Auto-vectorization using polyhedral compilation for an embedded ARM platform

Nieuwenhuizen, B.M.

B.M. Nieuwenhuizen

Auto-vectorization using polyhedral compilation for an embedded ARM platform

Bachelorscriptie

Scriptiebegeleiders:
H.J. Hupkes
T.P. Stefanov
J.T. Zhai

Datum Bachelorexamen: 20 augustus 2014

Mathematisch Instituut, Universiteit Leiden

Auto-vectorization using polyhedral compilation for an embedded ARM platform

Bas Nieuwenhuizen

Abstract

Various modern instruction set architectures contain SIMD instructions. By using these instructions, programmers and compilers can expose fine-grained parallelism on the processor and significantly increase the performance of their programs.

Analytical models such as the polyhedral model facilitate various transformations and analyses, which enable efficient automated selection of transformations.

In this thesis we examine applying polyhedral compilation to automatic vectorization by the compiler. In combination with an analytic model for the performance gains this enables efficiently selecting transformations to vectorize loops.

1 Introduction

Many programs spend most of their time in loops. It therefore became worthwhile for compilers to optimize loops. A significant part of the loops has few dependencies and can essentially have all iterations executed in parallel, especially those in multimedia and numerical applications.

A few processor architectures, trying to use that parallelism, introduced vector instructions that work on multiple data elements in parallel. However, using these instructions is often not easy and left for heuristics in the compiler or for the programmer.

If the programmer has to use these instructions the original source code becomes hard to read. Furthermore the programmer has to maintain multiple versions for different processors. Therefore, much research is being done in letting the compiler automatically use these instructions.

The polyhedral model [5] is used to model loops and perform complex transformations on them. Using these transformations it is possible to vectorize many loops that could not easily be vectorized by hand.

If the compiler decides which statements to vectorize and how to vectorize them, it can make decisions that do not result in the optimal performance. It is therefore necessary to estimate whether the transformation is profitable and which transformation is the most profitable.

One possible solution is to define a benchmark program and run that with all possible transformations. However, this is not practical as it is both intractable

to run such a program many times during a compilation and the program needs to explicitly generate another program for this method to work. Furthermore, it is also impractical to create a simple formula that captures all intricacies of the hardware platform, as modern hardware platforms are very complex. The alternative we explore in this thesis is therefore to estimate the profitability using an simple heuristic formula.

This thesis evaluates a set of transformations with a simple cost formula, based on a set of transformations and a formula proposed by Trifunovic et al.[10]. In particular, we evaluate two possible interpretations of the search space on an embedded ARM platform and compare it with a new search space that generalizes the other two.

Kong et al [9] also studied such a model, but focused on not only vectorization, but also cache behavior and the related loop tiling optimization.

This thesis is organized as follows: Section 2 describes the theory of the polyhedral model. Sections 3 and 4 present the search space and cost model for the optimizations. Section 5 describes our implementation. Section 6 presents an evaluation of that implementation and we conclude in section 7. Section 8 contains some ideas for further research.

2 The polyhedral model

The polyhedral model is a linear algebraic framework for representing the execution of parts of a program that is suitable for analysis and transformation. We start with a few definitions that are used in the definition for the polyhedral model.

Definition 2.1. *An affine map f from an n-dimensional vector space to an m-dimensional vector space is a map that can be decomposed into a linear map and a translation.*

It follows that every affine map f is a combination of a linear map g and a vector c such that

$$f(x) = g(x) + c$$

Definition 2.2. *A polyhedron P is a set in $\mathbb{R}^n$ such that there is a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$ such that*

$$P = \{p \in \mathbb{R}^n | Ap + b \geq 0\}$$

A polytope is a bounded polyhedron.

Definition 2.3. *A lattice is a set $L \subset \mathbb{R}^n$ such that*

$$L = \{\sum_{i=1}^{n} a_i b_i | a_i \in \mathbb{Z}\}$$

for a basis $b_1, \ldots, b_n \in \mathbb{R}^n$.

We consider a map f on an n-dimensional lattice affine if and only if there is an affine map $\mathbb{R}^n \to \mathbb{R}^n$ such that f is the restriction of that map to the lattice.

Definition 2.4. *Let $n, m \in \mathbb{N}$ and finite sets I, V be given. An instruction t is a tuple (i, r, w) where*

1. $i \in I$.
2. $r \subset V \times (\mathbb{Z}^n \overset{affine}{\to} \mathbb{Z}^m)$
3. $w \subset V \times (\mathbb{Z}^n \overset{affine}{\to} \mathbb{Z}^m)$

Wheres both r and w are finite. We call I the instruction alphabet and V is the set of variables. Furthermore, n is the dimensionality of the iteration vectors. We define for $k \in \mathbb{Z}^n$

$$\mathcal{R}_t(k) = \{(v, j) \in V \times \mathbb{Z}^m | \exists (v', f) \in r \colon v' = v \text{ and } j = f(k)\}$$

and similarly

$$\mathcal{W}_t(k) = \{(v, j) \in V \times \mathbb{Z}^m | \exists (v', f) \in w \colon v' = v \text{ and } j = f(k)\}$$

The sets I and V are mostly for bookkeeping and we assume in this paper that these sets are the same for all instructions in a program. Note that $\mathcal{R}_t(k)$ and $\mathcal{W}_t(k)$ are the array elements read and written by this instruction in iteration k. However not all arrays in a program have necessarily dimension m. This can be solved by letting m be the maximum number of dimensions and increasing the number of dimensions of arrays by adding dimensions with length 1.

Note that we use functions for the sets of accessed elements. These are used to model multiple executions of the same instruction. We can then say that the i'th execution reads elements $\mathcal{R}_t(i)$ and writes $\mathcal{W}_t(i)$. This definition generalizes that to a multidimensional description. As this model is used for loop transformations we call i the iteration vector.

Definition 2.5. *A statement S is a tuple $(\mathcal{D}_S, I_{S,d^S})$ where*

1. *$\mathcal{D}_S$ is the intersection of an d^S-dimensional polytope and a d^S-dimensional lattice.*
2. *I_{S,d^S} is a sequence $I^1_{S,d^S} = (i_1, r_1, w_1), I^2_{S,d^S}, \ldots, I^m_{S,d^S} = (i_m, r_m, w_m)$ of instructions for which the iteration vectors are contained in $\mathbb{Z}^{d^S}$.*

We call $\mathcal{D}_S$ the iteration domain of the statement.

The interesting property of a statement is not which array elements it reads and writes, but from which array elements it needs the values and which elements it changes. The difference is that if it writes an array element and reads

it in a later instruction, it does not depend on the value that is initially stored in the element. The set of read elements is

$$\mathcal{R}_S^*(k) = \bigcup_{u=1}^{|I_S|} \left(\mathcal{R}_{I_S^u}(k) \setminus \bigcup_{v=1}^{v-1} \mathcal{W}_{I_S^v}(k) \right)$$

Note that the map $k \mapsto \bigcup_{u=1}^{|I_S|} \mathcal{R}_{I_S^u}(k)$ can be decomposed in essentially a set of affine functions, which has computational advantages. $\mathcal{R}_S^*(k)$ does not have that property as for example instruction 1 can write to $(x, (k_1))$ and instruction 2 can write to $(x, (10 - k_1))$ for some x, which result in a dependency on an element except in the case that $k_1 = 5$. As the decomposition in a set of affine functions has computational advantages, such a superset of $\mathcal{R}_S^*(k)$ is often chosen. Furthermore as the map $k \mapsto \bigcup_{u=1}^{|I_S|} \mathcal{R}_{I_S^u}(k)$ is simple to compute and can be decomposed, we never use a set that is not contained in that set:

$$\mathcal{R}_s^*(k) \subseteq \mathcal{R}_s(k) \subseteq \cup_{u=1}^{|I_S|} \mathcal{R}_{I_S^u}(k)$$

Note that another solution to this problem is to decompose statements with multiple instructions into statements with a single instruction. This however increases the number of statements

For written elements we can use

$$\mathcal{W}_s(k) = \cup_{u=1}^{|I_S|} \mathcal{W}_{I_S^u}(k)$$

Note that the intersection of an n-dimensional polytope P and lattice with basis $b_1, \dots b_n$ can be represented by an integer polytope on $\mathbb{R}^n \times \mathbb{Z}^n$ with the the added constraints

$$x_i = \sum_{j=1}^{n} x_{n+j} (b_{n+j})_i$$

for all $i \in \mathbb{Z}, 1 \leq i \leq n$ using the parameters $x_1, \dots, x_{2n}$.

Definition 2.6. *A statement instance of a statement S in an iteration domain D_S for S is a pair (S, i) with $i \in \mathcal{D}_S$.*

Statement instances correspond to a single execution of a statement.

We do not yet have a representation of the execution order of statement instances. An intuitive solution is an affine map from statement instances to time. A multidimensional representation of the time using a lexicographic ordering is used due to limitations in the process that generates code from the polyhedral model.

Definition 2.7. *A schedule for a statement S is an injective affine function from $\mathcal{D}_S$ to a lattice.*

By putting multiple statements and corresponding iteration domains and schedule together we get a Static Control Part (SCoP).

Definition 2.8. *A Static Control Part (SCoP) is a finite set*

$$\{(S, \theta_S) \colon S \text{ is a statement and } \theta_S \text{ is a schedule for } S\}$$

such that for statements $S \neq T$ we have that

$$\theta_s(\mathcal{D}_S) \cap \theta_T(\mathcal{D}_T) = \emptyset$$

and the codomains of all schedules have an equal number of dimensions.

SCoPs are used to represent a maximal region of a program that can be represented by the polyhedral model. The parts of a program that can be represented by a SCoP can be characterized by the following conditions [4]

1. The only possible control flow consists of loops and if's with affine bounds.
2. (Multidimensional) arrays are the only data structures.
3. There is are no calls or they have been inlined.
4. Affine bounds, conditions and memory accesses depend only on outer loop counters, constants and a set of parameters that is constant during the execution of the SCoP.

Let us consider the following example

```
for(int i = 0; i < 1024; ++i) {
  a[i] = 0;                          // S1
  for(int j = i; j < 2048; ++j) {
    a[i] += b[i][j];                 // S2
  }
}
```

This example is trivially a SCoP, with the iterations domains

$$\mathcal{D}^{S_1} = \{i \in \mathbb{Z} \colon 0 \leq i \leq 1023\}$$

and

$$\mathcal{D}^{S_2} = \{(i, j) \in \mathbb{Z}^2 \colon 0 \leq i \leq 1023 \wedge j - i \geq 0 \wedge \text{J} \leq 2047\}$$

Furthermore, we have the sets of affine memory accesses

$$\mathcal{R}^{S_1}(i) = \emptyset$$
$$\mathcal{W}^{S_1}(i) = \{(a, (i))\}$$
$$\mathcal{R}^{S_2}(i, j) = \{(b, (i, j))\}$$
$$\mathcal{W}^{S_2}(i, j) = \{(a, (i))\}$$

And finally we have the schedules

$$\theta^{S_1}(i) = (i, 0)$$
$$\theta^{S_2}(i, j) = (i, 1, j)$$

Transformations

The transformations considered in this model are primarily concerned with the order of statements and their instances. To this end, we only consider transformations of the schedule and the iteration domain.

Definition 2.9. *A transformation is an invertible map f from a SCoP P_o to a SCoP P_n such that for all $(S', \theta_{S'}) = f((S, \theta_S))$ we have invertible maps g_S, h_S such that*

1. $\mathcal{D}_{S'} = g_S(\mathcal{D}_S)$
2. $\theta_{S'} = h_S \circ \theta_S \circ g_S^{-1}$
3. $|I_{S'}| = |I_S|$ *and for all* $I_S^k = (i, r, w)$ *we have*

$$I_{S'}^k = (i, \{v \circ g_S^{-1} : v \in r\}, \{v \circ g_S^{-1} : w\})$$

where h_S and g_S^{-1} are affine.

In essence a transformation can change schedules, but not the code that is executed. However, we allow a transformation on the iteration domain as we want to be able to increase the number of dimensions of the codomain of the schedule, which is not possible using an affine map. The instructions are changed accordingly to compensate for the change in iteration domains.

Furthermore, a transformation induces an isomorphism between statement instances. We can map the instance (S, i) to $(S', g_S(i))$. When considering this isomorphism, the only effect a transformation has is therefore changing the order of execution of statement instances.

Not every transformation is valid however, as after a transformation a statement instance could be executed that depends on an instance that is executed afterwards.

Definition 2.10. *Two statement instances (S_1, i_1) and (S_2, i_2) are conflicting if either*

1. $\mathcal{R}_{S_1}(i_1) \cap \mathcal{W}_{S_2}(i_2) \neq \emptyset$
2. $\mathcal{W}_{S_1}(i_1) \cap \mathcal{R}_{S_2}(i_2) \neq \emptyset$
3. $\mathcal{W}_{S_1}(i_1) \cap \mathcal{W}_{S_2}(i_2) \neq \emptyset$

We can then define the transformations that do not change program semantics.

Definition 2.11. *A transformation from P_o to P_n is semantics-preserving if and only if there are no conflicting statement instances (S_1, i_1) and (S_2, i_2) in SCoP P_o such that*

$$\theta_{S_1}(i_1) < \theta_{S_2}(i_2)$$

and their corresponding statement instance (S_1', i_1') and (S_2', i_2') have

$$\theta_{S_1'}(i_1') > \theta_{S_2'}(i_2')$$

Girbal proposed a family of schedules for a statement with a d^S dimensional iteration domain as follows:

$$\Theta^s = \left(\begin{array}{cccc|c} 0 & 0 & \dots & 0 & \beta_0^S \\ A_{1,1} & A_{1,2} & \dots & A_{1,d^S} & 0 \\ 0 & 0 & \dots & 0 & \beta_2^S \\ A_{2,1} & A_{2,2} & \dots & A_{2,d^S} & 0 \\ 0 & 0 & \dots & 0 & \beta_0^S \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{d^S,1} & A_{d^S,2} & \dots & A_{d^S,d^S} & 0 \\ 0 & 0 & \dots & 0 & \beta_0^S \end{array}\right)$$

where $A \in \mathbb{Z}^{d^S \times d^S}$ and $\beta \in \mathbb{Z}^{d^S+1}$. The affine scheduling function $\theta^s \colon \mathbb{Z}^{d^S} \to \mathbb{Z}^{2d^S+1}$ is then

$$\theta^S(i) = \Theta \begin{pmatrix} i \\ 1 \end{pmatrix}$$

Due to limitations in the code generation, the matrix A needs to be invertible [11]. Furthermore, we require the ranges of θ^{S_1} and θ^{S_2} to be disjunct for all pairs of different statements S_1, S_2.

Different classical loop optimization correspond to different parts of the scheduling matrix [6]:

1. Matrix A has to be an invertible matrix and is used for transformations such as loop interchange and other transformations that transform the execution order of instances within a statement.

2. β is used to transform the relative ordering of statement to each other. This is used in optimizations such as loop fusion.

Loop interchanges[6] correspond exactly to permutations of the rows of submatrix A of the schedule. For example

```
for(int i = 0; i < 1024; ++i)
  for(int j = 0; j < 512; ++j)
    arr[i][j]++;
```

with schedule

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$$

can be loop interchanged to

```
for(int j = 0; j < 512; ++j)
  for(int i = 0; i < 1024; ++i)
  arr[i][j]++;
```

with schedule

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ 1 \end{pmatrix}$$

We transposed the rows of the submatrix A from $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ to $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

Strip-mining[6] corresponds to replacing a loop index i in the iteration domain with $i = cj + k$ where $0 \leq k \leq c - 1$ and c is a constant, and subsequently transforming the schedule to make k the index of the innermost loop. If the innermost loop has no dependencies between iterations, the code generation can then vectorize this loop. For example

```
for(int i = 0; i < 1024; ++i)
  arr[i]++;
```

can be transformed to

```
for(int i = 0; i < 1024; i += 4)
  for(int j = 0; j < 4; ++j)
    arr[i + j]++;
```

The code generator can then emit code to execute the inner loop in parallel. The iteration domain is modified from $\{i \in \mathbb{Z} : 0 \leq i \leq 1023\}$ to $\{(i, j, t) \in \mathbb{Z}^3 : i = 4t \wedge 0 \leq j \leq 3 \wedge 0 \leq i + j \leq 1023\}$, where t is used as a variable to contract i to be a multiple of 4.

3 Optimization search space

How effective an optimization is depends on the set of solutions considered by the optimization. Optimizing in a large search space has the potential of finding a good solution that would be excluded by a small search space. However, searching through a large search space will also take more time. Therefore the size of the search space is a compromise between compilation time and run time.

There exist integer linear programming formulations of the set of valid schedules into which we could incorporate an analytical cost function. It has however already been established that this formulation introduces a sufficiently large number of variables to make this formulation impractical for a production compiler.

We therefore need to limit our search space to a subset of the semantics-preserving transformations. We will present three algorithms which generate a search space. All three search spaces are based on a combination of loop interchanges and strip-mining.

Let x be the SCoP that is to be optimized, n the number of statements in the SCoP and d the maximum dimensionality of the iteration domains. Furthermore let ϕ be the function that assigns each SCoP a cost, which we try to minimize.

Algorithm 1 Optimizer X_1

$x_{\min} \leftarrow x$
for $\sigma \in \{$ d-element permutations$\}$ **do**
  x' is x interchanged by the permutation σ.
  for $v = 1$ to d **do**
    y is x' with dimension v strip-mined for all statements.
    if $\phi(y) < \phi(x_{\min})$ **then**
      $x_{\min} \leftarrow y$
    end if
  end for
end for

Algorithm 1 searches exactly the loop interchange and strip-mine combinations, where all statements are transformed exactly the same way. This search space is similar to more traditional compiler optimizations that transform whole loops.

The algorithm evaluates up to $ndd!$ possible schedules. As the number of dimensions d is often very small, the algorithm is tractable even though the number of schedules is exponential in the number of dimensions.

This search space is limited, an example where it does not find the vectorization opportunities is

```
for(int k = 0; k < 1024; ++k) {
  for(int i = 0; i < 1024; ++i)
    for(int j = 0; j < 1024; ++j)
      v[i][j + 1] = v[i][j];        // A
  for(int i = 0; i < 1024; ++i)
    for(int j = 0; j < 1024; ++j)
      w[i+1][j] = w[i][j];          // B
}
```

where A is only vectorizable in i and B is only vectorizable in j.

Algorithm 2 is able to able to vectorize that example by greedily considering to strip-mine dimensions per statement. It searches through a larger search space. However, as it searches greedily through that space it still evaluates up to $ndd!$ schedules.

As the algorithm does not backtrack, it does not consider all combinations of strip-mine dimensions. In particular, the searched space X_2 is not a superset of X_1.

Furthermore, it ensures that it has a semantics-preserving schedule at all times by checking a complete schedule which does not strip-mine the statements that have not yet been considered. Note that this misses opportunities in loops as simple as

```
for(int i = 0; i < 1024; ++i) {
  for(int j = 0; j < 4; ++j)
    v[i][j] += w[i][j];          // A
```

Algorithm 2 Optimizer X_2

$x_{\min} \leftarrow x$
for $\sigma \in \{$ d-element permutations$\}$ **do**
  x' is x interchanged by the permutation σ.
  for $i = 1$ to n **do**
    $x'_{\text{orig}} \leftarrow x'$
    for $v = 1$ to d **do**
      y is x'_{orig} with statement i strip-mined in dimension v.
      if $\phi(y) < \phi(x')$ **then**
        $x' \leftarrow y$
      end if
    end for
  end for
  if $\phi(x') < \phi(x_{\min})$ **then**
    $x_{\min} \leftarrow x'$
  end if
end for

```
  v[i][0] = 1.0 / v[i][0];      // B
}
```

As B can not be strip-mined without strip-mining A in dimension i. However it can be more beneficial to vectorize B as divisions are often an order of magnitude slower than additions. Note that a loop interchange is not going to help, as the algorithm still chooses dimension j first for statement A.

All loop interchange and per statement strip-mining combinations generate $d^n d!$ schedules to consider, which is also exponential in the number of statements n.

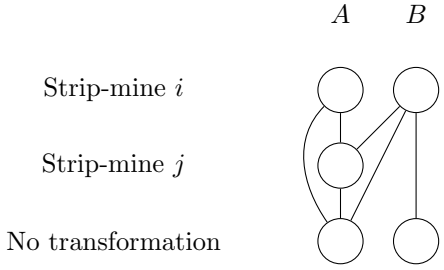
Theorem 3.1. *A transformation from P_o to P_n is semantics-preserving if and only if for all statements S_1 and S_2 in P_o the restriction of the transformation to those two statements is semantics-preserving.*

Proof. Suppose that the transformation is not semantics-preserving. Then there are statement instances i and j of statement $S_i, S_j \in P_o$ such that i depends on j and in P_n i is executed before j. The restriction of the transformation to S_1 and S_j also contains i and j and is therefore not semantics-preserving.

However, suppose that there are statements S_i and S_j in P_o such that the restriction of the transformation to S_i and S_j is not semantics-preserving. Then there are statement instance i and j in S_i and S_j such that i depends on j and in P_n i is executed before j. The original transformation transforms those instances in the same way and is therefore not semantics-preserving. □

Note that if we have a transformation for each single statement in a SCoP, we can combine them into a transformation for the entire SCoP. Furthermore it follows that this combined transformation is semantics-preserving if and only if all combinations of two transformations is semantics-preserving.

We can then construct a graph of transformations on statements in a SCoP P where each node corresponds to a transformation on a statement and there is an edge between two nodes if they are not equal and correspond to the same statement, or their combined transformation is not semantics-preserving. This graph is not simple as it can have loops: A transformation on a single statement can be not semantics-preserving and the corresponding node will therefore have a loop. The graph that corresponds to the last example is



Each independent set of the graph corresponds to a transformation on a subset of P, as it corresponds to at most one transformation per statement. By theorem 3.1 this combined transformation is semantics-preserving. On the other hand every semantics-preserving transformation that can be represented by a set of nodes in the graph corresponds to an independent set of nodes by theorem 3.1.

It follows that the maximum independent set in the graph has exactly $|P|$ elements, as at most $|P|$ transformations can be combined into a transformation and the identity transformation corresponds to a set with $|P|$ elements. Therefore each semantics-preserving transformation of P that can be represented by a set of nodes in the graph corresponds to a maximum independent set and each maximum independent set corresponds to a semantics-preserving transformation on P.

Furthermore if the cost model can be evaluated independently per statement, we can assign a weight to each node and the cost of a transformation is then the sum of the weights of the corresponding nodes. The problem of finding a semantics-preserving transformation can then be reduced to finding a maximum independent set with minimum weight. By subtracting a suitably large constant from all weights this reduces to a minimum weight independent set problem.

Algorithm X_3 considers every loop interchange and for every loop interchange it builds a graph of the strip-mine transformations and finds the minimum cost semantics-preserving transformation.

Using this reduction to calculate the optimal strip-mining transformations we get an algorithm that does $d^2n^2d!$ polyhedral operations to construction the graphs and $d^nd!$ simpler operations. This formula still contains the exponential part, but the underlying operations is much faster, especially if one considers that the polyhedral operations are non-polynomial operations themselves.

Note that search space X_3 with all loop interchanges on the entire SCoPs and per statement strip-mine transformations is a proper superset of X_1 and X_2. With an accurate cost model searching in X_3 therefore produces code that

Algorithm 3 Optimizer X_3

```
x_min ← x
for σ ∈ { d-element permutations} do
    x' is x interchanged by the permutation σ.
    build transformation graph
    y is x' transformed in optimal way according to the graph.
    if φ(y) < φ(x_min) then
        x_min ← y
    end if
end for
```

is at least as fast as the code generated using the other two search spaces.

4 Cost model

An analytic cost model is key to exploring the search space without having to explicitly construct every possible solution in the search space. Furthermore, having a simple model for the performance gains facilitates reasoning about which parts of the search space will contain the best solutions.

Both modern processors and the optimizations that are typically performed after vectorization are sufficiently complex to make an analytic model that perfectly models the performance impractical. Therefore, the cost model has to be an approximation.

The central assumption we use is that vectorized loops run faster except if there are factors that interfere with the vectorization. We assume that the most important interfering factor is that the vectorizer needs to insert extra instructions for memory accesses with a stride between iterations that is not equal to 1.

Our heuristic is based on the heuristic chosen by Trifunovic et al. [10]. They also based their cost model on the alignment of the memory accesses. However, LLVM does not store useful information about pointer alignment. We therefore decided not include the alignment component in the cost model.

Memory stride

Vector load and store instructions on ARM can only access series of consecutive values. If the compiler decides to vectorize a loop in such a way that the resulting program needs to load and store non-consecutive vectors, it has to include extra instructions to do these loads.

The stride of a memory access depends on the layout of the array that is accessed. As memory is a single-dimensional array the compiler needs to linearize the array. Most compilers store element i, j, k of an arrays with dimensions size L, M and N and elements size E at index $(iMN+jN+k)E$ and this layout is in fact suggested by several language standards [8], as they see a multidimensional array as an one dimensional array of arrays.

Formula

The formula used to calculate the cost of a statement $S = (\mathcal{D}_S, I)$ is then

$$\frac{|\mathcal{D}^S|}{VF} \left(\sum_{i \in I} c(i) \right)$$

Where $c(i)$ is the cost of the vectorized version of instruction i. For loads and stores this cost depends on the memory stride of the access as explained in an earlier subsection.

VF is the vectorization factor, which is the number of elements the vector instructions operate on. If the statement contains instructions that work with differing amounts of elements, we choose the largest one.

For the purpose of the cost formula, if an instruction works on $k \leq VF$ elements, it is counted $\frac{VF}{k}$ times. In our current implementation those element counts are all powers of two, which ensures that this quotient is integer and it is therefore possible to emit those instructions to work on the larger number of elements.

5 Implementation

We implemented the vectorization optimization in the Polly [7] framework with the LLVM compiler framework.

Polly uses a variant of the polyhedral model that is slightly different to the model described in section 2. The developers of Polly chose to represent functions as relations between the domain and codomain, which can be represented by integer polyhedra. It allows schedules that are not affine and the strip-mine transformations can therefore be done without transforming the iteration domain.

Determining the penalties

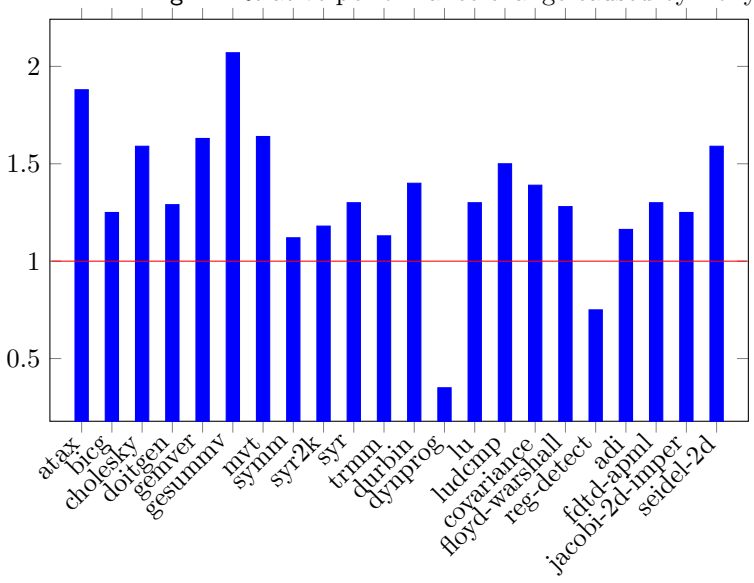
We used the following costs in the implementation:

stride	cost
0	1
1	2
2 or larger	4

For stride 0 there is a NEON instruction that loads a float and replicates it across the vector register in a single cycle. Note that there are no write with stride 4, as those would induce a dependency between instructions that prevent the vectorization.

For stride 1 we use the plain vector load and store instructions. Because LLVM does not have suitable alignment information, we cannot emit the aligned versions of these instructions. The nonaligned instructions, however, have an extra cycle latency. The primary problem with the alignment information in LLVM is that it does not store the alignment information of pointers, but of

Fig. 1: Relative performance change caused by Polly.



load and store instructions. If a pointer is accessed in a loop and the pointer is aligned in the first iteration then it typically is not in other iterations and the instructions have the lowest common denominator, which is not aligned.

The NEON instruction set contains load and store instructions that can access a specific value in a vector register. It follows that we can load and store values with larger strides by doing 4 separate loads and stores.

6 Evaluation

We experimentally evaluate the vectorization optimization on a modified version the Polybench/C [2] benchmark suite on a Zedboard[3], which contains an ARM Cortex A9 processor[1]. We compared it against the existing vectorizers, including the primary optimizer included in Polly.

We modified the Polybench suite to use global arrays instead of arrays allocated by a malloc wrapper in a different file. Firstly, memory allocation function provided by the system libraries for the Zedboard had a bug which resulted in memory corruption. Furthermore, LLVM could not detect that the arrays are disjunct and the Polly framework does not yet support generating separate versions and deciding at runtime which version is applicable. Doing the aforementioned modification solves both issues.

As the ARM NEON instructions only support single precision arithmetic, the benchmarks do not include results using double precision arithmetic. We tested both the tiny and small size benchmark configuration using the following configurations of the compiler:

Fig. 2: Geometric mean of the relative runtime performance.

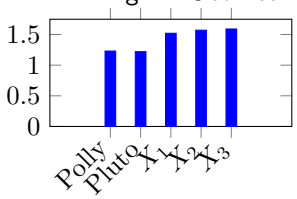
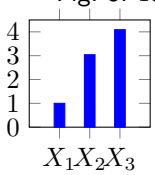


Fig. 3: Relative total compile times using the different search spaces.



1. Clang with `-O3` with Polly, but without any polyhedral optimizations.
2. Clang with `-O3` with Polly and a optimizer based on Pluto, which is an existing optimization based on the polyhedral framework.
3. Clang with `-O3` with Polly and the vectorization optimization using search space X_1.
4. Clang with `-O3` with Polly and the vectorization optimization using search space X_2.
5. Clang with `-O3` with Polly and the vectorization optimization using search space X_3.

We compare them against Clang with `-O3`, which include a loop vectorizer and basic block vectorizer that do not use the polyhedral framework.

Figure 1 shows that enabling Polly without any scheduling optimizations has significant performance improvements on average. The regression on dynprog is a consequence of bad interaction between Polly and the loop vectorizer. Most of the benchmarks have loops with very small bodies, which could explain the high variance of the results, as small changes in code can have relatively large changes in performance.

Figure 2 shows that extending the search space can result in better performance. All search spaces result in better code than Polly or Polly with Pluto.

5 shows the individual benchmarks. Our search spaces do not regress significantly for any benchmark compared to just Polly. Note the benchmark "ftfd-apml". It has equal runtimes for X_1 and X_3, but is slow for X_2. This shows that X_2 can result in worse solutions than X_1 and that this is not only theoretical, but also encountered in practice.

Figure 3 shows that searching through the larger search spaces does take significantly more time. Note that we can precalculate the cost for each strip-mine dimension for each statement. This is the most expensive part of the cost formula and we avoid calculating it multiple times. This brings the compile time much close to each other than in a naive implementation, as those parts of the cost formula dominate the time needed to determine the cost.

Furthermore, the compile times for the optimizations were mostly dominated by determining which transformations were semantics-preserving, as the time needed for a single query was around 0.6 milliseconds on average.

In X_3 the time needed to find the minimum weight independent sets was typically less than 1% of the compile time for all but one benchmark.

The compile and run times adhere to the typical tradeoff between the two.

7 Conclusion

We presented a set of optimizations which try to vectorize loop nests according to an approximate cost model. The optimizations use the polyhedral model and leverage the extra information that can be represented to enable complex transformations. The cost model can be used with just the polyhedral representation, which avoids code generation while searching for profitable vectorization strategies. We considered three search spaces for the optimizations, each of which is based on loop interchanges and strip-mining.

Furthermore, we implemented the optimizations in Clang and evaluated them using an ARM processor.

8 Discussion

On a Core i7, some benchmarks were slower with the vectorization pass than without. On closer examination these slowdowns were caused by one pattern: if we vectorize a statement in a loop and do not vectorize another statement in that loop we get mismatched code such as

```
for(int i = 0; i < 1024; ++i) {
  if(i % 4 == 0)
    vectorizedStatement1(i);
  statement2(i);
}
```

This is then optimized by the backend. The remainder is replaced by a bitwise and, but the is still an extra branch. The preferred solution in this code would be to transform into

```
for(int i = 0; i < 1024; i += 4) {
  vectorizedStatement1(i);
  for(int j = 0; j < 4; ++j)
    statement2(i + j);
}
```

However, this code still contains extra branches that can interact with the processor to get a significant slowdown. How to include this increase in complexity of the control flow in the performance model is still an open problem.

Furthermore, some examples used throughout the paper can be clearly vectorized, but the algorithm does not find it because it always strip-mines using the equation $i = j + k$ with $0 \leq k \leq 3$ and j a multiple of 4. In some of these cases the algorithm would be able to vectorize the code if we also allow j a specific value x such that $j \equiv x \mod 4$.

We found that some of the polyhedral operations were quite slow and we theorize that this is caused by the overly general notion of integer map in Polly, which causes more general and slower algorithms to be used.

References

[1] Cortex-a9 processor. `www.arm.com/products/processors/cortex-a/cortex-a9.php`, accessed 20 June 2014.

[2] Polybench/c. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`, accessed 20 Jun 2014.

[3] Zedboard. `http://www.zedboard.org/product/zedboard`, accessed 20 June 2014.

[4] Cédric Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, Pierre et Marie Curie, 2004.

[5] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[7] Tobias Grosser. Enabling polyhedral optimizations in llvm. Diploma thesis, Passau, 2011.

[8] ISO. Information technology – programming languages – c++. ISO 14882:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[9] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 127–138. ACM, 2013.

[10] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, pages 327–337. IEEE Computer Society, 2009.

[11] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2006.

More benchmark results

Fig. 4: Relative performance of schedule optimizations against Clang without Polly, part 1

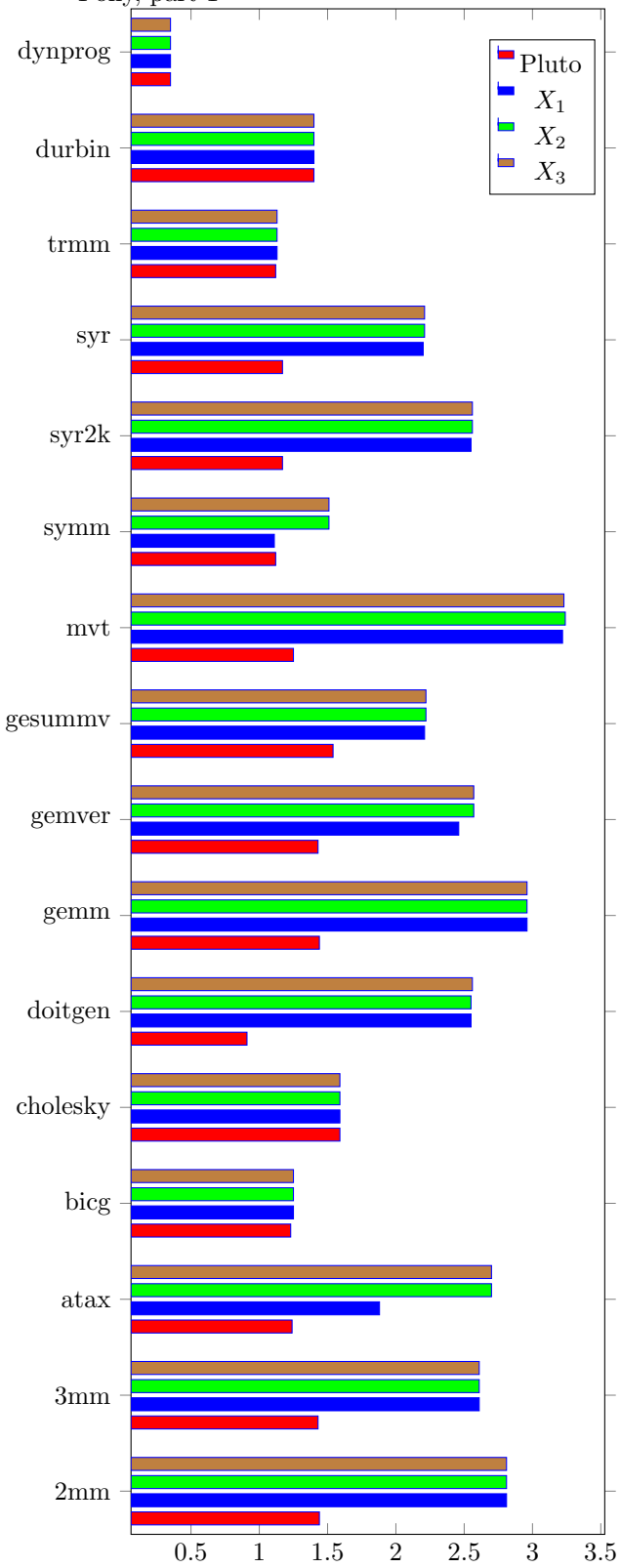


Fig. 5: Relative performance of schedule optimizations against Clang without Polly, part 2.