New Algorithms for SIMD Alignment ⋆

Liza Fireman⋆⋆[1], Erez Petrank⋆⋆⋆[2], and Ayal Zaks†[3]

[1] Dept. of Computer Science, Technion, Haifa 32000, Israel.
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.
[3] IBM Haifa Research Laboratory, Mount Carmel, Haifa 31905, ISRAEL.

Abstract. Optimizing programs for modern multiprocessor or vector platforms is a major important challenge for compilers today. In this work, we focus on one challenging aspect: the SIMD ALIGNMENT problem. Previously, only heuristics were used to solve this problem, without 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 provide *optimal* solutions for these two cases. The new algorithms employ dynamic programming and a MIN-CUT/MAX-FLOW algorithm as subroutines. We also discuss the relation between the SIMD ALIGNMENT problem and the MULTIWAY CUT and NODE MULTIWAY CUT problems; and we show how to derive an approximated solution to the SIMD ALIGNMENT problem based on approximation algorithms to these two known problems.

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 [25, 30, 14]. 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 many 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 often produced automatically by compilers (referred to as auto-vectorization or simdization). Explicit vector programming is time consuming and error prone. A promising alternative is to exploit vectorization technology to automatically generate SIMD codes from

⋆ This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).
⋆⋆ Email: liza@cs.technion.ac.il
⋆⋆⋆ Email: erez@cs.technion.ac.il. On sabbatical leave from the Computer Science Department, Technion, Haifa 32000, Israel.
† Email: zaks@il.ibm.com

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 [25].

One restrictive hardware feature that can significantly impact the effectiveness of simdization is the alignment constraint of memory units. In AltiVec [10], for example, a load instruction loads 16-byte contiguous memory from 16-byte aligned memory address (by ignoring the least significant 4 bits of the given 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 accessed 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 start at 16-byte aligned addresses.

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

The above code includes a loop with misaligned references. It requires additional realignment operations to allow vectorization 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 explicit shift instructions that execute inside the loop and therefore affect performance significantly. The problem is to automatically reorganize data streams in registers to satisfy the alignment requirements imposed by the hardware using 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 layout of the data in memory. This alternative suffers from several obvious limitations. In this paper the initial data alignment is assumed to be predetermined.

1.1 This work

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 Section 2. The main contribution of this paper is the presentation of two new algorithms that provide *optimal* solutions for two special cases. These special cases are quite general and cover many practical instances.

A polynomial-time algorithm for expressions with two alignments. For expressions that contain two distinct predetermined alignments, an efficient algorithm based on solving

a MINIMUM NODE S-T CUT problem can compute an optimal solution to the SIMD ALIGNMENT problem.

A polynomial-time algorithm for a single-appearance tree expression. For expressions that contain no common sub-expressions (i.e. form a tree) and where each array appears only once in the expression, an efficient algorithm based on dynamic programming can compute an optimal solution to the SIMD ALIGNMENT problem.

We stress that both cases are realistic and are common in practice. Also, the two cases do not supersede 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.

The SIMD ALIGNMENT problem can be mapped to the MULTIWAY CUT problem, and known algorithms for MULTIWAY CUT [11] can be used to solve the SIMD ALIGNMENT problem. However, known hardness results [12, 5] do not hold for the SIMD ALIGNMENT problem if a shifted stream may be used in both its original form and its shifted form (see Section 9). Furthermore, the mapping to the MULTIWAY CUT problem is more involved in this case (see Section 7).

1.2 Organization

In Section 2 we formally define the SIMD ALIGNMENT problem. In Section 3 we list useful heuristics proposed in the literature so far.In Section 4 we propose a graph representation of the SIMD ALIGNMENT problem, which will be used by the algorithms. In Sections 6 and 5 we present the efficient algorithms for the special case of expressions with only two alignments and for single-appearance tree expressions, respectively, and in Section 8 we present the effectiveness of these algorithms. Section 7 relates the SIMD ALIGNMENT problem to MULTIWAY CUT problems. Relevant prior art is described in Section 9 and Section 10 concludes.

2 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, sub-expressions (operations) and output operands, with an alignment value assigned to 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. However, a solution is feasible only if the inputs of *each* operation and output operand are properly aligned. An elaborate description of

how the shift operation is used with real platforms and a detailed example may be found in the thesis [15].

Shifting of data from one alignment value to another requires one shift operation but typically may require preliminary preparation of pre-loading and setting shift amount [23]. This preliminary work can often be placed before the loop where it is tolerable, whereas the shift operations themselves are part of the expression and must remain inside the loop. That is why our objective is to minimize the number of shifts.

3 Previous Heuristics

Previous work 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 section we shortly survey these heuristics, originally presented in [14], each of which can be shown to be sub-optimal for simple realistic instances [15].

- *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 simple policy is employed by the widespread GCC compiler [21, 24].
- *Eager-Shift Policy.* This policy shifts each load stream to the alignment of the store.
- *Lazy-Shift Policy.* This is a greedy policy of inserting shifts as late as possible in the expression. This policy does not specify how to break ties when different shifts may be used.
- *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 majority offset to the alignment of the store address.

4 An Abstraction of SIMD Alignment

In what follows, it will be useful to represent instances of the SIMD ALIGNMENT problem using annotated 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. This is because a single shift can be used for multiple alignments of the same array (e.g. to align both $a[i]$ to match $b[i+1]$ and $a[i+1]$ to match $c[i+2]$). Note, however, that this does not hold if only two distinct alignments are considered, as is the case in Section 6.

We call an expression in which each array appears with one alignment only a *single-appearance expression*. Most of the techniques employed in this paper relate to the study of graph algorithms. The representation of a single-appearance expression as a directed graph is the standard representation of expressions as graphs, except for two modifications. First, we add alignment labels to the nodes. Second, all appearances of the same array are represented by a single node. The nodes that represent the input and output streams are associated with alignment labels that signify the initial and final alignments, respectively, and the name of the array. Each operation (sub-expression) 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 following example:

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

The corresponding graph representation for the above expression is shown in Figure 1. This graph has three leaves (input nodes) labelled $1(b), 2(c), 1(d)$ and one root (output node) labelled $3(a)$. The labels signify the initial and final alignments and the array names. The operation nodes are labelled with the operation they represent. Note that the graph in this example in not a tree. It is a Directed Acyclic Graph (DAG).

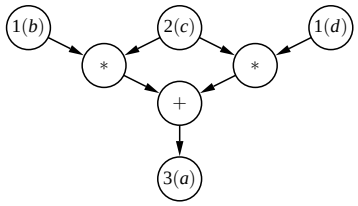


Fig. 1. A graph representation of $a[i+3] = b[i+1] * c[i+2] + c[i+2] * d[i+1]$.

4.1 A solution to a graph representation of a single-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-appearance SIMD ALIGNMENT problem is a labelling of the nodes. The cost of a solution for the graph $G(V,E)$ is the sum of the costs $c(v)$ associated with each node $v \in V$, where $c(v)$ is the number of distinct labels of v's successor nodes that are also different from the label of 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 the alignments of its successors (if different from its own). 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-appearance expression and therefore a shift must be done for a node if its successors 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 deal with sharing shifts among multiple alignment appearances of input operands or subexpressions. Therefore, the cost of the solution for a 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 2 we show an example of a graph with a given solution.

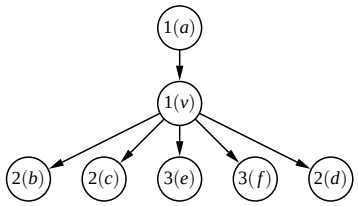


Fig. 2. A graph that exemplifies the cost of SIMD.

The labelling shown in Figure 2 costs only two shifts, because the descendants of v have only two distinct 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-appearance expression and* L *is a set of predetermined shift labels for the source and sink nodes.*
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:*

$$\sum_{v \in V} |\{c(u) \ : \ \exists u \in S(v) \ \ c(u) \neq c(v)\}|$$

where $S(v)$ *is the set of successor nodes 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.

5 A polynomial-time algorithm for single-appearance tree expressions

In this section we deal with expressions having an arbitrary number of alignments, but whose graph representations 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 in such cases.

In what follows, we consider the SIMD ALIGNMENT problem in its graph representation as defined in Section 4 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. The optimal solution is again 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 labelling can be translated into a solution for the SIMD ALIGNMENT associated instance in the following way. For every edge $(u,v) \in E$ connecting nodes of different labels: $c(u) \neq c(v)$, a shift is introduced from alignment $c(u)$ to alignment $c(v)$. Clearly, any labelling of the graph represents a *feasible* solution (though not necessarily an optimal one). The cost of a solution is equal to the number of such shifts, because shifts cannot be reused (the out-degree is 1).

Definition 3. *We say that a graph edge is a* shift edge*, with respect to a given alignment labelling of a tree, if its two incident nodes have distinct labels.*

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 T_v of T rooted at v. For each possible alignment $i \in I$, denote by $OPT_T(v,i)$ the minimum number of shift edges required by any labelling of T_v that assigns label i to node v. 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 *val* with an entry $val(v,i)$ for each node v and each possible shift $i \in I$. For each node v, the entry $val(v,i)$ represents a partial solution for T_v such that $val(v,i) = OPT_T(v,i)$. After computing the values $val(v,i)$ for all nodes v and alignments i, the algorithm uses them 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 edges are directed from the leaves to the root. The algorithm (see Algorithm 1) iterates over the nodes of the tree in topological order (starting from leaves and reaching the root at the end), filling the rows in the matrix $val(v,i)$. The base of this computation are the leaves for which a predetermined alignment label is provided. For each leaf v we set the value of $val(v,i)$ to be 0 if i is the predetermined alignment of v and ∞ otherwise. Then, the inner nodes of the tree are traversed in topological order, such that a node is visited after all its predecessors in the tree have been visited. The value of $val(v,i)$ for a node v with label i is computed by adding the costs associated with all incoming edges $(u,v) \in E$, where each such cost is the minimum of $val(u,j) + I_{(i \neq j)}$ taken over all $j \in I$, where $I_{(i \neq j)}$ is 1 if $i \neq j$ and 0 otherwise. Finally, the algorithm considers the entry $val(v,i)$ where v is the root node and $i = s_v$ is the predetermined alignment of the root as the cost of the solution. It is shown in the proof that this process computes $val(v,i)$ so that $val(v,i) = OPT_T(v,i)$, for each node v and $i \in I$.

Next, the tree is traversed from root to leaves in order to label all inner vertices in a consistent manner that matches the minimum number of shift edges. Given that each vertex has only one outgoing edge, its value is determined once, creating no conflicts.

Algorithm 1 Solving a Single-Appearance Tree expression.

Input: a tree $G(V,E)$ with the leaves and root having predetermined labels.
Output: a labelling $s(v)$ for all vertices.

1: **for** every node v in topological order (from leaves to root) **do**
2: **for** every $i \in I$ **do**
3: **if** v is a leaf, having predetermined alignment label s_v **then**
4: $val(v,i) = \begin{cases} 0 & i = s_v \\ \infty & i \neq s_v \end{cases}$
5: **else**
6: $val(v,i) = \sum_{u:(u\to v)\in E} \min_j \{val(u,j) + I_{(i\neq j)}\}$
7: where $I_{i\neq j}$ equals 1 if $i \neq j$ and 0 otherwise
8: **end if**
9: **end for**
10: **end for**

11: Set $s(v) = s_v$ for root and leaf nodes
12: **for** every node u in reverse topological order from root (excluding) to leaves (excluding) **do**
13: Let v be the unique successor of u: $(u \to v) \in E$.
14: $s(u) = \text{argmin}_j \{val(u,j) + I_{s(v)\neq j}\}$
15: where argmin_j is an index j for which the value $val(u,j) + I_{s(v)\neq j}$ is minimal.
16: **end for**

For a rigorous proof of correctness the reader is referred to the thesis [15]. The complexity of Algorithm 1 is governed by line 6 in which the algorithm computes the entries of the matrix val. There are $k|V|$ entries in this matrix where $k = |I|$, and for every node $v \in V$ we add $d^-(v)$ addends each requiring k comparisons, where $d^-(v)$ is the in-degree of v in the tree. Hence the total complexity of the algorithm is $O(k\sum_{v\in V} d^-(v)k) = O(k^2|E|) = O(k^2|V|)$.

6 A polynomial-time algorithm for expressions with only two alignments

In this section we present a polynomial-time algorithm for a restricted SIMD ALIGNMENT problem. We restrict the expression to be a SIMDG expression (and in particular, a single-appearance 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).

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 4. 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 5. 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.

Algorithm 2 for handling two alignments proceeds as follows. We start by considering the directed graph $G = (V,E)$ that represents the SIMD ALIGNMENT expression. 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. 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 (that is: $(u,w),(v,w) \in E$) are connected by an edge (u,v), if not already connected[4]. The direction of this (u,v) edge is immaterial, as we will be ignoring the directions of the edges E from now on. We further add two "terminal" nodes s_0 and s_1 that serve as 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 for s_1. In addition, sink nodes (which do not feed other nodes, but typically store into memory) are not expected to host shifts and should refrain from being cut nodes. This can be accomplished by merging each terminal node with all sink nodes connected to it, or by assigning infinite capacity (for max flow routine below) to sink nodes with predetermined labels[5]. Denote by H the obtained graph.

Next, we find a minimum node s-t cut C in H. Finding a solution to the MINIMUM NODE S-T CUT problem is possible in polynomial time using MAX FLOW algorithms [1, 9]. Denote by G' the (undirected) graph obtained by removing the cut C from H. Clearly s_0 and s_1 belong to G' and there is no path connecting s_0 and s_1 in G'. As the cut is in nodes, there may be more than two connected components on G'. 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 to label the remaining nodes. The labelled predecessor nodes (excluding those that belong to C) of each yet-unlabelled node must all have the same label, because every pair of such predecessors has been connected by an edge in H. We label each un-labelled node by the label of its predecessors. Source nodes (without predecessors) may belong to C, typically have predetermined labels which they retain. All remaining nodes are labelled

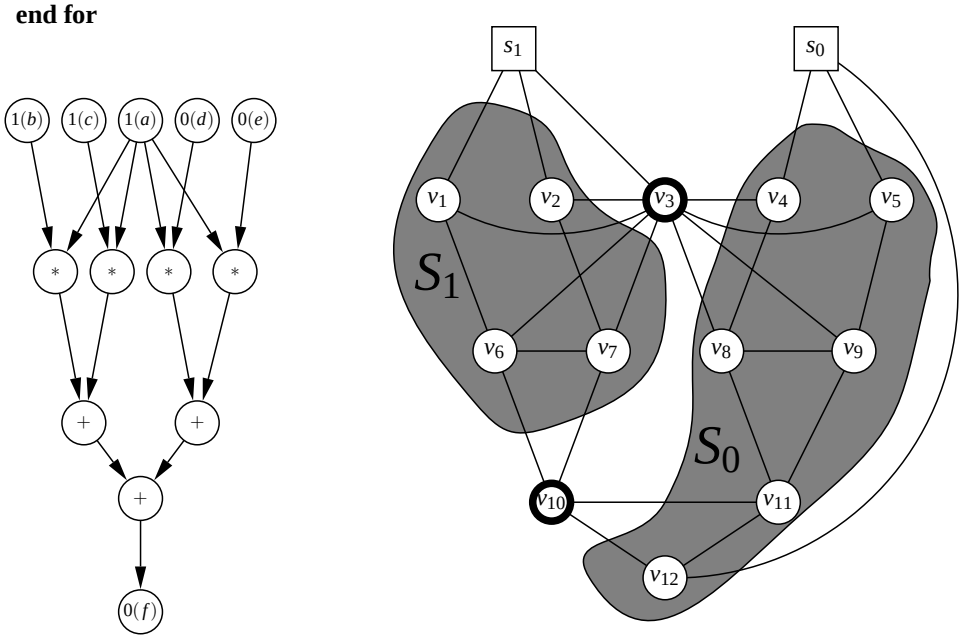
[4] Graphs with such additional edges are sometimes called *moral*, stressing that the parents of each node are "married". Note however that a parent may have more than one spouse.

[5] If a sink node does not have predetermined alignment, it will not belong to any minimum node-cut due to the "morality" property.

0 (we could label them 1 as well). Finally, a shift is provided for each node of the cut, to be applied to the result of the corresponding operation (i.e. *after* the operation is executed), from the label of the cut node to the "other" label (of one or more successor nodes).

An example. Consider the following code and its corresponding DAG which appears in Figure 3 (a). Two shifts are required to compute this expression.

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



(a) (b)

Fig. 3. (a) The DAG corresponding to $f[i] = (a[i+1]*b[i+1] + a[i+1]*c[i+1]) + (a[i+1]*d[i] + a[i+1]*e[i])$. (b) A minimum node $s_0 - s_1$ cut for the corresponding graph H.

Algorithm 2 produces the graph H with two additional terminals s_0 and s_1, and parent-connecting edges. It then finds a node-cut as in Figure 3 (b). We mark the cut nodes by encircling them with a bold line. The sets S_0 and S_1 are also marked.

Finally, we interpret the cut nodes as creating shifts in the computation. Array a is shifted from alignment 1 to alignment 0 (this is node v_3 in the cut). 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 $a[i+1]*b[i+1]+a[i+1]*c[i+1]$ is shifted from alignment 1 to alignment 0, enabling the execution of the final computation. Note that node v_8 has two predecessors both

with predetermined labels: v_3 and v_4. The label of v_8 is set to the (predetermined) label of v_4, because v_4 is not a cut-node (as opposed to v_3).

For a rigorous proof of correctness the interested reader is referred to the thesis [15]. The complexity of the algorithm is governed by the node cut phase whose complexity is $O((|V|_H)^2 \cdot |E|_H) = O(|V|^2 \cdot |V|^2) = O(|V|^4)$ [9], implying that the complexity of Algorithm 2 is $O(|V|^4)$.

7 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 6 (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 k 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.*

Multiway cuts appear, for example, in the problem of loop fusion for optimal reuse [13, 18, 16].

Definition 7 (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 k 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.*

We stress the difference between the above problems using the example in Figure 2. A minimal multiway cut for the undirected graph corresponding to the graph in Figure 2 includes **three** edges — when v is labelled 3. A minimal *node* multiway cut for this graph includes a **single** node — v. An optimal solution to the SIMDG problem for this graph costs **two** shifts — when v is labelled 1, indicating that it should be shifted to label 2 and to label 3. We now state the relations between the problems without proofs and derive approximation algorithms from them. (For more motivation and full proofs see [15].)

Lemma 1. *Every approximation algorithm to the* MULTIWAY CUT *problem with approximation ratio r provides an approximation algorithm to the SIMDG problem with approximation ratio $deg(G) \cdot r$ where $deg(G)$ is the maximum degree of graph G.*

Lemma 2. *Every approximation algorithm to the* NODE MULTIWAY CUT *problem with approximation ratio r yields an approximation algorithm with approximation ratio $(k-1) \cdot r$ for the SIMDG problem.*

For the MULTIWAY CUT problem there exist a (2-2/k)-approximation algorithm and a 3/2-approximation algorithm. For the NODE MULTIWAY CUT problem there exists a (2-2/k)-approximation [29]. Therefore we can deduce the following.

Corollary 1. *Using the* MULTIWAY CUT *approximation algorithms and Lemma 1 yield approximation algorithms for the SIMDG problem of approximation ratios* $deg(G) \cdot (2-2/k)$ *and* $deg(G) \cdot 3/2$.

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

8 Results

In Sections 6 and 5 we provided optimal algorithms for two special cases. In this section we demonstrate the practical advantage of using these algorithms compared to the heuristics mentioned in Section 3.

The effectiveness of Algorithm 1 was tested on complete binary tree expression graphs 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 2 to 7. For the trees randomly obtained as above, we ran Algorithm 1 and each heuristic described in Section 3. Table 1 tells for how many of the random trees (of depth d and k different alignments) none of the heuristics matched the optimal solution obtained by Algorithm 1. Note that as the size of the tree grows, the percentage of trees in which Algorithm 1 outperformed all of the heuristics grows rapidly. For intuition on where the heuristics fail the reader is referred to the thesis [15].

k	d=3	d=5	d=8
2	24.2%	94.6%	98.5%
3	28.4%	96.7%	98.8%
4	32.7%	95.4%	99.3%
5	27.4%	96.6%	98.1%
6	26.3%	94.7%	99.0%
7	29.2%	95.3%	98.7%
8	30.7%	96.4%	99.0%

Table 1. The percentage of test-runs in which Algorithm 1 outperformed all heuristics

width depth	3	4	5	6	7
3	23.0%	18.8%	30.5%	17.3%	24.3%
4	23.4%	21.5%	26.3%	35.2%	29.6%
5	23.3%	38.4%	28.8%	31.0%	24.7%
6	27.1%	32.2%	24.9%	38.1%	46.4%
7	37.4%	50.3%	32.3%	34.6%	45.4%
8	33.2%	34.3%	53.2%	42.8%	53.8%
9	45.6%	38.8%	40.3%	37.6%	48.9%

Table 2. The percentage of test-runs in which Algorithm 2 outperformed all heuristics

We now turn to examine the case of a general DAG with only two possible shift labels in their input and output vertices. The effectiveness of Algorithm 2 was tested on layer graphs of various depths and widths. Given the depth d and the width w, the vertices are determined and (assuming that operations are binary) two parents are randomly selected for each node. Random shift labels out of the two possible alignments are then assigned to the input and output nodes. Again, Algorithm 2 and all heuristics

were run on these randomly generated graphs and Table 2 shows the percentage of test-runs in which the algorithm outperformed all the heuristics. Further discussion and an example on which the heuristics fail appear in the thesis.

Finally, in order to check the shifting overhead on the execution of a program, we ran a program containing a loop that repeatedly computes an expression on the AltiVec platform, with various numbers of iterations. The optimal solution imposes one shift on this expression, whereas the best heuristic imposes two. The percentage of running time difference grows as the number of iterations increases, steadying eventually around 6%. More details appear in the thesis [15].

9 Related Work

There are two general approaches to optimizing code for SIMD architectures: the classical loop-based vectorization scheme [2] and the extraction of parallelism from acyclic code [19, 27]. 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 that of Eichenberger, Wu and O'Brien [14], which presents a set of heuristics for placing shifts in given expressions. These heuristics are described in Section 3. The study there first finds the shift schedule and then generates the relevant code. The shift schedule computed by our algorithms can be used to replace the first part of their study and feed their code generation to obtain more efficient code. A subsequent work [30] extends some of these heuristics to handle runtime alignment and alignment in the presence of length conversion operations.

Several compilers including VAST [28], GCC [21], compilers for SSE2 [3, 4] and VIS [8] provide re-alignment support using the Zero-Shift heuristic which shifts 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 [22]. Such architectures do not suffer (or suffer less) from the SIMD ALIGNMENT problem. Additional related work concentrates on detection of misalignment and techniques to increase the number of aligned accesses[20]. Our work deals with minimizing the number of shifts given a set of misaligned accesses, and is complementary to these techniques.

More heuristics for a more general case have been recently proposed by Ren et al. [26]. Typically, an SIMD platform may allow more advanced operations than just shifts. For example, Altivec allows any general projection of elements from two registers into a third register. Ren et al. propose a heuristic method to deal with streams of strides greater than one. This is an important related task that we do not consider here.

In [12, 6, 17, 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 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 no copies are made, so if data is moved it cannot be used in the original location (unless moved back again). In contrast, we assume that once a shift has been executed 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 to NP-Hardness proofs [12]. Furthermore, the ability to parallelize communication and computation costs is crucial to NP-Hardness proofs [5] but is not relevant to our model. Thus, these hardness results do not directly apply to 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.

10 Conclusion

Various challenging problems stand in the way of effective optimizations for 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 heuristics, with no guarantees on the quality of the obtained solution. In this paper we present 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 distinct alignments we present an algorithm that finds an optimal solution by solving the well known MINIMUM NODE S-T CUT problem. For the case where the input expression is a tree containing each array only once, we presented an algorithm that finds an optimal solution using dynamic programming. These two special cases cover many practical instances of the SIMD ALIGNMENT problem.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows*. Prentice Hall, 1993.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.
3. A. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance*. Intel Press, June 2004.
4. A. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *International J. of Parallel Programming*, 2:65–98, April 2002.
5. V. Bouchitt'e, P. Boulet, A. Darte, and Y. Robert. Evaluating array expressions on massively parallel machines with communication/computation overlap, 1995.
6. S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of POPL*, pages 16–28. ACM Press, 1993.
7. S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Trans. Program. Lang. Syst.*, 17(1):123–156, 1995.

8. G. Cheong and M. S. Lam. An optimizer formultimedia instruction sets. In *In Second SUIF Compiler Workshop*, August 1997.
9. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
10. M. Corporation. Altivec technology programming interface manual. June 1999.
11. E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts (extended abstract). In *Proceedings of the 24th ACM symposium on Theory of computing*, pages 241–251, New York, NY, USA, 1992. ACM Press.
12. A. Darte and Y. Robert. On the alignment problem. *Parallel Processing Letters*, 4(3):259–270, 1994.
13. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.*, 64:108–134, 2004.
14. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceeding of PLDI*, June 2004.
15. L. Fireman. The complexity of SIMD alignment. M.Sc. thesis, Technion — Israel Institute of Technology, Department of Computer Science, June 2006. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2006/MSC/MSC-2006-17.
16. G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, 1992.
17. J. R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *J. Parallel Distrib. Comput.*, 13(1):58–64, 1991.
18. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, 1993.
19. S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of PLDI*, pages 145–156, 2000.
20. S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of PACT*, 2002.
21. D. Naishlos. Autovectorization in gcc. In *Proceeding of GCC Developers Summit*, pages 105–118, 2004.
22. D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP Architecture. In *Proceedings of CASES*, pages 2–11, 2003.
23. D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proceedings of CGO*, pages 281–294, 2006.
24. D. Nuzman and A. Zaks. Autovectorization in gcc – two years later. In *Proceedings of GCC Developers Summit*, pages 145–158, 2006.
25. G. Ren, P. Wu, and D. Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.
26. G. Ren, P. Wu, and D. A. Padua. Optimizing data permutations for simd devices. In *Proceedings of PLDI*, pages 118–131, 2006.
27. J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *Proceedings of CGO*, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.
28. C. B. Software. VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit. http://www.psrv.com/vast altivec.html, 2004.
29. V. V. Vazirani. *Approximation Algorithms*, pages 38–40,155–160. Springer-Verlag, 1st edition, 2001.
30. P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proceedings of CGO*, pages 153–164, Washington, DC, USA, 2005. IEEE Computer Society.