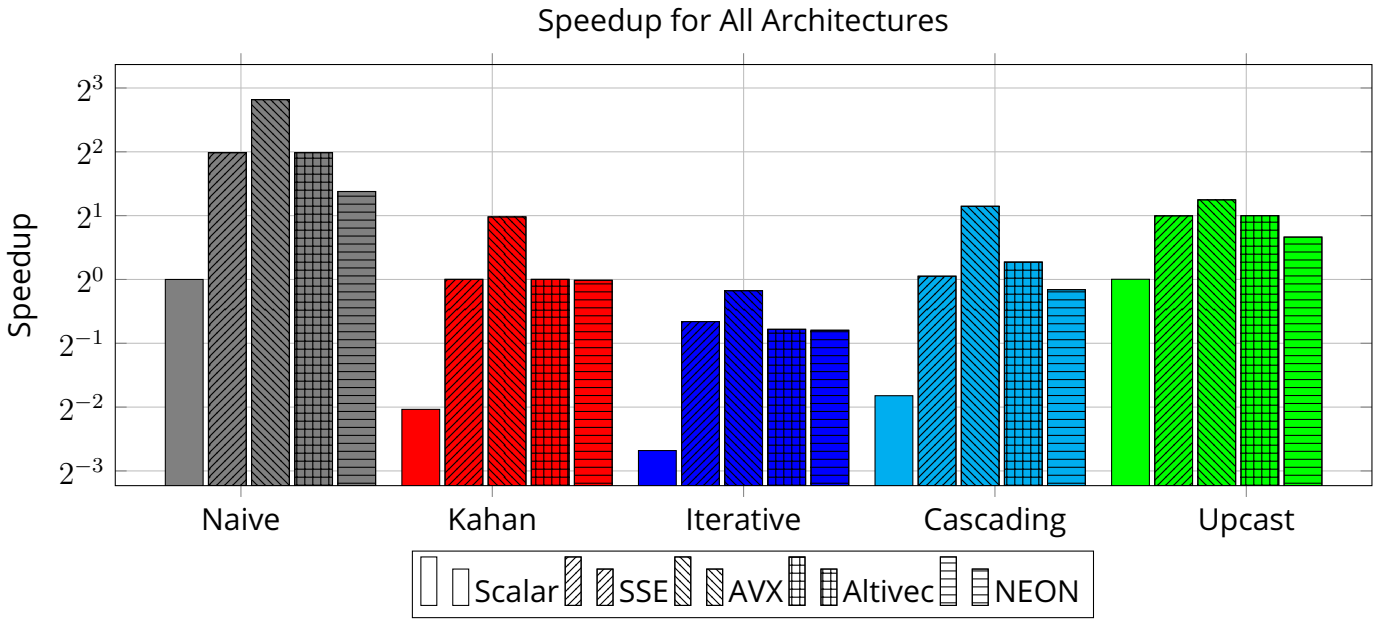

Figure 5.3: Speedup for different average calculation algorithms relative to naive scalar implementation at $N = 259200$. Each algorithm is presented from left to right as scalar, SSE, AVX, Altivec, and NEON. Bigger is better. Benchmarks performed using 32-bit floats to take advantage of SIMD acceleration.

the fastest in all scenarios, as it performs the fewest operations per element. The only operation performed within the loop is a single addition, which is a very fast operation to perform. The upcast method performs equally without vectorization, but naturally falls behind when vectorized because fewer higher precision values fit into an SIMD register. The Kahan summation method and cascading average perform nearly equally to each other, taking about four times as long as the naive method to perform their calculations. This is due to the extra operations performed per element. The Kahan average has four times as many addition/subtraction operations per element, leading it to take about four times as long to execute each iteration. The iterative implementation of the cascading average suffers from increased operations – on average two additions and divisions by two per element – and is susceptible to incorrect branch predictions. In practice, these divisions by two can be optimized to multiplications by 0.5, allowing faster computation. Finally, the iterative method takes about five times as long to perform its calculations. This represents the extra cost of a subtraction and an expensive division operation.

Figure 5.3 summarizes the speedups of each algorithm for each SIMD architecture tested. The SSE and Altivec results are almost exactly as expected for 128-bit SIMD registers. All algorithms except the upcast method show the expected speedup of 4 ($128 \div 32$) on both of these architectures. The upcast method only exhibits a speedup of 2 ($128 \div 64$), as the computations are performed using a higher precision. The NEON speedups are similar, as the SIMD registers are also 128-bits. The most computationally intensive and slowest algorithms – Kahan and Iterative – exhibit the same speedup of 4. However, the Naive, Cascading, and Upcast speedups are slightly lower, indicating that they are approaching the memory bandwidth limits of the machine. The AVX2 results for the Naive, Cascading, and Kahan algorithms show an expected speedup of 2 over the SSE results. This is as expected when moving comparing 256-bit SIMD registers to 128-bit SIMD registers. The Iterative algorithm shows a lower than expected speedup due to its use of a division operation in the computational loop. On the Haswell architecture used to perform the benchmarks, 32-bit SIMD division is faster than 64-bit SIMD division[49]. The Upcast algorithm also shows a lower than expected speedup due to the cost of converting the input data from float to double. The cost of this conversion is more expensive with AVX2 instructions than with SSE instructions[49].

5.3.2 . Precision

half C++ library

The half precision floating point format is not fully supported in most current computer hardware. For this reason, emulation is necessary in order to test the numerical performance of half precision algorithms. One option for emulation is the `half` C++ library[87][111], which implements a IEEE compliant half precision storage type. This library contains the functions for converting between half-

Operation	Naive	Kahan	Iterative	Cascade	Upcast
Add/Sub	N	4N	N	N	N
Div	1	1	N	N∗	1
Cast	0	0	0	0	N

∗ Most are substituted with multiplication operations

Table 5.1: Total number of operations per algorithm for an input of N values.

precision and architecturally supported types. Arithmetic operations are performed by converting to floating point, using native floating point arithmetic, then converting back to the half precision type for storage. Because the operations are performed in floating point precision, there may be a difference in numerical precision between the `half` C++ library and fully supported native half precision arithmetic.

```
#include <half.hpp>
int main(void) {
  using half = half_float::half
  half a = 10, b = 1.25;
  std::cout << a + b << std::endl;
  return 0;
}
```

Listing 5.1: Example of a C++ code using the half library.

Experimental Setup

Precision benchmarks were performed in C++ using the `half` header-only library to simulate IEEE compliant binary16 half-precision floating point numbers. Various example cases were tested in order to test the limitations of the various algorithms. Most of the example inputs have easy to calculate averages, but are unlikely to occur in practice. However, the random input could represent a potential real world case and the image input does represent a real world case.

Table 5.2 summarizes the results of the experiments performed. Any entry labelled n/a describes an instance where the result was either `inf` due to overflow in the case of the naive average or `nan` in the case of the kahan sum. The kahan summation results in `nan` rather than `inf` when overflowing due to line 7 of Algo 14. When the "sum" term overflows to `inf`, line 7 results in an "`inf - inf`" operation with a result of `nan`.

Overall, it appears that the biggest advantage of the Cascading Average method is that it avoids the catastrophic failure that the other methods are susceptible to. The sum then divide methods are susceptible to overflow issues that arise when $N\bar{a} > 65504$, while the iterative method fails for a sufficiently large number of elements because it fails to update the average when $A[i]/i < 1$ULP.

Input	N	Naive	Kahan	Iterative	Cascade
Increasing	100	9	1	0	0
+1	1000	n/a	n/a	0	0
	10000	n/a	n/a	993	1
Increasing	100	16	1	20	2
+0.001	1000	198	1	249	4
	10000	n/a	n/a	1486	4
Decreasing	100	9	1	0	0
-1	1000	n/a	n/a	0	0
	10000	n/a	n/a	993	1
Fixed Ratio	100	1	1	17	0
/(N/2)	1000	n/a	n/a	3	0
	10000	n/a	n/a	994	1
Fixed Diff	100	13	1	23	0
+(N/2)	1000	n/a	n/a	227	0
	10000	n/a	n/a	737	0
Rand1	1000	271	0	249	3
Rand2	1000	n/a	n/a	261	4
Rand3	1000	n/a	n/a	246	2
Rand4	1000	n/a	n/a	253	3
Fixed	1000	152	0	0	0
10	10000	1070	n/a	0	0
	100000	1259	n/a	0	0
	1000000	1277	n/a	0	0
+/-1	300	17	0	74	1
10-12	3000	709	1	128	1
	30000	1998	n/a	128	1
	300000	1401	n/a	128	1
Image	2073600	n/a	n/a	1037	3
	2073600	1761	391	1701	7

Table 5.2: Precision benchmark results. Results marked n/a indicate that the trial gave an invalid result (inf or nan).

It should be noted that for a sufficiently small input value range, the naive method will never visibly overflow. It will instead reach a point where $sum + a_i = sum$ because there are no a_i larger than one ULP. The Kahan accumulation method is much less likely to suffer this silent absorption failure due to the presence of a remainder variable which will eventually grow larger than 1 ULP.

5.3.3 . Addendum

Since completing the works presented here, the author has discovered that the `half` library used to emulate half precision floats allows temporary overflow and underflow due to its usage of expression templates. The library performs its computations using 32-bit floats, then reduces the results to 16-bit floats when assigning values. This means that any intermediate values during the computation are represented using 32-bit floats and not 16-bit floats. As a result, some calculations that should overflow do not actually overflow. This is evidenced by the difference between Algorithm 19 and Algorithm 20.

```
1  half a = 65504;
2  half b = 65504;
3  half c = a + b;
4  c = c / 2;
5  std::cout << c << "\n"
    ;
6  >>> inf
```

Algorithm 19: Usage of `half` that overflows

```
1  half a = 65504;
2  half b = 65504;
3  half c = (a + b) / 2;
4
5  std::cout << c << "\n"
    ;
6  >>> 65504
```

Algorithm 20: Usage of `half` that avoids overflowing

One way to circumvent this problem is to modify the Cascading Average algorithm to halve the entries being averaged before adding them, rather than adding before dividing. This would impose a small performance cost due to the extra division operation at every loop iteration, but avoid overflow issues. A side effect would be that very small input values are rounded to zero when divided. However, the effect of rounding small values to zero is much less significant than the effect of overflowing to `inf`.

6 - Fixed Precision Extension of nsimd

One means of accelerating and compressing neural networks being researched is the use of fixed-point formats[70][97]. However, experiments with fixed-point arithmetic tend to either neglect runtime performance, or be limited to FPGA applications with hardware implementations of fixed-point arithmetic. One reason for this may be the lack of a flexible, high performance fixed-point software library for CPUs.

6.1 . Fixed Point Format

Fixed-point arithmetic offers a few advantages over floating point arithmetic while presenting a few challenges of its own.

One of the main advantages of fixed point arithmetic compared to floating point is that it is simpler. It uses integer operations, which tend to be faster than floating point operations (previously shown in Table 1.1). In addition, the range is fixed, so there are no alignment concerns. This means that there are no problems from absorption or cancellation.

The fixed range is also a disadvantage, as it increases the risk of overflow. In addition, fixed point arithmetic is generally not explicitly hardware-supported. While some CPUs may include barrel-shifters to facilitate the implementation of fixed-point arithmetic, they do not include explicit fixed-point arithmetic units. As a result, fixed-point arithmetic must be emulated using integer operations. This reliance on integer operations creates a significant challenge – CPUs do not provide SIMD integer division. As a result, the division operation is to be avoided as much as possible when designing fixed-point algorithms.

As previously described in Section 3.0.3, this chapter will use the Qa.b format to describe fixed point numbers.

Fixed Point Arithmetic It should be noted that some of the basic arithmetic operations (add, sub, mul, div) have certain effects on the precision of fixed point numbers. This is especially important to keep in mind because fixed point arithmetic must be emulated using integers, as it is not natively supported by processors.

When fixed point numbers are contained within integers, a Qa.b number F is stored inside of an integer I. We may consider $I(x)$ to be the underlying integer representation of the fixed point value $F(x)$. Mathematically, the relationship between the two is

$$F_{a.b}(x) = I(x) \cdot 2^{-b}$$

From this, we can derive the properties of the basic arithmetic operations.

Addition and subtraction have the same straightforward properties.

$$F_{a.b}(x_1) + F_{a.b}(x_2) = I(x_1) \cdot 2^{-b} + I(x_2) \cdot 2^{-b} = [I(x_1) + I(x_2)] \cdot 2^{-b}$$

The result of addition or subtraction between two Qa.b numbers is another Qa.b number.

However, multiplication is not as straightforward

$$F_{a.b}(x_1) \cdot F_{a.b}(x_2) = [I(x_1) \cdot 2^{-b}] \cdot [I(x_2) \cdot 2^{-b}] = [I(x_1) \cdot I(x_2)] \cdot 2^{-2b}$$

Numerically, the result of a multiplication between two Qa.b numbers is a Q2a.2b number.

Division is not a simple matter either.

$$F_{a.b}(x_1) \div F_{a.b}(x_2) = [I(x_1) \cdot 2^{-b}] \div [I(x_2) \cdot 2^{-b}] = I(x_1) \div I(x_2)$$

Numerically, the result of a naive division between two Qa.b numbers is a Qa.0 number – there are 0 bits of decimal precision.

It should be noted that the multiplication operations output a larger number of decimal bits than the input formats (2b vs b bits). This means that a fixed point implementation must choose a rounding method when eliminating the extra bits. The simplest (and fastest) rounding method to implement is rounding down via truncation. A number of other methods exist, including but not limited to rounding up, rounding to even, and rounding to nearest (with various tie breakers). However, all fixed point implementations examined in the following section use the same truncation rounding method.

Section 6.5.2 will describe these problems and their solutions in greater detail. It is sufficient to describe the properties of the basic arithmetic operations, as all other functions provided will be constructed using these arithmetic operations.

6.2 . Fixed Point Software Libraries

A fixed point software library was needed in order to be able to perform some of the experiments presented in Chapter 7. Table 6.1 compares a few such libraries on the following desired qualities:

- **Math** - Whether the library contains math functions.
- **Vectorized** - Whether the library performs explicit vectorization of the provided functions.
- **Arbitrary Qa.b** - Whether the library supports arbitrary integer and decimal sizes when declaring fixed point types.
- **API** - The type of API offered by the library.

Library	Ref	License	Math	Vectorized	Multiple	Arbitrary Qa.b	API
liquid-fpm	[33]	GPL	Y	N	N	Y∗†	C Macros
libfixmath	[38]	MIT	Y	N	N	N	C99
CNL	[71]	Boost	Y	Y§	Y	Y‡	C++ templates
nsimd	[91]	MIT	Y	Y	Y	Y∗	C++ templates

∗ Extra bits are not zeroed during computation.
† a + b must equal 16 or 32.
‡ a is not actually configurable. It consumes all bits not used by b.
§Uses the deprecated boost.simd library.

Table 6.1: Comparison of features in various C++ libraries with fixed point support.

- **Multiple** - Whether the library supports multiple different formats within one code
- **Rounding** - All libraries use truncation as their default rounding mode for multiplication operations. As such, we do not consider it to be a distinguishing feature.

liquid-fpm is a submodule of the free and open-source liquid-dsp software library for digital signal processing. It leverages C macros in order to provide a flexible C interface for fixed point types of varying integer/decimal sizes. However, each integer/decimal combination must be defined ahead of time and the user must modify a header file in order to compile the corresponding dynamic library. As a result, it natively supports a maximum of two Qa.b formats – one 16-bit format and one 32-bit format. In addition, it only supports formats such that a + b is equal to either 16 or 32.

libfixmath is a free software library for Q16.16 fixed point numbers. It provides algorithms specifically designed for Q16.16 numbers and does not support any other fixed point formats.

CNL , or the Compositional Numeric Library, is a C++ library for fixed point numbers. It allows some, but not full flexibility in the allowed Qa.b sizes. Fixed point types are declared with template arguments for the base storage type and the location of the decimal point. However, the fixed point types will always fill the entire base type, limiting the possible supported integer/decimal combinations.

nsimd is a software library that simplifies SIMD programming. This chapter presents a fixed point library developed as a submodule of nsimd for the purpose

of meeting the needs described above. The fixed point submodule leverages C++ metaprogramming to allow for arbitrary integer/decimal combinations (provided they fit within a maximum of 64 bits). In addition, it is developed with vectorization in mind, as opposed to the other libraries which focus on scalar computations. The Qa.b definition is not strict – if the sum of a and b is less than the size in bits of a native integer type, the remaining bits can nonetheless contribute extra precision to computations as additional integer bits. However, nsimd clears these extra bits when performing comparison operations.

6.3 . nsimd

The **nsimd** C++ library mostly solves the problems presented in section 1.7.1. As shown in Algo. 21, it provides a single interface that allows one to write SIMD code that is portable between architectures and a set of functions that account for the memory alignment requirement. Code written using **nsimd** has a level of verbosity between that of scalar code and that of SIMD code written in assembly.

Algorithm 21 Using nsimd to write portable SIMD code

```
1  // SIMD (All architectures)
2  int align_beg = A.data() + (A.data() %
    NSIMD_MAX_ALIGNMENT);
3  int align_end = A.data() + A.size()
4              - (( A.data() + A.size() ) %
    NSIMD_MAX_ALIGNMENT);
5  for (int i = 0; i < align_beg; ++i)
6    A[i] = B[i] * C[i];
7  for (int i = align_beg; i<align_end; i+=nsimd::
    len(float()))
8    ns::pack<float> b = ns::load( &(B[i]) );
9    ns::pack<float> c = ns::load( &(C[i]) );
10   ns::pack<float> a = b * c;
11   ns::store( &(A[i]) , a );
12 for (int i = align_end; i < A.size(); ++i)
13   A[i] = B[i] * C[i];
14
```

The fixed point library we developed was implemented as a module of nsimd, so we must first describe how to use nsimd before approaching the fixed point library.

6.3.1 . Usage

C/C++/Advanced C++ APIs

As of this writing, nsimd provides 4 APIs – C89, C++98, C++11, and C++14 interfaces – providing varying levels of ease and style of use in terms of types provided and function calls. Each API contains all features provided by the APIs preceding it. For example, the C++98 API contains the C89 features and the C++14 API provides all features from all APIs. The C++98 API will be referred to as the basic C++ and the C++11 and C++14 APIs will be described together as the advanced C++ API, as the C++14 API only provides a few minor improvements over the C++11 API.

Includes To use the C89 and basic C++ APIs of nsimd, one simply needs to

```
#include <nsimd/nsimd.h>
```

This will automatically include all functions provided by nsimd. The advanced C++ API instead requires

```
#include <nsimd/cxx_adv_api.hpp>
```

Both the basic and advanced C++ APIs provide all types and functions inside of the `nsimd` namespace.

C89 Types For ease and consistency of use, nsimd provides aliases for the native built-in C scalar types, as well as the emulated half-precision floating point type (f16). A full list of the provided types (as of this writing) is provided in Table 6.2. For each provided scalar type (`{i/u/f}N`), nsimd provides a type corresponding to a SIMD register of that type (`v{i/u/f}N`)and a type corresponding to a SIMD register of logical values for the scalar type (`vl{i/u/f}N`). These logical values are the result of SIMD logical operations, such as `equals` or `greater than` operations. One example use case is the `if_else1` function, which outputs a result that depends on whether the logical values for each entry correspond to true or false. While these logical values can be considered equivalent to boolean true of false values, their bitwise representation can vary depending on the SIMD architecture in use.

	Signed Integer				Unsigned Integer				Float		
Scalar	i8	i16	i32	i64	u8	u16	u32	u64	f16	f32	f64
Vector	vi8	vi16	vi32	vi64	vu8	vu16	vu32	vu64	vf16	vf32	vf64
Logical	vli8	vli16	vli32	vli64	vlu8	vlu16	vlu32	vlu64	vlf16	vlf32	vlf64

Table 6.2: Types provided by the nsimd C89 API

Advanced C++ Types The advanced C++ APIs allow for greater control of low level details though the templated `pack` and `packl` structures. For basic usage, a `pack` only needs a type, such as `i8` or `uint32_t`, as a template option. Here are a few examples of variable declarations using the `pack` interface.

```
nsimd::pack<i32> v1;
nsimd::pack<uint64_t> v2;
nsimd::packl<float> v3;
```

As in the C89 API, a `pack<type>` represents a SIMD register of the corresponding type.

C89 Functions When calling an nsimd function using the C API, it is necessary to be aware of the target SIMD architecture when calling functions. This is because all functions take the form `nsimd_{function}_{extension}_{type}`. A few examples of function calls are listed below.

```
vf64  r1 = nsimd_add_cpu_f64( vf64 a1 , vf64 a2 );
vli32 r2 = nsimd_lt_avx2_i32( vi32 b1 , vi32 b2 );
vu16  r3 = nsimd_shl_aarch64_u16( vu16 c1 , int c2 )
 ;
```

Basic C++ Functions The basic C++ API for calling functions is simpler, thanks to function overloads – it is possible to have multiple versions of a function with the same name but different arguments. The SIMD extension is automatically inferred from the information provided by the input types (`vf64`, `vu16`, etc.). However, the type must still be specified when calling a function, as shown below.

```
vf64  r1 = nsimd::add( vf64 a1 , vf64 a2 , f64() );
vli32 r2 = nsimd::lt ( vi32 b1 , vi32 b2 , i32() );
vu16  r3 = nsimd::shl( vu16 c1 , int  c2 , u16() );
```

Advanced C++ Functions The advanced C++ API is simpler to use than the basic C++ API, as the `pack` structure provides all of the information the compiler needs to deduce the correct function to call, as shown below.

```
nsimd::pack<f64> r1 = nsimd::add( nsimd::pack<f64>
 a1 , nsimd::pack<f64> a2 );
nsimd::pack<int> r2 = nsimd::lt ( nsimd::pack<int>
 b1 , nsimd::pack<int> b2 );
auto r3 = nsimd::shl( nsimd::pack<u16> c1 , int c2 )
 ;
```

The two exceptions to this is the `load` type functions, which are unfortunately common. These two functions require the type to be explicitly specified, as show below.

```
nsimd::pack<f64> r1 = nsimd::loadu<nsimd::pack<f64
 >>( f64* p1 );
```

Operator Overloading The advanced C++ API overloads many common operators for arithmetic, bitwise operations, and logical operations for the pack structure. This allows the user to treat packs similarly to native variables when writing code. Below are a few lines demonstrating valid code that uses these overloads.

```
nsimd::pack <int> r1, r2, r3;
nsimd::packl<int> l1, l2, l3;
r3 = r1 + r2; // Addition
r3 = r1 * r2; // Multiplication
r3 = r1 / r2; // Division
r3 = r1 & r2; // Bitwise And
l3 = r1 > r2; // Comparison
```

Loop Unrolling The pack structure provided by the advanced C++ API allows for automatic loop unrolling via a second template argument. It can be declared with an integer value as a second template argument, representing the amount of unrolling to do. The example below unrolls the loop by a factor of 3.

```
for ( int i = 0 ; i < size ; i += nsimd::len( int()
 ) ) {;
  nsimd::pack<int,3> r1 = nsimd::loadu<nsimd::pack<
 int,3>>( *p1 );
  nsimd::pack<int,3> r2 = nsimd::loadu<nsimd::pack<
 int,3>>( *p2 );
  r1 = nsimd::sin( r1 * r2 );
}
```

Instructions for installing, navigating, and compiling nsimd are provided in Annex B.

6.3.2 . Developers - Adding a New Function

This section details how to add a new function to nsimd by walking through the steps involved using an example function that was added for use by the fixed-point module. All code will be written to various files in the `nsimd/egg` directory. The process involves adding the function as an 'operator', writing the code that generates the platform-specific code, and specializing (if necessary) the code that generates unit tests for the function. The function that will be used as an example is the `clz` function, which takes an integer as input and returns the number of bits set to zero above the strongest set bit.

Operator

The first step is to add a class representing the function to the `operators.py` file and fill it appropriately. The class must contain the following members:

- `full_name` - short description of the function.
- `signature` - signature that the function will have. Of the most common signature types, `v` represents a vector, `l` a vector of logicals, `b`, `p` an integer, and `s` a scalar.
- `categories` - the categories contain the documentation categories where the function should be listed.

In addition, the class may optionally contain the following members:

- `desc` - full description of what the function does.
- `types` - types which the function can take as input.
- `domain` - the input domain.
- `do_bench` - set to `False` if the function is not to be automatically benchmarked.
- `tests_ulps` - sets the allowed ULP tolerance of error in the unit tests.
- `bench_auto_against_X` - benchmark comparable function from X library for comparison.
- `tests_mpfr` - when performing unit tests, compare to the mpfr library.
- `cxx_operator` - C++ operator to overload with a call to this function.

As an example, the `clz` code is listed below.

```
class Clz(Operator):
    full_name = 'count leading zeroes'
    signature = 'v clz v'
    domain = Domain('R')
    categories = [DocMisc]
    types = common.iutypes
    desc = 'Count number of zero bits before the
   largest set bit'
```

The signature is set to `'v clz v'` because the function takes a vector as input and outputs another vector. In addition, the types are set to `common.iutypes` in order to limit the inputs to signed and unsigned integers.

Platforms

In order to generate code for the function to call, it is necessary to add the function to the various platform files. As of this writing, there are three such files – `platform_cpu.py` for CPUs with no SIMD extensions, `platform_x86.py` for x86 CPUs (SSE/AVX), and `platform_arm.py` for ARM CPUs (NEON/SVE). For each platform, there are three modifications that must be made per functions.

The first modification is to write the function capable of generating the platform-specific C code. This function will be given the SIMD extension (unless CPU) and the input type as function inputs and return a string containing the appropriate C/ASM code. Each platform provides its own `fmtspec` variable which contains a number of useful predefined strings. For example, here are a few of the predefined strings provided in `platform_arm.py`

- `simd_ext` : raw SIMD extension (aarch64, neon, sve)
- `typ` : input type
- `styp` : ARM register type corresponding to input type (eg. `in16x8_t` or `float32x4_t`)
- `to_typ` : output type
- `suf` : ARM function suffix corresponding to input type (eg. the `_s32` part of `vadd_s32`)
- `in0` : first input (`in1` - `in6` are also provided if needed)
- `typnbits` : total number of bits in the input type (eg. 64 for double, 8 for char)

These helpers contained in `fmtspec` make it possible to write the code below.

```
def clz(simd_ext, from_typ):
      if from_typ in ['i8', 'u8', 'i16', 'u16', 'i32', 'u32']:
        return '''\
        return vclzq_{suf}( {in0} );
        '''.format(**fmtspec)
      else:
        return emulate_op1('clz', simd_ext, from_typ)
```

The `suf` variable makes it easy to write a code that is generic for all input types. In cases where a function does not have a corresponding intrinsic, nsimd provides the `emulate_op1` function (among others), which generates the code necessary to call the scalar (CPU) version of the function being generated.

The next modification is to add an appropriate line to the `impls` array at the end of the file. This array contains a list of the functions to be implemented and

the functions they call to generate their C code. For the CPU `clz` function, this means adding the following line to the list.

```
'clz' : lambda : clz(from_typ),
```

Meanwhile, the x86 and ARM function calls also require information about the SIMD architecture being used.

```
'clz' : lambda : clz(simd_ext,from_typ),
```

The final modification is to update the header files that are needed by the function. This is done by adding to the `get_additional_include` function. If the code generated by the first step does not call any other nsimd functions, this step is not needed. For example, the CPU `clz` calls the nsimd `shrv` function, so the following lines of code are needed.

```
    elif func == 'clz':
        return  '''
                #include <nsimd/cpu/cpu/shrv.h>
                '''

```

Testing

By default, nsimd will generate unit tests for a function that compare the CPU outputs to the SIMD outputs for randomly generated inputs. If the SIMD output differs from the CPU output, the test will fail. For many functions, this is sufficient. However, some functions may have a limited input domain or special characteristics that may cause these outputs to differ for certain inputs.

The `clz` function falls in the latter category – for an input of 0, the output may depend on the compiler being used. This is because the CPU implementation calls compiler built-in functions when possible; these built-in functions do not all behave exactly the same for an input of 0. As a result, we wish to avoid inputs of 0 when testing, so we modify the code that generates the random inputs by adding the following lines in the appropriate location.

```
        if op.name == 'clz':
            vin_rand = 'vin1[i] = rand();\nwhile ( !
   vin1[i] ) vin1[i] = rand();'
```

This modifies the random input generation function to avoid outputting any 0 inputs.

Functions Added

A small number of functions needed to be added to the nsimd library in order to implement the fixed point algorithms presented in Section 6.5.

Architecture	i8	u8	i16	u16	i32	u32	i64	u64
x86 AVX	N	N	N	N	N	N	N	N
x86 AVX2	N	N	N	N	N	N	N	N
x86 AVX512	N	N	N	N	Y∗	Y∗	Y∗	Y∗
NEON	Y	Y	Y	Y	Y	Y	N	N
SVE	Y	Y	Y	Y	Y	Y	Y	Y

∗ Limited to Skylake architecture and above

Table 6.3: `clz` intrinsics supported by various architectures.

clz The `clz`, or count leading zeroes, is illustrated in Figure 6.1. This function takes integer types as inputs and returns the number of zero bits preceding the strongest bit of the input.

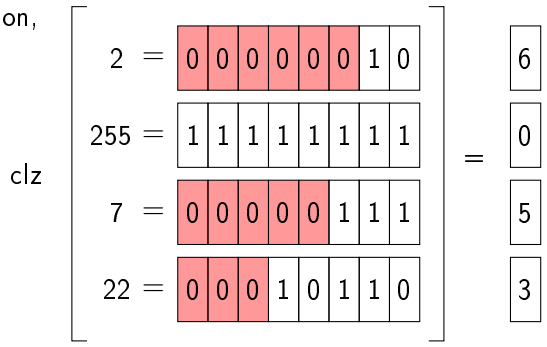

Figure 6.1: Example of the `clz` function.

Depending on the implementation, the output of the `clz` function may be unspecified for an input of zero. However, some implementations do make the effort to output the total number of bits in the input (eg. 32 for a 32-bit integer). In the nsimd implementation of `clz`, we choose not to make this effort. For performance reasons, the output of nsimd's `clz(0)` is the same as `clz(1)` − 31.

The scalar `clz` implementation attempts to use the equivalent compiler built-in function if available. Otherwise, variations of Algorithm 22 (adapted from [43, p. 97] and [106]) are used. The algorithm varies in length depending on the size (in maximum bits) of the input. However, the computations are essentially the same. Algorithm 22 performs a binary search for the location of the first nonzero bit in `x` while simultaneously updating the result value in `r`.

For the SIMD implementation of `clz`, we attempt to take advantage of intrinsic functions when available (see Table 6.3). At the moment, when intrinsics are not available the `clz` function falls back to the scalar implementation. This means that it will store the contents of the SIMD register, use the scalar `clz` function, and reload the results into an SIMD register. Document [5] was used to determine availability of intrinsics on the ARM SVE architecture.

shlv The `shlv`, or shift left variably, function is illustrated in Figure 6.2. It takes as input a register of integers to be shifted and a register of integer values listing the magnitude of the shift. The output is thus each input integer shifted left by

Algorithm 22 Algorithm used by nsimd to compute the `clz` function for 8 bit integers.

```
int8_t clz( int8_t arg0 ) {
  int8_t x = arg0;
  int8_t q, r;

  if ( x > 0xF ) q = 1;
  else           q = 0;
  q = q << 2;
  x = x >> q;
  r = q;

  if ( x > 0x3 ) q = 1;
  else           q = 0;
  q = q << 1;
  x = x >> q;
  r = r | q;

  r = r | ( x >> 1 );
  r = 7 - r;
  if ( r < 0 ) r = 0;

  return r;
}
```

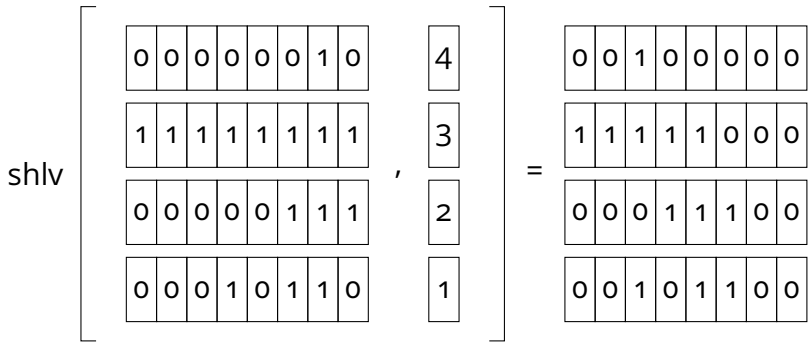

Figure 6.2: Example of the `shlv` function.

Architecture	i8	u8	i16	u16	i32	u32	i64	u64
x86 AVX	N	N	N	N	N	N	N	N
x86 AVX2	N	N	N	N	Y	Y	Y	Y
x86 AVX512	N	N	Y	Y	Y	Y	Y	Y
NEON	Y	Y	Y	Y	Y	Y	Y	Y
SVE	Y	Y	Y	Y	Y	Y	Y	Y

Table 6.4: `shlv` intrinsics supported by various architectures.

the corresponding magnitude.

The `shlv` function simply performs bitwise left shifts on the input register according to the magnitude register.

The scalar `shrv` implementation reuses code from the `shl` function which shifts all values by the same magnitude. However, it requires a slight modification in order to shift the inputs by different magnitudes. Algorithm 23 shows the scalar code generated for 16-bit unsigned integers.

Algorithm 23 Algorithm used by nsimd to compute the `shlv` function for 16 bit unsigned integers.

```
1 nsimd_cpu_vu16 nsimd_shlv_cpu_u16(nsimd_cpu_vu16
    a0, nsimd_cpu_vu16 a1) {
2   nsimd_cpu_vu16 ret;
3   ret.v0 = (u16)(a0.v0 << a1.v0);
4   ret.v1 = (u16)(a0.v1 << a1.v1);
5   ret.v2 = (u16)(a0.v2 << a1.v2);
6   ret.v3 = (u16)(a0.v3 << a1.v3);
7   return ret;
8 }
```

For the SIMD implementation of `shlv`, we attempt to take advantage of intrinsic functions when available (see Table 6.4). At the moment, when intrinsics are not available the `shlv` function falls back to the scalar implementation. This means that it will store the contents of the SIMD register, use the scalar `shlv` function, and reload the results into an SIMD register.

shrv The `shrv`, or shift right variably, function is illustrated in Figure 6.3. It takes as input a register of integers to be shifted and a register of integer values listing the magnitude of the shift. The output is thus each input integer shifted right by the corresponding magnitude.

If signed integers are given to the `shrv` function, it will perform arithmetic shifts which preserve the sign of the input. For unsigned inputs, the `shrv` simply

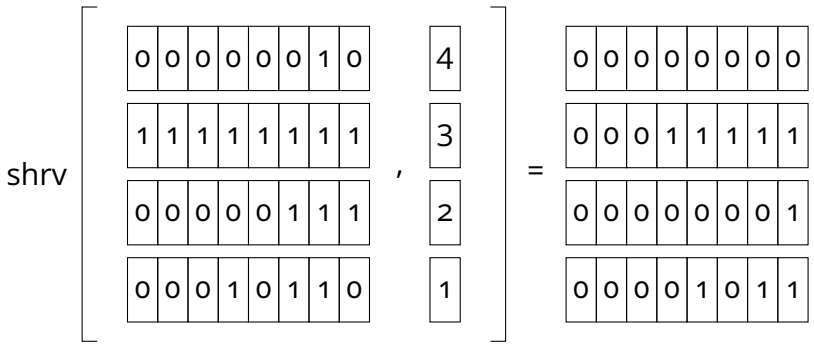

Figure 6.3: Example of the `shrv` function.

Architecture	i8	u8	i16	u16	i32	u32	i64	u64
x86 AVX	N	N	N	N	N	N	N	N
x86 AVX2	N	N	N	N	Y	Y	N	N
x86 AVX512	N	N	Y	Y	Y	Y	Y	Y
NEON	Y*	Y*	Y*	Y*	Y*	Y*	Y*	Y*
SVE	Y	Y	Y	Y	Y	Y	Y	Y

* Supported via left shift by negative values

Table 6.5: `shrv` intrinsics supported by various architectures.

performs a bitwise shift operation.

The scalar `shrv` implementation reuses code from the `shr` function which shifts all values by the same magnitude. However, it requires a slight modification in order to shift the inputs by different magnitudes. Algorithm 24 shows the scalar code generated for 64-bit signed integers. Note that lines 11-14 serve to preserve the sign of the input. For unsigned inputs, the computation consists of a single shift operation – line 16 without the masking operation.

For the SIMD implementation of `shrv`, we attempt to take advantage of intrinsic functions when available (see Table 6.5). At the moment, when intrinsics are not available the `shrv` function falls back to the scalar implementation. This means that it will store the contents of the SIMD register, use the scalar `shrv` function, and reload the results into an SIMD register. When comparing Table 6.5 to Table 6.4, one may note that levels of support are not identical depending on the direction of the shift.

div The `div` function performs division. Most architectures do not provide SIMD instructions for integer division, as seen in Table 6.6. As a result, vectorized integer division must be implemented through software.

Here we present the vectorization of Algorithm 25 labeled Algorithm D in [62] and coded in C in [43]. This algorithm performs the binary version of the decimal long division algorithm commonly taught in elementary school. Lines 9-

Algorithm 24 Algorithm used by nsimd to compute the `shrv` function for 64 bit signed integers.

```
nsimd_cpu_vi64 nsimd_shrv_cpu_i64(nsimd_cpu_vi64 a0, nsimd_cpu_vi64 a1) {
  union {
    i64 i;
    u64 u;
  } val;

  nsimd_cpu_vi64 ret;
  u64 mask;

  /* ------------------------------------------- */
  const int shift0 = 64 - 1 - a1.v0;

  val.i = a0.v0;
  mask = (u64)((val.u >> (64 - 1)) * ~(u64)(0) << shift0);

  ret.v0 = (i64)((val.u >> a1.v0) | mask);

  return ret;
}
```

12 of Algorithm 25 show the only portion of the algorithm which is not trivially parallelizable. Despite this, it is still possible to accelerate this algorithm via SIMD instructions. Algorithm 25 is only valid for positive inputs. Signed integers are converted to positive unsigned integers prior to the actual computation, then the result is given the appropriate sign before being output.

Note at line 5 of Algorithm 25 that the number of loop iterations is equal to the number of bits in the input type. This is the true limiting factor in how well integer division can be efficiently vectorized using this algorithm. Doubling the size of an input size quadruples the time needed to perform this integer division – half as many elements fit into an SIMD register and twice as many loop iterations are needed.

One peculiarity of this algorithm is that the result accumulator `result` needs no initialization. This is because each iteration of the loop updates the weakest current bit, then shifts the value left (multiplying by two). The current value of the `result` is never used for other computation. After the full number of iterations (equal to the number of total bits) have completed, none of the original uninitialized bits remain.

Algorithm 25 Scalar algorithm for bitwise long division of integers.

```
1  template<typename T> bitwise_long_division( T
   top , T bot ) {
2    T result;
3    T remainder = 0;
4
5    for ( int i = (8*sizeof(T)-1) ; i >= 0 ; --i
   ) {
6      result = result << 1;
7      remainder = remainder << 1;
8      remainder = remainder | ( ( top >> i ) & T
   (1) );
9      if ( remainder >= bot ) {
10       remainder = remainder - bot;
11       result = result | T(1);
12     }
13   }
14   return result;
15 }
```

Architecture	i8	u8	i16	u16	i32	u32	i64	u64
x86 AVX	N	N	N	N	N	N	N	N
x86 AVX2	N	N	N	N	N	N	N	N
x86 AVX512	N	N	N	N	N	N	N	N
NEON	N	N	N	N	N	N	N	N
SVE	Y	Y	Y	Y	Y	Y	Y	Y

Table 6.6: `div` intrinsics supported by various architectures.

Benchmarks

This section presents benchmarks performed on the `clz`, `shrv`, `shlv`, and `div` functions. All benchmarks present the speedup between the scalar CPU implementations and the SIMD implementations of the functions within nsimd. We choose not to perform any benchmarks on architectures of Intel AVX or lower, as they do not support any instructions for the implemented functions.

Raw benchmark data is provided in Annex C. Each graph presented contains a link to a table of raw data. Each raw data table provides a link back to the corresponding graph, so don't be afraid to click back and forth if you're reading this as a pdf.

Benchmarks were performed using the specifications described in Table 6.7. For convenience, Table 6.8 provides the ideal speedup for each register size.

SIMD	Compiler	SIMD Flag	Register Size	Frequency	RAM
AVX2	gcc 8.2	-mavx2	256	3.2 Ghz	16GB
AVX512	gcc 11.0.0	-mavx512f	512	2.4 Ghz	188GB
AARCH64	gcc 5.4	-marmv8-a	64	1.0 Ghz	126GB

Table 6.7: Specifications used to perform fixed point benchmarks.

clz Figure 6.4 shows the performance of the `clz` function. As expected from Table 6.3, the AVX2 `clz` offers no acceleration. The AVX512 `clz` provides

Register Size (bits)	8 bits	16 bits	32 bits	64 bits
64	8	4	2	1
128	16	8	4	2
256	32	16	8	4
512	64	32	16	8

Table 6.8: Ideal speedups for each combination of register size and data size.

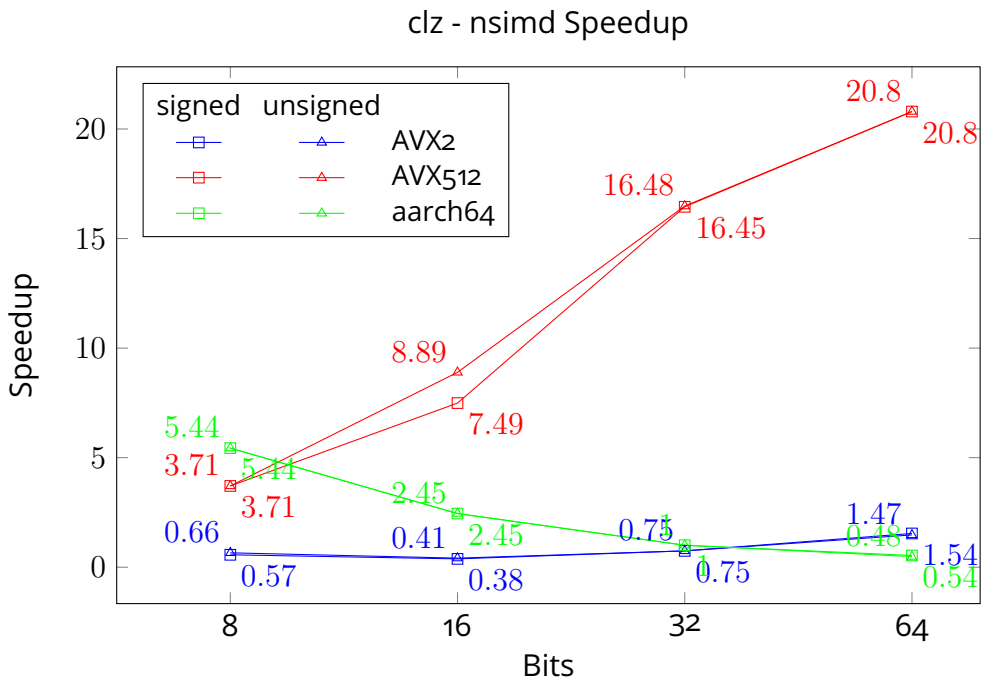

Figure 6.4: Speedup of the SIMD nsimd `clz` function compared to its scalar equivalent. Raw data in Table C.1.

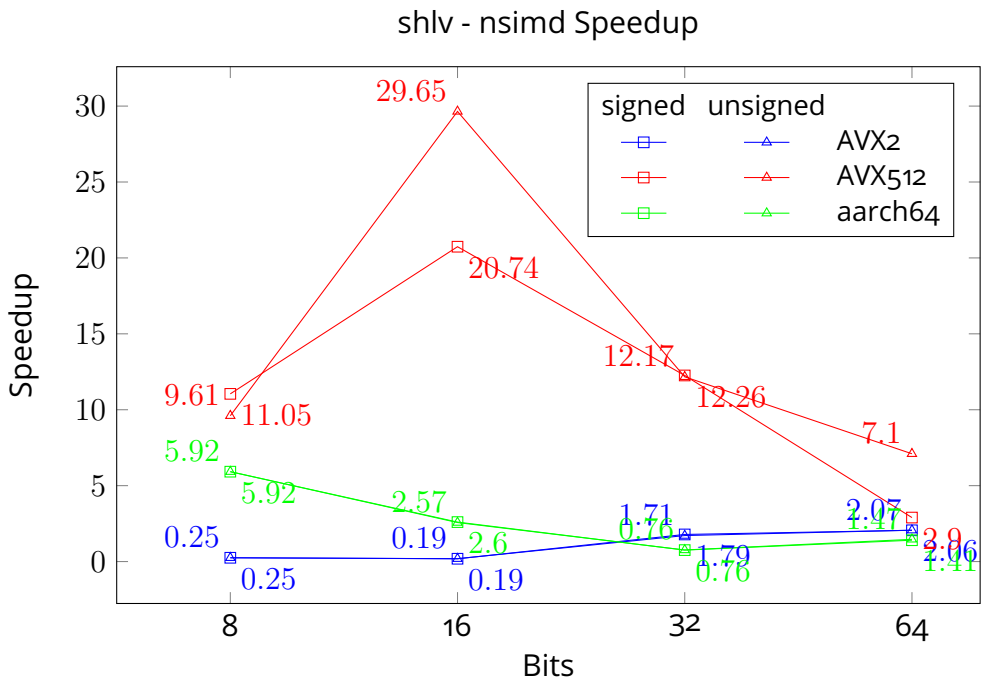

Figure 6.5: Speedup of the SIMD nsimd `shlv` function compared to its scalar equivalent. Raw data in Table C.3.

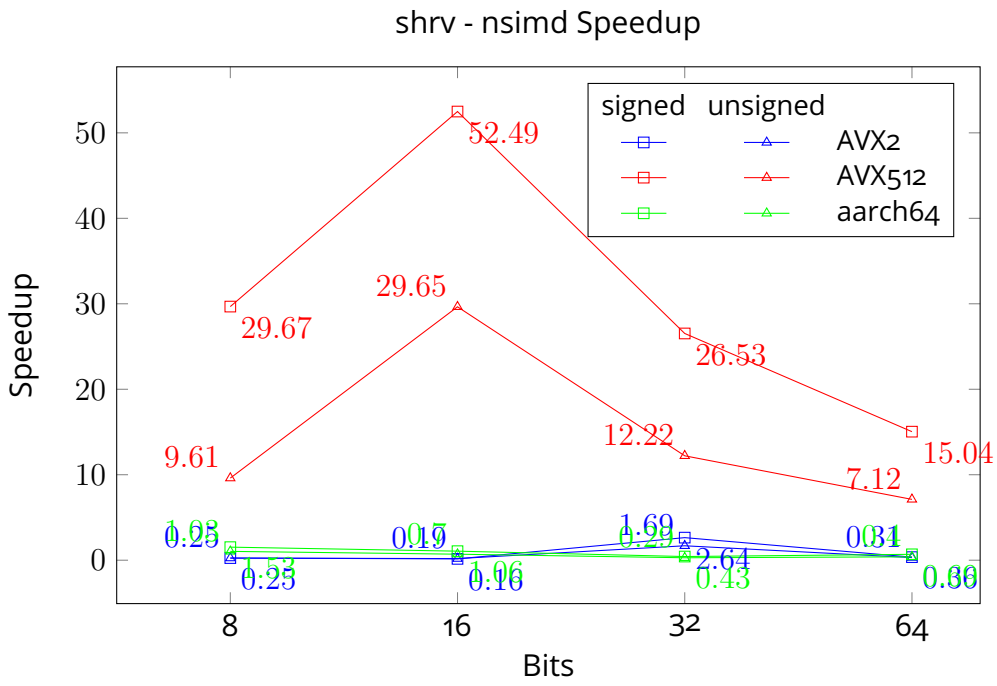

Figure 6.6: Speedup of the SIMD nsimd `shrv` function compared to its scalar equivalent. Raw data in Table C.2.

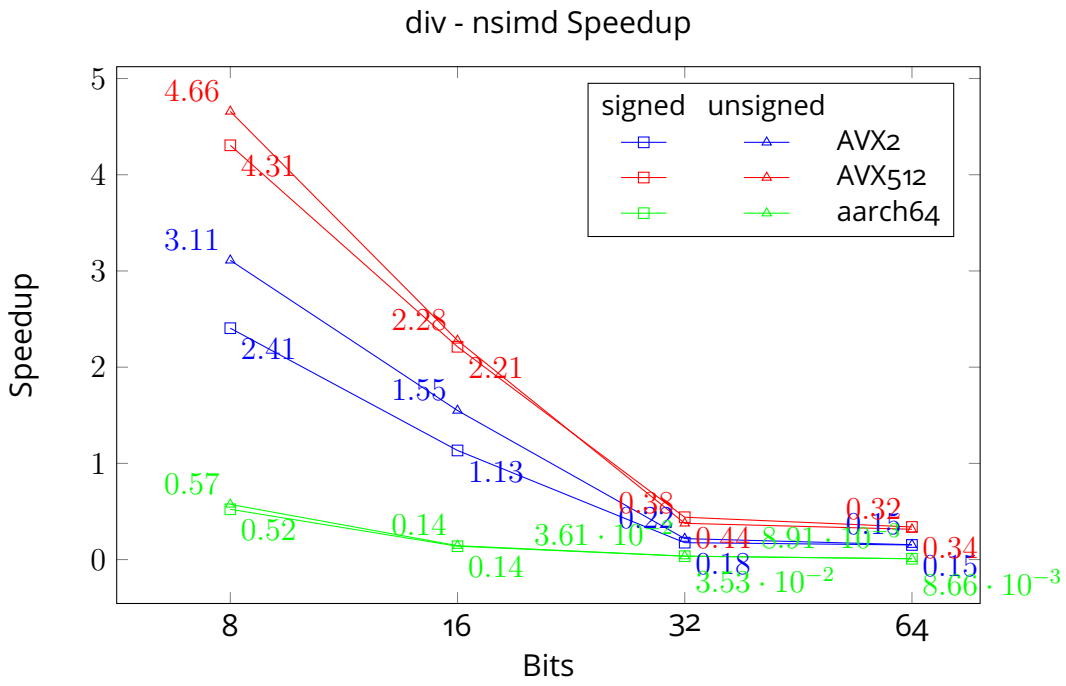

Figure 6.7: Speedup of the SIMD nsimd `div` function compared to its scalar equivalent. Raw data in Table C.4.

the greatest speedup, when supported, thanks to its largest register sizes. This speedup is significantly lower at 8 and 16 bits, as we must convert these to 32-bit types before being able to take advantage of the implemented intrinsic functions. However, the aarch64 `clz` does offer a speedup, although it is not as large as the ideal speedup. This is because the comparison between SIMD and scalar `clz` functions is actually a comparison between a CPU instruction (SIMD) and a compiler builtin function (scalar). It appears that the compiler builtin `clz` function is slightly faster than the SIMD instruction.

shlv Figure 6.5 shows the performance of the `shlv` function. As expected from Table 6.4, the AVX2 `shlv` only shows any speedup when there exists a corresponding CPU function at 32 and 64 bits. At 8 and 16 bits, the cost of storing the data in the heap, performing the scalar implementation, then loading the data back into an SIMD register degrades the performance compared to simply performing scalar computation. We also tested the `shlv` function using type conversions to the supported 32-bit type. However, the performance was equivalent to scalar emulation. For simplicity, we chose to keep the scalar emulation implementation. For the AVX512 `shlv` we see the greatest acceleration, once more thanks to the large register sizes. This speedup is reduced at 8 bits, as we must convert these to 16-bit types before being able to take advantage of the implemented intrinsic functions. The magnitude of the reduction is significantly greater for unsigned 8-bit types, as intrinsic functions only exist for converting signed 8-bit types to 16-bits. The SIMD aarch64 `shlv` scales perfectly linearly. However, the speedup is not so linear due to inconsistencies in the scalar performance. The scalar 32-bit `shlv` is actually faster than the SIMD instruction, while the 8 and 16 -bit scalar performances are similar to each other and the 64-bit scalar performance is significantly slower.

shrv Figure 6.6 shows the performance of the `shrv` function. One immediately visible feature is that, compared to the previous graphs, `shrv` performance differs noticeably between signed and unsigned types. This is because the `shrv` implements an arithmetic shift, requiring extra computation to preserve the sign of signed types. In comparison, the unsigned version can blindly shift using the CPU instruction. On the AVX2 architecture, the corresponding CPU instruction only exists for 32-bit integer types. This shows very clearly in the speedup graph, which only shows an AVX2 speedup at 32 bits. As with the `shlv` function, we use scalar emulation for unsupported types, as the performance is equal to type conversions. For the AVX512 `shrv` we see the greatest acceleration, once more thanks to the large register sizes. This speedup is reduced at 8 bits, as we must convert these to 16-bit types before being able to take advantage of the implemented intrinsic functions. As with the `shlv` function, the `shrv` function scales perfectly linearly on the aarch architecture. Once again, the scalar performance makes the speedup

appear nonlinear. The 32-bit scalar performance is noticeably faster than the others, while the 64-bit is significantly slower.

div Figure 6.7 shows the performance of the `div` function. As expected, the integer division algorithm implemented scales badly with the number of bits in the input type on all tested architectures. On the AVX2 architecture, we see that there is an actual speedup for both 8 and 16 -bit inputs. At 32 and 64 bits, the division algorithm is no longer worthwhile due to its poor scaling. On the AVX512 architecture, we see a more significant speedup at 8 and 16 bits. However, the poor scaling prevents any speedup at 32 and 64 bits. On the aarch64 architecture, the small register size (64 bits) limits the potential speedup. As a result, the algorithm is never parallel enough to recuperate the cost of the additional computations.

The final implementation of integer division in nsimd takes these results into account, applying the vectorized algorithm only where it is worthwhile.

6.4 . nsimd Fixed Point Type

This section describes the fixed point datatype added to nsimd for the purpose of performing fixed point operations. The goal was to provide a simple interface to a fixed point datatype which allows arbitrary Qa.b combinations. We achieve this goal through the use of C++ template metaprogramming.

6.4.1 . API

Base Type The fixed point extension of nsimd provides both a scalar interface and a SIMD interface, as shown below. The scalar interface takes the form `nsimd::fixed_point::fp_t<a,b>`. Meanwhile, there are two SIMD interfaces. The first is the standalone `nsimd::fixed_point::fpsimd_t<a,b>`, while the second makes use of the `nsimd::pack<T>` interface where one simply needs to place the scalar `fp_t` as the template argument `T`. The two SIMD interfaces are functionally equivalent. The `fpsimd_t` exists because the library was developed outside of nsimd and is used internally to represent fixed point SIMD registers. It should be noted that, for reasons described in the following section, a Qa.b number will be stored in the smallest sized integer capable of containing (a + 2b) bits. All `fp_t`s are signed. There is no current implementation of unsigned fixed point numbers.

```
1  using namespace nsimd::fixed_point;
2  fp_t<2,3> a1(3.1);    // Q2.3 (8 bits)
3  fp_t<8,8> a2 = 14.7; // Q8.8 (32 bits)
4  pack<fp_t<4,6>> v1;   // Vector of Q4.6 (16 bits)
5  pack<fp_t<16,16>> v2;  // Vector of Q16.16 (64 bits)
```

Operation	Operator	Scalar (`fp_t`)	SIMD (`pack<fp_t>`)	Example Usage
Add	`+`	Y	Y	`c = a + b`
Sub	`-`	Y	Y	`c = a - b`
Mul	`*`	Y	Y	`c = a * b`
Div	`/`	Y	Y	`c = a / b`
Add to self	`+=`	Y	N	`a += b`
Sub from self	`-=`	Y	N	`a -= b`
Mul self by	`*=`	Y	N	`a *= b`
Div self by	`/=`	Y	N	`a /= b`
Negative self	`-`	Y	N	`c = -a`
Cast to T	`T`	Y	N	`c = int(a)`
Equals	`==`	Y	Y	`if ( a == b )`
Not Equal	`!=`	Y	Y	`if ( a != b )`
Less	`<`	Y	Y	`if ( a < b )`
Less Equal	`<=`	Y	Y	`if ( a <= b )`
Greater	`>`	Y	Y	`if ( a > b )`
Greater Equal	`>=`	Y	Y	`if ( a >= b )`
Bitwise Or	`|`	Y	N	`c = a | b`
Bitwise Xor	`^`	Y	N	`c = a ^ b`
Bitwise And	`&`	Y	N	`c = a & b`
Bitwise Not	`!`	Y	N	`c = !a`

Table 6.9: List of currently overloaded operators to `fp_t` and `pack<fp_t>`.

Operators For ease of development, the fixed point extension of nsimd overloads a number of common operators for both the scalar and SIMD interfaces. A full list of currently supported operators with examples is listed in Table 6.9. As with the nsimd advanced C++ API, this allows the user to treat the fixed point types in a similar manner to native variables when developing code. At the moment, only operations between fixed point numbers of the same format are supported. As the nsimd fixed point extension is still a work in progress, some operators are still awaiting implementation.

Function Calls In addition to the overloaded operators, we also provide a small number of basic mathematical functions. When calling functions, the user does not need to specify the format of the fixed point number – it is automatically deduced through template metaprogramming. See below for some example function calls.

```
1  using namespace nsimd::fixed_point;
2  std::vector<fp_t<2,3>> v1(100);
3  nsimd::pack<fp_t<8,8>> b1, b2, b3;
4  v1[0] = cos( v1[1] );
5  v1[2] = log( v1[3] );
```

$a.b$	$0 < a + 2 \cdot b \leq 8$	$8 < a + 2 \cdot b \leq 16$	$16 < a + 2 \cdot b \leq 32$	$32 < a + 2 \cdot b \leq 64$
Scalar Type	`i8`	`i16`	`i32`	`i64`
Vector Type	`vi8`	`vi16`	`vi32`	`vi64`

Table 6.10: Storage types of valid Qa.b combinations

```
6   v1[4] = exp( v1[5] , v1[6]);
7   v1[7] = exp( v1[5] , 3 );
8   b1 = loadu<fp_t<2,3>>( v1.data() );
9   b2 = sin( b1 );
10  b3 = b2 * b1;
11  storeu( v1.data() , b3 );
```

6.4.2 . Implementation Details

Base Type Behind the scenes, the `fp_t` stores data in the smallest integer that fits twice the size of the decimal portion plus the size of the integer portion, as evidenced by Table 6.10.

$$\text{Minimum storage size(Qa.b)} = a + 2 \cdot b$$

We chose this storage limitation in order to be able to perform multiplication operations without upcasting the integer to a higher size. This provides two major benefits. First, it simplifies the development of vectorized algorithms, as it removes the need to decide when to upcast. Secondly, it provides performance benefits by removing conversions between vectorized data types. The conversion itself has a time cost and the proceeding operations performed using the upcasted type only operate on half as many values as the operations performed using the original type. For simple algorithms, the cost of the conversion can be significant compared to the cost of the actual computations. For more complex algorithms, the reduced potential acceleration leads to a significant performance cost. Section 6.5.2 will describe how the extra bits are used during multiplication and division computations.

We achieve this selective storage size by leveraging SFINAE (Substitution Failure Is Not An Error) via the use of the `std::enable_if` structure. This limits C++ compatibility to C++11 or newer, although the `std::enable_if` structure can be easily reimplemented using earlier C++ standards. Table 6.10 lists the implemented Qa.b combinations and their associated types. We limit support to types that fit in 64 bits, as that is the largest commonly physically supported integer type. If a code sets Qa.b such that $a + 2 \cdot b > 64$, the code will produce an error and not compile.

Helper Types All valid `fp_t` types define the following types for ease of development:

- `value_type` - the base type used to store scalar values.
- `positive_type` - unsigned equivalent of `value_type`
- `logical_type` - the base type used to store scalar logical values.
- `simd_type` - the nsimd type used to store SIMD registers.
- `simd_logical` - the nsimd type used to store SIMD logical registers.

These types can be useful when developing algorithms that use the raw data stored in the `fp_t`.

Alignment Fixed point numbers represented such that the last bit in the containing integer type contains the smallest decimal bit of the fixed point number. Any reserved or unused bits are contained in the strongest bits, as shown in Figure 6.8. This choice of alignment serves to enable the multiplication algorithm described in Section 6.5.2.

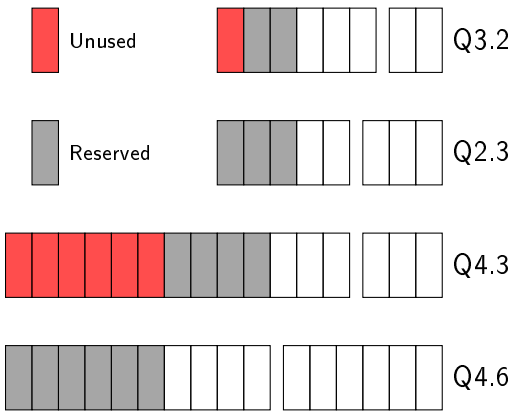

Figure 6.8: Examples of how various Qa.b formats are stored and aligned.

One notable side effect of this alignment is that the number of integer bits available during computations can be effectively larger than the number of integer bits assigned in the type declaration. For example, a `fp_t<4,4>` uses 16 bits of storage, but only requires 12 bits (8 storage + 4 reserved) for any fixed point computations. The remaining 4 bits are not zeroed during computation and thus effectively contribute 4 extra integer bits during some computations. In the future, it may be desirable to add a "strict" mode that zeroes these extra 4 integer bits after every computation.

Overflow In implementing the fixed point extension of nsimd, we chose to not make any special cases or error handling for overflow. This is similar to the behavior of the integer types used to store the values and perform the arithmetic. However, the presence of unused and reserved bits causes a slight difference in behavior from many signed integers. When a Qa.b number overflows past the a integer bits, it overflows through 0, rather than the largest negative value. This is because the signed integer storing the fixed point number has not yet overflowed to the largest negative value. If the magnitude of the overflow is sufficiently large to overflow

the underlying integer, the fixed point number will overflow in the same way as a signed integer.

6.4.3 . General Usage

Installation The fixed point extension of nsimd is included in the nsimd library as a module. As such, the installation instructions are the same (see Section B.1).

Organization The fixed point extension comprises entirely of header files. These files are located in the `nsimd/include/nsimd/modules` directory. At this level is the `fixed_point.hpp` file which implements the `pack` interface and makes the entire fixed point API accessible. At this level is also the `fixed_point` directory, containing most of the actual implementation. The important files and folders are as described below.

- ⊢ `modules`
  - ⊢ `fixed_point.hpp` - includes all files below. Implements `pack<fp_t` interface.
  - ⊢ `fixed_point`
    - ⊢ `fixed.hpp` - implements `fp_t` interface.
    - ⊢ `simd.hpp` - implements `fpsimd_t` interface.
    - ⊢ `fixed_math.hpp` - agglomerates `#includes` for all scalar math functions.
    - ⊢ `simd_math.hpp` - agglomerates `#includes` for all SIMD math functions.
    - ⊢ `constants.hpp` - defines a number of useful scalar constants.
    - ⊢ `function` - contains scalar function implementations.
      - ⊢ `simd` - contains SIMD function implementations.
    - ⊢ `helper` - functions that aid during development.

Compilation The procedure to compile the fixed point extension of nsimd is the same as for compiling nsimd alone. The only difference is that there is no current C interface for fixed point numbers.

Usage In order to make use of the fixed point extension, one only needs to include the following file.

```
1 #include <nsimd/modules/fixed_point.hpp>
```

This file includes all files necessary for use of the fixed point extension and allows the user to take advantage of the features previously described in Section 6.4.1.

6.5 . Algorithms & Performance

This section presents some of the mathematical functions implemented by the fixed point extension of nsimd. We break these functions into three categories – arithmetic functions, trigonometric functions, and other functions. Each individual function will be presented first in its scalar form, then its vectorized form. Generally, the scalar algorithms are designed to be vectorizable, so this aids in understanding the vectorized forms. Benchmarks will also be presented, comparing the scalar and vectorized performances to comparable fixed point libraries.

6.5.1 . Benchmark Design

The benchmarks presented below were obtained using the Google benchmark[37] tool. The computational loop to be repeated by the benchmark tool consists of a 1024 element input array (or arrays if multiple inputs) of random values. The result of the operation is stored into another array and the benchmark tool ensures that the computational loop is not optimized away. The benchmark tool outputs the raw time in nanoseconds. In order to obtain the cycles per element, we must divide this raw time by the number of elements and multiply by the clock frequency of the CPU used to perform the benchmark.

$$\text{cycles/element} = \frac{\text{raw time}}{\text{total elements}} \cdot \text{clock frequency}$$

Each benchmark compares the nsimd performance, both scalar and SIMD, to the other libraries listed in Section 6.2. Comparisons will be made with respect to the size of the underlying integer representations. For example, an nsimd `fp_t<8,8>` uses 32-bit integer storage, while a liquid-dsp `q16` uses 16-bit integer storage. This comparison is generous to nsimd, as some formats use more storage space with nsimd than the other libraries. Table 6.11 lists the C++ types used for each storage format.

Library	8 Bits	16 Bits	32 Bits	64 Bits
native	`int8_t`	`int16_t`	`int32_t`	`int64_t`
fp_t	`fp_t<2,3>`	`fp_t<4,6>`	`fp_t<16,8>`	`fp_t<16,16>`
pack	`pack<fp_t<2,3> >`	`pack<fp_t<4,6> >`	`pack<fp_t<16,8> >`	`pack<fp_t<16,16> >`
liquid		`q16_t`	`q32_t`	
fixmath			`Fix16`	
CNL∗	`<int8_t,power<-2»`	`<int16_t,power<-4»`	`<int32_t,power<-8»`	`<int64_t,power<-16»`

Value is the `T` in `cnl::scaled_integer<T>`.

Table 6.11: C++ types used for each combination of storage size and library.

The basic arithmetic functions will also be compared to the performance of the equivalent function on a same-sized integer. This allows us to compare the

performance to a natively supported type and see the cost of emulating fixed point numbers.

When examining the performances of the liquid-dsp and libfixmath libraries, it should be noted that these two libraries have compiled components. The cost of making a function call to the compiled functions will be very evident in some cases. In addition, these two libraries do not support all storage sizes. The liquid-dsp library supports 16 and 32 bit formats, while the libfixmath library only supports 32 bit formats.

All benchmarks were performed on the following machines:

SIMD	Compiler	SIMD Flag	Frequency	RAM
SSE4.2	gcc 8.2	-msse4.2	3.2 Ghz	16GB
AVX2	gcc 8.2	-mavx2	3.2 Ghz	16GB
AVX512	gcc 11.0.0	-mavx512f	2.4 Ghz	188GB
AARCH64	gcc 5.4	-marmv8-a	1.0 Ghz	126GB

Table 6.12: Machines used to perform fixed point benchmarks.

All benchmark binaries were compiled using the -O3 flag to obtain maximum performance for all libraries. In the library comparison benchmarks, the pack entries present the results using AVX2 vectorization.

As in Section 6.3.2, all graphs contain links to the corresponding raw data tables in Annex C. These data tables also contain links back to their corresponding graphs.

6.5.2 . Arithmetic Functions

Addition/Subtraction

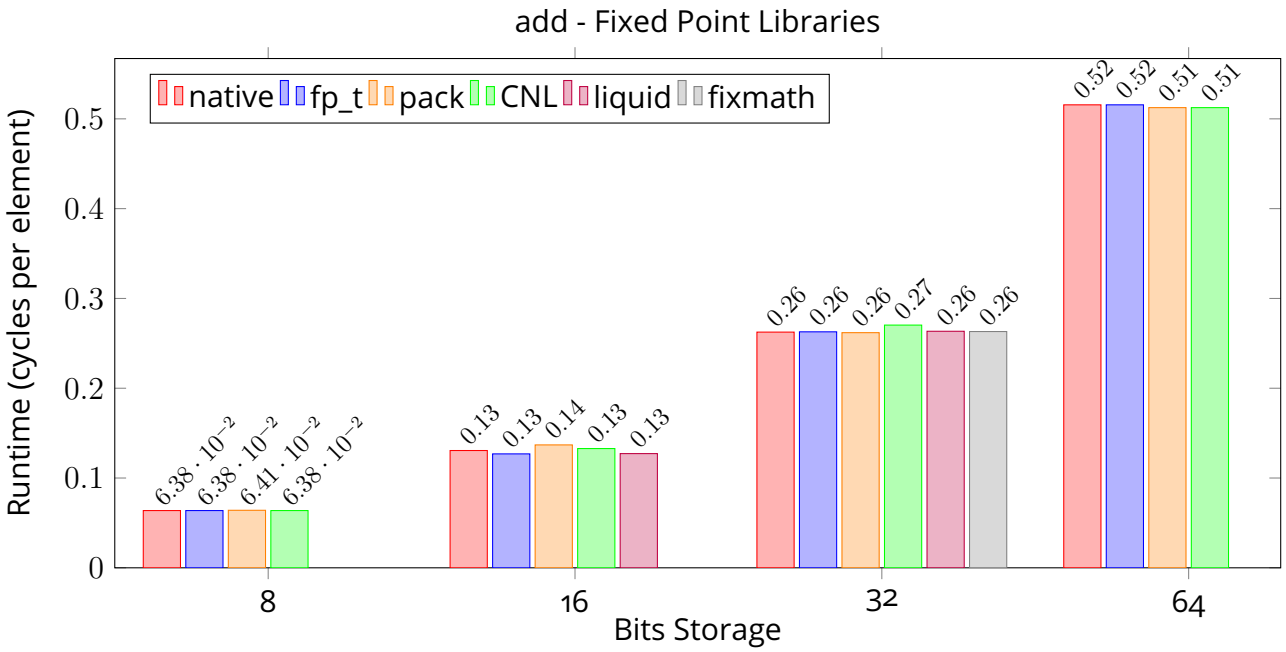

Figure 6.9: Comparison of fixed point library performances for the add function. Raw data in Table C.13.

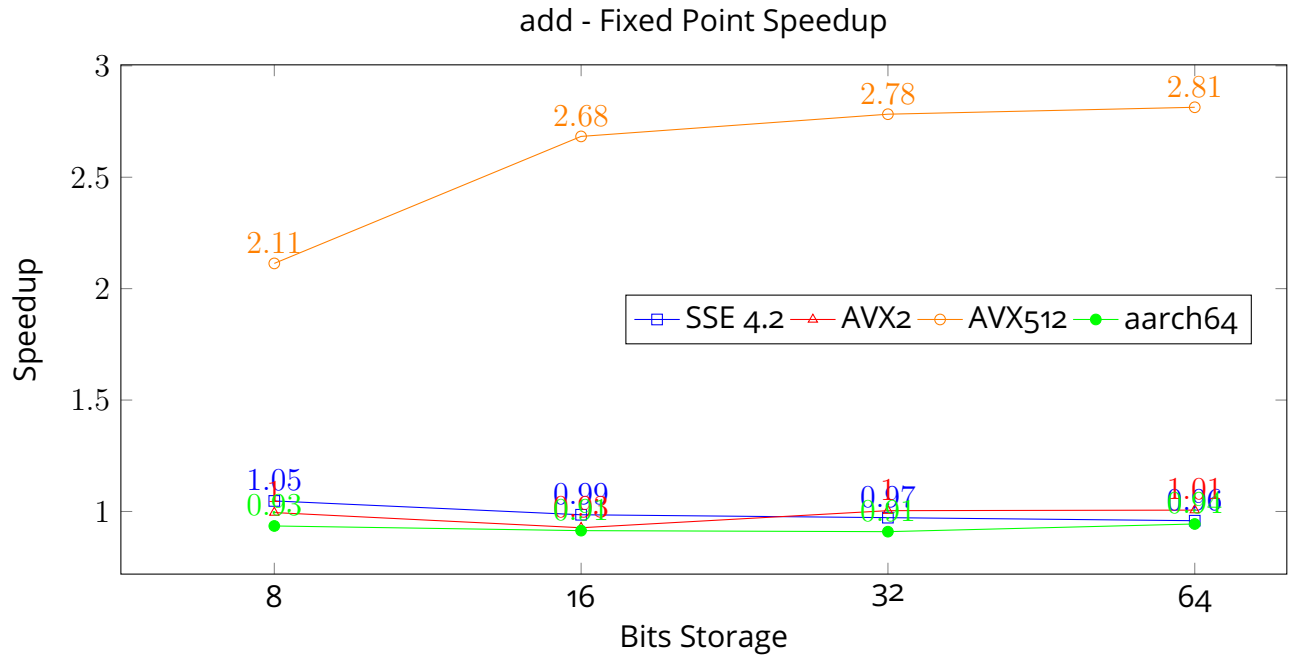

Figure 6.10: Speedup when using `pack<fp_t>` instead of `fp_t` for add on various architectures. Raw data in Table C.5.

As described in Section 6.1, addition and subtraction operations between fixed point numbers are straightforward. They are also similar enough that it is sufficient to present a single one of the two operations. It is enough to perform a simple integer addition or subtraction of the two values to obtain the correct fixed point result.

Comparison Figure 6.9 compares the performances of the add function for all fixed point libraries. All performances except for the pack<fp_t> are functionally identical for the this function. This is because the computation and the code are both simple enough for the compiler to automatically vectorize the add function.

Speedup Figure 6.10 compares the speedups obtained when using the pack<fp_t> structure to vectorize the add function on various architectures. Most speedup curves are fairly flat, as the compiler is capable of successfully vectorizing the scalar version. The one exception is AVX512, where the compiler has completely neglected to vectorize the scalar code.

Example with Numbers For example, if we wish to add 2.125 and 4.5 as Q3.3 values, we first convert the inputs to their implemented integer representations:

$$I(2.125) = 2.125 \cdot 2^3 = 17$$

$$I(4.5) = 4.5 \cdot 2^3 = 36$$

We can then perform an integer sum of the two:

$$17 + 36 = 53$$

And finally we can convert back from the integer representation to obtain the result.

$$53 \cdot 2^{-3} = 6.625$$

Of course, these conversions to and from integer representation are only performed when interacting with other numerical types and formats. When performing operations using only fixed point types of the same format, no conversions are ever necessary.

Multiplication

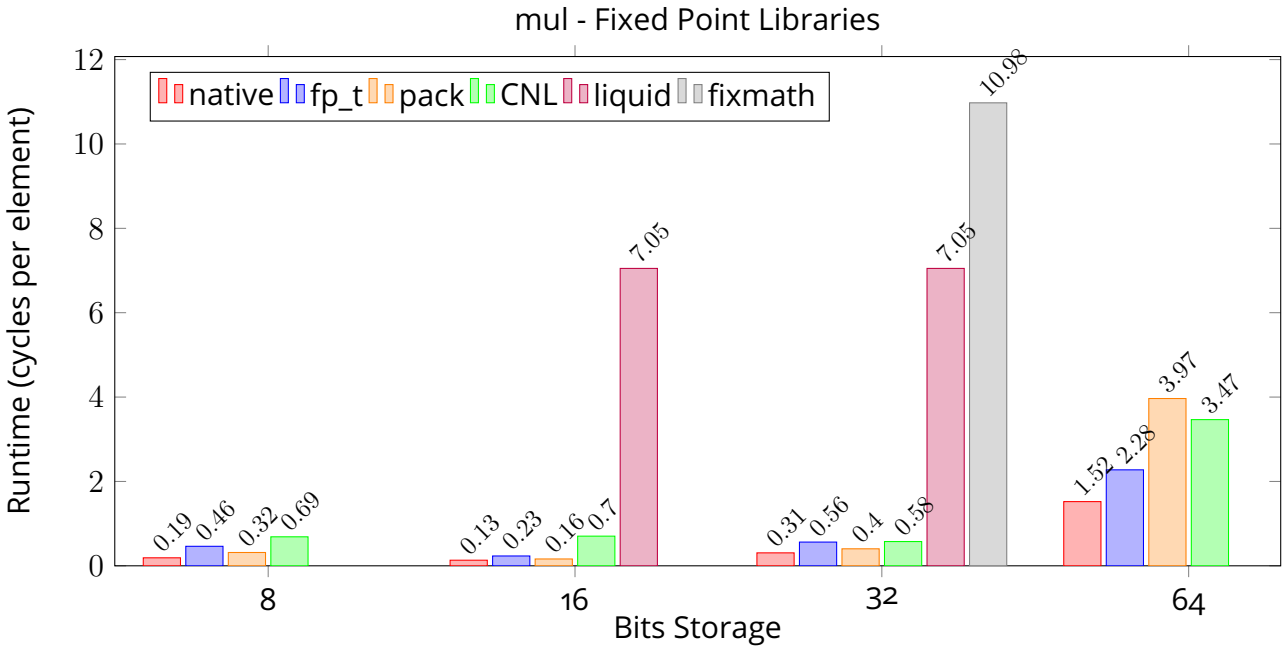

Figure 6.11: Comparison of fixed point library performances for the mul function. Raw data in Table C.14.

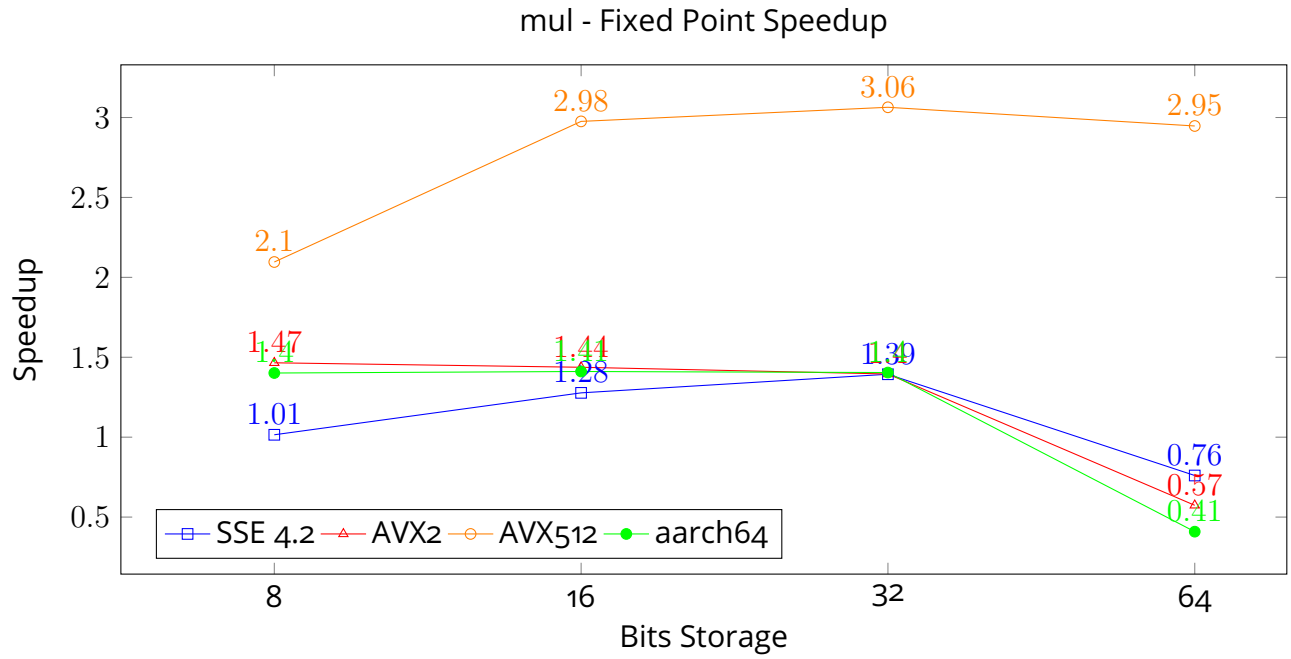

Figure 6.12: Speedup when using `pack<fp_t>` instead of `fp_t` for mul on various architectures. Raw data in Table C.6.

As described in Section 6.1, multiplication of two Qa.b numbers results in a Q2a.2b number. The first a bits and the last b bits of this multiplication are of no interest, as we wish to output a Qa.b number. We may neglect the first a bits, as we choose not to make special cases for integer overflow (Section 6.4.2). This is why we chose to use a + 2b bits to store a Qa.b value – a + 2b bits are sufficient to store the last Qa.2b bits of the multiplication. It is then sufficient to shift the remaining bits to the right (preserving the sign) in order to transform the Qa.2b number into a Qa.b number.

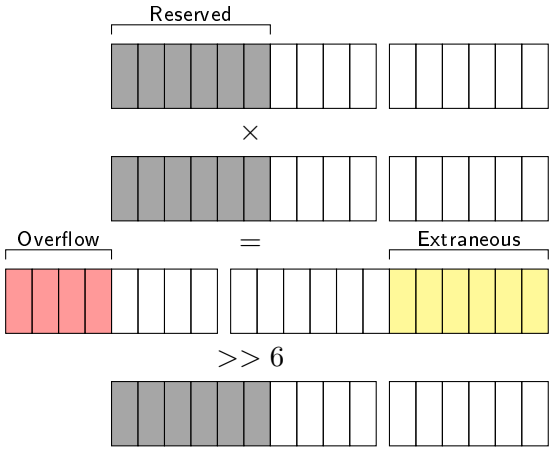

Figure 6.13: Visual breakdown of how nsimd multiplies two Q4.6 numbers.

Figure 6.13 shows a visual example of the nsimd implementation of fixed point multiplication of two Q4.6 numbers. These Q4.6 numbers are store in a 16-bit integer ($4 + 2 \cdot 6 = 16$), with 6 bit reserved for intermediate computations. After the integer multiplication, the top 4 bits of information are lost to overflow, as we would need 20 bits to store a Q8.12 number. However, the next 4 bits are the integer bits corresponding to our Q4.6 result. At this point, the result integer is not yet in Q4.6 format, it is in Q4.12. In order to return to the Q4.6 format, we must shift this result right by 6 bits. Finally, we obtain the Q4.6 result of the multiplication.

It should be noted that this implementation rounds off the extra bits via truncation. This rounding method was chosen for performance reasons, as a simple shift operation is faster than the computation required for any of the more advanced methods.

More formally, the transformations being performed are as follows

$$F_{a.b}(x_1) \cdot F_{a.b}(x_2) = [I(x_1) \cdot 2^{-b}] \cdot [I(x_2) \cdot 2^{-b}] = I(x_1)I(x_2) \cdot 2^{-2b} = F_{a.2b}(x_{result})$$

$$\left[F_{a.2b}(x_{result}) = I(x_{result}) \cdot 2^{-2b}\right] \cdot 2^{b} = I(x_{result}) \cdot 2^{-b} = F_{a.b}(x_{result})$$

Numerically, we achieve this via the following shift operation.

$$F_{a.2b}(x_{result}) >> b = F_{a.b}(x_{result})$$

In comparison, if we had chosen to only store a + b bits to store a Qa.b number, it would not be possible to store the a + 2b bits of result. Instead we would have to choose between two possible solutions to performing multiplication. The first solution would be to convert to a datatype capable of storing at least a + 2b bits. In most cases, this involves converting to a larger datatype, performing the multiplication and shift operations, then converting back

to the original datatype. Most fixed point implementations blindly upcast in order to store the full 2a + 2b result of the multiplication. Some SIMD implementations provide integer multiplication instructions that output a larger datatype (for example, 32-bit int multiplication results in a 64-bit output). However, it is still necessary to perform the shift operations with the larger datatype, then downcast the results back to the original datatype and pay the associated runtime performance costs. The second solution would be to compute only the middle a + b bits of the multiplication. This does not require any datatype conversions. However, it does significantly complicate the computations. In order to compute the middle a + b bits, our simple multiplication would become a series of shifts, multiplications, and additions.

Example with Numbers For example, if we wish to multiply 2.125 and 3.5 as Q3.3 values, we first convert the inputs to their implemented integer representations:

$$I(2.125) = 2.125 \cdot 2^3 = 17$$

$$I(3.5) = 3.5 \cdot 2^3 = 28$$

We can then perform an integer multiplication of the two:

$$17 \cdot 28 = 476$$

However, this operation actually overflows the 9 $(3+2\cdot3)$ bits available, so the value overflows down to

$$476\%256 = 220$$

We must then shift this value back from a Q3.6 integer representation to the Q3.3 integer representation.

$$356 \cdot 2{-}3 = 59.5 \rightarrow 59$$

Where we round the 0.5 down via truncation. Finally, we convert back from the integer representation to see that

$$59 \cdot 2^{-3} = 7.375$$

This is the closest Q3.3 value that does not exceed the numerical result of 7.4375.

Comparison Figure 6.11 compares the performances of the `mul` function for all fixed point libraries. Most evidently, the liquid-dsp and libfixmath bars show we see the cost of calling a function from a compiled library. At all sizes, the native integer multiplication is fastest, as the fixed point multiplications must perform extra computations after performing an integer multiplication themselves. All header-only libraries require take less than one cycle per element. This means that, once again, the compiler has succeeded in vectorizing the scalar codes. The scalar nsimd `fp_t` is generally the fastest fixed point library for multiplication. This is thanks to the implementation choice to choose a storage type that holds enough bits that no casting operations are needed, avoiding extra computation.

Speedup Figure 6.12 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the `mul` function on various architectures. However, it does not appear to have optimally auto-vectorized the 8, 16, and 32 bit codes, as the `pack` implementation does show a slight speedup in these cases.

The exception here is the SSE 4.2 architecture, where the speedup for 8-bit inputs is approximately one. This is because the architecture does not support native 8-bit integer multiplication. The nsimd implementation converts the 8-bit inputs to 16-bits in order to perform vectorized multiplication. It appears that the cost of this conversion neutralizes any gains from the vectorization.

Meanwhile, at 64 bits, we observe a loss in performance when using the `pack<fp_t>` structure. For the SSE 4.2 and AVX2 architectures, this is because there are no instructions that allow for 64-bit integer multiplication. It is not possible to use the same workaround as described for 8-bit inputs, so nsimd uses its scalar fallback. As a result, the SIMD multiplication can only be slower than the scalar multiplication. For the aarch64 architecture, this loss in performance is simply because the 64-bit SIMD register only contains a single 64-bit value. This makes it difficult to achieve any speedup at this level of precision.

Division

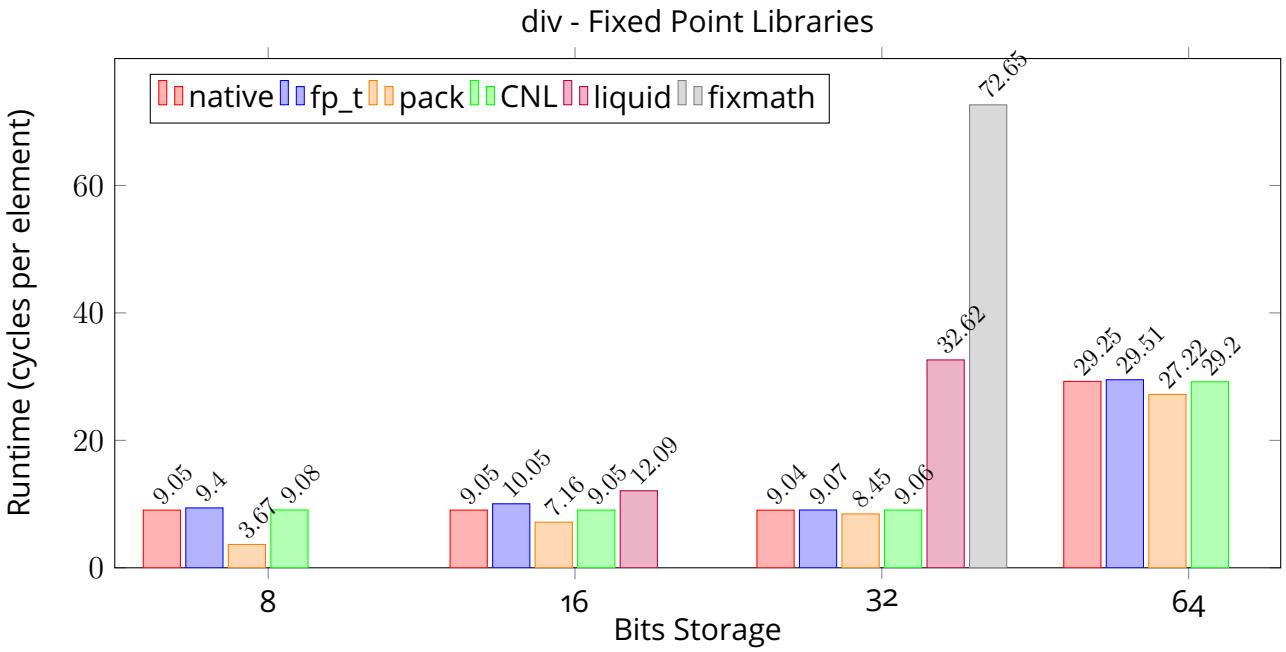

Figure 6.14: Comparison of fixed point library performances for the div function. Raw data in Table C.15.

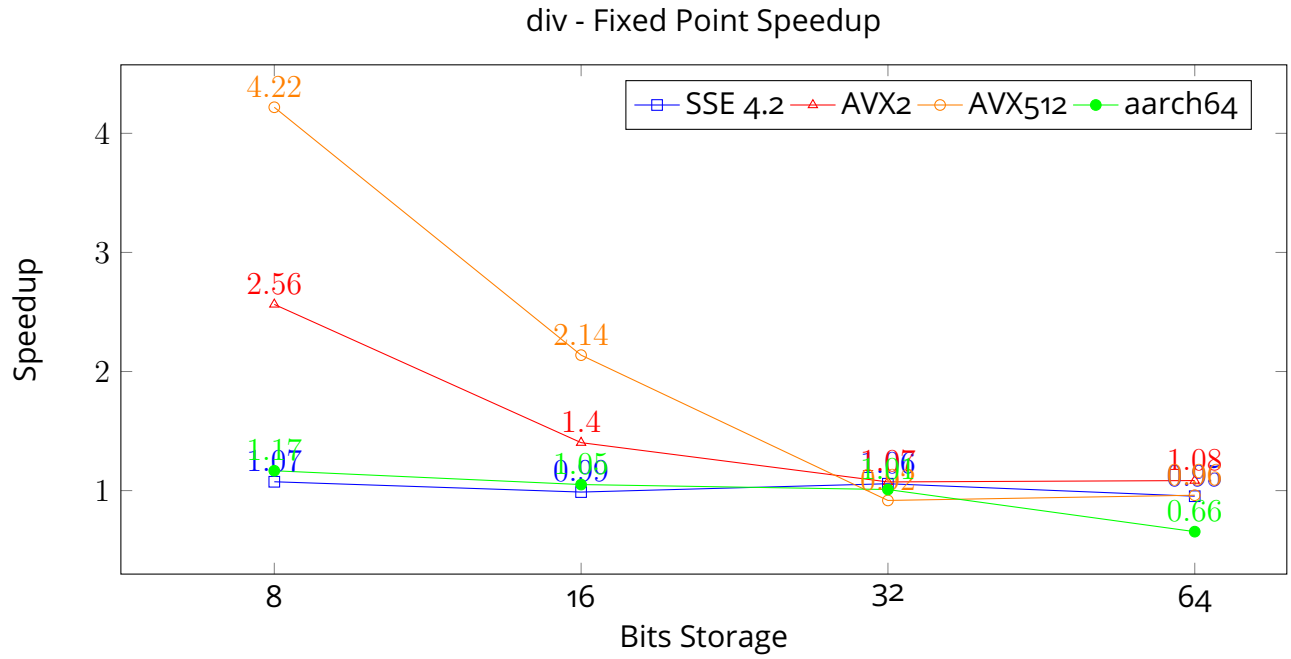

Figure 6.15: Speedup when using `pack<fp_t>` instead of `fp_t` for div on various architectures. Raw data in Table C.7.

As described in Section 6.1, division of one Qa.b numbers by another results in a Qa.0 number. However, we wish to produce a result with Qa.b precision. We achieve this by converting the numerator to Qa.2b format (by shifting the numerator b bits to the left) before performing integer division, as shown in Figure 6.16. The resulting value will be in the desired Qa.b format. This is always possible thanks to the b reserved bits.

More formally, the transformations being performed are as follows

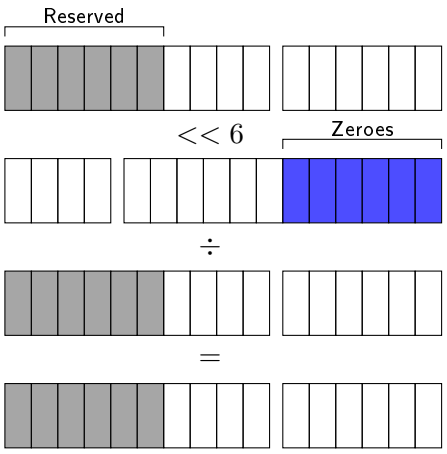

Figure 6.16: Example of the operations involved in scalar and 8/16-bit SIMD Q4.6 divisions.

$$\frac{F_{a.b}(x_1)}{F_{a.b}(x_2)} = \frac{I(x_1) \cdot 2^{-b}}{I(x_2) \cdot 2^{-b}}$$

$$\frac{F_{a.2b}(x_1)}{F_{a.b}(x_2)} = \frac{I(x_1) \cdot 2^{-2b}}{I(x_2) \cdot 2^{-b}} = \frac{I_1}{I_2} \cdot 2^{-b} = F_{a.b}(x_{result})$$

In comparison, if we had chosen to only store a + b bits to store a Qa.b number, it would not be possible to perform the shift without losing bits of information. In most cases, it would instead be necessary to convert to a datatype capable of storing at least a + 2b bits. This involves converting to a larger datatype, performing the shift and division operations, then converting back to the original datatype. Most fixed point implementations blindly upcast in order to store the 2a + 2b bits and be more than certain to store result of the shift operation.

Example with Numbers For example, if we wish to divide 2.25 by 2 using Q3.3 values, we first convert the inputs to their implemented integer representations:

$$I(2.375) = 2.375 \cdot 2^3 = 19$$

$$I(2.0) = 2.0 \cdot 2^3 = 16$$

Next we shift the numerator 3 bits left

$$19 \cdot 2^3 = 152$$

before performing the integer division

$$152 \div 16 = 9.5 \rightarrow 9$$

Where we round the 0.5 down via truncation. Finally, we convert back from the integer representation to see that

$$9 \cdot 2^{-3} = 1.125$$

This is the closest Q3.3 value that does not exceed the numerical result of 1.1875.

Comparison Figure 6.14 compares the performances of the `div` function for all fixed point libraries. Here we observe again observe similar performance among all header-only libraries. Looking at the `pack` bars, we see the effects of using the vectorized integer division implemented directly into nsimd for 8 and 16 bit inputs. At 32 and 64 bits, nsimd falls back to scalar division, displaying similar performance to the other division operations. Overall, we can observe that the cost of an integer division operation is dwarfs the cost of the shift operation, as the native integer division performance is similar to the fixed point division performance. Once more, the effect of calling a compiled function from a library can be seen at 32 bits with the liquid-dsp and libfixmath libraries.

Speedup Figure 6.15 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the `div` function on various architectures. We can observe the effects of the vectorized integer division of nsimd at 16 and 32 bits for the AVX2 architecture. For the other architectures, the variations in performance depend on the post-compilation cost of the nsimd scalar fallback mechanism.

6.5.3 . Trigonometric Functions

All trigonometric functions presented in this section were implemented using Taylor series approximations centered around zero. This choice was made in order to maximize comparability with the other fixed point libraries. While Taylor series approximations increase in error as the input gains distance from the center of the approximation, this error can be mitigated by exploiting the symmetry of trigonometric functions in order to reduce the input domain. In all functions presented, we reduce the input domain to $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

Sine

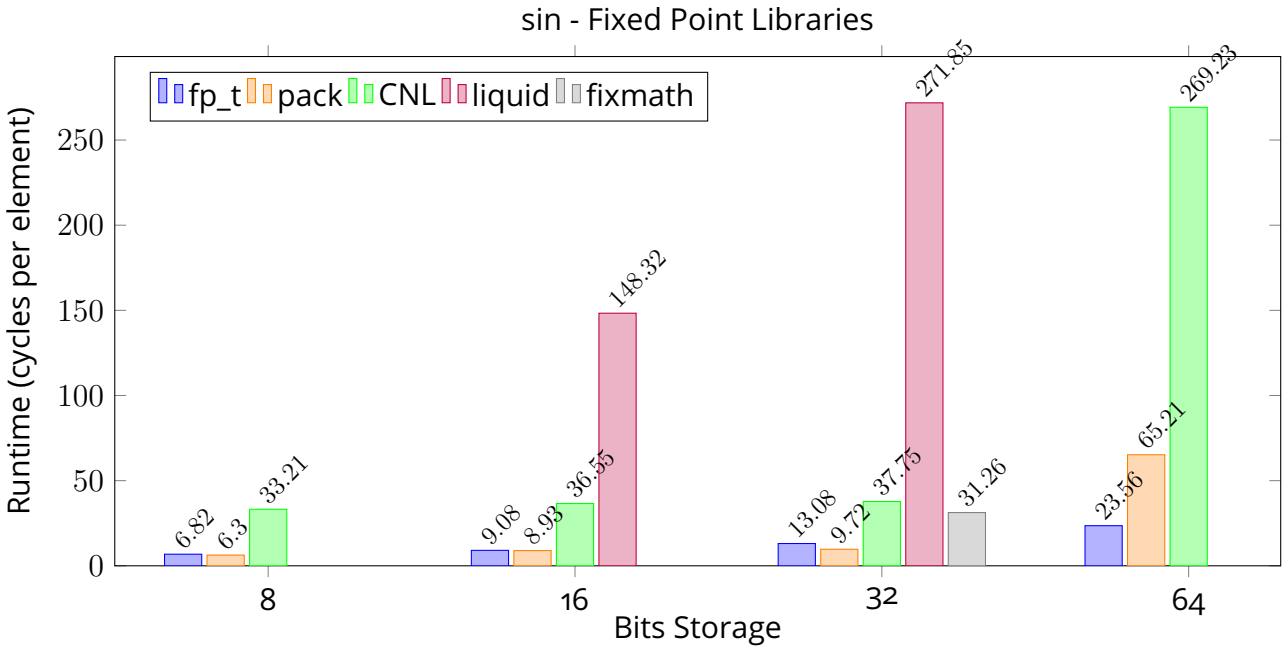

Figure 6.17: Comparison of fixed point library performances for the sin function. Raw data in Table C.18.

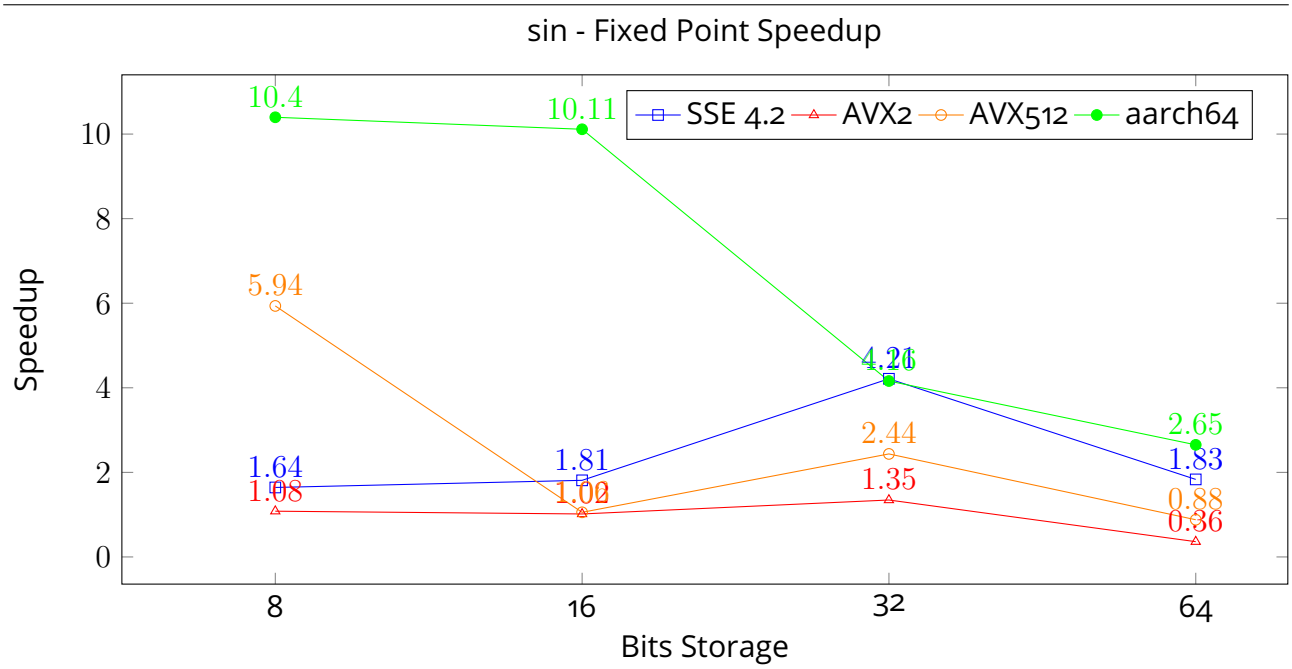

Figure 6.18: Speedup when using `pack<fp_t>` instead of `fp_t` for sin on various architectures. Raw data in Table C.10.

The sine function was implemented using a Taylor series expansion around zero (more specifically a Maclaurin series). The Taylor expansion of the sine around zero is

$$\sin(x) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)!} \cdot x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

We can factor the above equation into

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... = x \cdot (1 - \frac{x^2}{2 \cdot 3}(1 - \frac{x^2}{4 \cdot 5} - (1 - \frac{x^2}{6 \cdot 7}(1 - ...))))$$

This has the effect of reducing the number of computations, and reducing the magnitude of the values involved. The maximum power of x is x^2 and we avoid computing large exponentials. For example, the fourth term is $\frac{x^2}{6 \cdot 7}$ rather than $\frac{x^7}{7!}$. Reducing these magnitudes reduces the risk of numerical errors that arise from overflow or division by a large number negating a term. At the moment, we find that limiting computations to this fourth term provides sufficient numerical accuracy. In the future, it would be ideal to use SFINAE to determine the number of terms to use according to the number of decimal bits. Table 6.13 shows the level of precision each term of the expansion provides, where the Log2(denominator) entry is the approximate level of decimal bits needed for the term to have an impact.

Order of the term	1	2	3	4	5	6	7
Expression	x	$\frac{x^3}{3!}$	$\frac{x^5}{5!}$	$\frac{x^7}{7!}$	$\frac{x^9}{9!}$	$\frac{x^{11}}{11!}$	$\frac{x^{13}}{13!}$
Log2(denominator)	0	2.5	6.9	12.3	18.5	25.2	32.5

Table 6.13: The approximate impact of each `sin` Taylor expansion term on the final result.

In order to minimize the error that arises when inputs are distant from the center of the expansion (zero), it is necessary to resize inputs using symmetry. We exploit the following properties of the sine function in the following order to reduce the input domain:

$$\begin{aligned} \text{Initial} &: -\infty < x < \infty \\ \sin(-x) = -\sin(x) &: \quad 0 \leq x < \infty \\ \sin(x + n \cdot (2\pi)) = \sin(x) &: \quad 0 \leq x < 2\pi \\ \sin(x + \cdot\pi) = -\sin(x) &: \quad 0 \leq x < \pi \\ \sin(x) = \cos(x - \frac{\pi}{2}) &: -\frac{\pi}{2} \leq x < \frac{\pi}{2} \end{aligned}$$

Comparison Figure 6.17 compares the performances of the `sin` function for all fixed point libraries. The `fp_t` and `pack<fp_t>` performances are consistently faster than the other libraries. However, as the following paragraph will describe, this is more due to compiler optimizations than any code optimization. The CNL and libfixmath libraries are generally within a factor of 3x, while the liquid-dsp library shows slower performance.

Speedup Figure 6.18 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the `sin` function on various architectures.

On the SSE 4.2 and AVX2 architectures, the low speedup is actually due to the compiler optimizations of the scalar implementation. The C++ code contains a number of divisions by constant (calculable at compile-time) fixed-point values in order to perform the input range reduction described above. However, when disassembling the compiled code, there is no trace of any integer division – the compiler has replaced them with multiplication by by integer constants whose properties lead to effects equivalent to multiplying by the inverses of the original constants. This avoids the need to perform a computationally expensive integer division operation, greatly accelerating the overall computation. The remaining addition, multiplication, and shift operations are significantly faster than division operations. In addition, they are operations that the compiler is capable of auto-vectorizing. Meanwhile, the SIMD implementation does not contain this optimization of the division by constants.

On the aarch64 and AVX512 architectures, the compiler is not able to perform the same optimizations. This allows for a more impressive speedup. The AVX512 compiler is capable of vectorizing some parts of the code for only 16-bit inputs, reducing the possible observable speedup at this data point.

Cosine

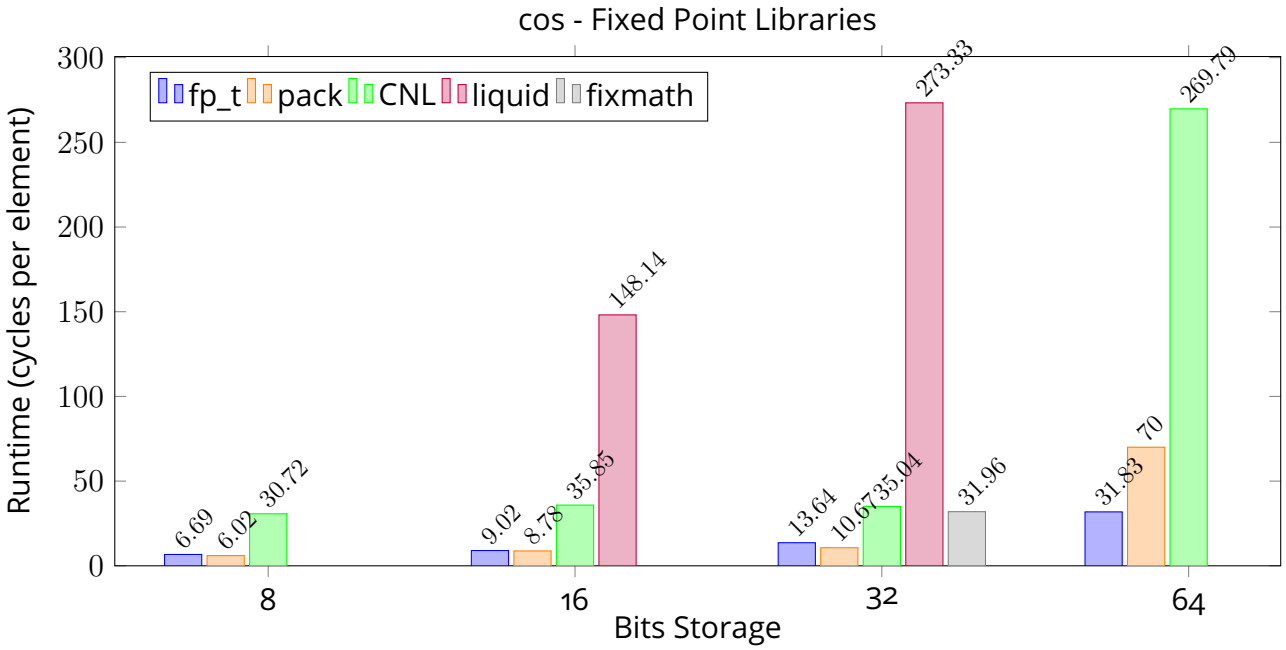

Figure 6.19: Comparison of fixed point library performances for the cos function. Raw data in Table C.19.

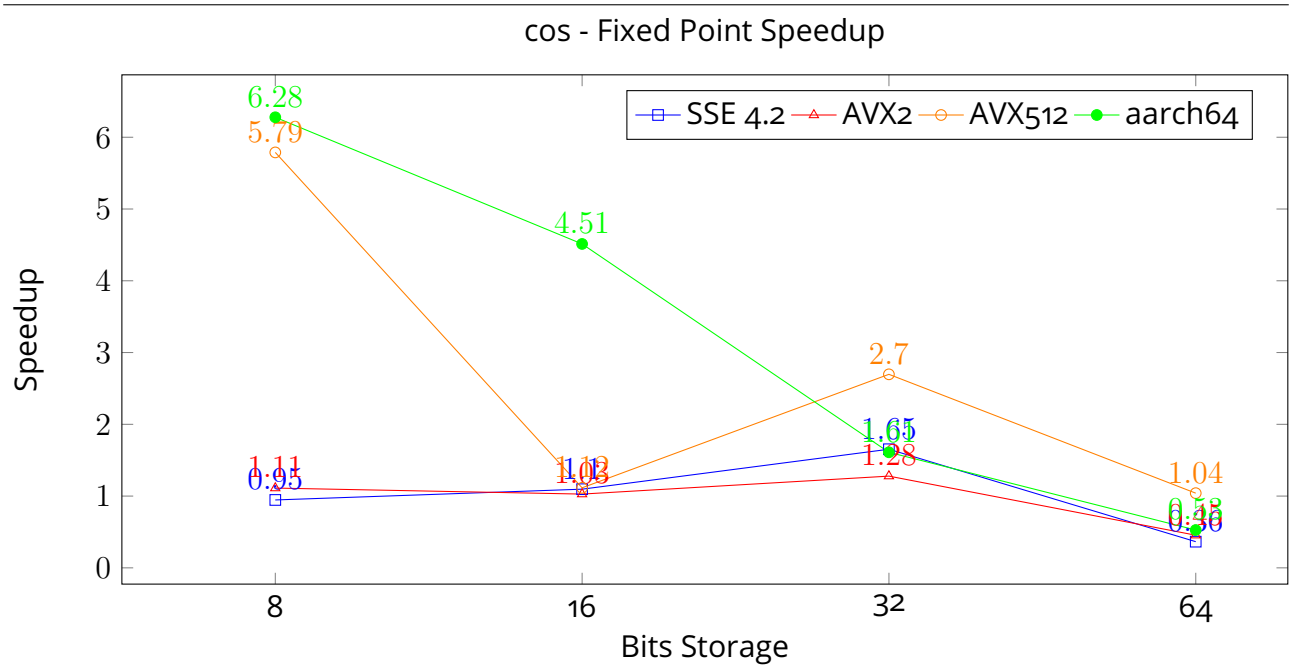

Figure 6.20: Speedup when using `pack<fp_t>` instead of `fp_t` for cos on various architectures. Raw data in Table C.11.

The cosine function was implemented using a Taylor series approximation around zero. The Taylor expansion of the cos around zero is

$$\cos(x) = \sum_{n=0}^{\infty} \frac{-1^n}{(2n)!} \cdot x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

We can factor the above equation into

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... = 1 - \frac{x^2}{1 \cdot 2}(1 - \frac{x^2}{3 \cdot 4} - (1 - \frac{x^2}{5 \cdot 6}(1 - ...)))$$

As with the sine function, this reduces the number of operations to perform and reduces the magnitude of the higher order terms. Once more, the computations are currently limited (here to the fifth term) with possible optimization via SFINAE.

We exploit the following properties of the cosine function in the following order do reduce the input domain:

$$\begin{aligned} \text{Initial} &: -\infty < x < \infty \\ \cos(-x) = \cos(x) &: \quad 0 \leq x < \infty \\ \cos(x + n \cdot (2\pi)) = \cos(x) &: \quad 0 \leq x < 2\pi \\ \cos(x + \cdot\pi) = -\cos(x) &: \quad 0 \leq x < \pi \\ \cos(x) = -\sin(x - \frac{\pi}{2}) &: -\frac{\pi}{2} \leq x < \frac{\pi}{2} \end{aligned}$$

Comparison Figure 6.19 compares the performances of the cos function for all fixed point libraries. The `fp_t` and `pack<fp_t>` performances are consistently faster than the other libraries. However, as the following paragraph will describe, this is more due to compiler optimizations than any code optimization. The CNL and libfixmath libraries are generally within a factor of 3x, while the liquid-dsp library shows slower performance.

SIMD Figure 6.20 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the cos function on various architectures. The cos function benefits from the same compiler optimizations as the `sin` function. As a result, the SSE and AVX2 scalar performances once more contain optimizations not present in the SIMD implementations, limiting the possible speedups.

Tangent

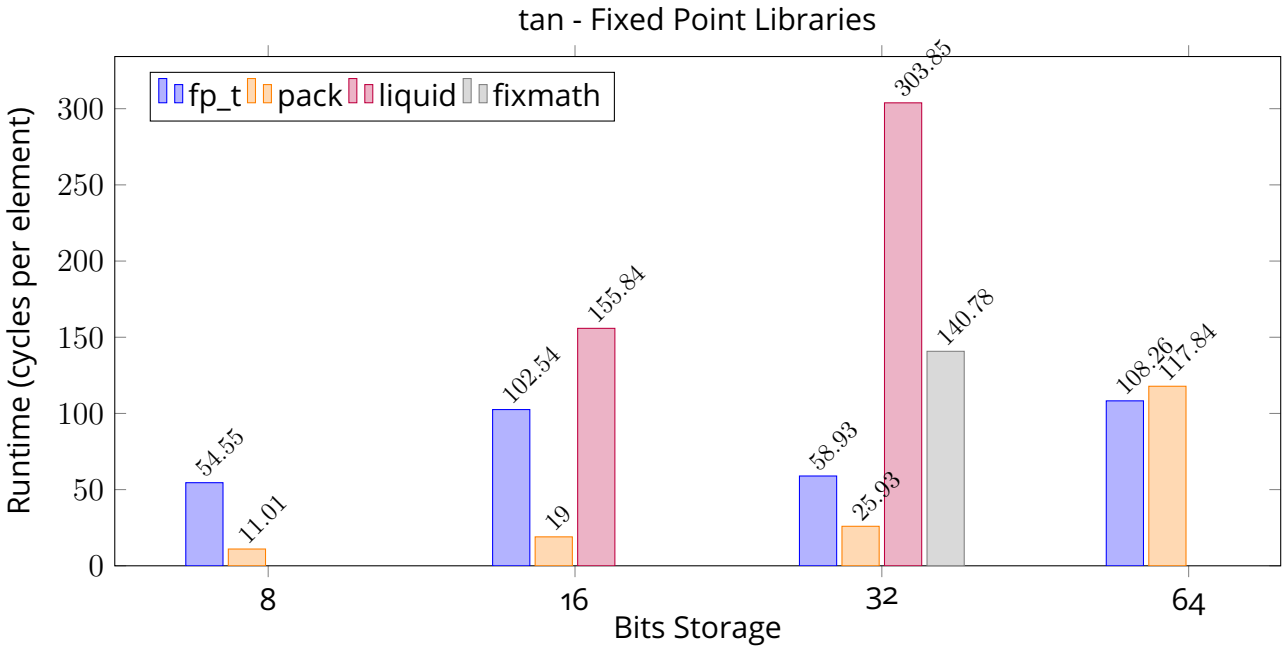

Figure 6.21: Comparison of fixed point library performances for the tan function. Raw data in Table C.20.

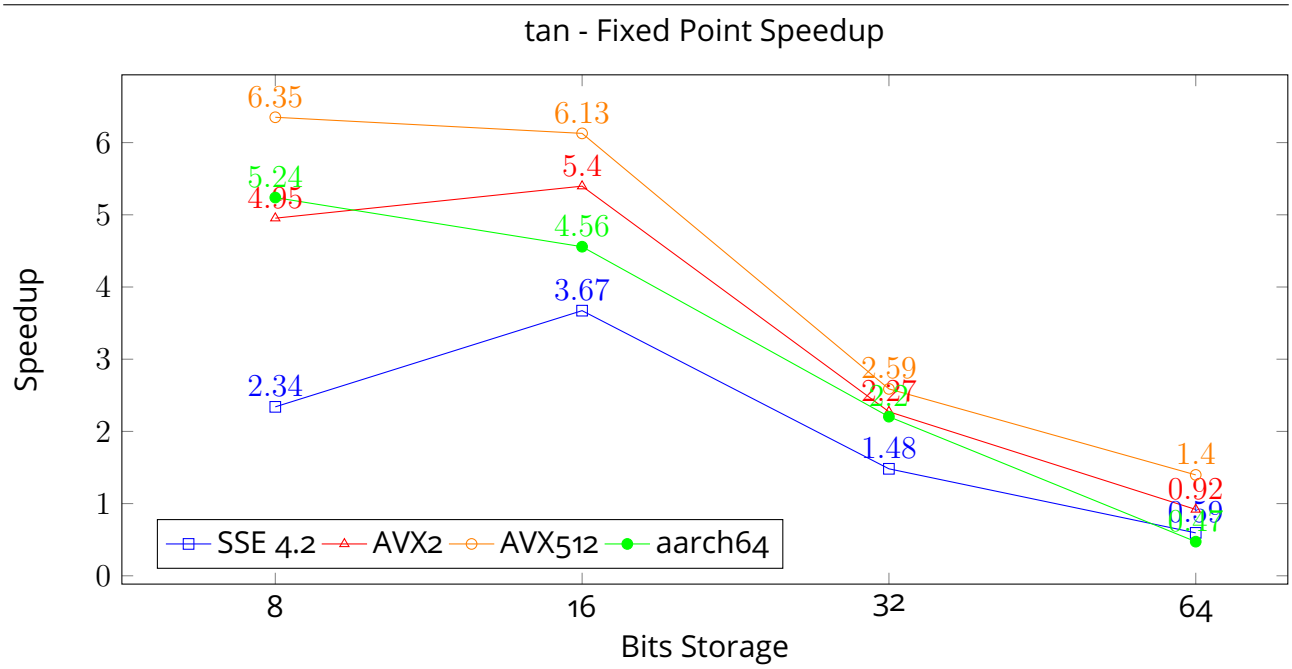

Figure 6.22: Speedup when using `pack<fp_t>` instead of `fp_t` for tan on various architectures. Raw data in Table C.12.

The tangent function is implemented using the sine and cosine functions via the following property.

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

It first performs range reduction using the properties of the tangent function, then calls the range reduced versions of the sine and cosine functions. We exploit the following properties of the tangent function in the following order do reduce the input domain:

$$\begin{aligned} \text{Initial} &: -\infty < x < \infty \\ \tan(-x) = -\tan(x) &: \quad 0 \leq x < \infty \\ \tan(x + n \cdot \pi) = \tan(x) &: \quad 0 \leq x < \pi \\ \tan(x + n \cdot \pi) = \tan(x) &: -\frac{\pi}{2} \leq x < \frac{\pi}{2} \end{aligned}$$

Comparison Figure 6.21 compares the performances of the `tan` function for all fixed point libraries. The CNL library is not present, as it does not present a `tan` function. While the scalar `fp_t` performance is still faster than the remaining libraries, the difference is reduced.

SIMD Figure 6.22 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the `tan` function on various architectures. Compared to Figures 6.18 and 6.20, it appears that the `tan` function does not benefit from as strong of compiler optimizations as the `sin` and `cos` functions. As a result, the vectorized `pack<fp_t>` implementation can show a better speedup than the previous functions.

The SSE 4.2 and AVX2 architectures exhibit a peak at 16 bits rather than 8 bits due to the usage of a number of operations that are implemented for 16, but not 8 bit inputs.

Reciprocal

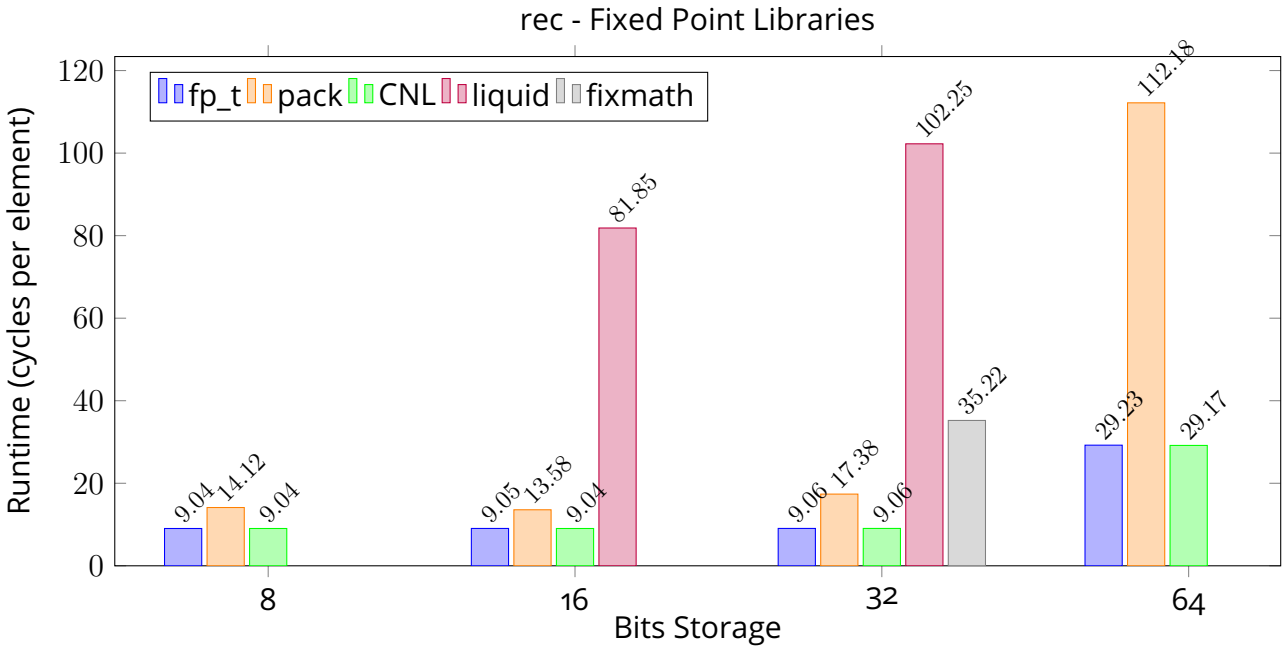

Figure 6.23: Comparison of fixed point library performances for the rec function. Raw data in Table C.16.

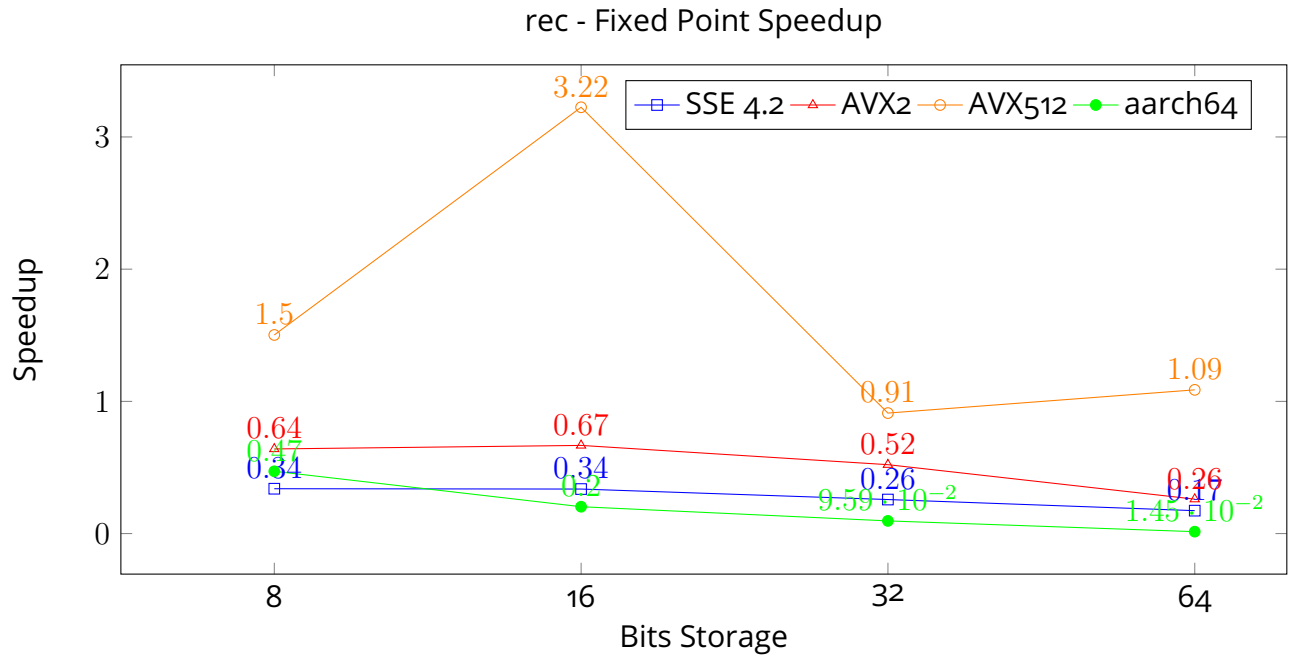

Figure 6.24: Speedup when using `pack<fp_t>` instead of `fp_t` for rec on various architectures. Raw data in Table C.8.

The reciprocal function can be implemented in two ways. The first way is to perform fixed point division of 1 by the input. However, SIMD instruction sets do not include the integer division required to do this. We present a long division algorithm which enables vectorized integer division for smaller integer types, but we still need an efficient reciprocal for larger integer types. For this, we use the Newton-Raphson iterative method to divide 1 by the input according to the following equation, where a is the input.

$$x_{i+1} = x_i \cdot (2 - (x_i \cdot a))$$

Depending on the choice of starting seed x_0, this formula may or may not converge to $x_f = \frac{1}{a}$. The formula stops updating when $x_{i+1} = x_i$, which can occur when

$$x_{i+1} = x_i = x_i \cdot (2 - (x_i \cdot a))$$

$$0 = -x_i + 2x_i - ax_i^2$$

$$0 = x_i - ax_i^2$$

$$0 = x_i(1 - ax_i)$$

$$x_i \in \{0, \frac{1}{a}\}$$

Here we see that there are two values which cause the iterations to stop updating – the correct answer ($\frac{1}{a}$) and zero. This means that we wish to avoid any seeds which cause the updates to pass through zero. We want to avoid values that satisfy the following formula

$$x_{i+1} = 0 = x_i \cdot (2 - (x_i \cdot a))$$

$$0 = 2x_i - ax_i^2$$

$$0 = x_i(2 - ax_i)$$

$$x_i \in \{0, \frac{2}{a}\}$$

This means that we want to choose the starting seed such that

$$0 < x_0 < \frac{2}{a}$$

Otherwise the final result will be zero. This appears to be a narrow window of possible starting values, but it is possible to numerically make a very good starting guess by exploiting the following relationship

$$(a = 2^{\log_2(a)})^{-1} \rightarrow \frac{1}{a} = 2^{-\log_2(a)}$$

While the exact $\log_2(a)$ is not trivial to compute numerically, its integer portion $\lfloor \log_2(a) \rfloor$ can be. Numerically, $\lfloor \log_2(a) \rfloor$ only requires that we locate the strongest set bit. This is possible

via the clz function implemented in Section 6.3.2. The clz function counts the number of zero bits before the strongest set bit. It can be used to compute $\lfloor \log_2(a) \rfloor$ via the formula

$$\lfloor \log_2(a) \rfloor = N_{\text{total bits}} - \texttt{clz}(a)$$

Because $\lfloor \log_2(a) \rfloor$ is an integer value, we can compute $2^{-\lfloor \log_2(a) \rfloor}$ via a shift operation. Remembering that the relationship between a fixed point value and its integer representation is

$$F_{a.b}(x) = I(x) \cdot 2^{-b}$$

this formula becomes

$$\lfloor \log_2(F_{a.b}(x)) \rfloor = \lfloor \log_2(I(x) \cdot 2^{-b}) \rfloor = N_{\text{total bits}} - \texttt{clz}(I(x)) - b$$

Our seed x_0 is thus computed via

$$F_{a.b}(x_0) = 2^{-(N_{\text{total bits}} - \texttt{clz}(I(x)) - b)} = 2^{\texttt{clz}(I(x)) + b - N_{\text{total bits}}}$$

$$I(x_0) = F_{a.b}(x_0) \cdot 2^b = 2^b \cdot 2^{\texttt{clz}(I(x)) + b - N_{\text{total bits}}} = 2^{\texttt{clz}(I(x)) + 2b - N_{\text{total bits}}}$$

Numerically, we can efficiently compute this power of 2 using shift operators. Unfortunately, not all shift operations define the behavior of shifting by a negative value. This means that we want to reframe the computation to only perform valid shifts in one direction. To accomplish this, we choose to start with a value of 2^{-b} (which sets the weakest bit to 1). As a result, any x_0 computation that requires a right shift is not representable in the Qa.b format and we do not need to specify the result. Finally, we can numerically compute our starting seed by beginning with

$$I(x_0) = I(2^{-b}) \cdot 2^{\texttt{clz}(I(a)) + 2b - N_{\text{total bits}}} = I(2^{-b}) << [\texttt{clz}(I(x)) + 2b - N_{\text{total bits}}]$$

In practice, we find that the choice of starting seed allows for rapid convergence. As such we limit the maximum number of SIMD iterations to 10. While this has yet to prove insufficient in our applications, more in depth tests are required to determine if 10 iterations are sufficient in all cases.

Example with Numbers For example, if we wish to compute the reciprocal of 15 using a Q8.12 (32-bit) format, we must first convert it to its integer representation.

$$15 = F_{8.12}(15) = I_{8.12}(15) \cdot 2^{-12}$$

$$I_{8.12}(15) = F_{8.12}(15) \cdot 2^{+12} = 15 \cdot 2^{12}$$

Next, we compute the starting seed.

$$\text{clz}(I_{8.12}(15)) = 16$$

$$I(x_0) = I(2^{-12}) << (16 + 24 - 32)$$

$$I(x_0) = I(2^{-12}) << 8 = 1 << 8 = 2^8$$

To see what number this integer format represents, we can convert it back to a fixed point value.

$$F_{8.12}(x_0) = I(x_0) \cdot 2^{-12} = 2^8 * 2^{-12} = 2^{-4} = \frac{1}{16} = 0.625$$

Finally, we use this starting seed to and perform our iterations to compute the reciprocal.

$$x_1 = 0.6025 \cdot (2 - (0.0625 \cdot 15)) = 0.06640625$$

$$x_2 = 0.06640625 \cdot (2 - (0.06640625 \cdot 15)) = 0.06665039$$

$$x_3 = 0.06665039 \cdot (2 - (0.06665039 \cdot 15)) = 0.06665039$$

Thanks to the choice of a good starting seed, the computation converges rapidly in 3 iterations.

Comparison Figure 6.23 compares the performances of the `rec` function for all fixed point libraries. The scalar `fp_t` implementation uses scalar integer division, while the SIMD `pack<fp_t>` implementation uses the Newton-Raphson algorithm described in this section. As a result, the scalar results are identical to the results presented for division (Section 6.5.2), but the vectorized results are slower.

Speedup Figure 6.24 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the `rec` function on various architectures. Here we see that the Newton-Raphson implementation is slower than the scalar implementation on most architectures for all input sizes. This is because of the high cost of the starting seed computation, which makes use of the `clz` and `shlv` functions described in Section 6.3.2. Either the necessary instructions do not exist (SSE4.2 and AVX2) or the register sizes are insufficient to amortize the extra computations (aarch64). The exception to this is the AVX512 architecture, which has the largest register size and provides appropriate intrinsic functions. Most interestingly, the `rec` function is twice as fast as the `div` function on AVX512 16-bit types. This indicates that, while of very limited usefulness today, the `rec` function will be worth using on future architectures with large SIMD register sizes and full support for the necessary intrinsic functions. In most cases, however, the `fp_t` `rec` function should be implemented using the `div` function instead.

Square Root

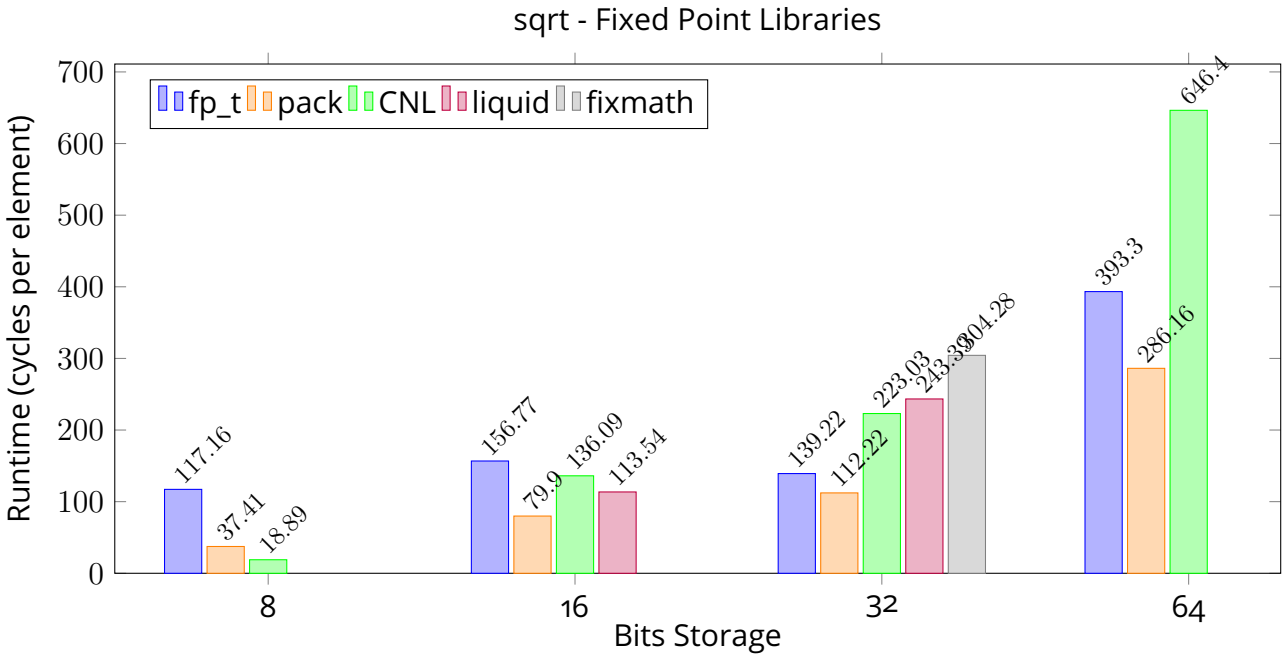

Figure 6.25: Comparison of fixed point library performances for the sqrt function. Raw data in Table C.17.

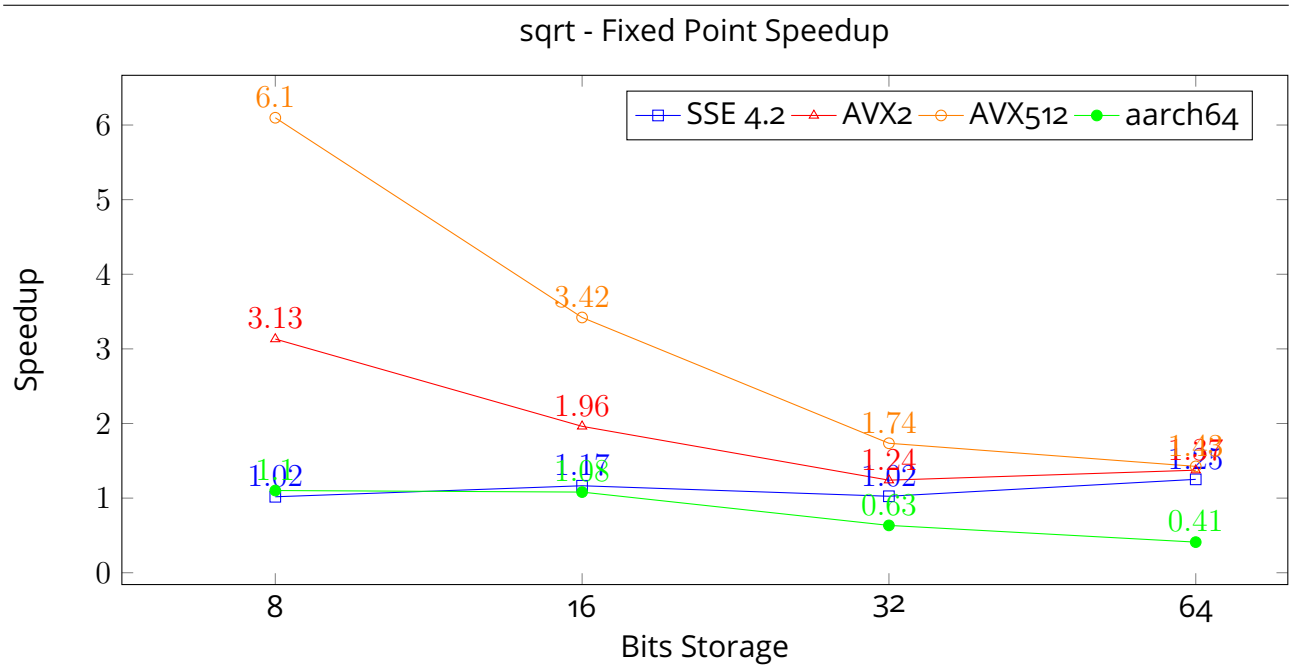

Figure 6.26: Speedup when using `pack<fp_t>` instead of `fp_t` for sqrt on various architectures. Raw data in Table C.9.

The square root function is implemented using the Newton-Raphson iterative method described in Section 3.2.2. This gives us the following formula for iterating towards the output.

$$x_{i+1} = \frac{1}{2} \cdot (x_i + \frac{a}{x_i})$$

We find that setting the starting seed x_0 to be equal to the input value to be sufficient – initial tests show that this allows tested inputs to converge within 10 iterations. As such, the vectorized implementation blindly performs 10 iterations to avoid the cost of checking if all inputs have converged. More in depth tests are necessary to determine if this 10 iterations are truly sufficient for all possible inputs.

Unlike the reciprocal function, there is only one simple numerical problem to avoid when implementing the square root – avoiding division by zero. This scenario can only occur when the initial seed is equal to zero, so – in theory – a single check before beginning the iterations is sufficient to avoid the problem. In practice, it is possible for the mathematical result to be smaller than one ULP, so this check must be performed with every loop iteration.

Example with Numbers For example, if we wish to compute the square root of 15 using a Q8.12 (32-bit) format, we perform the following iterations.

$$x_0 = 15$$
$$x_1 = \frac{1}{2} \cdot (15 + \frac{15}{15}) = 8$$
$$x_2 = \frac{1}{2} \cdot (8 + \frac{15}{8}) = 4.9375$$
$$x_3 = \frac{1}{2} \cdot (4.9375 + \frac{15}{4.9375}) = 3.98755$$
$$x_4 = \frac{1}{2} \cdot (3.98755 + \frac{15}{3.98755}) = 3.87451$$
$$x_5 = \frac{1}{2} \cdot (3.87451 + \frac{15}{3.87451}) = 3.87280$$
$$x_6 = \frac{1}{2} \cdot (3.87280 + \frac{15}{3.87280}) = 3.87280$$

In this example, we converge in 6 iterations.

Comparison Figure 6.25 compares the performances of the sqrt function for all fixed point libraries. At 8 bits, the CNL library is nearly by far the fastest. However, it scales poorly with size, as it implements a digit-by-digit algorithm for computing the square root. As we saw in Section 6.5.2, this type of algorithm scales very poorly with the number of bits. The scalar `fp_t` algorithm is much less efficient at 8 and 16 bits – even the liquid library is faster at 16 bits. However, it vectorizes well enough that the SIMD implementation is fastest for 16-bit and larger numbers.

Speedup Figure 6.26 compares the speedups obtained when using the `pack<fp_t>` structure to vectorize the sqrt function on various architectures. The `sqrt` performance graph resembles the graph showing the performance of integer division (Figure 6.7) for good reason. The limiting factor in the SIMD performance of the `sqrt` function is the presence of an integer division in each loop iteration.

6.6 . Conclusion

The fixed point computational library presented here presents a different approach than traditionally used when storing fixed point numbers. This approach allows us to avoid converting between different sized types when performing computations. While the gains are not always noticeable when performing scalar computations, this storage method allows us to write much more efficient vectorized code. In addition, it allows us to write simple scalar code for the compiler to optimize. Our use of template metaprogramming allows us to easily support a vast array of fixed point configurations. The result of all of this is a flexible, efficient computation library for fixed point numbers. However, this library is still very much a work in progress.

The first avenue of improvement is the addition of more functions. A few functions, such as the exponent and logarithm, already have scalar implementations but no corresponding vectorized implementations.

A second avenue of improvement is the optimization of the existing functions. For example, the compiler optimizations of the `sin` and `cos` functions should be applicable to all architectures both scalar and bectorized. Some optimizations would need to be performed via the use of SFINAE to select between different implementations depending on the fixed point format being used. For example, the square root algorithm implemented by CNL could be implemented and selected for use with 8-bit storage types. Another target for optimization is the trigonometric functions, which could vary the depth of the Taylor expansion depending on the number of decimal bits.

The third avenue of improvement is the user interface. There are two clear ways to improve the user interface of this fixed point library. The first is to add support for unsigned fixed point numbers. Unsigned fixed point numbers would not suffer the irregular overflow described in Section 6.4.2. In addition, some algorithms allow for different optimizations depending on whether the inputs are signed or not. The second user interface improvement is to add a strict mode which enforces the number of integer bits. It would cause significant performance loss – the simplest implementation would require an extra mask operation after every single operation performed on a fixed point number. However, this strict mode could be very useful when exploring the effects of strongly limited numerical precision as we will in Chapter 7.

7 - Neural Network Precision

The subject of which numerical precision to use for neural network inference is a subject of active research[42][2][17]. Candidates for useful numerical precisions are not limited to the traditionally supported types (int, float, char, etc.); some types have been researched and found interesting enough to be worth adding hardware support for, such as the float16[7][41] and bfloat16[98][52] formats. The goal of the research into useful numerical precisions is to determine precisions which strike a balance between runtime performance and numerical performance by reducing the precision as much as possible without losing fidelity of results. This chapter will first present a framework developed to perform experiments which vary the numerical precisions used in neural network inference, then it will present the results of the first experiments performed using this framework.

The framework presented in Section 7.1.2 is a neural network inference engine that allows the use of arbitrary data types. It is even possible to use user-defined custom types with minimal effort (Sec. 7.1.2). In addition to allowing the use of arbitrary types, it also allows the inference data types to be set per-layer. This flexibility enables a vast array of possible experiments, to the point that the question is not "What can we test?" but "What should we test?".

The experiments presented in Section 7.2 consist of computing the mAP (described Sec. 4.4) when performing inference of the PVANet network with various data types. Each experiment performs inference using two data types – 32-bit floating point and a specific custom data type. Which layers use which data types during inference depends on the experiment being performed. For example, the experiment performed in Section 7.3.2 shows the mAP when computing the Convolution layers with each chosen fixed-point data type and the remaining layers with 32-bit floats.

7.1 . Software

In order to test the effects of reducing the numerical precision of neural network inference, it is necessary to have an inference engine capable of using arbitrary types during inference. A survey of the existing engines failed to find any that met this requirement. As a result, we developed an in-house neural network inference engine capable of computing with arbitrary types via C++ template metaprogramming in order to perform the experiments presented in this section.

7.1.1 . Existing Inference Engines

Table 7.1 compares some of the most commonly used neural network software on the following categories:

- **Types** - Supported types. Integers, IEEE float16 and bfloat16 types are considered.

Info			Types				Features						
Library	Ref	License	int	f16	bf16	Ext	Mixed	SIMD	CUDA	OpenMP	Portable	Backend	API
Caffe	[56]	BSD-2	N	Y	N	N	Y∗	BLAS	Y	Y	N	C++	C++
PyTorch	[82]	BSD	Y	Y	Y	N	Y†	MKL	Y	Y	Y	C++	Python/C++
mxnet	[14]	Apache-2	N	Y	N	N	Y†	Varies‡	Y	Y	Y	C++	Many
TensorFlow	[1]	Apache-2	Y	Y	Y	Y	Y†	Eigen	Y	N	Y	C++	Many
Custom	7.1.2	TBD	Y	Y	Y	Y	Y	nsimd	N	Y	Y	C++	C++

∗ The base caffe does not support mixed-precision inference, but the NVIDIA fork does
† Limited types allowed in mixed-precision
‡ SIMD implementation depends on backend library being wrapped

Table 7.1: Feature comparison of various neural network software.

- **Ext** - whether the software can be **Ext**ended to support alternative datatypes.
- **Mixed** - Availability of mixed-precision inference.
- **SIMD** - Library used to achieve SIMD vectorization. BLAS and MKL only provide high-level vectorized functions.
- **CUDA** - Presence of CUDA support.
- **OpenMP** - Usage of OpenMP for parallelization.
- **Portable** - Portability between at least x86 and ARM platforms.
- **Backend** - Programming language of software backend.
- **API** - Programming language of provided APIs.

Caffe is a deep learning framework initially developed by the University of California Berkeley and now developed by Facebook. It is somewhat portable, although the portability takes the form of different forks of the codebase for different architectures. Despite using C++ templates in its codebase, it only supports IEEE 16, 32, and 64 bit floats. It does not allow for easy extension of the supported types. In addition, its mixed-precision capabilities are limited to mixing 16 and 32 bit floats.

PyTorch is a machine learning library which includes deep learning as one of its features. It is also owned by Facebook, and Caffe has recently been merged into the PyTorch library[12]. Being a python library, it is generally portable. It defines a number of types that can be used in computation including integers, booleans, bytes, half precision floats, and bfloat16 floats. In addition, its mixed-precision capabilities are limited to mixing 16 and 32 bit floats.

mxnet is a deep learning framework developed by Apache. In addition to providing deep learning functionality, it can also wrap other deep learning frameworks, allowing for greater portability. Despite using C++ templates in its codebase, it only supports IEEE 16, 32, and 64 bit floats. It does not allow for easy extension of the supported types.

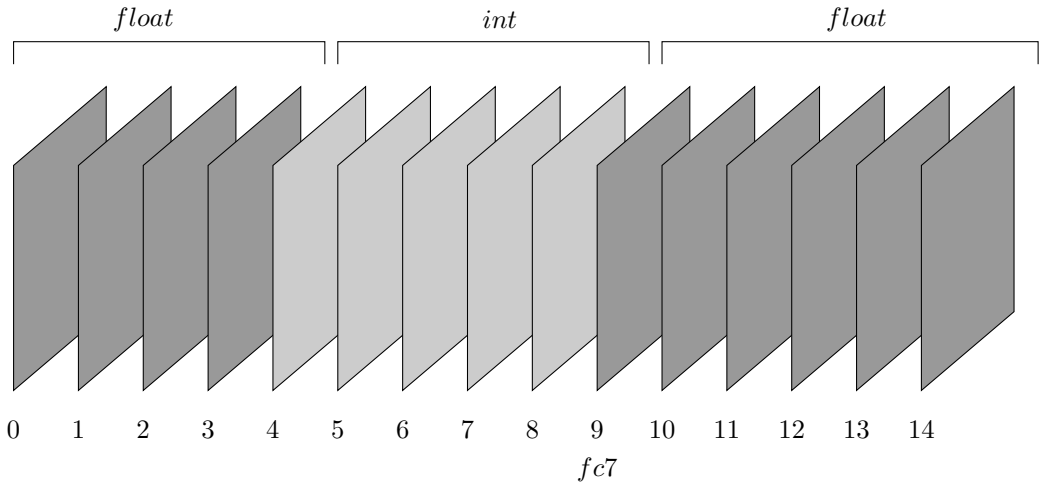

Figure 7.1: Example of a division of a neural network

TensorFlow is a software library developed by Google which contains machine learning and deep learning features. It is used on a variety of platforms and supports a variety of types. Its Python API allows for extending the supported datatypes via a `variant` umbrella type. However, usage of the `variants` appears to disable parallelization.

Custom This work presents a custom inference-only neural network engine developed by Agenium Scale with Université de Clermont-Auvergne. It consists entirely of header-only C++ code and makes use of template metaprogramming. The templates are used in such a way that adding new datatype support is relatively simple (see Section 7.1.2). In addition, platform-specific optimizations are implemented using the `nsimd` software library, allowing for both portability and performance.

7.1.2 . Custom Inference Engine

In order to perform the experiments presented in Section 7.2, it was necessary to develop a custom neural network inference engine. This engine was developed in C++ and leverages both template metaprogramming and operator overloading in order to support arbitrary data types. Usage of template metaprogramming is nonetheless limited to C++98 features to allow for wider portability of the engine.

The goal was to develop an engine capable of controlling the data types used in inference with layer-level granularity. Manually selecting the data type of each layer would be a tedious process, so the engine implements a system of *transitions* to define boundaries between regions using different data types. Each region between transitions consists of a series of layers using the same data type. Each transition represents a boundary where the data type being used changes.

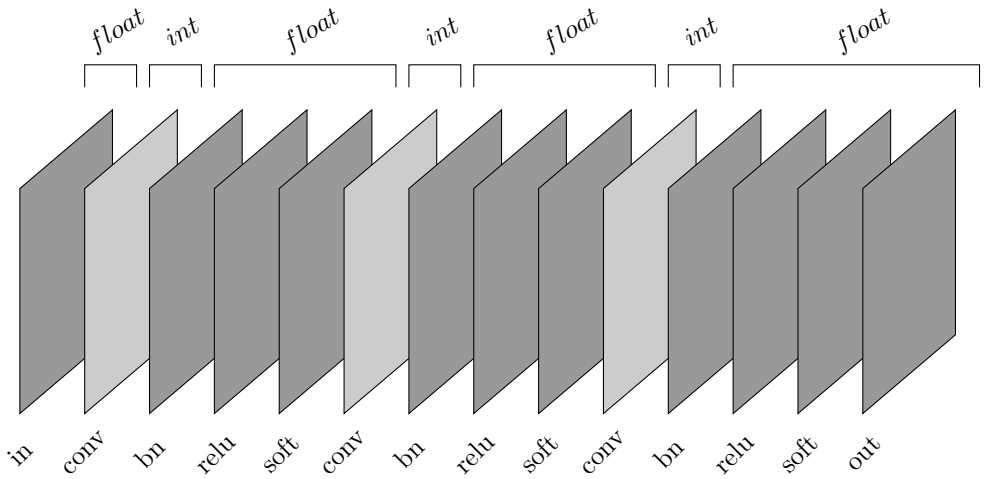

Figure 7.2: Example of a network using `int`s for Convolutions and `float`s for the rest.

Defining Transitions When writing code using the custom inference engine, transitions may be specified using either their number in the order of layers or their name in the `.prototxt` file defining the network. The layer listed in the transition is considered to be the first layer to use the new type. The data type used in the last transition will be used in all layers until the end of the network. The first defined data type will be used in all layers from the start until the first transition. If there is no layer corresponding to a defined transition, there will be a warning output to the standard output. For example, the code below defines the transitions needed to implement the regions defined in Figure 7.1.

```
1 std::vector<ns::Transition> transitions;
2 transitions.push_back( ns::Transition( ns::Type1 , ns::
   Type2 , 4 ) );
3 transitions.push_back( ns::Transition( ns::Type2 , ns::
   Type1 , "fc7" ) );
```

The transitions can be defined in any order. This array of transitions must be defined before creating a network object (to allow for memory optimizations).

Once the array of transitions is defined, the network object may be created. By default, networks using 2 and 3 different types during inference are supported. However, it is possible to support any number of data types by generating the appropriate header files. This is done by running the python script `tools/multi_network/gen_multi_network.py -n [# types]` with the number of types needed as an argument. Code generation is used for this instead of template metaprogramming for ease of development and to allow the possibility of compiling using older C++ standards. The object used to represent a network using X data types during inference is `ns::mixed_network_`X.

Loading a Network As of this writing, there are two ways to initialize a network object. The first way is to use a prebuilt `caffe` network to read the `.prototxt` and `.caffemodel` files and allow the network to copy the appropriate data. The second way is to convert the `.prototxt` and `.caffemodel` files to a custom format, then use the new files to load the network. In either case, the `multi_network` object is declared with a number of template arguments equal to the number of data types to be used plus one. The first template argument is the type used by the `.caffemodel` file in order to store the weights of the network. All following template arguments are the data types to be used during inference. Below is an example of a code that loads the information using a `caffe` object.

```
ns::mixed_network_2<float,int,float> custom_network(
 caffe_network , transitions );
```

Accessing Data It is fairly simple to access the data within the network object for loading inputs and accessing outputs. This is done using the `block_by_name` method of the network object. This function takes 3 arguments – the name of the layer to be accessed, and two pieces of information about the layer's data type. The first piece of information is the data types number in the order of types declared in the declaration. For example, in the listing above this paragraph, the first inference type listed in `int`, so it corresponds to `ns::Type1`. The second piece of information is the data type being requested. This must be specified due to the limitations of C++ compilers' type deductions.

```
int *ImInfo = custom_network.block_by_name( "im_info" , ns
 ::Type1 , int() );
ImInfo[0] = width;
ImInfo[1] = height;
```

The output of this method is a pointer to the raw data containing the requested data. It can be directly accessed using the [] operator (unlike the commonly used `std::shared_pointer` which only implements the operator since C++11).

Performing Inference Once the input layers are loaded, performing inference is as simple as the following code.

```
custom_network.Evaluate();
```

If one wishes to perform inference on a subsection of the network, there is also the `EvaluateFromTo( int , int )` method.

Supported Layers The custom inference engine currently supports only the layers necessary to use the PVANet and YOLO neural networks. The list of supported layers is as follows: BatchNorm, Concat, Convolution, Deconvolution, Eltwise, InnerProduct, Input, Pooling, Power, ProposalLayer (PVANet), ReLU, Reshape, ROIPooling, Scale, Softmax, Split, Upsample, Yolov3DetectionOutput (YOLO).

Supported Data Types

The custom inference engine is capable of performing inference using any data type that overloads the basic arithmetic operators (+-*/), implements three mathematical functions (sqrt, pow, exp), and implements the cast operator (operator()). If the data type meets these standards, adding support to the engine requires specializing the three mathematical functions in the include/nsmath.hpp file. For example, the following code adds fixed point capabilities to the engine:

```
template <uint8_t lf, uint8_t rt>
using fp_t = nsimd::fixed_point::fp_t<lf, rt>;

template <uint8_t a, uint8_t b>
fp_t<a, b> sqrt(fp_t<a, b> in) { return nsimd::fixed_point
 ::sqrt(in); }
template <uint8_t a, uint8_t b>
fp_t<a, b> pow(fp_t<a, b> in, fp_t<a, b> pow) {return nsimd
 ::fixed_point::exp(in, pow); }
template <uint8_t a, uint8_t b>
fp_t<a, b> exp(fp_t<a, b> in) { return nsimd::fixed_point::
 exp<a, b>(in); }
```

And the following code adds integer support to the engine:

```
template <> int sqrt(int in) { return int(std::sqrt(float(in)
 )); }
template <> int pow(int in, int pow) { return int(std::pow(
 float(in), float(pow))); }
template <> int exp(int in) { return int(std::exp(float(in)))
 ; }
```

Code Generator

In order to facilitate rapid prototyping and enable experiments that would be tedious to hand-code, a code generator is provided in the tools/gen_experiments/gen_experiments.py file. In its current state, the code generator only supports PVANet. However, it is designed to allow easy extension for other neural networks. In order to generate an experiment, the code generator requires two input files.

The first required file contains the layer information for the network being used. This file allows the code generation to occur much faster than if the generator loaded the entire network to obtain the layer information. The layer information file can be obtained by compiling src/cpp/layer_numbers.cpp and running it on the network of interest.

The second required file is a file containing the experiment parameters. It must be placed in the tools/gen_experiments/parameters/ directory. This file must define a number of

python functions that will be called by the code generator. The following code is a minimal example of a parameter file, commented to explain each function being defined.

```
# Data types to use : first entry will be the initial/entry type
def type_list():
  return [ "float" , "int" ]

# Type that caffe was trained with. 99% chance of float
def caffe_type():
  return "float"

# Layers that we need the pointers to
def output_layers():
  return [
            'rois'
          , 'bbox_pred'
          , 'cls_prob'
         ]

# Input layers that we need pointers to
def input_layers():
  return [
           'data'
         , 'im_info'
         ]

# Constraints: "I want this layer to have this type"
# Format of network is [ [ layer number , layer type , layer name , outputs... ] ... ]
def GenConstraints( network , types ):
  return SingleTransition( network , types , 10 )

def SingleTransition( network , types , num ):
  ret = []
  if ( num > len(network) ):
    print( "No transitions generated - num > number of layers " )
    return ret
  ret.append( [network[num  ][2] , types[1] ] )
  ret.append( [network[num+1][2] , types[0] ] )
  return ret
```

This example generates an experiment that uses the `float` data type for the first 10 layers

(layers 0-9), then uses the `int` data type for the remaining layers. For a more interesting example of what is possible with the code generator, the following definition of the `GenConstraints` function creates an experiment where all Convolution layers use `int` data types, while the rest of the network uses `float` data types. See Figure 7.2 to visualize the type of network that would be generated.

```
def GenConstraints( network , types ):
  return OnlyConvolution( network , types )

def OnlyConvolution( network , types ):
  ret = []
  for i in range( 0 , len(network) ):
    if ( network[i][1] == "Convolution" ):
      ret.append( [network[i  ][2] , types[1] ] )
      ret.append( [network[i+1][2] , types[0] ] )
  return ret
```

Generating the Code Once the appropriate input files have been written, the code generator can be used to generate the desired experiment(s). By default, with only the required flags, the code generated will use MPI to perform inference on all images in a directory and output the resulting bounding boxes in a chosen location.

- `-h` - help
- `-l [file]` - Required - layer information file created by running an auxiliary executable.
- `-o [file]` - Required - where to output generated C++ file.
- `-c [file]` - Required - configuration file that defines the experiment being generated.
- `-no-mpi` - Optional - disable MPI.
- `-vis` - Optional - visualize results using OpenCV.
- `-no-write` - Optional - do not output bounding boxes to a file.
- `-rec` - Optional - output images with bounding boxes.
- `-batch` - Optional - batched input. Generated code will take a list of files instead of a directory as input.

For prototyping, it is suggested to use the `-no-mpi`, `-vis`, and `-no-write` flags to easily visualize the results without storing them. This allows one to see if a set of parameters results in a visually coherent output. Otherwise, the default flags are ideal for inferring a large number of images in order to compute an mAP.

7.1.3 . Numerical Types

The tests presented in this chapter are performed using 3 categories of data types – natively supported types, customizable floating point types (discussed below), and customizable fixed point types(presented in Chapter 6. The purpose of using these data types is to explore what types could be interesting to use in neural network without limiting ourselves to the types currently supported by hardware.

Custom Floats

In order to perform experiments limiting the floating point format, a suitable data type is needed to represent the modified floats. A few qualities were sought after.

First, the data type needs to be capable of fully customizing and limiting the size of both the mantissa and the exponent. This disqualifies the Boost Multiprecision[9] library, as it does not allow full customization of the exponent.

Second, the data type needs to be capable of easily representing multiple different floating point formats. This disqualifies the MPFR[31] software library, as the exponent and mantissa bits are specified as global variables.

Finally, the data type needs to emulate as well as possible the format being chosen. This disqualifies any approaches leveraging expression templates, as they only respect the chosen format during storage operations and not while performing chains of intermediate computations.

Ultimately, it was simplest to code a limited custom floating point format. This custom implementation uses a native 32-bit float to store values and perform operations. After every operation performed upon a custom float, the exponent is checked for overflow. If the value should overflow, the value stored is $\pm$`infty`. Otherwise, a mask is applied to remove any extra precision in the mantissa.

There are three notable custom floating point configurations to consider when examining the results presented in Section 7.3.1. These are the IEEE 32-bit float (8-bit exponent, 24-bit mantissa), IEEE 16-bit float (5-bit exponent, 11-bit mantissa), and bfloat16 format (8-bit exponent, 8-bit exponent)

7.2 . Methodology

The ultimate goal is to reduce numerical precision at intervals that allow for better SIMD parallelism. However, the flexibility of the inference engine allows us to test an incredible variety of numerical precisions. We test broad ranges of numerical precision, regardless of SIMD performance, in order to see if any patterns emerge.

Test Design In order to study which numerical precisions can be acceptable, we chose to compute the mAP (described in Sec. 4.4) for a range of combinations of numerical precisions. All results presented here use 32-bit floating point types when not otherwise specified. The

custom numerical precisions used were the custom floating point types and the fixed point types.

Both types were tested in a similar fashion. For each experiment, one type of layer is chosen (for example, the Convolution layer). During inference, all layers of this type use the data type being studied. All other layer types use the default IEEE 32-bit float during inference. Inference is performed upon all images of the VOC-2007 dataset. The results are used to compute a VOC-2007 mAP for the given data type and layer being studied.

The experiments on fixed point formats and custom floats differ in one manner – exhaustivity. The custom floating point types are tested exhaustively for all configurations (184).

$$8 \text{ mantissa possibilities} * 23 \text{ exponent possibilities} = 184 \text{ configurations}$$

However, the fixed point format (Qa.b) is capable of many more configurations (2080), so it is not tested exhaustively.

$$\text{a} + \text{b} \leq 64 \rightarrow \sum_{i=0}^{64} i = 2080 \text{ configurations}$$

The methods used to select which fixed point configurations were tested are described in Section 7.3.2

Testing Hardware All experiments presented here were performed by our colleagues at the Université Clermont Auvergne using the Mesocentre, their computational cluster. The Mesocentre comprises 42 computational nodes with a total of 720 physical cores and 6 TB of RAM. The neural network inference computations performed in these experiments are trivially parallelizable and thus well suited to making use of such a computational cluster. We used MPI to parellelize the inference computations required in order to be able to compute the mAP for each configuration.

For context, every single mAP computation requires the computation of 4,952 inferences. At approximately 10 seconds per inference, this represents more than 8 hours of computation for a single data point of each graph presented in this section. Access to a computational cluster allowed the experiments to involve significantly more data points.

PVANet All experiments presented here are performed using the PVANet neural network (presented in Section 4.5). This network was chosen for a number of reasons.

- It is relevant as a state of the art network, having been published in 2016[46].
- The source code is open-source[45], allowing for reproduction of the published results.
- A number of different layer types (17) are used, allowing for varied experimentation.

- It uses a (slightly modified version of a) well-known open-source deep learning framework. Ensuring that the custom inference engine works for PVANet ensures that it is caffe-compatible.

Before running any tests, we ensured that the inference engine is capable of perfectly reproducing the mAP results obtained via the code on the PVANet code repository[45].

7.3 . Results

This section presents the results of the experiments described in Section 7.2, separated first by data type studied and then by layer type studied.

7.3.1 . Custom Floats

This section presents the results of computing the mAP when computing specific layers using all valid custom float configurations. For each layer presented, the layer in question is computed using a custom float format and the remaining layers using normal IEEE 32-bit floats. Each graph shows how the mAP varies with the size of both the mantissa and the exponent. Remember that, as described in Section 3.1, a 1-bit mantissa represents 2-bits of information thanks to normalization.

BatchNorm

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.3. Most evidently, this graph shows a dramatic cliff between a 5-bit exponent and a 6-bit exponent. This indicates the presence of a threshold below which the computation fails due to overflow and above which there are no problems. There appears to be no gain from going above 6 bits in the exponent.

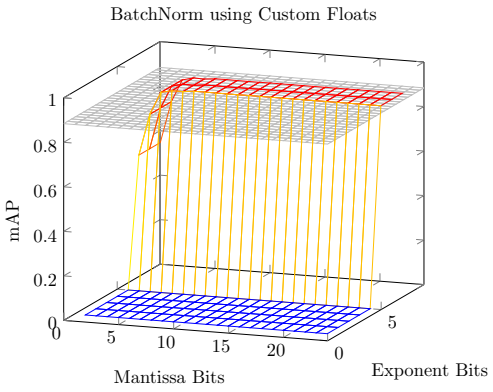

Figure 7.3: mAP using custom floats in the BatchNorm layer.

When looking at the effects of varying the number of bits in the mantissa, we see that the BatchNorm layer is very tolerant of low precision. For the usable exponents (6, 7, 8 bits), the mAP is approximately 0.57 for a 1-bit mantissa, 0.76 for a 2-bit mantissa, and a 3-bit mantissa is sufficient to nearly match the initial 0.88 mAP with floats.

Overall, the BatchNorm layer appears to allow for strongly reduced precision – it is possible to reduce the size to an 8-bit format (2-bit mantissa, 6-bit exponent) and still obtain reasonable results.

Convolution

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.4. Once more, this graph shows a dramatic cliff, this time between a 4-bit exponent and a 5-bit exponent. As in the BatchNorm layer, this appears to be a threshold effect such that once the 5-bit threshold is reached, there are no gains to be had by further increasing the number of bits in the exponent.

Figure 7.4 indicates that an 11-bit mantissa is sufficient to nearly replicate the performance of floats (mAP of 0.882 vs 0.883). On the lower end, 8 bits in the mantissa are the minimum to produce coherent results (mAP = 0.70).

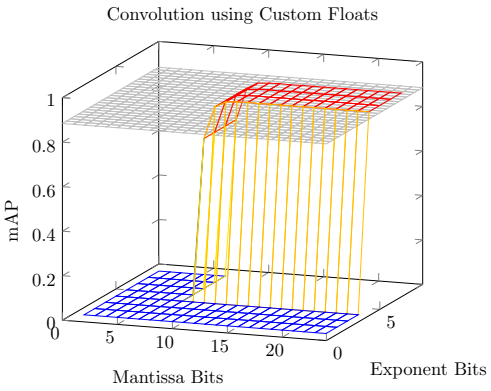

Figure 7.4: mAP using custom floats in the Convolution layer.

The Convolution layer can easily be computed using a 16-bit floating point format. An IEEE half-precision float provides 11 bits of mantissa and 5 bits of exponent, which is sufficient to nearly replicate the original performance. Meanwhile, a bfloat16 with an 8-bit exponent and 8-bit mantissa, is sufficient to achieve degraded results.

Deconvolution

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.5. As with the previous layers, this graph shows a cliff along the exponent axis between 3 bits and 4 bits. Once the 4-bit threshold is reached, there are no further gains to be had by increasing the size of the exponent.

Along the mantissa axis, Figure 7.5 is nearly flat. With a 1-bit mantissa, it is already possible to obtain an mAP of 0.87, leaving very little room for improvement before reaching the original mAP of 0.883. This indicates that the Deconvolution layer is fairly insensitive to specific values, as opposed to orders of magnitudes.

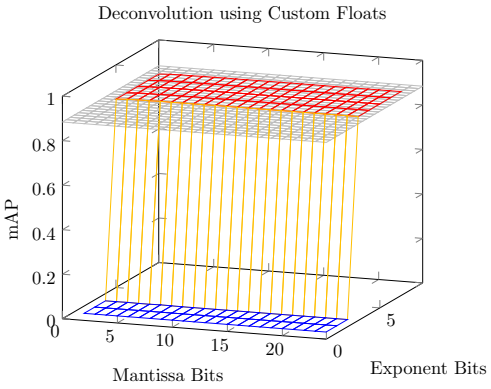

Figure 7.5: mAP using custom floats in the Deconvolution layer.

Overall, the Deconvolution layer can tolerate significantly reduced precision – it is possible to nearly replicate the original performance with only 5 bits (1-bit mantissa with 4-bit exponent).

Eltwise

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.6. This graph shows a less dramatic cliff along the exponent axis than the previous layers examined – it occurs between 3 bits and 5 bits. Once at the top of the cliff at 5 bits, there are no further gains from increasing the number of bits in the exponent.

Along the mantissa axis, a similar cliff can be observed. At 1 bit, the results are incoherent. At 2 bits, the results are degraded, but coherent (mAP = 0.74). At 3 bits, the results already approach the original performance (mAP = 0.86).

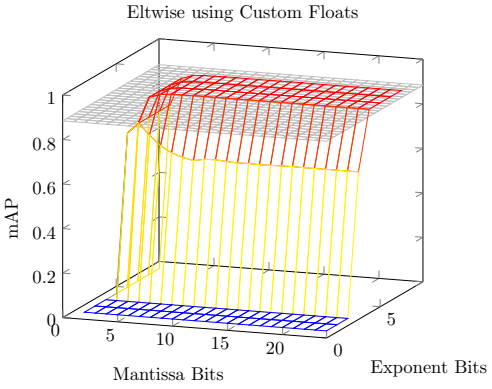

Figure 7.6: mAP using custom floats in the Eltwise layer.

Overall, the Eltwise layer does not need much precision – it is possible to nearly replicate the original performance with 8 total bits (3-bit mantissa and 5-bit exponent).

InnerProduct

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.7. This graph shows the usual cliff along the exponent axis between 2 bits and 4 bits. There are no gains from increasing the size of the exponent beyond 4 bits.

The mantissa axis shows the least steep incline so far, showing most of the growth between 7 and 11 bits. At 11 bits, the mAP (0.87) approaches the original performance using floats.

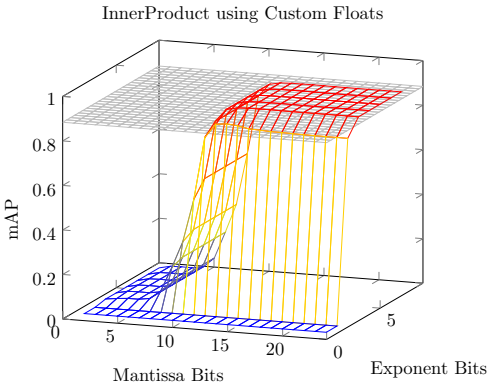

Figure 7.7: mAP using custom floats in the InnerProduct layer.

Overall, the InnerProduct computation can be done using a 16-bit format. With the 11 bits mantissa and 5 bits exponent of a IEEE half, it allows for an mAP of 0.87 – very nearly the original performance. However, a bfloat16 with 8 bits mantissa and 8 bits exponent only allows for an mAP of 0.32 – a strongly degraded result.

Pooling

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.8. This graph shows the cliff along the exponent axis between 3 bits and 4 bits. There are no gains from increasing the size of the exponent beyond 4 bits.

Along the mantissa axis, Figure 7.8 is nearly flat. With a 1-bit mantissa, it is already possible to obtain an mAP of 0.86, leaving very little room for improvement before reaching the original mAP of 0.883. This indicates that the Pooling layer is fairly insensitive to specific values, as opposed to orders of magnitudes.

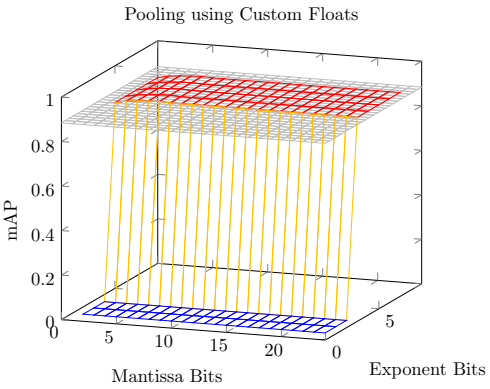

Figure 7.8: mAP using custom floats in the Pooling layer.

Overall, the Pooling layer can tolerate significantly reduced precision – it is possible to nearly replicate the original performance with only 5 bits (1-bit mantissa with 4-bit exponent).

ProposalLayer

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.9. As usual, this graph along the exponent axis shows a cliff between, this time between 4 and 5 bits. There appears to be no gain from going above 5 bits in the exponent.

Along the mantissa axis, a similar cliff can be observed. At 1 bit, the results are incoherent. At 2 bits, the results are degraded, but somewhat useful (mAP = 0.50). At 3 bits, the results are degraded, but coherent (mAP = 0.74). At 4 bits, the results already approach the original performance (mAP = 0.84).

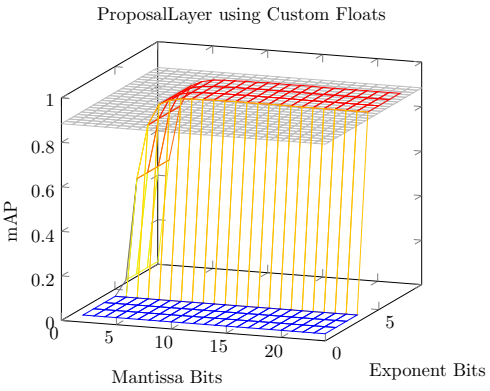

Figure 7.9: mAP using custom floats in the ProposalLayer layer.

Overall, the ProposalLayer layer can tolerate significantly reduced precision – it is possible to approach the original performance with only 8 bits (4-bit mantissa with 4-bit exponent).

ReLU

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.10. This graph shows a less dramatic cliff along the exponent axis than the previous layers examined – it occurs between 3 bits and 4 bits. Once at the top of the cliff at 4 bits, there are no further gains from increasing the number of bits in the exponent.

Along the mantissa axis, a similar cliff can be observed. At 1 bit, the results are incoherent. At 2 bits, the results are degraded, but coherent (mAP = 0.74). At 3 bits, the results already approach the original performance (mAP = 0.86).

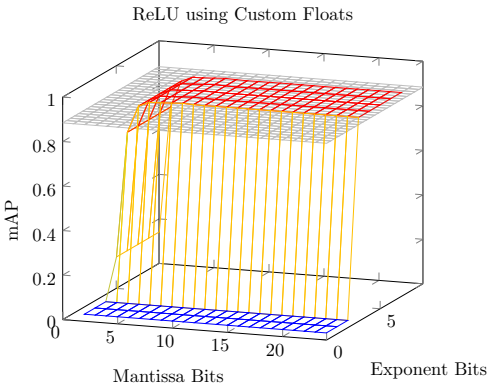

Figure 7.10: mAP using custom floats in the ReLU layer.

Overall, the ReLU layer does not need much precision – it is possible to nearly replicate the original performance with 8 total bits (4-bit mantissa and 4-bit exponent).

ROIPooling

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.11. As usual, this graph along the exponent axis shows a cliff between, this time between 4 and 5 bits. There appears to be no gain from going above 5 bits in the exponent.

Along the mantissa axis, a similar cliff can be observed. At 1 bit, the results are incoherent. At 2 bits, the results are degraded, but somewhat useful (mAP = 0.63). At 3 bits, the results already approach the original performance (mAP = 0.80).

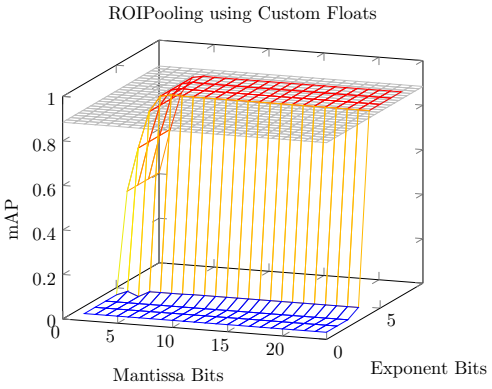

Figure 7.11: mAP using custom floats in the ROIPooling layer.

Overall, the ROIPooling layer can tolerate significantly reduced precision – it is possible to approach the original performance with only 8 bits (3-bit mantissa with 5-bit exponent).

Scale

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.12. This graph shows the usual cliff along the exponent axis between 3 bits and 5 bits. There are no gains from increasing the size of the exponent beyond 5 bits.

Along the mantissa axis, a similar cliff can be observed. At 2 bit, the results are incoherent. At 3 bit, the results are strongly degraded, but occasionally useful (mAP = 0.34). At 4 bits, the results are degraded, but somewhat useful (mAP = 0.76). At 5 bits, the results already approach the original performance (mAP = 0.86).

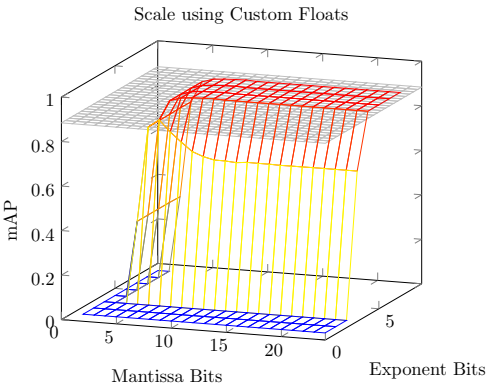

Figure 7.12: mAP using custom floats in the Scale layer.

Overall, the Scale computation can just barely be done using an 8-bit format with a 4-bit mantissa and 4-bit exponent (mAP = 0.76). However, it becomes much more reliable when adding one more bit to both the mantissa and exponent.

Softmax

The mAP for all tested combinations of mantissas and exponents is shown in Figure 7.13. This graph shows the cliff along the exponent axis between 0 bits and 2 bits. There are no gains from increasing the size of the exponent beyond 2 bits.

Along the mantissa axis, Figure 7.13 is nearly flat. With a 1-bit mantissa, it is already possible to obtain an mAP of 0.881, leaving very little room for improvement before reaching the original mAP of 0.883. This indicates that the Softmax layer is fairly insensitive to specific values, as opposed to orders of magnitudes.

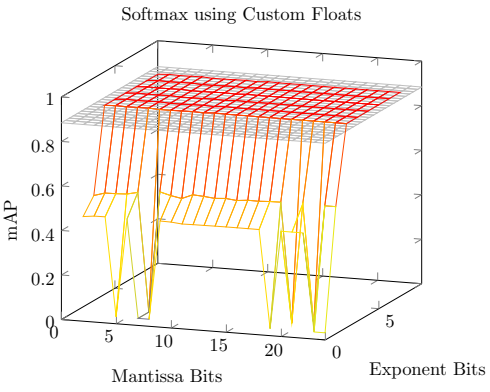

Figure 7.13: mAP using custom floats in the Softmax layer.

Overall, the Softmax layer can tolerate significantly reduced precision – it is possible to nearly replicate the original performance with only 3 bits (1-bit mantissa with 2-bit exponent).

7.3.2 . Fixed Point

This section presents the results of computing the mAP after performing inference where specific layers use certain fixed point formats with the nsimd fixed point module. For each layer presented, the layer in question is computed using various fixed point formats and the remaining layers using normal IEEE 32-bit floats. Each plot contains three graphs. The first graph represents an initial exploratory run that tests a variety of Qa.b formats by symmetrically increasing the number of bits in both the integer and decimal portions of the fixed point numbers. When this ceases to be possible due to the maximum storage size of $a + 2b < 64$, bits, we continue increasing the decimal bits while decreasing the number of integer bits. The second graph sets the integer bits to 3 and varies the number of decimal bits in order to explore the effects of having a low number of integer bits. The third graph sets the integer bits to 21 and varies the number of decimal bits in order to explore the effects of having a larger number of integer bits.

Each plot additionally contains a number of colored dashed lines. Every dashed line represents a change in the storage type of the fixed point numbers being represented by the curve of the same color. The storage type to the right of the dashed line is larger than the storage type to the left of the dashed line.

Integer Bits When examining the results in this section, the effects of the extra integer bits described in Section 6.4.2 (paragraph "Alignment") will become apparent in some cases. As a reminder, when the total number of bits required to store a fixed point number is less than the total number of bits available for storage, the extra bits can be used as extra integer bits. For example, a Q4.4 number requires 12 bits but is stored using 16 bits. These 4 extra bits can be used as integer bits during certain computations.

BatchNorm

Figure 7.14 shows the results of the fixed point experiments on the BatchNorm layer. The results are clear and simple – it is not possible to compute the BatchNorm layer using the nsimd fixed point module. This is due to the presence of certain parameter values of very low magnitude which are used as themselves and in their reciprocal form. As a result, the BatchNorm computation within PVANet network requires a larger number of integer and decimal bits than our numerical format allows.

Convolution

Figure 7.15 shows the results of the fixed point experiments on the Convolution layer. Along all three graphs, there is a cliff between 8 and 10 decimal bits where the mAP rises from near zero to nearly 0.6. After this cliff, there does not appear to be significant variation (with one exception explained below) in the mAP depending on the number of decimal bits. However, none of the tested fixed point formats approach the initial mAP of 0.883.

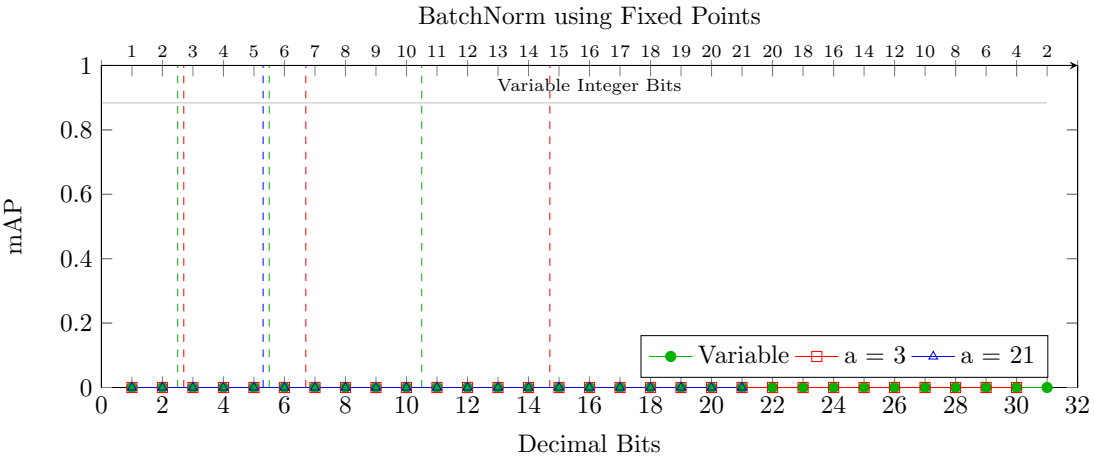

Figure 7.14: mAP using different fixed point formats in the BatchNorm layer. a = number of integer bits.

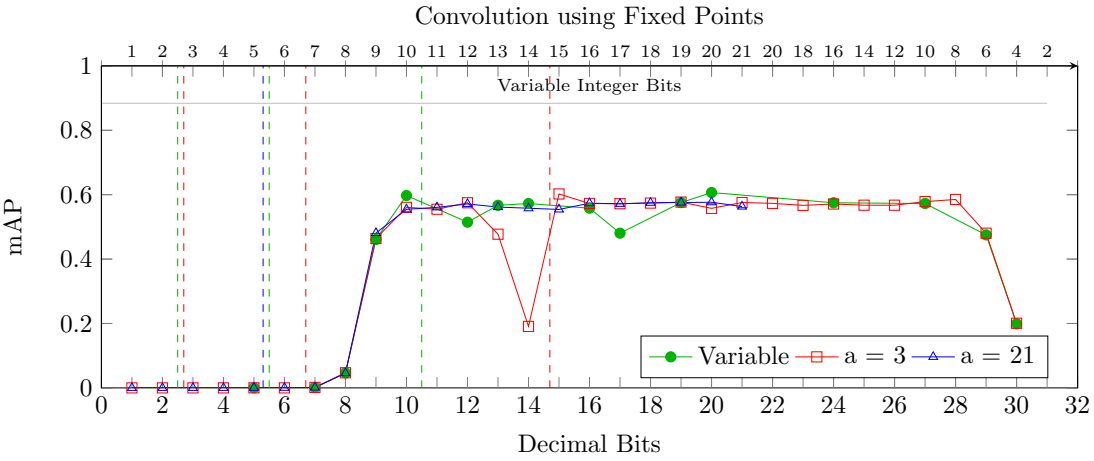

Figure 7.15: mAP using different fixed point formats in the Convolution layer. a = number of integer bits.

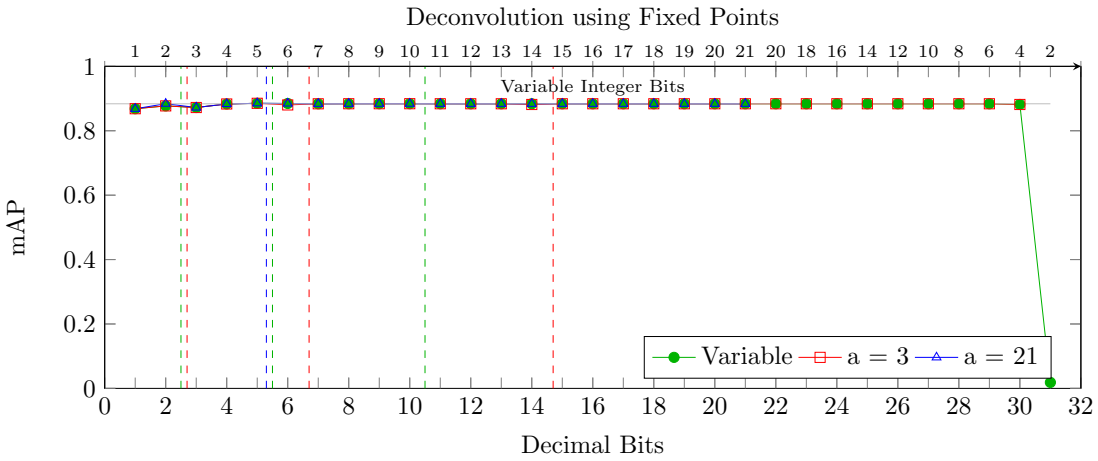

Figure 7.16: mAP using different fixed point formats in the Deconvolution layer. a = number of integer bits.

The Q3.X graph allows us to infer some of the effects of varying integer bits on the mAP, as evidenced by the dip at the Q3.13 and Q3.14 formats. These formats approach the upper limit of their storage types and thus do not benefit from many extra integer bits. The Q3.14 format effectively has 4 integer bits during computations, while the Q3.13 has 6 and the Q3.12 has 8. It thus appears that the lower bound to achieve the displayed mAP of 0.6 is 8 integer bits.

When comparing with Section 7.3.1, we can conclude that this loss of mAP is due to a lack of simultaneous integer and decimal precision in the types tested. Section 7.3.1 shows that a floating point value requires 11 bits of mantissa and 5 bits of exponent to approach the initial float32 mAP. Using fixed point numbers to represent the same range of values would require approximately 15 bits of integer bits and 25 bits of decimal bits. Unfortunately, this combination is not representable using our fixed point format, as it requires 65 bits of storage when accounting for the reserved bits. As a result, we fail to match the initial mAP with any tested fixed point format.

If future experiments are possible, they should be initially performed along the axes of 8 integer bits and 25 decimal bits.

Deconvolution

Figure 7.16 shows the results of the fixed point experiments on the Deconvolution layer. It appears that 4 integer bits are sufficient to obtain the initial mAP of 0.883, and any further increase is not needed. Due to the effective extra integer bits, the variable graph does not truly test integer bit numbers of 1 or 3. However, the rightmost point of the graph (Q2.31) does show that 2 integer bits are insufficient. This is coherent with the 4 exponent bit requirement found in Section 7.3.1.

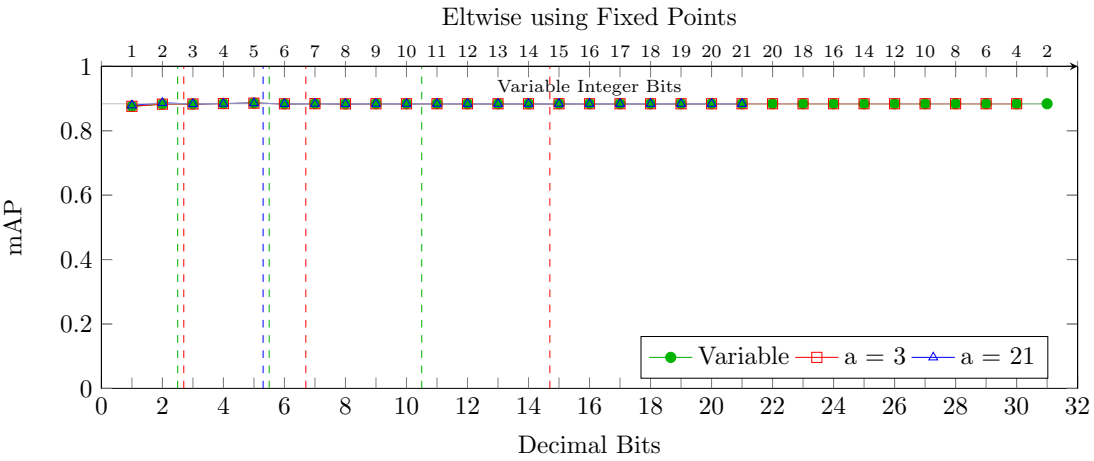

Figure 7.17: mAP using different fixed point formats in the Eltwise layer. a = number of integer bits.

Future experiments should test if 3 integer bits or 0 decimal bits are sufficient to maintain the 0.883 mAP.

Eltwise

Figure 7.17 shows the results of the fixed point experiments on the Eltwise layer. All tested configurations proved sufficient to replicate the initial mAP. This means that even the most extreme Q2.31 and Q7.1 formats were sufficient, which is more tolerant than the minimum 3 bit exponent and 5 bit mantissa found in Section 7.3.1. Future experiments would require a strict fixed point format in order to resolve the conflict between these findings.

InnerProduct

Figure 7.18 shows the results of the fixed point experiments on the InnerProduct layer. The most perplexing feature of this graph is the small peak at 6 decimal bits. This peak occurs on all three curves, so it cannot be discounted as a random occurrence. It also cannot be explained as a side effect of the unused bits in the storage type, as the three curves greatly differ in how they use the available bits. The Q3.X curve uses 15 bits with 1 extra, the Q21.X curve uses 33 bits with 31 extra, and the variable curve uses 18 bits with 14 extra. As of this writing, no viable explanation of this result has yet been found. Further experiments are required to test if this result is replicable and if it is truly consistent over the entire Qa.6 axis.

Ignoring the Qa.6 points, we observe the familiar presence of a cliff between 12 and 14 decimal bits along all three curves, showing that we can obtain an acceptable mAP of approximately 0.8 at 13 decimal bits and replicate the initial mAP with 14 bits. To the far right of the graph, we can observe that 2 integer bits are clearly insufficient, while 4 bits allow

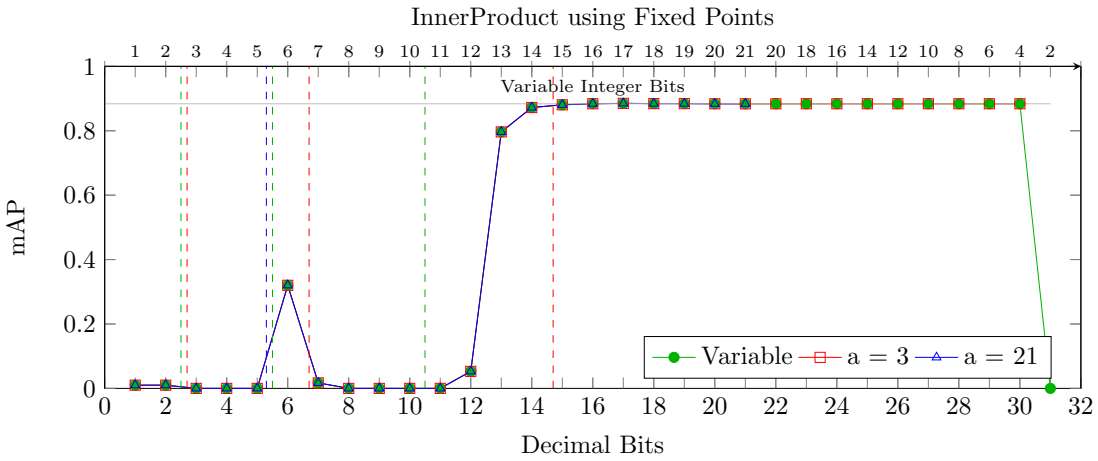

Figure 7.18: mAP using different fixed point formats in the InnerProduct layer. a = number of integer bits.

replication of the initial mAP. This is coherent with the findings of Section 7.3.1.

Pooling

Figure 7.19 shows the results of the fixed point experiments on the Pooling layer. Unfortunately, these results revealed and were impacted by a bug in the implementation of the Pooling layer that affects non-float types. Every Pooling layer in PVANet uses max pooling, which outputs the maximum value in a region. The implementation of this pooling type in the custom inference engine initializes the variable that stores the in-progress maximum value to be `-FLT_MAX`. However, this `-FLT_MAX` is far larger in magnitude than anything that a fixed point number can represent and causes behavior that depends on the fixed point format being converted to. This could have been avoided by instead initializing this value to the first value in the region to be examined.

This set of experiments must be re-run with the above bugfix in order to obtain usable results.

ProposalLayer

Figure 7.20 shows the results of the fixed point experiments on the ProposalLayer layer. The ProposalLayer results show two major inconsistencies.

The first inconsistency is the behavior of the Q3.X curve. When transitioning from 8-bit storage to 16-bit storage and 16-bit to 32-bit, we observe a drop to zero followed by a return to the baseline mAP. However, no such drop is observed when transitioning from 32-bit storage to 64 bits.

The second inconsistency is the Q3.2 mAP contrasted with the Q2.31 and the Q21.1

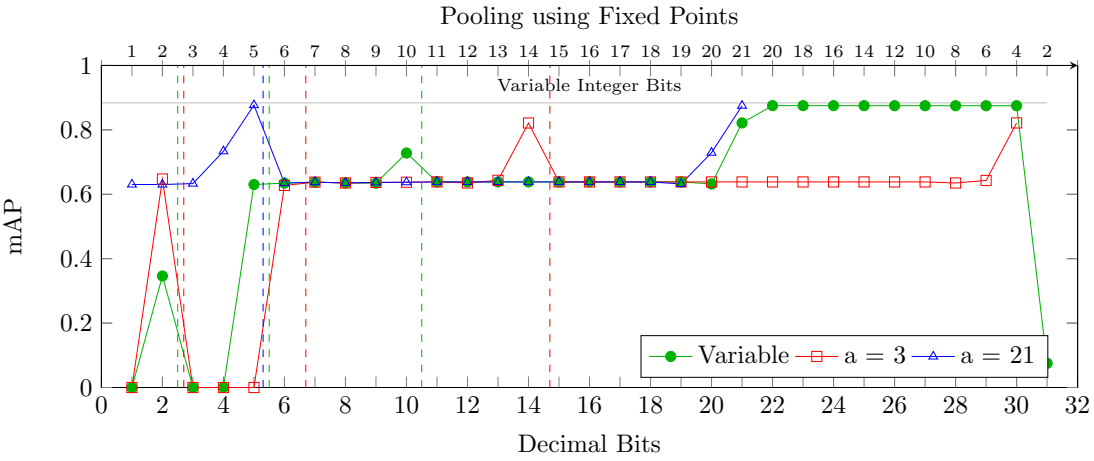

Figure 7.19: mAP using different fixed point formats in the Pooling layer. a = number of integer bits.

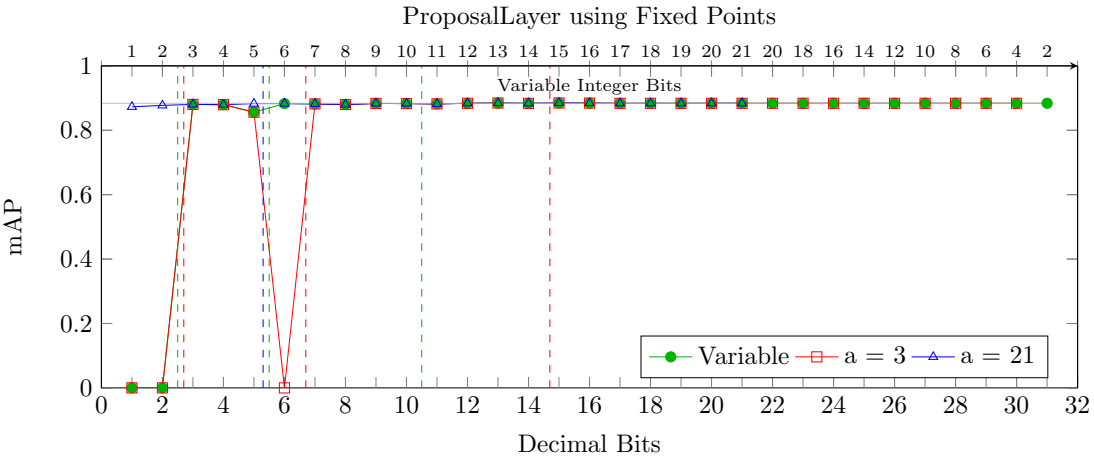

Figure 7.20: mAP using different fixed point formats in the ProposalLayer layer. a = number of integer bits.

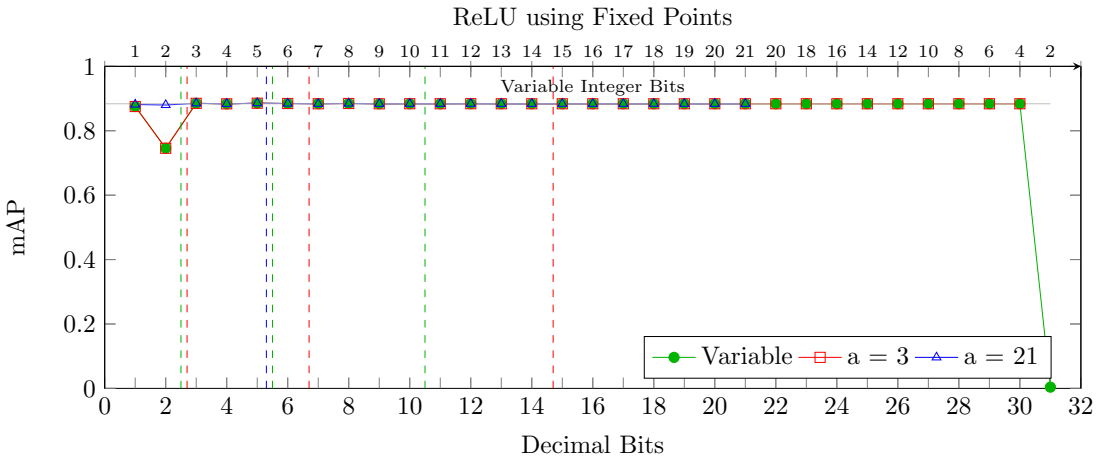

Figure 7.21: mAP using different fixed point formats in the ReLU layer. a = number of integer bits.

mAPs. At Q3.2, the mAP is zero, while both Q2.31 and Q21.2 display mAPs of 0.88. The Q2.31 format has less integer bits (declared and effective) while the Q21.1 has less decimal bits, yet both simultaneously display a higher mAP. This points towards a conclusion that the ProposalLayer requires either a certain number of decimal bits or a certain number of integer bits. However, this conclusion is not numerically coherent.

Neither inconsistency has yet been adequately explained. Further testing and verification of the source code are required in order to draw any real conclusions.

ReLU

Figure 7.21 shows the results of the fixed point experiments on the ReLU layer. The ReLU layer appears to have a minimum number of required integer bits and no dependence on the number of decimal bits. The Q2.31 data point shows that 2 integer bits appear to be insufficient to output acceptable results. In addition, we observe a small dip in the mAP at the Q3.2 and Q2.2 data points with 4 effective integer bits. However, neither the Q3.6 nor the Q3.14 show the same dip in mAP. This indicates that the small loss of mAP at 4 integer bits can be offset by having more decimal bits. Further testing with strict fixed point types would allow us to determine with certainty if this is the case.

ROIPooling

Figure 7.22 shows the results of the fixed point experiments on the ROIPooling layer. This graph resembles Figure 7.19, and for good reason. The ROIPooling layer suffers from the same type of software bug – it contains a variable that is initialized to `-FLT_MAX`. As a result, it is difficult to draw any suitable conclusions without re-running the experiments with the

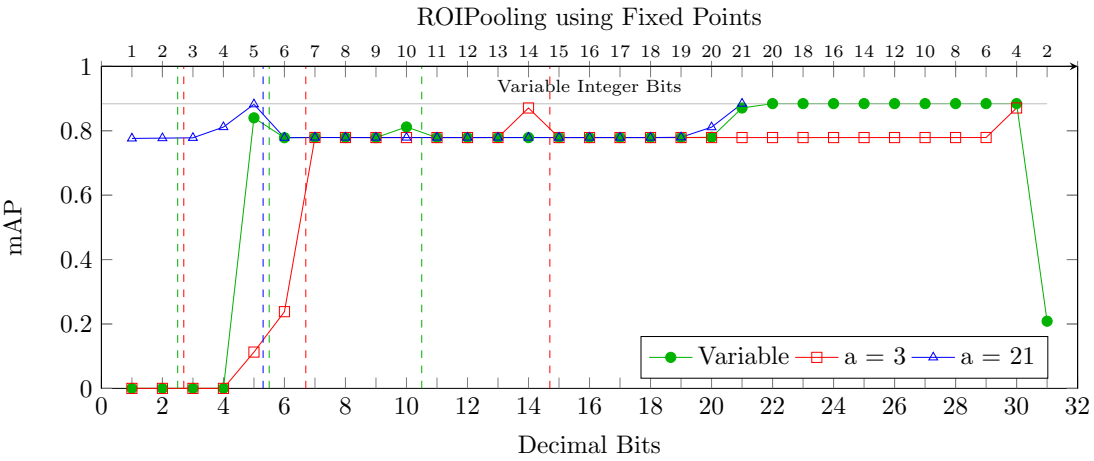

Figure 7.22: mAP using different fixed point formats in the ROIPooling layer. a = number of integer bits.

bug fixed.

Scale

Figure 7.23 shows the results of the fixed point experiments on the Scale layer. This graph shows a need for a minimum number of both integer and decimal bits. The Q21.X curve shows that, while 1 decimal bit can produce acceptable results, 2 decimal bits are needed to approach the initial mAP. For the integer bits, the curves consistently show that 4 bits is insufficient, 6 bits can be acceptable, but 8 are needed to approach the initial mAP. This can be seen in the dips in the variable and Q3.X curves which occur when the number of effective integer bits reduces before upgrading the storage type. These results appear to be coherent with the Section 7.3.1.

Softmax

Figure 7.24 shows the results of the fixed point experiments on the Softmax layer. These results are significantly worse than the results found in Section 7.3.1. In addition, the behavior of the curves is somewhat reminiscent of the Pooling and ROIPooling graphs. However, there does not appear to be any similar type of initialization bug in the Softmax layer. If this behavior is due to a software bug, it is a bug that has yet to be found.

Some differences may be due to the use of an exponential function, as the fixed point implementation has lower precision than the custom floating point implementation which computes the exponential using full float32 numbers (via `std::exp`). However, these differences are insufficient to explain the discrepancy between the fixed and floating point behaviors. We are still working to resolve this inconsistency.

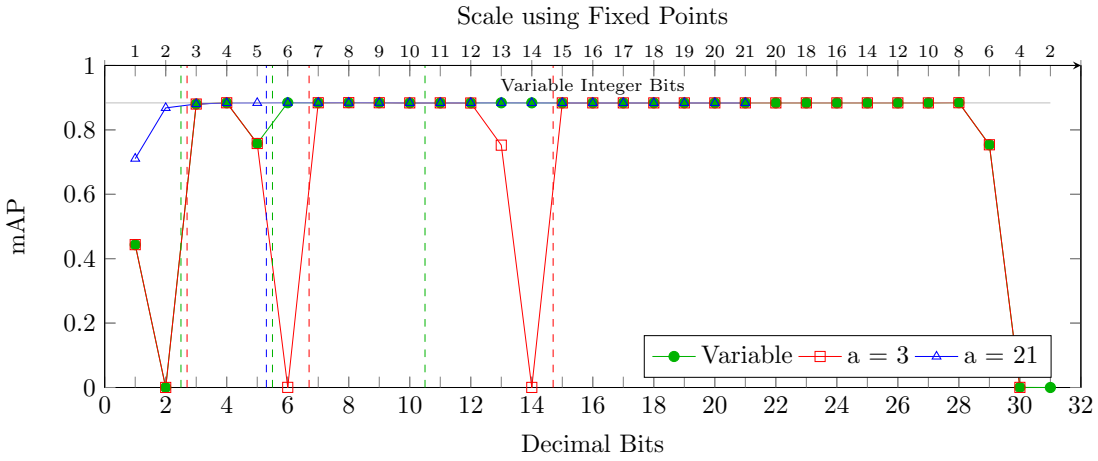

Figure 7.23: mAP using different fixed point formats in the Scale layer. a = number of integer bits.

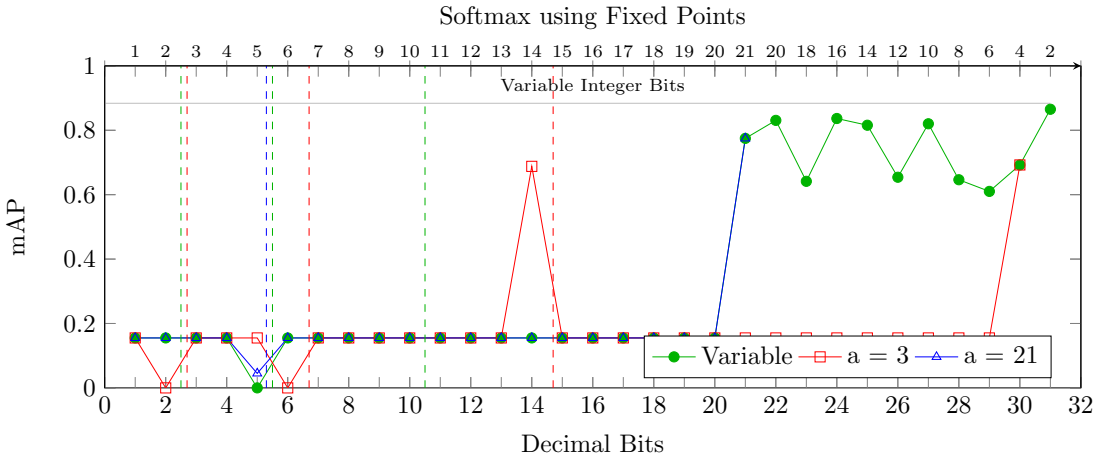

Figure 7.24: mAP using different fixed point formats in the Softmax layer. a = number of integer bits.

	Custom Float			Fixed Point		
Layer	Mantissa	Exponent	Total	Integer	Decimal	Total
BatchNorm	2	6	8	-	-	-
Convolution	11	5	16	-	-	-
Deconvolution	1	4	5	4	1	6
Eltwise	3	5	8	2	1	4
InnerProduct	11	5	16	4	14	32
Pooling	1	4	5	-	-	-
ProposalLayer	4	4	8	-	-	-
ReLU	4	4	8	4	3	10
ROIPooling	3	5	8	-	-	-
Scale	4	4	8	8	2	12
Softmax	1	2	3	-	-	-

Table 7.2: Minimum numbers of bits required to achieve $> 95\%$ of the initial mAP for tested layers.

7.4 . Conclusion

Table 7.2 summarizes the results of this chapter's experiments. In nearly all cases, the minimum number of bits required for a layer type is less with custom floating point numbers than fixed point numbers. This is due to the flexibility of floating point numbers compared to fixed point numbers.

In the floating point experiments, we can see that all layers can be compressed into 16 bits. Unfortunately, it is not quite possible to apply either of the natively support 16-bit floating point formats to all layers. The float16 (11-bit mantissa, 5-bit exponent) format does not have enough exponent bits to support the BatchNorm layer while the bfloat16 (8-bit mantissa, 8-bit exponent) format does not have enough mantissa bits to support the Convolution and InnerProduct layers. However, a combination of the two formats could allow the PVANet inference to be performed using entirely using natively supported 16-bit floating point numbers. If performed on hardware capable of accelerating computation involving both formats, this would achieve a theoretical 2x speedup over 32-bit floating point numbers.

The fixed point experiments feel much less complete than the floating point experiments. This is primarily due to the lack of exhaustivity – only a limited number of fixed point formats were tested. It was not even possible to truly test all of the desired formats, as the fixed point library used did not allow full control over the number of integer bits used during computation. In addition, some of the results suffered from avoidable software bugs, limiting even further the amount of information these experiments provide. These experiments would greatly benefit from being re-run exhaustively with the bugs fixed and with a strict fixed point mode allowing for precise control over the number of integer bits.

From the information we do have, it does not appear to be possible to perform inference of PVANet using only fixed point numbers. Most notably, the BatchNorm layer was confirmed to be incapable of supporting fixed point computation. However, some layers do appear capable of being computed using smaller fixed point numbers. For example, the Eltwise layer can be computing with as little as 4 bits – with some careful coding, it would be possible to fit two such computations within an 8-bit data type.

Conclusion

In this work, we have presented two useful applications – a high performance fixed point numerical library and a highly flexible numerical inference engine.

The custom inference engine presented in Chapter 7 offers an unparalleled level of numerical flexibility to neural network inference. No other artificial neural network library offers the same level of control over the the numerical types used during inference in terms of either allowed numerical precisions or layer-by-layer control over the precision used. While today it primarily has research applications, in the future it could conceivably be used for applications such as the design of neural network-specific ASIC hardware.

The vectorized fixed point computational library presented in Chapter 6 offers a flexible fixed point type while providing a runtime performance that is generally equivalent or greater than similar libraries. As a result, it can be applied to any application requiring fixed point numbers and rapid runtime performance.

Difficulties The overarching theme connecting all of the contributions is the evolution of numerical precision in the face of an evolving landscape of hardware support. Some of the numerical precisions discussed are implemented in recent hardware (float16), while some others are planned for support in the near future (bfloat16). The remaining precisions (custom floats and fixed point), are still worth considering for future support.

As a result of the general lack available hardware support for the numerical precisions of interest, it is currently necessary to emulate these numerical formats via software emulation. However, the choice of how to emulate remains a difficult question – a number of software libraries exist for each purpose, each with its own advantages and drawbacks. In Chapter 5, we chose to use an existing software library to emulate the float16 format. However, Section 5.3.3 shows that even for a relatively simple use case, the disadvantages can be significant. In the remaining works presented, we ultimately opted for the most flexible solution – implementing each desired numerical format ourselves. For fixed point software libraries, it was the only path towards simultaneously having full control over the Qa.b format and accelerating computations via vectorization. For the experiments on neural network inference, it was the only path towards having a simple implementation with full control over the floating point format. In addition, our experiments on neural network inference required the development of a custom inference engine, as no existing solution offers easy extension of numerical support along with the desired level of flexibility in arithmetic used during inference.

Applications The float16 average computation presented in Chapter 5 can be applied to any application that computes the average of a large set of numbers. This commonly occurs in machine learning algorithms such as meanshift or k-means.

The fixed point extension to nsimd presented in Chapter 6 has a number of applications thanks to its combination of speed and flexibility. One major use case is the embedded domain, where the use of fixed point precision is common. This domain affects many important applications, such as telecommunications, aerospace, and medical applications. Another possible use-case is for prototyping computations using various fixed point formats prior to implementing them on an FPGA or while designing an ASIC. Especially when designing an ASIC with a single specific use case, this can allow a manufacturer to achieve extra efficiency, allowing for lower energy consumption. One more domain that can make use of this work is finance, in the cases where monetary amounts sometimes need to be exactly represented and runtime performance can be critical.

The custom inference engine presented in Chapter 7 has some possible current applications, along with some applications that will become worthwhile as the field of ANNs matures. Currently, the most interesting application of the custom inference engine is to experiment with precision reduction. The inference engine allows for easily testing various numerical formats in order to obtain the best balance of accuracy and speed. As the state of the art neural networks in the field of ANNs begin to stabilize, it may become worthwhile to develop processors adapted to specific neural networks. In order for these processors to be maximally efficient in both runtime and energy usage, we will want to use the minimum possible numerical precision that provides accurate results. Our custom inference engine can allow for hardware manufacturers to achieve this balance.

Overall, all of the works presented combine into a useful toolbox for experimenting with modified numerical precision within applications and for potentially developing new hardware that perfectly adapts to its use case.

Future Works All of the works presented here can be expanded as hardware manufacturers continue to add new functionalities, in terms of both supported numerical precisions and vectorization technologies. The float16 format is already implemented on some of the latest hardware, while the bfloat16 format instructions are currently being added to others. Some architectures already provide limited instructions for using fixed-point numbers. These capabilities may be expanded in the future, allowing for faster fixed-point computation. On the vectorization side, there is already limited support for the float16 format and incoming support for the bfloat16 format. In addition to the additional numerical formats, each generation of vectorization hardware adds new instructions and/or expands the sizes of the registers. An example of the former is the upcoming Intel AMX instructions and an example of the latter the latter is the ARM SVE and SVE2 technologies. Every single one of these developments is an opportunity to update the works performed here, examining the effects of using these new technologies.

The study on computing an average presented in Chapter 5 can primarily be expanded in two directions. The first direction of expansion is to study various other simple-seeming algorithms implemented using the float16 format. The second direction would be to study

the effects of computing an average using various other numerical types. In addition, these two approaches can be combined, allowing for a number of possible future works.

The fixed point extension to nsimd presented in Chapter 6 is far from complete – there are a number of ways to expand and improve it. One improvement that would aid the works presented in Chapter 7 is the addition of a strict mode that clears the reserved and unused bits after every single operation. Another possible improvement is the addition of an unsigned fixed point type. If a user is within one bit of changing underlying storage types, this could make a significant difference. In addition, an unsigned fixed point type would exhibit well defined overflow. One final expansion could be the addition of a C API for easier integration into embedded applications.

Another important avenue for improvement of the fixed point extension to nsimd is the development of new algorithms and the optimization of the existing ones. For all functions presented, very few numerical approximation techniques were actually used. More techniques should be studied and, if relevant, applied in order to achieve gains in either performance or precision. For all functions using Taylor series approximations, we should apply SFINAE in order to modulate the depth of the approximation according to the available precision. Finally, the addition of new functions would increase the usability of the fixed point extension, as the number of supported functions is somewhat limited.

The custom neural network inference engine presented in Chapter 7, along with the experiments it enabled, can be expanded and improved in a few ways.

The inference engine itself can be improved in a few ways. Most importantly, the software bugs discovered when the experiments were performed must be fixed. Work is currently being done in order to enable quantization of layers. Without quantization, the trained network weights are simply reinterpreted prior to computation. With quantization, layers using various numerical precisions can be adapted to better fit into the numerical types being used.

The experiments can be expanded first by re-running the fixed point portions affected by software bugs. If possible, the fixed point experiments would also benefit from the same level of exhaustiveness as the floating point experiments. However, this would require a large amount of computational resources. It would also be interesting to perform a similar study of other state of the art neural networks. For example, the YOLO neural network is much more lightweight than PVANet, and could be more easily studied exhaustively.

A - Compiler Outputs

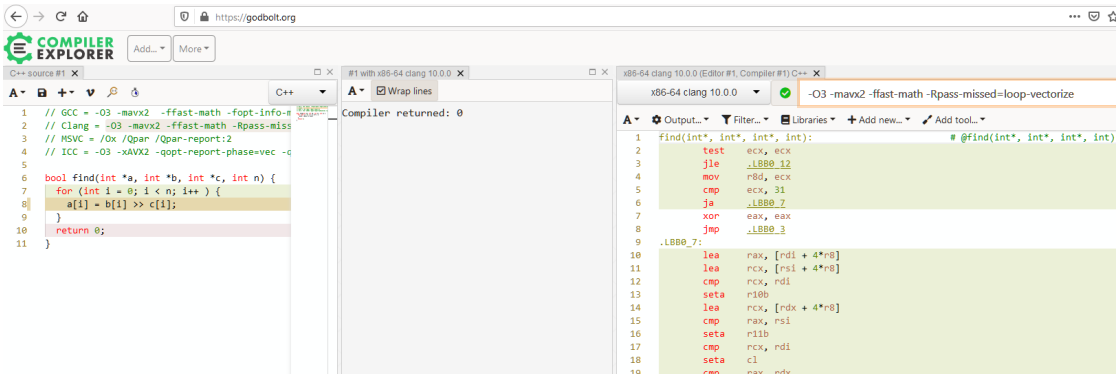

Figure A.1: Clang succeeding to vectorize Algo. 8.

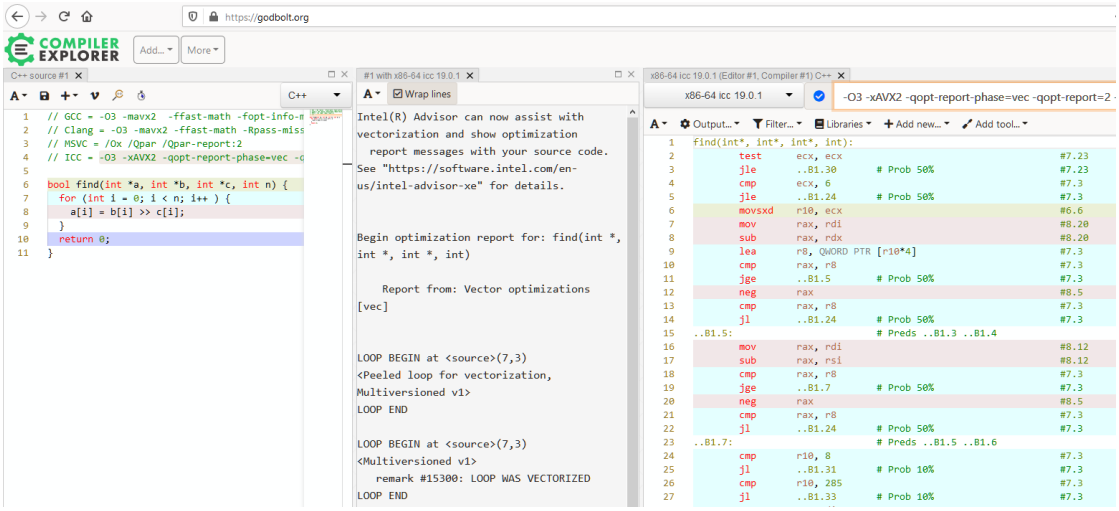

Figure A.2: icc succeeding to vectorize Algo. 8.

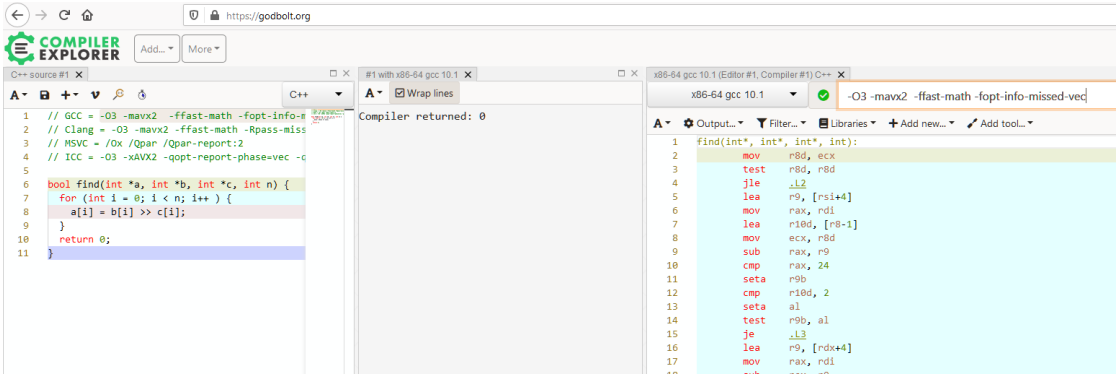

Figure A.3: gcc succeeding to vectorize Algo. 8.

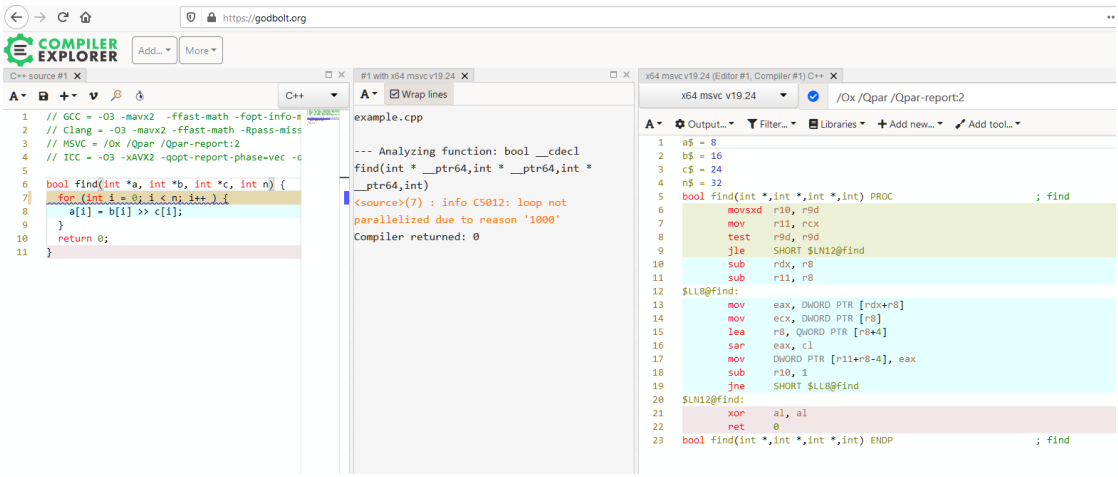

Figure A.4: MSVC failing to vectorize Algo. 8.

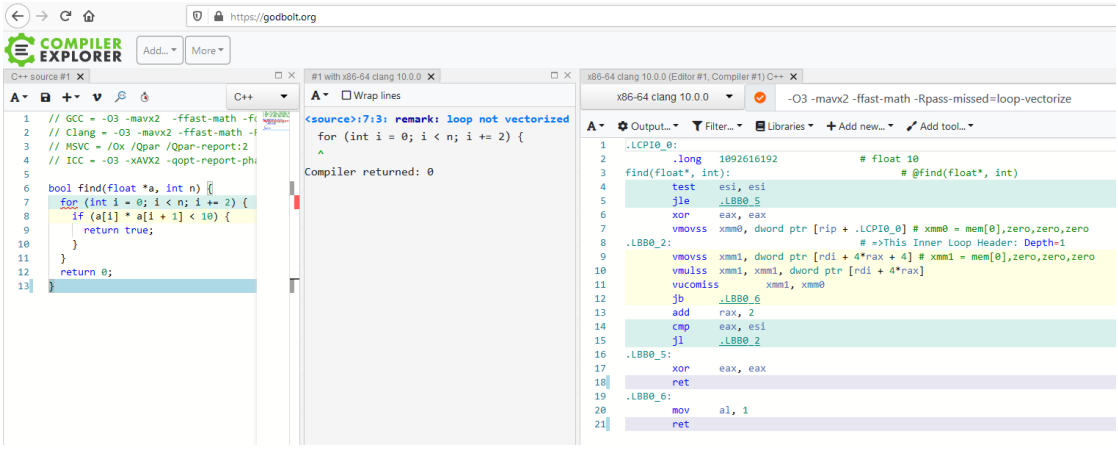

Figure A.5: Clang failing to vectorize Algo. 9.

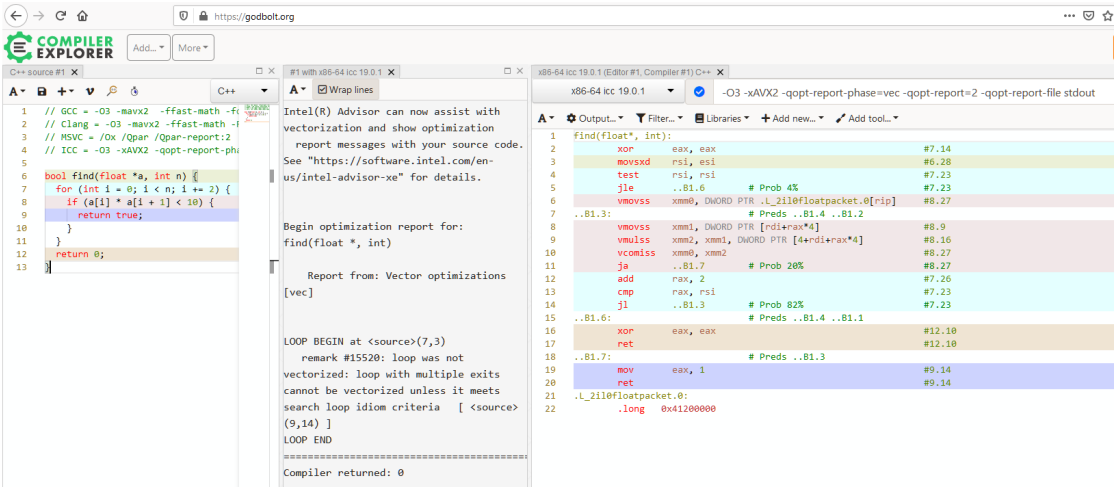

Figure A.6: icc failing to vectorize Algo. 9.

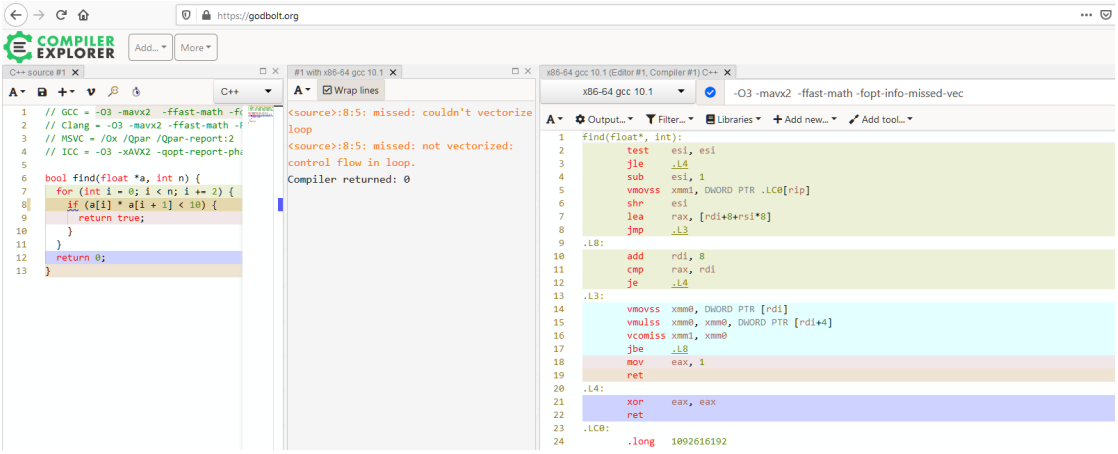

Figure A.7: gcc failing to vectorize Algo. 9.

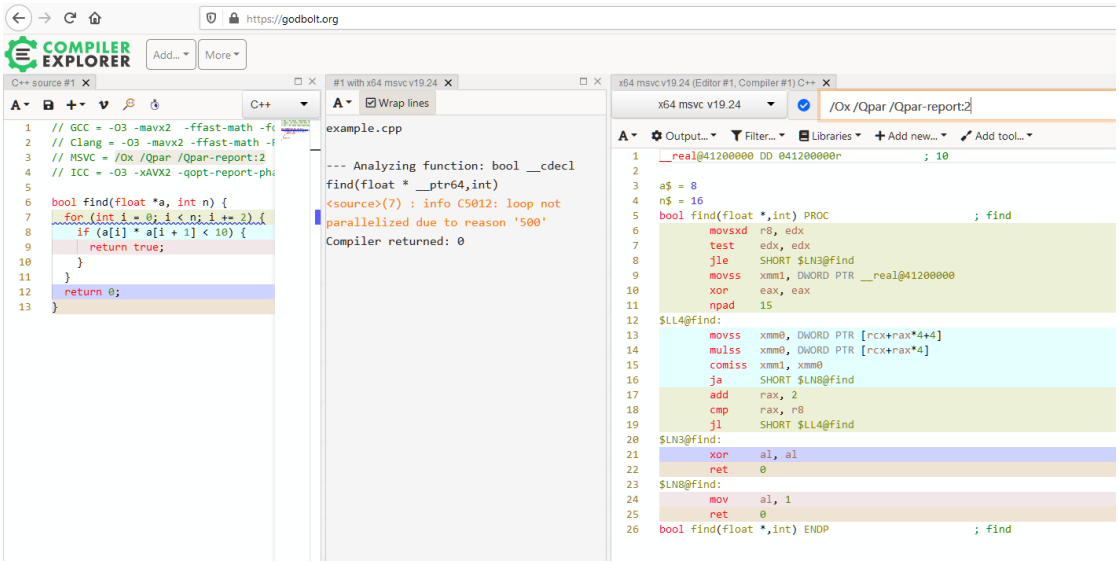

Figure A.8: MSVC failing to vectorize Algo. 9.

B - Installing nsimd

B.1 . Installation

Dependencies Python 3 is required to generate platform-specific code. CMake and a C++14 compiler are required to compile the library.

Optional Dependencies `ninja` is recommended over `make` for faster compilation. clang-format allows the python code to format the generated code. MIPP and Sleef, and Google Benchmark must be installed if performing benchmarks. MPFR and the Google Test library must be installed if performing unit tests.

Download In order to install nsimd, one must obtain the code via github.

```
git checkout https://github.com/agenium-scale/nsimd
```

Code Generation Next, the platform-specific code must be generated using the python script `egg/hatch.py`.

```
cd nsimd
python3 egg/hatch.py -A
```

The `-A` flag tells the script to "Generate code for all architectures, C and C++ APIs", as well as all tests and benchmarks. The `-h` flag provides a list of other flags that one may wish to use.

Compilation Once the code has been generated, we can compile the library.

```
mkdir build
cd build
cmake ..
make
```

This compiles nsimd's scalar functions. In order to compile for a SIMD architecture, the cmake command must include a `-DSIMD=<simd>` option, where `<simd>` is the target architecture (AVX2, SSE4_2, AARCH64, etc.).

B.2 . Organization

The root directory of `nsimd` is fairly straightforward. The functions of most of the directories are evident from their names.

- `benches` contains the benchmarks as defined by `egg/gen_benches.py`.
- `cmake` contains some cmake code to aid compilation.
- `doc` contains some basic documentation by default, plus the documentation generated by `egg/gen_doc.py`.
- `egg` contains the core of nsimd. It contains the python code that will generate all of the platform-specific functions, benchmarks, unit tests, code to be compiled, and some of the documentation.
- `include` contains the generated nsimd API and corresponding header-only code, as well as the fixed-point module.
- `scripts` contains a few auxiliary scripts.
- `src` contains the generated code that is used to compile all platform-specific libraries.
- `tests` contains the tests as defined by `egg/gen_tests.py`.

B.3 . Compilation

In order to compile C or C++ code using nsimd, it is necessary to provide certain compiler flags for the following information:

- path to includes (`-I/path/to/nsimd/include`)
- path to compiled library (`-L/path/to/nsimd/build`)
- link with compiled library (`-lnsimd_<arch>` - for example, `-lnsimd_avx2`)
- Activate correct SIMD instructions in compiler (`-m<architecture>` - for example, `-mavx2` or `-march=native`)
- Activate correct SIMD instructions in nsimd (`-DNSIMD_<arch>` - for example, `-DNSIMD_AVX2`)

C - Raw Benchmark Data

This annex provides the raw benchmark data (all times in nanoseconds) of all benchmarks performed in Chapter 6.

C.1 . Functions Added to nsimd

Type	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
i8	455	803	597	161	474	87.2
i16	497	1305	809	108	409	167
i32	643	862	857	52.1	328	327
i64	1605	1042	2142	103	1289	2403
u8	455	693	597	161	474	87.2
u16	497	1200	651	73.2	409	167
u32	643	863	857	52.0	328	327
u64	1605	1089	2142	103	1289	2658

Table C.1: Raw performance data in ns for the `clz` function. Used for Figure 6.4.

Type	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
i8	421	1681	2184	73.6	1262	826
i16	339	2169	1937	36.9	1200	1134
i32	272	103	1727	65.1	817	1906
i64	1459	4044	1940	129	1894	2750
u8	296	1181	707	73.6	729	706
u16	229	1214	759	25.6	569	809
u32	184	109	794	65.0	435	1522
u64	345	1122	918	129	1012	2549

Table C.2: Raw performance data in ns for the `shrv` function. Used for Figure 6.6.

Type	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
i8	296	1181	813	73.6	675	114
i16	228	1225	759	36.6	575	221
i32	184	103	798	65.1	329	435
i64	338	164	887	306	1283	912
u8	296	1181	707	73.6	675	114
u16	229	1213	759	25.6	568	221
u32	185	108	791	65.0	332	435
u64	339	164	916	129	1268	865

Table C.3: Raw performance data in ns for the `shlv` function. Used for Figure 6.5.

Type	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
i8	2595	1079	3221	748	2084	3985
i16	2586	2280	3329	1505	2118	15240
i32	2577	14661	3215	7304	2118	59973
i64	9395	61824	11599	34213	2126	245495
u8	2591	833	2902	623	2102	3664
u16	3149	2032	2902	1274	2120	14661
u32	3220	14771	2794	7412	2120	58789
u64	8987	58124	10683	33913	2123	238378

Table C.4: Raw performance data in ns for the `div` function. Used for Figure 6.7.

C.2 . Fixed Point Speedup

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	42.1	40.2	20.4	20.5	22.4	10.6	90.5	96.8
16	86.8	88.1	40.6	43.8	70.3	26.2	170	186
32	175	180	84.1	83.8	182	65.4	331	364
64	347	362	165	164	363	129	706	748

Table C.5: Raw performance data in ns for the fixed point `add` function. Used for Figure 6.10.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	204	201	148	101	98.3	46.9	199	142
16	129	101	74.6	51.9	98.2	33.0	388	275
32	354	254	180	129	239	78.0	771	549
64	729	959	728	1269	557	189	1503	3677

Table C.6: Raw performance data in ns for the fixed point `mul` function. Used for Figure 6.12.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	3214	2992	3007	1173	3219	763	2107	1806
16	3219	3259	3215	2292	3219	1506	2111	2009
32	2894	2736	2902	2705	3219	3509	2126	2107
64	9408	9873	9443	8709	11603	12071	2167	3307

Table C.7: Raw performance data in ns for the fixed point `div` function. Used for Figure 6.15.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	2893	8526	2893	4519	3218	2142	2104	4465
16	2892	8587	2897	4344	3218	998	1813	8922
32	2895	11242	2898	5560	3131	3435	1711	17836
64	9343	53860	9354	35899	11364	10453	1708	117657

Table C.8: Raw performance data in ns for the fixed point `rec` function. Used for Figure 6.24.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	37207	36549	37491	11971	57902	9496	28827	26165
16	50491	43317	50165	25568	77204	22568	34913	32300
32	44219	43144	44549	35911	72192	41605	28300	44609
64	125762	100483	125857	91570	176986	124174	28291	68960

Table C.9: Raw performance data in ns for the fixed point `sqrt` function. Used for Figure 6.26.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	3176	1933	2182	2017	8833	1488	22905	2203
16	4870	2688	2904	2857	2237	2120	29188	2886
32	8805	2090	4186	3109	10018	4112	21641	5202
64	7863	4288	7538	20866	8656	9837	29592	11163

Table C.10: Raw performance data in ns for the fixed point `sin` function. Used for Figure 6.18.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	3261	3447	2142	1927	8272	1429	21051	3353
16	4799	4378	2887	2809	2304	2066	32175	7129
32	8935	5406	4364	3414	12086	4482	23740	14745
64	10149	27816	10186	22401	10744	10324	32680	62015

Table C.11: Raw performance data in ns for the fixed point `cos` function. Used for Figure 6.20.

Bits	sse4_2	sse4_2(SIMD)	avx2	avx2(SIMD)	avx512	avx512(SIMD)	aarch64	aarch64(SIMD)
8	17581	7513	17455	3524	15763	2482	29861	5700
16	33022	8992	32813	6080	29524	4818	47837	10495
32	18716	12630	18859	8298	22777	8799	39780	18050
64	30511	51321	34642	37708	33160	23729	50157	106091

Table C.12: Raw performance data in ns for the fixed point `tan` function. Used for Figure 6.22.

Library	i8	i16	i32	i64
native	20.4	41.8	84.0	165
fp_t	20.4	40.6	84.1	165
pack	20.5	43.8	83.8	164
liquid		40.7	84.3	
fixmath			84.2	
CNL	20.4	42.5	86.5	164

Table C.13: Raw performance data in ns of various libraries for the fixed point `add` function. Used for Figure 6.10.

Library	i8	i16	i32	i64
native	60.7	42.5	98.2	487
fp_t	148	74.6	180	728
pack	101	51.9	129	1269
liquid		2256	2256	
fixmath			3512	
CNL	220	225	184	1109

Table C.14: Raw performance data in ns of various libraries for the fixed point `mul` function. Used for Figure 6.12.

Library	i8	i16	i32	i64
native	2897	2896	2894	9361
fp_t	3007	3215	2902	9443
pack	1173	2292	2705	8709
liquid		3869	10439	
fixmath			23248	
CNL	2904	2897	2900	9343

Table C.15: Raw performance data in ns of various libraries for the fixed point `div` function. Used for Figure 6.15.

Library	i8	i16	i32	i64
fp_t	2893	2897	2898	9354
pack	4519	4344	5560	35899
liquid		26193	32721	
fixmath			11271	
CNL	2893	2893	2898	9334

Table C.16: Raw performance data in ns of various libraries for the fixed point `rec` function. Used for Figure 6.24.

Library	i8	i16	i32	i64
fp_t	37491	50165	44549	125857
pack	11971	25568	35911	91570
liquid		36334	77885	
fixmath			97371	
CNL	6046	43550	71371	206850

Table C.17: Raw performance data in ns of various libraries for the fixed point `sqrt` function. Used for Figure 6.26.

Library	i8	i16	i32	i64
fp_t	2182	2904	4186	7538
pack	2017	2857	3109	20866
liquid		47464	86991	
fixmath			10002	
CNL	10626	11697	12081	86155

Table C.18: Raw performance data in ns of various libraries for the fixed point `sin` function. Used for Figure 6.18.

Library	i8	i16	i32	i64
fp_t	2142	2887	4364	10186
pack	1927	2809	3414	22401
liquid		47404	87466	
fixmath			10227	
CNL	9829	11473	11213	86334

Table C.19: Raw performance data in ns of various libraries for the fixed point `cos` function. Used for Figure 6.20.

Library	i8	i16	i32	i64
fp_t	17455	32813	18859	34642
pack	3524	6080	8298	37708
liquid		49870	97231	
fixmath			45048	

Table C.20: Raw performance data in ns of various libraries for the fixed point `tan` function. Used for Figure 6.22.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, 265–283. isbn: 9781931971331.

[2] Hande Alemdar, Nicholas Caldwell, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. "Ternary Neural Networks for Resource-Efficient AI Applications". In: *CoRR* abs/1609.00222 (2016). arXiv: 1609.00222. url: http://arxiv.org/abs/1609.00222.

[3] S Amat, S Busquier, and J.M Gutiérrez. "Geometric constructions of iterative functions to solve nonlinear equations". In: *Journal of Computational and Applied Mathematics* 157.1 (2003), pp. 197–205. issn: 0377-0427. doi: https://doi.org/10.1016/S0377-0427(03)00420-5. url: https://www.sciencedirect.com/science/article/pii/S0377042703004205.

[4] "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Trans. Math. Softw.* 28.2 (June 2002), 135–151. issn: 0098-3500. doi: 10.1145/567806.567807. url: https://doi.org/10.1145/567806.567807.

[5] ARM. "Arm C Language Extensions for SVE". In: (2019), pp. 1–368.

[6] ARM. *NEON Intrinsics Reference*. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics. Accessed 2020-05-18. 2020.

[7] *ARM Architecture Reference Manual Supplement: The Scalable Vector Extension(SVE), for ARMv8-A*. Revision ID081717, Accessed 2017-09-15. ARM. 2017.

[8] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. "A Comparison of Sorting Algorithms for the Connection Machine CM-2". In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '91. Hilton Head, South Carolina, USA: Association for Computing Machinery, 1991, 3–16. isbn: 0897914384. doi: 10.1145/113379.113380. url: https://doi.org/10.1145/113379.113380.

[9] Boost. *Boost Multiprecision*. https://github.com/boostorg/multiprecision. 2011.

[10] Steven Borowiec. *AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol*. https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol. 2016.

[11] *bSIMD: Introduction*. https://developer.numscale.com/bsimd. Accessed 2017-09-15. numscale, 2017.

[12] Caffe. *Caffe2 and PyTorch join forces to create a Research + Production platform PyTorch 1.0*. https://caffe2.ai/blog/2018/05/02/Caffe2_PyTorch_1_0.html. 2018.

[13] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. "MIPP: A portable C++ SIMD wrapper and its use for error correction coding in 5G standard". In: *WPMVP 2018 - Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing, Co-located with PPoPP 2018* 2018-March.February (2018). doi: 10.1145/3178433.3178435.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: (2015), pp. 1–6. arXiv: 1512.01274. url: http://arxiv.org/abs/1512.01274.

[15] S. Chevillard, M. Joldeş, and C. Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *Mathematical Software - ICMS 2010*. Ed. by K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama. Vol. 6327. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, 2010, pp. 28–31.

[16] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. "Natural Language Processing (Almost) from Scratch". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), 2493–2537. issn: 1532-4435.

[17] Matthieu Courbariaux and Yoshua Bengio. "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. url: http://arxiv.org/abs/1602.02830.

[18] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), 46–55. issn: 1070-9924. doi: 10.1109/99.660313. url: https://doi.org/10.1109/99.660313.

[19] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. "A set of level 3 basic linear algebra subprograms". In: *ACM Transactions on Mathematical Software (TOMS)* 16.1 (1990), pp. 1–17. issn: 15577295. doi: 10.1145/77626.79170.

[20] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. "Vectorization for SIMD Architectures with Alignment Constraints". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI '04. Washington DC, USA: Association for Computing Machinery, 2004, 82–93. isbn: 1581138075. doi: 10.1145/996841.996853. url: https://doi.org/10.1145/996841.996853.

[21] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. "Vectorization for SIMD Architectures with Alignment Constraints". In: *SIGPLAN Not.* 39.6 (June 2004), 82–93. issn: 0362-1340. doi: 10.1145/996893.996853. url: https://doi.org/10.1145/996893.996853.

[22] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. "Boost.simd: generic programming for portable simdization". In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 1–8.

[23] D. Etiemble and L. Lacassagne. "16-bit FP sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing". In: *International Conference on Parallel Processing, 2004. ICPP 2004.* 2004, 540–547 vol.1. doi: 10.1109/ICPP.2004.1327964.

[24] Daniel Etiemble, Samir Bouaziz, and Lionel Lacassagne. "Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing". In: *Proceedings of the 2005 3rd Workshop on Embedded Systems for Real-Time Multimedia* 2005 (2005), pp. 61–66. doi: 10.1109/ESTMED.2005.1518073.

[25] Daniel Etiemble and Lionel Lacassagne. "16-bit FP sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing". In: *Proceedings of the International Conference on Parallel Processing* (2004), pp. 540–547. issn: 01903918. doi: 10.1109/ICPP.2004.1327964.

[26] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[27] Michael Feldman. *Intel Lays Out New Roadmap for AI Portfolio*. https://www.top500.org/news/intel-lays-out-new-roadmap-for-ai-portfolio/. 2018.

[28] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960.

[29] Agner Fog. "Instruction tables". In: *Software Optimization Resources*. Vol. 4. 2017, pp. 199–214.

[30] Agner Fog. “VCL: C++ vector class library”. In: (2016), pp. 1–110. url: http://www.agner.org/optimize/{\#}vectorclass.

[31] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. “MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding”. In: *ACM Trans. Math. Softw.* 33.2 (June 2007), 13–es. issn: 0098-3500. doi: 10.1145/1236463.1236468. url: https://doi.org/10.1145/1236463.1236468.

[32] M. Frontini and E. Sormani. “Modified Newton’s method with third-order convergence and multiple roots”. In: *Journal of Computational and Applied Mathematics* 156.2 (2003), pp. 345–354. issn: 0377-0427. doi: https://doi.org/10.1016/S0377-0427(02)00920-2. url: https://www.sciencedirect.com/science/article/pii/S0377042702009202.

[33] Joseph D. Gaeddert. *liquid-fpm : Software-Defined Radio Fixed-Point Math Library*. https://github.com/jgaeddert/liquid-fpm.

[34] Giu.natale. *Example of a Roofline model*. 2016. url: https://commons.wikimedia.org/wiki/File:Example_of_a_Roofline_model.svg.

[35] Matt Godbolt. *Compiler Explorer*. https://godbolt.org/. 2020.

[36] David Goldberg. “What Every Computer Scientist Floating-Point Arithmetic Should Know About Floating-point Arithmetic”. In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[37] Google. *benchmark*. https://github.com/google/benchmark.

[38] Google. *libfixmath*. https://github.com/PetteriAimonen/libfixmath.

[39] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[40] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”. In: (2016). eprint: 1510.00149. url: https://arxiv.org/abs/1510.00149.

[41] Mark Harris. *New Features in CUDA 7.5*. https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/. Accessed 2017-08-10. 2015.

[42] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. “Leveraging the bfloat16 Artificial Intelligence Datatype for Higher-Precision Computations”. In: *Proceedings - Symposium on Computer Arithmetic* 2019-June (2019), pp. 69–76. doi: 10.1109/ARITH.2019.00019. arXiv: 1904.06376.

[43] Warren Henry. *Hacker’s Delight*. 2007, pp. 97–101. isbn: 9780321842688.

[44] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. url: http://arxiv.org/abs/1207.0580.

[45] Sanghoon Hong. *PVANet*. https://github.com/sanghoon/pva-faster-rcnn.

[46] Sanghoon Hong, Byung-Seok Roh, Kye-Hyeon Kim, Yeongjae Cheon, and Minje Park. "PVANet: Lightweight Deep Neural Networks for Real-time Object Detection". In: *CoRR* abs/1611.08588 (2016). arXiv: 1611.08588. url: http://arxiv.org/abs/1611.08588.

[47] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70. doi: 10.1109/IEEESTD.2008.4610935.

[48] Intel. "Intel 64 and IA-32 Architectures Optimization Reference Manual". In: *Intel Technology Journal* 09.03 (2005), pp. 1–660. issn: 15222594. doi: 10.1535/itj.0903.05.

[49] Intel. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 2020-05-18. 2020.

[50] Intel. *Math Kernel Library*. https://software.intel.com/en-us/.

[51] Intel. *Performance Benefits of Half Precision Floats*. https://software.intel.com/content/www/us/en/develop/articles/performance-benefits-of-half-precision-floats.html. 2012.

[52] Intel Corporation. "BFLOAT16-Hardware Numerics Definition". In: November (2018), p. 7. url: www.intel.com/design/literature.htm..

[53] 3Dfx Interactive. *Glide 2.2 Programming Guide*. San Jose, CA 94134, 1997.

[54] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[55] M. H. Ionica and D. Gregg. "The Movidius Myriad Architecture's Potential for Scientific Computing". In: *IEEE Micro* 35.1 (2015), pp. 6–14.

[56] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[57] N. Jouppi, C. Young, N. Patil, and D. Patterson. "Motivation for and Evaluation of the First Tensor Processing Unit". In: *IEEE Micro* 38.3 (2018), pp. 10–19.

[58] Sylvain Jubertie, Ian Masliah, and Joel Falcou. “Data layout and SIMD abstraction layers: Decoupling interfaces from implementations”. In: *Proceedings - 2018 International Conference on High Performance Computing and Simulation, HPCS 2018* (2018), pp. 531–538. doi: 10.1109/HPCS.2018.00089.

[59] William Kahan. “Further remarks on reducing truncation errors”. In: *Communications of the ACM* 8.1 (1965), pp. 40–41.

[60] Povilas Kanapickas. *libsimdpp*. https://github.com/p12tic/libsimdpp. 2017.

[61] Keil. *Half-precision floating-point intrinsics*. http://www.keil.com/support/man/docs/armclang_ref/armclang_ref_wtr1519643241579.htm. 2012.

[62] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. isbn: 0201896834.

[63] Matthias Kretz and Volker Lindenstruth. “Vc: A C++ library for explicit vectorization”. In: *Software: Practice and Experience* 42 (Nov. 2012). doi: 10.1002/spe.1149.

[64] Jim Kukunas. *Power and Performance: Software Analysis and Optimization*. 2015, pp. 1–284. isbn: 9780128008140. doi: 10.1016/C2013-0-18946-5.

[65] L. Lacassagne, D. Etiemble, and S. A. Ould Kablia. “16-Bit Floating Point Instructions for Embedded Multimedia Applications”. In: *Proceedings - International Workshop on Computer Architecture for Machine Perception, CAMP* (2005), pp. 198–203. doi: 10.1109/camp.2005.1.

[66] Maximilian Lam, Zachary Yedidia, Colby Banbury, and Vijay Janapa Reddi. *Quantized Neural Network Inference with Precision Batching*. 2020. arXiv: 2003.00822 [cs.LG].

[67] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. “Cholesky Factorization on SIMD Multi-Core Architectures”. In: *J. Syst. Archit.* 79.C (Sept. 2017), 1–15. issn: 1383-7621. doi: 10.1016/j.sysarc.2017.06.005. url: https://doi.org/10.1016/j.sysarc.2017.06.005.

[68] Daniel Lemire and Leonid Boytsov. “Decoding billions of integers per second through vectorization”. In: *CoRR* abs/1209.2137 (2012). arXiv: 1209.2137. url: http://arxiv.org/abs/1209.2137.

[69] Fengfu Li and Bin Liu. “Ternary Weight Networks”. In: *CoRR* abs/1605.04711 (2016). arXiv: 1605.04711. url: http://arxiv.org/abs/1605.04711.

[70] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. “Fixed Point Quantization of Deep Convolutional Networks”. In: *CoRR* abs/1511.06393 (2015). arXiv: 1511.06393. url: http://arxiv.org/abs/1511.06393.

[71] John McFarlane. *Compositional Numeric Library*. https://github.com/johnmcfarlane/cnl.

[72] Ralf Möller. "Design of a low-level C++ template SIMD library". In: (2016). url: www.ti.uni-bielefeld.de/html/people/moeller/tsimd{_}warpingsimd.html.

[73] Barry Lee Mowday. "PL/STAR, a structured assembly language for the CDC STAR-100". MA thesis. College of William & Mary - Arts & Sciences, 1979, p. 86.

[74] Pritam Mukherjee et al. "A shallow convolutional neural network predicts prognosis of lung cancer patients in multi-institutional computed tomography image datasets". In: *Nature Machine Intelligence* 2.5 (2020), pp. 274–282. issn: 2522-5839. doi: 10.1038/s42256-020-0173-6. url: https://doi.org/10.1038/s42256-020-0173-6.

[75] Jean-michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-pierre Jeannerod, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. 2009. isbn: 9780817647049.

[76] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), 89–100. issn: 0362-1340. doi: 10.1145/1273442.1250746. url: https://doi.org/10.1145/1273442.1250746.

[77] Lars Tyge Nielsen. "Understanding N(d1) and N(d2): Risk-Adjusted Probabilities in the Black-Scholes Model". In: *Finance* 14.October (1993), pp. 95 –106. issn: 07526180.

[78] NVIDIA. *CUDA 11 Features Revealed*. https://developer.nvidia.com/blog/cuda-11-features-revealed/. 2020.

[79] NVIDIA. "NVIDIA Tesla P100". In: *White Paper* (2018).

[80] NVIDIA. "NVIDIA Turing GPU". In: *White Paper* (2018).

[81] NVIDIA Corporation. *Cg Language Toolkit*. 2003.

[82] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. url: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[83] K. Peou, A. Kelly, J. Falcou, and C. Germain. “A Case Study on Optimizing Accurate Half Precision Average”. In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2018, pp. 356–363.

[84] Andrea Petreto, Arthur Hennequin, Thomas Koehler, Thomas Romera, Yohan Fargeix, Boris Gaillard, Manuel Bouyer, Quentin L. Meunier, and Lionel Lacassagne. “Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU Architectures”. In: *Conference on Design and Architectures for Signal and Image Processing, DASIP* 2018-October (2018), pp. 25–30. issn: 21649766. doi: 10.1109/DASIP.2018.8597004.

[85] Stéphane Piskorski, Lionel Lacassagne, Samir Bouaziz, and Daniel Etiemble. “Customizing CPU instructions for embedded vision systems”. In: *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors* (2007), pp. 59–64. issn: 10636862. doi: 10.1109/ASAP.2007.4429961.

[86] Quantstack. *xsimd*. https://github.com/xtensor-stack/xsimd. 2018.

[87] Christian Rau. *half: Half-precision floating point library*. http://half.sourceforge.net/. Accessed 2017-09-15. 2017.

[88] E. Y. Remez. “Sur la détermination des polynômes d’approximation de degré donnée”. In: *Comm. Soc. Math. Kharkov* 10 (1934), pp. 41–63.

[89] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65.6 (1958), pp. 386–408. issn: 0033295X. doi: 10.1037/h0042519.

[90] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 696–699. isbn: 0262010976.

[91] Agenium Scale. *nsimd*. https://github.com/agenium-scale/nsimd. 2019.

[92] Thomas J. Scott. “Mathematics and Computer Science at Odds over Real Numbers”. In: *Proceedings of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’91. San Antonio, Texas, USA: Association for Computing Machinery, 1991, 130–139. isbn: 0897913779. doi: 10.1145/107004.107029. url: https://doi.org/10.1145/107004.107029.

[93] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. 2003, p. 642. isbn: 0071230076.

[94] N. Shirazi, A. Walters, and P. Athanas. “Quantitative analysis of floating point arithmetic on FPGA based custom computing machines”. In: *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. 1995, pp. 155–162. doi: 10.1109/FPGA.1995.477421.

[95] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. url: http://arxiv.org/abs/1712.01815.

[96] Sergi Siso, Wes Armour, and Jeyarajan Thiyagalingam. “Evaluating auto-vectorizing compilers through objective withdrawal of useful information”. In: *ACM Transactions on Architecture and Code Optimization* 16.4 (2019). issn: 15443973. doi: 10.1145/3356842.

[97] Roman A. Solovyev, Alexandr A. Kalinin, Alexander G. Kustov, Dmitry V. Telpukhov, and Vladimir S. Ruhlov. “FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations”. In: *CoRR* abs/1808.09945 (2018). arXiv: 1808.09945. url: http://arxiv.org/abs/1808.09945.

[98] Nigel Stephens. *BFloat16 processing for Neural Networks on Armv8-A*. https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/bfloat16-processing-for-neural-networks-on-armv8_2d00_a. 2019.

[99] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. “A Transprecision Floating-Point Platform for Ultra-Low Power Computing”. In: *CoRR* 1711.10374 (2017). arXiv: 1711.10374. url: http://arxiv.org/abs/1711.10374.

[100] Sid-Ahmed-Ali Touati, J. Worms, and Sébastien Briais. “The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation”. In: *Concurrency and Computation: Practice and Experience* 25 (July 2013). doi: 10.1002/cpe.2939.

[101] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.August (2019). issn: 1476-4687. doi: 10.1038/s41586-019-1724-z. url: http://dx.doi.org/10.1038/s41586-019-1724-z.

[102] Dorothy Wedel. “Fortran for the Texas Instruments ASC System”. In: *SIGPLAN Not.* 10.3 (Jan. 1975), 119–132. issn: 0362-1340. doi: 10.1145/390015.808411. url: https://doi.org/10.1145/390015.808411.

[103] S. Weerakoon and T.G.I. Fernando. “A variant of Newton’s method with accelerated third-order convergence”. In: *Applied Mathematics Letters* 13.8 (2000), pp. 87–93. issn: 0893-9659. doi: https://doi.org/10.1016/S0893-9659(00)00100-2. url: https://www.sciencedirect.com/science/article/pii/S0893965900001002.

[104] Paul J. Werbos. *Applications of advances in nonlinear sensitivity analysis*. 2005. doi: 10.1007/bfb0006203.

[105] R. Whaley, Antoine Petitet, and Jack Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27 (Nov. 2000).

[106] Wikipedia. *Find First Set*. https://en.wikipedia.org/wiki/Find_first_set.

[107] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), 65–76. issn: 0001-0782. doi: 10.1145/1498765.1498785. url: https://doi.org/10.1145/1498765.1498785.

[108] Kent E. Wires and Michael J. Schulte. “Reciprocal and Reciprocal Square Root Units with Operand Modification and Multiplication”. In: *J. VLSI Signal Process. Syst.* 42.3 (Mar. 2006), 257–272. issn: 0922-5773. doi: 10.1007/s11265-006-4186-0. url: https://doi.org/10.1007/s11265-006-4186-0.

[109] Jack M Wolfe. “Reducing Truncation Errors by Programming”. In: *Communications of the ACM* 7.6 (1963), pp. 355–356.

[110] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. “Improving Neural Network Quantization without Retraining using Outlier Channel Splitting”. In: *CoRR* abs/1901.09504 (2019). arXiv: 1901.09504. url: http://arxiv.org/abs/1901.09504.

[111] Jeroen Van Der Zijp. *Fast Half Float Conversions*. 2010. url: http://www.fox-toolkit.org/ftp/fasthalffloatconversion.pdf.