The Complexity of SIMD Alignment

Liza Fireman
The Complexity of SIMD Alignment

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science

Liza Fireman

Submitted to the Senate of the Technion — Israel Institute of Technology

SIVAN, 5766 Haifa June, 2006
This Research Thesis was done under the supervision of Erez Petrank in the Department of Computer Science

The generous financial help of the Technion is gratefully acknowledged
# Contents

Abstract 1

1 Introduction 3
   1.1 Technique used ................................................. 4
   1.2 Organization .................................................... 5

2 Related Work 7

3 An Overview of the SIMD ALIGNMENT Problem 9

4 Previous Heuristics 13

5 An Abstraction of SIMD Alignment 15
   5.1 A solution to a graph representation of a single-alignment-appearance ... 16

6 A polynomial-time algorithm for expressions with only two alignments 19
   6.1 An example ...................................................... 21
   6.2 Correctness ....................................................... 23
   6.3 Complexity Analysis ............................................ 27

7 A polynomial-time algorithm for single-appearance tree expressions 29
   7.1 An example ...................................................... 31
   7.2 Correctness ....................................................... 33
   7.3 Complexity Analysis ............................................ 35

8 The MULTIWAY CUT and the NODE MULTIWAY CUT Problems 37
9 Results
  9.1 Measurements for tree expression .......................... 43
  9.1.1 An Example .............................................. 44
  9.2 Measurements for expressions with only two alignments .......... 44
  9.2.1 An Example .............................................. 46
  9.2.2 Runtime Overhead ......................................... 47

10 Conclusion and Open Problems 49

References 49

Hebrew Abstract 7
List of Figures

5.1 A graph representation of $a[i + 1] = b[i + 1] + c[i + 2] * d[i + 1]$. 16
5.2 A graph representation of $a[i + 3] = b[i + 1] * c[i + 2] + c[i + 2] * d[i + 1]$. 16
5.3 A graph that exemplifies the cost of SIMD. 17
6.1 The DAG corresponding to $f[i] = (a[i + 1] * b[i + 1] + a[i + 1] * c[i + 1]) + (a[i + 1] * d[i] + a[i + 1] * e[i]);$ 22
6.2 The graph $H$ corresponding to the DAG of Figure 6.1 23
6.3 A minimum node $s_0 - s_1$ cut for the graph in Figure 6.2. 24
7.1 The corresponding tree to the expression $a[i + 2] = b[i + 1] * c[i + 2] + d[i + 2] * e[i + 3]$ 32
7.2 Computing $val$ values for the tree in Figure 7.1. 32
7.3 Output labelling and implied shift edges for the tree in Figure 7.1. 33
8.1 A graph that exemplifies the differences between the problems. 38
9.1 A tree for which Algorithm 2 outperforms all the heuristics 44
9.2 An optimal labelling for the tree of Figure 9.1. 45
9.3 A tree for which Algorithm 2 outperforms all the heuristics 45
9.4 An optimal labelling for the tree of Figure 9.3. 45
9.5 A DAG in which Algorithm 1 outperforms all the heuristics 46
1 A graph representation of the expression $a[i + 1] = b[i + 1] + c[i + 2] * d[i + 1]$. 1
2 A graph representation of $a[i + 3] = b[i + 1] * c[i + 2] + c[i + 2] * d[i + 1]$. 2
List of Tables

9.1 The percentage of test-runs in which algorithm 2
outperformed all the heuristics ........................................ 43

9.2 The percentage of test-runs in which Algorithm 1
outperformed all the heuristics ........................................ 46

9.3 Run time of a program that represents Figure 9.5 ............... 47
Abstract

Optimizing programs for modern multiprocessor or vector platforms is a major important challenge for compilers today. Various problems in this domain are not yet thoroughly understood. In this work, we focus on one such problem: the SIMD ALIGNMENT problem. In this problem we are given a code that includes a loop with misaligned references. Such a code requires additional realignment operations to allow parallelization on SIMD architectures because of SIMD alignment constraints. The realigning of data is achieved by shifting streams of data after reading them into the SIMD registers and before applying the SIMD operations. Shifts are executed inside the loop and therefore affect the performance significantly. The problem is to automatically reorganize data streams to satisfy the alignment requirements imposed by the hardware with a minimum number of shift executions.

Previously, only heuristics were used to solve this problem, without any guarantees on the number of shifts in the obtained solution. We study two interesting and realistic special cases of the SIMD ALIGNMENT problem and present two novel and efficient algorithms that solve the problem optimally for these two cases. In the first case, we deal with expressions whose number of different alignments is not restricted, but their graph representation form a tree and any array appears in the expression at most once. For this special case we show a polynomial-time algorithm using dynamic programming. In the second case, the graph representation can be any DAG, but we restrict the expression to contain only two distinct predetermined alignments associated with the input and output operands. We use a MIN-CUT/MAX-FLOW algorithm as a subroutine.

We also discuss the relation between the SIMD ALIGNMENT problem and the MULTIWAY CUT and NODE MULTIWAY CUT problems. We show how to achieve an approximated solution to the SIMD ALIGNMENT problem based on approximation algorithms to these two known problems.
Chapter 1

Introduction

Designing effective optimizations for modern architectures is an important goal for compiler designers today. This general task is composed of many non-trivial problems, the solution to which is not always known. In this paper, we study one such problem, the SIMD ALIGNMENT problem, which emerges when optimizing for multimedia extensions. Previously, only heuristics were studied for this problem [20, 24, 13]. In this paper we present two novel algorithms that obtain optimal solutions for two special cases. These special cases are actually broad enough to cover most practical instances of the SIMD ALIGNMENT problem.

Multimedia extensions have become one of the most popular additions to general-purpose microprocessors. Existing multimedia extensions are characterized as Single Instruction Multiple Data (SIMD) units that support packed, fixed-length vectors, such as MMX and SSE for Intel and AltiVec for IBM, Apple and Motorola. Producing SIMD codes is sometimes done manually for important specific application, but is mostly produced automatically by compilers (referred as simdization). Explicit vector programming is time consuming and error prone. A promising alternative is to exploit vectorization technology to automatically generate SIMD codes from programs written in standard high-level languages. However, simdization is not trivial. Some of the difficulties in optimizing code for SIMD architectures stem from hardware constraints imposed by today’s SIMD architectures [20].

One such restrictive hardware feature that can significantly impact the effectiveness of simdization is the alignment constraint of SIMD memory units. In AltiVec [10], for example, a load instruction loads 16-byte contiguous memory from 16-byte aligned memory, ignoring the least significant 4 bits of the memory address. The same applies to store instructions. Now consider a stream: given a stride-one memory reference in a loop, a memory stream corresponds to all the contiguous locations in memory addressed by that memory reference over the lifetime of the loop. The alignment constraint of SIMD memory units requires that streams involved in the same SIMD operation must have matching offsets.

Consider the following code fragment, where integer arrays a, b, and c are aligned.

\[
\text{for } (i = 0; i < 1000; i++) \text{ do} \\
\quad a[i] = b[i + 1] + c[i + 2]; \\
\text{end for}
\]
The above code includes a loop with misaligned references. It requires additional realignment operations to allow parallelization on SIMD architectures with alignment constraints. In particular, unless special care is taken, data involved in the same computation, i.e., \(a[i], b[i+1], c[i+2]\), will be relatively misaligned after being loaded to machine registers. To produce correct results, this data must be reorganized to reside in the same slots of their corresponding registers prior to performing any arithmetic computation.

The realigning of data in registers is achieved by shifting. Shifts are executed inside the loop and therefore affect the performance significantly. The problem is to automatically reorganize data streams in registers to satisfy the alignment requirements imposed by the hardware with a minimum number of shift executions. Prior research has focused primarily on vectorizing loops where all memory references are properly aligned. An important aspect of this problem, namely, the problem of minimizing the number of shifts for aligning a given expression has been studied only recently.

An alternative to performing shift operations at runtime is to modify the way the data is laid-out in memory. This alternative suffers from several limitations and drawbacks: it can accommodate only one preferred alignment, whereas multiple computations may exist that do not agree on a common preference; the data may be allocated and accessed at various places, requiring whole-program analysis and optimization to optimize its layout effectively; and finally, changing the layout increases the memory consumption. In this paper the initial data alignment is assumed to be predetermined.

### 1.1 Technique used

In this work we investigate the computational complexity of the SIMD ALIGNMENT problem. A formal definition of the problem, motivation, and examples from current modern platforms appear in Chapter 3. The main contribution of this paper is the presentation of two new algorithms that solve the problem optimally for two special cases. These special cases are quite general and cover most practical instances.

**A polynomial-time algorithm for a single-alignment-appearance expression with two alignments.** For the special case, in which all arrays in the input expression appear in only (one out of) two different alignments, we provide an efficient algorithm that computes the optimal solution. The algorithm proposed employs an algorithm for the MINIMUM NODE S-T CUT problem as a subroutine.

**A polynomial-time algorithm for a single-appearance tree expression.** For the special case that the given expression is a tree and each array appears only once in the expression, we provide an efficient algorithm that solves the SIMD ALIGNMENT problem optimally. The algorithm uses dynamic programming to find the shifting schedule that yields the minimum number of shifts required to compute the expression.

We stress that both cases are realistic and are common in practice. Also, the two cases do not contain one another: one case is broader in the sense that it works on any expression, not necessarily a tree, and the other case is broader in the sense that it applies to an arbitrary number of alignments.
1.2 Organization

In Chapter 3 we define the SIMD ALIGNMENT problem, provide motivation from real platforms, and make a few observations. In Chapter 4 we list useful heuristics proposed in the literature so far and demonstrate how they may fail. In Chapter 5 we propose a graph representation of the SIMD ALIGNMENT problem, which will be used by the algorithms. In Chapter 6 we present the efficient algorithm for the special case of expressions with only two alignments. In Chapter 7 we present the algorithm for the special case of single-appearance tree expressions. In Chapter 9 we present the effectiveness of the algorithms in Chapters 6 and 7. We relate our work to prior art in Chapter 2 and present our conclusions and future work in Chapter 10.
Chapter 2

Related Work

There are two general approaches for generating optimized code targeting SIMD architectures: the classical loop-based vectorization scheme [2], and the extraction of parallelism from straight-line (or non loop-based) code [15, 21]. Our scheme applies generally to the SIMDization of any expression, although due to the overheads associated with shifts it is more relevant to expressions that reside in loops as in the loop-based scheme.

The work that is most closely related to ours is [13], which presents a set of heuristics for placing shifts in given expressions. These heuristics are described in detail in Chapter 4. The original paper also studies the details of implementation. The study there is partitioned into first finding the shift schedule and then generating the relevant code.

The shift schedule which is the output of our algorithms can be used to replace the first part of their study and feed their code generation to obtain a more efficient code. A subsequent work [24] extends some of the heuristics first presented in [13] to handle runtime alignment and alignment in the presence of length conversion operations. Our quest of finding optimal solutions was not considered.

Several compilers including VAST [22], GCC [18], compilers for VIS [8] and SSE2 [3, 4] provide re-alignment support, using the Zero-Shift heuristic, shifting all arrays to alignment zero before each operation. Our work can be used to further improve the code generated by such compilers. Some SIMD architectures provide re-alignment capabilities without requiring explicit shift operations [19]. Such architectures do not suffer from the SIMD ALIGNMENT problem. Additional related work concentrates on detection of misalignment and techniques to increase the number of aligned accesses [16]. Our work deals with minimizing the number of shifts given a set of misaligned accesses, and is complementary to these techniques.

In [12, 6, 14, 7] an interesting similar, yet different, problem is considered. They consider the efficient distribution of data to a set of distributed processors so that the communication required to compute the given program expressions is minimized. The distribution of array elements is restricted to affine transformations. There is a cost for communication if during an operation a processor has to access array cells that are not included in the data distributed to this processor. On one hand, this problem generalizes ours as shifts are a special case of general communication, and putting array entries in subsequent locations in the memory is a special case of affine transformations on array entries. However, these works assume a severe restriction which makes their case very different than ours in practice. They assume that no copies are made so if data is moved, it
cannot be used in the original location. We assume that once a shift has been executed then the data stream can be used both in its shifted form and in its original form without paying any extra cost. The inability to use streams in this manner is crucial to several positive and negative results, and in particular the NP-Hardness proof in [12]. Furthermore, the ability to parallelize communication and computation costs is crucial to the NP-Hardness proof of [5] but is not relevant to our model. Thus, these hardness results do not hold with our problem. Another difference which is less crucial but should be noted when comparing the results is that we assume predetermined alignments of the arrays, whereas these papers assume that they can set the alignment of the involved arrays. This assumption always allows a no-shift solution to a single-appearance tree expression. Such a solution does not always exist in our formulation.
Chapter 3

An Overview of the SIMD ALIGNMENT Problem

We begin by defining the SIMD ALIGNMENT problem.

**Definition 1** The SIMD ALIGNMENT problem.

**Input:** An expression containing input operands, operations and output operands, with an alignment value for every input and output operand.

**Solution:** A specification of shifts for some input operands and operations, such that the inputs to each operation all have the same alignment values and the inputs to output operands have the desired alignment values.

**Cost:** The number of shifts in the solution.

Note that for each operation of the given expression, the solution may specify several shifts if the result of the operation is needed in different alignments, or it may specify no shifts at all if the result is needed only in the same alignment as its inputs. This applies to input operands as well — an input operand may be shifted before being used in an operation. However, a solution is feasible only if the inputs of each operation and output operand are properly aligned.

Note also that if an expression includes constant operands, their vectorization does not involve alignment restrictions and does not require shifts, so we do not consider them part of the expression. Similarly, if we concentrate on an expression in a loop, we disregard loop-invariant operands whose shifts can be placed outside the loop.

In some cases the original alignment of an operand can be set to any desired value (e.g. by padding a local array that is set to start at an aligned address). We treat such operands similar to sub-expressions — their alignment is not predefined, but rather determined by the algorithm according to an optimal placement of shifts.

This paper focuses on finding a feasible solution with a minimal number of shifts to a SIMD ALIGNMENT instance. We proceed with describing how the shift operation is used with real platforms. We discuss properties of these operations and relate the low level description to the abstract definition above.
To illustrate the operation of shifts for alignment, consider the example below dealing with arrays \(a\), \(b\) and \(c\) of 4 byte elements. Suppose that the architecture supports SIMD instructions that operate on 4 elements of 4 byte each (as is the case with Altivec and SSE). In order to vectorize this loop, one effectively unrolls the loop to create 4 copies of the loop body and then packs adjacent elements into vectors:

```plaintext
for (i = 0; i < 250; i++) do
  \(a[i : i + 3] = b[i + 1 : i + 4] + c[i + 2 : i + 5]\);
end for
```

However, most vector architectures support access to aligned memory efficiently, whereas access to unaligned memory is not supported or incurs a heavy penalty. In the example, if \(a\), \(b\) and \(c\) are properly aligned on 16 byte boundaries, \(a[i : i + 3]\) can be accessed efficiently but not so \(b[i + 1 : i + 4]\) and \(c[i + 2 : i + 5]\). To cope with this restriction, a pair of aligned accesses are used together with an instruction for extracting the desired elements. For example,

\[
\begin{align*}
t_1 &= b[i : i + 3]; \\
t_2 &= b[i + 4 : i + 7]; \\
t_3 &= \text{shift}(t_1, t_2, 1);
\end{align*}
\]

are used to extract \(b[i + 1 : i + 4]\) from \(b[i : i + 7]\). The third parameter of the \textit{shift} operation indicates the misalignment, or shift amount. In Altivec this can be implemented as follows, using the Altivec C API (as supported by GCC):

```plaintext
v0 = vec ld (b, 0); \\
v1 = vec ld (b, 16); \\
v2 = vec lvsl (b, 4); \\
v4 = vec perm (v0, v1, v2); // v4 now holds b[1 : 4].
```

Stores of vectors into misaligned addresses can be handled in a similar way: a pair of adjacent aligned vectors are loaded, modified by inserting the desired vector into the appropriate position, and stored back.

Continuing with the example, notice that in the next iteration we need to access the next 4 elements \(b[i + 5 : i + 8]\), etc. Having already loaded elements \(b[i + 4 : i + 7]\) in the current iteration, we reuse them in the next iteration as follows:

```plaintext
v0 = vec ld (r3, 0); // v0 = b[0 : 3] \\
sv = vec lvsl (r3, 4); \\
u0 = vec ld (r4, 0); // u0 = c[0 : 3] \\
us = vec lvsl (r4, 8);
for (i = 0; i < 250; i++) do
  v1 = vec ld (r3, 16); // v1 = b[i + 4 : i + 7]; \\
u1 = vec ld (r4, 16); // u1 = c[i + 4 : i + 7]; \\
v2 = vec perm (v0, v1, sv); \\
u2 = vec perm (u0, u1, su); \\
w0 = v2 + u2; \\
vec store (w0, r5, 0) \\
v0 = v1; \\
u0 = u1; \\
r3 = r3 + 16; \\
r4 = r4 + 16;
```

10
The example above demonstrates the prevalent technique for handling misaligned accesses — by shifting them to produce a vector register containing the desired elements. This treatment requires one shift operation per misaligned access inside the loop, and additional preparations of pre-loading and setting shift amounts in appropriate registers before the loop.

It may not be necessary, however, to shift every misaligned access independently. By considering the operands involved in each operation, Wu et al. [24] showed that fewer shifts could suffice — the important goal is to shift all operands that feed an operation into a common position, not necessarily the aligned position. To see this, consider the following example:

```
for (i = 0; i < 250; i++) do
  a[i : i + 3] = b[i + 1 : i + 4] + c[i + 1 : i + 4];
end for
```

Here, instead of shifting both \(b\) and \(c\) which have the same alignment value, we can compute the operation \((b + c)\) and shift its result to be stored properly aligned:

```
v0 = vec_ld(r3, 0); // v0 = b[0 : 3]
u0 = vec_ld(r4, 0); // u0 = c[0 : 3]
w0 = v0 + u0;
for (i = 0; i < 250; i++) do
  v1 = vec_ld(r3, 16); // v1 = b[i + 4 : i + 7];
u1 = vec_ld(r4, 16); // u1 = c[i + 4 : i + 7];
w1 = v1 + u1;
t = shift(w0, w1, 1);
vec_store(t, r5, 0)
v0 = v1;
u0 = u1;
w0 = w1;
r3 = r3 + 16;
r4 = r4 + 16;
r5 = r5 + 16;
end for
```

In any case, any shifting of data from one alignment value to another requires one shift operation and preliminary preparation of pre-loading and setting shift amount. For expressions that appear inside loops, this preparation can be placed before the loop and is therefore often tolerable, whereas the shift operations become part of the expression and must remain inside the loop. That is why our objective is to minimize the number of shifts.

Expressions in loops have always been natural candidates for SIMD parallelism, with increased performance improvements for loops with large counts and the ability to hide long latencies using the loop prologue and epilogue. Note that the SIMD ALIGNMENT problem is not restricted to loop-based simdization only, but applies to Superword-Level Parallelism as-well [15, 21].
Chapter 4

Previous Heuristics

Previous work has concentrated on identifying operations that can be vectorized assuming all operands are aligned. Several simple heuristics have been proposed to solve the alignment problem. In this chapter we shortly survey these heuristics and provide some worst-case (yet realistic) examples for each heuristic. A more detailed account of these methods and their implementation appears in [24].

- **The Zero-Shift Policy.** This policy shifts each misaligned load stream to offset zero, and shifts the store stream from offset zero to the alignment of the store address. This is the simplest policy and it is currently employed by the widespread GCC compiler [18]. The motivation is to obtain a feasible solution in the simplest manner. This heuristic might apply many redundant shifts, for example, the trivial case:

```c
for (i = 0; i < 1000; i++)
    a[i + 1] = b[i + 1] + c[i + 1];
end for
```

The code includes a loop with aligned references, therefore no shift is required, but the zero-shift heuristic applies three redundant shifts.

- **The Eager-Shift Policy.** This policy shifts each load stream to the alignment of the store. This clearly creates a feasible solution, but the cost is unclear. This heuristic might apply many redundant shifts, for example:

```c
for (i = 0; i < 1000; i++)
    a[i + 2] = b[i + 1] + c[i + 1] + d[i + 1] * e[i + 1];
end for
```

The right side of the assignment can be calculated without applying any shift at all, then, a single shift is required to store the result. However, the eager-shift heuristic applies four shifts, one for each input stream.

- **The Lazy-Shift Policy.** This policy applies the greedy policy by inserting a shift only at the point where it is required to make the next operation possible. The Lazy-Shift heuristic alone does not exactly determine which shifts are output. If several alternative shifts are possible, it does not specify which of them should be selected. Consider the following loop.
\begin{verbatim}
for (i = 0; i < 1000; i++) do
    a[i+1] = b[i+1] + c[i+2] * d[i+1];
end for
\end{verbatim}

The Lazy-Shift heuristic does not determine anything in this situation because no operation can be done without applying a shift as a first step. In this case, shifting \(c\) towards \(d\) is optimal, yielding one shift overall, but shifting \(d\) towards \(c\) implies two or three shift executions overall (depending on how the algorithm makes its second decision).

- **The Majority Policy.** This policy shifts each load stream to the majority of the alignments of the input and output streams, and shifts the store stream from the offset of the majority to the alignment of the store address. The majority heuristic might execute many redundant shifts. For example:

\begin{verbatim}
for (i = 0; i < 1000; i++) do
    a[i+2] = b[i+2] + c[i+2] + d[i+1] * e[i+1];
end for
\end{verbatim}

This expression can be calculated with a single shift, applied on the result of \(d[i+1] * e[i+1]\). But the majority heuristic would apply two shifts.
Chapter 5

An Abstraction of SIMD Alignment

In what follows, it will be useful to represent instances of the SIMD ALIGNMENT problem as graphs. We provide a graph representation for instances in which an array appears in the expression in one alignment only. The proposed representation may also be used in the general case, but for arrays appearing with multiple alignments the cost of the solution cannot be easily translated from graphs to expressions. We call an expression in which each array appears with one alignment only a single-alignment-appearance expression. Most of the techniques employed in this paper relate to the study of graph algorithms. The representation of a single-alignment-appearance expression as a directed graph is the standard representation of expressions as graphs, except for two modifications. First, we add alignment labels. Second, all equal array appearances are united in a single node. The nodes that represent the input and output streams are associated with alignment labels that signify the initial alignment that the arrays have in the expression, and the name of the array. Each operation is also represented by a graph node. The operation nodes are labelled with the operation they carry. The nodes for input streams of an operation are connected to the node of the operation by incoming edges. Consider the example below.

\[
\text{for } (i = 0; \ i < 1000; \ i++) \ do \\
\quad a[i+1] = b[i+1] + c[i+2] \ast d[i+1]; \\
\text{end for}
\]

The graph representation corresponding to the expression above forms a tree, as shown in Figure 5.1.

The tree in Figure 5.1 has three leaves, representing input vertices labelled 1(d), 2(c), 1(b) and one root node representing the output stream and labelled 1(a). The labels signify the initial alignments and the array names. The operation nodes are labelled with the operation they represent.
Consider a second example.

\[
\text{for } (i = 0; i < 1000; i++) \text{ do } \\
\quad a[i+3] = b[i+1] * c[i+2] + c[i+2] * d[i+1]; \\
\text{end for}
\]

The corresponding graph representation to the above expression is shown in Figure 5.2. This graph has three leaves (input vertices) labelled 1(d), 2(c), 1(b) and one root (output node) labelled 3(a). Note that the graph in this example is not a tree. It is a Directed Acyclic Graph (DAG).

\[
\text{for } (i = 0; i < 1000; i++) \text{ do } \\
\quad a[i+3] = b[i+1] * c[i+2] + c[i+2] * d[i+1]; \\
\text{end for}
\]

5.1 A solution to a graph representation of a single-alignment-appearance

An important property of the expression execution is that once we shift a stream, we can use the shifted stream repeatedly without paying more shifts. In addition, even if we shift a stream we can still use its original alignment. We consider this property in the solution representation and its cost definition.
A solution to the graph representation of a single-alignment-appearance SIMD ALIGN-MENT problem is a labelling of the nodes. The cost of a solution for a single node \( v \) is the number of different labels of its descendants that are also different from its own label. The cost of a solution for the graph \( G(V, E) \) is a summation of the costs for all nodes in \( V \). We claim that this cost of the graph solution is equal to the cost of the corresponding solution of the expression. We interpret the solution to the graph as shifting specifications for the expression execution as follows. Each operation is executed at the alignment that is the label of its corresponding node in the graph. A stream represented by node \( v \) should be shifted from the alignment represented by its label, to all the alignments of its descendants that have different labels. This specifies a valid execution of the expression because all operations have their input stream shifted to the same alignment. We need to show that the computed cost represents the minimal number of shifts required to execute the operations at the alignment specified by the graph solution. The expression is a single-alignment-appearance expression and therefore a shift must be done for a node if its descendants do not have the same label. If an array appears with more than one alignment in the expression, shifting it once could save shift to another use of this array. We do not allow using recognition of common subexpressions in order to save shifts. Therefore, the cost of the solution in the graph is exactly the number of shifts that should be executed in order to compute the expression with the alignments specified by the graph solution. In Figure 5.3 we show an example of a graph with a given solution.

![Graph Example](image)

Figure 5.3: A graph that exemplifies the cost of SIMD.

The labelling shown in Figure 5.3 costs only two shifts, because the descendants of \( v \) have only two different shift labels that are also different from its own label (2 and 3). Therefore, \( v \) should be shifted from alignment 1 to alignments 2 and 3, enabling the execution of the rest of the computation without any further shift.

We are now ready to define the problem SIMDG.

**Definition 2 (The SIMDG Problem)**

**Input:** \( (G, L) \) where \( G(V, E) \) is a DAG representation of a single-alignment-appearance expression and \( L \) is a set of predetermined shift labels for the leaves and the root.

**Solution:** a labelling \( c \) for all nodes, which is an extension of the given labelling \( L \).

**Cost function:** for a labelling \( c \) the cost is:

\[
\sum_{v \in V} |\{c(u) : \exists u \in D(v) \quad c(u) \neq c(v)\}|
\]

where \( D(v) \) is the set of the descendants of \( v \).

**Goal:** finding a solution with minimum cost.
From this point on we stick to the graph representation and consider the SIMDG problem rather than the original SIMD ALIGNMENT problem.
Chapter 6

A polynomial-time algorithm for expressions with only two alignments

In this chapter we present a polynomial-time algorithm for a restricted SIMD ALIGNMENT problem. We restrict the expression to be a SIMDG expression that contains only two distinct predetermined alignments associated with the input and output operands. Note that such restricted cases can appear in practice, when there are only two possible alignment values (0 and 1, e.g. when vectorizing for pairs of elements) or when more than 2 alignment values exist but all input and output operands are confined to two values (not necessarily 0 and 1).

An important (and problematic) property of the expression execution is that once we shift a stream, we can use the shifted stream repeatedly without paying more shifts. In addition, even if we shift a stream we can still use its original alignment. Here is an example in which a shifted stream is used thrice (but executed only once), and also used in its original alignment:

\[ f[i] = (a[i] * b[i+1]) + (a[i] * c[i+1]) + (a[i] * d[i+1]) + (a[i] * e[i]); \]

This expression can be computed using only two shifts. The first shift is applied to the stream \( a \), from 0 to 1. The shifted stream can then be used thrice to compute the subexpressions \( (a[i] * b[i+1]) \), \( (a[i] * c[i+1]) \), and \( (a[i] * d[i+1]) \). The original stream \( a \) is then used in its original alignment to compute \( (a[i] * e[i]) \). A second shift is applied on the result of \( (a[i] * b[i+1]) + (a[i] * c[i+1]) + (a[i] * d[i+1]) \), from 1 to 0, and then the final result can be computed. Note that if we had three different arrays, instead of three appearances of the same array \( a \), we would have needed more shifts. Thus, an algorithm for solving the SIMD ALIGNMENT problem must notice such opportunities and exploit them, if possible. In Chapter 7 below we restrict the input instances to have only a single appearance of each array, and hence no common sub-expressions. This restriction enables us to solve the problem with more than two alignments.
Our algorithm uses a variant of the standard cut problem in graphs. In this variant, the cut is specified by nodes and not by edges. Let us first define a node cut in a graph.

**Definition 3 A node s-t cut** [1].

*Given a connected undirected graph \( G = (V, E) \) and two specified vertices \( s, t \in V \), for which \( (s, t) \notin E \), a node s-t cut is a subset of \( V \setminus \{s, t\} \) whose removal from the graph disconnects the vertices \( s \) and \( t \) from each other.*

We now define the minimum node s-t cut problem that we use to solve the variant of the SIMD ALIGNMENT problem restricted to two alignments. An optimal solution to the minimum node s-t cut problem can be constructed in polynomial time using a max-flow algorithm [1, 9].

**Definition 4 Minimum node s-t cut problem.**

*Input:* a connected, undirected graph \( G = (V, E) \) and two specified vertices \( s, t \in V \), for which \( (s, t) \notin E \).

*Problem:* find a node s-t cut with a minimum number of nodes.

Given an expression as an input to the SIMD ALIGNMENT problem, we represent it as a graph and then use an algorithm for minimum node s-t cut to construct a minimum node s-t cut, which is then used to provide a solution to our original problem in terms of minimum shifts. We start by making an observation on shifting a stream. Consider the code below.

```plaintext
for (i = 0; i < 1000; i++) do
    d[i + 1] = a[i] * b[i + 1] + a[i + 1] * c[i];
end for
```

This expression can be computed using two shifts: one applied to stream \( a \), from 0 to 1, and the other applied to \( c \), from 0 to 1. The observation we wish to make, is that even though stream \( a \) appears with both alignments in different places the expression, each appearance may require a shift to reach the other alignment. That is, \( a \) needs to be shifted from 0 to 1 independent of the appearance of \( a \) elsewhere with an original alignment of 1.

The algorithm’s pseudo-code appears in Algorithm 1. We start by considering the directed graph \( G = (V, E) \) that represents the SIMD ALIGNMENT expression. We denote the two alignments by 0 and 1 for clarity; the algorithm depends neither on the values of the alignments nor on the possible number of alignments. Next, we construct an undirected graph \( H \) by performing the following actions to the graph \( G \). First, each pair of nodes \( u \) and \( v \) that share a common successor node \( w \) \( ((u, w), (v, w) \in E) \) are connected by an edge \( (u, v) \), if not already connected. \(^1\) The direction of this \( (u, v) \) edge is immaterial, as we ignore the directions of the edges \( E \) from now on. We further add two “terminal” nodes \( s_0 \) and \( s_1 \) that will serve as the source and target nodes \( s \) and \( t \) for the minimum node s-t cut problem. An edge is added between \( s_0 \) and each node whose alignment label is predetermined to 0, and similarly to \( s_1 \). Denote by \( H \) the obtained graph.

Next, we find a node s-t cut \( C \) in \( H \). Finding a solution to the minimum node s-t cut problem is possible polynomial time via max flow algorithms [1, 9]. Denote by \( G' \)

\(^1\)Graphs with such additional edges are sometimes called *moral*, stressing that they “marry” the parents of each child. Of course, this ignores the fact that a parent may have more than one spouse.
the (undirected) graph obtained by removing the cut \( C \) from \( H \). By definition of a node s-t cut, \( s_0 \) and \( s_1 \) belong to \( G' \) and there is no path connecting \( s_0 \) and \( s_1 \) in \( G' \). Since the cut is in nodes, there may appear more than two connected components. All nodes in the component \( S_0 \) that contains \( s_0 \) are labelled 0 and all nodes in the connected component \( S_1 \) that contains \( s_1 \) are labelled 1. We now return to the original (directed) graph \( G \) and look at the remaining nodes that have not been labelled yet. We will show in Lemma 3 below that a node in the cut cannot have predecessors that are labelled inconsistently so far. (It will follow from the fact that each two parents are connected, i.e., that \( H \) is a moral graph.) We label each un-labelled node by the color of its predecessors. Finally, all remaining nodes are colored 0 (we could color them 1 as well).

**Algorithm 1** Solving an expression with two alignments

**Input:** a DAG \( G = (V, E) \) with some vertices having predetermined labels \( s(v) \).

**Output:** a labelling \( s(v) \) for all vertices.

1. Add two terminals \( s_0, s_1 \).
2. Add an edge \((s_0, v)\) to every node \( v \) with a predetermined alignment label 0.
3. Add an edge \((s_1, v)\) to every node \( v \) with a predetermined alignment label 1.
4. Add an edge \((u, w)\) for every pair \( u, w \) that have a common successor \( v \notin \{s_0, s_1\} \).

Let \( H \) be the undirected graph obtained by ignoring the directions of the edges in the resulting graph.

Find a minimum node s-t cut \( C \) in \( H \).

Let \( G' \) be the (undirected) graph obtained after removing the nodes of the cut \( C \) from \( H \).

Let \( S_0 \) be the set of nodes that are reachable by a path from \( s_0 \) in \( G' \).

Similarly, let \( S_1 \) be the set of nodes that are reachable from \( s_1 \) in \( G' \).

Define an initial labelling: label every node in \( S_0 \) with alignment label 0, and every node in \( S_1 \) with alignment label 1.

Let \( R = V \setminus (S_0 \cup S_1) \). The set \( R \) consists of \( C \) and the set of nodes that are isolated from both \( s_0 \) and \( s_1 \) in \( G' \).

Labelling nodes in \( R \): consider the original (directed) graph \( G \) and label each \( v \in R \) according to its predecessors. If \( v \) has a predecessor in \( S_0 \), then set its alignment label to 0. Similarly, if \( v \) has a predecessor in \( S_1 \), set its label to 1.

Final labelling: Label all remaining nodes (in \( R \)) with 0.

**Interpretation of the labels:** Shifts are provided for the nodes of the cut. A shift is applied to the result of the operation corresponding to the cut node (i.e. after the operation is executed).

### 6.1 An example

We now provide an example of an expression and how the algorithm operates on it. Then, we prove the algorithm’s correctness formally. Consider the following example:

```plaintext
for (i = 0; i < 1000; i++) do
    f[i] = (a[i + 1] * b[i + 1] + a[i + 1] * c[i + 1])
    + (a[i + 1] * d[i] + a[i + 1] * e[i]);
end for
```
This expression can be computed using a minimum of two shifts: one shift applies to array \( a \) from alignment 1 to alignment 0, serving the computation of \( a[i+1] \times d[i] \) and \( a[i+1] \times e[i] \). The other shift applies to the result of the subexpression \( (a[i+1] \times b[i+1] + a[i+1] \times c[i+1]) \), from alignment 1 to alignment 0.

The algorithm starts by considering the DAG \( G = (V, E) \) representing the expression as shown in Figure 6.1.

![Diagram of DAG](image)

Figure 6.1: The DAG corresponding to \( f[i] = (a[i+1] \times b[i+1] + a[i+1] \times c[i+1]) + (a[i+1] \times d[i] + a[i+1] \times e[i]) \):

Algorithm 1 continues by producing the graph \( H \) with two additional terminals \( s_0 \) and \( s_1 \), and connecting parents.

The corresponding graph \( H \) after step 4 of the algorithm is shown in Figure 6.2. Note, for example, that the edge \( (v_1, v_3) \) was added because \( v_1 \) and \( v_3 \) are both predecessors of \( v_6 \).

Next, a cut in \( H \) is found as shown in Figure 6.3. We mark the cut nodes in the following figure by encircling them with a bold line. (In this example, it is easy to see that a minimum cut is of 2 nodes and is unique). The sets \( S_0 \) and \( S_1 \) are also marked.

Finally, we interpret the cut nodes as creating shifts in the computation as follows. Node \( v_3 \) (which initially represented the input array \( a \) with alignment label 1) is a cut node. Therefore, the array \( a \) is shifted from alignment 1 to alignment 0. After this shift is executed, the subexpressions \( a[i+1] \times d[i] \) (node \( v_8 \)) and \( a[i+1] \times e[i] \) (node \( v_9 \)) can both be computed with alignment 0. Array \( a \) will be used with its original alignment 1 to compute \( a[i+1] \times b[i+1] \) (node \( v_6 \)) and \( a[i+1] \times c[i+1] \) (node \( v_7 \)). If we look at the graph interpretation, the shift operation affected a subset of \( v_3 \)'s successors but not all of them. As previously stressed, even though the array \( a \) is shifted, it may still be used with its original alignment. Node \( v_{10} \) is the other cut node, therefore, the result of the computation \( a[i+1] \times b[i+1] + a[i+1] \times c[i+1] \), is shifted from alignment 1 to alignment 0, enabling the
execution of the final computation (node $v_{12}$) in alignment 0 and completing the expression by storing its result (node $v_{13}$) without any further shifts.

### 6.2 Correctness

To formally prove the algorithm, we start by rigorously stating how an alignment labelling of the graph is interpreted as an execution of the represented expression.

**Definition 5** A labelling $L : V \rightarrow \{0, 1\}$ of an expression (directed) graph $G(V, E)$ is proper with respect to a subset $C \subseteq V$ of the nodes if for each edge $(v \rightarrow w) \in E$, either $L(v) = L(w)$ or $v \in C$ (or both).

To compute the expression that is represented by a graph $G(V, E)$, and a proper labelling $L : V \rightarrow \{0, 1\}$ with respect to a subset $C$ of the nodes, we apply a shift to each node in $C$. We need to show that this allows computing the expression, i.e., that the inputs to any operation are consistently aligned. Since the labelling is proper, then for each node $v$, each predecessor of $v$ either has the same alignment as $v$, or it is shifted before $v$ is executed. A predecessor that has the same alignment as $v$ creates no problem in the computation, even if it is shifted, because we can use the original stream ignoring the shift. On the other hand, if a predecessor has an alignment label that is different from $v$, then it is in the set $C$ and it is shifted. Since there are two alignments, the shifted stream has the same label as $v$. 
Figure 6.3: A minimum node $s_0 - s_1$ cut for the graph in Figure 6.2.

חיתוך בצומתים שמתאימים לכרך 6.2 $s_0 - s_1$.
Thus the computation is possible. Furthermore, the number of shifts required to compute the expression is exactly the number of nodes in the set $C$.

We will formally show that, on one hand, any execution of the expression must have at least as many shifts as the size of an $s_0 - s_1$ node cut in $H$ (as defined in Algorithm 1). We will further show that, on the other hand, the output of the algorithm is a proper labelling of the expression graph with the set $C$ being a minimum $s_0 - s_1$ node cut in the graph $H$. We will then conclude that the algorithm outputs an optimal solution to the SIMD alignment problem, when restricted to two alignments.

Lemma 1 The minimum number of shifts required to compute the given expression is at-least the size of a minimum node cut that disconnects $s_0$ and $s_1$ in the graph $H$, as defined in Step (5) of Algorithm 1.

Proof: We show that any proper execution of the expression implies a proper alignment labelling of the expression graph, and that any proper alignment labelling of the expression graph implies a node cut in $H$, where the number of nodes in the cut equals the number of shifts executed. First, consider a proper computation of the expression. This means that the inputs to any operation has the same alignment. Label each node by the alignment of its inputs (and label input nodes according to their initial labelling). Let the set $C$ contain all nodes whose output is shifted in the computation. Clearly, this yields a proper labelling of the graph $G$ with respect to $C$.

Now, given the above proper labelling of the nodes in $G$ with respect to the set $C$, we would like to show that $C$ is an $s$-$t$ node cut of the graph $H$. This means that the number of shifts in the given computation equals the size of a node cut $C$ of $H$. Consider any (undirected) path from $s_0$ to $s_1$ in $H$. At some point, the path must cross an edge $(u, v)$ such that $u$ is labelled 0 and $v$ is labelled 1. We would like to show that one of these nodes is in $C$. If $(u \rightarrow v)$ or $(v \rightarrow u)$ is an edge in $G$, then we are done, since, by the definition of $C$, $u \in C$ or $v \in C$ and this path breaks when $C$ is removed from $H$. Otherwise, $(u, v)$ is not an edge in $G$ and it is an edge that was added to $H$ because it connects two predecessors of a common node $w$ in $G$. However, the label of $w$ must be different from either $u$ or $v$, and therefore, by definition of $C$ either $v \in C$ or $u \in C$ and we are done. □

It remains to show that Algorithm 1 produces a proper alignment labelling of the graph $G$ (and $H$) with respect to a minimum $s_0 - s_1$ node cut $C$ of $H$. The set $C$ will be the one computed in Step (6) of the algorithm. By definition it is a minimum $s_0 - s_1$ node cut of $H$. It remains to show that the labelling is proper. This is done in the rest of this section. We start with a disjointness property of the sets created in the algorithm.

Lemma 2 The sets $S_0$, $S_1$ and $R \setminus C$, as defined in Algorithm 1 are disconnected sets in the graph $H$, in the sense that there is no edge in $H$ connecting a vertex in one set to a vertex in another.

Proof: We first establish the fact that $S_0$ and $S_1$ are disjoint, otherwise the lemma trivially breaks. If there exists a node $v$ that is both in $S_0$ and in $S_1$, then the nodes $s_0$ and $s_1$ are not disconnected in $G'$, contradicting the fact that we have removed an $s_0$-$s_1$ node cut from $H$ to create $G'$. Now, given that $S_0$ and $S_1$ are disjoint, suppose that there exists an edge $(v, u)$ in $H$ which connects $u \in S_0$ to $v \in S_1$. By the definition of $S_0$ and $S_1$, the nodes $u$
and \( v \) are not in the cut \( C \). Since \( G' \) contains all \( S_0 \) and \( S_1 \), the nodes \( s_0 \) and \( s_1 \) are not disconnected in \( G' \): the node \( u \) is reachable from \( s_0 \) in \( G' \), and a path can continue from \( u \) to \( s_1 \) via the edge \( (u,v) \) in \( G' \). This, again, contradicts the fact that we have removed an \( s_0-s_1 \) node cut from \( H \) to create \( G' \). Thus, \( S_0 \) and \( S_1 \) are disconnected from each other in \( H \).

By definition of \( R \), it is disjoint from \( S_0 \) and \( S_1 \). We now show that \( R \setminus C \) is disconnected from both \( S_0 \) and \( S_1 \). W.l.o.g., consider a node \( v \) in \( S_0 \) and a node \( u \) in \( R \setminus C \) and suppose, by way of contradiction, that \( (u,v) \) is an edge in \( H \). Since both \( u \) and \( v \) are not in \( C \), then they both exist in \( G' \). But if \( v \) is reachable from \( s_0 \) in \( G' \), then so is \( u \), contradicting the assumption that \( u \) is not in \( S_0 \). \( \Box \)

Next, we assert that a cut node does not have predecessors in both \( S_0 \) and \( S_1 \). Recall that any discussion on successors or predecessors corresponds to the original directed graph \( G \).

**Lemma 3** Let \( S_0 \) and \( S_1 \) be the sets defined in Algorithm 1, then a node \( w \) in the cut \( C \) cannot have one predecessor in \( S_0 \) and another predecessor in \( S_1 \).

**Proof:** By the definition of the graph \( H \), if \( w \) is a node in \( G \) and the nodes \( u \) and \( v \) are predecessors of \( w \), then an edge \( (u,v) \) exists in \( H \) — this edge is added to \( H \) is Step (4) of Algorithm 1. Therefore if \( v \) is in \( S_0 \) and \( u \) is in \( S_1 \) (or vice versa), then \( s_0 \) and \( s_1 \) are not disconnected by removing the nodes of \( C \) from \( H \), contradicting the fact that \( C \) is an \( s_0-s_1 \) node cut of \( H \). \( \Box \)

We continue with showing that the output labelling is proper with respect to \( C \). Namely, for each node \( w \), each predecessor of \( w \) is either in \( C \), or has the same alignment label as \( w \). This property trivially holds for nodes in \( S_0 \) or \( S_1 \) because they are only connected to nodes with the same label or to cut nodes. Next, we show this holds for nodes that belong to the cut \( C \).

**Lemma 4** For any cut node \( w \in C \), \( w \) is given the same label as all its predecessors that are not in \( C \).

**Proof:** If \( w \) has a predecessor \( u \in S_0 \) (or in \( S_1 \), resp.), it is labelled in Step (12) with 0 (or 1, resp.). By Lemma 3, all its predecessors that are not in \( R \) must be labelled consistently with it. We next note that \( w \) cannot have a parent in \( R \setminus C \). Suppose in way of contradiction that \( p \in R \setminus C \) is a predecessor of \( w \). Since both \( u \) and \( p \) are predecessors of \( w \), then an edge \( (u,p) \) must exist in \( H \) (added in Step (4) of Algorithm 1). Such an edge contradicts Lemma 2.

Otherwise, \( w \) has no predecessors in \( S_0 \) or \( S_1 \). This means that it gets the 0 alignment label in Step (13) of the algorithm. Also, all its predecessors that are not in \( C \) must be in \( R \setminus C \). By Lemma 2, all nodes in \( R \setminus C \) cannot have a predecessor in \( S_0 \) or \( S_1 \) and so all the nodes in \( R \setminus C \) are also given the 0 label in Step (13) and we are done. \( \Box \)

To show that the labelling output by Algorithm 1 is proper with respect to \( C \), we need to assert that each node \( v \notin C \) has the same alignment as all its successors. This is shown in the following lemma.
Lemma 5  In the output labelling of Algorithm 1 each node \( v \notin C \) has the same alignment label as all its successors.

Proof: Let \( v \notin C \). If \( v \in S_0 \) (or \( S_1 \)) then all its successors are labelled 0 (or 1). This is clear for a successor in \( S_0 \) (or \( S_1 \), resp.). Otherwise, the successor is a cut node and it is labelled 0 (or 1, resp.) in Step (12) of Algorithm 1. Finally, by Lemma 2 successors of \( v \) cannot be in \( R \setminus C \).

Otherwise, \( v \) is in \( R \setminus C \) and then, by Lemma 2, its successors must belong to \( R \). We claim that all of them are labelled 0 by Algorithm 1. Suppose, by way of contradiction, a predecessor \( p \) labelled 0 in Step (13). Node \( c \in C \) can only be labelled 1 in step (12), because it has a predecessor \( p \in S_1 \). This means that both \( p \) and \( v \) are predecessors of \( c \) in the original graph \( G \). Therefore, in the graph \( H \) there must be an edge between the node \( v \in R \setminus C \) (added in Step (4) of the algorithm) and the node \( p \in S_1 \). But such an edge contradicts Lemma 2. \( \Box \)

Corollary 6  The algorithm above computes a solution with the minimum number of shifts required to compute the expression.

Proof: By Lemma 5, the algorithm outputs a proper labelling of the expression graph \( G \) with respect to a minimum cut \( C \) in \( H \). By Lemma 1, this implies the minimum number of shifts required to compute the input expression. \( \Box \)

6.3 Complexity Analysis

We now analyze the time complexity of Algorithm 1. Let \( |V| \) be the number of vertices of the graph and \( |E| \) be the number of edges. We examine each of the six phases.

Phase 1 (lines 2-3): In this phase the algorithm adds two nodes \( s_0 \) and \( s_1 \), and edges to connect these nodes to the input and output nodes. The complexity of this stage is \( O(|V|) \).

Phase 2 (line 4): In this phase the algorithm connects every pair of nodes \( u, w \) which have a common successor. The complexity of this stage is \( O(|V|^2) \).

Denote by \( |V|_H \) the number of nodes in the induced graph \( H \). After the first two phases \( |V|_H = |V| + 2 = O(|V|) \). Denote by \( |E|_H \) the number of edges in the induced graph \( H \). After the first two phases \( |E|_H = |E| + O(|V|^2) = O(|V|^2) \).

Phase 3 (line 6): The complexity of calculating the minimum node cut is \( O((|V|_H)^2 \cdot |E|_H) = O(|V|^2 \cdot |V|^2) = O(|V|^4) \) [9].

Phase 4 (lines 7-11): In this phase we label the nodes in \( S_0 \) and \( S_1 \). The labelling can be done by using BFS/DFS traversal. The complexity of this phase is \( O(|V|_H + |E|_H) = O(|V|^2) \) [9].

Phase 5 (lines 12): In this phase we label the cut nodes which have predecessors from \( S_0 \) or \( S_1 \). Each incoming edge is being examined at most once. Therefore, the complexity of this phase is \( O(|E|_H) = O(|V|^2) \).

Phase 6 (lines 13): In this phase we label the rest of the nodes in the set \( R \). The complexity of this phase is \( O(|V|_H) = O(|V|) \).

To sum up, the bottleneck is Phase 6, making the complexity of Algorithm 1 \( O(|V|^4) \).
Chapter 7

A polynomial-time algorithm for single-appearance tree expressions

In this chapter we deal with expressions whose number of different alignments is not restricted, but their graph representation form a tree and any array appears in the expression at most once. We show that the SIMD ALIGNMENT problem can be solved in polynomial-time using dynamic programming, when restricted to such single-appearance tree expressions.

In what follows, we consider the SIMD ALIGNMENT problem in its graph representation as defined in Chapter 5 and denote the input graph by $T = (V, E)$. The graph $T$ is a directed tree, with edges oriented from leaves to root. We further denote by $I$ the set of all predetermined alignment labels appearing in the leaves and the root of the given tree. We consider only solutions that restrict the labelling of the inner nodes to alignment labels in $I$. As discussed in Chapter 6, the optimal solution is among the considered solutions.

A solution to the graph representation of a SIMD ALIGNMENT problem is again an alignment labelling of the operation nodes in the graph, i.e., a complete alignment labelling of the entire graph. Such a solution of the graph can be translated into a solution for the SIMD ALIGNMENT associated instance in the following way. When an operation is labelled by alignment $d$, all its input streams that are not $d$-aligned are shifted to alignment $d$ before executing the operation. Clearly, any labelling of the graph represents a feasible solution (though not necessarily an optimal one). The cost of a solution is the number of shifts executed. In the graph representation, this translates to edges whose ends are not labelled consistently. If a node is labelled with a different label from its successor, then a shift must be executed before the successor can use the stream in its input. We call such an edge (that implies a shift) a shift edge. The cost of the solution is exactly the number of shift edges.

**Definition 6** We say that a graph edge is a shift edge, with respect to a given alignment labelling of a tree, if its two adjacent nodes do not have the same labels.

example the optimal SIMD ALIGNMENT solution applies one shift to array $c$ from alignment 2 to alignment 1. The corresponding optimal solution for the corresponding tree is the one that labels all the operation nodes with alignment label 1.
The dynamic programming algorithm computes incremental solutions to the problem by considering larger and larger subtrees. The optimal solution of a subtree is computed using the values computed for its immediate subtrees. In particular, let \( v \) be a node in the tree \( T \) and consider the subtree of \( T \) for which \( v \) is the root. For each possible alignment \( i \in I \), denote by \( OPT_T(v, i) \) the minimum number of shift edges required to label the subtree rooted at \( v \), such that the resulting alignment label of the node \( v \) is \( i \). Note that the optimal labelling and the corresponding cost may be different for different \( i \)'s in \( I \).

The dynamic programming algorithm computes the entries of a matrix, \( \text{val} \), with an entry \( \text{val}(v, i) \) for each node \( v \) and each possible shift \( i \in I \). For each node \( v \), the entry \( \text{val}(v, i) \) represents a partial solution. It will later be shown that the output numbers \( \text{val}(v, i) \) equal \( OPT_T(v, i) \) for the expression tree \( T \). After computing the values \( \text{val}(v, i) \) for all nodes \( v \) and alignments \( i \), the algorithm uses the computed values to label the tree optimally.

Recall that the tree representing the expression has leaves representing the input operands and a root representing the output. The direction of the edges are from the leaves to the root. We start by setting up (any) numbering of the nodes from the root to the leaves so that for each two nodes numbered \( i \) and \( j \), if \((i \rightarrow j) \in E\) then \( j < i \). A BFS traversal or any pre-order traversal from the root to the leaves (traversing edges in reverse direction) may be used to obtain such numbering.

The algorithm’s pseudo-code appears in Algorithm 2. It starts by filling the matrix \( \text{val}(v, i) \) from the leaves to the root, going over the vertices from \( n \) to 1 according to their numbers in reverse order. At the base of this computation, we have the nodes at the leaves, for which a predetermined alignment label is provided. For each such leaf \( v \) we set the value of \( \text{val}(v, i) \) to be 0 if \( i \) is the predetermined alignment of the node \( v \) and \( \infty \) otherwise. Then, the inner nodes of the tree are traversed according to their numbering in reversed order. This order guarantees that a node is traversed only after its predecessors in the input tree \( G \) have been traversed. The value of \( \text{val}(v, i) \) for a node \( v \) with label \( i \), is computed locally by looking at its incoming edges and checking the costs incurred by all these edges. In particular, to compute \( \text{val}(v, i) \), we sum over all predecessor nodes \( u \) with edges \((u \rightarrow v) \in E\). For each such \( u \), we take the minimum, over the alignment values \( j \in I \), of \( \text{val}(u, j) + I_{i \neq j} \), where \( I_{i \neq j} \) is an indicator variable which equals 1 if \( i \neq j \), and 0 otherwise. Last, the algorithm computes the entries \( \text{val}(\text{root}, i) \). These values are pre-determined by the input labelling to the SIMD ALIGNMENT problem (because the root represents the output stream). Therefore, we set the value of \( \text{val}(\text{root}, i) \) to be

\[
\sum_{u: (u \rightarrow v) \in E} \min_j \{\text{val}(u, j) + I_{i \neq j}\} \text{ if } i \text{ is the predetermined alignment of the node } v \text{ and } \infty \text{ otherwise.}
\]

We will show in Lemma 7 below that this process computes \( \text{val}(v, i) \) so that \( \text{val}(v, i) = OPT_T(v, i) \).

After traversing the whole tree from the leaves to the root, the algorithm chooses the alignment value of the root to be its predetermined alignment determined as by the given labelling to the SIMD ALIGNMENT problem. If this alignment is \( i \), then the cost of the solution will be \( \text{val}(\text{root}, i) \).

Then, the tree is traversed backwards from the root to the leaves in order to label all inner vertices in a consistent manner that matches the minimum number of shift edges. For this process to work correctly, we must assume that each vertex has a single outgoing edge. This is true because the underlying graph is a tree and each array appears in the expression.
at most once. Given that each vertex has only one outgoing edge, the labelling process is doable using the information stored in \( \text{val}(v, i) \), as shown in Lemma 8. The labelling obtained in this manner is an optimal labelling of the tree, as shown in Lemma 9.

Algorithm 2 Solving a Single-Appearance Tree expression.

**Input:** a tree \( G(V, E) \) with the leaves and root having predetermined labels \( s_v \).

**Output:** a labelling \( s(v) \) for all vertices.

Run a BFS algorithm from the root towards the leaves and number all vertices according to the traversal.

```plaintext
for node \( v = n \) to 2 do
  if input node \( v \) has a predetermined alignment label \( s_v \) then
    for all \( i \in I \) do
      \( \text{val}(v, i) = \begin{cases} 
        0 & i = s_v \\
        \infty & i \neq s_v 
      \end{cases} \)
    end for
  else
    // Node \( i \) has no predetermined alignment label
    \( \text{val}(v, i) = \sum_{u:(u \rightarrow v) \in E} \min\{\text{val}(u, j) + I\}_{j \neq i} \} \)
    Where \( I_{i \neq j} \) equals 1 if \( i \neq j \) and 0 otherwise
  end if
end for

// A special treatment for the root (output) node \( v \) which
// has a predetermined alignment label \( s_v \) and number 1:
\( \text{val}(1, i) = \begin{cases} 
  \sum_{u:(u \rightarrow v) \in E} \min\{\text{val}(u, j) + I_{i \neq j} \} & i = s_v \\
  \infty & i \neq s_v 
\end{cases} \)
\( s(1) = \text{val}(1, s_1) \)

for node \( u = 2 \) to \( n \) do
  Let \( v \) be the unique node such that \((u \rightarrow v) \in E\).
  \( s(u) = \arg\min_j \{\text{val}(u, j) + I_{s(v) \neq j} \} \)
  where \( \arg\min_j \) is an index \( j \) for which the value \( \text{val}(u, j) + I_{s(v) \neq j} \) is minimal.
end for
```

7.1 An example

We now provide an example of an expression and how the algorithm operates on it. Then, we prove the algorithm’s correctness formally. Consider, for example, the following expression:

```plaintext
for \( (i = 0; i < 1000; i++) \) do
  \( a[i+2] = b[i+1] \ast c[i+2] + d[i+2] \ast e[i+3] \);
end for
```

In this expression each array appears only once and its graph representation forms a tree. It is therefore adequate for Algorithm 2. The corresponding tree for this expression is shown in Figure 7.1.
The tree has four input nodes labelled 1(b), 2(c), 2(d) and 3(e), and one output node labelled 2(a). The algorithm computes the values \( \text{val}(v, i) \) for each node \( v \) and for each alignment \( i \) as described in Figure 7.2.

The alignment to the root is set to the predetermined alignment label. In the example above the root alignment is 2. The cost of the solution will be \( \text{val}(1, 2) \) which is 2 in this case. Then, the tree is traversed backwards from the root to the leaves in order to label all
inner vertices, i.e., to produce the solution. All the values that were chosen by the algorithm are marked by a bold line in Figure 7.3. The output labelling is written on the nodes. The shift edges, that are induced from the algorithm choices, are also marked by a bold line.

![Figure 7.3: Output labelling and implied shift edges for the tree in Figure 7.1.](image)

In the solution of labelling the tree in the example above, there are two shift edges. One of them is the edge between array \( b \), which is labelled with alignment label 1, and the multiplication operation node \( v_3 \), which is labelled with alignment label 2, and the other is the edge between the array \( c \), which is labelled with alignment label 3, and the multiplication node \( v_4 \), which is labelled with label 2. Therefore, array \( b \) should be shifted from alignment 1 to alignment 2, and array \( e \) should be shifted from alignment 3 to alignment 2, enabling the execution of the rest of the computation in alignment 2 without any further shift.

### 7.2 Correctness

We now prove the algorithm’s correctness. We start by asserting that the values of the matrix \( \text{val} \) represent \( \text{OPT}(v, i) \) as desired.

**Lemma 7** For each node \( v \) in the tree and for each possible alignment \( i \in I \), the value \( \text{val}(v, i) \) that the algorithm computes is the minimum number of shift edges required to label the subtree of \( v \) conditioned on the alignment labelling of \( v \) being \( i \), i.e., \( \text{val}(v,i)=\text{OPT}(v,i) \).

**Proof:** We prove the lemma by induction on the nodes, ordered by their numbers in descending order. Namely, from the leaves to the root.
**Basis:** the leaf \( \ell = n \) and any other leaf node is a pre-labelled node and the lemma trivially holds by the initialization process. The value \( \text{val}(\ell, i) \) is set to 0 for the alignment label and to \( \infty \) for any other alignment.

**Induction Step:** consider a node \( \ell \) that is not a leaf. By the induction hypothesis, the lemma holds for all nodes \( k = n, n - 1, \ldots, \ell + 1 \). By the numbering property, every predecessor, \( p \), of \( \ell \) satisfies \( p > \ell \), and thus, \( \text{val}(p, i) = \text{OPT}(p, i) \) for all \( i \in I \).

We next relate the value of \( \text{OPT}(\ell, i) \) to the values of \( \text{OPT}(p, j) \) for its predecessors \( p \) and all possible alignments \( j \in I \). First, since we are labelling a vertex \( v \) with alignment \( i \), we add one shift edge for each predecessor of \( v \) that is not labelled with \( i \). Second, since the input graph is a tree, each predecessor’s subtree is disjoint from the other subtrees and the cost (number of shifts) for each subtree of a predecessor is independent of the labelling of the subtrees of the other predecessors. Therefore, to find \( \text{OPT}(v, i) \), we need to sum over the cost of each predecessor. Consider a predecessor \( p \). Its contribution to the cost of setting the label of \( \ell \) to \( i \) is either \( \text{OPT}(p, i) \) or \( \text{OPT}(p, j) + 1 \) for \( j \neq i \). The addition of 1 to \( \text{OPT}(p, j) \) is required since, if we set the label of \( p \) to \( j \neq i \) and the label of \( \ell \) to \( i \), then the edge \( (p \rightarrow \ell) \) becomes a shift edge. Selecting the smallest such expression yields the optimum for that subtree. Of course, if we set the label of \( p \) to \( i \), then no such shift edge is created. Summing up,

\[
\text{OPT}(\ell, i) = \sum_{p : (p \rightarrow \ell) \in E} \min_j \{\text{OPT}(p, j) + I_{i \neq j}\}
\]

where the second equality is correct because of the induction hypothesis and the final equality stems from the way \( \text{val}(\ell, i) \) is computed by the algorithm.

We now claim that the labelling output by Algorithm 2 is a labelling with cost \( \text{OPT}(\text{root}, i) \), where \( i \) is the predetermined label of the root. By the definition of \( \text{OPT}(\text{root}, i) \), this labelling is optimal. By Lemma 7, we need to show that the labelling output by Algorithm 2 bears a cost of \( \text{val}(\text{root}, i) \). This is asserted in the following lemma.

**Lemma 8** Algorithm 2 outputs a labelling for which if node \( v \) is labelled \( i \), then the cost (number of shift edges) of the subtree rooted at \( v \) is \( \text{val}(v, i) \) as computed by Algorithm 2 in the first loop.

**Proof:** We prove the lemma by induction on the nodes, ordered by their numbers in descending order (from the leaves to the root).

**Basis:** the leaf \( \ell = n \) and any other leaf node is a pre-labelled node, the algorithm labelled \( \ell \) with its predetermined label, which costs 0, and the lemma trivially holds.

**Induction Step:** consider a node \( \ell \) that is not a leaf. By the numbering property, every predecessor, \( p \), of \( \ell \) satisfies \( p > \ell \). By the induction hypothesis, the labelling output by the algorithm to all vertices \( n, n - 1, \ldots, \ell + 1, \ell \) satisfies the lemma. This holds for all predecessors of \( \ell \). Suppose the algorithm labels node \( \ell \) with \( i \). By the specification of the algorithm, each predecessor \( p \) is labelled with an index \( s(p) = j \) for which the value \( \text{val}(p, j) + I_{i \neq j} \) is minimal. Therefore, the cost for the subtree rooted at \( v \), with respect to the labelling of the algorithm, is
\[
\sum_{p : (p \rightarrow v) \in E} \text{val}(p, s(p)) + \sum_{p : (p \rightarrow v) \in E} I_{s(p) \neq i} = \\
\sum_{p : (p \rightarrow v) \in E} (\text{val}(p, s(p)) + I_{s(p) \neq i}) = \text{val}(v, i)
\]

\[\square\]

**Corollary 9** Algorithm 2 yields an optimal solution to single-appearance tree instances of the SIMD ALIGNMENT problem.

**Proof:** By the definition of \(OPT\), the cost of the optimal solution is \(OPT(root, i)\), where \(i\) is the predetermined labelling of the root node. By Lemma 7 \(OPT(root, i) = \text{val}(root, i)\). By Lemma 8, the labelling output by Algorithm 2 has cost \(\text{val}(root, i)\), which is optimal. \(\square\)

### 7.3 Complexity Analysis

We now analyze the time complexity of Algorithm 2. Let \(|V|\) be the number of vertices in the graph. Since the graph is a tree, then \(|E| = |V| - 1\). Denote by \(k = |I|\) the number of different alignment labels in the input expression. We analyze the complexity of Algorithm 2 according to its three phases:

**Phase 1** (line 1): In this phase the algorithm sets up numbering of the nodes using BFS traversal. The complexity of BFS traversal is \(O(|V| + |E|) = O(|V|)\) [9].

**Phase 2** (lines 2-16): In this phase the algorithm computes the entries of the matrix \(\text{val}\). Each edge is examined exactly \(k\) times, for each possible shift \(i \in I\). In each examination the algorithm calculates the minimum value as shown in lines 9 and 15. The complexity for calculating the minimum in line 9 is \(O(k)\). Therefore, the total complexity for this phase is \(O(k^2|E|) = O(k^2|V|)\).

**Phase 3** (lines 17-21): The minimum value is calculated once for each edge. Therefore, the total complexity for this phase is \(O(k|E|) = O(k|V|)\).

To sum up, the complexity of the full algorithm is

\[O(|V|) + O(k^2|V|) + O(k|V|) = O(k^2|V|).\]
Chapter 8

**The MULTIWAY CUT and the NODE MULTIWAY CUT Problems**

We now compare the SIMDG alignment problem to the MULTIWAY CUT and the NODE MULTIWAY CUT problems. We show the relations and the differences between the SIMDG problem and these two known problems.

**Definition 7 (Multiway Cut)** Given an undirected graph $G(V,E)$ and a set of terminals $S = \{s_1, s_2, \ldots, s_k\} \subseteq V$, a multiway cut is a set of edges whose removal disconnects the terminals from each other. The Multiway Cut Problem is the problem of finding a multiway cut with minimum weight, where the weight is the sum over the weights of the cut-edges.

**Definition 8 (Node Multiway Cut)** Given an undirected connected graph $G(V,E)$ and a set of terminals $S = \{s_1, s_2, \ldots, s_k\} \subseteq V$, a node multiway cut is a subset of $V \setminus S$ whose removal disconnects the terminals from each other. The NODE MULTIWAY CUT Problem is the problem of finding a node multiway cut with minimum weight. Here the weight of the cut is the sum of the weights of the cut-nodes.

It is known that the MULTIWAY CUT and the NODE MULTIWAY CUT problems are NP-hard for any fixed $k \geq 3[23]$. 

Note that a MULTIWAY CUT induces a labelling $c : V \to \{1, \ldots, k\}$ in the following way. Consider the graph $G$ after removing the cut edges. For each $1 \leq i \leq k$, all nodes in the component that contains $s_i$ are labelled $i$. Any remaining node is labelled 1. The edges in the cut are exactly the edges that have one side labelled differently from the other side. We would like to minimize the number of these edges.

The NODE MULTIWAY CUT also induces a labelling $c : V \to \{1, \ldots, k\}$. Consider the graph $G$ after removing the cut edges. For each $1 \leq i \leq k$, all nodes in the component that contains $s_i$ are labelled $i$. Each cut node is labelled as one of its neighbors. All remaining nodes are labelled 1. For the above labelling, only cut nodes have neighbors with different labels than the cut node label. The problem is to minimize the number of the cut nodes.

A solution for the SIMDG problem is a labelling for all nodes. Only nodes that should be shifted have descendants with different labels than the label of the parent. The cost of
a solution for a single node \( v \) is the number of different labels of its descendants that are also different from its own label. The cost of a solution for the graph \( G \) is a summation of the costs for all nodes in \( V \).

We stress the difference between the above problems with an example:

![Diagram](image)

Figure 8.1: A graph that exemplifies the differences between the problems.

A minimal multiway cut for the undirected graph corresponding to the graph in Figure 8.1 includes three edges, when \( v \) is labelled 3. A minimal node multiway cut in the undirected graph corresponding to the above graph includes only one node, the node \( v \). A solution to the SIMDG problem costs only two shifts when \( v \) is labelled 1, \( v \) should be shifted to shift label 2 and to shift label 3.

The three problems coincide in a few special cases. Consider a SIMDG input graph \((G(V,E), L)\) with only two different shift labels in \( L \). We denote by \( H_G \) the undirected moral graph constructed from \( G \) in the following way. First, each pair of nodes that share a common successor node are connected by an edge, if not already connected. We further add two “terminal” nodes \( s_0 \) and \( s_1 \). An edge is added between \( s_0 \) and each node whose alignment label is predetermined to 0, and similarly to \( s_1 \). A labelling to \( V \) in \( G \) is induced from the NODE MULTIWAY CUT of \( H_G \) in the following way. All nodes in the component that contains \( s_i \) are labelled \( i \). Each cut node will be labelled by the label of its predecessors in \( G \). All remaining nodes are labelled 1. For the above special case, the NODE MULTIWAY CUT and the SIMDG problem coincide, the labelling induced from the NODE MULTIWAY CUT of \( H_G \) is a solution to the SIMDG problem in \( G \). We have used this fact in Chapter 6. When the problem’s corresponding graph is a tree with a single appearance of each array then the solution to all three problems have the same cost.

From the labelling induced from the problems above, we can see that solutions for the MULTIWAY CUT and NODE MULTIWAY CUT problems can be easily modified to become a feasible solution to the SIMDG problem but with a different cost. This is stated in the following lemmas.

Given a SIMDG instance \((G, L)\) we construct an undirected graph \( G' \) as follows. We unite all the nodes with predetermined shift label \( i \) in \( G \) to a terminal node \( s_i \) in \( G' \) for every shift label \( i \in L \). All other nodes in \( G \) are also nodes in \( G' \). The edges connected to the node with predetermined label \( i \) in \( G \) are connected to the terminal \( s_i \) in \( G' \). All other edges in \( G \) are also edges in \( G' \). Note that \( G' \) may contain parallel edges. We denote by \( S \) the set \( \{ s_i \mid i \in L \} \).
Lemma 10  For any instance \((G, L)\) of SIMDG problem:

\[
OPT_{SIMDG}(G, L) \leq OPT_{MULTIWAY \text{ CUT}} (G', S).
\]

**Proof:** Given a MULTIWAY CUT of \((G', S)\), the same labelling corresponding to the cut is a solution to the SIMDG problem of \((G, L)\). The MULTIWAY CUT of \((G', S)\) induces a labelling of the nodes in \(G'\) as explained above. Let us consider the same labelling for the nodes in \(G\). In \(G\), shifts should be executed only for the streams that have descendants with different labels, in which case the corresponding edge in \(G'\) is a cut edge.

Moreover, the cost of the cut of graph \(G'\) might be higher than the cost of corresponding solution to the SIMDG problem of \(G\). The cost of the cut is the number of edges in the cut. There might be more than one edge from a certain node to nodes with same labels. In this case we pay for each one of these edges. However, the cost of handling these edges in the SIMDG problem is only one, because a shift to such a stream can be executed only once and later be used without paying more. Therefore, the cost of any feasible MULTIWAY CUT solution of \(G'\) is an upper bound for the cost of the SIMDG corresponding solution of \(G\).

Given an SIMDG instance \((G, L)\). We denote by \(H_G\) the undirected moral graph constructed from \(G\) in the following way. First, each pair of nodes that share a common successor node are connected by an edge, if not already connected. We further add \(k\) “terminal” nodes \(s_i\) for \(1 \leq i \leq k\). An edge is added between \(s_i\) and each node whose alignment label is predetermined to \(i\).

Lemma 11  For any SIMDG instance \((G, L)\)

\[
OPT_{SIMDG}(G, L) \leq (k - 1) \cdot OPT_{NODE \text{ MULTIWAY \text{ CUT}}}(H_G, S).
\]

**Proof:** Given a NODE MULTIWAY CUT for the moral graph \(H_G\) induced from \(G\), the same labelling corresponding to the cut is a solution to the SIMDG problem of \(G\). A cut node is labelled with the label of the nodes which are its predecessors in \(G\) that are not cut nodes. Shifts should be executed only to the streams represented by cut nodes. Note that a node in \(G\) cannot have predecessors that are labelled inconsistently, unless at least one of them is a cut node, because \(H_G\) is a moral graph, so each two parents of the same node in \(G\) are connected in \(H_G\). Therefore, the computation can be executed according to the labelling induced from the cut and only cut nodes should be shifted. We shift a cut node to match its descendants labels.

However, the cost of the obtained solution to the SIMDG problem might be higher than the cost of the cut itself. A certain cut node might have descendants with more than one shift label, which means it should be shifted more than once in order to match the alignments of its descendants. Since \(k\) is the number of alignments in the graph, \(k - 1\) is an upper bound on the number of shifts a cut node might need. Therefore, the cost of the SIMDG solution for the graph \(G\) is bounded by \(k - 1\) times the cost of the NODE MULTIWAY CUT for the induced moral graph \(H_G\).

There are approximation algorithms for the MULTIWAY CUT and the NODE MULTIWAY CUT problems. In the following lemmas we show how to use them to get approximation algorithms to the SIMDG problem.

39
Lemma 12 For any instance \((G, L)\) of SIMDG problem:

\[
OPT_{\text{MULTIWAY CUT}} (G', S) \leq \text{deg}(G) \cdot OPT_{\text{SIMDG}}(G, L).
\]

Proof: Given an optimal solution for the instance \((G, L)\), consider the set of outgoing edges that leave the nodes that should be shifted according to this optimal solution. We claim that this set is a multiway cut for \(G'\). Consider the undirected graph \(G''\) corresponding to the graph \(G\) after removing this set of edges. For every \(i\), nodes labelled \(i\) are not reachable from other nodes with different labels in \(G''\). \(G''\) is similar to \(G'\) except for the edges removed and the united nodes with same predetermined labels. Thus, this set of edges is a multiway cut in \(G'\) too, which completes the proof. \(\square\)

Lemma 13 Every approximation algorithm to the MULTIWAY CUT problem with approximation ratio \(r\) is an approximation algorithm to the SIMDG problem with approximation ratio \(\text{deg}(G) \cdot r\) where \(\text{deg}(G)\) is the maximum degree of graph \(G\).

Proof: The output of the approximation algorithm for SIMDG on a graph \(G\) will be the induced labelling of the cut that is the output of the approximation algorithm for NODE MULTIWAY CUT on \(G'\). We claim that this suggested algorithm is a \(\text{deg}(G) \cdot r\) approximation algorithm, i.e \(\frac{\text{APPROX}_{\text{SIMDG}}(G, L)}{\text{OPT}_{\text{SIMDG}}(G, L)} \leq \text{deg}(G) \cdot r\) where \(\text{APPROX}_{\text{SIMDG}}(G, L)\) is the cost of the output of the approximation algorithm, and \(\text{OPT}_{\text{SIMDG}}(G, L)\) is the cost of an optimal solution to the SIMDG problem of \(G\). This claim follows from the inequalities:

\[
\begin{align*}
\text{APPROX}_{\text{SIMDG}}(G, L) &\leq \text{APPROX}_{\text{MULTIWAY CUT}} (G', S) & (8.1) \\
&\leq \text{OPT}_{\text{MULTIWAY CUT}} (G', S) \cdot r & (8.2) \\
&\leq \text{OPT}_{\text{SIMDG}}(G, L) \cdot \text{deg}(G) \cdot r. & (8.3)
\end{align*}
\]

For a given instance \((G, L)\), inequality 8.1 follows from Lemma 10. \(\text{APPROX}_{\text{MULTIWAY CUT}}\) is an approximation algorithm with approximation ratio \(r\) and therefore the inequality 8.2 holds. Inequality 8.3 follows from Lemma 12. \(\square\)

For the MULTIWAY CUT problem there exist a \((2-2/k)\)-approximation algorithm [23] and a \(3/2\)-approximation algorithm [23].

Corollary 14 Using the MULTIWAY CUT approximation algorithms and Lemma 13 yield approximation algorithms for the SIMDG problem. We obtain a \(\text{deg}(G) \cdot (2-2/k)\)-approximation algorithm and a \(\text{deg}(G) \cdot 3/2\)-approximation algorithm for the SIMDG problem.

We now obtain an approximation algorithm from the NODE MULTIWAY CUT approximation algorithm.

Lemma 15 For any instance \((G, L)\) of SIMDG problem:

\[
OPT_{\text{NODE MULTIWAY CUT}} (H_G, S) \leq OPT_{\text{SIMDG}}(G, L).
\]
Proof: Given an optimal solution for the instance \((G, L)\), consider the set of nodes that should be shifted according to this optimal solution. We claim that this set is a node multiway cut for \(H_G\). Consider the graph \(G''\) corresponding to the graph \(G\) after removing the nodes that have to be shifted according to the optimal solution. For every \(i\), nodes labelled \(i\) are not reachable from other nodes with different labels in \(G''\). Thus, this set of nodes is a node multiway cut in \(G''\). Now let us add to \(G''\) the edges between every two parents of the same node in \(G\). Every two parents of the same node in \(G\) must have the same label, or else one of them should have been shifted in order to compute their successor. So the set of removed nodes is still a NODE MULTIWAY CUT after the addition of these edges. Note that \(H_G\) is similar to \(G''\) except the cut nodes and the edges connected to them. Therefore, this set of nodes is a NODE MULTIWAY CUT in \(H_G\), which completes the proof. \(\Box\)

Lemma 16 Every approximation algorithm to the NODE MULTIWAY CUT problem with approximation ratio \(r\) is an approximation algorithm with approximation ratio \((k - 1) \cdot r\) for the SIMDG problem.

Proof: The output of the approximation algorithm for SIMDG on graph \(G\) will be the induced labelling of the NODE MULTIWAY CUT which is the output of the approximation algorithm on \(H_G\). We claim that this suggested algorithm is an \(r \cdot (k - 1)\) approximation algorithm, i.e \(\frac{\text{APPROX}_{\text{SIMDG}}(G, L)}{\text{OPT}_{\text{SIMDG}}(G, L)} \leq r \cdot (k - 1)\) where \(\text{APPROX}_{\text{SIMDG}}(G, L)\) is the cost of the output of the approximation algorithm, and \(\text{OPT}_{\text{SIMDG}}(G, L)\) is the cost of an optimal solution to the SIMDG problem of \(G\). This claim follows from the inequalities:

\[
\begin{align*}
\text{APPROX}_{\text{SIMDG}}(G, L) & \leq \text{APPROX}_{\text{NODE MULTIWAY CUT}}(H_G, L) \cdot (k - 1) \\
& \leq \text{OPT}_{\text{NODE MULTIWAY CUT}}(H_G, L) \cdot r \cdot (k - 1) \\
& \leq \text{OPT}_{\text{SIMDG}}(G, L) \cdot r \cdot (k - 1).
\end{align*}
\]

For a given instance \((G, L)\), inequality 8.4 follows from Lemma 11. APPROX\(_{\text{NODE MULTIWAY CUT}}\) is an approximation algorithm with approximation ratio \(r\) and therefore the inequality 8.5 holds. Inequality 8.6 follows from Lemma 15. \(\Box\)

For the NODE MULTIWAY CUT problem there exists a \((2-2/k)\)-approximation [23].

Corollary 17 Using the NODE MULTIWAY CUT approximation algorithm and Lemma 16, we obtain an approximation ratio of \((2 - 2/k) \cdot (k - 1)\) for the SIMD ALIGNMENT problem.
Chapter 9

Results

We have proven in Chapters 6 and 7 that the algorithms that we have proposed for the two special cases are optimal. In this chapter we demonstrate the practical advantage of using these algorithms compared to all the heuristics mentioned in Chapter 4.

9.1 Measurements for tree expression

The effectiveness of Algorithm 2 was tested on complete binary trees of various depths. Denote by \( k \) the number of possible different alignments in the tree. For each depth \( d \) we consider the full binary tree of depth \( d \) and randomly generate the alignments (shift labels) of the input vertices (the leaves) and the alignment of the output vertex (the root) in the range of 1 to \( k \). We also let the bound \( k \) range from 1 to 7.

We have run Algorithm 2 and all the heuristics on these randomly generated trees. Algorithm 2 always outputs an optimal solution and therefore the cost of its solution is at least as good as any of the solutions of the heuristics mentioned in Chapter 4. Table 9.1 summarizes our results of comparing Algorithm 2 to all the heuristics. In the table cell that represents depth \( d \) and at most \( k \) different alignments, we report the percentage of test-runs in which our algorithm performed strictly better than all the heuristics.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( d=3 )</th>
<th>( d=5 )</th>
<th>( d=8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>24.2%</td>
<td>94.6%</td>
<td>98.5%</td>
</tr>
<tr>
<td>3</td>
<td>28.4%</td>
<td>96.67%</td>
<td>98.8%</td>
</tr>
<tr>
<td>4</td>
<td>32.7%</td>
<td>95.37%</td>
<td>99.26%</td>
</tr>
<tr>
<td>5</td>
<td>27.4%</td>
<td>96.57%</td>
<td>98.1%</td>
</tr>
<tr>
<td>6</td>
<td>26.3%</td>
<td>94.74%</td>
<td>99%</td>
</tr>
<tr>
<td>7</td>
<td>29.2%</td>
<td>95.31%</td>
<td>98.68%</td>
</tr>
<tr>
<td>8</td>
<td>30.65%</td>
<td>96.43%</td>
<td>98.99%</td>
</tr>
</tbody>
</table>

Table 9.1: The percentage of test-runs in which algorithm 2 outperformed all the heuristics.
The table shows that as the size of the tree grows, the percentage of trees in which Algorithm 2 outperformed all of the heuristics grows rapidly. This result is probably due to the approach of the heuristics that does not consider the structure of the tree as opposed to our algorithm. The Eager-Shift heuristic and the Majority heuristic take into consideration only the shift labels of the leaves and root. The Lazy-Shift heuristic, considers only the labels of a node’s parents, before labelling a node.

9.1.1 An Example

In Figures 9.1 and 9.3 we show explicit examples of trees for which none of the heuristics achieve an optimal solution. These trees demonstrate how all heuristics can fail simultaneously, and provide some intuition as to why this happens.

For the tree in Figure 9.1, the Majority policy and the Eager-Shift policy heuristics output a solution composed of 4 shifts, both execute shifts to all the arrays with shift label 1. The Lazy-Shift policy does not specify a single solution in this case, but it may output a solution composed of as many as 5 shifts. However, as can be seen in Figure 9.2 only three shifts are needed in the optimal solution.

For the tree in Figure 9.3, the Majority policy heuristic outputs a solution composed of 6 shifts, shifting all the arrays to shift label 1. The Eager-Shift policy heuristic outputs a solution composed of 7 shifts, shifting all the arrays to shift label 2. As before, the Lazy-Shift policy does not specify a single solution, but it may output a solution composed of as many as 7 shifts. However, the optimal solution, as shown in Figure 9.4, has only 5 shifts.

9.2 Measurements for expressions with only two alignments

We now turn to examine the case of a general graph with only two possible shift labels in their input and output vertices. The effectiveness of Algorithm 1 was tested on layer graphs of various depths and widths. Given the depth and width, the vertices are determined and it remains to randomly generate the edges. Assuming that operations are binary, we randomly selected two parents for each node. Random shift labels out of the two possible alignments...
Figure 9.2: An optimal labelling for the tree of Figure 9.1

Figure 9.3: A tree for which Algorithm 2 outperforms all the heuristics

Figure 9.4: An optimal labelling for the tree of Figure 9.3
Table 9.2: The percentage of test-runs in which Algorithm 1 outperformed all the heuristics.

<table>
<thead>
<tr>
<th>width</th>
<th>depth</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>23%</td>
<td>18.81%</td>
<td>30.5%</td>
<td>17.3%</td>
<td>24.3%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23.4%</td>
<td>21.5%</td>
<td>26.3%</td>
<td>35.2%</td>
<td>29.6%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>23.3%</td>
<td>38.4%</td>
<td>28.8%</td>
<td>31.0%</td>
<td>24.7%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>27.1%</td>
<td>32.2%</td>
<td>24.9%</td>
<td>38.1%</td>
<td>46.4%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>37.4%</td>
<td>50.3%</td>
<td>32.3%</td>
<td>34.6%</td>
<td>45.4%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>33.2%</td>
<td>34.3%</td>
<td>53.2%</td>
<td>42.8%</td>
<td>53.8%</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>45.6%</td>
<td>38.8%</td>
<td>40.3%</td>
<td>37.6%</td>
<td>48.9%</td>
<td></td>
</tr>
</tbody>
</table>

were assigned to the input and output nodes. We have run Algorithm 1 and all the heuristics on these randomly generated graphs. In table 9.2 we report the percentage of test-runs in which the algorithm outperformed all the heuristics.

9.2.1 An Example

In the examples of the trees in Figures 9.1 and 9.3, the Lazy-Shift policy did not specify a single solution, but at least one of its possible solutions was an optimal solution, since the decision to postpone a shift where possible never precludes an optimal solution. In the case of general graphs with only two possible shift labels, the decisions explicitly specified by the Lazy-Shift policy can actually preclude the optimal solution. We show an example of such a case in Figure 9.5.

![Figure 9.5: A DAG in which Algorithm 1 outperforms all the heuristics.](image)

The optimal solution to the graph in Figure 9.5, which is the output of Algorithm 1, requires only a single shift, applied to node $v$, shifting it from label 1 to label 2. The Eager-Shift policy heuristic will output a solution with four shifts. The Lazy-Shift policy
Table 9.3: Run time of a program that represents Figure 9.5

will defer shifting as long as possible, until it reaches the nodes \( u \) and \( w \), yielding a cost of two shifts.

### 9.2.2 Runtime Overhead

To check the shifting overhead on the execution of a program, we have run a program containing a loop that repeatedly computes the expression represented in Figure 9.5 on the Altivec platform, with various numbers of iterations. The optimal solution imposes one shift whereas the heuristic imposes two. The percentage of running time difference between using the optimal solution, which is the output of Algorithm 1, and using the Lazy-Shift heuristic solution, grows as the number of iterations increases, steadying eventually around 6%. In table 9.3 we report the execution times of a program corresponding to Figure 9.5 using one shift as the optimal solution and using two shifts as the Lazy-Shift policy solution.
Chapter 10

Conclusion and Open Problems

Various challenging problems stand in the way of effective optimizations for parallel and vector platforms. In this paper, we focused on the SIMD ALIGNMENT problem. In most previous work, simdization was studied assuming the input streams are all aligned. This is not the case in practice. Previous study of the SIMD ALIGNMENT problem offered only heuristics, with no guarantees on the quality of the obtained solution. In this paper, we presented two novel efficient algorithms that solve the SIMD ALIGNMENT problem optimally for two important special cases. For the case in which the input expression has only two different alignments we presented an algorithm that finds the optimal solution using known algorithms for the MINIMUM NODE S-T CUT problem as subroutines. For the case in which the input expression is a tree and each array appears only once in the expression, we presented an algorithm that finds the optimal solution using dynamic programming. These two special cases cover many practical instances of the SIMD ALIGNMENT problem.

There are various questions that still remain open for this problem. First, is the general SIMD ALIGNMENT problem NP-Hard? If so, it would be interesting to find approximation algorithms that solve the general case and analyze their approximation ratio. Also, it would be interesting to see lower bounds on the ability to approximate the problem, assuming $P \neq NP$. On the other hand, if the general SIMD ALIGNMENT problem is not NP-Hard, then can we present an algorithm that finds an optimal solution for the general case?
References


הסיבוכיות של מערכות סופר פלוס מוחשבים וקטרができות

לידיה פדרמן
הספריות של יישור עבורי מחשבים קסוריים

תודה על מחקר

לשם מחלי חקל של הדרישות לקבצלת הון
מנשערים למ던עים במintree השתיו

לידת מיכאלי

הון לסנטה סנטינו — מון סנטולינו לישראלי
лит 2006

שם תשע"ו

חפף
## תוכן עניניים

<table>
<thead>
<tr>
<th>מספר</th>
<th>ייעוד</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>תקציר באנלוגיה</td>
</tr>
<tr>
<td>3</td>
<td>התקפה</td>
</tr>
<tr>
<td>4</td>
<td>תכנית קורומית</td>
</tr>
<tr>
<td>5</td>
<td>מבנה קורומית</td>
</tr>
</tbody>
</table>
| 6     | SIMD של יישור ל- 
| 5.1   | יצוג פתרון ביעילות |
| 19    | אלגוריתמים פולינומייםubo יישויים עם שני פרבר תחתי |
| 21    | דוגמא |
| 23    | נתונים האלגוריתמים |
| 27    | נקודות סיבוכיות |
| 29    | אלגוריתמים פולינומייםubo יישויים עם ביטוי ייחודי מספר הייחודי המופע לכל מטר |
| 31    | דוגמא |
| 33    | נתונים האלגוריתמים |
| 35    | נקודות סיבוכיות |
| 37    | בקשת תחתי ר-זרדי ובשעיה תחתי ר-זרדי ייחודיים |
## תוצאת

<table>
<thead>
<tr>
<th>מספר</th>
<th>תכנית</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>תוצאת מידה עבורה על יטואין 9.1</td>
</tr>
<tr>
<td>44</td>
<td>תוצאת מידה עבורה על ספלי וכפי הוזה 9.1.1</td>
</tr>
<tr>
<td>44</td>
<td>תוצאת מידה עבורה על בספלי כפי הוזה 9.2</td>
</tr>
<tr>
<td>46</td>
<td>תוצאת 9.2.1</td>
</tr>
<tr>
<td>47</td>
<td>השפעה על זמני הזרמת 9.2.2</td>
</tr>
<tr>
<td>49</td>
<td>מסכמת ושאלות פתרון</td>
</tr>
<tr>
<td>49</td>
<td>רישום מקורות</td>
</tr>
<tr>
<td>6</td>
<td>תקציר מורחב</td>
</tr>
</tbody>
</table>
רשימה אורבירה

\[ a[i + 1] = b[i + 1] + c[i + 2] \times d[i + 1] \]

\[ a[i + 3] = b[i + 1] \times c[i + 2] + c[i + 2] \times d[i + 1] \]
רשימת טבלאות

<table>
<thead>
<tr>
<th>מספר</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
</tr>
<tr>
<td>9.2</td>
</tr>
<tr>
<td>9.3</td>
</tr>
</tbody>
</table>

43. אבחון הבדייקות בין ייצוגי האלגוריתם 2 טובים ונר משל החורשות
46. אבחון הבדייקות בין ייצוגי האלגוריתם 1 טובים ונר משל החורשות
47. עמדה הראיה בכר הדורות네투 השמעת ביאור

9.5. עמדה הראיה בכר הדורות네투 השמעת ביאור
תקציר מורחב

העפשתямиיה של תочки מעבת למלת הפרומת מרובים מחוברים בפוסיבי ואראיסטטיקות הקטניות היא
אצטגר מרשימים למדוניע מחדרים בים. נוספים שעчрיו בח baise הזן נתקלה לעמק.سقطה ז

אנו נמאמקים באחת מהמימיות והלאו: עיצי מצורה ל- SIMD המתקינים התווספת הצרה. בבריכ
וה זן קוד שלול של לולא ענ הפיתואת למזרחי ומקורות התאדות התאדות בודיקו היוצרת. בבריכ

יונשות הניבים בא construcciónים - הה קרément את הידיו העופת לشرو השפת עד השזר SIMD
здоров המחצית ארבעה קובליות. הזרועים הזרוע של SIMD ידועי השרי עד פ_sun המידיע פוש קתקים
ברצל של SIMD ידועי צל קלע התאדות.

//תגון בכותרת הקוד הבא שאם מ总局פית המועריכים

for (i = 0; i < 1000; i++) do
  a[i] = b[i + 1] + c[i + 2];
end for

קוד שעפול כל להלוכי עם מ الجنس מוגנוב בحطות שנות, קולמר הפרת מראני שאמינ
מוטרות האראש ביחס לשתייה. נזרעות פעלת ייעור לנט הצלב על מעין אלפער החשף מקבלי באוד- SIMD
סיקורית. בבריכ, אוPu אף קי מינתי פעלת מוניות לישון, המודע המנורחב ביבוש, קולמר
"עיצה בגרדה שיאטייה מוניות באמציה לינטמסים, רק לא אלפער החשון
מקבלי. די עלולים אחת אחת הכנה שי צלדת מזפר את המידיע בגרניטים, שהשיחוב יוביל
חלבצבים באומני.

הלישון מ tua של המידע בגרניטים מתבצע על ידי פעילת זויה. הזרוע אלא מתרבות בנוק
הלולאה כל מפעילה על מה ביצוע של תונותታת ב焠 יישפנויות. הביעי היא סדר מתרחש את
המדוע בגרניטים על מעין קמיס וدني ירישת היצירת ההכלה על הדי בחרות והד מערער מספר
Iterable文化旅游.

עדי, היריסטיקות ש_IMAGES-18125 משלחת claro-23644, אלא של התוכנית בוגר למספר הזרוע בפרות:

מותבך. לק פוריט על היריסטיקות המקובלת:

- הדריסטיקות ה-23644 משלחת את ה-0434-2 החסויות bleibt้า התאמות לפי הקטנות הפוטל
  והכל להחלות כל מת ה-0434-2 שבחלות מקבלי הקטנות הפוטל
- הדריסטיקות ה-23644-20520, משלחת את מפעילה ה-0434-2 הפוטל ע-ה-0434-2 באילןמקבלי
  התואמות של מקבלי עילןמקבלי הפוטל
- הדריסטיקות ה-23644-20520 משלחת את מפעילה ה-0434-2 הפוטל ע-ה-0434-2 באילןמקבלי
  המפעילים הפרוטוקולים复古-255 חתות
- רופינס הדריסטיקות המ-8204-29414 משלחת את הפרוטוקולים复古-255 שמצרפת את ה-0434-2 באילןמקבלי
  הפרופס בแปลก אוצרות ההשתתף האלח אול בחרות והד מערערлас
  בבריכו המ总局פים במלת ה-0434-2 באילןמקבלי הפרופס בחרות והד מערערлас.
for \( i = 0; i < 1000; i++ \) do
\[
a[i + 1] = b[i + 1] + c[i + 2] \ast d[i + 1];
\]
end for

Figure 1: A graph representation of the expression \( a[i + 1] = b[i + 1] + c[i + 2] \ast d[i + 1] \).

for \( i = 0; i < 1000; i++ \) do
\[
a[i + 3] = b[i + 1] \ast c[i + 2] + c[i + 2] \ast d[i + 1];
\]
end for

Figure 1: A graph representation of the expression \( a[i + 1] = b[i + 1] + c[i + 2] \ast d[i + 1] \).

1(d), 2(c), 1(b)

for \( i = 0; i < 1000; i++ \) do
\[
a[i + 3] = b[i + 1] \ast c[i + 2] + c[i + 2] \ast d[i + 1];
\]
end for

for \( i = 0; i < 1000; i++ \) do
\[
a[i + 1] = b[i + 1] + c[i + 2] \ast d[i + 1];
\]
end for
Figure 2: A graph representation of $a[i+3] = b[i+1] * c[i+2] + c[i+2] * d[i+1]$. 

$$a[i+3] = b[i+1] * c[i+2] + c[i+2] * d[i+1].$$