Multi-Vehicle Map Fusion using GNU Radio

E. Akin Sisbot, Augusto Vega, Arun Paidimarri, John-David Wellman, Alper Buyuktosunoglu, Pradip Bose {EASISBOT, AJVEGA, APAIDIMA, WELLMAN, ALPERB, PBOSE}@US.IBM.COM
IBM Research, Yorktown Heights, NY

David Trilla DAVID.TRILLA@BSC.ES
Barcelona Supercomputing Center, Barcelona, Spain

Abstract

In this paper, we present a representative open-source application for fully/semi-autonomous vehicles operating as a collaborative *swarm* using GNU Radio. This application is the driver of the DARPA-sponsored EPOCHS project led by IBM, which focuses on agile heterogeneous system-on-a-chip (SoC) development. The EPOCHS Reference Application (ERA) incorporates local sensing, creation of occupancy grid maps, vehicle-to-vehicle (V2V) communication of grid maps between neighboring vehicles using GNU Radio-based dedicated short-range communication (DSRC), and map fusion to create a joint higher-accuracy grid map.

This paper analyzes the GNU Radio-based DSRC transceiver workload on a Xilinx Zynq UltraScale+ MPSoC, identifying computation kernels for software and/or hardware acceleration. In particular, we present optimizations of *Viterbi decoding* and *complex exponential* that result in a measured 5.4× performance improvement over a CPU-only baseline, with projections of a fully-optimized version of this hardware accelerator showing significantly higher benefits.

1. Introduction

Recent advances in AI and sensors provide greater capabilities for autonomous vehicles, but require more powerful hardware processing support. Many chip manufacturers continue to develop faster and/or more efficient hardware solutions to help accelerate machine learning and data processing. Most of this development, however, is *ad hoc* design, not systematically driven from the application to the silicon design. The *Efficient Programmability Of Cognitive Heterogeneous Systems* (EPOCHS) project focuses on novel methodologies to enable rapid development of programmable, heterogeneous system-on-a-chip (SoC) designs for ultra-efficient deployment and execution of a target application domain.

Autonomous vehicles and advanced driver assistance systems (ADAS), through their suite of sensors, are expected to significantly improve driving safety and dramatically reduce traffic accidents. Combining the on-board sensor capability with vehicle-to-vehicle (V2V) communications, autonomous vehicles become even more aware of their environment. Recognizing the benefits, V2V communication efforts are being driven by many major automotive manufacturers, including BMW, General Motors, Daimler, Honda, and Toyota. The current solution adopted by these manufacturers is through *dedicated short-range communications* (DSRC), a variation of IEEE 802.11 Wi-Fi standard operating in the 5.9-GHz band.

In the specific context of autonomous cars, the addition of V2V connectivity can provide significant benefits. One of the most critical challenges that the automotive industry faces these days, for example, is related to the rate of *false predictions* while the car is driving and perceiving the environment. False predictions (either false negatives or false positives) can be alleviated through the use of arrays of sensors to build redundancy into the on-board system; however this approach can be economically inefficient and will not necessarily solve the problem. In cases where the vehicle is poorly positioned for any on-board sensor to reduce the false prediction rate, adding sensors cannot effectively reduce these errors. V2V communications can enable *swarm-based perception* use cases, where vehicles make use of other vehicles' sensor data (i.e. other perspectives) to effectively reduce false prediction rates, as Figure 1 depicts. Even though individ-

Proceedings of the 1st GNU Radio Conference, Copyright 2016 by the author(s).

ual vehicles may have a relatively poor vision of the surrounding environment (due, for example, to bad weather conditions or line-of-sight occlusion), the real-time fusion of their local views can result in more reliable navigation. This remark is further supported by observations from the swarm robotics domain, where several works have shown that the successful completion rate of a task by a group of robots improves with the number of robots in the swarm. Object recognition accuracy, for example, increases significantly when performed in a cooperative manner and with a relatively large number of robots involved in the process (Giusti et al., 2012). Similarly, navigation delays are cut down when the navigation activities are conducted cooperatively by a robot swarm (Ducatelle et al., 2014). We anticipate the emergence of a similar behavior in groups of vehicles when the proper swarming elements are in place.

To guide the development of a heterogeneous SoC, we developed an open-source reference application called ERA (EPOCHS Reference Application) that models a set of connected autonomous/semi-autonomous vehicles operating in a common environment. Each autonomous vehicle in ERA uses its on-board sensors to generate local occupancy grid maps, which it also communicates to other nearby vehicles using DSRC. When a vehicle receives occupancy maps from its nearby neighbors, it merges the received ones with the locally-generated occupancy maps, expanding the scope and increasing the accuracy of this vehicle's perception. A vehicular system could use a dedicated DSRC modem for the V2V communications such that ERA would only need to implement the higher layers of the communication stack. However, we propose to integrate the computation from the entire modem (PHY and MAC layers) into the main SoC. This would provide opportunities for maximal hardware reuse and more optimal load balancing across accelerators and general purpose cores in the chip. In this regard, ERA incorporates an open-source DSRC IEEE 802.11p transceiver implemented using GNU Radio (Bloessl, 2018; Bloessl et al., 2018). The continuous stream processing required for the protocol presents a key challenge for real time operation, especially within a power-constrained embedded SoC.

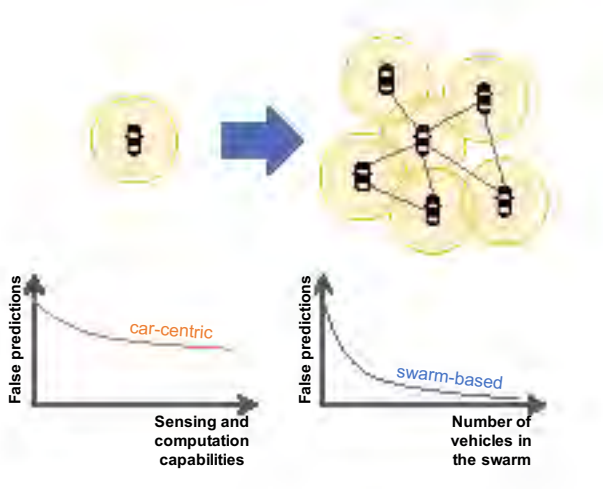


Figure 1. Increasing the on-board sensing and computation capabilities do not necessarily reduce false predictions by the same factor. V2V communication and swarm intelligence are expected to alleviate this problem.

This paper describes ERA, our local sensing and V2V communications application suite, and presents a novel system-level performance evaluation methodology that we apply to ERA. We illustrate this methodology with an analysis of the V2V communication subsystem, including performance hot-spots, and the potential of various acceleration schemes to improve the performance and efficiency of the implemented system.

2. EPOCHS Reference Application

This work is conducted as part of our project, sponsored by the DARPA MTO Domain-Specific System on Chip (DSSoC) Program (DSSoC, 2019). In this context, ERA (ERA, 2019) serves as the testbed and driving application domain for the development of our domain-specific SoC design methodology and associated technologies. Specifically, ERA is an open-source application that enables multi-vehicle (cooperative) sensor fusion in future autonomous/connected cars, using elements of computer vision and vehicle-to-vehicle (V2V) communications. It models an autonomous vehicle that incorporates an on-board *sensing and mapping fabric* for multi-modal sensing and mapping. ERA extends these local perception capabilities with a *communication and consensus fabric* consisting of V2V communications and multi-vehicle map fusion. The ultimate goal is to explore and demonstrate the improvement in perception capability by sharing information with other vehicles, and participating in collaborative swarm intelligence.

2.1. System Overview

ERA is built using the Robot Operating System (ROS) (Quigley et al., 2009) middleware and GNU Radio (GNU Radio Foundation, Inc., 2019) to implement the V2V software defined radio. The overall architecture of ERA for two agents in sufficiently close proximity to communicate with one another is shown in Figure 2. The major components of ERA are the following:

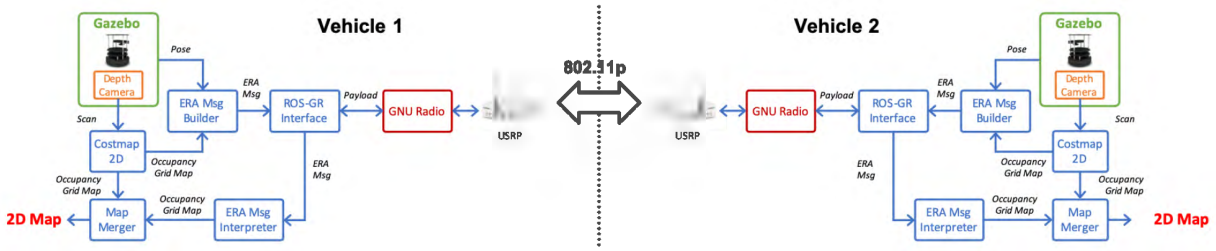
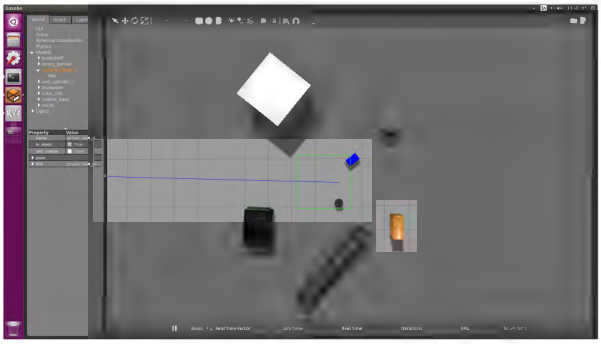
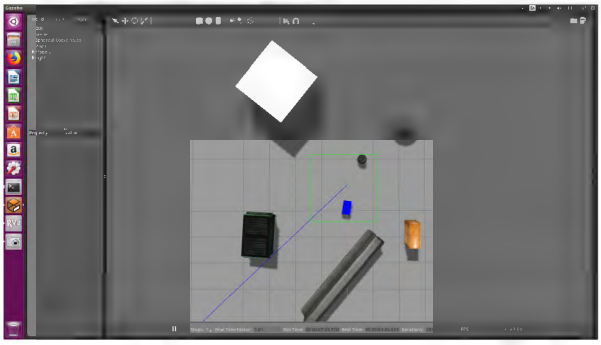


Figure 2. Overall ERA Architecture.

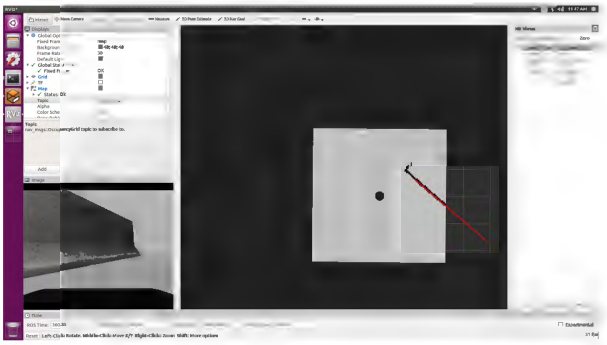


(a)



(b)

Figure 3. Gazebo simulations for vehicles *1* and *2*.



(a)

(b)

Figure 4. Occupancy grids created by vehicles *1* and *2*.

2.1.1. Gazebo

We use Gazebo (Koenig & Howard, 2004) to simulate the sensors feeds, the position of the vehicles (agents), and the physics of the world. The Gazebo simulator creates sensor data, and makes it available to the overall system. In our implementation each agent is equipped with an RGB-D camera. Gazebo generates a color image along with a depth image based on the environment and agent's position. Figure 3a shows a sample simulated environment for *vehicle 1*. In this scenario, the current vehicle is represented by the black circle in the middle of the screen. The remote vehicle (*vehicle 2*) is represented by the blue box. Figure 3b shows the simulated environment for *vehicle 2*, where the local vehicle (*vehicle 2*) is represented by the black circle whereas the remote vehicle (*vehicle 1*) is represented by the blue box. The simulated vehicles (agents) are equipped with simulated RGB-D cameras (such as Microsoft Kinect, or Intel Realsense), which provide *point clouds* as output. We then transform the point clouds to 2D laser scans by projecting them onto the ground.

2.1.2. Costmap 2D

Costmap 2D is a ROS package in the 2D Navigation Stack (ROS, 2019). This package builds occupancy grid maps centered around the agent using 2D laser scans as input. An occupancy grid map is a $m \times n$ matrix where each element represents a square region sized r. Thus the overall size of the occupancy grid

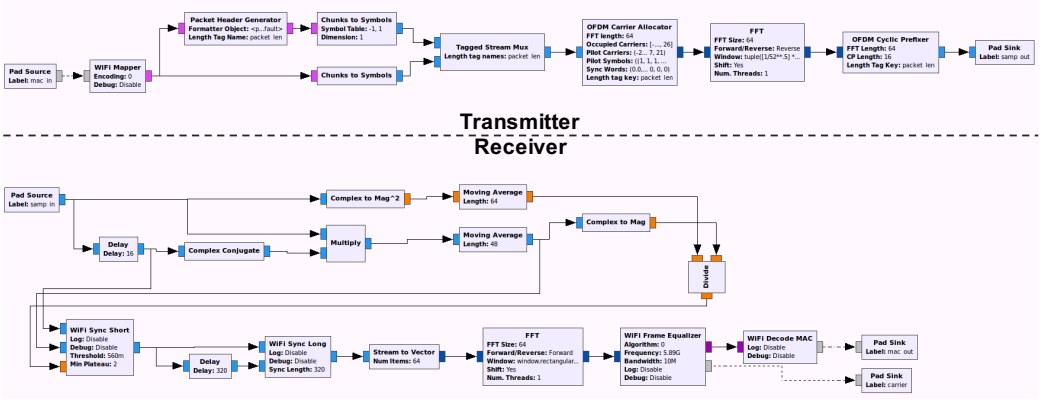


Figure 5. GNU Radio-based 802.11p transceiver flowgraph based on (Bloessl, 2018; Bloessl et al., 2018).

is $m \times n \times r$ m^2. The resolution and size of the grid depend on the sensor's specifications and environment characteristics.

Each element in the occupancy grid has one of the following values:

- *free*, a location free from any obstacles;
- *occupied*, a location occupied by an obstacle;
- *unknown*, a location the sensor doesn't cover.

The *costmap* calculated from the simulated environment for *vehicle 1* is shown in Figure 4a and for *vehicle 2* in Figure 4b.

2.1.3. ERA Message Builder

This block is in charge of packing the necessary information in a message and preparing it for transmission. The messages exchanged between vehicles include each vehicle's local occupancy grid and its *pose* (position and orientation) information. The information is time-stamped, and the ID of the agent is also attached. Overall, an ERA message is composed of following data:

- *ID*, the unique ID of the vehicle;
- *timestamp*, the time of the sensor data acquisition;
- *pose*, the position and the orientation of the vehicle at time *timestamp*;
- *occupancygrid*, the occupancy grid map;

2.1.4. GNU Radio

This block is in charge of broadcasting ERA messages using the adopted DSRC IEEE 802.11p protocol. Figure 5 presents the GNU Radio flowgraph, an open-source software defined radio implementation of the IEEE 802.11p standard (Bloessl, 2018; Bloessl et al., 2018). To reduce the message payload size, ERA applies LZ4 compression (LZ4, 2019) to the message before injecting it into the transmission pipeline. Once the transmission payload has passed through the GNU Radio transmitter pipeline, we send it to a USRP (Universal Software Radio Peripheral) device to be broadcast over the air.

When a message is received by the receiver's USRP device, it passes through the GNU Radio receiver pipeline, and the subsequent message payload is sent through LZ4 decompression to recover the original ERA message.

2.1.5. ERA Message Interpreter

This block parses received ERA messages and makes the embedded occupancy grid map available to the overall system. The first time the received vehicle ID is "seen", a representation of the remote vehicle is created in the Gazebo simulator. Otherwise, the location of the existing remote vehicle in the simulator is updated.

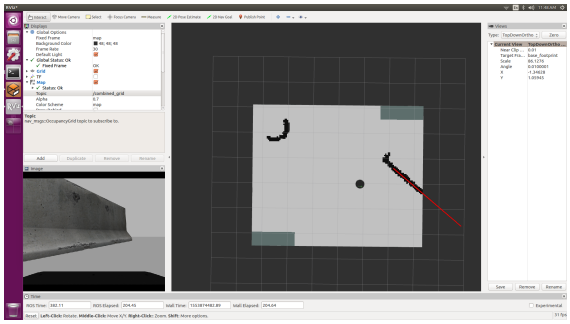


Figure 6. Occupancy grid map created in *vehicle 1*.

2.1.6. Map Merger

The last process in ERA is the Map Merger, where the remote map received through DSRC, and the local map built by Costmap 2D are merged into a single occupancy grid of potentially higher accuracy. Map Merger uses the time stamp to obtain a synchronized pair of maps, and using the *pose* information received in the ERA message, applies the correct transformation and overlays the local map to the remote one. The resulting map is an accumulation of the two maps, providing a larger scope on the world, with greater aggregate information than any individual map. Figure **??** shows the combined map created in *vehicle 1*.

3. Performance Evaluation of V2V Communications using GNU Radio

This section focuses on the GNU Radio components within ERA to identify the most critical (CPU intensive) blocks in the 802.11p transceiver. We use a Xilinx Zynq UltraScale+ MPSoC (ZCU102, 2019) as the development platform, featuring a quad-core ARM Cortex-A53 CPU (1.2 GHz) and an FPGA substrate. This SoC provides heterogeneous computing resources along with the programmable logic for rapid architecture exploration. We use Ubuntu 16.04 and GNU Radio 3.7.9.3 in our evaluation platform.

We create a stand-alone "mini" version of ERA to simplify the evaluation, which consists of transceiver components configured in a loop-back mode. Packets are injected at a specified rate and passed to the transmitter pipeline. The output of this pipeline is routed back to the receiver pipeline. This approach effectively simulates the full ERA DSRC transceiver and is executed continuously, sending and receiving messages at a desired rate.

We use the Linux *perf* (Linux Kernel Organization, 2019) performance analysis tool to monitor performance with low overhead, recording the performance data and the associated location in the program, in order to ascribe the sample to a given function/code location. Figure 7a shows the execution time fraction per GNU Radio **block**. The following blocks account for 60% of the total execution time: *sync_short*, *frame_equalizer*, *decode_mac*, and *sync_long*.

Figure 7a also shows that each of these four blocks consume more than 14% of the total execution time, and all of them belong to the receiver pipeline. Figure 7b illustrates some of the underlying **functions** in the GNU Radio 802.11p pipeline that take more than 2% of the DSRC workload execution time. Functions in this figure are different from the blocks shown in Figure 7a — functions are smaller than blocks, and a GNU Radio block can make calls to multiple functions. Similarly, multiple blocks can make calls to the same functions.

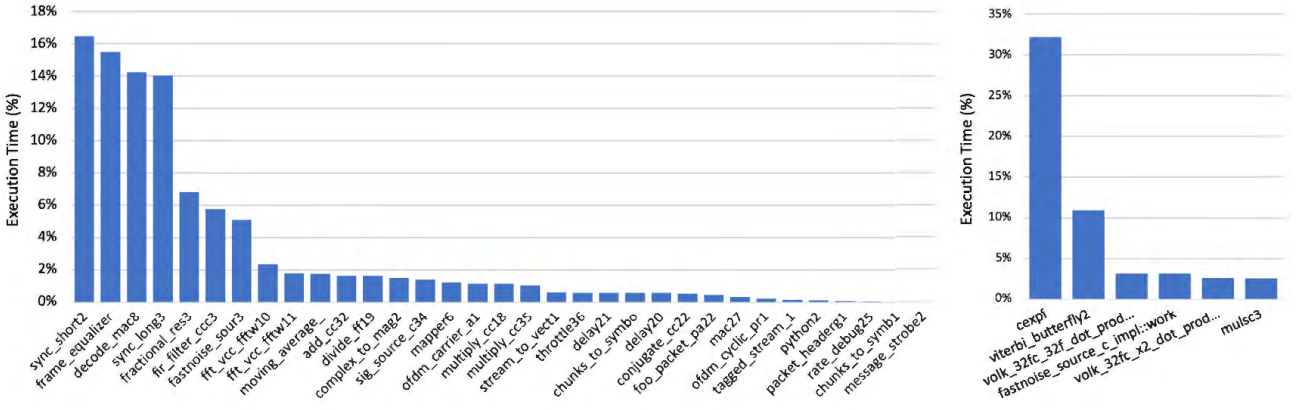
The most time consuming **functions** from Figure 7b are *cexpf* and *viterbi_butterfly2*, taking 32% and 11% of the execution time, respectively. In considering the most time consuming **blocks** of the DSRC application, we note that three different blocks use *cexpf*: *sync_short2*, *sync_long3* and *frame_equalizer*. Similarly, both the *decode_mac* and *frame_equalizer* blocks use the *viterbi_butterfly2* function — however the contribution to execution time of the *viterbi_butterfly2* function in *frame_equalizer* is negligible (less than 0.1%). Given that these functions (*cexpf* and *viterbi_butterfly2*) comprise approximately 43% of the total DSRC application run-time, we determine that these two are the best candidates for explicit acceleration.

3.1. Throughput Optimization Opportunities

As indicated in the previous section, the most time consuming GNU Radio blocks belong to the receiver — specifically, the *cexpf* and *viterbi_butterfly2* functions called from them. Compared to the transmitter (where frames can be pre-computed before they are streamed to the radio), the receiver must keep pace with the incoming data rate to avoid packet losses. In addition, the vehicle will transmit one message but can potentially receive N messages per time period (where N is the number of nearby cars) and thus the stressor is on the worst-case portion of the receiver's performance. More collaboration (i.e. more received inputs) per time period means better decision making and better collaborative intelligence.

In order to assess the acceleration opportunities of the receiver pipeline, we replace all the calls to *cexpf* and *viterbi_butterfly2* with "dummy" (empty) ones. Specifically, these calls just return proper precomputed output without making any computation — in other words, these are "ideal" calls that take almost no time. The goal is to determine how much effort should be dedicated to accelerating each part of the code, by indicating the level of acceleration achievable for the overall system.

Figure 8 shows the throughput when different injection rates (x-axis) are used. The throughput is measured by injecting a fixed-content 1500-byte packet into the transmitter at that rate, and measuring the throughput of the DSRC pipeline. The *baseline* (solid orange line) shows a maximum throughput of 378 kb/s. The ideal throughput would continue to follow the input



(a) Transceiver blocks.

(b) Transceiver functions.

Figure 7. Execution time fraction.

rate (dashed dark blue line). The other lines in the figure show projections of the throughput when ideal versions of the *cexpf* and *viterbi_butterfly2* functions are used. For readability, we limit the projections to cases where each function is singly accelerated to near-zero computational effort, and then both in concert. These results therefore identify an upper bound on the maximum performance that can be achieved by accelerating these functions, which in this case reaches slightly over 600 kb/s. Also, the optimization of *cexpf* yields a better performance improvement than the optimization of *viterbi_butterfly2*, and the combination of both optimizations yields the highest return.

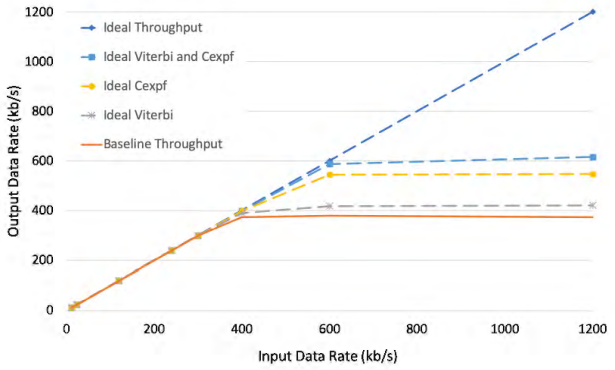


Figure 8. Performance throughput at different input rates, comparing different (ideal) optimizations for the two most time consuming functions.

4. Acceleration Alternatives

Section 3.1 establishes the upper bound on the maximum performance that can be obtained by accelerating the *cexpf* and *viterbi_butterfly2* functions. This section moves on to determining the actual, realizable performance acceleration alternatives for the adopted Zynq UltraScale+ MPSoC. We specifically consider *vector* and *FPGA* implementations, which are discussed next.

4.1. Understanding functionality

The *cexpf* function is a call to the standard C math library that computes complex base-e exponentials. Given a complex input $a + bi$, the *cexpf* function returns the value $r = e^{a+bi}$. The result of this operation can be computed using Euler's formula $e^{a+bi} = e^a * (cos(b) + sin(b)i)$. The computation of the *sine* and *cosine* functions can similarly be performed by some number of steps (determined by the desired precision) of well-known Taylor's expansions (e.g. $sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...$).

The *Viterbi decoder* (Viterbi, 1967) uses a software implementation of the Viterbi algorithm to decode convolutionally encoded data. In our case, it operates on a convolutional code with constraint length of $K = 7$ and $rate = 1/2$ [1]. The *viterbi_butterfly2* function performs a butterfly operation, the basic Viterbi decoder operation.

4.2. Vector Implementations

We use vector implementations of the *cexpf* and *viterbi_butterfly2* functions using native ARM NEON calls (NeonLib, 2019). We initially considered the possibility of using the Vector-Optimized Library of Kernels (VOLK) (VOLK, 2019), but its ARM implementation does not yet include several operations (like *sine* and *cosine*, used in *cexpf*). Although we could have

[1] Used in some BPSK, QPSK and 16-QAM modes of IEEE 802.11p.

implemented these kernels in VOLK, we decided to rely on ARM NEON due to time limitations.

4.3. FPGA Implementations of *cexpf*

The *cexpf* function of $a + bi$ can be computed by $e^a * (cos(b) + sin(b)i)$. We implement a dual-datapath pipeline in the programmable logic of our Xilinx Zynq UltraScale+ MPSoC that computes e^a and $(cos(b) + sin(b)i)$ in parallel, and multiplies them to generate the full e^{a+bi} output. Figure 9 shows the block design of the accelerator, which takes a 64-bit input containing two 32-bit floating-point numbers: the real and imaginary parts. The imaginary component is sent to a floating-point unit to convert into the fixed-point representation required by the CORDIC unit (Xilinx, 2019). The CORDIC unit then computes the sine and cosine, and returns two fixed-point values which are converted back to 32-bit floating-point values. The real component is sent through a floating-point unit that computes its exponent; then the sine, cosine and real portions are sent to the final multipliers to obtain the result. This is a simple implementation of the pipeline, using Xilinx components and the base GNU Radio buffer data interfaces; as such, it does not represent the full performance advantage possible from a custom implementation of the same functionality.

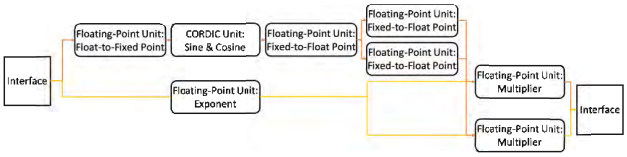


Figure 9. Block design of the *cexpf* accelerator datapath.

With the main pipeline in place, we explore several implementations for its interface with the CPU. The Xilinx IPs naturally support direct AXI bus interfacing, allowing the pipeline to be driven directly by the bus using built-in buffering and back-pressure control. This provides a means to use the hardware accelerator much like a (distant) functional unit. Xilinx also supports a variety of other interfacing options that allow bulk input data transfers, decoupling the data write and accelerator computation. The three main approaches that we adopt in this work are:

FIFO Implementation: It uses Xilinx *first-in, first-out* (FIFO) buffers to store the accelerator's inputs and outputs. In this case, the CPU stores all the input data into the FIFO buffers and signals the pipeline to start reading/writing input/output values from/to the FIFO buffers. Once this process is completed, the CPU copies the data back from the output FIFO buffers to main memory. The top part of Figure 10 depicts this scheme.

Block RAM Implementation: In this case, the CPU transfers the input data into the accelerator's Block RAM (BRAM), and then signals the pipeline to start. The output is written back into an output BRAM. Although the CPU still must do explicit copies to/from the BRAM, it can use more efficient block memory move operations (rather than single-item reads and writes). The middle part of Figure 10 depicts this scheme.

DMA Based Implementation: This implementation adds a DMA engine to the FPGA that directly moves data to/from the accelerator's BRAM from/to DRAM, freeing the CPU from doing this. The bottom part of Figure 10 depicts this scheme.

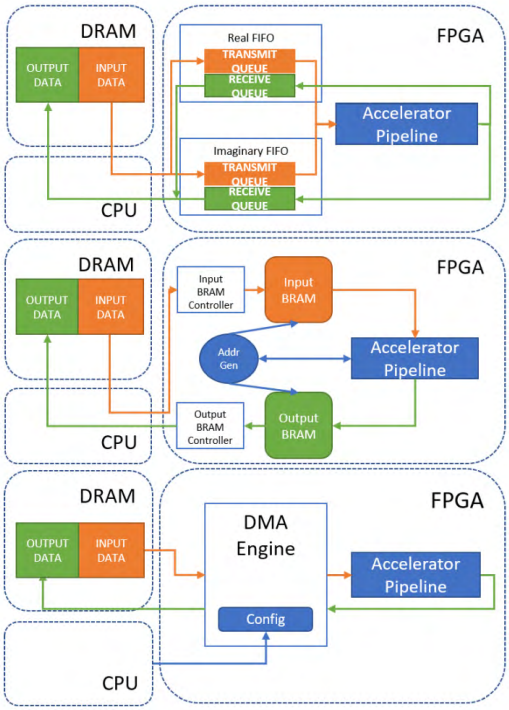


Figure 10. Studied complex exponential accelerator interfaces: FIFO (top), BRAM (middle), and DMA (bottom).

In this dual-datapath pipeline implementation, it becomes critical to equalize the number of stages of both datapaths in such a way that they contain the same number of pipeline stages. If the two real- and imaginary-input datapaths contain disparate numbers of stages, then the shorter one can expose stalls affecting the performance of the entire accelerator. This happens because buffers on the shorter pipeline will fill up waiting for the other datapath to provide its data.

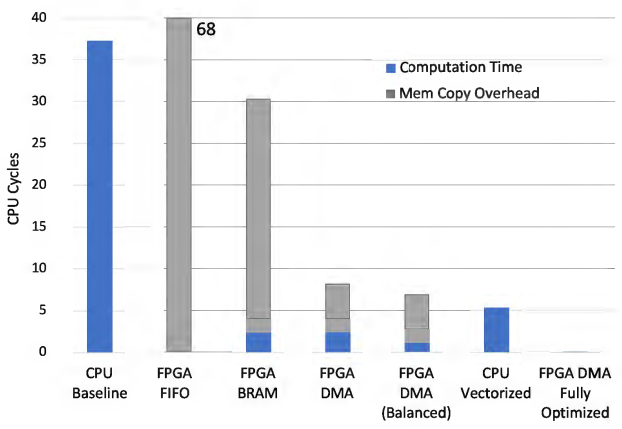


Figure 11. Execution time per operation of different accelerator options for the complex exponential function. The average time is computed using 4096 complex exponential operations. *CPU Baseline* shows the original model performance. FPGA implementations break execution times into exposed computation and memory copy overhead for (left to right) AXI FIFO buffers, FPGA Block RAM memory, and a DMA engine without and with well-balanced datapaths. *CPU Vectorized* is a NEON SIMD optimized version of the code, and *FPGA DMA Fully Optimized* shows a projection of a more optimized implementation running at 300 MHz with four parallel computation engines and memory-copy elimination.

As a result, we also implement and evaluate a *balanced* datapath in this study.

4.4. Performance Comparison of *cexpf* Implementations

Figure 11 shows a performance comparison of the different accelerator implementations of the *cexpf* function on a test bench decoupled from the entire application. We plot the average execution time to compute 4096 complex exponent operations on random input data, and break it out into computation time and data movement overhead. The performance is reported in CPU cycles sampled at 100 MHz. The baseline is the standard C math library executed in the ARM CPU, which requires 37 CPU cycles, with no additional overhead to move input or output values.

The simple AXI FIFO-based pipeline turned out to be notably slower than the CPU, at 68 cycles per complex exponent output, where the actual computation time was negligible compared to the memory copy overhead. In this case, the CPU must write each input value into the same FIFO buffer address (and similar for reading outputs), requiring each subsequent transfer to wait for the previous one to complete before the next transfer can be initiated.

The Block RAM (BRAM) implementation corrected this problem by providing a span of memory addresses that can be simultaneously accessed by the CPU to write inputs or read outputs. As a result, we obtain a significant speedup over the FIFO implementation, bringing the per-operation execution time down to about 30 cycles. Note, however, that there is still a significant data movement overhead in this case.

The next implementation uses the DMA engine, which is in charge of directly accessing DRAM DDR memory and feeding the accelerator without CPU intervention. This configuration improves the execution time to 8 cycles. Examination of the throughput, however, indicated stalling due to imbalance in the accelerator pipeline reflecting unnecessary stalls. The corresponding *balanced* version realizes a performance of 6 cycles per output (with an effective computation time close to 1 cycle).

We also consider the performance acceleration from running a vectorized ARM NEON (NeonLib, 2019) version of *cexpf*. This implementation runs entirely on the ARM core, resulting in an execution time of 5.3 cycles per operation — slightly more efficient than the DMA implementation and offering a 7× speedup over the CPU baseline.

The DMA *balanced* design described above still incurs memory copy overhead, which is due to the fact that data has to be first moved into physically addressable DRAM to allow the DMA engine to access it. Ideally, we could instead map the workload's input and output data to known physical address space and completely avoid this overhead. We estimate that this single modification would provide a 33× speed-up, reducing the per-operation cost from 37 to 1.1 cycles. Implementing this change, however, does require modification of GNU Radio, and also has security implications that are outside the scope of this paper.

Further performance optimizations of the *FPGA DMA Balanced* design can be achieved by increasing the width of the AXI bus (from the current 64 bits to 128 or 256 bits) and replicating the accelerator pipeline to allow multiple parallel computations in the accelerator. A 4-way replication could provide a 4× speedup for the computational portion of the accelerator, reducing the 1.1 cycles per operation to 0.275 cycles. Finally, the clock rate of the FPGA can be cranked up from 100 MHz to 300 MHz (which is a stable AXI bus frequency) to theoretically achieve an extra 3× throughput speedup, reducing the overall computation portion to just about 0.1 cycles per operation. The *FPGA DMA Fully Optimized* case in Figure 11 shows the performance projection when all these optimiza-

tions work together.

4.5. Overall Workload Performance Analysis

Once we have estimates of the realistic levels of performance acceleration available from the accelerator options, we can determine the impact on the overall DSRC transceiver performance. Figure 12 shows how the throughput of the application changes when varying the injection rate similarly to Figure 8, but now using realistically achievable acceleration speedups measured from our actual implementations.

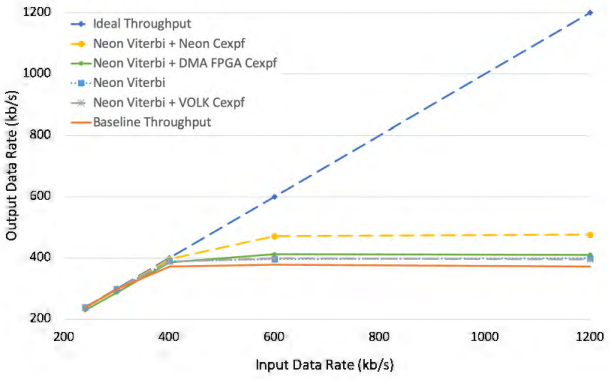


Figure 12. Performance throughput of the overall DSRC transceiver application at different input rates, when realistic acceleration implementations are in place.

As shown, the vectorized implementation (*Neon Viterbi + Neon Cexpf*) is the one that yields the best throughput: 470 kb/s for a 1200-kb/s input rate. Some distance behind that is the actual *cexpf* FPGA implementation using DMA engines, offering 415 kb/s. Recall that the FPGA implementation plotted here still requires accessing physically-addressed DRAM, which vastly diminishes the performance gains that can be obtained. These results provide useful insights of the kind of accelerators that can provide the highest benefits for DSRC applications.

5. Conclusion

This work introduces our ERA software suite, which models intelligent vehicles operating in an open world and sharing information to enable greater collective (swarm) insights. Specifically, ERA models intelligent vehicles using on-board sensors to identify objects in the nearby locus of their world view, and communicates this information with other in-range vehicles allowing each to form a greater composite world view.

Using ERA, we consider the computing requirements for an autonomous vehicle workload, and determine that the *vehicle-to-vehicle* (V2V) receiver functionality performance is critical to the efficient exchange of information between in-range vehicles. We develop a focused V2V workload driver that exercises the GNU Radio V2V DSRC pipeline within ERA at a programmable messaging rate, and allows ready investigation of the performance bottlenecks and hot-spots. In this V2V DSRC implementation, we identify the receiver task as the critical functionality, and in that two main functions (*Viterbi decoding* and *complex exponential*) that contribute significant performance overheads.

Having identified functions that constitute approximately 43% of the receiver pipeline execution time, we then consider techniques to accelerate their performance, and measure the full receiver pipeline speedup. We evaluate a simple FPGA implementation of the *complex exponential* function with balanced datapath, which provides 5.4× performance improvement over a CPU-only baseline. Projections of a fully-optimized version of this hardware accelerator show that the computation portion could provide 58× speedup over the vectorized code, but the performance is currently constrained by the need for redundant memory copy (from GNU Radio buffers to physically addressable memory).

As we progress in our program, we continue to apply our performance analysis methodology throughout the full ERA workload, and to identify performance bottlenecks. This process progressively identifies the best candidates for acceleration, and expresses the performance potential of such acceleration in the context of the full running workload. In this way, we can explore the full workload performance space, iteratively improve the underlying hardware design, and eventually settle on a fully optimized SoC implementation.

6. Acknowledgements

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document is approved for public release: distribution unlimited.

References

Bloessl, Bastian. IEEE 802.11 a/g/p transceiver for GNU radio. https://github.com/bastibl/gr-ieee802-11, 2018.

Bloessl, Bastian, Segata, Michele, Sommer, Christoph, and Dressler, Falko. Performance assessment of IEEE 802.11p with an open source SDR-based prototype. *IEEE Transactions on Mobile Computing*, 17(5):1162–1175, May 2018. doi: 10.1109/TMC.2017.2751474.

DSSoC. DARPA DSSoC Program. https://www.darpa.mil/program/domain-specific-system-on-chip, 2019.

Ducatelle, Frederick, Di Caro, Gianni A., Förster, Alexander, Bonani, Michael, Dorigo, Marco, Magnenat, Stéphane, Mondada, Francesco, O'Grady, Rehan, Pinciroli, Carlo, Rétornaz, Philippe, Trianni, Vito, and Gambardella, Luca M. Cooperative navigation in robotic swarms. *Swarm Intelligence*, 8(1):1–33, Mar 2014. ISSN 1935-3820. doi: 10.1007/s11721-013-0089-4. URL https://doi.org/10.1007/s11721-013-0089-4.

ERA. ERA GitHub. https://github.com/IBM/era/, 2019.

Giusti, A., Nagi, J., Gambardella, L., and Di Caro, G. A. Cooperative sensing and recognition by a swarm of mobile robots. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 551–558, Oct 2012. doi: 10.1109/IROS.2012.6385982.

GNU Radio Foundation, Inc. GNU radio. http://www.gnuradio.org, 2019.

Koenig, N. and Howard, A. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pp. 2149–2154 vol.3, Sep. 2004. doi: 10.1109/IROS.2004.1389727.

Linux Kernel Organization. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org, 2019.

LZ4. Extremely fast compression algorithm. https://github.com/lz4/lz4, 2019.

NeonLib. Simple ARM NEON optimized sin, cos, log and exp library. http://gruntthepeon.free.fr/ssemath/neon_mathfun.html, 2019.

Quigley, Morgan, Conley, Ken, Gerkey, Brian P., Faust, Josh, Foote, Tully, Leibs, Jeremy, Wheeler, Rob, and Ng, Andrew Y. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

ROS. ROS 2D navigation stack. http://wiki.ros.org/navigation, 2019.

Viterbi, Andrew J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, 1967.

VOLK. Vector-optimized library of kernels. http://libvolk.org, 2019.

Xilinx. Xilinx LogiCORE™ CORDIC IP. https://www.xilinx.com/products/intellectual-property/cordic.html, 2019.

ZCU102. ZCU102 evaluation board user guide. https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/, 2019.