Lab 5: SoC Design and Optimization

Christos Vasileiou
CE435: Embedded Systems
Department of Electrical and Computer Engineering,
University of Thessaly, Volos, Greece
chrivasileiou@uth.gr

Dimitrios Voulgaris
CE435: Embedded Systems
Department of Electrical and Computer Engineering,
University of Thessaly, Volos, Greece
dvoulgaris@uth.gr

***Abstract*- System performance is a topic that has been extensively analyzed and optimized using conventional techniques. However, given the CPU frequency limitations of the last decades as well as the decay of Moore's Law, innovative architectures have arisen in order to address the issue. FPGAs are a relatively new concept of programmable hardware logic that has been proposed to alleviate exactly this problem. They are to be used isolated as microcontrollers, or in embedded or larger, heterogeneous systems collaboratively with CPU as hardware accelerators. FPGA-based computing systems have been shown to provide superior performance for many application-specific computations in comparison to general-purpose architectures. This laboratory report is aiming in juxtaposing the performance optimization achieved using solely software techniques with a combination of software and hardware ones. A high-performance *Sobel* edge-detector acceleration core has been developed in Vivado HLS tool and compared to its counterpart: an extensively software optimized *Sobel* edge-detector application. Our results, which emphasize on the performance aspect, reveal the dominance of the core over the application.**

I. Introduction

This report has been drawn up having the dual scope of providing a detailed description of the procedure we followed in order to address the requests of Lab 5, while at the same time it is intending in underlining the results that emerged from the various experiments of software and hardware approaches.

$$O_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad O_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Fig. 1: Sobel kernels

The primary aim of this laboratory exercise can be summarized in familiarizing with Vivado HLS tool by experimenting with a relatively code-wise simple, yet demanding image processing algorithm. *Sobel* filter is a well-known image processing operator which receives an input image and transforms it so as its edges are emphasized. Simplifying its operation, we can say that it performs a 2-D spatial gradient measurement giving emphasis on regions with high

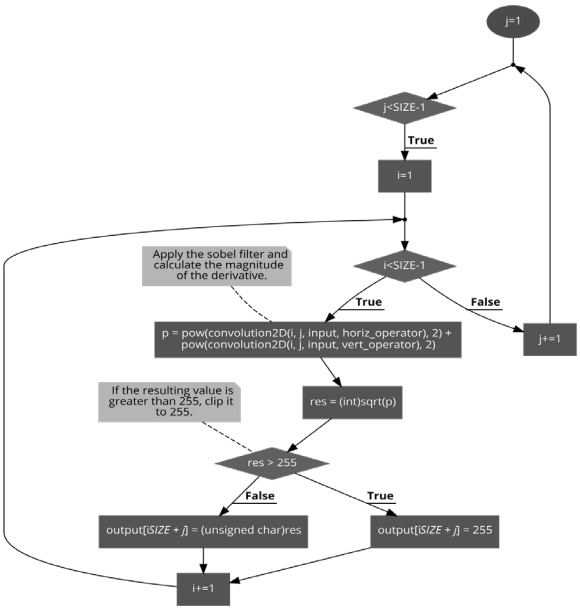

Fig. 2: Sobel function dataflow

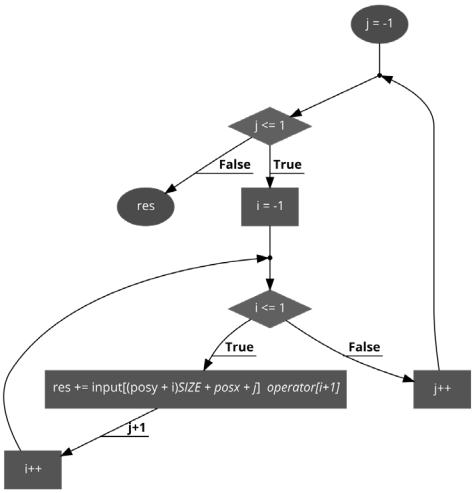

Fig. 3: Convoluton function dataflow

spatial frequency (edges). Typically it calculates the approximate absolute gradient magnitude at each point in an input gray scale image using a pair of 3x3 convolution kernels as shown in Fig. 1.

The basic *Sobel* algorithm consists of two nested loops which calculate the output image pixel by pixel applying the convolution function (Fig. 2). The latter is responsible for performing a discrete convolution between the 9 elements of the *Sobel* filter and the respective elements of the original image as shown in Fig. 3. It is an inherent characteristic of the *Sobel* algorithm that the computation of the parametrical input image pixels is possible only after image padding. We would rather discard these pixels without creating any severe image distortion than burden the CPU, or FPGA with the additional padding procedure. As the analysis continues, special references will be made concerning this matter.

This very C/C++ code has been chosen for the experiment implementation. In particular, the entire procedure is divided in three steps: The first regards the software optimization of the code. Having at our disposal a bibliography brimming with software optimization techniques, we have profiled the code, found its bottlenecks and performance-limiting sections and addressed them combining the appropriate methods. This is to be found in the second part of the report. The second step consists of a hardware implementation of the *Sobel* filter using Vivado HLS. This is a powerful high-level synthesis tool that transforms C/C++ code into HDL such as Verilog. Exploiting Xilinx' Zynq-7000 SoC ZC702 Evaluation Kit that is available in our lab, we have created an IP that applies the filter at an input image deploying specifically designed hardware. This IP is used as accelerator by ARM Cortex-A9, which is the board's processing unit. Their communication is possible through an AXI4-Lite Interface. A stepwise walkthrough shall follow at sections IV and V of this report. Finally, the last step regards the analysis of the results from both implementations, as well as their comparison.

The rest of this report is structured as follows. Section II explains the background information that led to the software implementation. Section III presents our experimental results with regards to the software application. Section IV describes the methodology for the hardware implementation along with the required background. Section V presents the respective results. Section VI compares briefly the results from the previous two implementations.

II. Software Implementation Background

The system that was set up to support these experiments is based on Vivado tool: A Zynq processing unit was used along with the demanded peripherals as well as the drivers that that enabled the transaction between CPU and the SD card of the Kit. The *Vivado Bock Diagram* had no particular peculiarity, since the heart of this implementation is to be found in the *SDK* development suite which is provided along with *Vivado* and at which the C/C++ source code was developed.

Monitoring and analyzing the performance of an algorithm can be a very complicated task, nevertheless, willing to eliminate its complexity we simply made use of the library "*xscutimer.h*" which is provided by the Evaluation Kit. This library eases the programmer by allowing the access of drivers for using the hardwired timers of Cortex-A9. Thus, we added a timer before and after the computational part of the algorithm exacting its computational duration. The rest part of the source code is considered invaluable, yet was not part of the total measured time since its purpose is solely to prepare the input/output files for processing, print user information in the screen etc.

With regards to the timing measurements, in order to achieve the maximum possible quality of result, we performed each execution 10 times and extracted the average execution time. Cold cache misses, system preparation and other factors may interfere in the program execution and alter its behavior. Therefore, every timing reference mentioned in the rest of this report shall refer to the algebraic average of these execution times.

Ensuring the correctness of the algorithm after every optimization, in software or hardware level is of essence. Given that visual verification is not enough, we created a "golden" output file which we considered the desirable result. This file is co-computed with the under-test file in order to extract the PSNR value. This value should tend to infinity in order the result to be completely correct and the files to be identical. Let it be noted that neither this computation contributes to the total execution time.

The aforementioned *Sobel* code, although particularly short, it offers a wide variety of changes that allow its performance boost. The final applied optimizations are: *compiler optimization, loop unrolling, loop interchange, function inlining, loop invariant code motion, common sub-expression elimination, strength reduction and algorithm-specific optimizations*.

A. Compiler optimization

Compilers conceal so much complexity behind their operation and are, among others, capable of performing independent code optimizations. *SDK* gives the flexibility of choosing both compiler and also setting

```
for (i=1; i<SIZE-1; i+=1) {
 for (j=1; j<SIZE-1; j+=1 ) {
   x1=0;
   x2=0;

   x1=-input[i*SIZE-SIZE+j-1]
      -input[i*SIZE+j-1]*2
      -input[i*SIZE+SIZE+j-1]
      +input[i*SIZE-SIZE+j+1]
      +input[i*SIZE+j+1]*2
      +input[i*SIZE+SIZE+j+1];

   x2= input[i*SIZE-SIZE+j-1]
      -input[i*SIZE+SIZE+j-1]
      +input[i*SIZE-SIZE+j]*2
      -input[i*SIZE+SIZE+j]*2
      +input[i*SIZE-SIZE+j+1]
      -input[i*SIZE+SIZE+j+1];

   p = (x1*x1) + (x2*x2);
   res = (int)sqrt(p);
   if (res > 255)
    output[i*SIZE+j]=255;
   else
    output[i*SIZE+j]=res
 }
}
```

Fig. 4: Common sub-expressions such as "*i*SIZE*" are calculated multiple times inside a nested loop. "*math.h*" power function is substituted by a simple multiplication while the *sobel* kernels are broken down to their components.

```
for (i=1,k=0;i<SIZE-1;i+=1) {
 k =k+size;
 for (j=1; j<SIZE-1; j+=1 ) {
   x1=0;
   x2=0;
   m=k-SIZE+j;
   x1=-input[m-1]
      -(input[i*SIZE+j-1]<<1)
      -input[k+SIZE+j-1]
      +input[m+1]
      +(input[i*SIZE+j+1]<<1)
      +input[k+SIZE+j+1];

   x2= input[m-1]
      -input[k+SIZE+j-1]
      +input[m]*2
      -(input[k+SIZE+j]<<1)
      +input[m+1]
      -input[k+SIZE+j+1];

   p = (x1*x1) + (x2*x2);
   res = (int)sqrt(p);
   if (res > 255)
    output[k+j]=255;
   else
    output[k+j]=res
 }
}
```

Fig. 5: Computations that occur multiple times inside the nested loop are eliminated. The "expensive" operation of multiplication is replaced by its addition equivalent or by shift circuits while *sobel* kernels are broken down to their components.

compiler flags that determine the desirable optimization degree. In our case, we opted for *gcc* and set the optimizations to the maximum level possible: *-O3*. This decision came as a result of experimentation with all available optimization levels.

B. Loop Unrolling

This technique is particularly famous and indeed sometimes performed by the compiler indirectly. However, explicitly unrolling a loop contributes in increasing the instruction window, and therefore the compiler can schedule instructions selecting from a larger set. Moreover, eliminating loops alleviate the system from extra "book-keeping" that is demanded when a loop is called.

In our case, we chose, as a first step, to completely unroll the loop residing in "*convolution2D*" function. The finite and relatively small number of iterations (9 iterations) allows that the loop be completely unrolled without causing excessive register usage or code burst.

C. Loop Interchange

A more careful look of the code and particularly of the way the output is calculated and written back in memory evinces that, given the large image used as input, every mechanism assisting in more effective memory usage is cancelled. The nested loop is constructed in such a way that elements are accessed in a row-by-row manner, disturbing the optimal memory access. In order to invert this fact what is demanded is to compute elements row-wise (column-by-column), or, in other words, to interchange the variables used to iterate the loops. So, *j* becomes *i* and *i* becomes *j*.

D. Function Inlining

A profound way to minimize the execution time by lessening the application's need for logistics is to inline frequently-called and relatively small functions. Data copying and stack transferring which are demanded when a function is called can add a severe overhead to the total execution time, especially when the function call frequency is high.

In our experiment, this is the case of the function "*convolution2D*". Inlining it into the main body of the "*sobel*" function eliminates the execution time, as it will be shown later, by removing the extra overhead that a function call adds.

E. Strength Reduction

Being a fundamentally mathematical algorithm, *sobel* filter uses mathematical expressions and formulas that are defined in "*math.h*" library. Some of these formulas such as "*pow*" or "*i*SIZE*" can be substituted by their simpler and computationally "lighter" mathematical equivalencies. Others like multiplications in which the multiplier is power of 2 can be performed by a shifter circuit, which is much faster and spatially smaller. Others, however, such as square root can only be approximated by computational (iterative or not) methods that may be faster, yet inaccurate. Thus, such a substitution was not realized, rather was left as a potential future extension.

F. Common Sub-expression Elimination

When loops constitute the main part of an algorithm, it is usually the case that many same computations are repeated in their bodys. Every loop of the *sobel* computation refers to a close neighborhood of pixels, the position of which has to be calculated for each one of these. The value of one output pixel is function the values of the corresponding input pixel, as well as of the latter's circumferential ones. Calculations aiming in determining the pixel position are repeated and can subsequently be grouped instead of being performed for every element independently.

G. Algorithm-specific Optimizations

The basic idea behind *sobel* filter is the convolution between an input image and the *sobel kernels*. These kernels are very specific and, since our application is exclusively optimized to perform a particular task, they can be broken down to their components. One entire row of each kernel consists solely of zeros; therefore its multiplication with the input elements has no contribution to the final result. Furthermore there are several unit elements in each kernel (4 ones in each kernel, so 8 in totals). In both previous cases an arithmetic unit would have been required to calculate a result that is either beforehand known as 0 or can be computed with much less computational effort.

Disassembling filter kernels and transferring their contribution to the main function by performing simple arithmetic operations shall prove beneficial to the overall performance of the application.

In Fig.4 we pinpoint some common sub-expressions that occur when kernels of Fig. 1 are disassembled and the mathematical operations are profound. In Fig. 5 we lay out the final code version of the main computational part, after the total optimization set that was applied.

III. Software Implementation Results

In this section we provide the results that were obtained during our software development. We focused our attention in the application completion time; however, willing to achieve a stable execution, we monitored the standard deviation among different program executions and kept track of the influence that each optimization has upon this value.

In Fig.6 we present the overall decrease achieved by the optimizations. The sharpest plot decline corresponds to the execution time saved when *compiler optimizations* are applied. Algorithmic performance presents an increase of more than 94% due to these complex improvements that compiler adds.

Zooming in the optimizations applied by us, in Fig.7, we observe that *loop unrolling*, although theoretically beneficial, little has to offer since the execution time remains roughly the same. This is not the case, nevertheless. As stated before, forcing the compiler to perform optimizations at the highest possible level, results in indirect code manipulations that are not visible to the programmer. Keeping that in mind it seems that unrolling the loop that resides in "*convolution2D*" function is of these manipulations; thus the time improvement that this technique offers is hidden in the total time improvement that *–O3* optimization level achieves.

On the other hand, *loop interchange* exhibits a rather apparent improvement in execution time. This can be justified by gaining an insight in the system that the experiments were run: Cortex A9 is equipped with a 32KB L1 Data cache and a 1MB L2 Data cache. Provided that we use images sized 1024x1024 (i.e. 1MB) every accessed address is 1023 bytes apart from the previously accessed one. In the best case scenario we would assume the entire image reside in L2 Data cache, therefore avoiding main memory accesses. However this is not the case as output image is also accessed and consequently occupying a part of the cache.

So, in order to compute one column of the output image, not only will there be no L1 hit, but also data will be fetched from main memory multiple times. And this shall be the case for the computation of every column.

L1 miss rate and even more L2 miss rate are more than performance limiting. Improving cache access pattern can utilize cache techniques such as prefetching as well as minimize the overall cache miss rate improving the performance by more than 10%.

Function inlining combined with the rest of the optimizations offers a significant reduction in execution time. Using an input image of 1MB requires that the helping computational function be called more than

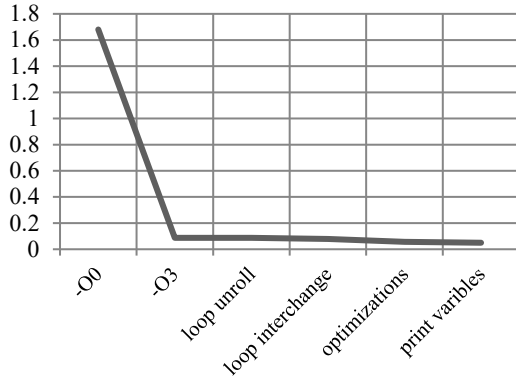

Fig. 6: Total execution time where all applied optimizations are visible.

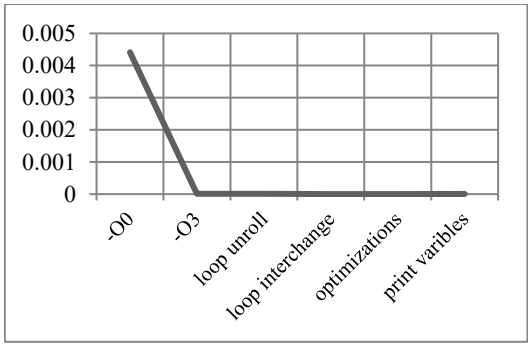

Fig. 8: Standard deviation

1,004,000 times, one for every pixel of the output image. Taking the burden of the administrative cost that accompanies a function call is visible in the execution time. This code enhancement combined with the rest code alterations described before offer a total boost of ~26% in the overall application performance.

What is of essence since we could not explain it, nonetheless included it in our final code version is the performance improvement when a "*printf*" is added. In particular, printing the values of two variables at the non-measured part of the code seems to boost the overall performance. As shown in Fig. 7 this increase in performance is around 13%.

Standard deviation is almost annihilated when compiler optimizations are applied (Fig. 8), while the additional code modifications seem to reduce it even more so that the final code version achieves a stable execution pattern.

IV. Hardware Implementation Methodology

FPGAs are consisted of un-configured hardware modules that allow the user to program them in order to perform a very specific operation. In our case, this operation is called *sobel* filter and was realizable thanks to the tool ecosystem that Xilinx provides. The entire procedure can be divided in 3 steps. As first step, Vivado HLS was used aiding in the transition from C/C++ code

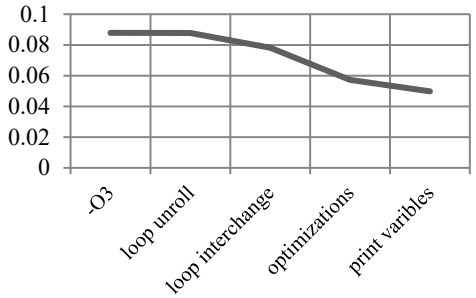

Fig. 7: Total execution time where some applied optimizations are visible.

to Verilog one. This tool offers the required sub-tools that allow packaging the new hardware into an ip making portable and available for usage. The second step includes creating and programming (in C) the peripheral processing system that surrounds the ip. In the third and final one we return to Vivado in order to perform code optimizations that result in more efficient hardware and thus in increased system performance.

A. IP Creation and Interconnection

The code that is provided by the software implementation cannot be used directly as it is in the Vivado HLS environment. On the contrary it has to be divided in two parts: one that shall be the future hardware ip and one that will be used as testbench, in first place, and as the main code that shall later control the CPU. We have chosen only the main computational function to be transferred in hardware, discarding the PSNR computation. Moreover, complying to the software algorithm, we opted for omitting the circumferential input pixels rather than padding the input image. This technique can be a future program extension.