EigenEdge: Real-Time Software Execution at the Edge with RISC-V and Hardware Accelerators

Kuan-Lin Chiu*
chiu@cs.columbia.edu
Columbia University
New York, NY, USA

Guy Eichler*
guyeichler@cs.columbia.edu
Columbia University
New York, NY, USA

Biruk Seyoum
biruk@cs.columbia.edu
Columbia University
New York, NY, USA

Luca P. Carloni
luca@cs.columbia.edu
Columbia University
New York, NY, USA

ABSTRACT

An important goal in the field of real-time computation at the edge is to achieve balance between low-latency requirements and strict low-power constraints. Into this equation, we would like to incorporate simple Application Programming Interfaces (APIs) for software development and utilization of open-source IPs that encourage reusability in the public domain. One big challenge is to bridge the gap between APIs that simplify the implementation of complex algorithms but mostly rely on CPU-centric computing paradigms, and lightweight heterogeneous hardware architectures designed for the constraints of real-time computation at the edge. We introduce a hardware/software co-design approach that combines software applications designed with Eigen, a powerful open-source C++ library that abstracts linear-algebra workloads, and real-time execution on heterogeneous System-on-Chip (SoC) architectures. We use ESP, an open-source SoC design platform that allows us to integrate the CVA6 RISC-V processor and custom hardware accelerators. With FPGA-based experiments, we show that our approach provides significant performance and energy efficiency gains, while maintaining the simplification provided by high-level software development.

KEYWORDS

Edge Computing, System-on-Chip, FPGA, Eigen, RISC-V, Real-Time.

ACM Reference Format:
Kuan-Lin Chiu, Guy Eichler, Biruk Seyoum, and Luca P. Carloni. 2023. EigenEdge: Real-Time Software Execution at the Edge with RISC-V and Hardware Accelerators. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3576914.3587510

1 INTRODUCTION

While the development of Mobile Edge Computing (MEC) and Internet-of-Things (IoT) is rapidly growing, developers and researchers are trying to find ways to bridge the gaps between complex computation-intensive workloads in popular domains and the limited resources of constrained IoT platforms [13, 27, 43, 44, 46]. These optimized implementations take advantage of the synergy between software and hardware development, and maximize the potential of limited hardware resources for the exponentially growing need in computational abilities [2, 5, 6, 17, 18, 47].

New applications tend to rely more on the development with increasing abstraction, where specialized libraries and benchmarks allow developers to isolate the implementation of the algorithms from the infrastructure of the system. These libraries provide optimized and configurable implementations for commonly used kernels, and allow the developers to focus on the performance of the application as a whole [12, 26, 29, 37].

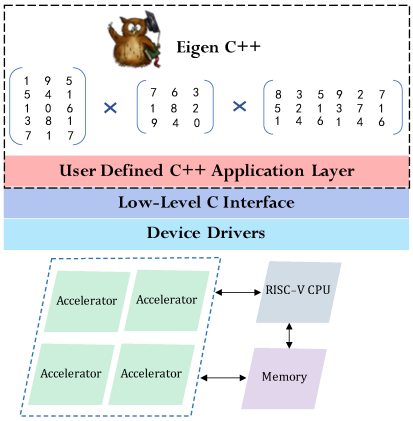

Fig. 1: The EigenEdge hardware/software co-design approach.

One popular library is *Eigen* [29]. It provides linear-algebra (LA) methods that can be used for various applications in the fields of Quantum Computing, Robotics, Healthcare, and more [7, 11, 28, 30, 35, 39]. The *Eigen* syntax enables developers to perform fundamental LA operations (matrix and vector manipulation, factorization, decomposition, etc.) with minimal lines of code. In addition, the library is highly optimized for commercial processors, and supports multi-threading with OpenMP [3].

To evaluate the performance of *Eigen* under different levels of parallelism, we conducted an analysis on Intel Xeon platforms with different numbers of processor cores and a clock frequency of 2.2 GHz. On each platform we ran the same application that uses *Eigen* to compute the product of two 1000×1000 matrices. We tested the applications with multi-threading enabled and disabled. The results of the analysis are summarized in Fig. 2. As expected, enabling multi-threading with multiple cores provides better performance. For instance, with an 8-core processor, the computation takes only 1.8 seconds with multi-threading enabled, and 7.4 seconds with multi-threading disabled. On a single-core Intel processor, however, multi-threading provides no substantial benefit over a single-threaded execution. Nevertheless, lightweight edge devices with a single-core processor will not benefit from multi-threading with *Eigen*. This limitation motivated us to explore alternative solutions for running *Eigen* workloads on edge devices as heterogeneous SoCs.

In addition, to exploit the simplifications offered by *Eigen* for IoT and MEC requires bridging the gap between software applications

*Both authors contributed equally to this research.

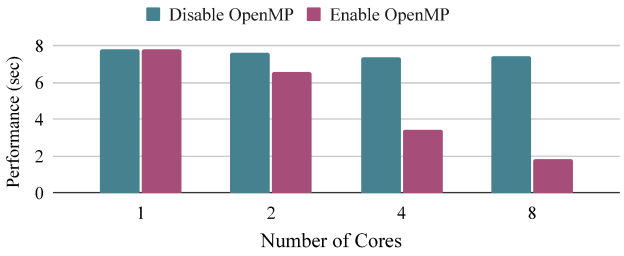

Fig. 2: Execution on varying numbers of Intel Cores with OpenMP enabled/disabled.

that use *Eigen* and lightweight heterogeneous SoC architectures that are based on low-power processors and specialized hardware accelerators [5, 8, 41, 42, 47]. As we seek a solution to bridge this gap, we aim at making it available for open-source projects to support its accessibility and reusability in the public domain.

We present EigenEdge (Fig. 1), a software architecture that allows any C++ application that uses *Eigen* to take advantage of specialized hardware accelerators embedded in a heterogeneous SoC. EigenEdge maintains the abstraction in the syntax of *Eigen*, while decoupling the system-level integration from the development of the application. EigenEdge simplifies hardware/software co-design and combines the open-source *Eigen* C++ library, the ESP SoC design platform [33], the CVA6 RISC-V processor [45], and hardware accelerators. FPGA-based experiments show that offloading computations from *Eigen* to the accelerators results in performance and energy-efficiency gains over similar *Eigen* workloads running on CVA6 alone. With the integration of more hardware accelerators, EigenEdge can close the gap among abstract software, efficient hardware, and real-time computation at the edge.

2 BACKGROUND AND RELATED WORK

This section provides a summary of the main projects we combined for the design of EigenEdge and a discussion of related work in the field of hardware/software co-design.

Eigen C++ Linear-Algebra Library. Linear-Algebra (LA) is the basis of almost all areas in mathematics. LA provides concepts that are used in many areas of computer science as graphics, cryptography, machine learning, computational biology, and more. The *Eigen* library provides a clean C++ API for using LA objects and methods. *Eigen* is a public library and has been widely used in the backend of several application domains including AI [39], vision/graphics [1, 28], quantum computing [30], robotics [38], and healthcare [11]. *Eigen* contains a benchmark of LA operations with support for dense matrices and vectors of varying sizes, which can be decided statically or dynamically. While providing limited support for operations on sparse matrices, *Eigen* has been extended to support several specialized features, such as non-linear optimization [14], polynomial solvers, FFT, etc [29].

Eigen uses C++ expression templates to optimize the code at compile time. It uses the OpenMP library to enable multi-threading, as well as automatic vectorization for several compilers and architectures that support SIMD instructions. *Eigen* achieves superior performance by applying optimization techniques that avoid redundant memory accesses, allocate memory statically for fixed-size

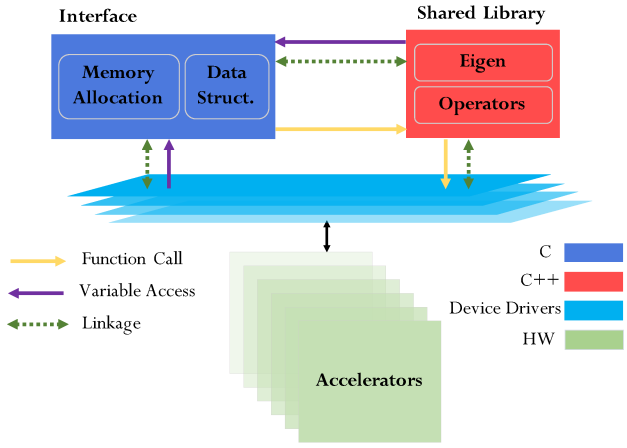

Fig. 3: EigenEdge software architecture.

matrices, support lazy compiler evaluations, and globally-optimized compilation. Finally, *Eigen* has also been extended to support other programming languages such as Python, Java, and R [25].

The ESP Platform. ESP is an open-source research platform for heterogeneous SoC design [33]. ESP combines a scalable architecture and a flexible methodology [9]. The tile-based architecture includes accelerator tiles, processor tiles, memory tiles (each containing a channel to main memory), and I/O tiles. The methodology supports several flows for the design of hardware accelerators, and simplifies the integration of third-party accelerators [16]. The processor tiles can contain one of the following cores: RISC-V 64-bit CVA6 [45], SPARC 32-bit LEON3 [15], and RISC-V 32-bit Ibex [31]. Each tile has a socket that interfaces it with a multi-plane packet-switched network-on-chip (NoC) with a 2D-mesh topology.

Related work. Hardware/software co-design techniques help achieve broader exploration of the design space at the system level and better implementations in terms of performance and energy efficiency [5, 6, 17, 36, 47]. This approach has been adopted in several works on a variety of application domains. In edge computing, co-design approaches have already been used in the implementations of several applications including image processing [23], multimedia and signal processing [4, 20], smart-networks [10], mapping for navigation [19, 32], and more. The booming world of AI and machine learning is also taking advantage of this approach [6, 24, 34, 40], and there is a strong trend in adding intelligence into the edge [6]. Our work can be leveraged by many projects in these fields that use embedded systems for computational problems at the edge.

3 SOFTWARE ARCHITECTURE

Fig. 3 presents the software architecture of EigenEdge. The design is based on three main components: (1) a C++ shared library, (2) a C interface, and (3) device drivers to control hardware accelerators at runtime. The software architecture in Fig. 3 implements a link between the C application and the C++ shared library, and two other links with the device drivers. All the links are created at compilation time.

The C++ Shared Library. As part of the EigenEdge software architecture (Fig. 3) we created a shared library (`eigen_run.so`). This library is the key to run any software application that uses

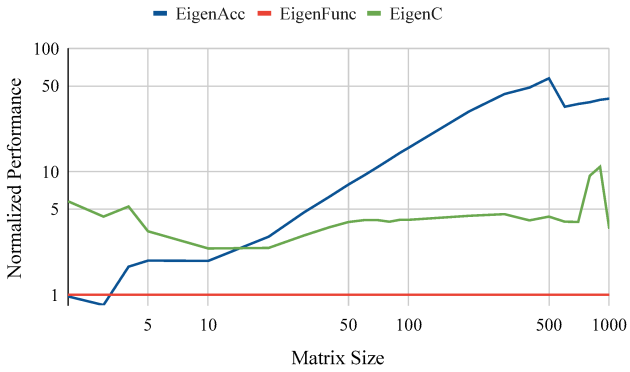

Fig. 4: Performance of matrix-matrix multiplications.

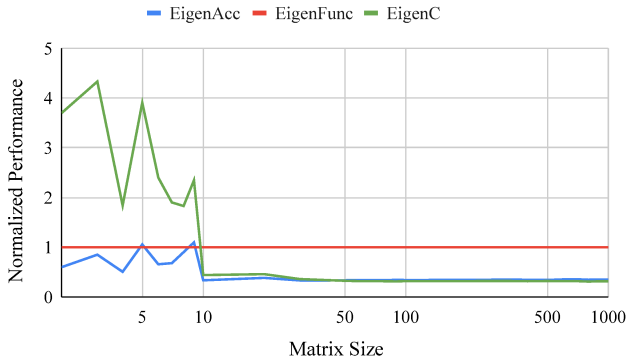

Fig. 5: Performance of matrix-vector multiplications.

Table 1: System setup for the experiments

Processor	CVA6, RISC-V, 64-Bit, 78 MHz
L1 Cache	ICache: 16KB, DCache: 32KB
Accelerator	1 General Matrix Multiplication (gemm)
Memory	2.5 GB DDR4-2400 memory
Evaluation Board	proFPGA Virtex UltraScale XCVU440
Operating System	Linux v4.20.0
HLS Tool	Cadence Stratus 20.25
Synthesis Tool	Xilinx Vivado 2019.2

Eigen on an embedded edge device. In EigenEdge, the applications simply need to be added to the shared library as new functions, and declared within an `extern "C"` linkage symbol that allows them to be accessed by a function call from a C application (Fig. 3). The `extern "C"` symbol is necessary because parts of the software architecture are written in C, while the applications are written in C++. The C parts are compiled with a `gcc` compiler while the C++ parts are compiled with a `g++` compiler. This required a non-trivial solution for linking the different parts of the project.

The shared library includes *Eigen* and allows any application to use all the features and methods *Eigen* provides. However, the shared library contains the implementation of customized operators (product, addition, subtraction, division, etc.) that overload the built-in operators in *Eigen*. The new operators interface with *Eigen* data types as matrices and vectors, extract their parameters, and establish the connection with the hardware accelerators on the SoC. The operators are able to configure the relevant hardware accelerators according to the specific task by having a variable access to the C interface (Fig. 3). The operators also invoke the accelerators through function calls to the device drivers. This feature is enabled by the link between the ESP device drivers library (`esp_driver.a`) and `eigen_run.so`. Specifically, we used the `extern "C"` symbol inside `eigen_run.so` to declare a linkage to specific functions inside `esp_driver.a`, so the operators could call these functions.

The C Interface. As shown in Fig. 3, the EigenEdge architecture includes a C interface. This interface can execute the applications inside `eigen_run.so` through regular function calls, thanks to the mentioned `extern "C"` linkage that was used in the declarations of the functions inside the shared library.

The C interface allocates the memory space for each accelerator in the SoC and initializes the data structures that hold the parameters to configure each accelerator. Base memory addresses and pointers to the parameters inside the data structures are passed as arguments to the C++ functions in `eigen_run.so`, and accessed later by the device drivers when the accelerators are invoked (Fig. 3). When a function from `eigen_run.so` is called, the addresses and the pointers are stored in global variables. Following that, the relevant operators that are used in the applications can dynamically configure and invoke the accelerators with the function calls to the device drivers.

The Device Drivers. The ESP platform [33] provides the infrastructure to interface with hardware accelerators through Linux device drivers. The library that includes the device drivers is implemented in C and exposes the ESP runtime API that allows software applications to configure and invoke the accelerators through function calls. For EigenEdge, we did not need to modify the device drivers but only link them properly to the C++ shared library (`eigen_run.so`), and integrate the function calls within the customized operators.

In addition, ESP provides optimized hardware accelerators which were utilized in our experiments (Section 4). As mentioned, these accelerators are invoked transparently in EigenEdge thanks to the specialized operators that we added to the shared library.

4 EXPERIMENTAL RESULTS

Experimental Setup. We designed a heterogeneous SoC by using ESP. We used an FPGA-based experimental setup with the main characteristics summarized in Table 1. The SoC is equipped with a RISC-V CVA6 processor, a memory channel, an I/O tile, and a general matrix-matrix multiplication accelerator (`gemm`) provided from ESP. The accelerator was designed in synthesizable SystemC by using Cadence Stratus HLS 20.25.

The complete SoC was synthesized with Xilinx Vivado 2019.2 with a target clock frequency of 78 MHz. We deployed the design on a Xilinx Virtex Ultrascale XCVU440 FPGA board. All evaluations reported in this section were performed by implementing custom Linux software applications that follow the EigenEdge approach and use the same software structure described in Section 3. We first implemented unit tests to investigate the effect of different sizes of matrix multiplications on the overall performance and energy efficiency. We then extended this concept to a real-world algorithm: Extended Kalman Filter [21], whose computation also relies on matrix multiplications.

Table 2: Power consumption in the SoC

CPU	Accelerator	Memory	I/O
0.2W	0.077W	0.064W	0.089W

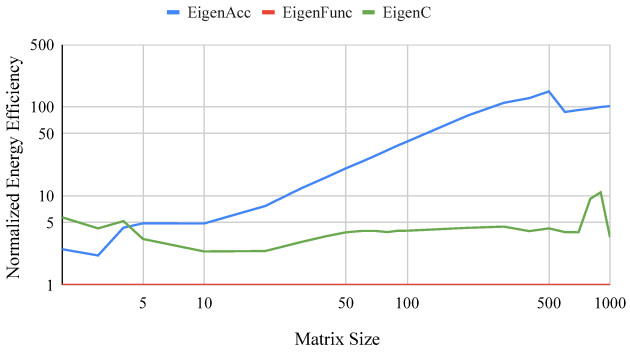

Fig. 6: Energy efficiency of matrix-matrix multiplications.

General matrix-matrix multiplication. In linear algebra, Matrix multiplication is used in a wide range of computations. Given two matrices $A_{m\times n}$ and $B_{n\times p}$, their product is the matrix $C_{m\times p} = A*B$, where each element C_{ij} is the sum of the dot products between elements $\{A_{ik}\}_{k\in n}$ and $\{B_{kj}\}_{k\in n}$. The general algorithm for matrix multiplication involves a triple nested loop that iterates over the rows of A, the columns of B, and the multiplied elements. Loops are good targets for optimizations during hardware design with high-level synthesis tools, which can unroll and pipeline their iterations as well as parallelize the execution of the non-dependant parts of the computation.

We evaluated the performance of the EigenEdge hardware/software co-design approach by implementing *Eigen* applications with overloaded operators that can invoke the gemm accelerator from ESP. Specifically, we performed tests under three different scenarios:

- `EigenFunc`: The program runs with the original *Eigen* library, where the matrix multiplication is done with the built-in *Eigen* operator for matrix multiplication.
- `EigenC`: The program runs with a custom operator for the product between two matrices, that implements the product in a native C fashion, and overloads the built-in *Eigen* operator.
- `EigenAcc`: The program runs with a custom operator that overloads the built-in *Eigen* operator, but offloads every matrix multiplication to the specialized gemm accelerator.

The execution in `EigenFunc` and `EigenC` depends only on the performance of the software running on the CVA6 processor, while `EigenAcc` offloads most of the computation to the gemm accelerator instead. In each of the above scenarios, we designed tests that perform matrix multiplications with varying sizes of A and B. All tests were written with the original *Eigen* syntax, but the operators were used according to the scenario (`EigenFunc`, `EigenC`, `EigenAcc`). The dimensions of the matrices $A_{m\times n}$ and $B_{n\times p}$ were set to $m = n = p = size$, while the program sweeps all *size* values in the range of $\in \{1, 1000\}$.

Fig. 4 shows the normalized performance in logarithmic scale of `EigenAcc` and `EigenC` with respect to `EigenFunc`. As the size of the matrices increase, `EigenAcc` outperforms the scenarios that depend on the performance of the CVA6 processor alone (`EigenFunc` and `EigenC`). This confirms that the specialized architecture of the accelerator, which is based on parallelism of the dot products, is beneficial for large-size matrices. `EigenAcc` provides 57× maximum speedup with respect to `EigenFunc` with $size = 500$ and 11× maximum speedup with respect to `EigenC` with $size = 900$. For small size matrices ($4 \leq size \leq 15$) `EigenAcc` is outperformed by `EigenC`, but still provides a maximum speedup of 2.3× with respect to `EigenFunc`. For even smaller matrices ($size < 4$), `EigenAcc` performs worse than the built-in `EigenFunc`. The reason for the slowdowns with small matrices is the fact that the computation becomes negligible compared to the software overhead that is required to invoke the accelerator.

Table 3: EKF latency and normalized energy efficiency

Test run	Latency	Energy Efficiency
`EigenFunc`	88.78 sec	1.00
`EigenC`	50.59 sec	1.75
`EigenAcc`	81.74 sec	2.82

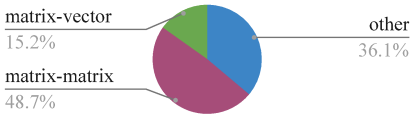

Fig. 7: Runtime profiling of the *EKF C++* application.

In a similar fashion, we repurposed the gemm accelerator to perform matrix-vector multiplications and did a similar sweep on the matrix sizes as before. As shown in Fig 5, the `EigenAcc` did not provide meaningful speedup with respect to the others, and for large-size matrices, `EigenFunc` provided the best performance. We assume that the reason for this result is the design of the gemm accelerator, which is not optimized for accelerating matrix-vector multiplications, but only matrix-matrix multiplications. The gemm accelerator prioritizes reusing the same values from the matrix for several computations and parallelizes them, while in matrix-vector computation each value of the matrix is used only once.

Table 2 reports the FPGA power consumption of each tile, including CPU, memory, I/O, and the gemm accelerator, which was obtained from the post-synthesis reports of Xilinx Vivado. The energy efficiency is defined as the reciprocal of the multiplication between power and latency. Fig. 6 shows the normalized energy efficiency in logarithmic scale of `EigenAcc` and `EigenC` with respect to `EigenFunc`. `EigenAcc` achieved a maximum of 125× gain in energy efficiency for $size = 400$, while `EigenC` achieved a maximum 11× for $size = 900$. `EigenAcc` provides better energy efficiency than `EigenC` as the matrix size grows, starting with a size equal to 3.

Extended Kalman Filter (EKF). We also tested our approach on real-world applications that can be implemented on edge devices. EKF is a non-linear extension of the Kalman Filter, which is a widely used algorithm in linear systems [22]. EKF is used in various applications, including satellite tracking, target tracking, and autonomous vehicle navigation. *EKF C++* is an open-source implementation of the EKF based in the *Eigen* library [7]. The application fuses LIDAR and RADAR sensor readings to estimate the location and velocity of a vehicle.

We profiled the implementation of the application and checked the relative runtime for operations of matrix-matrix multiplication, matrix-vector multiplication, and others. As shown in Fig. 7, the results of the profiling confirm that matrix-matrix multiplication is the most used operator, taking almost 49% of the total runtime of the application.

We chose to integrate the *EKF C++* application into the EigenEdge shared library (Section 3) and evaluated it on FPGA. We investigated the same three scenarios as we did for the matrix multiplication (`EigenFunc`, `EigenC`, and `EigenAcc`), and focused on performance and energy efficiency. The results are summarized in Table 3. They show that `EigenAcc` provides a latency improvement of 9.2% when compared to the original (`EigenFunc`). Even though `EigenC` achieves a greater latency improvement of 43%, the normalized energy efficiency of `EigenAcc` is 61% better than the one for `EigenC` and 180% better than `EigenFunc`. The slowdown with `EigenAcc` can be explained by the fact that the *EKF C++* application uses only small matrices, with the largest size being 4×4. According to our analysis in Fig. 4, we expected `EigenC` to provide a lower latency. However, as Fig. 6 suggested, `EigenAcc` was expected to provide better energy efficiency even for small-size matrices. Hence, the case study of *EKF C++* application confirms our prior analysis reported in Fig. 4, while the better energy efficiency of `EigenAcc` compared to `EigenC`, even for small-size matrices, confirms the prior analysis reported in Fig. 6.

5 FUTURE WORK

In Section 4 we showed that EigenEdge can potentially provide significant gains in performance and energy-efficiency for the case of multiplying large-size matrices. In our case study, the *EKF C++* application uses relatively small matrices, and did not maximize the potential of the EigenEdge approach. Applications that use larger-size matrices, such as ones for graphics and computer vision, may be a better fit for EigenEdge.

EigenEdge was built to be scalable and adaptable for future needs. The operators described in Section 3 can call any accelerator that is present in the SoC, such as ones for Fast Fourier Transform (FFT), Discrete Wavelet Transform (DWT), Singular Value Decomposition (SVD), 2D-Convolution, and more. The accelerators can be integrated into the SoC by using ESP, along with *Eigen* software applications that utilize these kernels. We expect to extend EigenEdge with more operators that invoke hardware accelerators and replace the built-in CPU-centric operators from *Eigen*.

Furthermore, complex tasks can be split into multiple kernels, and run in parallel on multiple specialized accelerators by using a single operator or method. This will yield further performance and energy efficiency gains for a variety of computationally-intensive and power-hungry applications, enabling their execution in real-time on lightweight edge devices.

6 CONCLUSION

We presented EigenEdge, a software approach that enables hardware/software co-design to balance low-latency requirements and low-power constraints for real-time computation at the edge. The seamless development flow of EigenEdge combines a simple API, a software architecture, and the access to efficient hardware accelerators in heterogeneous SoCs. EigenEdge keeps the system-level integration hidden from the application level, which maintains abstraction, and supports the promotion of further research and development of applications for embedded systems.

Acknowledgments. This work was supported in part by DARPA and in part by the NSF (A#: 1764000). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory and DARPA or the U.S. Government.

REFERENCES

[1] GTSAM: a sensor fusion library for robotics and vision. 2023.
[2] Azzam Alhussain et al. 2022. Hardware-Efficient Deconvolution-Based GAN for Edge Computing. In *Proc. of CISS*.
[3] Seonmyeong Bak et al. 2022. OpenMP Application Experiences: Porting to Accelerated Nodes. *Parallel Comput.* (2022).
[4] Soumya Basu et al. 2018. Heterogeneous and Inexact: Maximizing Power Efficiency of Edge Computing Sensors for Health Monitoring Applications. In *Proc. of ISCAS*.
[5] Beatriz Blanco-Filgueira et al. 2019. Deep Learning-Based Multiple Object Visual Tracking on Embedded System for IoT and Mobile Edge Computing Applications. *IoT-J* (2019).
[6] Oliver Bringmann et al. 2021. Automated HW/SW Co-Design for Edge AI: State, Challenges and Steps Ahead. In *Proc. of CODES*.
[7] Extended Kalman Filter C++. 2017. https://github.com/mez/extended_kalman_filter_cpp
[8] Maurizio Capra et al. 2019. Edge Computing: A Survey on the Hardware Requirements in the Internet of Things World. *Future Internet* (2019).
[9] Luca P. Carloni. 2016. The Case for Embedded Scalable Platforms.
[10] Yao-Chung Chang et al. 2018. Campus Edge Computing Network Based on IoT Street Lighting Nodes. *IEEE Systems Journal* (2018).
[11] QUIT: Processing Quantitaive MRI Data. 2022. https://github.com/spinicist/QUIT
[12] Robert David et al. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. In *Proc. of MLSys*.
[13] Guy Eichler et al. 2021. MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces. In *Proc. of ICCD*.
[14] IFOPT: Interface for Nonlinear Optimizers. 2022. https://github.com/ethz-adrl/ifopt
[15] SPARC LEON3 Cobham Gaisler. 2011. www.gaisler.com/index.php/products/processors/leon3
[16] Davide Giri et al. 2021. Accelerator Integration for Open-Source SoC Design. *IEEE Micro* (2021).
[17] Cong Hao et al. 2021. Enabling Design Methodologies and Future Trends for Edge AI: Specialization and Codesign. *IEEE Design & Test* (2021).
[18] Jude Haris et al. 2021. SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference. In *Proc. of SBAC-PAD*.
[19] Yuquan He et al. 2021. Picovo: A Lightweight RGB-D Visual Odometry Targeting Resource-Constrained IoT Devices. In *Proc. of ICRA*.
[20] Maher Jridi et al. 2018. SoC-Based Edge Computing Gateway in the Context of the Internet of Multimedia Things: Experimental Platform. *JLPEA* (2018).
[21] Simon J. Julier et al. 1997. New Extension of The Kalman Filter to Nonlinear Systems. In *Proc. of AEROSENSE*.
[22] Rudolph Emil Kalman. 1960. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering* (1960).
[23] W Kayankit et al. 2009. Hardware/Software Co-Design for Line Detection Algorithm on FPGA. In *Proc. of ECTI-CON*.
[24] Hamed F Langroudi et al. 2019. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge. *arXiv* (2019).
[25] Eigen Other Languages. 2022. http://eigen.tuxfamily.org/index.php?title=FAQ#Other_languages
[26] Cheng Li et al. 2020. The Design and Implementation of a Scalable Deep Learning Benchmarking Platform. In *Proc. of CLOUD*.
[27] En Li et al. 2019. Edge AI: On-demand Accelerating Deep Neural Network Inference via Edge Computing. *TWC* (2019).
[28] CGAL: Computational Geometry Algorithms Library. 2022. https://www.cgal.org/
[29] Eigen C++ Library. 2010. http://eigen.tuxfamily.org
[30] Quantum++: A Quantum Computing Library. 2022. https://github.com/softwareQinc/qpp
[31] RISC-V Ibex lowRISC. 2022. https://github.com/lowRISC/ibex
[32] Dipan Kumar Mandal et al. 2019. Visual Inertial Odometry at the Edge: A Hardware-Software Co-Design Approach for Ultra-Low latency and Power. In

Proc. of DATE.
[33] P. Mantovani et al. 2020. Agile SoC Development with Open ESP. In *Proc. of ICCAD.*
[34] Matías Mendieta et al. 2019. A Novel Application/Infrastructure Co-Design Approach for Real-Time Edge Video Analytics. In *Proc. of SoutheastCon.*
[35] Eigen Projects. 2023. https://eigen.tuxfamily.org/index.php?title=Main_Page#Projects_using_Eigen
[36] Adrian Sampson et al. 2015. Hardware-Software Co-Design: Not Just a Cliché. In *Proc. of SNAPL.*
[37] Changyang She et al. 2019. Cross-Layer Design for Mission-Critical IoT in Mobile Edge Computing Systems. *IoT-J* (2019).
[38] ROS: Robot Operating System. 2022. https://www.ros.org/
[39] Eigen Backend Inside Tensorflow. 2023. https://github.com/tensorflow/tensorflow/tree/5dcfc51118817f27fad5246812d83e5dccdc5f72/third_party/eigen3
[40] Kizheppatt Vipin. 2019. ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms. In *Proc. of ICFPT.*
[41] Xiaying Wang et al. 2020. An Accurate EEGnet-Based Motor-Imagery Brain-Computer Interface for Low-Power Edge Computing. In *Proc. of MeMeA.*
[42] Ming Xia et al. 2021. SparkNoC: An Energy-Efficiency FPGA-Based Accelerator Using Optimized Lightweight CNN for Edge Computing. *JSA* (2021).
[43] Huihui Xue et al. 2020. Edge Computing for Internet of Things: A Survey. In *Proc. of iThings.*
[44] Wei Yu et al. 2017. A Survey on the Edge Computing for the Internet of Things. *IEEE Access* (2017).
[45] Florian Zaruba et al. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *ITVL* (2019).
[46] Mi Zhang et al. 2020. Deep Learning in the Era of Edge Computing: Challenges and Opportunities. *Fog Computing: Theory and Practice* (2020).
[47] Zongwei Zhu et al. 2019. A Hardware and Software Task-Scheduling Framework Based on CPU+ FPGA Heterogeneous Architecture in Edge Computing. *IEEE Access* (2019).