Vrije Universiteit Brussel

Faculty of Applied Sciences and Engineering
INDI Dept.

Performance Analysis of a Real-Time Video Processing System

Graduation thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Engineering: Electronics-ICT

Frank Vanbever

Promotors: Em. Prof. Dr. Ir. Erik D'Hollander
Prof. Dr. An Braeken

Advisors: In. Bruno Tiago Da Silva Gomes

January 2014

Abstract

acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Up until halfway the first decade of the new millennium it was possible to gain computing performance whilst also being able to maintain the sequential programming paradigm. This was due to Moore's law, stating that the number of transistors on integrated circuits double approximately every two years. There was no need for research into explicit parallelism because the next generation of computing devices was just around the corner which would make the research obsolete.To perpetuate the sequential programming paradigm several innovations such as multiple issue, deep pipelines and out of order execution were introduced into processors which were inefficient in both the use of transistors and power. Eventually though it became impossible to progress any further whilst still supporting the sequential paradigm. The integrated circuit industry was unable to continue decreasing the size of MOSFETs whilst continuing to increase the clock frequency. The industry had hit what is called the power wall. The solution to this problem was to go over to parallel processors, meaning that there is more than one processing unit working at a time. A lot of real world applications are parallel, and hardware can be made parallel with relative ease. The problem lies in the programming model, how to exploit this parallelism and make programming for these parallel architectures easier and transparent for the programmer.

1.1 Computing Components

1.1.1 Multicore Processor

The multicore processor is the solution presented for the aforementioned problems by the traditional CPU manufacturers such as Intel and AMD. The idea behind this type of processor is to place a number of cores (currently up to eight) on the same die. This presents a compromise between maintaining sequential performance whilst also providing a certain advantage of parallel processing. Parallel programming for these processors presents certain challenges whilst their modest parallelism cannot provide a dramatic improvement in power performance. Multicore processors are unlikely to be a one-size-fits-all solution to the parallel problem.[3]

1.1.2 Graphics Processing Units

Graphics Processing Units Graphics processing units are a type of coprocessor in in traditional computers meant to process images for output to the display. Recently however there has been increased interest in the GPGPU, the general purpose graphics processing unit. These processors implement a different paradigm, namely the manycore paradigm. A GPU is a processor with hundreds single instruction multiple data cores, each of which is heavily multi-threaded. Because of this large amount of cores the FLOPS (floating point operations per second) is unrivalled. [4]GPU's, due to their SIMD nature present some problems, conditional execution paths for example, present a serious overhead on the GPU. GPU's are programmed with either OpenCL (open standard) or CUDA (proprietary to Nvidia)

1.1.3 Field Programmable Gate Array

FPGAs are devices containing a vast amount of configurable logic linked by programmable connections. This logic is comprised of lookup tables grouped together into configurable logic blocks. Any combinatorial function can be programmed into these LUT's. Next to these uncommitted logic blocks a typical FPGA also contains several blocks with a specific function such as block ram and DSP multipliers. FPGAs are an interesting competitor in the parallel processing field because they aren't constrained by the Von Neuman architecture. FPGAs follow the dataflow paradigm in which the data flows through the logic. Implementing a data-flow is inherently parallel. The different stages in the datapath can also be made sequential effectively making the datapath a pipeline. The fine grained nature of FPGAs also means that the bitwidth can be adapted to the application.

1.2 Berkeley Dwarves

Image processing algorithms are very compute intensive. These makes them prime targets for exploiting parallelism and implementing them on parallel architectures. Which platform is the best fit however is dependent on both the algorithm and the data. A common method to subdivide parallel algorithms is presented in [3], the so called ”dwarfs”. These 13 dwarves are classes of algorithms in which the membership is defined by a similarity in computation and data movement.These 13 dwarfs are classes of algorithms in which the membership is defined by a similarity in computation and data movement. The dwarfs are:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

A thorough review of these dwarfs and what kind of computation and communication they entail goes beyond the scope of this document. More information can be found on the Berkeley View Wiki [5] and an updated view can be found in [2]. Finding out which dwarf is most suited for which platform is a very labour-intensive task. In [5] a theoretical analysis of dwarf performance on different accelerators in heterogeneous systems is given. A first point to note is that for floating point operations GPU’s are hard to beat. Fixed point numbers are a way to overcome this problem. Another point to note is that conditional elements and costly communication can wreak havoc on the accelerator’s performance. In Figure 1.1 the analysis is represented by a Venn diagram. In this diagram * denotes fixed point operations whilst ˆ denotes floating point operations.[5]

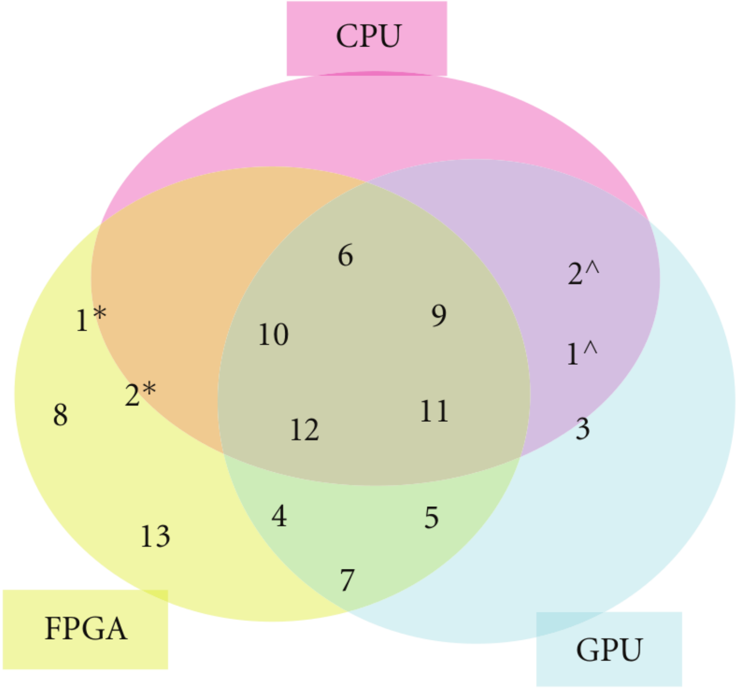


Figure 1.1: Analysis of different hardware accelerators in regards to performance on a certain dwarf

1.3 GUDI Project

This thesis is inspired by the work of the GUDI project. GUDI is an acronym for "A Combined **G**P-GP**U**/FPGA **D**esktop for accelerating **I**mage processing applications". The research starts from the observation that there is a large need for computing power to process data using computationally intensive image processing algorithms. Conventional Off-the-shelf Desktop computers don't have the necessary processing power to satisfy this demand. A lot of image processing algorithms exhibit parallelism which can be exploited by the right architecture. Two such massively-parallel architectures are GP-GPU's and FPGA's. The GUDI project has for goal to investigate the possibilities and limitations of a computer with such a heterogeneous architecture. It is an investigation into which technologies and which development tools perform best in different situations. The means through which this is done is through the implementation and performance measurement of algorithms. The ultimate goal is to split an algorithm into several parts which are executed on the technology (CPU,GPU,FPGA) most fit for the job so as to ensure optimal speed-ups.

Chapter 2

Platform Overview

2.1 Zynq-7000

The Zynq-7000 System on Chip combines a dual core ARM Cortex-A9 with Xilinx programmable logic in a single device. This combination of a CPU and an FPGA on the same device is not a new phenomenon, with examples of previous generations being the PowerPC based Xilinx Virtex-II Pro and some models of the Virtex 4 and Virtex 5 series FPGA's. The two most notable differences between these generations is the shift from PowerPC based architectures to ARM based architectures, and a notable shift in emphasis from HDL centered design to a more programmer centric view with an emphasis on high level languages.

2.1.1 Processing System

The Zynq-7000 series SoC is split into two parts: The processing system (PS) and the programmable logic (PL). The Processing system (PS) contains an Application Processor Unit (APU), memory interfaces and I/O peripherals.

APU The APU is a Dual ARM Cortex-A9 CPU which implements version 7 of the ARM ISA as well as Thumb and Jazelle instruction sets. Each core has a NEON Media Processing Engine supporting SIMD vector and scalar single-precision floating-point and integer computation and scalar double-precision floating-point computation. Each core has 32 KB instruction and 32 KB data caches and there is 512 KB shared L2 cache and 256 KB of shard on-chip SRAM memory. The APU also has a snoop control unit to maintain L1 and L2 coherency. This snoop control unit also controls the Accelerator Coherency Port, a 64-bit AXI slave port from the programmable logic, which performs the role of master, to the processing system which serves as slave. This allows direct communication between the PS and the PL through the L2 caches or on chip memory with guaranteed coherency. The also has an on-board 8-channel DMA controller with 4-channels reserved for PS to/from memory and 4 for PL to/from memory transers. Finally the Processing system also contains an interrupt controller.

Memory Controller The Memory controller supports a number of memory technologies. The system has a DDR controller which supports DDR2 and DDR3 memory, a Quad-SPI controller which converts normal memory read operations to SPI and vice versa, and a Static Memory Controller which supports NAND and SRAM/NOR type memory.

I/O Peripherals The Processing system contains quite a lot of industry standard I/O peripherals for external data communication.

- GPIO
- 2 Gigabit Ethernet Controllers
- 2 USB controllers
- 2 SD/SDIO controllers
- 2 SPI controllers
- 2 CAN controllers
- 2 UART controllers
- 2 I^{2}C controllers

These peripherals are connected to multiplexed I/O buffers which enable to externalize these signals to up to 54 pins. If there is a need for more I/O pins the signals can be routed into the PL through the extended MIO, where they can be routed directly to package pins or peripherals in the PL.

2.1.2 Programmable Logic

The programmable logic provides the same functionality that can be expected from a Xilinx FPGA. The PL in 7z010 and 7z020 Zynq SoCs is based on Artix-7 FPGAs whereas the PL in 7z030, 7z045 and 7z100 SoCs is based on Kintex-7 FPGA logic. This PL can be coupled through a couple of different interconnects, with varying degrees of interconnectedness between the PL and the PS. Of note here is that the PS has to be booted first and the PL logic has to be configured from the PL at boot or at a later time. This is another example of the shift to a more software centered view. The system has all the features one can expect from an FPGA: configurable logic blocks with look-up tables, a number of 36 KB block RAMs, DSP348E slices and configurable IO. The PL side also contains an Analog to Digital converter, and in the larger varieties of the Zynq SoC an integrated PCI Express block.

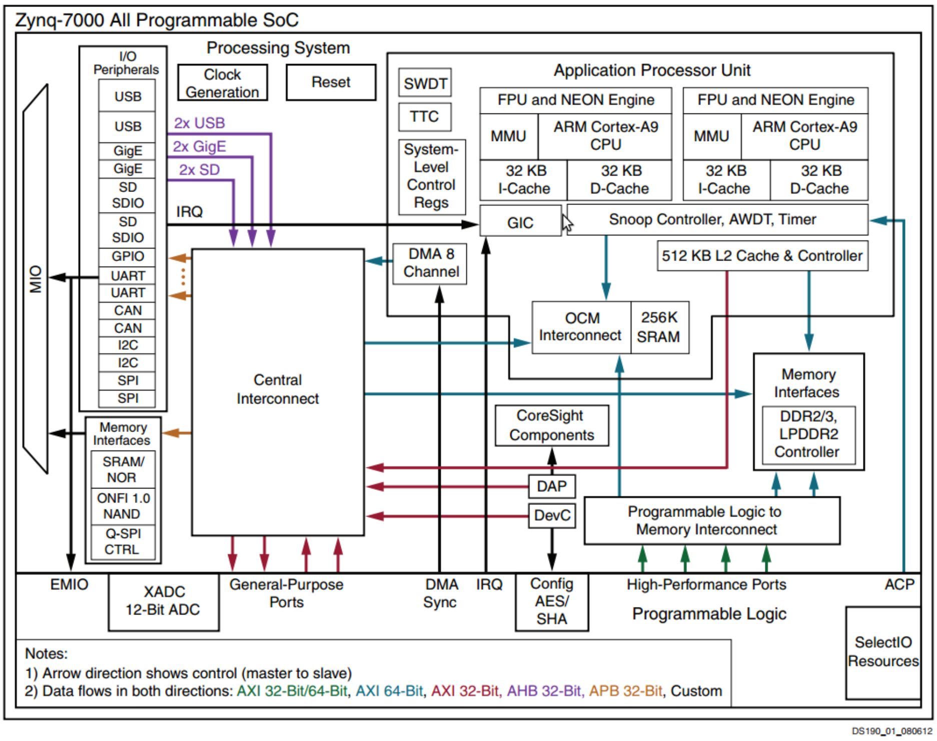


Figure 2.1: Zynq -7000 SoC overview

2.1.3 Interconnect

The interconnect system has proven to be the The Advanced Microcontroller Bus Architecture (AMBA) is an open standard specification for a bus system used in SoCs to interconnect and manage different functional blocks.
The AXI specification which is part of AMBA is targeted at high performance, high frequency systems. The Achilles' heel of HPC is usually costly communications. It would be interesting to research the performance of an on-chip interconnect system and a typical PCIe bus.Due to the recent launch of the zynq-7000 architecture there is little to no academic information to be found about it.
Profiling for this platform is done through the Xilinx SDK software. It needs to be noted that the profiler cannot profile the hardware implemented in the programmable logic. For this other tools have to be used.[23]
The board which will be used for this implementation is the Xilinx ZC702 which contains the XC7Z020 CLG484-1 AP SoC combined with the Video and Imaging kit. This contains a high definition camera and the necessary board to interface it with the SoC.

Chapter 3

High Level Synthesis

A recurring theme in the literature is the relative difficulty of implementing an algorithm on an FPGA compared to conventional implementation techniques on CPU's and GPU's. Both development time and place-and-route take considerably more time compared to programming/compiling for more traditional architectures. [5]–[7]. High level synthesis tools enable a designer to implement an algorithm in a high level language and to have it compiled and synthesized into hardware. These tools enable faster prototyping and implementation.[8] The latest generation of these tools uses C or variants of this language to enable programmers without a background in HDL design to benefit from the advantages of FPGA accelerators without facing the steep learning curve of learning a HDL such as VHDL or Verilog. For existing HDL designers HLS tools these tools present a reduction in the number of lines of code that are needed to describe the design.[9] These tools enable to shift the focus from low-level implementation details to the development and improvement of the algorithm in a rapid prototyping fashion.[10] HLS tools have a long history dating back to the 1970's but only recently have these tools matured enough to become adopted by industry. These tools present an interesting evolution and a possible paradigm shift in hardware design and prototyping.[11] Two such tools will be used for this master thesis: the Riverside Optimising Compiler for Configurable Computing (ROCCC) and Xilinx Vivado.

3.0.4 Riverside Optimising Compiler for Configurable Computing

ROCCC is a C-to-vhdl compiler which focusses on FPGA based code acceleration. It implements a subset of the C language on which it performs loop analysis techniques to provide increasing throughput with less usage of area. [12] The generated VHDL is independent from FPGA platforms and supports code reuse through the use of modules. ROCCC uses the streaming paradigm, in which data is represented by streams, a data format similar to the way arrays are stored in memory. These streams pass through a set of operations called kernels. This way of representing data makes it possible to express parallelism and is relatively easy mapped to the FPGA hardware. This paradigm removes the need for area-coslty soft-core processors.[13]

This streaming paradigm is also what enables the platform independence of the ROCCC hardware. As long as the data is delivered to the system in the form of a stream it can be used. Another important feature of ROCCC are the so-called smart buffers. These attempt to utilize the data-locality of certain applications to increase the performance. This is done by utilizing intelligent data reuse to minimize the number of off-chip memory acceses.

3.0.5 Xilinx Vivado High Level Synthesis Tool

Vivado High-Level Synthesis is part of the Xilinx Vivado design suite and is the product of the acquisition of AutoESL and the re-branding of their AutoPilot High-Level Synthesis tool. Vivado represents the next evolution of Xilinx tools fitting in their vision of an ”all programmable world”. It allows C, C++ and SystemC code to be synthesized into VHDL or Verilog code. Functional simulation can be done in C, which is a great improvement over the typical VHDL or Verilog simulation. The Vivado tool is based on the Eclipse platform and incorporates the C Development tool (CDT).[15] Due to the recent release of Vivado there is not much academic information to be found about this tool.

Chapter 4

Performance Analysis

4.1 Pragma's influencing the memory architecture implementation

The way memory accesses are implemented are an important factor influencing the performance of an IP-Core. Buffers have a large influence on the operational intensity. Increasing operational intensity moves the implementation from memory bound area of the roofline model to the compute bound area of the roofline model.

In *High Level* programming languages memory gets abstracted to variables and array's. These abstractions need to be translated into something that can be implemented in hardware. For FPGA's this means choosing between *block ram* or registers. The process of translating the memory constructs into the most fitting type of physical memory is controlled by the HLS compiler but can be influenced by using directives or pragma's. Especially the way arrays are translated into hardware is of importance. For this purpose a couple of directives are available.

Resource lets the programmer determine which component will be used to map a certain array to.

Array_Map Maps several smaller arrays to the same memory to decrease the resource consumption.

Array_Partition Determines how a certain array will be partitioned into smaller arrays each using their own memory to avoid the bottleneck of having to perform multiple consecutive reads. This directive also allows to partition an array completely into registers.

Array_Reshape Will rearrange an array so the elements have a larger word width. This improves the performance of the memory while maintaining the same resource consumption.

In systems doing video processing buffers are usually employed to exploit the spatial and temporal data locality. In the example of the TRD there are 2 abstractions implemented: `ap_linebuffer` and `ap_window`.

4.1.1 `ap_linebuffer` Class

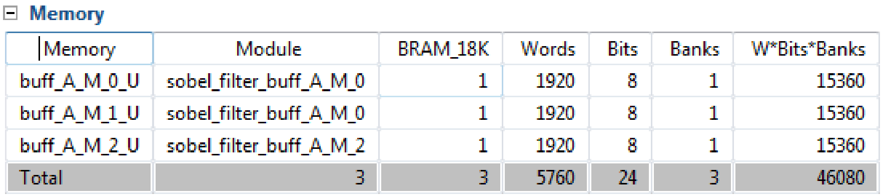
The class `ap_linebuffer` is a generic C++ implementation of the linebuffer described in XAPP793. A linebuffer is described as a multi-dimensional shift-register. A linebuffer needs to be able to be read and written to in the same cycle to maximize performance. The dual port nature of block RAM makes it the ideal component for this abstraction. Because the `ap_linebuffer` class is generic its behavior needs to be defined in the application. The template for the `ap_linebuffer` class is `<typename T, int LROW, int LCOL>`. A type, the number of rows and the number of columns need to be specified. This is done in the `sobel.h` file by the following line:

```
typedef ap_linebuffer<unsigned char, 3, MAX_WIDTH> Y_BUFFER;
```

The parameters of the template are used to determine the size of the only variable of the class, -the array M of type T with LROW rows and LCOL columns.
This array get partitioned by the following directive:

```
#pragma AP ARRAY_PARTITION variable=M dim=1 complete
```

This means that the first dimension, the number of rows, get partitioned into different block RAMs. This is also reported by the vivado HLS tool. buff_A is the line buffer used throughout the implementation.



⊟ Memory

Memory	Module	BRAM_18K	Words	Bits	Banks	W*Bits*Banks
buff_A_M_0_U	sobel_filter_buff_A_M_0	1	1920	8	1	15360
buff_A_M_1_U	sobel_filter_buff_A_M_0	1	1920	8	1	15360
buff_A_M_2_U	sobel_filter_buff_A_M_2	1	1920	8	1	15360
Total	3	3	5760	24	3	46080

Figure 4.1: paritioning of an array in multiple block RAM instances

4.1.2 ap_window Class

The second class used in the application is a generic implementation of the memory window described in XAPP793. It is a combination of shift-registers forming a 2-dimensional data storage element of N pixels centered on a pixel P. Usually these are implemented as flip-flops because they contain relatively few elements who need to be simultaneously available for a calculation. This is achieved through completely partitioning the memory into registers, preventing it from being implemented by a block RAM. The template of `ap_window` is `<typename T, int LROW, int LCOL>`. These parameters are used for the only variable in the class, an array M of type T with LROW rows and LCOL cols. Analog to the linebuffer class the programmer here also needs to define the type, number of rows and number of columns of array M. The array gets paritioned into registers by the following directive:

```
#pragma AP ARRAY_PARTITION variable=M dim=0 complete
```

The `dim=0` means that all dimensions should be partitioned. The `complete` keyword signifies that this partitioning should be done for the whole array.

4.1.3 influence of memory architecture on the operational intensity

This hierarchical structure of the memory influences the operational intensity of the algorithm. The numerator is determined by the number of bytes being processed by the core. Every iteration one value gets read from the external memory and one value gets written. There are 32 bits per pixel, so there are 4 bytes per pixel. this means that with height H and width W there are:

$$4 * 2 * (H * W)$$

bytes being read/written to/from the memory. This is the denominator in the expression of the operational intensity.
The numerator is dependent on the number of pixels being calculated by the core. The core used in the TRD doesn't calculate the pixels on the outer rim of the image and instead uses these as padding.

```
if( row <= 1 || col <= 1 || row > (rows-1) || col > (cols -1)){

                edge.R = edge.G = edge.B = 0;

}
//Sobel operation on the inner portion of the image
else{
```

```
                edge = sobel_operator( ... );
}
```

This branching needs to be taken into consideration for the expression of the computational intensity. The expression for the numerator is given by:

$$(H * W) - [(4 * W) + 4 * (H - 4)]$$

the complete expression is then given by:

$$\frac{(H * W) - [(4 * W) + 4 * (H - 4)]}{4 * 2 * (H * W)}$$

4.2 Pramga's influencing throughput

4.2.1 Original TRD

The original TRD has 3 pragma's applied to it:

- `set_directive_loop_flatten -off "sobel_filter/sobel_filter_label0"`
- `set_directive_dependence -variable &buff_A -type inter -dependent false "sobel_filter/sobel_filter_label0"`
- `set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"`

These all influence the system in a distinct way.

Loop Flattening This directive combines nested loops. This removes the need for the clockcycle needed to enter and leave the loop. It needs to be applied to the inner loop of a set of nested loops. In the TRD loop flattening is explicitly disabled.

Dependence The compiler tries to identify dependencies between calculations or resources. Sometimes this automatic identification of dependencies is too conservative because the compiler doesn't have some information. For this reason the *dependence* directive exists, allowing the programmer to explicitly state that there are or aren't dependencies for a certain variable. There are two types of dependence:

Inter The dependence is between different iterations of the same loop. If the dependence is set to false this will allow the loop to be unrolled

Intra The dependence is inside the iteration. If the dependence is set to be false the compiler will attempt to reorder the operations for the most optimal performance.

In the case of the TRD the inter-dependence of the variable buff_A is set to false.

Pipeline The directive set_directive_pipeline is used to control the pipelining of loops and functions. Each function or loop on which this directive is used can read a new input every N clockcycles. This variable N is called the *Initiation Interval* or II for short. In the case of the TRD pipelining is applied to the inner loop, with an Initiation Interval equal to 0.

Throughput

Given the analysis generated by Vivado HLS presented in table 4.2 it is possible to calculate the throughput of the system. First of all it needs to be noted whether the system satisfies the timing requirements. The analysis gives us an estimated clock period of 4.2 ns with an uncertainty of 0.62 ns placing it well within the bounds of the required 5 ns clock period. The system needs 22 cycles to complete. Of these, there are 2 initialization cycles, 20 cycles to finish the outer loop, of which 19 cycles are the inner loop. The system employs pipelining on the innermost loop, which results in an initiation interval of 1.
N is the number of cycles necessary to calculate one frame:

$$N = \text{init cycles} + \text{outer loop iterations} * (\text{iteration cycles} + \text{inner loop iterations} - 1)$$

These cycles all take a certain time to complete:

$$total\ time = N * cycle\ time$$

The number of frames per second is then given by:

$$FPS = \frac{1}{total\ time}$$

Entering the numbers found in table **??** gives us a value of 95.37 frames per second. Given that HDMI has a 60 Hz refresh rate this system satisfies that constraint.

4.2.2 No pragma's

The original system performance satisfies the real time constraint placed on the system. To study the impact the directives have on the performance of the system all directives were removed from the system. The analysis generated by Vivado HLS is presented in the second column of table **??**. The first observation that can be done is that the system performs the same operation in only 16 clock cycles instead of 22, a decrease of 27%. Because the system has no pipelining the initiation interval is 14 cycles. A new value gets read each iteration.
The number of cycles needed to calculate one frame is given by:

$$N = init\,cycles + outer\,loop\,iterations * (inner\,loop\,iterations * inner\,loop\,cycles)$$

Knowing the number of cycles the throughput can be calculated the same way it is done in section 4.2.1. This gives us a value of 0.47 frames per second, a 203 times decrease in performance.

4.2.3 Loop flatten on

The next effect that can be studies is the effect of turning loop flattening on. This is done with the following directives:

- `set_directive_dependence -variable &buff_A -type inter -dependent false "sobel_filter/sobel_filter_label0"`
- `set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"`
- `set_directive_loop_flatten "sobel_filter/sobel_filter_label0"`

4.2.4 Loop flattening with Initiation Interval 2

	BRAM_18K	**DSP48E**	**FF**	**LUT**
Original directives	3	19	1487	1412
No pragma	5	4	802	1475
loop_flatten_on	3	23	1764	1668
loop_flatten_II_2	3	23	1777	1875
no_dependence	3	19	1487	1412
no_pipeline	5	4	802	1475
only dependence	5	4	786	1534
only loop flattening	5	8	1050	1783
only pipelining	3	23	1851	1849

Table 4.1: Utilization Estimates

	Original directives	No pragma	loop_flatten_on	loop_flatten_II_2	no_dependence
Estimated Clock (ns)	4,2	4,35	5,25	4,2	4,2
Uncertainty (ns)	0,62	0,62	0,62	0,62	0,62
cycle time (ns)	5	5	5,25	5	5
total cycles	22	16	28	29	23
init	2	2	8	8	2
outer loop	20	14	20	21	21
inner loop	19	13	N/A	N/A	20
pipelining outer	no	no	yes	yes	no
pipelining inner	yes	no	N/A	N/A	yes
Initiation Interval	1	14	1	2	1

Table 4.2: Analysis Data

	no_pipeline	**only dependence**	**only loop flattening**	**only pipelining**	**only pipelining II 2**
Estimated Clock (ns)	4,35	4,35	4,35	4,2	4,2
Uncertainty (ns)	0,62	0,62	0,62	0,62	0,62
cycle time (ns)	5	5	5	5	5
total cycles	17	16	22	31	31
init	2	2	8	8	8
outer loop	15	14	14	23	23
inner loop	14	13	N/A	N/A	N/A
pipelining outer	no	no	no	yes	yes
pipelining inner	no	no	no	no	no
Initiation Interval	15	14	14	2	2

Table 4.3: Analysis Data Continued