JOHANNES KEPLER
UNIVERSITY LINZ

Author
Christoph Aigner

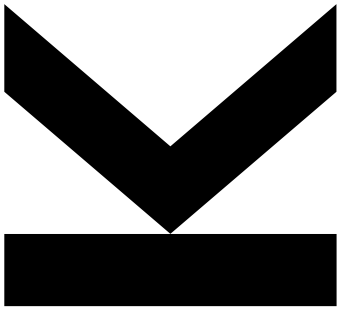
Submission
Institute for System Software

Thesis Supervisor
DI Lukas Makor

Assistant Thesis Supervisor
Dr. Gergö Barany (Oracle Labs)

October 2023

Improving Vectorization of Fold Loops in a Dynamic Compiler



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Parallelization is the next big step in the quest to further increase the performance of applications [1], but writing parallel code is a quite cumbersome task that might increase program complexity and development times significantly. Therefore, to utilize the parallelization capabilities of modern hardware despite non parallel code, the compiler can automatically transform linear code to a parallel representation in a process called (auto-)vectorization. A well suited target for vectorization are loops because sometimes multiple loop iterations can be executed in parallel resulting in a significant reduction in execution time.

Loops that accumulate a set of data into a single value are called fold loops. These loops are especially tricky to vectorize because the current loop iteration depends on the previous one. To still be able to extract parallelism, the calculation needs to be transformed using mathematical properties like associativity, commutativity and distributivity.

This work contributes to the Graal Compiler, a highly optimizing dynamic compiler that already contains an extensive loop vectorizer, to improve the already existing implementation for vectorizing fold loops that contain a multiplication and an addition, also called hash code like loops. The new implementation does not only yield a performance improvement for loops that were vectorized previously (see section 6.1), but also allows this optimization to be used not only on multiplications and additions, but also on other arithmetic operations, namely left-shifts, bit-wise XORs and bit-wise ORs. Additionally, this work expands the Graal Compiler's capabilities to vectorize fold loops that contain subtractions, which are neither associative nor commutative.

Kurzfassung

Der nächste Schritt im Bestreben nach immer schnelleren Ausführungszeiten von Programmen ist, mehrere Berechnungen nebenläufig auszuführen. Das Schreiben von solch parallelem Programmcode ist jedoch mit vielen Hindernissen wie zum Beispiel deutlich komplexerem Code verbunden, die die Fehlerrate und die Entwicklungszeit deutlich erhöhen. Um nun trotzdem von den Möglichkeiten der parallelen Datenverarbeitung zu profitieren, die moderne CPUs bieten, kann ein Compiler automatisch linearen Code in parallelen Code überführen. Dieser Prozess wird auch "automatische Vektorisierung" genannt. Schleifen eignen sich meist besonders gut für automatische Vektorisierung, da hier oft mehrere Durchläufe nebenläufig ausgeführt werden können.

Schleifen, ein einziges Ergebnis aus einer Menge von Daten errechnen, werden Reduktionsschleifen genannt. Solche Reduktionsschleifen sind besonders schwierig zu vektorisieren, da immer ein Schleifendurchlauf auf dem Ergebnis des Vorausgegangenen beruht. Für das Umformen der gegebenen Berechnung in eine parallelisierbare Form werden mathematische Eigenschaften wie Assoziativität, Kommutativität und Distributivität benötigt.

Diese Arbeit beschreibt einen Beitrag zum Graal Compiler, einem stark optimierenden dynamischen Compiler, der schon automatisch geeignete Schleifen vektorisiert. Die Vektorisierungsfunktion für Reduktionsschleifen mit einer Multiplikation und einer Addition (auch genannt Hash-Code ähnliche Schleifen) wurde im Zuge dieser Arbeit verbessert, um einerseits schnellere Ausführungszeiten für bereits vorher vektorisierte Schleifen zu erzielen (siehe Abschnitt 6.1) und andererseits diese Optimierung auch auf andere arithmetische Operationen wie Linksverschiebung, bitweises XOR und bitweises OR anwendbar zu machen. Zusätzlich wurde der Graal Compiler um die Fähigkeit erweitert, Reduktionsschleifen mit Subtraktionen, welche weder assoziativ noch kommutativ sind, zu vektorisieren.

Contents

1 Introduction

As the generational increase in straight line (non parallel) execution speed starts to decrease [1], alternative methods of increasing execution performance need to be leveraged. One of those methods is parallelism. In addition to conventional multi-threading / multi-processing, a simpler form of parallelism is data parallelism. Data parallel processing takes place within a single thread and just processes multiple elements of data in parallel to increase data throughput significantly.

With the advent of data parallel Single Instruction Multiple Data (SIMD) instructions in desktop computing facilitated by Intel's MMX Instruction Set Architecture (ISA) extension to its x86 architecture released in 1996 [2], new possibilities for software optimizations emerged. Today, the leading ISAs in desktop and high power computing provide ISA extensions for SIMD instructions.

Writing SIMD code manually is a very cumbersome task. Usually writing such code locks the developed software to a specific platform that supports the used instructions which therefore requires the developer to strike a balance between platform independence, development effort and performance. Therefore, most developers opt to write purely scalar code which leaves significant performance gains on the table. Automatically transforming the scalar code to SIMD code through compiler optimizations is a great way of leveraging SIMD instructions while maintaining platform independence on the source code level. This process is called automatic vectorization.

A Just-In-Time (JIT) compiler has the unique opportunity to take platform independent intermediate code and transform it to a highly optimized machine code representation specially tailored to the target machine. This allows the compiler to leverage instructions for optimizations that may not always be available but are available on the target machine. Additionally, old intermediate code may receive significant performance increases by executing it using newer JIT compilers which employ better optimizations than the ones available at the time of shipping the original piece of software.

A common target for vectorization are loops, because the same short code snippet is executed repeatedly. Therefore, data parallelisms may arise that can be processed with SIMD instructions. Using SIMD instructions, multiple iterations of the target loop may be executed in parallel. This thesis focuses on a special category of loops called *fold-loops* or *reduce-loops* which are loops that accumulate elements of a set of data (usually an array) into a single value. These loop shapes are especially hard to vectorize due to restrictions that arise when reshaping the calculation to extract data parallelism that are discussed in section 3.3.

This work contributes to the loop vectorizer of the Graal Compiler, a highly optimizing JIT compiler for Java Bytecode, that generates data parallel code for such loops. After an introduction to vectorization and fold-loops (section 2), the difficulties and restrictions when vectorizing fold-loops are presented in section 3. Following this, improvements made to the vectorization of fold loops containing 2 different operations are presented in section 4. Further, this thesis tackles the problem of vectorizing fold loops containing subtractions in section 5. Lastly, in section 6 the contributions are evaluated and benchmark results are presented.

2 Background

This chapter introduces some terms and concepts used in this thesis. It starts out presenting Java and GraalVM in section 2.1, followed by an introduction into parallelism in section 2.2 and finally introducing SIMD and vectorization in section 2.3.

2.1 GraalVM

Java is an object oriented, garbage collected programming language designed with platform independence in mind. It was originally developed by Sun Microsystems in 1995 [3] which later was acquired by Oracle. Java is designed to be compiled to a general intermediate representation called Java Bytecode (abbreviated to "bytecode" in the following text). To execute a given bytecode program, a run-time, often in the form of a virtual machine (VM), is needed. The Java Virtual Machine (JVM) executes the given bytecode by either interpreting the bytecode or by translating the byte to machine code at run time (also called Just-In-Time (JIT) compilation) to achieve significantly better performance.

2.1.1 A Java Virtual Machine

There is an abundance of high performance JVMs available for users to run Java programs. The most popular of them is the open source reference implementation by Oracle called Java HotSpot VM. GraalVM is a JVM developed by Oracle Labs based on HotSpot and like HotSpot, the GraalVM employs tiered compilation. This means, execution of the given bytecode starts in an interpreter. During interpretation profiling data about the program is collected. If a bit of code is executed often enough, the code is compiled with a quick JIT compiler called C1 or client compiler. Finally, especially hot spots of the program are compiled with a highly optimizing JIT compiler using the profiling data collected earlier which in HotSpot is called C2 or server compiler.

The GraalVM replaces the C2 JIT compiler with its own implementation called the Graal Compiler, which is written in Java. Additionally, GraalVM provides support for many different languages like JavaScript, Ruby, Python or R via the Truffle Language Implementation Framework [4].

2.1.2 Graal Intermediate Representation

Graal uses an intermediate representation (IR) that has the structure of a directed graph in static single assignment (SSA) form. It is a superposition of a control flow graph (CFG)

and a data flow graph (DFG) [5]. The CFG is modeled by nodes having successors while the DFG is modeled by nodes having inputs. To enable traversing the CFG or DFG in any direction, all nodes connected via successor edges or input edges are also linked in the reverse direction via usage edges and predecessor edges.

Listing 1: Example of simple if clause [5]

```
1 ...
2 if (cond) {
3     result = value1 + value2;
4 } else {
5     result = value2;
6 }
7 return result;
```

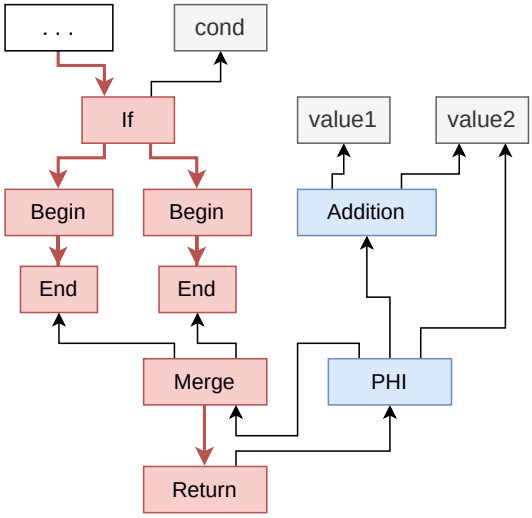


Figure 1: IR of the code from listing 1

To further illustrate the GraalIR, a short example taken from the paper "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler" [5] is presented in listing 1. Figure 1 contains the IR generated for the piece of code in listing 1. The successor edges that represent control flow are red downward edges while the black upward pointing edges are the input edges that denote value flow.

2.2 Parallelization

A common and very powerful strategy to increase application performance is to leverage the hardware's capability to process data simultaneously.

2.2.1 Free Lunch is Over

In the famous article "The Free Lunch Is Over" [1] which was released in 2005 the end of free performance gain (without needing to change the software, the so called "free performance lunch") by ever improving hardware is proclaimed. Straight-line execution speed starts to approach hard physical limits like the speed of light. As improvement of straight-line execution speed slows down, the logical next step to further increase

computing performance is to start processing data in parallel. To still deliver a substantial uplift in performance across generations of hardware, modern day chips provide increasing capabilities of parallelization.

The main issue with gaining performance via parallelization is, that it does come at the cost of significantly increased complexity of the software because in order to leverage these new capabilities, parallelism needs to be baked into the software. Therefore, to meet ever increasing requirement in computational capacity, a shift in how to write efficient and fast code is inevitable.

Additionally, parallelization is implemented in many different, sometimes unique, ways on different platforms and systems. To perfectly leverage the theoretical maximum performance, developers often times need to write platform dependent code. This leads to significantly increased development times because the same piece of software often needs to run on a wide variety of hardware and operating system configurations therefore requiring sometimes substantial parts of the software to be written for multiple configurations.

2.2.2 Task-Parallel Programming

The most well known form of parallelism in programs is task-parallel programming, where an application is separated into multiple tasks that run in parallel in separate threads or processes on multiple CPU cores. These tasks may share data but have independent execution flows. This logical separation needs to be done in software. Parallel programming comes with a large set of unique pitfalls to be considered by the developer, which significantly increases the complexity of the software.

Additionally, multi-threading / multi-processing comes with a considerable computational overhead when creating and managing multiple threads and processes. Finally, special algorithms that consist of a parallelizable computational pattern need to be used to be able to profit from concurrent programming.

2.2.3 Data Parallelism

Data Parallelism in contrast revolves around the idea of processing batches of data at once. Those sets of similar data are called vectors. Instead of iterating over the scalar elements of the vector and processing one at a time, modern time CPUs often offer instructions which can execute the same instruction on a set of values at once. These instructions are called Single Instruction Multiple Data (SIMD) instructions. Still, these

instructions do have their limitations as the number of elements that are processed in parallel cannot be any arbitrary number but is usually restricted to a small power of 2.

Data parallelism can be leveraged by Graal to prolong the availability of free performance lunch. It can be used to further improve single threaded performance. Generating data parallel code from scalar code is, while still a challenging task in its own right, much easier than automatically rewriting an application to run on multiple threads, because data parallelism does not have to consider concepts like locking of resources. Furthermore, adding data parallelism to the generated code does not come with the significant overhead in managing and scheduling multiple threads.

Additionally, because Java programs do not ship as binaries already compiled to machine code but rather only compiled to the platform independent intermediate language Java Bytecode, even older programs can profit from improvements in the Graal JIT compiler without the need to be changed or recompiled.

Data parallel code will be called vector code or vectorized code for the rest of this thesis.

2.3 Single Instruction Multiple Data

Data parallelism is commonly implemented in hardware as SIMD instructions. Nearly all modern day high power instruction set architectures (ISAs) like x86, ARM or IBM PowerPC define an extension for SIMD instructions.

2.3.1 Intel Advanced Vector Instructions

The ISA with the highest share in the high power computing market is the Intel x86 architecture. This ISA defines multiple different extension for SIMD instructions like MMX, 3DNow!, EMMX, multiple versions of Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). This introduction to vector instructions will focus on the latter two of those ISA extensions.

SSE defines 8 additional 128 bit wide vector registers called XMM0-XMM7. The 64bit extension to Intel x86 called AMD64 or x86-64 adds 8 additional vector registers XMM8-XMM15 to CPUs that also support SSE. AVX extends the width of the vector registers to 256 bits calling them YMM0-YMM15 while still only allowing floating point arithmetic. AVX2 includes support for integer instructions on these vector registers. The AVX512 extension, additionally to increasing the register width to 512 bit, again doubles the vector register count resulting in 32 vector registers ZMM0-ZMM31. The lower parts of these vector registers can still be addressed via XMM0-XMM31 or YMM0-YMM31.

Most current x86 based CPUs support AVX2 which offers instructions that interpret the content of vector registers as a packed set of smaller data types like single or double precision floating point numbers or integers of various sizes and executing the same operation on each recognized element. Using these instructions (which will be called vector instructions in this thesis) we can easily implement data parallelism as described before.

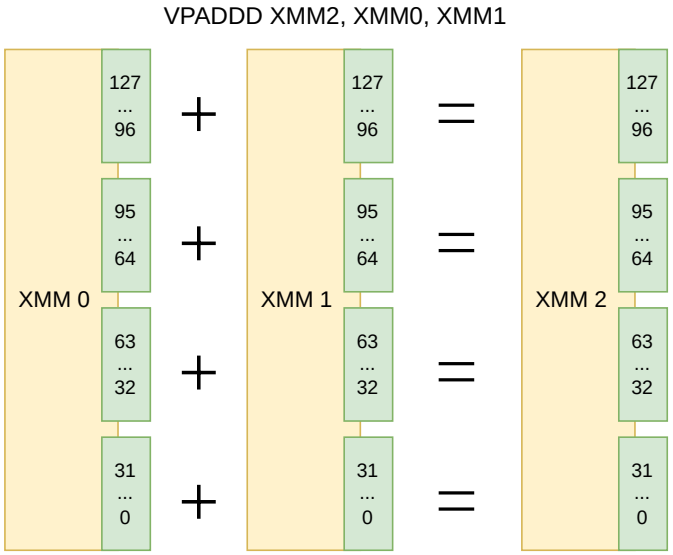


Figure 2: Add packed double-word integers from XMM0, XMM1 and store in XMM2

To further illustrate the functionality of SIMD instructions, figure 2 represents the AVX2 instruction `VPADDD XMM2, XMM0, XMM1` (Add packed double-word integers from XMM0, XMM1 and store in XMM2) [6]. Using this instruction, the 128 bit wide SIMD register XMM0 is interpreted as 4 32 bit wide integers that are added to the 4 32 bit wide integers in XMM1, Finally, the result is then stored in XMM2. The 4 independent additions in this calculation are done in parallel, therefore (depending on the micro architecture) taking similar time as a single scalar addition.

2.3.2 Automatic Vectorization

Automatic vectorization is a technique where an optimizing compiler automatically derives a vectorized representation from source code written using scalar operations. This process can yield significant performance gains [7]. Automatic vectorization can generally be separated into two different approaches, linear-code vectorization and loop vectorization.

2.3.2.1 Linear-Code Vectorization

The first one is linear-code vectorization where the compiler tries to find inherent data parallelisms in the source code and generates vector code for the given calculation. In combination with careful loop unrolling, this technique can be used to vectorize loops. For example GCC only relies on a linear-code superword-level parallelism (SLP) vectorizer which, in conjunction with loop unrolling, that is aware of possible later vectorization, can vectorize loops effectively [8]. LLVM and Graal feature separate linear-code and loop vectorizers [9].

Linear-code vectorization can not only be used to transform entire loops to a data parallel form but also to vectorize smaller suitable scalar calculations like for example matrix multiplications, or loads of 4 adjacent values from memory.

2.3.2.2 Loop Vectorization

Since loops that execute the same operation on a set of data, like an array, are a common target for vectorization [10], vectorizers that specifically handle the transformation of entire loops have been devised. The advantage of building a loop vectorizer over simply combining loop unrolling with a linear-code vectorizer is, that vectorization does not rely on loop unrolling to guess the appropriate number of iterations to unroll. Additionally, a loop vectorizer that takes the entire loop into consideration may be able to more easily extract parallelism that may require extensive transformation of the loop body, like mixed arithmetic folds (see section 4). Such loop shapes might be harder to detect if the loop has been unrolled prior.

If a loop is vectorized, multiple iterations of the given loop can be executed at once by conceptually splitting the given array into smaller parts that fit into vector registers and iterating over those batches using vector instructions instead of iterating over the scalar elements of the array.

Listing 2: Exemplary code representing a vectorized map calculation

```
1  int[] A = new int[10];
2  int[] B = new int[10];
3  int[] C = new int[10];
4  initArrays(A, B);
5
6  // scalar computation
7  for (int i = 0; i < a.length; i++) {
8      C[i] = A[i] + B[i];
9  }
10
11 // vectorized computation
12 int i;
13 for (i = 0; i + 4 <= a.length; i += 4) {
14     C[<i, i+1, i+2, i+3>] = A[<i, i+1, i+2, i+3>] + B[<i, i+1, i+2, i+3>];
15 }
16 for (; i < a.length; i++) {
17     C[i] = A[i] + B[i];
18 }
```

The example shown in listing 2 is written in a Java-like pseudo code where `<1, 2, 3, 4>` represents a vector of 4 elements. Operations on these vectors are implied to be executed element wise and in parallel. As can be seen in the scalar version of the loop, all elements of arrays `A` and `B` are added and the result is stored in array `C`. While the scalar version processes just 1 element per loop iteration, the vectorized loop can process 4 elements per iteration by executing the addition on 4 elements of the given arrays simultaneously. Due to the fact that the given arrays can be of any length and not just multiples of the vector length, an extra scalar loop is necessary to process the remaining elements that did not fill a vector that could be processed in the vector loop. This additional loop is called a scalar tail loop. In the given example, elements 0 through 7 would be processed within the vector loop while the remaining elements 8 and 9 would be handled by the scalar tail loop.

Vectorizing loops comes at the cost of significantly increased code size but the maximum theoretical performance gain, when large sets of data are processed in this fashion, is a factor close to the vector length.

3 Fold Loops

The following chapter introduces the fundamentals of fold loops (sections 3.1 and 3.2). Further in section 3.3, the problem of vectorizing fold loops containing floating point arithmetic is discussed.

3.1 Fold Loop Patterns

Map and *fold* loops are common loop patterns for manipulating sets of data. Map loops take a set of data and execute an operation on each individual element thereby producing a new set of data (see the example in listing 2). Fold loops on the other hand do not produce a new set of values like map loops, but they rather fold the given data set into a single scalar value called the *accumulator* via a given computation. This thesis will focus on fold loops, since these are often more complex to vectorize and need special treatment.

Listing 3: Calculating the sum of all elements in an array

```
1 long sum = 0;
2 for (int i = 0; i < array.length; i++) {
3     sum = sum + array[i];
4 }
```

Listing 4: Counting the number of non `null` elements in an array

```
1 int count = 0;
2 for (int i = 0; i < array.length; i++) {
3     if (array[i] != null) {
4         count++;
5     }
6 }
```

Listing 5: Calculating a simple hash code

```
1 int hash = 1;
2 for (int i = 0; i < array.length; i++) {
3     hash = hash * 31 + array[i];
4 }
```

Fold loop patterns appear in various different contexts. A few common examples for fold loops are a simple sum of an array (e.g. listing 3), counting how many elements of an array satisfy certain criteria (e.g. counting the number of null pointers in an array of objects as shown in listing 4), or calculating a simple hash code (e.g. listing 5).

In contrast to the map loop discussed in listing 2, the computation in fold loops does depend on the result of previous loop iterations. Therefore, additional transformation steps and restrictions on vectorizability are required.

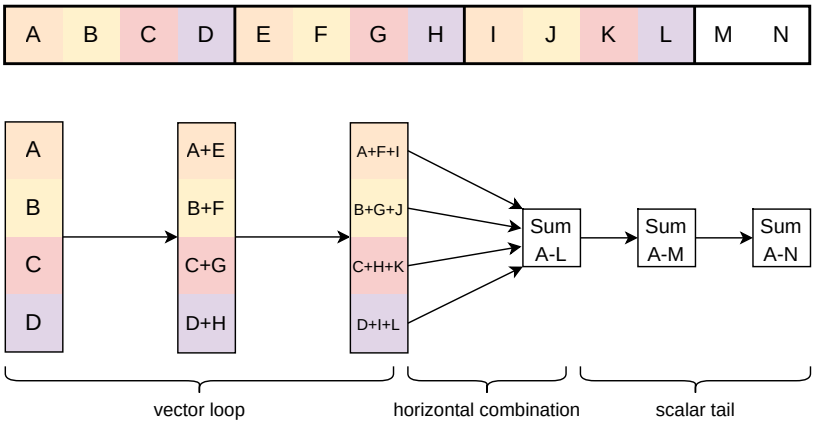


Figure 3: Vectorized sum calculation

Figure 3 shows an example for a vectorized calculation of the sum of all array elements (see listing 3) for an array of length 14 containing elements A through O and with a vector length of 4. For vectorization, the accumulator variable *sum* is extended to an accumulator vector. In each element of the vector every nth $+ i$ element ($n = \textit{vector length}$, $i = \textit{element index}$) of the array is accumulated independently. Once there are fewer than n elements left to process, the vector loop can not be executed anymore. To obtain a scalar result from the accumulator vector, the elements of the vector are combined horizontally using the operation defined in the body of the original loop. Finally, the remaining elements are processed using a scalar tail loop.

3.2 Accumulator Path

Listing 6: Fold loop containing arithmetic not on the accumulator path

```
1 int accumulator = 5;
2 for (int i = 0; i < a.length; i++) {
3     accumulator = accumulator + a[i] * b[i];
4 }
```

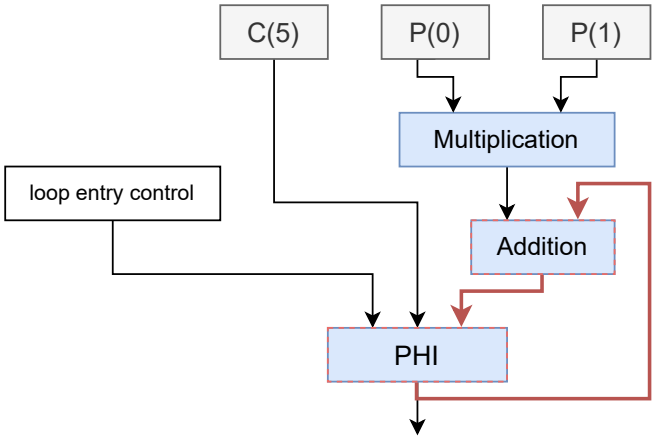


Figure 4: Diagram of the loop body of listing 6

Figure 4 shows the body of the loop in listing 6. Control flow for the loop as well as loading of array elements is omitted for simplicity sake. P(0) represents the element of array `a` needed for the current loop iteration. P(1) represents the same for array `b`. C(5) represents the constant 5. The recursive calculation marked in red is the accumulator path in the loop.

The accumulator path is the recursive path starting and ending at a PHI controlled by the loop begin.

As can be seen, the multiplication in not contained in the accumulator path, therefore this multiplication can conceptually be extracted into a separate map loop leaving a simpler minimized version of the fold loop that only contains the accumulator path.

Due to this possible minimization, the restrictions on fold loops discussed in this thesis exclusively apply to the calculation along the accumulator path.

3.3 Non-associative Rings with Identity

Vectorizing scalar fold loops may require various properties found in non-associative rings with identity. For vectorizing loop shapes similar to listing 3 or listing 4, the first 3 conditions in definition 1 are needed while for vectorizing listing 5 the conditions 1, 2, 3, 5 and 7 are needed (see section 4.1). For vectorizing folds containing subtractions as described in section 5.2, conditions 1 through 4 are required.

Definition 1 (Non-associative Ring with Identity)**.** *A* **non-associative ring with identity [11, 12]** *is defined as a set S and two operations $\circ$ and $*$ that satisfy the following conditions:*

1. $\forall a, b, c \in S : (a \circ b) \circ c = a \circ (b \circ c)$
   The operation $\circ$ needs to be associative.

2. $\forall a, b \in S : a \circ b = b \circ a$
   The operation $\circ$ needs to be commutative.

3. $\exists e_\circ \in S : \forall a \in S : e_\circ \circ a = a \circ e_\circ = a$
   The operation $\circ$ needs to have an identity element.

4. $\forall a \in S : \exists b \in S : a \circ b = b \circ a = e$
   Each element needs to have an inverse in S with respect to the operation $\circ$.

5. $\forall a, b, c \in S : (a \circ b) * c = (a * c) \circ (a * b)$
   The operations $\circ$ and $*$ need to be right-distributive.

6. $\forall a, b, c \in S : a * (b \circ c) = (a * b) \circ (a * c)$
   The operations $\circ$ and $*$ need to be left-distributive.

7. $\exists e_* \in S : \forall a \in S : e_* * a = a * e_* = a$
   The operation $*$ needs to have an identity element.

Definition 2 (Subring)**.** *Let $(A, \circ, *)$ be a ring like structure. A* **subring [13]** *of A is a subset $B \subseteq A$ such that:*

1. $e_\circ \in B \wedge e_* \in B$
   The identity elements for both operations need to be contained in B.

2. $\forall x \in B : \exists y \in B : x \circ y = e_\circ$
   Each element in B needs to have an inverse element with respect to $\circ$ in B.

3. $\forall x, y \in B : x \circ y \in B \wedge x * y \in B$
   B needs to be closed over $\circ$ and $*$.

For the set of all real numbers in conjunction with addition and multiplication it can be proven that this combination is a non-associative ring with identity with the additive neutral element $e_+ = 0$ and the multiplicative neutral element $e_\cdot = 1$. But because computers can not represent any arbitrary real number, we can not assume the afore mentioned properties for the set of all numbers representable in computer arithmetic. Two representations for numbers commonly used in digital computing are integer numbers of fixed length and floating point numbers as defined in the IEEE 754 standard.

3.3.1 Integers

It can be proven that the set of all integers $\mathbb{Z}$ in combination with addition and multiplication is a non-associative ring with identity with the additive neutral element $e_+ = 0$ and the multiplicative neutral element $e_\cdot = 1$.

Though due to the limitation in size to a power of two in the number of digits in binary representation, integers in digital arithmetic are effectively taken modulo 2^b (where $b = \mathit{bitcount}$) with each operation, thereby forming a subring to the ring of integers $\mathbb{Z}$ with addition and multiplication.

This holds because the set of integers $\mathbb{Z}_{2^b}$ always includes the additive neutral element 0 and the multiplicative neutral element 1 for $\mathit{bitcount} \in \mathbb{N}^*$. Furthermore, $\mathbb{Z}_{2^b}$ always contains the inverse element y to an element x with respect to addition which can be calculated via $y = (2^b - x)\%2^b$. Finally, because the result of every operation is taken modulo 2^b resulting in an element in $\mathbb{Z}_{2^b}$, $\mathbb{Z}_{2^b}$ is closed under addition and multiplication.

Therefore we can conclude that $(\mathbb{Z}_{2^b}, +, \cdot)$ is a subring to $(\mathbb{Z}, +, \cdot)$. With this, we can assume all properties of non-associative rings with identity for digital integer arithmetic.

3.3.2 Floating Point Numbers

Floating point numbers according to IEEE 754 are just mere approximations of the numbers they represent. Additionally, IEEE 754 defines some bit patterns as *Not a Number* (NaN). Furthermore, the standard defines +0 and -0 as well as positive and negative infinity. Due to accumulating rounding errors, IEEE 754 is not associative or distributive with respect to addition and multiplication [14].

Compilers must preserve the programs semantics through the entire optimization process. Therefore, folds containing floating point arithmetic can not be vectorized. Some compilers like the LLVM compiler provide an option called `-ffast-math` that enables treating floating point operations as if they were a non-associative ring with identity. This allows those compilers to optimize floating point arithmetic more aggressively and to vectorize folds containing floating point arithmetic. GraalVM does not provide this option. If only the highly optimizing compiler vectorizes such a computation, the same code might yield different results depending on whether the code is executed in the interpreter, compiled using the C1 compiler or compiled using the highly optimizing compiler. This might result in significant problems for other parts of the code during execution.

To further emphasize that folds containing floating point arithmetic can not be vectorized, an example that computes the sum on an array of floats exhibiting a standard

fold loop pattern is presented in listing 7. At the start, an array of floats is filled with pseudo-random floating point values. First the result for the standard scalar implementation is calculated. Then the result of a calculation resembling the vectorized version of the scalar calculation is computed. This calculation is not necessarily vectorized in the compiler. Finally, both results are printed to the standard output.

Listing 7: Demonstration of floating point rounding error incurred by vectorization

```
1  Random rng = new Random(0xCAFEBABE);
2  float[] data = new float[0xBEEF];
3  for (int i = 0; i < data.length; i++) {
4      data[i] = rng.nextFloat() / data.length;
5  }
6
7  // scalar computation
8  float sumS = 0;
9  for (int i = 0; i < data.length; sumS += data[i++]) ;
10
11 // vectorized computation
12 float[] acc = new float[] {0.0f, 0.0f, 0.0f, 0.0f};
13 int i;
14 for (i = 0; i + 4 <= data.length; i += 4) {
15     acc[0] += data[i];
16     acc[1] += data[i + 1];
17     acc[2] += data[i + 2];
18     acc[3] += data[i + 3];
19 }
20 float sumV = acc[0] + acc[1] + acc[2] + acc[3];
21 for (; i < data.length; sumV += data[i++]) ;
22
23 System.out.printf("scalar:␣.␣.␣.␣.␣%.10f\n" +
24                   "vectorized:␣.␣.␣%.10f\n", sumS, sumV);
```

Running the Java code snippet from listing 7 on GraalVM CE version 17.0.6 generates the following output:

```
scalar: . . . . 0.5002341270
vectorized: . . 0.5002374649
```

As can be seen in the output, the results of the scalar calculation and the vectorized calculation start to diverge at the 6th digit past the decimal point. This is due to the scalar computation picking up a different amount of floating point rounding error than the vectorized calculation. Therefore, these two calculations can not be considered semantically equal which implies that this optimization can not be used for floating point values.

4 Vectorizing Mixed Arithmetic Fold Loops

This chapter presents the additions made to the Graal Compiler with respect to the vectorization of fold loops that contain more than 1 unique arithmetic operation. After a quick introduction to hash code like loops, an improvement to the existing vectorization algorithm is presented in section 4.3. Subsequently, the vectorization algorithm is expanded to handle arbitrary multipliers (in section 4.4.1) and to also handle bit-wise exclusive or, bit-wise or and left-shift (in sections 4.4.2 and 4.4.3).

4.1 Hash Code Like Loops

Mixed arithmetic folds are defined as fold loops containing multiple different arithmetic operations along the accumulator path. A widely used example for such loops are fold loops that contain a multiplication and an addition along the accumulator path, so called *hash code like loops*.

Listing 8: The method body of the method `Arrays#hashCode(int[] a)` taken from the Java Standard Library

```
1 if (a == null) return 0;
2 int result = 1;
3 for (int element : a) {
4     result = 31 * result + element;
5 }
6 return result;
```

In listing 8 the method `Arrays#hashCode(int[])` is presented. The first operation in the accumulate statement we call the *multiplicative operation* while the operation directly after the multiplicative operation we call the *secondary operation* of a hash code like calculation.

This computational pattern is not only used in simple and quick hash codes, but also in a family of hashes considered cryptographically safe called Poly1305 [15]. There, among other minor changes that do not alter the computational pattern, instead of a constant, a key is used for the multiplication to derive the hash.

4.2 Previous State

The previous implementation of vectorizing these hash code like loops is based on a master's thesis called "Idiom-driven innermost loop vectorization in the presence of cross-iteration data dependencies in the HotSpot C2 compiler" by William Sjöblom [16] which

proposes a solution for vectorizing single statement loops with data dependence on previous iterations of the loop. The multiplication constant inside the loop was limited to be $2^n \pm 1$ where $n \in \mathbb{N}$. Additionally, only the combination of a single multiplication and a single addition was recognized as vectorizable, despite other similar loop shapes containing different operations occurring in real world code.

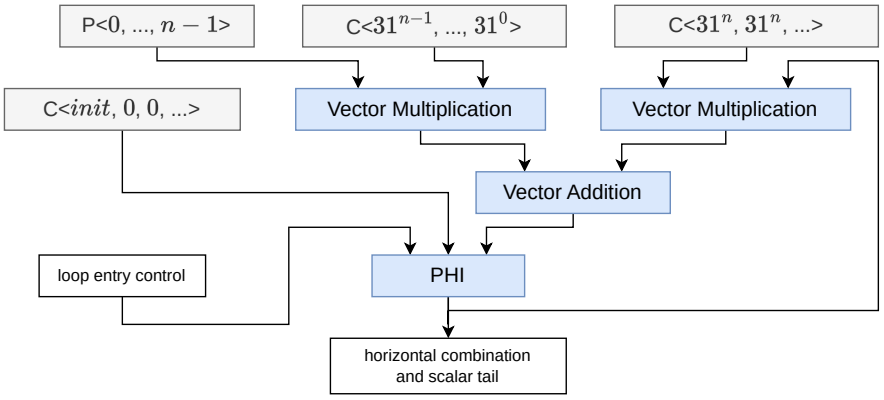


Figure 5: Previous method of vectorizing hash code like loops

Figure 5 depicts the main vectorized loop of the version of the calculation in the loop body in `Arrays#hashCode(int[])` obtained via the previous algorithm for vectorization. It only depicts the calculation inside the main vector loop obtained via vectorization. Control flow, associated checks, as well as loading of array elements are omitted for simplicity sake. The parameter vector depicted by $P < 0, ..., n-1 >$ represents a vector of array elements to be processed in the given loop iteration, while vectors with constant elements are depicted as $C < 1, 2, ... >$. The vector length n is a variable dependent on the capabilities of the target architecture. The value *init* is the initial value the accumulator variable contains before the hash code like loop is entered, for example $init = 1$ for listing 8.

4.3 Improving the Vectorization of Hash Code Like Loops

As presented in section 4.2, the previous implementation for vectorizing hash code like loops uses 2 multiplications and an addition inside the loop. This can be improved upon by sinking the left vector multiplication found in figure 5 below the loop.

To demonstrate the transformation of the calculation needed to extract data parallelism in hash code like loops, a simple example will be used. The following example calculates the hash of an array a of length 6 and a vector length of 2 according to the

implementation of `Arrays#hashCode` from the Java Standard Library presented in listing 8. For demonstration purposes the array length is chosen as a multiple of the vector length to omit the scalar tail needed to handle arrays of arbitrary length in the example.

$result =$

$= (((((1 \cdot 31 + a[0]) \cdot 31 + a[1]) \cdot 31 + a[2]) \cdot 31 + a[3]) \cdot 31 + a[4]) \cdot 31 + a[5]$

Original shape of the calculation

$= 1 \cdot 31^6 + a[0] \cdot 31^5 + a[1] \cdot 31^4 + a[2] \cdot 31^3 + a[3] \cdot 31^2 + a[4] \cdot 31^1 + a[5] \cdot 31^0$

Using right-distributivity to eliminate parentheses and adding a $\cdot 31^0$ using the existence of a neutral element for the right side of the multiplication

$= a[0] \cdot 31^5 + a[2] \cdot 31^3 + a[4] \cdot 31^1 + 1 \cdot 31^6 + a[1] \cdot 31^4 + a[3] \cdot 31^2 + a[5] \cdot 31^0$

Using the associative and commutative properties of addition to reorder the calculation

$= 0 \cdot 31^7 + a[0] \cdot 31^5 + a[2] \cdot 31^3 + a[4] \cdot 31^1 +$
$1 \cdot 31^6 + a[1] \cdot 31^4 + a[3] \cdot 31^2 + a[5] \cdot 31^0$

Using the existence of an additive identity element to add to the calculation to bring the number of operands up to a number divisible by the chosen vector length

$= (0 \cdot 31^6 + a[0] \cdot 31^4 + a[2] \cdot 31^2 + a[4] \cdot 31^0) \cdot 31^1 +$
$(1 \cdot 31^6 + a[1] \cdot 31^4 + a[3] \cdot 31^2 + a[5] \cdot 31^0) \cdot 31^0$

Again using right-distributivity and additive associativity to extract one multiplication per line. The previous implementation did not include this step.

$= (((0 \cdot 31^2 + a[0]) \cdot 31^2 + a[2]) \cdot 31^2 + a[4]) \cdot 31^1 +$
$(((1 \cdot 31^2 + a[1]) \cdot 31^2 + a[3]) \cdot 31^2 + a[5]) \cdot 31^0$

Using right-distributivity to gain a uniform pattern

To obtain a vectorized calculation from the last calculation, each line can be interpreted one element of a SIMD register and each expression in parentheses can be interpreted as an iteration of the loop. Therefore, each loop iteration in the example consists of a single vector multiplication with 31^2 followed by a vector addition of the next array elements. After the loop the accumulator vector needs to be multiplied with the vector $< 31^1, 31^0 >$ while horizontal combination is done using addition. All remain-

ing elements that did not fit into a SIMD register will get taken care of using the usual scalar tail.

$$
\begin{aligned}
result \; = \; & (((0 \cdot 31^1) \cdot 31^2 + a[0] \cdot 31^1) \cdot 31^2 + a[2] \cdot 31^1) \cdot 31^2 + a[4] \cdot 31^1 + \\
& (((1 \cdot 31^0) \cdot 31^2 + a[1] \cdot 31^0) \cdot 31^2 + a[3] \cdot 31^0) \cdot 31^2 + a[5] \cdot 31^0
\end{aligned}
$$

Using right-distributivity to gain a uniform pattern from the 4th step of the equation presented above.

As hinted earlier, the previous implementation did not perform step 5 of the transformation, which then results in the calculation presented right above. Here, the familiar pattern of 2 multiplications and one addition inside the loop, like already shown in figure 5, emerges.

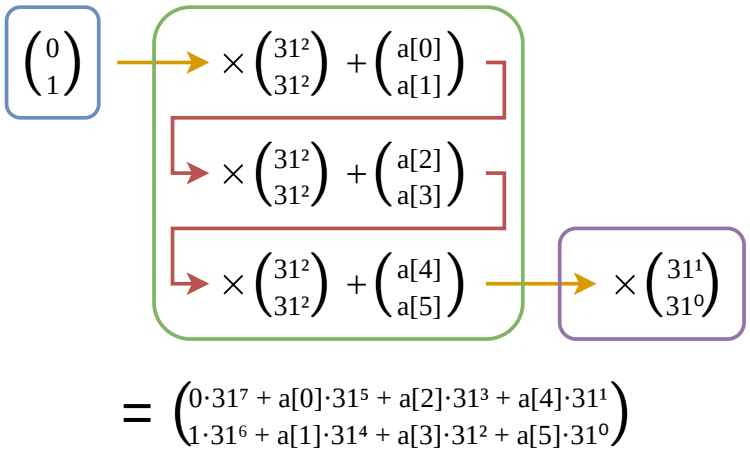


Figure 6: Illustration of the vector loop

Figure 6 features an illustration of the vector loop of the previous example for the new algorithm. Marked in blue is the initial state of the accumulator vector. Marked in green are 3 iterations of the vector loop. As can be seen, in each loop iteration, the previous result is multiplied by the vector $< 31^2, 31^2 >$ (this multiplication is implied to be element-wise) and subsequently added to a vector of array elements. Finally, marked in purple, the result of the vector loop is multiplied by the vector $< 31^1, 31^0 >$. To now obtain a scalar result, all elements of the vector result are summed up to obtain the scalar result $0 \cdot 31^7 + a[0] \cdot 31^5 + a[2] \cdot 31^3 + a[4] \cdot 31^1 + 1 \cdot 31^6 + a[1] \cdot 31^4 + a[3] \cdot 31^2 + a[5] \cdot 31^0$ which is exactly the fourth step in the previous example.

Listing 9: Vectorized hash like computation

```
1  // vector loop
2  int i;
3  IntVector accVector = <0, ..., 0, init>;
4  for (i = 0; i + vlen < a.length; i += vlen) {
5      IntVector temp = accVector * <c^vlen, c^vlen, ...>;
6      accVector = temp + a[<i, i+1, ..., i+(vlen-1)>];
7  }
8
9  // horizontal combination
10 accVector = accVector * <c^(vlen-1), ..., c^1, c^0>;
11 int result = sum(accVector);
12
13 // scalar tail
14 for (; i < a.length; i++) {
15     result = result * c + a[i];
16 }
```

Listing 9 depicts the resulting calculation for a hash like fold loop for an integer array in a Java-like pseudo code. The vector length *vlen* and the constant *c* are left as variables to demonstrate the generic nature of this reshaped calculation.

As can be seen, the vector loop that processes the majority of the calculation contains only a single multiplication and one addition. The horizontal combination with addition is preceded with the multiplication that has been sunk out of the loop.

During literature search I found out that this improvement of sinking one multiplication below the loop has already been done before. The family of cryptographic hashing algorithms called Poly1305 use a similar pattern of multiplication and addition in a loop to calculate a hash using a key. A SIMD algorithm for calculating the Poly1305 hash that extracts the multiplication from the loop has been presented in a paper called "Vectorization of Poly1305 Message Authentication Code" by Martin Goll and Shay Gueron in 2015 [17].

4.4 Generalizing Hash Code Loops

Investigations on the Java Standard Library revealed that hash code like loop patterns do not only occur with a constant multiplier of $2^n \pm 1$ in combination with addition but rather come in a variety of different flavors. The aim of this contribution is to open up the detection and vectorization of hash code like loop patterns to also include patterns that do not closely match the implementation in `Arrays#hashCode(int[])` (see listing 8).

4.4.1 Detecting Arbitrary Multipliers

Because multiplications are quite costly operations [18], in Graal multiplications with constants are optimized to a faster representation consisting of shifts, negates, additions and subtractions immediately upon insertion into the IR. These optimizations range from simply replacing the multiplication with a single left-shift (if the multiplication constant is a power of 2) to quite complex patterns of 2 left-shifts and a subtraction if the constant multiplier consists of all ones followed by all zeros (as demonstrated in listing 10). The implementation of this can be found on the public Graal GitHub in the `MulNode#canonical` method [19].

Listing 10: Canonicalization of a multiplication

```
1 // original calculation
2 result = input * 0b01111000;
3
4 // optimized calculation
5 result = (input << 7) - (input << 3);
```

To accurately detect such optimized versions of multiplications with constants, a new algorithm was devised. In contrast to the previous detection algorithm that works in a bottom up fashion, the new detection works in a top down manner following usage edges along the data flow in the IR graph.

Since a possible multiplier of a hash code like loop is always a direct usage of the accumulator, detecting the accumulator in a top down fashion allows for additional arithmetic in the loop that uses the hash code like computation.

The detection starts at the accumulator and in a first step all usages inside the loop that are multiplications and left-shifts are collected. If the accumulator is used in more than 1 multiplication or more than 2 left-shifts, the algorithm used for vectorizing hash code like loops can not handle this. Therefore, such loops are already rejected while detecting the constant multiplier.

The most simple case of a successful detection is, when just a single multiplication is detected. In this case no optimization was made to the multiplication and the detection is trivial. If the input that is not the accumulator is a constant, this value is the desired constant multiplier, if the input is not a constant, no constant multiplier was found and the detection of a vectorizable hash code like loop terminates.

The method `MulNode#canonical(Stamp, ValueNode, long, NodeView)` [19] can produce 6 different optimized shapes which are presented in figure 7. The constant inputs n and k for the given left-shift operations in figure 7 are incorporated into the node description to reduce the total node count in order to simplify the resulting graphs.

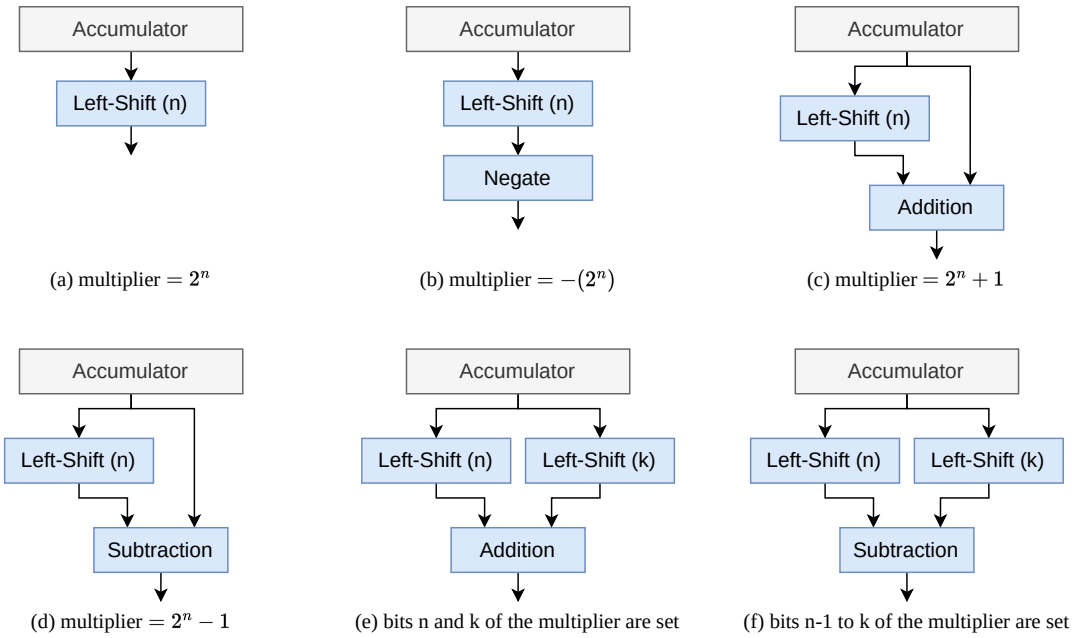


Figure 7: Possible shapes of a multiplication with a constant

If a singular left-shift by a constant is found as a direct usage of the accumulator inside the loop, it can be concluded that if a multiplication with a constant is present, it is of the form presented in figure 7 (a) through (d). To determine the exact multiplier, the operation using the left-shift's result needs to be evaluated. If this operation is a negate, the calculation shape is of type (b) and the desired constant can be computed via $const = -(2^n)$.

If the operation using the left-shift's result is an addition, the shape of the calculation could be (a) or (c). To determine if the addition is part of the optimized multiplication, it needs to be checked if the input edge not connecting this operation to the previous left-shift, connects to the accumulator. If this is the case, the desired constant can be calculated via $const = 2^n + 1$, else the addition is not part of the multiplication and the constant is $const = 2^n$ following the calculation shape presented in (a).

If the operation using the left-shift's result is a subtraction, the detection works similar to the handling of an addition described above. The constant can either be calculated via $const = 2^n - 1$ or $const = 2^n$ based on if a shape of type (a) or (d) is detected.

If 2 left-shifts with constants are detected as direct usages the accumulator inside the loop, the multiplication can be of type (e) or (f). If these two left-shifts do not have a singular common usage, no constant factor is detected and the search for an optimizable, hash code like calculation is terminated. If, however, the common usage of

the two left-shifts is an addition, the detected shape is (e) constant can be calculated via $const = 2^n + 2^k$. This results in a constant with exactly two bits (at n and k) set like for example 0b10100 for $n = 4$ and $k = 2$.

If the common usage of the two constant left-shifts is subtraction, the detected shape is (f) and the constant is calculated via $const = 2^n - 2^k$. The optimization leading to a shape like presented in (f) is applicable for constants like 0b11100 for which $n = 5$ and $k = 2$. The bits $n - 1$ to k are set, the rest are zeros.

Because this algorithm does not traverse the entire accumulator path, the remaining operations along the accumulator path need to be checked separately to ensure the loop actually represents a vectorizable hash code like calculation.

With this new algorithm for detecting arbitrary multipliers in hash code like loops implemented, the Graal loop vectorizer now can vectorize hash code like loops containing any constant multiplication in combination with addition as the secondary operation.

4.4.2 Shifts and XORs

To apply the vectorization algorithm presented in section 4.3, the properties 1, 2, 3, 5 and 7 of non-associative rings with identity (definition 1) are needed. Therefore, to vectorize left-shifts (in formulas abbreviated to '<<') as the multiplicative operation in combination with bit-wise XORs (in formulas presented as '`XOR`') as the secondary operation of a hash code like loop, these properties need to be proven in combination with $\mathbb{Z}$.

For the set S from definition 1 take $\mathbb{Z}$, for $\circ$ take `XOR` and for $*$ take $<<$. The bit-wise XOR operation is defined as an operation where, in binary representation, every bit is joined separately via the logical XOR operation which has the logic table presented in table 1.

XOR	0	1
0	0	1
1	1	0

Table 1: Logic table for XOR

1. (associativity): The logical XOR operation is associative as can be deduced using the logic table 1. Because the bit-wise XOR operation over $\mathbb{Z}$ consists of multiple independent logical XOR operations it can be concluded that therefore also bit-wise XOR is associative.

2. (commutativity): The logical XOR operation is commutative and using the argument presented in point 4.4.2 this property can be derived for bit-wise XOR.

3. (XOR identity): The identity element for bit-wise XOR is 0.

5. (right-distributivity): The k-th bit of the result of the calculation $x << y$ can be expressed as $(x << y)_k = x_{k-y}$. Additionally, the k-th bit of the result of the calculation x `XOR` y can be expressed as $(x \texttt{ XOR } y)_k = x_k \texttt{ XOR } y_k$.

   Using these representations, the calculation for the k-th bit of the left side of the equality in definition 1.5 can be written as
   $((a \texttt{ XOR } b) << c)_k = (a \texttt{ XOR } b)_{k-c} = a_{k-c} \texttt{ XOR } b_{k-c}$
   while the calculation for the right side of the calculation can be written as
   $((a << c) \texttt{ XOR } (b << c))_k = (a << c)_k \texttt{ XOR } (b << c)_k = a_{k-c} \texttt{ XOR } b_{k-c}$
   As can be seen, the final results of the calculations match, therefore it can be concluded, bit-wise XOR and left-shift are right-distributive.

7. (left-shift identity): Although there is no general identity element for left-shift, the identity element for the right side of the left-shift operation is 0. This suffices for the transformation presented in section 4.3.

Finally, it needs to be proven that the afore proven properties also hold for the subset $\mathbb{Z}_{2^n}$ with $n = bitcount$. It can be said that $0 \in \mathbb{Z}_{2^n}$. Additionally, x `XOR` y produces a result of at maximum $2^{max(x_{len},y_{len})+1} - 1$ which is an element of $\mathbb{Z}_{2^n}$ because $x_{len} \leq n \wedge y_{len} \leq n$. Finally, because the top bits of the result get cut in left-shift, the result is also always in $\mathbb{Z}_{2^n}$.

With this proven, mixed arithmetic fold loops with a left-shift and bit-wise XORs can be vectorized using the same algorithm used for multiplication and addition.

4.4.3 Shifts and ORs

To prove the afore mentioned vectorization does work with a left-shift and bit-wise OR operations, a very similar argument to the one presented in section 4.4.2 can be made.

OR	0	1
0	0	1
1	1	1

Table 2: Logic table for OR

A look at the logic table of the logical OR operation found in table 2 reveals that, in contrast to the logical XOR operation, the logical OR operation (and by extension also the bit-wise OR operation) can not be reversed. This, however, does not pose a problem since this optimization does not use inverses. All necessary properties of the algebraic structure ($\mathbb{Z}_{2^n}$, `OR`, $<<$) can be proven in the same way as presented for the bit-wise XOR operation in section 4.4.2.

Given all properties proven above, we can now vectorize all hash code like loops containing not only a multiplication combined with addition, but also a left-shift combined with bit-wise XOR and bit-wise OR.[1]

[1] 2 examples for uses in the JDK are:

1. `https://github.com/openjdk/jdk/blob/3bcfac18c39d83bf876787e7ce422831bab0db2f/src/java.base/share/classes/java/io/ObjectStreamClass.java#L1861-L1863`
2. `https://github.com/openjdk/jdk/blob/3bcfac18c39d83bf876787e7ce422831bab0db2f/src/java.base/share/classes/jdk/internal/jimage/decompressor/CompressIndexes.java#L99-L103`

5 Subtractions in Fold Loops

This chapter presents a possible approach to vectorize fold loops containing a mixture of additions and subtractions. First, in section 5.2 a simple approach is presented that has been implemented into the Graal Compiler. In the second part, in section 5.3.1, a more complex approach to lift the restrictions incurred by the simple approach is presented.

5.1 Problem Statement

To perform the transformations necessary to extract data parallelism presented in section 3.1, the operations on the accumulator path need to be associative, commutative and have a neutral element. The subtract operation is neither associative nor commutative, therefore loops containing a subtraction can not be vectorized using the standard algorithm for fold loops. This work adds an algorithm to the Graal Loop Vectorizer to be able to handle subtractions in fold loops under certain conditions presented in section 5.3.

5.2 Basic Idea

Listing 11: Fold loop containing a subtraction

```
1 int acc = init;
2 for (int i = 0; i < a.length; i++) {
3     acc = acc - a[i];
4 }
```

Looking at the loop in listing 11 the loop contains the operation `acc = acc - a[i]` which can be rewritten as `acc = acc + (-a[i])`. With this, the accumulator path only contains a single addition which is associative and commutative while the extracted map operation consists of a single negate operation. This allows the calculation to be reordered arbitrarily therefore enabling vectorization similar to a fold loop containing only additions.

Listing 12: Vectorized fold loop containing a subtraction

```
1  // vector loop
2  int i;
3  IntVector accV = <init, 0, ..., 0>;
4  for (i = 0; i + vlen < a.length; i += vlen) {
5      accV = accV - a[<i, i+1, ..., i+(vlen-1)>];
6  }
7
8  // horizontal combination
9  int acc = sum(accV);
10
11 // scalar tail
12 for (; i < a.length; i++) {
13     acc = acc - a[i];
14 }
```

After transforming the calculation to extract data parallelism, the map operation can be reintegrated into the fold calculation. This results in a vector loop containing subtractions, the horizontal combination being done with addition instead of subtraction and the scalar tail resorting to the original loop as shown in listing 12.

The fact that the subtraction from the original scalar loop is effectively preserved in the vector loop, therefore not interfering with other arithmetic, and the fact that both fold loops containing additions and subtractions are combined horizontally using addition, allow subtractions and additions to be mixed arbitrarily in fold loops for the loop to be still vectorizable.

5.3 Accumulator Path Restriction

The keen reader might have noticed a significant limitation of the approach presented previously in section 5.2.

Listing 13: Accumulator path traversing the right input edge of a subtraction

```
1 int acc = init;
2 for (int i = 0; i < a.length; i++) {
3     acc = a[i] - acc;
4 }
```

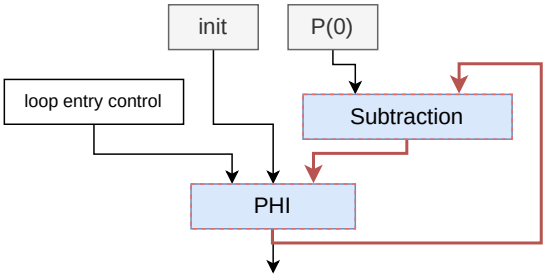


Figure 8: Accumulator path of the loop in listing 13

This simple approach does not allow the accumulator path to traverse the right input edge of a subtraction. The example loop presented in listing 13 uses the accumulator as a subtrahend, therefore the accumulator path traverses such a right input edge as can be seen in figure 8. Looking at an example input for this calculation using an array of length 6, the calculation $result = a[5] - (a[4] - (a[3] - (a[2] - (a[1] - (a[0] - init)))))$ can be obtained, which can not be transformed as easily to extract data parallelism as the approach presented in section 5.2 might suggest.

5.3.1 Lifting the Accumulator Path Restriction

To lift this accumulator path restriction, a more complex algorithm needs to be devised. Revisiting the last example in section 5.3, the calculation can be transformed to $result = a[5] - a[4] + a[3] - a[2] + a[1] - a[0] + init$ using the existence of an additive inverse element. As can be seen, every second component of the calculation is inverted. Recognizing this pattern, this calculation can now be transformed further assuming a vector length of 2.

$$
\begin{aligned}
result = \quad & a[5] - a[4] + a[3] - a[2] + a[1] - a[0] + init \\
& \text{initial calculation} \\
= \quad & 0 \quad - a[0] - a[2] - a[4] + \\
& init + a[1] + a[3] + a[5] \\
& \text{reordering the calculation and adding 0 in between} \\
= \quad & (0 \quad + a[0] + a[2] + a[4]) \cdot (-1) + \\
& (init + a[1] + a[3] + a[5]) \cdot (1) \\
& \text{negating every other line in the calculation starting from the top to} \\
& \text{eliminate the subtractions}
\end{aligned}
$$

In the last expression, each line can again be interpreted as an element of the accumulator vector while each column of additions can be interpreted as an iteration of the vector loop. The horizontal combination of the vector elements is preceded by inverting every other vector element starting at index 0. This vector can be expressed generically as $< (-1)^{vlen-1}, ..., (-1)^1, (-1)^0 >$ or $< -1, 1, -1, 1, ... >$. Finally the vector elements are combined via addition.

The similarities between this approach to vectorize subtractions in fold loops and the approach to vectorizing hash code like loops presented in section 4.3 are not coincidental. The calculation `accumulator = a[i] - accumulator` inside the loop can be rewritten as `accumulator = accumulator * -1 + a[i]`. This means, a fold loop with an accumulator path traversing the right edge to a subtraction can be seen as a hash code like loop with addition and a constant multiplier of -1 and it can be therefore vectorized using the same algorithm.

5.3.2 Vectorizing Arbitrary Add-Sub-Patterns

To now be able to vectorize any fold loop containing an arbitrary combination of additions and subtractions, the calculation inside the loop needs to be resolved in a manner that the accumulator path only contains additions and negations. A possible algorithm to do that would be to collect all inputs to the accumulator path starting at the accumulator itself while inverting every input to the accumulator path that traverses the right edge of a subtraction to have the accumulator path only consist of additions. If the accumulator path itself traverses the right edge of a subtraction, every previous input including the accumulator itself is negated.

At the end, the accumulator path would either only consist of additions which would be vectorizable using the approach presented in section 5.2, or it would contain a singular negation of the accumulator at the start followed by only additions which could be vectorized using the approach presented in section 5.3.1. With this, arbitrary combinations of additions and subtractions could be vectorized in fold loops.

This approach is illustrated by the following example. The steps are annotated below the transformations of the equation. Each annotation starts with the operation along the accumulator path that is handled next.

$$
\begin{aligned}
acc &= (c[i] - ((a[i] - acc) + b[i])) - d[i] \\
&= (c[i] - ((a[i] + (-acc)) + b[i])) - d[i] \\
&\quad [a[i] - acc]\text{: extract the negation of the accumulator} \\
&= (c[i] - (a[i] + (-acc) + b[i])) - d[i] \\
&\quad [acc + b[i]]\text{: noting to do for additions} \\
&= (c[i] + (-a[i]) + acc + (-b[i])) - d[i] \\
&\quad [c[i] - acc]\text{: accumulator path traverses the right edge to a subtraction,} \\
&\quad \text{therefore all previous inputs to the accumulator path need to be inverted} \\
&= c[i] + (-a[i]) + acc + (-b[i]) + (-d[i]) \\
&\quad [acc - d[i]]\text{: accumulator path traverses the left edge to a subtraction, therefore} \\
&\quad \text{only the negation of the right input is extracted.}
\end{aligned}
$$

The final version of the equation is a sequence of additions where the accumulator is not negated. Therefore, like discussed above, this sequence can be vectorized with the approach presented in section 5.2 after this transformation has been applied.

This algorithm, however, has not been implemented into the Graal Compiler because tests on real world code showed that the accumulator path almost never traverses the right edge to a subtraction in a possibly vectorizable fold loop (0 occurrences in all JDK classes). Therefore almost all folds containing subtractions can be vectorized using the simple approach presented in section 5.2 which was implemented into the Graal Compiler. An example taken from the JDK for a subtraction in a fold loop that is now considered vectorizable can be found in the `java.awt.GridBagLayout` class.[2]

5.4 Subtractions in Hash Code Like Loops

The final contribution of this thesis to the Graal Compiler is an implementation to vectorize hash code like loops that have a subtraction as a secondary operation or as additional arithmetic below the hash code like calculation along the accumulator path. While subtractions below the hash code like calculation will be vectorized automatically using the

[2] `https://github.com/openjdk/jdk/blob/3bcfac18c39d83bf876787e7ce422831bab0db2f/src/java.desktop/share/classes/java/awt/GridBagLayout.java#L1405-L1406`

simple approach presented in section 5.2, subtractions as the secondary operation to a hash code like calculation need special treatment.

If the accumulator path traverses the left input edge, the negation can simply be extracted and later reintegrated into the calculation, in the same fashion as explained earlier in section 5.2. If the accumulator path traverses the right edge to the subtraction below the multiplicative operation, instead of inverting the accumulator, the multiplication constant can be inverted to achieve the same result. This can be seen in the final paragraph of section 5.3.1. After this point the secondary operation is considered to be a simple addition and the hash code like computation is vectorized according to this new shape.

6 Evaluation

Finally, some of the afore mentioned improvements for hash code like loops contributed to the Graal Compiler will be evaluated.

For the following benchmarks, 2 systems were used. The first machine contains "11th Gen Intel Core i7-1165G7" CPU which implements the Intel Tiger Lake micro architecture for the AMD64 instruction set (from here on forth called the "AMD64 system"). The second machine is a server system that contains a "AppliedMicro X-Gene 3 APM883832-X3" CPU which implements Ampere Computing's Skylark architecture for the AArch64 instruction set (from here on forth called the "AArch64 system"). All benchmarks were done by executing 15 warm-up iterations followed by 5 iterations where the throughput is measured. The final result is computed by executing this routine 2 times and averaging the 10 measured iterations.

6.1 Sunk multiplication

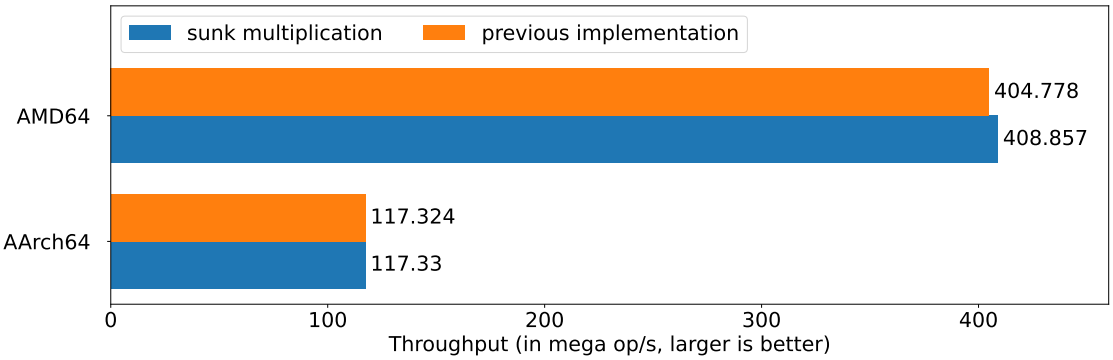


Figure 9: Hash code benchmark with small `int[]`

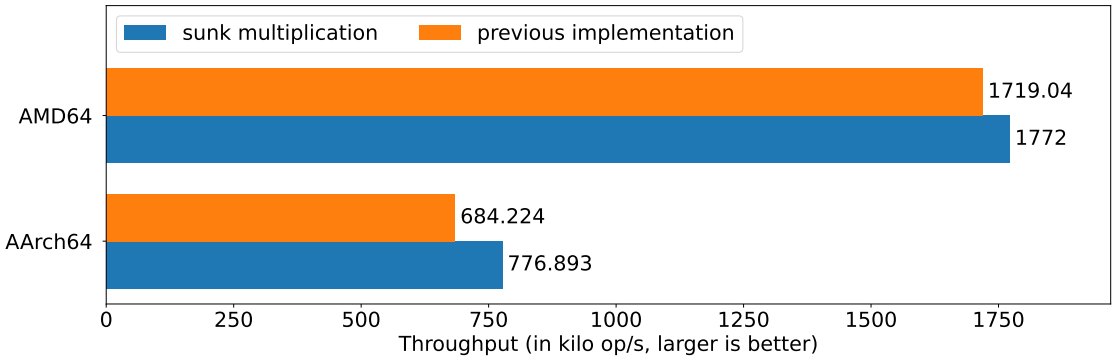


Figure 10: Hash code benchmark with large `int[]`

I implemented the improved algorithm for vectorizing hash code like loops presented in section 4.3 into the Graal Compiler.

Figures 9 and 10 show peak performance benchmarks for hashing integer arrays with the optimization of sinking one multiplication below the loop presented in section 4.3. For figure 9 an integer array of length 4 is used while for figure 10 an integer array of length 2048 is used.

In figure 9 the results for the previous implementation and the new algorithm described earlier lay within margin of error. This makes sense since the array size used is so small that the produced vector loop, that contains one less multiplication, is not executed very often.

On large arrays an increase in performance on the AMD64 system by 3% and an increase in performance on the AArch64 machine by 13.5% can be detected. Even though a multiplication, which can be quite costly, has been extracted from the loop, the performance increase did not match expectations. By taking a closer look at the previous implementation of the vectorization of hash code like loops presented in figure 5, it can be seen that the two vector multiplications do not have a data dependency between them. Therefore, pipelining in the CPU hides the latency added by the second vector multiplication.

The vector multiplication instruction `VPMULLD` for up to 256-bit SIMD registers on the Intel Tiger Lake micro architecture which is implemented in the "Intel 11th Gen Intel Core i7-1165G7" processor used in the benchmarks in figure 9 and 10, adds 10 clock cycles of latency to the calculation while the execution of data independent multiplications can be started at a rate of 1 instruction per clock cycle due to pipelining [18]. This means, that sinking the multiplication below the loop reduces the clock cycles spent multiplying

inside the loop from 11 cycles to 10. Additionally, 10 clock cycles are added to the calculation below the loop.

The AArch64 CPU model appears to have less sophisticated pipelining than the Intel CPU, so it cannot hide the effects of the second multiplication in the loop to the same extent as the Intel CPU which explains the difference in performance gain for the two tested systems.

6.2 Generalized hash code like loops

Listing 14: Hash code like computation using shift and XOR

```
1 long acc = 0;
2 for (int i = 0; i < array.length; i++) {
3     acc = (acc << 8) | array[i];
4 }
```

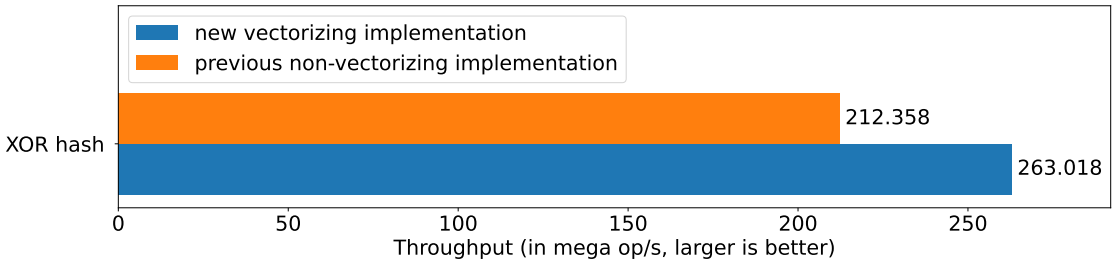


Figure 11: Benchmark for hash code like loop presented in listing 14

Hash code like fold loops with bit-wise OR or XOR are often used for constructing `int` or `long` values from an array of bytes. In listing 14 such a combination is presented. Figure 11 depicts the benchmark results for building a 64-bit integer value from a byte array of length 8 using the calculation presented in listing 14. This benchmark was executed on the AMD64 system The previous implementation that could not vectorize this loop pattern managed a throughput of 212.4 mega op/s while the new implementation that vectorizes this pattern managed to achieve a throughput of 263 mega op/s. This is an improvement of 23.9% for this loop shape.

With the new implementation, on AVX512 this loop is vectorized with a vector length of 4 because 4 64-bit integers fit into a 256-bit sized YMM SIMD register (a YMM register is used in this case because the version of GraalVM used did not fully support 512-bit wide ZMM registers yet). This results in a theoretical maximum performance increase of 4x which will not be achieved since SIMD instructions tend to be slightly slower than their

scalar counter parts. Additionally, horizontal combination and scalar tail add overhead that harms performance quite significantly for smaller array sizes.

In this particular example, the horizontal combination uses a vector multiplication instead of a element wise shift for the multiplication below the loop presented in section 4.3, because Intel AVX512 only defines an instruction for shifting all elements of a vector by the same amount but not by different amounts each. Therefore, we have to resort to a vector multiplication which is a significantly slower operation than a shift. This results in the performance gain of this transformation to be only 23.9% for a byte array of length 8.

7 Conclusion

While vectorizing loops can result in a significant uplift in performance [7], extracting data parallelism automatically from inherently scalar code is quite a challenging problem. Often, significant refactorization of the given calculation using various mathematical properties is needed. The requirement of such properties (as discussed in section 3) restricts the optimization possibilities for loop vectorization quite significantly, especially for fold loops.

An example for such restrictions is, that floating point arithmetic is not associative as shown in section 3.3.2. Therefore, these operations can not be vectorized in fold loops.

Tests on the loop vectorizer of the Graal compiler have shown that the previous implementation was already very capable of vectorizing loops containing simple arithmetic and sometimes even simple control flow. The loops that can not yet be vectorized in the Graal compiler are mostly more complex loops like for example fold loops that contain multiple different arithmetic operations, fold loops that contain operations that are not associative or commutative and loops that contain more complex control flow. Such loops need special treatment to be vectorized. This thesis provides this special treatment for some loops like hash code like loops and fold loops containing subtractions.

First, the problem of vectorizing fold loops containing mixed arithmetic was tackled. There, the previous implementation was improved by extracting one of the two multiplications from the loop in section 4.3. Afterwards in section 4.4, the previous implementation was expanded upon to allow arbitrary multipliers as well as a combination of left-shifts and bit-wise XORs or bit-wise ORs.

Finally, a new implementation to vectorize subtractions, which are neither associative nor commutative, was added to the Graal compiler in section 5.

References

[1] H. Sutter, "The free lunch is over," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[2] Intel, "MMX technology technical overview," 2023. Available at `https://www.intel.com/content/dam/develop/external/us/en/documents/mmx-manual-tech-overview-140701.pdf`.

[3] Oracle, "Moved by Java timeline," 2020. Available at `https://www.oracle.com/java/moved-by-java/timeline/`.

[4] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013* (A. L. Hosking, P. T. Eugster, and R. Hirschfeld, eds.), pp. 187–204, ACM, 2013.

[5] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck, "An intermediate representation for speculative optimizations in a dynamic compiler," in *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, Indianapolis, IN, USA, 28 October 2013* (C. Bockisch, M. Haupt, S. Blackburn, H. Rajan, and J. Gil, eds.), pp. 1–10, ACM, 2013.

[6] Intel, "Intel 64 and IA-32 architectures software developer's manual," 2023. Available at `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4`.

[7] Konstantin, "Improving performance with simd intrinsics in three use cases," 2020. Available at `https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/`.

[8] I. Rosen, "Auto-vectorization in GCC," 2023. Available at `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[9] LLVM Project, "Auto-vectorization in LLVM," 2023. Available at `https://llvm.org/docs/Vectorizers.html`.

[10] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 4, pp. 491–542, 1987.

[11] E. Weisstein, "Nonassociative algebra," 2023. Available at https://mathworld.wolfram.com/NonassociativeAlgebra.html.

[12] E. Weisstein, "Unit ring," 2023. Available at https://mathworld.wolfram.com/UnitRing.html.

[13] E. Weisstein, "Subring," 2023. Available at https://mathworld.wolfram.com/Subring.html.

[14] P. W. Markstein, "The new IEEE-754 standard for floating point arithmetic," in *Numerical Validation in Current Hardware Architectures, 6.1. - 11.1.2008* (A. A. M. Cuyt, W. Krämer, W. Luther, and P. W. Markstein, eds.), vol. 08021 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[15] S. Bhattacharyya and P. Sarkar, "Improved SIMD implementation of poly1305," *IET Inf. Secur.*, vol. 14, no. 5, pp. 521–530, 2020.

[16] W. Sjöblom, "Idiom-driven innermost loop vectorization in the presence of cross-iteration data dependencies in the hotspot c2 compiler," Master's thesis, Linköping University, Software and Systems, 2020.

[17] M. Goll and S. Gueron, "Vectorization of poly1305 message authentication code," in *12th International Conference on Information Technology - New Generations, ITNG 2015, Las Vegas, NV, USA, April 13-15, 2015* (S. Latifi, ed.), pp. 145–150, IEEE Computer Society, 2015.

[18] A. Fog, "Instruction tables," 2022. Available at https://www.agner.org/optimize/instruction_tables.pdf.

[19] C. Häubl and A. Prokopec, "GraalVM MulNode#canonical," 2017. Available at https://github.com/oracle/graal/blob/a60993b28921cc1545dbd8975767701b0f552aa4/compiler/src/jdk.internal.vm.compiler/src/org/graalvm/compiler/nodes/calc/MulNode.java#L129-L171.

List of Code Snippets

List of Definitions

List of Figures

List of Tables