Zubair Nawaz

Recursive Variable Expansion

A Transformation for Reconfigurable Computing

Recursive Variable Expansion

A Transformation for Reconfigurable Computing

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 11 januari 2011 om 12:30 uur

door

Zubair NAWAZ

Master of Science in Computer Science
Lahore University of Management Sciences
geboren te Lahore, Pakistan

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. ir. H.J. Sips

Copromotor: Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft, promotor
Dr. K.L.M. Bertels	Technische Universiteit Delft, copromotor
Prof. dr. O. Nieto-Taladriz García	Technical University of Madrid
Prof. Dr.-Ing. J. Becker	University of Karlsruhe
Prof. Dr.-Ing. M. Berekovic	Technishe Universität Braunschweig
Prof. dr. ir. M. J. T. Reinders	Technische Universiteit Delft
Dr. P. Diniz	University of Southern California
Prof. dr. K.G. Langendoen	Technische Universiteit Delft, reservelid

Zubair Nawaz

Recursive Variable Expansion - A Transformation for Reconfigurable Computing
Delft: TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica - III
PhD Thesis Technische Universiteit Delft.

Met samenvatting in het Nederlands.

ISBN:978-90-72298-11-9

Subject headings: compiler optimization, high performance computing, reconfigurable computing, automatic pipeline design algorithm, Smith-Waterman acceleration, dynamic programming acceleration.

Printed in The Netherlands

This dissertation is dedicated to my father

Raja Muhammad Nawaz

My inspiration and the one who gave me every opportunity to what I am.

Recursive Variable Expansion

A Transformation for Reconfigurable Computing

Zubair Nawaz

Abstract

RECONFIGURABLE computing, in which general purpose processor (GPP) is augmented with one or more FPGAs, is increasingly used for high-performance computing where massive fine-grain parallelism and pipelining can be exploited. A challenge is to exploit such massive parallelism on FPGAs and more specifically how to map an application on the heterogeneous underlying platform.

Similar to software compilers, hardware compilers can use loops to exploit such parallelism. The existence of a dependence between data is one the constraints that limits parallelism in a program. In this dissertation, we propose a transformation called Recursive Variable Expansion (RVE), which can be applied to an important category of loops. It removes all the data dependences by expanding the variable with its dependence expression until the expression becomes only a function of known variables. We classify two types of expressions, one which expands polynomially, and other which expands exponentially on the number of input variables. Irrespective of the type of expression, when we map an expression on an FPGA, the area (LUT) required on an FPGA is proportional to the number of terms in the expression.

We present an automated pipeline design algorithm for the problems whose expression expands polynomially. This algorithm determines the largest pipeline size that fits the FPGA. Furthermore, the algorithm also ensures that the time to feed the data is less than the time to process an instruction through the pipeline. We apply this algorithm to DCT, a widely used signal processing kernel, which shows a comparable performance to the hand optimized implementation.

The exponentially expanding version is applicable to the category of dynamic programming (DP) problems for which RVE is combined with dataflow. We demonstrated better performance than dataflow only, which is the best technique known so far for such problems. We generalize the approach by proposing a framework such that the technique can be applied to a large range of DP problems.

Finally, we validate the proposed DP framework using the Smith-Waterman (SW) algorithm, which is a widely used, computation and data intensive application in bioinformatics. We show that our implementation yields a 2.29x speedup at the cost of 2.82x more area as compared to the conventional dataflow systolic array implementation. Moreover, we propose a parallel FPGA design for SW traceback stage, whose bandwidth requirement is also well within limits of the current off-the-shelf FPGA boards.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms and Symbols

ASIC	Application Specific Integrated Circuit
DWARV	DelftWorkbench Automated Reconfigurable VHDL Generator
DCT	Discrete Cosine Transform
DP	Dynamic Programming
FIR	Finite Impulse Response filter
FPGA	Field Programmable Gate Arrays
GPP	General Purpose Processor
HDL	Hardware Description Language
HLL	High Level Language
HPC	High Performance Computing
ISDG	Iteration Space Dependence Graph
LCS	Longest Common Subsequence problem
MCSS	Maximum Contiguous Subsequence Sum problem
MM	Matrix Multiply
NW	Needleman-Wunsch algorithm
PE	Processing Element
RVE	Recursive Variable Expansion
RVENP	Recursive Variable Expansion with No Precomputation
RVEP	Recursive Variable Expansion with Precomputation
SW	Smith-Waterman algorithm
VHDL	Very High Scale Integrated Circuits Hardware Description Language

1 Introduction

THERE are many computer applications from various fields whose computational demands exceed conventional processor's capability. A few examples include applications in the domain of financial analytics, bioinformatics, data mining, medical imaging and scientific computations. Even though all these applications have different program requirements, performance is a common objective.

Over last few years, we have seen a shift towards heterogeneous systems for high performance computing (HPC). In heterogeneous systems, a general purpose processor is augmented with application specific hardware or processors. This heterogeneous system of processors can be on multiple boards or one board connected with high bandwidth interconnections or can be on a single chip. The application specific hardware gives better performance/area and performance/power for specific applications as compared to a homogeneous system of processors, therefore overall, heterogeneous systems reduce the area and power requirements. The application specific hardware can be an FPGA, GPU, Cell, ASIC or some other application specific processor. Convey HC-1ex and Tianhe 1A are some of the latest examples of such systems, which use FPGAs and GPUs respectively as co-processors.

Recently, FPGAs have also taken the interest of the HPC community. In this dissertation, we are considering reconfigurable computing based on field programmable gate arrays (FPGAs) as co-processors. FPGAs consist of logic and memory blocks, interconnected through a programmable network. The logic block is a programmable device, which holds the configuration of the FPGA. A programmer can make the FPGA application specific by changing its configuration. An FPGA can be reconfigured an unlimited number of times. The computations are implemented spatially, computing in parallel millions of operations across the silicon chip. Moreover, logic and memory blocks can

be arranged into a deep pipeline to exploit pipeline parallelism. Even though FPGAs operate at a lower frequency than the GPPs, they usually outperform GPPs in many applications due to the higher degree of parallelism they can exploit.

However, it is hard for many software-oriented programmers in HPC community to program FPGAs. They are usually proficient with high level languages (HLL), whereas FPGAs require deep hardware knowledge and the ability to program using a hardware description language (HDL). Moreover, they are also required to extract parallelism for a specific kernel, which is usually written for a single processor.There exist some advanced compiler tools such as the hArtes toolchain [2] that provide substantial (semi) automatic support for the entire mapping process. One of the components of this toolchain is the DWARV hardware compiler that generates synthesizable VHDL from C-code.

In this dissertation, we propose a transformation called Recursive Variable Expansion (RVE) which can automatically expose parallelism from the code more than the conventional techniques. The resulting parallelized code can be mapped on to FPGA for increased performance. We have also applied our technique to a class of optimization problems which are solved by dynamic programming technique. RVE should be seen as part of the DWARV hardware compiler that can generate highly optimized VHDL for a specific set of loops.

In the next section, we present a motivational example that describes our technique. In Section 1.2, we present the type and conditions of the problems on which our solutions can be applied. In Section 1.3, we present the contributions made in the dissertation. Finally in Section 1.4, we describe the organization of each chapter of the dissertation.

1.1 Motivational Example

Figure 1.1a depicts a simple real world example of a 4-tap FIR filter. We would like to extract maximum parallelism to get the result in minimum time. Loop unrolling is a widely used transformation in reconfigurable computing to extract the parallelism [27]. When the inner loop is fully unrolled and all the statements are scheduled with respect to dependences, we get the modified code as shown in Figure 1.1b and the scheduled data flow graph (DFG) as shown in Figure 1.1e. The adders are chained, due to data dependences of the statements.

When applying RVE, we can transform the code to remove the dependences

```
for i = 0 to n
    d[i]= 0
    for j = 0 to 3
        d[i]=d[i]+s[i+j]xc[j]
    end for
end for
```

(a) 4-tap FIR filter source code

```
for i = 0 to n
S1   d[i]= 0
S2   d[i]=d[i]+s[i+0]xc[0]
S3   d[i]=d[i]+s[i+1]xc[1]
S4   d[i]=d[i]+s[i+2]xc[2]
S5   d[i]=d[i]+s[i+3]xc[3]
end for
```

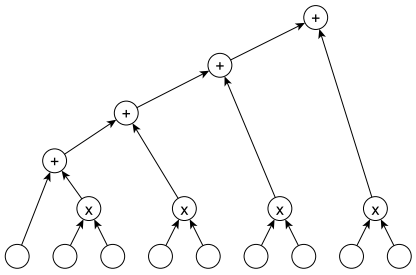
(b) FIR filter code after inner loop fully unrolled

```
S1  d[i]= 0
S2  d[i]=s[i+0]xc[0]
S3  d[i]=s[i+0]xc[0]+s[i+1]xc[1]
S4  d[i]=s[i+0]xc[0]+s[i+1]xc[1]+s[i+2]xc[2]
S5  d[i]=s[i+0]xc[0]+s[i+1]xc[1]+s[i+2]xc[2]+s[i+3]xc[3]
```

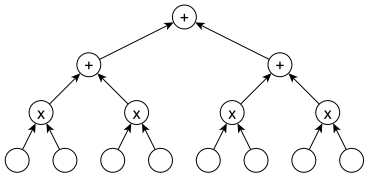
(c) RVE is applied to unrolled code

```
for i = 0 to n
S5  d[i]=s[i+0]xc[0]+s[i+1]xc[1]+s[i+2]xc[2]+s[i+3]xc[3]
end for
```

(d) FIR filter code after RVE



(e) DFG, when fully unrolled



(f) DFG, when RVE is applied

Figure 1.1: Motivational example

resulting in a shorter tree depth of the dataflow graph. In Figure 1.1c, we substitute the values of the dependent variable in the unrolled code as given in Figure 1.1b. Statement $S5$ in Figure 1.1c computes $d[i]$ and is free of dependences. We call this transformation as *Recursive Variable Expansion*. The transformed FIR code after applying RVE is shown in Figure 1.1d. The data flow graph for the resulting code after applying RVE is shown in Figure 1.1f, which shows that FIR can be computed more efficiently by applying RVE instead of using unrolling only.

1.2 Applicability Conditions

When RVE is applied to an arithmetic expression, the number of terms can grow. The resulting expression can even grow to exponential number of terms. The growth of expression changes the way RVE is applied. Therefore, we distinguish between two types of the expressions, based on the exponential and polynomial growth of the terms when RVE is applied to them.

In order to be able to apply RVE to the polynomially expanding problems, a number of conditions needs to be satisfied, which are as follows:

1. The bounds of the loops must be known at the compile time.
2. The indexing of the variables should be a function of surrounding loop iterators and/or constants.
3. If the kernel produces more than one output variables, then the length of the generic expression for those output variables should be equal.

However, when we deal with dynamic programming problems that expand exponentially when RVE is applied, the following constraint needs to be respected.

1. In each recursive expression, there is a constant number of recurrence terms and the distance vector between the current and all the dependent recurrence terms remains the same.

1.3 Contributions

This dissertation makes the following contributions.

- It describes a transformation called RVE for a certain class of problems that removes all the data dependences. Hence, increased parallelism can be achieved compared to existing loop transformations. We obtained speedups up to 77x as compared to GPP, when area is not the constraint. (Chapter 3)

- For polynomially expanding problems, we propose an automatic pipeline design algorithm which chooses an optimal pipeline size keeping in view the resources like area and memory bandwidth available on the FPGA. We apply the algorithm on a kernel from real world applications showing a comparable performance to the hand optimized implementation at the cost of more area. (Chapter 4)

- For exponentially expanding problems, we propose a generic framework for dynamic programming problems as well as two variants of the RVE algorithm, named RVE with no pre-computation (RVENP) and RVE with pre-computation (RVEP). When applied to various dynamic programming problems, we demonstrate that they outperform any known technique. We obtained speedups of up to 3.01x at the cost of 1.68x area overhead. (Chapter 5)

- The RVEP and RVENP solutions presented in Chapter 5 are limited to dynamic programming problems that do not clip values. We have also extended our RVE solution to the DP problems which clip the values. (Chapter 6)

- We present an extensive case-study by applying RVEP extended for clipping to the Smith-Waterman (SW) algorithm, which is one of the most widely used algorithm in bioinformatics. We show that our implementation gives 2.29x speedup at the cost of 2.82x more area as compared to the dataflow implementation, which is the highest speedup reported in the literature[1]. (Chapter 6)

- We propose a parallel FPGA design for RVEP Smith-Waterman traceback implementation, that can compute the alignment after performing the matrix fill for the whole database once. Moreover, it addresses the memory bandwidth issue that arises after changing the traceback from serial to parallel. We show that our technique can reduce the memory bandwidth requirement from 49.36 Gb/sec to 8 Kb/sec even for existing approaches. (Chapter 7)

[1] to the best of our knowledge

1.4 Dissertation Organization

The dissertation is organized as follows:

One of the advantages of RVE is that it eliminates data dependences that can occur inside loops. Chapter 2, therefore, discusses the different existing loop transformations and how they address some of these dependences and what issues are still unsolved.

In Chapter 3, we describe a transformation called Recursive Variable Expansion (RVE), which removes data dependences. Furthermore, we investigate the maximum parallelism that can be achieved, when area is not the constraint. For that reason, we have implemented four widely used kernels on FPGA and have shown its performance gain over GPP based execution.

Chapter 4 presents the extension for polynomially expanding expressions and allows an upper bound to be placed on the available area on which the RVE expression will be mapped.

Chapter 5 deals with the dynamic programming (DP) problems. We present a generic algorithm to apply RVE to DP problems, that can generate highly efficient circuits. We devise two variants of our technique namely RVENP and RVEP.

Smith-Waterman (SW) is a widely used bioinformatics algorithm. In chapter 6, we apply RVENP and RVEP to Smith-Waterman. It also addresses the clipping to zero problem and devise a general algorithm to avoid this.

In Chapter 7, we propose a parallel traceback design for SW on FPGA, which can give the alignment immediately after performing the matrix fill.

Finally, we conclude our dissertation in Chapter 8 by summarizing the chapters presented and emphasizing the contributions made in this research. We also discuss future research directions.

2

Data Dependences and Loop Transformations

LOOPS are an important source of performance improvement. The highest payoff is achieved by parallelizing the various iterations of the loop nest, which can be assigned to different processors. So, a major speedup can be obtained through *loop parallelization*. However, a loop in a program is not always parallelizable due to data dependences. Transformations are needed to make it work and this is done either manually or automatically.

The selection of suitable parallelization techniques among many available techniques depends upon the type of loop nests and dependency relationships encountered. Usually there are more than one loop level optimization technique that can be applied to the code. Finding the right combination and schedule of optimisation is a challenge as different choices will lead to different outcomes. There is also a need to verify that the transformation does not change the meaning of the program, which is called the *legal transformation*. As the proposed transformation Recursive Variable Expansion (RVE) eliminates data dependences, we dedicate this chapter to an in-depth discussion of data dependences and loop transformations. We first present the different dependences and then show how different transformations can change the order of execution to avoid dependences.

2.1 Dependence Relation

A dependence relation governs the sequence of data access to a memory. Given a program, if there is a constraint on the execution order of any two statements, then there is a dependence relationship between these statements. Any ordering based transformation that does not modify the dependence relationship

among statements is guaranteed to run the program correctly. There are two types of dependences at a higher level. First is due to a dependence arising from the control flow. It is called the control dependence. For example :

```
S1 if (x<7) then
S2   y=0
S3 end if
```

We cannot interchange $S_1$and S_2 in the program because the execution of S_2 depends upon S_1. So, there is a control dependence between S_1 and S_2, written as $S_1 \delta^c S_2$.

The other type is designed to ensure the correct order of data definition and data use. This is called *data dependence*. For example,

```
S1 a = 4
S2 b = 3
S3 c = 2 * a * b
```

Statement S_3 depends on both S_1 and S_2, as the variables used in S_3 are defined in S_1 and S_2. So S_3 must come after S_1 and S_2 in the transformed program, however, the order of S_1 and S_2 does not matter, so S_1, S_2, S_3 is the same as S_2, S_1, S_3 and they both produce the same result.

Data dependence is further divided into three types of dependences given below.

True dependence

A statement computes and stores a value in a variable and some later statement/statements use the variable. For example

```
S1 a = 10
S2 b = c + a
```

So S_1 must be executed before S_2, as the value written in S_1 is used in S_2. This type of dependence is also called flow dependence and is denoted by $S_1 \delta S_2$.

Antidependence

A statement reads a value stored in a variable and some later statement computes a new value for the same variable. For example:

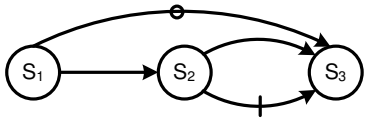


Figure 2.1: Dependence graph

```
S1 b = a + 10
S2 a = c + 5
```

Here again, S_1 and S_2 cannot be interchanged in the transformation, as after interchange b will get a wrong value. Antidependence in the above example is denoted by $S_1\delta^{-1}S_2$. Antidependence is not restricted like the true dependence. One can avoid this restriction by using two memory variables, like a_1 in S_1 and a_2 in S_2. Now, even if the statement S_2 is executed before S_1, the old value is still in a_2.

Output dependence

Both the statements write the same variable. For example

```
S1 a = 10
S2 .....
S3 a = b
```

S_1 and S_3 cannot be interchanged in the transformation as if there is another statement using the variable a after these definitions, then the variable a will get the wrong value in it. Output dependence in the above example is denoted by $S_1\delta^o S_2$.

2.1.1 Representing the Dependence Relations

Like other relations, the dependence relations between different statements can be represented by a graph [128]. It is not only a convenient way to understand the dependence relationship, but there are also many well known graph algorithms which can help to analyze the structure more closely. In a graph, each statement is represented by a node and the relationship between two statements is represented by a directed edge from a node called source to a node called sink. There can be more than one type of dependence between the same pair of vertices, therefore, the graph is essentially a multigraph.

```
S1 a = b + 10
S2 c = a * 5
S3 a = c/2
```

The following dependences in the above example are also represented in the dependence graph in Figure 2.1.

$S_1\delta S_2$, $S_1\delta^o S_3$, $S_2\delta S_3$, $S_2\delta^{-1} S_3$

2.1.2 Loop Dependence Analysis

When there is a loop, then each statement in a loop is executed many times. There can be not only a dependence relation between the different statements in the same iteration, but also a dependence relation between the same/different statement(s) in the different iterations. For example

```
for i = 1 to n
S1 a[i] = a[i-1] + c
S2 b[i] = a[i] * a[i-1]
end for
```

In the above example, when $i = k$, S_2 reads $a[k]$, which is computed in the last statement S_1 in the same iteration, this is called the *loop independent dependence* [60]. Whereas, in the same iteration $i = k$, S_1 reads the value of $a[k-1]$ computed by the same statement S_1 in iteration $i = k-1$. Similarly, in iteration $i = k$, S_2 reads $a[k-1]$ computed by the statement S_1 in iteration $i = k-1$. The latter two are called the *loop carried dependences* [60].

In the example above, i is called the *loop index*, here its lower bound is 1 and its upper bound is n. Generally we have loops like this:

```
for i = L to U step S
   ...
end for
```

In the example given above, the loop index i has lower bound L and the upper bound is U and the step is S. In this case, the loop index does not indicate which iteration is it executing. When $i = L$, it does not mean that the loop is in the L^{th} iteration, as it is its 1^{st} iteration. So, we have an iteration number which defines the iteration the loop is in. The compiler optimization techniques

```
for i1 = l1 to u1
 for i2 = l2 to u2
 ...
  for in = ln to un
   a[f1(i1, ... , in), ... , fm(i1, ... , in)] = ...
   ... = a[g1(i1, ... , in), ... , gm(i1, ... , in)]
  end for
 ...
 end for
end for
```

Figure 2.2: Perfect nest with n loops

usually require that the loop index should run from 1 to some upper bound in steps of 1. This can be done using *loop normalization* [60]. For this chapter, we implicitly consider unit loop step, otherwise it will be mentioned explicitly.

Figure 2.2 shows the generalized perfect loop nest of n loops, in which array a of m dimensions is read and written in different statements. f_i and g_i are the functions on the n dimensional loop index resulting in the i^{th} dimension of a. This generalized loop may read and write the same locations in different iterations giving rise to a different type of dependence.

A unique iteration in a nested loop is characterized by a vector $i = (i_1, i_2, \ldots, i_n)$ of n dimensions, called the *iteration/index vector* [11, 60]. The leftmost loop index represents the outermost loop. Each loop index is bounded by the corresponding lower and upper limits for that index.

The execution order is important for the dependence, as one needs to know which iteration is executed before some other iteration. Let i be a vector, i_k is the k^{th} element of i, and $i[1 : k]$ be the leftmost k elements of i. We say that iteration i is executed before the iteration j, denoted by $i \prec j$ and formally defined as [11].

$$i \prec j \text{ iff } \exists k : (i_k < j_k \wedge \forall m < k : i_m = j_m) \tag{2.1}$$

It means that an iteration vector i is executed before another iteration vector j if and only if any statement executed in the iteration described by the iteration vector i is executed before any statement executed in the iteration described by the iteration vector j.

```
for I = 1 to n
    for J = 1 to n
        a[I+1,J-1]=a[I,J]+a[I,J+1]
    end for
end for
```

(a)

```
for I = 1 to n
    for J = 1 to n+1
        a[J]=a[J]+a[J-1]+a[J+1]
    end for
end for
```

(b)

Figure 2.3: Loop carried dependences examples

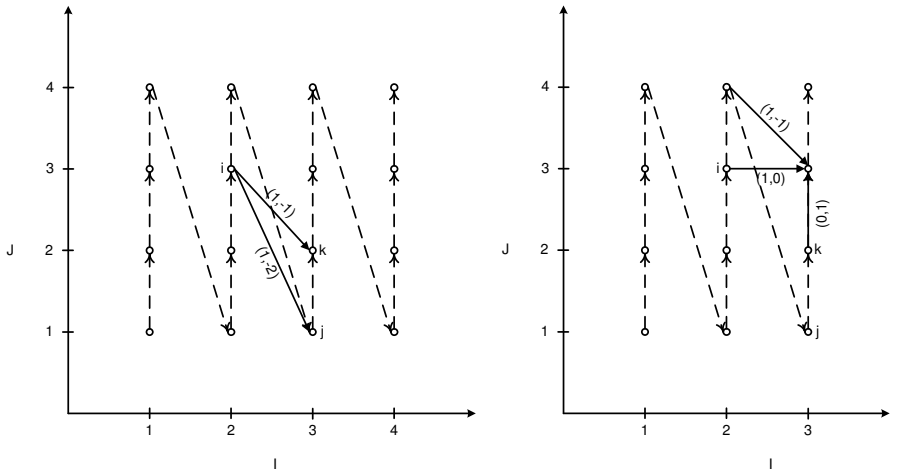
Formally, we say that there is a loop carried dependence from the iteration i to the iteration j, if both of them access the same memory location in which one of them is writing it, and $i \prec j \wedge f_k(i) = g_k(j)$, *where* $1 \leq k \leq n$ [11]. It means that there is a dependence between two different iterations when the values of subscripts are the same in these iterations. If there are no two such iterations, then the memory access is independent across all the iterations.

The loop carried dependence examples are given in Figure 2.3. Consider the iteration $i = [2,3]$ in Figure 2.3a when $a[3,2]$ is written, the same value is read in later iterations $j = [3,1]$ and $k = [3,2]$. It means that there is a true dependence from iteration i to iteration j and k. We represent the dependence between these iterations as $i \rightarrow j$ and $i \rightarrow k$.

2.1.3 Iteration Space

The iteration space is a good representation for understanding the way loops are executed and which loop carried dependences exist among the different iterations [128]. An iteration space of n-nested do loops is represented by an n-dimensional discrete Cartesian space. The iteration space of the loop nest is the set of all possible values of $i = (i_1, i_2, \ldots, i_n)$, i.e. the set of all points in n dimensions is constrained by their respective limits, described formally as $\{i \in Z^n : l_x \leq i_x \leq u_x, x = 1, 2, \ldots, n\}$. Each do-loop is represented by a unique axis in the iteration space. For example, for the code shown in Figure 2.3a, we have two loops given by loop indices I and J, so we have a 2-dimensional discrete Cartesian space shown in Figure 2.4a.

The space is composed of discrete points which represent the possible iteration vectors. The dotted arrow lines show the order in which the iterations will be executed. The dependence between different iterations is shown by the solid arrow lines. This type of graph is called the *Iteration Space Dependence Graph*



(a) Iteration space dependence graph for example in Figure 2.3a with $n = 4$

(b) Iteration space dependence graph for example in Figure 2.3b with $n = 3$

Figure 2.4: Iteration Space dependence graphs

(ISDG) [128]. In an ISDG, a dependence arrow is drawn between the source and the sink iteration vectors. Figure 2.4a shows the ISDG for the example shown in Figure 2.3a.

An ISDG and the iteration space are helpful to understand the dependence relationship in the loops and especially when the loop transformations are applied.

2.1.4 Distance Vectors

It is sometime beneficial to have an idea about the distance between the source and the sink of a dependence in the iteration space. If we have $i \rightarrow j$, then the distance vector $d(i,j)$ of length n is defined such that $d(i,j)_k = (j_k - i_k)$, where $1 \leq k \leq n$ [60, 68, 86]. In the example given in Figure 2.3a, we have a distance vector for the same statement due to $i \rightarrow j$ is $(1,-2)$ and due to $i \rightarrow k$ is $(1,-1)$, so, we have the distance vectors $\{(1,-2),(1,-1)\}$.

Given a distance vector $d(i,j)$, showing the distance between the source i and the sink j, the first non-zero element must be positive. The first non-zero negative element implies that the sink occurs before the source, which is not possible.

Another example given in Figure 2.3b has the distance vectors $\{(0,1),(1,0),(1,-1)\}$. The iteration space dependence graph is given

in Figure 2.4b.

2.1.5 Direction Vectors

Sometimes it is not possible to find the dependence distance between the iterations at compile time. However, there is enough information to get an idea about the dependences between the iterations by using the direction vectors [128, 129].

Given $i \rightarrow j$, we define the direction vector $D(i,j)$ as a vector of length n such that

$$D(i_k, j_k) = \begin{cases} < & \text{if } i_k < j_k \\ = & \text{if } i_k = j_k \\ > & \text{if } i_k > j_k \end{cases} \quad \text{where } 1 \leq k \leq n \tag{2.2}$$

The direction vectors for example in Figure 2.3a are $\{(<,>),(<,>)\}$ and for example in Figure 2.3b are $\{(=,<),(<,=),(<,>)\}$. Given $i \rightarrow j$, another way to define the direction vector $D(i,j)$ as a vector of length n is as follows [60]:

$$D(i_k, j_k) = \begin{cases} < & \text{if } d(i_k, j_k) > 0 \\ = & \text{if } d(i_k, j_k) = 0 \\ > & \text{if } d(i_k, j_k) < 0 \end{cases} \quad \text{where } 1 \leq k \leq n \tag{2.3}$$

For a loop carried dependence, there is an important characteristic called the *level* of a loop carried dependence, defined as the index of the leftmost non-"=" of $D(i,j)$. It means that it is the index of the outermost loop index that varies between the source and the sink. The outermost loop is counted as 1.

```
for I = 1 to n
   for J = 2 to n-1
      a[I,J] = a[I,J-1]
   end for
end for
```

The level of the dependence for the above example is 2, as $D(i,j)$ is $(=,<)$.

```
for I = 1 to n
    for J = 1 to n
        C[I,J]=A[I,J]+B[I,J]
    end for
end for
```

(a) Both loops are parallelizable

```
for I = 1 to n
    for J = 1 to n
        A[I,J]=A[I,J-1]+A[I,J-2]
    end for
end for
```

(b) Outer loop is parallelizable

```
for I = 1 to n
    for J = 1 to n
        A[I,J]=A[I-1,J]+A[I-1,J+1]
    end for
end for
```

(c) Inner loop is parallelizable

Figure 2.5: Parallelizable loops

2.2 Loop Transformations

The maximum gain in performance is achieved from the part of the program which takes the maximum time of the program - iterative loops. The loops are an important source of the performance improvement. The loop transformations change the order of execution of statements in the loops, so that more performance can be achieved.

The benefits and the legality are the two things that need to be discussed for every loop transformation. The benefits of the loop transformation cannot be evaluated without the knowledge of the underlying hardware architecture. A loop is transformed only if it is legal. A loop transformation is called *legal* if the transformation preserves the dependence relationship. Another way to say the same thing is that the transformation is legal if the transformed dependence vector is lexicographically positive as described in Section 2.1.4.

There are many loop transformations from parallelizing software compilers, which are commonly used in the compilers for the reconfigurable systems. In this section, we will briefly discuss the loop transformations which are the most significant to enhance the performance of the reconfigurable systems.

2.2.1 Parallelizable Loops

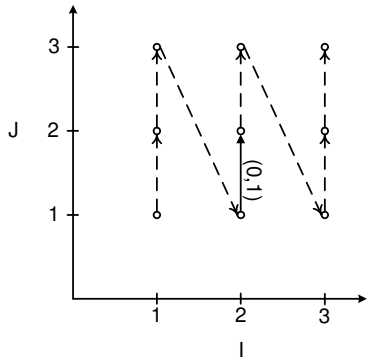
In case of the vector architecture machine, we always look for loops which are parallelizable. For example, the loop given in Figure 2.5a can be easily recognized as suitable for the vector machine, since it does not have any loop

```
for I = 1 to n
    for J = 2 to n
        A[I,J]=A[I,J-1]+C
    end for
end for
```

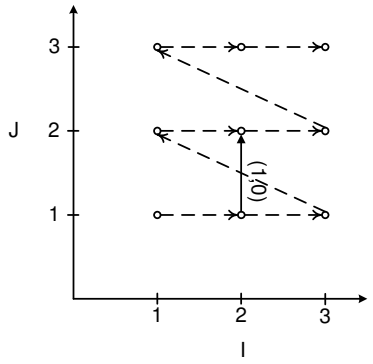
(a) Before loop interchange

```
for J = 1 to n
    for I = 2 to n
        A[I,J]=A[I,J-1]+C
    end for
end for
```

(b) After loop interchange



(c) ISDG for Figure 2.6a



(d) ISDG for Figure 2.6b

Figure 2.6: Code and ISDG for Loop Interchange

carried dependences and both the loops are parallelizable. Therefore to check whether a loop is parallelizable, we check whether that loop induces loop carried dependences or not. Formally, we define the k-th loop is parallelizable in distance vector $V = (v_1, \ldots, v_k, \ldots, v_n)$ [11], when

$$\forall V, v_k = 0 \vee \exists l < k : v_l > 0 \tag{2.4}$$

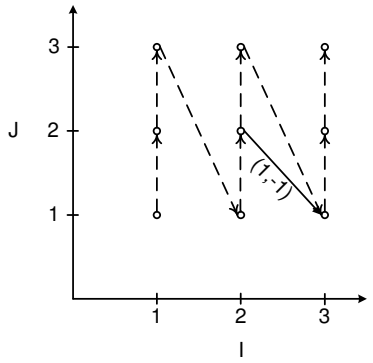
The example in Figure 2.5a has the distance vectors $\{(0,0),(0,0)\}$, which also means that both loops are parallelizable. Now we look at the two other examples which have loop carried dependences, but which can still be parallelized. The example in Figure 2.5b is outer loop parallelizable. Its distance vectors are $\{(0,1),(0,2)\}$, its outer loop has zero in both the vectors. The example in Figure 2.5c has a distance vector $\{(1,0),(1,-1)\}$. The inner loop is parallelizable, since its distance is zero in the first vector. Although in the second vector the distance is -1, but the distance of the outer loop is greater than zero.

```
for I = 1 to n
    for J = 1 to n
        A[I+1,J]=A[I,J+1]+C
    end for
end for
```

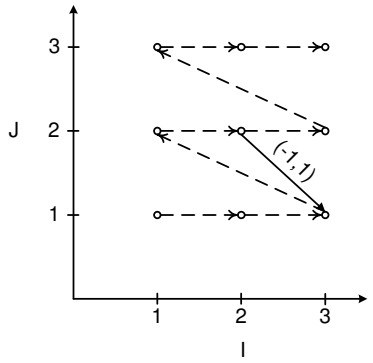
(a) Before loop interchange

```
for J = 1 to n
    for I = 1 to n
        A[I+1,J]=A[I,J+1]+C
    end for
end for
```

(b) After loop interchange



(c) ISDG for Figure 2.7a



(d) ISDG for Figure 2.7b

Figure 2.7: Illegal loop interchange

2.2.2 Loop Interchange

One of the most useful transformations to enhance the performance is the loop interchange, in which two loops are interchanged in a perfect loop nest [8, 125, 128]. Generally a compiler interchanges a sequential inner loop with an outer loop which carries no loop carried dependences. It has many benefits, such as enabling vectorization, improving vectorization and improving data locality. Figure 2.6a shows an example in which the inner loop cannot be parallelized/vectorized due to the loop carried dependency. The distance vector for the example in Figure 2.6a is $\{(0, 1)\}$. After loop interchange in Figure 2.6b, the distance vector becomes $\{(1, 0)\}$, in which the inner loop does not have any loop carried dependency and it can be parallelized. The ISDG after loop interchange in Figure 2.6d shows that dependence vector is lexicographically positive and hence legal.

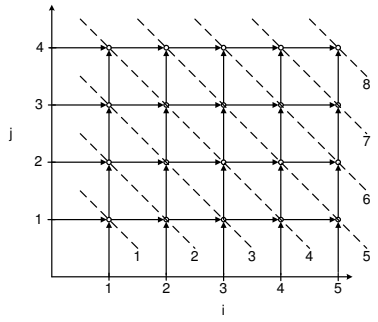
It is not always possible to legally apply loop interchange, as after loop interchange distance vector can violate the legality definition. For example, the code in Figure 2.7a is legal, but when the loop interchange is applied (as shown in Figure 2.7b), it is no more legal and it is easily seen in the ISDG for the code

```
for i = 1 to n
    for j = 1 to m
        A[i,j]=A[i-1,j]+A[I,j-1]
    end for
end for
```

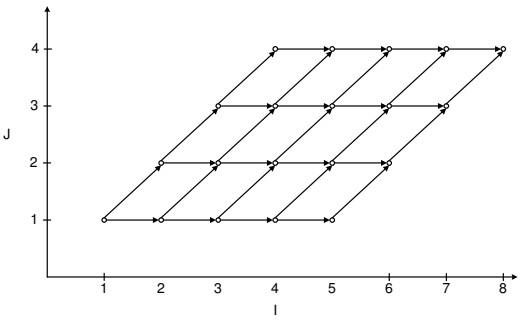
(a) Before loop skewing

```
for I = 1 to n+m-1
    for J = max(1,I-n+1) to min(m,I)
        A[I-J+1,J]=A[I-J,J]+A[I-J+1,J-1]
    end for
end for
```

(b) After loop skewing



(c) ISDG for code in Figure 2.8a



(d) ISDG for code in Figure 2.8b

Figure 2.8: Loop Skewing

after the loop interchange in Figure 2.6d, where the distance vector becomes $\{(-1,1)\}$, and is not lexicographically positive. Formally, the two loops i and j in a perfect loop nest of m loops are legal to interchange, when each distance vector $d = (d_1, \cdots, d_i, \cdots, d_j, \cdots, d_m)$ in the original loop nest is lexicographically positive after conversion to $d' = (d_1, \cdots, d_j, \cdots, d_i, \cdots, d_m)$ [8, 11, 127].

2.2.3 Loop Skewing

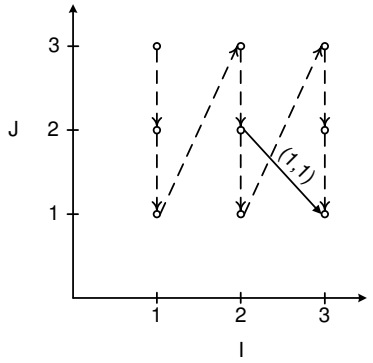
Loop skewing is a quite useful and widely used transformation for parallel processing and is combined with loop interchange [86, 126], which is used for the wavefront or hyperplane computation [76]. Lets look at a simple example shown in Figure 2.8a. Its ISDG is shown in Figure 2.8c and its dependence vectors are $\{(1,0),(0,1)\}$, and none of the loops are parallelizable as described in Section 2.2.1. However, if we change the way the elements are traversed to the sequence as followed by the diagonal lines shown in Figure 2.8c, then all the elements along each diagonal line can be computed in parallel as they are independent of each other. Similarly, all the elements in the next diagonal can be computed in parallel and so on. This diagonal computation is called the wavefront computation. The iteration space is changed from a rectangle to a parallelogram, whereas the memory references remain the same to achieve

```
for I = 1 to n
    for J = n to 1 step -1
        A[I+1,J]=A[I,J+1]+C
    end for
end for
```

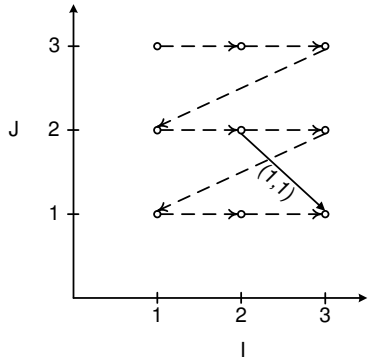
(a) loop reversal for code in Figure 2.7a

```
for J = 1 to n
    for I = 1 to n
        A[I+1,J]=A[I,J+1]+C
    end for
end for
```

(b) after loop interchange for code in Figure 2.9a



(c) ISDG for code in Figure 2.9a



(d) ISDG for code in Figure 2.9b

Figure 2.9: Typical use of loop reversal

loop skewing. The code after the loop skewing is shown in Figure 2.8b, and its iteration space and the dependence is shown in Figure 2.8d.

For a two dimensional space as given by the code in Figure 2.8a, the iteration space is redefined by introducing two new variables I and J. Where I and J are defined as:

$$I = i + j - l_i \tag{2.5}$$

$$J = j \tag{2.6}$$

where l_i is the lower bound for the outer loop i in Figure 2.8a. Similarly, bounds for the outer and inner loops are modified accordingly as shown by code in Figure 2.8b. The way memory is referenced, is not changed in the transformed code and, therefore, iteration variables are written in terms of a new iteration space as given by Equations 2.5 and 2.6.

2.2.4 Loop Reversal

Loop reversal reverses the direction of the loop iteration on which it is applied [122]. It is often used with the other loop transformations like the loop interchange, loop skewing and loop fusion [13, 128]. It can also improve the cache performance. The loop reversal of some loop i negates the i^{th} entry of each distance vector d associated with the loop, similarly, the corresponding entry in all the direction vectors is reversed. The loop reversal is legal if each reversed distance vector d' is lexicographically positive. Lets look at how the example in Figure 2.7a can be made suitable for loop interchange after loop reversal. The code in Figure 2.7a has distance vector $\{(1,-1)\}$ and as described earlier in Section 2.2.2, the code is not suitable for the loop interchange. After loop reversal shown in Figure 2.9a, the distance vector is changed to $\{(1,1)\}$, which is perfect for the loop interchange as shown in Figure 2.9b. It is also used to enable loop fusion when the two loops have different directions.

2.2.5 Strip Mining

Strip mining or loop sectioning is often used by vectorizing compilers to divide a single loop into two nested loops [6, 9, 81]. The outer loop iterates between the consecutive strips and the inner loop iterates each iteration within a strip. The maximum number of iterations of the inner loop is equal to the maximum vector length of the machine. Thus a loop in Figure 2.10a is transformed to two nested loops in Figure 2.10b, where s is the size of the strip. It is used for SIMD compilation [124] as well as for improving the memory performance. It is always legal to apply the strip mining, however it adds a dimension to the iteration space as it splits a loop into two loops as shown in Figures 2.10c and 2.10d.

2.2.6 Loop Tiling

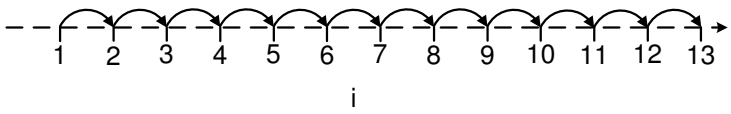
Strip mining can also be applied to multiple loops, called loop blocking. When block loops are interchanged and moved to the outer position, it is called loop tiling. It is primarily used to improve the memory performance [6, 42, 75]. It partitions the iteration space into tiles/blocks, so that the memory access in the loop remains in the cache. The tile size can also be chosen to suit the available vector operations in a vector machine. The program in Figure 2.11a with distance vector $\{(1,0)\}$ is first stripmined to the code in Figure 2.11b with the distance vector $\{(0,1,0,0)\}$ and then the loops are interchanged to Figure

```
for i = 1 to n
    A[i]=B[i-1]+C
end for
```

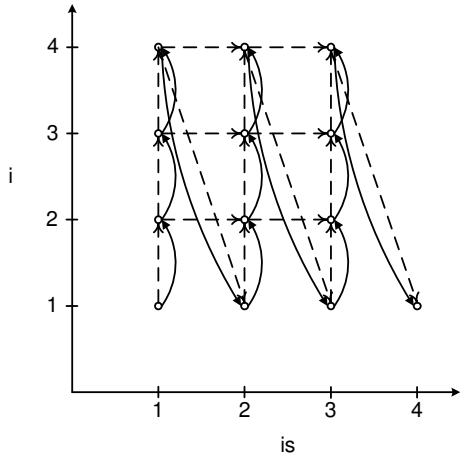
(a) code before strip mining

```
for is = 1 to n step s
    for i = is to min(n,is+s-1)
        A[i]=B[i-1]+C
    end for
end for
```

(b) code after strip mining



(c) ISDG before strip mining with $n = 13$



(d) ISDG after strip mining with $n = 13$ and $s = 4$

Figure 2.10: Strip mining

```
for i = 1 to n
    for j = 1 to m
        A[i,j]=A[i-1,j]+C
    end for
end for
```

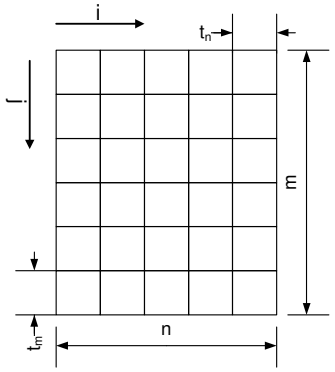
(a) original loop

```
for it = 1 to n step t_n
    for i = it to min(n,it+t_n)
        for jt = 1 to m step t_m
            for j = jt to min(m,jt+t_m)
                A[i,j]=A[i-1,j]+C
            end for
        end for
    end for
end for
```

(b) Loop blocking: strip mining the two loops

```
for it = 1 to n step t_n
    for jt = 1 to m step t_m
        for i = it to min(n,it+t_n)
            for j = jt to min(m,jt+t_m)
                A[i,j]=A[i-1,j]+C
            end for
        end for
    end for
end for
```

(c) Loop tiling: loop interchange of the code in Figure



(d) Iteration Space

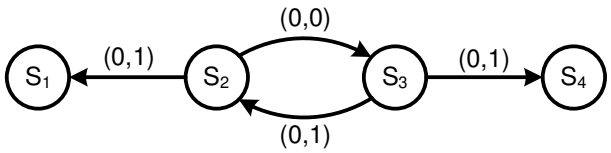
Figure 2.11: Loop Tiling

```
for i = 1 to n
    for j = 1 to m
S1      A[i,j]=B[i,j-1]+5
S2      B[I,j]=C[I,j-1]*2
S3      C[I,j]=1/B[I,j]
S4      D[I,j]=C[I,j-1]-1
    end for
end for
```

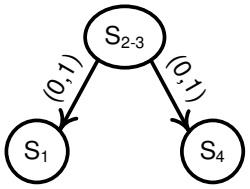
(a) original code

```
for i = 1 to n
    for j = 1 to m
S2      B[I,j]=C[I,j-1]*2
S3      C[I,j]=1/B[I,j]
    end for
end for
for i = 1 to n
    for j = 1 to m
S1      A[i,j]=B[i,j-1]+5
    end for
end for
for i = 1 to n
    for j = 1 to m
S4      D[I,j]=C[I,j-1]-1
    end for
end for
```

(b) After loop distribution



(c) Dependence graph for code in Figure 2.12a



(d) Dependence graph for code in Figure 2.12b

Figure 2.12: Loop distribution

2.11c with distance vector $\{0, 0, 1, 0\}$ to complete loop tiling. $t_n \times t_m$ defines the tile size. The outer two loops in Figure 2.11c step the tiles and the inner two loop steps the elements in each tile. Loop tiling is legal if loop interchange is legal.

2.2.7 Loop Distribution

Loop distribution or loop fission splits a single loop into more than one loop, showing the same iteration space as the original [67, 86]. Each loop contains a subset of the statements of the original loop. It is used to improve the memory performance and also used to remove loop carried dependences [83]. We look at the example in Figure 2.12a, its dependence graph is given by Figure 2.12c. There is a loop carried dependency at loop j. Strongly connected components

```
for i = 1 to n
S1    A[i] = B[i] + 1
end for
for i = 1 to n
S2    C[i] = A[i] / 2
end for
for i = 1 to n
S3    D[i] = 1 / C[i+1]
end for
```

(a) original code

```
for i = 1 to n
S1    A[i] = B[i] + 1
S2    C[i] = A[i] / 2
S3    D[i] = 1 / C[i+1]
end for
```

(b) Illegal loop fusion

```
for i = 1 to n
S1    A[i] = B[i] + 1
S2    C[i] = A[i] / 2
end for
for i = 1 to n
S3    D[i] = 1 / C[i+1]
end for
```

(c) Legal loop fusion

Figure 2.13: Loop Fusion

are found in the dependence graph, then the statements in a strongly connected component are kept in the split [67]. The order of execution among the different loops is determined by the acyclic graph among the strongly connected components as shown in Figure 2.12d. The code is transformed as given by Figure 2.12d, where S_2 and S_3 are kept in the same loop followed by the loops containing S_1 and S_4. The resulting code is shown in Figure 2.12d, where the loops containing S_1 and S_4 can be executed in parallel.

2.2.8 Loop Fusion

Loop fusion is the opposite of loop distribution and it merges two or more adjacent loops with the same loop limits into one loop. When the loop limits do not match then other loop transformations like loop peeling [11] can be applied to match the loop limits. The fused loop can increase cache locality when all the loops access the same memory area in the cache [11, 61, 83]. It is also used with other loop transformations to improve the cache performance [82, 130]. After fusion, the larger loop bodies enable more effective scalar optimizations such as common subexpression elimination and instruction scheduling [105].

The loop overhead is also decreased after the loop fusion.

Loop fusion is legal when all the dependence relations are preserved. Lets look at the code in Figure 2.13a. In this figure, all three loops have the same limits so they can be merged together. When, we merge all three loops, the code in Figure 2.13b is obtained. This fusion is illegal, as the dependence relation is not preserved. The original code has dependences $S_1 \delta S_2$ and $S_2 \delta S_3$, whereas, the code after fusion has the dependences $S_1 \delta S_2$ and the loop carried $S_3 \delta^{-1} S_2$. The fused loop will not generate correct results as none of the values of C used by S_3 are created in S_2. However, if only the first two loops are merged, then, the dependence relation is preserved and therefore is legal as shown in Figure 2.13c [82].

2.2.9 Loop Unrolling

Loop unrolling replaces copies of the loop with u copies of the loops and then iterates with step u. The number of copies, u, is called the unrolling factor and the original loop is called the rolled loop. It reduces the overhead of executing the loop and increases the chance of instruction level parallelism as it increases the block size and the scheduler can pack more instructions. Similarly, in a larger block there are more chances of the common-subexpression elimination and the induction variable optimization. It also enables software pipelining which is discussed in the next section. A simple example of loop unrolling is shown in Figure 2.14. The underlying machine can issue one load, one add and one add with a constant in one cycle. All instructions have a single clock latency except the load instruction which takes two cycles. Figure 2.14b shows the assembly code of the loop body in Figure 2.14a, which takes 4 cycles. When the loop body is unrolled 4 times and the registers are allocated to avoid the interference, it takes 16 cycles as shown by Figure 2.14c. When instruction scheduling is applied to the unrolled loop, it takes 7 cycles as some independent instruction can be parallelized as shown in Figure 2.14d.

2.2.10 Software Pipelining

Most parallelism can be achieved when the loops are fully unrolled. The instruction scheduler then tries to optimally pack the instructions, however, this will increase the code size. If loop is unrolled to some reasonable factor, a pattern can be seen repeating itself again and again, which can be considered as a kernel in the loop until that pattern is finished. Lets again look at the example in Figure 2.14b . The loop is unrolled 12 times and after which instruction

```
for i = 1 to n
    sum=sum+a[i]
```

(a) original code

```
LD R1,0(R0)
nop
ADD R2,R2,R1
ADDI R0,R0,#4
```

(b) Loop body in assembly language

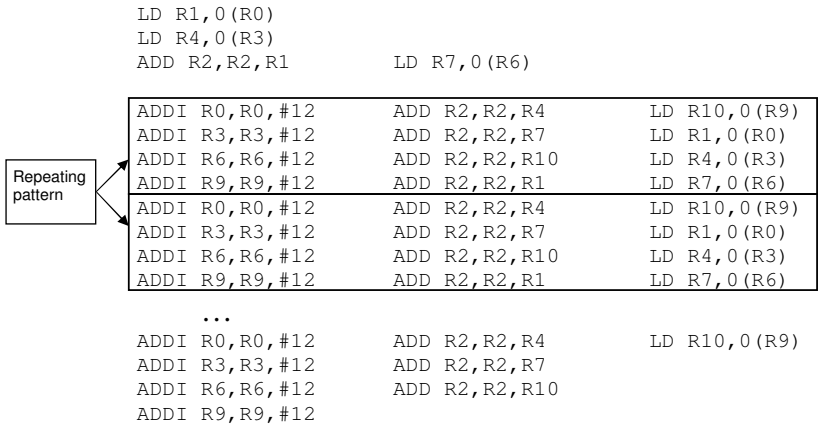
```
LD R1,0(R0)
nop
ADD R2,R2,R1
ADDI R0,R0,#12

LD R4,0(R3)
nop
ADD R2,R2,R4
ADDI R3,R3,#12

LD R7,0(R6)
nop
ADD R2,R2,R7
ADDI R6,R6,#12

LD R10,0(R9)
nop
ADD R2,R2,R10
ADDI R9,R9,#12
```

(c) Loop body unrolled 4 times and allocate registers

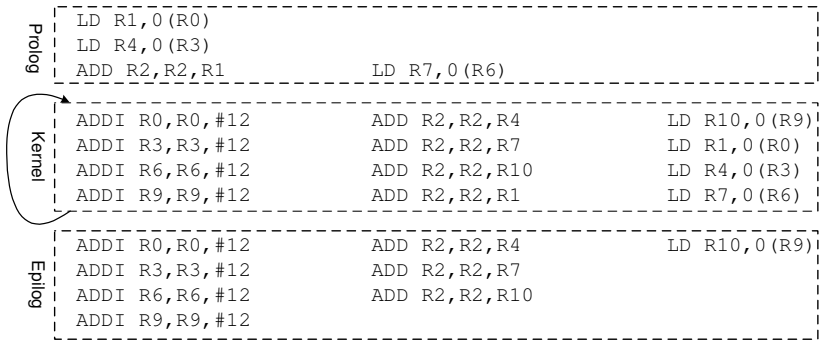
```
LD R1,0(R0)
LD R4,0(R3)
ADD R2,R2,R1        LD R7,0(R6)
ADDI R0,R0,#12      ADD R2,R2,R4        LD R10,0(R9)
ADDI R3,R3,#12      ADD R2,R2,R7
ADDI R6,R6,#12      ADD R2,R2,R10
ADDI R9,R9,#12
```

(d) Loop unrolling with Instruction scheduling

Figure 2.14: Loop Unrolling



(a) Loops unrolled and with instruction scheduling



(b) Software Pipelined

Figure 2.15: Software Pipelining

scheduling is applied. We obtain the code in Figure 2.15a. A repeating pattern can be easily identified, which is called the kernel. That code can be called in a loop until it vanishes as shown in Figure 2.15b. The code before the kernel is called the prolog and after the kernel is called the epilogue. This way the code size is reduced as compared to a fully unrolled loop and the same parallelism is achieved.

2.3 Summary and Conclusion

In this chapter, we discussed the different concepts and techniques that focus on loops as a source of performance improvement. Crucial in any loop optimization is the notion of dependence as they contribute the main barrier towards parallelization. First, this chapter gives an overview of the data dependences. Later, it describes few loop transformations which are oftenly used in reconfigurable systems.

3

Recursive Variable Expansion

In this chapter, we introduce a transformation called Recursive Variable Expansion (RVE). This transformation is meant for high performance computing where FPGA area is not the major concern and performance is the main objective. In this chapter, we present the basic idea of our transformation and how it provides high degree of parallelism by removing data dependences. This is achieved by backward substituting for those variables which are creating dependences. This chapter highlights the acceleration that can be achieved when area is not a constraint. Furthermore, we distinguish between two types of expressions, namely polynomially and exponentially expanding.

More specifically, the contribution of this chapter is:

- A loop transformation technique called **Recursive Variable Expansion** (RVE), which removes all the data dependences from the program; then, the parallelism is only bounded by the amount of resources one has. In the suggested transformation, we have assumed that the area available on the FPGA is infinite. In contrast to prevalent loop transformation techniques which are basically developed for computers with multiple processors, the suggested single transformation exploits the flexibility and area of FPGA to give more parallelism without making wide selection and scheduling among other loop transformations.
- We obtained speedups of up to 77 times on Virtex II Pro platform FPGA compared to the software only implementation for the considered kernels running on a PowerPC processor.

The chapter is organized as follows. In section 3.1, we discuss the related work. Section 3.2 presents a motivational example in which we compare RVE with another parallelizing transformation. Section 3.3 formally describes the

Recursive Variable Expansion along with its benefits and limitations. Section 3.4 presents the experimental validation and results obtained.

3.1 Related Work

The idea behind RVE is not new. After the publication of our first paper, we found the reference to the work of Muroaka et al. [69], who in 1972 proposed a similar technique that he framed *statement substitution* [69, 86] . Kuck et al. claimed that statement substitution extracts more parallelism than any other transformation, when there is unlimited parallelism available [67]. The problem is that they did some theoretical calculation and no machine was built, as there was no machine which has that much parallelism. Later, similar techniques like *look ahead computation* [37, 38, 63, 73] and *block back-substitution* [102] were proposed. In all these techniques, the recurrence is iterated M times, expanded and rearranged to calculate the result of M iterations of the original recurrence. In these transformations, there are lot of redundant computations, however the critical path is reduced by using *tree height reduction algorithm* [12, 16, 64, 65, 69, 71, 86, 92].

Our RVE transformation is different from these techniques as recurrence is iterated, expanded and rearranged to the full extent of the loop. It does not stop there, it is further extended to other loop bodies. This makes it suitable even for small loops, which otherwise do not exploit the parallelism and there are free hardware resources available. In contrast to block back-substitution, it is not limited to the innermost loop iteration.

A large part of programs has at least a first order recurrence equation [30]. Recurrence equations are one of the most difficult parts to parallelize [69]. A lot of work has been done to solve linear recurrence equations in parallel [25, 30, 40, 46, 56, 59, 63, 73, 74, 84, 86, 100, 109, 116, 121]. A formalization of the problem and resulting graphs are presented in [59]. An algorithm for computing a general class of recurrence equations using *statement substitution* and tree height reduction algorithm is given in [86]. Kogge and Stone et al. [63] present *recursive doubling decomposition algorithm* to solve m^{th}-order recurrence equation for parallel machines like Illiac IV. Buzbee and few others used the cyclic reduction technique to solve the problem [25, 30, 54]. Partition method is also used to solve such equation, which takes fewer resources than cyclic reduction [29, 40, 121].

One important technique that we will be using in RVE is tree height reduction. It computes an arithmetic expression by making a parse tree with the

lowest height. An *arithmetic expression* is any well formed string containing atleast one operator and two variables. It can be composed of variables, operators and brackets. Here, we assume that operators are only of four types (i.e. $+, -, \times, /$). These arithmetic expressions can be computed in many ways by making parse trees for the expressions. The goal is to compute the expression with a parse tree which has the lowest height. This problem is called *tree height reduction*. In tree height reduction, the arithmetic expression is tranformed to another equivalent expression by using associative, commutative and distributive laws, such that the tree height is the lowest. Tree height reduction techniques have been studied by a number of researchers. Many issues have been investigated such as the optimal way to achieve the best tree height reduction [12, 16, 64, 65, 69, 71, 86, 92]. What are the bounds for such algorithms [14, 23, 24, 66, 70, 71] ? How many resources are required? [14,24,30,66,71]. All these approaches are purely theoretical and most of them have assumed that all the operators take the same time, except [65] and [71], where different operators can take different time.

Our work is different from the earlier in two ways.

1. All the earlier transformations were applied on a basic block, whereas we propose to do it for whole kernel.
2. We present a working implementation that is also used for validation purposes.

3.2 Motivational Example

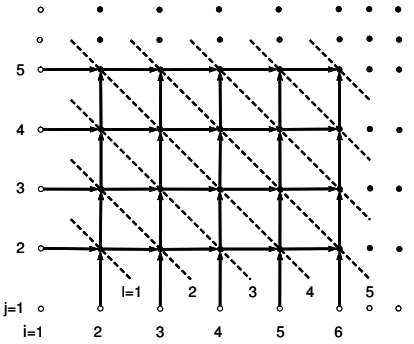
To clearly explain the loop transformation proposed by us, we use a motivational example (see Figure 3.1a). Figure 3.1c shows the iteration space with dependences among the various iterations of the loop nest. During the execution, $A[3, 4]$ is written in iteration $(3, 4)$ and read in iterations $(4, 4)$ and $(3, 5)$, so the distance vectors are $(4 - 3, 4 - 4), (3 - 3, 5 - 4) = (1, 0), (0, 1)$. Without any kind of transformation, none of the loops can be parallelized as given by Equation 2.4. Statement reordering is also not possible due to $S_1 \delta S_2$, which is used by instruction scheduling in the backend of the compiler. It is not useful to implement this loop nest on FPGA because no parallelism will be extracted and implementation will only decelerate the overall application as the FPGA operates at a lower frequency compared to the general purpose processor (GPP).

```
for i = 2 to n
  for j = 2 to m
S1  A[i, j]= A[i-1, j] + c
S2  B[i, j]= A[i, j]+ A[i, j-1]
  end for
end for
```

(a) A Simple example code

```
for I = 2 to m+n
  for J = max(2,I-n) to min(m,I)
S1  A[I-J, J]= A[I-J-1, J] + c
S2  B[I-J, J]= A[I-J, J]+ A[I-J, J-1]
  end for
end for
```

(b) After Loop Skewing



(c) Iteration Space after loop skewing showing dependences among the iterations. White points are inputs and black points are computed.

Figure 3.1: Motivational Example

Let's look at two ways to parallelize the example in Figure 3.1a. First is loop skewing, which is widely used loop optimization for such kind of code and second is Recursive Variable Expansion, which will be introduced to show that it outperforms loop skewing by consuming more area. We have tried to estimate time and area for both the transformations. For estimation, we have assumed that each addition takes one cycle and the area taken by the FPGA is directly proportional to the number of terms being added.

3.2.1 Applying Loop Skewing Transformation

As shown in Figure 3.1c, if the sequence to access the array A is changed to a sequence followed by slanted dashed lines, then it does not violate the dependences between the iterations and it also keeps the same program behavior. For the given example , first the single black point on slanted line with $I = 1$ is executed, then the two black points on slanted line with $I = 2$ at parallel, then the three black points on slanted line with $I = 3$ simultaneously, and so on. This sequence of execution is valid, since it does not modify the dependences

of the program and all the points executed in parallel are independent to each other. This particular type of transformation is called loop skewing [13, 126].

In the transformed code (see Figure 3.1b), suppose m and n are greater than $3 + 4 = 7$; then $A[3,4]$ is written in iteration $(7,4)$, while the same is read in iteration $(8,4)$ and $(8,5)$, so the distance vectors are $(8-7, 4-4), (8-7, 5-4) = (1,0), (1,1)$, which shows that the inner loop is parallelizable by the rule discussed above.

Area estimate for Loop Skewing. The number of operands to be added is 4 (i.e. $A[I-J-1, J]$, c, $A[I-J, J]$, $A[I-J, J-1]$) for each iteration of the inner loop. As all the iterations of the inner loop are expanded, then the maximum number of operands to be added will be $4\times$ *maximum number of iterations of the inner loop* for some outer loop. Since the lower bound for the inner loop is $max(2, I-n)$, the lower bound for the inner loop remains 2 until I becomes $n+2$, so the upper bound for the same I becomes $min(m, n+2)$. Using this upper and lower bound, the number of iterations for the inner loop is going to be $min(m, n+2) - 1$. After this, as the lower bound is increased by one with the increase of I, it either increases the upper bound with the same amount if I is smaller than m, or it remains m, if m is smaller than I, which shows that the maximum number of iterations for the inner loop is $min(m, n+2) - 1$. So the maximum terms that need to be added are $4 \times (min(m, n+2) - 1)$.

Time estimate for Loop Skewing. The inner loop is parallelizable, if it is fully expanded on to the FPGA and executed in parallel, then the time for each iteration of the outer I loop is 2 *cycles* for the two adders which cannot be executed in parallel as $S_1 \delta S_2$. The total time to execute the transformed code will be $2 \times (m + n - 1)$ *cycles*, as the number of times the outer loop will be iterated is $m + n - 1$.

3.2.2 Applying Recursive Variable Expansion Transformation

When Loop Skewing is applied, only the inner loop is parallelizable. However, let us take the same example and fully unroll both loops. As it is, unrolling will not provide any parallelism. If we look at the example in Figure 3.1a with $i = 4$ and $j = 3$ (see Figure 3.2), $A[3,3]$ is needed to compute $A[4,3]$, however if we replace $A[3,3]$ with its computation, then $A[4,3]$ is no more dependent on $A[3,3]$, but rather on $A[2,3]$. We repeat this procedure until $A[4,3]$ is only the function of input variables, then it can be computed without waiting for other results, if provided enough resources. The same can be done for every statement.

```
A[4, 3]= A[3, 3] +c
       = A[2, 3] +c +c
       = A[1, 3] +c +c +c
```

Figure 3.2: Recursively substituting the values

Area Estimate for Recursive Variable Expansion. The way the transformation is applied, we can get the number of operands to be added in example of Figure 3.1a by a recurrence equation. Given that the statement S_1 is only dependent on i. Suppose the number of operands to be added in S_1 is denoted by $T(i)$ and is given by $T(i) = T(i-1) + 1$. The solution to this equation is $T(i) = i$. Similarly, suppose the number of operands to be added in S_2 is denoted by $S(i)$ is given by $S(i) = 2T(i)$, which solves to $S(i) = 2i$, so the number of operands to be added for both the statements is $\tau(i) = S(i) + T(i) = 3i$. If we expand, as discussed earlier, both the statements for all iterations until they cannot be expanded further, then the total number of operands to be added τ is given by

$$\tau = \sum_{i=2}^{n}\sum_{j=2}^{m}\tau(i) = \frac{3}{2}(m-2)(n^2+n-2) = O(n^2m) \tag{3.1}$$

Suppose the output of the loop nest or variable which is used (live variable) later is only $B[n, m]$, then there is no need to expand all the terms in all iterations on to FPGA, because after expansion, $B[n, m]$ is a function of only inputs, which can be computed readily. Since we are doing this expansion at compile time, all the other terms can be discarded keeping only $B[n, m]$ expanded on to the FPGA. Then the total number of operands to be added is $\tau = S(n) = 2n$.

Time Estimate for Recursive Variable Expansion. The expressions expanded in width contain only addition operators and there are no dependences in the expression. The addition operator is associative, therefore the most efficient way to add them efficiently is adding in parallel in a complete binary tree fashion (see Figure 3.3), where each level takes one cycle. Then $O(\log n)$ cycles is required to add n terms. Since all the terms are expanded and executed in parallel then maximum time taken by any statement will be the one which has maximum number of operands to be added, which in this example is $2n$. So the overall time will be $O(\log n)$ cycles.

Time and area estimation for both transformations is summarized in Table 3.1. It shows that the time to compute the code in Figure 3.1a is $O(logn)$ in Re-

	Time (cycles)	Area (no. of operands to be added)
Loop Skewing	$2(m+n-1)$	$4 \times (min(m, n+2) - 1)$
Recursive Variable Expansion	$\log(n)$	$\frac{3}{2}(m-2)(n^2+n-2)$

Table 3.1: Time and Area estimates for different transformations for example in Figure 3.1a

cursive Variable Expansion as compared to linear in m or n for loop skewing. This speedup is at the expense of larger area. However, the area required is not large when the live variable is only $B[n, m]$ as it can be only $2n$.

3.3 Recursive Variable Expansion

Recursive Variable Expansion (RVE) is a transformation which removes all data dependences among different statements in a program, thereby making suitable for parallel execution. The basic idea is the following. If any statement G_i is waiting for some statement H_j to complete for some iteration i and j respectively due to some data dependency, both of the statements can be executed in parallel, if the computation done in H_j is replaced with all the occurrences of the variable in G_i which create the dependency with H_j. This makes G_i independent of H_j at the cost of redundant computation. Similarly, computations can be substituted for all the variables which creates dependences in other statements. This process can be repeated recursively till all the statements are function of known values and all data dependences are removed. Hence, all the statements can be executed in parallel provided the required resources are available. The transformation is named *Recursive Variable Expansion*, because all variables that create dependences are recursively substituted with their values. The suggested loop transformation is applied in the following steps

- Given a program, loops that spend most of the time in the whole program are identified using profiling information.
- The identified loops are fully unrolled.
- The values for every assignment statement are recursively substituted.

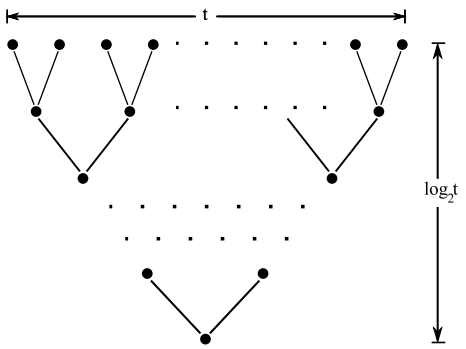


Figure 3.3: Best case for binary operation on t terms

- Constant folding is applied and then any possible computation during the compile time is done to reduce the computation at runtime.
- Tree height reduction algorithm is applied [69]. The resulted tree will provide the sequence to parallelize the statements with least number of levels.

Since the values for the variables are recursively substituted, thereby repeating the same pattern again and again until all the statements are expanded in width and are functions of only known values. This repetition of the same pattern resulting in known independent variables gives the ability to largely parallelize the operations depending only on the precedence of the operators and whether the operators are associative. For associative binary operators, a statement of t terms is expected to be computed in $O(\log_2 t)$ levels (see figure 3.3) by using recursive doubling decomposition algorithm or tree height reduction [63, 68].

Do all the expressions solve so efficiently? In the next section, we classify the type of expression which can affect the way RVE can be applied.

3.3.1 Classification of Expressions

We distinguish between two different kinds of recurrence equations: first results in an expression with a polynomial number of terms and second with an exponential number of terms.

The polynomial recurrence equation is defined as follows

$$A(n, m) = bA(n-1, m) \oplus c \tag{3.2}$$

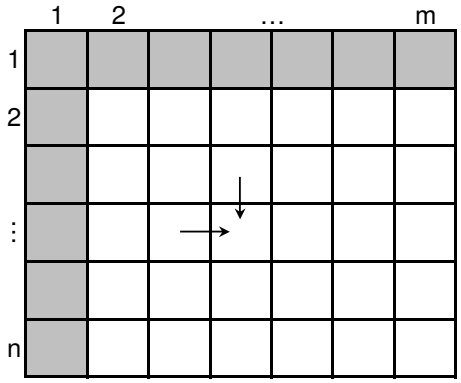


Figure 3.4: Recurrence in two variables. Grey boxes are the inputs

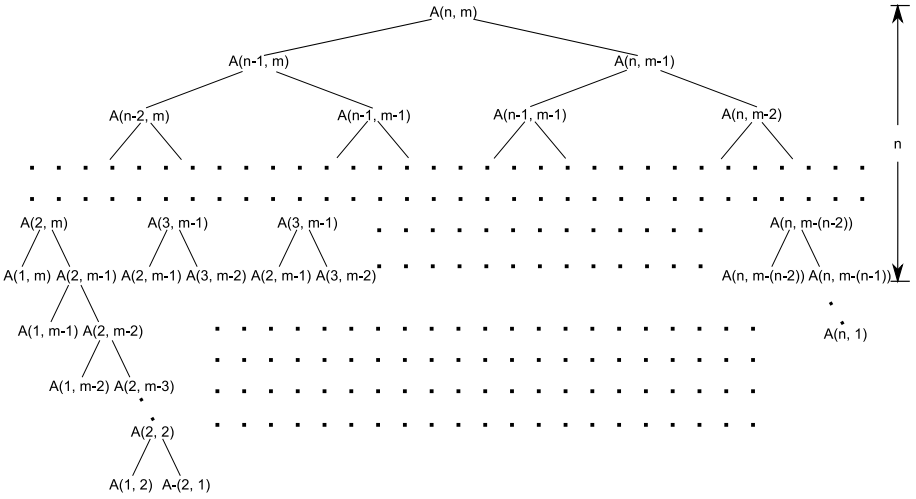


Figure 3.5: Lower bound for the worst case of Recursive Variable Expansion

where $\oplus$ is a binary operator. It is called first order recurrence equation as the solution is dependent on one of its predecessor. Such an expression expands polynomially when RVE is applied. When $\oplus$ is an associative operator and we apply tree height reduction to the expanded expression, it becomes $\log(n^k) = k \log n$, which is logarithmic in input size. The code in Figure 3.1a is an example of such recurrence equation.

Second and higher order recurrences expand exponentially when applying RVE. Some of the exponentially expanded expression can be reduced to polynomial number of terms. Lets look at one such example of second order recurrence equation, which expands exponentially but can be reduced to polynomial number of terms.

$$A(n, m) = A(n-1, m) \oplus A(n, m-1) \tag{3.3}$$

Suppose $m \geq n$, then the recursion tree [114] for this recurrence is shown in Figure 3.5. Figure 3.4 shows the way, it is computed. It is a recurrence in two index variables and both reduce by one at each level. Since every node has two children, then the number of nodes at level n are 2^{n-1}. It means that the number of terms which needs to be computed are at least exponential in 2. Even if it is assumed that there is infinite area, it would be useless to apply RVE as the time to compute after tree height reduction will be linear as $\log(2^{n-1}) = O(n)$. This becomes even worse for higher order recurrences.

However, if we look at the recursion tree more closely, we find many duplicate terms in the tree. Lets assume that $\oplus$ in Equation 3.3 is $+$. Can we take benefit of this structure? Yes, we can, no matter how large the expression is expanded, we know that the expression will ultimately be only a function of input variables.There are only $n + m - 1$ input variables, therefore, the unique terms are only $n + m - 1$. The operator between each term is addition and addition of c unique x terms can be written as cx. The equation 3.3 can be represented as.

$$\begin{aligned} A(n, m) \;=\; & a_1 A(1,1) + a_2 A(2,1) + \cdots + a_n A(n,1) \\ & + a_{n+1} A(1,2) + a_{n+2} A(1,3) + \cdots \\ & + a_{n+m-1} A(1, m) \end{aligned} \tag{3.4}$$

where $a_1, a_2, \cdots, a_{n+m-1}$ are the constant multiplication factor for input terms $A(1,1), A(2,1), \cdots, A(1, m)$ respectively. To find each constant multiplication factor, we need to count the respective terms. Since there are an expo-

nential number of leaf nodes, a naive method of counting will take exponential time. As there is a lot of overlapping, it is beneficial to use the bottom up approach of dynamic programming as shown in Algorithm 3.1. We maintain an array of input variables with count of each variable. Initially the count for all variables is zero. The *count* function on a subexpression keeps track of the count of input variables in that subexpression. The maximum number of input variables for which the count variable can be modified is $n + m - 1$, therefore this counting can be done in $nm(n + m - 1)$ steps. This expression reduction is done at compile time and Algorithm 3.1 ensures that the compilation may not take exponential time.

Algorithm 3.1 Counting exponential leaves

```
for i = 2 to n
  for j = 2 to m
    count(A(n, m)) =count(A(n − 1, m)) +
count((A(n, m − 1))
  end for
end for
```

We can reduce the exponential number of terms to n^k, where k is some constant, if the following conditions are met:

1. Number of inputs are polynomial
2. All the operators in the expanded expression are associative with respect to each other.
3. The operators applied on some number of the same terms in the expanded expression reduce to one operator which can be a same (like max.) or a different operator. For example, addition of the same terms can be reduced to a single multiplication, similarly multiplication of the same terms can be reduced to power and subtraction to division.

Let's now look at another example of Smith-Waterman (SW). SW is a dynamic programming based problem, which if expanded fully with RVE can generate an exponential number of terms which cannot be reduced to a polynomial number of terms. It is defined by the following recurrence equation:

$$F(i,j) = \max \begin{cases} F(i,j-1) + g \\ F(i-1,j-1) + x(i,j) \\ F(i-1,j) + g \\ 0 \end{cases} \tag{3.5}$$

Equation 3.5 is a third order recurrence equation. Like Equation 3.3, it also generates exponential number of terms, as every term generates four children and the level is decreased by one. The max operator also supports reduction as $\max(c,c,c) = \max(c) = c$. However, the problem is that it can not be reduced to a polynomial number of terms as the operator between the four subexpressions is max. Although, max alone is associative in nature, but max and addition are not associative with each other. Therefore, a term in one subexpression cannot be added to a term in other subexpression, which hinders the reduction to a polynomial number of terms. We present ways to effectively apply RVE on SW and other similar dynamic programming problems in Chapter 5 and 6.

A loop body with control structure can also grow into a exponential number of terms when applied with RVE. Lets look at a simple example in Figure 3.6a, which has a control structure in a loop. When we apply RVE to this, the number of statements will expand exponentially as shown in Figure 3.6b. We deal with one type of control structure in dynamic programming problems by using predicated execution in Chapter 5. The general case of control structure in a loop structure will be dealt in future work.

The success of RVE depends on the statements having associative binary operators. Because once such statements are expanded in width and are functions of known and independent variables, then these associative operators allow us to largely parallelize the operations by applying tree height reduction. Even if there are some non-associative operators, they can be handled in a special way [69]. An expression of t terms and depth d of parenthesis can be computed in $O(1+2d+\lceil \log_2 n \rceil)$ levels using at most $\lceil \frac{n-2d}{2} \rceil$ processing elements [70].

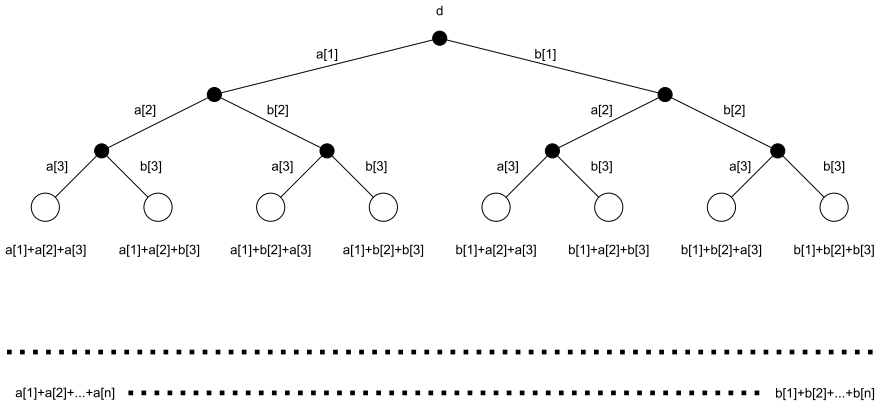
3.3.2 Constraints of RVE

RVE can be applied to a class of problems, which satisfy the following conditions.

1. The bounds of the loops must be known at the compile time.

```
d = 0
for i = 1 to n
   if (d > c[i])
      d = d + a[i]
   else
      d = d + b[i]
end for
```

(a) Loop with a control structure



(b) Graph showing exponential expansion

Figure 3.6: Example with a control structure

2. The indexing of the variables should be a function of control variables and/or constants.

3.3.3 Benefits of RVE

If these restrictions are satisfied, the following benefits hold:

- It removes all the data dependences from the part of the program on which it is applied, then the parallelism is only bounded by the amount of resources one has.
- Unlike other loop optimizations, RVE is more appropriate for hardware implementation, as it does not use memory to store intermediate results and hence minimizes the memory accesses. Initially all the input variables are read once and then all the computations are done using those values and finally the output is written back to the memory.
- This single transformation exploits more parallelism without making wide selection and scheduling other loop transformations.
- This transformation can also work with non-perfectly nested and unnormalized loops.

3.4 Experimental Results

This section presents the experimental setup, results and discussion. It demonstrates the benefit of our transformation. We compare an FPGA implementation to a pure software execution on the GPP. To this purpose, we have chosen four kernels. We look at area consumption and execution time for these kernels.

3.4.1 Kernels

As there is no support for floating point computations on the PowerPC processor included in Virtex II Pro platform, only integer version for the following kernels are chosen to make a fair comparison.

- SOBEL is a 3×3 convolution mask over an integer image.

- MM is a 16-bit integer matrix multiplication of a 12×6 matrix by a 6×4 matrix.
- FIR is a finite impulse response filter with 16 tap over 32 consecutive 8-bit elements.
- DCT is an integer implementation of 2D Discrete Cosine Transform.

Each application is written as a standard C program. The selected kernels have 2D (or more) non perfectly nested loops.

3.4.2 Software and Hardware Implementation

Each kernel executed on the PowerPC is referred to as the pure software execution. For the pure software implementation, the kernels written in C are compiled using GCC 4.2.0 with level 3 optimization, in which the inner loop is fully unrolled. The compiled codes are simulated for IBM PowerPC 405 processor immersed into the FPGA fabric, which runs at 250MHz. To estimate the time taken by the software only implementation, the machine instructions are counted and segregated into computation and memory access instructions using PSIM simulator [4].

To accelerate the kernels by parallelizing them on FPGA, we apply the Recursive Variable Expansion transformation as described in Section 3.3, which outputs only those statements that are used later in the program. The resulting expressions are free from any loop and are largely expanded. All the input variables required to compute the statements are read from the memory and stored in the registers on the FPGA. Since the VHDL code is intrinsically parallel, all the possible computations are performed in parallel according to the sequence provided by the tree height reduced expression. Finally the results are written back to memory. We have used two memory models: on chip memory and the DDR on the board, to evaluate the performance.

The Field-Programmable custom computing machine (FCCM) used in our experiment is Molen prototype targeted on the Virtex II Pro platform FPGA of Xilinx as described in [72]. Xilinx ISE version 8.2.022 , XST and ModelSIM SE are used to generate, synthesis and simulate the VHDL respectively.

3.4.3 Results

In this section, we describe the performance improvement due to RVE, when the different kernels are mapped on FPGA as compared to when mapped on

	Pure Software Execution					Hardware Execution on Virtex II Pro platform								
		On Chip		DDR				On Chip			DDR			
	Computation (cyc)	Mem (cyc)	Total (cyc)	Mem (cyc)	Total (cyc)	Frequency (MHz)	Computation (cyc)	Mem (cyc)	Total (cyc)	Speed up	Mem (cyc)	Total (cyc)	Speed up	Area (Slices)
Sobel	59	84	143	560	619	127.14	4	43	92.4	1.53	215	430.64	1.44	541
MM	797	1224	2021	8160	8957	118.67	2	102	219.1	9.22	966	2039.3	4.39	64792
FIR	1025	1536	2561	10240	11265	129.60	2	32	65.6	39.05	286	555.55	20.28	13850
DCT	6439	9147	15586	60980	67419	110.81	3	86	200.8	77.63	966	2186	30.84	1067552

Table 3.2: Performance and Area Utilization for Software only and Virtex II platform

GPP. The results are summarized in Table 3.2. The time is measured in clock cycles and area in case of FPGA is measured in slices. Since the PowerPC runs at 250 MHz. and the synthesis on Virtex II Pro platform gives lower frequency, the time for the execution on the Virtex II Pro platform is normalized according to the clock speed of the general purpose processor (PowerPC 405) to make a fair comparison. Then, based on the normalized time for Virtex II Pro, speedup is calculated. The speedup for the hardware implementation compared to the software one for the given kernels is between 1.5 and 77 times in case of on chip memory. A large gain is achieved in the computation time, but also the memory access is improved as the whole block of the memory is read instead of a variable and stored in the registers, from where data can be accessed efficiently. Currently, the bottleneck is memory, so the speedup can be enhanced further if the time to access the memory is improved. The memory access in DDR is more expensive than on chip memory, therefore the speedup is reduced and ranges from 1.4 to 30 times the software only execution. The results also depict that the speedup is higher for computation intensive kernels (DCT) than the memory access intensive (Sobel). The area occupied by the first three kernels is well within the area provided by the current FPGAs, like Virtex 4-XC4VLX200 contains 89,088 slices [5]. However, the area covered by the DCT is around 10 times more than the available in today's FPGAs. The area estimates for DCT are made by extrapolating the linear regression model between the number of operations and the area occupied. The time estimates for DCT are made by mapping only one out of 64 elements on the FPGA and taking that time for the whole block of 64 elements, as we have assumed unlimited resources, all the 64 elements can be mapped on to the FPGA and can be computed in parallel.

3.5 Summary and Conclusion

In this chapter, we have presented the Recursive Variable Expansion transformation which allows to eliminate dependences that may exist in certain kinds of loops. Given certain conditions, RVE allows to parallelize loops by performing backward substitution up to the point that the entire expression is defined in terms of the known variables. Moreover, conventional loop transformations require to read and write the result before and after any operation, which is easily avoided in our transformation. The whole computation starts after once reading the data from the memory. The input is fed to the operations, which compute and forward the results to other operations in the sequence till

the circuit generates the output of the computation. So our loop transformation reduces the memory access time, which is a very common bottleneck in many other transformations used to parallelize the code on FPGA. The RVE transformation removes all the data dependences in the applied region and a highly parallelized code is generated. This loop transformation was applied on four kernels taken from various applications and the generated code was implemented on Molen prototype and it showed speedup ranging from 1.5 to 77 times depending upon the intensity of the computation with respect to memory access. This performance gain was at the cost of more area on the FPGA. Three out of four kernels could be mapped on current FPGAs. We also identified the type of expressions according to the number of terms after expansion, when RVE is applied. The first type of expressions grows polynomially whereas the second type expands exponentially.

In the next chapter, we will look at the polynomial recursive expression and devise an automatic and flexible pipelining for Recursive Variable Expansion. It automatically chooses a largest part of the code to pipeline that can be mapped on the available area and data required to feed the pipeline is less than the available bandwidth.

Note.

The content of this chapter is based on the the following paper:

Z. Nawaz, O.S. Dragomir, T. Marconi, E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, *Recursive Variable Expansion: A Loop Transformation for Reconfigurable Systems*, proceedings of International Conference on Field-Programmable Technology 2007, pp. 301-304, Kokurakita, Kitakyushu, Japan, December 2007.

4

Pipelined Design for RVE

EVEN though the hardware capacity has substantially improved, a number of constraints still limit potential performance improvement, of which area is a dominant one. The other major constraint is the memory bandwidth for many data intensive kernels. Therefore, it would be useless to waste area for enhancing the parallelism, when the data to feed the circuit is not available at the required interval [15].

This chapter deals with the expressions that expand polynomially when RVE is applied as defined in the last chapter. The goal is to achieve the maximum parallelism given some limited hardware resources and bandwidth. We present an algorithm to automatically generate a pipeline design for the Recursive Variable Expansion (RVE) algorithm, which not only helps in restricting the area, but also tries to utilize free resources, if any. Pipelining is a technique, in which various similar tasks are sequenced and overlapped in time, so that all the available resources are being utilized at the same time by scheduling different tasks to different stages. In this chapter, we introduce a flexible pipelining design algorithm for RVE, which not only fulfills the area constraints on a given FPGA but also hides the memory access latency by overlapping the memory access with the computation. The proposed algorithm can be applied on the problems that fulfill the constraints listed in Section 3.3.2 as well as the following constraint.

- When a kernel produces more than one output variables, then the length of the generic expression for those output variables should be equal.

The contributions of this chapter are as follows:

1. For the loops that contain polynomially expanding expressions, we extend the RVE basic algorithm by proposing a pipelined design that take

area and bandwidth constraints into consideration. We only deal with those loops that expand polynomially.

2. We validate the pipeline design algorithm using a real world kernel, showing a comparable performance to the hand optimized implementation at the cost of more area.

The chapter is organized as follows. In the next section, we present the related work. Section 4.2 presents the background information. A formal description of our pipeline design is given in Section 4.3. The algorithm to solve the optimal pipeline design is described in Section 4.4. Section 4.5 describes the architecture to hide the memory access latency. Section 4.6 describes the experimental setup and the results are presented and discussed. Finally the chapter is summarized in Section 4.7.

4.1 Related Work

Extensive research has been done in the area of loop pipelining. Few techniques only work with intra-loop dependences in the loop nest. *Software pipelining* [7] is one such technique, in which the compiler generates a schedule where various iterations execute concurrently in an overlapped manner. This technique was primarily developed for VLIW architecture and is now also used for reconfigurable computing. It is used in the Garp Compiler [26], which only pipelines inner loops with intra-loop dependences. The other reconfigurable compilers which use the software pipelining are NAPA-C [44] and PICO [104]. In a similar approach [107], the iterative modulo scheduling of software pipelining is integrated with the retiming and slowdown [77] (that is used to pipeline synchronous circuit) to reduce the pipelining delays in the reconfigurable hardware. In addition to dealing with the intraloop dependences in the loop nest, our algorithm can also produce a pipeline for any type of the loop nest with the loop carried dependences in RVE polynomially expanded expression.

Loops with loop carried dependences are more difficult to parallelize and pipeline. A significant work has been done in exploiting the parallelism for the loop carried dependences. For example in the pipeline vectorization [123], various loop transformations like the loop unrolling, loop tiling, loop fusion and loop merging are used to remove the loop-carried dependences in the innermost loop. Beside this, the retiming technique [77] is used in pipeline vectorization [123] for an efficient pipelining. Some new loop transformations

like the unroll and squash [96] are also proposed to deal with the inner loops containing loop carried dependences. In contrast, our pipeline algorithm removes the loop carried dependences from all the loops, hence provide extra parallelism.

Some techniques use the dataflow graph approach instead of the conventional loop transformations. In these technique, the functions or loops waiting for some data may start computing as soon as the required data is available, which can be out of order. One of the earliest example is the technique described by Ziegler et al. [134], which uses FIFO buffers to synchronize between the subsequent stages. Another is called the Reconfigurable Dataflow control scheme [112], which is also applicable to nested loops with loop carried dependences. This approach uses the Tagged-Token execution model [118] to control the sequence of execution. A more recent technique called the pipelining of sequences of loops [99] uses a more fine grain synchronization and buffering scheme. An iteration of a loop starts before the end of the previous iteration, as soon as the data is available. Interstage buffers are maintained, which signal and trigger the subsequent stages. Therefore, the sequence of the production and consumption of the data can be different. Hash functions are used to reduce the size of the interstage buffers. As our pipeline algorithm is based on RVE, which removes all the loop carried dependences, therefore the computation can be done out of order.

As mentioned earlier, our pipelining technique is meant for RVE, which is similar to the techniques like back substitution [69], look ahead computation [63, 73] and block back-substitution [102]. In all these techniques, the recurrence is iterated M times, expanded and rearranged to calculate the result of M iterations of the original recurrence. The different pipelining approaches used for look ahead computation are clustered look ahead, scattered look ahead and block processing [94]. Identification of certain algebraic structures which allow to apply the well known look ahead computations has been done in [37, 38].

Our RVE technique is different as the recurrence is iterated, expanded and rearranged to the full extent of the loop. It does not stop there and further extended to the other loop bodies as discussed later in Section 4.6 for the DCT code. This makes it suitable even for sequence of small loops, where other techniques exploit limited parallelism even though hardware resources would be available. In contrast to the block back-substitution, it is not limited to the innermost loop iteration. Our pipelining approach is similar to block processing, in which inputs are processed in the form of non-overlapping blocks to gen-

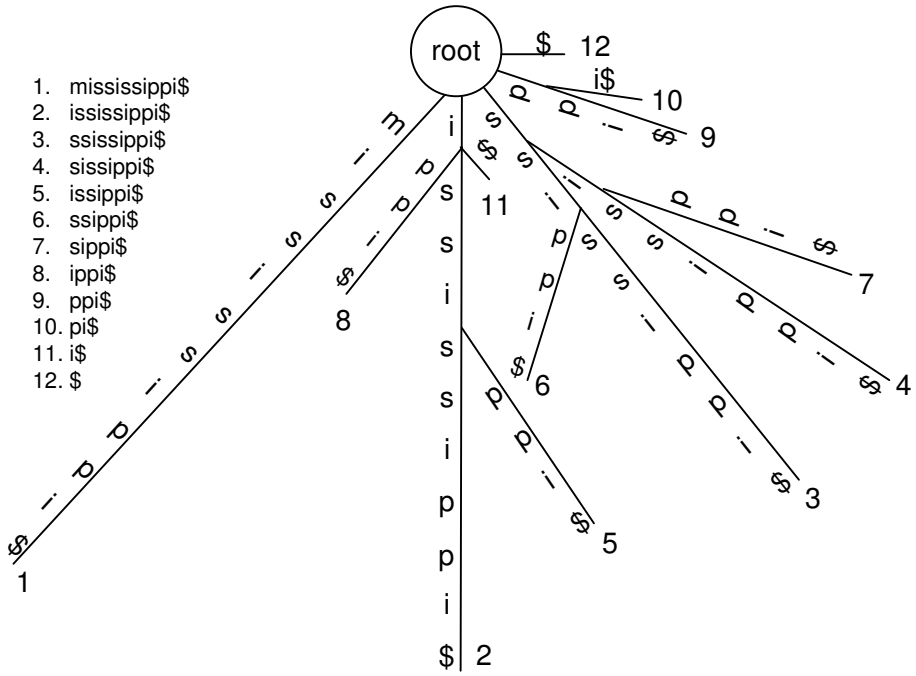
erate non-overlapping blocks of outputs. The difference in our and the earlier pipelining approach is in the way the block is chosen. As mentioned earlier, they expand the loop to a predefined size M and then make a circuit for that. In our algorithm, we expand the entire loop body loop bodies and then try to find the largest repeated pattern that satisfies the memory and area constraint. There are two advantages of doing this. First, we can find larger pattern even if the loop is small. Second, when all the loops are expanded, we can remove the redundant computations which are present due to intraloop dependences or redundant registers which contain the same variables. Since we have expanded all the loop bodies, there are more chances of finding redundant computations and registers. The core idea of our technique is that first we expand the recurrence to its full extent and then try to find the redundant computations and registers which can be removed without sacrificing the throughput.

4.2 Basic Concepts

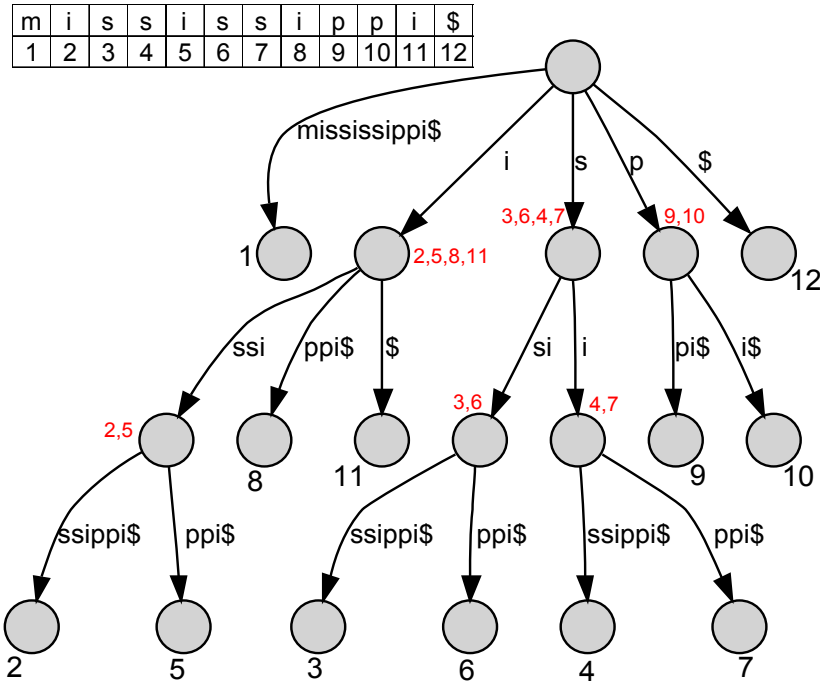
We first introduce the basic concepts that we need to formalize the problem statement. A string is a finite set of symbols from an alphabet Σ. The number of symbols in an alphabet set is defined by $|\Sigma|$. The length of a string T denoted by $|T|$ is the number of symbols in that string. Let Σ^* denotes the set of all finite length strings formed by using symbols from the alphabet Σ. The zero-length empty string, denoted by ε, also belongs to Σ^* . The *concatenation* of two string T and S is written as TS, i.e. the string T followed by the string S, where $|TS| = |T| + |S|$. We say that the string S is a *suffix* of the string U, if $U = TS$ for some $T \in \Sigma^*$. $T[i]$ denotes the i^{th} character of T. $T[i..j]$ is the *substring* $T[i]T[i+1]...T[j]$ of T. A *Kleene star* of a string T, denoted by T^*, is the set of all the strings obtained by concatenating zero or more copies of string T. We define $T^+ = TT^*$, which means that T^+ is the smallest set that contain T and all the strings that are concatenation of more than one copies of T.

4.2.1 Suffix Trees

A suffix tree is a data structure that is used to efficiently solve many string search related problems. We are going to use this data structure extensively to solve our pipeline design problem. This data structure is built by preprocessing the search string at the start of search. A simpler way to understand a suffix tree is to look at the *suffix trie* [48] first. We take an example string



(a) Suffix trie of mississippi



(b) Suffix tree of mississippi

Figure 4.1: Suffix tree for mississippi

$S =$ *mississippi* that ends with *$,* as end marker, then make a suffix trie of it as shown in Figure 4.1a. A suffix trie [48] is a type of tree which has at most $|\Sigma|$ branches from each node. The edge labels on the path from root to each leaf node i give a suffix $S[i..L]$, where L is the length of S. The starting character of the label of every edge coming out of a node is unique. Once a suffix trie is constructed for string S, searching a pattern s of length l in S is very simple because of the organization of the suffix trie for S. We start from the root and choose the path in the tree which matches the characters of the search pattern s until we exhaust all the characters of s or we do not find a path in the suffix trie starting with the corresponding character in s. When all the characters of s are exhausted, we got a match in String S, otherwise a mismatch, therefore it takes $O(l)$ time to find a pattern of length l. The reason for not using the suffix trie is its time to construct it, which is $O(L^2)$, which makes overall time as quadratic. Other best known string searching techniques like Knuth-Morris-Pratt (KMP) [62] and Boyer-Moore (BM) [22] take time $\Theta(L+l)$, which makes them better than suffix trie for the string matching purpose.

We can beat the KMP and BM algorithm in the pattern searching if we change the data structure to the Suffix tree, which is a suffix trie after *path compression*. The path compression is a process in which edges of the nodes with a single descendant are compressed to a single edge until a node with more than one descendant is found. Therefore, every internal node in the suffix tree except the root has at least two children. The suffix tree for $S =$*mississippi* is shown in Figure 4.1b. The advantage of suffix tree over suffix trie is that it is built in $O(L)$ time [115] as compared to $O(L^2)$ for a string of length L.

In this chapter, we are interested in finding *repeats*[1] in a string. Given a string S, a repeat r is any substring of S which is found in S at more than one position. We compare two approaches to find a repeat, one using the best known string matching algorithm and other using the suffix tree.

The simplest approach to find all the repeats is to start from a pattern of length 2 and find all the repeated expressions for all the possible patterns of length 2 in the expression E of length L. Increase the pattern length by one and try to find the repeated expressions for all the patterns of that length until we get some pattern length $l+1$ for which the repeated pattern is not found for any possible pattern of that length. This means that the last pattern length l for which there were some repeated expression or expressions is the largest repeat. The upper bound for l is $\frac{L}{2}$. If we use one of the best string matching algorithm

[1]Perhaps *repeating term* would be a better name, but the term *repeat* is a standard in the literature.

```
for i=1 to 5
  A[i]=0
  for j=1 to 4
    A[i]=A[i]+d[j]*i
  end for
  A[i]=A[i]>>8
end for
```

Figure 4.2: A simple example

like Knuth-Morris-Pratt [62], string matching for one pattern will take $\Theta(L)$, as there are $\Theta(L)$ possible patterns for any pattern length $l \leq \frac{L}{2}$. Therefore, to find repeated pattern for one pattern length will take $\Theta(L^2)$. Since the possible pattern lengths can be $\frac{L}{2} - 1$, ranging from 2 to $\frac{L}{2}$, it will take $\Theta(L^3)$ to find the optimal repeat.

Using the suffix tree as the data structure is a well known in bioinformatics and is the best known repeat finding method [48]. Lets again take the example S =*mississippi* and look at its suffix tree in Figure 4.1b. Every internal node except the root in suffix tree has at least two children. It essentially means that the string s which is made by concatenating the labels of the edges from the root to any specific internal node, is present in the main string S equal to the number of children of that internal node. For Example, i is present in *mississippi* 4 times at positions $2, 5, 8, 11$, similarly *issi* is present twice at position $2, 5$. Therefore, every path from the root to any internal node is a repeat, which can be found in $O(L)$. There can be no more than $O(L)$ internal nodes or repeats, as there are L leaf nodes and every internal node has at least two children, therefore, it would not take more than $O(L^2)$ to find the all the repeats.

4.3 Problem Statement

In this section, we first present a motivational example to provide a sketch of the pipelined RVE. Then, in Section 4.3.2, we describe the problem formally.

```
A[1]=A[1]>>8
    =(A[1]+d[4]*1)>>8
    =(A[1]+d[3]*1+d[4]*1)>>8
    =(A[1]+d[2]*1+d[3]*1+d[4]*1)>>8
    =(A[1]+d[1]*1+d[2]*1+d[3]*1+d[4]*1)>>8
    =(0+d[1]*1+d[2]*1+d[3]*1+d[4]*1)>>8
    =(d[1]*1+d[2]*1+d[3]*1+d[4]*1)>>8
...
A[5]=(d[1]*5+d[2]*5+d[3]*5+d[4]*5)>>8
```

Figure 4.3: Expanded expressions after applying RVE on example in Figure 4.2

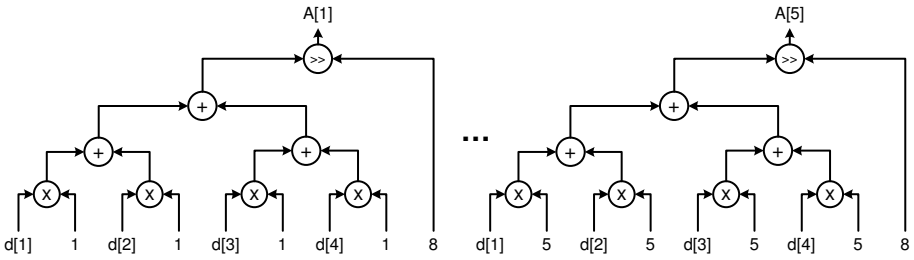


Figure 4.4: Circuits for Figure 4.3.

4.3.1 Motivational Example

We use the simple example shown in Figure 4.2 in the rest of this chapter to illustrate the RVE technique and to show how we can perform the computation in the example in a pipelined way. $d[1]$, $d[2]$, $d[3]$ and $d[4]$ are the four inputs and $A[1]$, $A[2]$, $\cdots$, $A[5]$ are the five outputs to the example in Figure 4.2. After applying RVE, we get the *expanded expressions* as shown in Figure 4.3. As all the loop carried dependences are removed, we can get all the outputs by computing their respective expanded expressions in parallel by using the recursive doubling algorithm for each output as shown in Figure 4.4. Computing like this gives a lot of parallelism, at the same time it requires a lot of area. This area can be reduced if all the circuits can be pipelined.

When a circuit is to be made from an expression, then the type and sequence of operators along with the type of the operands are important. Therefore, the expanded expressions in Figure 4.3 can be transformed to the *generic expressions* in Figure 4.5, by replacing the variables with their types. In Figure 4.5, i stands for an integer and c for a constant. The information in a generic expression is sufficient enough to infer the type and sequence of the operators along

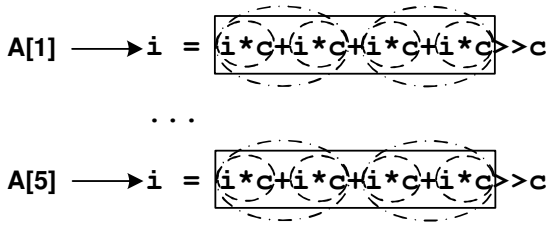
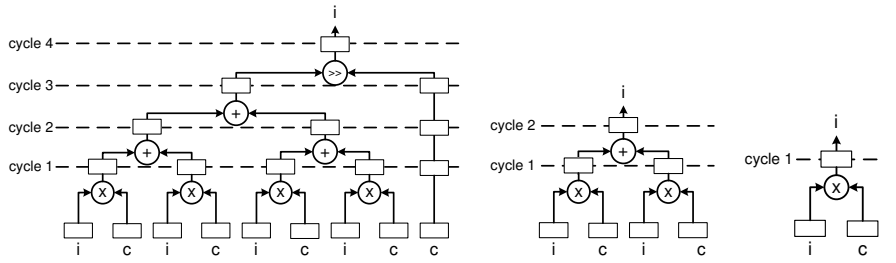


Figure 4.5: Generic Expressions



(a) Pipelined circuit for *(i*c+i*c+i*c+i*c)>>c* **(b)** Pipelined circuit for *i*c+i*c* **(c)** Pipelined circuit for *i*c*

Figure 4.6: Pipeline circuit for repeats in generic expression as given in Figure 4.5

with the type of the operands. Figure 4.5 shows that the generic expression (i.e. $(i*c+i*c+i*c+i*c) \gg c$) for all the outputs ($A[1], A[2], \cdots, A[5]$) is the same, which means that the sequence and type of the operators in a circuit of all outputs is the same. We can save area and compute all the outputs by just making a circuit for one output and using each stage of the circuit for different outputs. For that, we need to insert intermediate registers to store the intermediate values of all the stages in progress as shown in Figure 4.6a. However, if the memory or area constraints are not met for the chosen expression, then we can divide the expression into some smaller repeated equivalent sub-expressions such that when we make a circuit for any of those sub-expressions, it satisfies the area and memory constraints. This smaller sub-expression can be pipelined easily as it is small enough to satisfy the area and memory constraints and there are more than one such expressions, for which the corresponding data can be provided accordingly. For example in Figure 4.5, some repeats are: $i*c+i*c$ repeated 10 times and $i*c$ repeated 20 times. The corresponding pipelined circuits are shown in Figure 4.6b and Figure 4.6c. This means that the problem of enumerating the pipelining candidates for an expanded expression is equivalent to finding the repeated equivalent subexpressions or the *repeats* in

the corresponding generic expression. The chances of finding various repeats is very high in a RVE generic expression because it is generated from the loop body without the conditional statements which is doing some repetitive task, as shown in Figure 4.5.

Let E be a generic expression of length L. There can be many possible repeated sub-expressions $e \in \{e_{l_1}, e_{l_2}, \ldots, e_{l_j}\}$ with corresponding number of repeats $n \in \{n_{l_1}, n_{l_2}, \ldots, n_{l_j}\}$, where l_j is the length of the sub-expression e_{l_j} , $l_1 \leq l_2 \leq \ldots \leq l_j$ and $l_j \leq \frac{L}{2}$. The repeated sub-expression e in a generic expression E is defined as

$$E = (uev)^+ (xey)^+ \tag{4.1}$$

where $u, v, x, y \in \Sigma^*$. In other words, e is any non-overlapping sub-expression in E which is repeated at least twice.

4.3.2 Problem Statement

Following are the notations used to define the problem statement. Let

- E denotes the generic expression of length L.
- A_E is the estimated area on FPGA required by expression E.
- T_E is the time to transfer data for expression E from memory.
- T_C is the time to compute the expression E on FPGA F.
- A_F is the available area on the FPGA F.

Let $A_E > A_F$, which means that the expression E as a whole cannot be mapped on to the FPGA or $T_E > T_C$, which means that the data transfer for expression E cannot be hidden. Find such k non-trivial repeated expressions $e_{G_r} \in \{e_{l_1}, e_{l_2}, \ldots, e_{l_j}\}$ of length $l_{G_r} \in \{l_1, l_2, \ldots, l_j\} > 1$ for $1 \leq r \leq k$ and $k \geq 1$, which is repeated n_{G_r} time where $n_{G_r} \in \{n_{l_1}, n_{l_2}, \ldots, n_{l_j}\}$ such that

$$n_{G_r} l_{G_r} = \max_{1 \leq i \leq j} n_{l_i} l_i \tag{4.2}$$

for which $A_{e_{G_r}} \leq A_F$ and $T_{e_{G_r}} \leq T_{c_{G_r}}$. The condition $n_{G_r} l_{G_r} \leq L$ is always true. where

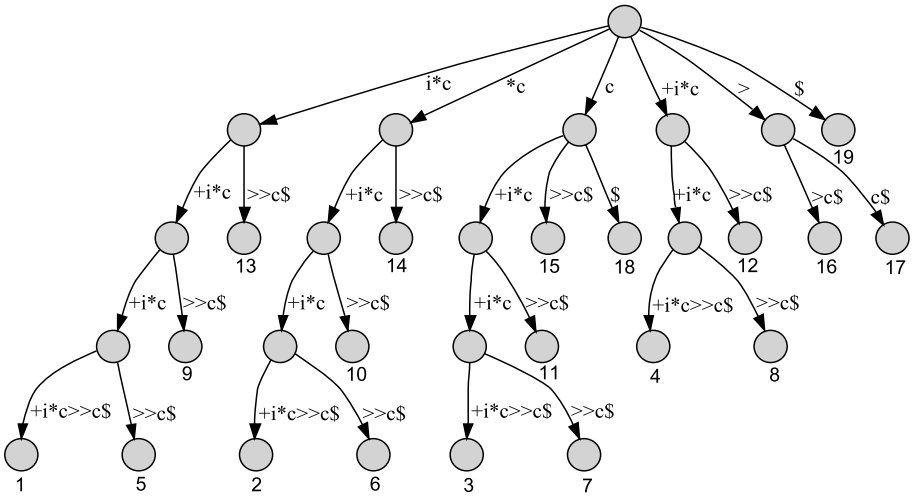


Figure 4.7: Suffix tree of *i*c+i*c+i*c+i*c>>c*

- $A_{e_{G_r}}$ is the area required by the expression e_{G_r} when mapped on to the FPGA F,
- $T_{e_{G_r}}$ is the time to transfer the data of expression e_{G_r} from the memory, and
- $T_{c_{G_r}}$ is the time to compute the expression e_{G_r} on the FPGA F.

Equation 4.2 means that we choose all the k repeated expressions whose product is the maximum among all the expressions, when the length of each expression is multiplied with the corresponding number of times it is repeated in E and they also meet the memory and the area constraints. The lengths of those expressions should be greater than 1 to make it non-trivial.

Finally we would choose the repeat e and call it an *optimal repeat*, which satisfies Equation 4.2 as well as the following equation.

$$e = \left\{ e_{G_m} \mid l_{G_m} = \max_{\forall e_{G_r}} l_{G_r} \right\} \tag{4.3}$$

Equation 4.3 means that we will choose the expression e_{G_m}, which has the maximum length among all e_{G_r} when $k > 1$.

4.4 Flexible Pipelining Design Algorithm

This section describes the flexible pipelining design algorithm. The five main steps in our algorithm are as follows:

4.4.1 Find possible candidates for pipelining

As mentioned in Section 4.3.1, finding all possible candidates for pipelining is equivalent to finding all repeats in a generic expression E. We use suffix tree for that purpose as described in Section 4.2.1.

By making a suffix tree for the generic expression E as shown in Figure 4.5, it gives us all the repeats along with their start positions in E. Some of the repeats in $(i*c+i*c+i*c+i*c) \gg c$ as shown by Figure 4.7 are $i*c+i*c+i*c$, $*c+i*c+i*c$, $+i*c+i*c$, $i*c+i*c$ and $*c+i*c$. Every repeat is a candidate for converting into a circuit for pipelining. The list of the repeats can be refined by removing non-valid repeats like $*c+i*c+i*c$, $+i*c+i*c$, $*c+i*c$ etc. To remove non-valid repeats, we fully parenthesize the generic expression E according to the priority of the operators to make it $((i*c)+(i*c)+(i*c)+(i*c) \gg c)$ and then build suffix tree from it. We filter only those repeats which are properly parenthesized and remove any substring after matching the closing parenthesizes. For example, non-trivial valid repeats in parenthesized E are $(i*c)$, $(i*c)+(i*c)$ and $(i*c)+(i*c)+(i*c)$. The non trivial non-overlapping valid repeats in parenthesized E are $(i*c)$ and $(i*c)+(i*c)$.

4.4.2 Select the optimal repeat from among the possible candidates.

Once the candidate repeats are shortlisted, we find the effective lengths l_e of the repeat e by removing all the parenthesizes and apply Equation 4.2 and Equation 4.3 to all of them to get the optimal repeat. In the example, the shortlisted candidates from the generic expression $(i*c+i*c+i*c+i*c) \gg c$ are $(i*c)$ and $(i*c)+(i*c)$ with effective lengths 3 and 7 and frequencies 4 and 2 respectively. Applying Equation 4.2 gives $\max(3\times 4, 7\times 2) = 7\times 2$, which selects $(i*c)+(i*c)$ considering it satisfies the memory and area constraints as the only option, which becomes the optimal repeat after applying Equation 4.3. We call our algorithm a flexible pipelining design algorithm as it chooses the best among many candidates with different area and memory requirements

and can adapt when the requirements are changed.

4.4.3 Feed data to pipeline

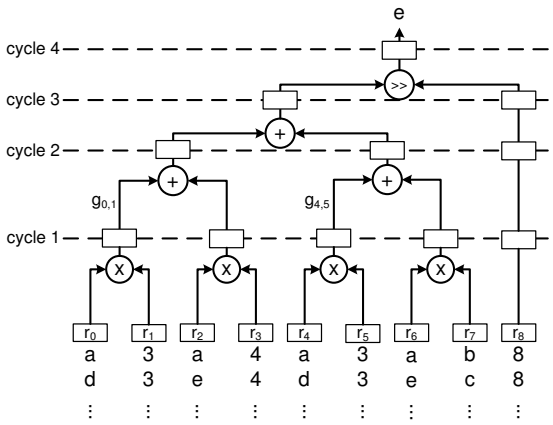
After selecting the optimal repeat for the pipelined circuit, we extract the data that needs to be fed to the registers after each cycle at the input of the pipelined circuit. We extract it by comparing the generic expression with the expanded expression in all the regions where e is repeated, as there is a one to one correspondence between the expanded expression and the corresponding generic expression as shown in Figure 4.8a. If l_e is the effective length of the optimal repeat e, then we define the number of operands in optimal repeat e as o_e, which excludes the number of operators from the effective length l_e.

4.4.4 Eliminate redundant expressions

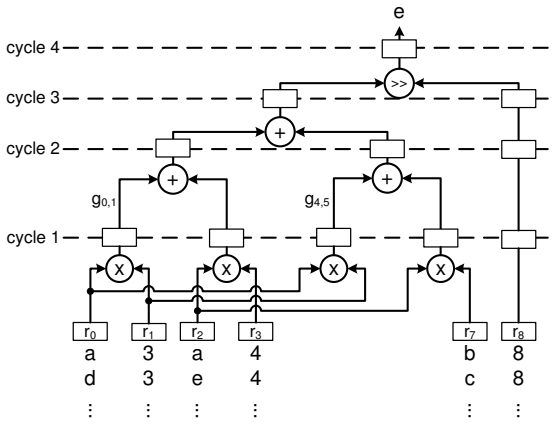
Since many variables and computations can be repeated in RVE, there are many redundant operands or operators that can be removed without sacrificing the speed. This, in turn, can reduce the area requirement. When o_e values that need to be fed are finalized after every cycle in the start of the pipeline. Then there is a chance to remove some redundant registers. As the data fed to them after each cycle is the same as fed to other registers. Let $r_1, r_2, \ldots, r_{o_e}$ be the positions of the o_e registers. Let n be the total number of cycles to compute the expression e. Let $d_1, d_2, \ldots, d_n$ and $f_1, f_2, \ldots, f_n$ be the data fed to the registers at position r_j and r_k, where $1 \leq j \leq o_e$ and $1 \leq k \leq o_e$ and $j \neq k$ in cycles $c_1, c_2, \ldots, c_n$. If $d_i = f_i$ for $1 \leq i \leq n$, then one of the registers can be removed. As both of the registers carry the identical data in respective cycles, and a link from one register can be fed to the operator, which initially takes the input from the other register. Let's assume that the optimal repeat is $e = (i * c + i * c + i * c + i * c) \gg c$, then Figure 4.8a shows the circuit for the expression e and also the data fed to each register. Since data in the registers r_4, r_5 and r_6 is the same as r_0, r_1 and r_2 respectively for every n cycles, we can easily remove registers r_4, r_5 and r_6 and feed their values from r_0, r_1 and r_2 respectively for every n cycles. This is shown in Figure 4.8b. Doing this will reduce the area as we have eliminated 3 registers in the given example. Now according to Figure 4.8b, e can be evaluated as

$$e = (r_0 * r_1 + r_2 * r_3 + r_0 * r_1 + r_2 * r_7) \gg r_8 \tag{4.4}$$

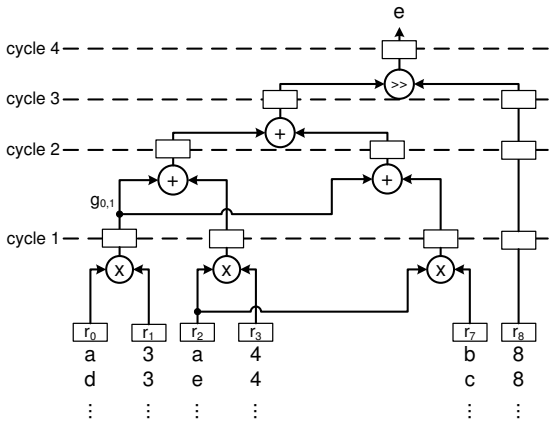
We can further try to reduce the area by removing some redundant operators



(a) before optimization



(b) after register optimization



(c) after operator optimization

Figure 4.8: Area optimization

```
for i=1 to 5
  A[i]=e^1+e^2>>8     (E)
end for
```

Figure 4.9: Computing kernel in Figure 4.2 using optimal repeat *i*c+i*c*

and the registers. In Equation 4.4, subexpression $r_0 * r_1$ is repeated once. We can remove the computation of the second occurrence of the subexpression $r_0 * r_1$ by providing the result from the first occurrence. This is done by feeding the data needed at the link $g_{4,5}$ from the link $g_{0,1}$ as shown in Figure 4.8c. This further reduces an operator and a register in the given example. We can again find such repeated sub-expression by using the suffix tree method as described in Section 4.4.1. The bigger the expanded expression is, more are the chances to find the redundant registers and the common operators. The chances of finding the registers with same values in each cycle and common operators is high because of the nature of RVE.

4.4.5 Convert optimal repeat to a pipeline circuit

Once the optimal repeat e is selected, it is converted to a deep pipelined circuit and mapped on to the FPGA. When e is evaluated, then the expression E can be computed serially as given by Equation 4.1 either on the GPP or the FPGA. In the given example, $E = (i * c + i * c + i * c + i * c) \gg c$ and let the optimal repeat be $e = i * c + i * c$, then e is computed for two different sets of inputs using the pipelined circuit of Figure 4.6b, and let the results are temporarily saved as e^1 and e^2. The example in Figure 4.2 is changed to Figure 4.9 and can be computed on the GPP or the FPGA by making a pipelined circuit for E, provided enough area is available.

After applying the pipelining to an expanded expression E, it is divided into very few serial computations as compared to the number of iterations of the loop body, which means extensive parallelism. Beside extensive parallelism, another advantage of finding the optimal repeat is the minimal memory accesses, as a large part of the expanded expression E is computed in a large pipelined circuit for e without saving the intermediate results in the memory.

If the kernel produces some number of output variables, when RVE is applied to those output variables, then it is recommended that the length of the generic expression for those output variables should be equal as in Figure 4.5, the length of the generic expression for all the 5 outputs is the same. This is

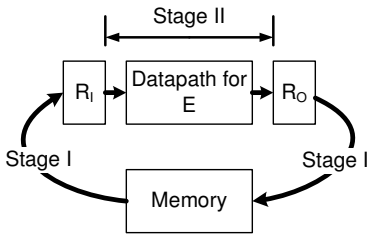


Figure 4.10: Architecture to balance datapath with memory access

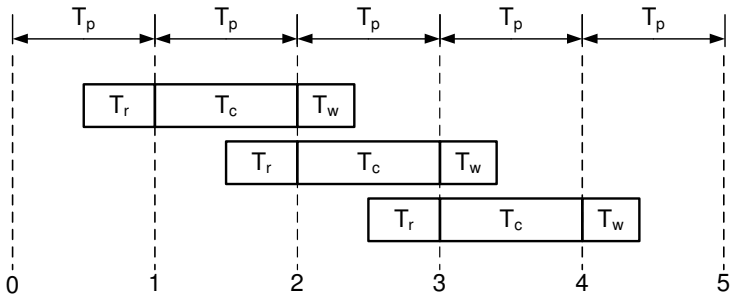


Figure 4.11: 2 stage pipelining, when $T_p = T_c \geq T_r + T_w$

a limitation for the current pipelining algorithm. However, there are many kernels from real life applications which satisfy this limitation like DCT, Finite Impulse Response (FIR) filter and matrix multiplication.

4.5 Balancing the Datapath and Memory Access Operations

Usually kernels are continually run in many applications, therefore, there is a need to balance the datapath with the memory access operation. We are considering the worst case scenario in which reads from and writes to memory are not possible at the same time, therefore it is recommended to divide the kernel computation as a two stage pipeline as shown in Figure 4.10. In the first stage, all the data computed earlier and saved in register set R_O is written to the on chip memory, then data for the next iteration of the kernel is read from the memory into a register set R_I for one run of the kernel. In the second stage, data is read from R_I and datapath operations are done and output is saved in the register set R_O as shown in Figure 4.10. As both the stages, read/write (Stage I) and computation (Stage II), use their own resources, we can pipeline them

```
for (i=0; i<8; i++) {
  for (j=0; j<8; j++) {
    s1=0; s2=0;
    for (k=0; k<8; k++) {
      s1+=(block[8*i+k])*(c1[j][k]);
      s2+=(block[8*i+k])*(c2[j][k]);
    }
    tmp1[8*i+j]=s1; tmp2[8*i+j]=s2;
  }
}
for (i=0; i<8; i++) {
  for (j=0; j<8; j++) {
    s1=0; s2=0;
    for (k=0; k<8; k++) {
      s1+=(c1[i][k])*tmp1[8*k+j];
      s2+=(c1[i][k])*tmp2[8*k+j]
          +(c2[i][k])*tmp1[8*k+j];
    }
    s2+=8388591;
    out[8*i+j]=((s2>>8)+s1)>>16;
  }
}
```

Figure 4.12: DCT code

as shown in Figure 4.11 and call it a memory-computation pipeline. Let the time for reading the memory, doing the datapath operations and writing back to the memory are T_r, T_c and T_w respectively. The latency of this pipeline is defined as $T_p = \max(T_r + T_w, T_c)$. The best is to choose the largest datapath for which $T_p = T_c \geq T_r + T_w$ provided it also fits the area on the FPGA. By doing this, the memory access latency is totally concealed and the datapath is computed efficiently in every stage by using the optimal resources. The pipelining approach in Section 4.4 is different from what is discussed here as the pipelining in Section 4.4 refers to pipelining in the datapath operations. Reading at the beginning and writing to the memory at the end of the kernel has two advantages. First, the total time to access the memory is minimized as all ports are used on every cycle. Second, the ordering of the accesses become irrelevant.

Table 4.1: Memory access time

Description	cycles
Time to read 8-bit 64 elements of 8×8 block	$\frac{8\times64}{64} \times 3 = 24$
Time to transfer 2 parameters through XREG	$2 \times 3 = 6$
Time to write 9-bit 64 elements	$\frac{9\times64}{64} \times 1 = 9$
Total memory access time for a kernel of DCT	39

4.6 Experiments and Results

In this section, we brief the experimental setup to implement the pipelined design algorithm and then discuss the results.

In order to facilitate the experiments, we have developed a program to automatically generate the pipelined design in VHDL. The resulting synthesizable VHDL code is used in the Xilinx simulator to obtain the results.

We use a Molen [117] prototype targeted on the Xilinx Virtex II pro platform XC2VP30 FPGA, which contains 13696 slices. The automatically generated code is simulated and synthesized on ModelSIM and Xilinx XST of ISE 8.2.022 respectively.

To evaluate and demonstrate the pipelining algorithm for RVE, we have used an integer version of the DCT as given in Figure 4.12, which satisfies all the constraints of the technique given in Sections 3.3 and 4.4. We automatically generate VHDL for two versions of RVE, one is with the area optimization and the other is without the area optimization. The results of the two versions of automatically optimized and different pipeline sizes for the DCT are compared to the hand optimized and pipelined DCT core[2] provided by Xilinx on the same platform. All the implementations take 8-bit input block element and output DCT of 9-bit. The port size to access on chip memory is 64 bits. It takes 3 cycles to read from the on-chip memory and store it in register set R_I, whereas it takes 1 cycle to write from R_O to the on-chip memory. The total memory access time to transfer the data for a block of DCT is 39 cycles as given in Table 4.1.

A kernel of DCT outputs 64 elements whose generic expressions are the same. We find the best candidate for the optimal repeat e for the DCT, which comes out to be the generic expression of one element, therefore, we refer to it as *full element* in the experiment. This is the largest repeat which satisfies the

[2]https://secure.xilinx.com/webreg/clickthrough.do?cid=55758

Table 4.2: Comparison of automatically optimized DCT with Xilinx's hand optimized DCT core

	Frequency	latency	Computation time for a block of 8×8 (cycles)		Slices
	(MHz)	(cycles)	(cycles)	(ns)	
Xilinx DCT core	171.223	92	64	373.8	1213
DCT full element	121.479	13	64	526.9	9215
DCT $\frac{1}{3}$ element	265.354	8	192	723.6	2031
DCT full element opt.	120.943	13	64	529.2	4939
DCT $\frac{1}{3}$ element opt.	265.354	8	192	723.6	1820

area and memory requirements for XC2VP30 FPGA, therefore, it is chosen as the optimal repeat. However, if there is less area available on FPGA, the next largest repeat is one third of the generic expression of the one element, therefore we refer to it as $\frac{1}{3}$ *element*. When area optimization is applied, we refer to them as *full element opt.* and $\frac{1}{3}$ *element opt.,* respectively.

Table 4.2 shows the results for different implementations of the DCT after synthesis. The Xilinx DCT core is hand optimized by knowing the properties of a 2D DCT. A 1D DCT is only implemented with buffering and taking the transpose of the 8×8 block. Initially, 1D DCT is computed from the inputs, then the output is transposed and fed back to the same 1D DCT circuit to produce the 2D DCT. The generated code is very small and well pipelined, therefore, it has very few slices and high frequency. However the initial latency is high due to transposition and computing again the 1D DCT. Once the initial latency is spent, the circuit produces an entire DCT block every 64 cycles.

Our automatic optimization does not take advantage of the knowledge of the properties of the 2D DCT. It takes the unoptimized code of the 2D DCT, follows some generic steps to apply the RVE and then designs a flexible deep pipeline as discussed in Section 4.4, trying to satisfy the area and memory constraints. The code generated for *DCT full element* is very large as compared to the hand optimized, therefore it has low frequency. However, it extracts lots of parallelism and utilizes the resources to its capacity and produces an output of the DCT block every 64 cycles with a lower initial latency of 13 cycles, which is basically the depth of the pipeline. The code for the DCT one-third is relatively small but still larger than the Xilinx DCT core. It produces better frequency than Xilinx core at the cost of 3 times more cycles and lower initial latency of 8 cycles to compute one DCT block. The time to compute DCT using one third element is increased by 37% with a 78% decrease in area as compared to computing with the full element.

When area optimization is applied to both pipelining candidates, then the area is reduced by 46.4% for the *full element* and by 10.4% for the $\frac{1}{3}$ *element.* The reduction in area after the area optimization for the *full element* is larger than $\frac{1}{3}$ *element*, which shows that there is more potential for reducing area in larger pipelines.

The results show that our pipelining design algorithm for RVE, which applies on some limited type of problems, gives a comparable performance at the cost of extra hardware than the hand optimized code. Although, it is not better than the hand optimized in performance, the main benefits of our approach is automated design, optimization, and hardware generation of kernels starting from

a program code. The design time and quality of the optimization for the hand optimized is quite variable depending on the human intelligence. Whereas, the design time and quality of the optimization in our technique is quite deterministic. Secondly, this automated approach can be used to automatically generate high performance code for other kernels which satisfy the given constraints for which the hand optimized codes are not generated and saves a lot of design time.

4.7 Summary and Conclusion

In this chapter, we have presented a pipelining algorithm for RVE, which automatically generates an extensively parallel and pipelined VHDL code for a certain class of problems, which can compare in performance with hand optimized codes. Although the algorithm produces better performance for large area FPGA, still it can be used to get good performance for reasonably small FPGA. Our algorithm is a good choice for kernels, for which hand optimized codes are not available, area is not major concern and high performance is the requirement in short design time.

In the next chapter, we will look at a class of dynamic programming problems, which produces exponential number of terms when RVE is applied on it naively. We will describe a better approach to tackle such problems.

Note.

The content of this chapter is based on the the following paper:

Z. Nawaz, T. Marconi, T. P. Stefanov, K.L.M. Bertels, *Flexible Pipelining Design for Recursive Variable Expansion*, International Parallel and Distributed Processing Symposium, pp. 915-922, Rome, Italy, May 2009.

5

RVE for Dynamic Programming Problems

DYNAMIC programming (DP) is a powerful method, which is typically used to compute a large number of discrete optimization problems in various fields. Examples of DP problems[1] are Knapsack problem, Traveling salesman problem, Smith-Waterman, shortest paths, Viterbi algorithm and Planner's problem. Beside the optimization problems, it is also used in other problems as computing Fibonacci numbers and Binomial coefficients.

Optimization problems are usually very important problems and take considerable amount of time to compute. There is always a need to solve them quickly, possibly using parallel computation. Therefore such problems are good candidates for hardware acceleration.

Earlier in Chapter 3, we saw that when RVE was applied to the Smith-Waterman problem (DP problem), it expanded exponentially. In this chapter, we show that our technique can be applied to a large class of DP problem, which have a constant number of dependences for which the distance vector between the current and constituent subproblem or subproblems is also constant. Generally, each element in dynamic programming problems is computed using dataflow on FPGA. Whereas, we present a hybrid approach to accelerate such problems. We use RVE to compute the blocks of elements in parallel and use dataflow between the blocks. The contribution of this chapter is as follows:

- We have devised the formulation of a generic framework as well as two variants of the RVE algorithm, named RVE with no pre-computation (RVENP) and RVE with pre-computation (RVEP). When applied to various Dynamic Programming problems, we demonstrate that they outper-

[1]The term 'dynamic programming problems' refer to problems that can be formulated in dynamic programming [120].

form any known technique.

Both RVENP and RVEP can generate highly parallel, pipelined hardware accelerators for DP problems using Reconfigurable systems. Later, we will discuss under what conditions the two versions gives better performance than the other. As these algorithms expose more parallelism, it is possible to achieve more acceleration than any other parallel technique at the cost of extra area on FPGA. These algorithms are especially suitable for cases where high performance is a priority and extra area can be used to achieve this.

This chapter describes a methodology for the application of our algorithm on 4 representative problems to show its speedup and area overhead as compared to a dataflow implementation. Later a generalized version of DP problems is defined that shows that the devised steps are generic enough to tackle a large class of DP problems.

The chapter is organized as follows. In the next section, we present the related work. In Section 5.2, we briefly introduce the four representative problems. Then in Section 5.3, we describe some generic steps and apply them on each of the problems to clarify its effects. In Section 5.4, we have compared the hardware acceleration by the both RVE implementations with dataflow. Section 5.5 describes how these steps are generalized to be applicable on a large number of DP problems. Finally, Section 5.6 summarizes the chapter.

5.1 Related Work

Our acceleration is based on RVE, which is similar to techniques like back substitution [69], look ahead computation [63, 73] and block back-substitution [102]. The first known implementation of look ahead in DP problems was done in [36], where it was applied to the Viterbi algorithm and showed its potential for DP problems. It was shown that the add-compare-select (ACS) operation, which is nonlinear in nature, is difficult to parallelize. Later, this work was extended to DP problems [94, 95], again showing only the Viterbi algorithm example and without much discussing the general properties of DP. Fettweis [37, 38] also presented the algebraic properties for identifying the class of problems on which look ahead computation can be applied. As resources were very limited at that time, parallelism exposed was less. No hint was given to tackle conditional statements in generic DP formulations.

Systolic arrays have been the choice of many researchers for mapping dynamic programming problems on to a VLSI chip. Li and Wah [119] have classified

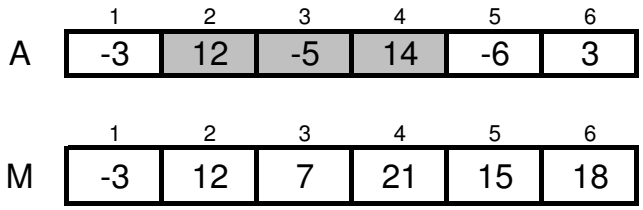


Figure 5.1: MCSS problem

DP problems according to the functional equations. They have also proposed systolic arrays for each type. Few have devised two-dimensional systolic arrays for the dynamic programming problems [28, 47, 80]. Others have proposed a linear systolic arrays to map dynamic programming problems [87,97]. In this chapter, we compare our implementation with the linear systolic array implementation of the four representative dynamic programming problems.

5.2 Representative Problems

In this section, we describe some problems that are representative for a broad range of DP problems. Here Maximum Contiguous Subsequence, Needleman-Wunsch and Longest Common Subsequence are optimization problems.

5.2.1 Maximum Contiguous Subsequence Sum (MCSS) Problem

Given an array of n real numbers $a_1, a_2, \cdots, a_n$. We want to find i and j such that $\sum_{k=i}^{j} a_k$ is maximized and $1 \leq i \leq j \leq n$ [20]. The solution to this problem is trivial, if all the numbers are non-negative. However, it becomes interesting when there are some negative numbers as well. Practically this problem can be seen as finding the maximum number of people in a room over a time where we get some numbers representing the people entering or leaving the room over that time. Let $M[j]$ defines the max sum over all the windows ending at j, then $M[j]$ can be computed using the following recurrence equation.

$$M[j] = \max \begin{cases} M[j-1] + A[j] \\ A[j] \end{cases} \tag{5.1}$$

A vector M is filled starting from left side using the Equation 5.1. Figure 5.1

		G	A	C	G	G	A
	0	-2	-4	-6	-8	-10	-12
G	-2	**1**	-1	-3	-5	-7	-9
A	-4	-1	**2**	0	-2	-4	-6
T	-6	-3	**0**	1	-1	-3	-5
C	-8	-5	-2	**1**	0	-2	-4
G	-10	-7	-4	-1	**2**	1	-1
G	-12	-9	-6	-3	0	**3**	1
A	-14	-11	-8	-5	-2	1	**4**

Figure 5.2: Matrix, for NW $g = -2$ and $x[i,j] = 1$ when $S[i] = T[j]$ otherwise -1. Elements in bold show the traceback.

shows an example which computes vector M from vector A and MCSS is 21, where $i = 2$ and $j = 4$. Using dynamic programming, it is solved in $O(n)$.

5.2.2 Fibonacci Numbers

The Fibonacci numbers $N[i]$ is given by the following equation.

$$N[i] = N[i-1] + N[i-2] \tag{5.2}$$

where $N[0] = 0$ and $N[1] = 1$

5.2.3 Needleman-Wunsch (NW) Algorithm

Needleman-Wunsch (NW) is a global alignment algorithm for the two biological sequences [91]. The optimal alignment score $F[i,j]$ for two sub-sequences $S[1..i]$ and $T[1..j]$ is given by the following recurrence equation:

$$F[i,j] = \max \begin{cases} F[i,j-1] + g \\ F[i-1,j-1] + x[i,j] \\ F[i-1,j] + g \end{cases} \tag{5.3}$$

where $F[0,0] = 0, F[0,j] = g \times j$ and $F[i,0] = g \times i$, for $1 \leq i \leq n, 1 \leq j \leq m$, n and m are lengths of S and T respectively. The $x[i,j]$ is the score for match/mismatch, depending upon whether $S[i] = T[j]$ or $S[i] \neq T[j]$. The

		a	b	a	**c**	**d**	a	c	**c**
	0	0	0	0	0	0	0	0	0
c	**0**	0	0	0	1	1	1	1	1
a	0	**1**	1	1	1	1	2	2	2
d	0	**1**	**1**	**1**	1	2	2	2	2
c	0	1	1	1	**2**	2	2	3	3
d	0	1	1	1	**2**	3	3	3	3
d	0	1	1	1	2	**3**	**3**	**3**	3
c	0	1	1	1	2	3	3	4	**4**

Figure 5.3: LCS

g is some constant penalty for inserting a gap in any sequence. An example of NW algorithm is shown in Figure 5.2, where a matrix is generated and the two sequences are placed along the row and the column. First, the top row and left column are filled with boundary conditions. Then the rest of the matrix is filled using Equation 5.3 starting from the top-left corner. The elements are filled from left to right and from top to bottom. After filling the whole matrix, a traceback to find the optimal solution is started from the bottom right corner. For most of the DP problems, the traceback is done in a similar way. The global alignment as a result of the traceback shown in Figure 5.2, is

```
GA-CGGA
|||||||
GATCGGA
```

5.2.4 Longest Common Subsequence (LCS) Problem

Given a string of characters, if some of the characters are deleted from that string, then the resulting string is called a *subsequence*. For example, $Z = \langle a, d, c \rangle$ is a subsequence of $X = \langle a, b, a, c, d, a, c, d \rangle$. Given two sequences X and Y, we say that Z is a *common subsequence* of X and Y, if Z is a subsequence common to both X and Y. LCS is defined as the longest of all the possible subsequences of X and Y [114]. The Unix *diff* command works by finding the LCS of two files, where each line is treated as a character. Let $c[i, j]$ is the length of the LCS for sequences X_i and Y_j, then its formulation for $i, j > 0$ is given by:

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j, \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } x_i \neq y_j. \end{cases} \tag{5.4}$$

where $c[i,j] = 0$ for $i = 0$ or $j = 0$. An example of LCS is shown in Figure 5.3. Similar to NW, a matrix is filled by using Equation 5.4 and then traceback is started from bottom right corner of the matrix. The LCS as we get from Figure 5.3 is $\langle a, c, d, c\rangle$. The condition in the recursive formulation along with finding the maximum make it different from the previous three examples. Another very well known problem which has a similar structure is Knapsack problem.

5.3 Generic RVE Algorithm for DP Problems

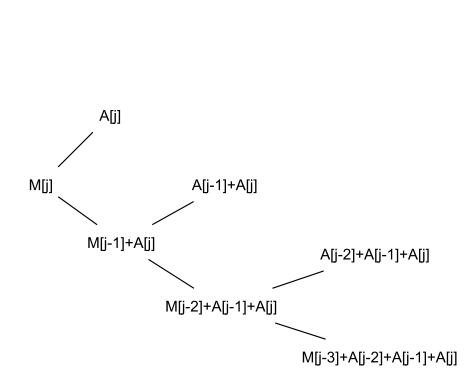
In this section, we describe the generic steps that can largely increase the parallelism, consisting of applying a modified version of RVE and subsequently a dataflow based transformation. First we expand the equation by partially applying RVE on the DP recurrence equation and remove the redundant sub-equations. The remaining sub-equations are grouped together. As will be discussed below, certain terms can be precomputed. If we exploit that possibility then we obtain the RVE with pre-computation version, denoted as RVEP. If not exploited, then it is called RVE without pre-computation, denoted as RVENP. Finally, all the unknown variables in a block are computed in parallel and subsequent blocks are computed in a dataflow manner. We first illustrate the approach using the four representative problems described above.

5.3.1 Step 1: Apply RVE

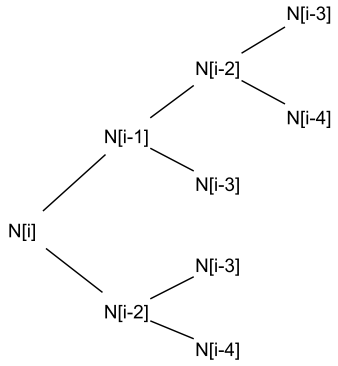
We partially apply RVE on all the representative problems in Section 5.2 and we get the recursion trees shown in Figure 5.4. The edge labels in Figure 5.4d define the condition as A defines $x_i = y_j$, B defines $x_i = y_{j-1}$, C defines $x_{i-1} = y_{j-1}$, D defines $x_{i-1} = y_j$ and A', B', C', D' are the complement of A, B, C, D respectively.

5.3.2 Step 2: Remove redundant sub-equations

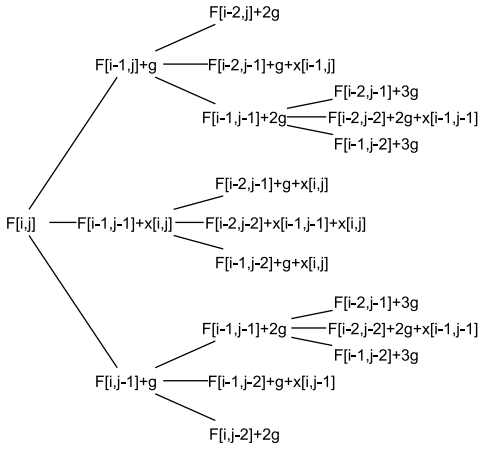
Once the expression is expanded, we identify some redundant nodes in the recursion trees in case of optimization problems as they can be removed without



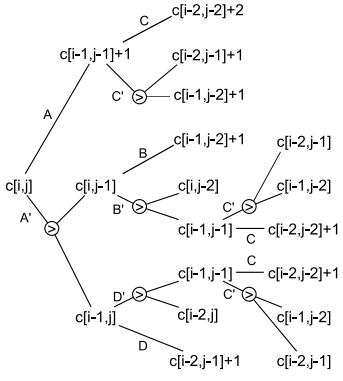
(a) Maximum Contiguous Subsequence Sum



(b) Fibonacci number



(c) NW



(d) LCS

Figure 5.4: Partially RVE expanded recursion trees

affecting the solution.

In case of MCSS, there is no redundant node, therefore, nothing can be removed and we get the following equation.

$$M[j] = \max \begin{cases} a & M[j-3] + A[j-2] + A[j-1] + A[j] \\ b & A[j-2] + A[j-1] + A[j] \\ c & A[j-1] + A[j] \\ d & A[j] \end{cases} \tag{5.5}$$

Fibonacci number is not an optimization problem, so nothing can be removed.

In case of NW, there are some leaf nodes which are redundant, therefore we can reduce the number of sub-equations in the *max* equation by removing redundant nodes. For NW, the reduced equation after removing the redundant nodes is the following.

$$F[i,j]=\max \begin{cases} i & F[i,j-2]+2g \\ ii & F[i-1,j-2]+g+x[i,j-1] \\ iii & F[i-1,j-2]+3g \\ iv & F[i-1,j-2]+g+x[i,j] \\ v & F[i-2,j-2]+2g+x[i-1,j-1] \\ vi & F[i-2,j-2]+x[i-1,j-1]+x[i,j] \\ vii & F[i-2,j-1]+3g \\ viii & F[i-2,j-1]+g+x[i,j] \\ ix & F[i-2,j-1]+g+x[i-1,j] \\ x & F[i-2,j]+2g \end{cases} \tag{5.6}$$

The 13 leaf nodes in Figure 5.4c are reduced to 10 sub-equations in Equation 5.6.

If conditional statements are part of the recurrence equation of a DP problem, then it is not obvious to remove the redundancies as shown in Figure 5.4d. Here, the conditional statements are mixed with *max* statements. The non-associative nature of conditional statements make it difficult to get benefit after applying RVE. However we can still remove redundant nodes after making a small algebraic transformation. It is algebraically correct to take the maximum value of all the unique nodes and any statement will be only effective when its accompanying conditional statement is also true, otherwise it will be 0. This is

similar to predicated execution, which was implemented in Cydra 5 computer [98]. The $c[i,j]$ after RVE expansion can thus be written as following.

$$c[i,j] \quad = \quad \begin{matrix} c[i-2,j-2]+2 & A1 \\ \max \begin{cases} c[i-1,j-2]+1 & A2 \\ c[i-2,j-1]+1 & A3 \\ c[i,j-2] & A4 \\ c[i-2,j-2]+1 & A5 \\ c[i-1,j-2] & A6 \\ c[i-2,j-1] & A6 \\ c[i-2,j] & A7 \end{cases} \end{matrix} \tag{5.7}$$

where $A1 = A \wedge C$, $A2 = (A \wedge C') \vee (A' \wedge B)$, $A3 = (A \wedge C') \vee (A' \wedge D)$, $A4 = (A' \wedge B')$, $A5 = (A' \wedge B' \wedge C) \vee (A' \wedge D' \wedge C)$, $A6 = (A' \wedge B' \wedge C') \vee (A' \wedge D' \wedge C')$ and $A7 = A' \wedge D'$. Here $A \wedge C$ means A AND C, $A \vee C$ means A OR C. The Equation 5.7 has only 8 sub-equations as compared to 13 leaf nodes in Figure 5.4d.

5.3.3 Step 3: Group sub-equations

After having removed the redundant sub-equations, we can group and simplify the remaining components of the equation.

Equation 5.5 for MCSS is already grouped and cannot be simplified any further.

The Fibonacci number as given in Figure 5.4b is simplified and grouped as following

$$N[i] = 3N[i-3] + 2N[i-4] \tag{5.8}$$

In NW, Equation 5.6 can be rearranged and simplified to the follows:

$$F[i,j] = \max \begin{cases} i & (F[i,j-2] \succ F[i-2,j]) + 2g \\ ii & F[i-1,j-2] + C_1 \\ iii & F[i-2,j-2] + C_2 \\ iv & F[i-2,j-1] + C_3 \\ v & 0 \end{cases} \tag{5.9}$$

where $C_1 = ((g + (x[i, j-1] \succ x[i, j])) \succ 3g)$, $C_2 = ((2g + x[i-1, j-1]) \succ (x[i-1, j-1] + x[i, j])) = (2g \succ x[i, j]) + x[i-1, j-1]$ and $C_3 = (3g \succ (g + (x[i, j] \succ x[i-1, j])))$ for Equation 5.6. Here $\succ$ is defined as the max operator.

In LCS, Equation 5.7 can be rearranged and simplified to the following.

$$c[i,j] = \max \begin{cases} c[i-2, j-2] + 2 & A1 \\ \begin{cases} c[i-1, j-2] + C_1' & A2 \vee A6 \\ c[i-2, j-1] + C_2' & A3 \vee A6 \end{cases} \\ c[i, j-2] & A4 \\ c[i-2, j-2] + 1 & A5 \\ c[i-2, j] & A7 \end{cases} \tag{5.10}$$

where $C_1' = \max \begin{cases} 1 & \text{if } A2 \\ 0 & \text{if } A6 \end{cases}$ and $C_2' = \max \begin{cases} 1 & \text{if } A3 \\ 0 & \text{if } A6 \end{cases}$.

It is possible that $A2$ and $A6$ are true at the same time. Similarly for $A3$ and $A6$.

5.3.4 Step 4: Precompute cost function

Precomputation for an iteration means that part of the computation for the current iteration can be done in the previous iteration, because all variables are known. This can reduce the critical path and increase parallelism with little increase in area on an FPGA.

In Equation 5.5 of MCS, the contents of sub-equation b,c and d are known from the start. Therefore they can be pre-computed for next j^{th} iteration denoted by j' while $M[j]$ is computed.

Pre-computation cannot be done for Fibonacci numbers as all its contents are dynamic and nothing is known from the start in Equation 5.8.

In NW, the contents of C_1, C_2 and C_3 in Equation 5.9 are known in advance and can be precomputed. While $F[i, j]$ is being computed C_1, C_2 and C_3 for next (i, j) iteration defined as (i', j') can be computed in parallel as shown in Figure 5.15b.

Similarly, in LCS, the contents of C_1' and C_2' in Equation 5.10 are known in advance, therefore, C_1',C_2', $A4$ and $A7$ for the next (i, j) iteration defined as

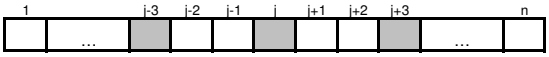


Figure 5.5: MCSS vector

Figure 5.6: Fibonacci vector

(i', j') can be computed in parallel with the computation of $c[i, j]$ for the current i, j values. In other words, C_1', C_2', $A4$ and $A7$ to be used in next iteration (i', j') are computed in current iteration (i, j). The circuit for LCS as given by Equation 5.10 is shown in Figure 5.15c. C_1' and C_2' are further reduced to $C^*[i, j]$ in Figure 5.15c.

When the pre-computation is applied, then we speak of RVEP (RVE with pre-computation) and in case it is omitted we speak of RVENP (RVE with no precomputation). The choice of choosing between them is left to the user.

5.3.5 Step 5: Fill the block and mix with dataflow

Next, we look at the elements which can be filled in parallel and also figure out the elements whose computation can be avoided without affecting the solution of the problem, as the computation of the output can be arranged without them too.

Equation 5.5 of MCSS can be mapped to a vector as shown in Figure 5.5, where the j^{th} index of vector M is computed from $j - 3^{rd}$ index. Similarly, $j - 1^{st}$ and $j - 2^{nd}$ indices can be computed from $j - 3^{rd}$ index in parallel to the computation of j^{th} index, as there is no dependency among them. Later the $j + 1^{st}$, $j + 2^{nd}$ and $j + 3^{rd}$ indices are computed in parallel from j^{th} index, which are done serially after j^{th} index computation. A one dimensional systolic array is used to compute the three unknowns as shown in Figure 5.7.

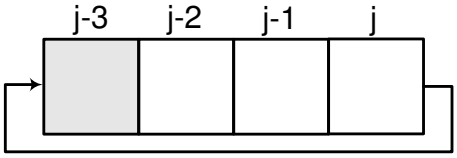


Figure 5.7: Systolic array for MCSS

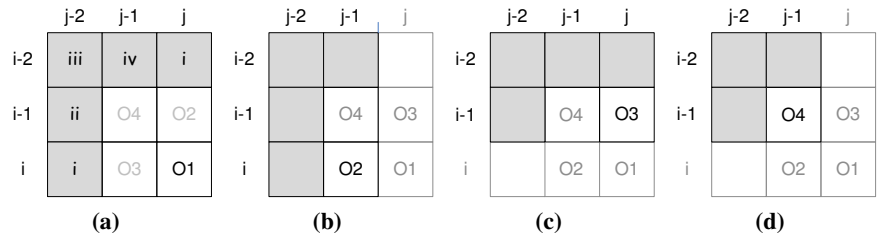


Figure 5.8: NW matrix for $B = 2$ that shows the elements from which $F(i-i^{'}, j-j^{'})$ are computed. The shaded square represents already known values.

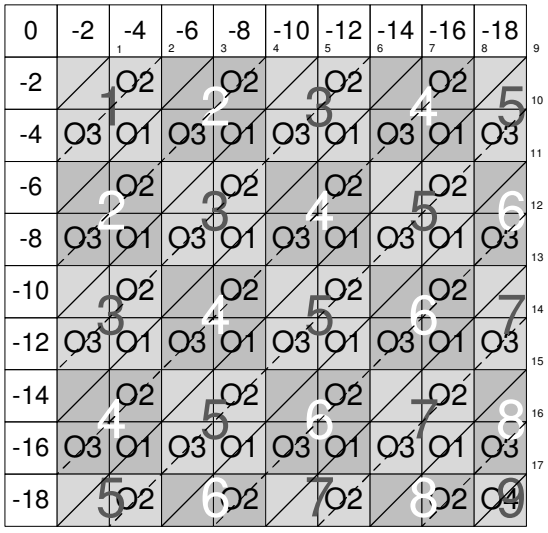


Figure 5.9: Sequence of fill of the $F(i, j)$ scoring matrix of Equation 5.9, starting from the top left light shaded square numbered 1 (represent the time instance to compute) and moving diagonally down as shown by trailing numbers. All the squares with the same number can be executed in parallel. Antidiagonal lines show the dataflow.

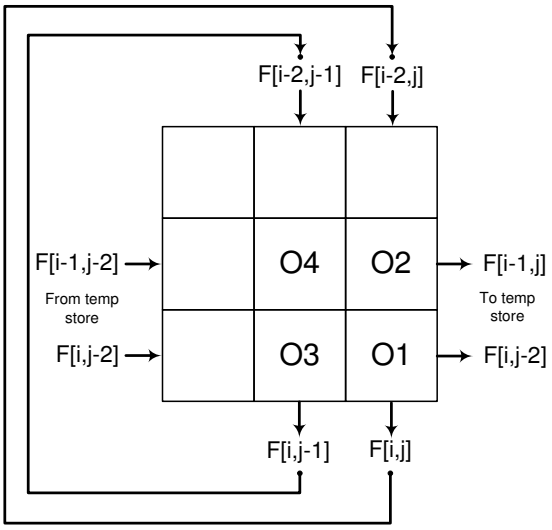
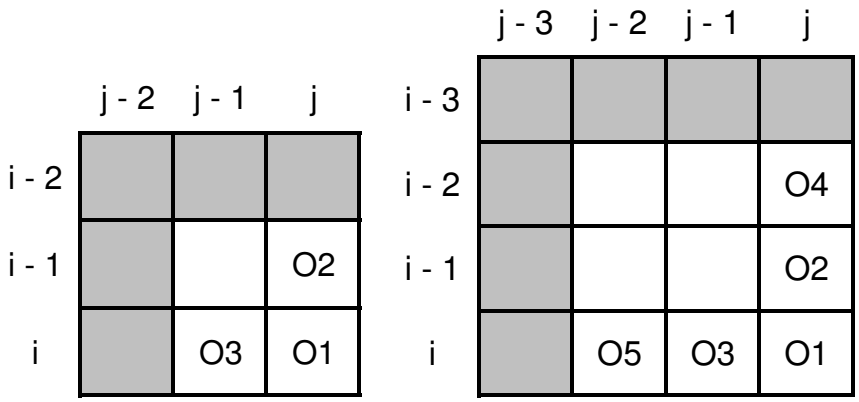


Figure 5.10: Systolic array

Similarly, Equation 5.8 of Fibonacci number can be mapped to the vector shown in Figure 5.6. The i^{th} index of vector N is computed from $i-3^{rd}$ and $i-4^{th}$ indices. Likewise, $i-1^{st}$ and $i-2^{nd}$ indices can be computed from $i-3^{rd}$ and $i-4^{th}$ indices in parallel to the computation of the i^{th} index. A one dimensional systolic array similar to Figure 5.7 is used to compute Fibonacci numbers.

In NW, Equation 5.9 when mapped on to tabular form gives us a 3×3 matrix, where the terms to be computed (*O1* to *O4*) are represented by a 2×2 block as shown in Figure 5.8. Figure 5.8a shows how $F(i,j)$ (i.e. *O1*) is calculated from Equation 5.9. Similarly we can compute $F(i,j-1)$ (i.e. *O2* in Figure 5.8b), $F(i-1,j)$ (i.e. *O3* in Figure 5.8c) and $F(i-1,j-1)$ (i.e. *O4* in Figure 5.8d) using the same steps as applied for *O1*. All the unknown variables in a block can be computed in parallel, as there are no dependences among them. The whole matrix can be filled as shown in Figure 5.9, which is like dataflow at block level. We define the size of the unknown block as the *blocking factor* B, which is set to 2. In the example discussed here, we assume that the matrix size is the multiple of blocking factor for simplification. Otherwise, we can always find the next smaller matrix which satisfies this condition, and continue with our method. This structure of matrix filling motivates us to use a systolic array as shown in Figure 5.10. This shows that when a block is computed, the outputs taken out in the vertical direction are fed back as input for the same block circuit to compute the next iteration in the fill. The horizontal data is



(a) for b=2 **(b)** for b=3

Figure 5.11: matrix to show elements to be found

stored in some temporary storage like BRAM, which is used in some later iteration. Similar to NW, LCS matrix can also be filled.

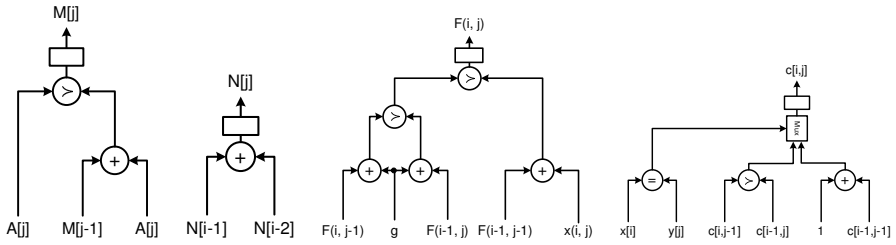
In most of the DP problems, we can avoid filling the whole matrix and still can obtain the optimal solution as is the case with NW and LCS. Figure 5.8 shows how some elements depend upon elements which are already computed. Since we know that the maximum value is always at the bottom right corner, *O1* will contain the maximum value. *O1* is chosen from any of five boundary element shaded as gray in Figure 5.8a, therefore the traceback will lead to that element. Similarly, *O2* and *O3* can also be traced back to any of the boundary elements shaded in gray in Figure 5.8b and 5.8c respectively. The optimal solution is obtained while tracing back like this. There is no need to find the elements inside the block as shown in Figure 5.11a, since the solution can be obtained even without computing it. This saving can be increased from $\frac{1}{4}$ to $\frac{4}{9}$, when we increase the blocking factor from 2 to 3 as shown in Figure 5.11b. The example given in Figure 5.12 shows the traceback without completely filling the matrix with blocking factor $B = 2$, thus avoiding area which is otherwise used to compute the element inside the block.

Similarly for computing the Fibonacci numbers shown in Figure 5.6, we only compute the gray shaded elements, as they suffice to generate the solution.

We cannot avoid filling some elements in case of MCSS, as according to algorithm, the traceback starts from the maximum value in the vector, which can be located anywhere in the matrix. Therefore, we have to completely fill the vector.

		G	A	C	G	G	A
	0	-2	-4	-6	-8	-10	-12
G	-2		-1		-5		-9
A	-4	-1	**2**	0	-2	-4	-6
T	-6		**0**		-1		-5
C	-8	-5	-2	**1**	0	-2	-4
G	-10		-4		**2**		-1
G	-12	-9	-6	-3	0	**3**	1
A	-14		-8		-2		**4**

Figure 5.12: Example showing the traceback for NW Algorithm after RVE is applied with b=2



(a) MCSS **(b)** Fibonacci number **(c)** NW **(d)** LCS

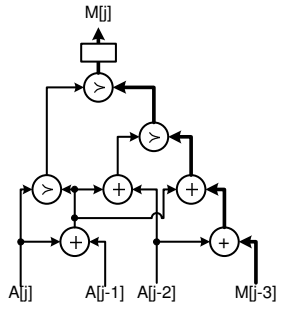
Figure 5.13: Circuits for each element in dataflow

This reduction technique can be used for most of the DP problems, as in most of them, the traceback starts from the bottom right corner in a matrix or from the right end in case of the vector. Therefore, in cases where the maximum is at the bottom right corner, the user can switch to this reduction.

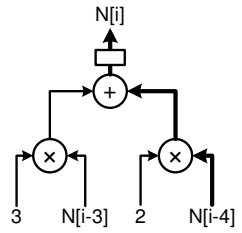
5.4 Performance Evaluation

This section presents the implementation details for the four representative DP problems and then presents the performance comparisons for the dataflow only, RVENP and RVE versions..

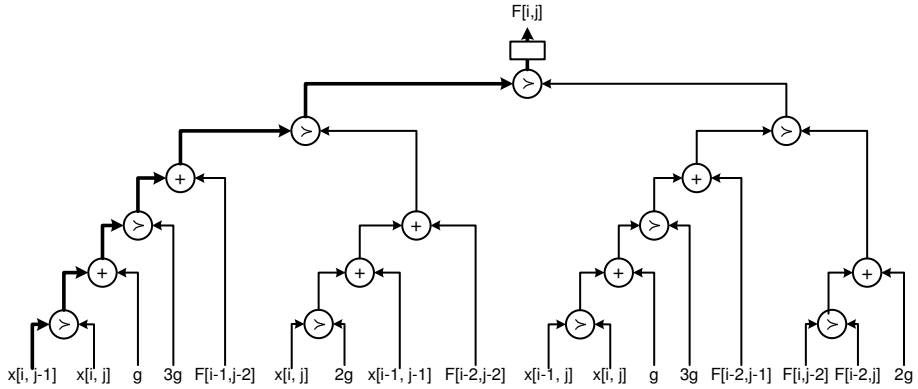
The circuit diagrams for implementing one element using dataflow for all the four problems using Equation 5.1, 5.2, 5.3 and 5.4 are shown in Figure 5.13. Similarly, the circuit diagrams for implementing one element using RVENP



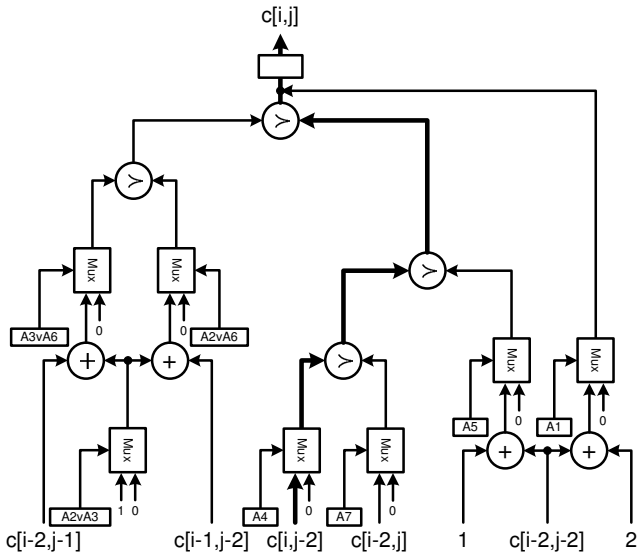
(a) MCSS as given by Equation 5.5



(b) Fibonacci as given by Equation 5.8

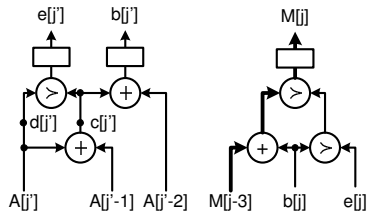


(c) NW as given by Equation 5.9

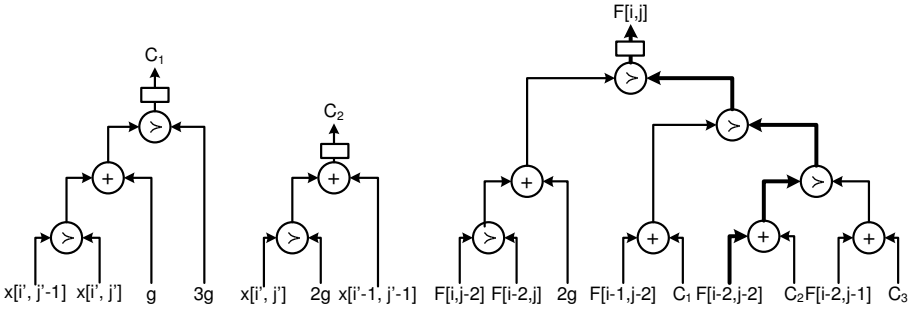


(d) LCS as given by Equation 5.10

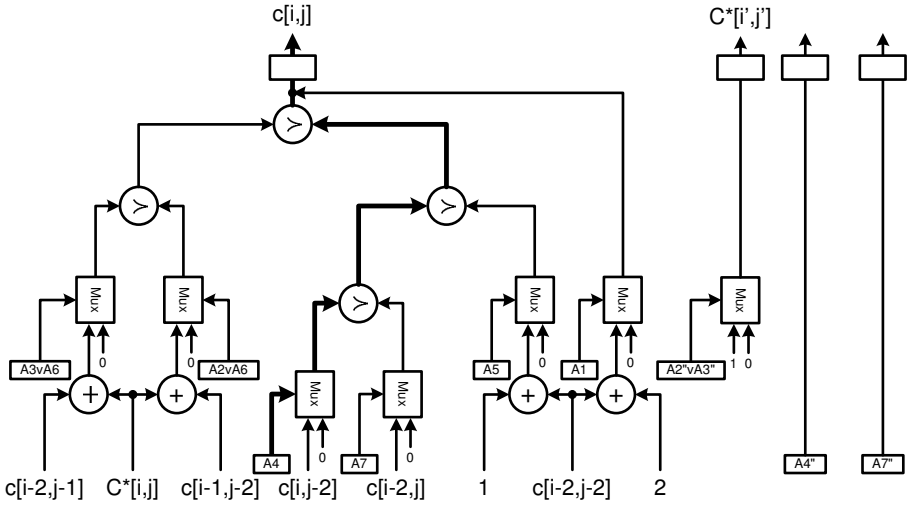
Figure 5.14: Circuits for each element in RVENP, bold lines define the critical path



(a) MCSS as given by Equation 5.5, where e[j'] and b[j'] are pre-computed for next iteration



(b) NW as given by Equation 5.9. C_1, C_2 and C_3 are precomputations. C_3 is similar to C_1.



(c) Cicuit for LCS as given by Equation 5.10. $A2''$,$A3''$ and $A4''$,$A7''$ are precomputations.

Figure 5.15: Circuits for each element in RVEP, , bold lines define the critical path

Table 5.1: Results to show time and hardware utilized

Type	Implementation	Frequency (MHz.)	Time to compute $n \times m$ elements (ns)	Speed-up	Slices	area overhead
NW (B=2) $n = 40, m = 40$	RVEP	88.75	439.43	1.69	8018	2.29
	RVENP	78.91	494.24	1.5	7557	2.16
	Systolic	106.58	741.25	1	3500	1
LCS (B=2) $n = 40, m = 40$	RVEP	97.68	399.27	1.03	3772	3.62
	RVENP	115.73	337	1.22	3641	3.49
	Systolic	192.20	411.02	1	1043	1
MCSS (B=3)	RVEP	159.26	6.28	2.47	118	3.02
	RVENP	101.71	9.83	1.58	120	3.08
	Systolic	193.09	15.54	1	39	1
Fibonacci (B=4)	RVENP	202.55	4.94	3.01	69	1.68
	Systolic	269.14	14.88	1	41	1

and RVEP for all the representative problems is shown in Figure 5.14 and 5.15.

To appreciate the performance of the two RVE techniques, we have implemented the systolic array for both the RVE techniques and the systolic array of dataflow technique for the four problems, targeted for Xilinx 4 platform, which contains 25280 slices. We call the systolic array dataflow implementation as *Systolic* in Table 5.1. The processing element design is written in VHDL and simulated and synthesized on ModelSim 6.5 and Xilinx 10.5 respectively. The results are summarized in Table 5.1 and graph in Figure 5.16. The speedup and area-overhead is computed with respect to the dataflow systolic array implementation.

We have implemented NW by using both RVE techniques with a blocking factor $B = 2$ and then have compared it to its systolic array dataflow imple-

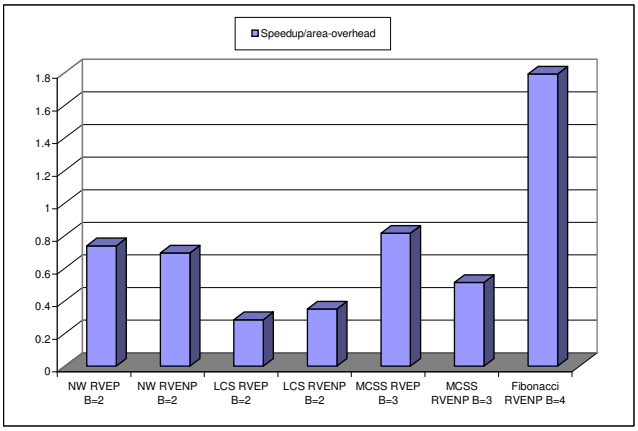


Figure 5.16: Graph to show speedup/ area-overhead w.r.t. systolic dataflow

mentation. The results show that the both RVEP and RVENP implementations are 1.69x and 1.5x faster than the systolic implementation respectively. These speedups are at the cost of more area, which is 2.29x and 2.16x respectively higher than the systolic implementation. Figure 5.16 shows that speedup/area-overhead for RVEP is better than RVENP for NW as the critical path for RVEP is shorter than RVENP as shown in Figure 5.14c and 5.15b. The exclusion of pre-computation from the critical path makes the critical path short. The slight increase in slices for RVEP with respect to RVENP is due to the extra registers needed to store the intermediate pre-computed element.

We have implemented LCS using both RVE techniques with a blocking factor $B = 2$. The results show that the time to compute the block of $n = 40$ and $m = 40$ using RVEP implementation is approximately the same as the time for the systolic approach. The speedup/area-overhead for RVEP is less than RVENP as shown in Figure 5.16. Therefore, it is not beneficial to apply RVEP. However, there is a 1.22x speedup in case of RVENP on top of Systolic with less area usage than RVEP but still substantially more area than in the Systolic case. In an attempt to explain the lack of good enough performance improvement for RVEP, we found that RVENP is faster in spite of the same critical path (see Figures 5.14d and 5.15c). The reason is that the portion of elements that are pre-computed is not enough to compensate for the additional routing complexity in the RVEP circuitry.

We have chosen $B = 3$ for both RVE techniques in the implementation of MCSS. The RVEP implementation is 2.47x faster than the systolic implementation but with twice the area usage. The RVENP implementation is only 1.58x

faster than systolic also with around twice more area usage. This shows that RVEP results in a faster implementation than RVENP with similar area consumption. The critical path of the RVEP implementation is shorter than the RVENP implementation. The area of RVEP is slightly lower despite the use of more registers because it uses less adders than the RVENP implementation as shown in Figures 5.14a and 5.15a.

There is only RVENP implementation for the Fibonacci algorithm, as it does not have any pre-computation part. We have implemented the multipliers in Figure 5.14b with shifts and adds. We choose $B = 4$ for RVENP implementation of Fibonacci numbers. The results show that RVENP implementation of Fibonacci is 3.01x faster than the systolic implementation at the cost of 1.68x area. Figure 5.16 shows that the speedup/area-overhead for RVENP in case of Fibonacci is better than dataflow, because we are able to optimize the RVENP circuit for area more than before.

The results show that usually RVEP gives a higher speedup than the RVENP implementation shown for both NW and MCSS. However, the RVENP solution is preferable than RVEP when there are little or no pre-computations to be done as in the case of LCS. The speedup of both the RVE techniques depends upon the structure of the recurrence equation and the number of associative operators in the recurrence equation which will be explained in the next Section.

5.5 Applicability of the RVE Techniques to DP Problems

In this section, we generalize the RVE variants and define a framework such that they can be applied to the class of dynamic programming problems as long as they comply to the earlier formulated conditions: a constant number of dependences and the constant iteration space distance.

DP is usually applied to a class of optimization problems which exhibit two basic properties [114].

1. The problem exhibit optimal substructure property. An optimization problem has a *optimal substructure* if the optimal solution to the problem depends on the optimal solution of its subproblems.

2. There exist *overlapping subproblems* meaning that one can find the solution by recursion over the same subproblem. The total number of distinct

subproblems is polynomial in the input size.

The optimal substructure implies that DP problems are defined by a recurrence functional equation whose left hand side is the function name and the right hand side is max/min expression of some monotonous cost function [17]. The general form of DP optimization problem with a constant number of dependences, for which the RVE approach is applicable is the following:

$$P[I_{i,j}] = \begin{array}{c} \prod P[I_0] \oplus_2 C_0 \quad \text{if } A(I_0) \\ \vdots \qquad\qquad\quad \vdots \\ \prod P[I_h] \oplus_2 C_h \quad \text{if } A(I_h) \\ M \begin{cases} \prod P[I_{h+1}] \oplus_2 C_{h+1} & \\ \vdots & \text{if } A(I_{h+1}) \\ \prod P[I_m] \oplus_2 C_m & \end{cases} \end{array} \tag{5.11}$$

where $\prod P[I_k] = P[I_{k,0}] \oplus_1 P[I_{k,1}] \oplus_1 \cdots \oplus_1 P[I_{k,n_k}]$, for $0 \leq k \leq m$

Here $P[I_{i,j}]$ is the optimization problem that needs to be solved. $P[I_{k,l}]$ for $0 \leq l \leq n_k$ is a subproblem of $P[I_{i,j}]$. Furthermore, M = max or min depending upon the nature of the problem, $I \in \mathbb{Z}^n$ defines the position of problem in the n-dimensional space, $I_{i,j} - I_{k,l}$ remains constant for any iteration, which means the position of the subproblems is invariant w.r.t to the reference it is being computed. $\oplus_1$ is the operator between the recursive functions and $\oplus_2$ is the operator between the recursive function and the cost function C_i, which can be constant or a function of I. In most of the DP problems, $\oplus_1$ and $\oplus_2$ are associative in nature. $A(I_g)$ for $g = 0, 1, 2, \cdots, h+1$ is the condition that chooses the relevant solution accordingly. If $n_k = 0$ then it is called *monadic* otherwise it is called *polyadic*. The general form of monadic DP problems with constant number of dependences derived from Equation 5.11 is the following.

$$P[I_i] = \begin{array}{c} P[I_0] \oplus_2 C_0 \quad \text{if } A(I_0) \\ \vdots \qquad\qquad\quad \vdots \\ P[I_h] \oplus_2 C_h \quad \text{if } A(I_h) \\ M \begin{cases} P[I_{h+1}] \oplus_2 C_{h+1} & \\ \vdots & \text{if } A(I_{h+1}) \\ P[I_m] \oplus_2 C_m & \end{cases} \end{array} \tag{5.12}$$

Here $P[I_i]$ is the optimization problem that needs to be solved. $P[I_k]$ for $0 \leq k \leq m$ is a subproblem of $P[I_i]$. NW, LCS and MCSS are monadic

DP problems with a constant number of dependences and that follow Equation 5.12. Equation 5.12 shows that the optimal solution of $P[I]$ depends on the optimal solution of its m subproblems, therefore, if we apply RVE, then the number of sub-equations will be exponential in m, which generally cannot be solved efficiently. This can be seen from Figure 5.4c and 5.4d for NW and LCS respectively, where every node at each level is expanded to 3 nodes. Therefore, we apply the mixed solution by combining RVE and dataflow to limit the growth of sub-equations to be solved as shown in Figure 5.9 in case of NW.

The Fibonacci algorithm as given by Equation 5.2 is one of the simplest polyadic DP problem and it follows Equation 5.11, where $k = 0$ and $l = 1$. It is not an optimization problem and also have no condition. Finding Binomial coefficient by using Pascal's triangle is another polyadic DP problem. Similar to monadic DP problems, polyadic problems also expand exponentially, therefore, requiring the hybrid approach.

Since DP problems have the overlapping subproblems property, redundant leaf nodes obtained after RVE application can be removed without loss of generality in the min/max equation. We can even improve the overall speedup when the number of overlapping problems is higher as only one has to be solved to find the global solution. This is seen in Section 5.3.2, where 13 leaf nodes in case of NW were reduced to 10 and 13 leaf nodes to 8 in case of LCS.

The cost function C_i as given by Equation 5.11 and 5.12 is either a constant or a function of I, which is known in advance, therefore C_i can be precomputed for the next position I' which is also known while the optimization problem for the current position I is computed. E.g., C_1, C_2 and C_3 were precomputed for the next (i, j) values, while $F[i, j]$ for the current (i, j) is being computed in case of NW described in Section 5.3.4. This increases parallelism and can help in reducing the critical depth of the pipeline.

The depth reduction of a pipeline for a problem after RVE is applied, depends upon number of reasons. The main reason is that when RVE is applied, it increases the number of terms in each sub-equation, between which an associative operator is applied. More the terms in a sub-equation with associative operator between them, more they can be computed efficiently by computing them in a balanced tree as done in the *recursive doubling decomposition* algorithm [63]. Secondly, after RVE, there are a number of sub-equations which are identical as the DP problems have overlapping subproblem property. We can eliminate those sub-equations while taking the max/min, as taking the max/min of two identical sub-equations is the same. Taking into account all these fac-

tors, we can explain the difference of acceleration between MCSS, Fibonacci, NW and LCS as compared to traditional dataflow approach.

To minimize the memory access, RVE blocks connected with other RVE blocks in a dataflow manner, can be arranged efficiently in a systolic array as described in [119] for elements connected with other elements in dataflow.

5.6 Summary and Conclusion

In this chapter, we have presented a generic algorithm to apply RVE to DP problems, that can generate highly efficient circuits. We have devised two variants of our technique to implement on FPGA called as RVENP and RVEP. Both can accelerate the DP problems better than the well known dataflow approach. Section 5.5 provided a detailed discussion of how Equation 5.11 can be applied to the class of DP problems as long as they comply to the conditions stated explicitly in the beginning of the chapter. In the next chapter, we apply the same technique on Smith-Waterman (SW) algorithm, which is again a type of DP problems and is very widely used in molecular biology.

Note.

The content of this chapter is based on the the following paper:

Z. Nawaz, T. P. Stefanov, K.L.M. Bertels, *Efficient hardware generation for dynamic programming problems*, proceedings of International Conference on Field-Programmable Technology 2009, pp. 348-352, Sydney, Australia, December 2009.

6

Acceleration of Smith-Waterman

SEQUENCE alignment is one of the most widely used operations in computational biology. It is typically used to compare newly determined sequences to known sequences in a database and to find the similarities among them. This helps in discovering functional, structural and evolutionary information in biological sequences of DNA, RNA and proteins. There are two types of sequence alignment namely local and global. The choice of alignment depends upon the type of problem, but the most widely used is local alignment. Smith-Waterman (SW) algorithm [106] computes the optimal local alignment. It is used to align two apparently dissimilar sequences which include some pattern which is highly conserved. The algorithm finds that highly conserved pattern and ignores the patterns that show little similarity. This algorithm belongs to the family of dynamic programming problems, which has time and space complexity $O(mn)$, where m and n are lengths of the sequences being aligned. Although this complexity seems to be acceptable, the exponential growth in bio-sequence databases of known sequences makes this complexity extremely challenging to manage [19, 41]. Therefore as the database size grows larger, faster algorithms become important to quickly compare and align the sequences. Even a small relative improvement can still have a large absolute impact.

There are some heuristic techniques like FASTA [79] and BLAST [10], which also compute the local alignment. These algorithms have lower time complexity and are thus faster. However, they do not guarantee to find the optimal alignment. As Smith-Waterman belongs to the category of dynamic programming problems, it seems likely that it can be parallelized to obtain a shorter execution time.

Smith-Waterman algorithm belongs to the category of exponentially expanding RVE expansion. Therefore, we have to use the RVENP and RVEP variants in

order to control this exponential expansion. In this chapter, we apply RVENP and RVEP on SW and show that our implementation accelerates more than dataflow approach at the cost of more area. This increase of area is not a problem because of two reasons. First, we are looking at high performance computing, where we are ready to give away more area to achieve higher speed. Second, future generations FPGA's will have enough area thanks to Moore's law.

The main contributions of this chapter are

1. To provide an actual implementation of RVENP and RVEP for different versions of SW and varying blocking factors for RVE. As stated above, we obtain performance improvements ranging from 1.46x up to 2.29x at the expense of up to 3x higher area consumption.

2. Modify RVE such that the algorithm adopts to clipping to zero factor and give correct optimal values.

The rest of the chapter is organized as follows. In the following section, we briefly describe the Smith Waterman algorithm and how it is implemented on a uniprocessor. In section 6.2, we present the related work. Section 6.3 describes the way the RVE technique is applied to SW and also shows the mapping to actual circuits. Section 6.4 discusses the experimental setup and obtained results showing the execution time and hardware usage for different versions of SW with different blocking factors of RVE. Finally, we conclude the chapter in Section 6.5.

6.1 The Smith-Waterman algorithm

Smith-Waterman algorithm is a dynamic programming algorithm, which fills a two-dimensional matrix with optimal alignment scores as shown in Figure 6.1. The two sequences to be aligned are placed along the row and column of the matrix. The matrix is filled row-wise starting from the top-left corner. After the matrix is filled, a traceback is performed starting from the maximum value in the matrix, like 6 from Figure 6.1.

Let $S[1..m]$ and $T[1..n]$ be the two sequences of length m and n for the sequence alignment. The *optimal alignment score* $F[i,j]$ for the two subsequences $S[1..i]$ and $T[1..j]$ is given by the following recurrence equation.

		G	T	C	G	C	A	A	C
	0	**0**	0	0	0	0	0	0	0
T	0	0	**2**	0	0	0	0	0	0
C	0	0	0	**4**	**2**	2	0	0	2
C	0	0	0	2	3	**4**	2	0	2
A	0	0	0	0	1	2	**6**	4	2
T	0	0	2	0	0	0	4	5	3
G	0	2	0	1	2	0	2	3	4

Figure 6.1: Matrix for an example of SW algorithm, when $a = -2$ and $x(i,j) = +2$ when *S[i]=T[j]* otherwise -1. Elements in the traceback are shown in bold.

$$F[i,j] = \max \begin{cases} D[i,j] \\ F[i-1,j-1] + x[i,j] \\ E[i,j] \\ 0 \end{cases} \tag{6.1}$$

and

$$D[i,j] = \max \begin{cases} F[i,j-1] + a \\ D[i,j-1] + b \end{cases}$$

$$E[i,j] = \max \begin{cases} F[i-1,j] + a \\ E[i-1,j] + b \end{cases}$$

where $F[0,0] = D[0,0] = E[0,0] = F[0,j] = D[0,j] = E[0,j] = F[i,0] = D[i,0] = E[i,0] = 0$, for $1 \leq i \leq m$ and $1 \leq j \leq n$. The $x[i,j]$ is the similarity score obtained from a scoring/similarity matrix for the corresponding characters $S[i]$ and $T[j]$. Equation 6.1 is called the SW algorithm for the *affine gap penalties* [45]. There are many scoring matrices and the choice of a scoring matrix depends upon the type of biological sequence and the goal of analysis. The scoring matrix has both the negative and the positive values. a is the penalty for opening a gap and b is the penalty of the following gaps in any sequence. Both a and b have the negative values. For a local alignment, $b > x_l > a$, i.e. the lowest score in the scoring matrix x_l is greater than the continuing gap penalty b and is less than the opening gap penalty a. Other-

wise, the alignment will have more gaps and will eventually change from a local to a global type of alignment, even though a local alignment algorithm is used [31, 52, 108]. This observation is used in Section 6.3 to simplify the RVE optimized equation for the affine gap penalties. Equation 6.1 takes care of the fact that probability of having a gap is low, but probability of having n consecutive gaps is not as bad as $n \times a$.

Sometimes, a simplified version of SW algorithm is also used, in which $a = b$. This is called *SW algorithm for linear gap penalties* and Equation 6.1 can be simplified as following:

$$F[i,j] = \max \begin{cases} F[i,j-1] + a \\ F[i-1,j-1] + x[i,j] \\ F[i-1,j] + a \\ 0 \end{cases} \tag{6.2}$$

Here $x_l > a$, which is used in Section 6.3 to simplify the RVE optimized equation for linear gap penalties.

Once, the whole matrix is filled, we find the maximum score in the whole matrix and then start the traceback from that element to one of the three elements from which the alignment score is calculated. This process is repeated till the score drops below a certain threshold or to zero. In the traceback, if the corresponding row and the column elements match then the alignment is computed from the top-left element, otherwise, it is computed from any of the three elements depending on which of them produces a maximum. When an element is computed from the top element then there is a gap in the sequence along the row, and similarly when an element is computed from the left element then there is a gap in the sequence along the column. The local optimal alignment for the example in Figure 6.1 is as follows.

```
TCGCA
|||||
TC-CA
```

The computation of optimal alignment score $F[i,j]$ as given by Equation 6.1 and 6.2 takes constant time, and since there are $m \times n$ elements to be computed, the time complexity for SW algorithm is $O(mn)$. The traceback takes $O(m+n)$ steps, as the longest path in the $m \times n$ matrix is from top left to the bottom right, which is $O(m+n)$, and the time to determine the source of the computation for an element is constant. We need to keep the table of size $m \times n$ to compute the optimal score as well as to perform traceback, therefore,

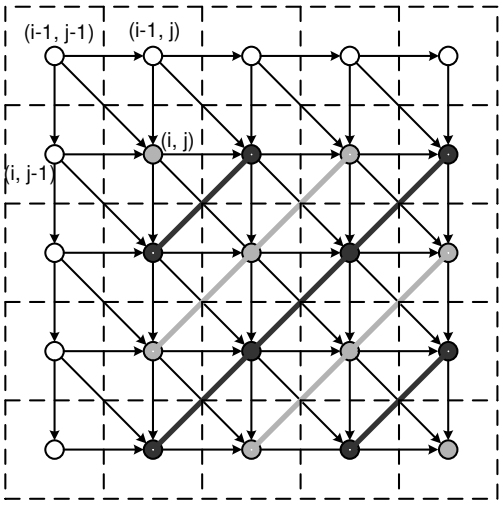


Figure 6.2: Data dependence graph for Equation 6.1 and 6.2 (different shades of gray in circles show the elements which can be executed in parallel).

the space complexity for the algorithm is also $O(mn)$.

To parallelize SW algorithm we need to look at its data dependence graph as shown in Figure 6.2. Blank circles are the elements after the initialization with the boundary conditions. Any iteration (i, j) cannot be executed until iterations $(i-1, j)$, $(i-1, j-1)$ and $(i, j-1)$ are executed first, due to the data dependences. However, if we traverse the elements in a wavefront manner starting from the top-left corner as shown in Figure 6.2, all the elements in the diagonal can be executed in parallel. The degree of parallelism is constrained to the number of elements in the anti-diagonal. The maximum number of the processing elements required will be equal to the number of elements in the longest anti-diagonal (l_d). l_d is defined as follows:

$$l_d = \min(m, n) \tag{6.3}$$

Here, we have assumed that the processing elements are equal in number to the length of the shorter sequence. Theoretically, the lower bound to the number of steps required in this parallel implementation is equal to the number of anti-diagonals required to reach the bottom-right element is as follows:

$$m + n - 1 \tag{6.4}$$

The profiling of SW algorithm shows that filling the matrix takes 98.6% of the

overall time to find the optimal alignment [110]. Therefore, filling the matrix is an obvious choice to be accelerated on an FPGA.

6.2 Related Work

As sequence alignment is one of the most widely used operation in computational biology, a lot of work has been done to accelerate it by using different hardware. In addition to the specific architectures designed for sequence alignment, many solutions for special purpose hardware like SIMD, CELL/BE and FPGAs have been devised [51].

Several implementations for SIMDs have been proposed as MGAP, Kestrel and Fuzion [21,32,103]. A recent implementation is done on a Intel Xeon 2.0 GHz using a technique called Striped Smith-Waterman, which claims to achieve a speedup of six times over the other SIMDs implementations [34]. SIMDs contain the general purpose processors, therefore, it is programmable and is used for a wider range of applications like image processing and scientific computing. The drawback is that they are expensive. The striped SW has also been modified for Cell/BE [35, 113].

FPGAs are a good choice to accelerate sequence alignment as they provide a lot of parallelism. A linear systolic implementation of the dataflow approach is widely used for this purpose. Some of the solutions based on FPGAs using the dataflow are given in [18,78,93,131–133]. Recently, Jiang et al. [57] modified SW algorithm formula by introducing a new variable, thereby, reducing the critical path.

The implementations similar to ours are presented in [49,50]. Both of them are mere implementation of RVENP technique [88] and model only SW algorithm with linear gap penalty. In [50], a hardware implementation is done based on a rectangular systolic array implementation. [49] is more close to our current implementation, as it is also a linear systolic array implementation. We have not only implemented RVENP but also RVEP using a linear systolic array for both linear and affine gap penalties.

6.3 Application of RVE to SW Algorithm

Instead of computing one element, we can compute a block of $k \times k$ elements in parallel, by applying the RVENP or RVEP for a blocking factor $B = k$.

When it is applied for a blocking factor $B = 2$ to Equation 6.1 and Equation 6.2, we get the following equations for $F[i,j]$ in a 2×2 block.

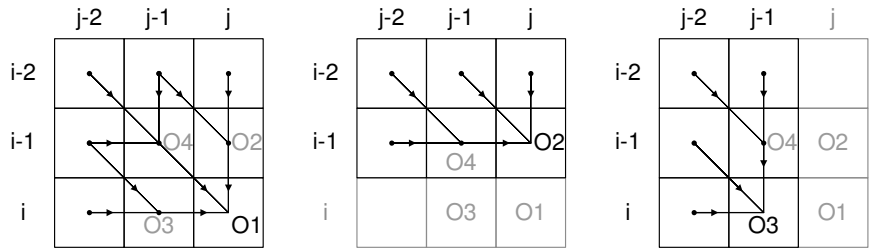
$$F[i,j] = \max \begin{cases} i & D[i,j-2] + 2b \\ ii & F[i,j-2] + c_2 \\ iii & F[i-2,j] + c_2 \\ iv & E[i-2,j] + 2b \\ v & (D[i-1,j-2] \succ E[i-2,j-1]) + c_1 \\ vi & F[i-1,j-2] + c_3 \\ vii & F[i-2,j-1] + c_4 \\ viii & F[i-2,j-2] + c_5 \\ ix & 0 \end{cases} \tag{6.5}$$

where $c_1 = b + x[i,j]$, $c_2 = a + b$, $c_3 = a + (x[i,j] \succ x[i,j-1])$, $c_4 = a + (x[i,j] \succ x[i-1,j])$ and $c_5 = x[i,j] + x[i-1,j-1]$ in Equation 6.5. Here $\succ$ is defined as the max operator.

$$F[i,j] = \max \begin{cases} i & (F[i,j-2] \succ F[i-2,j]) + 2a \\ ii & F[i-1,j-2] + C_1 \\ iii & F[i-2,j-2] + C_2 \\ iv & F[i-2,j-1] + C_3 \\ v & 0 \end{cases} \tag{6.6}$$

where $C_1 = a + (x[i,j-1] \succ x[i,j])$, $C_2 = x[i,j] + x[i-1,j-1]$ and $C_3 = a + (x[i,j] \succ x[i-1,j])$ in Equation 6.6.

Equation 6.5 or 6.6 can be visually understood as finding the longest path in a graph reduced from the dependency graph by removing some of the edges which can never be a part of the solution. Equation 6.6 gives the longest path in the reduced graph showing the reduced paths from the known vertices ($F[i,j-2]$, $F[i-1,j-2]$, $F[i-2,j-2]$, $F[i-2,j-1]$ and $F[i-2,j]$) to an unknown vertex O1($F[i,j]$) as shown in Figure 6.3a. Likewise, Equation 6.5 and 6.6 to compute $F[i,j]$, we can find the equations to compute $F[i-1,j]$ (O2) and $F[i,j-1]$ (O3) in parallel for affine and the linear gap penalties. Figure 6.3b and 6.3c show the reduced path for RVE optimized equation for $F[i-1,j]$ (O2) and $F[i,j-1]$ (O3) using the linear gap penalties. The diagonal edges in these graphs can have the positive or the negative weights depending upon the $x[i,j]$ value. All the vertical and horizontal edges always have negative weights and



(a) *F[i,j]* (O1) **(b)** *F[i-1,j]* (O2) **(c)** *F[i,j-1]* (O3)

Figure 6.3: Reduced graphs

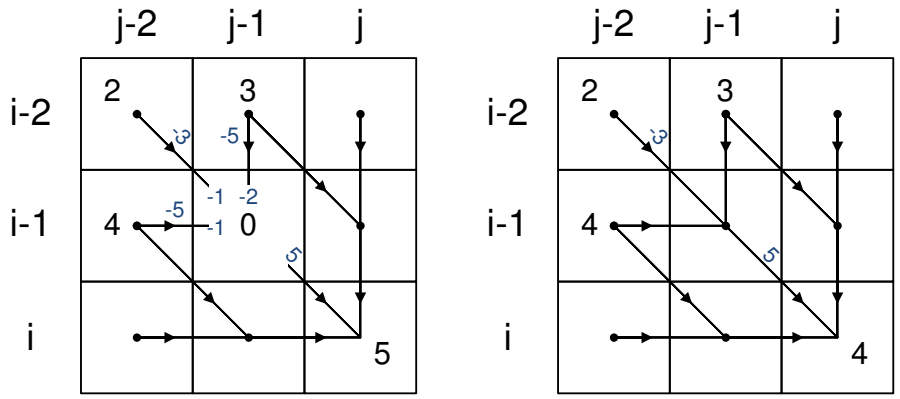
represent a gap in the alignment. The unknown variables (O1, O2, O3 and O4) can be computed in parallel, as shown in Figure 6.3 for the blocking factor ($B = 2$). Similarly, Equation 6.5 can be taken as representing the longest path in a similar graph with three levels [58].

6.3.1 Clipping Error

An aspect of SW is that negative values for the optimal score are not allowed and then always clipped to zero. This clipping introduces an error in the computation of the final optimal values when using the RVE method and thus has to be compensated for. In this section, we describe the patch to RVE which solves this issue.

In original SW algorithm Equations 6.1 and 6.2, which is for an element computation, $F[i,j]$ can never be negative, as a negative value is clipped to zero. Algebraically, the formulas for computing $F[i,j]$ as given in Equation 6.5 and 6.6 are correct, however due to the clipping it can produce smaller values than produced by the original formula for SW element as given by Equations 6.1 and 6.2.

This problem is explained in Figure 6.4, suppose $F[i,j]$ is computed from $F[i-2,j-2]$ and $F[i-1,j-1]$, $F[i,j]$ path. Lets look at the computation of $F[i,j]$ using Equation 6.2. Figure 6.4a shows a scenario when all the unknown elements (i.e. $F[i-1,j-1]$, $F[i,j-1]$, $F[i-1,j]$ and $F[i,j]$) are computed in a dataflow sequence using Equation 6.2. First $F[i-1,j-1]$ is computed from $F[i-2,j-2]$ to 0, as $\max(3-5,2-3,4-5,0) = 0$, then $F[i,j]$ is computed from $F[i-1,j-1]$ (as $F[i-1,j-1] + x[i,j] = 0+5 = 5$) to 5. However, when using the sub-equation (*iii*) of Equation 6.6 (as shown in Figure 6.4b) $F[i,j]$ is computed to 4. This difference is due to the clipping to



(a) computed using Equation 6.2 **(b)** computed using Equation 6.6

Figure 6.4: Clipping error

zero of the intermediate vertex $F[i-1, j-1]$ value as governed by Equation 6.2, when all the paths give the negative values. The problem is that Equation 6.6 computes $F[i, j]$ without explicitly computing the intermediate vertices and ignoring the clipping to zero for these intermediate vertices when the value becomes negative. A naive solution is to check all the paths that contain any intermediate vertex to compute an unknown element. Therefore, to compute O1 ($F[i, j]$), there are 7 paths which contain some intermediate vertices, which need to be checked for the clipping as shown in Figure 6.3a. Similarly to compute O2 ($F[i-1, j]$) and O3($F[i, j-1]$), there are two paths that need to be checked for the possible clipping as shown in Figures 6.3b and 6.3c, respectively. So overall for SW algorithm with the linear gap penalties, 11 paths need to be checked for the clipping error, when $B = 2$. We now show that it is enough to check only one path for the clipping error in case of $B = 2$.

To simplify the understanding, lets redefine the Equation 6.2 for some intermediate vertex p as shown in Figure 6.5.

$$F_p = \max \begin{cases} f_p \\ 0 \end{cases} \tag{6.7}$$

where $f_p = \{F_1 + w_1, F_2 + w_2, F_3 + w_3\}$.

According to Equation 6.7, F_p is clipped to zero when $f_p < 0$.

We call a vertex r is patched for the clipping, when the effects of the clipping are taken into account.

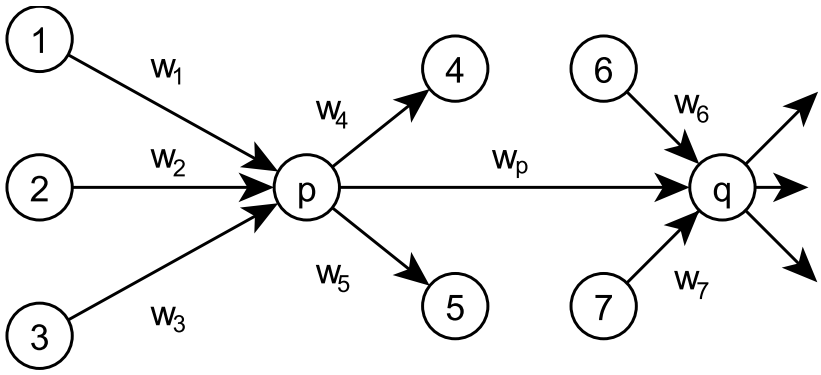


Figure 6.5: Graph to show intermediate vertex

Theorem 6.3.1 : Given a reduced graph having paths from known vertices to an unknown vertex q for the RVE optimized SW algorithm equation. Let p be an intermediate vertex in a path to compute the vertex q, which satisfy the clipping to zero requirement (i.e. $f_p < 0$). Then only those paths going through p need to be clipped, whose outgoing edge w_p from p has a positive weight and $f_p + w_p > 0$.

Proof. We look at the computation of F_q for some vertex q by using the RVE optimized equation, which has an intermediate vertex p satisfying the clipping to zero requirement (i.e. $f_p < 0$). Let p have three incoming edges e_1, e_2 and e_3 from vertices 1, 2 and 3 with weights w_1, w_2 and w_3 and three outgoing edges e_4, e_p and e_5 to vertices 4, p and 5 with weights w_4, w_p and w_5, respectively. Similarly, q have three incoming edges from vertices 6, p and 7 with weights w_6, w_p and w_7 and three outgoing edges as shown in Figure 6.5.

We prove this theorem for Equation 6.6, the proof is generalized enough to be extended for Equation 6.5 also. Lets look at the computation of F_q by using Equation 6.6 in which p is an intermediate vertex. We show that even when p satisfies the properties for the clipping as given by Equation 6.7 (i.e. $f_p < 0$), we will not clip F_p to zero while computing F_q until $w_p > 0$ and $f_p + w_p > 0$.

Lets look at an outgoing edge w_p from p to q which can have a negative or a positive weight and $f_p < 0$.

When w_p is negative (i.e. $w_p < 0$), then $f_p < 0$ does not effect the evaluation of the optimal value F_q for the vertex q. As $f_p + w_p < 0$, and is clipped to zero at the vertex q by Equation 6.6, when all the other incoming paths to q are also negative (i.e. $f_6 + w_6 < 0$ and $f_7 + w_7 < 0$). Otherwise, F_q gets the value of

$\max(f_6 + w_6, f_7 + w_7)$, assuming the intermediate vertices 6 and 7 have been patched for clipping to zero. In this case, there is no need to correct the vertex p for clipping as it will not effect the result.

When w_p is positive (i.e. $w_p > 0$), and produces $f_p + w_p > 0$, then $f_p + w_p$ produces a lower value than when computed by Equation 6.2 in the evaluation for F_q, when $\max(f_p + w_p, f_6 + w_6, f_7 + w_7) = f_p + w_p$, since $f_p + w_p < F_p + w_p$ as $f_p < 0$ and $F_p = 0$. Therefore, f_p needs to be corrected to zero in Equation 6.6 for producing the correct result. However, when $w_p > 0$ and produces $f_p + w_p < 0$, then it is similar as stated in positive weight case, and no correction is needed.

Similarly, using the same methodology, the theorem can also be proved for Equation 6.5. ■

Theorem 6.3.1 assumes that p is an intermediate vertex which satisfies the clipping requirement (i.e. $f_p < 0$). There are still many paths going to vertex q through vertex p and they need to be checked for clipping to zero (e.g. in Figure 6.5, paths from 1 to q, 2 to q and 3 to q). Now to check whether p satisfies the clipping to zero requirement, we prove that the checking of any one the path to p for clipping to zero, and then clipping to zero is sufficient to make a valid clipping instead of checking all the paths to p.

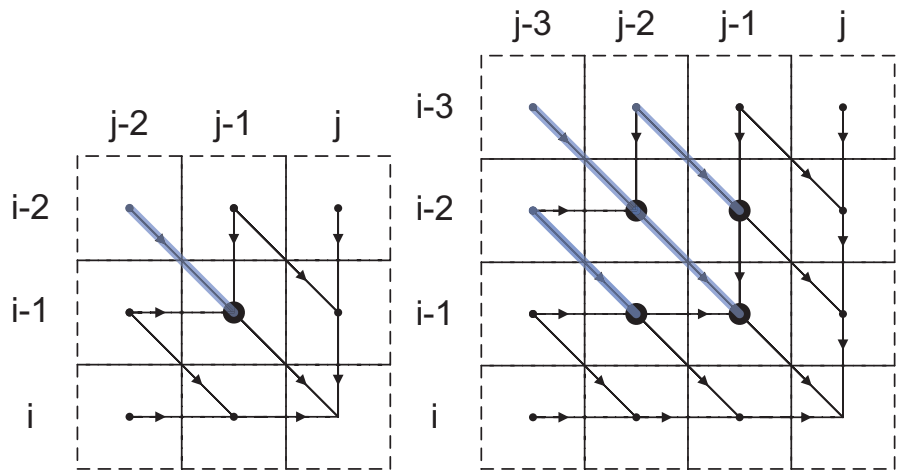
Theorem 6.3.2 Given a reduced graph for the RVE. let p be an intermediate vertex, which needs to be checked for clipping. Then checking of any one of the paths to p for clipping to zero and then making the correction, if that path has negative weight, will correct the clipping error for all the paths via p.

Proof. To compute F_q at a vertex q, Equation 6.5 and 6.6 can be written as F_q= max(weights of all the paths in a reduced graph from known vertices to a vertex q). p is one of the intermediate vertices where clipping to zero needs to be checked. Let's assume that a path going through p gives the max value, as the clipping effect at p is only visible at vertex q, when it is maximum, then

$$F_q = \max(f_p + w_p, 0) \tag{6.8}$$

where $f_p = \{F_1 + w_1, F_2 + w_2, F_3 + w_3\}$

As given by the statement, we check only one path to the vertex p, let it be from the vertex 2 to the vertex p, for clipping to zero, i.e. we check whether $F_2 + w_2 < 0$. If $F_2 + w_2 < 0$, we call vertex 2 a potential candidate for the clipping, then we make the correction and put $F_2 + w_2 = 0$, otherwise we keep the same. So, if all the other paths to p (i.e. from vertex 1 and vertex 3) are



(a) Block with B=2 **(b)** Block with B=3

Figure 6.6: Patch for clipping. The intermediate vertices with +ve outgoing edges are shown by large black circles. The paths that need to be checked for clipping are shown by bold lines

also negative (i.e. $F_1 + w_1 < 0$ and $F_3 + w_3 < 0$), then according to Equation 6.8, correction to zero in the path from vertex 2 corrects the clipping error for all the paths to p and returns $F_q = w_p$. Otherwise, any other positive edge takes the effect and no clipping error correction is required. It means that F_q can never have a negative value according to Equation 6.8, and it will compute w_p, when $f_p < 0$. ■

6.3.1.1 Patch

Theorems 6.3.1 and 6.3.2 define a patch to correct the clipping error in an RVE block of any blocking factor. First, according to Theorem 6.3.1, we shortlist the intermediate vertices with possible +ve outgoing edges. Then according to Theorem 6.3.2, for each shortlisted vertex, we consider only one path from the known vertices to a shortlisted intermediate vertex to check whether it is the potential candidate for the clipping. Let p is one of the shortlisted intermediate vertex and we choose path from any known vertex i to vertex p to check whether $F_i + w_i < 0$, where F_i is the optimal score at vertex i and w_i is the weight of the path from i to p. If $F_i + w_i < 0$, we put $F_p = 0$, otherwise keep the same. This way, the clipping error can be corrected for the whole block.

Now according to Theorems 6.3.1 and 6.3.2, we need to check only one path going through $F[i-1, j-1]$ while computing $F[i, j]$ using Equation 6.6 as shown in Figure 6.6a, instead of 11 paths as described earlier for $B = 2$. Similarly, we check only four paths for clipping in case of $B = 3$ as shown in Figure 6.6b. We implemented this patch and found that our implementation gives the same result as the results of the original SW Equations 6.1 and 6.2.

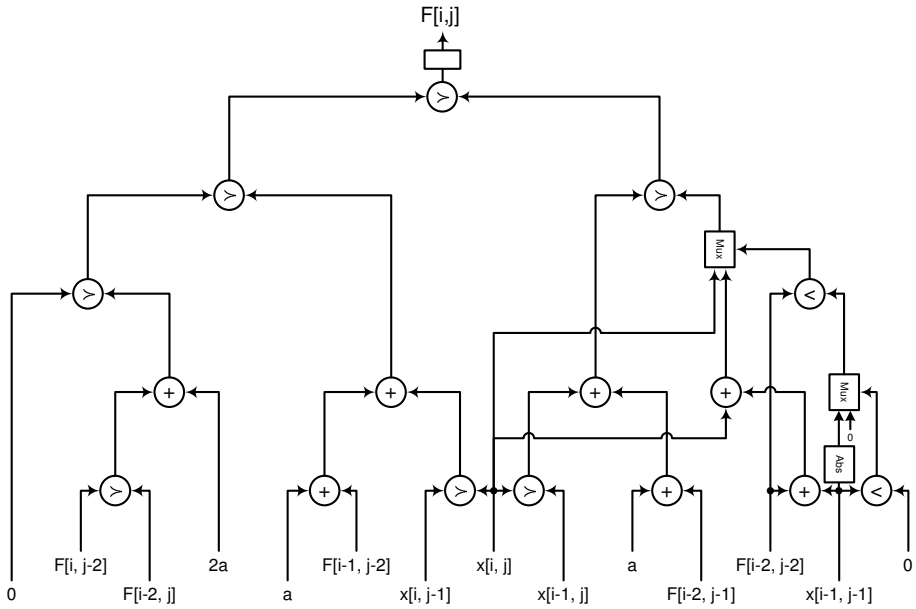
6.3.2 Mapping Equations to Circuits

There are two ways Equations 6.5 and 6.6 can be mapped to the circuits as discussed in Chapter 5. The first, RVENP (RVE with No Pre-computation), is a simple implementation of the RVE equation for each iteration (i, j).

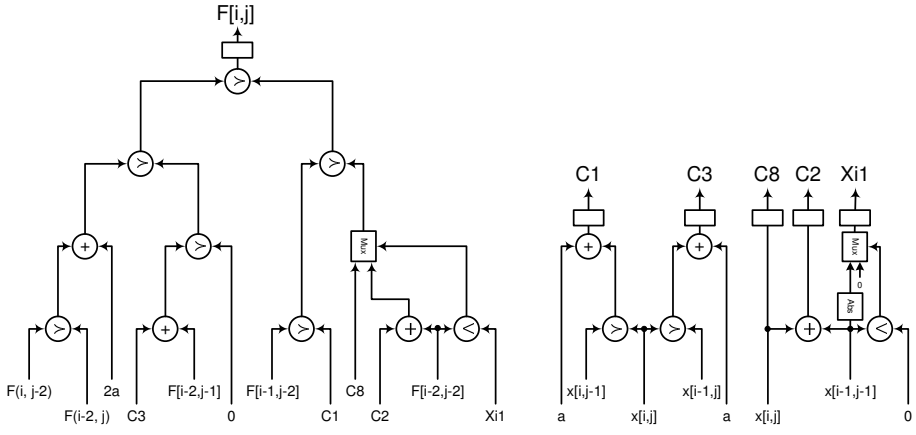
In the second implementation, we divide the RVE equation for one iteration (i, j) into two parts. One that can be computed prior to the current iteration (i, j) (pre-computed) as it is known earlier and second, which becomes known at the current iteration (i, j), this implementation is called as RVEP (RVE with pre-computation). In Equation 6.5, $c_1, c_2, \ldots, c_5$ for any iteration can be computed in advance as all its content are known in the start of the computation. Similarly, in Equation 6.6, C_1, C_2 and C_3 can be pre-computed. The RVE implementation for SW algorithm with linear gap penalty is shown in Figure 6.7a and RVEP implementation is shown in Figure 6.7b. Both these figures include correction for the clipping error.

The pre-computation increases the parallelism by dividing the same equation into two parts and computing both of them in parallel. This, however, slightly increases the area with the introduction of the extra registers to save the pre-computed values. This way, pre-computation may also reduce the critical path depending upon the problem. First implementation, RVENP, of SW algorithm for the linear gap penalties has the critical path from $F[i, j-2]$ to $F[i, j]$ as shown in Figure 6.7a, which has 5 levels. However, in the RVEP implementation, the critical path is reduced to 4 levels from $F[i, j-2]$ to $F[i, j]$ as shown in Figure 6.7b.

A block in optimal value matrix is computed using the circuit in Figure 6.7a or 6.7b and other circuits to compute $F[i-1, j]$, $F[i, j-1]$ and $F[i-1, j-1]$, we call all these circuits as the block circuit. Similarly, we map $\frac{n}{B}$ block circuits on the FPGA. The blocks in the optimal value matrix is computed using the block circuits as shown in Figure 6.8a where block circuit v computes the optimal value block (u, v) in cycle u. This structure of filling motivates us to use a systolic array as shown in Figure 6.8b. This shows that when a block is

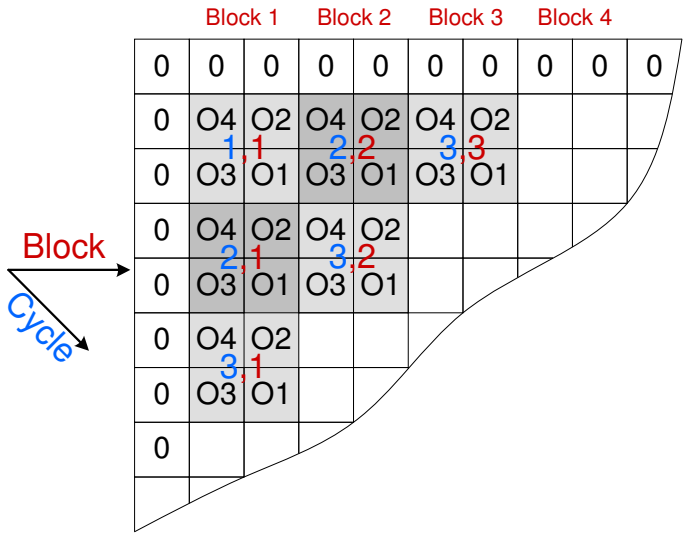


(a) RVENP

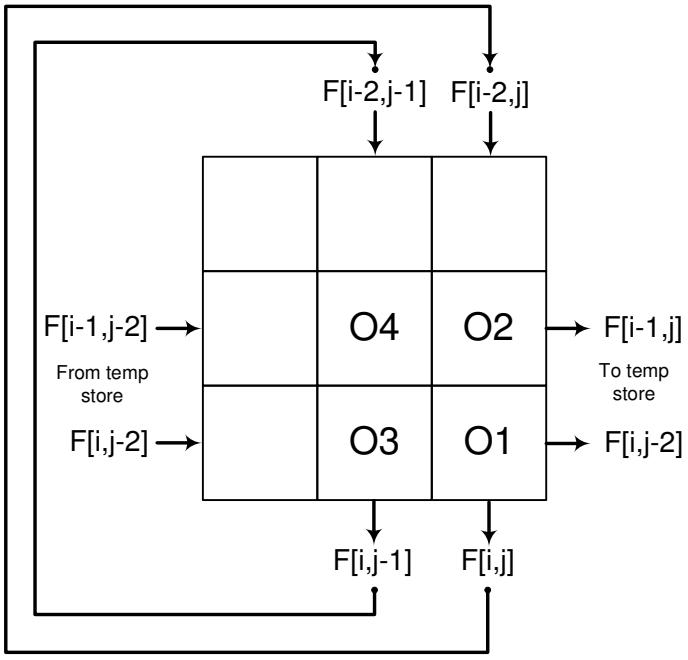


(b) RVEP

Figure 6.7: $F[i,j]$ computation in a block for SW with linear gap penalties



(a) Sequence of fill, Blocks at the top show the hardware to be used in each column. Number in red represent the block used and the number in blue represent the cycle in which it is used.



(b) Systolic array

Figure 6.8: Filling the whole table

Table 6.1: Results to show time and hardware utilized

Type	Implementation	Frequency (MHz.)	Time to compute $n \times m$ elements (ns)	Speed-up	Slices	overhead
SW (B=2) Linear $n = 40, m = 40$	RVEP	79.33	491.60	1.65	9150	2.05
	RVENP	70.31	554.74	1.46	8683	1.94
	Systolic	97.59	809.51	1	4468	1
SW (B=3) Linear $n = 36, m = 36$	RVEP	73.215	286.818	2.29	12335	3.21
	RVENP	56.148	374.01	1.76	11790	3.06
	Systolic	98.797	657.93	1	3848	1
SW (B=2) Affine $n = 26, m = 26$	RVEP	60.35	414.25	1.68	12497	2.85
	RVENP	59.32	421.48	1.65	12872	2.94
	Systolic	72.959	699.01	1	4380	1

computed, the output taken out in the vertical direction is fed back as input for the same block circuit to compute the next iteration in the fill. The horizontal data is stored in some temporary storage like BRAM, which is used in some later iteration.

6.4 Performance Evaluation

We compared two variants of the RVE implementation with single element systolic array implementation for three different cases by changing the blocking factor B and two variants of the SW algorithm. The results show that the RVE

implementation is faster than the single element systolic array implementation. Furthermore, the results show that the RVEP implementation is faster than the RVENP. We have generated the PE design in VHDL and targeted the Xilinx Virtex II pro platform, which contains 13696 slices. The code was simulated and synthesized on ModelSIM SE 6.5 and Xilinx ISE 10.1 respectively.

First, we have implemented the linear systolic dataflow implementation of Equation 6.1 and 6.2, which is equivalent to other comparable implementations on FPGA [57, 93, 131, 132]. We call this implementation as *Systolic* in Table 6.1.

The query sequence and corresponding row of the scoring matrix is loaded as the preprocessing step. This loading is done for aligning the query sequence with all the known sequences in the database. However, when a new sequence is to be aligned with all the sequences in the database, we can load the new unknown sequence and its corresponding row of the scoring matrix by utilizing the partial reconfiguration of the reconfigurable fabric.

We have implemented the protein sequence alignment using Blossum 62 as the scoring matrix, in which the highest value is 11 and the lowest is -4. Therefore, 5 bits can be used to store an element in the LUT. The data width for each block computation is 16 bit, which is sufficient to align a sequence of more than 5k length. We have chosen protein sequence alignment as it has higher values for $x[i, j]$ and therefore, the same design can be used for DNA sequence alignment by changing the corresponding LUTs.

The number of block processing elements p available for computing the sequence alignment depends upon the size of FPGA used. This also limits the size l of the query sequence used, as the length of the query sequence that can be accommodated on the FPGA is equal to $q = B \times p$. In this chapter, we are producing the results when $l \leq q$. However, if the size of query sequence l is larger than that can fit on the available number of PEs (i.e. $l > q$) on the FPGA, then two solutions can be used. First simpler and easier solution is to split the length l into k parts, such that $((k-1) \times q) < l \leq (k \times q)$, and then execute k passes sequentially by storing the intermediate results in some temporary storage. The second method, which is more suitable for this kind of implementation, but expensive to implement is to use high end machines like CrayXD1, which can have 150 FPGAs mounted on a single machine [110]. All these FPGA can be used to map the circuits and run in parallel when required.

There is no issue to accommodate the length d of database sequence in the design. It can be very large as it only depends upon the size of temporary storage available, BRAM in our case for storing the intermediate results.

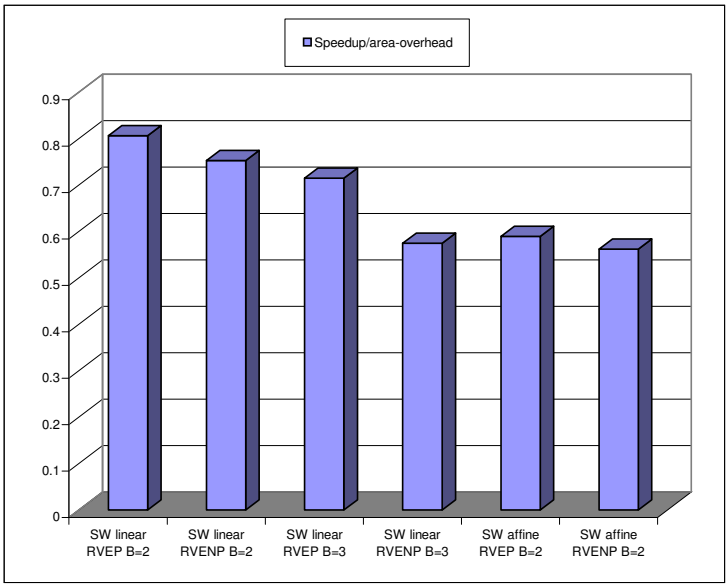


Figure 6.9: Graph to show the speedup/area-overhead w.r.t dataflow for SW

In the first case, we selected SW algorithm with linear gap penalties and applied both the RVE techniques with $B = 2$ and compared with the Systolic implementation. To compute a block of 40×40 elements, RVEP is 1.65x faster than the Systolic implementation, which is used in all previous SW parallel implementations. Similarly, RVENP is 1.46x faster than the Systolic implementation. This speedup is at the cost of around 100% area on top of the Systolic implementation. In the second case, we compute a block of 36×36 elements for the same SW version, however the blocking factor for RVE is increased to 3. By increasing the blocking factor, the speedup is increased to 2.29x in the case of RVEP and 1.76x in the case of RVENP as compared to the Systolic implementation for the same block computations. This speedup is again at the cost of around 2.82% more area than the Systolic implementation. This increase of speedup by increasing the blocking factor shows that the speedup can be improved by increasing the blocking factor which, however, also increases the area consumed. The higher speedup of RVEP as compared to RVENP is due to the reduction of critical path, which is done by splitting the computations into two parts. This requires more intermediate registers to store the pre-computation results. Therefore, it consumes a bit more area than RVENP.

Finally, we implemented SW algorithm with the affine gap penalty and chose

$B = 2$ for the RVE implementations. RVEP shows a speedup of 1.68x more than the Systolic implementation, which is also more than the RVENP's 1.65x speedup. The interesting thing was that its area consumption was less than the RVENP, despite the fact it used extra registers to store the intermediate pre-computation results. Both of these RVE implementations were optimized for speed. The critical path of both RVE techniques took the same number of levels, that's why the difference in speedup is not big. However, in the case of RVEP, we were able to increase the reuse of sub-expression in Equation 6.6 as compared to RVENP, while keeping the speed as the priority. Therefore, in the case of RVEP, we were able to process the same block in less area than the RVENP. This also reduced the net delay for the RVEP as compared to the RVENP and created a small difference in time. These results are summarized in Table 6.1. The graph depicted in Figure 6.9 shows that the speedup/area-overhead of RVEP is better than RVENP, therefore, it is recommended to use RVEP to accelerate Smith-Waterman algorithm.

6.5 Summary and Conclusion

In this chapter, we have applied the generic steps devised in Chapter 5 to an important bioinformatics algorithm, Smith-Waterman. Smith-Waterman formula contains a clipping factor that does not allow negative values. This leads to incorrect values when RVE is applied. We have proposed an efficient patch that compensates that error for SW RVE implementation with any blocking factor. We have implemented two RVE based variants, RVENP and RVEP, for two versions of SW formulation and with varying blocking factor which produce more speedup for SW than the widely used dataflow approach. We implemented RVENP and RVEP on Xilinx Virtex II pro platform showed 2.29x more speedup at the cost of 2.82x more area than dataflow approach. Likewise Chapter 5, the results show that RVE techniques are again better than dataflow to get high performance when area utilization is not the major restriction. Furthermore, we also show that RVEP gives better performance than RVENP for SW problem.

In the next chapter, we will describe a pipelined design for SW traceback that returns the sequence alignment immediately after once performing the matrix fill for all the sequences in the database. We will also discuss the memory bottleneck that arises due to the change in procedure. We have devised the solution for this memory bottleneck, which can be easily implemented on the current off-the-shelf FPGA boards.

Note.

The content of this chapter is based on the the following paper:

Z. Nawaz, M. Shabbir, Z. Al-Ars, K.L.M. Bertels, *Acceleration of Smith-Waterman Using Recursive Variable Expansion*, 11th Euromicro Conference on Digital System Design (DSD-2008), pp. 915-922, Parma, Italy, September 2008.

Z. Nawaz, H. Sumbul, K.L.M. Bertels, *Fast Smith-Waterman hardware implementation*, International Parallel and Distributed Processing Symposium, pp. 1-4, Atlanta, USA, April 2010.

7

A parallel Smith-Waterman traceback

In the last chapter, we saw that there are two stages in Smith-Waterman (SW) algorithm namely matrix fill and traceback. First, we fill the matrix with optimal score found, then we find the maximum of the optimal score. Finally we perform the traceback starting from the maximum value. This procedure is performed for all the sequences in the database. Since the matrix fill stage takes 98.6% of the overall time [110], all FPGA implementations use FPGAs for accelerating the matrix fill stage.

There are two methods to perform the sequence alignment on a reconfigurable system. In the first method, the matrix is filled on an FPGA and then the matrix data is sent to the GPP, where the traceback is performed. This method creates a memory bottleneck in any off-the-shelf FPGA board. In the second method, a sequence is shortlisted by finding the maximum value after performing the matrix fill stage for the whole database. Later, that maximum value and the index of the corresponding sequence is transferred to the GPP. The matrix fill stage for the shortlisted sequences is repeated on the GPP and the traceback is performed to get the optimal alignment.

In this chapter, we propose a parallel FPGA design of the SW traceback, which performs the alignment immediately after completing the matrix fill for all the sequences in the database. This way, we can avoid the second matrix fill stage for the shortlisted sequences at the expense of more area consumption. It can be easily implemented on off-the-shelf FPGA boards as it uses the BRAM and bandwidth within limits of the current FPGA boards. The main benefits of the proposed technique are as follows:

1. The proposed solution gives the alignment after scanning the database once. We show that the bandwidth requirements is within the limits of present day FPGAs.

2. The whole solution can be easily implemented as a pure FPGA based implementation without needing a GPP.

3. Our solution is generic and can be used to design hardware for any dataflow systolic array implementation.

In this chapter, we propose a hardware design for a RVEP SW [90] implementation, which has a higher bandwidth requirement than the classical dataflow implementation for the same size of matrix. However, this design can be easily adapted to address the bandwidth issue for a dataflow systolic array implementation.

The Chapter is organized as follows. The next section presents the related work. Section 7.2 gives the detailed description of the memory bandwidth bottleneck in case of SW. Then Section 7.3 discusses the proposed solution. Section 7.4 discusses the design challenges and their possible solutions. Section 7.5 discusses the different solutions and bandwidth requirements depending upon the what is needed. Finally Section 7.6 summarizes contents of the chapter.

7.1 Related work

Several people have worked on approaches using the first method. Hoang and Lopresti [53] gave a linear systolic array implementation on a SPLASH reconfigurable logic array, in which the data of the matrix fill was stored in memory and then a traceback was performed. They only used the edit distance formula, which is a special case of Smith-Waterman algorithm. This method requires substantial memory bandwidth which is not available, as will be shown in Section 7.2. Yamaguchi et al. [131] and Moritz et al. [85] implemented SW on a linear systolic array. They both applied compression and saved direction vectors of 2 bits for each element instead of 16 bits. The compression reduced the memory bandwidth requirement. However, still it was high enough to be implemented using off-the-shelf FPGA boards and became the bottleneck as described later in Section 7.2.

Most of the implementations follow the second method, in which FPGAs are only used to find the maximum value after filling the matrix [18, 33, 57, 111, 133].

Our implementation is more close to the first method. Our goal was to avoid the memory bandwidth problem such that off the shelf FPGAs can be used.

Table 7.1: Bandwidth requirement for different implementations

Implementations	p	d_w	Frequency	Time I	Bandwidth I
		bits	MHz.	sec	Gb/sec
Zhang07 [133]	384	20	66.7	1.56×10^{-4}	49.36
Oliver05 [93]	252	16	55	1.86×10^{-4}	27
Jiang07 [57]	80	20	82	1.23×10^{-4}	13.04
Nawaz10 [90]	40	16	79.3	6.33×10^{-5}	12.6

7.2 Memory Bandwidth Bottleneck

In this section, we describe the memory bandwidth bottleneck that arises when we use the first method. Here, we assume to store only the direction vector, which is of 2 bits as in [85, 131]. A double buffering technique can be used, in which one can keep two copies of the direction matrix in which one stores the data alternatingly. When the FPGA is computing the next sequence alignment and storing that result in one buffer, the other buffer can be transferring its contents to the shared memory for later traceback use by the GPP. The time to transfer the direction matrix from memory should not be more than the matrix fill time for the next pair of sequences, so that the memory bandwidth does not become the bottleneck.

In order to better quantify the memory bottleneck problem, we discuss the memory requirements for different implementations. In Table 7.1, we present the numbers for four different SW implementations. The first three are dataflow implementations and the last is the RVEP implementation. p represents the maximum number of PEs that can be accommodated on the FPGA available in the respective implementation. We take $m = 10000$, which is a higher end value for a sequence, as only 13 out of 468851 protein sequences are longer than 10000 symbols in the UniProt database and 99.5% of the sequences are less than 1000 symbols [55]. The storage size of an element in the optimal value matrix is defined as d_w. The frequency used for computing each element in case of a dataflow implementation and for computing a block in case of the RVEP implementation is given under Frequency. Time I is the time needed to fill the whole matrix for one sequence alignment by the respective implementations. Bandwidth I is computed for the transfer of the direction matrix and is computed by the following formula:

$$\text{Bandwidth I} = \frac{2nm}{\text{Time I}} \tag{7.1}$$

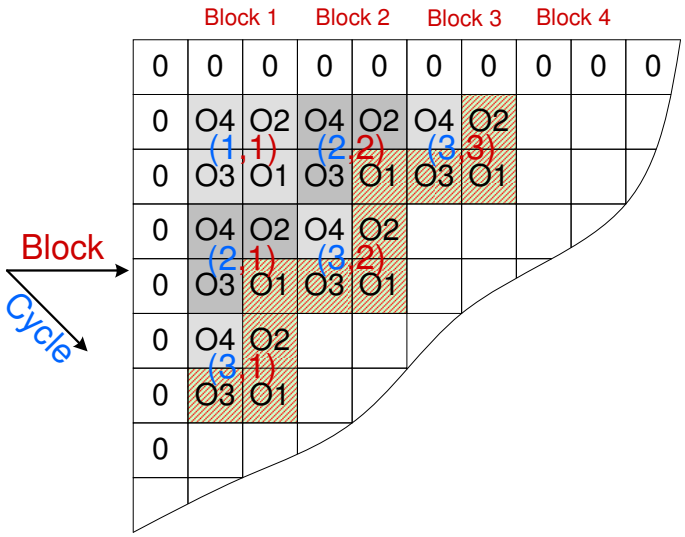


Figure 7.1: Row to store the optimal values for $B = 2$. Blocks at the top show the hardware to be used in each column. Block (u, v) specifies that v block circuit is used in u cycle to compute the optimal value.

When combining all these data, we obtain a bandwidth requirement of up to 49.36 Gb/sec which is more than what is available on even the largest FPGA's. The Xilinx-6 FPGA Connectivity Development Kit enables advanced connectivity designs with PCI Express 1.1/2.0, Ethernet, SATA and other proprietary high-speed serial protocols with line rates up to 6.5 Gb/sec [3].

7.3 Compression and Backtracking

In order to solve the bottleneck as quantified above, we propose to perform back tracking in addition to the compression. Even though this twofold solution is presented here in the context of the RVEP for Smith Waterman, it can be easily modified to be useful for any classical dataflow implementation. In this section, we are using $B = 2$,$p = 40$, $m = 10000$, $n = 1000$, and $d_w = 16$, where B is the blocking factor, p represents the maximum number of PEs that can be accommodated on the FPGA, n and m are lengths of the sequences and d_w is the data width of the optimal values in bits.

During the matrix fill, the elements in a block can be computed only from the adjacent elements in the preceding block [89]. As depicted in Figure 7.1, all the elements in the anti-diagonal of blocks in cycle 4 can be computed using

		G	T	C	G	C	A	A	C
	0	**0**	0	0	0	0	0	0	0
T	0	0	**2**	0	0	0	0	0	0
C	0	0	0	**4**	**2**	2	0	0	2
C	0	0	0	2	3	**4**	2	0	2
A	0	0	0	0	1	2	**6**	4	2
T	0	0	2	0	0	0	4	5	3
G	0	2	0	1	2	0	2	3	4

(a) Matrix for an example of SW algorithm, when $a = -2$ and $x(i,j) = +2$ when *S[i]=T[j]* otherwise -1. Elements in the trace back are shown in bold.

		G	T	C	G	C	A	A	C
	0	**0**	0	0	0	0	0	0	0
T	0	0	**2**	0	0	0	0	0	0
C	0	0	0	**2**	**1**	2	0	0	2
C	0	0	0	3	2	**2**	1	0	2
A	0	0	0	0	3	3	**2**	1	1
T	0	0	2	0	0	0	3	2	1
G	0	2	0	2	2	0	3	3	2

(b) Direction matrix for the example in Figure 7.2a

Figure 7.2: Scoring matrix and its corresponding directon matrix

the adjacent elements from the preceding anti-diagonal of blocks, as shown by the pattern filled elements. Hence to compute the optimal score for the current anti-diagonal of blocks, we only need to store the optimal score data for the adjacent elements from the preceding anti-diagonal in the BRAM using a FIFO buffer. We thus avoid to store the entire matrix of optimal values. The size of memory required to store this is $p \times 2B \times d_w = 20 \times 2 \times 2 \times 16 = 1280$ bits, where p is the number of block PEs that can be accommodated on the FPGA.

Instead of the optimal value matrix, we only store the direction matrix which contains direction vectors to construct the sequence alignment. There are only 3 directions from which an element can be computed. So similar to [85, 131], only 2 bits are needed to indicate the direction it is computed from. As described earlier in Section 6.1, the traceback is stopped beyond a threshold value. We give the direction value 0 in the direction matrix for the corresponding threshold value in an optimal value matrix. Similarly we fill a 1,2 or 3 value in the current element of the direction matrix, if the current element is computed from the left, top left or top element respectively. The maximum value 6 in the matrix from Figure 7.2a is computed from the top-left element, therefore, the corresponding element in the direction matrix contains 2 as shown in Figure 7.2b. The required storage space is determined by two factors: the first is the row to keep the optimal values given by $p \times 2B \times d_w$ and second to keep the direction matrix which is $2nm$. So the total space required in BRAM is $p \times 2B \times d_w + 2nm = 1280 + 2 \times 10^7$ bits, which is less than $d_w(nm) = 16 \times 1000 \times 10000 = 64 \times 10^7$ bits which are required without

compression. Now there are FPGAs available with 6.5×10^7 bits BRAM [1].

Since the bandwidth requirement is still high after compression, we propose to move the traceback stage to the FPGA, which further reduces the bandwidth requirement. It means the complete solution is now on the FPGA, which is easier to maintain as it is in one place. So, the traceback is performed on the direction vector in BRAM and the alignment results are sent to the main memory, which are far less than the whole direction matrix. We perform another task, which starts the traceback from the direction value in the direction matrix corresponding to the maximum value in the optimal value matrix in BRAM and transfer only direction vectors which come across the traceback path. For the example shown in Figure 7.2a, the maximum value is 6, so we go to the corresponding element in the direction matrix shown in Figure 7.2b, which is 2 and start the traceback from there. We perform the traceback according to the direction vector and get the traceback as shown in Figure 7.2b, which is the same as in Figure 7.2a. The length of the traceback path is $O(max(m, n))$, which is the worst case and usually the length of the traceback is far less than this.

7.4 Design Overview

In this section, we present the design overview for the proposed technique for an RVEP implementation of Smith Waterman with parameter $B = 2$. As explained above, the traceback is done in parallel with the matrix fill stage. This design is on top of the circuit for optimal value computation as given in Chapter 6. The proposed design is composed of computing the maximum of the values stored in the optimal value matrix, generating the corresponding direction matrix, storing the direction values in BRAM and finally doing the traceback on the direction vectors starting from the maximum optimal value. This implementation can be easily modified for RVEP with higher blocking factors. The details are discussed in the following sections.

7.4.1 Computing max in the optimal value matrix

The first step is to find the maximum value in the optimal value matrix. We do that by first finding the maximum of the block and then finding the maximum

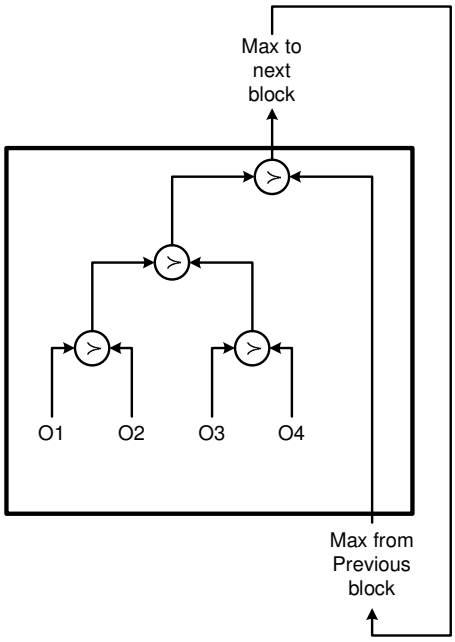
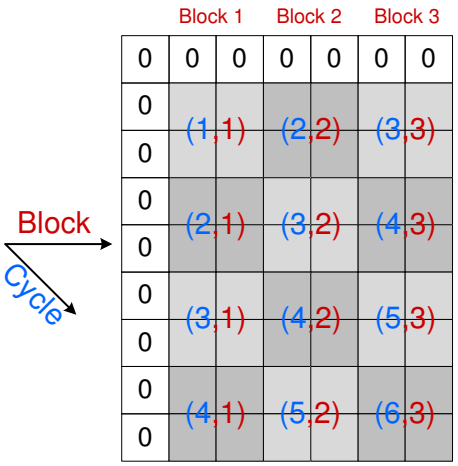


Figure 7.3: Finding max. for the block of 2×2

among all the blocks in a block column and then finding the maximum value among all the block columns to find the maximum of the matrix.

As mentioned earlier in Chapter 6, we compute the optimal values in a systolic array. After the optimal values are computed for a block, we compute the maximum of the optimal values from the previous and current blocks in the same block column by using the circuit in Figure 7.3. In the meanwhile, the optimal values for the next block in the same block column are computed systolically by the computation block. This continues until the optimal values for the column block and later in the next cycle the maximum of the whole column is computed. The resource and time used for computing the optimal value and then the maximum for the matrix shown in Figure 7.4a is given in Figure 7.4b. It shows that first the optimal value for block $(1, 1)$ i.e. $C_{1,1}$ is computed. In the next cycle, $C_{2,1}$ is computed systolically and the maximum of block $(1, 1)$, i.e. $M_{1,1}$ is computed. The maximum of block column 1 is computed in cycle 5 and similarly the maximum of block column 2 is computed in cycle 6. The maximum of block column 1 and 2 is computed in cycle 7 by using a single comparator as given by m_{1_2}. At the same cycle 7, the maximum of the column 3 is computed and then the same comparator that is used to compute m_{1_2}



(a) Computation block. A number in red represent the block used and the number in blue represents the cycle in which it is used.

Time (cycles)	Block 1			Block 2			Block 3			column max
1	$C_{1,1}$									
2	$C_{2,1}$	$M_{1,1}$	$T_{1,1}$	$C_{2,2}$						
3	$C_{3,1}$	$M_{2,1}$	$T_{2,1}$	$C_{3,2}$	$M_{2,2}$	$T_{2,2}$	$C_{3,3}$			
4	$C_{4,1}$	$M_{3,1}$	$T_{3,1}$	$C_{4,2}$	$M_{3,2}$	$T_{3,2}$	$C_{4,3}$	$M_{3,3}$	$T_{3,3}$	
5		$M_{4,1}$	$T_{4,1}$	$C_{5,2}$	$M_{4,2}$	$T_{4,2}$	$C_{5,3}$	$M_{4,3}$	$T_{4,3}$	
6					$M_{5,2}$	$T_{5,2}$	$C_{6,3}$	$M_{5,3}$	$T_{5,3}$	
7								$M_{6,3}$	$T_{6,3}$	m_{1_2}
8										m_{2_3}
9										

(b) Computation sequence for the computation block in Figure 7.4a, C stands for Computing the optimal value, M for finding the Maximum of the optimal value in a block and T for generating the direction vectors for elements in the block. Red arrows show the computation of max block and direction vector for the corresponding computation block. Blue arrows show the computation of maximum of block columns.

Figure 7.4: Computation block and the sequence to compute it

in cycle 7 is used to compute the maximum of block 3 and all block columns < 3, we call it m_{2_3} in cycle 8. This way the maximum of all elements of the optimal value matrix is computed.

Algorithm 7.1 Pseudo-code to generate the direction vector for an element (i, j)

```
if (F[i,j]=0)
   output=0
else if (F[i,j]=F[i,j-1]+a)
   output=1
else if (F[i,j]=F[i-1,j]+a)
   output=3
else
   output=2
```

7.4.2 Generating the direction matrix

The direction vectors for a direction matrix block are computed after the computation of the optimal value block. In Figure 7.4b, the direction vectors $T_{i,j}$ for any block (i, j) are computed in the next cycle to $C_{i,j}$. Similar to compute optimal values and computing a maximum for a block, direction vectors of a block are also computed systolically. The pseudo-code to generate a direction value for an element is shown in Algorithm 7.1.

7.4.3 Storing direction vectors in BRAM

After the direction vectors for a block have been generated, they are stored in BRAM as an intermediate result. All the values of the direction vectors for the same i block are generated in parallel as shown in Figure 7.5. In this section, we are looking at Figure 7.5 as an example where $n = 8$ and $B = 2$, $b = 4$ and $r = \frac{n}{B} = 4$.

Three regions can be distinguished as shown in Figure 7.5. The first one where the number of blocks needed to be written to the memory is increasing starting from 1 to $(r-1) = 3$ blocks and generating $1 \times b = 4$ addresses to $(r-1)b = 12$ addresses. The second one where the number of blocks remain constant at r blocks to generate $rb = 16$ addresses and the third one where the number of blocks again decreases from $(r-1) = 3$ to 1 block.

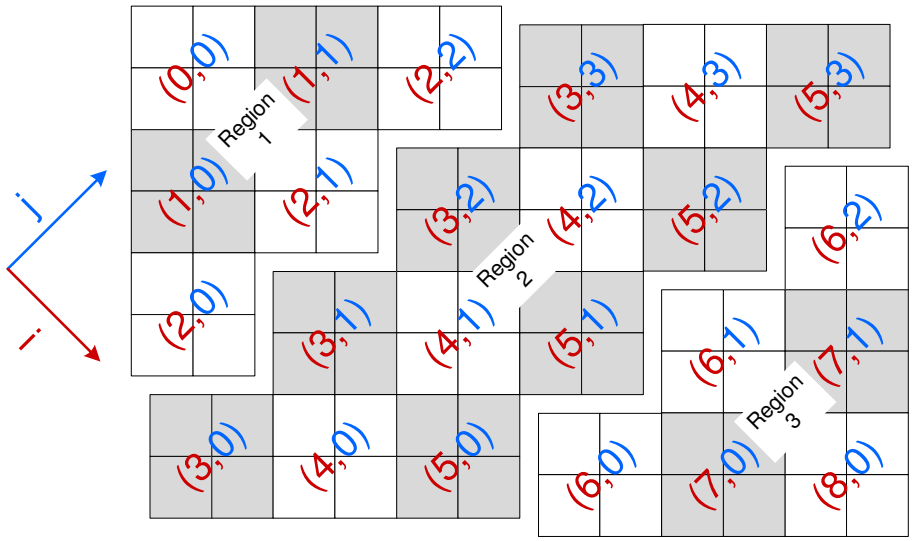


Figure 7.5: Classification of BRAM according to the way to fill the direction matrix of 8×12, with $B = 2$ and $b = 4$

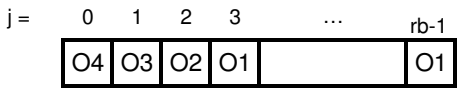


Figure 7.6: Elements stored in BRAM

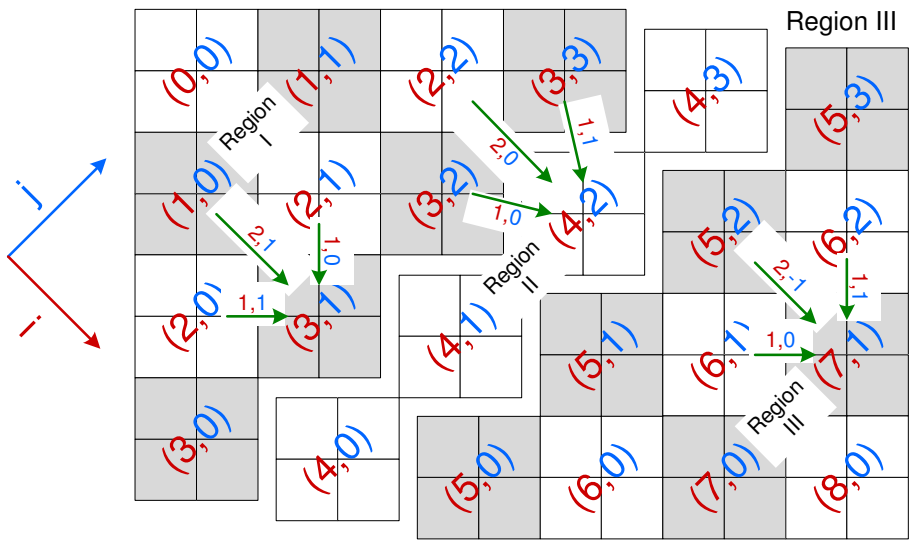


Figure 7.7: Classification of BRAM according to the direction vectors among the neighboring blocks, Region I: $i < r$; Region II: $i = r$ and Region III: $i > r$, here $r = 4$

It is interesting to choose the suitable width in BRAM to store these direction vectors. Lets look at two extremes. One possibility is to generate an address for each block and store it separately. In this case, $2b = 8$ bits are allocated to store $b = 4$ elements. Therefore, in Region 2, we need r memory ports for writing into BRAM. The second possibility is to generate one address for the whole anti-diagonal of blocks. This way $2rb = 32$ bits are allocated to store $rb = 16$ elements, and we need at most one port to write. In this case, we need to do stuffing in Region 1 and 3 and some memory will be wasted, however the advantage is that only one memory port is required.

We have chosen the second option and we made the BRAM width $= 2rb = 32$ bits. The way elements are stored is shown in Figure 7.6. This whole BRAM width is filled in one cycle using one write port. The data is always filled starting from 0.

One can further optimize the memory waste by choosing the BRAM width between these two extremes.

Algorithm 7.2 Pseudo-code for traceback

```
GetTrace(i,j,trace){
  k = j mod b
  if ( ( (i<r) and ((j>2) and
  !( (j>=i*4) and (j<i*4+3) ) ) )
  or ( (i>=r) and ((j>2) and !( (j>=r*4-4)
  and (j<r*4-1) ) ) ) and (m>0) {
   I = GetI(k,m(i,j)) // Get I from Figure 7.8
   J = GetJ(k,m(i,j)) // Get J from Figure 7.8
   trace = trace + m(i,j) // + is concatenation
   GetTrace(I,J,trace)
  } else {
  trace = trace + m(i,j)
 }
}
```

	m(i,j)		
	1	2	3
k=0	I=i-1	I=i-2	I=i-1
	J=j-2+d	J=j-1+e	J=j+1+d
k=1	I=i-1	I=i-1	I=i
	J=j-2+d	J=j-3+d	J=j-1
k=2	I=i	I=i-1	I=i-1
	J=j-2+d	J=j+1+d	J=j+1+d
k=3	I=i	I=i	I=i
	J=j-2+d	J=j-3	J=j-1

where d=0,e=0 if i<r; d=b, e=b if i=r and d=b, e=2b if i>r

Figure 7.8: BRAM Address translation

7.4.4 Traceback

The traceback is started from the point in the direction matrix corresponding to the maximum value of the optimal value matrix. The data is stored in a different coordinate system (rotated at 45 degree to horizontal) as compared to what was earlier suggested by the formula in Equation 6.2 and secondly each 2-dimensional block is linearized as shown in Figure 7.6. We need to translate the traceback formula accordingly.

There are three regions classified by the direction vectors between the blocks as shown in Figure 7.7. The traceback is possible to three neighboring blocks. The block direction vectors are different in different regions as shown in Figure 7.7. The direction vectors among the neighboring blocks remain the same in each region and hence the traceback formula too. The traceback code, which also takes care of each of these regions is given in Algorithm 7.2. The first call to the traceback code is GetTrace(i,j,ε), where ε is the empty string and (i, j) is the position of the maximum value in the optimal value matrix. The trace variable in Algorithm 7.2 will finally give the traceback path from which the alignment can be easily computed. The traceback code has a linear time complexity and takes less time than the time to fill the matrix in either the dataflow or RVEP case. The address translation is summarized in Figure 7.8. $m(i, j)$ refers to the direction value at (i, j) position in the BRAM.

7.5 Experimental Validation

In order to assess the impact of our compression scheme with traceback, we explore two extreme cases of memory transfer. The memory bandwidth required for these two cases is given in Table 7.2, which is a extension of Table 7.1. The first case, considered as the worst case is as follows. We keep two direction matrices for double buffering to do the sequence alignment continuously. When one matrix is being filled, the other can be used to transfer the traceback result of the sequence alignment done recently. We generate the direction matrix for every alignment one after the other; however, we do not transfer the traceback direction vector for every alignment. The idea is that we start the alignment, find the maximum score in the optimal value matrix of the first pair, declare it as global maximum score and send its traceback direction vector to the main memory. We proceed with the second pair, generate the direction matrix, find its maximum score, but only send its traceback direction vector when the maximum score of the second pair is larger than the global

Table 7.2: Comparison of bandwidth requirement for different implementations

Implementations	P.E.	Data width	Frequency	Time I	Time II	Bandwidth I	Bandwidth II	Bandwidth III
		bits	MHz.	sec	sec	Gb/sec	Mb/sec	Kb/sec
Zhang07 [133]	384	20	66.7	1.56×10^{-4}	2.48	49.36	128.8	8
Oliver05 [93]	252	16	55	1.86×10^{-4}	3.02	27	107.2	6.64
Jiang07 [57]	80	20	82	1.23×10^{-4}	2.02	16.3	162.4	9.92
Nawaz10 [90]	40	16	79.3	6.33×10^{-5}	1.13	12.6	316	17.6

maximum score and declare that as global maximum. We will repeat this for the rest of the known sequences in the database.

In the worst case, we need to transfer the traceback vector for every comparison when every next sequence in the database has a higher maximum optimal value than the current sequence. In that case, we need to send $O(2 \times max(m, n))$ bits for some unit time (which is equal to the time to fill the matrix, Time I in Table 7.2) instead of $O(2nm)$ bits, which is linear as compared to quadratic in normal case. The chances of this worst case are close to impossible. We have also computed the bandwidth requirement for our proposed technique keeping in mind the worst case referred to as Bandwidth II in Table 7.2. Here, the maximum bandwidth is 316 Mb/sec which is easily achievable in normal FPGA boards.

The best case is described as follows. We find the maximum value in the optimal value matrix for the first sequence alignment, find the traceback vector along with the maximum optimal value. We set the maximum value as global maximum and also save the traceback vector in BRAM. We repeat the same for the next alignment and compare the maximum value with the previously saved global maximum value. The previously saved global maximum is replaced with the maximum value of the current alignment, if the maximum value of the current sequence is larger than the global maximum. The traceback vector corresponding to the new global maximum is also saved. We repeat this for the rest of the sequences in the database. Finally, we are left with the maximum value of the sequence and its corresponding traceback vector, which is transferred to the main memory. In this scenario, the traceback path and maximum will be sent once after scanning the entire database, which contains 468851 protein sequences in case of UniProt [55]. The time it takes to scan the whole database is given under the header Time II. The memory bandwidth requirement for this implementation using Time II is shown as Bandwidth III in Table 7.2. The maximum bandwidth under heading Bandwidth III is 17.6 Kb/sec.

The implementation strategy can be easily changed, if there is a requirement for computing some k-best alignments from the database. The bandwidth for any such strategy will fall in between Bandwidth II and Bandwidth III.

Our solution is better than other approaches described in [33, 93, 111] in the sense that it gives the optimal alignment between the unknown sequence and the known sequence in the database after once scanning through the database and there is no need to repeat the sequence alignment for some smaller subset. Secondly, the whole solution is based on FPGA, so there is no need to maintain the solution at two different places, which is cheaper and at lower risk.

7.6 Summary and Conclusion

In this chapter, we have proposed a parallel FPGA design of the SW traceback phase, which constructs the optimal alignment between the unknown sequence and its closest known sequence from the database. We have shown that compression alone is not enough to address the memory bandwidth problem. We have proposed to perform a traceback on the compressed data to reduce the memory bandwidth and can be easily implemented on current off-the-shelf FPGA boards. Moreover, we have proposed a hardware design for a SW RVEP implementation that can be easily extended to any other dataflow systolic array implementation.

Note.

The content of this chapter is based on the the following paper:

Z. Nawaz, M. Nadeem, H. V. Someren, K.L.M. Bertels, *A parallel FPGA design of the Smith-Waterman traceback*, proceedings of International Conference on Field-Programmable Technology 2010,pp. 1-6, Beijing, China, December 2010.

8 Conclusions

FPGAs are increasingly becoming a choice in high-performance computing due to high degree of parallelism they provide. The traditional HDL compilers use many loop transformations to extract the required parallelism. The area on FPGA is increasing with an increase in transistor density due to the trend called Moore's law. This motivated us to investigate the methods that can utilize the extra area available on FPGA to extract more parallelism beyond the traditional loop transformations.

In this final chapter, we first summarize the work done in the earlier chapters and also present the contributions made in each of them in the next section. Finally, in Section 8.2, we propose the future research directions.

8.1 Summary and Contributions

We started by discussing a number of loop optimizations and explained the kind of dependences that limit the parallel execution in Chapter 2.

The goal of loop parallelization techniques is to change the sequence to execute the instructions, so that the dependences are not violated and more computations can be performed in parallel. In Chapter 3, we introduced a transformation called Recursive Variable Expansion (RVE) as a more powerful loop transformation suitable for a large class of problems. RVE removes all the data dependences in an expression by a backward substitution, which results in expansion. We identified two types of recurrences namely polynomially expanding and the other exponentially expanding. We showed analytically by an example that our transformation achieved more acceleration than the traditional approaches like loop skewing. We applied RVE on four kernels and then compared the obtained results with a GPP implementation. It showed

speedups to 77x as compared to the GPP implementation. The basic version of RVE assumes unlimited area, an assumption which was relaxed later.

The contribution made in Chapter 3 can be summarized as

- We proposed a transformation called Recursive Variable Expansion. When applied to a certain class of problems, all the data dependences are removed and a high degree of parallelism can be achieved.

In Chapter 4, we imposed the constraints like the limited available area and usable memory bandwidth. We tried to build a pipelined hardware design for the problems that expand polynomially when RVE is applied. We used the suffix tree data structure to find repeating patterns in the generic version of the expanded expression. The largest repeating pattern that satisfies the area and memory constraints determines the pipeline circuit. Furthermore, without compromising performance, we optimized the area of the pipelined circuit.

The contribution made in Chapter 4 can be summarized as :

- We presented an automatic pipelining design algorithm for polynomially expanding expression that defines an optimal pipeline which satisfies both the area and memory bandwidth constraints. We applied this algorithm on the 2D DCT kernel, which showed that it produces a pipeline circuit with comparable performance to the hand optimized pipeline circuit for 2D DCT kernel provided by Xilinx.

Chapter 5 dealt with the dynamic programming (DP) problems, which are characterized by an exponential expansion when RVE is applied. We proposed a hybrid approach that mixed the RVE with the dataflow to limit the growth. We chose four representative DP problems, and applied RVE to them. Finally, we generalized our approach to show that it can be applied to a large class of DP problems, which have a constant number of dependences.

The contribution made in Chapter 5 can be summarized as :

- We proposed a generic framework and two variants of the RVE based algorithms (RVENP and RVEP) that can accelerate a large number of DP problems more than a pure dataflow approach.

In Chapter 6, we provided an in-depth discussion of an important bioinformatics algorithm, Smith-Waterman (SW). Smith-Waterman is also a DP problem. The fact which makes it different from most of the other DP problems is that

the SW formula contains a clipping factor which does not allow negative values. However, when RVE based algorithms devised in Chapter 5 are applied, they exclude the clipping factor for the intermediate vertices. As a consequence, RVE cannot be applied as such but needs modifications to compensate for this error.

The contributions in Chapter 6 can be summarized as :

- We applied RVEP and RVENP on SW algorithm and showed that our solution outperforms the dataflow approach.
- We proposed a generic algorithm that defines a patch to extend the RVE formulations and takes care of the clipping factor on a RVE block for any blocking factor.

Finally, we further optimize our RVE and dataflow implementation of SW by performing the traceback at the same time when performing the matrix fill. This requires a higher bandwidth, which is not available on off-the-shelf FPGA boards. To this purpose, we have developed a compression scheme to address the bandwidth constraints and storage needs.

The contributions in Chapter 7 can be summarized as :

- We proposed a new pipelined SW traceback FPGA design that computes the alignment immediately after scanning the database.
- We resolved the memory bandwidth bottleneck problem that arises due to the change in the design from serial to parallel traceback.

8.2 Future Directions

In Chapter 3, we proposed our transformation Recursive Variable Expansion (RVE). We applied this transformation to kernels which have certain constraints, which also limits the type of kernels on which it can be applied. However, RVE can be applied to more kernels if we relax some of these constraints. One of the constraints is that we apply RVE to a loop body which is free from any conditional statement. Another assumption made for the application of RVE is that the bounds of the loops must be known at compile time. In future, we plan to relax these constraints by using techniques developed by Ghuloum et. al [39, 43] and Schlansker et. al [101]. This way RVE can be applied to more kernels.

In Chapter 4, we proposed a pipeling design algorithm for the polynomially expanding expression that chooses an optimal pipeline that fits both the area and memory bandwidth requirement. However, our approach was only applicable to equal sized generic expressions. We can relax this constraint by using loop tiling with RVE expansion of some constant number of steps, where loop tiling will iterate between the RVE blocks.

In Chapter 5 and 6, we proposed an algorithm that dealt with a large class of dynamic programming problems. We applied our algorithm semi-automatically on a number of such problems. A fully automatic implementation of the algorithm that becomes part of the compiler would be a useful enhancement. This way, more dynamic programming problems that satisfy the given constraints can be accelerated. In the current work, we have kept the blocking factor to at most 3, because the problem size becomes so large to be handled manually. When fully automated, we could investigate the speedups for large block sizes. Another intriguing extension can be to use rectangular blocks instead of square blocks to investigate the performance gains at the cost of consumed area.

As said earlier in Chapter 1, we plan to make RVE as part of the DWARV compiler, as we can identify the loops and expressions, on which it can be applied efficiently.

In Chapter 7, we have proposed a parallel FPGA design for Smith-Waterman traceback stage. We plan to implement that design on FPGA to see the performance benefit that can be achieved at the cost of consumed area. A complete solution of the sequence alignment problem, that takes inputs from the user on a computer, computes the alignment with all the sequences in the database, and then return the final alignment result at the end is being done on Convey supercomputer.

Bibliography

[1] 7 Series FPGAs Overview. online: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[2] hArtes. online: http://www.hartes.org/.

[3] New Xilinx Virtex-6 and Spartan-6 FPGA Connectivity Development Kits. online: www.xilinx.com/products/devkits/EK-V6-ML605-G.htm.

[4] PSIM. online: http://sourceware.org/psim/.

[5] Virtex-4 family overview. online: http://www.xilinx.com/bvdocs/publications/ds112.pdf.

[6] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Transactions on Computers*, 30(5):341–356, 1981.

[7] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[8] John R. Allen and Ken Kennedy. Automatic loop interchange. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 233–246, New York, NY, USA, 1984. ACM.

[9] John Randal Allen. *Dependence analysis for subscripted variables and its application to program transformations*. PhD thesis, Houston, TX, USA, 1983.

[10] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, pages 403–410, 1990.

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[12] J. L. BAER and D. P. BOVET. Compilation of arithmetic expressions for parallel computations. In *Proceedings of the IFIP Congress*, pages 340–346. North-Holland Pub. Co., Amsterdam, 1968.

[13] U. Banerjee. Unimodular Transformations of Double Loops. *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, 1991.

[14] U. Banerjee, Shyh-Ching Chen, D. J. Kuck, and R. A. Towle. Time and Parallel Processor Bounds for Fortran-Like Loops. *IEEE Transactions on Computers*, 28(9):660–670, 1979.

[15] Nastaran Baradaran and Pedro C. Diniz. A compiler approach to managing storage and memory bandwidth in configurable architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(4):1–26, 2008.

[16] James C. Beatty. An axiomatic approach to code optimization for expressions. *Journal of the ACM (JACM)*, 19(4):613–640, 1972.

[17] R. Bellman and S.E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, NJ, 1962.

[18] K. Benkrid, Ying Liu, and A. Benkrid. A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(4):561 –570, apr. 2009.

[19] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, Barbara A. Rapp, and David L. Wheeler. GenBank. *Nucleic Acids Research*, 28(1):15–18, 2000.

[20] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1999.

[21] M.J. Borah, M. Bajwa, R.S. Hannenhalli, and S. Irwin. A SIMD solution to the sequence comparison problem on the MGAP. In *Proceedings of International Conference on Application Specific Array Processors*, 1994.

[22] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[23] Richard P. Brent. The Parallel Evaluation of Arithmetic Expressions in Logarithmic Time. In *Complexity of Sequential and Parallel Numerical Algorithms*, pages 83–102. Academic Press, 1973.

[24] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.

[25] B. L. Buzbee, G. H. Golub, and C. W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM Journal on Numerical Analysis*, 7(4):627–656, 1970.

[26] Timothy J. Callahan and John Wawrzynek. Adapting software pipelining for reconfigurable computing. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–64, New York, NY, USA, 2000. ACM.

[27] João M.P. Cardoso and Pedro C. Diniz. Modeling Loop Unrolling: Approaches and Open Issues. In *Proceedings of the 4th International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS '04)*, 2004.

[28] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(4):461–491, 1986.

[29] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical Parallel Band Triangular System Solvers. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):270–277, 1978.

[30] Shyh-Ching Chen and D. J. Kuck. Time and Parallel Processor Bounds for Linear Recurrence Systems. *IEEE Transactions on Computers*, 24(7):701–717, 1975.

[31] M O Dayhoff. Survey of new data and computer methods of analysis. In *Atlas of Protein Sequence and Structure*, volume 5, page 29, 1978.

[32] Andrea Di Blas, David M. Dahle, Mark Diekhans, Leslie Grate, Jeffrey Hirschberg, Kevin Karplus, Hansjorg Keller, Mark Kendrick, Francisco J. Mesa-Martinez, David Pease, Eric Rice, Angela Schultz, Don Speck, and Richard Hughey. The UCSC Kestrel Parallel Processor. *IEEE Transactions on Parallel and Distributed Systems*, 16:80–92, January 2005.

[33] Philippe Faes, Bram Minnaert, Mark Christiaens, Eric Bonnet, Yvan Saeys, Dirk Stroobandt, and Yves Van de Peer. Scalable hardware accelerator for comparing DNA and protein sequences. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 33, New York, NY, USA, 2006. ACM.

[34] Michael Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

[35] MS Farrar. Optimizing Smith-Waterman for the Cell Broadband Engine. http://farrar.michael.googlepages.com/SW-CellBE.pdf.

[36] G. Fettweis and H. Meyr. Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck. *IEEE Transactions on Communications*, 37:785 – 790, 1989.

[37] G. Fettweis, L. Thiele, and G. Meyr. Algorithm transformations for unlimited parallelism. *IEEE International Symposium on Circuits and Systems*, pages 1756–1759 vol.3, May 1990.

[38] G.P. Fettweis and L. Thiele. Algebraic recurrence transformations for massive parallelism. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 40(12):949–952, Dec 1993.

[39] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. *ACM SIGPLAN Notices*, 29:135–146, June 1994.

[40] D. D. Gajski. An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines. *IEEE Transactions on Computers*, 30(3):190–206, 1981.

[41] Michael Y. Galperin. The Molecular Biology Database Collection: 2007 update. *Nucleic Acids Research*, 35:D3–D4(1), January 2007.

[42] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 229–254, London, UK, 1988. Springer-Verlag.

[43] Anwar M. Ghuloum and Allan L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. *ACM SIGPLAN Notices*, 30:58–67, August 1995.

[44] Maya B. Gokhale, Janice M. Stone, and Edson Gomersall. Co-Synthesis to a Hybrid RISC/FPGA Architecture. *Journal of VLSI Signal Processing Systems*, 24(2-3):165–180, 2000.

[45] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, December 1982.

[46] A.C. Greenberg, R.E. Ladner, M.S. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Information Processing letters*, 5:31–35, 1982.

[47] C. Guerra and R. Melhem. Synthesizing non-uniform systolic designs. In *International Conference on Parallel Processing*, 1986.

[48] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[49] L. Hasan and Z. Al-Ars. An Efficient and High Performance Linear Recursive Variable Expansion Implementation of the Smith-Waterman Algorithm. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 3845–3848, September 2009.

[50] L. Hasan, Z. Al-Ars, Z. Nawaz, and K.L.M. Bertels. Hardware Implementation of the Smith-Waterman Algorithm Using Recursive Variable Expansion. In *Proceedings of 3rd International Design and Test Workshop IDT08*, December 2008.

[51] L Hasan, Z. Al-Ars, and S. Vassiliadis. Hardware Acceleration of Sequence Alignment Algorithms - An Overview. In *Proceedings of Inernational Conference on Design & Technology of Integrated Systems in Nanoscale Era*, pages 96–101, September 2007.

[52] S Henikoff and JG Henikoff. Amino Acid Substitution Matrices from Protein Blocks. In *Proceedings of the National Academy of Sciences*, volume 89, pages 10915–10919, 1992.

[53] Dzung T. Hoang and Daniel P. Lopresti. FPGA Implementation of Systolic Sequence Alignment. In *International Workshop on Field Programmable Logic and Applications*, 1992.

[54] R. W. Hockney. A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. *Journal of the ACM (JACM)*, 12(1):95–113, 1965.

[55] E. J. Houtgast. Scalability of Bioinformatics Applications for Multicore Architectures. Master's thesis, T U Delft, 2009.

[56] L. Hyafil and H. T. Kung. The complexity of parallel evaluation of linear recurrence. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 12–22, New York, NY, USA, 1975. ACM.

[57] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun. A Reconfigurable Accelerator for Smith-Waterman Algorithm. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(12):1077–1081, Dec. 2007.

[58] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT press, 2004.

[59] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[60] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[61] Ken Kennedy and Kathryn S. Mckinley. Typed fusion with applications to parallel and sequential code generation. Technical report, Rice University, 1993.

[62] Donald E. Knuth, Jr. James H. Morris, , and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[63] P M Kogge and H S Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22:786–793, 1973.

[64] G. P. Kozhevnikova and A. K. Sinitskii. Tree transformation problem in microparallelism algorithms. *Cybernetics and Systems Analysis*, 19(5):604–614, September 1983.

[65] Paul William Kraska. *Parallelism exploitation and scheduling*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1972.

[66] D. J. Kuck and K. Maruyama. Time Bounds on the Parallel Evaluation of Arithmetic Expressions. *SIAM Journal on Computing*, 4(2):147–162, 1975.

[67] David J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29–59, 1977.

[68] David J. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc. New York, NY, USA, 1978.

[69] D.J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. *Transactions on Computers*, C-21:1293– 1310, 1972.

[70] Muraoka Y. Kuck, D.J. Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity. *Acta Informatica*, 3:203–216, September 1974.

[71] H. T. Kung. New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences. *Journal of the ACM (JACM)*, 23(2):252–261, 1976.

[72] G.K. Kuzmanov and S. Vassiliadis. Arbitrating Instructions in an $\rho\mu$-coded CCM. In *Proceedings of the 13th International Conference on FPL'03*, pages 81–90, September 2003.

[73] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[74] S. Lakshmivarahan and Sudarshan K. Dhall. New Parallel Algorithms for Solving First-Order and Certain Classes of Second-Order Linear Recurrences. In *Proceedings of ICPP*, pages 843–845, 1985.

[75] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM.

[76] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974.

[77] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[78] Hsien-Yu Liao, Meng-Lai Yin, and Yi Cheng. A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. In *26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 2, pages 2817 –2820, 2004.

[79] D. J. Lipman and W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol.*, 183:63–98, 1990.

[80] B. Louka and M. Tchuente. Dynamic programming on two-dimensional systolic arrays. *Information Processing Letters*, 29(2):97–104, 1988.

[81] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of ACM (JACM)*, 24(1):121–145, 1977.

[82] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of Loops for Parallelism and Locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[83] Kathryn Mckinley, Ken Kennedy, Ken Kennedy, and Kathryn S. M C Kinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, 1994.

[84] Xiangzhen Qiao Mi Lu and Guanrong Chen. A parallel algorithm for evaluating general linear recurrence equations. *Circuits, Systems, and Signal Processing*, 15:481–504, 1994.

[85] Guilherme L. Moritz, Cristiano Jory, Heitor S. Lopes, and Carlos R. Erig Lima. Implementation of a Parallel Algorithm for Protein Pairwise Alignment Using Reconfigurable Computing. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pages 1 –7, Sep. 2006.

[86] Yoichi Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1971.

[87] Jean Myoupo. Mapping dynamic programming onto modular linear systolic arrays. *Distributed Computing*, 6:165–179, 1993. 10.1007/BF02242705.

[88] Z. Nawaz, M. Shabbir, Z. Al-Ars, and K.L.M. Bertels. Acceleration of Smith-Waterman Using Recursive Variable Expansion. In *11th Euromicro Conference on Digital System Design (DSD-2008)*, pages 915–922, September 2008.

[89] Z. Nawaz, T. P. Stefanov, and K.L.M. Bertels. Efficient hardware generation for dynamic programming problems. In *International Conference on Field-Programmable Technology*, December 2009.

[90] Z. Nawaz, H. Sumbul, and K.L.M. Bertels. Fast Smith-Waterman hardware implementation. In *International Parallel and Distributed Processing Symposium*, April 2010.

[91] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[92] Alexandru Nicolau and Roni Potasmann. Incremental tree height reduction for high level synthesis. In *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 770–774, New York, NY, USA, 1991. ACM.

[93] T.F. Oliver, B. Schmidt, and D.L. Maskell. Reconfigurable architectures for bio-sequence database scanning on FPGAs. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52(12):851–855, Dec. 2005.

[94] K.K. Parhi. Look-ahead in dynamic programming and quantizer loops. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 1382–1387, May 1989.

[95] K.K. Parhi. Pipelining in dynamic programming architectures. *IEEE Transactions on Signal Processing*, 39(6):1442–1450, Jun 1991.

[96] Darin Petkov, Randolph Harr, and Saman Amarasinghe. Efficient Pipelining of Nested Loops: Unroll-and-Squash. In *Proceeding of the International Parallel and Distributed Processing symposium*, 2002.

[97] V. K. Prasanna Kumar and Yu-Chen Tsai. Mapping dynamic programming onto a linear systolic array. *The Journal of VLSI Signal Processing*, 1:335–343, 1990. 10.1007/BF00929926.

[98] B.R. Rau. Cydra 5 directed dataflow architecture. In *COMPCON Spring 88 33rd IEEE Computer Society International Conference*, pages 106 – 113, 1988.

[99] Rui Rodrigues, Joao M. P. Cardoso, and Pedro C. Diniz. A Data-Driven Approach for Pipelining Sequences of Data-Dependent Loops. In *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228, Washington, DC, USA, 2007. IEEE Computer Society.

[100] Ahmed H. Sameh and Richard P. Brent. Solving Triangular Systems on a Parallel Computer. *SIAM Journal on Numerical Analysis*, 14(6):1101–1113, 1977.

[101] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, pages 40 – 51, 1994.

[102] Michael S. Schlansker and Vinod Kathail. Acceleration of First and Higher Order Recurrences on Processors with Instruction Level Parallelism. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–429, London, UK, 1994. Springer-Verlag.

[103] Bertil Schmidt, Heiko Schröder, and Manfred Schimmler. Massively Parallel Solutions for Molecular Sequence Analysis. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2002. IEEE Computer Society.

[104] Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. In *ASAP '00: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Washington, DC, USA, 2000. IEEE Computer Society.

[105] Sharad K. Singhai, Kathryn, and S. McKinley. A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality . *The Computer Journal*, 40, 1997.

[106] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[107] Greg Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 177–186, New York, NY, USA, 2002. ACM.

[108] D.J. States, W. Gish, and S.F. Altschul. Improved Sensitivity of Nucleic Acid Database Search Using Application-Specific Scoring Matrices. In *Methods: A companion to Methods in Enzymology*, volume 3, pages 66 – 77, 1991.

[109] Harold S. Stone. Parallel Tridiagonal Equation Solvers. *ACM Transactions on Mathematical Software (TOMS)*, 1(4):289–307, 1975.

[110] Olaf Storaasli and Dave Strenski. Experiences on 64 and 150 FPGA Systems. In *Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute (RSSI'08)*, 2008.

[111] Olaf O. Storaasli and Dave Strenski. Exploring Accelerating Science Applications with FPGAs. In *Proceedings of the Third Annual Reconfigurable Systems Summer Institute (RSSI'07)*, 2007.

[112] Henry Styles, David Barrie Thomas, and Wayne Luk. Pipelining Design with Loop Carried Dependencies. In *International Conference on Field-Programmable Technology*, 2004.

[113] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, 1:107, 2008.

[114] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, McGraw Hill, second edition, 2001.

[115] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[116] H. A. van der Vorst and K. Dekker. Vectorization of linear recurrence relations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):27–35, 1989.

[117] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The Molen Polymorphic Processor. *IEEE Transactions on Computers*, 53(11):1363– 1375, November 2004.

[118] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4), 1986.

[119] Benjamin W. Wah and Guo jie Li. Systolic processing for dynamic programming problems. *Circuits, Systems, and Signal Processing*, 7(2):119–149, June 1988.

[120] Benjamin W. Wah and Guo-jie Li. Systolic processing for dynamic programming problems. *Circuits, Systems, and Signal Processing*, 7:119–149, 1988. 10.1007/BF01602094.

[121] H. H. Wang. A Parallel Method for Tridiagonal Equations. *ACM Transactions on Mathematical Software (TOMS)*, 7(2):170–183, 1981.

[122] Dorothy Wedel. Fortran for the Texas Instruments ASC system. In *Proceedings of the conference on Programming languages and compilers for parallel and vector machines*, pages 119–132, New York, NY, USA, 1975. ACM.

[123] M. Weinhardt and W. Luk. Pipeline vectorization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(2):234–248, Feb 2001.

[124] Michael Weiss. Strip mining on SIMD architectures. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 234–243, New York, NY, USA, 1991. ACM.

[125] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[126] M. J. Wolfe. Loop Skewing: The Wavefront Method Revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[127] Michael Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[128] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press Cambridge, MA, USA, March 1989.

[129] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[130] Jingling Xue. Enabling Loop Fusion and Tiling for Cache Performance by Fixing Fusion-Preventing Data Dependences. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 107–115, Washington, DC, USA, 2005. IEEE Computer Society.

[131] Yoshiki Yamaguchi, Yosuke Miyajima, Tsutomu Maruyama, and Akihiko Konagaya. High Speed Homology Search Using Run-Time Reconfiguration. In *12th International Conference on Field-Programmable Logic and Applications*, pages 281–291, London, UK, 2002. Springer-Verlag.

[132] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong. A Smith-Waterman Systolic Cell. In *FPL'03*, 2003.

[133] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *HPRCTA '07: Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications*, pages 39–48, New York, NY, USA, 2007. ACM.

[134] Heidi E. Ziegler, Mary W. Hall, and Pedro C. Diniz. Compiler-Generated Communication for Piplined FPGA Applications. In *DAC2003*, 2003.

List of Publications

International Conferences

1. **Z. Nawaz,** M. Nadeem, H. V. Someren, K.L.M. Bertels, *A parallel FPGA design of the Smith-Waterman traceback*, proceedings of International Conference on Field-Programmable Technology 2010,pp. 1-6, Beijing, China, December 2010.

2. **Z. Nawaz,** H. Sumbul, K.L.M. Bertels, *Fast Smith-Waterman hardware implementation*, International Parallel and Distributed Processing Symposium, pp. 1-4, Atlanta, USA, April 2010.

3. **Z. Nawaz,** T. P. Stefanov, K.L.M. Bertels, *Efficient hardware generation for dynamic programming problems*, proceedings of International Conference on Field-Programmable Technology 2009, pp. 348-352, Sydney, Australia, December 2009.

4. **Z. Nawaz,** T. Marconi, T. P. Stefanov, K.L.M. Bertels, *Flexible Pipelining Design for Recursive Variable Expansion*, International Parallel and Distributed Processing Symposium, pp. 915-922, Rome, Italy, May 2009.

5. L Hasan, Z. Al-Ars, **Z. Nawaz,** K.L.M. Bertels, *Hardware Implementation of the Smith-Waterman Algorithm Using Recursive Variable Expansion*, Proceedings of 3rd International Design and Test Workshop IDT08, pp. 135-140, Monastir, Tunisia, December 2008.

6. **Z. Nawaz,** M. Shabbir, Z. Al-Ars, K.L.M. Bertels, *Acceleration of Smith-Waterman Using Recursive Variable Expansion*, 11th Euromicro Conference on Digital System Design (DSD-2008), pp. 915-922, Parma, Italy, September 2008.

7. **Z. Nawaz**, O.S. Dragomir, T. Marconi, E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, *Recursive Variable Expansion: A Loop Transformation for Reconfigurable Systems*, proceedings of International Conference on Field-Programmable Technology 2007, pp. 301-304, Kokurakita, Kitakyushu, Japan, December 2007.

Local Conferences

1. **Z. Nawaz,** T. Marconi, T. P. Stefanov, K.L.M. Bertels, *Optimal pipeline design for Recursive Variable Expansion*, ACACES 2009, pp. 85-88, Terrassa, Spain, July 2009.

2. L Hasan, Z. Al-Ars, **Z. Nawaz,** *A Novel Approach for Accelerating the Smith-Waterman Algorithm using Recursive Variable Expansion*, Annual Workshop on Circuits, Systems and Signal Processing (ProRISC 2008), pp. 40-45, Veldhoven, The Netherlands, November 2008.

3. **Z. Nawaz,** M. Shabbir, Z. Al-Ars, K.L.M. Bertels, *Acceleration of Biological Sequence Alignment using Recursive Variable Expansion*, ProRISC07, Veldhoven, The Netherlands, December 2007.

Non-related Publications

1. T. Abdullah, K.L.M. Bertels, L.O. Alima, **Z. Nawaz**, *Effect of the Degree of Neighborhood on Resource Discovery in Ad Hoc Grids*, Proceedings of the International conference on Architecture of Computing Systems (ARCS), Hannover, Germany, February 2010

Samenvatting

RECONFIGURABLE Computing, waarin een general purpose processor (GPP) is uitgebreid met een of meerdere FPGAs, wordt steeds meer gebruikt voor high performance computing, waar op grote schaal fijnmazig parallellisme en pipelining toegepast kan worden. Het is een uitdaging om zulk grootschalig parallellisme op FPGAs aan te wenden en, specifieker, om een applicatie op het onderliggende heterogene platform te verdelen.

Vergelijkbaar met software compilers kunnen hardware compilers loops gebruiken om zulk parallellisme uit te buiten. Het bestaan van afhankelijkheden tussen verschillende data is een van de beperkingen die het parallellisme in een programma begrenst. In deze dissertatie stellen we een transformatie genaamd Recursive Variable Expansion (RVE) voor, welke toegepast kan worden op een belangrijke categorie van loops. Het verwijdert alle data afhankelijkheden door de variabele met zijn afhankelijkheidsexpressie uit te breiden totdat de expressie slechts een uitdrukking is in gekende variabelen. We onderscheiden twee typen expressies: één welke polynomiaal groeit, en een andere welke exponentieel groeit in termen van het aantal invoer-variabelen. Ongeacht het type van de expressie, wanneer we een expressie toewijzen aan een FPGA, dan is de benodigde oppervlakte (LUTs) op de FPGA evenredig aan het aantal termen in de expressie.

We presenteren een geautomatiseerd pipeline ontwerp-algoritme voor de vraagstukken die polynomiaal groeien. Dit algoritme bepaalt de grootste pipline-grootte dat op de FPGA past. Bovendien verzekert het algoritme ook dat er minder tijd nodig is om de data toe te voeren dan om een instructie door de pipeline te laten verwerken. We passen dit algoritme toe op DCT, een op grote schaal gebruikte kernel voor signaalverwerking, die vergelijkbare prestaties vertoont met de handgeoptimaliseerde implementatie.

De exponentieel expanderende versie is toepasbaar op de categorie van dynamisch programmeerbare (DP) problemen, waarvoor RVE gecombineerd word met dataflow. We tonen aan dat RVE betere prestaties levert dan alleen de toepassing van dataflow, terwijl deze tot dusver de beste techniek is voor dergelijke problemen. We generaliseren onze benadering door een raamwerk voor te stellen waarin ze toegepast kan worden op een breed scala aan DP problemen.

Ten slotte valideren we het voorgestelde DP-raamwerk met het Smith-Waterman algoritme. Dit algoritme is een veelgebruikte, rekenintensieve en dataintensieve applicatie in de bio-informatica. We tonen dat onze implemen-

tatie een snelheidswinst van 229% behaalt ten koste van 282% van de oorspronkelijke oppervlakte in verhouding tot de conventionele dataflow systolic array implementatie. Daarenboven, stellen wij een parallel FPGA-ontwerp voor, voor het SW traceback stadium, waarvoor de benodigde bandbreedte ook voldoende binnen de perken van de huidige beschikbare FPGA-borden.

Acknowledgments

In the name of Allah the most gracious, the most merciful

First and foremost, I thank Allah SWT for endowing me with health, patience and the knowledge to complete this thesis.

This dissertation would not have been possible without the help of many people who helped me in making it a reality. I owe my deepest gratitude to Prof. Stamatis Vassiliadis, who accepted me as a PhD student in his group. He shared the basic idea of RVE with me and asked me to explore it further in my PhD. I am especially grateful to my supervisor Koen Bertels, who was supportive of my ideas and gave me the freedom to decide about the next levels of research. His help and support was always there for technical as well as personal problems. I really enjoyed working with him. I am thankful to Zaid al-Ars for encouraging me to continue working on Smith-Waterman problem. I am grateful to Todor Stefanov, with whom I have worked on two papers. He critically reviewed my work and gave me helpful suggestion to improve it. I am fortunate to work with Hans van Someren, who has been always generous with his time to discuss my ideas, provided positive feedback and also gave me the insight of the working compilers. I am thankful to Prof. Henk Sips for acting as a promotor. I would like to extend my gratitude to the PhD examination committee, especially to Pedro Diniz, whose detailed comments helped to improve the dissertation.

I am also fortunate to work with many other wonderful people. I would like to thank especially Elena, Ozana, Thomas, Mudassir, Laiq, Ekin and Nadeem, who helped me in implementing the algorithms. I extend my thanks to Carlo, Yana, Vlad, Kamana, Mojtaba, Arash and Roel from the Delft Workbench team. It has been nice working with them. I express my sincere thanks to Jae, Sandra, Tariq, Laiq, Naeem, Faisal, Fakhar, Aqeel and Nadeem for providing me a good company over these years and having long non-technical discussions. I am thankful to Roel Meeuws for translating synopsis and propositions into Dutch and Tariq Abdullah for proof-reading my thesis. My thanks also go to Lidwina and Monique for their administrative assistance and to Bert, Eric and Eef for their technical support through out these years.

I have been very lucky to have a good social life in Delft through the acquaintance of many Pakistani friends and their families. We used to have regular family gathering during the weekends and on other social events, which my family and I enjoyed a lot.

I would like to thank Sarmad Abbasi, who introduced me with theoretical computer science. He urged (induced) me to think many fundamental questions about computer science. He was always available to discuss the theoretical problems. He motivated me to pursue PhD, which was not in my plan earlier.

I am highly indebted to my parents, who did a lot for their children. They provided me with the best possible education that I could think of. This would not have been possible without their prayers, blessings and sacrifices. I am also thankful to my younger siblings Khurram, Bilal and Annie for their love and support.

I am really grateful to my wife for her love and support over the years. This PhD would not have been possible, if she had not relieved me of the household activities. She almost single handedly managed our children Dayyan and the newborn Rayyan. Finally, Dayyan deserves special thanks for his patience and understanding during the times when I used to work till late in the office.

Zubair Nawaz
Delft, The Netherlands, 2011

Curriculum Vitae

Zubair Nawaz was born on February 15, 1973 in Lahore, Pakistan. He did his BS Mechanical Engineering from Ghulam Ishaq Khan (GIK) Institute of Engineering Sciences and Technology in 1997. From 1997 to 1999, he worked as piping design engineer at DESCON Engineering Limited Lahore. He graduated with MS in Computer Science from Lahore University of Management Sciences (LUMS) in 2002. Later, he joined Punjab University College of Information Technology (PUCIT) at University of the Punjab, Lahore as a faculty member in 2002, where he taught undergraduate and graduate level Computer Science courses.

In November 2005, he joined Computer Engineering Lab, in EEMCS faculty at Delft University of Technology for pursuing his PhD. He worked on loop transformations for reconfigurable computing under the supervision of Prof. Stamatis Vassiliadis and Koen Bertels. The research conducted by him is presented in this thesis.