High Level Design For High Speed FPGA Devices

Man. Ng     mcn99
Department of Computing
Imperial College

June 13, 2002
Acknowledgement

Before starting the report, I would like to thank the following people for helping me throughout the project. Without their help, it would be impossible for me to finish the project:

I would like to thank my supervisor Dr. Wayne Luk for giving me a lot of useful advices and encouragement throughout the project. He also guided me towards the problems I should focus on during the implementation.

I would like to thank Professor Yang for letting me to implement his gel image processing algorithm on hardware. He also gave me references and example sources to understand the theories underneath. And I would like to thank for his teaching in his excellent multimedia course. The course conveyed many useful concepts for me to understand the gel image processing

I would like to thank Altaf and Shay. They are two Ph.D research students who helped me a lot throughout the implementation of the application.
Abstract

In the project, I have discovered a systematic approach for high-level hardware design. With this approach, I successfully implemented the sophisticated gel image processing on high speed hardware. In the report, I will also introduced a new technique which can automate the process of high-level hardware performance optimization by rearranging the code sequence so that the it can be run at minimum number of clock cycles. The report will be split into 4 Chapters:

Chapter 1 is Introduction. It includes the background, all the related works and my contribution to the project.

Chapter 2 is Optimization. In this chapter, I will focus on the techniques for optimization. I will also demonstrate some techniques which can automate the optimization process.

Chapter 3 is Hardware Development. In this chapter, I will generalize the steps of converting a software programme into hardware. These include several techniques which can improve the performance or save the hardware resources.

Chapter 3 is Case Study : Gel Image Processing. In this chapter, I will use gel image processing as an example to show the effect on resource and performance of the techniques discussed in chapter 2. In this chapter, I will also compare the performance of the application between two devices and the software version: Pilchard and RC1000.

Chapter 5 is Conclusion. It includes the assessed achievements and expected future works.

There is also an online version available for this report, the URL is:

http://www.doc.ic.ac.uk/~mcn99/project/report.pdf
# Contents

Acknowledgement ......................................................... i
Abstract ................................................................................ ii

## 1 Introduction .................................................................. 1
  1.1 Background and Related Works .................................... 1
    1.1.1 Field Programmable Gate Arrays (FPGAs) [1] .......... 1
    1.1.2 Pilchard [2] ..................................................... 2
    1.1.3 RC1000 [3] .................................................. 2
    1.1.4 VHDL [4] ..................................................... 2
    1.1.5 Handel-C [5] ................................................ 3
    1.1.6 Extending the Handel-C language [7] ................. 3
  1.2 Contribution ............................................................ 3

## 2 Optimization ............................................................ 5
  2.1 Performance Optimization .......................................... 5
    2.1.1 Balance The Delay Of Each Path ......................... 5
    2.1.2 Basic Parallelism ........................................... 8
    2.1.3 Re-arrange Code Sequence ................................ 8
    2.1.4 Add Register To Store Intermediate Result .......... 11
    2.1.5 Pipelining .................................................... 12
  2.2 Space Optimization .................................................. 15
    2.2.1 Optimal Width Variables ................................ 15
    2.2.2 Reuse Component .......................................... 15
  2.3 Evaluation ............................................................. 16
    2.3.1 Equations ................................................... 16
    2.3.2 Reasoning By Using the Equation ...................... 17
  2.4 Summary ............................................................... 18

## 3 Systematic High Level Hardware Design ....................... 19
  3.1 Design Flow ......................................................... 19
  3.2 Program Analysis ................................................... 21
    3.2.1 Four Guidelines For Program Analysis ............... 21
  3.3 Direct Conversion .................................................. 23
4 Case Study: 2-D Gel Image Processing 25
4.1 2-D Gel Image Processing 25
4.2 Program Analysis 25
  4.2.1 Guideline 1: Choose the part which is executed most of the time 26
  4.2.2 Guideline 4: Avoid floating point calculation if possible 27
  4.2.3 Guideline 2: Choose the part with low data dependency 28
  4.2.4 Guideline 3: Choose the part with low communication overhead 28
4.3 Direct Conversion And Balance Delay 29
  4.3.1 Performance Optimization 1: Basic Parallelism 30
  4.3.2 Performance Optimization 2: Rearrange Code Sequence 32
  4.3.3 Performance Optimization 3: Addition of Register To Store Intermediate Result 34
  4.3.4 Performance Optimization 4: Pipeline 35
  4.3.5 Space Optimization: Reuse Of Component 36
4.4 Evaluation and Conclusion 37
4.5 Summary 38

5 Conclusion 39
  5.1 Assessed Achievement 39
  5.2 Future Works 39
## List of Figures

1.1 Design Flow For Handel-C ........................................ 3
2.1 Breaking Up Complex Operations ................................. 6
2.2 Basic Parallelism .................................................. 8
2.3 Non-optimized code ............................................... 10
2.4 Steps To Rearrange Code ......................................... 11
2.5 Register Storing Intermediate Result ......................... 11
2.6 Conversion to Pipeline ........................................... 13
2.7 Effect Of Pipeline ................................................ 14
2.8 Shared Expression in Handel-C ................................. 16

3.1 Hardware Development Steps ..................................... 20

4.1 Case Study: Break Up Complex Operations .................... 29
4.2 Case Study: Summary Of Basic Version ........................ 30
4.3 Case Study: Basic Parallel Method ............................. 32
4.4 Case Study: Rearrange Code Example .......................... 33
4.5 Case Study: Addition of Register .............................. 34
4.6 Case Study: Comparison Between Different Performance Op-
    timizations ....................................................... 35
4.7 Case Study: Reuse of Component ............................... 36
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Case Study: Non-Optimized vs Basic Parallel</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Case Study: Without Rearrangement vs Rearrangement</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Case Study: Non-Optimized vs Rearrange+Reg Addition</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Case Study: Pipeline vs Other Versions</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Case Study: Non-Optimized vs Component Reuse</td>
<td>37</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the emergent of Handel-C [5], a C-like hardware language, a complete high level FPGA design approach is realized. However, most of developers will stick on the lower-level language such as VHDL when they are aiming to design high performance hardware. It is because developers have greater control on the actual circuit implementation in low-level approach. But low-level design probably will reach its limit when FPGA chips grow bigger and bigger. Developers will not be able to develop new application quick enough with low level design which consists of billions of gates. A high-level approach will then be the answer. The purpose of this project is to introduce a systematic way of developing high performance hardware under high-level approach.

1.1 Background and Related Works

In this section, I am going to present the materials that are necessary to understand the content of this report.

1.1.1 Field Programmable Gate Arrays (FPGAs) [1]

Like Programmable Logic Devices (PLDs), FPGA is a piece of hardware which is programmable. However, while the size of PLDs is limited by power consumption and time delay, FPGA can easily implement designs with million of gates on a single IC. The re-programmable nature of FPGA allows developers implements design with shorter development times and lower cost than an equivalent custom VLSI chips. It worths mentioning that development of FPGA is faster than Moore’s Law with capacity doubling every year. With millions of gates available on the newest chip, FPGA is an ideal platform to develop reconfigurable system which is capable of execute complicate application at performance. Therefore, FPGA is the chip I am developing application for.
1.1.2 Pilchard [2]

Pilchard is a reconfigurable computing platform employing a field program-
vable gate array (FPGA) which plugs into a standard personal computer’s
133MHz synchronous dynamic RAM Dual In-line Memory Modules (DIMMS)
slot. Comparing to traditional FPGA devices which are utilizing the PCI
interface, Pilchard allows data to be transferred to and from the host com-
puter in much shorter time, due to the higher bandwidth as well as the
lower latency of the DIMM interface. However, as DIMMS is not originally
designed for Input/Output (I/O), extra control signals will be needed for
Pilchard to indicate the start and the end of data processing. As a result,
high level behavioral design approach is preferable to low level structural
design approach for developing applications for Pilchard. That’s proves
why it is vital to have a systematic way of high level development for high
performance FPGA.

1.1.3 RC1000 [3]

RC1000 is a 32-bit PCI card designed for reconfigurable computing ap-
lications. It has full board support package in Handel-C with libraries
which ease the circuit design for this device. It also features 4 SRAM
banks (2Mbytes each) on the board which can be accessed by the FPGA
or host CPU. The board can be configured to be run between 4000KHz to
100MHz. This device is very different from Pilchard in many aspects. In
the report, I will show that the development steps introduced in this project
is general and can be applicable to application development on different
devices.

1.1.4 VHDL [4]

VHDL is one of the first high-level languages emerged in the market for
designing applications with programmable logic devices. VHDL provides
high-level language constructs that enable designers to describe large cir-
cuits and bring products to market rapidly. It supports the creation of
design libraries in which to store components for reuse in subsequent de-
signs. Because it is a standard language (IEEE standard 1076), VHDL
provides portability of code between synthesis and simulation tools, as well
as device-independent design. It also facilitates converting a design from a
programmable logic to an ASIC implementation. The disadvantage of this
language is it is not completely high level, the language still expects user
to know the hardware behaviors of the components. Therefore, I decided to
use another even higher level hardware language, i.e. Handel-C.
1.1.5 Handel-C [5]

Handel-C is a high level C-like programming language designed for compiling program into hardware images of FPGAs or ASICs. Handel-C provides some extra features which are not appeared in C to support few hardware optimizations. One of those is the language supports specifying the width of each signal so that just optimization can be achieved by targeting the exact resources needed by Handel-C compilers. Handel-C compilers target hardware directly by mapping the program into hardware at the netlist level in xnf or edif format. The advantage of Handel-C over VHDL is that it doesn’t expect users to know too much about the hardware in low level which VHDL does. It is a completely high level language! Figure 1.1 shows the design flows I will adopted in converting Handel-C program to hardware. Although several tools are involved in different steps, but users won’t need to worry about the hardware detail. Because what users need to do is just clicking several buttons to launch the program for converting the file into next step, it is as simply as that.

Figure 1.1: Design Flow For Handel-C

1.1.6 Extending the Handel-C language [7]

Dong U Lee, a Ph.D students, has invented a language which supports both hardware and software. His approach is to combine both C and Handel-C language. In the language, user can specify which part is done by software and which part is done by hardware. In the project, he also developed an more friendly interface for communication between the host and the FPGA device. However, the number of devices currently supported by this language is limited. That’s why I finally gave up on using this language.

1.2 Contribution

- I have developed an easy but efficient optimization method which can rearrange code so that it can be run in minimum of cycles.
- I have developed a systematic design flow for high level hardware design target for high speed devices
• I have implemented the complicated 2D gel image processing on hardware
Chapter 2

Optimization

In this chapter, I am going to discuss various methods to optimize the high level code. Optimization is the main part which we try to exploit and utilize parallelism to achieve speed up which PC software normally is not able to do so because of the limited resources of CPU. The main focus of this chapter will be on how to automate these optimizations processes. We will also discuss some evaluation equation so to measure the speedup we can achieve after optimization.

2.1 Performance Optimization

This is exploiting the potential parallelism of the program and then run different non-conflicting operations at the same clock cycle to acquire speed up. In normal applications, tens of even hundreds of operations which run sequentially on PC’s CPU may be able to run in parallel. However, PC can’t run them in parallel because of the limited hardware resource. But by designing specific hardware to run as many operations as possible in parallel, significant speed up can obtain. This is the main reason why application on FPGA can sometimes run faster than the corresponding software version even though FPGA hardware run at much slower clock speed (of course we also need to take account of the CPI, but even then PC CPU can still do the same individual operation much faster). There are several techniques we could apply:

2.1.1 Balance The Delay Of Each Path

Balancing the delay of each path is important because the hardware clock speed will at most be the same as the path with longest delay. Therefore, if the delay of one particular path is much later than the others, then it means we have wasted resource as other paths is capable of running at much higher speed. By balancing the delay, it can make sure that the the
parallel optimization will be optimal in later stage. The delay of a path can be defined as:

\[ T_{\text{delay}} = T_{\text{logic}} + T_{\text{routing}} \]  

(2.1)

where \( T_{\text{delay}} \) is the total delay of the path  
\( T_{\text{logic}} \) is the delay due to logic  
\( T_{\text{routing}} \) is the delay due to routing

Therefore, reducing the delay is done by reducing one of the \( T_{\text{logic}} \) or \( T_{\text{routing}} \) or both. There are 2 main steps to achieve this:

- Break up complex operation into simple operations;
- Use components with pre-defined placement and routing constraints

**Breaking Up Complex Operation**

First of all, the simplest step to do is to break the complicated operation into several simpler operations. This step effectively reduce the logic in each operation thus reduce \( T_{\text{logic}} \) in equation 2.1. In software program, the effect of complicated operation normally will run as quick or even quicker than the simpler operations with the same result as the compiler will optimize the instructions execution for us. However, for hardware, a complex operation mean it needs a longer clock to finish, while other simpler operations need not take that long to finish. Figure 2.1 shows an example of breaking up a complex operation into simple operations. In this example, we can see that sometimes extra registers are needed to store intermediate result of the calculation to make the operation simple enough.

![Figure 2.1: Breaking Up Complex Operations](image)

**Predefined Placement and Routing Component**

If the result of the first method is not satisfactory enough or the timing constraints still isn’t met, we can then use a timing analyzer provided by the FPGA chip developer to find out the longest paths. For example, we
can use Timing Analyzer coming along with Xilinx ISE Foundation 4 for timing analysis of Xilinx FPGA chips. After finding out the longest path, we will then know which operation run too slow. At this time, we could try 2 methods to increase the speed of this operation. The first is try to write a constraint file ourselves which specify explicitly the placement and routing of this component. This could enhance the timing as the tools which automatic do the placement and routing is normally not very smart. Then we will include this constraint file before the placement and routing process. However, this approach require the developer to have fair amount of knowledge on the FPGA chip they are using. This includes knowledge on the primitives component supported by the chip and the relative placement and routing of these primitives which can achieve the minimum delay. The second approach is easier, it is to use the macro of the predefined placement and routing components defined by the chip developer. Put it simpler, the chip developer has done the job for you, and we should just use it! For Xilinx, they have a program called Core Generator which does exactly what I mentioned. How to include these components in Handel-C program is specified in Handel-C menu. However, users must know that the timing of output and input of these components requires extra care. Because of the language limitation of Handel-C. The input signal will always arrive one cycle late into the component. This step will reduce $T_{routing}$ of equation 2.1 because by nicely placing the logic blocks the routing of the signal will be much shorter thus the delay due to routing will significantly reduced.

Possibility of Automating This Process

Above, we have discussed 2 methods to achieve this step. This step is difficult to be automated, because it is difficult to define what is complex operations and what isn’t. This depends on the device and chip we use as well as the functionality of the program. For device which need a very restricted timing constraint. A 16 bits multiplication may be considered as complex while for some other device which cannot run at high speed, it may be a waste to break up the operations into very simple one which can run at very high speed as the device won’t be run at that speed anyway. However, we can borrow the idea from Lee again to make automation of this process possible. While compiling the source, we can include information about the device we are using as well as the timing constraint. The library of the device will include the delay of each logic unit. It will also include some information on how the FPGA development tools will route the signals. Then the compiler will be able to approximate the $T_{logic}$ and $T_{routing}$ thus $T_{delay}$ of each path. The result will then be compared with the timing constraints specified. If violation is detected, the compiler will use the 2 methods mentioned above to balance the delay of each path until the constraint is met.
2.1.2 Basic Parallelism

This is the first step of the actual performance optimization process and is the easiest. The following rules can be applied to automate the process. Scan through the program sequentially. Group as many operations as possible into one clock cycle until there is violation of data dependency. Then repeat the process again for the next cycle. As we have discussed how to detect data dependency in earlier section, this process is possible to be done automatically. Figure 2.2 is a simple example on how to achieve this. We can see that without parallel execution, it takes 8 cycles to complete the 8 operations. However, with parallel execution, it takes only 2 cycles.

![Figure 2.2: Basic Parallelism](image)

2.1.3 Re-arrange Code Sequence

Sometimes, a program can have high parallelism, but the execution sequence of the operations prevents the method just mentioned above to achieve that level. For example, the code in figure 2.3 will have the same effect on the above example shown in figure 2.2. But if we use the "basic parallelism" method. We will need 4 cycles to finish the operations instead of 2.

We can solve the problem by re-arranging code sequence of the code so that the code can run in least cycles as possible. For example, the code above can first change to the same sequence as in 2.2. Then we will apply "basic parallelism" method again. For this process to be automated by compiler, the compiler will need to have fair amount knowledge and reasoning on the program. I have discovered a method to automate this process:
1. Firstly, choose a group of codes to start with, preferably in the innermost loop.

2. Create an empty table with variables names as index and labels as value. The label is at the formal var:n where var is the name of a variable and n is the number specifying the operation sequence.

3. Scan through the code sequentially. For each variable assignment (Either modification/initialization), assign a label to the operation following the rules listed below:

   **step 1**  search the table to find out the label of the variable being assigned to.

   **step 2a** if no entry is found, the variable is first being assigned. Add an entry in the table, the content (label) is specified as:

   **step 3ai** if the variable is assigned a constant value or a signal from outside the block we are working with, specify the label as var\_name:1 where var\_name is the name of the variable being assigned.

   **step 3aii** if the variable value depends on other variables, get the labels of these variables from the table. Assign the label of the variable same as the labels we got with the biggest order but with the order incremented by 1. Eg, for a = b + c, if label for b is d:3 and c is e:4, then label for a should be e:5.

   **step 2b** if an entry is found, the variable has been assigned before. Get the label of that variable.

   **step 3b** Update the label of the variable as specified in step 3ai and step 3aii but with a change that when finding the biggest order label, we should include label of itself in the comparison. For example, if a = b + c and the labels of b, c are same as above but the label of a this time exists in the table with value a:5 then the new label will be a:6. Note that we can treat constant as having order of 0 when doing comparison.

   **step 4** associate the operation with the label we just specified.

4. After labelling all the operations, we will rearrange the operations such that operations with the same order are placed together.

5. Now ”basic parallelism” method will return code which can run in minimum cycles which is the same as the highest order of the labels within the block.

6. Then we can work with one outer loop, repeat from step two again until the whole program is covered.
The method works because what it does is to push all operations to execute at the earliest cycles. As the result, the modified code must be able to execute at minimum number of cycles. It is obvious that this method can be automated quite easily. By combining this method with the basic parallelism method mentioned before, a surprising efficient parallelisation tools is developed!

Figure 2.4 shows examples of how the method works.

Figure 2.3: Non-optimized code
### 2.1.4 Add Register To Store Intermediate Result

Just re-arranging the operations sometimes is still not enough. If we find that there are a lot of operations on the same variable while other variables just got few operations on them. Then we will come out with many operations in the first few cycles while there are just operations on that variable in the last few cycles. So, can we balance the operations in each cycles? The possible solution is to add registers to store Intermediate Result of the variable and run the calculation of the Intermediate result concurrently to shorten the cycles taken. This technique doesn’t always work and is difficult to automated. Because this requires a lot of reasoning and logics. Figure 2.5 shows an example of this technique. We can see that before modifying the code, it will need to take 3 cycles to finish the operations, but take only 2 after we modified the code.

---

**Figure 2.5: Register Storing Intermediate Result**

```
20

a = 1;  a : 1
a = a + 1;  a : 2
b = 2 * 3;  b : 1
b = b + 4 * 5;  b : 2
b = b + 6 * 7;  b : 3
c = 3;  c : 1
c = c + 3;  c : 2
d = 4;  d : 1
d = d + 4;  d : 2
```
2.1.5 Pipelining

Pipelining is an implementation technique whereby multiple tasks are overlapped in execution. Hennessy and Patterson [6] described pipeline in chapter 3 of their book as:

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, though on a different car.

Ideally, we can start the next task after every cycle. Then when, the pipeline is full, the throughput will be a task per cycle in regardless of how many cycles it take for the task to finish. Therefore, in principle, if the task need 100 cycles to complete and you have enough tasks to make the pipeline full most of the time. It will takes approximately 100 times less cycles to finish all the tasks! However, the speedup won’t be 100. Because:

- Pipeline requires extra control logics thus increase overhead.
- Pipeline requires extra registers to store the intermediate result thus increase the delay.

Moreover, developers need to remember that each task won’t take shorter to finish. So pipelining is ideal for calculation of large set of data which goes through the same calculation.

There are some points we need to care about when implementing pipeline:

- Structural Hazard occurs when there are not enough hardware resources to deal with the overlapped task execution. For example, we cannot multiple read data from the same ram instance in the same clock cycle. Solution to it is to make sure enough resources have been created for the pipeline.

- As the register you used to store the intermediate value is changed every cycle now, the data will be lost unless we have extra registers to store the data. So if the result at one pipeline stage is needed several stages later in the pipeline, we will need to make sure that we add extra registers to store and pass the result value whenever the task proceeds into the next pipeline stage. So that when as task advances to the new stage, the correct data needed by the task will also come with the task.

- Data Hazard occurs when data needed by the task is not yet ready when it enter to a new pipeline stage. We need to design the pipeline carefully so that it won’t happen.

Figure 2.6 is an example of converting a handle-c program into pipeline. The function will return a result which is the input times 5 factorial. Figure 2.7 shows the effect of the pipeline suggested in the above example.
Figure 2.6: Conversion to Pipeline

```plaintext
n=0;
while(n<100)
{
  a=input[n];
  a=a*2;
  a=a*3;
  a=a*4;
  par{
    output[n]=a;
    n++;
  }
}
```

```
par{
  n[0]=0;
  n[4]=0;
}
while(n[4]<100)
{
  par{
    par{ //stage1
      a[0]=input[n[0]];
      n[1]=n[0]; //preserve n
      n[0]++;
    }
    par{ //stage2
      a[1]=a[0]*2;
      n[2]=n[1]; //preserve n
    }
    par{ //stage3
      a[2]=a[1]*3;
      n[3]=n[2]; //preserve n
    }
    par{ //stage4
      a[3]=a[2]*4;
      n[4]=n[3]; //preserve n
    }
    par{ //stage5
      output[n[4]]=a[3];
    }
  }
}
```
Figure 2.7: Effect Of Pipeline

Before Pipelining

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc2</th>
<th>cc3</th>
<th>cc4</th>
<th>cc5</th>
<th>cc6</th>
</tr>
</thead>
<tbody>
<tr>
<td>task1</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task2</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task3</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After Pipelining

<table>
<thead>
<tr>
<th>cc1</th>
<th>cc2</th>
<th>cc3</th>
<th>cc4</th>
<th>cc5</th>
<th>cc6</th>
</tr>
</thead>
<tbody>
<tr>
<td>task1</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task2</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task3</td>
<td>read</td>
<td>*2</td>
<td>*3</td>
<td>*4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2 Space Optimization

Space optimization has totally different purpose to time optimization. While time optimization normally mean using more resources to do several operations at the same time to acquire speed up, space optimization concern about how to make the work done by least resources possible. It is particular important in FPGA design as the less resource we use, the more operations we can do within the chips. I am now going to discuss some techniques for space optimization:

2.2.1 Optimal Width Variables

Remember the direction conversion of variable into standard width in 4.1? Indeed, if we know the upper and lower bound of the values that the variable will take, we will find that some bits are never used in most of the cases. For example, a variable which takes value with range 0-1024 will not need 32 bits. 11 bits unsigned value(0-2047) is enough to cover the range. A variable with less width will need a smaller register to store it. Operations taking operands with smaller width will require less logic gates to implement than the same operation with wider operands. As mentioned in equation 2.1.\( T_{\text{logic}} \) contributes to one of the component of \( T_{\text{delay}} \). Thus, this optimization not only save more resources, but also reduces \( T_{\text{logic}} \) and, therefore, \( T_{\text{delay}} \).

Altaf, an Ph.D student in Imperial College, is currently working on a project [8]. The aim of the project is automating the resource optimization by deciding the optimal width of the floating point variable. His approach is by running through simulation sufficient number of times, then use statistical approach to find out the optimal width of the variable.

2.2.2 Reuse Component

If the same operation with different operands with the same width is executed in more than once in different clock cycle. We can create just one component instance and reuse the component with different operands. This approach doesn’t affect the delay of the circuit, but it save gates of the replicated components! In order to reuse the component as many times as possible, we can even drop the constraint so that same operations which take shorter operands can also use the shared component. This works because operations which take longer operands are always compatible with the same operations which take shorter operands. One may argue that it will increase the delay of the operation because the component use more logic than it needed. Thus true, in principle, we can run the operation faster if we just use the exact logic needed. However, as FPGA devices don’t support variable clock for different operation. The clock speed is simply determined by the longest delay path in the program. Therefore, there is nearly no ef-
fect on the program by sharing the component as the component is needed anyway in the operation which will utilize all its logic.

In Handel-C the sharing of the component can easily specified by shared expression. Figure 2.8 shows the sample code for doing this. For more detail, it is always useful to read the Handel-C manual from help file of the DK1 design suite.

![Figure 2.8: Shared Expression in Handel-C](image)

```
a = 3 * 4;
b = 6 * 7;
c = 11 * 13;
d = 9 * 9;

shared expr mult(a, b) = a*b;
a = mult(3, 4);
b = mult(6, 7);
c = mult(11, 13);
d = mult(9, 9);
```

### 2.3 Evaluation

It won’t be know what the optimization can achieve if we don’t have method to evaluate the result. Therefore, in this section we are going to discuss the technique involved in evaluation. I will introduce some equations so that developer can make conclusion about the implementation. Is it worthy to implement the program in hardware? If the result is not as desirable currently, will it be worthy to do so when technology advances? Or can we think of another approach which can acquire much higher speed up? These are the question we want to answer in this step.

#### 2.3.1 Equations

\[ T_{exec} = T_{config} + T_{comm} + T_{proc} \]  \hspace{1cm} (2.2)

Now we will introduce some extra equations which can help us to evaluate the result.

Assume that hardware reconfiguration is rarely during execution, then \( T_{config} \approx 0 \), thus, the equation becomes:

\[ T_{exec} = T_{comm} + T_{proc} \]  \hspace{1cm} (2.3)
$T_{\text{proc}}$ can be determined by:

$$T_{\text{proc}} = n \times t$$

(2.4)

where $n$ is the number of cycle and $t$ is the clock delay.

t can be determined by:

$$t = 1/c$$

(2.5)

where $c$ is the clock speed

therefore

$$T_{\text{proc}} = n/c$$

(2.6)

$n$ can be obtained in simulation and $c$ can be obtained from the timing report after circuit compilation.

we can also obtained $T_{\text{comm}}$ by:

$$T_{\text{comm}} = w/b$$

(2.7)

where $w$ is number of bytes needed to be transferred between the host and the FPGA device in each execution. $b$ is the bandwidth of the interface which the device using to communicate with the host PC.

Sub them all together

$$T_{\text{exec}} = w/b + n/c$$

(2.8)

From this equation, we can see that optimization isn’t just about reducing number of cycles and increasing the clock speed. We also need to think about the data communication. Sometimes, speed up is obtained not because you can pass the data quicker, but because you have a better method to pass around the data. For example, if we can clever manipulation the data so that data is only read for every $n$ executions. Then the $w$ involved in each execution will in average $o+i/n$ where $o$ is the amount of data output to host while $i$ is the amount of data input from host.

### 2.3.2 Reasoning By Using the Equation

We know the $T_{\text{exec}}$ of the software. So we will know that whether our hardware implementation will achieve any speed up. If not, we will then try to reasoning it:
1. First of all, is \( w/b \) bit larger than \( T_{exec} \)? If so, no matter how much speed up the hardware achieve, there will be no speed up at all, because most of the time is spent on transferring the data. Unless you use a device which utilizes a communication interface with much higher bandwidth, it is not worthy to implement the program in hardware.

2. If \( w/b \) is considerably smaller than the expected \( T_{exec} \), assume \( c \) is fixed. i.e. the clock speed remains the same, calculate the \( n \) required to achieve such a speed up. Then think about whether it is possible to exploit more parallelism to achieve this number of cycles.

3. If you find that it is not much you can do with \( n \), then try to fix \( n \) and calculate the \( c \) require to achieve the speed up? Then think about whether current technology allow your program run at that speed? Is it possible that the speed-up will be achieved if you use the chip with the most advance technology? Or do you expect the device which can run at the speed you require will soon enter the market?

4. is there any constraint dragging your approach? For example, may be your approach can run 10 times faster if you can read all the memory location simultaneously

5. Finally, it is worth to think whether you should choose another part of the program to be implemented in hardware which potentially has more parallelism to be exploited.

### 2.4 Summary

In this chapter, I have talked about optimization process. I introduced a code rearrangement method which can effectively make a code run in minimum cycles of code in parallel. Later in the chapter, I introduced equations for evaluation which is an important part for the optimization process. By the equations, we know that execution speed not only depends on how fast the the FPGA chips can run and how many cycles you have reduce. It is also important to have systematic method for reducing the communication cost between the host and the FPGA devices.
Chapter 3

Systematic High Level Hardware Design

In this chapter, I am going to introduce five general steps of converting a software programme into hardware. I will go through each step in detail in the next few chapters.

3.1 Design Flow

Figure 3.1 shows the general development steps for converting software program to hardware circuits in Handel-C. They are listed as follows:

1. The first step is program analysis. It is used for determining which part of the program we would like to convert into hardware. This step is very important as it affect the performance of the final result. The basic idea is to find the critical path of the program which is executed most of the time. Thus the speed up of this part of the program in hardware will also improve the whole program significantly.

2. The second step is direct conversion. That is converting the software program directly into hardware without implementing any kind of optimizations yet. This step ensures that the hardware will behave exactly the same as the software.

3. The third step is balancing the delay. That is balancing delays of each path of the hardware. This is important as the hardware clock delay will be the same as the path with longest delay. Therefore, if the delay of one particular path is much later than the others, then it means we have wasted resource as other paths is capable of running at much higher speed.
4. The forth step is *optimization*. It is important to apply each optimization technique at one time. After applying the technique, we will see whether the program has meet its requirement. If not, we will return to step 3. If so, we will proceed to step 5.

5. The final step is *evaluation*. That is evaluating the actual result of this hardware version of the program. Then draw a conclusion which considers the following questions: Does the hardware provide speedup of the program? If not, in what circumstances it could? Should we try different approach to implement the program?... etc

We can see that the last 3 steps are indeed discussed in the Optimization Chapter we mentioned above. The remaining two will be discussed in this chapter.

Figure 3.1: Hardware Development Steps

- **Step 1.** Program Analysis. Determine which part of the program to be converted into.
- **Step 2.** Direct conversion from software to hardware without parallelism or any other optimizations.
- **Step 3.** Balance the delay of each path.
- **Step 4.** Optimization. Apply one optimization technique at one time. Check whether the requirement is met. Repeat to Step 3 if not.
- **Step 5.** Evaluate the actual result. Draw conclusion.
3.2 Program Analysis

In this section we will look at the first step of the high level hardware design flow. Program Analysis is the most important among the five steps as the effect of the implementation will not be significant or even negligible if the wrong part of the program is chosen to be implemented in hardware.

3.2.1 Four Guidelines For Program Analysis

I have developed 4 guidelines for this step. These guidelines are not forced to be followed, but can give the developers the idea of what kind of programs are preferable to be converted.

Guideline 1: Choose the part which is executed most of the time

The reason is obvious, if we implement the part which rarely executed in hardware, even we can achieve a great speed up at that part, it will has little effect on the actual program as that part is not run frequently anyway. It is preferable if we can find out the most inner loop of the program and implement it in hardware in order to make the most from the limited hardware. However, as the technology advanced, FPGA chips nowadays are getting bigger which can accommodate millions of gate and support faster real time reconfiguration, Implementing the whole program in hardware is more possible now.

Guideline 2: Choose the part with low data dependency

There shouldn’t be high data dependency at the part of program to be implemented in hardware. Bear in mind that, FPGA chips are normally slower than PC’s CPU. This is because extra logics are needed to support reconfiguration on FPGA. Thus, speed up is achieved purely on exploiting potential parallelism within the program. High data dependency will reduce the potential parallelism we could achieve as data at one part depends on the data at the other parts which restrict us from processing them in parallel.

Guideline 3: Choose the part with low communication overhead

The execution time for the program in hardware is:

\[ T_{exec} = T_{config} + T_{comm} + T_{proc} \]  \hspace{1cm} (3.1)
where $T_{\text{exec}}$ denote the total execution time
$T_{\text{config}}$ denote the time for reconfiguration
$T_{\text{comm}}$ denote the time for transferring data
between the host and the FPGA device
$T_{\text{proc}}$ denote the time for actual data processing

As we can see in equation 3.1, the total execution time of
the hardware is not just the time for processing data. It also
includes the time to configure the FPGA device($T_{\text{config}}$) and
the time to send initial data and get result from the FPGA
device($T_{\text{comm}}$). If we don’t need to reconfigure the FPGA
device at real time, $T_{\text{exec}}$ is zero once we configure the de-
vice for the first time. However, $T_{\text{comm}}$ will be non-zero ev-
every time we execute the program. The overhead depends on
how often and how much data we need to transfer between
the host and the device. If large amount of data transfer
is needed most of the time, the speed up may not be favor-
able as $T_{\text{comm}}$ contribute to most fraction of $T_{\text{exec}}$. However,
some tricks could be applied to solve the problem in some
cases. For example, if most of the data remain the same in
sequence of executions of the program, we can leave the data
in the FPGA device without trashing it after the first exe-
cution. Then the $T_{\text{comm}}$ of the subsequent executions just
involve transferring the data which is different.

**Guideline 4 Avoid floating point calculation if possible**

It is known that PC can deal with floating point calculation
better than FPGA chips. With dedicated FPU unit and
well designed pipelined hardware, floating point calculation
is recommended to be done on PC. Although FPGA also has
library supporting floating point calculation, it cannot run
at as high the clock speed as integer calculation. However,
as many researchers from both commercials and academics
are currently working on this issue, I can foresee that this
guideline will soon be not-applicable. Meanwhile, avoiding
floating point calculation is still recommended. If developers
encounter floating point calculation, they should try to rea-
sion about the range of the data and try to find out whether
it is possible to use integer calculation to emulate the float-
ing point calculation.

For all the hardware development steps, this step is the most difficult to
be automated. Because this involves reasoning what the program actually
does. Therefore, it is difficult to generate some general rules for automating
this step. However, some ideas can be borrowed from current compiler
techniques to facilitate the process. For example, Hennessy and Patterson [6] have discussed techniques for compiler to find out data dependencies and possibility of eliminating them.

3.3 Direct Conversion

In this section, we will look at the second step of the development process. It is direct conversion from software language into hardware language. This step doesn’t involve application of any kind of optimization. The reason of leaving it for later stage is we want to make sure that the hardware program can behave exactly the same as the software program. This is for debugging purpose. If we apply the optimization techniques at the beginning, it is hard to debug the program when something goes wrong because you will not be able to find out in which step the error occurs.

This part is quite straight forward as Handel-C is a high level like language. The only concern is inserting extra code to deal with the communication between the host and the FPGA device. This part of the code correspond to the communication cost($T_{comm}$) mentioned in eq 3.1. One of its job is to get and setup initial data needed to be processed from the host, another is to transfer the result back to host after the data is processed by the FPGA device. As mentioned above, this part is very important. It is important that we try to find a way which we can transfer minimum data in every execution. One of the probability is try to keep as many reusable data as possible in FPGA by having a lot of RAM. Another is cleverly partitioned the data so that the data kept in FPGA is reused as many time as possible before being thrashed out. The code for the communication is not easy to be automated. It is because different devices have different method to communicate with the host. However, last year, an excellent BENG project from Dong U Lee [7] has suggested possible solution to this problem. He developed a new language which unifies the hardware language and the software language. Within the language, user can define which part is done by hardware and which part is done by software. By specifying the device being used, his compiler will generate the communication code according to the device library automatically. Therefore, for the compiler to support a new device, the chip developers simply write the library for it. This approach prevents normal programmers from spending a lot of time to check the data sheet of each device they are working in order to write the communication program.

3.4 Summary

In this chapter, we have discussed the general steps for high level design for high speed FPGA devices. In the second part, I have mentioned the
technique involved to minimal the data involved in communication.
Chapter 4

Case Study: 2-D Gel Image Processing

In this chapter, I am going to show how the techniques discussed in Chapter 2 can be applied in real life. I will go through the development steps introduced in Chapter 2 by using 2-D Gel Image Processing as an example. We will try to compare the performance of Pilchard against RC1000. Sadly, I failed to implement the gel processing on pilchard yet due to some difficulties. I will explain it in detail in later section.

4.1 2-D Gel Image Processing

2-D Gel image processing is a technique used in proteomic research. Its purpose is to match the gel patterns in 2 images taken in different time. However, it is used to be difficult to match the patterns accurately and efficiently because there are always misalignments at different scales when the images are taken. A novel registration technique based on image intensity distribution rather than selected features was introduced in a recent paper [9] by Dr. S. Vesser, Dr. M. J. Dunn and Dr. G. Z. Yang. The method uses a multi-resolution representation of the gel profiles and exploits the fact that coarse approximation to the optimal matching can be extracted efficiently from low-resolution images. Moreover, by using multi-resolution image registration, the misalignments can be removed systematically. In this chapter, the implementation of the code is based on a software which utilizes this technique.

4.2 Program Analysis

As mentioned in chapter 3, the first step of the hardware development is program analysis.
4.2.1 Guideline 1: Choose the part which is executed most of the time

And in this step, the most important rule is to find out the innermost loop of the program which is executed most of the time. Otherwise, the effect of the implementation won’t be obvious. To achieve that, we need to know how the algorithm of the technique works. Here is the brief registration algorithm used in the program:

**step 1** Set detail level \( l \) to 0

**step 2** Blur images \( I_1, I_2 \) by setting resolution to 5

**step 3** Optimize parameters from an initial rigid transformation \( t \) in \( T_{\text{rigid}} \)

**step 4** While \( l \) smaller than 5 do the followings, go to **step 4**, otherwise finish

**step 5** Subdivide \( t \)

**step 6** if not finish all squares \( a_{i,j}, a_{i+1,j}, a_{i,j+1}, a_{i+1,j+1} \) in the grid of \( t \), then go to **step 7**, otherwise go to **step 8**

**step 7** Optimize the control points \( c_{i,j}, c_{i+1,j}, c_{i,j+1}, c_{i+1,j+1} \) by using BFGS to maximize \( f(c) = \text{corr}(l_1, t_c(I_2)) \) in the affected area, return to **step 6**

**step 8** Increment detail level \( l \)

**step 9** Blur images \( I_1, I_2 \) by setting resolution to \( 5 + l \), return to **step 4**

Basically, the idea of the algorithm is to divide the 2 input images \((I_1, I_2)\) into certain number of blocks(grids). The grids of image1 \((I_1)\) are always square in shape while the shape of the grids of image2 \((I_2)\) is determined by the control points \((c_{i,j}, c_{i+1,j}, c_{i,j+1}, c_{i+1,j+1})\). A target image \((t_c(I_2))\) is produced by transforming the pixels inside each grid of \( I_2 \) to the corresponding square grid in \( I_1 \). Similarity and its derivative against the control points are then calculated between the \( I_1 \) and \( T_c(I_2) \). Similarity is the correlations of pixels between the 2 images. Then the derivatives will be used to adjust the control points so that the similarity between \( I_2 \) and \( T_c(I_2) \) will increase. This process will repeat until the increase of similarity is satisfactory.

The whole process mentioned above will be carried on different resolutions of the image. Starting from low resolution to high resolution. The number of grids is determined by the resolution the process is working at. The control points derived by lower resolution will determine the initial control points of the process in higher resolution. It is obvious now which step of the algorithm cost most of computational time, it is **step 7**: ”Optimization of Control Points”. So I choose this part is the starting point of my hardware implementation.
4.2.2 Guideline 4: Avoid floating point calculation if possible

So we have chosen the part which is executed most of the time. Now it’s time to look whether it is suitable to be implemented in hardware. In the first glance, the program violates this guideline, because the optimization of the control points uses the derivatives which involve a lot of floating point calculation. At this stage, the first thing we should do is try to see whether it is possible to use integer calculation to emulate floating point calculation. The useful technique is to scale the floating point values so that certain amount of the lowest significant bits of the integer represents the value behind the decimal. But floating point number normally can accommodate much larger range of values than integer of the same width. That means we normally need to drop some of the least significant bits of the floating point number when transferring it to integer. The criteria for this technique to work is dropping these bits won’t affect the accuracy of the program.

Sadly, I realize that it is not the case in this program. It is deal to the fact that some intermediate result of the algorithm involves multiplication of floating point values with large values, especially when the program is processing at high resolution which means more number of pixels are involved. That’s mean dropping of the least significant values will have significant effect on the final result. But that’s not the end of the story. When I reasoned more deeply into the program, I found out that the optimization indeed can be separated into 2 parts.

- Transformation and Calculation of Similarity and Derivatives, and
- BFGS Optimization

Only the second part is floating point calculation intensive. The first part can be implemented mostly on integer calculation. Now, we have to ask the question: Is the first part still the part which executed most? Before answering the question, let’s look at the following equation:

\[ c = w \times i \times CPI \quad (4.1) \]

where
- \( w \) denotes number of data unit in the program
- \( i \) denotes average number of instructions per data unit
- \( CPI \) denotes average cycles per instruction
- \( c \) denoted total number of cycles of the program

Now I can tell you the answer is yes, although both parts are executed same number of time, part 1 indeed is the most computational intensive. The reason is most of the pixels of both images are involved in the calculation of part 1, while part 2 just works on the control points. Number of control points is smaller than number of pixels in orders! Therefore \( w \) in equation 4.1
of part 1 is much larger than part 2. But both $i$ and $CPI$ are nearly the same in both part. $i$ is nearly the same because both part involves nearly same amount of calculation per data. $CPI$ is the nearly the same as long as the pipeline in the CPU is full which is true most of the time in this case as both parts involves large amount of calculation. Therefore, by the equation, $c$ in part 1 is much larger than $c$ in part 2. As PC’s CPU has fixed clock speed for all instructions, we can conclude that part 1 is the most computational intensive part of the program.

4.2.3 Guideline 2: Choose the part with low data dependency

Ok, does the part of code chosen above meet this criteria? The answer is yes again. It is because for each target pixel, its value is entirely determined on 4 pixels from the Image2 and the value isn’t affected by other target pixel. This means in principle we can calculate all the target pixels at the same time. It reflects an extremely low data dependency. The calculation of similarity and derivative also has low data dependency. It is because they are calculated from the target pixels which have low data dependency.

4.2.4 Guideline 3: Choose the part with low communication overhead

By equation 2.1, $T_{comm}$ contributes one part of the $T_{exec}$. Therefore, it is important that $T_{comm}$ won’t be too large. As mentioned in the previous chapter, $T_{comm} = \frac{w}{b}$ which is the number of bytes of data needed to be transfer divided by the bandwidth of the communication interface. The amount of data needed to be transferred is:

$$w = I_1 + I_2 + cp + T_c(I_2) + d + s$$

where $I_1$ and $I_2$ are the source images, $cp$ is array of control points, $T_c(I_2)$ is the transformed image, $d$ is an array of intermediate results which can determine the derivatives and $s$ is an array of intermediate results which can determine similarity. At highest resolution level, each image size is $512 \times 512$ bytes. There are $17 \times 17$ control points. Each control point has $x, y$ coordinates of 4 bytes. Each control point is also associated with six 8bytes $d$ values. There are five $s$ values, each of 8 bytes. Therefore, the total number of bytes to be transferred is 802656 bytes. For Pilchard, the bandwidth($b$) of 64 bits 100MHz DIMM interface is $8 \times 100MHz = 800$MB/s. Thus, $T_{comm}$ is 802656/800 = 1024 × 1024 which is less than 0.001s! And for RC1000, the bandwidth($b$) of the 32 bits PCI33 bus is $4 \times 33$MHz = 132MB/s. And therefore, $T_{comm}$ is 802656/132 = 1024 × 1024 which is around 0.01s. Figures for both devices are not high because the execution
time of the program is over 3s normally, therefore, we can carry on the implementation.

4.3 Direct Conversion And Balance Delay

The second step of the development steps is converting the code into Handel-C without applying any form of optimization. This steps ensure that the basic version of your hardware implementation behaves exactly the same as the software version.

Luckily, there isn’t any recursive functions in the code I am converting. Therefore, this step appears to be straight forward. The program can compile successfully after following the six steps specified in chapter 4. There is one thing needed to be mentioned, the code for communication is for simulation at this stage. It is for the sake of debugging since debugging the program in simulation is much easier than debugging it in hardware. In simulation, we can check how the signals(variables) changed in each cycle. But in hardware, this will be impossible. Before starting the simulation, I use "Breaking Complex Operations" technique to balance the delay of each path so that the most complex operations involved is 16bits multiplication.

For example, the code in left hand side of the figure 4.1 is modified as in right hand side. This bit of code is part of the transformation. What it is doing is trying to calculate the weight mean of 2 pixels. Note that the calculation here involve the technique of using integer calculation to emulate the calculation of floating point numbers.

Figure 4.1: Case Study: Break Up Complex Operations

```
<table>
<thead>
<tr>
<th>Complex Op</th>
<th>Simple Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>J0 = extend(J00, width(J0))&lt;&lt;10 +</td>
<td>J0 = extend((J01 - J00), width(J0)) ;</td>
</tr>
<tr>
<td>extend((J01 - J00), width(J0)) *</td>
<td>J0 *= (signed) (0@WX) ;</td>
</tr>
<tr>
<td>((signed)(0@WX)) + 512;</td>
<td>J0 += extend(J00, width(J0))&lt;&lt;10 ;</td>
</tr>
<tr>
<td></td>
<td>J0 += 512;</td>
</tr>
</tbody>
</table>
```
To save time in simulation, I compile the code which is working in lower resolution at $32 \times 32$ and with less number of grids (4 grids with each of size $16 \times 16$. However, the problem size is specified in header, the actual code can be compiled in variable problem size by just changing the header. After compilation by Handel-C compiler, the following result is obtained: 1198568 NAND gates, 2047 FlipFlops, 28428 memory bits are used. In simulation, 132490 clock cycles are needed for data processing. Note that I just count the cycles in data processing, i.e. cycles in $T_{\text{proc}}$. But ignoring the cycles spent in communication. It is because different values will be obtained for different interface. Therefore, it is not meaningful to include the cycles taken for simulation to setup the data. If this version can run at pilchard, the clock will be 100MHz. Thus it will take $132490/(100 \times 1024 \times 1024) \approx 0.0013$ s for data processing. However, I have tried my best to make the program work with pilchard without success. The program looks too complicated for Handel-C compiler to generate circuit which can run at that speed. I try to add more Cores component to reduce the delay. At the end, nearly the whole program except the control signals is using the Cores component. The approach is no longer high level and virtually same as using low level approach like VHDL. As this report focuses on high-level development approach, I decided not to continue this Core replacing process. I believe that it is possible to make the program work on pilchard, but currently the highest level approach possible is structural implementation on VHDL which is still very low-level. I believe that in later time, when a smarter compiler which can produce optimized circuit from behavior language like Handel-C, the program will be able to work at 100MHz. For RC1000, the implementation can work at 12.5MHz and, therefore, takes 0.01s to finish the job. Table 4.2 summarizes the properties of this version.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>1198568</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>2047</td>
</tr>
<tr>
<td>Memory(bits)</td>
<td>28428</td>
</tr>
<tr>
<td>Clock Cycles</td>
<td>132490</td>
</tr>
<tr>
<td>Pilchard(in principle)</td>
<td>0.0013s</td>
</tr>
<tr>
<td>RC1000</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

4.3.1 Performance Optimization 1: Basic Parallelism

Applying the Basic Parallelism method mentioned in Pg. 8, simply scan through the code sequentially, group the operations to the same clock cycle until violation of data dependency. This leads us to a program which achieve
basic parallelism. Figure 4.3 shows the effect of the method on fraction of
the actual code. We can see before applying the method, it takes 8 cycles to
finish that portion. After applying the method, it takes only 4 cycles. This
part of the code is the accumulators of the arithmetics of image1 pixels and
target pixels. These values are then used to calculate the mean or square
mean values of the images. Before compilation and simulation, I expected
that this version will take almost same amount of resources but with less
cycles for the data processing. Because in both versions, I used the same
amount of operators and without explicitly specifying to share resource,
you should use nearly same amount of gates except there is extra logics for
control in this parallel version. The result of compilation and simulation is
as expect. This version uses 1200614 NAND gates, 2233FlipFlops and 28428
memory bits. And the simulation showed that 56496 clock cycles are used
for data processing. Table 4.1 shows the differences between this version
against the version without any optimization. Note that the speedup in this
version will be the same as speedup in cycles when implemented in hardware.
It is because there isn’t increase in delay in any path as no extra logic is
used in each path. If this version works on pilchard, it will take 0.00054s for
each execution for data processing. In RC1000, the program again works at
12.5MHz and takes 0.0043s.

<table>
<thead>
<tr>
<th></th>
<th>Non-Optimize</th>
<th>Basic Par</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>1198568</td>
<td>1200614</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>2047</td>
<td>2233</td>
</tr>
<tr>
<td>Memory(bits)</td>
<td>28428</td>
<td>28428</td>
</tr>
<tr>
<td>Clock Cycles</td>
<td>132490</td>
<td>56496</td>
</tr>
<tr>
<td>Pilchard(principle)</td>
<td>0.0013s</td>
<td>0.00054s</td>
</tr>
<tr>
<td>RC1000</td>
<td>0.01s</td>
<td>0.0043s</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>2.345</td>
</tr>
</tbody>
</table>

Table 4.1: Case Study : Non-Optimized vs Basic Parallel
4.3.2 Performance Optimization 2: Rearrange Code Sequence

As mentioned in chapter 2, Basic Parallelism methods normally cannot exploit the full parallel potential of the program. Now we will use the Rearrange Code Sequence method to see how much more parallelism we can get. Figure 4.4 shows the exactly the same part of the code in figure 4.3. But we can see that by applying this method, we can achieve higher parallelism as only 2 cycles are needed instead of 4 cycles! Note that, although the first assignments in the code was referenced to another variables I and J, we can give them labels of I:0 and J:0. It is because they are signals from outside into this block of code, we can treat them as if they are constants. In this version, I expect the hardware resources used and the clock speed of the hardware implementation are about the same again because of the same arguments mentioned above. But I expect further reduce number of cycles in the simulation. The actual figure is: 1200768 NAND Gates, 2247 FlipFlops, 28428 Memory bits and 29864 clock cycles for data processing. So by rearrangement of the code sequence then application of the basic parallelism technique, it reduces the number of cycles by nearly half than just applying basic parallelism method alone! Again this version fail to work on pilchard. If 100MHz can be achieved, the delay will be 0.00028s. RC1000 can work on 12.5MHz again, which result in delay of 0.0023. Table 4.2 compares the effect of basic parallelism method with and without rearrangement of the code sequence.
### Table 4.2: Case Study: Without Rearrangement vs Rearrangement

<table>
<thead>
<tr>
<th></th>
<th>Non-Optimized</th>
<th>Basic Par</th>
<th>Rearrange</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>1198568</td>
<td>1200614</td>
<td>1200768</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>2047</td>
<td>2233</td>
<td>2247</td>
</tr>
<tr>
<td>Memory (bits)</td>
<td>28428</td>
<td>28428</td>
<td>28428</td>
</tr>
<tr>
<td>Clock Cycles</td>
<td>132490</td>
<td>56496</td>
<td>29864</td>
</tr>
<tr>
<td>Pilchard (principle)</td>
<td>0.0013s</td>
<td>0.00054s</td>
<td>0.00028s</td>
</tr>
<tr>
<td>RC1000</td>
<td>0.01s</td>
<td>0.0043s</td>
<td>0.0023s</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>2.345</td>
<td>4.436</td>
</tr>
</tbody>
</table>

### Figure 4.4: Case Study: Rearrange Code Example

```
J = I * J ;
I J S u m  + =  I J ;   I : 2
I S u m  + =  I ;   I : 1
J S u m  + =  J ;   J : 1
J S q u a r e d  =  J * J ;   J : 1
J S q u a r e d S u m  + =  J S q u a r e d ; J : 2
I S q u a r e d  =  I * I ;   I : 1
I S q u a r e d S u m  + =  I S q u a r e d ; I : 2
```

par { //cycle1
   I J = I * J ;   I : 1
   I S u m  + =  I ;   I : 1
   J S u m  + =  J ;   J : 1
   J S q u a r e d  =  J * J ;   J : 1
   J S q u a r e d S u m  + =  J S q u a r e d ; J : 2
}

par { //cycle2
   I J S u m  +=  I J ;   I : 2
   J S q u a r e d S u m  +=  J S q u a r e d ; J : 2
   I S q u a r e d S u m  +=  I S q u a r e d ; I : 2
}`
4.3.3 Performance Optimization 3: Addition of Register To Store Intermediate Result

This technique is used to remove data dependency to reduce the number of cycles required for execution. In version is built upon the parallel version which has been optimized by using the code rearrangement technique. Figure 4.5 shows one part of the code which this technique is utilized. We can see that the modified code can run 1 cycle less than the original code. The code has the same effect as that mentioned in 4.1. I don’t expect much speedup in this version as this technique isn’t applicable to many part of the code. And normally, you will just be able to squeeze one or two cycles less after doing a lot of code transformation. This version gives the result of: 1201656 NAND Gates, 2309 FlipFlops, 28428 Memory bits and 27816 cycles in data processing. In theory, pilchard can finish the job in 0.00027s while RC1000, again works in 12.5MHz takes 0.0021s. Table 4.3 compares the speedup of this version with the non-optimization while Figure 4.6 shows the speedup so far.

Figure 4.5: Case Study: Addition of Register

Table 4.3: Case Study: Non-Optimized vs Rearrange+Reg Addition

<table>
<thead>
<tr>
<th></th>
<th>Non-Optimized</th>
<th>Reg Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>1198568</td>
<td>1201656</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>2047</td>
<td>2309</td>
</tr>
<tr>
<td>Memory (bits)</td>
<td>28428</td>
<td>28428</td>
</tr>
<tr>
<td>Clock Cycles</td>
<td>132490</td>
<td>27816</td>
</tr>
<tr>
<td>Pilchard (principle)</td>
<td>0.0013s</td>
<td>0.00027s</td>
</tr>
<tr>
<td>RC1000</td>
<td>0.01s</td>
<td>0.0021s</td>
</tr>
<tr>
<td>Speedup</td>
<td>1</td>
<td>4.76</td>
</tr>
</tbody>
</table>
4.3.4 Performance Optimization 4 : Pipeline

Sadly, I failed to implement the whole part of the transformation and calculation of similarity and derivatives in pipeline. Because it will result in a 30 stages pipeline which is very complicated and will have much overhead in control signals. However, I do extract the transformation part out and successfully implement it in 14 stages pipeline. The pipeline version is built upon the parallel version. Assume there is enough data to keep the pipeline full most of the time. Then this version will have nearly 14 times less cycles than the parallel version! Table 4.4 shows the detail of different versions of the transformation program. The pipeline version have nearly 20 times less cycle than the non-optimized version. However, it uses twice as much resources and extra memory. Extra memory is needed to replicate the data so that several memory location can be accessed at the same cycle in the pipeline. Moreover, don’t expect we can achieve 20 times speed up. As mentioned before, pipeline will increase the overhead by registers setup and additional control logic. As expected, after compilation, RC1000 fails to work at 12.5MHz but reduces to 6MHz. What even worse is the program doesn’t behave as it expected although simulation suggests it should. I have asked help from other researchers from the lab and they say it could be due to memory congestions which occurs when memory banks on RC1000 are accessed every cycle.
Table 4.4: Case Study: Pipeline vs Other Versions

<table>
<thead>
<tr>
<th></th>
<th>Non-Optimized</th>
<th>Parallel</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>108840</td>
<td>110794</td>
<td>171970</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>634</td>
<td>873</td>
<td>1810</td>
</tr>
<tr>
<td>Memory (bits)</td>
<td>16780</td>
<td>16780</td>
<td>41356</td>
</tr>
<tr>
<td>Clocks Cycles</td>
<td>37761</td>
<td>14258</td>
<td>2173</td>
</tr>
<tr>
<td>Pilchard (principle)</td>
<td>0.00036s</td>
<td>0.00014s</td>
<td>0.00002s</td>
</tr>
<tr>
<td>RC1000 (pipe not work)</td>
<td>0.00289s</td>
<td>0.00109s</td>
<td>0.00035s</td>
</tr>
</tbody>
</table>

4.3.5 Space Optimization: Reuse Of Component

Apart from time optimization for higher performance, we can have another approach: Space Optimization which allows the program to execute at less resource. In the example in figure 4.7, I create a multiplier component which is shared by several operations. This version is based on the non-optimized version. I expect the same number of cycles in execution but with less resources used. The figures obtained are: NAND gates 413668, FlipFlops 2047, memory bits 28428, 132490 cycles. So all the figures are same as non-optimized version except it uses nearly a third of the original resources! Table ?? summarize the result again.

Figure 4.7: Case Study: Reuse of Component

\[
\begin{align*}
I J & = I * J; \\
I S q u a r e d & = I ^ 2; \\
J S q u a r e d & = J ^ 2; \\
\text{shared expr } & \text{ mult}(a, b) = a * b; \\
I J & = \text{mult}(I, J); \\
I S q u a r e d & = \text{mult}(I, I); \\
J S q u a r e d & = \text{mult}(J, J); \\
\end{align*}
\]
### Table 4.5: Case Study: Non-Optimized vs Component Reuse

<table>
<thead>
<tr>
<th></th>
<th>Non-optimized</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND Gates</td>
<td>1197568</td>
<td>413668</td>
</tr>
<tr>
<td>FlipFlops</td>
<td>2047</td>
<td>2047</td>
</tr>
<tr>
<td>Memory(bits)</td>
<td>28428</td>
<td>28428</td>
</tr>
<tr>
<td>Clock Cycles</td>
<td>132490</td>
<td>132490</td>
</tr>
<tr>
<td>Pilchard(principle)</td>
<td>0.0013s</td>
<td>0.0013s</td>
</tr>
<tr>
<td>RC1000</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
</tbody>
</table>

#### 4.4 Evaluation and Conclusion

So will the hardware implementation could be faster than the software version? If not, in what circumstances it will be? Let’s look at the biggest problem with image $512 \times 512$. The number of instruction will be equal to the number of cycles of the non-optimized version. For this problem the number of cycles is 31105020. Therefore $T_{exec}$ for software in 1GHz computer is $31105020/(1024 \times 1024 \times 1024)$. This is based on the assumption that there is no cache miss and pipeline is always full so that the throughput is 1 instruction per cycle. It will take 0.029s for the software to complete the task. So for pilchard without reconfiguration, $T_{comm} + T_{proc}$ needs to be smaller than 0.029s. $T_{comm}$ for pilchard calculated in program analysis was 0.001s. Therefore, $T_{proc}$ needs to be smaller or equal to 0.028s to obtain speedup. Pilchard needs to be run at either 100MHz or 133MHz. Assume it is running at 133MHz, then in 0.028s, there will be 133MHz $\times$ 0.028s = 3904897 cycles available for data processing in order for pilchard to meet the performance of software. Therefore, we must achieve parallelism of averaging $31105020/3904897 = 7.966$ instructions per cycle. Which looks possible if we can run the pipeline version on pilchard at the preferred speed. However, the highest speedup of the parallel version on RC1000 is 4.76. In order for RC1000 to meet the performance of 1GHz PC, $T_{proc} = T_{exec} - T_{comm}$ assuming zero reconfiguration time. $T_{proc}$ is 0.028s -0.01s = 0.018s. If 4.76 speedup is achieved, that means we have to execute $31105020/4.76 = 6534668$ cycles in 0.018s. Thus requiring the clock speed with 367 MHz! Currently, RC1000 can run at most 100MHz. So RC1000 probably won’t be able to speedup the program at all. However, is it the end of story? No! We can still thing of the following questions:

1. The current design just make use of 1/4 of the Virtex2000E chips. That means in principle, if I can use all the resources. It can work at 4 grid at the same times. Which even in the non-pipeline version, it will achieve 4.76 $\times$ 4 speed up = 19 times less cycle! Moreover the
biggest chip available now has 4 times more gates than virtex2000E which mean we can achieve another 4 times speedup, i.e. 16 grids at the same time, which means 76 times speedup.

2. Not yet! Assume if we have infinite memory, then we can save both the source images and target of all resolutions into the memory, for each execution, we just need to pass the new control points from the host to FPGA and FPGA just need to pass back the similarity and derivatives back to the host. The data involved is then $802656 - 2$ source image - target image $= \cdot$. The $T_{comm}$ will become $0.00012s$ for a PCI bus. and $0.000015s$ for Pilchard! image what the communication cost will be if a device can utilize DDR ram bus or RDRAM bus? Therefore, we can regard $T_{comm}$ as 0.

Now if we compare the result again with the software version. The $T_{exec}$ for hardware will be $31105020cycles/76 = 409276$ cycles. For the device to be run as fast as hardware, the clock speed need to be $409276/0.029 = 14$MHz! Therefore, if the assumption holds. Only 14 MHz is needed. So what I can work on now is try to modify the communication method so that virtually the constraint by memory is eliminate.

We can conclude that for Pilchard, it is possible to speed up the program if we can run over 8 instructions per clock cycle. But for RC1000, a extremely high clock speed will be needed if we want any speed up at all.

4.5 Summary

In this chapter, I have run through the hardware development steps by using 2-D Gel Image Processing as an example. After this chapter, readers should know how various high-level hardware development techniques are applied throughout the development process.
Chapter 5

Conclusion

5.1 Assessed Achievement

1. I have also discovered some possible ways of automation of converting software code into high performance hardware design. For example, a simple procedure is introduced to aid the process of rearranging the sequence of the software code which can then group to run in parallel in minimum number of cycles.

2. I have developed a systematic approach for high level hardware development aimed at high performance. I divided the development process into 5 phrases.

3. By going through these phrases step by step, I successfully converted the complicated 2D gel image processing into several workable hardware versions in RC1000. For each application of a different optimization technique, there is significant improvement from the previous version.

5.2 Future Works

The following are the future works with the projects:

- develop an efficient time partitioning method to enable gel image processing implemented on limited memory device.

- develop a tool/algorithm which can automated implementation of optimal pipelining.

- utilize the development techniques to implement more applications on even more time demanding devices

- try floating point implementation for the whole 2D gel image processing.
• modify the communication method for gel image processing.
Bibliography


