Accelerating Image Processing Algorithms for the RADIO Project's Assistant Robot System

Fynn Schwiegelshohn, Muhammed Al Kadi, Philipp Wehner, Philipp Smoluk, Michael Hübner, Diana Göhringer

ESIT - Chair for Embedded Systems of the Information Technology
MCA - Application-Specific Multi-Core Architectures Research Group
Ruhr University Bochum
Universitätsstraße 150
44801 Bochum
fynn.schwiegelshohn@rub.de; muhammed.alkadi@rub.de; philipp.wehner@rub.de
philipp.smoluk@rub.de; michael.huebner@rub.de; diana.goehringer@rub.de

Abstract: This paper presents an acceleration method for an image processing algorithm from the RADIO EU project. The algorithm was profiled and the compute intensive tasks were then mapped on programmable hardware. The accelerator design was done with Vivado HLS. The accelerator implements the image processing algorithm as OpenCL kernel and optimizes its performance based on the analysis of the generated hardware. However, the OpenCL code does not reach acceptable speedups when implemented as is. For better performance , several optimizations steps have to be exceuted. The results show that software generated as hardware on programmable hardware can reach a speedup of up to 1.32. For this, further knowledge of the underlying hardware is required.

1 Introduction

In the EU project RADIO [KAV+16], a robot assistant is used to monitor the health of an end user with the help of image data. The images generated by the robot, should provide indications about the end users health, mental capabilities and behavior. One aspect, which needs to be monitored within the RADIO project, is the process of the end user standing up and getting out of bed [SWR+15]. This algorithm has already been developed in software. Because several of these algorithms are running continuously on the robots processing platform, the resource usage is very high and timing constraints might not be met. Therefore, the tasks which are compute intensive can be accelerated with the help of programmable hardware. The RADIO robot platform has two main processing platforms. The first is an Intel NUC, the second is an Avnet PicoZed [Inc]. The NUC is responsible for controlling the base platforms sensors and actuators. Therefore, it is directly connected to the TurtleBot2 base platform via USB. The Avnet PicoZed serves as accelerator platform to reduce the computation load of the Intel NUC. Additional devices that are placed on the robot are an Asus Xtion Pro camera and a Hokuyo laser scanner. These two devices can either be connected to the Intel NUC or to the Avnet PicoZed. The setup of the robot platform is depicted in Figure 1. The image processing software should be accelerated by the Avnet Picozed. In the course of generating the accelerator for this algorithm, the use of parallelism should also be considered.

Section 2 introduces the image processing algorithm that is accelerated in this paper. Then in section 3, the profiling results of the algorithm are introduced in order to determine which components of the algorithm should be accelerated. Section 4 introduces the OpenCL code implementation of the algorithm while explaining how to achieve the best possible accelerator results. The speedup and evaluation results are presented in section 5. Finally, section 6 concludes this paper.

2 RADIOs Image Processing Algorithm

The algorithm for monitoring the state of the patient is based on center of gravity calculation and can be divided into 4 to 5 parts, depending on whether mark ups are activated or not.

1. Reading of the most recent image frame.

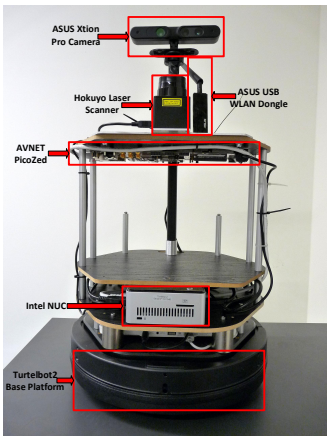

Figure 1: The RADIO robot platform and all its connected devices.

2. Detection of movement.
3. Calculation of center of gravity.
4. Evaluation of center of gravity.
5. Drawing mark ups.

2.1 Reading image frame

The image data is provided by the Asus Xtion Pro camera of the RADIO robot platform. The image data is sent via USB directly to the NUC which publishes the received frames via its robot operating system to the Avnet Picozed where it is processed. The image frame is then read by the software and saved to a 3-dimensional array. The first two dimensions indicate the pixels positions whereas the third dimension stores the color values of the RGB color channel. Each color is coded with 8 bit, resulting 24 bit color payload. Given that the Asus Xtion Pro camera provides images with the size of 640×480 pixels, the resulting array size is $640 \times 480 \cdot 3 = 921600$ or 900 KiB.

2.2 Detection of Movement

The algorithm loads to subsequent frames and compares both image frames with each other in order to detect changes or movement within the two image frames. In order to reduce the impact of small movements of the camera or image noise, the comparison does not only take place on the subtracted image, but rather on blocks of pixels with the size 10×10. Within these blocks the mean value of all subtracted color channels is calculated. If this value exceeds a certain threshold, the respective block is flagged as active to show that a change has occurred. This can be seen in Figure 2. While the person is moving out of the bed, the pixel blocks that detect movement are highlighted in red.

2.3 Center of Gravity Calculation

After all blocks have either been detected as active or inactive, the center of gravity can be calculated. In this case, the center of gravity is calculated through the mean value the positions of all active blocks. In Figure 2, this can be seen by the bright green sphere. Because the active blocks are positioned in the middle of the image and in the lower right corner of the image, the center of gravity lies directly in the between the detected hotspots of movement.

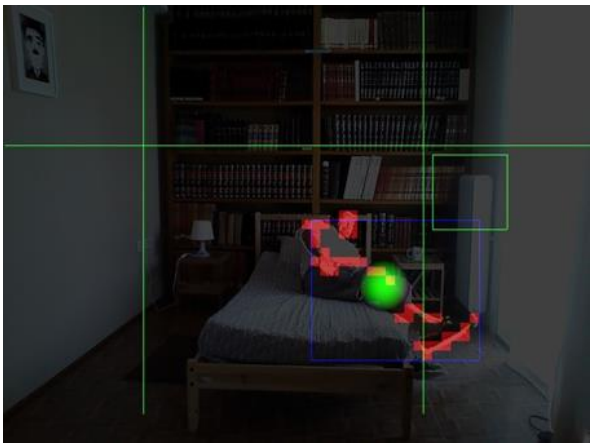

Figure 2: The image processing algorithm when detecting movement.

2.4 Center of Gravity Evaluation

Now that the position of the center of gravity has been determined, its position needs to be analyzed and interpreted. If the y coordinate of the center of gravity exceeds a certain threshold, the algorithm assumes that the observed person has gotten out of bed. Several of these thresholds exist. Some are shown in Figure 2. These are the green lines in the image. All lines symbolize a threshold which indicates a new state of the monitored person.

2.5 Markup Drawing

In order to optimize and help debug the algorithm, markups can be drawn into the image. When drawing markups, all color values which differ more than the value 40 compared to the prior frame are set to 70. If the pixels differ less than 40, the color values are quartered. Additionally as already seen in Figure 2, the pixel within an active block will be colored red. This is done by adding the value 128 to the red channel. This calculation is saturated, meaning that the resulting value never exceeds 255.

3 Profiling Results

In order to optimally accelerate the image processing algorithm with programmable hardware, the compute intensive components need to be identified. This is done with the help of profiling. The Picozed is a System on Chip with an dual core ARM Cortex A9 processor and integrated programmable hardware. The image processing algorithm is first executed on the ARM processor. There, the performance of the algorithm is determined and the potential hardware accelerated components are identified. From the software side, the algorithm consists of several subblocks which are further analyzed during the profiling. These are described in Table 1. For each profiling run, the algorithm is executed 20 times in order to mitigate the impact of outliers. The used profiler is gprof [GKM04] and the results are presented in Figure 3(a) for the algorithm with markups and in Figure 3(b) for the algorithm without markups. As can be seen in both Figures, the algorithm spends most of total processing time in the *checkBoxes* function. In the case with activated markups, the amount is 61.70% and 51.20% without activated markups. Because the *copyToImageData* function is only requires for debug purposes, this function will not be implemented in the final algorithm design. Therefore, the timing value for this function is ignored. In the case of activated markups, all the data required for the *annotateBoxes* function is generated by the *checkBoxes* function. Because both function are executed sequentially, it is possible to generate hardware accelerators for both functions.

Table 1: Attributes of the three numerical methods in comparison

Function name	Task
copyToRGB	Copy the received image data to a 3-dimensional array
checkBoxes	Calculate the mean value of the color value differences over the last 2 frames and indicate the active blocks.
annotateBoxes	If markups are activated, indicate the pixel changes and highlight the active blocks.
process_function	Calculates the center of gravity and determines its position. This is the function that calls *checkBoxes* and *annotateBoxes*.
copyToImageData	Copy the processed image data from the 3-dimensional array to a ROS compatible array for debug purposes.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
61.70	2.32	2.32	61440	0.04	0.04	checkBoxes
17.02	2.96	0.64	20	32.00	32.00	copyToImageData
16.49	3.58	0.62	20	31.00	31.00	copyToRGB
3.72	3.72	0.14	13199	0.01	0.01	annotateBoxes
1.06	3.76	0.04	20	2.00	125.00	process_function

(a) with activated markups

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
51.29	1.39	1.39	61440	0.02	0.02	checkBoxes
26.57	2.11	0.72	20	36.00	36.00	copyToImageData
21.03	2.68	0.57	20	28.50	28.50	copyToRGB
0.74	2.70	0.02	20	1.00	70.50	process function

(b) without activated markups

Figure 3: Profiling results of the algorithm

4 Hardware Accelerator Design

In order to efficiently switch between the algorithm with and without markup functionality, two OpenCL kernels are designed. This enables an efficient implementation of only one query in order to determine which kernel version will be executed. OpenCL kernels consist of workgroups and workitems. In this case, a workgroup stands for one pixel block and a workitem stands for one pixel. The designed kernel will then be called $640 \times 480 = 307200$ times for each pixel pair. The first step is to calculate the difference of all color vlaues of each pixel pair in a workgroup. If the differential value exceeds the value 40, the pixel value is set to 70, otherwise the value is divided by 4. As soon as each workitem of a workgroup completes the differential calculation, the first workitem of the workgroup will calculate the mean value of all workitems. The mean value is then saved to an external array which is accesible by the CPU for further processing. If the mean value exceeds the threshold vlaue of 30, the block will be highlighted in red. Figure 4 shows the kernel implementation as schematic and as pseudocode.

The initial version of the OpenCL code can be generated with 100 MHz. Figure 5(a) shows the resource requirements of the initial hardware version. This core is compared to a software implementation on the dual core processor of the Picozed. The execution time of the algorithm on software takes approximately 17547 μs. The generated hardware requires 88404 μs, meaning the hardware accelerator requires $88404\mu s \cdot 100MHz = 8840400$ cycles to execute the algorithm. This results in a speedup of 0.2. In order to achieve a accelerator which actually accelerates the image processing algorithm, further optimization steps have to be executed.

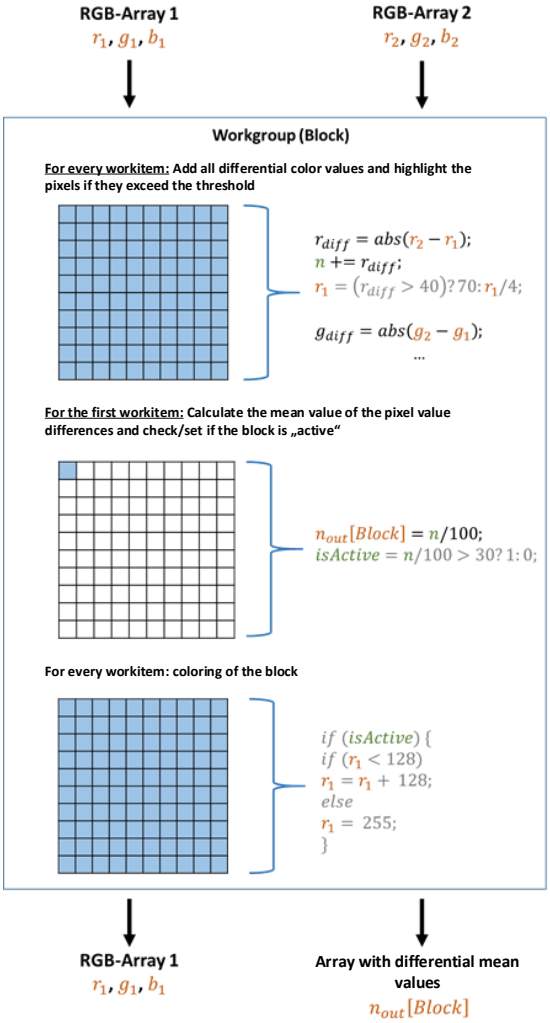

Figure 4: Schematic view of the kernel design annotated with pseudocode

The first optimization steps is to efficiently let the accelerator read the image data from the DDR memory. This is done with the command *async_work_group_copy*. This command transmits a user defined number of sequential bytes from memory via a burstmode to the accelerator. The transmission of one frame is executed stepwise in order to reduce the resource usage of the BRAM on the programmable hardware. Because one image always lies sequentially in memory, only one transmission command per frame is required. After this step, the estimate cycles to complete the algorithm are in a range from $4729607 - 5712647$ cycles, which means a performance improvement of $36\% - 46\%$ compared to the initial implementation. This performance improvement however comes at the cost of an increased resource utilization as can be seen in Figure 5(b). Here, the number of used BRAM blocks has increased from 2 to 74 while all other resource remain almost constant.

Because the number of required BRAMs is very high, the memory requirements of the accelerator are reduced in the second optimization step. Currently, every color value is transmitted as a 4 Byte value to the BRAMs although a 1 Byte value would suffice. Therefore, all three color values are stores in one 4 Byte value on the software side and then transmitted to the accelerator. This reduces the data transmission by 2/3 from 14535 cycles to 4935 cycles. By performing this optimization, the performance of the accelerator is increased while also reducing the resource utilization. This is shown in Figure 5(c). The number of BRAMs is reduced from 74 to 42 and the LUT resource utilization is reduced by 2% compared to the first optimization. The estimated cycle number is also further reduced to $2272007 - 2947847$ cycles which is an performance improvement of 52% compared to the first optimization step.

Because image processing algorithms perform many operations on each pixel individually, these operation are executed in a loop. These loop can be parallelized on hardware. Parallelizing loop can be done through loop pipelining or through loop unrolling. While loop pipelining reuses the already available components

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1827
FIFO	-	-	-	-
Instance	2	4	662	812
Memory	-	-	-	-
Multiplexer	-	-	-	2599
Register	-	-	2160	-
Total	2	4	2822	5238
Available	280	220	106400	53200
Utilization (%)	~0	1	2	9

(a) Resource utilization of the initial OpenCL kernel

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1591
FIFO	-	-	-	-
Instance	2	4	662	812
Memory	72	-	0	0
Multiplexer	-	-	-	1340
Register	-	-	1786	150
Total	74	4	2448	3893
Available	280	220	106400	53200
Utilization (%)	26	1	2	7

(b) Resource utilization of the OpenCL kernel after optimizing the data access.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	691
FIFO	-	-	-	-
Instance	2	4	662	812
Memory	40	-	0	0
Multiplexer	-	-	-	1245
Register	-	-	1383	140
Total	42	4	2045	2888
Available	280	220	106400	53200
Utilization (%)	15	1	1	5

(c) Resource utilization of the OpenCL kernel after optimizing memory requirements.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	2142
FIFO	-	-	-	-
Instance	2	-	662	812
Memory	36	-	0	0
Multiplexer	-	-	-	1344
Register	-	-	1927	140
Total	38	1	2589	4438
Available	280	220	106400	53200
Utilization (%)	13	~0	2	8

(d) Resource utilization of the OpenCL kernel after optimizing loop execution.

Figure 5: Resource utilizations of the different optimization steps.

for parallelization, loop unrolling requires separate resources in order to increase the degree of parallelism. Therefore, loop pipelining requires less additional resources than loop unrolling. The algorithm has 5 loops that can benefit from either loop unrolling or loop pipelining. In the case of this algorithm, no performance difference is detected when using loop unrolling compared to loop pipelining. Because loop pipelining requires less hardware resources, loop pipelining is used for all 5 loops. Figure 5(d) shows the resource utilization when employing loop pipelining for the algorithm. Through loop pipelining, the resource requirements of the BRAMs are reduced even further from 42 to 38. The number of DSP blocks is also reduced from 4 to 1 and the FFs are slightly increased as well as the LUT resource usage. This optimization further increased the performance compared to the last optimization step, leading to cycle number of $1273607 - 1586951$ which is an acceleration of $44\% - 46\%$.

After these three optimization steps, the accelerator is again compared to the software implementation of the algorithm.

5 Evaluation and Results

In order to evaluate the performance of the accelerator on the real hardware, the accelerator must first be implemented on the PicoZed platform. THis is done with the Vivado tool provided by Xilinx. The accelerator must be connected to the processing system in order to receive the image data from the DDR memory. The architecture design is shown in Figure 6. It can be seen, that the processing system and the accelerator are not directly connected to each other. The accelerator is connected to an AXI peripheral interconnect which is connected to the processing system via the AXI4 protocol. This interconnect enables the processing system to control the the accelerator by writing to specified register to start, stop, or initialize the accelerator. The second port of the accelerator is connected to an AXI memory interconnect which enables the accelerator to access the DDR memory to receive and transmit data.

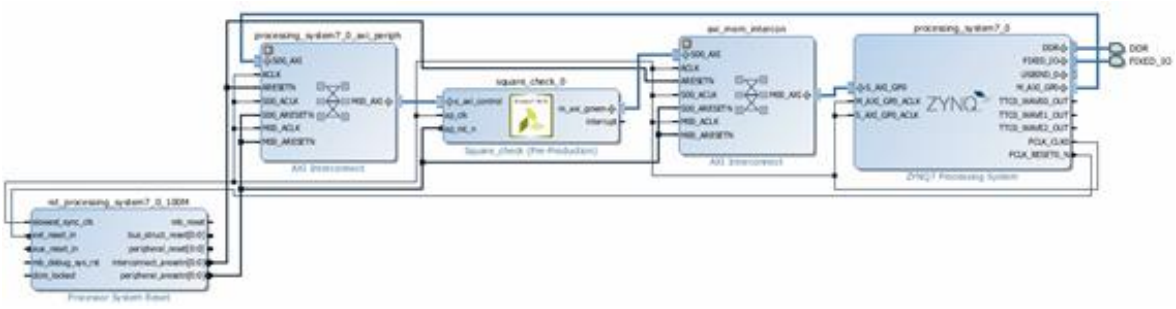

Figure 6: Architecture design of the accelerator connected to the processing system.

Table 2: Measured execution times of each optimization step and of the software implementation

Measurement platform	Execution time	Speedup
Software (ARM)	$17547\mu s$	1
Initial Implementation	$88404\mu s$	0.2
First optimization	$48687\mu s$	0.37
Second optimization	$23401\mu s$	0.75
Third optimization	$13290\mu s$	1.32

Table 2 shows the execution times of the different implementation versions. For all implementations, the clock frequency of 100 MHz is used. The ARM processor is running at 666 MHz. It can be seen that the initial and up until the second optimization hardware version, the software version outperforms the hardware implementation. This changes in the third optimization where the hardware implementation reaches a speedup of 1.32 compared to the software version. All hardware implementations can further increase their performance compared to the software implementation by increasing the clock frequency.

6 Concluding Remarks

In this paper, a image processing algorithm is analyzed and implemented as OpenCL kernel on programmable hardware. While the initial version of the generated hardware reduced the performance of the image processing algorithm compared to the software implementation, this paper shows that with more knowledge of the hardware generation toolflow, further performance increase can be achieved without changing the source code for the underlying hardware generation. This leads to further promising possibilities, enabling software designers to generate and analyze their own hardware accelerators effectively reducing the design gap between software and hardware designers.

References

[GKM04] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004.

[Inc] Xilinx Inc. ZYNQ 7000 All Programmable SoC. Technical report.

[KAV+16] G. Keramidas, C. Antonopoulos, N. S. Voros, F. Schwiegelshohn, P. Wehner, J. Rettkowski, D. Ghringer, M. Hbner, S. Konstantopoulos, T. Giannakopoulos, V. Karkaletsis, and V. Mariatos. Computation and communication challenges to deploy robots in assisted living environments. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 888–893, March 2016.

[SWR+15] F. Schwiegelshohn, P. Wehner, J. Rettkowski, D. Ghringer, M. Hbner, G. Keramidas, C. Antonopoulos, and N. S. Voros. A Holistic Approach for Advancing Robots in Ambient Assisted Living Environments. In

Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on, pages 140–147, Oct 2015.