Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2011-04

M.Sc. Thesis

RTL Implementation of an Optical Flow Algorithm (Lucas) Using the Catapult C High-Level Synthesis tool

Xianli Ren

Abstract

With the development of the technology, today's digital systems' growing design complexity has outpaced the traditional RTL design flow. The manual steps of micro-architecture definition, hand written RTL, simulation, debug and area/speed optimization through RTL synthesis are becoming more and more time consuming that gives the call of higher level abstraction in digital design. Catapult C synthesis tool, a C/C++ based hardware synthesizer, was released by Mentor Graphics as a solution of high complex digital system design. With this tool, designers are able to describe a complex system in a more productive abstraction level and Catapult C will generate an accurate RTL description turned to the target technology.

This thesis presents a practical introduction to C/C++ based high-level synthesis with Catapult C synthesis tool including tips of writing efficient synthesizable C/C++ code presented. In the design work of this thesis, the optical flow algorithm "Lucas" is implemented into hardware by Catapult C. The simulation results shows that with the clock frequency of 100MHz, the generated hardware has a minimum latency of 133.46ms for processing three images, which means it can reach a processing speed of 22.47 frames per second.

RTL Implementation of an Optical Flow Algorithm (Lucas) Using the Catapult C High-Level Synthesis tool

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Microelectronics

by

Xianli Ren
born in Hangzhou, China

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"RTL Implementation of an Optical Flow Algorithm (Lucas) Using the Catapult C High-Level Synthesis tool"** by **Xianli Ren** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: My Graduation Date

Chairman: _____
prof.dr.ir. A.J. van der Veen

Advisor: _____
dr.ir. Rene van Leuken

Committee Members: _____
dr.ir. A.V. Genderen

_____

Abstract

With the development of the technology, today's digital systems' growing design complexity has outpaced the traditional RTL design flow. The manual steps of micro-architecture definition, hand written RTL, simulation, debug and area/speed optimization through RTL synthesis are becoming more and more time consuming that gives the call of higher level abstraction in digital design. Catapult C synthesis tool, a C/C++ based hardware synthesizer, was released by Mentor Graphics as a solution of high complex digital system design. With this tool, designers are able to describe a complex system in a more productive abstraction level and Catapult C will generate an accurate RTL description turned to the target technology.

This thesis presents a practical introduction to C/C++ based high-level synthesis with Catapult C synthesis tool including tips of writing efficient synthesizable C/C++ code presented. In the design work of this thesis, the optical flow algorithm "Lucas" is implemented into hardware by Catapult C. The simulation results shows that with the clock frequency of 100MHz, the generated hardware has a minimum latency of 133.46ms for processing three images, which means it can reach a processing speed of 22.47 frames per second.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Rene van Leuken for his guidance and help when I met difficulties in my progress. I can remember there was one time when I got stuck in by several problems, I was very anxious, Prof. Rene van Leuken called up everyone who has used the same tool as I have to discuss my problems and fixed them.

Next, I want to thank Tao Xu, a Phd student in Circuit and System group, he is very kind and always willing to help me. Also I appreciate Mr. Antoon Frehe's help on my server accesses.

I am grateful for my friend Vashishth Krishan Chaudhri, our daily coffee break was a good memory and I hope his mother can get better. Also thanks to my friend Guanyu Yi for helping me on my Latex learning.

I want to thank my roomate Junfeng Jiang and other Chinese and international friends, I appreciate their help and companionship during my study.

Finally, I want to thank my parents and brother for their endless love and support.

Xianli Ren
Delft, The Netherlands
My Graduation Date

Contents

Abstract v

Acknowledgments vii

1 Introduction 1
- 1.1 The need of high-level synthesis . . . 1
- 1.2 Brief introduction of optical flow algorithm . . . 2
- 1.3 Thesis goals . . . 3
- 1.4 Contributions & Results . . . 3
- 1.5 Thesis outline . . . 3

2 From C/C++ to Hardware Description Language 5
- 2.1 About Catapult C Synthesizer . . . 5
- 2.2 C/C++ mapping to HDLs . . . 5
  - 2.2.1 Differences between C/C++ and HDLs stuctures . . . 5
  - 2.2.2 Data types . . . 6
  - 2.2.3 Interfaces . . . 8
  - 2.2.4 Hierarchical design . . . 8
- 2.3 Efficient synthesizable C/C++ writing style . . . 9
  - 2.3.1 Unsynthesizable writing styles . . . 9
  - 2.3.2 Coding with better synthesizability . . . 11
  - 2.3.3 Loop controls . . . 14
- 2.4 Summary . . . 16

3 Optical flow algorithm "Lucas" 19
- 3.1 Optical flow method . . . 19
  - 3.1.1 Introduction of optical flow method . . . 19
  - 3.1.2 Principle of optical flow algorithm . . . 21
- 3.2 "Lucas" algorithm . . . 23
  - 3.2.1 Principle of "Lucas" algorithm . . . 23
- 3.3 Summary . . . 26

4 System architecture design 27
- 4.1 Code clean up . . . 27
  - 4.1.1 System flow chart . . . 27
  - 4.1.2 Stackblur algorithm . . . 28
  - 4.1.3 Compute derivative . . . 28
  - 4.1.4 Compute velocity . . . 28
- 4.2 Hardware system architecture . . . 29
  - 4.2.1 System architecture . . . 29
  - 4.2.2 Datatype redefine . . . 31
  - 4.2.3 Block interfaces . . . 31

4.2.4 Memory access . . . 32
4.2.5 Use of Catapult C build in library . . . 33
4.2.6 Parallel design . . . 33
4.2.7 Testbench design . . . 34
4.3 Summary . . . 35

5 Code conversion and simlulation results **37**
5.1 Code conversion . . . 37
5.1.1 Import design . . . 37
5.1.2 Apply constraints . . . 38
5.1.3 Scheduling . . . 39
5.1.4 Generating HDL code . . . 39
5.1.5 Schematics . . . 40
5.1.6 reports . . . 41
5.2 Simulation results . . . 42
5.3 Different design constraints . . . 44
5.4 Compare with hand write VHDL code . . . 46

6 Conclusion **51**
6.1 Summary of writing synthesizable C/C++ code . . . 51
6.1.1 Hardware descriptive ability of C/C++ . . . 51
6.2 Summary of Catapult C . . . 51
6.3 Summary of results . . . 52
6.4 Future work . . . 52

List of Figures

1.1 Traditional RTL design flow . . . 1
1.2 Catapult C design flow[7] . . . 2

2.1 Basic C/C++ data types and corrsponding representation in high-level synthesis[21] . . . 7
2.2 Bit-accurate Algorithm C data types[22] . . . 8
2.3 Inferring serial hardware[12] . . . 11
2.4 Inferring parallel design[13] . . . 12
2.5 Common sub-expression allow hardware reuse[14] . . . 12
2.6 Loop using one iterator[15] . . . 13
2.7 Loop using multiple iterators[16] . . . 14
2.8 Conditional “if” creates dependency chain[17] . . . 15
2.9 Conditional “if” splits dependency chain[18] . . . 15
2.10 Loop unrolling[8] . . . 16
2.11 Loop merging[9] . . . 16
2.12 Loop pipelining[10] . . . 17

3.1 Motion field . . . 20
3.2 Optical flow field[4] . . . 21
3.3 Optical flow field in images . . . 22
3.4 principle of “lucas” algorithm . . . 24
3.5 Neighbourhood pixel weight . . . 25

4.1 System flow chart . . . 27
4.2 Principle of stackblur algorithm . . . 28
4.3 Compute derivatives . . . 29
4.4 Hardware system architecture of “Lucas” algorithm . . . 30

5.1 Import design . . . 37
5.2 Setup design . . . 38
5.3 Design constraints . . . 39
5.4 Gantt chart of stackblur block(partly) . . . 40
5.5 Operation element details . . . 41
5.6 Output file after HDL generation . . . 42
5.7 Top-level RTL schematic . . . 43
5.8 First page of Top core inside RTL schematic . . . 44
5.9 Critical path schematic . . . 45
5.10 Area score . . . 45
5.11 Latency, clock and loop report . . . 46
5.12 C testbench result . . . 47
5.13 Modelsim simulation waveform . . . 48
5.14 Solution table . . . 49
5.15 Bar chart . . . 49

5.16 XY plot chart 50

6.1 Synthesis result 52
6.2 Hierarchical synthesis makes function overlap[11] 53

List of Tables

5.1 Simulation result comparison 43
5.2 Comparison of previous hand wirte deisgn and this design 47

1 Introduction

With the advance of semiconductor technology, the digital system complexity has been increasing so fast that tranditional register-transfer level (RTL) design flow can hardy maintain its design productivity and efficiency. Thus improving the RTL design flow has become a hot spot in research.

1.1 The need of high-level synthesis

High-level synthesis, sometimes also called behavioral level synthesis, electronic system level synthesis or algorithmic synthesis, is a technology used to interprete algorithmic level design into hardware architecture.

The traditional RTL design flow is shown in Figure 1.1, we can see that when the design integration scales up, all the manual work would become more and more complex and time consuming.

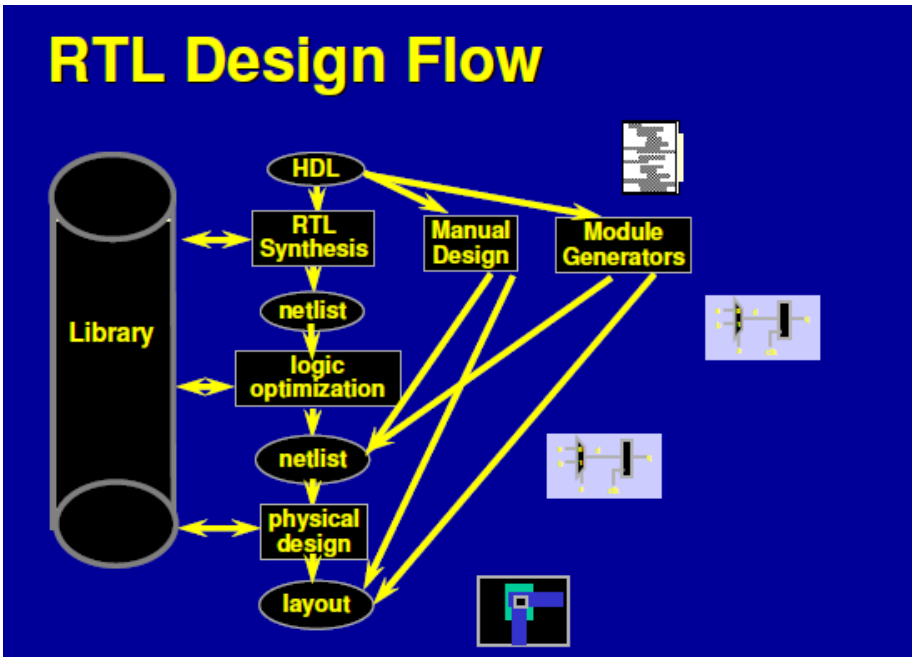

Figure 1.1: Traditional RTL design flow

In order to keep or even increase designers' productivity, high-level abstraction models are introduced and used without concern of what is inside the blocks and how could the blocks be implemented. Many of Electronic Design Automation solution companies

have presented their own high-level synthesis tools, including the ability of determining the data transfering, operation timing, storage as well as high-level optimization options to help designers to balance the trade-off between area or resource usage and latency. In this thesis, one of the tools called Catapult C, developed by Mentor Graphics is introduced and used for hardware implementation of the Optical Flow Algorithm, "Lucas".

Figure 1.2 shows the new RTL design flow with the help of Catapult C. Here we can see that the Micro-architecture definition, RTL design and RTL area/timing optimization could be automated by Catapult Synthesis and no more manual work maybe needed.

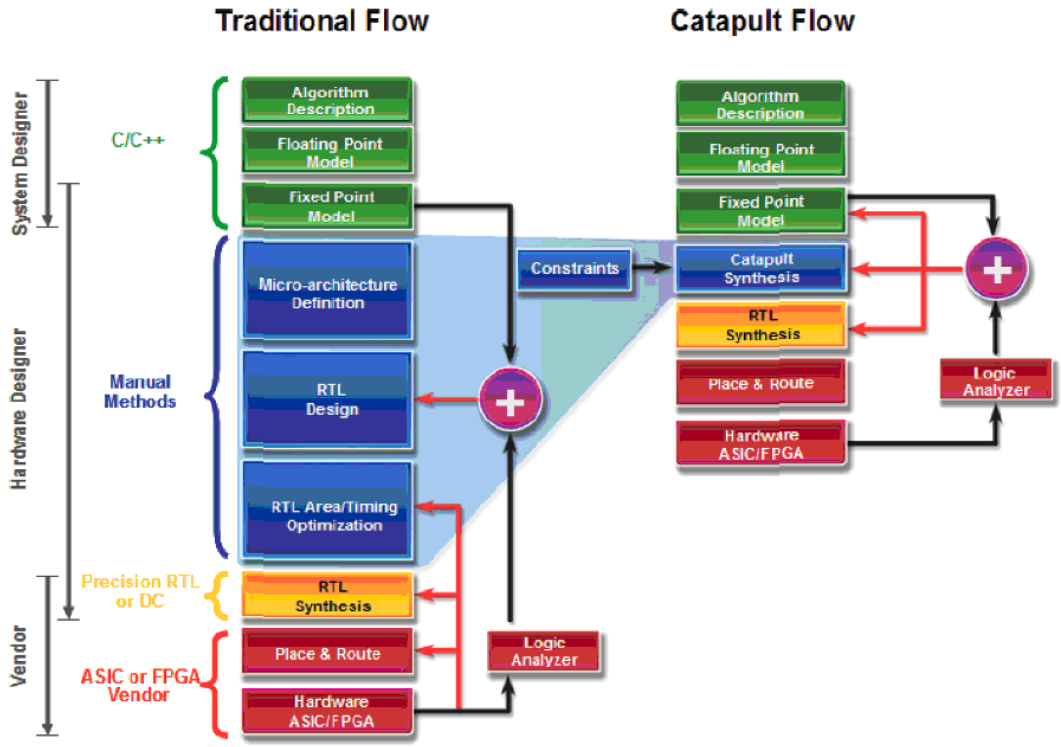

Figure 1.2: Catapult C design flow[7]

1.2 Brief introduction of optical flow algorithm

Optical flow represents the distribution of apparent velocities of movement of brightness patterns in an image. The concept of Optical Flow was originated from the study of human visual system, the term "optical flow" was first indroduced by American psychologist James J.Gibson in his paper "$The Perception of the Visual World$"[19] in 1950.

The real study of Optical Flow Computation began after 30 years in the early

1980s by *Horn&Chunck*[6]*andLucas&Kanade*[5]. Until now, varieties of Optical Flow Algorithm have been developed and used in the field of computer vision.

1.3 Thesis goals

This thesis has the following goals.

- Study the transformation from C/C++ to RTL hardware in the perspective of High-Level Synthesis.
- Study the capabablity of the High-Level Synthesis tool, Catapult C.
- Design the hardware of "Lucas" algorithm in hardware-oriented C code.
- Finally, use High-Level Synthesis tool Catapult C to convert the high-level C code into RTL code and verify.

1.4 Contributions & Results

The major contributions of this thesis project are listed in below:

- A methodology and coding tips for writing efficient synthesizable C code.
- Developed synthesizable "Lucas" C code.
- Converted the synthesizable C code to RTL and verified its correctness.
- Optimized the C code by adding design constraints and compare the results with a hand written solution.

1.5 Thesis outline

The rest of the thesis is organized as follows:

Chapter 2 presents the way that C/C++ code is being projected to hardware. After that, rules of how to write efficient hardware-oriented C/C++ code are disscussed.

Chapter 3 presents the principle of optical flow algorithm "Lucas" in detail.

Chapter 4 presents the system architecture as well as the design flow of this project.

Chapter 5 shows the Catapult C RTL implementation flow as well as the simulation results.

Chapter 6 is the conclusion.

From C/C++ to Hardware Description Language

2

High-level synthesis technology is introduced in order to reduce the increasingly growing complexity of ASIC design, with its help the designers are able to raise the design abstract level and put more effort on behavioral level (C/C++) design and optimization.

However, not all C/C++ code can be converted to RTL level description and not all synthesizable C/C++ code can be successfully converted into efficient RTL descriptions.

Thus, in this chapter, C/C++ based high-level synthesis tool Catapult C will be first introduced in 2.1, rules of C/C++ to HDLs based on Catapult C will be presented in 2.2, tips and advices about how to write efficient synthesizable C/C++ code will be discussed in 2.3.

2.1 About Catapult C Synthesizer

Catapult C is a C/C++ based high-level synthesis tool developed by Mentor Graphics. It has the following main features:

- C/C++ editor and compiler.
- C/C++ algorithm synthesizability analysis.
- Architecture constraints.
- RTL hardware generation and optimization.
- SystemC Verification
- Automatic generation of SystemC testbench

2.2 C/C++ mapping to HDLs

In this part, all the rules of C/C++ to HDLs will be based on Catapult C.

2.2.1 Differences between C/C++ and HDLs stuctures

As we know, HDL is well known for its ability of describing parallel operations, while C/C++ is for serial operation description, they have the following features in their structures:

- C/C++ consists of functions, while HDLs are made up of modules or entities. Thus, the main task of C/C++ based high-level synthesis is to convert C/C++ functions to high performance, multi-threaded RTL modules. Subfunctions in C/C++ corresponds to the sub modules/entities.
- In C/C++, a function call is made when a subfunction is needed, the function variables or return values or even global variables can be used as communications between one function and its outside world. Accordingly, in HDLs modules/entities can be instantiated when needed, they use their input and output ports to communicate with outside.

  However, their nature is different. First, the instantiation in HDLs reflects physical interconnections among the ports. Second, in C/C++ one function can be repeatively and limitlessly called while in HDLs each time an instantiation of the same module leads to creation of a new hardware block.
- There is no timing in C/C++, there is no concept of clocking or synchronous/asynchronous control. To solve this problem, C/C++ based high-level synthesizer will create the timing according to the data flow and timing constrains.
- In C/C++, operations are executed line by line in serial while in HDLs there are descriptions for parallel operations. Thus, one other important task for C/C++ based high-level synthesizer is to analyze the data dependencies and parallel the operations automatically.

2.2.2 Data types

This part presents the basic data types in C/C++ and the way they will be mapped into bit-accurate HDL data types. Also, the algorithm C data type will be introduced here.

1. standard C/C++ data types

   The table in Figure 2.1 shows the basic C/C++ integral data types and their corresponding data types in HDLs in high-level synthesis. As we see standard C/C++ data types don't have the ability of bit-accurate precision control, neither can they give fixed-point fractional description. Thus there are several C/C++ libraries being created to solve this problem, such as Algorithm C and System C data types.

2. Algorithm C data types

   Algorithm C data types are developed for Catapult C users to write arbitrary-length bit-accurate algorithms that can be synthesized into hardware.

   There are two commonly used Catapult C data types: $ac\_int < W, S >$ and $ac\_fix < W, I, S, Q, O >$.

   For $ac\_int < W, S >$, the precision is determined by the template parameter W, which is the integer that indicates the bit-width; and S, which is a boolean

Table 2-1. Integral data types

C++ Code	VHDL	Verilog	Signed
`bool MY_Var;`	`STD_LOGIC MY_Var;`	`reg MY_Var;`	No
`char MY_Var; //avoid signed char MY_Var; signed char int MY_Var;`	`STD_LOGIC_VECTOR (7 downto 0) MY_Var;`	`reg [7:0] MY_Var;`	Yes
`unsigned char MY_Var; unsigned char int MY_Var;`	`STD_LOGIC_VECTOR (7 downto 0) MY_Var;`	`reg [7:0] MY_Var;`	No
`short MY_Var; signed short MY_Var; signed short int MY_Var;`	`STD_LOGIC_VECTOR (15 downto 0) MY_Var;`	`reg [15:0] MY_Var;`	Yes
`unsigned short MY_Var; unsigned short int MY_Var;`	`STD_LOGIC_VECTOR (15 downto 0) MY_Var;`	`reg [15:0] MY_Var;`	No
`int MY_Var; signed MY_Var; signed int MY_Var;`	`STD_LOGIC_VECTOR (31 downto 0) MY_Var;`	`reg [31:0] MY_Var;`	Yes
`unsigned int MY_Var; unsigned MY_Var;`	`STD_LOGIC_VECTOR (31 downto 0) MY_Var;`	`reg [31:0] MY_Var;`	No
`long MY_Var; signed long MY_Var; signed long int MY_Var;`	`STD_LOGIC_VECTOR (31 downto 0) MY_Var;`	`reg [31:0] MY_Var;`	Yes
`unsigned long MY_Var; unsigned long int MY_Var;`	`STD_LOGIC_VECTOR (31 downto 0) MY_Var;`	`reg [31:0] MY_Var;`	No
`long long MY_Var; signed long long MY_Var; signed long long int MY_Var;`	`STD_LOGIC_VECTOR (63 downto 0) MY_Var;`	`reg [63:0] MY_Var;`	Yes
`unsigned long long MY_Var; unsigned long long int MY_Var;`	`STD_LOGIC_VECTOR (63 downto 0) MY_Var;`	`reg [63:0] MY_Var;`	No

All variables created by Catapult will have the LSB in the 0 bit location.

Figure 2.1: Basic C/C++ data types and corrsponding representation in high-level synthesis[21]

to determine if it is a signed or unsigned integer. S=true means it is a signed integer, S=false means it is a unsigned number.

For fixed-point data type $ac\_fixed < W, I, S, Q, O >$, there are five template parameters, W is an integer and gives the whole bit-width, I is an integer and gives the integer parts bit-width, S is a boolean and determines if it is signed or not, Q and O are the quantization and overflow modes, they can be left blank.

The table in Figure 2.2 shows the basic bit-accurate Algorithm C data types and their numerical ranges.

Table 2-1. Numerical Ranges of ac_int and ac_fixed

Type	Description	Numerical Range	Quantum
ac_int<W, false>	unsigned integer	0 to 2^W- 1	1
ac_int<W, true>	signed integer	-2^{W-1} to 2^{W-1}- 1	1
ac_fixed<W, I, false>	unsigned fixed-point	0 to (1 - 2^{-W}) 2^I	2^{I-W}
ac_fixed<W, I, true>	signed fixed-point	(-0.5) 2^I to (0.5 - 2^{-W}) 2^I	2^{I-W}

Figure 2.2: Bit-accurate Algorithm C data types[22]

2.2.3 Interfaces

As is mentioned before, C/C++ function variables or return values or global variables can be used as communication data types among functions, in order to precisely synthesize a function into a hardware block, designers are recommended to use variables and return values as interfaces.

1. Input ports

   Function variables that are pass-by-value will be synthesized into input ports. Functions variables that are pass-by-reference but only be read and never written in the function will also be synthesized into input ports.

2. Output ports

   Function variables that are pass-by-reference but only be written and never read will be synthesized into output ports. Function return value with also be synthesized into output ports.

3. Inout ports

   Function variables that are pass-by-reference and be both written and read will be synthesized into inout ports.

Here is an example of the top level.

```
#pragma design top                    // indication of top level block
void top(
unsigned char f1i[316*252],           //input port
m_type norm_vels1_y[316*252]          //output port
)                                     //m_type is a user defined datatype
/*---Whether it's input or output is determined by the function body---*/
```

As we can see, this top level function has no return value and all variables are arrays, which means they are pass-by-references.

2.2.4 Hierarchical design

When it comes to building a system which contains mutiple blocks that can run in concurrent or sequential, designers should apply HLS contrains (in Catapult GUI) as

well as some recommeded coding styles.

The code below shows part of the top level description of this thesis project, the hierarchy and dependency of the blocks can be easily identified from the way it is written.

```
#pragma design top                          //indication of top level
void top(

)
{

    /*-----------data ports for blocks----------*/
    int f1o[316*252],f2o[316*252],f3o[316*252];    //output of stackblur
    int Ix[316 * 252],Iy[316 * 252],It[316 * 252];    //der_3x3 output
    /*--------------stack blur-----------------*/
    stackblur(f1i,f2i,f3i,f1o,f2o,f3o);
    /*--------------compute derivitives-------------------*/
    compute_ders_3x3(Ix,Iy,It,f1o,f2o,f3o);
    /*--------------compute velocities--------------------*/
    compute_vels(Ix,Iy,It,full_vels_x,full_vels_y,norm_vels1_x,
       norm_vels1_y);

}
```

As we see, there are three sub-blocks inside the top block, stackblur, compute_ders_3x3 and compute_vels, each of them connectes to others by sharing the same arrays as function variables. According to their data dependencies, we can tell that they are connected in serial.

2.3 Efficient synthesizable C/C++ writing style

There is no doubt that C/C++ based high-level synthesizer will largely reduce the design cycle , but it can not be perfect in every aspect. Synthesiability is becoming a more important issue for Catapult C users.

Here in this section, unsynthesizeble writing styles will be first introduced in 2.2.1 and efficient synthesizable C/C++ writing styles will be presented in 2.2.2.

2.3.1 Unsynthesizable writing styles

Here we list the unsynthesizable writing styles.

1. Loops without finite bounds

   Loops without finite bounds can not be synthesized into hardware, thus Catapult C will examine in its starting point of synthesis and gives error. A synthesizable loop should have the following elements:

   - A constant start point.

- A constant stop point.
- A constant increment.

2. Global variables

   Global variables being used in hardware oriented C/C++ code leads to difficulty in compilation and synthesis, the hardware generated may not function as expected. Thus designers should avoid using global variables.

3. Dynamic interface mapping

   In C/C++ coding, the following kind of function head writing style is very common.

```
void function0(
unsigned char img[PIC_X][PIC_Y], //image pixel value
unsigned w,                      // width of the image
unsigned h,                      // height of the image
)
```

However, this is unsynthesizable. The array indexes PIC_X and PIC_Y are dynamic variables, which means the unsigned char variable of the function can be any one of the elements in the array (or menory), but we never know which one it is. Here are two synthesizable writing styles:

```
void function1(
unsigned char img_in,          // memory reading port
unsigned char img_out,         // memory writing port
unsigned address,             // memory address
bool    read_en,             // read enable
bool    write_en,            // write enable
unsigned w,                  // width of the image
unsigned h,                  // height of the image
)
```

```
void function2(
unsigned char img[w*h]        // The port size equals to the memory size
unsigned w,                  // width of the image
unsigned h,                  // height of the image
)
```

In function1, variable img_in, img_out, address, read_en, and write-en can be synthesized into reading, writing, address, read enable, and write enable ports respectively to connect with a dual port memory block. This style is more like a HDL description, though it is synsizable, but has lost the convenience of using high-level synthesis.

Function2 has simply used the whole array (memory) "img" as variables, then Catapult C will automatically create the read&write interface for this kind of writing style. Thus it is the preferable style.

Please note that the clock will be automatically created by the synthesizer.

2.3.2 Coding with better synthesizability

1. Using of parentheses in expressions

   Arithmetic expressions are costy as they consume a great many hardware. With the help of the C/C++ based high-level synthesizer, designers are able to manage the amount of hardware to be used by using parentheses properly in C/C++ code.

   Figure 2.3 and Figure 2.4 shows how parentheses affect the synthesis result.

   The version shown in Figure 2.4 has no parenthese used, which infers a serial design.

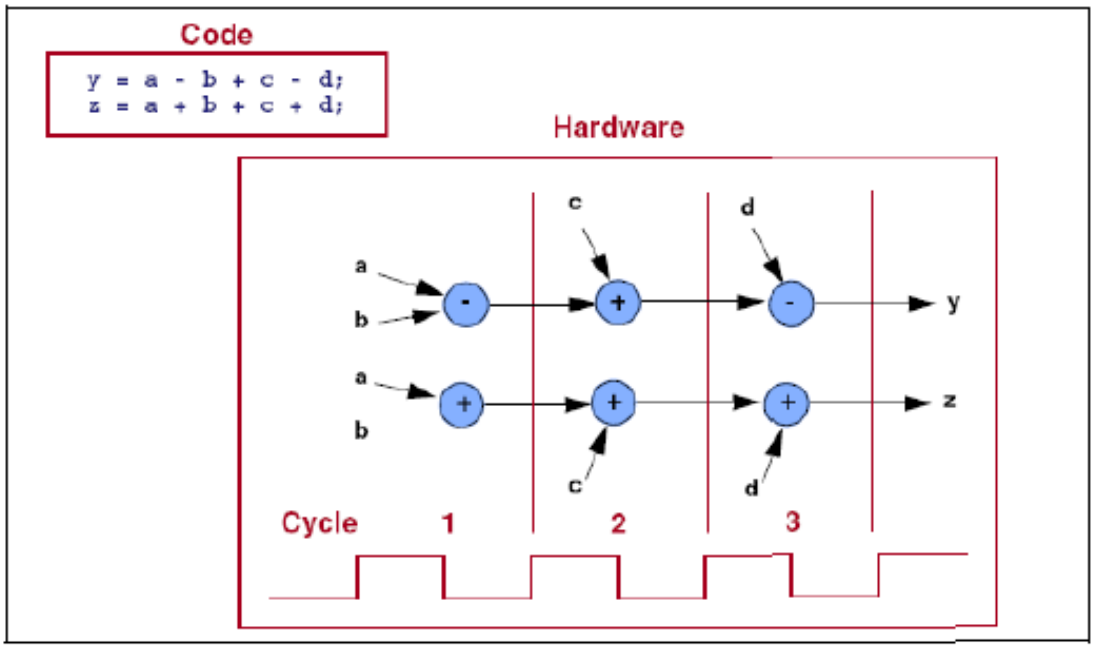

Figure 2.3: Inferring serial hardware[12]

   The version shown in Figure 2.5 has parentheses used, which infers a parellel design.

   Figure 2.5 shows that proper using of parentheses in sub-expression can reduce the resource usage.

2. Shorten the dependency chain

   A short dependency chain is always prefered to achieve small latency and to fit the design into a higher clock frequency.

   Now we present some of the typical suggestions can be followed to reduce the dependency chain.

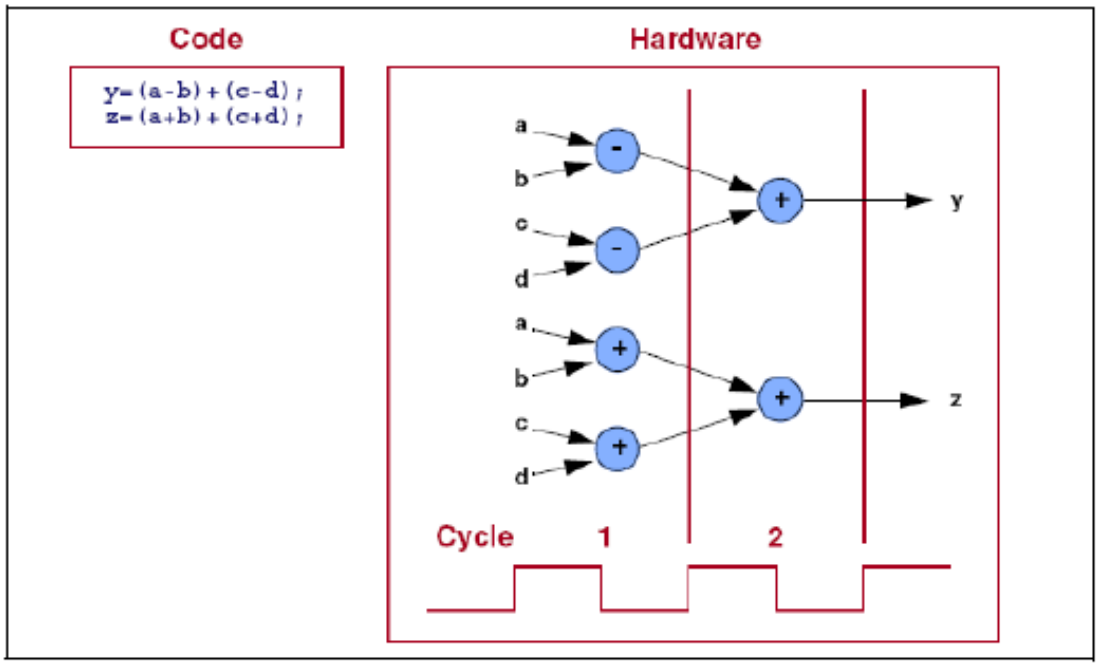

Figure 2.4: Inferring parallel design[13]

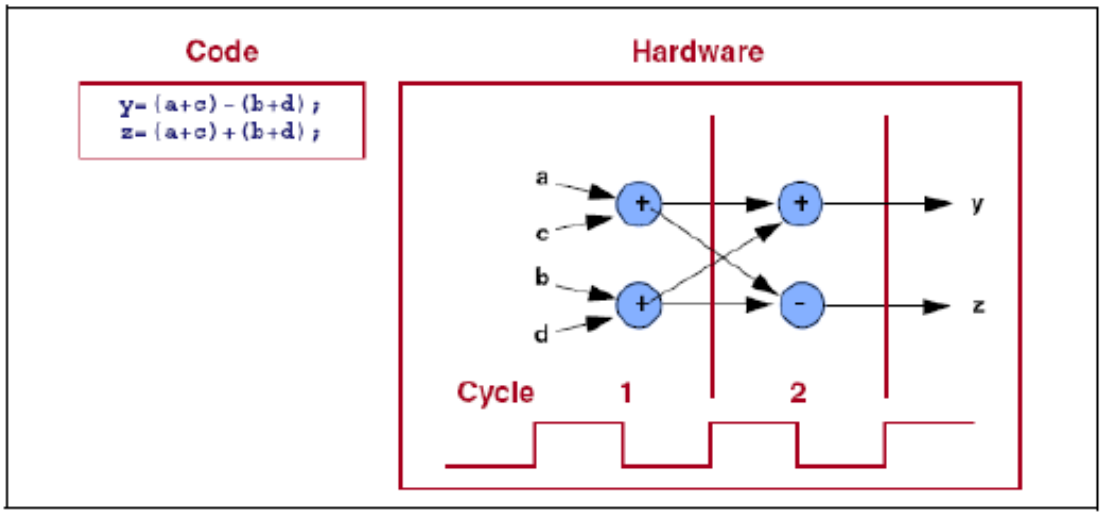

Figure 2.5: Common sub-expression allow hardware reuse[14]

The first suggestion is to use more iterators in a loop. Here is an example of using only one iterator or multiple iterators in Figure 2.6.

As we see in the one-iterator version, indexes of in1 and in2 are calculated by mod operator, which leads to a long dependency chain in this loop.

Figure 2.7 shows an example of multiple iterators being used.

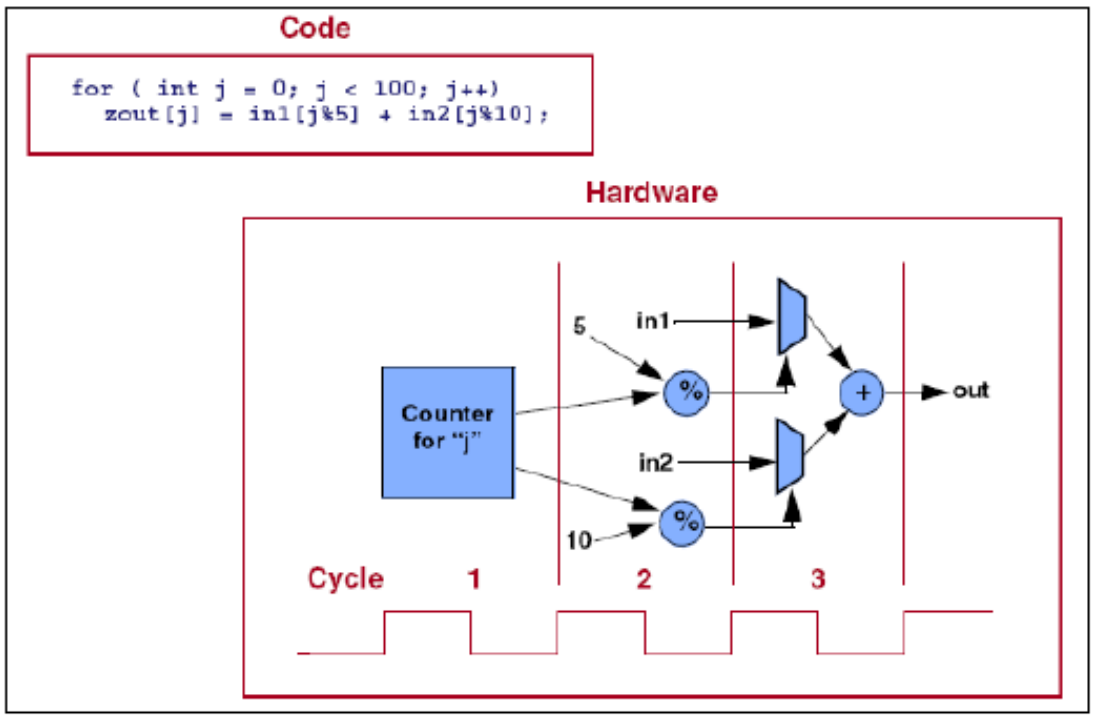

Figure 2.6: Loop using one iterator[15]

The version shown in Figure 2.7 with three independent loop iterators is faster than the previous version shown in Figure 2.6 because the dependency chain has been split into shorter ones, but please note that extra hardware is required to control the iterators.

In addition to using more loop iterators, writing the conditional logic in a proper way also helps in shortening the dependency chain. Using of "else" can generate mutually exclusive conditions and thus split the dependency chain.

Figure 2.8 shows an example of using conditional "if" logic.

Figure 2.9 shows an example of using conditional "else" logic.

As we can see, when the else is used, the synthesizer knows it is a mutually exclusive condition and will split the dependency chain into two, as a result, the design becomes faster.

The last suggestion is to use more small memory blocks instead of a few large ones. The reasons are simple. First more memory blocks means more datas can be read or write at the same time thus the dependeny chain can be shorten compared with when large memory is used, also smaller memory size means the multiplexer size can also be smaller.

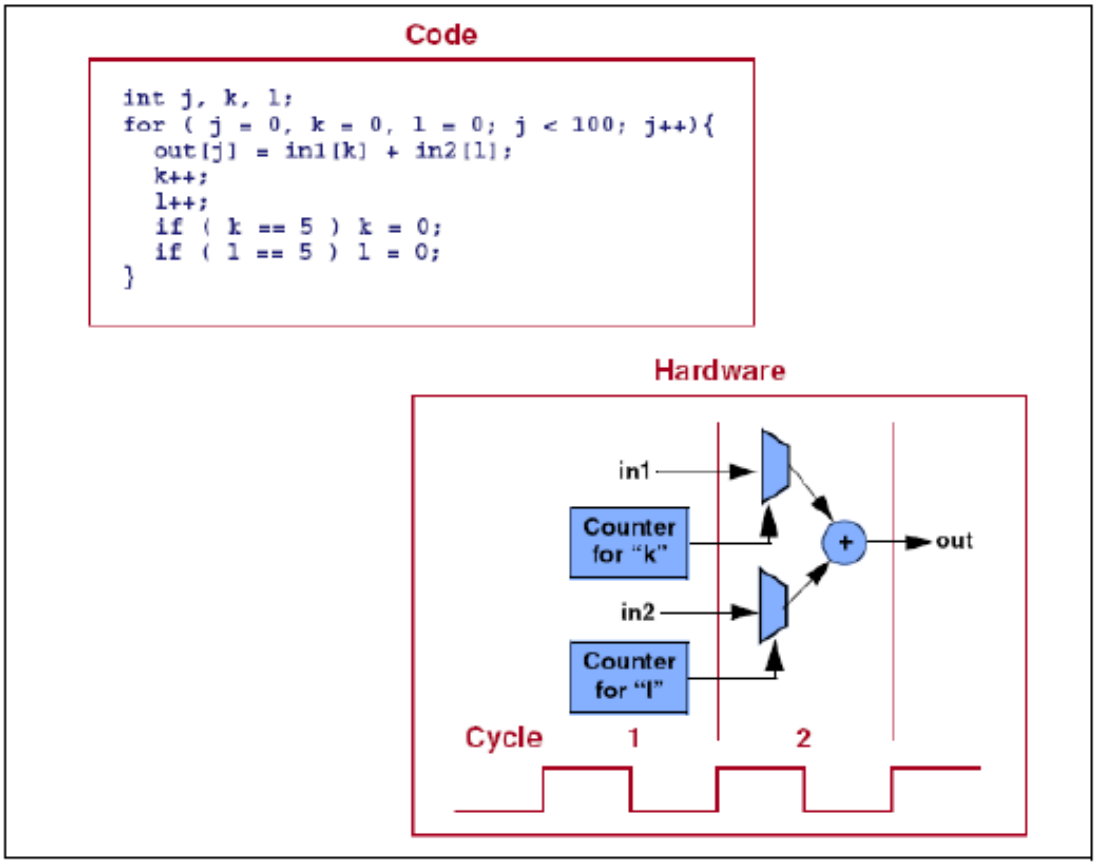

Figure 2.7: Loop using multiple iterators[16]

2.3.3 Loop controls

Catapult C provides loop control options as part of the architecture constraints. There are thre kinds of loop control: loop unrolling, loop merging and loop pipelining.

Loop unrolling is a technique that reduces the loop iteration numbers by flattening the loop body. Here in Catapult C, when you apply loop unrolling, you actually unrolled this loop fully such that only one iteration is left.
Here is an example shown in Figure 2.10, on the left side, the loop is left rolled, the synthesis result shows one adder is used and the latency is four cycle; while the right side shows when the loop is fully unrolled, in synthesis four adders are needed and the latency is one cycle.
If we say loop unrolling is a instruction-level parallelism, loop merging represents a loop-level parallelism. In loop merging, various sequential loops can be merged into one loop with the same functionality as the original loops. This technique helps to reduce latency and area consumption in the design.

In Catapult C, users just need to tick on the option to enable or unable the loop merging, if loop merging is enabled, Catapult will automatically find out whether

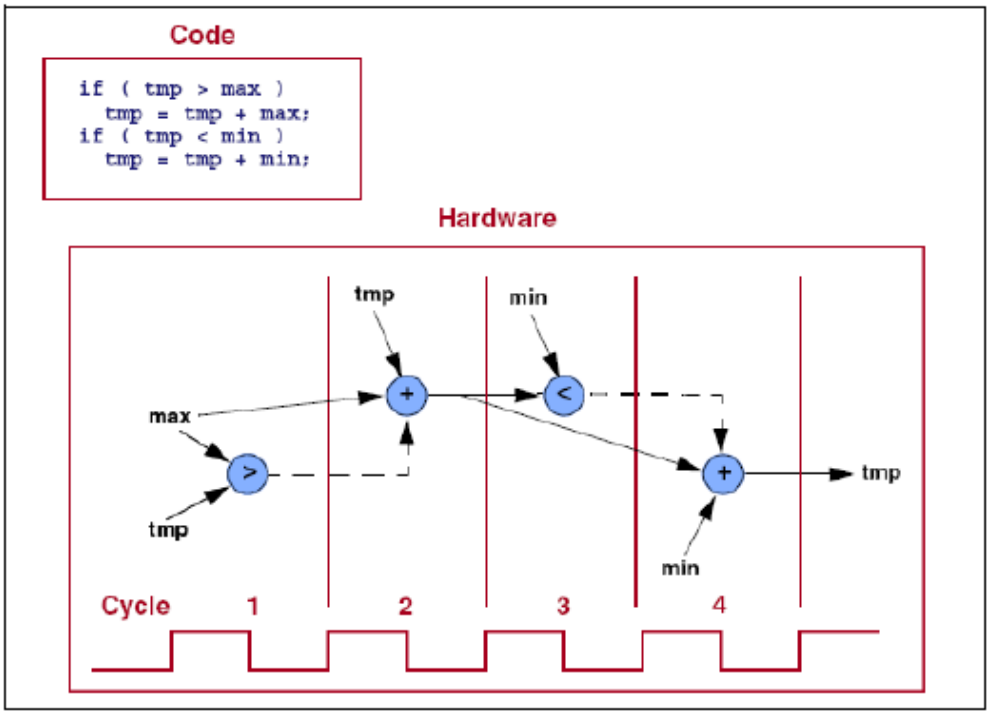

Figure 2.8: Conditional "if" creates dependency chain[17]

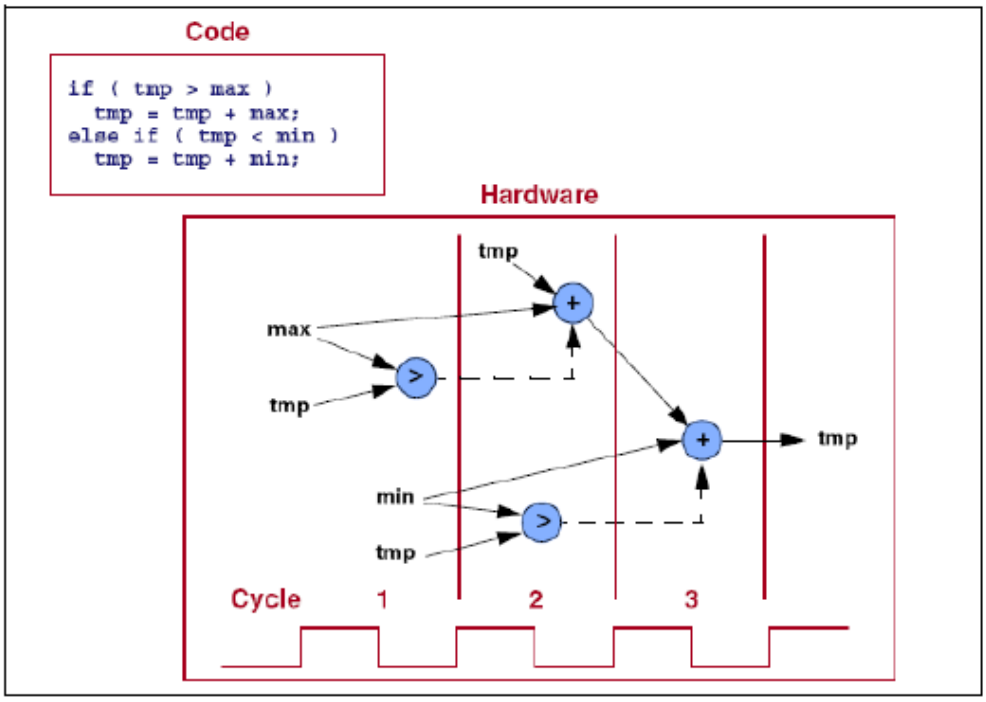

Figure 2.9: Conditional "if" splits dependency chain[18]

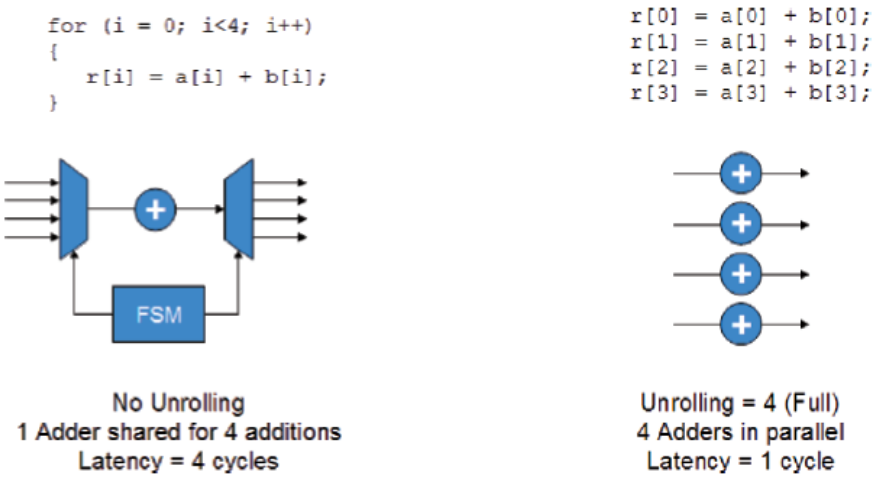

Figure 2.10: Loop unrolling[8]

and how can it be merged and do the merging according to the analysis result.
Figure 2.11 shows how sequential loops being merged. We can see that the second loop can be merged into the first bigger one.

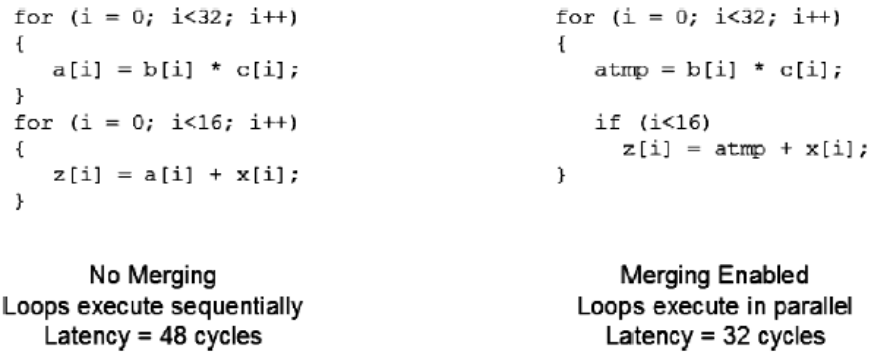

Figure 2.11: Loop merging[9]

Pipelining a loop can increase the throughput of the loop and is also very simple to do through Catapult C. What the users need to do is to determine whether to pipeline a loop or not and what the initial interval is.
Figure 2.12 shows the principle of pipeline.

2.4 Summary

C/C++ based high-level synthesizer Catapult C provide a new solution for digital system design, as a result, a new study area intended for helping the designers to write high quality hardware-oriented C/C++ code has been created. In the following chapters, an image processing algorithm "Lucas" will be implemented in hardware

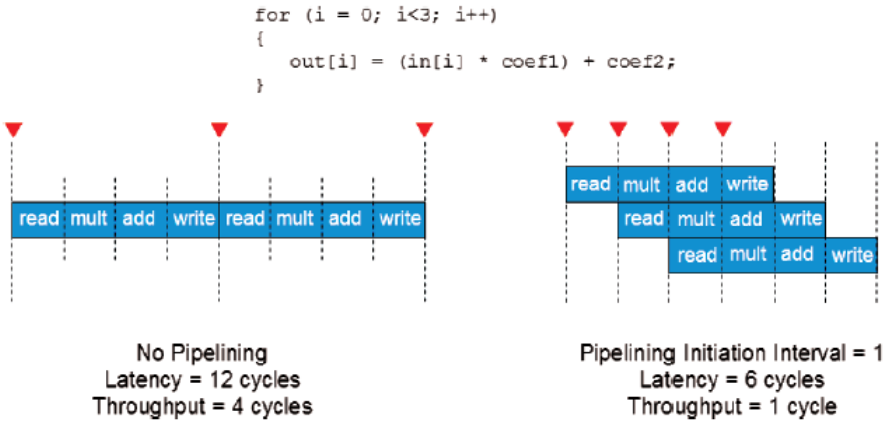

Figure 2.12: Loop pipelining[10]

using Catapult C, to give us a better understanding of how this C/C++ based high-level synthesizer works.

3 Optical flow algorithm "Lucas"

One important reseach area in the field of Computer Vision is analysis of the dynamic scenes through sequential images. Dynamic scenes refers to scenes with object movements inside, its not only a function of spatial positions, but also a function of time. The picture sequences taking from a dynamic scene are called dynamic image sequences. Between any neighboring frames, there are grayscale and color difference in some part of the pixels.

The basic task of dynamic scene analysis is to find out the movement information, detect and track the moving objects from the dynamic image sequences. Athough relevant study has been continued for decades, various kinds of motion estimation algorithms have been developed, finding an algorithm with good robustness, high accuracy and high performance is still a very challenging task.

Here in this thesis, the focus is not about finding out a better algorithm, but to get an efficient hardware solution in terms of processing speed by using high-level synthesis tool. Thus a mature and well developed object movement tracking algorithm, optical flow algorithm "Lucas", is analyzed and used. Optical flow algorithm will be introduced in section 3.1, the detailed analysis of "Lucas" algorithm will be presented in section 3.2.

3.1 Optical flow method

3.1.1 Introduction of optical flow method

As is mentioned in the introduction, the concept of optical flow is originally introduced by an American psychologist James J. Gibson, in his study of human visual system. Optical flow is the brightness pattern of the apparent motion of objects in a visual scene, it arises from relative motion between the object and the observer. When our eyes are looking at a moving object, its like continuous optical images flow through our retina, thus called optical flow.

When we talk about the optical flow method in computer vision, we are actually studying the optical flow field. To understand what optical flow field is, first we need to know another term, motion field. Motion field describes three-dimensional motion in the real world, including three-dimensional velocity in terms of velocity magnitude and its direction.

However, when we observe the world from a camera, what we see is a sequence of flat images, the three-dimensional real world has been squeezed into a two-

dimensional plane. To describe the motion of the objects in the images, optical flow field is used as a substitute of motion field.

Optical flow field is a two-dimensional velocity field presented in an image as a projection from the three-dimensional motion field. Once we have the optical flow field calculated, we know the object movements in the image, though it is not exactly the real world motion field, but its still useful. That is why optical flow method is regarded as one of the classical solutions for motion estimation.
Figure 3.1 shows an example of motion field and example of optical flow field is shown in Figure 3.2. Here we can see in Fig 3.2, optical flow method calculates the overall

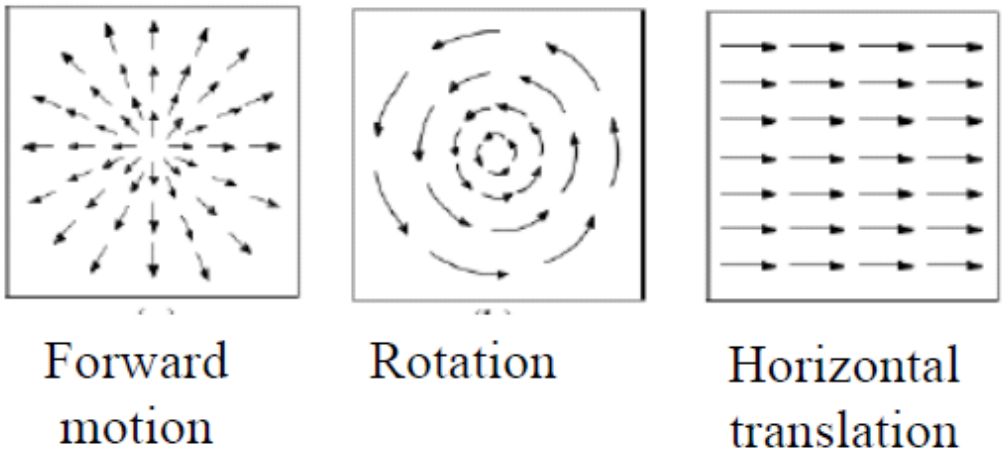

Figure 3.1: Motion field

movements in the image.
Below shows some of the applications of optical flow method:

1. Velocity Measurement

   Velocity measurement is one basic application of optical flow method. Assume a robot is travelling at a certain altitude, given the altitude of the robot, the velocity can be calculated by analyzing the pixel velocity.

2. Altitude Measurement

   As a converse of velocity measurement, altitude can be calculated from the velocity. Anyone who has been on a flight knows, when looking out of the window of an airplane, the higher we are, the slower we find the ground scene moves. Assume an aircraft moves with a constant speed, by analyzing the pixel velocity on pictures taken by downward facing camera, the altitude can be calculated.

3. Tracking

   Since optical flow method has no interest in identifying any specific object, the way of tracking should contain the background subtraction[1] part. After removing the

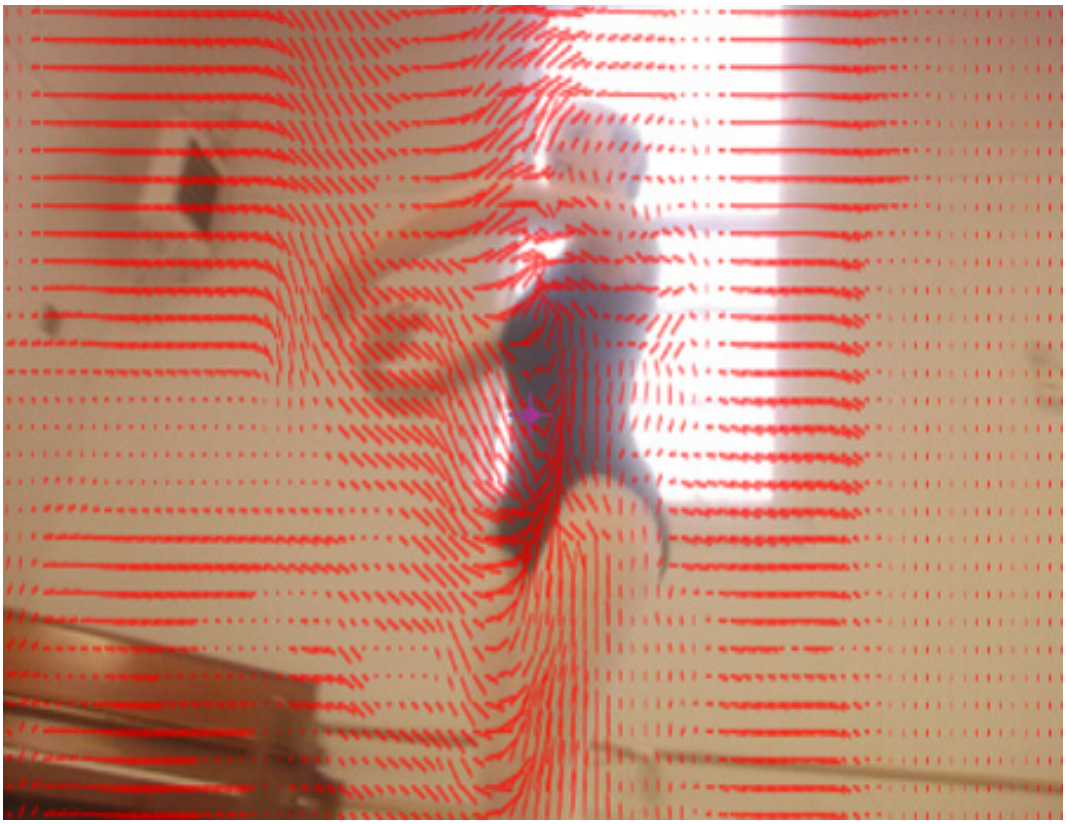

Figure 3.2: Optical flow field[4]

irrelevant parts, the rest of work lies in analyzing the pixel velocity along with the pixel to distant ratio.

4. Three-dimensional Scene Recovery

   As is mentioned before, optical flow field is a projection from the three-dimensional real world motion field. By analyzing the pixel movements, we can see the velocity field of different parts of an object in an image are different because of its shape and the placement.

5. Navigation

   Optical flow method can be used to supervise the navigation of robot[20]. With a camera put on the robot, by calculating the optical flow field of the observed environment continuously, we can get the obstacle positions, the robot heading direction, the time to collision and the depth.

3.1.2 Principle of optical flow algorithm

In optical flow theory, every pixel in an image has its brightness (or intensity value), the movements of the objects can be illustrated by the movements of the pixel intensity values in a sequence of images.

One critical prerequisite of optical flow algorithm is the assumption called brightness constancy. Let's explain it here. Assume there is a point, positioned in (x, y) and possesses an intensity value $I(x, y)$, after some time t, due to the movement of the object or the observer, what we find on the later frame is that this pixed has moved to somewhere else and possessed another position and intensity value $(x + \delta x, y + \delta y)$ or $(x(t), y(t))$ and $I(x + \delta x, y + \delta y)$ or $I(x(t), y(t))$.
Figure 3.3 shows the optical flow in different frames of images.

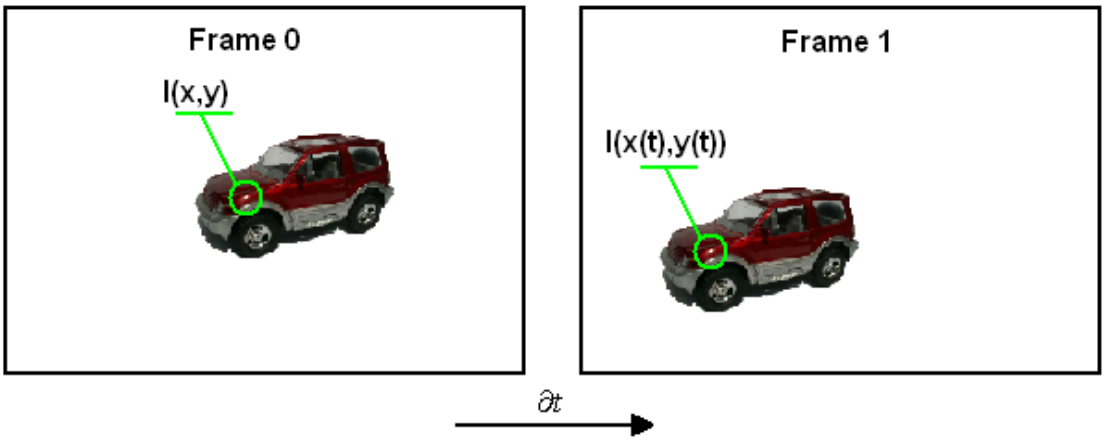

Figure 3.3: Optical flow field in images

Brightness constancy here means, given one point in an image taken from a dynamic scene, no matter where this point moves to in the following frames, we assume its intensity value will not change within some time.
Equation 3.1 shows the brightness constancy:

$$I(x, y) = I(x + \delta x, y + \delta y) = I(x(t), y(t)) \tag{3.1}$$

If we add time as another parameter, we get equation 3.2:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) = I(x(t), y(t), t + \delta t) \tag{3.2}$$

By applying Taylor series expansion, we get equation 3.3:

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + HighOrderTerms \tag{3.3}$$

Discard the high order terms and apply the brightness constancy, substitute equation 3.2 into equation 3.3, we can get the optical flow constraint equation, which is shown in equation 3.4.

$$\frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0 \tag{3.4}$$

Then both sides divided by δt, we get equation 3.5:

$$\frac{\partial I}{\partial x}\frac{\delta x}{\delta t} + \frac{\partial I}{\partial y}\frac{\delta y}{\delta t} + \frac{\partial I}{\partial t} = 0 \tag{3.5}$$

Where $\delta x/\delta t = u_x$ which is the velocity in x direction, $\delta y/\delta t = u_y$ the velocity in y direction, $\delta I/\delta t = I_x$, $\delta I/\delta t = I_y$, $\delta I/\delta t = I_t$. Then we can get the final version of constraint in equation 3.6.

$$I_x \times u_x + I_y \times u_y + I_t = 0 \tag{3.6}$$

By solving this function , we can have the optical flow field $\vec{u} = (u_x u_y)^T$ calculated. So what we actually have to do is calculating the intensity value derivitives(I_x, I_y, I_t) from a sequence of images taken from a dynamic scene.

3.2 "Lucas" algorithm

"Lucas" algorithm, abbreviation of "Lucas-Kanade" algorithm, developed by Bruce D. Lucas and Takeo Kanade, was regarded as a classical differential method for computing the optical flow field.

3.2.1 Principle of "Lucas" algorithm

As we know in the earlier section, calculating the derivitives of the intensity value is the essential part of solving the optical flow constraint function.

However, equation 3.6 has two variables u_x and u_y, which means even if we provide all the other parameters, the two terms can not be calculated by a single function, there must be other constraint functions. Here "Lucas" algorithm gives a solution.

"Lucas" algorithm assumes the pixels in a neighbourhood share the same optical flow field, which means we will get more than one constraint functions to solve two variables.

Figure 3.4 simply shows the principle of the "Lucas" algorithm. There are three prerequisites for this assumption:

1. Brightness constancy.

2. Frames have to be consecutive, time interval between each frame has to be small and the object movements should not be speedy.

3. Neighbourhood pixels have the similar motion and they keep together.

The above conditions have to be met, otherwise the accuracy of this algorithm will be dramatically affected.

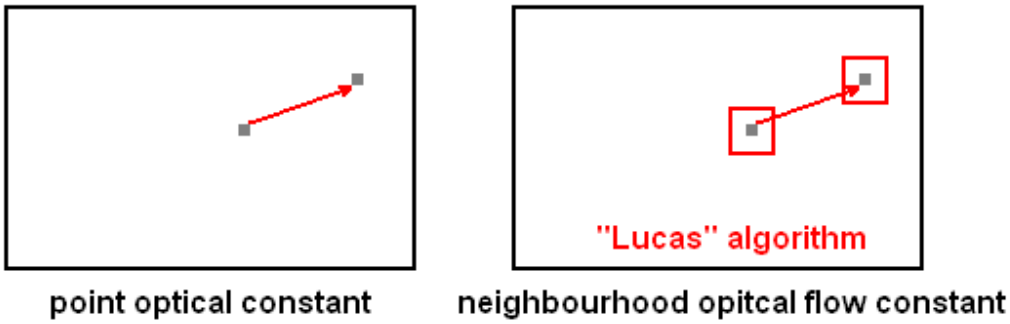

Figure 3.4: principle of "lucas" algorithm

In "Lucas" algorithm, we get the following equations:

$$\begin{aligned} I_{x1} \times u_x + I_{y1} \times u_y &= -I_{t1} \\ I_{x2} \times u_x + I_{y2} \times u_y &= -I_{t2} \\ &\dots \\ I_{xm} \times u_x + I_{ym} \times u_y &= -I_{tm} \end{aligned} \tag{3.7}$$

After transform the above into a matrix, we get equation 3.8.

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xm} & I_{ym} \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} = - \begin{pmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tm} \end{pmatrix} \tag{3.8}$$

Actually, solving equation 3.8 can not give us the right result, beause with the assumption that all neighbourhood pixels share the same velocity (optical flow), this set of linear equations are not determinant.

Thus we have to find some limits for this overdetermined set of linear funtions, equation 3.9 shows the new matrix with errors counted in. The errors come from the neighbourhood optical flow constant assumption.

$$\begin{pmatrix} I_{x1}W_1 & I_{y1}W_1 \\ I_{x2}W_2 & I_{y2}W_2 \\ \vdots & \vdots \\ I_{xm}W_m & I_{ym}W_m \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} + \begin{pmatrix} I_{t1}W_1 \\ I_{t2}W_2 \\ \vdots \\ I_{tm}W_m \end{pmatrix} = \begin{pmatrix} \varepsilon_1 W_1 \\ \varepsilon_2 W_2 \\ \vdots \\ \varepsilon_m W_m \end{pmatrix} \tag{3.9}$$

The parameters $W_1 - W_m$ represent the weights of the neighbourhood pixels' contribution in the calculation. Figure 3.5 shows an example of 3x3 neighbourhood pixel weights. Then we can apply the leastsquare solution to achieve the least errors. Here is the cost function:

Figure 3.5: Neighbourhood pixel weight

$$\varepsilon = \sum_{i=0}^{m} W_i^2 (I_{xi} \times u_x + I_{xi} I_{yi} \times u_y + I_{xi} I_{ti})^2 \tag{3.10}$$

To get the least ε, we calculate the derivatives and find out the zero point of it:

$$\begin{aligned} \frac{\partial \varepsilon}{\partial u_x} &= \sum_{i=0}^{m} W_i^2 (I_{xi}^2 \times u_x + I_{xi} I_{yi} \times u_y + I_{xi} I_{ti}) = 0 \\ \frac{\partial \varepsilon}{\partial u_y} &= \sum_{i=0}^{m} W_i^2 (I_{yi}^2 \times u_x + I_{xi} I_{yi} \times u_x + I_{yi} I_{ti}) = 0 \end{aligned} \tag{3.11}$$

Then we get the new equation:

$$\begin{pmatrix} \sum_{i=0}^{m} W_i^2 I_{xi}^2 & \sum_{i=0}^{m} W_i^2 I_{xi} I_{yi} \\ \sum_{i=0}^{m} W_i^2 I_{xi} I_{yi} & \sum_{i=0}^{m} W_i^2 I_{yi}^2 \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} + \begin{pmatrix} \sum_{i=0}^{m} W_i^2 I_{xi} I_{ti} \\ \sum_{i=0}^{m} W_i^2 I_{yi} I_{ti} \end{pmatrix} = 0 \tag{3.12}$$

let $\begin{pmatrix} \sum_{i=0}^{m} W_i^2 I_{xi}^2 & \sum_{i=0}^{m} W_i^2 I_{xi} I_{yi} \\ \sum_{i=0}^{m} W_i^2 I_{xi} I_{yi} & \sum_{i=0}^{m} W_i^2 I_{yi}^2 \end{pmatrix} = M$, $\begin{pmatrix} u_x \\ u_y \end{pmatrix} = \vec{u}$, $\begin{pmatrix} \sum_{i=0}^{m} W_i^2 I_{xi} I_{ti} \\ \sum_{i=0}^{m} W_i^2 I_{yi} I_{ti} \end{pmatrix} = b$.
We can get the following equation.

$$\vec{u} = M^{-1} \vec{b} \tag{3.13}$$

This equation is solvable under the below mentioned conditions:

1. The matrix M is nonsingular[3].
2. The matrix is not too small due to noise.
3. The matrix should be well-conditioned[2].

If the matrix M is singular, we can only get the norm velocity instead of the full velocity. To find out if the matrix is solvable, the eigenvalue and eigenvectors are required. Given the matrix M is in the following form:

$$\begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix} \tag{3.14}$$

The determinant is:

$$determinant = m_{00} \times m_{11} - m_{10} \times m_{01} \tag{3.15}$$

The eigenvalues are :

$$\begin{aligned} eigenvalue0 &= \frac{m_{00} + m_{11} + \sqrt{(m_{00} + m_{11})^2 - 4 \times determinant}}{2} \\ eigenvalue1 &= \frac{m_{00} + m_{11} - \sqrt{(m_{00} + m_{11})^2 - 4 \times determinant}}{2} \end{aligned} \tag{3.16}$$

The eigenvectors can be calculated in the following way:

$$\begin{aligned} eigenvector0 &= \frac{\begin{pmatrix} m_{01} \\ eigenvalue0 - m_{00} \end{pmatrix}}{\sqrt{(m_{01})^2 - (eigenvalue0 - m_{00})^2}} \\ eigenvector1 &= \frac{\begin{pmatrix} m_{01} \\ eigenvalue1 - m_{00} \end{pmatrix}}{\sqrt{(m_{01})^2 - (eigenvalue1 - m_{00})^2}} \end{aligned} \tag{3.17}$$

Till here, the principle of "Lucas" algorithm has been presented.

3.3 Summary

Optical flow method gives us an opportunity to learn the visual world from images. It is a very useful method and still under research. In combination with other techniques, optical flow field has a very wide usage in the field of computer vision. In next chapter we will go the design part of this thesis project.

4 System architecture design

The task of this thesis project is to implement "Lucas" algorithm in hardware with C/C++ based high-level sysnthesis methodology developed by Mentor Graphics. In this chapter, the original "Lucas" C code will be analyzed and cleaned up in section 4.1, the hardware system architecture will be presented in section 4.2.

4.1 Code clean up

The original C code gives many options of subfunctions to be used in each step. The purpose of code cleaning is to get rid of the unused parts including the code for processing function arguements and the unused subfunctions to get a clean code without any instructional inputs. In the following sections, the chosen subfunctions of the design will be presented one by one.

4.1.1 System flow chart

The system flow chart of "Lucas" code is presented in Figure 4.1 to illustrate how the system works in a straightforward way. As we see in Figure 4.1, the images have to go

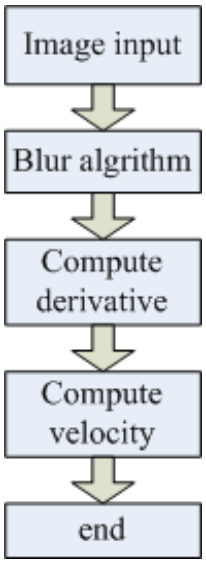

Figure 4.1: System flow chart

through three main steps to get the optical flow field calculated. Here we see a blur algorithm being used to smooth the images in order to have more continuous derivative.

4.1.2 Stackblur algorithm

The choosen smoothing algorithm is radius=3 stackblur algorithm. The principle of stackblur algorithm is shown below in Figure 4.2. Figure 4.2 shows the principle of

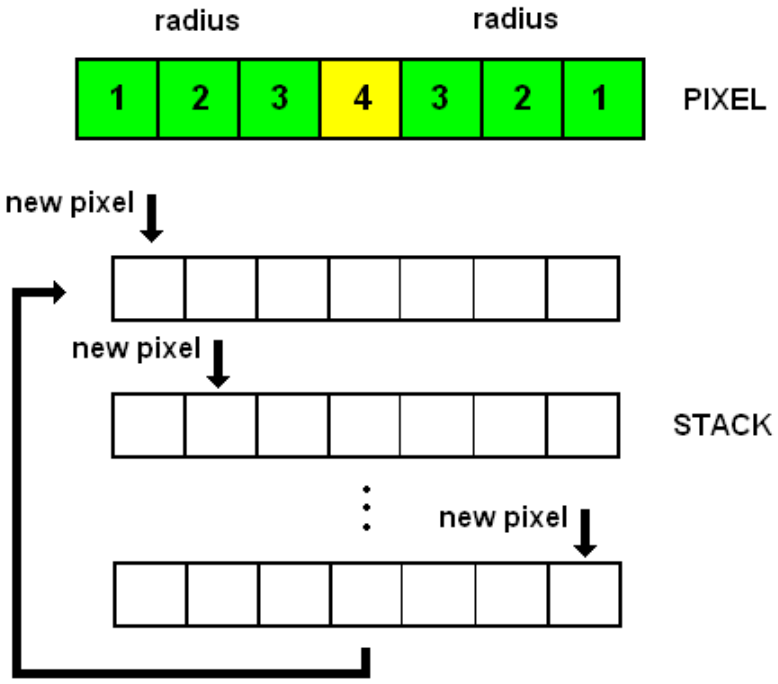

Figure 4.2: Principle of stackblur algorithm

stackblur algorithm, first we look at the above colored block array, yellow block represents the pixel being smoothed, green ones are the pixels used to smooth the yellow one, the numbers are the weights of the pixels in computation, there is also a smoothing step in the column which hasn't be shown here. The blank block array under the colored one represents the stack being used to store the pixel values for calculation. Here we can see, the new pixel value is stored into the stack one after another in a circular way.

4.1.3 Compute derivative

For this part, a 3×3 derivative computation algorithm is chosen, which means three frames of consecutive images are needed. The instensity value derivative with x, y and time are calculated respectively by simple subjections. Figure 4.3 presents the simple way of calculating the intensity value derivatives, under the condition that the three frames of images have to be taken with a small time interval.

4.1.4 Compute velocity

The velocity computation is a very essential part of the whole "Lucas" source code. It comprises of three sub-parts, the first part is used to calculate the matrix

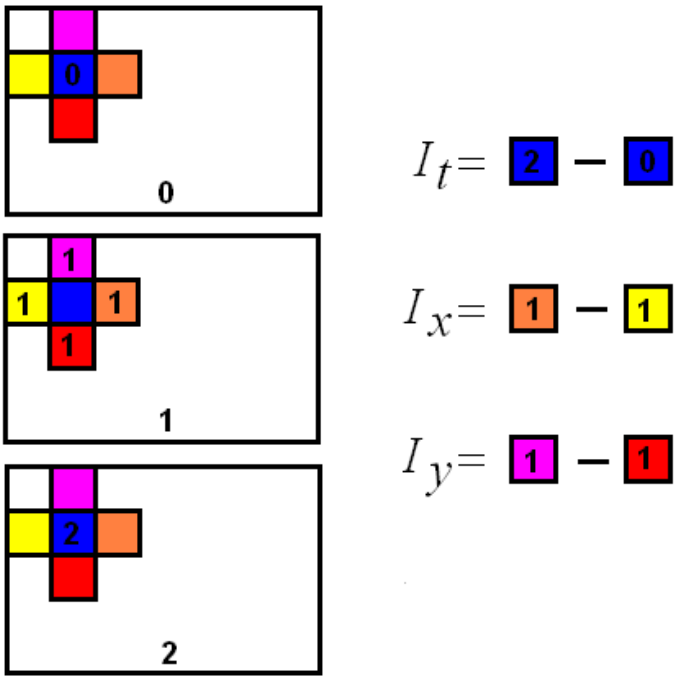

Figure 4.3: Compute derivatives

being used for the least square solution, the second part is for solving the matrix and the last part is for identifying the velocity types and computing the velocity values.

The oringinal code gives options in the matrix calculation part, there are choices for the 5×5 weighted window and 3×3 weighted window. Here the weighted 3×3 window size is choosen. The principle of which has been shown in Figure 3.5 and equation 3.12.

Hereby the code clean up part is finished, what we have choosen is radius=3 stackblur, 3×3 derivative computation and weight 3×3 window. The next section will present the hardware system architecture.

4.2 Hardware system architecture

The difference between high level C/C++ code and HDL code as well as the principle of converting C/C++ to HDLs have been discussed in detail in chapter 2. This section is going to show us how those theories work on converting "Lucas" C code to RTL code with the help of Catapult C.

4.2.1 System architecture

The hardware system architecture is shown in Figure 4.4. This system architecture is designed based on the software system flow chart in Figure 4.1. As we see clearly, the

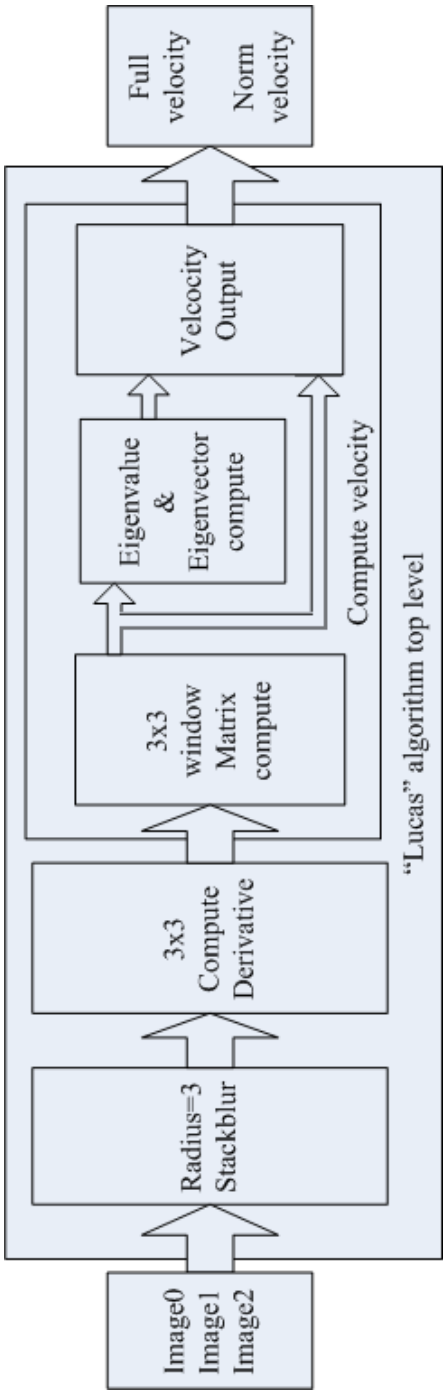

Figure 4.4: Hardware system architecture of "Lucas" algorithm

top level has the image datas as input and velocity datas as output. Inside top level there are three big blocks, radius 3 stackblur block, 3×3 compute derivative block and velocity compute block, the last one also consists of three sub blocks, weighted 3×3 window matrix calculation block, eigenvalue block and velocity output block.

What we then need to do is to redesign the blocks based on the original C code to make them useable and synthesizable, also the correct communications among the blocks have to be assured.

4.2.2 Datatype redefine

In Catapult C every datatype has to be fixed-point in order to be synthesizable, that's the prerequisite for hardware design

First let us look at the datatypes being used in the design.

1. Unsigned char datatype is used for the image data, its representation in hardware is a 8-bit binary number.

2. Int datatype has the representation of 32-bit binary numbers in hardware, it is used for the smoothed image data, derivative data and some of the intermediate signals and variables in the velocity compute block.

3. The algorithm C fixed-point datatypes $ac\_fixed < 32, 16, true >$ and $ac\_fixed < 32, 16, false >$ are used to replace any of the float datatypes originally being used for such as, the matrix data, eigenvalues, eigenvector elements and velocity data.

Then we can move on to take care of the ports and interconnections among the inner blocks.

4.2.3 Block interfaces

Interface define is one key element of a synthesizable C/C++ design, it has to be defined explicitly like the way HDLs do, the type and size of the interface ports have to be static. The top level of this design has three input ports and four output ports, it is shown below.

```
#pragma design top                    //indicatoin of top level function

void top(
unsigned char f1i[316*252],              //input image1
unsigned char f2i[316*252],              //input image2
unsigned char f3i[316*252],              //input image3
m_type full_vels_x[316*252],            //output full velocity x
m_type full_vels_y[316*252],            //output full velocity y
m_type norm_vels1_x[316*252],          //output norm velocity x
m_type norm_vels1_y[316*252]           //output norm velocity y
)
```

As we can see each port will be assigned to an exterior memory block after synthesis. The intermediate interconnections are shown below.

```
{
/*-----------data ports for blocks----------*/
    int f1o[316*252],f2o[316*252],f3o[316*252];   //output of stackblur
    int Ix[316 * 252],Iy[316 * 252],It[316 * 252];   //der_3x3 output
    /*--------------stack blur-----------------*/
    stackblur(f1i,f2i,f3i,f1o,f2o,f3o);
    /*--------------compute derivatives-------------------*/
    compute_ders_3x3(Ix,Iy,It,f1o,f2o,f3o);
    /*--------------compute velocities--------------------*/
    compute_vels(Ix,Iy,It,full_vels_x,full_vels_y,norm_vels1_x,
        norm_vels1_y);
}
```

As we can see six memory blocks of int type are defined inside the top level to be used as interconnections among the three sub-blocks, array f1o, f2o, f3o are used as the ouputs and inputs of stackblur block and compute_ders_3x3 block respectively; array Ix, Iy, It are used as the outputs and inputs of compute_derivative block and compute velocities block respectively.

4.2.4 Memory access

Since the the interfaces have been defined differently from the way it was in the original code, the data access inside the blocks also need to be reformed. As we can see in section 4.2.3, memories are used as interconnections between the blocks, which means each of the data being processed has to be accessed from a memory. The example shown below is part of the compute derivative block, from where we can clearly find out how memory access being established.

```
/* 1D kernel in t direction*/
It[pic_y * x + y] = (pic2[pic_y * x + y] - pic0[pic_y * x + y]  ); //
    pic_y=imgage width

//It[x][y] = (int)(pic[2][x][y] - pic[0][x][y]  ); original code

 /* 1D kernel in x direction*/
for (p=0;p<=1;p++) {                    //To avoid two read action on same
     RAM
    if(p==0)
        tmp=pic1[pic_y * (x-1) + y];
    else
       Ix[pic_y * x + y] = (pic1[pic_y * (x+1) + y] - tmp);
}

//Ix[x][y] = (int)(pic[1][x+1][y] - pic[1][x-1][y]); original code
```

We can see the original used code has been commented out. In the rewritten code, no format convert funtion such as (int) can be used because it can not be compiled by the synthesizer, also, memory address need to be recalculated. Apart from that,

one extra "for" loop is used to prevent the read action on the same Ram at the same time, because as we see the calculation of I_x involves two read actions from the same memory.

4.2.5 Use of Catapult C build in library

The C/C++ has its own mathematics library, including functions such as sine, cosine, square root etc.

However, they are not yet compilable by the Catapult C synthesizer. Because such kind of heavy computing complexity operations should be well designed and optimized for different data types and have the ability of pipelining to meet the requirements of the growing clock frequency.

Therefore Mentor Graphics has developed its own C++ synthesizable mathematics library for Catapult C, operational modules such as division, square root, sine, cosine are included, some of them have been further classified into fixed-point and integer versions.

In this project, the fixed-point square root function in Catapult C Mathematics library has been used to replace the sqrt() function which is originally used. Also the fixed-point division module has been used. Note that designers have to include math/mgc_ac_math.h .

4.2.6 Parallel design

In order to acclerate the system operation speed, parallel designs are recommended.

The code below roughly shows the difference between the original and redesigned code of stackblur algorithm.

```
/*  original "Lucas" code  */

for(k=0;k<FRAMES;k++)  // FRAMES is the number of images being used
{
      stackblur((*inpic)[k], (*pic_x), (*pic_y), radius,  radius);
              //pixel value,   width ,  height, x radius, y radius
}

/*  redesigned code  */

      stackblur(f1i,f2i,f3i,f1o,f2o,f3o);)
     // f1i,f2i,f3i are the three input images; f1o,f2o,f3o are the three
         output images
```

Regardless of the difference between the two functions' variables, in the original code the function is recalled every time as an image comes in, while in the redesigned version all three images comes in together and be processed in parallel, which saves a lot of

computing time. Of course the internal code of stackblur have to be redesigned to parallelize the process.

4.2.7 Testbench design

Testbench is an important part of digital design. For this C/C++ based high-level digital design method, testbench in C/C++ is supported and required for the gcc based simulation. Another amazing feature about Catapult C is it can transform the C/C++ testbench into system C automatically, thereby we can run the modelsim simulation right after the generation of the HDL code needless of writing a system C or VHDL or verilog testbench.

The C testbench fo this project consists of the following parts:

1. Define interfaces
2. Read input data
3. Run with input data
4. Display output data

In below, some parts of the testbench are shown.

```
/*-----------data ports for blocks----------*/
    unsigned char f1i[316*252],f2i[316*252],f3i[316*252];
    // inputs of stackblur

    m_type full_vels_x[316*252],full_vels_y[316*252];
    // output full velocity

    m_type norm_vels1_x[316*252],norm_vels1_y[316*252];
    // output norm velocity
/*------------read input images-----------*/
    fp1=fopen("yos.8","rb");
    fread(&fhead[0],1,32,fp1);       //the first 32 values are the head
       code of the ras image format
    fread(&f1i[0],1,316*252,fp1);
    fclose(fp1);

    fp2=fopen("yos.9","rb");
    fread(&fhead[0],1,32,fp2);
    fread(&f2i[0],1,316*252,fp2);
    fclose(fp2);

    fp3=fopen("yos.10","rb");
    fread(&fhead[0],1,32,fp3);
    fread(&f3i[0],1,316*252,fp3);
    fclose(fp3);
/*----------run inputs in top level------------*/
```

```
    CCS_DESIGN(top)(f1i,f2i,f3i,full_vels_x,full_vels_y,norm_vels1_x,
        norm_vels1_y);
```

As we see there is no synthesizability requirement for the testbench, we can use normal C/C++.

4.3 Summary

Writing synthesizable C/C++ code requires a deep understanding of the algorithm as well as a concept of hardware architecture.The C/C++ to HDLs converting process and simulation results will be shown and discussed in the next chapter.

Code conversion and simlulation results

5

Now that we have prepared the written C code and testbench, the next is to convert the C code to RTL language as well as verify its functionality and efficiency by Catapult C. Apart from that, Catapult C can apply some optimizations automatically as well as optionally, in that way we can refine the design to meet different hardware requirements. In the following sections, the code conversion will be presented in section 5.1, the simulation results will be presented in section 5.2.

5.1 Code conversion

In this section, the Catapult C operation flow will be presented step by step.

5.1.1 Import design

The first step is to import the source code and testbench into a Catapult C project, note that the test bench has to be excluded from the compilation. The process is shown in Figure 5.1. We can see the Synthesis Tasks window on the left, in which the whole design process is listed step by step, also we see there is a solution window on the upper left, which shows different solution for one design, any change of the code or constraints will generate a new solution.

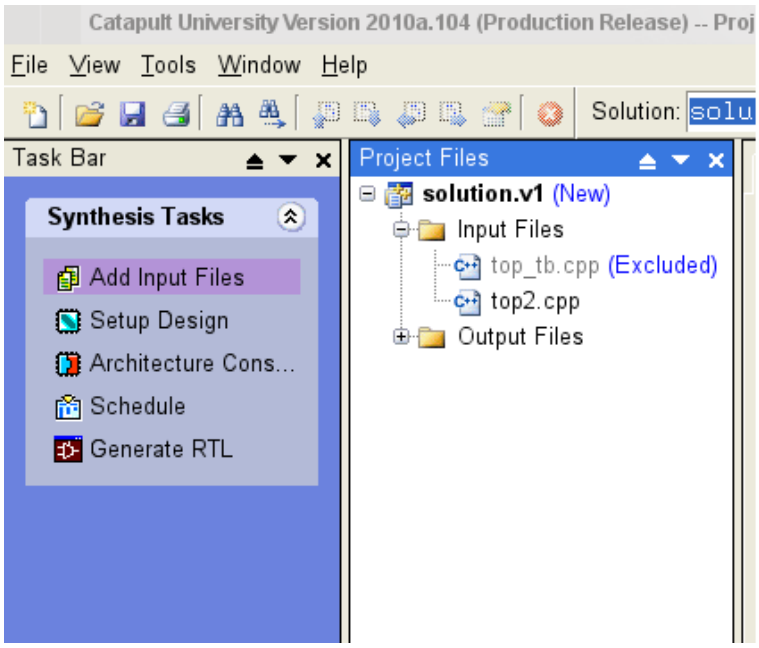

Figure 5.1: Import design

Next, we need to setup the design, in which the technology, design library, design frequency, handshake mode and top level function can be set. Figure 5.2 shows the graphic user interface of the setup design process, we can see from Figure 5.2 the Xinlinx Virtex 5 VLX220TFF1738 FPGA is chosen as the target device. The accerlerated library, base FPGA library, single port RAM is chosen as the compatible libraries. The clock frequency is set to be 50MHz.

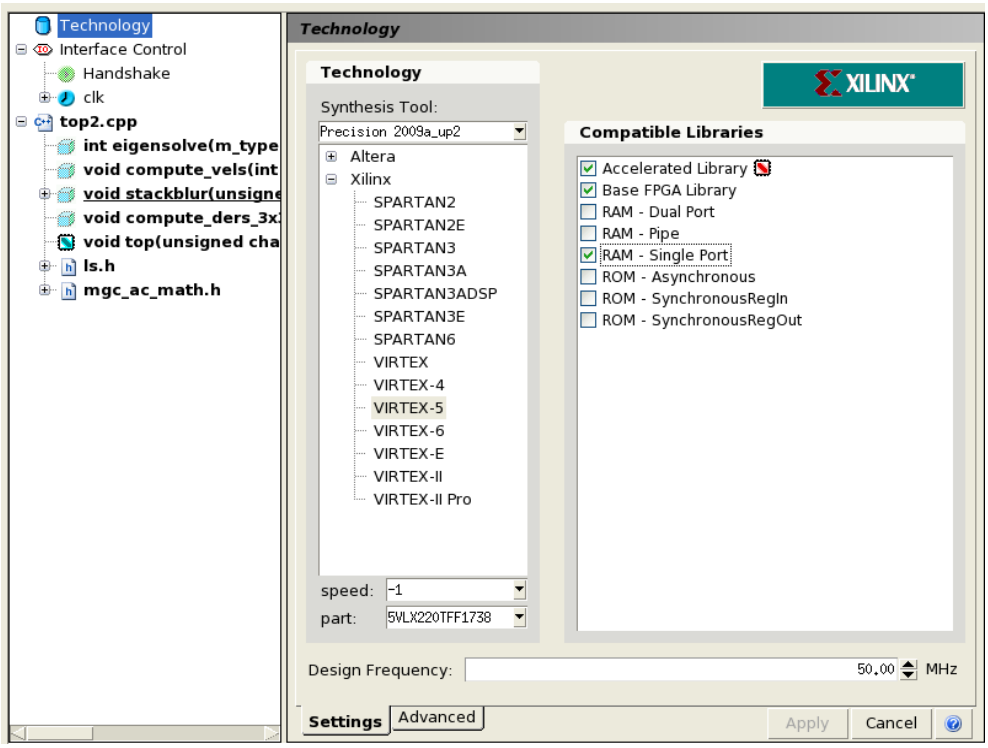

Figure 5.2: Setup design

5.1.2 Apply constraints

After the design has been setup, we need to apply some design constraints in order to meet the design requirements. In this design we want to have a fast processing speed regardless of the resource usage, thereby we will try to parallelize or pipeline the design as much as we can. Figure 5.3 shows the design constraints settings in this project, as we see the software offers options like unroll or pipeline the loop which can be found in Figure 5.3.

As we see in Figure 5.3, the blue circles represent the loops left rolled in the design flow, the green and yellow little blocks mean that the loop has been fully unrolled, the blue circle with red arrow means the loop has been pipelined, the initial interval time is highlighted in blue, the blue square around the loop means the loop merging has been deactivated.

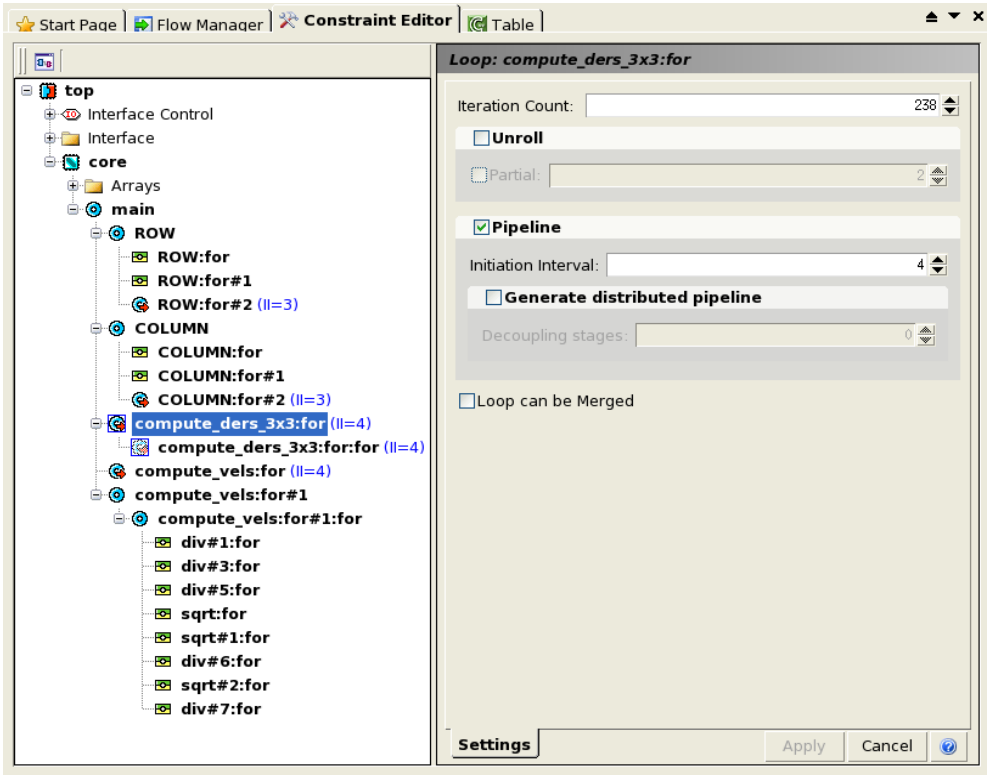

Figure 5.3: Design constraints

5.1.3 Scheduling

Now we can click on schedule task in synthesis tasks to do the automatic scheduling. As one of the main tools of algorithm analyzing in Catapult C, the scheduling window graphically shows the hierarchical nesting of loops, the relative runtime of each loop, the types of operatons, the operations scheduling sequences and the dependencies between operations. Textually, the loop names operations and components, the area, delay are shown. Also, all items can be cross-probed to the source code.

The Gantt chart can be shown in compacted or expended mode. In compacted mode, all information are tightly arranged to save space on the screen. In expanded mode, we can see every detail of the operation scheduling. Scheduling details of radius=3 stackblur block are partly shown in Figure 5.4, in which we can clearly see the operation timings as well as their dependencies.
When we move our mouse indicator onto the operation elements, we can see the detailed information of this operation element. An example is shown in Figure 5.5.

5.1.4 Generating HDL code

If the scheduling has been smoothly done, we can click on the Generate RTL task to generate all the other output files, including HDL files, report files, schematics.
Figure 5.6 shows the project files after the complete HDL generation process.

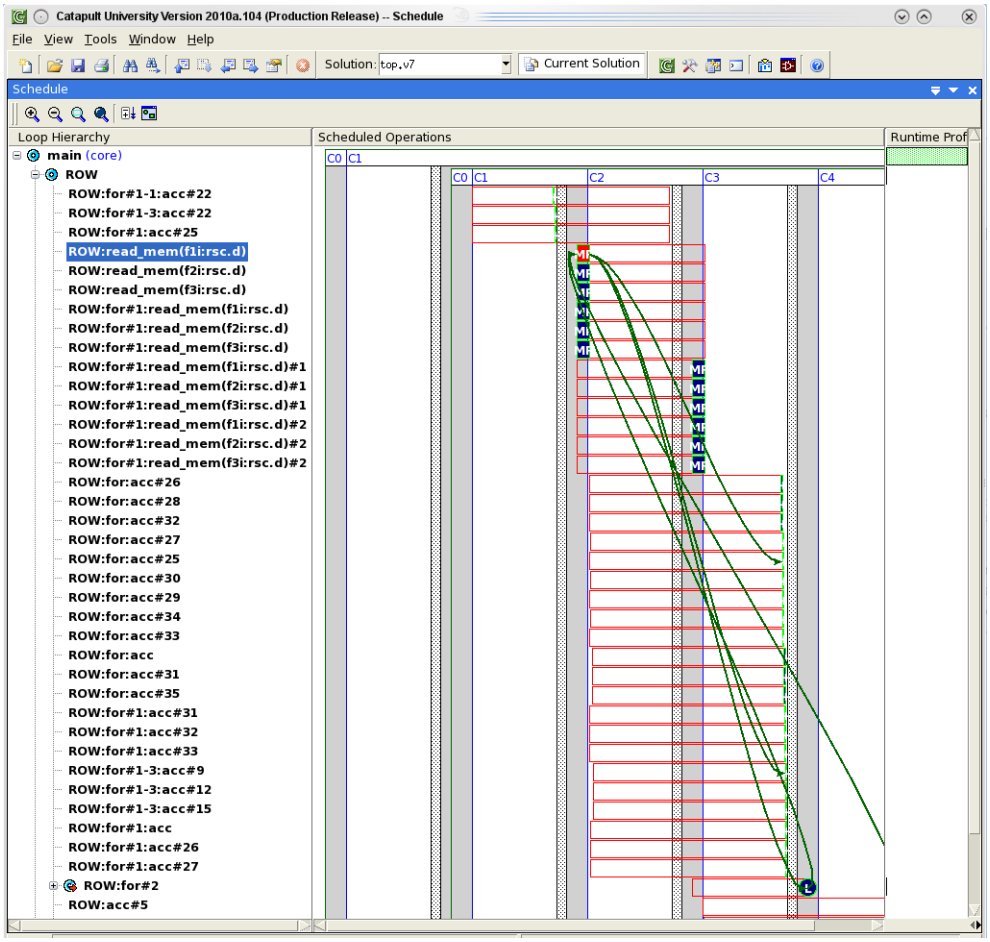

Figure 5.4: Gantt chart of stackblur block(partly)

Catapult C generates two types of HDL codes, the *cycle* and *rtl*.

- The cycle output is for fast simulation because the I/O of the design is cycle accurate. In this project it's not used.
- The RTL output is for RTL level simulation.

The generated VHDL code *rtl.vhdl* has 17155 lines.

5.1.5 Schematics

Catapult C outputs four kinds of schematic files, RTL, DataPath, CriticalPath and CriticalMap.The DataPath and CriticalPath are filtered from the RTL schematic.
The schematics are shown in a hierarchical way. We can go into lower-level blocks by double click in the upper-level blocks.

Figure 5.7 shows the top-level schematic, Figure 5.8 shows the first page of the top core inside RTL schematic, there are 202 pages in all.

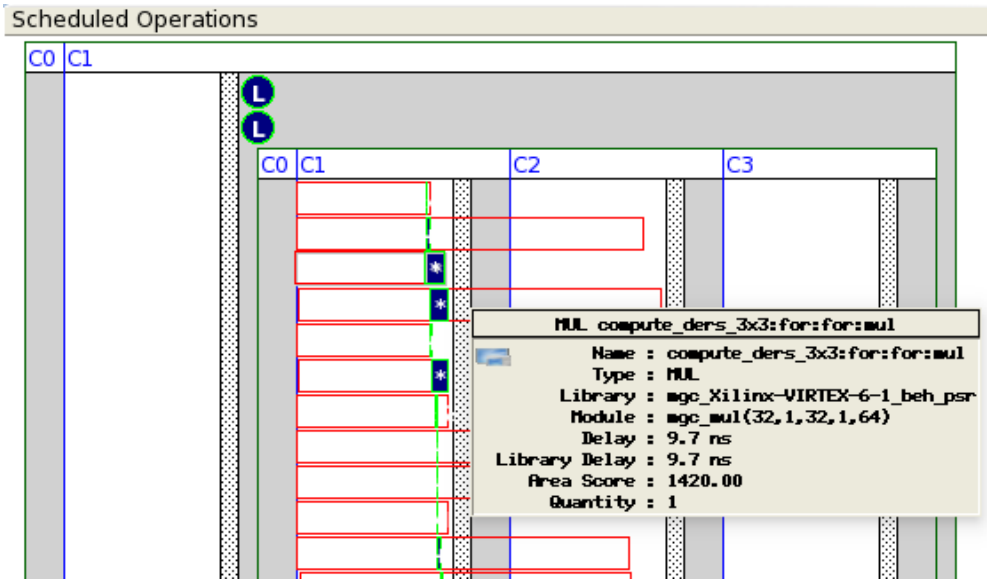

Figure 5.5: Operation element details

As we see in Figure 5.7 the big block on the left side is the top core, the six small blocks on theside are the six intermediate ports.

The Critical Path schematic is shown in Figure 5.9, we can see the window is divided into three parts, the schematic in the top, the path table in lower left side and the path quick view in the lower right side. There are ten most critical paths listed in the path table and the selected one will be highlighted in the schematic, the detailed timing data about each instance in the selected path is shown in the path quick view. Also It is also possible to find out the point to point path delay.

5.1.6 reports

The Catapult C will automatically generate two reports for each solution.

- The *cycle.rpt* is a high-level architecture/algorithm report which includes informations regarding I/O ports, memory resources, loops and latency.
- The *rtl.rpt* includes informations about materials usage, area, register-to-variable mappings and timing.

Figure 5.10 shows the area score in *rtl.rpt*. Figure 5.11 shows part of the *cycle.rpt*, from where we can find the process general information, clock information and loop information.

From Figure 5.11, we can find that with the clock frequency set to be 50MHz the generated design has a latency of 136.58 ms for processing three frame of images.

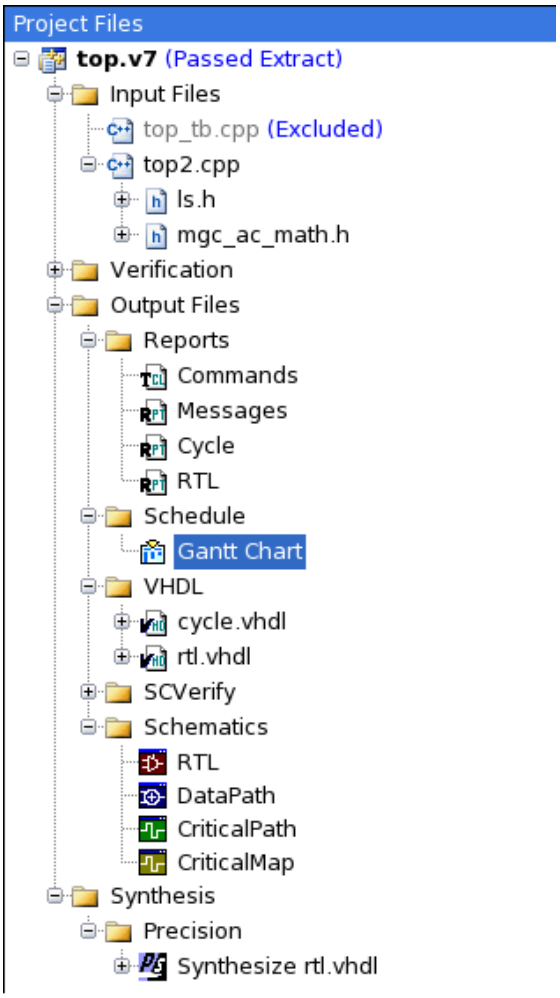

Figure 5.6: Output file after HDL generation

5.2 Simulation results

Catapult C provides a gcc based simulation on user provided C/C++ source code and testbench. Once the RTL is generated, Catapult C provides a push-button verification flow that allows simulation of the generated code against the source code and testbench. However, in this project, the output velocity of the RTL simulation is slightly different from that of the souce code simulation, the reason lies in the difference between the C++ datatype $float$ and algorithm C datatype $ac\_fixed < 32, 16, ture >$. Thereby for this design, the souce code and generated HDL code are simulated separately and compared manually.

The source code simulation can be done by double clicking on the Original Design + Testbench in the verification folder and the results are shown in Figure 5.12. Three sets of the velocity datas are choosen to be printed out.

For RTL simulation, Catapult C will automatically generate a makefile to take

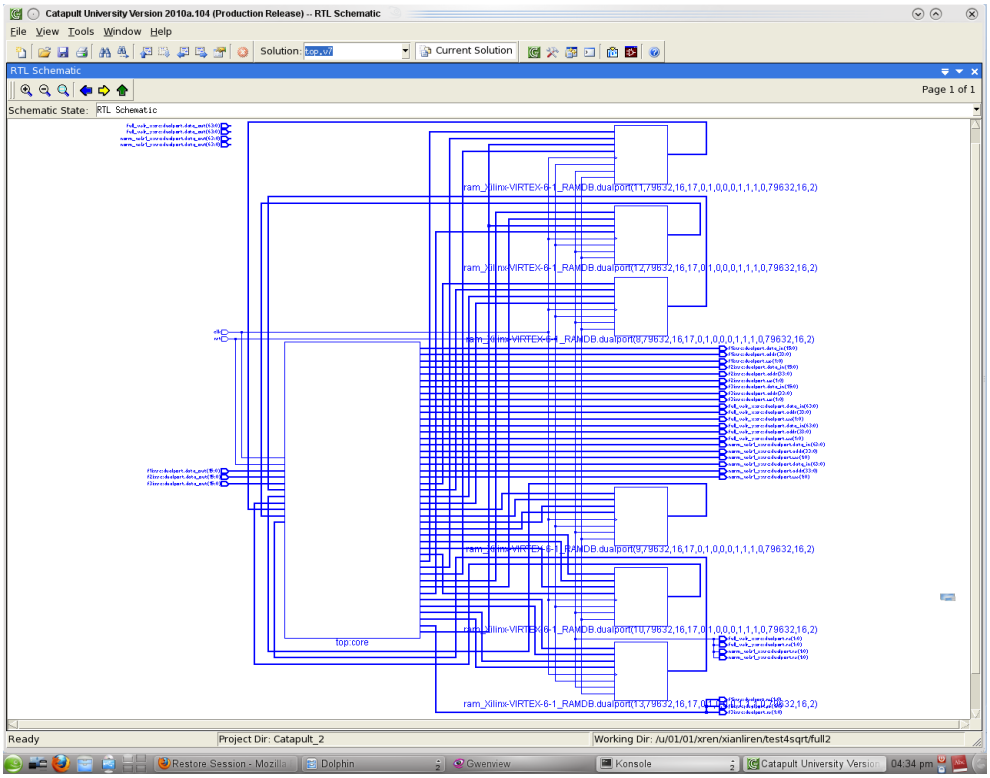

Figure 5.7: Top-level RTL schematic

care of compiling and linking with Modelsim, no extra testbench needed because Catapult C will convert the user supplied C/C++ testbench into SystemC.

Figure 5.13 shows part of the Modelsim simulation wave form, we can see all the interface signals are automatically included into the wave form.

Table 5.1 shows the comparison between the results of source code and RTL simulation. We can see in Table 5.1, the modelsim simulation results are almost

	C simulation result			
	full velocity x	full velocity y	norm velocity x	norm velocity y
pixel address	6787	6787	7963	7963
pixel data	2.77983	-2.28939	0.56255	-0.19875
	modelsim simulation result			
	full velocity x	full velocity y	norm velocity x	norm velocity y
pixel address	6787	6787	7963	7963
pixel data(Hex)	0002C7A3	FFFDB5EA	00009003	FFFFCD1F
pixel data(Dec)	2.7798	-2.2879	0.5625	-0.1987

Table 5.1: Simulation result comparison

the same with the gcc simulation results, which means this C to RTL conversion is

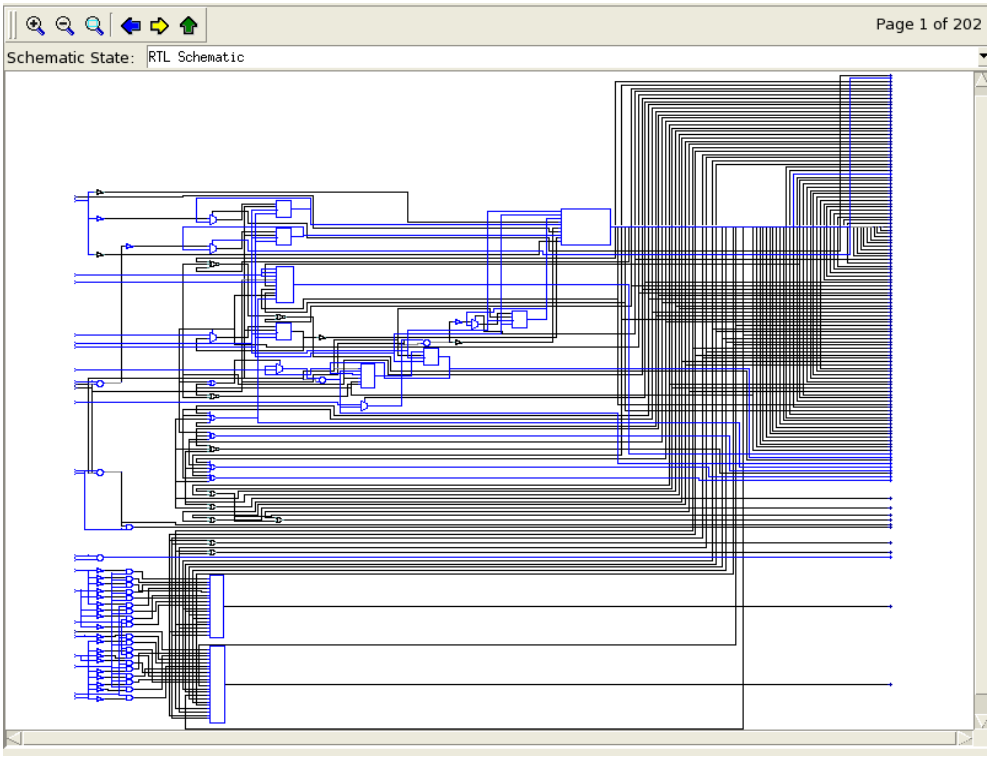

Figure 5.8: First page of Top core inside RTL schematic

successful in terms of functionality.

5.3 Different design constraints

After verifying the design functionality, we can move on to optimize the design to get a faster solution. Different synthesis clock frequencies and design constraints have been applied to generate several solutions.

The following four solutions have been implemented and verified.

1. Clock frequency 50MHz, no design constraints added, loop merging disabled for the *compute_ders_3* $\times$ 3 block. In the comparison table and charts (Figure 5.14, Figure 5.15, Figure 5.16) this solution is called top.v3.

2. Clock frequency 50MHz, design constraints setting as shown in Figure 5.3, loop merging disabled for the *compute_ders_3* $\times$ 3 block. This is the version previously implemented in the before sections of chapter 5. In the comparison table and charts (Figure 5.14, Figure 5.15, Figure 5.16) this solution is called top.v2.

3. Clock frequency 100MHz, design constraints setting as shown in Figure 5.3, loop merging disabled for the *compute_ders_3* $\times$ 3 block. Higher clock frequency has been tried but errors happened because the feedback path is too long to schedule design with current pipeline, thus 100MHz is the maxim frequency used in this

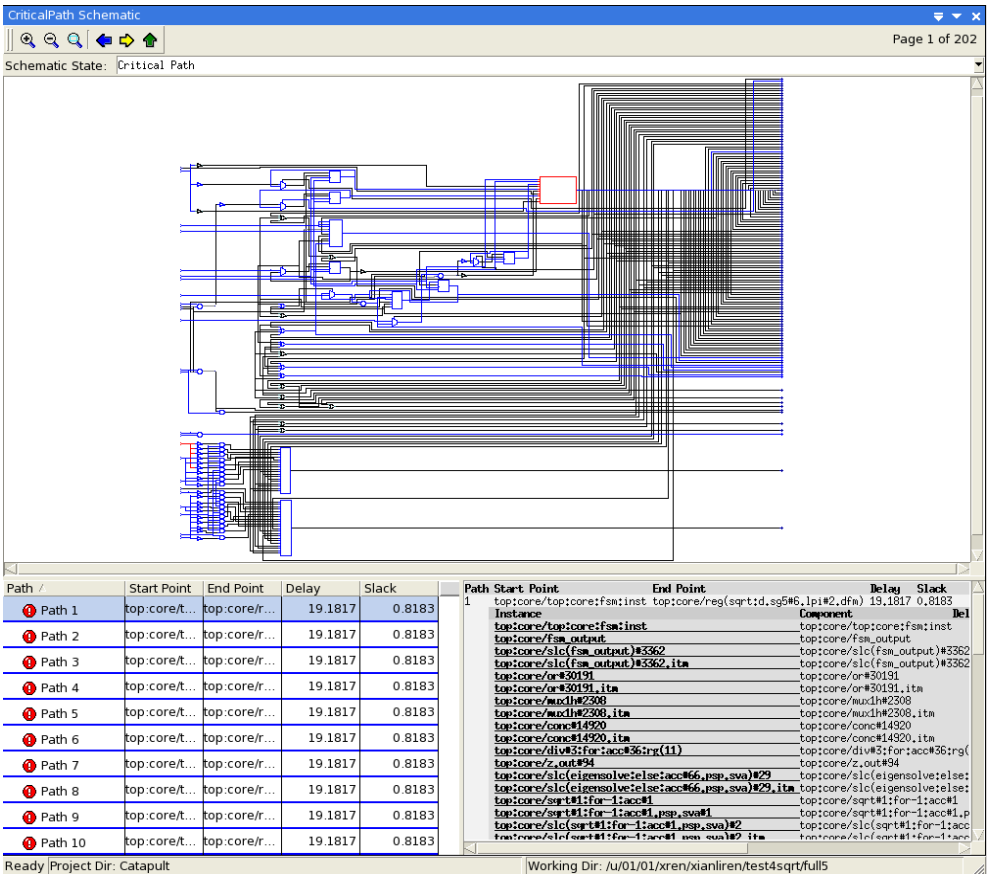

Figure 5.9: Critical path schematic

Area Scores

	Post-Scheduling	Post-DP & FSM	Post-Assignment
Total Area Score:	36064.2	96811.9	43759.9
Total Reg:	0.0	0.0	0.0
DataPath:	**36064.2 (100%)**	**96810.9 (100%)**	**43758.9 (100%)**
MUX:	6392.1 (18%)	24795.6 (26%)	16908.9 (39%)
FUNC:	29545.4 (82%)	70664.6 (73%)	22481.4 (51%)
LOGIC:	126.7 (0%)	1350.7 (1%)	4368.6 (10%)
BUFFER:	0.0	0.0	0.0
MEM:	0.0	0.0	0.0
ROM:	0.0	0.0	0.0
REG:	0.0	0.0	0.0

Figure 5.10: Area score

project. In the comparison table and charts (Figure 5.14, Figure 5.15, Figure 5.16) this solution is called top.v1.

Figure 5.14 shows the table which lists the area and latency of the four different solutions of design.

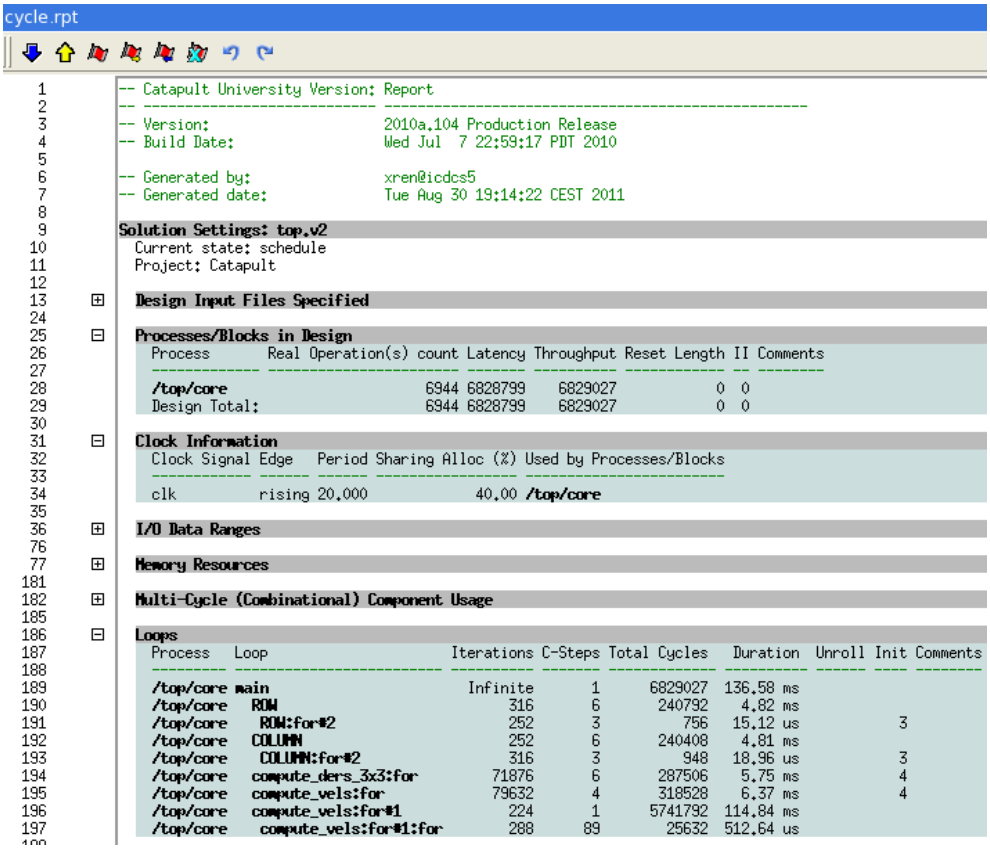

Figure 5.11: Latency, clock and loop report

Figure 5.15 shows the bar chart of the area scores of different solutions, we can see that in each solution memory and mux takes up most of the area, the top.v3 solution is the raw solution without any extra architecture contraints added (except that loop merging is disabled on the compute derivative block), it costs the least area but it's also the slowest one. Figure 5.16 shows the XY plot chart, X and Y axises represent the total area score and latency time respectively. We can see clearly that top.v1 is the fastest solution and top.v3 is the slowest solution.

5.4 Compare with hand write VHDL code

There is a previous work on the same "Lucas" algorithm done by a previous MSc student T.M.F.Hurkmans[23], which is about implementing the algorithm in hardware with hand-write VHDL. Thus it is meaningful to compare the two solutions to see if C/C++ based high-level synthesis flow can really help in digital IC design.

Table 5.2 shows the features of the two solutions. Please note that this design is not fully pipelined, it has a latency for processing three images, thereby the FPS(frames per second) of this design is calculated by how many pictures it can process in one second.

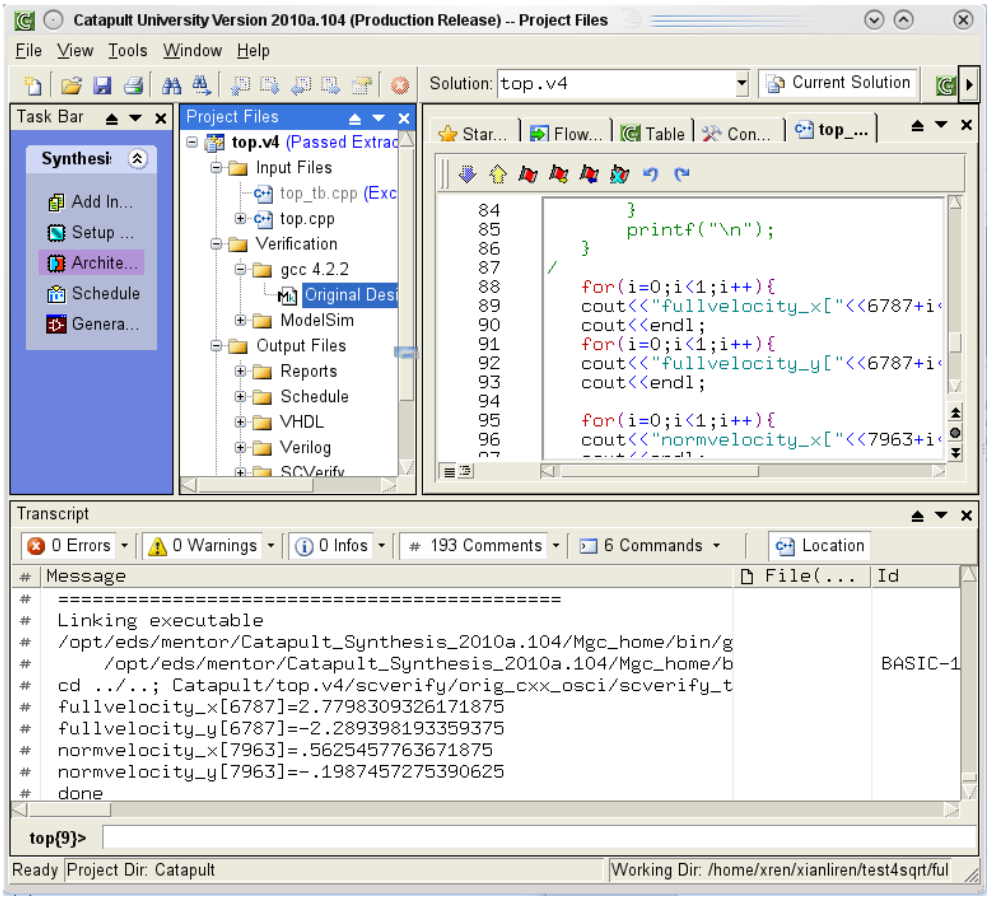

Figure 5.12: C testbench result

Solution	Hand written VHDL	Catapult C generated VHDL
Clock frequency (MHz)	30.884	50.00
Image size	316*252	316*252
FPS	15.99	22
LUTs	7006 (Virtex 5)	14251 (Virtex 5)
I/O pins	447 (Virtex 5)	438 (Virtex 5)
FF	2802 (Virtex 5)	6586 (Virtex 5)
DSP48s	106 (Virtex 5)	47 (Virtex 5)

Table 5.2: Comparison of previous hand wirte deisgn and this design

As we can see in table 5.2, compared with the formal design, this design can be run at a higher clock frequency than that of the hand written one and it's processing speed is also faster.

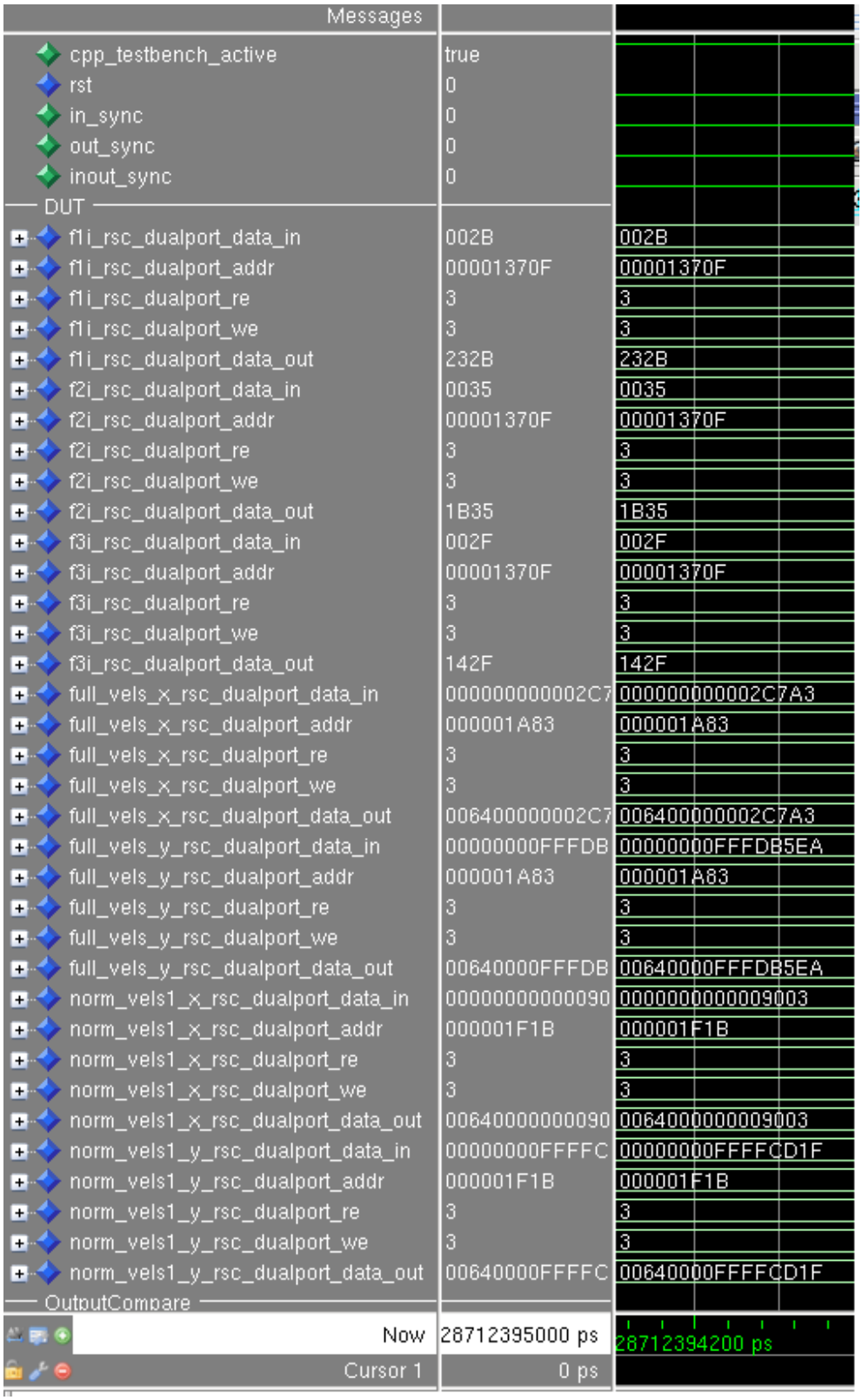

Figure 5.13: Modelsim simulation waveform

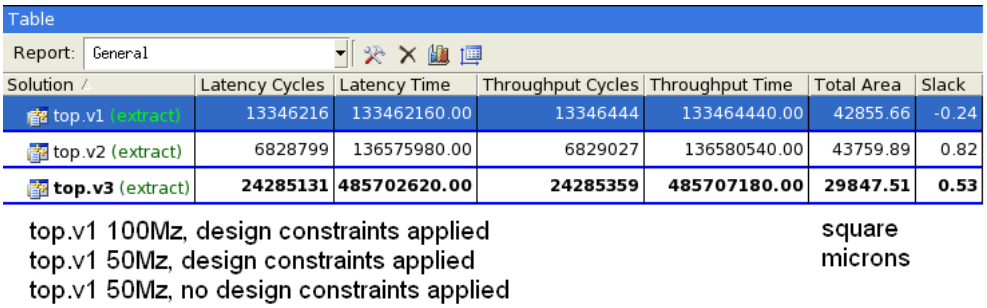

Figure 5.14: Solution table

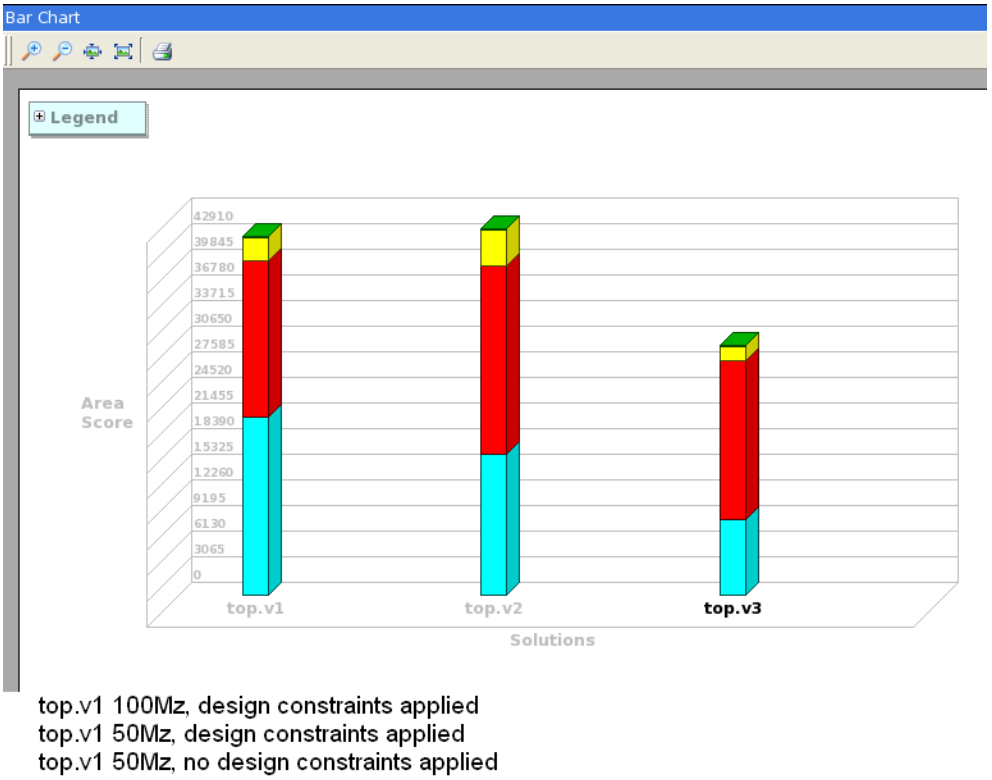

Figure 5.15: Bar chart

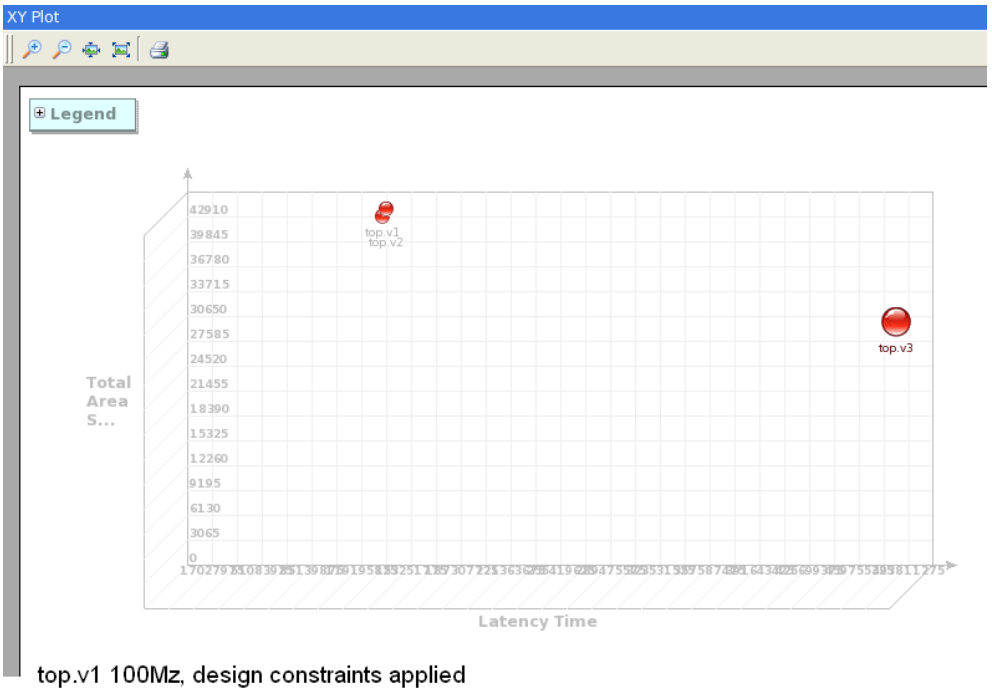

Figure 5.16: XY plot chart

6 Conclusion

Now that this design has been finished, it is time to draw the conclusion of this work. This chapter is divided into three parts, the summary of writing synthesizable C/C++ will be shown in section 6.1, the summary of the Catapult C benefits will be shown in section 6.2, section 6.3 will present the future work.

6.1 Summary of writing synthesizable C/C++ code

Here I want to give a summary on the steps of wirte synthesizable C/C++ code in this design.

- Analyze the source code, discard any unused or instructional part, make sure the rest of the code is all indispensable.
- Draw the system architecture graph. Better to map each block to a destination subfunction in the source code.
- Redefine the data types, change any unsynthesizble data types into fixed-point.
- Redefine the function variables, as they will be converted into block ports, the correct data transfer as well as the explicit size of ports should be assured.
- Now take care about the inside of the blocks, make sure all the code is synthesizable. Rewrite any unsupported write styles, replace any unsupported functions like $fabs()$, $sin()$ by a hand written one or the one in the Catapult C library.
- Try some special writing styles to meet the goals, please refer to chapter 2.
- Write the C++ testbench and move on to the verify and converting stage with Catapult C

6.1.1 Hardware descriptive ability of C/C++

6.2 Summary of Catapult C

With Catapult C, the design period can be dramatically reduced, for example, designers can change the synthesis clock frequency in the Catapult C GUI and then the synthesizer will automatically analyse the datapath and do the scheduling; while in the traditional design flow, changing the clock frequency may lead to a lot of manual work, for example lots of registers may need to be displaced.

Catapult C is not only a tool for C/C++ to RTL conversion, it is also a very

good algorithm/hardware architecture analysis tool. It shows graphically the system structure, data path, scheduling gantt chart, schematics as well as charts for area, latency comparision.

6.3 Summary of results

The design can be run at a frequency of 50MHz, the process speed is 22 frames of images per second. The image size is 316×252. The design is compiled with the Xilinx Virtex 5 FPGA library, which means it is possible to be synthesized and implemented onto an FPGA.

The synthesis tool is chosen to be Mentor Graphics Precision RTL Synthesis, the synthesis results shows that this design can be synthesised into Virtex 5VLX330TFF1738 FPGA. Figure 6.1 shows the resource usage.

```
*******************************************************************
Device Utilization for 5VLX330TFF1738
*******************************************************************
Resource                          Used    Avail   Utilization
-------------------------------------------------------------------
IOS                               438     960      45.62%
Global Buffers                    1       32        3.12%
LUTs                              14251   207360    6.87%
CLB Slices                        3563    51840     6.87%
Dffs or Latches                   6586    207360    3.18%
Block RAMs                        222     324      68.52%
   RAMB18                         0
   RAMB36                         222
DSP48Es                           47      192      24.48%
-------------------------------------------------------------------
```

Figure 6.1: Synthesis result

6.4 Future work

Some work can be done in the future to improve the design and find out more about writing better synthesizable C/C++ code.

- Run the design on FPGA.
- Pipeline the full design, let it be able to process consecutive image streams.

  Catapult C provides function level pipeline for complex system consisting of various processing stages, which is called hierarchical synthesis, as is shown in Figure 6.2.
- Improve the algorithm to shorten the dependency chains.

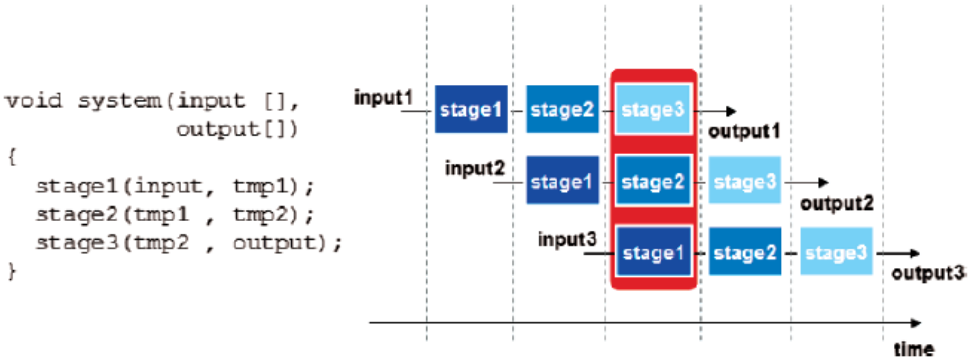

Figure 6.2: Hierarchical synthesis makes function overlap[11]

Bibliography

[1] Background Subtraction. http://www.societyofrobots.com/programming_computer_vision_tutorial_pt4.shtml#background_subtraction.

[2] Condition number. http://en.wikipedia.org/wiki/Condition_number.

[3] Invertible matrix. http://en.wikipedia.org/wiki/Invertible_matrix.

[4] Optical Flow Filter for Smooth,Keyframe-able Motion Effects. http://www.borisfx.com/image/bcc/avx/optical.jpg.

[5] T Kanade BD Lucas. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence*, pages 121–130, 1981.

[6] B.Horn and B.Schunck. Determining Optical Flow. *Artificial Intelligence*, 1981.

[7] *High-Level Synthesis: from Algorithm to Digital Circuit*, page 33. Springe, 1 edition, 2008.

[8] *High-Level Synthesis: from Algorithm to Digital Circuit*, page 41. Springe, 1 edition, 2008.

[9] *High-Level Synthesis: from Algorithm to Digital Circuit*, page 42. Springe, 1 edition, 2008.

[10] *High-Level Synthesis: from Algorithm to Digital Circuit*, page 42. Springe, 1 edition, 2008.

[11] *High-Level Synthesis: from Algorithm to Digital Circuit*, page 43. Springe, 1 edition, 2008.

[12] *Catapult C Synthesis C++ to Hardware Concepts*, page 44. Mentor Graphics, 2010.

[13] *Catapult C Synthesis C++ to Hardware Concepts*, page 45. Mentor Graphics, 2010.

[14] *Catapult C Synthesis C++ to Hardware Concepts*, page 46. Mentor Graphics, 2010.

[15] *Catapult C Synthesis C++ to Hardware Concepts*, page 48. Mentor Graphics, 2010.

[16] *Catapult C Synthesis C++ to Hardware Concepts*, page 49. Mentor Graphics, 2010.

[17] *Catapult C Synthesis C++ to Hardware Concepts*, page 50. Mentor Graphics, 2010.

[18] *Catapult C Synthesis C++ to Hardware Concepts*, page 51. Mentor Graphics, 2010.

[19] J.J Gibson. *The Perception of the Visual World*. Houghton Mifflin, 1950.

[20] D.Djebrouni D.Meriche Kahlouche Souhila, O.Djekoune. On the use of optical flow in robot navigation. In *IEEE International Conference on Signal Processing and Communications*, pages 1287–1290, 2007.

[21] *Catapult C Synthesis C++ to Hardware Concepts*, page 12. Mentor Graphics, 2010.

[22] *Catapult C Synthesis C++ to Hardware Concepts*, page 14. 2010.

[23] T.M.F.Hurkmans. System Performance Analysis and Fixed-Point Architecture of a Gradient-Based Optical Flow Algorithm. 2009.