- ORIGINAL ARTICLE -

Optimization of a Line Detection Algorithm for Autonomous Vehicles on a RISC-V with Accelerator

Optimización de un Algoritmo de Detección de Líneas para Vehículos Autónomos en un RISC-V con Acelerador

María José Belda[1], Katzalin Olcoz[1], Fernando Castro[1], and Francisco Tirado[1]

[1]*Complutense University of Madrid, Madrid 28040, España*
{mbelda.katzalin.fcastror.ptirado}@ucm.es

Abstract

In recent years, autonomous vehicles have attracted the attention of many research groups, both in academia and business, including researchers from leading companies such as Google, Uber and Tesla. This type of vehicles are equipped with systems that are subject to very strict requirements, essentially aimed at performing safe operations –both for potential passengers and pedestrians– as well as carrying out the processing needed for decision making in real time. In many instances, general-purpose processors alone cannot ensure that these safety, reliability and real-time requirements are met, so it is common to implement heterogeneous systems by including accelerators. This paper explores the acceleration of a line detection application in the autonomous car environment using a heterogeneous system consisting of a general-purpose RISC-V core and a domain-specific accelerator. In particular, the application is analyzed to identify the most computationally intensive parts of the code and it is adapted accordingly for more efficient processing. Furthermore, the code is executed on the aforementioned hardware platform to verify that the execution effectively meets the existing requirements in autonomous vehicles, experiencing a 3.7x speedup with respect to running without accelerator.

Keywords: Autonomous vehicles, Firesim, Image processing, Matrix accelerator, RISC-V

Resumen

En los últimos años los vehículos autónomos están atrayendo la atención de muchos grupos de investigación, tanto del ámbito académico como del empresarial, entre los que se incluyen investigadores pertenecientes a empresas punteras como Google, Uber o Tesla. Los sistemas de los que están dotados este tipo de vehículos están sometidos a requisitos muy estrictos relacionados esencialmente con la realización de operaciones seguras, tanto para los potenciales pasajeros como para los peatones, así como con que el procesamiento necesario para la toma de decisiones se realice en tiempo real. En muchas ocasiones, los procesadores de propósito general no pueden por sí solos garantizar el cumplimiento de estos requisitos de seguridad, fiabilidad y tiempo real, por lo que es común implementar sistemas heterogéneos mediante la inclusión de aceleradores. En este artículo se explora la aceleración de una aplicación de detección de líneas en el entorno de vehículos autónomos utilizando para ello un sistema heterogéneo formado por un core RISC-V de propósito general y un acelerador de dominio específico. En particular, se analiza dicha aplicación para identificar las partes del código más costosas computacionalmente y se adapta el código para un procesamiento más eficiente. Además, se ejecuta dicho código en la mencionada plataforma hardware y se comprueba que su procesamiento efectivamente cumple con los requisitos presentes en los vehículos autónomos, experimentando una reducción de 3.7x en su tiempo de ejecución con respecto a su ejecución sin acelerador.

Palabras claves: Vehículos autónomos, Firesim, Procesamiento de imágenes, Acelerador de matrices, RISC-V

1 Introduction

In the technological era in which we live, we every day strive to make all the usual tasks as automatic as possible in order to gain free time. In addition, we try to achieve scenarios that are impossible right now, such as smarter power grids, fully autonomous vehicles or smart cities. This is why the Internet of Things (IoT) arises, as we need new technologies to design these systems. Most of them are on-board systems, so they need to get a trade-off between power consumption and delivered performance. In particular, in this work we focus on autonomous vehicles.

Autonomous driving systems aim to enable vehicles to drive on the road without human intervention [1, 2, 3]. Therefore, these systems must guarantee the safety and integrity of the vehicle, for which they must take a series of decisions in real time, including

moving the steering wheel to ensure that the correct trajectory is followed, detecting obstacles in the path (pedestrians, animals, objects...), activating the braking mechanism when necessary and others. For this purpose, it is essential that the vehicle has a camera that records images of the route and processes them in real time to ensure the correct and safe operation of the vehicle. This image processing requires considerable computing power, but at the same time, when talking about on-board systems, it is essential to keep energy consumption at low levels so the vehicle does not loose autonomy [4].

For these reasons, autonomous vehicles require onboard automatic systems to process the recorded images that allow certain operations such as line and edge detection. Currently, the most widely used algorithms for this type of processing require high performance and their basic kernel is matrix and vector multiplication. It is therefore highly desirable that this type of algorithms could be executed in one of the many domain specific accelerators that have emerged in recent years.

In this paper we propose to accelerate a line detection application employed in autonomous cars by using different heterogeneous systems made up of a general-purpose RISC-V core working at low frequency and a domain-specific accelerator. For this purpose, the application is deeply analyzed in order to identify the computationally intensive parts of the code and adapted consequently for a more efficient processing. As it will be explained in Section 3, the hardware platform used in this work includes, on the one hand, a general-purpose BOOM processor, which is an out-of-order RISC-V core [5], and on the other hand, the Gemmini [6] accelerator, specifically designed for matrix multiplication. This platform was chosen because the RISC-V architecture, in addition to being open source, allows the integration of accelerators and their potential adaptation in a very simple way. Furthermore, the RISC-V instruction set architecture (ISA) is highly modular, allowing to choose exactly the functionalities needed, which is especially useful in IoT environments.

This paper leverages two image processing algorithms: 1) the Canny algorithm for edge detection of an image, and 2) the Hough transform, oriented to find imperfect instances of objects within a certain class of shapes by means of a voting procedure. In Section 4 we perform a detailed analysis of both algorithms codes, in order to identify the computational load of the different functions included in these programs, as well as the available parallelism. Moreover, we schedule some functions to run on the accelerator, while the rest of the algorithm is executed on the processor, aimed to optimize the total execution time and consequently to meet the strict requirements of performance, consumption and safety imposed by autonomous vehicles. The experimental evaluation carried out in Section 5 reports a speedup of 3.7x when executing these algorithms with respect to the baseline where no accelerator is employed. Finally, Section 6 concludes the paper.

2 Basic notions and state of the art

In this section we explain some basic notions related to autonomous vehicles. We also provide details on the RISC-V-based development environment that we employ, including the tools used that make it possible the evaluation of the proposal presented in this paper.

2.1 Autonomous vehicles

Autonomous vehicles are equipped with several sensors, as shown in Fig. 1, including video cameras, which are responsible for obtaining the data that serve as input to the processing system. The purpose of this data processing is to recognize the environment which the vehicle is driving through, and as a result, to make the appropriate decisions at any time, so as to ensure that the vehicle can reach its destination efficiently and safely. In this aspect, autonomous vehicles have levels of driving automation from 0 (No automation) to 5 (Full automation), as explained in [7]. In the first levels, from 0 to 2, the vehicle has very little capacity to act (in level 2 it can only perform steering and acceleration) and all the responsibility lies on the driver. In contrast, the automatic system monitors the driving environment in levels 3 to 5, being this last one the ideal scenario in which the vehicle is completely autonomous, even not providing controls for the driver. So, there is a gap between levels 2 and 3. Between these levels there is also a technological gap, since generating hardware and software capable of monitoring the environment in real time becomes significantly difficult. However, this gap is progressively disappearing and this work aims to contribute to this.

Notably, certain safety decisions are related to the correct recognition of the trajectory to be followed by the vehicle, based on the images recorded by the camera. In addition to allowing the car to follow the correct route, this functionality also involves restricting the likelihood of an accident. For this purpose, computer vision algorithms are commonly used in these processing systems [9, 10] and, in particular, quite approaches use Canny algorithm to detect edges combined with the Hough transform to detect road lines [11, 12]. Therefore, in this paper we focus on improving the performance of these algorithms which are the basis of lane detection. The problem with these algorithms is their very high computational cost. In addition to this, there is a need for data processing to be performed in real time so that the vehicle could react with immediacy to changing situations that may occur during the journey. It is also highly desirable that the energy consumption associated with such processing

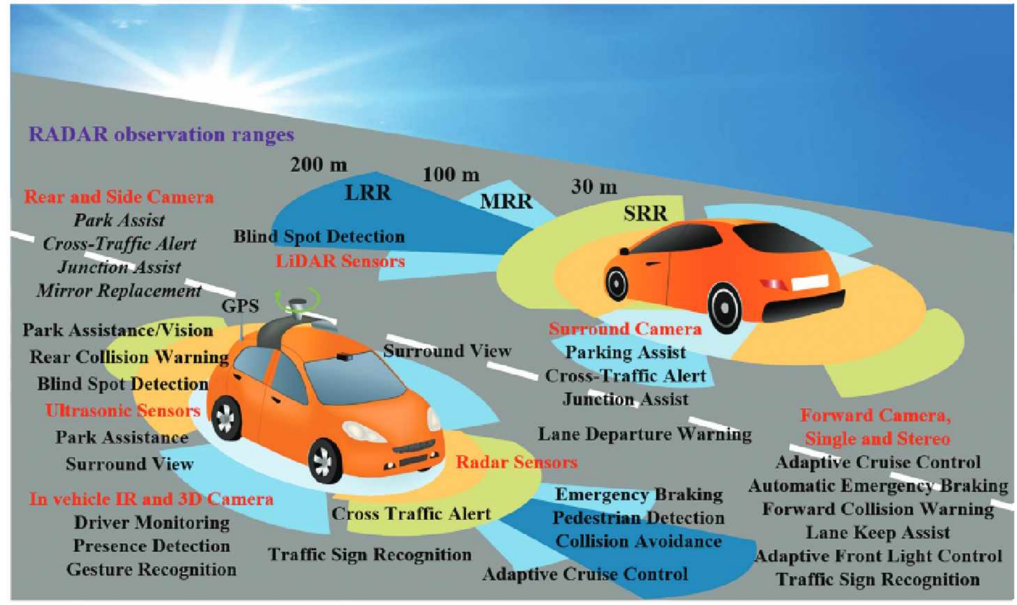

Figure 1: Integrated sensors on an autonomous vehicle [8].

was as low as possible, so that the vehicle's autonomy was not affected.

Autonomous driving systems are essentially composed of three classes of sub-systems [1, 2]: *scene recognition*, *route planning* and *vehicle control*, consisting of a set of algorithms each. In particular, as shown in Fig. 2, *scene recognition*, the class in which this article falls, comprises three essential tasks, namely 1) *localization*, which precisely establishes the vehicle's location, 2) *object detection*, which identifies objects of interest in the vehicle's environment (such as other vehicles, pedestrians or road signs, with the aforementioned objective of avoiding accidents and also traffic violations), and 3) *object tracking*, which, since the object detection algorithm is carried out on each frame of the image, is responsible for relating its results to other frames in order to predict the trajectories of moving objects. These three tasks account for a very high percentage of the total computation time required [1] and therefore constitute bottlenecks that significantly limit the ability of conventional processors to satisfy the existing restrictions in the design of this type of systems. For this reason, it is being proposed to incorporate some type of accelerator to the on-board processing systems that helps the processor to fulfill the strict time limits in which it must operate.

2.2 RISC-V-based development environment

In order to carry out the implementation and evaluation of our proposal, which will be explained in the following section, a series of software tools have been used, as detailed next:

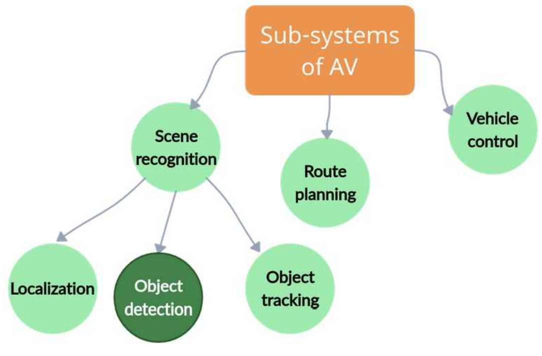

Figure 2: Schematic of the subsystems of an autonomous vehicle.

2.2.1 Chipyard.

Chipyard [13] is an environment for the design and evaluation of hardware systems that consists of a set of tools and libraries designed to provide an integration path between open-source tools and commercial tools for the development of Systems on Chip (SoC). The environment provides a range of components for design construction as well as for compilation and simulation. Among these components there are several RISC-V cores and accelerators, including the BOOM core and Gemmini accelerator that make up the heterogeneous system chosen in this paper and that will be detailed in Section 3. The simulation of the complete system accelerated with FPGA is one of the types of simulation supported by Chipyard, using the FireSim tool described below.

2.2.2 FireSim.

FireSim [14] is a hardware simulation platform that runs on Amazon cloud services and automatically deploys the FPGA services in the cloud when needed. In particular, the user can generate the RTL of an own design and run it on these FPGAs, obtaining the same results as if the circuit was physically deployed.

2.2.3 Amazon Web Services.

Amazon Web Services [15] is a cloud services platform that offers from training courses in new technologies –such as artificial intelligence or IoT– to infrastructure services –such as storage or cloud computing. We focus on cloud computing because it offers a wide range of hardware platforms, including EC2 F1 instances that correspond to FPGAs, giving us the versatility we need to synthesize designs and to simulate the execution of applications on them.

3 Platform design

The platform employed in our experiments features a general-purpose processor equipped with an accelerator –implemented as a systolic array architecture– for matrix multiplication. Both components have been developed by the Computer Architecture group at Berkeley University [6]. The accelerator communicates with the processor through the RoCC (Rocket Co-Processor) interface, which allows the accelerator to receive the specific instructions that the processor sends, as shown in Fig. 3. In the following two sections we describe the processors and the accelerator used.

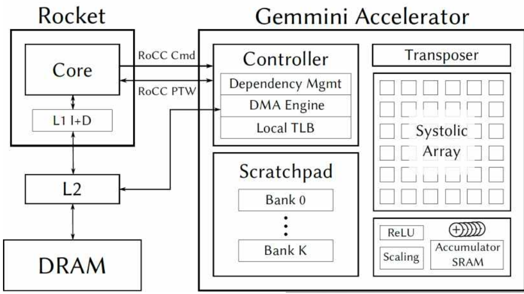

Figure 3: Architecture of our heterogeneous platform [6].

3.1 Processors

As Fig. 3 illustrates, our system features a core plus an accelerator. In our experiments we opted to employ either the Rocket or the BOOM (Berkeley Out-of-Order Machine) processor. Both of them are written in Chisel and implement the RV64GC instruction set. Also, they are easily parameterizable and can be synthesized. Notably, the cores are configured by using the Rocket Chip SoC generator [16].

The main differences between both cores lie in the pipeline characteristics: while the Rocket core features an in-order 5-stage pipeline, the BOOM core is equipped with a deeper out-of-order pipeline, which is inspired by those of MIPS R10000 and Alpha 212645 [5]. Consequently, the BOOM core is expected to deliver higher performance when executing our line detection algorithm. However, this comes at the expense of higher energy consumption than that of the Rocket core. Therefore, we experiment with both processors in order to check if the speedup reported by the BOOM core is significant enough to cancel out the energy constraints.

3.2 The Gemmini Accelerator

The Gemmini matrix multiplication accelerator relies on a 2D systolic array architecture, as shown in Fig. 3, to perform matrix multiplications in an efficient fashion. In addition to this systolic array, it also features a scratchpad memory with multiple banks and an accumulator, which has more bits than that of the systolic array. Besides, the implementation allows to choose, at compile time, between two specific calculation mechanisms: output-stationary or weight-stationary.

Customized instructions –out of RISC-V standard– are available for the Gemmini accelerator, so that it is equipped with its own instruction queues that make it possible to execute concurrently with the processor. The Gemmini programming model can be broken down into three different levels. In the high-level we can run Open Neural Network Exchange (ONNX) models, being the accelerator itself in charge of mapping the ONNX kernel to the accelerator by means of dynamic dispatch. In the mid-level we use a hand-tuned library including C macros to perform data transfers between the main memory and the accelerator's scratchpad memory, which should be explicitly defined, as well as to automate the calculation of the block size used to split a matrix and to perform the full multiplication in a transparent way for users. Among available functions we highlight the following: *tiled_matmul*, to run a tiled matrix multiplication with hardcoded tiling factors; *tiled_conv*, to apply a convolution with hardcoded tiling factors; *tiled_matmul_auto*, to run a tiled matrix multiplication with automatically calculated tiling factors; *gemmini_mvin*, to move data from the main memory to the scratchpad and *gemmini_mvout*, to move data from the scratchpad to the main memory. Finally, at the low-level, we can write our own mid-level kernels with low-level assembly instructions.

4 Adapting image processing algorithms

As stated previously, the aim of this work is to accelerate image processing algorithms employed to guide autonomous vehicles. Notably, we focus on those algorithms targeted to detect road lines from road images. In this section we first introduce the basic algorithms used (the Canny algorithm and the Hough transform). Then, we show the full algorithm that we have employed in this work as starting point for line detection and, finally, we propose some changes to this algorithm oriented to improve its efficiency and performance without impacting on accuracy.

4.1 Canny Algorithm

Among the edge detection methods developed to date, the Canny algorithm is one of the methods more strictly defined that provides a satisfactory and reliable detection. Thus, it has become one of the most popular algorithms targeting edge detection.

This algorithm relies on calculus of variations, which allows to find an analytical function to approximate the real curve (i.e., the road lines) as accurately as possible. The procedure followed by the Canny algorithm [17] can be broken down into 5 stages as shown next:

1. Noise reduction: applying the Gauss filter for image smoothing.
2. To find the intensity gradient of the image.
3. Magnitude threshold to the gradient: applying a threshold to the gradient for discarding edge false positives.
4. Double threshold: applying again a threshold to the gradient for highlighting the potential edges.
5. Hysteresis: removing weak or disconnected edges.

Algorithm 1 shows the pseudo-code we employed to apply the Canny algorithm, broken down into the 5 stages aforementioned. Essentially, it includes multiplications of consecutive matrices and conditions checking in order to detect edge points.

Algorithm 1 Canny algorithm summarized pseudo-code.

1: float $NR \leftarrow$ mask $*$ image ▷ Stage 1: Noise reduction
2: float $G_x \leftarrow$ mask $*$ NR ▷ Stage 2: Gradient intensity
3: float $G_y \leftarrow$ mask $*$ NR
4: float $G \leftarrow \sqrt{G_x^2 + G_y^2}$
5: float $\phi \leftarrow \arctan(|G_y|/|G_x|)$
6: **if** $\phi[*] \geq$ threshold$_\phi$ **then** ▷ Stage 3: Gradient threshold
7: float $\phi \in \{0, 45, 90, 135\}$
8: **end if**
9: **if** $\phi[*] \geq$ threshold$_\phi$ && $G[*] \geq$ threshold$_G$ **then** ▷ Stage 4: Double threshold
10: int edge$[*] \leftarrow 1$
11: **end if**
12: **if** $G[*] \geq$ threshold$_G$ && edge$[*] == 1$ **then** ▷ Stage 5: Hysteresis
13: int image_out$[*] \leftarrow 255$
14: **end if**

4.2 Hough Transform

The Hough transform is a technique of features extraction which is employed in multiple fields involving image processing, as computer vision or image digital processing. The goal of the algorithm is to find imperfect objects among certain classes of objects by means of a voting procedure. This procedure lies in creating a space with the values assigned to each pixel, so that the resulting local maximums in the so called accumulator space are the possible detected objects.

Generally, the classical Hough transform was only applied to the detection of straight lines, but in recent years it has been modified and currently it is employed for the detection of arbitrary curves, as ellipses or circles.

Algorithm 2 illustrates the code we employed to apply the Hough transform [18]. In this code, for each edge point previously detected with the Canny algorithm, the Hough transform draws a set of straight lines going through that point, recording the amount of lines going through each image pixel. Hence, those points with more lines going through them will correspond to a line in the original image.

Algorithm 2 Hough transform summarized pseudo-code.

1: For each edge point (i, j)
2: **if** image$[i * width + j] \geq 250$ **then**
3: $\theta \leftarrow 0$
4: **while** $\theta \leq 180$ **do**
5: float $\rho \leftarrow j * \cos\theta + i * \sin\theta$
6: accumulators$[(\rho + c_\rho) * 180 + \theta]$++
7: θ++
8: **end while**
9: **end if**

4.3 Line Detection

Once we have described the two previous algorithms, we now employ a combination of both as well as another specific code targeted to detect with higher accuracy the lines that demarcate lanes in conventional ways. For this purpose, given a certain input image, we first apply the Canny algorithm and then the Hough

transform, so that we can apply a function (*Get lines coordinates*) to detect lines in the resulting image. In Algorithm 3 we show the code of the mentioned function, which involves a search of local maximums in the preprocessed image and the generation of a straight line going through closest maximums.

Algorithm 3 Get lines coordinates algorithm summarized pseudo-code.

```
1: For each image point (ρ, θ)
2: if accumulators[∗] ≥ threshold then ▷ It is a local maximum
3:     max ← accumulators[*]
4:     if accumulators[neighbourhood(*)] ≥ max then ▷ We check its neighborhood
5:         max ← accumulators[neighbourhood(*)]
6:     end if
7: end if
8: lines.add(x1, y1, x2, y2) ▷ We save the two points that demarcate the straight line
```

4.4 Delivering higher performance

In the previous sections we have described the original code of the algorithms, which indeed deploys many floating point variables. Therefore, it is advisable to replace them by integer variables without any loss in accuracy. We effectively made these changes in the code and we verified that no accuracy loss occurs when detecting lines in an image. Fig. 4 shows the original image with detected lines highlighted in red. The analytical results corresponding to the lines detected with the original algorithm and with the simplified one do match, and also the second algorithm has performed slightly faster. Details on these modifications can be found in [19].

Apart from this change, we also performed a profiling of the full code divided into three stages: 1) original image loading, 2) lines detection and 3) generation of an output image with the detected lines. Accordingly to the results obtained, we opted for not generating an output image (that is not needed by our system) due to the high cost associated, as shown in Table 1. In doing so, we are able to reduce the execution time by 4.2x as we can derive from data reported in Table 2. It is worth noting that the time values illustrated in the mentioned tables are approximate since the profiling was not performed on the target platform, but on an Intel i7 processor running Linux. However, in order to report time values as accurate as possible, the algorithms were run several times so that the tables show the average values obtained. According to numbers from Table 2, our attention is drawn to the line detection phase since it accounts for almost 70% of the execution time.

In addition, we have performed another specific profiling of the stages of the line detection algorithm in

Table 1: Phased *profiling* of the full code.

	Time(μs)	% over total
Image load	43803	7,32%
Line detection	98171	16,42%
Image generation	456030	76,26%
Total	598004	

Table 2: Phased *profiling* of the full code excluding the generation of the output image.

	Time(μs)	% over total
Image load	43485	30,58%
Line detection	98714	69,42%
Total	142199	

order to know in which parts of the processing the acceleration efforts should be focused. Table 3 illustrates that the most time-consuming part is the application of the Canny algorithm, which accounts for more than 87% of the total execution time. Therefore, we will focus on accelerating this stage of image processing.

Table 3: Phased *profiling* of the line detection algorithm.

	Time(μs)	% over total
Canny algorithm	90265	87,64%
Hough transform	12275	11,92%
Get coordinates	459	0,45%
Total	102999	

5 Experimental results

In this section, we first describe the hardware platforms as well as the workloads employed in our experiments, and then we detail the results obtained.

5.1 Platforms generated

All the components used in the designs generated are written in Scala, so it is easy to modify their main features such as number of registers or number of Re-Order Buffer (ROB) entries. Notably, we generate several designs: while all of them include one (or more) Rocket or BOOM cores, they may include or not the Gemmini accelerator.

Apart from the cores, for the sake of fairness the remaining components in the different designs generated (such as memory, clock frequency or buses) are the same in all of them. Hence, all designs have an L2 –shared in multicore platforms– 4MB size. In order to optimize the design to fit into smaller FPGAs, the option MCRams is enabled in the FireSim platform configuration for all designs. This option allows the FPGA simulation tool (Golden Gate [20]) to simulate the RAM via serialized accesses with a decoupled model [14].

Figure 4: Original image with detected lines highlighted in red.

Platforms including the Gemmini accelerator can only be designed to work at 50MHz while the remaining ones can reach 80MHz. Thus, the later have been designed both at 50 and 80 MHz for a fair comparison against designs equipped with Gemmini. Notably, the platforms generated are:

1. *Platform* 1: Rocket single core.
   This architecture includes a single *Big* Rocket core. There are four different sizes for the core, namely *Big, Medium, Small* and *Tiny*, with different features such as the size of L1-cache. The *Big* Rocket is the only one providing Floating Point Unit. It also has by default the parameters shown in Table 4. More information on the details of the configuration can be found in [19].

2. *Platform* 2: Rocket dual core.

   This is the same configuration as Platform 1 but it includes two *Big* Rocket cores. This dual configuration also has the option MTModels enabled in the FireSim platform configuration, so that each core is simulated with a separate thread of execution on a shared underlying physical implementation [14].

3. *Platform* 3: Heterogeneous Rocket single core + Gemmini Accelerator.

   This architecture is made up by a *Big* Rocket core and a Gemmini matrix multiplication accelerator, which has been designed with default options: 16x16 8-bit systolic array, both dataflows supported (output-stationary and weight-stationary), float data type supported, a set of accumulator registers with 64B of total capacity, a 256KB scratchpad with 4 banks, a small TLB with 4 entries and a bus width of 128 bits.

4. *Platform* 4: BOOM Single core.

   This architecture includes a single *Large* BOOM core. There are different macros for defining BOOM cores of *Giga, Mega, Large, Medium* and *Small* sizes. The main differences between the one that we are using and the rest is the number of entries in the ROB and some L1-cache parameters. Thus, in the configuration *WithNLargeBooms* the value of notable parameters are shown in Table 4. More information on the details of the configuration can be found in [19]. The Large size was chosen because it is just big enough to provide the required performance with minimum power consumption.

5. *Platform* 5: BOOM dual core.

   This is the same configuration as Platform 4 but it includes two *Large* BOOM cores, with the MTModels option enabled.

6. *Platform* 6: Heterogeneous BOOM single core + Gemmini Accelerator.

   This architecture is made up by a *Large* BOOM core and a Gemmini matrix multiplication accelerator, which has been designed with the default options explained earlier.

5.2 Workloads generated

Different workloads were designed for running on the platforms described in the previous section. They are the following:

1. *Workload* 1: Multithreaded application on top of Linux buildroot distribution.

   In this workload, a multithreaded application (with each thread computing the addition of 2

Table 4: Platform configuration options.

		Big Rocket	Large Boom
I&D Cache	Size	16KB	32KB
	Sets	64	64
	Ways	4	8
	Prefetching	no	disabled
TLB	Sets	1	1
	Ways	32	512
BTB Entries		28	28
ROB Entries		no	96
FPU		yes	yes
Branch predictor entries		no	128

long arrays, as explained in [19]) is executed on top of Linux. It has been specifically designed to fully exploit the parallel features of the platforms, so that it can be used to evaluate the maximum performance obtainable in the different multicore designs. This value will serve as an upper bound when we evaluate the performance achieved by our target application.

2. *Workload* 2: Line detection algorithm on top of Linux buildroot distribution.

   In this workload, the modified version of the line detection application explained in Section 4 is executed on top of Linux.

3. *Workload* 3: Line detection algorithm for bare-metal platforms with Gemmini.

   In this workload, in addition to the modifications in Section 4, we have modified the line detection algorithm to add matrix multiplications. In the original version, this algorithm multiplies some mask values to a pixel neighborhood manually by writing the corresponding scalar multiplications. We have rewritten these multiplications in a matrix form, obtaining a 5x5 matrix for the mask and a 5x5 neighborhood matrix for each pixel. As for the platform, the differences with respect to the previous workload are that this platform includes a Gemmini accelerator for matrix multiplication and the fact that no operating system is available for this platform. Thus, matrix multiplications in the code have to be replaced by calls to a Gemmini multiplication. As previously explained, some C macros are provided with the designs that make it possible to easily programming the accelerator. First, data need to be moved from the main memory to the scratchpad memory in Gemmini, then the multiplication is performed in tiles and finally the results are transferred back to the main memory. We will use the *tiled_matmul_auto* function that receives the dimensions of both matrices as input parameters and automatically splits the multiplication in blocks of suitable size for the systolic array and memory, thus performing the whole multiplication. Finally, system calls not available outside Linux were removed from the code and their functionality was implemented in an equivalent way.

5.3 Experiments

In this section we show the results obtained from the execution of the workloads on the different platforms designed. The metrics measured are clock cycles and instructions retired provided by the performance counters of the target platforms.

5.3.1 Experiment 1: Execution of a multithreaded application on single core and dual core platforms both with Rocket and BOOM cores.

The goal of this experiment is to verify the maximum performance attainable in the different platforms by using a massively parallel application. Therefore we employ *Workload 1*, configured with as many independent threads as the number of cores in the system, i.e., 1 or 2 depending on the specific platform.

The target platforms in this case include both single and dual core processors (either Rocket or BOOM, running at 80MHz) that correspond to the *Platforms* 1, 2, 4 and 5 previously described.

The results of the experiment are shown in Table 5, both for a simulation in which the main loop is executed once (column labelled *N_times = 1*) and 8 times (column *N_times = 8*). The number of clock cycles for the experiment with 8 iterations is 8 times the one of the single iteration experiment. Besides, speedup of the dual core version with respect to the single core is very close to 2x for both Rocket and BOOM. Finally, comparing the performance of the different cores, BOOM achieves almost 2.2x higher performance than Rocket, so that a single BOOM core outperforms a dual core Rocket running at the same frequency for this highly parallel application.

Thus, it has been verified that multithreaded applications are being correctly simulated in the multicore

Table 5: Cycles when executing Workload 1 on Platforms 1, 2, 4 and 5.

	N_times = 1	N_times=8
	Cycles	
Rocket singlecore	2.01×10^9	1.59×10^{10}
BOOM singlecore	9.17×10^8	7.31×10^9
Rocket dualcore	9.97×10^8	7.99×10^9
BOOM dualcore	4.53×10^8	3.66×10^9
Speedup BOOM vs Rocket	2.19x	2.18x
Speedup Rocket dual vs single	2.02x	1.99x
Speedup BOOM dual vs single	2.02x	1.99x

platforms, achieving the expected speedup. Furthermore, the comparison between both types of cores has been established.

5.3.2 Experiment 2: Execution of the line detection application on Rocket and BOOM single cores.

This second experiment involves simulating the execution of the line detection application (*workload* 2) on the Rocket and BOOM single core platforms employed in the previous experiment (Platforms 1 and 4), also running at 80MHz. In Table 6 we report the number of clock cycles and instructions retired corresponding to each of the different parts of the line detection algorithm, as well as the average cycles per instructions (CPI) value. In addition, we calculate the actual time from the cycles and clock frequency, resulting in times of around half second. In particular, for the Rocket core we obtain a total execution time of 0.648s and for the Boom core 0.327s. As shown, the CPI for the Hough transform is higher than 3 in both Rocket and BOOM platforms. Moreover, its execution on the BOOM processor almost matches the time reported on the Rocket platform, as the multiple data dependencies in the code make out-of-order capabilities useless.

On the other hand, the Canny and the GetCoordinates algorithms exhibit lower CPI numbers in both platforms, achieving a speedup of 2x when executing on the Boom processor with respect to Rocket, due to the greater instruction level parallelism that can be extracted from both algorithms. Recall that the Canny algorithm is the most relevant part of the line detection application, consuming close to 90% of the total execution time (as shown in Table 3). In conclusion, using the BOOM core for the execution of the workload is interesting in terms of the global speedup achieved.

5.3.3 Experiment 3: Execution of the line detection application on heterogeneous platforms with a Rocket or BOOM single core and a Gemmini matrix multiplication accelerator.

This experiment consists on simulating the execution of the modified line detection application (*workload* 3) on the heterogeneous single core platforms made up by a Rocket or BOOM processor plus a Gemmini matrix multiplication accelerator running at 50MHz.

Table 7 shows first the results obtained in the simulation of Workload 3 (line detection application for bare metal) on a Rocket single core (used as baseline for computing speedups) and a BOOM single core, both running at 50MHz. As the first row shows, BOOM is 41% faster than Rocket. The execution results from the previous section, that is, those corresponding to Workload 2 (line detection application for Linux) on Rocket and BOOM single core at 80MHz are also compared to the baseline execution, achieving speedups of 2.09x and 3.76x respectively. It is worth noting that although the code of Workloads 2 and 3 does not exactly match, it performs the same functionality. Finally, the results from the simulation of Workload 3 on heterogeneous platforms in which matrix multiplications are performed using the Gemmini accelerator are also recap in Table 7. According to them, speedups of 2.36x and 3.7x are reported for Rocket and BOOM based platforms respectively, with respect to the baseline. Although these speedups can be considered as significant, they are far from the maximum values attainable by the accelerator. The reason is that the size of the matrices employed is smaller than that of the systolic array, which indeed is not fully utilized.

Furthermore, in the graph shown in Fig. 5 we can see the time corresponding to all the single core and heterogeneous experiments. The first thing we notice is that the out-of-order execution of the Boom core is beneficial for the Canny algorithm, leaving the Rocket core as the slowest by far at both 50 and 80MHz. Furthermore, we see how the combination of the cores with the Gemmini accelerator at 50MHz gives us a similar time to the same cores without accelerator at 80MHz, which gives us a great benefit in terms of consumption by running at a lower clock frequency which should be taken into account in the field of autonomous vehicles, as it would provide greater autonomy. In addition, we note that the shortest time is under half a second, in particular 300ms, and we achieve it with the combination of the Boom core and the Gemmini accelerator at a clock frequency of 50MHz. Thus, a vehicle travelling at 50km/h could run the algorithm every 4 metres approximately and if necessary, options such as mounting several systems in parallel or slightly increasing the clock frequency for faster processing could be explored.

In conclusion, for this application with small matrices, both platforms based on the BOOM core deliver similar performance (speedup of around 3.7x with re-

Table 6: Cycles, instructions retired and CPI when executing Workload 2 on Platforms 1 and 4 at 80MHz.

		Cycles	Instructions	CPI	Time(ms)
Rocket singlecore	Canny	2.18×10^9	9.06×10^8	2.40	648,38
	Hough	3.32×10^8	9.35×10^7	3.55	98,86
	Coordinates	6.49×10^6	3.47×10^6	1.87	1,93
Boom singlecore	Canny	1.08×10^9	9.06×10^8	1.19	327,10
	Hough	3.16×10^8	9.35×10^7	3.38	96,07
	Coordinates	3.2×10^6	3.47×10^6	0.92	0,97
Speedup Boom vs Rocket	Canny	2.02x	1.00x	2.02x	1.98x
	Hough	1.05x	1.00x	1.05x	1.03x
	Coordinates	2.03x	1.00x	2.03x	1.99x

Table 7: Speedup results when executing Workload 2 on Platforms 1 and 4 at 80MHz, and Workload 3 on Platforms 3, 4, 6 at 50MHz, with respect to execution of Workload 3 on Platform 1 at 50 MHz.

	Speedup vs Rocket singlecore 50MHz			
	Canny	Hough	Coordinates	Total
Boom singlecore 50MHz	1.44x	1.04x	1.85x	1.41x
Rocket singlecore 80MHz	2.26x	0.98x	1.07x	2.09x
Boom singlecore 80MHz	4.57x	1.03x	2.18x	3.76x
Rocket + Gemmini 50MHz	2.54x	1.16x	1.03x	2.36x
Boom + Gemmini 50MHz	4.43x	1.07x	1.98x	3.70x

spect to the Rocket baseline), being the BOOM single core at 80MHz slightly faster than the BOOM + Gemmini at 50MHz. Even in this non favourable scenario, the accelerator allows to report high performance working at a lower frequency, being more power efficient than the single core platform running at higher frequency.

6 Conclusions and future work

In this paper we have explored the acceleration of a line detection algorithm in the autonomous car environment using a heterogeneous system consisting of a general-purpose RISC-V core and a domain-specific accelerator. In particular, we analyzed the application to identify the most computationally intensive parts of the code and adapted it accordingly for more efficient processing.

The first conclusion we extract from this work is that RISC-V architecture provides a hw-sw ecosystem that is well suited for IoT in general and autonomous vehicle systems in particular, due to its versatility and modularity, which allows to generate platforms adapted to different scenarios. In fact, in this work, we designed six different platforms covering a wide spectrum of alternatives: on one side single and dual core homogeneous systems, and on the other side heterogeneous platforms with a single core plus a matrix multiplication accelerator –all of them including high performance BOOM cores or more efficient Rocket cores.

Also, a multithreaded application with high data parallelism has been designed to analyze the performance of the homogeneous platforms built. Thus, it has been verified that multithreaded applications are being correctly simulated in the multicore platforms, achieving the expected speedup. Furthermore, the comparison between both types of cores determined that a single BOOM core is up to 2.19 times faster than a Rocket one.

Finally, the original application of line detection has been modified in order to decrease its execution time without losing accuracy, and it has also been adapted for bare metal and Gemmini execution. We simulated the application on all designed platforms. BOOM-based platforms reported the best performance numbers, achieving speedups of 3.7x with respect to the baseline (a single Rocket core running at 50MHz), and being the single BOOM core running at 80MHz slightly faster than the BOOM + Gemmini platform at 50MHz. As previously stated, even working at a lower frequency the accelerator allows to report high performance, being more power efficient than the single core counterpart working at a higher frequency. It is worth noting that our goal in this work was to explore how an domain-specific accelerator was able to accelerate the baseline execution (just using a conventional single core) in applications belonging to autonomous vehicles environment.

As future work, other applications which involve multiplication of big matrices can be adapted to heterogeneous platforms in order to implement more of the functionalities required for autonomous vehicles. Moreover, Gemmini is expected to achieve much higher speedups for inference using neural networks, as shown in [6], so exploring this issue constitutes an interesting avenue for future work.

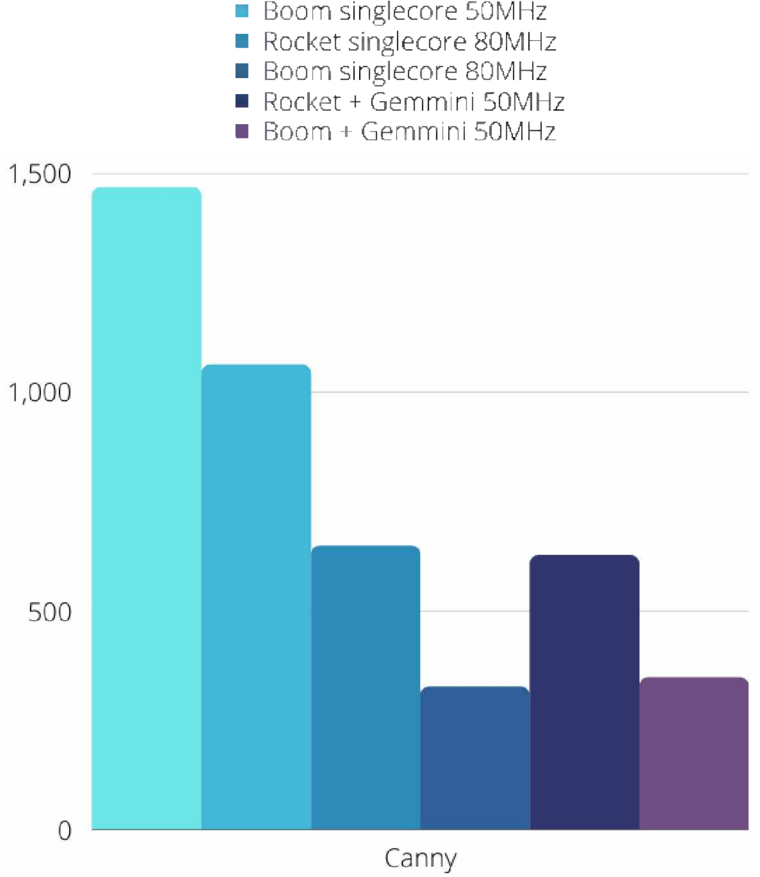

Figure 5: Time results when executing Workload 2 on Platforms 1 and 4 at 50MHz and 80MHz, and Workload 3 on Platforms 3 and 6 at 50MHz.

Competing interests

The authors have declared that no competing interests exist.

Funding

The present work has been funded by the Comunidad de Madrid through project S2018/TCS-4423 and by the Ministry of Science, Innovation and Universities through project RTI2018-093684-B-I00.

Authors' contribution

MJB wrote the programs, conducted the experiments, analyzed the results and wrote the manuscript; KO and FC conceived the idea, analyzed the results and wrote the manuscript; FT revised the manuscript. All authors read and approved the final manuscript.

References

[1] S.-C. Lin *et al.*, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, p. 751–766, 2018.

[2] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, pp. 60–68, 11 2015.

[3] P. Bose, A. J. Vega, S. V. Adve, V. S. Adve, and V. J. Reddi, "Secure and resilient socs for autonomous vehicles," in *Proceedings of the 3rd International Workshop on Domain Specific System Architecture (DOSSA)*, pp. 1–6, 2021.

[4] B. Yu *et al.*, "Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations," in *Proceedings of 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1067–1081, 2020.

[5] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Proceedings of the 4th Workshop on Computer Architecture Research with RISC-V (CARRV)*, pp. 1–7, 2020.

[6] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, pp. 769–774, 2021.

[7] "The 6 levels of vehicle autonomy explained." Available at: `https://www.synopsys.com/automotive/autonomous-driving-levels.html`. Accessed on 2022-09-07.

[8] O. Vermesan *et al.*, *IoT technologies for connected and automated driving applications. Internet of Things - The Call of the Edge*, pp. 306–332. River Publishers, Oct. 2020.

[9] R. Coppola and M. Morisio, "Connected car: technologies, issues, future trends," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, pp. 1–36, 2016.

[10] T. Rateke *et al.*, "Passive vision region-based road detection: A literature review," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–34, 2019.

[11] F. Bounini, D. Gingras, V. Lapointe, and H. Pollart, "Autonomous vehicle and real time road lanes detection and tracking," in *IEEE Vehicle Power and Propulsion Conference (VPPC)*, pp. 1–6, 2015.

[12] G. Zhang, N. Zheng, C. Cui, Y. Yan, and Z. Yuan, "An efficient road detection method in noisy urban environment," in *Proceedings of IEEE Intelligent Vehicles Symposium*, pp. 556 – 561, 2009.

[13] A. Amid *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[14] S. Karandikar *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 29–42, 2018.

[15] "Amazon web services (aws)." Available at: `https://aws.amazon.com/es`, 2021. Accessed on 2022-09-07.

[16] K. Asanovic *et al.*, "The rocket chip generator. eecs department," *University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016.

[17] J. F. Canny, "Finding edges and lines in images," *Theory of Computing Systems - Mathematical Systems Theory*, p. 16, 1983.

[18] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, p. 11–15, 1972.

[19] M. J. Belda, "Image processing in autonomous vehicles on a risc-v with accelerator," *Master Thesis, UCM*, 2022.

[20] A. Magyar *et al.*, "Golden gate: Bridging the resource-efficiency gap between asics and fpga prototypes," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019.

Citation: M.J. Belda, K. Olcoz, F. Castro and F. Tirado. *Optimization of a line detection algorithm for autonomous vehicles on a RISC-V with accelerator.* Journal of Computer Science & Technology, vol. 22, no. 2, pp. 129–140, 2022.
DOI: 10.24215/16666038.22.e10
Received: April 22, 2022 **Accepted:** September 7, 2022.
Copyright: This article is distributed under the terms of the Creative Commons License CC-BY-NC.