ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

High-Level-Synthesis του αλγορίθμου Όρασης Υπολογιστών Harris σε FPGA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Π.Γαλάνης

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής καθηγητής Ε.Μ.Π.

Αθήνα,Ιούνιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

High-Level-Synthesis του αλγορίθμου Όρασης Υπολογιστών Harris σε FPGA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ιωάννης Π.Γαλάνης

Επιβλέπων: Δημήτριος Σούντρης
Αναπληρωτής καθηγητής Ε.Μ.Π.

Εγκρίθηκε απο την τριμελή εξεταστική επιτροπή

……	……	……
Κιαμάλ Πεκμεστζή Καθηγητής ΕΜΠ	Δημήτριος Σούντρης Αν. Καθηγητής ΕΜΠ	Γεώργιος Οικονομάκος Επ.Καθηγητής ΕΜΠ

….....

Ιωάννης Π.Γαλάνης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Οι αλγόριθμοι Όρασης Υπολογιστών γίνονται ολοένα και περισσότερο δημοφιλείς σε σημερινές εφαρμογές. Συνήθως,εισάγουν σημαντικό φόρτο εργασίας στις εφαρμογές,εξαιτίας της αυξημένης πολυπλοκότητάς τους αλλά και του τεράστιου μεγέθους των δεδομένων που χρησιμοποιούν. Γι'αυτό,δεν εκτελούνται αποτελεσματικά από μονάδες γενικού σκοπού. Αντιθέτως, υλοποιούνται ικανοποιητικά από ειδικού σκοπού υλικό(FPGA ή ASIC) για να βελτιστοποιηθεί η απόδοσή τους.

Σε αυτή τη διπλωματική εργασία,ασχολούμαστε με τον αλγόριθμο ανίχνευσης γωνιών των Harris & Stephens. Σκοπός μας είναι να παράσχουμε μία software λύση στο ζήτημα της υλοποίησης του αλγορίθμου σε FPGA,χρησιμοποιώντας το εργαλείο Vivado High-Level Synthesis της εταιρείας Xilinx. Αφού περάσουμε με επιτυχία την διαδικασία σύνθεσης και παράξουμε την περιργαφή επιπέδου καταχωρητή,ξεκινάμε να εισάγουμε ορισμένες βελτιστοποιήσεις,ώστε να επιτύχουμε υψηλότερες επιδόσεις. Τελικά,εκμεταλλευόμενοι τις διαδικασίες βελτιστοποίησης του Vivado HLS σημειώσαμε μεγάλη επιτυχία μειώνοντας το χρόνο εκτέλεσης,αυξάνοντας την απόδοση και χρησιμοποιώντας λιγότερη μνήμη. Η συσκευή στην οποία στοχεύεται η υλοποίηση και με βάση την οποία πήραμε μετρήσεις είναι η πλακέτα Kintex-7 της Xilinx.

Τα αποτελέσματα της παρούσας εργασίας παρουσιάστηκαν στο συνέδριο HiPEAC 2015,Workshop in Recofigurable Computing (WRC) in Amsterdam,2015(*https://www.hipeac.org/2015/amsterdam/schedule/#wshop)*
*"***A Framework for Rapid System-Level Synthesis Targeting to Reconfigurable Platforms :A Computer Vision Study "**,*Dionysios Diamantopoulos, Ioannis Galanis, Kostas Siozios, George Economakos, and Dimitrios Soudris*.

Το κείμενο της διπλωματικής οργανώνεται ως εξής:

Στο Κεφάλαιο 1 υπάρχει η εισαγωγή στα FPGA και γίνεται ειδική αναφορά στον ειδικό τρόπο προγραμματισμού του.

Στο Κεφάλαιο 2 αναλύεται ο επιστημονικός κλάδος της Όρασης Υπολογιστών,καθώς και η σχέση του με τα FPGA.

Ακολούθως στο Κεφάλαιο 3 ,παρουσιάζουμε τον αλγόριθμο ανίχνευσης γωνιών Harris. Αρχικά,εξηγούμε τον τρόπο λειτουργίας του,δίνοντας λεπτομέρειες για τις βασικές του συναρτήσεις. Στη συνέχεια κάνουμε μία γενική εκτίμηση των αναγκών μνήμης του αλγορίθμου,χρησιμοποιώντας το εργαλείο valgrind.

Στο Κεφάλαιο 4 περιγράφονται οι μετασχηματισμοί που ήταν απαραίτητο να γίνουν ώστε η υλοποίηση να μπορεί να περάσει από τη διαδικάσια της σύνθεσης. Στη συνέχεια,αναλύεται η στρατηγική σχεδιασμού καθώς παρουσιάζονται αναλυτικά οι βελτιστοποιήσεις που έγιναν. Επόμενα,στο κεφάλαιο 5 συνοψίζονται τα επιτεύγματα που αφορούν τη βελτίωση της απόδοσης της υλοποίησης του αλγορίθμου(χρόνος εκτέλεσης,μνήμη διεκπαιρεωτική ικανότητα-throuhgput) και τη εξοικονόμηση της χρήσης των διαθέσιμων πόρων. Τέλος,στο κεφάλαιο 6 παραθέτουμε τις μελλοντικές κατευθύνσεις της επιστημονικής έρευνας με βάση τις τελευταίες τάσεις στον κλάδο των Συστημάτων σε Ψηφίδα(System on Chip - SoC).

Λέξεις κλειδιά: Όραση Υπολογιστών,Σύνθεση Υψηλού Επιπέδου,Αλγόριθμος Harris,FPGA,Kintex-7, Ανίχνευση γωνιών, Vivado HLS.

Abstract

Computer Vision algorithms become more and more popular in modern applications. They usually introduce significant performance workload,due to their increased complexity and intensive size of the data input. This is why they are not efficiently performed by general-purpose computing systems. However,they are adequately implemented onto specific hardware(for example ASICs or FPGAs) in order to optimize their execution.

In this diploma thesis,we deal with the Harris & Stephens corner detection algorithm. Our purpose is to provide a software solution of mapping the algorithm onto an FGPA device,using the Vivado High-Level Synthesis tool of Xilinx. After going through the synthesis flow and produce the RTL description successfully,we started introducing several optimizations,in order to achieve higher performance. We began from simple transformations and continued to more complex ones,which aimed at transforming the structure of our implementation. Finally,we took advantage of Vivado HLS optimization directives and reached great success by reducing the runtime,increasing throughput and requiring less memory. The target device of our implementation which gave those measures is the Xilinx Kintex-7 board.

This work was presented in HiPEAC conference in Workshop in Recofigurable Computing (WRC) in Amsterdam,2015(*https://www.hipeac.org/2015/amsterdam/schedule/#wshop)*
*"***A Framework for Rapid System-Level Synthesis Targeting to Reconfigurable Platforms :A Computer Vision Study ",***Dionysios Diamantopoulos, Ioannis Galanis, Kostas Siozios, George Economakos, and Dimitrios Soudris*

The thesis text is organized as follows:

In Chapter 1,there is an introduction to FPGA devices and a more specific reference is made for their unique programming style.

In Chapter 2 it is discussed the scientific field of Computer Vision and its relationship with FPGAs.

In the following chapter,Chapter 3,we present the Harris corner detection algorithm. Firstly,we explain its functionality,giving details for its basic functions. Then we make a profiling for the algorithm's memory needs,using the valgrind tool.

Chapter 4 describes the transformations that were necessary so the implementation could be synthesized. Then, our design strategy is explained in detail and then we perform several optimizations that are discussed extensively. Next,in Chapter 5 we discuss the achievements in performance of our implementation(latency,memory and throughput) and the utilized resources. Finally,in Chapter 6 we present the possible future work according to the latest trends in the field of hybrid systems that use System on Chip (SoC) architecture.

Keywords: Computer Vision,High-Level Synthesis,Harris algorithm,FPGA,

Kintex-7, Corner detection, Vivado HLS.

Acknowledgements

For the completion of this thesis, principally I would like to express my sincere gratidute to Prof. Mr Dimitrios Soudris for inspiring me through his teaching and reasearch.I would also like to thank him for trusting me to deal with such a demanding,as well as interesting scientific task.

I would like to also thank Dr. Manolis Lourakis for his contribution of the C source code.

In addition,this thesis would not have been completed succesfully without the contribution of PhD student Dionysios Diamantopoulos. His relentless effort of supporting my reaserch with answering any question I had and giving immediately solution to any obstacle we encountered, played a major role in completing our research succesfully.I would like to thank,also,Mr George Lentaris for supporting us with valuable ideas throughout this work. Finally,Iwould like to thank Dr Kostas Siozios for his precious comments all over our reaserch and for his contribution in composing our conference paper.

Furthermore I would like to thank all my teachers throughout all of my education years that helped me to develop my way of thinking .Of course I could not forget the support of all of my friends all these years in National Technical University of Athens. I would like to give them my gratitude for all the times we worked together in lab sessions,cooperated to finish projects,and studied hard to complete our studies succesfully.

Last but not least,I would like to give a special thank to my family that gave me the opportunity to pursue my dreams and become a qualified engineer.Their endless love and support throughout all these years gave me the opportunity to complete successfully my studies in NTUA.

Table of Contents

Chapter 1:Introduction

1.1 Introduction to FPGA

An FPGA(field-programmable gate array) is an electronic device that consists of an integrated circuit (IC) that allows its user to configure it for a variety of applications. In fact, almost every algorithm that is computable can be performed by an FPGA device. Unlike the Application-Specific Integrated Circuit(ASIC),FPGA 's main feature is that it can be dynamically re-programmed without any restriction after being manufactured. Thus the functionality of the design can be updated to any possible change in late design cycle and adapt to new,higher standards. To do so,a FPGA contains a large number of programmable logic blocks,alongside with reconfigurable interconnects which can be re-connected in many different combinations,depending on the application 's requirements. Most FPGA include additional resources to implement complicated digital operations, such as high-speed transceivers, high-speed I/Os, memory elements like blocks of RAM or flip-flops(FFs),and also analog components like analog-to-digital converters(ADCs) and digital-to-analog converters(DACs). [1]

1.1.1 History

FPGA 's ancestors where simple programmable logic devices(PLDs) and programmable read-only memory (PROM) .PROM was the non-volatile memory that can be loaded with information. It could have been programmed either in a factory-level or a user-lever(field-programmable). PLDs were electronic devices which contained an array of logic gates OR and logic gates AND,also both factory and field-programmable. In the 80's,Xilinx co-founders R.Freeman and B.Vonderschmitt introduced the first commercial field-programmable gate array. In the '90's,FPGAs production grew explosively [2] . Other vendors emerged and

the market percentage was shared. In early '90's,they were initially used in telecommunications and networking and later that decade expanded to consumer,automotive and industrial fields of market.

1.1.2 Latest Trends

Traditionally, FPGAs where slower,consumed more energy and achieved less functionality than the ASICs. However,nowadays FPGAs have evolved significantly and they can provide solutions that can be preferred from an ASIC one. They can achieve:

- low power
- increased speed
- low materials cost
- increased possibilities for re-configuration 'on-the-fly'.
- short time to market
- low non-recurring engineering(NRE) costs

In addition,according to Xilinx's estimations,there are recent technology and market changes that are changing the FPGA/ASIC relation:

- Integrated circuit costs grow sharply
- ASIC high-complexity extends design time
- R&D resources are decreasing
- Costs for slow time-to-market is increasing
- Financial constraints in a poor economy are driving low-cost technologies

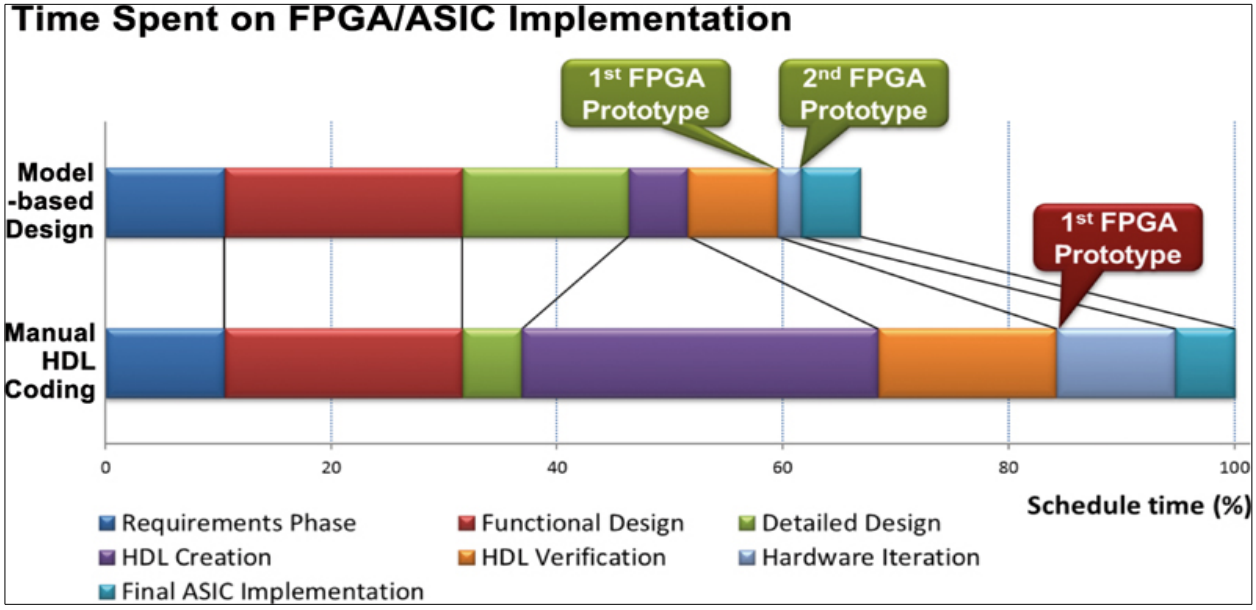


Figure 1: Time Spent on FPGA/ASIC Implementation

Source:Mathworks.com

Since FPGAs have lower material cost,can enter the market in a short amount of time and the NRE costs are constantly reducing,the cost per unit will finally be less than ASIC at higher volumes.

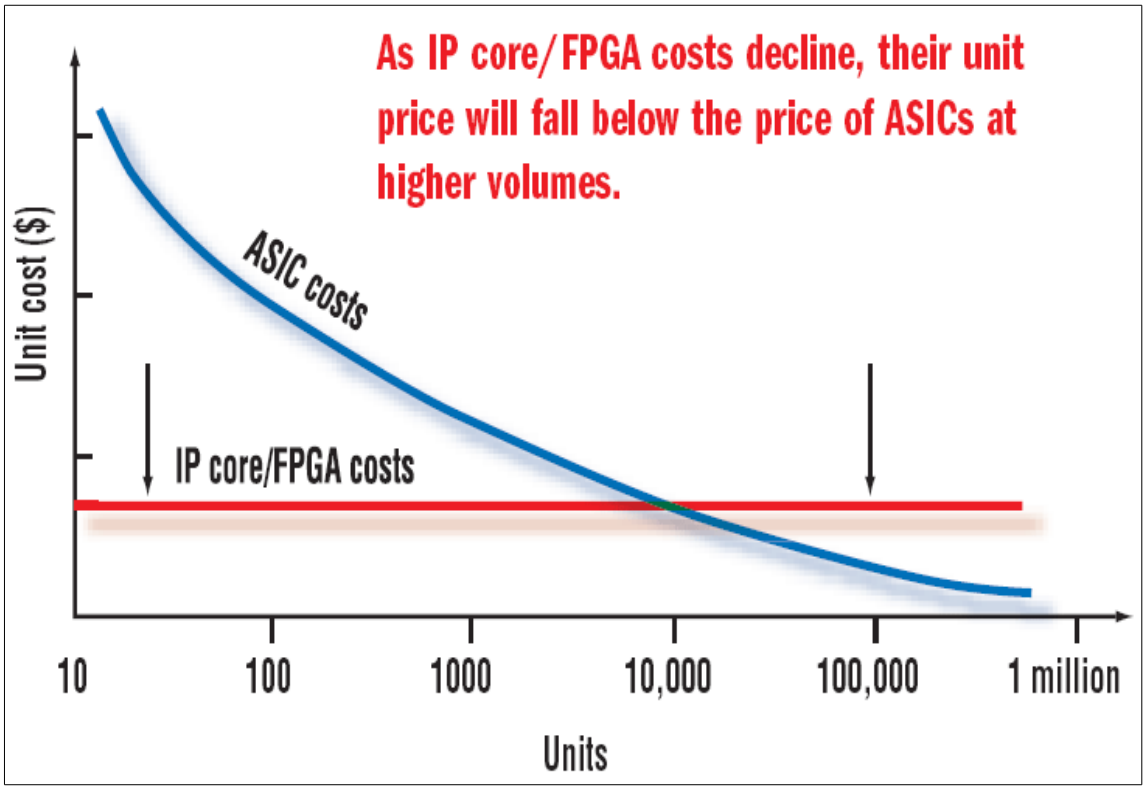


Figure 2: Fpga/Asic cost per unit

These trends make FPGAs a highly flexible alternative than ASICs for a larger number of higher-volume applications than they have been historically used for, to which the company attributes the growing number of FPGA design starts:

- 2005: 80,000[3]
- 2008: 90,000[4]

This evolution would not have been achieved if there were not the explosive increase of the logic gates of FPGA :

- 1982: 8192 gates, Burroughs Advances Systems Group, integrated into the S- Type 24-bit processor for reprogrammable I/O.[5]
- 1987: 9,000 gates[6]
- 1992: 600,000, Naval Surface Warfare Department[7]
- Early 2000s: Millions[8]

As a result,it is concluded that FPGA market has expanded significantly through the past three decades:

- •1985: First commercial FPGA: Xilinx XC2064 [6]
- •1987: $14 million [6]
- •≈1993: >$385 million [6]
- •2005: $1.9 billion [9]
- •2010 estimates: $2.75 billion [9]

As we can observe in the graph below,in 2010 the market was dominated by Altera and Xilinx,but there were other smaller vendors too and all together shared the market.

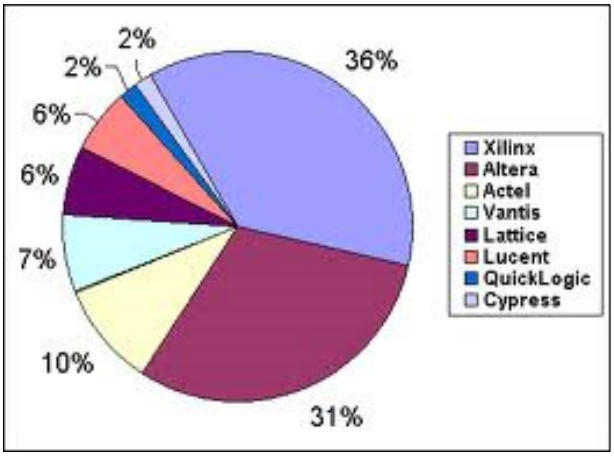


Figure 3: FGPA market share in 2010

Latest estimations,though,have shown that approximately 90% of the market in 2012 was shared between Altera and Xilinx(Xilinx 47%, Altera 41%), with combined revenues in excess of $4.5B and a market cap over $20B.In future, we expect that the programmable logic fabric will continue to rise, since the major companies insist on investing heavily for new technologies and manufacturing.

It is also the main feature of FPGA,that can execute an implementation in parallel,thanks to their concurrent nature,that makes them faster than a soft microprocessor in a variety of applications. So recently, the main trend in FPGA

technology is to combine the advantages of the logic blocks of the traditional FPGA design with embedded microprocessors and the required peripherals to develop a whole system-on chip(SoC) device.[1]

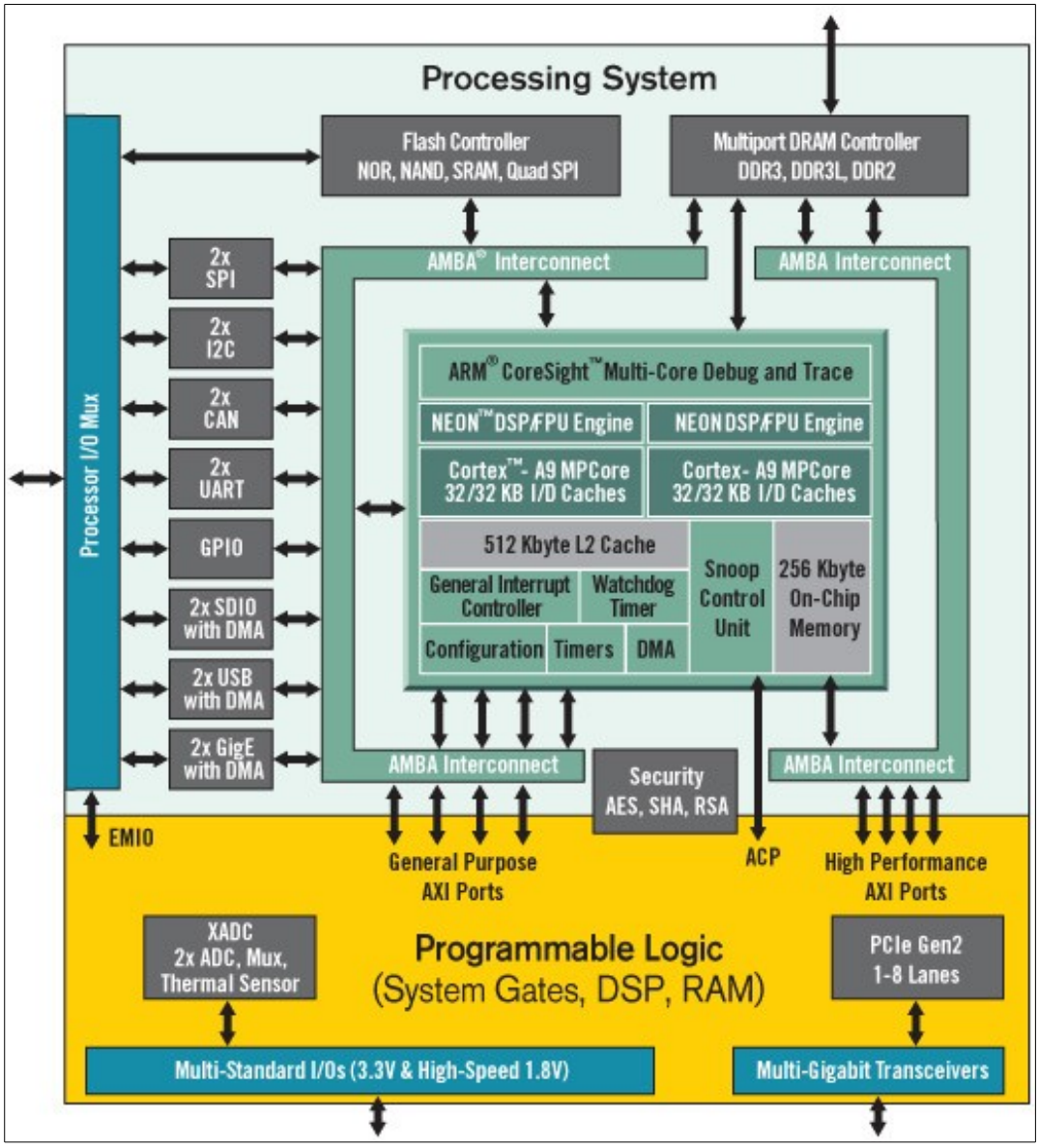


Figure 4: System-on-Chip: Zynq series

Source: Xilinx

In 2010, Xilinx presented the first SoC device(Zynq™-7000) that combined

features of an ARM microcontroller (hard-core implementations of a 32-bit processor, memory, and I/O) with an FPGA core. By including an ARM processor-based platform into FPGA family enables developers to apply a conjunction of serial and parallel processing to their embedded system designs, for which the general trend has been to progressively increasing complexity. The high level integration (commercial levels at 28nm) are able to cut power consumption and power leakage,resulting in a smaller design,less parts cost an more reliable implementation,since most failures in modern electronics take place on PCBs connections and not inside the actual chips.[10]

1.1.3 Applications

As the FPGA technology evolves rapidly,the number of applications they are used in has been expanded. From their initial purpose,ASIC prototyping,to Aerospace and Defense,from medical electronics to consumer electronics, there is almost no field of the modern electronic industry that has not been affected my the rise of the FPGA technology.[11]

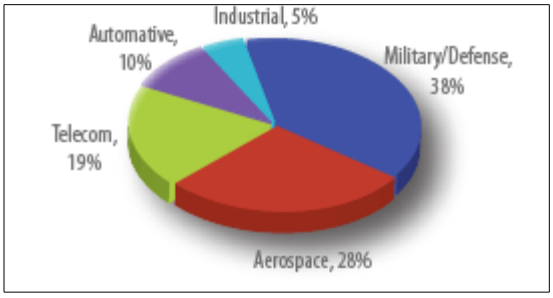


Figure 5: Fpga market applications
Source:Aldec

1.1.4 FPGA Architecture

An overall view of a FPGA board would reveal several electronic components that cooperate in order to implement the desired digital circuit. The main functional unit of the FPGA board is the Configurable Logic Blocks(CLB). Each board contains a large number of CLBs,which are organized in a two dimensional array and the are interconnected via horizontal and vertical routing channels.

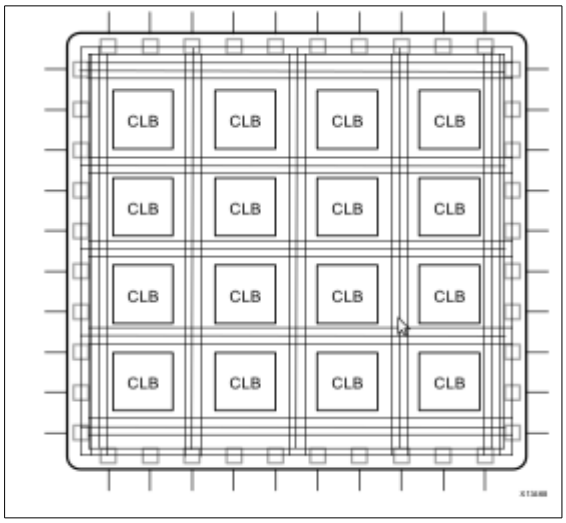


Figure 6: Array of CLBs

There are also several I/O blocks that allow the device to communicate with the outside environment. A more detail view of array of CLBs would show the following figure:

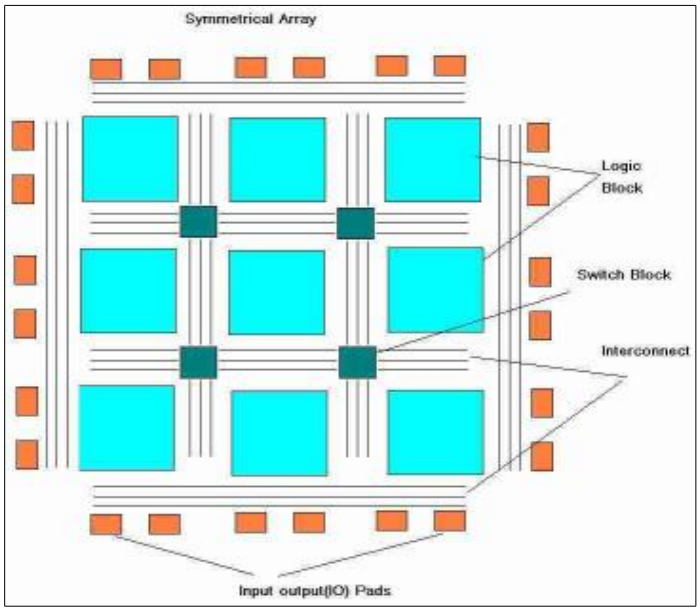


Figure 7: Structure of FPGA

Each CLB comprises a number of slices ,each of one contains a number of logic cells. A logic cell consists of the following:

- Look-up table(LUT):Responsible for logic operation.
- Flip-Flop(FF) : Stores the result of the LUT.
- Network connection units:Connect each element to one another.
- Input/Output(I/O) pads:Physical ports to interchange data in and out of the board

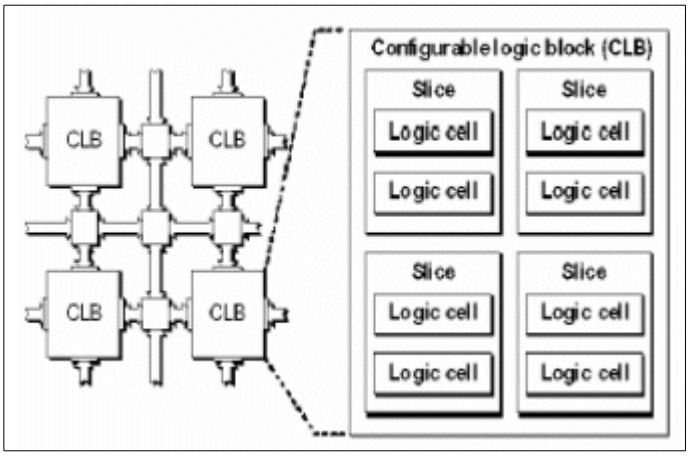


Figure 8: FPGA basic element

Combining all of the CLBs is responsible for implementing any kind of application. Responsible for making different designs are the switch boxes,which are configured each time depending on the circuit they implement. Actually,there switch box consists of a matrix with 6 pass transistors,as it is shown below:

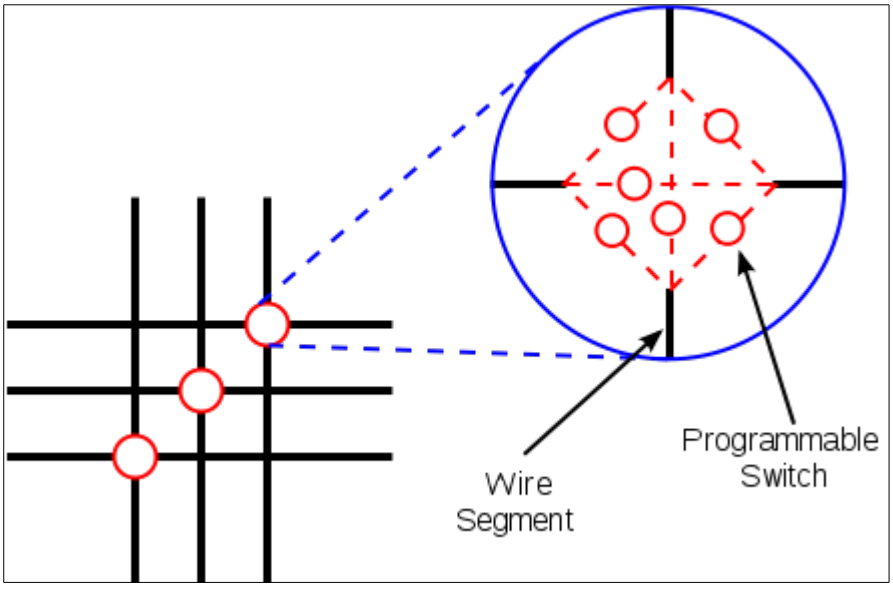


Figure 9: Programmable interconnect

However, this architecture brings limitations,in terms of computational throughput,resources and clock frequency.[12]

1.1.4.1 Memory

FPGA boards are equipped with various memory elements that can be used as RAM,ROM or shift-registers. These units are block RAMs (BRAMs),LUTs and shift registers.

The BRAM is a dual-port RAM component embedded into the FPGA board that can achieve storage of a large set of data. Two types of BRAMs with different capacity are usually instantiated: 18k or 36k bits. The total number of these memories devices is always specific in every board. Also,the dual port operation of these memories can provide access to different locations in the same clock cycle(parallel behavior).

1.1.4.2 LUT

In every modern FPGA device, LUTs are the fundamental elements that can apply every logical function of N boolean variables. In fact, it is a truth table that depending on the input values,generates different functions to produce output. Since the number of inputs is N, the maximum output values that a LUT can calculate is 2^N ,which corresponds to the memory locations that are accessed by the LUT. Hence,the number of implemented functions are 2^{N^N} .A regular value of N is a Xilinx FPGA board is 6.

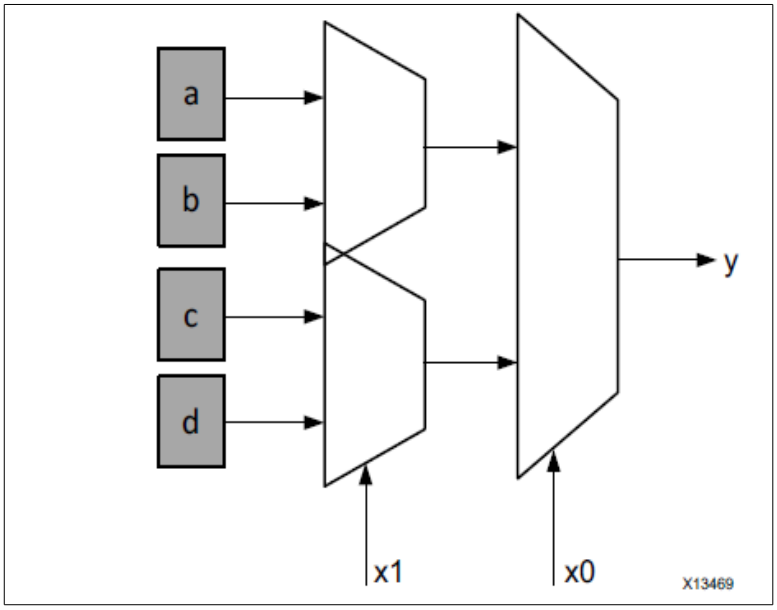


Figure 10: Functional Representation of a LUT as Collection of Memory Cells

Source: Xilinx

1.1.4.3 Flip-Flops

Flip-flops are the basic unit of storage in FPGA design. Each flip-flop includes several inputs:data input,clock input,clock enable,reset and one data output. The functionality of the FF is to preserve the value for more than one clock cycle (when the enable input is ON). If a new data input occurs,only if clock value and clock enable are to logic 1 (or ON in other words),then the input data value is passed to the output.

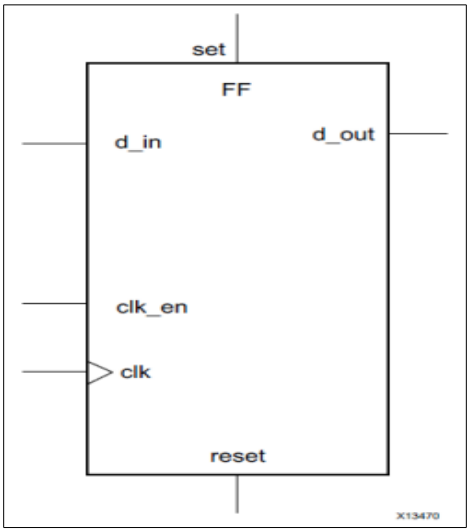


Figure 11: Structure of a Flip-Flop
Source:Xilinx

Modern FPGA devices are equipped with additional components which increase the computational efficiency of the board. Such elements are:

- Embedded memories(RAMs,ROMs and shift-registers)
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

The combination of all these components results in the whole architecture schematic of the modern FPGA devices:

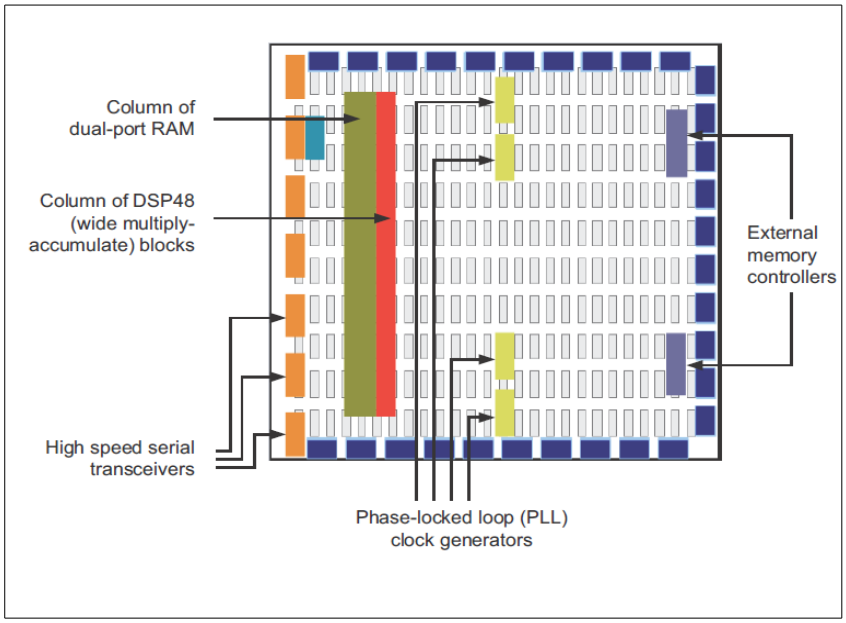


Figure 12: Modern FPGA structure

Source:Xilinx

1.1.4.4 DSP Blocks

Probably the most complex computational unit into the Fpga fabric is the DSP block. Modern FPGA vendors have established actual DSP devices into Fpga,in order to support the increasing amount of computational load. Dsp's consist of adders,subtractor units and multipliers, combined to compose an arithmetic logic unit(ALU).

1.2 Programming the FPGA

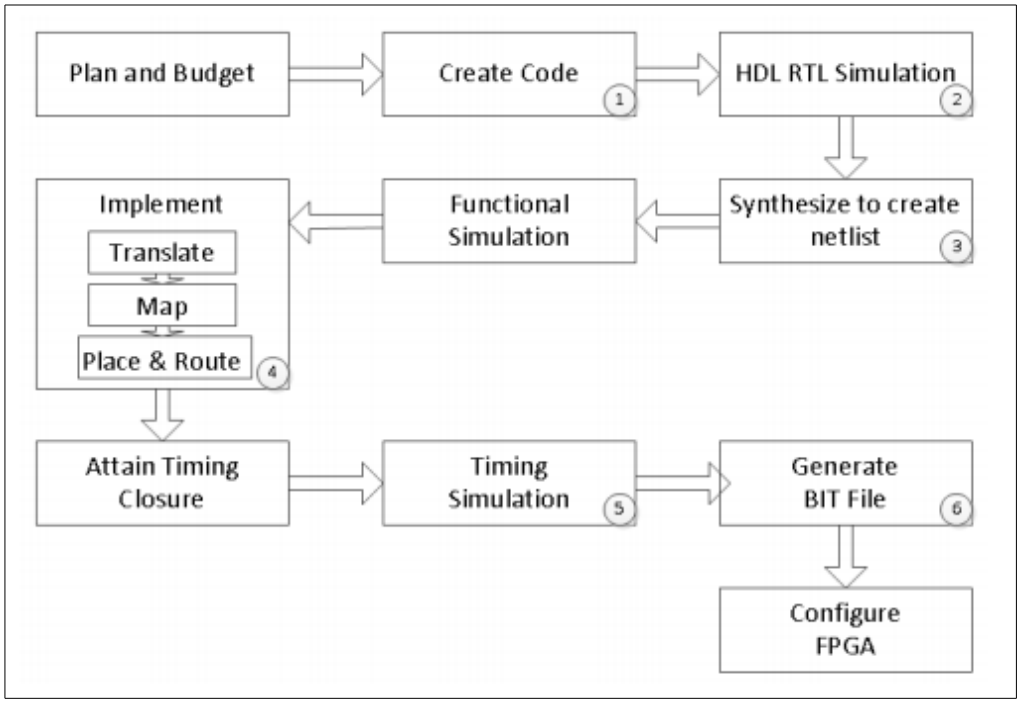


Figure 13: FPGA design flow

Source:Xilinx

Traditionally,to program an FPGA device the user has to provide a register-transfer-level(RTL) description,which is applied by code in a hardware description language(HDL). At most cases,FPGA boards are accompanied with a design tool by the vendor which is used to generate the technologically-mapped netlist. This netlist is implemented to the actual board via a special process,called place-and-route(PnR). Once the output is verified(with validation of the generated map , simulation and examination if the meeting the timing constraints are met),then the binary file called bitstream is produced. Finally,the bitstream file is loaded to the FPGA board via a serial interface(JTAG) or an external memory and in the end the board is programmed-configured to the desired circuit.

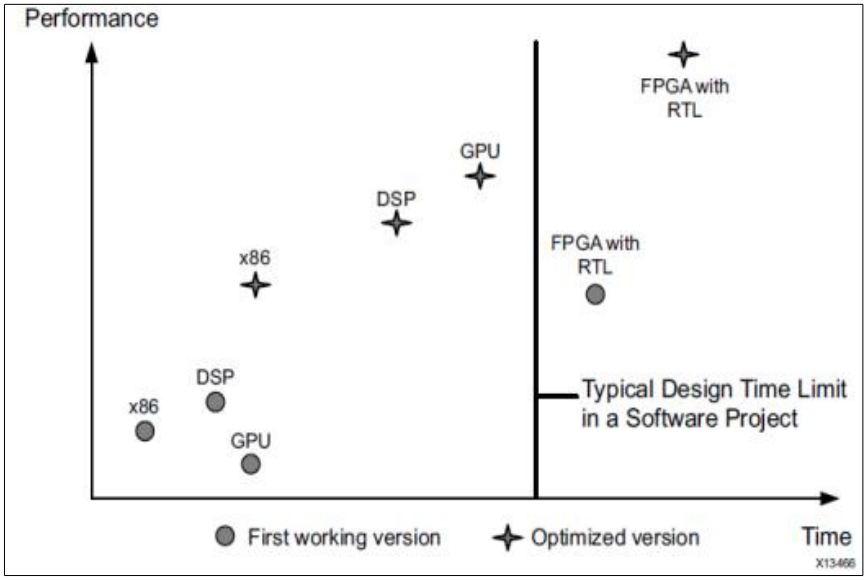


Figure 14: Design Time vs. Application Performance with RTL Design Entry

Source: Xilinx

Historically,FPGA programming is implemented by the two most common HDLs, VHDL and Verilog. After the developer has written the code, the next step in the design circle is to simulate the RTL description in every stage of the design process. To do that, an specific validation program is created ,called testbench,which can determine if the implementation requirements are met and if not, warn the user that there is an error. Since no errors occur,the design flow continues to the next stage: the specialized software produces the netlist by the synthesis procedure and it is simulated again to confirm that there are no errors. Finally, the design is applied to the FPGA board.

It is obvious from above that the programming flow of FPGAs has a lot of complex stages that may cause substantial delays in the completion of a project. We could compare the difficulties in HDL programming of an FPGA design,to the assembly language programming in software engineering. Thus,as it is shown in the figure below,the traditional FPGA design flow with RTL results in limitations in terms of implementation time and achievable performance for different computation platforms.

In that previous figure is demonstrated that despite the higher performance for both initial and optimized implementation,compared to standard and specialized processors,FPGA development time required to arrive at this performance is far beyond the duration of a typical software development time. Therefore, FPGAs were usually used when the design requirements could not have been met with any other means,such as multiple-processor designs. [13]

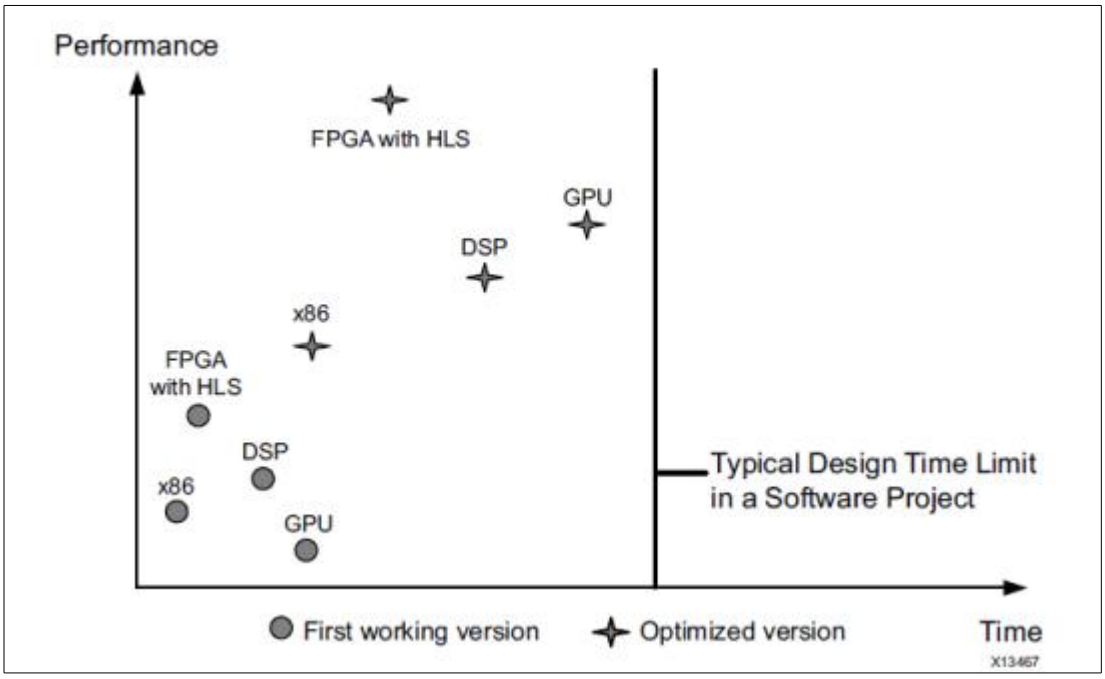


Figure 15: Design Time vs. Application Performance with Vivado HLS Compiler
Source: Xilinx

However,recent advances in that field have come out and they are able to remove any difference between the programming methods of a typical processor and an FPGA. As there are compilers for high-level languages,like C,to different processor architectures,Xilinx created Vivado® High-Level Synthesis(HLS),which is a compiler that provides the same functionality for C/C++ programs targeted to different FPGA boards. The results of the comparison between the HLS compiler and other,standard or specialized compilers,is figured below.[13]

It is presented a large difference in favor of the HLS compiler,as it achieves the highest performance,within the design time of a x86 processor and a DSP. That gap is justified if we realize that assuming the five-stage pipeline,5 consequent instructions would execute in a processor like the following:

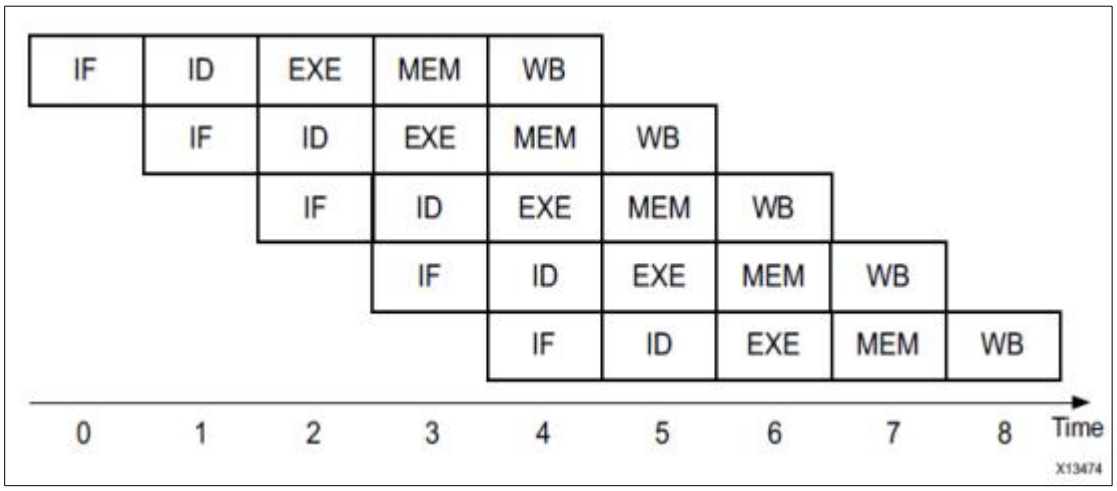


Figure 16: 5-stage Pipeline

Source: Xilinx

However,the natural execution on an FPGA board is not being held on a common computational platform. It executes each single program on a custom circuit. If the program changes,then the implementation changes and so does the circuit.
So,the "exe" stage for a single instruction appears below:

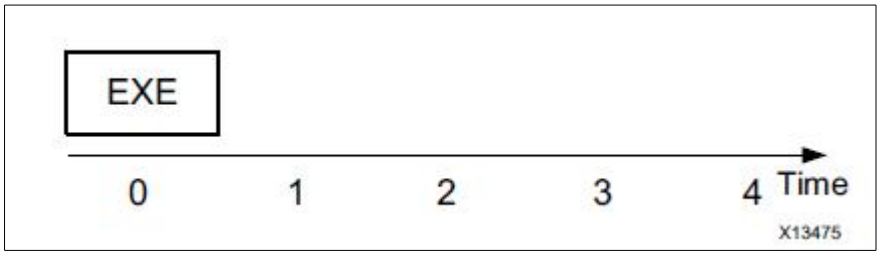


Figure 17: Execution stage for a single instruction

Source: Xilinx

Therefore,given the same set of 5 instructions as before,it is easy to come to the following figure:[13]

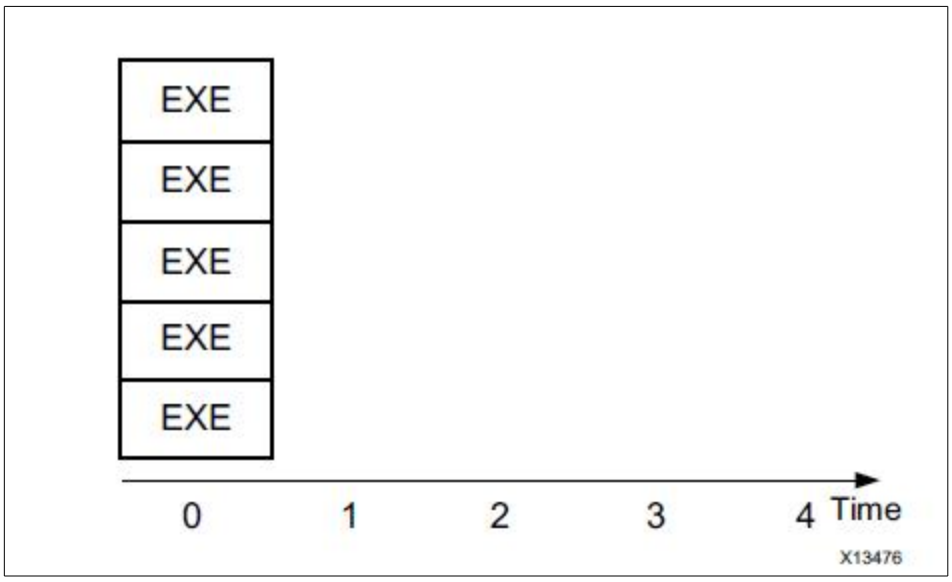


Figure 18: Concurrent execution

Source: Xilinx

Comparing the previous graphs we can come to the conclusion that FPGAs have a nominal performance that is lot faster than a processor. Of course,actual numbers depend on each application,but in general FPGAs are at least 10x faster than a processor,concerning computationally demanding algorithms. Therefore, FPGAs have grown explosively the past decade, concerning computational performance,since they can accomplish true parallel execution and high complexity operations better than CPUs. Alongside with the evolution of processing capabilities because of Moore's Law,the gap between FPGA and CPU performance continues to grow. The chart below demonstrates the comparison of FPGA and CPU performance.

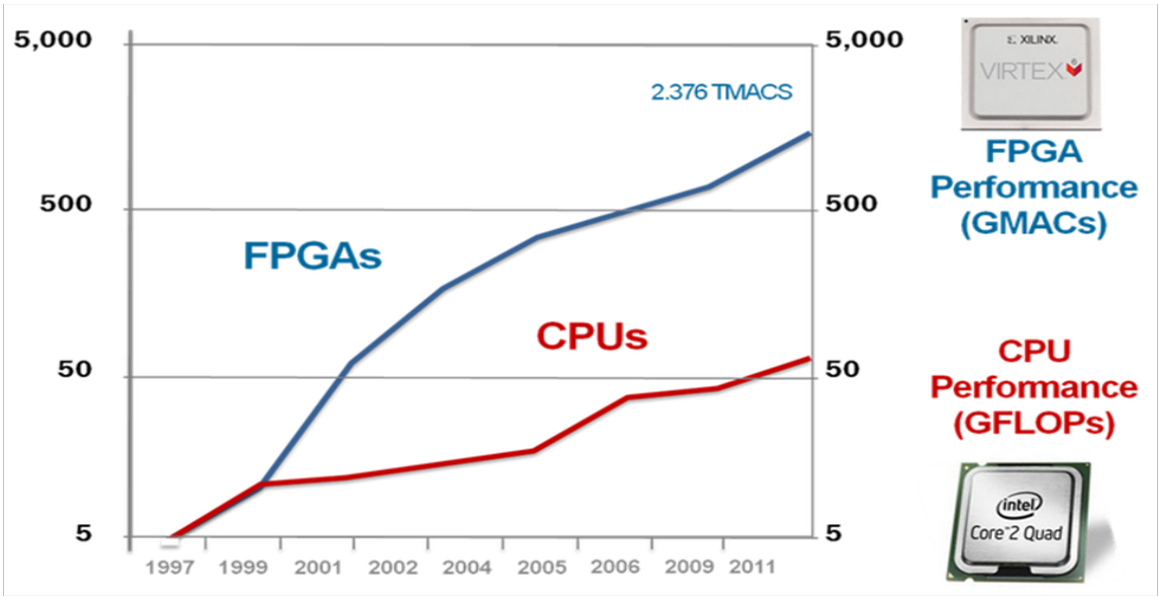


Figure 19: Moore's law comparing FPGA and CPU performance.
Source: National Instruments

One fundamental difference between programming a processor and an FPGA is that FPGA lacks of on-chip memory. So,the HLS tool builds a fast memory architecture and thus the implementation can access one or more memory banks independently.

Another feature that a software engineering has to adapt to when programming an FPGA device is that dynamic memory allocation is not available. Thus,the regular processor code has to be adapted to the special FPGA requirements,as it is presented below:

Processor Code

```
void foo(......)
{
   int *A = (int *)malloc(10 * sizeof(int));
   ....
   free(A);
}
```

FPGA Code

```
void foo(......)
{
   int A[10];
   ....

}
```

Figure 20: FPGA static memory allocation

Source: Xilinx

Except for the memory issues,HLS compiler handles operations (arithmetic and logical) differently than a standard processor.[13] A software developer faces several restrictions when it comes to optimize the performance of an application. The only effect that can be applied is trying to limit as the dependencies between sequential operations as much as possible, or improving memory access pattern in order to increase cache performance. However, HLS has not such constraints: the compiler builds the circuit that specified to the application and the developer has the opportunity to optimize the design's throughput,power consumption and latency. For example,assuming we have the following set of operations:

$$A[i]=B[i]*C[i];$$
$$D[i]=B[i]*E[i];$$
$$F[i]=A[i]+B[i];$$

If the previous instruction set is executed in a standard processor, the only dependency is that A[i] and D[i] must be computed before F[i] (Read-After-Write). The execution would be like that:

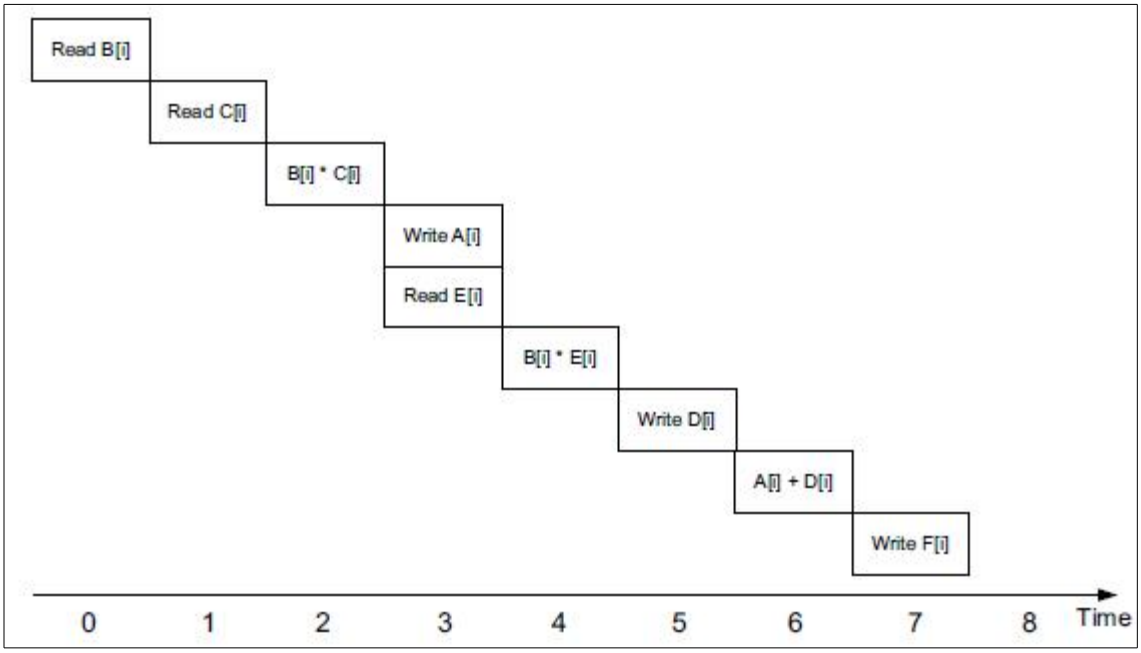


Figure 21: Execution of Example Code on Processor

Source: Xilinx

However, in the HLS compiler the previous set would be executed in less cycles:

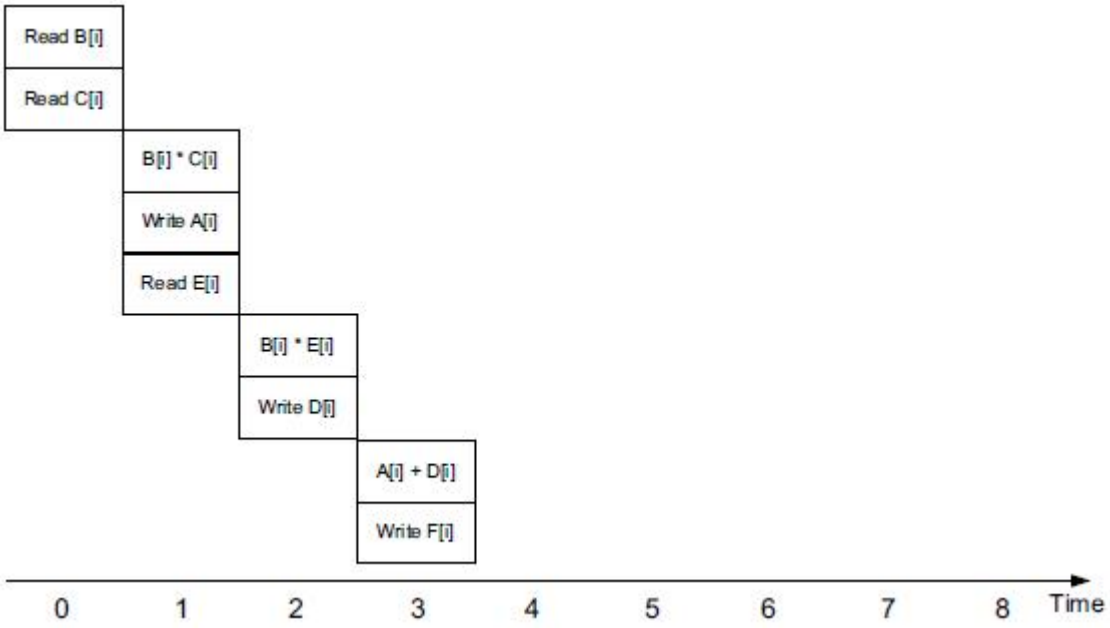


Figure 22: Default execution of HLS Code on an FPGA

Source: Xilinx

This happens because HLS compiler creates a custom memory architecture, depending only on each algorithms requirements. HLS corresponds the arrays to different memory banks,whereas on the processor case the arrays are stored in the same memory space and this is why delays occur.[13]

Same with the HDLs, writing C/C++ with the HLS tool needs software validation,respectively. The reason to write a software test bench is to validate that the software implementation runs without any segmentation faults and that functionality of the implemented algorithm is has correct functionality. A rule of thumb is that the test bench must reach at least 90% code coverage to be advised as an adequate test bench. Hence,the test vectors examine all branches in case statements,conditional if-else statements and for loops. The test bench alerts the user whether the algorithm code behaves the way it is expected. If not, prints an appropriate message informing for the wrong functionality.

A simple example of an algorithm code(on the right) with the corresponding software test bench(on the left) are shown below:

Test Bench Code

```
int main()
{
   int i;
   int B[10];
   int C[10];
   int result;

   for(i=0; i < 10; i++){
        B[i] = i;
        C[i] = i;
   }
   result = example(B,C);
   return result;
}
```

Algorithm Code

```
int example(int B[10], int C[10])
{
   int i;
   int A=0;

   for(i=0; i < 10; i++){
         A += B[i] * C[i];
         if(i == 11)
              A = 0;
   }
   return A;
}
```

Figure 23: Example of Code coverage

Source: Xilinx

In the example above,concerning the assignments, the instruction A=0 will never be executed. HLS is able to recognize if there is an unreachable statement like the above and cuts it off the configurable circuit.[13]

After the software test bench is build the next step to the HLS design flow is to implement the process of co-simulation. From one hand the test bench can detect most of possible errors in the design,but it cannot verify if the functionality remains correct after the implementation being transformed to concurrent execution. The co-simulation stage checks if the C/C++ test bench and the generated RTL have the same behavior. To do so,HLS generates a hardware emulator and simulates how the RTL would function on the device. [13]

Chapter 2: Computer Vision

2.1 Definition

With the term Computer Vision(CV) we refer to the scientific field that includes all the methods for gathering,processing and analyzing data from the real world(usually images) and generate arithmetic or symbolic information as results. The target is to extend the ability of human vision so as to derive conclusions from image processing,which consists of applying several math fields(like algebra,geometry,probability theory or statistics) alongside with laws of physics and learning theory. [14][15]Hence,the field of Computer Vision is multi-scientific fields which can be depicted in the following figure:

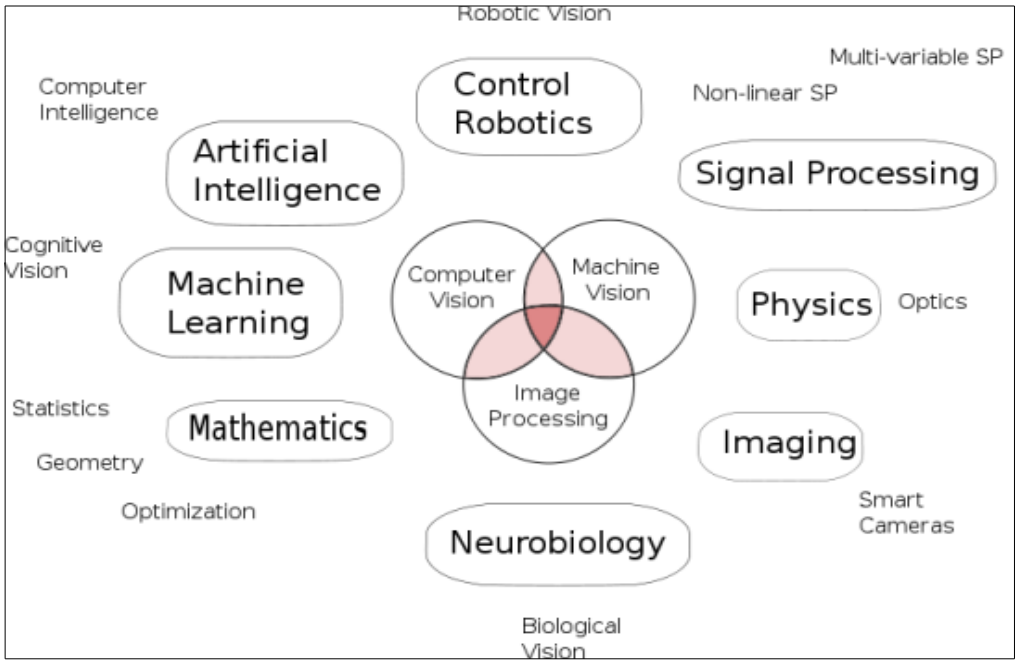


Figure 24:Scientific fields of Computer Vision

Source: Wikipedia

2.2 Applications

Computer Vision algorithms have evolved rapidly the past few years and they cover a vast field of applications:

- Automotive:

  Depending on the application ,CV 's assistance ranges from supporting drivers or pilots in a variety of real-time situations,to more fully autonomous applications of small wheel-robots,cars,aerial vehicles. The main use of CV for all those kind of vehicles is to navigation. In the first category, CV supporting systems warn drivers for obstacles or help pilots for landing of aircraft. In the second, vehicles use CV algorithms to create a map of the environment and thus be able to navigate to the desired route. Space exploration is being held with fully autonomous land-based vehicles,like NASA's Mars Exploration rover or ESA's ExoMars Rover.[16] Some car industries make efforts to introduce unmanned driving cars,but there are some improvements to be done until they reach commercial use.

- Industry(*machine vision,video surveillance,manufacturing applications)*

  In industrial applications,CV provides the necessary information to support the manufacturing process. Usually, products are being examined through automatic procedures to find any imperfections. Alternatively, robotic arms measure position and orientation to implement pick-and-place processes.

- Medical Imaging

  Computer Vision algorithms provide information from image data in order to make medical diagnosis for patients. The image data may vary:X-ray images, ultrasonic images or tomography images. The output could be measurements of organ dimensions,blood flow or more complex information about the structure of the brain or the quality of medical treatments.

Reconfigurable computing has been successful also in many compute intensive areas, including DNA matching,encryption/decryption, image processing, neural networks.

2.3 Computer Vision and FPGA

The complexity of Computer Vision algorithms combined with the growing demand for applications with intensive amount of information(like high definition images or videos) leads to greater amount of computational power. So,general purpose CPU's can satisfy only those applications which have low complexity. For more demanding requirements specific processors like GPUs can perform better.
However, CV algorithms become more complex,as they often demand for example,nonlinear optimizations,in order to be more accurate. In addition, concerning image processing, ordinary image sizes range from 512x512 pixels to 1024x1024. The resulting computational load can reach,sometimes,the level of several million operations. In these cases, VLSI based devices,like ASICs, can meet high performance expectations,for example high-throughput in a real-time CV system. Still,the ASIC approach encounters difficulties: the cost is high,design is time consuming and because the ASIC architecture is not reconfigurable,the design cannot be updated after shipping.

One the other hand, the unique feature of reconfigurability of FPGAs can surpass the limitations created by an ASIC. With the recent improvements in FPGA technology, we can manage to reach very high performance with an FPGA,close enough to an ASIC. The natural concurrent behavior of reconfigurable platforms is a great advantage in implementing CV algorithms. For example, one of the most common operations in image processing and digital signal is convolution. Many machine vision systems use two-dimensional convolution for image filtering, edge detection, and template matching. Convolution of a regular image has four loops and its overall complexity is $O(N^2)$ .With a 3x3 or 5x5 kernel,convolution computations can demand several millions of multiplications and additions. In a standard processor this operation could be quite time consuming,but in FPGA it can be implemented simultaneously. Compared to ASICs, the design can be updated very easily in a time to make any modifications or improvements to the algorithm's implementation. This happens because the FPGA board can be reprogrammed(reconfigured) in a matter of hours, but ASIC needs separate hardware to be added. This sequence of re-programming reminds the CPU operation, where a program is loaded and after another one. Moreover, FPGAs can reach applications beyond CPU. For example, FPGAs

operate in slower clock frequency than CPUs and thus they can be preferred in space robotics applications, since radiation prevents usage of fast-clock CPUs.[17]

Nevertheless, FPGA applications display one major drawback when it comes to implement CV algorithms that use primarily fixed point arithmetic. In this case, computations that include floating-point operations occupy excessively much of the available resources and this becomes even worse,when the operations are repeated many times(like in the convolution example) ,in order to accelerate the implementation,taking advantage of the concurrent behavior of the FPGA architecture. One solution is to make a profile of the data inputs and then determine the minimum digits required to balance precision with the FPGA's available resources. Another solution is to limit those operations as much as possible without losing any accuracy,since automotive vehicles driven by CV algorithm depend strongly on orientation information calculated in floating point operations and thus any deviation of the correct route would have disastrous results.[18]

The optimal solution,though is to embed actual DSPs into the FPGA board. For example ,Xilinx has included DPS blocks into the FPGA fabric. Each DSP contains of three different elements:an add/subtract unit which is connected to a multiplier which has cascade connection to the final add/subtract/accumulator engine. Hence,each DSP block is capable of computing functions of the following form:

$$p = a \times (b + d) + c$$

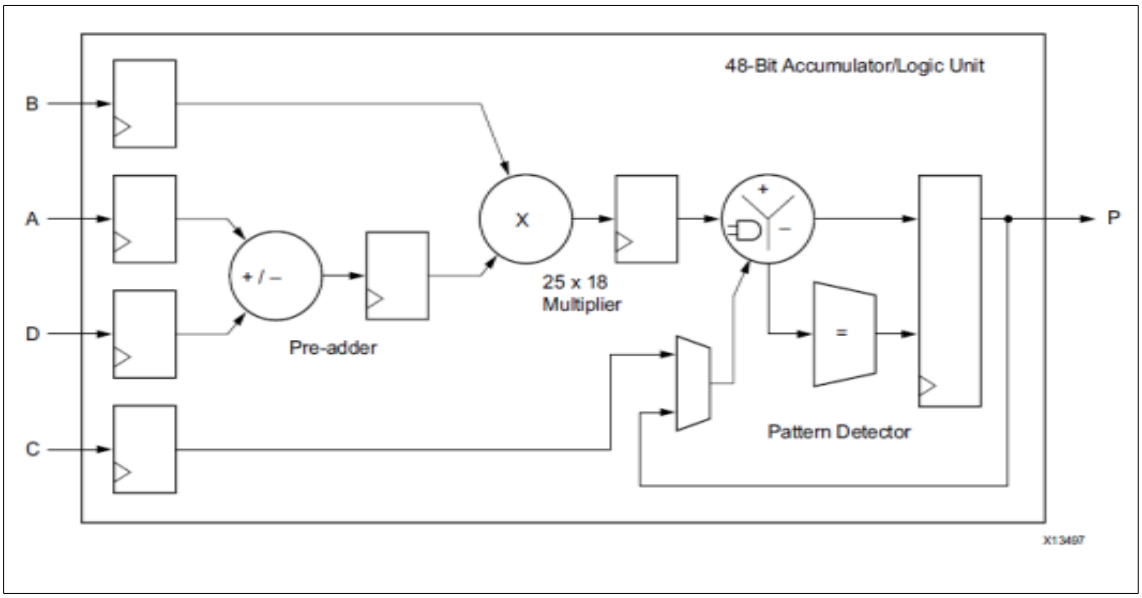


Figure 25: Structure of a DSP Block

Source: Xilinx

Chapter 3:Harris Corner Detector

3.1 Introduction to feature detection

One of the main targets of the field of Computer Vision is to extract features from images and supply the results as inputs to systems,in order to make important decisions(such as triggering an actuator to move a robot hand). This process is called feature detection and refers to all methods and operations that are necessary to calculate at every pixel of an image whether or not satisfies the criteria of each feature. The result is a subset of the image,containing either isolated points,continuous lines or connected regions. Although there is not a clear definition of the meaning of feature, usually we refer to feature as an interesting part of an image,which is repeated two or more times throughout the image.

There have been developed several feature detection algorithms,varying on the desired feature detected, the computational complexity and repeatability. We could divide them into the following groups:

- Edges

  With the term “edges” we refer to the locations in an image where there is a border (an edge) between two regions. There is no a predefined shape for an edge,since it can contain everything. To compute an edge,most algorithms rely on the fact that edges consist of sets that include pixels on an image that have high value gradient magnitude. Hence, edges have one dimensional structure.

- Corners/Interest points

  The term “corner” was suggested when image processing algorithms were detecting edges in the fist place and then they were using the results to compute corners,by determining where there was a strong change in direction. Thus,the algorithms evolved and they stopped calculating exclusively edges, but they were searching for strong values of curvature in

the image gradient. However,it was claimed that those algorithms could detect false corners,when for example detected a small white dot in a black background. These points were named interesting points.

- Blobs/regions of interest or interest points

  In contrast with corners,which are point features,blobs detect region like features of an image. But,they do usually include a centered point or a local maximum. In that sense,we could include blob detectors as interest point operators.

- Ridges

  In case we have stretched objects in an image,it is necessary to use ridges. A ridge could be described as an one-dimensional line that constitutes a symmetry axis and plus its width depends on the local ridge point. Nevertheless, calculating ridge points in gray-scale images is computationally heavier than detecting edges,corners or blobs.

In this thesis,we are going to deal with the field of edge detection. A corner is calculated as an intersection of two edges or as a point where there are two strong and different edge directions in a local region around the point. Corner detection(equal term of edge detection) is usually used in motion detection,image registration,video tracking,3D modeling and object recognition.

3.1.1 Moravec detector

One of the fist efforts in corner detection was Moravec detection algorithm which determines a corner as a point of low self-similarity.[19] The algorithm checks whether the neighborhood of a centered pixel resembles with the other local pixels,by computing the sum of squared differences between the two sections. If the sum has a low value,then the algorithm implies more similarity. So,if the pixel has intensity value that is similar to its neighbors,then the regions will not differ. However, if the pixel belongs to an edge,then it is obvious that the two regions that are in vertical direction to the edge will have strong differences in intensity values, whereas in a parallel direction there would be plenty of similarities. If the pixel belongs in a section that intensity values vary in all directions, then all of the neighborhood patches will look different. Hence,the

corner strength is defined as the lowest sum of squared differences(SSD) between the region of the centralized pixel and its neighbors in all directions(horizontal,vertical and the two diagonals). If the value of SSD is a local maximum, then that point is considered to be an point of interest. Nevertheless, the Moravec detector has a strong drawback:it is not isotropic,meaning that if there is an edge that is in a different direction of its neighbors, then the smallest SSD will be high and thus the edge will be considered a corner incorrectly.

In mathematical terms,this relationship can be presented as follows:

$$E(u,v)=\sum w(x,y)*[I(x+u,y+v)-I(x,y)]^2$$

where:

- E is the computed sum of square differences
- W(x,y) is the window function which can be graphically defined as:

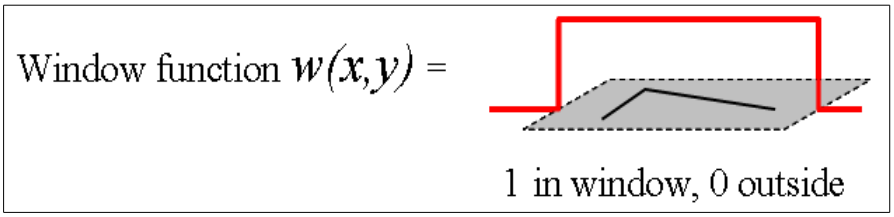


Figure 26: Binary window function

- I(x,y) is the intensity of the pixel
- I(x+u,y+v) is the shifted intensity

We compute shifted intensity in four directions:(u,v)={(1,0),(1,1),(0,1),(-1,1)}
The algorithm searches for the local maximal in min{E(u,v)}. The mathematical formula of the Moravec detector confirms the problems mentioned above:

- because of the binary window function, the response of the algorithm is very vulnerable to noise.
- The step of the operator is 45 degrees and thus important information is eliminated.
- Only the minimum value of E(u,v) is taken into account.

3.1.2 The Harris & Stephens / Plessey / Shi–Tomasi corner detection algorithm

Because of the previous mentioned problems of Moravec detector, Harris and Stephens improved Moravec's detector by taking into account the differential value of the corner,regarding the direction directly and not using shifted regions. [19]
This corner value is often called autocorrelation(introduced in the actual paper). In fact,in paper the mathematical formulas are the clearly calculating the sum of square differences(SSD). The weighted sum is calculated as in Moravec detector :

$S(x,y)=\sum\sum w(u,v)[I(u+x,v+y)-I(u,v)]^2$,where we consider a two-dimensional gray-scale image.

An important development compared with Moravec's operator is that Harris algorithm considers all small shifts and not with 45 degree step. Thus,a Taylor expansion is used to compute $I(u+x,v+y)$ approximately. Letting Ix and Iy be the partial derivatives of the Intensity of an image,we have:

$$I(u+x,v+y)\approx I(u,v)+I_x(u,v)x+I_y(u,v)y$$

Thus the sum expression becomes:

$$S(x,y)\approx\sum\sum w(u,v)(I(u,v)+I_x(u,v)x+I_y(u,v)y)^2$$

In a matrix form,the previous relationship can be rewritten as follows:

$S(x,y)\approx(x\ y)A\begin{pmatrix}x\\y\end{pmatrix}$,where A is the structure tensor:

$$A=\sum_u\sum_v w(u,v)\begin{bmatrix}I_x^2 & I_xI_y\\ I_xI_y & I_y^2\end{bmatrix}=\begin{bmatrix}\langle I_x^2\rangle & \langle I_xI_y\rangle\\ \langle I_xI_y\rangle & \langle I_y^2\rangle\end{bmatrix},$$

where the brackets $\langle\ldots\rangle$ imply averaging (summation over u,v).

Another improvement comparing to Moravec detector is that the window function is now Gaussian,guaranteeing isotropic response. The general form of a Gaussian window function is presented below:

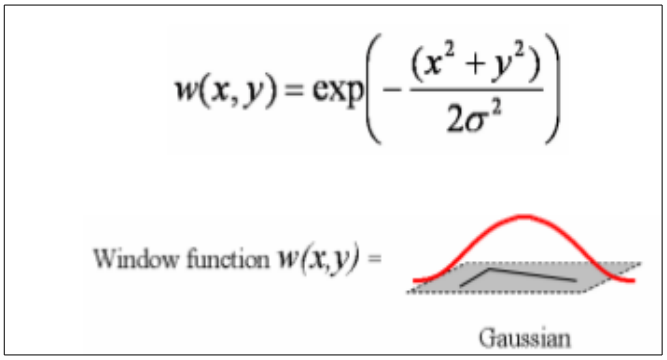


Figure 27: Gaussian window function

In Harris algorithm a corner is considered to have a large variation of S in all of the directions of the vector $(x\,y)$. In mathematical form, this can be expressed through the eigenvalues of the matrix A. If an interest point is examined,then matrix A should have two eigenvalues with grand value. Considering the magnitudes of the eigenvalues, we can determine the following cases:

- $\lambda_1 \approx 0$ and $\lambda_2 \approx 0$,then this point is not of interest.
- $\lambda_1 \approx 0$ and λ_2 has a grand positive value,then at this point there is an edge.
- Both λ_1 , λ_2 have grand positive values,then at this point we have found a corner.

$$M_c = \lambda_1 \lambda_2 - \kappa\,(\lambda_1 + \lambda_2)^2 = \det(A) - \kappa\ \mathrm{trace}^2(A)$$

Nevertheless,because computing the eigenvalues requires a big workload,Harris and Stephens suggested an alternative function M_c which is presented below:

considering that $det(M) = \lambda_1 \lambda_2$ and $trace(M) = \lambda_1 + \lambda_2$ and k is a factor that is chosen depending on the sensitivity level required.
From literature,an accepted value of k is between 0.04–0.15.[20]

A graphical representation of the classification of image points,according to the eigenvalues of M is shown below:

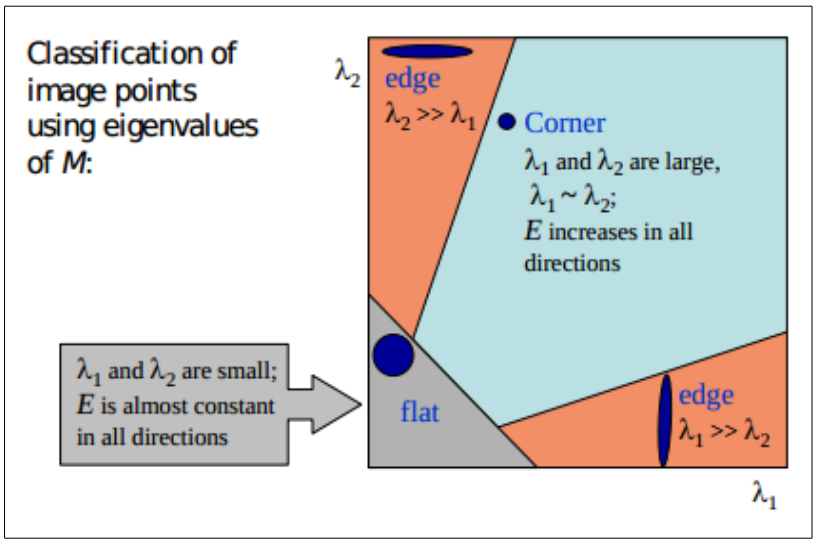


Figure 28: Classification of Image points

Source: Chris Harris & Mike Stephens Plessey Research Roke Manor, United Kingdom © The Plessey Company pic. 1988

3.2 Implementation of Harris Algorithm

For the evaluation of our framework we employed a C-code implementation of Harris & Stephens algorithm[20] ,provided by Dr. Manolis Lourakis[*http://users.ics.forth.gr/~lourakis*]. We present the following flow chart to describe the implementation of the algorithm:

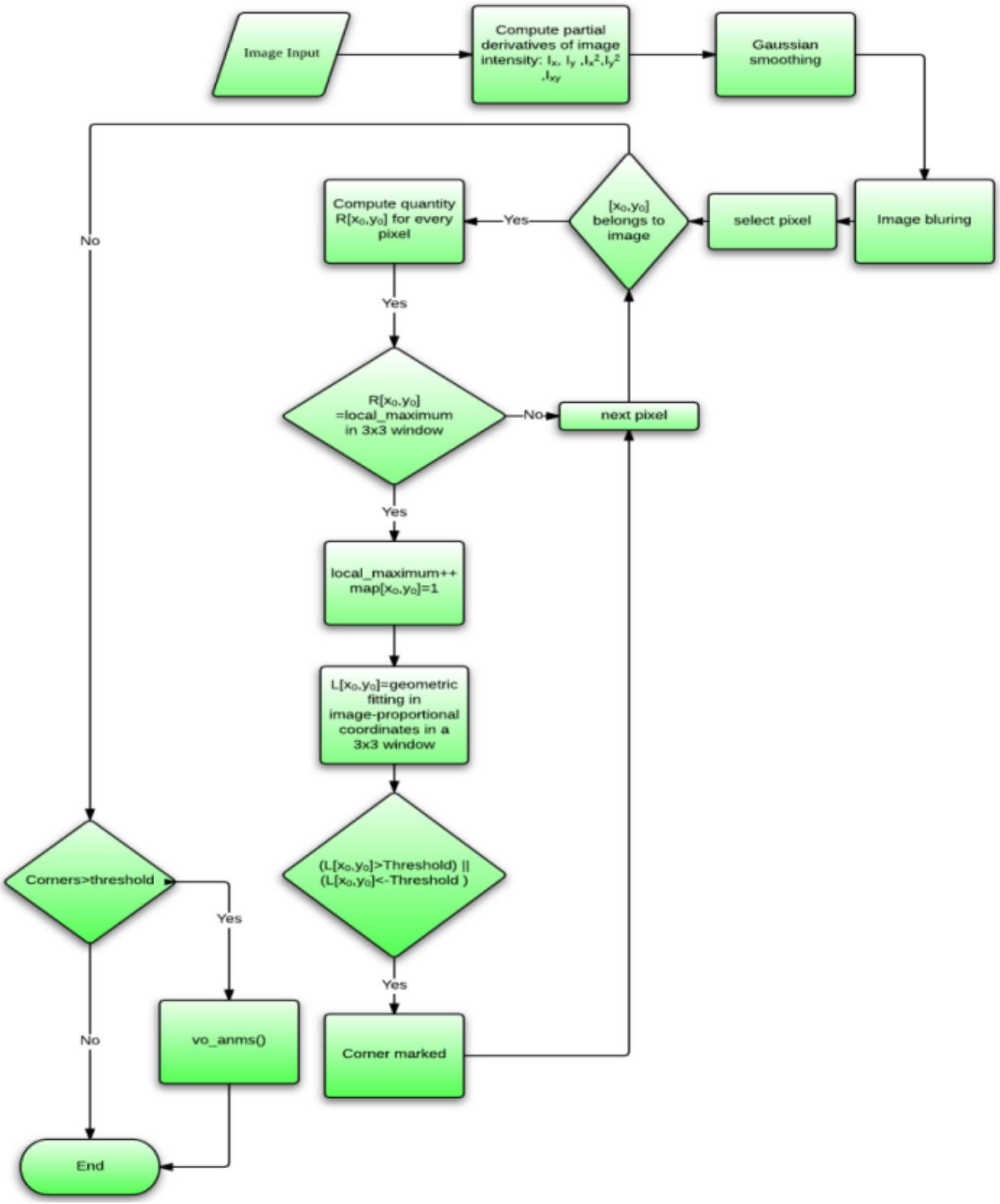


Figure 29: Harris algorithm flow chart

At first,the input image is lead to the function that calculates the partial derivatives of the intensity I of the image in every pixel. That function is called imgradient(). It is presented below in pseudo-code style:

```
     int imgradient5_smo(image[width,height],width,height,gradx,grady){
       for (i=[0,width-1]){
           for ( j=[0,width-1]){
           /* 5-tap derivative kernel          */
     /     * derivative_kernel=[-1 -3 0 3 1]
      /  * smoothing_kernel=[1 6 12 6 1] == [1 6 2*6 6 1]

                 wrkx[i,j]=image[i,j]*derivative_kernel;
                 wrky[i,j]=image[i,j]*smoothing_kernel;
           next_line;
                 }
           for (i=[0,width-1]){
              for ( j=[0,width-1]){
                 gradx[i,j]=wrkx[i,j]*smoothing _kernel;
                 grady[i,j]=wrky[i,j]*derivative_kernel;

           next_line;
                 }


}
```

In the first two-for loop,the algorithm computes in each pixel the derivative and the smoothed value respectively by accumulating the local sum of the 5 value kernel. It uses the values of the previous two and the next two pixels in the particular row. Then,each value is normalized by the equivalent factors.

In the second two-for loop,the kernels are interchanged. The multiplication is done by the previous two and next two pixels vertically,following the same procedure as before. The resulting gradx,grady arrays have the derivative values in each pixel. Notice that as it is obvious,the pixels of the first and the last two rows of the image and the fist two and the last two pixels of each column do not have the derivative values because the algorithm uses windows that go beyond the limits of the image in those regions of pixels.

Because the algorithm cannot calculate the desired values at the borders of the image,there is another function called imgradient_bfill() which computes the differences in those regions .

Then,a Gaussian kernel is formed with mean value μ=0 and standard deviation σ=1. We compute ,according to the previous quantities,the following Gaussian kernel: `kern=[1 12 55 90 12 1]`,which is symmetric and normalized.

Then,image is filtered by imgblur() which is the function that performs an horizontal and then sequentially a vertical convolution,using the Gaussian kernel formed before. The horizontal convolution is held by a window that is 7 points wide and takes into consideration three pixels before and three after. Respectively, the vertical convolution considers the tree pixels above and the three below the center pixel.

```
int imgblurg(gradx,grady,gradxy){

/* separability: convolve horizontally ... */
for (i=[0,height]){
      for (j=[0,width]){
          wrkx[i,j]=convolution_with_gaussian_kernel_horizontal(gradx,gaussian_kernel);
          wrky[i,j]=convolution_with_gaussian_kernel_horizontal(grady,gaussian_kernel);
          wrkxy[i,j]=convolution_with_gaussian_kernel_horizontal(gradxy,gaussian_kernel);
      }
}
  /* ... then convolve vertically */
for (i=[0,height]){
      for (j=[0,width]){
           gradx2[i,j]=convolution_with_gaussian_kernel_vertical(gradx,gaussian_kernel);
           grady2[i,j]=convolution_with_gaussian_kernel_vertical(grady,gaussian_kernel);
           gradxy[i,j]=convolution_with_gaussian_kernel_vertical(gradxy,gaussian_kernel);
      }
   }

}
```

The quantities gradx2,grady2,gradxy correspond to I_x^2, I_y^2, I_{xy} respectively.

Next,we present a typical example of horizontal convolution graphically[20]:

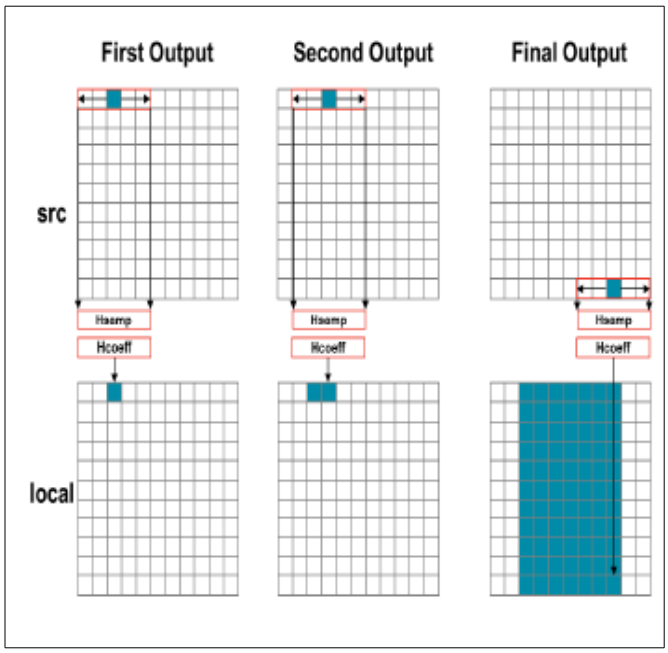


Figure 30: Horizontal Convolution
Source: Xilinx

In that figure we observe that since our window function is a 5-sample kernel,the convolution cannot start at the first pixel. Otherwise,it would need to include pixel values that are actually outside the image. Thus,it starts from the third pixel and ends in the width-3 pixel in each row. The first 5 pixels of the first row of the source image(Hsamp) are used to be convolved with the 5 samples of the kernel(Hcoeff). The first output is now calculated. The same process continues in the second set of Hsamp,until the final value of the last row is computed.

A typical C code for implementing the horizontal convolution would be the following:

```
HconvH:for(int col = 0; col < height; col++){
        HconvWfor(int row = border_width; row < width - border_width; row++){
                int pixel = col * width + row;
                Hconv:for(int i = - border_width; i <= border_width; i++){
                        local[pixel] += src[pixel + i] * hcoeff[i + border_width];
                }
        }
}
```

Then,in the vertical convolution,we perform the same calculations,but in the vertical direction this time,as it shown next:

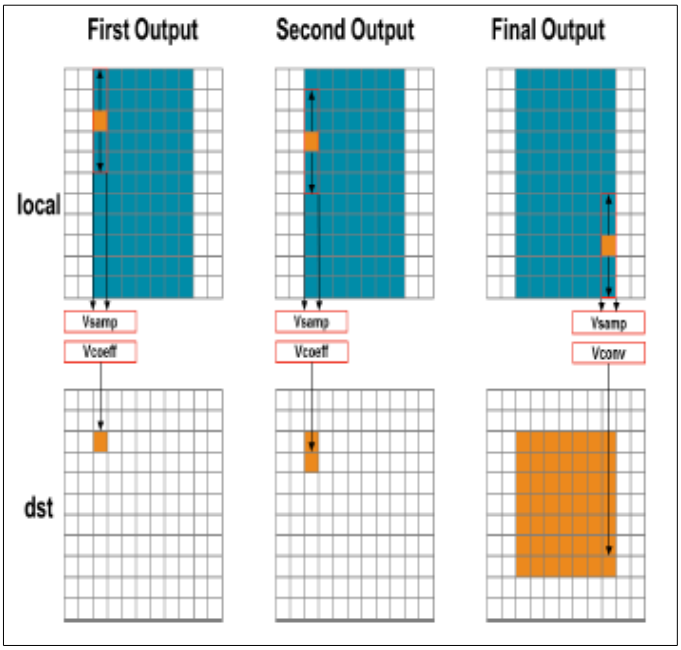


Figure 31: Vertical Convolution

Source: Xilinx

Likewise,we start to compute the destination from the third row,since our kernel is again 5-sample. Consequently,we end in the height-3 row. Then,the process follows the way the horizontal is implemented:all the 5 data samples convolve with the convolution coefficients,Vcoeff in vertical case. After the first value is created using the first 5 samples in the vertical direction,the next set of 5 samples is used to calculate the second output. That process is repeated until the value of the final pixel in the last column is calculated. A typical C code for implementing the vertical convolution would be the following:

```
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
        VconvW:for(int row = 0; row < width; row++){
                int pixel = col * width + row;
                Vconv:for(int i = - border_width; i <= border_width; i++){
                        int offset = i * width;
                        dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
                }
        }
}
```

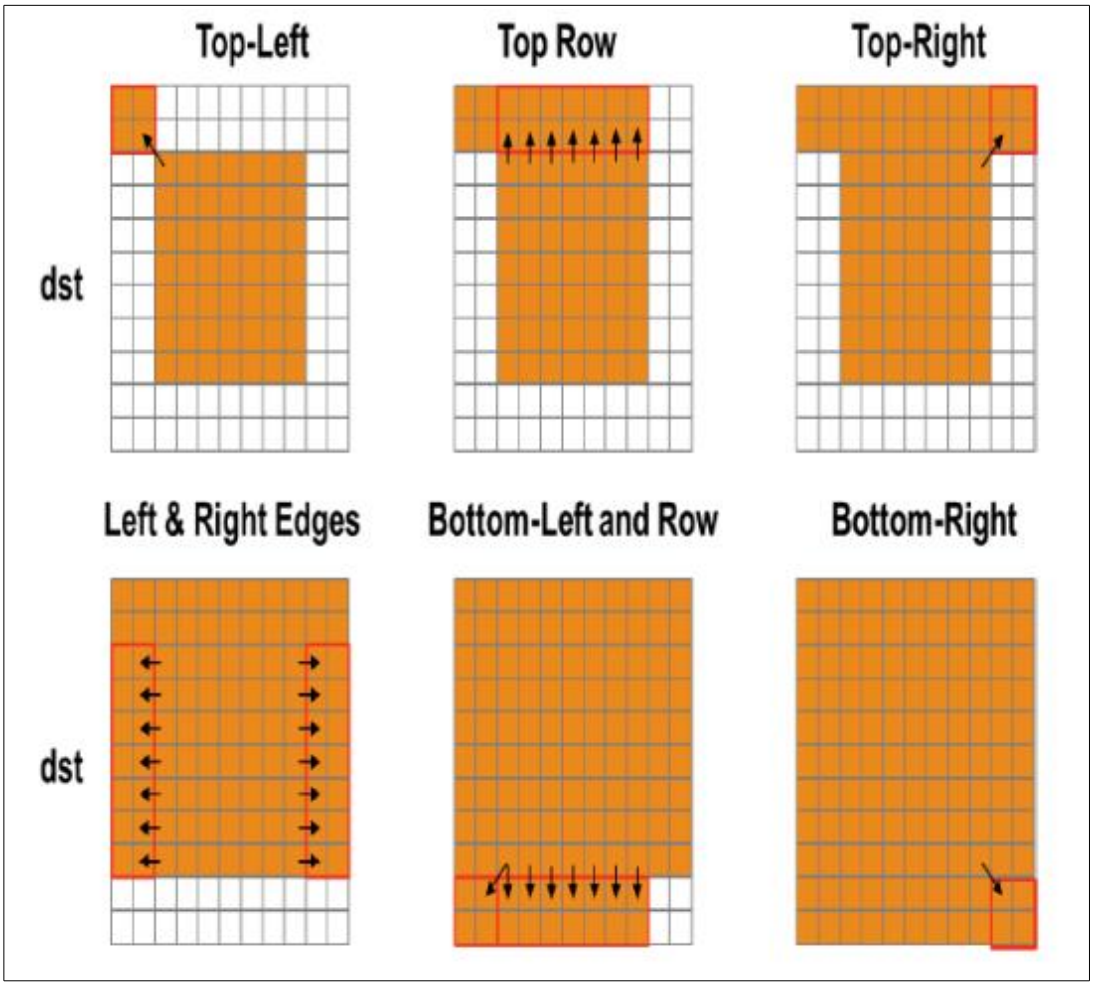


Figure 32: Convolution Border Samples

Source: Xilinx

In the last figure we can notice that the output image is smaller in both horizontal

and vertical directions,due to the convolution border effect. So,we have to fill in the border pixels with data. We can create those values by simply copying the nearest pixel's value in the convolved output.

A simple approach to apply that solution is to handle each case differently:

- Top border

```
int border_width_offset = border_width * width;
int border_height_offset = (height - border_width - 1) * width;
// Border pixels
Top_Border:for(int col = 0; col < border_width; col++){
        int offset = col * width;
        for(int row = 0; row < border_width; row++){
                int pixel = offset + row;
                dst[pixel] = dst[border_width_offset + border_width];
}
for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + row];
}
for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + width - border_width - 1];
        }
}
```

- Side border

```
Side_Border:for(int col = border_width; col < height - border_width; col++){
int offset = col * width;
for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[offset + border_width];
}
for(int row = width - border_width; row < width; row++){
        Int pixel = offset + row;
        dst[pixel] = dst[offset + width - border_width - 1];
        }
}
```

- Bottom border

```
Bottom_Border:for(int col = height - border_width; col < height; col++){
int offset = col * width;
for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + border_width];
}
for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + row];
}
for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + width - border_width - 1];
        }
}
```

Consequently,it is calculated in every pixel the quantity $R=detA-k*(traceA)^2$,where $A=\begin{bmatrix} I_x^2 & I_{xy} \\ I_{xy} & I_y^2 \end{bmatrix}$ .All the values of R are stored in the array called "cornerness[]".

Then,if the value of R in the arbitrary pixel (x_0, y_0) is local maximum in a 3x3 window,then that pixel is marked as a local maximum and thus takes a value of 1 in a binary map that represents the corners with the logical value of 1,and

other not-interesting pixels with 0.

Finally,at pixels that have the value of “1” in the map,it is applied a geometrical fitting in image proportional coordinates. It is determined the quantity,consider it $L(x_0, y_0)$. If $L(x_0, y_0)$ is not zero,then the coordinates of that pixel ,after being corrected by sub-pixel calculations, are stored in the corner array. When that procedure is completed,the corners are drawn in the image,using the coordinates from the corner array.

There is another function that is executed only if there is a large number of corners detected. The operation of that function is that given the corner map(called "cmap") that was computed in a previous step that has marked the candidate corners,lets say n. First,it sorts the corners in descending order depending on its corner's strength,meaning the corresponding value of R for each pixel. Then,it selects N<n corners and distributes them "uniformly" across the image. Then cmap is returned,containing only the selected N corners.

The pseudo-code is presented below:

```
/* Adaptive Non-maximal Suppression using the scheme of MSR-TR-2004-133 */

float vo_anms_schemeA(cmap,conrerness,n,N){

      quicksort(cmap,indexes,n);
      for (i=0;i<n;i++){
            compute_the_k_strongest_cornerness();
            /*find the closest distance to j among the first k strongest corners*/
            min_distance[i]=find_closest_distance_to_j();
             /* select the N largest values in r */
            threshold=quick_Select(min_distance,k);
            for(i=[0,k]){
                  if(r[i]<threshold)erase_from_map();
            }
         }
      return cmap[];
      }
```

That was a general structure of the implementation of Harris corner detector. After,we used the valgrind tool ,we produced a rough profiling of the percentage of the cycles that ever part of the algorithm needs. To do that,we open a terminal and we type:

```
valgrind --tool=callgrind ./harris
```

where *./harris* is the executable of the implementation.[21]

Callgrind is a tool of valgrind that makes the particular estimation. We open the output with *kcachegrind*,another tool of valgrind. The output is shown below:

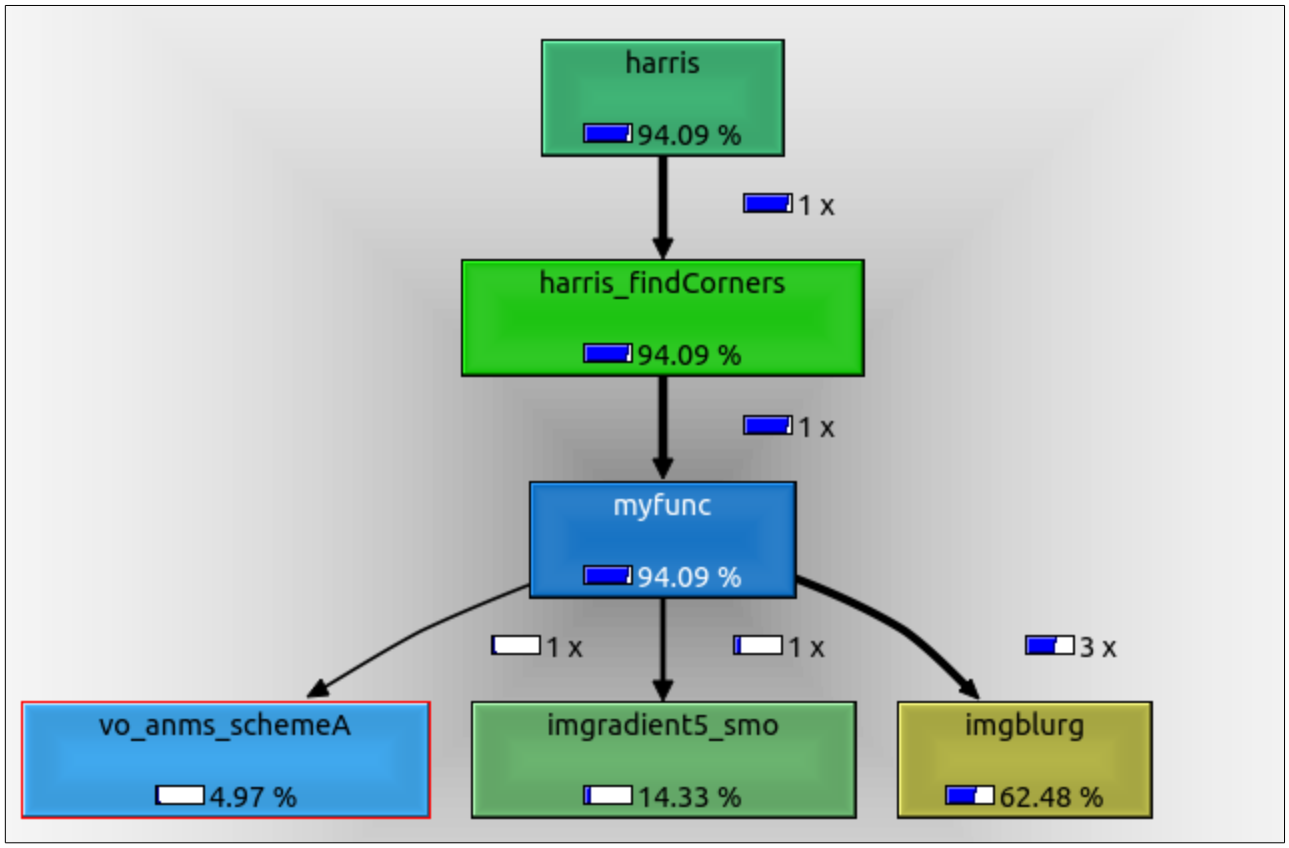


Figure 33: Harris cycle estimation

To be clear,*''my_func()''* is the function that combines the results of the other functions and performs the processing:local maximum calculations,filling the binary map of the candidate corners and finally draw the correct corners on the output image. As we can observe,"my_func" demands about 12% of the total cucles,"vo_anms_schemeA()" about 5%,"imgradient" about 14% and "imgblurg()" about 62%. The reason for that large difference is that the process of bluring is performed three times,for matrices : I_x^2, I_{xy}, I_y^2 ,where I is the intensity of the image. As a result,the previous figure is a guide that can help us to emphasize where to put effort in order to optimize the parts of the implementation that are more computationally intensive.

3.3 Vo_anms() analysis

3.3.1 Non-recursive implementation

In that point we will analyze the functionality of "vo_anms()" function. The main characteristic of that function is that is has recursive operation. It uses the classic quicksort function to sort in a descending the "strength" of each corner. However,since our target device is an FPGA board,we know that recursive functions are not synthesized,unless we are able to define the depth of the recursive tree at compile time. Because we were not able to determine that,it was inevitable that we changed the quicksort to a non-recursive implementation. So,we present below the non-recursive version of quicksort:

```
void quicksort_flint(int  farr[],int arr[])
{
  i=0;
  beginning[0]=0; end[0]=n;
  while (i>=0) {
    Left=beginning[i]; Right=end[i]-1;
    if (L<R) {
      select_pivot_value();
      select_pivot_index();if (i==end)stop();
      while (Left<Right) {

      find_the_less_element_than_pivot_from_the_right();
      move_it_to_the_left();
      find_the_greater_element_than_pivot_from_the_left();
            move_it_to_the_right();
      }
      select_new_pivot();
      beginning[i+1]=Left+1;
      end[i]=Left;
      i++;}
     else {
      i--;
    }
   }
//reverse array-descending sorting
descending_sorting();
return;
}
```

In that version,in every iteration we select new pivot at first and then we start to compare the elements with the pivot from the left and from the right of the array.

The first step is to select a pivot. We follow the most simplest approach and we select as pivot the first element in the array. Starting from the right end of the array,when we find an element that is less than the pivot at position "Right",we move it to the left side-at position "Left". Then we start to look up from the left side for position "Left +1" and when we find an element that is greater than the pivot we move it to the position "Right". Finally we select as new pivot the value at the "Left" position and we start again the previous procedure. When the array is sorted,we have to re-arrange items in order to achieve descending order sorting,instead of the ascending one that was initially applied.

3.3.2 Select the strongest corners

After the sorting,vo_anms finds the k-strongest corners,by calculating the corners that have R-value[see below] that is over a threshold. From those corners,we compute the minimum distance to the j-th corner(j={1,..n}). When having measured all the minimum distances we have to determine which of them are the N strongest(N is variable and is user's choice). Therefore we use the function "quickSelect()",which is presented below in pseudo-code:

```
int quickSelect(int a[n], int n, int k)
{
// a[] contains the minimum distances
// n:size of the array
// k:number of elements to select
 l=0; r=n-1;
  do{

   select_the_last_element_as_pivot();
    for(i=j=l; j<n-1; ++j)
      compare_element_pivot();
      if (element<=pivot)swap(element,pivot);
      next_element();
      if(a[j]<=pivot){
        swap(a[j],pivot);
            i++;// next element
        }
  }while (not_sorted(k));
  return a[i];// the k-th strongest value
```

In each iteration,left of the selected pivot there are values that are less equal than pivot. Depending on the value of k,the do-while loop is repeated until a[i] has the value that is the k-th strongest. That value is returned to vo_anms() and plays the role of threshold. We scan all the previously calculated minimum distances and we only keep those that are above the threshold. Whichever corner is less than threshold is erased. This is done by marking the corner's position on cmap with the value of '0',so it is not listed as corner any more.

Chapter 4: Harris Implementation

4.1 Harris syntesizable version

In this thesis,we evaluate our proposed framework with the High-Level-Synthesis of Harris Computer Vision algorithm. We selected Xilinx Vivado HLS as a state-of-art HLS industrial tool [13].Our final target is to synthesize the C-implementation of Harris algorithm,in order to be able to be mapped to the FPGA board. To do so,we used the Vivado® Design Suite,and especially the Vivado High Level Synthesis (HLS) tool. Our target board is the Xilinx kintex7- xc7k325tffg900-2.

Moreover,the final target of this thesis is to optimize the implementation, by

- Accelerating the algorithm's performance
- Achieving optimal area utilization

The first challenge was to remove any dynamical memory allocations from the code. Since our target is an FPGA board,it is apparent that HLS compiler must know exactly how much memory it is required at compile time and not at linking time. Thus,any malloc() functions where eliminated and were replaced by static memory allocations,meaning arrays. The result was that there was a lot of memory acquired by the start of the implementation and so there was mandatory to start making memory adjustments.

4.1.1 Memory optimizations

One of the first obstacles that every engineer has to deal with when it comes to program an FPGA device is the constraint of memory. Unlike other general purpose systems,where the amount of memory is more than sufficient,in reconfigurable machines it demands a lot of adaptations to be made in algorithms 's implementations, sometimes with inevitable loss in quality of the produced application.

As a first step,a simple improvement is to adjust the word-length,depending on the application' s needs. Although in general purpose computing word-length is already defined by the architecture of each processor,reconfigurable computing allows the customization of every single variable 's word-length,in order to accomplish optimal trade-offs in numerical accuracy,speed and power consumption.

So,the designer is able to achieve the most sufficient hardware implementation of the algorithm at which different word-lengths are used for different internal variables,depending on their size. Thus,some word-lengths are reduced,without decreasing the level of accuracy. In fact,sometimes it is observed that accuracy is less sensitive in some variables than to others. So it is possible to cut down some extra bits in order to shrink the area of the hardware used, without losing any sufficient information.

Hence,selecting the optimal word-length for each variable can be very hard problem. Actually,as demonstrated in it is NP-hard,even for systems that have special mathematical properties that simplify the problem. Nevertheless,there have been published several approaches to that problem:some of them can considered as heuristics,providing an area/signal quality trade-off. Others,offer simplifying assumptions on error properties and others present optimal approaches that can be applied to algorithms that have special mathematical properties.[22]

Our approach to that problem was based on the special mathematical properties of the algorithm. Particularly, our goal is to reduce the area utilized by the arrays of the implementation. Those arrays are:

- $I_x[], I_y[], I_{xy}[], I_x^2[], I_y^2[]$,which all contain integer values
- $Cornerness[]$,which contains all the R-values(see chapter 3) with float arithmetic.

The sample image we used as input to our implementation is 256x256 ,gray-scale and shown below:

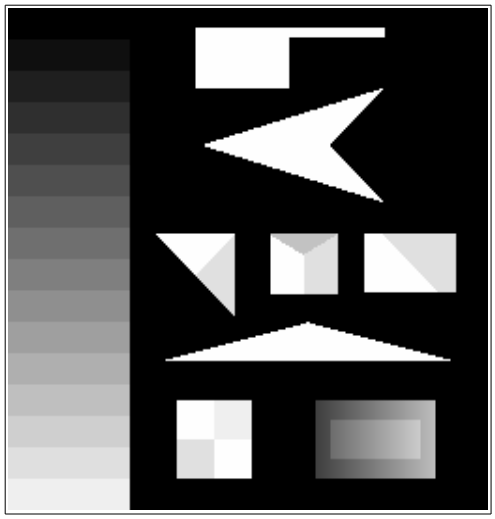


Figure 34: Sample input image

Analyzing the algorithm,we found out that each element of the array "Cornerness[]" is compared with its neighbors in a 3x3 window. Thus, what we need is to reduce the word-length,but try to maintain the size relationship between the elements. What was found out is that making a cast from float to integer arithmetic did not have negative effect on the output's accuracy. By the term accuracy here we mean counting the variation of the coordinates of the corners detected,comparing to the initial ones.

With all of our arrays including values of 32bit,a first try of synthesizing the project gave as the results below:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	-	111	11458	24269
Memory	964	-	64	38
Multiplexer	-	-	-	38
Register	-	-	12	-
Total	964	111	11534	24350
Available	890	840	407600	203800
Utilization (%)	108	13	2	11

Figure 35: Harris initial report

The table above is the Synthesis report created by the Vivado High Level

Synthesis (HLS).As it can be observed,the initial version of the algorithm actually needs more memory than it is available in the FPGA board,making it impossible to be mapped.

So,we analyzed more the elements that the arrays of partial gradients contain,finding out that in most times the length of 32 bits was a redundant luxury. Then,our decision was to reduce to half the word-length(16bits) of the following arrays: $I_x[], I_y[], I_{xy}[], I_x^2[], I_y^2[]$ and examine the accuracy of the output. In that case,we did have some some variation but its was inside of the permitted error margin. In fact,using the Euclidean norm, we compute the error margin as follows:

$$error\ margin = \frac{\sum \|z\|}{corners\ detected} \quad , \quad \|z\| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad \forall\, i = 1,\ldots n \quad , j = 1,\ldots m$$

where n is the initial detected corners and m is the detected corners of the optimized version. The error margin was calculated to be 0.5% . The gain of the memory utilization is shown below:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	-	71	10794	23539
Memory	564	-	32	19
Multiplexer	-	-	-	22
Register	-	-	12	-
Total	564	71	10838	23585
Available	890	840	407600	203800
Utilization (%)	63	8	2	11

Figure 36: 16 bit optimization

As it is figured,the reduction of BRAM_18K utilization was cut off by about 45% and the DSP48E by 5%,making possible to map the design onto the actual Kintex7 board.

Consequently,another try was to decline the word-length of the "Cornerness[]" array 's elements to half,because for the detailed synthesis report it was depicted that a lot of memory banks were occupied in the 32bit version. However,it was detected a large declination of the corners detected,and the actual coordinates of each corner. However,with detail analysis of the elements of that

array,we found out that we could represent efficiently its contents with less bits,but definitely more than 16 bits.

In that case,C-language can't provide any help to make a custom word-length. C-based predefined data types have word-length multiple of 8 (8,16,32,64 bits) .So,since RTL description can support any arbitrary data length, it is possible that needless hardware would be occupied. For example,in the case that we perform a multiplication,the standard unit to do that in a Xilinx FPGA is the DSP48. This contains a 18*18 bit multiplier. Thus,if,for example,a 17-bit multiplication is needed,then using C data types you have to implement a multiplier which is 32*32 bit occupying 3 DSP48 macros,when just one is required,resulting in unnecessary overuse of FPGA's hardware. [23]

In that case,we can take advantage of Vivado HLS arbitrary precision data types. Vivado contains libraries that provide data types which can define variables of any custom width. For example, user can define variables of 10 bit,24 bit or 35 bit, while using the standard C data types those variables would be 16,32,64 bit respectively. In the figure below it is shown the summary of the supported arbitrary precision data types of the Vivado HLS:

Language	Integer Data Type	Required Header
C	[u]int<W> (1024 bits)	#include "ap_cint.h"
C++	ap_[u]int<W> (1024 bits) Can be extended to 32K bits wide.	#include "ap_int.h"
C++	ap_[u]fixed<W,I,Q,O,N>	#include "ap_fixed.h"
System C	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits)	#include "systemc.h"
System C	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h"

Figure 37: Arbitrary Precision Data Types

Source: Xilinx

The great advantage of the Vivado 's data types is that you do not have to sacrifice accuracy,as a trade-off in hardware area. The engineer has the opportunity to adjust variables to smaller bit-widths and then re-execute C simulation to confirm that the functionality is still correct or inside the allowed error margin. In

the previous example,the 35-bit hardware would have been implemented without losing any accuracy, and saving the remaining 29-bit for other needs of the design. Hence,shorter bit-lengths can provide smaller and faster hardware circuits. We can place more logic in the FPGA and the implementation can be executed at higher clock frequencies.

In our implementation,we performed specific analysis of the necessary word-length of the array's elements. The arrays where divided into 2 categories:

- gradient content arrays
- Cornerness content arrays

For the first category,as said before,16-bit word-length brought no problem to accuracy. Trying to reduce it even more,we found out bottleneck at 15-bits(further reduction had unacceptable results). In the second category,after some trial-and-error techniques,we concluded to data width of 24 bits. The results of Synthesis procedure are shown below:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	-	69	10650	23306
Memory	499	-	30	18
Multiplexer	-	-	-	21
Register	-	-	12	-
Total	499	69	10692	23350
Available	890	840	407600	203800
Utilization (%)	56	8	2	11

Figure 38: Arbitrary precision types optimization

As it is depicted,from the previous 16-32 bit version,our implementation uses 65 less BRAMs,that is 8% reduction. Notice that the error remained the same. Furthermore,any bit decrease would bring inaccuracies in the output that are beyond the acceptable error margin,which is not desirable. So, it is necessary to

carry out other optimizations to decline more the area of the FPGA that is required by the Harris implementation.

4.1.2 Vo_anms() synthesis

Vo_anms() function is a bit of expensive computationally,since it performs a variety of complex operations. We can observe below that vo_anms() has increased the percentage of resources utilization in all categories:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	26	75	13825	31329
Memory	499	-	30	18
Multiplexer	-	-	-	21
Register	-	-	12	-
Total	525	75	13867	31373
Available	890	840	407600	203800
Utilization (%)	58	8	3	15

Figure 39: Harris utilization with Vo_anms()

We can observe that there is an offset added in BRAM_18K utilization,that is 26 block rams,initially. Then,we perform some changes concerning data reuse in vo_anms() and we have a new synthesis report:

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	9	75	13912	31815
Memory	499	-	30	18
Multiplexer	-	-	-	21
Register	-	-	12	-
Total	508	75	13954	31859
Available	890	840	407600	203800
Utilization (%)	57	8	3	15

Figure 40: Harris Vo_anms() optimized

We can observe that the BRAM utilization has reduce from 26 to 9 %. The other resources have not been changed,since the operations that vo_anms() implements remain the same.

4.2 Parametric Fragmentation of input image

To make our design more memory efficient,we examined the structure of the algorithm. We started form the idea of observing the core of the algorithm,which is the part that computes the gradients of the image's intensity,then performs the horizontal and vertical convolution and eventually marks the corners on the image wherever there is a local maximum in a 3x3 window that passes the whole image. We concluded that if we wanted cut of the size of the arrays,we should execute the core of the algorithm for only a fraction of the initial input image. Thus,the core would be performed multiple times. We decided,then,to divide the image in parts that are power of 2 in order to have easier calculations. The factor of fragmentation was 2,4,8,16. We present graphically the way we did the fragmentation:

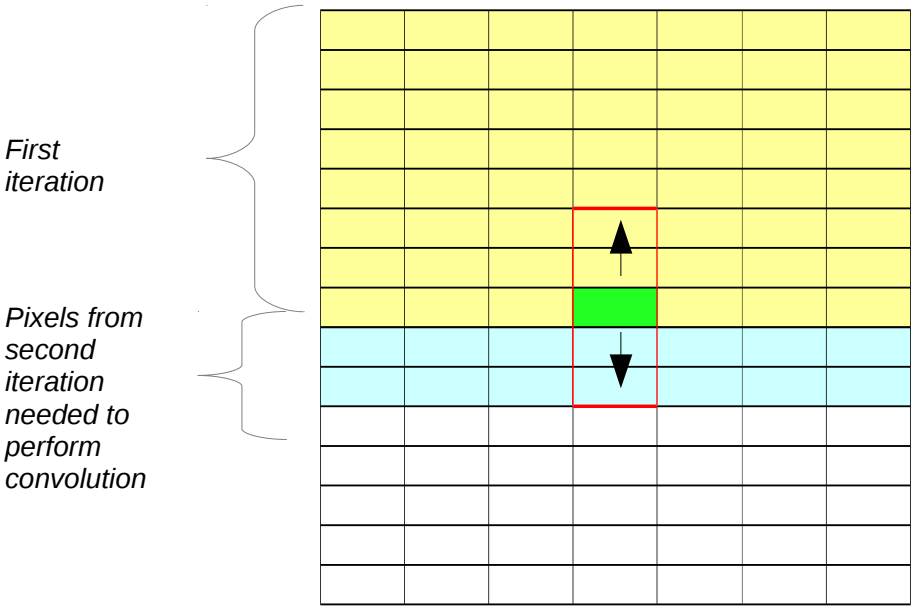


Figure 41: First iteration

At the first iteration,the convolution window(red rectangle) moves throughout the the yellow region. The double for-loop would be like following:

```
for(i=0;i<height/factor+2; ++i{
      for(j=0; j<width; ++j){
            perform_calculations();
            }
      }
```

We notice that the border for the external loop is not height/factor,where factor={1,2,4,8,16}. Since the convolution window in vertical direction is 5,in the border the center pixel(the green one) needs values from the next iteration. So,we need to include 2 more lines in the first iteration to keep the functionality of the algorithm correct. This is why the border of the external loop has been extended to height/size+2. In the second iteration,the pattern is a little bit different,as we can

see below. In that case,same as before we need to have the values of the previous's iteration region,so we do not start from the $i=\frac{height}{factor_{i-1}}$,but from $i=\frac{height}{factor_{i-1}}-2$

The loop now is changed :

```
for(i=height/ factor_{i-1} ;i<height/factor+2; ++i{
      for(j=0; j<width; ++j){
            perform_calculations();
            }
      }
```

We still keep the upper bound of i ,and that is the patter we apply for the general case of the i-iteration. In the last iteration,we follow the approach of the first iteration with the difference that we only need the 2 previous row's values.

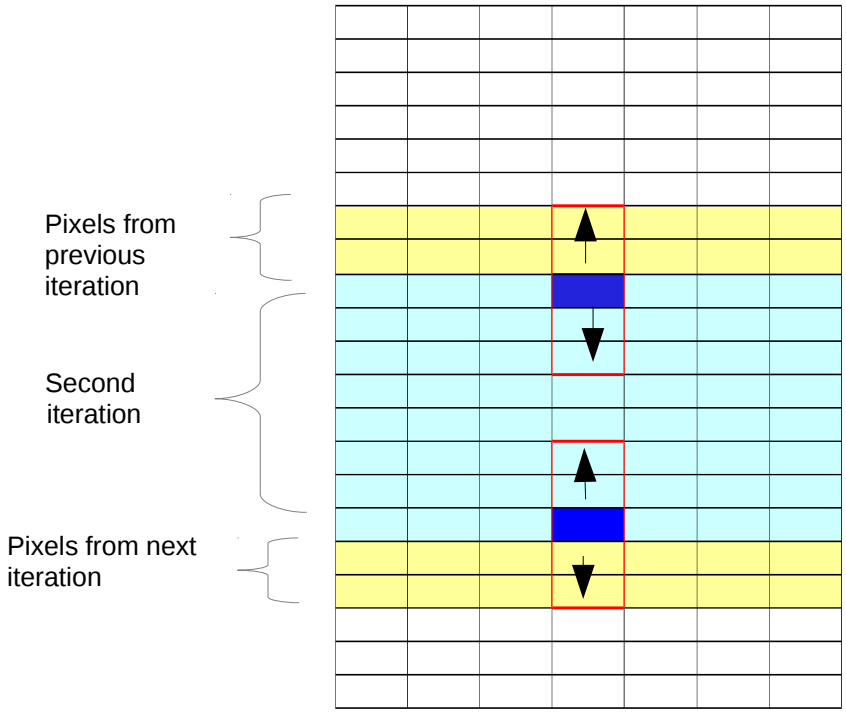


Figure 42: Second iteration

That approach is computationally expensive,since we actually use twice the same

values. However,this aliasing is necessary since otherwise the convolution would not be correct and the candidate corner pixels at borders would not be detected.

The error margin was calculated with two ways:

- one that counts the difference between the detected corners of each version and the correct corners
- the other that counts the percentage of the corners that their coordinates differ from the correct ones inside the desired error margin

Test Case 1

Initially,the correct output of the algorithm with input the previous gray-scale image is presented below:

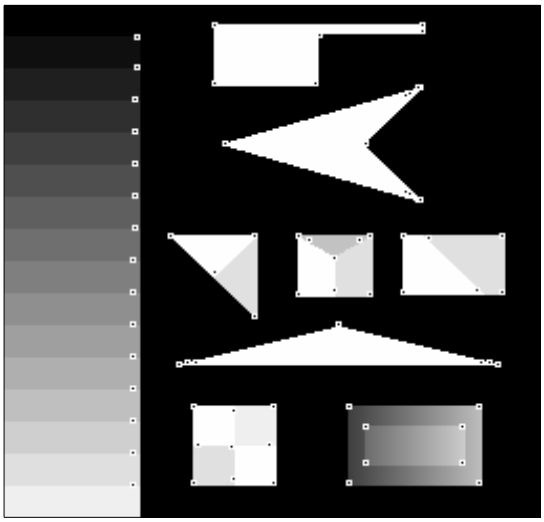


Figure 43: Correct output

Below are shown the two kinds of error:

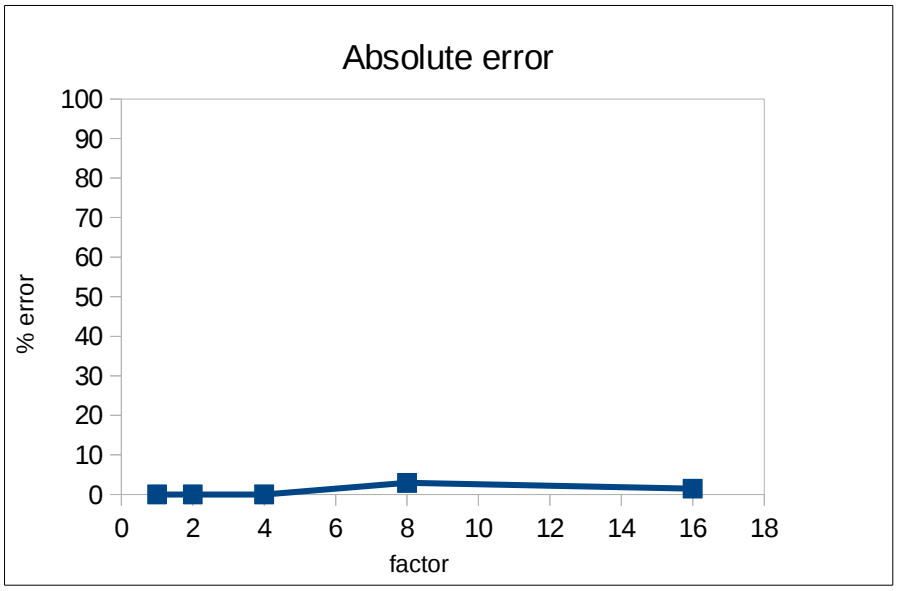


Figure 44: Absolute error

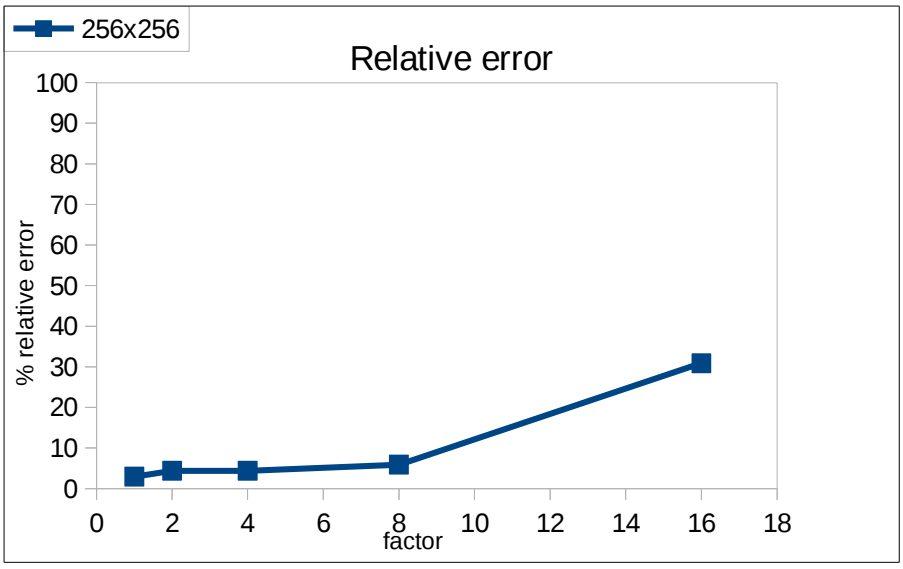


Figure 45: Relative error

In each version we counted the resources that were required for the implementation and are depicted in the following charts:

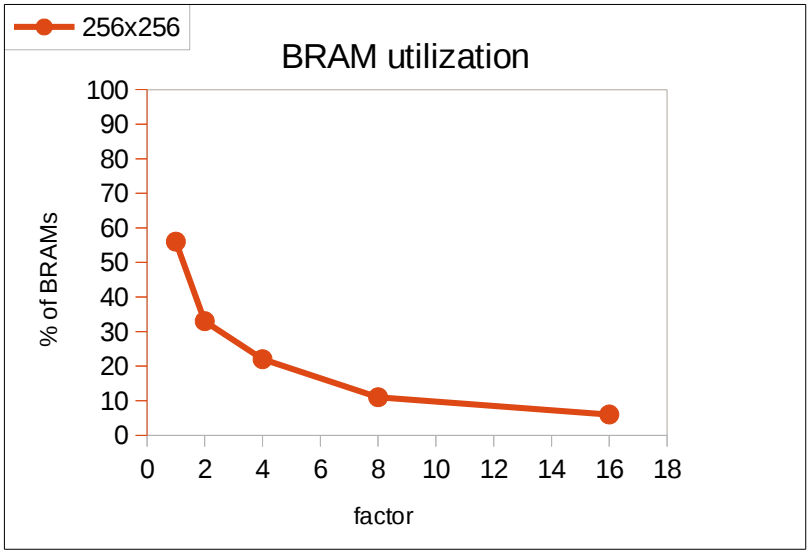


Figure 46: BRAM utilization

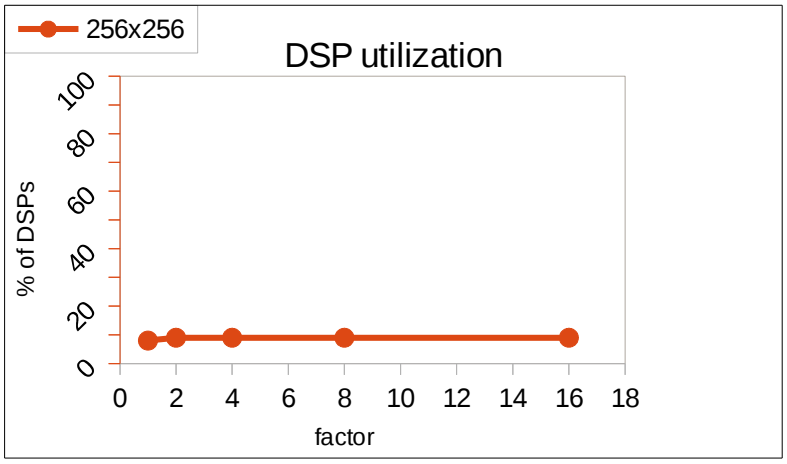


Figure 47: DSP utilization

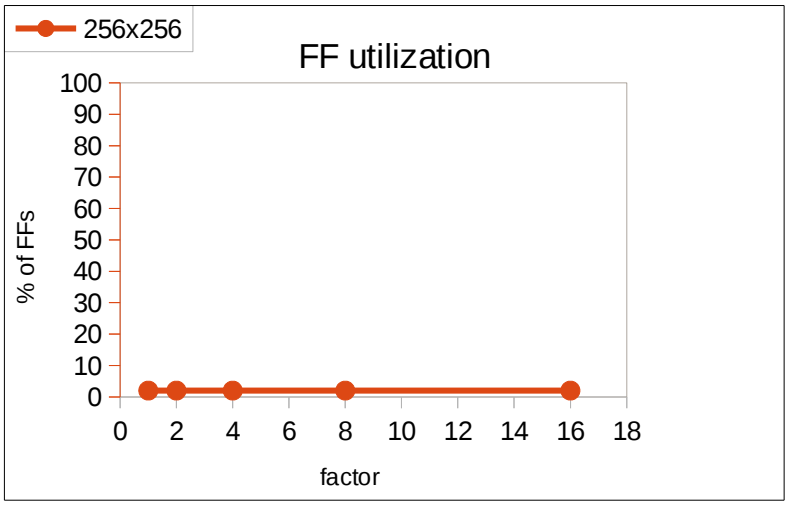


Figure 48: FF utilization

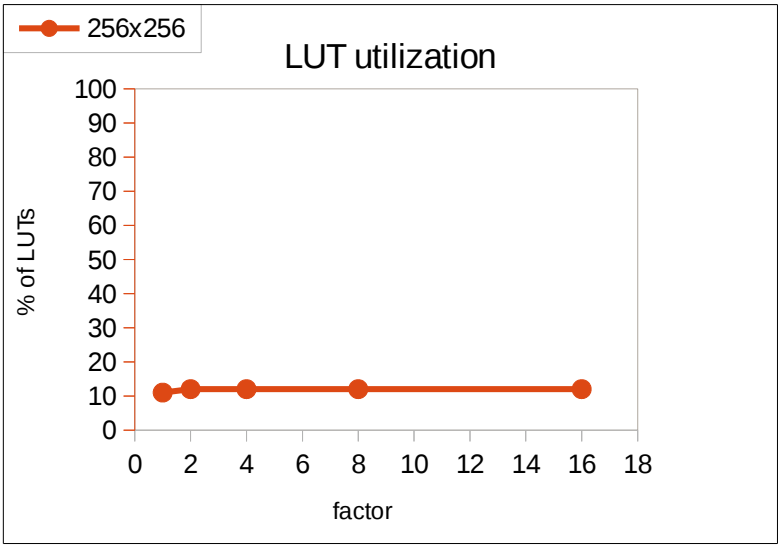


Figure 49: LUT utilization

As expected,since the size of the arrays was declined by 2,4,8,16 each time,likewise the percentage of BRAMs was decreased exponentially. The other parts of the resources(DPSs,FFs and LUTs) remained steady,as there was not any important change in the logic that was implemented. In the following table we present accumulated utilization of FPGA resources for Test Case 1:

Fpga resources	BRAM_18K	BRAM_18K %	DSP48E	DSP48E %
Factor				
1	499	56	69	8
2	301	33	76	9
4	202	22	76	9
8	103	11	76	9
16	56	6	76	9
Available	890	-----	840	-----

Fpga resources	FF	FF %	LUT	LUT %
Factor				
1	10676	2	23351	11
2	11399	2	24652	12
4	11324	2	24584	12
8	11420	2	24565	12
16	11431	2	24537	12
Available	407600	-----	203800	-----

In that case we could refer to another application that was made from scratch by Microprocessors and Digital Systems Lab [to be sumbitted] implemented Harris Algorithm without using High-Level Synthesis. However,this version was completely hand-made and the code was written straightaway to VHDL. The implementation was mapped onto Xilinx Virtex-6 board(XC6VLX240T-2) with frequency at 172MHz and for input image size 512x384. The results are shown in the following table:

LUTs	DSPs	Slices	BRAMs 36K	Time
11477	10	4045	82	14 *

The star in the time cell denotes that the time is to be optimized.

Test Case 2

Then we tried to examine the algorithm's performance with an image of the same size,but with more corners to detect. We simply added several objects in the previous image,which changes as follows:

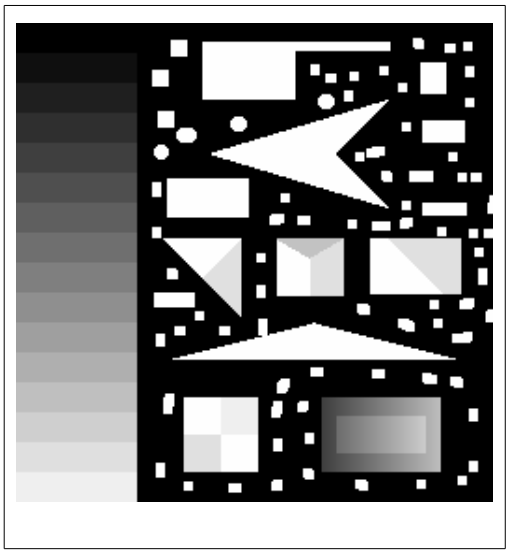


Figure 50: Sample image 2

The algorithm reacted adequately and found all the corners of the image:

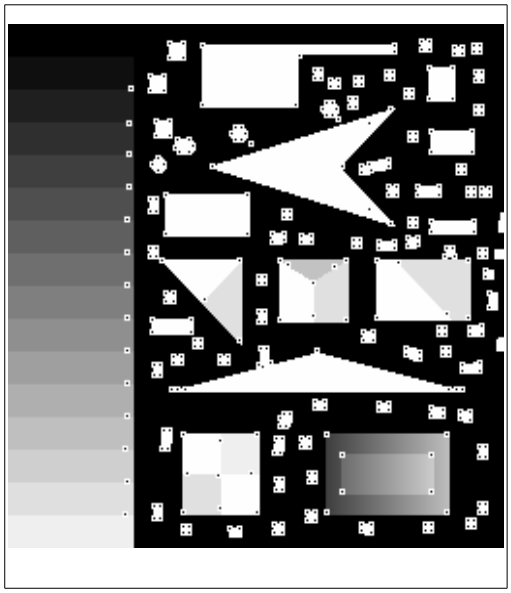


Figure 51: Sample image 2 output

In that case,we do not need to calculate the resources again,since the size of the image is the same. But,we do have to measure the error ,depending on the level of fragmentation of the input image. Below,we present the absolute and relative error:

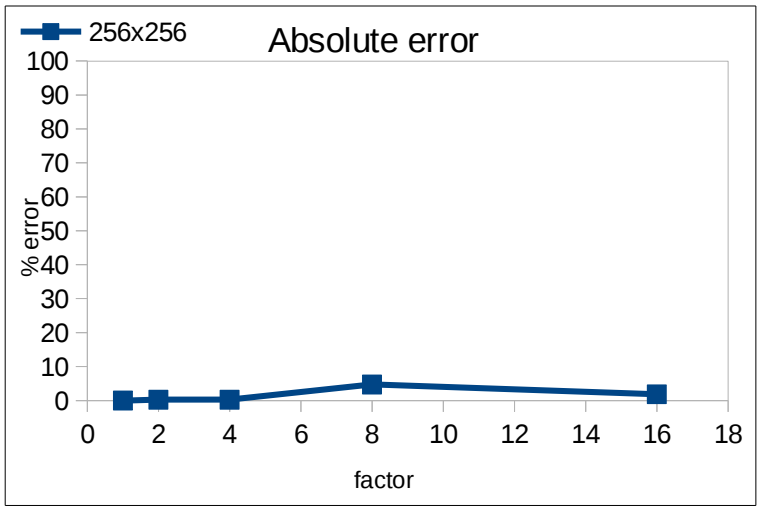


Figure 52: Absolute error

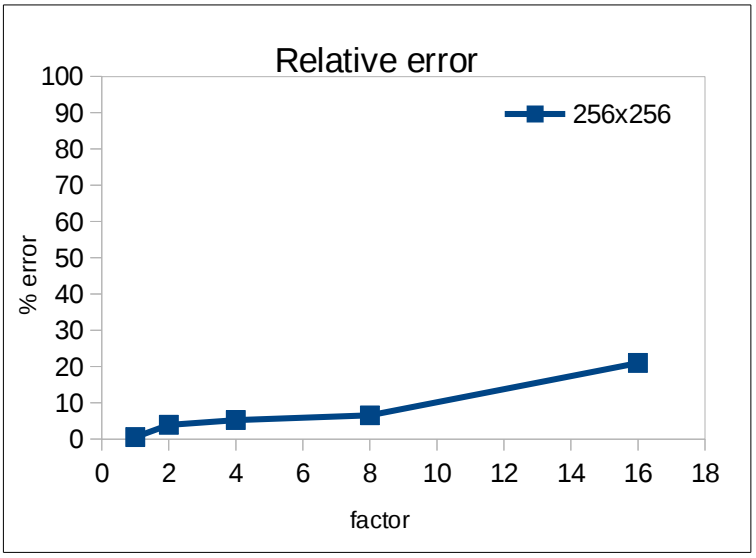


Figure 53: Relative error

Test Case 3

In that test case,we put as input a significantly larger image with 4-times bigger width and height,resulting in a 16-times the size of the initial image.

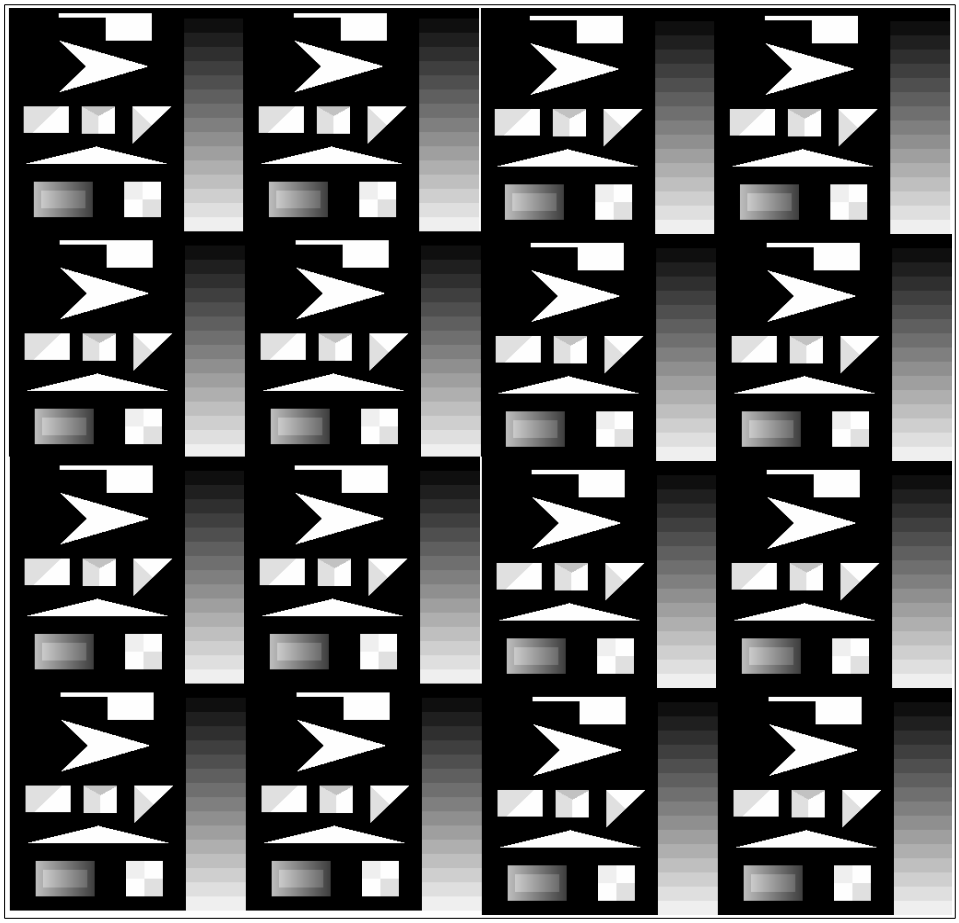


Figure 54:Input sample image 1024x1024

That image is actually 16-times the initial sample image and thus is of size 1024x1024. We experimented our algorithm's performance and as it was obvious the memory problem in that case was huge. The percentage of BRAMs occupied depending on the level of fragmentation is figured below:

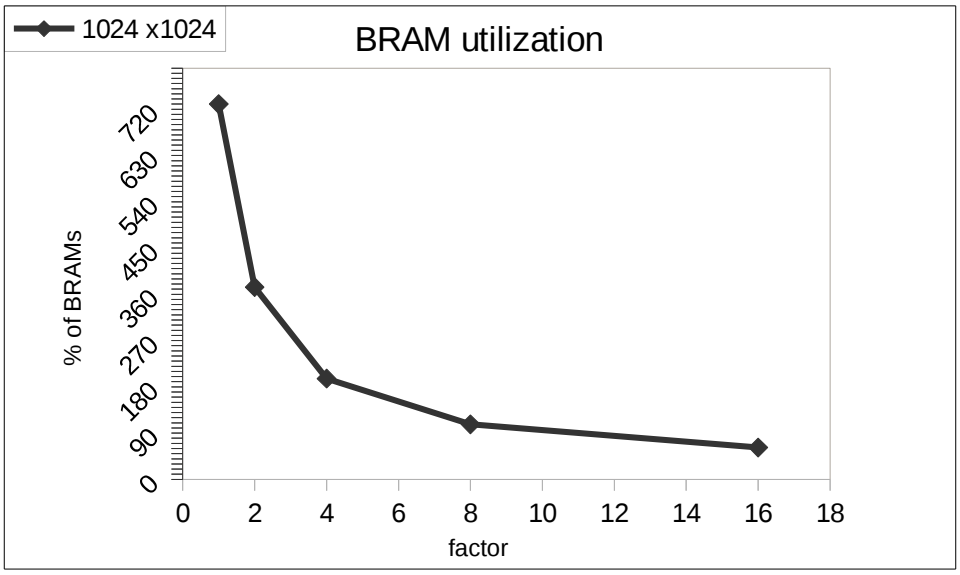


Figure 55: BRAM utilization

As expected, only with the value factor=16 the algorithm could actually be mapped onto FPGA,because only in that case the percentage of BRAM utilization is below 100%(actually is about 60%). In all other cases,the memory needs exceed beyond the available hardware. The other resources have very similar values,as the amount of logic used in the processing of the arrays did not change. In the following table we present the total resource utilization in Case 3:

Fpga resources	BRAM_18K	BRAM_18K %	DSP48E	DSP48E %
1	6499	730	69	8
2	3331	374	76	9
4	1747	196	76	9
8	955	107	76	9
16	559	62	74	8

Fpga resources	FF	FF %	LUT	LUT %
1	10823	2	23607	11
2	11511	2	24914	12
4	11496	2	24847	12
8	11537	2	24835	12
16	11411	2	24767	12

We also counted the absolute and the relative error of the sample 2 image output:

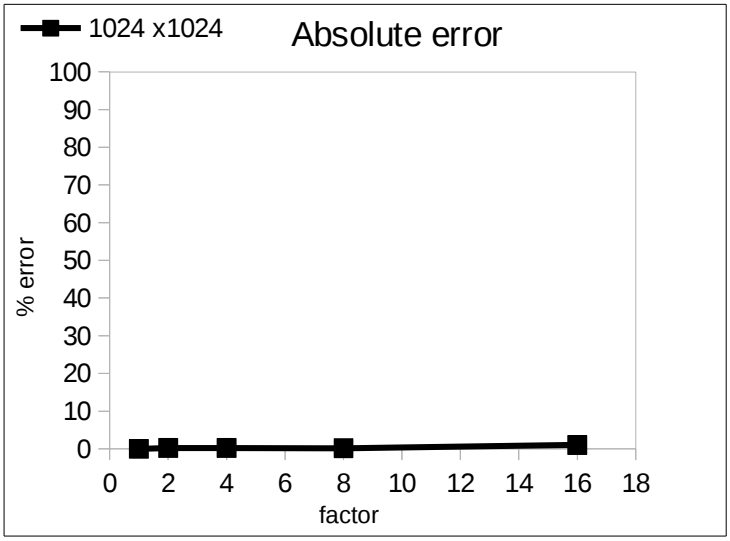


Figure 56: Absolute error

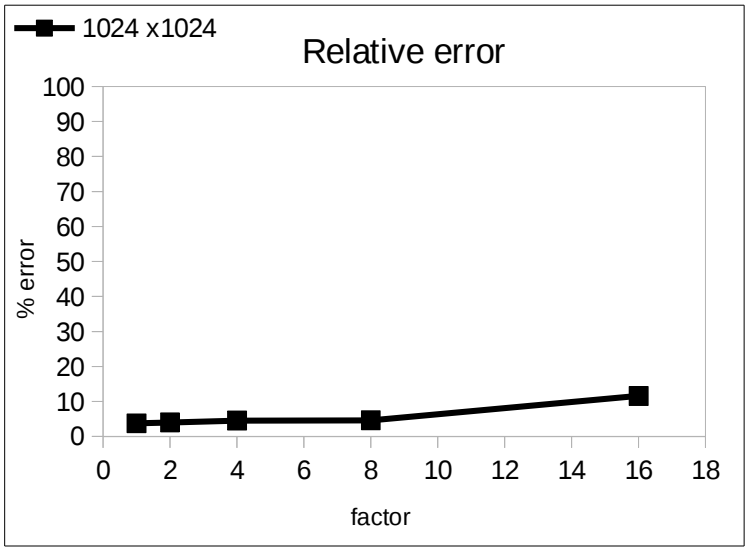


Figure 57: Relative error

4.3 Synthesis Optimizations

In this chapter we are going to discuss the various optimization methodologies that can be performed using the features of Vivado High-Level Synthesis tool,in order to produce a circuit's architecture that meets the design's desired performance specifications,satisfying the area constraints. In general,Vivado HLS has an automatic procedure to handle each design. By default,it tries to create the most optimal implementation,according to the design 's requirements. The clock is the first constraint to be determined and Vivado HLS uses the specification of the target device to decide which is the maximum number of operations that can be executed,within a clock cycle. After achieving the optimum clock frequency,Vivado HLS produces the synthesized circuit and makes optimizations automatically according to the following goals:

- Throughput
- Latency
- Area

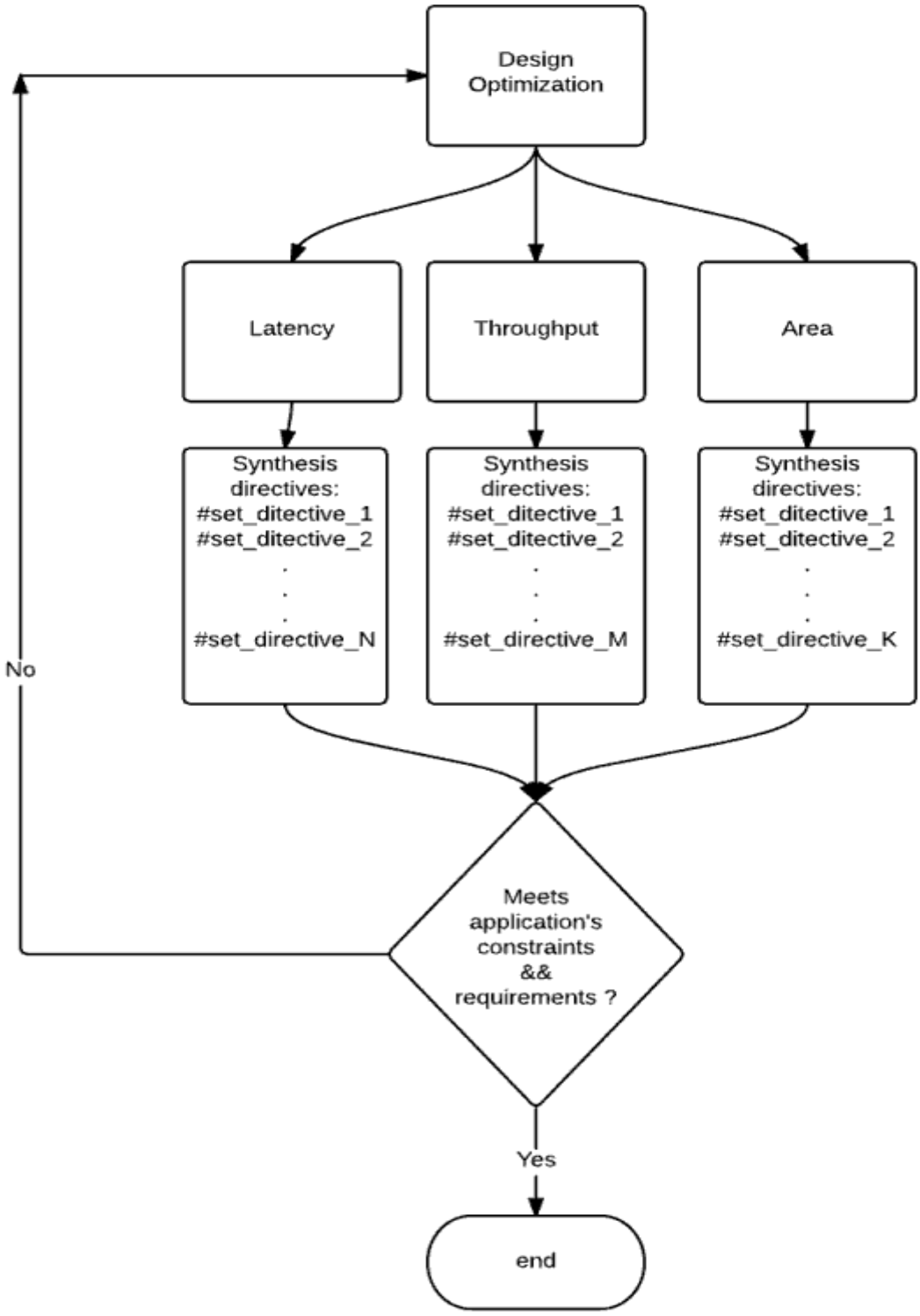


Figure 58: Design optimization strategy

In addition to the default synthesis operations,Vivado HLS provides a number of synthesis directives and configurations which can make optimizations in each of the previous three sectors,depending on the application requirements. Next,we present a flowchart that shows the general optimization strategy.[6]

4.3.1 Throughput Optimizations

The first step to maximize the application's throughput is to minimize the interval between new inputs and thus reach a peak on the output rate. In order to achieve that,a number of optimization are available through Vivado HLS[6]. Next,we present a flow chart of throughput optimization design flow:

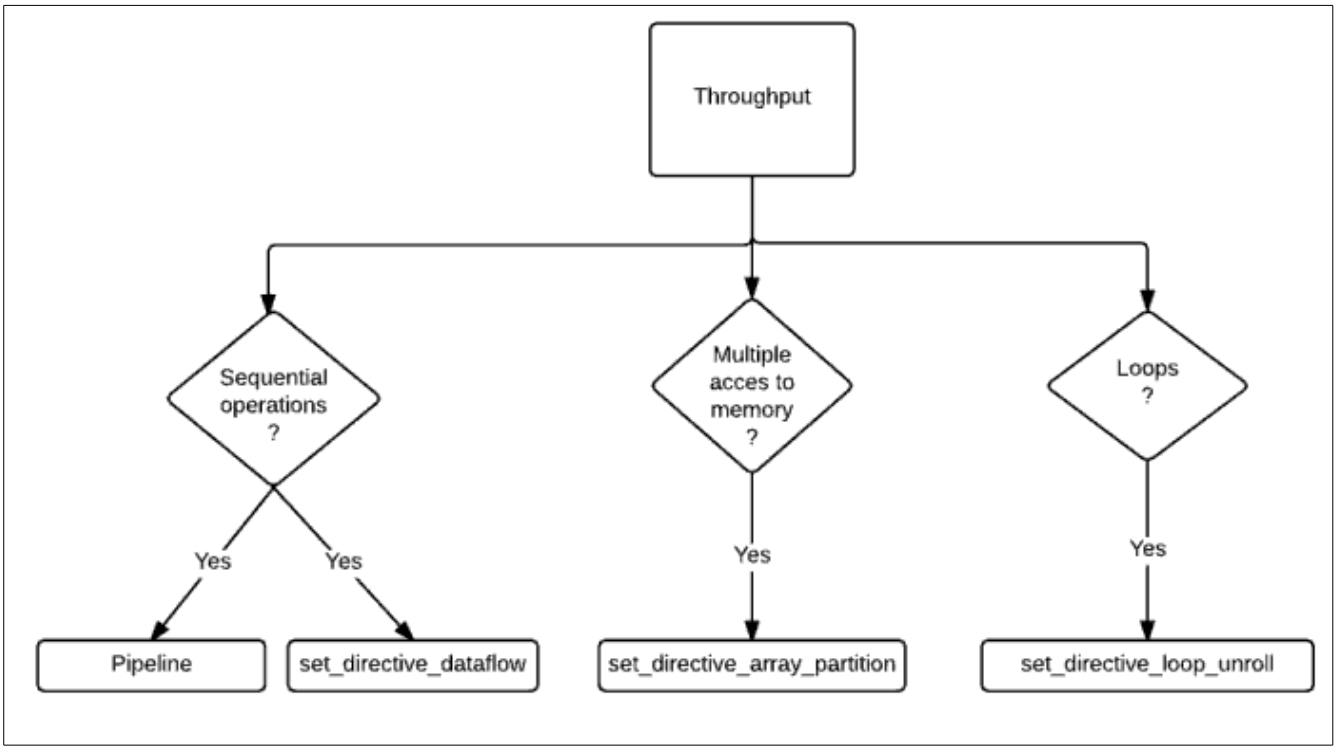


Figure 59: Design strategy-Throughput

4.3.1.1 Pipeline

Pipelining can give the opportunity to operations to be executed in parallel:each task does not have to complete all of its operations before it begins the next set of assignment. It can be applied to either functions or loops. Then,we present a

simple example of how pipeline can improve a function's throughput.

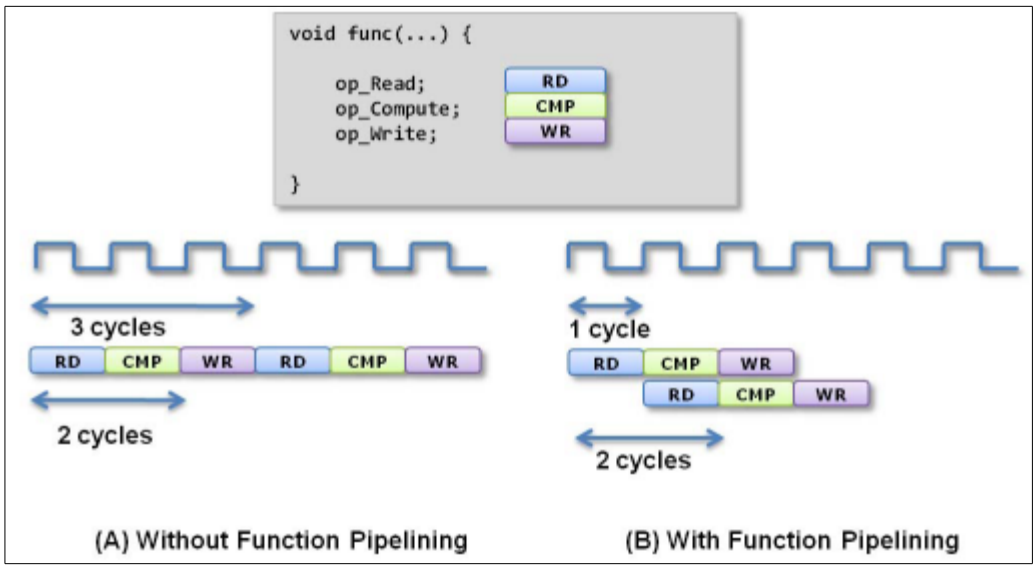


Figure 60: Pipeline behavior

Source: Xilinx

In the non-pipelined version,the function reads every 3 cycles and produces the output value in every 2 cycles. So,the function has an Initiation Interval (II) of 3,and a latency of 2. In the pipelined version,our function reads every 1 cycle (II=1) and still has latency=2. [6]

Next,we can see the changes from a standard sequential loop to a pipelined one with concurrent execution:

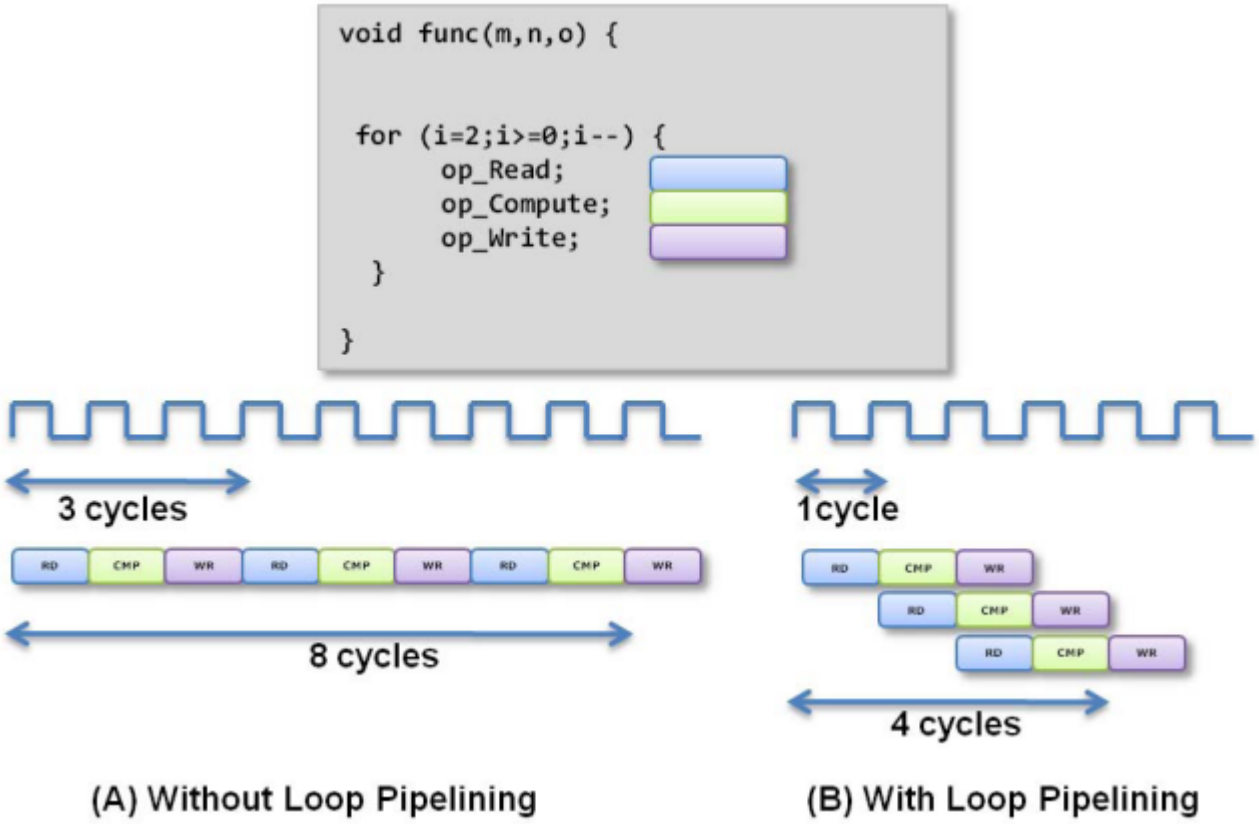


Figure 61: Loop pipeline

Source: Xilinx

In figure (A),every 3 cycles we have new input and every 8 cycles we have new output value. In the pipelined version,our program reads new value in every 1 clock cycle and the number of cycles needed to produce new output has been cut off to 4 cycles. Hence,there is significant reduction in both II and latency,without using more hardware recourses.

To apply PIPELINE synthesis directive,we have to place the following pragma in the C source code inside within the boundaries of the required location:

```
#pragma HLS pipeline II=<int>
```

where II is the desired initiation interval.

4.3.1.2 Dataflow

Dataflow optimization technique can be applied when sequential code is executed. The target is to change the sequential order of functions or loops and make it concurrent. It is one of the most powerful methods to improve the circuit's throughput. In the next figure it is depicted how the Dataflow technique allows sequential execution of three functions to overlap and thus increase the overall throughput and cut down latency.[6]

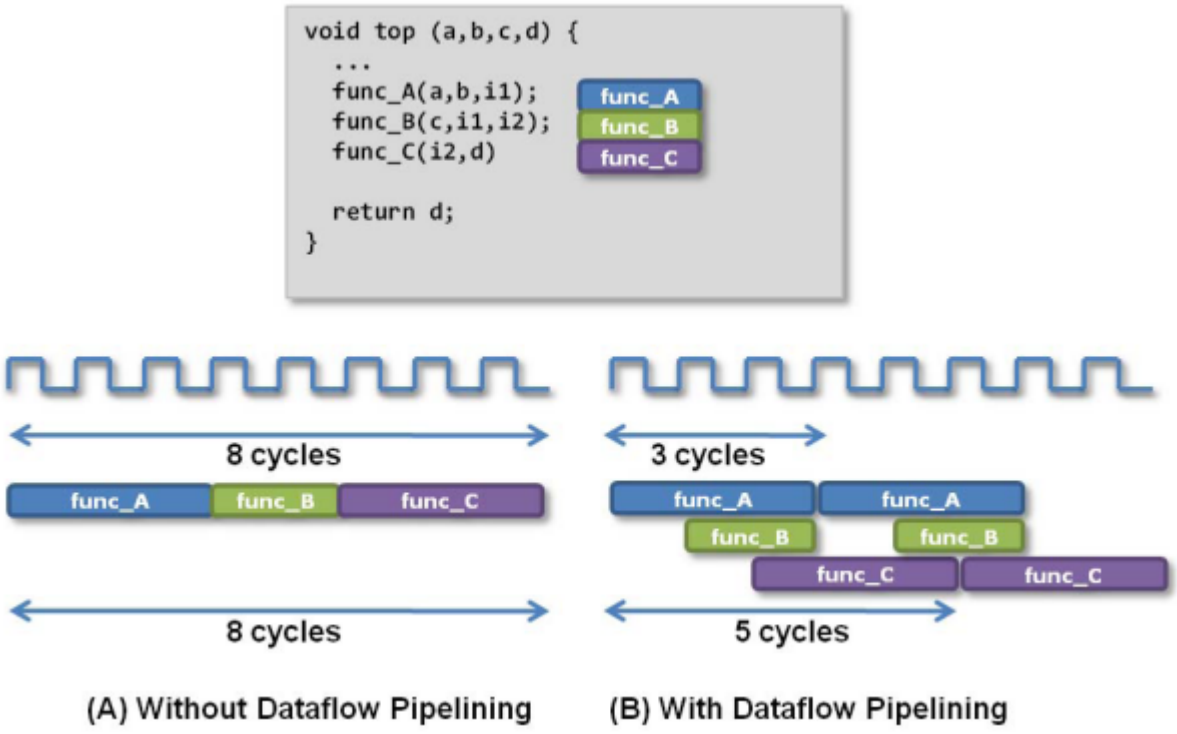


Figure 62: Dataflow behavior

Source: Xilinx

In figure (A) where there is no dataflow applied,our implementation takes 8 cycles to produce an output and the same time for func_A to read a new input. However,in figure (B),dataflow pipelining allows func_A to read a new input every 3 cycles(lower II) and the overall implementation takes now 5 cycles to produce its output(shorter latency).

To perform dataflow directive,we simple add the following pragma into the C source code,inside the region we wish to apply dataflow behavior:

```
#pragma HLS dataflow
```

4.3.1.3 Array partition

A common issue when applying pipeline synthesis directive is that Vivado HLS creates a warning that it cannot reach the desired initiation interval (II) of 1,since it cannot assign a load or a write operation onto a memory because of memory ports limitation. That problem is usually created by arrays. We know that they are implemented by Vivado HLS as block RAMs which have a maximum of two data ports. Thus,it is consequent that the throughput of a read/write(load/store) is bordered.

A possible solution to that limitation is to split the array into multiple smaller arrays(a single block RAM into multiple smaller ones respectively),creating more number of available ports. To perform that partition,we can use the equivalent synthesis directive called ARRAY_PARTITION. Vivado HLS gives as the opportunity to make three types of array partition:

- **block**:the initial array is split into blocks of equal size,containing consecutive elements of the initial block
- **cyclic**: the initial array is split into equally sized blocks interpolating the elements of the original array
- **complete**: the default choice is to split the array into its individual elements. This is actually degenerate a memory into registers.

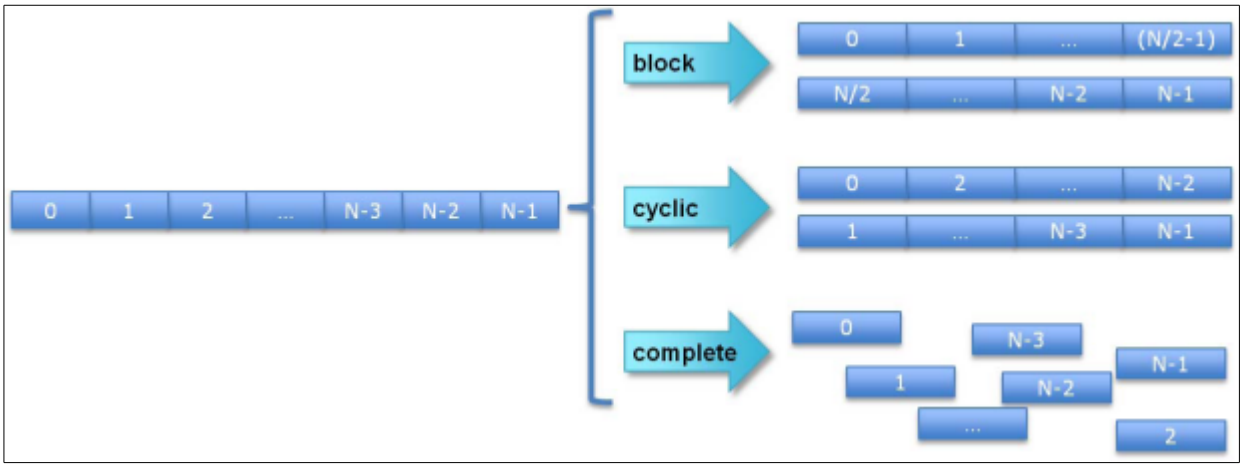


Figure 63: Array partition

Source: Xilinx

To apply array partition directive,we add the following command in the source code:

#pragma HLS array_partition variable=<variable> <block,cyclic,complete> factor=<int> dim=<int>

where variable is the desired array to be partitioned,factor is the desired level of partition and dim is the desired dimension to apply the partition.

We decided to apply partition to the "cornerness[]" array(the one which contains the R-values) because it is the array with the largest word-length. However,the results did not have any positive effect on our implementation. Actually, the timing was a little worse and the area utilization increased(number of BRAMs,LUTs and DPSs),as we present below:

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	5
FIFO	-	-	-	-
Instance	-	166	20061	36641
Memory	571	-	30	18
Multiplexer	-	-	-	21
Register	-	-	12	-
Total	571	166	20103	36685
Available	890	840	407600	203800
Utilization (%)	64	19	4	18

Figure 64: Area utilization of array_partition

The numbers correspond to the first version without word-length optimizations. The initial numbers without array_partition where 63,8,2,11 % respectively.

4.3.1.4 Loop unrolling

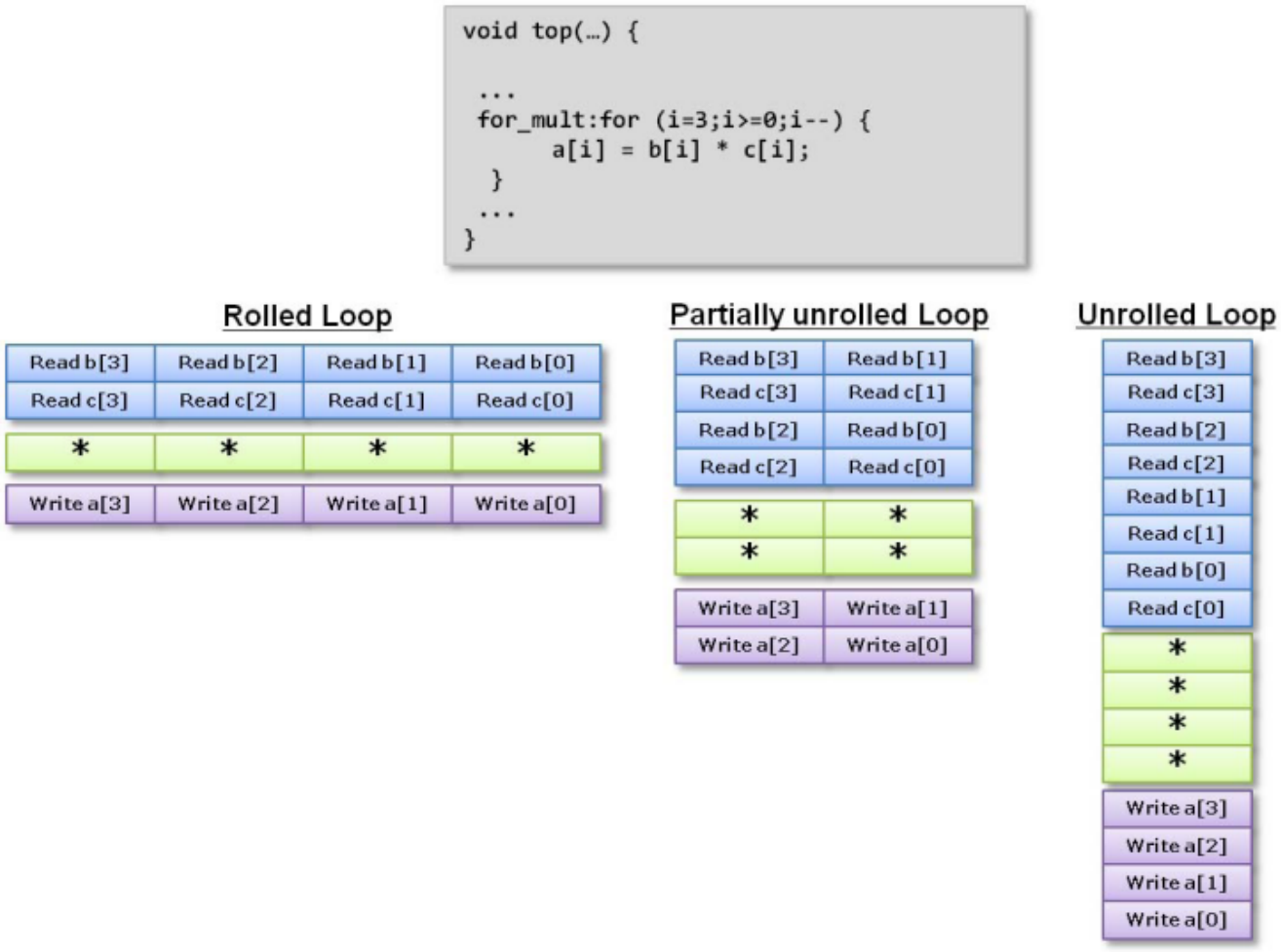


Figure 65: Loop unroll

Source: Xilinx

Vivado HlS also has the strength to fully or partially unroll for-loops in an automatic way,with applying the equivalent synthesis directive UNROLL. The way unroll directive can change a set of code is depicted in the following figure: It is concluded that with just applying the unroll directive,every user can produce a variety of different implementations,based on the varying unroll depth[6]. There are three ways to perform loop unrolling:

- **Rolled loop**:in the case that the loop is rolled, each iteration is executed in a single clock cycle. The initial version requires four cycles to be completed and needs one multiplier and a block RAM. Notice here that BRAM can be a single-port RAM.
- **Partially unrolled loop:** in that case,the loop is partially unrolled by the factor of 2,so this version needs two multipliers and dual-port RAMs,since data has to be read and written in the same clock cycle twice. However,the improvement is that the unrolled version needs 2 cycles to be completed:both initiation interval and latency are cut to half,comparing to the rolled version.
- **Unrolled loop:** That case is when the loop is fully unrolled. That is,it its completed in one cycle. In that example,the implementation requires four multipliers. The main constraint here is that we need to support 4 reads and 4 writes executions in the same cycle. Thus,we need to perform array partition,since block-RAMs have maximum 2 ports.

The unroll directive can be applied inside the source code like following:

```
#pragma HLS unroll skip_exit_check factor=<int>
```

within the desired code region.

4.3.2 Latency optimizations

After making throughput optimizations,Vivado HLS provides three kind of synthesis directives that allow us to either reduce latency on our design,or indicate the desired value of latency[6]. In the following chart we present the three of them:

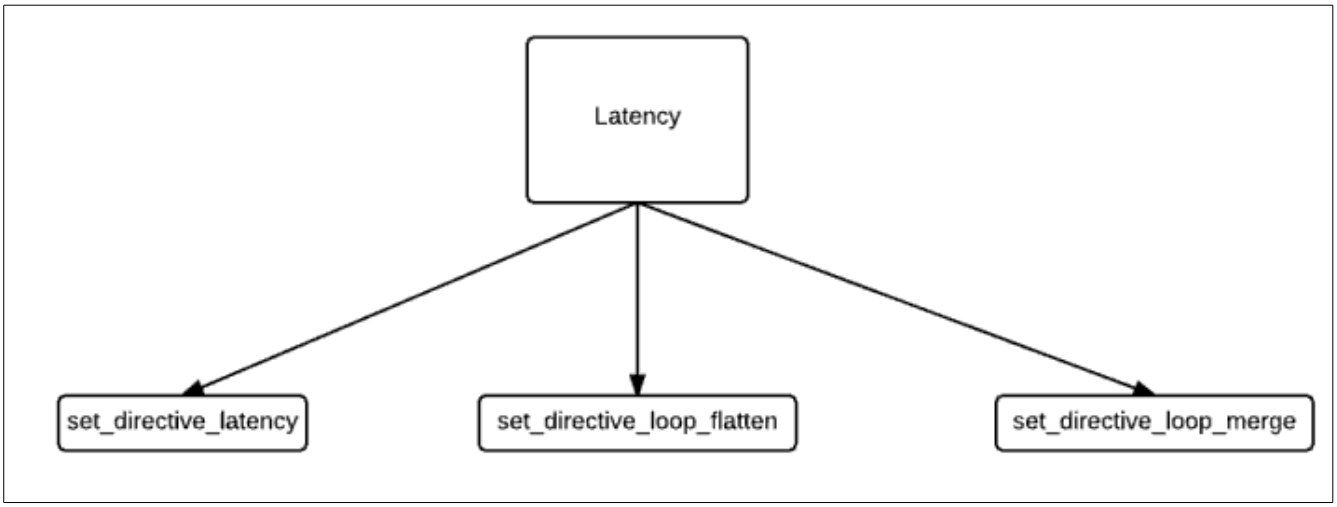


Figure 66: Latency optimizations

4.3.2.1 Latency directive

Vivado HLS supports use of latency constraint,which is defined by the LATENCY directive and it can be applied either in function loop or region. The syntax is simple:

```
#pragma HLS latency min=<int> max=<int>
```

where min and max specify the minimum and maximum desired latency,respectively. So,when we place latency directive,Vivado HLS has to make sure that all operations inside the function or region must be completed within the desired number of clock cycles. If Vivado cannot achieve the desired latency,then it relaxes the constraint,so as to try to achieve the best possible latency.

A simple example of how to apply latency directive is shown below:

```
Loop_A: for (i=0; i<N; i++) {
#pragma HLS latency max=10
        ..Loop Body...
}
```

In that case,we placed latency directive inside the loop and so we give Vivado HLS the command to limit each iteration's latency to 10 clock cycles.

If we wish to place latency directive to all iterations,we place the latency directive just outside the loop:

```
Region_All_Loop_A: {
#pragma HLS latency max=10
Loop_A: for (i=0; i<N; i++)
        {
                ..Loop Body...
        }
}
```

4.3.2.2 Loop merge directive

When we deal with code that contains lots of sequential loops,there is an additional unnecessary overhead that can increase the clock cycles needed(latency). In the figure below we present an example of two sequential loops and how that programming style can have negative effect on the design' s performance.

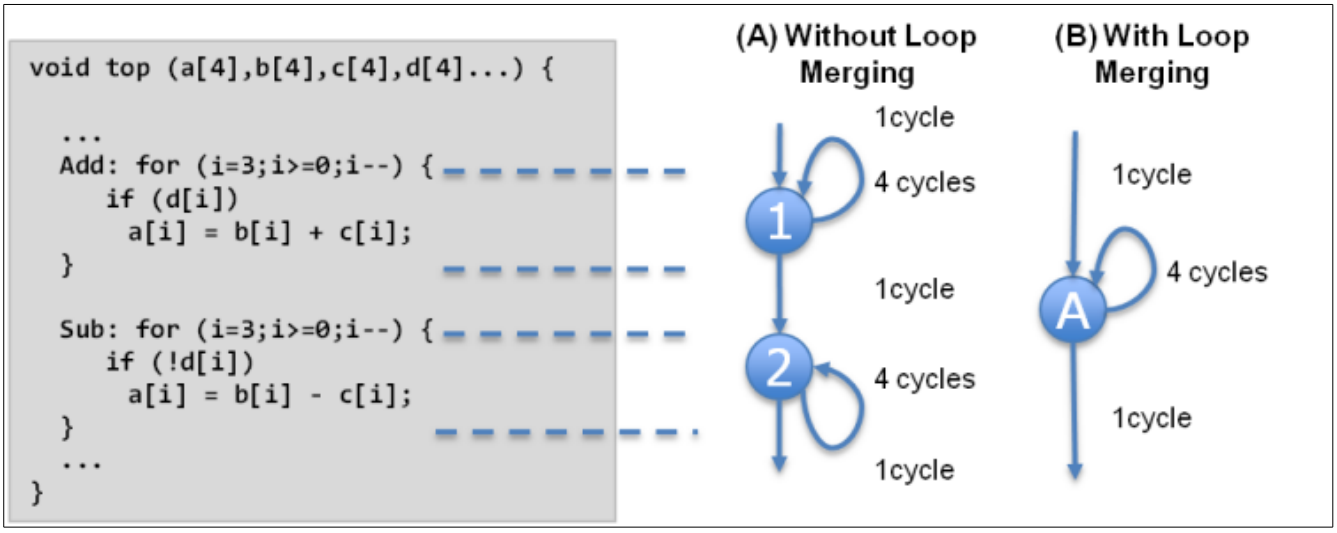


Figure 67: Loop merging

Source: Xilinx

As it is shown,in the first case the implementation takes 1 cycle to enter the ADD loop,4 cycles to do the operations and 1 cycle to leave the loop and enter the next one,SUB loop. The sub loop needs the same 4 cycles to complete its operations and 1 cycle to leave the loop. In total,the non-merged version needs 11 clock cycles to be completed. Whereas, in the merged version we notice that we need 1 cycle to enter the merged loop,4 cycles to do its operations and 1 cycle to leave the loop. In total,the merged version takes 6 loops,almost half of the non-merged version.

To apply loop_merge optimization all we have to do is to place the following pragma inside the region we wish to merge:

```
#pragma HLS loop_merge
```

Besides the reduction of the clock cycles,loop merging allows concurrent execution. In the previous example,we need to use a dual-port block RAM in order to perform add and sub operations in the same cycle. In general,there are some limitations concerning loop merging:

- Loop bounds must have the same values. If they are constants,then the maximum value is used as the bound in the merged loop. If one bound is variable and the other one is constant,then merging cannot be applied.
- No dependencies are allowed between the loops operations.

In our implementation of Harris algorithm,loop merging was not applicable because in both functions that compute partial derivatives and do convolution(imgradient and imblurg respectively) there are dependencies that concern computations of future elements that do not allow loops to be merged.

Vivado HLS produced the following error:

@E [XFORM-522] Cannot merge loops in region 'label0': data dependence(s) between loops prevent merging.

4.3.2.3 Loop flatten directive

Alike before,it is known that there are additional clock cycles to enter or leave nested loops. In the following example,it requires one clock cycle to move from one loop to another:

```
void foo_top { a, b, c, d} {
...
Outer: while(j<100)
        Inner: while(i<6)// 1 cycle to enter inner
                ...
                LOOP_BODY
                ...
        } // 1 cycle to exit inner
   }
  ...
}
```

Taking into account the outer loop bounds,it requires 100 cycles to enter and 100 to leave the inner loop. So,in total there are 200 additional loops necessary. Thus,we perform loop flatten directive in order to flatten all kinds of nested loops(perfect and semi-perfect)into a single loop hierarchy,achieving less number of cycles needed to execute all the operations in the loop. In addition,flattened loops are able to be optimized as a single loop,achieving greater level of optimization in the united loop body. To apply loop flatten directive,we simply have to add the following pragma in the C source code inside the desired region:

```
#pragma HLS loop_flatten
```

4.3.3 Timing Results

Applying all the optimizations above we selected the timing results from the co-simulation stage of design. The co-sim was implemented in SystemC. The output figure is shown below:

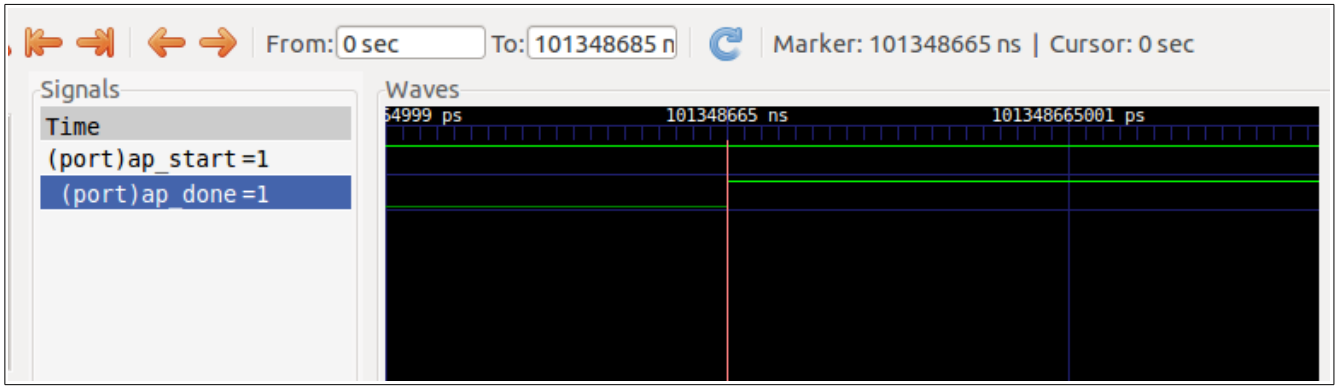


Figure 68: SystemC output

When the signals ap_start and ap_done are both in logic 1 then we know that the output result is ready and we count the point in which the result is created. In the figure above,the point is $t_0 = 101{,}346\,ms$ .In the same way we measured all of the timing points which are shown in the following tables:

	time(ms)			
	no optimizations	loop flatten	pipeline	pipeline+array partition
128x128x	25.729	24.278	12.893	13.4
256x256	101.346	95.631	50.8	52.662
1024x1024	24398	23022.2	3103.7	3225.9

Table 1: Timing results(1/2)

	time(ms)	
array partition(no pipeline)	unroll(unroll_factor=10)	unroll(unroll_factor=50)
27,831	24,193	24,085
109,626309	95,296227	94,870815
6700,03494	5824,22282	5798,2229

Table 2: Timing results(2/2)

4.3.4 Area optimizations

4.3.4.1 Data types and Bit-lengths

As discussed before,the bit-lengths of variables in C can affect the size of the memory required for the RTL implementation. With Vivado arbitrary precision types we can adjust the word-length exactly to the design' s needs,decreasing the number of operations and possibly enlarge the initiation interval(II) and cut off latency. We took advantage of the Vivado types in previous section,achieving great memory save.[6]

4.3.4.2 Function Inlining

Inlining functions actually removes any function hierarchy,embedding all in one single function. Inlining can possible reduce area utilization because it permits the components of each inlined function to "cooperate" or optimized more effectively. Sometimes,when there are small functions Vivado HLS inlines them automatically. Inline directive has also the capability of making all the functions below inlined by using the recursive option. So,if we use apply inline directive recursive option to the top-level function,we actually remove all the hierarchy of the design. To perform inlining,we simply add the following pragma in the C source code within the function or region we wish to inline:

```
#pragma HLS inline <region | recursive | off>
```

With the option "off " we can eliminate functions or regions to be inlined,when we have placed inline directive at the top-function,or prevent Vivado HLS from automatically inlining them.

In our particular case,since as we mentioned before,the most computationally expensive function is imblurg(),we decided to inline it to our top function in order to reduce computational overhead and make the RTL implementation more resource effective.

4.3.4.3 Directive array_map

A usual technique to reduce the percentage of memory utilization when an implementation consists of many small arrays is to combine the small arrays into a larger one. In general,when an array is mapped into a block RAM there is the possibility that the size of the array does not cover the whole capacity of the block RAM. So,a more effective use of the FPGA's resources would be to create a large array that contains all the small ones. Hence,the redundant memory units would be used optimally.

To perform mapping small arrays into a larger one we simply have to place the directive array_map into our implementation by adding the following pragma into the source code:

```
#pragma HLS array_map variable=<variable> instance=<instance> <horizontal, vertical> offset=<int>
```

The variable corresponds the array we are applying the directive,instance is the new name of the target array of the mapping and the offset is the integer value which defines the absolute offset in the target array for current mapping operation. Then,we have to choose between the two types of array mapping:

- **horizontal mapping**:the default type of mapping which combines the original arrays into a sequential order creating a single bigger array.
- **vertical mapping**: this type creates a new array with longer word-length than the original small arrays.

In our case,we firstly implemented horizontal mapping in the gradient arrays by

placing the following pragma into the C source code:

```
#pragma HLS ARRAY_MAP variable=gradx instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=grady instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=gradxy instance=array4 horizontal
#pragma HLS ARRAY_MAP variable=gradx2 instance=array4 horizontal
#pragma HLS ARRAY_MAP variable=grady2 instance=array4 horizontal
```

We arranged together the first gradients(gradx and grady) into array3 because the are used together,and then the rest of the gradients (gradxy,gradx2 and grady2) into array4.In the following table we present the synthesis report:

	No array_map	array_map horizontal	array_map vertical
	BRAM_18K		
Total	499	454	499
Available	890	890	890
Utilization(%)	56,00%	51,00%	56,00%

Table 3:Array_map results

It is shown that with the horizontal mapping the utilization percentage is reduced by 5% or 45 BRAMs. However,the vertical mapping did not bring any improvement in area utilization. That difference can be explained by the detailed memory report of the Vivado HLS:

```
* Memory:
+--------------+--------------------------------+---------+-------+-----+------+-------------+
|    Memory    |             Module             | BRAM_18K| Words | Bits| Banks| W*Bits*Banks|
+--------------+--------------------------------+---------+-------+-----+------+-------------+
|array3_U      |harris_findCorners_array3       |      150|  67584|   30|     1|      2027520|
|array4_U      |harris_findCorners_array4       |      225|  67584|   45|     1|      3041280|
|cmap_U        |harris_findCorners_cmap         |        4|  65536|    1|     1|        65536|
|cornerness_U  |harris_findCorners_cornerness   |      120|  67584|   24|     1|      1622016|
|gmask_U       |harris_findCorners_gmask        |        0|      7|   15|     1|          105|
+--------------+--------------------------------+---------+-------+-----+------+-------------+
|Total         |                                |      499| 268295|  115|     5|      6756457|
+--------------+--------------------------------+---------+-------+-----+------+-------------+
```

Figure 69: array_map vertical

In the horizontal version,we truly have improvement because we take advantage of the redundant memory blocks of the non array_map version. So,instead of having

five arrays of 75 BRAM_18K in the initial version,which makes a total of 375 block RAMs,we map them into two larger arrays,array3 and array4 respectively,with 135 and 195 block-rams. In total,330 block-rams,that makes 45 less. However,vertical mapping just created two larger arrays of 150+225=375 block-rams making no impact on area utilization. That happened because as we can derive from the Bits column,the bit length was doubled in the first array and grew up three times in the second. So,we stay for the horizontal version.

4.3.4.3 Directive Resource

In Vivado HLS when a C operator is used,like +,-,* or /, in synthesis step they are implemented as hardware cores. Vivado can select automatically the optimal core for each case. However,using the RESOURCE directive user can determine exactly which operator to be used in RTL description. The syntax of resource directive is simply adding the following pragma to the C source code:

```
#pragma HLS resource variable=<variable> core=<core>
```

Variable is the argument that can be an array,an arithmetic operation or a function argument and the core is the desired specific library resource which is going to implement the variable in the RTL behavior.

In our implementation we selected a dual-port asynchronous RAM ,implemented with LUTs. The results for Test Case 1(image input 256x256) compared with the initial non-dual port version are shown in the next table:

Fpga resources	265x256	256x256 dual port
	% BRAM Utilization	% BRAM Utilization
Factor=1	56	17
Factor=2	33	10
Factor=4	22	7
Factor=8	11	3
Factor=16	6	2

Table 4: Dual port Ram results 256x256

And are graphically presented in the following chart:

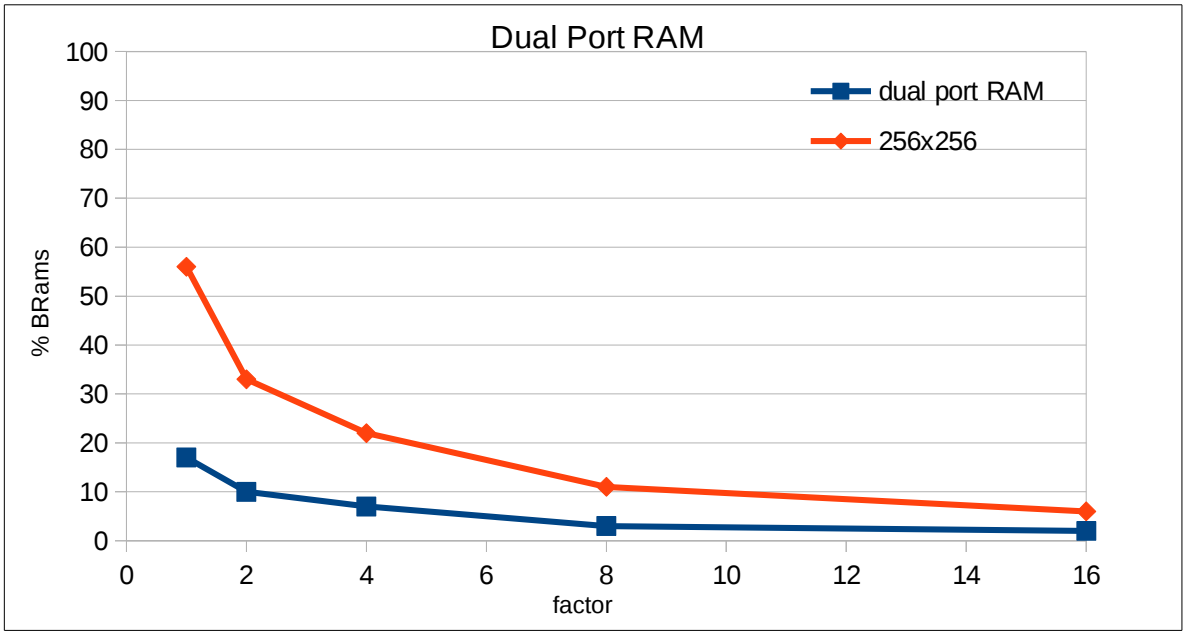


Figure 70: Dual port RAM 256x256

For Test Case 2(input image 1024x1024) we do the same comparison in the following table:

Fpga resources	1024x1024	1024x1024 dual port
	% BRAM Utilization	% BRAM Utilization
Factor=1	730	226
Factor=2	374	118
Factor=4	196	64
Factor=8	107	37
Factor=16	62	24

Table 5: Dual port Ram results 1024x1024

And are graphically presented in the following chart:

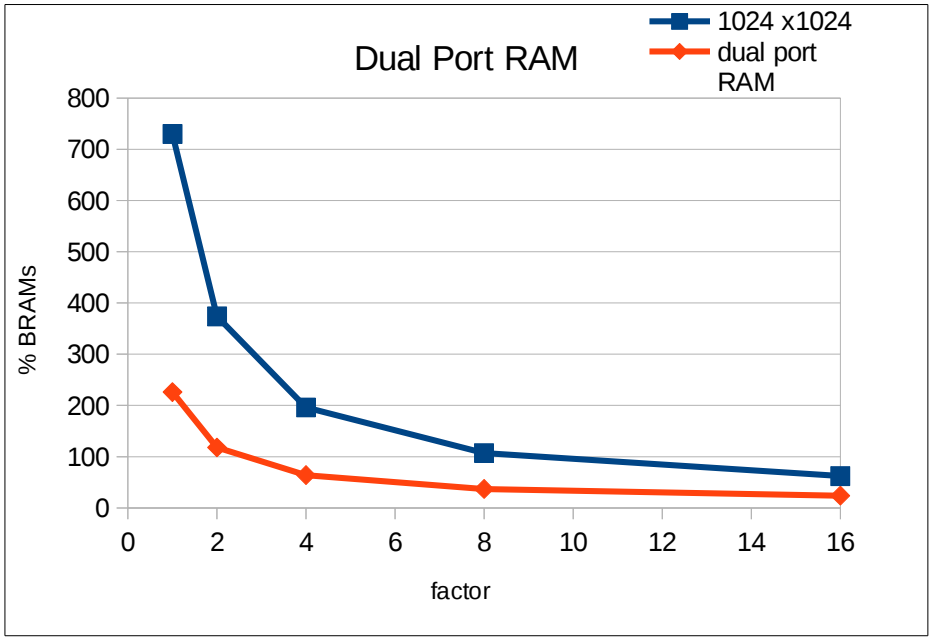


Figure 71: Dual port Ram 1024x1024

4.4 Synthesizable dynamic memory allocation

Despite all the possible memory optimization techniques,we encountered a bottleneck considering the memory utilization. The main reason is the natural operation of the FPGA,since it allows only static memory allocation,which sets the lower boundary of memory optimization.

We overcame this issue by incorporating an HLS-synthesizable dynamic memory management library[27],which was called "Memluv". The next figure presents the flow chart of the dynamic memory allocator:

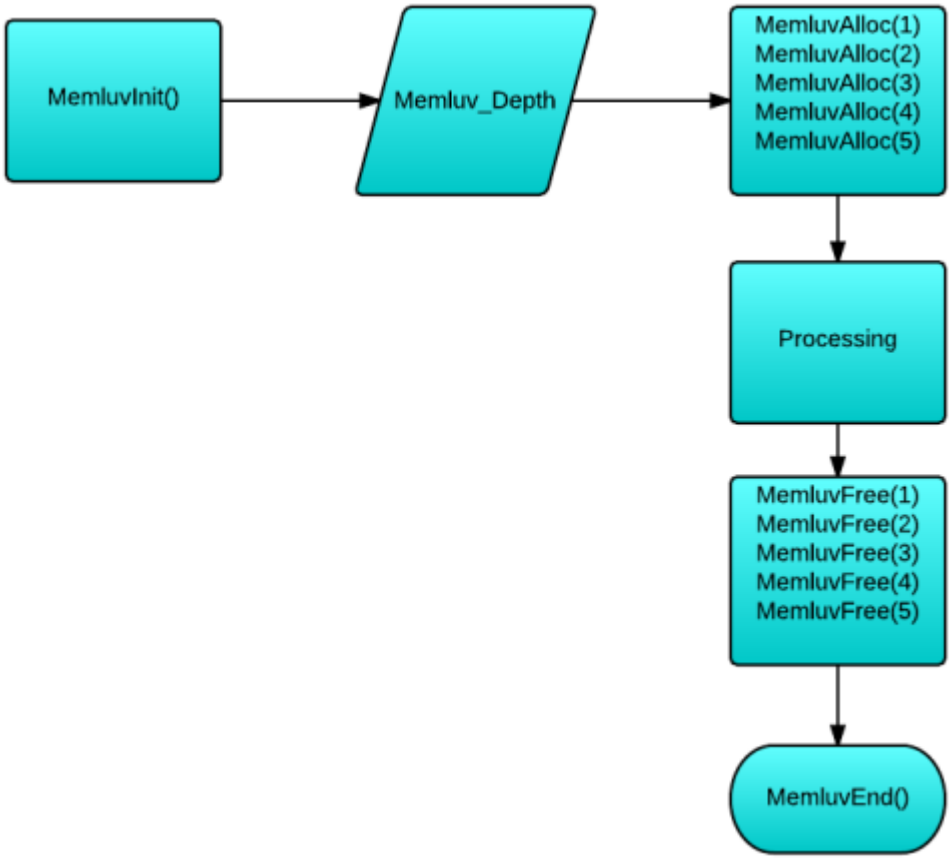


Figure 72: Memluv Allocator Flow Chart

In the flow chart below we can see that Memluv allocator starts with function MemluvInit().Then,it expects the user to define the actual size of the memory to be allocated-"Memluv_Depth". Then,we can apply "MemluvAlloc()",which allocates the desired number of bytes. We can perform multiple times " MemluvAlloc()",but it is mandatory that the sum of the partial number of bytes does not exceed the total memory allocated initially,that is the " Memluv_Depth'. Otherwise,a segmentation fault appears. After the memory space is used,we can free it with the function "MemluvFree()" and allocate it again,if necessary.In the end,we terminate the process by the function"MemluvEnd()".

The main advantage of Memluv allocator is that we can define a certain memory space and do multiple allocations that cover the memory needs of our application.

Therefore,we applied Memluv allocator to our implementation and we present the results below(compared to static memory allocation):

	BRAM_18K	BRAM_18K %	DSP48E	DSP48E%
static allocation	564	63	81	9
memluv allocation	667	74	78	9

	FF	FF%	LUT	LUT %
static allocation	12766	3	28602	14
memluv allocation	13621	3	30710	15

It is depicted that memluv allocator needs more block RAMs than the static memory allocation,as expected.

Then we tried to import more harris cores inside our implementation. Then we present the results:

- Harris cores=2

	BRAM_18K	BRAM_18K %	DSP48E	DSP48E%
static allocation	724	81	81	12
memluv allocation	667	74	101	12

	FF	FF%	LUT	LUT %
static allocation	16993	4	34037	16
memluv allocation	18106	4	37595	18

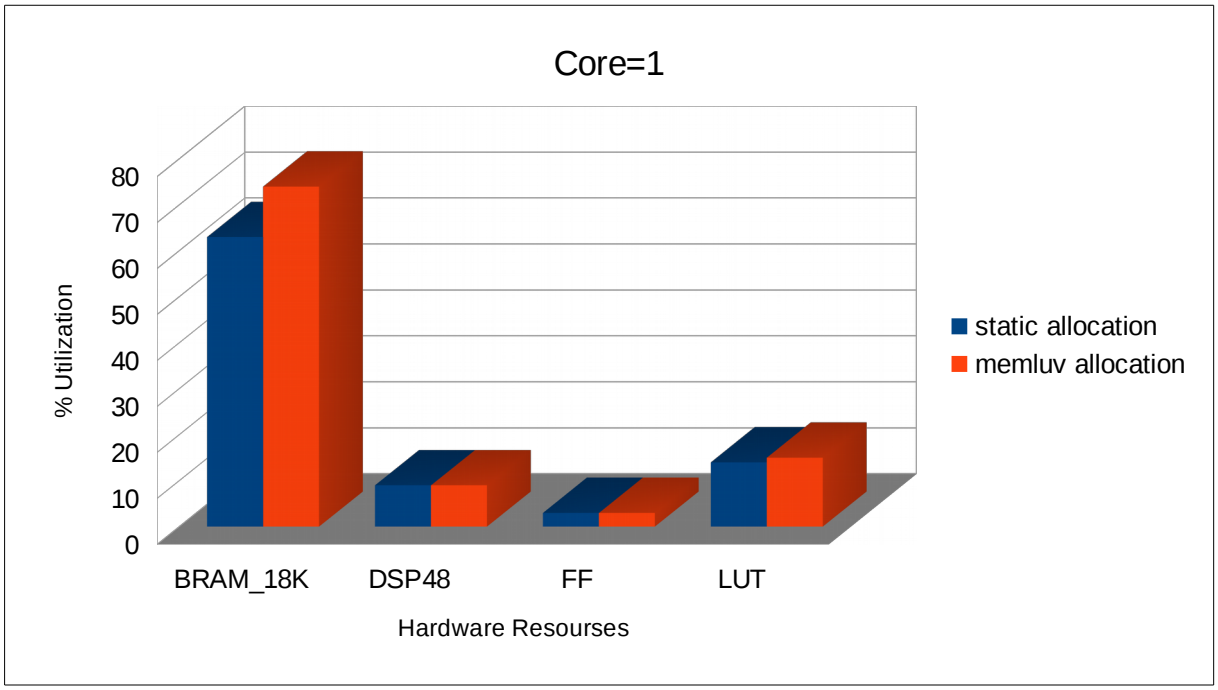
- Harris cores=4

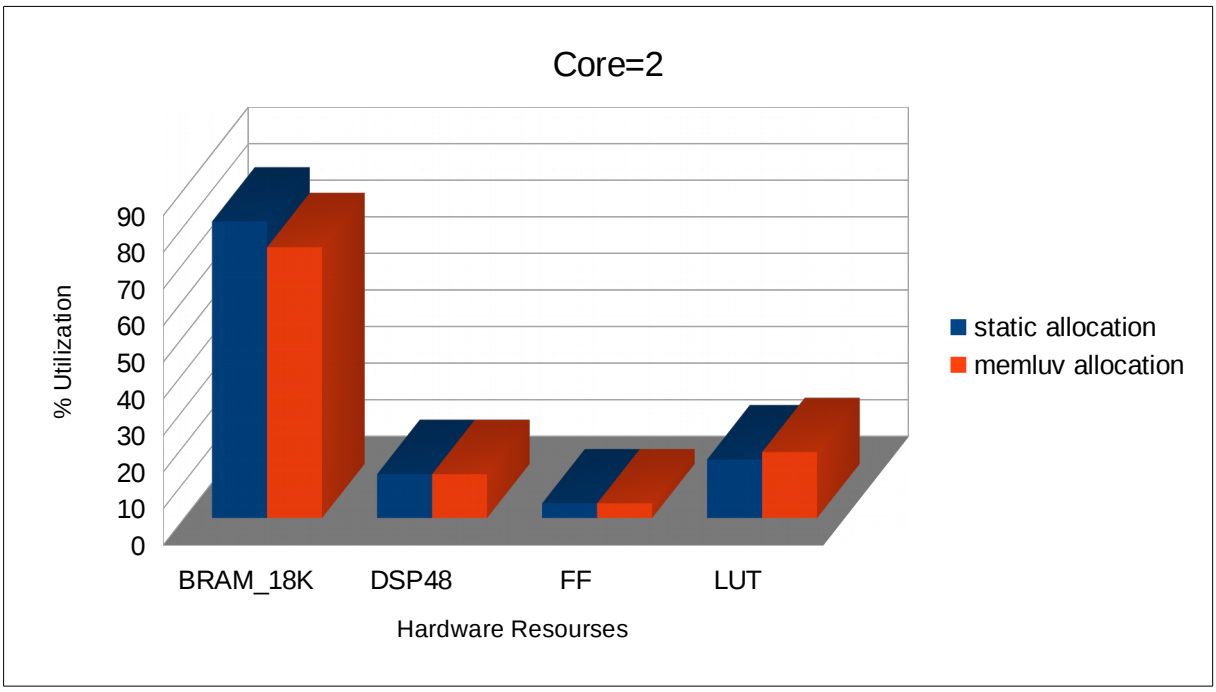
	BRAM_18K	BRAM_18K %	DSP48E	DSP48E%
static allocation	1044	117	145	17
memluv allocation	667	74	147	17

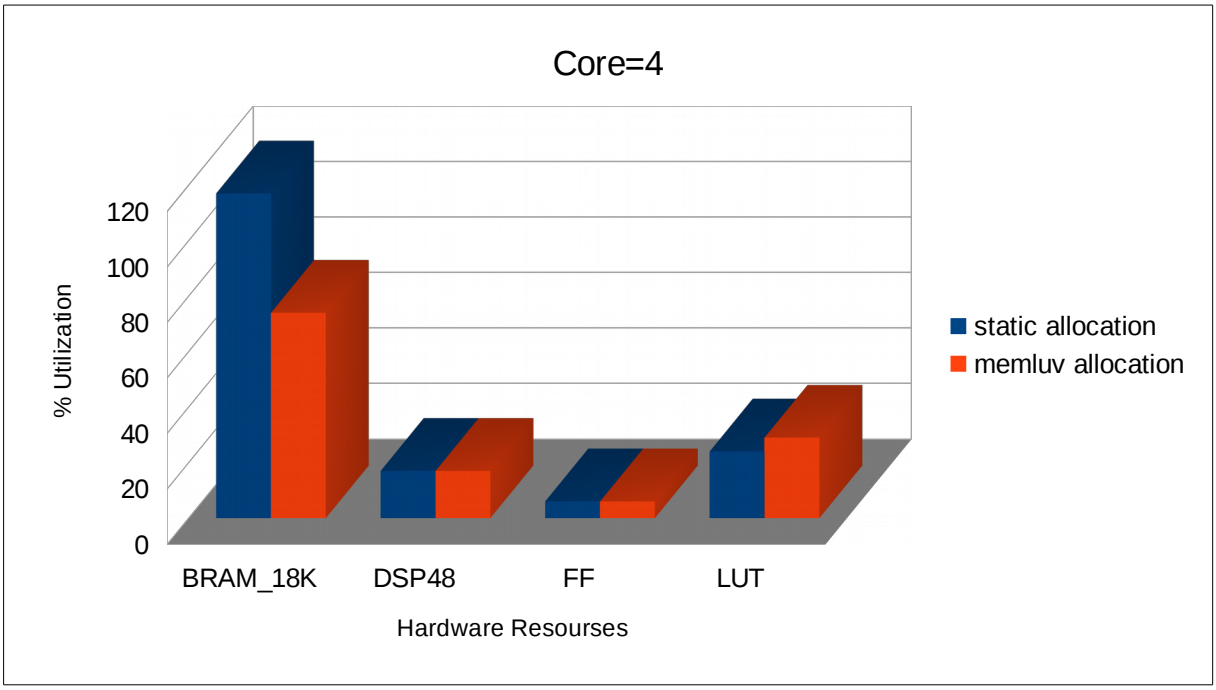
	FF	FF%	LUT	LUT %
static allocation	25002	6	49340	24
memluv allocation	28354	6	59576	29

We realize that with static memory allocation it would not be possible to implement 4 Harris cores onto FPGA,since it exceeds the available BRAM memory. However,using dynamic memory allocation permits the multiple execution of Harris cores without any more need for memory resources.

Obviously,the multiple cores cannot be executed simultaneously,but sequentially. That means we could embed more Harris algorithms,with the use of a single FPGA board. If dynamic memory allocation could not be synthesized,we should use another FPGA board or an array of FPGA boards. In both cases,the cost of our implementation would be significantly higher. In the following charts we present the previous measures graphically:




Core=2
% Utilization
90
80
70
60
50
40
30
20
10
0
BRAM_18K
DSP48
FF
LUT
Hardware Resourses
static allocation
memluv allocation


Core=4
% Utilization
120
100
80
60
40
20
0
BRAM_18K
DSP48
FF
LUT
Hardware Resourses
static allocation
memluv allocation

Chapter 5: Conclusion

In this thesis we examined an applicability study of modern HLS for computer vision algorithms. As a case study,we employed the Harris & Stephens algorithm [19] targeting a Kintex-7 FPGA device. We examined 2 cases with 2 different input image sizes(256x256 and 1024x1024) and one case with 256x256 size and larger number of detectable corners. In all of the cases, our implementation reacted efficiently.

So,our proposed methodology is to apply task-level algorithmic transformations and then continue with system-level optimizations through HLS directives.

The first step was to make a profile of our C-code implementation using valgrind tool. Based on our results,we concentrated our attention to a specific function that occupies about 62% of our total execution cycles.

At next step,we performed a number of optimizations to improve the initial performance,in terms of throughput,latency and area utilization.

The first step was to optimize the word-length and replace all the floating point arithmetic with integers. Next,we took advantage of the arbitrary precision data types that Vivado HLS provides (fixed point replacements of build-in C data types).The error that was expected to occur was measured inside the desired error margin,which is deviation of 0.1 in the coordinates of each detected corner. We also eliminated all the dynamic memory functions(remove malloc()/free()) and replaced them with static memory allocations(e.g arrays).

Then we applied parametric fragmentation of the input image and thus the processing procedure changed:all of the calculations repeated multiple times for each one of the image fragments.

Finally, we performed a number of optimizations using the synthesis directives that Vivado HLS provides. We placed directives for increasing throughput(pipeline,unroll,array partition),for minimizing latency(loop merge,loop flatten) and to reduce area utilization(array map,resource,inline).

We also examined another case study with implementing a dynamic memory allocator that is synthesizable and thus can be used instead of static memory procedures. The result is that our implementation is able to embed multiple algorithm cores,in our special case is Harris detector,without requesting more

memory. However,it is inevitable that there would be necessary more resources besides memory(like LUT's,DSP's and FFs).

Combining all of the possible solutions led in a design space exploration. Band-devision techniques provide a reduction of about 9x BRAM decrease at the best case. Our proposed prototyping framework also achieved a simulation environment of up to about 6x faster .

Therefore,our implementation proved that FPGA's can accelerate significantly a high-complexity Computer Vision algorithm,like Harris corner detection, and deal with the intensive computational load efficiently.

Chapter 6:Future Work

During the past few years,traditional ways of performance improvement have to be reconsidered,since Moore's Law has to be re-explained. According to the new tendencies, hardware designers have to deal with new constraints:the exponential clock rate growth has reached an end so the new era is to double the number of cores per chip,instead of doubling the clock frequency every 18 months[24]. Furthermore,over the last few years,networking systems are the on the focus of attention concerning their performance,since they are enriched with more and more capabilities in software layer[25]. The solution to both needs is next-generation System-on-Chip (SoC) communications processors that combine multiple cores with multiple hardware acceleration engines.

Until recently,the solution to every computational challenge was Moore's Law-doubling the processor performance every 18 months. However,the data growth outnumbers Moore's Law and so general purpose processors,despite of embedding multiple cores,cannot reach today's performance requirements. They are just too slow to implement functions that are executed in the core of several popular applications,like cryptographic security encryption/decryption,digital signal processing or traffic management,which are necessary for achieving Quality of Results(QoR). Sometimes,it happens these functions to be executed sequentially,so the presence of multiple cores cannot provide simultaneous access. For all the previous reasons,such computationally intensive functions are usually implemented in hardware.

The procedure of implementing one of these functions in hardware is presented in that thesis. Especially,the Harris & Stephens corner detection algorithm is successfully mapped onto a specific FPGA device and its performance is optimized through the design process.

A step forward would be to embed a multi-core accelerator system alongside with a CPU processor,creating a *Sytem-on-Chip(SoC)*.FPGAs are now powerful computing devices and they are suitable for use as fine-grained accelerators. This trend is currently followed by vendors and what is under research is to combine a

vendor's IP (intellectual property) into a custom acceleration engine,within a SoC. That is,to manufacture a CPU-FPGA hybrid chip that consists of traditional CPU cores with FPGAs in a single chip. Thus,with FPGAs integrated into CPU's,each chip will be possible to be customized to optimal performance to specific workloads. Until now what has been announced is that FPGAs are to be embedded into data center CPUs to deal with web-based, storage or networking workloads. [26]

One of the largest sector vendors,Microsoft,announced recently that they used FPGA to accelerate data center performance and reported several impressive results :

- 95% more throughput
- only 10% more power
- 30% total cost of ownership

It is also revealed by Intel that they plan to integrate FPGAs into CPUs with estimated performance improvement at about 20x.

There are still some challenges into co-operation of CPUs and FPGAs,like the integration process itself. Another issue is the memory sharing and the coherence protocols that should be applied,or generally the communication between the components of the hybrid chip.

In the end,however,those limitations will be surpassed and reconfigurable computing will be widely used in distributed systems,providing an efficient solution to increase CPU's performance. So,Hardware accelerators are capable of keeping pace along with the intensive grow of data volume and give a reliable alternative to CPU-based multi-core systems.

References

[1] Mike Thompson, EE Times "*Mixed-signal FPGAs provide Green Power"*,7/2/2007, (URL:*http://www.eetimes.com/document.asp?doc_id=1271543*)

[2] Peter Clarke, Xilinx, *"ASIC vendors talk licensing", EETimes*, 6/22/2001, (URL:*http://www.eetimes.com/document.asp?doc_id=1180867)*

[3] Dylan McGrath,EE Times, "*Gartner Dataquest analyst gives ASIC, FPGA markets clean bill of health"*, 6/13/2005,(URL:*http://www.eetimes.com/document.asp?doc_id=1154636)*

[4] Xilinx Inc, "*Virtex-4 Family Overview"* , August 30, 2010, (URL:*www.xilinx.com/support/documentation/data_sheets/ds112.pdf*

[5] David W. Page, LuVerne R. Peterson, " *Re-programmable PLA"*, Jan 11, 1983(URL:*http://www.google.com/patents/US4508977)*

[6]International Directory of Company Histories, Vol. 16. St. James Press, Xilinx, Inc. History, 1997

[7] Wayback Machine,"*History of FPGAs*", April 12, 2007(URL:*https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm)*

[8] Clive Maxfield(2004), "*The Design Warrior's Guide to FPGAs: 1st Edition:Devices,Tools and Flows"*,Elsevier. p. 4,ISBN: 978-0750676045

[9] Dylan McGrath, *EE Times*, "*FPGA Market to Pass $2.7 Billion by '10, In-Stat Says",* May 24, 2006.

[10] Xilinx, "*Zynq-7000 All Programmable SoC Technical Reference Manual"*, September 19, 2014

[11] Aldec Inc. (URL:"*https://www.aldec.com/en/company*)

[12] Rahul Gargon, "*FPGA news roundup:Microsoft "Catapult",Intel's hybrid and Xilinx OpenCL*", June 21,2014 (URL:"*http://www.anandtech.com/show/8189/fpga-news-roundup-microsoft-catapult-intels-hybrid-and-xilinx-opencl- ")*

[13] Xilinx, “Introduction to FPGA Design with Vivado High-Level Synthesis” , July 2, 2013(URL:*http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf)*

[14] Reinhard Klette, “*Concise Computer Vision:An introduction into Theory and Algorithms”*,Springer 2014,ISBN:978-1447163190

[15] Linda G. Shapiro and George C. Stockman, “*Computer Vision”*, 2001,ISBN:978-0130307965

[16] Katz, Gregory, "*2018 mission: Mars rover prototype unveiled in UK"*, 27 March 2014

[17] Nalini K. Ratha,Anil K. Jain, "*Computer Vision Algorithms on Reconfigurable Logic Arrays"*,IEEE Transactions on Parallel and Distributed Systems,January 1999

[18] W. James MacLean ," *An Evaluation of the Suitability of FPGAs for Embedded Vision Systems* ", Computer Vision and Pattern Recognition – Workshops.IEEE Computer Society Conference, 2005

[19] Chris Harris & Mike Stephens, “A Combined Corner and Edge Detector”, Plessey Research Roke Manor, United Kingdom,1988

[20]Xilinx, “*Vivado Design Suite User Guide:High-Level-Synthesis*”, May 30, 2014

[21] http://valgrind.org/docs/manual/quick-start.html#quick-start.prepare, Valgrind,

[22] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung, “*Reconfigurable computing: architectures and design methods”*, March 2005

[23] Xilinx,“*Vivado Design Suite User Guide:Getting Started*”, July 25, 2012

[24] Rahul Gargon, "FPGA news roundup:Microsoft "Catapult",Intel's hybrid and Xilinx OpenCL", June 21,2014

[25] David Sonnier, "Next-generation multicore SoC architectures for tomorrow's communications networks", December 11th, 2012

[26] Rahul Gargon, "FPGA news roundup:Microsoft "Catapult",Intel's hybrid and Xilinx OpenCL", June 21,2014

[27] Dionysios Diamantopoulos, Sotirios Xydis, Kostas Siozios, and Dimitrios Soudris,"*Dynamic Memory Management in Vivado-HLS for Scalable Many-Accelerator Architectures"*,11th International Symposium on Applied Reconfigurable Computing 15-17 April 2015