MASTER'S THESIS

Automatic vectorization for the CVP

by

M.F.A. Coenen

Supervisors: Prof. dr. ir. C.H. van Berkel, Dr. S. Balakrishnan

Eindhoven, August 2004
Preface

This report is my Master’s thesis, resulting from my graduation project for my studies in Computing Science at the Eindhoven University of Technology. The graduation project was performed at the Embedded Systems Architectures on Silicon (ESAS) department of Philips Research Laboratories in Eindhoven.

This report describes the research I have done on automatic vectorization for the CVP-processor. I have described techniques for automatic vectorization and implemented them, which resulted in an experimental compiler. I have tested this experimental compiler with a few algorithms, indicating how well the compiler performs and what improvements could be done in order to improve this performance.

I would like to express my gratitude to Srinivasan Balakrishnan for his supervision at Philips Research, Kees van Berkel for his role as my university supervisor, Rick Nas for his aid on various topics and finally Albert van der Werf for allowing me to perform my graduation project in the ESAS group.

Last but not least I want to thank my parents, my brother and my girlfriend for their support and care. They have always stimulated me to aim for the highest and without their support I would certainly not have come this far.
Contents

1 Introduction ................................................. 1
   1.1 General introduction .................................. 1
   1.2 The Co-Vector-Processor (CVP) ......................... 1
   1.3 The Open64 compiler ..................................... 4
   1.4 Problem statement and objectives ....................... 5
   1.5 Related work ............................................ 6
       1.5.1 Cray .............................................. 6
       1.5.2 IRAM .............................................. 7
       1.5.3 GCC .............................................. 7
       1.5.4 VAST-C ........................................... 7
       1.5.5 Intel ............................................. 7
       1.5.6 Superword-level parallelism ......................... 7

2 Basic vectorization theory ................................. 8
   2.1 Notation ............................................... 8
   2.2 Vectorization .......................................... 9
   2.3 Dependences ........................................... 9
       2.3.1 Flow dependences ................................ 9
       2.3.2 Anti-dependence ................................ 10
       2.3.3 Output dependence ................................ 10
       2.3.4 Dependences and vectorization .................... 10
   2.4 Dependence graphs ...................................... 11
       2.4.1 The array dependence graph ....................... 11
       2.4.2 The statement dependence graph ................... 12
   2.5 Vectorization techniques ............................... 13
       2.5.1 Stripmining ........................................ 13
       2.5.2 Scalar Expansion .................................. 15
       2.5.3 Partial Vectorization .............................. 17
       2.5.4 Statement Reordering ............................... 19

3 Scalar Expansion expanded ................................. 21
   3.1 Problem definition .................................... 21
   3.2 Proposed solution ..................................... 22
   3.3 Transformation derivation .............................. 23
       3.3.1 Introduction ...................................... 23
       3.3.2 Derivation ........................................ 23
       3.3.3 Generalization .................................... 27

4 Implementation .............................................. 30
   4.1 The Open64 architecture ............................... 30
   4.2 Important data-structures .............................. 30
       4.2.1 The array-dependence graph (ADG) ................ 31
       4.2.2 The statement-dependence graph (SDG) ........... 31
       4.2.3 The control flow graph (CFG) ..................... 31
       4.2.4 The Def-Use and Use-Def chains ................... 32
       4.2.5 The DO_LOOP_INFO (DLI) structure ................ 33
   4.3 Implementation of the vectorizing transformations ... 33
       4.3.1 Conditions for vectorization ...................... 33
       4.3.2 Stripmining ....................................... 34
       4.3.3 Scalar Expansion .................................. 34
       4.3.4 Partial vectorization .............................. 35
       4.3.5 Statement reordering ................................ 35
4.4 WHIRL2C code generation ..................................... 36
  4.4.1 Vector loops .............................................. 36
  4.4.2 Operators ................................................ 36
  4.4.3 Loads and stores of variables ........................... 37

5 Vectorized algorithms .............................................. 38
  5.1 FIR filter .................................................... 38
    5.1.1 Non-vectorized Implementation ....................... 38
    5.1.2 Vectorization opportunities ........................ 38
  5.2 Golay correlator ............................................ 41
    5.2.1 Non-vectorized implementation ....................... 41
    5.2.2 Vectorization opportunities ........................ 42
  5.3 Vertical peaking ........................................... 46
    5.3.1 Non-vectorized implementation ....................... 46
    5.3.2 Vectorization opportunities ........................ 47

6 Conclusions ...................................................... 49
  6.1 Achievements ............................................... 49
  6.2 Suggestions for future study ............................ 50

A Golay vector code .............................................. 51

B Peaking vector code ........................................... 53

References ........................................................ 55
1 Introduction

1.1 General introduction

The software modem research project is conducted in the Embedded Systems Architectures on Silicon (ESAS) group at Philips Research Eindhoven. It developed a vector processor (CVP) intended to serve as a low-cost, low-power programmable solution for the baseband processing involved in various third generation wireless communication standards [1].

In order to meet the computational requirements of these standards while at the same time dissipating as little power as possible, a high degree of parallelism is used. This involves both data and instruction level parallelism.

In these algorithms some form of data parallelism may implicitly be present. The discovery and subsequent exploitation of the data level parallelism present in these algorithms, called vectorization, is currently the responsibility of the programmer.

This thesis covers the automation of this process with a so-called vectorizing compiler. There are several arguments to automate the process of vectorization:

- Even experienced programmers might miss opportunities for vectorization.
- It reduces the effort of the programmer.
- It enables porting of code across multiple implementations of CVP. For example, the vector length might change.

Chapter 2 describes some basic vectorization techniques. Chapter 3 describes a vectorization technique named Scalar Expansion in more detail and extends its applicability. Chapter 4 describes the implementation of an experimental vectorizing compiler using the technique of the previous chapters in more detail. Chapter 5 describes some algorithms which have successfully been vectorized using this vectorizing compiler and how efficiently they are vectorized. We will name criteria for efficiency later. Finally chapter 6 covers the conclusion and suggestions for future study on the subject.

1.2 The Co-Vector-Processor (CVP)

Most modern processors, such as the Intel Pentium line of processors, are scalar processors (although they do sometimes contain a few vector instructions). In order to explain the key differences between a scalar processor and a vector processor such as the CVP, we first describe a scalar processor and then give some types of parallelism (among which vector-parallelism) used to speed up performance of processors.

A scalar processor is only able to manipulate one or two data elements (a data element is a single value that is stored somewhere in the memory of the architecture) at a time. For example, it can add the value of variable A to the value of variable B and store the result of that operation in C. In order to do this, the processor must first fetch the variables A and B from memory, perform the arithmetic operation 'add' and store the result in C. These operations are encoded in instructions. Therefore a processor first has to decode the instruction and eventually load/store values to/from memory and perform arithmetic operations.

Two such instructions are performed consecutively in figure 1. Instruction-decoding and loading variables from memory both increase the duration of the entire operation.
In order to speed up computations on processors, different types of parallelism can be exploited. We discuss three types of parallelism:

- Pipeline Parallelism
- Very-Long-Instruction-Word (VLIW-)Parallelism
- SIMD Parallelism

Almost all processors use pipelining. Pipelining is a technique whereby multiple instructions are overlapped in execution. A pipeline typically consists of multiple stages, such as instruction decoding, fetching operands and performing arithmetic calculations. As soon as a stage has finished its job, it can immediately start a new job. For example, as soon as an instruction has been decoded, the next instruction can already be decoded. This concept is depicted in figure 2. Assuming each of the stages (which are executed in parallel) takes one unit of execution time, often referred to as a cycle, we can now produce a result every cycle instead of every three cycles.

In order to increase the instruction-level parallelism even further, VLIW-parallelism can be used. For example, a processor can contain two arithmetic units (in VLIW terms also named functional units). A VLIW-architecture packs a number of instructions, each addressing a functional unit, together in a single instruction (hence the term VLIW). For example, a VLIW instruction can specify that one arithmetic functional unit should add A and B and another arithmetic functional unit should add C and D. This will allow the processor to perform these two additions at the same time. However, the result of the second addition could depend on the result of the first. If so, the second addition unit should 'stall' until the first completes. In a conventional processor the processor itself needs to detect when to stall - it is implemented in hardware. In a VLIW-architecture any dependencies are determined
Figure 3: Computations on a scalar processor and a vector processor

by the compiler. It is the job of the compiler to make sure that the instructions respect the dependencies of the calculation performed and to determine which functional units may operate in parallel. Therefore VLIW-architectures do not need the additional hardware the conventional processors need.

Vector processors take this concept yet one step further. Instead of manipulating only one or two data elements like a scalar processor, a vector processor can perform operations on many more elements at once. This is also called data parallelism or Single-Instruction-Multiple-Data (SIMD). A vector processor typically has instructions that specify an operation to be performed on a vector of elements.

Suppose for example that we have two arrays A and B with 4 data elements and we want to add them pair-wise and store the result in C, i.e. $C[0] = A[0] + B[0]$, $C[1] = A[1] + B[1]$, etc etc. On a scalar processor, this calculation would require four additions, eight loads from memory and four stores back to memory. If we assume that a load, an addition and a store are each represented by a single instruction, this computation could be written in $4 \times (1 + 2 + 1) = 16$ instructions for a scalar machine.

On a vector processor such as the CVP, this calculation can be encoded in a single instruction. This instruction loads the four values of A into a vector register, loads the four values of B into a vector register, performs a vector add (which adds the values in the vector registers pair-wise) and finally stores four values back in C. Typically, the single instruction on the vector processor is executed a lot faster than the 16 instructions on the scalar processor. The difference between a scalar and a vector processor in this computation is shown in figure 3. In that figure $A[0 : 3]$ refers to the vector comprised of elements 0, 1, 2 and 3 of the array A. The number of elements in a vector is called the vector length or $V_L$. In the example of figure 3 the vector length is four.

Most vector processors, including the CVP, are only able to load elements that are consecutive in memory as a vector. Such accesses are called ‘unit stride’ or ‘accesses with stride 1’. For example, it would not be possible to directly load the values $A[0], A[12], A[16]$ and $A[30]$ in a single vector, assuming the array elements of A are laid out serially in memory.
The CVP implements each of these three parallelism types: it features multiple, pipelined, functional units (addressable with VLIW-instructions), each of which can operate on vectors of data elements. Note that the CVP also contains a scalar datapath, which means it can also work with scalar variables.

Typically we abstract away from these low-level instructions and program applications for these processors in a high-level programming language such as C. The CVP processor is programmed using a language called CVP-C. It extends the C programming language with vector types and so-called vector intrinsics. These vector intrinsics represent vector operations that can be executed on the CVP. For example, there is a vector intrinsic named `vadd(v1, v2)` which adds the two vectors `v1` and `v2` pair-wise.

We use pointer arithmetic to read from large vector buffers in memory. For example, if we want to read two vectors of four elements each (\(V_L = 4\)) from an array `A` (which is laid out serially in memory) and a pointer named `A_ptr` is pointing to the first element of the array `A`, we can do this as follows:

```c
// Assume A_ptr points to A[0]
// `v_t` is the type for a vector in CVP-C
v_t First_Vector = *(A_ptr);  // Loads elements A[0:3] in First_Vector
A_ptr = A_ptr + 1;           // The pointer now points to A[4]
```

A vector load is aligned if the first element of the vector to be loaded is a multiple of the vector size \(V_L\). For example, if \(V_L = 4\) such as in the example above, vector loads starting at element `A[0]` or `A[12]` are aligned, whereas vector loads starting at element `A[3]` or `A[6]` are unaligned. Unaligned loads typically require more cycles.

In order to be able to apply the vectorization techniques described in this thesis, the vector processor must meet the following requirements:

- The vector processor must support the most basic arithmetic operations on vectors, such as addition, subtraction, multiplication and division.
- The vector processor must be able to do unaligned loads. Some algorithms require unaligned loads. Stores may be aligned.
- The vector length \(V_L\) must be known at compile-time.
- The vector processor must support basic scalar operations, including address calculation as shown in the CVP-C pointer example.

This thesis is therefore certainly not specific to CVP.

1.3 The Open64 compiler

Open64 is a suite of optimizing compiler development tools for Intel Itanium systems running Linux [2]. The Open64 project is the continuation of the SGI Pro64 compiler suite which was released under the GNU General Public License (GPL) [3]. This means the source is freely available to anyone and may be modified under the terms of the GPL.

The Open64 compiler suite contains a frontend for the C, C++, and Fortran 90/95 languages. The frontend can translate each of these input languages to a tree-based, hierarchical intermediate language named WHIRL. Several modules in the backend then have the opportunity to optimize this WHIRL tree before it is finally translated to machine code. More information about the Open64 compiler and its architecture can be found in chapter 4.
1.4 Problem statement and objectives

The problem statement of this graduation project is as follows:

Given a (scalar) program written in C, automatically vectorize it by using an experimental vectorizing compiler with state-of-the-art vectorization techniques and generate vector code in the CVP-C language.

The problem statement is shown in figure 4. Note that the CVP-C code that is generated by the vectorizing compiler is only a part of the chain. The generated CVP-C is the input for yet another tool, the High Level Assembler (HLA), which finally emits code that is executable on CVP. An assumption of this project is that the HLA will take care of both pipelining and VLIW-Parallelism and the vectorizing compiler can focus strictly on SIMD parallelism. This partitioning of the three areas is depicted in figure 5.

The first goal of this graduate project is to investigate existing vectorization techniques. Most vectorization techniques aim at vectorizing loops with array accesses, because repeatedly applying operations on consecutive array elements could indicate these operations can be applied on a vector instead, as a vector can be seen as an array of elements. These techniques are state-of-the-art in vectorization and are for example described in [4], [5] and [6].

The second goal of this project is to implement these vectorization techniques in an existing compiler framework in order to create an experimental vectorizing compiler. It is experimental because it only contains a subset of all available vectorization techniques. An existing framework is to be used because the development of a new compiler would be too time-consuming and beyond the scope of this project. Open64 was chosen as this compiler framework because it allows source-to-source
transformation: An input file written in C/C++ can be translated to a WHIRL
tree, then (vectorizing) transformations can be done on the WHIRL tree, which fi-
ally can be translated back to C or Fortran. This is important because this allows
us to 'feed' the compiler C-code, vectorize it on the WHIRL intermediate language
level and from that generate CVP-C code again. This exactly matches the problem
statement shown in figure 4. If applied on Open64, the problem can be approached
as seen in figure 6.

The third goal is to test the implementation of the experimental compiler using
a few simple but relevant programs as input. These programs are the FIR filter,
the golay correlator and a video peaking algorithm. Each of these is described in
more detail in chapter 5.

Any of these three programs (which are known to be vectorizable) might not be
vectorized by the first implementation of the experimental compiler, which is using
only some of the state-of-the-art vectorization techniques. The fourth goal therefore
is innovation, to research new techniques which allow for successful vectorization of
these three algorithms.

The fifth goal of the project is to evaluate the CVP-C code generated by the
experimental compiler. This is done by comparing the compiler-generated CVP-C
code of each of the algorithms mentioned with CVP-C code that is written by a
programmer. Some criteria that can be used to evaluate these programs are:

- The number of cycles each of the CVP-C programs take;
- A comparison of the number of (unaligned) loads and the number of arithmetic
  operations used by each of the CVP-C programs;
- An analysis of human-invented optimizations that the compiler does not apply
  and what the impact on performance is as a result of that.

1.5 Related work

Vector architectures and their corresponding vectorizing compilers date back even
to the seventies. Originally vector architectures were only used in super-computers
such as the Cray [7]. For a long time super-computers were considered to be their
only application. However, because of the enormous increase of multimedia
applications during the last few years, many modern architectures such as the Intel
Pentium line again feature special vector instructions for multimedia. The key ob-
ervation is that multimedia applications often repeat the same set of operations
over an input sequence of video frames or audio samples, thus allowing for data
parallelism [8]. Here we describe some related work to indicate that the research
area of vectorization and vectorizing compilers is popular and important.

1.5.1 Cray

Cray [9] is one of the pioneers in both vector architectures and vector compilers.
While CDC and Texas Instruments produced the first vector machines, both were
memory to memory architectures. Therefore each instruction required one or more
slow memory access. Cray introduced the CRAY-1 in 1976, which was the first vector architecture to use vector registers [7] and therefore a lot faster than the CDC and TI machines. In addition, it was also the fastest scalar machine in the world at that time and hence very successful. With this machine came the Cray Fortran (CFT) compiler, which was the first automatically vectorizing compiler for Fortran.

1.5.2 IRAM

The IRAM research project done at the university of Berkely, California aims to understand the entire spectrum of issues involved in designing general-purpose computer systems that integrate a processor and DRAM onto a single chip. The first implementation of an IRAM chip is a vector processor named VIRAM1 [10]. To generate code for the VIRAM1, a vectorizing compiler is being used. It is based on the Cray vectorizing compiler. More information on the compiler design and some benchmarks are described in [11].

1.5.3 GCC

One of the most recent efforts in the area of vectorization is the extension of the well-known GCC compiler with vectorization techniques [12]. Currently only the most basic vectorization techniques are applied under quite restrictive conditions within the compiler. However, more complex vectorization techniques are already planned to be implemented and once complete the (open) source of this vectorizer might be a very valuable resource to the area of vectorization.

1.5.4 VAST-C

Crescent Bay Software currently sells a compiler for the Motorola AltiVec vector architecture. It contains a large number of known transformations and supports the use of `pragma's` (hints to the compiler) to aid the vectorization process [13].

1.5.5 Intel

While Intel provided specific SIMD instruction-sets such as MMX with their Pentium processors, until recently it was up to the programmer to recognize their usability and exploit them in algorithms. For that purpose Intel provided libraries for many common algorithms that used the SIMD-capabilities. The current Intel compiler however supports automatic vectorization [14]. Intel Press has also released a book covering automatic vectorization in June 2004 [15], indicating their renewed interest in vectorization and vectorizing compilers.

1.5.6 Superword-level parallelism

A somewhat different approach to vectorization is described in [16]. Classic vectorization techniques such as the ones described in this thesis are aimed at replacing individual instructions that are executed a number of times inside a loop by vector equivalents. The basic idea of superword-level parallelism is to pack statements together in so-called superwords by unrolling a loop a number of times. Their claim is that slp transformations are simpler and extract more parallelism than traditional vectorizing compilers.
2 Basic vectorization theory

This chapter covers some basic vectorization techniques that can be used by a compiler to automatically vectorize a program. First we define the notation we use to clarify these vectorization techniques. Then we will define some aids which each of the techniques need in order to determine whether it can be applied without changing the semantics of the program.

All these techniques are aimed towards the vectorization of inner loops which contain reads and writes to array variables. Since such a loop often serially reads or writes an array variable, opportunities might exist for reading or writing this array variable in parallel - as a vector.

2.1 Notation

Vectorization was defined in chapter 1 as making implicit data parallelism, if any, explicit in a particular form. Eventually, we would like this form to be CVP-C. For introducing the different vectorization techniques however, we will use a more simple form. This form will be based on Dijkstra's Guarded Command Language (GCL) [17].

Below is a very simple loop in GCL:

Program 1

\[
\begin{align*}
\text{var } A, B : \text{array}[0..64] \text{ of int;} \\
\text{var } n : \text{int;} \\
n := 1; \\
\text{do } n \leq 64 \\
\quad A[n-1] := B[n]; \\
\quad n := n + 1; \\
\text{od}
\end{align*}
\]

The loop corresponds to the following code in C:

\[
\begin{align*}
\text{for (int } n = 1; n \leq 64; n++) \\
\quad A[n-1] = B[n];
\end{align*}
\]

Note that although in each GCL program the loop-counter \(n\) is a scalar variable inside the loop, we will ignore it when discussing vectorization. This is because in these programs \(n\) is purely used for calculating the addresses of the variables.

Since we would like to use vector operations, we will also need to define vector types and vector operations. In table 1 we define the type of operations available.

<table>
<thead>
<tr>
<th>Operator description</th>
<th>Operator semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary operator</td>
<td>vector := vector + vector</td>
</tr>
<tr>
<td>Binary operator</td>
<td>vector := vector + scalar</td>
</tr>
<tr>
<td>Unary operator</td>
<td>vector := vector/ vector</td>
</tr>
<tr>
<td>Vector load</td>
<td>vector := memory[start:stop]</td>
</tr>
<tr>
<td>Vector store</td>
<td>memory[start:stop] := vector</td>
</tr>
</tbody>
</table>

Table 1: Types of vector operations

In GCL, we will write a vector as a set of elements from an array. The vector consisting of array elements 0 to (and including) 63 of the array \(A\), is written as \(A[0:63]\). Thus, \(A[0:63]\) can be seen as a vector containing 64 elements. All (vector
or scalar) operators are written in infix notation using their common symbols, i.e ' + ' for addition.

2.2 Vectorization

Now consider program 1 again. There is some data parallelism present, since instead of assigning only one element of B to A, we might as well assign all elements at the same time by using a vector of length 64:

Program 2

\[ \text{var } A, B : \text{array}[0..64] \text{ of } \text{int}; \]
\[ A[0..63] := B[1..64]; \]

We therefore now have vectorized this loop, by making the available data parallelism explicit in our notation. We store a vector of 64 elements from B into a vector of 64 elements in A. Every GCL statement that has been vectorized is written in bold for the remainder of this thesis.

2.3 Dependences

Statements (source-level statements or intermediate language statements) in a loop might depend on each other in a certain way. For example, one statement may read a variable that is written by another statement. There may be multiple dependences between two statements, since a statement can access more than one variable. There are three types of dependences between statements: flow dependences, anti-dependences and output dependences. In order to express the them formally, we define:

- The READ set READ(S) of a statement S is the set of memory locations (referred to by variables) that may be read by this statement.
- The WRITE set WRITE(S) of a statement S the set of memory locations that may be written by this statement.

2.3.1 Flow dependences

A statement Y is flow-dependent on a statement X if a memory location that is written by X is subsequently used by Y, i.e. \( (\text{WRITE}(X) \cap \text{READ}(Y) \neq \emptyset) \land (X \prec Y) \). Here the \( \prec \) symbol refers to lexical precedence in program-order. An example of flow dependence:

Program 3

\[ \text{var } A, B, C : \text{array}[0..65] \text{ of } \text{int}; \]
\[ \text{var } n : \text{int}; \]
\[ n := 0; \]
\[ \text{do } n < 64 \]
\[ \quad s1 : A[n] := B[n + 1]; \]
\[ \quad s2 : C[n] := A[n]; \]
\[ \quad s3 : n := n + 1; \]
\[ \text{od} \]

We have labelled the statements s1, s2 and s3. Since \( \text{WRITE}(s1) \cap \text{READ}(s2) \equiv \{A[n]\} \) and \( s1 \prec s2 \), s2 is flow dependent on s1.
2.3.2 Anti-dependence

A statement $X$ is anti-dependent on a statement $Y$ if a memory location that is read by $X$ is subsequently written by $Y$, i.e. $(\text{READ}(X) \cap \text{WRITE}(Y) \neq \emptyset) \land (X \prec Y)$. An example of anti-dependence:

Program 4

```plaintext
var A, B, C : array[0..65] of int;
var n : int;
n := 0;
do n < 64
   s1 : A[n] := B[n];
   s2 : B[n] := C[n + 1];
   s3 : n := n + 1;
```

od

Since $\text{READ}(s1) \cap \text{WRITE}(s2) \equiv \{B[n]\}$ and $s1 \prec s2$, s1 is anti-dependent on s2.

2.3.3 Output dependence

A statement $Y$ is output-dependent on a statement $X$ if a memory location that is written by $X$ is subsequently written by $Y$, i.e. $(\text{WRITE}(X) \cap \text{WRITE}(Y) \neq \emptyset) \land (X \prec Y)$. An example of output dependence:

Program 5

```plaintext
var A, B, C, D : array[0..65] of int;
var n : int;
n := 0;
do n < 64
   s1 : A[n] := B[n];
   s2 : D[n] := A[n];
   s3 : A[n] := C[n + 1];
   s4 : n := n + 1;
```

od

Since $\text{WRITE}(s1) \cap \text{WRITE}(s3) \equiv \{A[n]\}$ and $s1 \prec s3$, s3 is output-dependent on statement s1.

2.3.4 Dependences and vectorization

As we will see shortly, dependences between statements can limit the amount of available data parallelism. Consider the following program:

Program 6

```plaintext
var A, B, C : array[0..65] of int;
var n : int;
n := 1;
do n < 65
   A[n - 1] := B[n];
   n := n + 1;
```

od
It would be erroneous to write the program as follows:

Program 7

```plaintext
var A, B, C : array[0..65] of int;
var n : int;
A[0..63] := B[1..64];
```

In this program, 64 values of B are assigned to A even before these 64 values of B are written by the second vectorized statement. In the original program, the value of B used in the \((n+1)^{th}\) iteration is the value assigned in the \(n^{th}\) iteration. If the two vectorized statements are switched, the problem still exists, because the computation of B depends also on the computation of A, i.e. the dependency is cyclic.

Though some special forms of cyclic dependences, which will be discussed later, can in fact be vectorized, in most cases cyclic dependences between statements indicate these statements are not vectorizable. Therefore an important part of the analysis done in order to determine the applicability of vectorization is in dependence determination and analysis.

2.4 Dependence graphs

Dependence relations can often be visualized more clearly in a graph. The Open64 compiler provides dependence graphs at two different levels, the array dependence graph (ADG) at the load/store level of array accesses and the statement dependence graph (SDG) at the statement level. Given the following program:

Program 8

```plaintext
var A, B, C, D : array[0..64] of int;
var n : int;
n := 1;
do n < 64
   s1 : A[n] := B[n];
   s3 : D[n] := C[n+1];
   s4 : B[n] = C[n] + 2;
   s5 : n := n + 1;
```

We will describe the two graphs for this program.

2.4.1 The array dependence graph

The array dependence graph or ADG of program 8 is shown in Figure 7. Every node in the ADG represents a statement in the program. We have labelled the nodes in the ADG in accordance with the labels of program 8.

An edge between two statements indicates that a dependence exists between these two statements. When a node \(p\) has an incoming edge from \(q\), this means that \(p\) is dependent on \(q\). For example, the ADG shows an edge from \(s2\) to \(s4\). This indicates the flow dependence between \(s2\) and \(s4\): \(s2\) assigns variable \(C[n]\), \(s4\) uses the variable \(C[n]\). Therefore \(s4\) depends on \(s2\).
The numbers depicted alongside the edges represent the dependence distance. The dependence distance is the vector difference between the source iteration and the target iteration of a dependence.

In order to compute this difference, we will label the iteration space using index variable iteration vectors. This vector is composed of the loop index variables for each nest of the loop.

\[
  iv = \begin{pmatrix}
    I_1 \\
    I_2 \\
    \vdots \\
    I_n
  \end{pmatrix}
\]

\(I_k\) is the value of the loop index variable for the \(k\)th nested loop of that iteration. For example, in statement \(s1\) the value of \(A[n]\) is written. In the next iteration, \(s2\) reads the value written by \(s1\) in the previous iteration by accessing \(A[n-1]\). The iteration vectors (of size 1, because the loop is not nested) for source and target are:

\[
  iv_{source} = (n - 1)
\]

\[
  iv_{target} = (n)
\]

Thus, \(iv_{target} - iv_{source} = 1\), which is the number 1 alongside one of the edges in the graph. Additionally, \(s2\) reads \(A[n]\) also, which explains the second edge from \(s1\) to \(s2\) with a dependence distance of zero.

2.4.2 The statement dependence graph

The statement dependence graph or SDG of this program is shown in figure 8. The statement dependence graph shows all flow, anti- and output-dependences between statements in a loop in the same way the ADG does. The primary difference between the SDG and the ADG can be seen by observing that there is only one edge from \(s1\) to \(s2\) - the result of the statement abstraction level instead of the array load/store abstraction level. Furthermore, the statement dependence graph will also contain dependences between scalar variables.
The statement dependence graph is used by several loop transformations and as we will see shortly it’s also used by vectorization techniques.

2.5 Vectorization techniques

From the previous it follows that in order to vectorize a loop, two primary requirements must be met:

- Each statement in the loop must be mappable on the types and operators described in section 2.1;
- There may be no dependency cycles involving any two statements inside the loop;

For example, an assignment to a scalar variable inside a loop violates the first requirement (we cannot store a scalar variable with the vector operations shown in the table of section 2.1) and complicates vectorization. The vectorization techniques described here either relax these requirements or transform the program to meet these requirements.

2.5.1 Stripmining

As mentioned in the introduction, we define $V_L$ as the number of elements that can be stored in a single vector. In order to transform a loop with computations on array variables to computations on vector variables (requirement one), the iteration count of that loop must be equal to the vector length $V_L$. To see why this is necessary, consider a loop with $N$ iterations that contains the statement $A[n] := B[n] + C[n]$. We do have a vector operation for this statement, but this operation operates on the fixed vector length $V_L$ and not on any arbitrary length $N$. Since programmers are in general not concerned with any vector length at all, most loops have an iteration count that does not equal the vector length.

In order to transform these loops into loops where the number of iterations equals the vector length $V_L$, a technique named ‘stripmining’ can be used [4]. Stripmining a loop with $N$ iterations is done by introducing an outer loop with $N \div V_L$ iterations, in which the original loop is run repeatedly with $V_L$ iterations. This inner loop is called the strip loop. In case $N$ is not a multiple of $V_L$, the original
loop body is still run for the last $N \mod V_L$ iterations after the outer loop has finished. For example, assuming $V_L = 64$:

**Program 9**

```plaintext
var A, B, C : array[0..100] of int;
var n : int;
n := 0;
do n < 100
   A[n] := B[n + 1];
   C[n] := A[n];
   n := n + 1;
od
```

can be transformed into:

**Program 10**

```plaintext
var A, B, C : array[0..100] of int;
var m, n, m_upper : int;
m := 0;
m_upper := (100 div 64) * 64;
do m < m_upper
   n := m;
do n < m + 64
      A[n] := B[n + 1];
      C[n] := A[n];
      n := n + 1;
   od
   m := m + 64;
od
do n < 100
   A[n] := B[n + 1];
   C[n] := A[n];
   n := n + 1;
od
```

The inner loop of the nested loop now has $V_L$ iterations. If we have determined there are no dependency cycles between the statements, each of the statements inside this loop can now be replaced by its vector equivalent. This operation has become simple now, since the iteration count is equal to $V_L$. This is shown in program 11.

14
Program 11

\[
\begin{align*}
\text{var } A, B, C &: \text{array}[0..100] \text{ of int;} \\
\text{var } m, n, m_{\text{upper}} &: \text{int;} \\
m &: = 0; \\
m_{\text{upper}} &: = (100 \div 64) \times 64; \\
do m < m_{\text{upper}} \\
& \quad A[m : m + 63] := B[m + 1 : m + 64]; \\
& \quad C[m : m + 63] := A[m : m + 63]; \\
& \quad m := m + 64; \\
od \\
do n < 100 \\
& \quad A[n] := B[n + 1]; \\
& \quad C[n] := A[n]; \\
& \quad n := n + 1; \\
od
\end{align*}
\]

This transformation can be done on any loop without further requirements.

2.5.2 Scalar Expansion

Scalar expansion is the replacement of an assignment to a scalar variable by an assignment to an array variable. To see why this transformation is useful, consider the following program:

Program 12

\[
\begin{align*}
\text{var } a, n &: \text{int;} \\
\text{var } B, C &: \text{array}[0..64] \text{ of int;} \\
n &: = 1; \\
do n <= 64 \\
& \quad s1 : a := B[n - 1]; \\
& \quad s2 : C[n] := a; \\
& \quad n := n + 1; \\
od
\end{align*}
\]

The statement dependence graph of this program is depicted in figure 9.

The graph contains a cycle between statements s1 and s2. The dependency of s2 on s1 is easy to recognize - s2 uses the value assigned by s1. The reason s1 also depends on s2 is the existence of an anti-dependency: s1 must be executed after
the execution of s2 in the previous iteration and before the execution of s2 in the current iteration.

In order to remove this cycle, the scalar variable can be converted to an array variable. We can replace any assignment to the scalar variable a with an assignment to A[n], and every use of a scalar variable a with a use of A[n]. Applied on program 12 this technique results in program 13.

**Program 13**

```plaintext
var a, n : int;
var A, B, C : array[0..64] of int;
n := 1;
do n <= 64
   s1 : A[n] := B[n - 1];
   s2 : C[n] := A[n];
   n := n + 1;
```

As can be seen in figure 10, now there is only a flow dependency from s1 to s2 and the program can be vectorized. Program 14 shows the vector code.

**Program 14**

```plaintext
var a, n : int;
var A, B, C : array[0..64] of int;

A[1 : 64] := B[0 : 63];
C[1 : 64] := A[1 : 64];
```

Note that the variable a might be used in other scalar code after the loop has executed. We then say that variable a is live on loop exit. In order to make sure this scalar code uses the correct value of a, we should store the last value assigned to the array replacement of a back to a. In this example, that would mean introducing a statement a := A[64] after the loop exit.

This transformation helps to meet both stated requirements for vectorization: it converts scalar variables to arrays, allowing them to be replaced with vector variables, thereby also eliminating cycles in the dependency graphs.

Scalar Expansion is already implemented in the Open64 compiler, since cycle-elimination can be useful for other (non vector-processor related) optimizations too. In chapter 3 a more detailed analysis of scalar expansion and the requirements for its application are given.
2.5.3 Partial Vectorization

Some programs have a cyclic dependency that cannot be broken by transformations. This does not exclude other statements from vectorization however - the program might still be partially vectorized. Consider the following program:

Program 15

\begin{verbatim}
var A, B, C, D : array[0..65] of int;
var n : int;
n := 1;
do n <= 64
    s1 : A[n] := B[n];
    s3 : D[n] := C[n + 1];
    s4 : B[n] = C[n] + 2;
    s5 : n := n + 1;
\end{verbatim}

The statement dependence graph is given in figure 11. There is a cyclic dependency between s2 and s4: a flow-dependence between statements s2 and s4 due to $C[n]$ and a flow-dependence between s4 and s2 because s4 writes a value ($B[n]$) that is read by s2 in the next iteration ($B[n - 1]$).

This means s2 and s4 cannot be vectorized. That however does not exclude other statements from vectorization.

A cycle in the graph can be defined as a strongly connected region. For any two nodes (V1, V2) in a strongly connected region, there must be a path from V1 to V2. A strongly connected component then is the maximal strongly connected region.

Once we find the strongly connected components in a statement graph, we can exclude the statements they represent from vectorization because we know they are all cyclic dependent on each other. To depict this more clearly, we can compute the so-called acyclic condensed graph. The acyclic condensed graph (ACG), for a graph G is a DAG with a node for each strongly connected component in G. An edge between two nodes in the ACG exists whenever there is an edge between two nodes in G from distinct strongly connected components.

The Acyclic Condensed Graph for the program is shown in figure 12. Indeed statements s2 and s4 form a strongly connected component together and hence must be executed in a non-vectorized loop. Statements s1 and s3 have no cyclic dependencies (in fact they are not dependent on each other at all). Therefore, we can replace s1 and s3 with equivalent vector code. The resulting program is:
Program 16

var $A, B, C, D$ : array[0..65] of int;
var $n$ : int;
$n := 1$;


do $n <= 64$
  $s4 : B[n] = C[n] + 2$;
  $s5 : n := n + 1$;
od
2.5.4 Statement Reordering

Sometimes statements must be reordered to enable vectorization. Consider the following program:

Program 17

```plaintext
var A, B, C, D : array[0..64] of int;
var n : int;
n := 1;
do n <= 64
    s1 : A[n] := B[n - 1];
    s2 : B[n] := D[n];
    n := n + 1;
```

The statement dependence graph of this program is given in figure 13. There are no cyclic dependences, but the statements cannot be executed in vector-mode in the given order. In fact this is what the statement dependence graph tell us - s2 must be executed prior to s1 because s1 depends on s2. For the scalar program, this is no problem since in the first execution of s1 the B-value used is undefined and all subsequent executions of s1 are preceded by s2, thus satisfying the dependency. In vector-mode however, up to \( V_L \) values of B are used by the first execution of s1. \( V_L - 1 \) of these values would have been defined in the scalar program, but not in the vectorized program.

In order to solve this, the statements must be reordered according to the statement dependence graph. This transformation must be done in order to generate correct vector-code. A pseudo-algorithm that does this on a statement graph \( G=(V,E) \):

```plaintext
While there are nodes in V
    - Take a node \( p \) from \( V \) with no incoming edges
    - Remove all outgoing edges of \( V \)
    - Remove \( p \) from \( V \)
    - Add \( p \) at the tail of the ordered list of statements
End
```

This algorithm will only work on graphs without cycles. The resulting statement list is ordered according to the dependences of the statement dependence graph. The resulting vector code for this particular program after statement reordering is shown in program 18.
Program 18

var $A, B, C, D : array[0..64] of int;$

$s2 : B[1 : 64] := D[1 : 64];$
$s1 : A[1 : 64] := B[0 : 63];$
3 Scalar Expansion expanded

When we attempted to vectorize the peaking algorithm, which is described in further
detail in chapter 5, we discovered that the conditions under which scalar expansion
is performed by the Open64 compiler are too restrictive. In [5] and [18] no conditions
for scalar expansion are given at all. This is wrong, because applying scalar
expansion the way it is described in this literature does not work for all scalar vari-
ables, as we will see shortly. Other literature, such as [4] mentions a condition for
scalar expansion. While this condition is indeed necessary to apply scalar expansion
as described in that literature, we can slightly modify the technique to make this
condition obsolete and expand the applicability of the technique.

In this chapter we describe a program which has scalar variables that are not
scalar expanded by the Open64 compiler. First we will explain why they are not
expanded. Then we will propose a solution in the form of a transformation. Finally,
we will derive this transformation using formal methods and prove its correctness.

3.1 Problem definition

Consider the following program:

```
var a, b: int
var n: int
n := 0;

do n < N
   a := b;
b := values[n];
results[n] := a + b;
n := n + 1;

```

Remember that the variable n is used only for address calculations and hence we
ignore it as a scalar variable. The results array is filled with values from the values
array. The variable a is actually a delayed value of b - it is assigned the value b had
in the previous iteration (if any). When this relatively simple program is fed to the
Open64 compiler, it only scalar expands variable a, but not b. As we have seen in
the previous chapter, the presence of an assignment to a scalar variable results in a
cycle in the dependence graph which prevents us from vectorizing this program.

Further investigation on the conditions for scalar expansion of scalar variables
in Open64 shows that if a variable has uses that are not preceded by a definition in
the same iteration, that variable is not scalar expanded. Such a variable is said to
have an 'upward exposed use' [4]. In the given program b has an upward exposed
use, since it is read once before it is assigned in the same iteration. Consider what
would happen if we expanded the scalar b by applying scalar expansion as described
in section 2.5.2:
\textbf{Proposed solution}

In order to vectorize this program, we must remove the cycle corresponding to the scalar variable \( b \). Therefore, we would still like to promote it to an array variable. The key observation is that the scalar value used by the upward exposed use of \( b \) is, except for the first iteration, the assigned value of the previous iteration. This indicates there would at least be a solution for all iterations except the first, namely to use the value of the previous iteration (at index \( n - 1 \)) for all upward exposed uses. Applying this idea to the program, we get:

\begin{verbatim}
var a, b : array[0..N] of int
var n : int
n := 0;

do n < N
  a[n] := b[n - 1];
  b[n] := values[n];
  results[n] := a[n] + b[n];
  n := n + 1;
od
\end{verbatim}

Indeed for all iterations except the first (it accesses \( b[-1] \) there) this program is correct. In order to investigate this transformation more closely, we will derive this transformation using formal techniques such as shown in [17].
3.3 Transformation derivation

3.3.1 Introduction

The method used for this derivation is based on Dijkstra’s program calculus. It uses logic predicates to reason about programs. A predicate that holds before a certain program or statement is called a pre-condition, a predicate that holds after a certain program or statement is called a post-condition. Often programs will be annotated with these logic predicates. Predicates known to be true at a certain point in the program are enclosed by braces, as in:

\[
\begin{align*}
    a &:= 7; \\
    \{a == 7\}
\end{align*}
\]

Another important notation used is that of replacement: Given a predicate P, P(n := n+1) means 'P where n is replaced by (n+1)'. Extensively used in this proof are loop-invariants. Loop-invariants are predicates on variables in a program that, once proven, may be assumed to hold at the beginning of every loop iteration. Proving a loop-invariant is usually done by induction. To prove a loop-invariant correct one must:

- Prove that loop-invariant P holds before the loop is started
- Given a loop-invariant P and a loop with counter n, one should prove that at the end of each iteration P(n := n+1) holds also. In order to prove P(n := n+1), one may assume P is true

Our goal is to replace scalar variables with array variables. We may replace a scalar variable \( x \) with an array variable \( arr.x[y] \), if we make sure that before every statement that reads \( x \) the condition \( \{ arr.x[y] == x \} \) holds. Therefore we will establish conditions that say something about the scalar variables \( a \) and \( b \), such that we can guarantee their array equivalents will contain the same values before each use of \( a \) and \( b \). Once we can guarantee that \( \{ arr.a[y] == a \} \) and \( \{ arr.b[y] == b \} \) before every use of either \( a \) or \( b \), we have proven that the replacement is correct.

3.3.2 Derivation

Consider the following program:

```plaintext
var a, b : int
var n : int
n := 0;

do n < N
    a := b;
    b := values[n];
    results[n] := a + b;
    n := n + 1;
od
```

This program can be seen as a function operating on values from the \( values \) array and storing them to the \( results \) array. We are however not so much concerned with
what exactly this program calculates - we would like to introduce array variables throughout this program for the sake of scalar expansion.

Therefore, we introduce array variables va, and vb. The array variable va[i] will contain the value of a in iteration i, after it has been calculated. The same goes for vb. In order to replace the variables a and b in the statement

\[ \text{results}[n] := a + b; \]

we must be sure that va and vb carry the value of their scalar equivalent before this assignment. We can guarantee this by assigning the the values of the scalars to their corresponding array variables. In the case of a we therefore introduce a statement va[n] := a and likewise we introduce a statement for b.

```plaintext
var a, b : int
var n : int
var va, vb : array[0..N] of int
n := 0;

do n < N
    a := b;
    b := values[n];
    va[n] := a, vb[n] := b;
    \{ va[n] == a \land vb[n] == b \}
    results[n] := va[n] + vb[n];
    n := n + 1;
od
```

We can now express the computation in terms of the variables va and vb.

\[ < \forall i : 0 \leq i < N : \text{results}[i] = \text{va}[i] + \text{vb}[i] > \]

The loop can be proven correct by using loop invariants:

\[ P_0 : < \forall i : 0 \leq i < n : \text{results}[i] = \text{va}[i] + \text{vb}[i] > P_1 : 0 \leq n \leq N \]

Both \( P_0 \) and \( P_1 \) hold before the loop starts (\( P_0(n := 0) == P_1(n := 0) == \text{true} \)) and both also hold at the end of each loop with n replaced by (n+1). However, we would like to replace all occurrences of a and b in the loop by their array representatives. The first statement uses the value of b, which has not been assigned yet in that iteration. It is actually the value that was assigned to b in the previous iteration, or for the first iteration it is the value which might have been assigned to b before entering the loop. We previously defined these uses as 'upward exposed uses'. In order to convert this scalar variable to a corresponding array variable, we must have a pre-condition which maps the values of a and b to their array equivalents. At this point the only thing we know is given by the loop-invariants \( P_0 \) and \( P_1 \). However these do not say anything about the values of a and b and their relation with va and vb. We therefore introduce a new loop invariant \( P_2 \):

24
$P_2 : (n \equiv 0 \lor (va[n-1] \equiv a \land vb[n-1] \equiv b))$

We assume that negative array indices return an undefined value, and that for any value $x$ (undefined $\equiv x$) $\equiv$ false. First, we will prove that $P_2$ holds before the loop:

$P2(n := 0) \\
\equiv \{Replacement\} \\
(0 \equiv 0 \lor (va[-1] \equiv a \land vb[-1] \equiv b) \\
\equiv \{Calculus\} \\
true$

Now we must prove that $P_2$ is invariant under the increment of the index $n$:

$P_2(n := n + 1) \\
\equiv \{Replacement\} \\
(n + 1 \equiv 0 \lor (va[n] \equiv a \land vb[n] \equiv b) \\
\equiv \{P_2, n > 0, Calculus\} \\
(va[n] \equiv a \land vb[n] \equiv b)$

Therefore, we must establish the post-condition

$\{va[n] \equiv a \land vb[n] \equiv b\}$

in order for the loop invariant $P_2$ to hold. Clearly, the assignment:

$va[n] := a, vb[n] := b;$

establishes this, and since it is not falsified by the assignment to results[$n$] we may conclude this is indeed a post-condition of the loop of the current program. Therefore we conclude this invariant is valid. The resulting program is as follows:

```
var a, b : int
var n : int
var va, vb : array[0..N] of int
n := 0;
{P_0 : \forall i : 0 \leq i < n : results[i] = va[i] + vb[i]} \\
{P_1 : 0 \leq n \leq N} \\
{P_2 : (n \equiv 0 \lor (va.(n-1) = a \land vb[n-1] \equiv b))}

do n < N \\
{P_0 \land P_1 \land P_2} \\
a := b; \\
b := values[n]; \\
va[n] := a, vb[n] := b; \\
{va[n] = a \land vb[n] = b}
```
\[
\begin{align*}
\text{results}[n] & := \text{va}[n] + \text{vb}[n]; \\
\{P_0(n := n + 1) \land P_1(n := n + 1) \land P_2(n := n + 1)\} \\
n & := n + 1;
\end{align*}
\]
\[\{P_0 \land P_1 \land P_2 \land n = N \Rightarrow \forall i : 0 \leq i \leq N : \text{results}[i] = \text{va}[i] + \text{vb}[i] \}\]

In all cases where \( n \) is greater than zero, we may safely conclude that \( \{(\text{va}.(n-1) == a \land \text{vb}[n-1] == b)\} \) and therefore we can now, in those cases, replace the assignment of \( b \) to \( a \) by the array equivalent \( \text{vb}[n-1] \). The loop invariant \( P_2 \) is however not strong enough; in the first iteration, where \( n \) is zero, we cannot deduce that \( \text{va} \) and \( \text{vb} \) contain the correct values. Therefore, we split off the first iteration of the loop and 'peel' it outside the loop. We duplicate the loop body above the loop. For simplicity, we will assume \( N \geq 0 \), which implies the first iteration will always be executed. The program is as follows:

\[
\begin{align*}
\textbf{var} & \ a, b : \text{int} \\
\textbf{var} & \ n : \text{int} \\
\textbf{var} & \ \text{va, vb} : \text{array}[0..N] \ of \ \text{int} \\
n & := 0; \\
\{\text{Duplication of the loop body, } N \geq 0\} \\
a & := b; \\
b & := \text{values}.0; \\
\text{va}.0 & := a, \text{vb}.0 := b; \\
\{\text{va}.0 = a \land \text{vb}.0 = b\} \\
\text{results}.0 & := \text{va}.0 + \text{vb}.0; \\
n & := n + 1; \\
\{P_0 : < \forall i : 0 \leq i < n : \text{results}[i] = \text{va}[i] + \text{vb}[i] >\} \\
\{P_1 : 0 \leq n \leq N\} \\
\{P_2 : (n == 0 \lor \text{va}.(n-1) == a \land \text{vb}[n-1] == b))\} \\
\textbf{do} & \ n < N \\
\{P_0 \land P_1 \land P_2 \land n > 0\} \\
a & := \text{vb}[n-1]; \\
b & := \text{values}[n]; \\
\text{va}[n] & := a, \text{vb}[n] := b; \\
\{\text{va}[n] == a \land \text{vb}[n] == b\} \\
\text{results}[n] & := \text{va}[n] + \text{vb}[n]; \\
\{P_0(n := n + 1) \land P_1(n := n + 1) \land P_2(n := n + 1)\} \\
n & := n + 1; \\
\{P_0 \land P_1 \land P_2 \land n == N \Rightarrow < \forall i : 0 \leq i \leq N : \text{results}[i] = \text{va}[i] + \text{vb}[i] >\}
\end{align*}
\]

Indeed we see that the duplication of the loop body does not violate the loop invariants and therefore the calculated result will still be the same. Given our definition for replacement, we may now replace the uses of \( a \) and \( b \) by the array values they are known to have:

\[
\begin{align*}
\textbf{var} & \ a, b : \text{int} \\
\textbf{var} & \ n : \text{int} \\
\textbf{var} & \ \text{va, vb} : \text{array}[0..N] \ of \ \text{int}
\end{align*}
\]
\[ \begin{align*} n & := 0; \\
\{ \text{Duplication of the loop body} \} \\
a & := b; \\
b & := \text{values}.0; \\
va.0 & := a, vb.0 := b; \\
\{ va.0 == a \land vb.0 == b \} \\
results.0 & := va.0 + vb.0; \\
n & := n + 1; \\
\{ P_0 : \forall i : 0 \leq i < n : results[i] = va[i] + vb[i] \} \\
P_1 & : 0 \leq n \leq N \\
P_2 & : (n == 0 \lor (va.(n-1) == a \land vb.(n-1) == b)) \\
do \ n < N \\
\{ P_0 \land P_1 \land P_2 \land n > 0 \} \\
a & := vb[n-1]; \\
b & := \text{values}[n]; \\
\{ a = vb[n-1] \land b = \text{values}[n] \} \\
va[n] & := vb[n-1], vb[n] := \text{values}[n]; \\
\{ va[n] = a \land vb[n] = b \} \\
results[n] & := va[n] + vb[n]; \\
\{ P_0(n := n + 1) \land P_1(n := n + 1) \land P_2(n := n + 1) \} \\
n & := n + 1; \\
\od \\
\{ P_0 \land P_1 \land P_2 \land n == N \Rightarrow \forall i : 0 \leq i \leq N : results[i] = va[i] + vb[i] \} \end{align*} \]

In the loop, the scalar variables \( a \) and \( b \) are no longer used for computing the result. Therefore, we may safely remove the assignments to these variables. This leaves us a loop with only array statements (except for the counter), which is what we originally set out for. In fact, the scalar variables from the pooled first iteration can be removed also by assigning them directly to their array equivalents.

3.3.3 Generalization

Reasoning in a manner similar to that in the previous section, we can prove that programs of the form:

\[ \begin{align*} \text{do } & n < N \\
& \ldots := \theta; \\
& a := \ldots; \\
& \ldots := \theta; \\
& \ldots \end{align*} \]

\[ n := n + 1; \]

\[ \od \]

can be transformed into:

\[ \]
\[\ldots := a;\]
\[va.0 := \ldots;\]
\[\ldots := va.0;\]
\[\ldots;\]
\[n := n + 1;\]

\textbf{do} \ n < N
\[\ldots := va[n - 1]; \; \{Upward\;Exposed\;Use\}\]
\[va[n] := \ldots;\]
\[\ldots := va[n];\]
\[\ldots;\]
\[n := n + 1;\]
\textbf{od}

For this transformation to work we do have to assume that the program contains no conditional assignments to \(a\) (we will see why that is necessary shortly). In general, we may conclude that expansion of scalar variables to array variables is possible if we introduce an additional loop-invariant, which states which array index contains the value of the scalar variable in the previous iteration. Usually the replacing array variable will be indexed by \(\text{loop.counter} - 1\) for all upward exposed uses. All writes to the scalar expanded variable will be indexed by \(\text{loop.counter}\). From this it follows that all ‘normal’ uses, which are preceded by a write in the same iteration, can be indexed by \(\text{loop.counter}\) also. The example shown above would use the invariant \(P_2: va[n - 1] = a\).

Suppose, however, that an assignment to \(a\) is conditional:

\textbf{do} \ n < N
\[\ldots := a;\]
\[\text{if} \ (\text{cond})\]
\[\quad a := \ldots\]
\[\text{fi}\]
\[\{We\;need\;to\;guarantee\;va[n] = a\;in\;order\;to\;replace\;a\;with\;va[n]\}\]
\[\ldots := a;\]
\[n := n + 1;\]
\textbf{od}

If we simply replace all assignments and uses of \(a\) we would get:

\textbf{do} \ n < N
\[\ldots := va[n - 1];\]
\[\text{if} \ (\text{cond})\]
\[\quad va[n] := \ldots\]
\[\text{fi}\]
\[\{We\;cannot\;guarantee\;va[n] = a\;in\;order\;to\;replace\;a\;with\;va[n]\}\]
\[\ldots := va[n]; \{Thus\;this\;is\;incorrect!\}\]
\[n := n + 1;\]
\textbf{od}
However because of the conditional assignment we can no longer guarantee $P_2$. This behavior is due to the fact that scalar variables keep the same value over iterations even when they are not assigned again. The replacing array variables which are indexed by the loop-index however do not keep the same values over iterations. Therefore, at the beginning of the loop, we introduce a statement $va[n] := va[n - 1]$. This will already satisfy the loop-invariant $P_2(n := n + 1)$ for the next iteration, because $va[n] == va[n - 1] == a$. We specify the following program, in which we have introduced that extra statement:

\[
\begin{align*}
d & n < N \\
\{ va[n - 1] == a \} \\
va[n] & := va[n - 1] \\
\{ va[n] == a \therefore P_2(n := n + 1) \}
\end{align*}
\]

$S_1$ may contain any sequence of statements, for example the entire loop body of the previously shown program with conditional assignments.

We must prove $S_1$ will not invalidate the condition $P_2(n := n + 1)$, which we have already established. The only statements that will invalidate this invariant are assignments to either $a$ or $va[n]$. Suppose $S_1$ contains an assignment $a := value$. We will then introduce a statement $va[n] := value$ immediately below this assignment, thus satisfying $P_2(n := n + 1)$ again. Since all assignments to $va[n]$ are introduced this way, no assignment to $va[n]$ will violate $P_2(n := n + 1)$. Thus we may conclude $P_2(n := n + 1)$ is a valid postcondition for $S_1$ and that $P_2$ is loop-invariant. The assignments $a := value$ and $va[n] := a$ can then be merged into a single assignment $va[n] := value$.

By applying this technique we can scalar expand scalar variables even if they have upward exposed uses and conditional assignments. This will result in the elimination of cycles in the dependence graph and therefore aids vectorization.
4 Implementation

This chapter discusses the implementation of the described vectorization techniques in the Open64 compiler, resulting in an experimental vectorizing compiler. First a short description of the existing Open64 compiler architecture and implementation is given. Then the intermediate language of Open64, on which all transformations are performed, is discussed. The Open64 compiler also contains some important data-structures that need to be maintained by all transformations. Finally, the actual implementation of the vectorization techniques is described.

4.1 The Open64 architecture

Open64 is a very modular compiler. For the interface between several modules the compiler uses an intermediate language called WHIRL. WHIRL is hierarchical in that it has different levels. The front-end of the compiler translates C/C++/Fortran programs to Very High WHIRL, the highest level of the Intermediate Language (IL). At this point, the IL still very much resembles the original source program. In later phases, the compiler transforms this IL to High, Mid, Low and finally Very Low WHIRL. This process is also named 'lowering stages'. The lower the level, the more close it is to the target architecture.

Besides the hierarchical nature of WHIRL it is also tree-oriented. At each of the five WHIRL levels, the program under transformation is represented in a so-called Whirl Node (WN) tree. The WN is therefore the primary data-type of the intermediate language. Each WN contains an opcode, indicating which type of treenode it is. A WN may also have any number of children. For many program constructs, such as addition, the number of children and their relative positions are fixed. For more information concerning the WHIRL language, we refer the reader to [22].

The Open64 compiler features a few important modules:

- The Inter-Procedural Analysis (IPA), a module that analyzes procedures and optimizes them. For example, it may determine that the argument of a function is a constant and use this fact in that function. The IPA module contains roughly 150000 lines of code.

- The Loop Nest Optimizer (LNO), a module that analyzes loops and implements several optimizations for them, such as common subexpression elimination, dead-code elimination and loop fission (splitting). The LNO module contains roughly 190000 lines of code.

- WHIRL2c, a module that can translate Very High and High WHIRL back to C-code. This module contains roughly 30000 lines of code.

The program flow of the passes in the Open64 compiler framework is graphically depicted in figure 14.

4.2 Important data-structures

The Open64 compiler maintains a few important data-structures. Since program transformations often change the structure of the program, they must modify these data-structures to reflect these changes. For example, a statement s1 is removed from a loop because the result it computes is not used by any other statements. The result s1 computes is dependent on s2 and this dependency is shown in the ADG and SDG. Upon removal of the statement s1, both the ADG and SDG need to be adapted.
4.2.1 The array-dependence graph (ADG)

The array dependence graph was described previously in chapter 2. In the LNO-phase of the compiler an array dependence graph is built by calling `Build_Array_Dependence_Graph()`. It is kept up-to-date by all array related transformations.

4.2.2 The statement-dependence graph (SDG)

The statement dependence graph was also described in chapter 2. The statement dependence graph for any loop can, unlike the ADG, be generated at any time in the LNO-phase by calling `Build_Statement_Dependence_Graph()`. An Acyclic Condensed Graph can be generated from the SDG at any time by calling `Acyclic_Condensation()` on a SDG.

4.2.3 The control flow graph (CFG)

The control flow graph is an abstract representation of a procedure or program. Each node in the graph represents a basic block. A basic block contains zero or more statements, none of which may be jumps or jump targets. Edges are then used
to represent jumps in control flow. For example, a simple if-then-else-construct can be represented in a CFG as shown in figure 15.

The CFG is constructed automatically by Open64 when calculating the def-use chains described in the next section.

4.2.4 The Def-Use and Use-Def chains

The Open64 compiler maintains both Def-Use and Use-Def chains, which keep information about definitions and their uses. A definition of a variable reaches a use if there is a path in the CFG from the definition to the use that does not contain any other definitions of that same variable. We will illustrate a def-use chain for the following program:

Program 19

\[
\begin{align*}
\text{var } & a, b : \text{int;} \\
b & := 4; \\
a & := 0; \\
\text{if } & b > a \\
& a := a + 1; \\
b & := b + 3; \\
& a := b \times 2; \\
\text{fi} \\
a & := a + 4;
\end{align*}
\]

First, we will label each definition and use of the variables with a subscript in order to be able to refer to a specific definition or use of a particular variable.

Program 20

\[
\begin{align*}
\text{var } & a, b : \text{int;} \\
b_0 & := 4; \\
a_1 & := 0; \\
\text{if } & b_2 > a_3 \\
& a_5 := a_4 + 1; \\
b_7 & := b_6 + 3; \\
& b_8 <= a_9 \{\text{This is the else condition}\} \\
a_{11} & := b_{10} \times 2; \\
\text{fi} \\
a_{13} & := a_{12} + 4;
\end{align*}
\]

Below is a table with reaching definitions for each use:

<table>
<thead>
<tr>
<th>Use</th>
<th>Reaching definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_2)</td>
<td>(b_0)</td>
</tr>
<tr>
<td>(a_3)</td>
<td>(a_1)</td>
</tr>
<tr>
<td>(a_4)</td>
<td>(a_1)</td>
</tr>
<tr>
<td>(b_8)</td>
<td>(b_0)</td>
</tr>
<tr>
<td>(a_9)</td>
<td>(a_1)</td>
</tr>
<tr>
<td>(b_{10})</td>
<td>(b_0)</td>
</tr>
<tr>
<td>(a_{12})</td>
<td>(a_6, a_{11})</td>
</tr>
</tbody>
</table>
Observe that because of the if-then-else construct in the program, the compiler cannot link the use $v_{12}$ to a single definition. The compiler also keeps a reverse table, which links each definition to zero or more uses. The use-def and def-use chains are constructed before the LNO is invoked and therefore must be kept up-to-date by all transformations. The compiler offers an interface with some functions for this. The most important function for that end is $\texttt{Add\_Def\_Use()}$, which links a use (identified by a WN) to a definition (also identified by a WN). It updates both the def-use chain and the use-def chain. Def-use and use-def chains are used by several program transformations, such as scalar expansion, in the Open64 compiler.

### 4.2.5 The \texttt{DO\_LOOP\_INFO} (DLI) structure

The DLI-structure is a structure mapped to each Do-Loop. It contains a lot of information about the loop. The most important are given below, where bold names indicate fields added in the experimental compiler:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>The lower bound of the loop</td>
</tr>
<tr>
<td>UB</td>
<td>The upper bound of the loop</td>
</tr>
<tr>
<td>Step</td>
<td>The step size of the loop</td>
</tr>
<tr>
<td>Is_Inner</td>
<td>Whether the loop is an inner loop</td>
</tr>
<tr>
<td>Has_Calls</td>
<td>Whether the loop contains calls to functions</td>
</tr>
<tr>
<td>Has_Gotos</td>
<td>Whether the loop contains goto statements</td>
</tr>
<tr>
<td>Has_Exits</td>
<td>Whether the loop has any premature exits</td>
</tr>
<tr>
<td>Is_Vect_Candidate</td>
<td>Whether the loop is a candidate for vectorization</td>
</tr>
<tr>
<td>Is_Vectorizable</td>
<td>Whether the loop is vectorizable</td>
</tr>
</tbody>
</table>

This structure can provide us with some of the information needed to determine whether a loop is a candidate for vectorization. When we create a new loop in a transformation, we need to create a new \texttt{DO\_LOOP\_INFO} structure and call $\texttt{LNO\_Build\_Do\_Access()}$, which fills in the structure.

### 4.3 Implementation of the vectorizing transformations

Since the LNO module maintains a lot of information, such as dependence graphs for every loop in the program, this is the most logical place to implement vector optimizations and transformations. After the LNO and Vector optimizations, we route the resulting High WHIRL to the WHIRL2C module, thus generating C-code. Additionally, we modify the WHIRL2C module to output vector intrinsics whenever we translate a loop that is vectorizable. The main vectorization part can be found in \texttt{osprey1.0/be/lno/vector_opts.cxx}.

#### 4.3.1 Conditions for vectorization

Before a loop is even a candidate for vectorization, the experimental compiler checks a number of conditions. Some conditions are inherent to the problem of vectorizing and therefore \textbf{hard} conditions. Others conditions simply exist because they indicate program code for which vectorization techniques are not implemented yet, but may be in the future. These conditions are given in the table below:
<table>
<thead>
<tr>
<th>Condition</th>
<th>Hard condition?</th>
</tr>
</thead>
<tbody>
<tr>
<td>The loop must be an inner loop</td>
<td>No</td>
</tr>
<tr>
<td>The loop may not contain conditionals</td>
<td>No</td>
</tr>
<tr>
<td>The loop must have a constant step of 1</td>
<td>No</td>
</tr>
<tr>
<td>The loop must have a simple (constant) lower bound</td>
<td>No</td>
</tr>
<tr>
<td>All data-types used in the loop must be of equal size</td>
<td>No</td>
</tr>
<tr>
<td>The loop may not contain goto's, premature exits or procedure calls</td>
<td>Yes</td>
</tr>
<tr>
<td>The number of loop iterations must at least be equal to ( V_L )</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 4.3.2 Stripmining

Stripmining is almost always a necessary transformation if we want to vectorize a loop. Stripmining is done first, because it does not hinder the other transformations - it only limits the number of iterations of the inner loop to the vector length. The function that stripmines loops is named `Stripmine_LOOP`. Suppose we stripmine a loop with \( N \) iterations. The basic flow of this function is then as follows:

1. Create a new variable `strip_div_i`
2. Add the statement `strip_div_i = \( \frac{N}{V_L} \) * V_L`
3. Create a new outer loop, the index variable `outer_ind` walks from the lower bound until `strip_div_i` in steps of \( V_L \)
4. Create and initialize a DLI-structure for the outer loop, set that it is not vectorizable
5. Add `defs` and uses to the chain for `strip_div_i` and `outer_ind`
6. Create a copy of the original loop, make it walk from `strip_div_i` to `strip_div_i + V_L` in steps of 1
7. Create and initialize a DLI-structure for the copied loop, set that it is vectorizable
8. Add `defs` and uses for `outer_ind` in the inner loop
9. Add the copy to the body of the outer loop
10. Call `LNO_Build_Access` on both DLI-structures
11. Remap the lower bound of the original loop to `strip_div_i`
12. Add the use of `strip_div_i`
13. Insert the outer loop before the original loop
14. Update the Array Dependence Graph

### 4.3.3 Scalar Expansion

If the compiler finds any assignments to scalar variables inside the loop, it will attempt to scalar expand it. Scalar expansion can only be done if a number of conditions are met. The function `Scalar_Expandable()` takes a WN that represents an assignment to a scalar variable and checks whether it can be scalar expanded inside the loop it is in. A variable is `live` on loop exit if it is still used after the loop has terminated. If a variable is live on loop exit, the last value kept by the scalar expanded variable must be transferred back to the original scalar value (the
scalar expanded variable will cease to exist after loop exit). If the variable is scalar expandable and is not live on loop exit, the function Scalar_Expandable() returns SE.EASY; if it is scalar expandable and live on loop exit, it returns SE.HARD; if it is not scalar expandable it returns SE.NONE. In the experimental compiler we have added return values SE.UEU.EASY to indicate that the scalar variable has upward exposed uses and is not live on loop exit and SE.UEU.HARD to indicate that the scalar has upward exposed uses and is also live on loop exit.

Once Scalar_Expandable() has indicated that a scalar variable can be scalar expanded, the procedure to scalar expand it is as follows:

1. If the scalar variable has any upward exposed uses, peel off the first iteration of the loop.

2. Introduce a new array SE.i, where i is a counter that is different for each scalar variable to be expanded.

3. Walk through the def-use and use-def chains of the scalar variable to find all defs and uses of this particular scalar variable inside the loop.

4. For each def (write) of this scalar variable, replace it with a write to SE.i[loop.counter]

5. For each use, determine if it is an upward exposed use using the def-use chains.
   - If so, replace it the scalar variable use with SE.i[loop.counter – 1]. If the use is not upward exposed, replace the scalar variable use with SE.i[loop.counter]

6. Check whether the variable is live on loop exit. If so, store the last value assigned to the array variable back to the scalar variable.

4.3.4 Partial vectorization

The basic approach of partial vectorization is to construct an acyclic condensed graph from the statement dependence graph, and split up the loop in multiple loops using the ACG. Each strongly connected component containing more than one statement is moved to a new loop with identical bounds, and this loop is then marked non-vectorizable. Given the ACG=(V,E) of a loop L, a pseudo-algorithm for partial vectorization is as follows:

F = None
While there are nodes in V
  - Take a node p from V with no incoming edges
    - If p is a node representing more than one statement:
      - add two new loops with the same bounds as L but with an empty body below loop F, or below L if F is not set
      - move the statements represented by p from L to the new first body and mark this loop non-vectorizable
      - Set F to the second body
    - Else if p is a node representing only one statement:
      - If F is set, move p from L to F, otherwise do nothing
  - Remove p from V
End

4.3.5 Statement reordering

After all other optimizations, the statements can be reordered using the algorithm given in 2.5.3. The implementation of this algorithm is pretty straightforward. Statement reordering using this algorithm is allowed within any inner loop.
4.4 WHIRL2C code generation

The WHIRL2C module can translate Very High WHIRL and High WHIRL back to C-code. Since we want to output CVP-C-code that contains additional vector intrinsic functions (which represent actual vector instructions on the CVP architecture), this module is to be modified also.

The WHIRL2C translates WHIRL nodes based on their opcode: for each WN opcode there is a translation function available, which may call other translation functions on the children of that WN. Each translation function gets a buffer of tokens which contains the (already) translated output as an argument. Thus the buffer of tokens is passed top-down through the WN-tree, along with the translation. Each translation function may add tokens (such as ‘a = b + c;’) to the buffer. The interface of the token buffer can be found in `osprey1.0/be/WHIRL2C/token_buffer.h`.

Additionally, each translation function gets a CONTEXT argument. The context is a bit-vector variable containing information about the current status of the translation. The context can thus be checked against certain flags, defined in `osprey1.0/be/WHIRL2C/whirl2c_common.h`. We defined an additional flag in the context named `CONTEXT_VECTOR_LOOP` and some macros to set/unset this flag and to check whether the flag is set.

4.4.1 Vector loops

We want to emit vector intrinsics only when we translate a loop that is marked for vectorization. The translation function for loops is named `WN2C_do_loop`. In this translation, we check whether the passed WN (which is a loop) is marked vectorizable. If so, instead of adding the `for (...)` header we call a new translation function named `WN2C_vect_do_loop`. In `WN2C_vect_do_loop` we then set the `CONTEXT_VECTOR_LOOP` flag, thus indicating that we are currently processing a loop that can be vectorized. Then, we call the translation functions for each of the children of the loop WN, which for an inner loop are the statements that are contained within the loop.

4.4.2 Operators

In several operator translation functions we added a check whether the flag `CONTEXT_VECTOR_LOOP` was set. If so, the translation should be different in those cases because we want to output vector-code intrinsics. For example, the function translating the binary operator plus will output ‘arg1 + arg2’ when `CONTEXT_VECTOR_LOOP` is not set, but if it is set it will output the corresponding vector intrinsic ‘vadd(arg1, arg2)’. The most common arithmetic operator translation functions have been adapted this way. Additionally, the intrinsics that represent binary operators may have only two arguments, whereas in C it is valid to write $d = a + b + c$. In vector-code we introduce a temporary variable, like

```c
    temp1 = vadd(b+c);
    d = vadd(a, temp1);
```

A special case is the multiply-and-accumulate (MAC) instruction. It performs the equivalent of the statement $a = a + b \cdot c$ in a single instruction. This instruction can be executed on CVP when $a$ and $b$ are vectors and $c$ is a scalar, or when $a$, $b$, and $c$ are all vectors. This is implemented by searching for a WHIRL tree which matches the $a = a + b \cdot c$ statement. If such a statement exists, a MAC intrinsic (vmac) is emitted instead of the corresponding multiply, add and store intrinsics.
4.4.3 Loads and stores of variables

Loads and stores of variables present an altogether different challenge. Inside a strip loop that is marked as vectorizable each array variable should become a vector variable in the output. An array variable with the size of multiple vectors can then be seen as multiple vectors that are consecutive in memory. To refer to the correct location of a vector variable, we use pointers. First of all, we assume that the first element of each vector variable (representing an array) is aligned. Given the following code:

```c
for (int i = 0; i < 8; i++) {
    A[i] = A[i+8];
}
```

we assume that element \( A[i] \) is aligned. From this it follows that each positive offset from \( i \) that is a multiple of the vector length is aligned. If the vector length \( V_L \) were 8, the memory location \( A[i+8] \) would be considered to be aligned also. However if \( V_L \) were 16, \( A[i+8] \) would be an unaligned access.

Next, assuming that for each array-variable \( \text{var.x} \) that is indexed by the strip loop counter now represents a vector variable, there is a pointer \( \text{var.x.ptr} \) pointing to each \( \text{var.x} \). Each \( \text{var.x.ptr} \) is initialized to the first element of the array variable it points to. Therefore initially we can assume \( \text{var.x.ptr} \) points to \( \text{var.x}[0] \) and assuming the loop counter \( i \) starts at zero (this is taken care of by the compiler) points to \( \text{var.x}[i] \). Then, at the end of each iteration this pointer is increased by one. Since the \( \text{var.x} \) is now seen as a vector variable and \( \text{var.x.ptr} \) is a pointer to a vector variable, a pointer increase by one means the pointer now points at the next vector. In the next strip loop we can then again assume that \( \text{var.x.ptr} \) points to \( \text{var.x}[i] \). Note that the pointer increase at the end of a strip loop is not yet generated by the experimental compiler - we add them by hand.

Now given that \( \text{var.x.ptr} \) points to \( \text{var.x}[i] \), for each load of \( \text{var.x} \) we check whether the offset is aligned - that is a multiple of the vector length. If it is, we simply divide the offset by the vector length to get the increase of the pointer needed to load from the correct memory location. For example, a load of \( \text{var.x} \) with offset 32 \( (\text{var.x}[i + 32]) \) can be written as \( *(\text{var.x.ptr} + 2) \), assuming \( V_L = 16 \) and the \( * \) operator denotes pointer dereferencing.

For loads that are not a multiple of the vector length and therefore unaligned there is a special intrinsic function named \( \text{vload.unal()} \). It takes a pointer to an element (which is not aligned) and then reads an entire vector from that element on. However since we only have pointers that point to entire vectors, we need an additional intrinsic which allows us to point at an individual element inside a vector (from where the unaligned read will take place). For that purpose the intrinsic named \( \text{padd(ptr, offset)} \) exists. The first argument is a pointer to an aligned element of the vector variable, the second argument indicates the desired element within that vector. An unaligned vector read with offset 6 \( (\text{var.x}[i + 6]) \) can then be read by calling \( \text{vload.unal(padd(var.x.ptr, 6))} \);

Stores are generated likewise, except for the fact that the CVP does not support unaligned stores.
5 Vectorized algorithms

This chapter describes a number of algorithms and how they can be automatically vectorized using the techniques described in previous chapters. First, a non-vectorized implementation of each algorithm is given. Then we will show what techniques are applied in order to automatically vectorize it. Finally, we will evaluate the generated code by comparing it to hand-written code. Since the scheduler (which schedules each of the instructions in the CVP-C code and shows the number of cycles it takes to execute that code) was only available at the end of this project, we can only give a quantitative performance analysis in terms of cycles for the go-slay algorithm. For the other two algorithms, we will discuss any differences with a hand-written version and reason about them.

5.1 FIR filter

The Finite Impulse Response filter algorithm is perhaps the most known in the DSP-world. It is a very simple algorithm described by:

\[ y[n] = \sum_{k=0}^{k<T} c[k] * x[n + k] \]

Here the variable \( x \) represents an array of input values, the array \( y \) represents an array of output values and the variable \( c \) is an array of coefficients. The number of coefficients \( T \) is also called the number of taps.

5.1.1 Non-vectorized Implementation

A straightforward GCL implementation of this algorithm is shown below (we assume that all elements of array \( y \) have been initialized to 0 for simplicity):

Program 21

```gcl
var N, T : int;
var x, y : array[0..N + T] of int;
var c : array[0..T] of int;
var k, n : int;
n := 0;
do n < N
   k := 0;
do k < T
      y[n] := y[n] + c[k] * x[n + k];
      k := k + 1;
   od
   n := n + 1;
```

5.1.2 Vectorization opportunities

As a first attempt we may try to vectorize the inner loop. Doing this results in the following program:
Program 22

\[
\text{var } N, T : \text{int}; \\
\text{var } x, y : \text{array}[0..N + T] \text{ of int}; \\
\text{var } c : \text{array}[0..T] \text{ of int}; \\
\text{var } k, n : \text{int}; \\
n := 0; \\
d \text{ do } n < N \\
\quad y[n] := y[n] + c[0 : T - 1] * x[n : n + T - 1]; \\
\quad n := n + 1; \\
\text{od}
\]

Note that the inner loop can be stripmined (assuming \(T \geq V_L\)), but for clarity we have not applied this straightforward transformation here (assume for now \(T = V_L\)). Stripmining the inner loop has some disadvantages though. If \(T\) is very small \((T < V_L)\) it is not vectorizable at all. If \(T\) is no multiple of the vector length \(V_L\), we need to do an additional scalar loop for all elements larger than \(\frac{T}{V_L} \times V_L\). This can be a severe limitation for some of the programs that we want to execute on the CVP. For example, it is not very uncommon in video-processing to have 3-tap, 9-tap or 11-tap filters. Finally, we only calculate one value of output \(y\) at a time. The CVP architecture however could store an entire vector of output elements. It is known that a vectorized version of FIR is able to produce \(V_L\) outputs at a time \([19]\).

We can modify the algorithm to produce \(V_L\) outputs at a time if we stripmine the outer loop instead (we will assume \(N\) is a multiple of \(V_L\) for this example):

Program 23

\[
\text{var } N, T : \text{int}; \\
\text{var } x, y : \text{array}[0..N] \text{ of int}; \\
\text{var } k, n : \text{int}; \\
n := 0; \\
d \text{ do } n < N \\
\quad k := 0; \\
\quad \text{do } k < T \\
\quad \quad l := n; \\
\quad \quad \text{do } l < V_L \\
\quad \quad \quad y[l] := y[l] + c[k] * x[l + k]; \\
\quad \quad \quad l := l + 1; \\
\quad \quad \text{od} \\
\quad k := k + 1; \\
\text{od} \\
n := n + V_L; \\
\text{od}
\]

This program thus multiplies \(V_L\) input values with one coefficient repeatedly, thereby producing \(V_L\) output values each time. The advantages of stripmining the outer loop are that the vector parallelism obtained is not dependent on the number of filter taps, but on the number of input samples \(N\). The number of input samples \(N\) is large because we usually have a stream of data, for example in video-processing a stream of pixels. Secondly now we can produce \(V_L\) results simultaneously, thus exploiting the full store bandwidth of the vector processor. If we replace the strip loop by their corresponding vector instructions, the program is as follows:
Program 24

```c
var N, T : int;
var x, y : array[0..N] of int;
var k, n : int;
n := 0;
do n < N
    k := 0;
do k < T
        y[n : n + V_L - 1] := y[n : n + V_L - 1] + c[k] * x[n + k : n + V_L - 1 + k];
        k := k + 1;
    od
od
```

The vectorizing compiler contains a few checks to recognize algorithms for which outer loop stripmining is the most optimal solution. The most important condition for outer loop stripmining is that the array index of all stores is the outer-loop index. Thus we ensure that we produce multiple writes in a stripmined loop. We show the CVP-C vector code that was generated by the compiler with some additional comments below:

```c
v_t *x_ptr; // Assume x_ptr points to x[0]
v_t *y_ptr; // Assume y_ptr points to y[0]
for (n = 0; n < N; n+=V_L) {
    int k;
    for (k = 0; k < T; k++) {
        // This loop writes y[n:n+V_L-1] T times
        // In the first iteration x_ptr will load x[n:n+V_L-1] (k == 0)
        v_t x_unaln_load = *(x_ptr);
        *(y_ptr) = vmac16(*(y_ptr), c[k], x_unaln_load); // Write y[n:n+V_L-1]
        x_ptr = padd(x_ptr, 1); // In the next iteration x_ptr
        // will load x[n+k:n+k+V_L-1]
    }
    y_ptr++; // Increase pointer to start producing the next V_L outputs
}
```

The most important difference with program 25 is that a MAC intrinsic function (vmac16) has been used to translate the $y[n : n + V_L - 1] := y[n : n + V_L - 1] + c[k] * x[n + k : n + V_L - 1 + k]$ statement.
5.2 Golay correlator

The Golay correlator algorithm is described in [20]. It processes input values through several stages. Each stage uses one or more delayed values of the outputs of the previous stage. We analyze the scalar version of the algorithm to detect any vectorization opportunities and point out whether any difficulties for vectorization exist.

5.2.1 Non-vectorized implementation

Below is an implementation of the Golay correlator written in C.

```c
#include <stdio.h>

#define INP_SIZE 8000

typedef short GolayType; // Short to match complex data width

GolayType output[INP_SIZE]; // Output buffer
GolayType input[INP_SIZE + 6]; // Input buffer

GolayType golay_1, golay_2;
GolayType golay_1_delay[INP_SIZE + 6], golay_2u[INP_SIZE + 8], golay_3[INP_SIZE + 48];
GolayType golay_4u[INP_SIZE + 192], golay_4[INP_SIZE + 128], golay_5[INP_SIZE];

int main() {

    for (int i = 0; i < INP_SIZE; i++) {

        // Stage 1
        golay_1 = input[i] + input[i + 2] + input[i + 4] - input[i + 6];

        // Stage 2
        golay_2u[i+8] = golay_1_delay[i + 7] + golay_1;
        golay_2l = golay_1_delay[i + 7] - golay_1;
        golay_1_delay[i+8] = golay_1;

        // Stage 3
        golay_3[i+48] = golay_2u[i] + golay_2l;

        // Stage 4
        golay_4u[i+192] = golay_3[i] + golay_3[i+32] + golay_3[i+16] - golay_3[i+48];
        golay_4l[i+128] = golay_3[i] - golay_3[i+16] + golay_3[i+32] + golay_3[i+48];

        // Stage 5
        golay_5[i] = golay_4u[i] + golay_4u[i + 128] - golay_4l[i] + golay_4l[i+128];
    }
}
```
5.2.2 Vectorization opportunities

The statement dependence graph for the golay algorithm is given in figure 16. The label of each node refers to the assignment statement of the label. For example, the node labelled \textit{golay.2l} refers to the statement \textit{golay.2l = golay.1.delay[i + 7] - golay.1}.

The cycles in the statement dependence graph correspond to the \textit{golay.1} and \textit{golay.2l} variables, which are scalars. By applying scalar expansion on these variables, these cycles disappear. The statement dependence graph after scalar expansion is depicted in figure 17. Note that the compiler also had to do some statement reordering: in the scalar program, it was fine to assign a value to \textit{golay.1.delay[i + 8]} after reading \textit{golay.1.delay[i + 7]}. However in vectorized form reading from \textit{golay.1.delay[i + 7]} means reading an entire vector, including the element \textit{golay.1.delay[i + 8]} which is still to be assigned. Therefore, the assignment to \textit{golay.1.delay} must be done before any reads in the \textit{golay.2l} and \textit{golay.2u[i]} assignments. Since there are now no cycles in the statement dependence graph and for each statement a vector equivalent exists, we can stripmine the loop and mark it for vectorization. The resulting vector-code can be found in Appendix A. On the next page we show a GCL version of the vectorized program (some variable declarations were left out for readability):
Program 25

var INP_SIZE : int;
var i : int;
i := 0;
do i < INP_SIZE;

    {Stage1}
golay_1[i : i + V_L - 1] := input[i : i + V_L - 1] + input[i + 2 : i + 2 + V_L - 1]
+ input[i + 4 : i + 4 + V_L - 1] + input[i + 6 : i + 6 + V_L - 1]
golay_1_delay[i + 8 : i + 8 + V_L - 1] := golay_1[i : i + V_L - 1];

    {Stage2}
golay_2u[i + 8 : i + 8 + V_L - 1] := golay_1_delay[i + 7 : i + 7 + V_L - 1] + golay_1[i : i + V_L - 1];
golay_2l[i : i + V_L - 1] := golay_1_delay[i + 7 : i + 7 + V_L - 1] - golay_1[i : i + V_L - 1];

    {Stage3}
golay_3[i + 48 : i + 48 + V_L - 1] := golay_2u[i : i + V_L - 1] + golay_2l[i : i + V_L - 1];

    {Stage4}
golay_4u[i + 192 : i + 192 + V_L - 1] := golay_3[i : i + V_L - 1] + golay_3[i + 16 : i + 16 + V_L - 1]
+ golay_3[i + 32 : i + 32 + V_L - 1] - golay_3[i + 48 : i + 48 + V_L - 1]
golay_4l[i + 128 : i + 128 + V_L - 1] := golay_3[i : i + V_L - 1] - golay_3[i + 16 : i + 16 + V_L - 1]
+ golay_3[i + 32 : i + 32 + V_L - 1] + golay_3[i + 48 : i + 48 + V_L - 1];

    {Stage5}
golay_5[i : i + V_L - 1] := golay_4u[i : i + V_L - 1] + golay_4u[i + 128 : i + 128 + V_L - 1]
- golay_4l[i : i + V_L - 1] + golay_4l[i + 128 : i + 128 + V_L - 1];

i := i + V_L;

od
The corresponding vector-code in CVP-C generated by the compiler can be found in Appendix A.

The vector-code generated by the compiler has been given as input to the High Level Assembler (which was described in the introduction). The HLA 'scheduled' the code by packing the instructions in VLIW and pipelining them. The resulting assembly contains the number of cycles the CVP would take to execute the schedule. The loop body execution time of this vector-code is 48 cycles, whereas code that is vectorized by hand takes only 23 cycles - which is roughly a factor two faster.

The main cause of this difference is the pointer addressing generated by the compiler. For example, suppose we load a vector from memory with an offset of 32 elements. Assuming a vector contains 8 elements, the compiler will output:

```
var_load_32 = *(var_ptr + 4);
```

This addressing scheme is however not supported by the CVP. It only supports pointer addition to the same register (p = p + 1 is legal, p = q + 1 is not) and the scheduler will only accept direct dereferencing of pointers ( *(p) is legal, *(p + 4) is not). Therefore, for each access with an offset the following code has to be used:

```
tmp_ptr = var_ptr;
tmp_ptr = tmp_ptr + 4;
var_load_32 = *(tmp);
```

Since the golay algorithm contains quite a few loads with offset, for each of these loads an additional temporary pointer is used. Additionally, the scheduler does not yet know how long it should keep these temporary variables alive. Given the fact that the CVP has a limited number of pointer registers available, this results in spilling - storing a pointer value in memory and restoring it when it is needed again.

The hand-optimized version of the golay algorithm explicitly uses the same pointer for each unaligned load of a variable. Consider a part of the compiler-generated vector-code for stage 3 of the golay algorithm:

```
golay_3_load_16 = *(golay_3_ptr + 2);
golay_3_load_32 = *(golay_3_ptr + 4);
golay_3_load_48 = *(golay_3_ptr + 6);
```

This will result in 3 temporary pointers being used. The hand-optimized version however explicitly uses one pointer and increases it by the correct amount:

```
golay_3_ptr +=2;
golay_3_load_16 = *(golay_3_ptr);
golay_3_ptr +=2;
golay_3_load_32 = *(golay_3_ptr);
golay_3_ptr +=2;
golay_3_load_48 = *(golay_3_ptr);
```

These so-called pointer-walks are therefore much more efficient. It would be possible to implement these pointer-walks in the code generation phase of the compiler, but a more logical place to implement it would be in the scheduler itself. This is because pointer-walks introduce constraints on the order of execution of the loads and hence reduce scheduling freedom.

A second cause of the bad performance of the compiler-generated vector-code is the use of three unaligned loads on the input vector, which is shown in figure 18. The hand-optimized version needs only one aligned load and an additional buffer
to store the previous aligned load. To see why this is sufficient, notice that the vector load of input[i+2] loads elements input[i+2:i+9], the vector load of input[i+4] loads elements input[i+4:i+11] and the vector load of input[i+6] loads elements input[i+6:i+13]. All of these elements actually reside in two vectors, which are the values input[i:i+15].

The hand-optimized version always keeps the two subsequent vectors representing input[i:i+15] available. From these two vectors we can then compose vectors that consist of input[i+2:i+9] etc. etc.. For this purpose CVP has dedicated instructions (select and shuffle) that can select different elements from two vectors and 'shuffle' them into a single vector. This is shown in figure 19.

In order to implement this optimization in the compiler, the compiler would need to determine that there are a number of unaligned accesses falling within the boundary of two aligned vectors.
5.3 Vertical peaking

The vertical peaking algorithm is used in video pixel processing [?]. It consists of a number of operations on a set of pixels, where each iteration uses some of the computed values of the previous iteration.

5.3.1 Non-vectorized implementation

The implementation below is written in C. The type line is equal to an array of type short, the types aperture and par_vpk are both structs containing only members of type short. Therefore any access to a member of these structs can be seen as an access to a member of type short. The CLIP macro refers to operators using min and max functionality. Since the compiler does not support this, CLIP(a, b, c) has been re-defined as the addition a + b + c.

```c
#include "types.h"

line main_p4, main_p3, main_p2, main_p1;
line main_0, main_m1, main_m2, main_m3, main_m4, main_peaked;
aperture main_ap;
par_vpk main_vpp;

void Vert_peaking_ref2( line p4, line p3, line p2, line p1, line _0, line m1, line m2, line m3, line m4,
aperture ap, par_vpk vpp,
line peaked )
{
  looper px;
  short fourth_Fs, sixth_Fs, low_band;
  short sum_m1, sum_0, sum_p1;
  short gained, gained_abs, gained_q, gained_in;
  short scaled, sm_factor, cored;
  int hi_res;

  for (px = 0; px < 256; px++) {
    sum_m1 = sum_0; /* Upward exposed use of sum_0 */
    sum_0 = sum_p1; /* Upward exposed use of sum_p1 */

    fourth_Fs = (-p2[px] + 0[px] + 0[px] - m2[px]) * vpp.vpk.1.4th_Fs.gain[0];
    sixth_Fs = (-p3[px] + 0[px] + 0[px] - m3[px]) * vpp.vpk.1.6th_Fs.gain[0];
    hi_res = (vpp.vpk.low_band.coeff[4] * ((int)p4[px] + (int)m4[px]) +
              (vpp.vpk.low_band.coeff[3] * ((int)p3[px] + (int)m3[px]) +
               (vpp.vpk.low_band.coeff[2] * ((int)p2[px] + (int)m2[px]) +
                (vpp.vpk.low_band.coeff[1] * ((int)p1[px] + (int)m1[px]) +
                 (vpp.vpk.low_band.coeff[0] * ((int)_0[px])

    low_band = CLIP(-8192,((short)((hi_res + 16) >> 5)),8191);

    // fourth_fs, sixth_fs and low_band are scalar flow dependencies
    sum_p1 = fourth_Fs + sixth_Fs + low_band;

    // Here the flow dependencies from sum_0, sum_m1 and sum_p1 come together
    gained = CLIP(-2047,((sum_m1 + sum_0 + sum_0 + sum_p1 + 2) >> 2.2047);

    // From here there are some subsequent arithmetic operations executed on gained
    gained_q = gained >> 4;
    gained_abs = ABS( gained );
    gained_in = gained_abs << vpp.vpk.smart[0];
    scaled = gained_in >> 4;
    sm_factor = 12800/(100 + scaled);
    cored = MIN( (gained_q + vpp.vcore.thd[0]), 0 );
```
5.3.2 Vectorization opportunities

The SDG for the vertical peaking algorithm is shown in figure 20. Again the label of the vertices refers to the statement assigning the variable corresponding to the label. Though there are more scalar variables in the program listing, most of them were only storing intermediate results between arithmetic operations. Therefore, the compiler optimized these away and they do not show in the dependence graph.

The sum.m0 and sum.0 are actually delayed values of the sum.p1 variable. Because of this, both sum.0 and sum.p1 have upward exposed uses. The cycles in the dependence graph can be removed by applying the scalar expansion technique described in chapter 3. The SDG of the algorithm after scalar expansion is shown in figure 21.

There are no more cycles in the new dependence graph. The dependence graph also shows that the statements need to be reordered in order to produce valid vector-code: sum.p1 must be stored before sum.0 is stored, after which sum.m1 can be stored. The corresponding CVP-C vector-code generated by the compiler is shown in Appendix B.

When compared to a version that is vectorized by hand, a few key differences become clear. First of all the compiler-version does not contain the min and max instructions, because the experimental compiler does not support them yet.

Secondly, some of the values used in the computation of the hi.res variable are constant during each iteration. Each of the subterms (vpp.vpk.low.band.coeff[x]) is multiplied with vpp.vpk.low.band.gain[0]. Since the result of this multiplication is the same for every iteration, the programmer has moved these multiplications before the loop-start. Thus before the loop starts vpp.vpk.low.band.coeff[x] * vpp.vpk.low.band.gain[0] is calculated for each x ∈ {0, 1, 2, 3, 4}. The results are then used inside the loop, which saves 5 multiplications per loop iteration. It would be possible to implement this in the experimental compiler by determining which expressions are loop-invariant and calculate the results of those expressions before the loop starts.
Another difference is that the hand-written version uses MAC instructions to calculate the expressions found in the hi.res statement. Consider a similar statement $a = b*(c+d)+e*(f+g)$. This statement can be rewritten as $a = b*c+b*d+e*f+e*g$, which in turn can be written as:

\[
\begin{align*}
    a &= b \ast c; \\
    a &= a + b \ast d; \\
    a &= a + e \ast f; \\
    a &= a + e \ast g;
\end{align*}
\]

The last three statements can directly be translated to MAC instructions. The experimental compiler however is not yet able to recognize these kind of MAC patterns in a statement like $a = b \ast (c + d) + e \ast (f + g)$. It should however be no problem to implement this in the vectorizing compiler.
6 Conclusions

In this chapter we describe the conclusions of this graduation project. Finally, we will give some suggestions for future study.

6.1 Achievements

We have investigated a number of state-of-the-art vectorization techniques found in the literature. For each of these techniques we have described its purpose and shown its application on some abstract GCL programs.

Secondly, we have implemented these techniques in the Open64 compiler framework. We have documented the global architecture of the Open64 compiler and the implementation of each of these techniques. This resulted in an experimental vectorizing compiler, which addresses the problem statement: It takes a program written in the C language, attempts to vectorize it using the vectorization techniques and finally outputs vector-code in the form of CVP-C.

We have tested the experimental compiler using the FIR, golay and video peaking programs. Based upon the initial results of the experimental compiler, we have derived new vectorization techniques in order to improve the results of the experimental compiler: scalar expansion for variables with upward exposed uses and outer loop stripmining for the FIR algorithm. For the scalar expansion technique we have used program calculus to derive the transformation, thereby also proving its correctness.

Although the experimental compiler only contains a few vectorization techniques, these techniques appear to be sufficiently effective for vectorizing the FIR, golay and video peaking programs. For each of these programs we have indicated if and why their performance is not on track with code that is vectorized manually by a programmer. We have also indicated whether the performance-increasing optimizations in the hand-written program can be implemented in the vectorizing compiler also.

The current limitations of the experimental compiler are as follows:

- It does not support conditionals in the loop body;
- It does not support so called intra-operations. One example of an intra-operation is the intra-add instruction, which adds all the elements in a vector and stores the result in a scalar variable;
- It does not support min and max instructions and does not recognize MAC patterns more advanced than \( a + b \times c \) (used in hand-written peaking);
- It does not support shuffle and select instructions to avoid multiple unaligned writes (used in hand-written golay);
- It does not automatically update the pointers to the vectors after each iteration. It is usually trivial to add these manually, but implementing it would save the programmer some work;
- The loop may only contain references to array or scalar variables. Pointer analysis is not done by Open64, hence if the loop contains any pointers the dependence; information for the loop is incomplete and we cannot vectorize it;
- Many CVP-C programs use circular vector buffers and circular pointers. Often a vector is only needed a limited amount of time (or iterations). When a vector is no longer needed, the same vector buffer can be overwritten again by making it circular.
6.2 Suggestions for future study

In the achievements section we have indicated a few limitations of the experimental compiler. These limitations could be addressed by researching why a specific limitation is there, how it can be removed and how much performance will be gained if this limitation were removed.

For some of the optimizations used in hand-written code, such as the pointer-walks used in the golay algorithm, it is not completely clear whether they should be implemented in the vectorizing compiler or in the High Level Assembler. These optimizations should be evaluated more closely in order to determine where and how they can be applied best.

The 'shuffle' instruction available on CVP has been mentioned briefly in the discussion of the peaking algorithm. The shuffle instruction can shuffle the elements inside a vector given a certain pattern. Consider figure 22, which shuffles the elements of a vector of length four, thereby reordering the elements. In certain algorithms, such as the Fast Fourier Transform (FFT), use of the shuffle instruction can improve performance. It is worth investigating whether the experimental compiler can recognize, by analyzing the program and data flow between variables, when it would be beneficial for performance to use a shuffle instruction. The recognition of shuffle patterns (data permutations) in scalar programs will be the greatest challenge there.
Appendix

A  Golay vector code

/* STAGE 1 */
input_unalnload_2 = vload_unal(padd_16(input_ptr, 2));
input_unalnload_4 = vload_unal(padd_16(input_ptr, 4));
input_unalnload_6 = vload_unal(padd_16(input_ptr, 6));

vtemp_1 = vadd_16(input_unalnload_2, *input_ptr);
 vtemp_0 = vadd_16(input_unalnload_4, vtemp_1);
 // golay_1 = input[i] + input[i+2] + input[i+4] - input[i+6];
golay_1 = vsub_16(vtemp_0, input_unalnload_6);
 // golay_1_delay[i+8] = golay_1
*(golay_1_delay_ptr + 1) = golay_1;

/* STAGE 2 */
golay_1_delay_unalnload_7 = vload_unal(padd_16(golay_1_delay_ptr, 7));
// golay_21 = golay_1 - golay_1_delay[i+7];
golay_21 = vsub_16(golay_1, golay_1_delay_unalnload_7);
// golay_2u[i+8] = golay_1 + golay_1_delay[i+7];
*(golay_2u_ptr + 1) = vadd_16(golay_1, golay_1_delay_unalnload_7);

/* STAGE 3 */
// golay_3[i+48] = golay_2u[i] + golay_21;
*(golay_3_ptr + 6) = vadd_16(golay_21, *golay_2u_ptr);

/* STAGE 4 */
golay_3_load_16 = *(golay_3_ptr + 2);
golay_3_load_32 = *(golay_3_ptr + 4);
golay_3_load_48 = *(golay_3_ptr + 6);

vtemp_3 = vsub_16(*golay_3_ptr, golay_3_load_16);
 vtemp_2 = vadd_16(golay_3_load_32, vtemp_3);
 // golay_41[i+128] = golay_3[i] - golay_3[i+16] + golay_3[i+32] + golay_3[i+48];
*(golay_41_ptr + 16) = vadd_16(golay_3_load_48, vtemp_2);

vtemp_5 = vadd_16(*golay_3_ptr, golay_3_load_32);
 vtemp_4 = vadd_16(golay_3_load_16, vtemp_5);
 // golay_4u[i+192] = golay_3[i] + golay_3[i+32] + golay_3[i+16] - golay_3[i+48];
*(golay_4u_ptr + 24) = vsub_16(vtemp_4, golay_3_load_48);

/* STAGE 5 */
golay_4u_load_128 = *(golay_4u_ptr + 16);
golay_4l_load_128 = *(golay_4l_ptr + 16);

vtemp_6 = vadd_16(*golay_4u_ptr, golay_4u_load_128);
 vtemp_6 = vsub_16(vtemp_7, *golay_4l_ptr);
 // golay_5[i] = golay_4u[i] + golay_4u[i+128] + golay_4l[i + 128] - golay_4l[i];
*(golay_5_ptr) = vadd_16(golay_4l_load_128, vtemp_6);
// POINTER UPDATE PHASE
// Since the compiler cannot yet determine when the pointers
// can be safely updated, it is done at the end of the loop
input_ptr++;
golay_2u_ptr++;
golay_3_ptr++;
golay_4u_ptr++;
golay_4l_ptr++;
golay_5_ptr++;
B Peaking vector code

// Loop body for vertical peaking generated by vectorizing compiler
// (with comments and clean-ups)

// Calculation of fourth_Fs
vtemp_3 = vsub_16(main_0, main_p2);
vtemp_2 = vadd_16(main_0, vtemp_3);
vtemp_1 = vsub_16(vtemp_2, main_m2);

// Calculation of sixth_Fs
vtemp_6 = vsub_16(main_0, main_p3);
vtemp_5 = vadd_16(main_0, vtemp_6);
vtemp_4 = vsub_16(vtemp_5, main_m3);

// mi_0 == vpp.vpk_1_4th_Fs_gain[0]
// mi_1 == vpp.vpk_1_6th_Fs_gain[0]
// Both are stored outside the loop by the compiler
// because they are (simple) loop invariants
// The compiler does not yet recognize multiplications
// with loop invariants and hence they are not placed in temps
// From this follows:
// vtemp_0 = fourth_Fs + sixth_Fs
vtemp_0 = vadd_16(mi_0 * vtemp_1, mi_1 * vtemp_4);

// Now follows the calculation of hi_res
vtemp_11 = vadd_16(main_p1, main_m1);
vtemp_12 = vadd_16(main_p2, main_m2);
// vtemp_10 = vpp.vpk_low_band_coef[1] * (int)p1[px] + (int)m1[px])
// + vpp.vpk_low_band_coef[2] * (int)p2[px] + (int)m2[px])
vtemp_10 = vadd_16(mi_6 * vtemp_11, mi_7 * vtemp_12);

vtemp_14 = vadd_16(main_p4, main_m4);
vtemp_15 = vadd_16(main_p3, main_m3);
// + vpp.vpk_low_band_coef[3] * (int)p3[px] + (int)m3[px])
vtemp_13 = vadd_16(mi_8 * vtemp_14, mi_9 * vtemp_15);
vtemp_9 = vadd_16(vtemp_10, vtemp_13);
// vtemp_8 = vtemp_9 + vpp.vpk_low_band_coef[0] * (int)_0[px]
vtemp_8 = vadd_16(vtemp_9, main_0 * mi_5);

// vtemp_7 = hi_res + 16 >> 5 (mi_4 == vpp.vpk_low_band_gain[0])
vtemp_7 = vmsar_32(mi_4 * vtemp_8 + 16, 5);

// sum_p1[px + 1] = vtemp_0 + vtemp_7 = fourth_fs + sixth_Fs + low_band
*(padd_16( sum_p1_ptr , 1 )) = vadd_16(vtemp_0, vtemp_7);

// sum_0[px + 1] = sum_p1[px]
*(padd_16( sum_0_ptr , 1 )) = *( sum_p1_ptr );
// sum_m[px + 1] = sum_0[px]
*(padd_16( sum_m_ptr , 1 )) = *( sum_0_ptr );

// Load sum_p1, sum_0 and sum_m unaligned
sum_0_unalnload_1 = vload_unal(padd_16(sum_0_ptr, 1))
sum_m1_unalnload_1 = vload_unal(padd_16(sum_m1_ptr, 1))
sum_pi_unalnload_1 = vload_unal(padd_16(sum_pi_ptr, 1))

vttemp_22 = vadd_16((sum_m1_unalnload_1), (sum_0_unalnload_1));
vttemp_21 = vadd_16((sum_0_unalnload_1), vtemp_22);

vttemp_20 = vadd_16((sum_pi_unalnload_1), vtemp_21);
// vtemp_19 = gained[px]
vttemp_19 = vmasr_32(vtemp_20 + 2, 2);

vttemp_18 = vmasr_32(vtemp_19, 4);
// mi_3 == vpp.vpk_smart[0], so vtemp_24 = gained_in = gained_abs << vpp.vpk_smart[0]
vttemp_24 = vmasl_32(vtemp_19, mi_3);
// vtemp_23 = scaled = gained_in >> 4
vttemp_23 = vmasr_32(vtemp_24, 4);
// temp_17 = cored * sum_factor, MIN defined as + here
vttemp_17 = vmpy_16(mi_2 + vtemp_18, 12800 / (vtemp_23 + 100));
// vtemp_16 = (cored+sum_factor + 64) >> 7
vttemp_16 = vmasr_32((vtemp_17 + 64), 7);
// peaked = vtemp_16
*(main_peaked_ptr) = vtemp_16;

// pointer updates for each of the variables are done here
sum_0_ptr++;
sum_m1_ptr++;
sym_pi_ptr++;
main_peaked_ptr++;
References


[10] Joseph Gebis, Sam Williams, David Patterson, Christos Kozyrakis, *VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM*


