Dataflow Accelerator Architecture for Autonomous Machine Computing (Invited Paper)

Shaoshan Liu
Shenzhen Institute of Artificial Intelligence and Robotics for Society
China

Yuhao Zhu
University of Rochester
USA

Bo Yu*
Shenzhen Institute of Artificial Intelligence and Robotics for Society
China

Jean-Luc Gaudiot
University of California, Irvine
USA

Guangrong Gao
University of Delaware
USA

ABSTRACT

Commercial autonomous machines is a thriving sector, one that is likely the next ubiquitous computing platform, after Personal Computers (PC), cloud computing, and mobile computing. Nevertheless, a suitable computing substrate for autonomous machines is missing, and many companies are forced to develop *ad hoc* computing solutions that are neither principled nor extensible. By analyzing the demands of autonomous machine computing, this article proposes Dataflow Accelerator Architecture (DAA), a modern instantiation of the classic dataflow principle, that matches the characteristics of autonomous machine software.

KEYWORDS

Autonomous driving, data flow, computing architecture.

1 AUTONOMOUS MACHINE: THE NEXT UBIQUITOUS COMPUTING PLATFORM

Commercial autonomous machines is a thriving sector. With a projected average compound annual growth rate (CAGR) of 26%, by 2030 this sector will have a market size of $1 trillion [1]. Autonomous machine is on the verge of becoming the next ubiquitous computing platform, after personal computers, cloud computing, and mobile computing.

The continuous proliferation of autonomous machines, however, depends critically on an efficient computing substrate, driven by higher performance requirements and the miniaturization of machine form factors. Despite recent advancements in autonomous machine systems design from major industrial organizations as Google [2], Tesla [3], Mobileye [4], Nvidia [5], the computing architecture of autonomous machines still remains a largely open research question. This is because completely independent teams have approached the problem, resulting in a bevy of solutions, some replications but certainly no consensus.

The fragmentation, while undesirable, is understandable in that different companies target autonomous machines in different forms (e.g., cars, aerial drones, service robots, and industrial robots), each naturally differing in missions, design constraints, computational capabilities, and mechanical characteristics [6]. A better understanding of the underlying issues, a formalization of the common problems, and a certain unification of the solutions would all yield a more efficient approach to the design process.

*The corresponding author.

This article first analyzes the challenges in designing an efficient computing substrate for autonomous machines, particularly focusing on why contemporary mobile Systems-on-a-Chip (SoC), a seemingly natural choice, is a bad fit. Surprisingly, classic dataflow architectures, while not enjoying practical adoption for general-purpose computing, are well-suited for autonomous machine workloads — in principle. We propose Dataflow Accelerator Architecture (DAA), a modern instantiation of the dataflow principle in the era of hardware specialization. We describe why DAA matches the characteristics of autonomous machine workloads, and discuss key technologies that could enable DAA as a mainstream computing substrate for autonomous machines.

2 SOFTWARE PIPELINE OF AUTONOMOUS MACHINES

From a thirty-thousand-foot view, the software pipeline continuously consumes data from a diverse array of sensors and produces control commands (e.g., brake, turn) to the actuator of the vehicle. Fig. 1 shows the software pipelines of two representative autonomous machines, a high-end L4 self-driving car (left) and a low-end home cleaning robot (right). They differ in implementation but follow the same principle. We use an L4 autonomous vehicle we developed as a running example to describe the software pipeline.

In order to generate the control commands, the control module, the last module of the pipeline, requires a navigation plan, i.e., a path, which is generated by the path planning module. To generate a path the planning module in turn requires two pieces of information: how the environment looks like and where the agent is in the environment. The former comes from the prediction module (which predicts the motion of surrounding objects) and the latter comes from the localization module. To predict the motion of an object, the prediction module relies on the past trajectory of the object, which comes from the tracking module. The tracking module fuses the perception results from various sensors such as Radar, cameras, and LiDAR.

For the vehicle to be responsive, the software stack must generate control commands at a *prescribed firing frequency* (PFF), e.g., 10 Hz in our L4 self-driving car here. This timing requirement at the output in turn translates to timing requirements at interior nodes in the *macro dataflow graph* (M-DFG) of the software pipeline. For instance, the planning task must execute at the PFF, since planning and control are pipelined; the 2D perception task usually executes

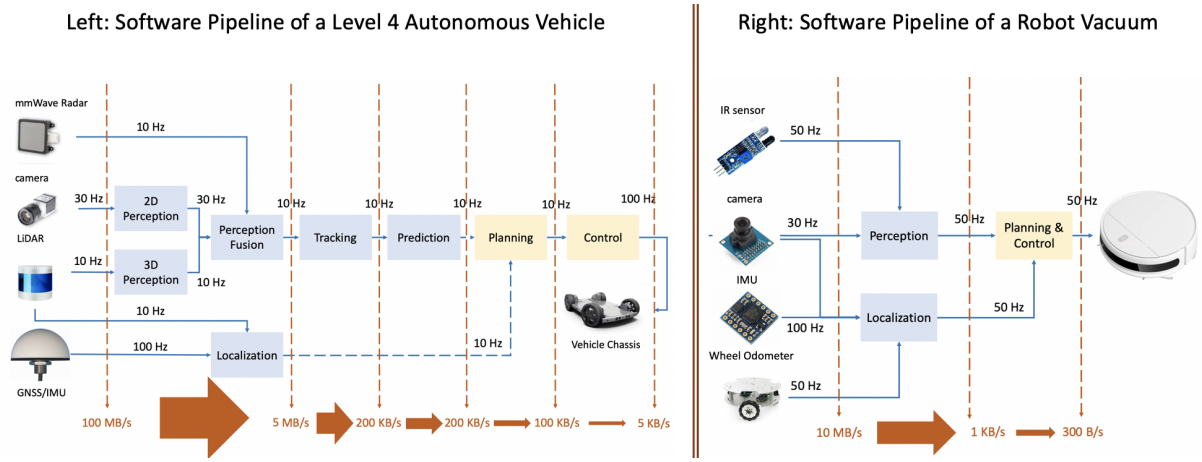

Figure 1: Left: the software pipeline of a level-4 autonomous vehicle. Right: the software pipeline of a robot vacuum. Other software organizations are possible.

at a frequency several-fold higher than the PFF, since the tracking module uses *a sequence of* 2D perception results to track objects.

3 WHY NOT USE COMMERCIAL SOCS?

Commercial (mobile) SoCs would have been an ideal computing substrate for autonomous machines. The incentive is two-fold. First, since mobile SoCs have reached economies of scale, it would have been beneficial to piggyback on such affordable, backward-compatible computing systems. Second, mobile SoCs integrate domain-specific accelerators, a.k.a., Intellectual Property (IP) blocks in semiconductor parlance, which are specialized for particular algorithm domains and provide large performance and energy-efficiency benefits. In fact, mobile SoCs have been demonstrated to support mobile robots [7, 8].

We initially explored the possibility of using high-end mobile SoCs, such as the Qualcomm Snapdragon [9] and Nvidia Tegra [10], to support autonomous driving workloads, but concluded that mobile SoCs are ill-suited for autonomous machines. Fig. 2 shows the results from one such effort using Nvidia TX2 [11], a typical mobile SoC [12].

In this design, we map the perception task to the GPU and the rest to the CPU with extensive use of SIMD instructions. Fig. 2a and Fig. 2b show the latency and energy consumption, respectively, of an Intel Coffee Lake CPU, an Nvidia GTX 1060 GPU, and the TX2, all executing the same software pipeline. Fig. 2a shows that TX2 is much slower than the GPU, leading to an excessively high end-to-end latency of about 1 second. The long latency also diminishes the energy benefits of mobile SoCs. Fig. 2b shows that TX2 consumes more energy over the GPU. Investigating the results reveals three sources of inefficiencies.

Lack of Specialization. The processing capability of mobile SoCs is too low for realistic autonomous machine workloads due to the lack of hardware specialization. This is perhaps not all that surprising given that mobile SoCs are not built with the autonomous machine software stack in mind. Today's SoCs integrate accelerators such as GPUs, Neural Processing Units, and Digital Signal Processors (DSP), but few, if any, are dedicated to algorithms used in autonomous machines. Most of the algorithms are mapped thus to the GPU in an SoC, which serve a "general-purpose" accelerator. Extensive prior work shows that the speed of an autonomous machine task can be accelerated by up to one of magnitude on a dedicated accelerator over a GPU [13, 14].

The lack of hardware specialization in commercial SoCs also presents a significant challenge for sensor synchronization, which is critical since autonomous machines fuse diverse sources of sensors. Commercial SoCs lack dedicated hardware for sensor synchronization and, as a result, often synchronize sensor data in software. Software synchronization either introduces a high synchronization latency (several milliseconds) or, worse, produces incorrect synchronization results as sensor data timestamps are not directly obtained in hardware [12, 15].

Inefficient Task Communication. Mobile SoCs do not optimize the communication between hardware accelerators. Instead, data communication requires redundant copy through the main memory, which introduces high time and power overhead for each sensor frame. For instance, when using DSP to accelerate image processing, the CPU has to explicitly copy images from sensor interface to DSP through the entire memory hierarchy [16]. On the Snapdragon 820 SoC, copying data through the CPU introduces a 3 ms latency overhead and a 1 W power overhead [17]. Even worse, since many robotic systems were built on top of managed run-time environments (e.g., Android), data copying could trigger garbage collections and stall the entire pipeline [18].

Centralized Task Coordination. Different accelerators are coordinated by the CPU. When a producer accelerator finishes its job, it sends an interrupt to the CPU. The CPU executes the corresponding interrupt service routine, which in turn triggers the driver of the consumer accelerator. The consumer driver, once again executes on the CPU, configures and invokes the consumer accelerator, essentially "relaying" the task from the producer to the consumer.

Frequently invoking the CPU prevents the CPU from entering deep sleep mode, leading to excessive energy waste. Our measurements show that the CPU services interrupts tens of times per

second, whereas the CPU needs to be idle for over 1 second for it to be beneficial to enter the deep sleep mode (due to the wake-up overhead). On average, the CPU could contribute to up to half of the SoC power, offsetting the energy efficiency of accelerators.

4 DATAFLOW ACCELERATOR ARCHITECTURE (DAA) TO THE RESCUE

The limitations call for a new architectural model, which we call the *dataflow accelerator architecture* (DAA). DAA has two ingredients. First, it incorporates a diverse set of domain specific accelerators to address the increasing performance requirement of ever more complicated algorithms. Second, DAA organizes accelerators in a dataflow fashion to remove the inefficiency of centralized coordination and communication. The dataflow organization of accelerators translates the orders of magnitude efficiency gains of individual accelerators to the end-to-end application.

This section first discusses the principles of classic dataflow architecture and how they inherently address the limitations of SoC architectures. We then discuss unique traits of the autonomous machine workloads that overcome conventional detractors that kept dataflow architectures of the past from gaining practical adoption.

4.1 From Dataflow to Dataflow Accelerator Architectures

Principles of Dataflow Architectures. Dataflow concepts originated in the 1970s and 1980s, with pioneering work by Jack Dennis, Arvind, and others [19, 20]. The central idea of dataflow architectures was to do away with the classic von Neumann architecture, where instructions are executed in the explicit order specified by the control flow. Control flows limit the window in which the instruction-level parallelism (ILP) can be exploited, presenting an artificial performance roadblock. In a dataflow architecture, execution is data-driven in that an instruction, in principle, executes as soon as all its inputs are available rather than when the control flow gets to it.

Dataflow architectures rely on the dataflow graph (DFG), which captures the data dependencies and, thus, the full ILP in a program. In a DFG, each node is an instruction and edges represent direct data communications between instructions. By explicitly expressing the data dependency among instructions, a DFG allows an instruction to fire whenever its operands are ready. It is worth noting that out-of-order superscalar processors that dominate today's high-performance CPU market are essentially restricted dataflow machines that exploit ILP in a small portion (as large as the instruction window can accommodate) of the DFG [21] [22].

Dataflow Accelerator Architecture. We see an analogy between the bottlenecks in conventional programs and those in autonomous machines software, both of which can be addressed by the dataflow principle. The key is to view the software stack of an autonomous machine, such as those in Fig. 1, as a macro dataflow graph (M-DFG), where each node represents a high-level task such as localization and motion planning. The granularity of each task (M-DFG node) is rather coarse, usually equivalent to billions of instructions when compiled to a conventional CPU ISA.

One could view the M-DFG as an ISA for the autonomous machine software stack, and view today's SoCs as an implementation

(a) Latency comparison. (b) Energy comparison.

Figure 2: Performance and energy comparison of three platforms running the same software pipeline. On TX2, we use the Pascal GPU for depth estimation and object detection and use the ARM Cortex-A57 CPU (with SIMD capabilities) for localization. Other mappings are explored, too, but result in worse performance and/or energy-efficiency.

of the ISA. This implementation maps each M-DFG node to an IP block. Just like how the speed of a conventional program is limited by the control flow, the M-DFG, when executed on an SoC, is also control-limited in that the IP blocks must be centrally coordinated by the CPU and communicated through the memory, as demonstrated in Section 3.

This realization gives rise to the notion of dataflow accelerator architecture, where accelerators directly communicate with each other through dedicated on-chip buffers and coordinate autonomously without the CPU's intervention. This architectural model has two benefits. First, it exposes higher levels of parallelisms with each accelerator (M-DFG node) firing whenever its input data are ready. Second, it accelerates the firing rates of M-DFG nodes by making operands more readily available to consumers; this is achieved by allowing producers and consumers to directly communicate using a per-accelerator on-chip buffer rather than through the main memory.

The DAA takes inspiration from the block variant of the dataflow architecture, which partitions a dataflow graph into blocks (a.k.a. megainstructions) and either 1) executes each block in a data flow manner and uses control flow across blocks [23] or 2) uses dataflow across blocks while executing each block following its control flow [24]. The DAA model has a similar flavor in that each M-DFG node is essentially a block. By removing the CPU coordination and allowing M-DFG nodes to directly communicate, DAA essentially enforces the dataflow model across blocks. However, DAA does not specify how each block is executed. A block could be mapped to a CPU, executing by following the control flow, or (more commonly) mapped to a dedicated accelerator that is designed to fully exploit the ILP exposed by the dataflow.

One might find similarities between the on-chip buffers in DAA and the so-called *system caches* in contemporary mobile SoCs such as the Apple's A-series and Qualcomm's Snapdragon-series SoC. The main difference is that a system cache is shared across all the SoC-generated traffics without management, whereas each communication buffer in a DAA is dedicated and optimized for a pair of producer-consumer communication. Note, however, that the physical implementation of the DAA buffers can either be distributed or centralized.

4.2 Opportunities for Autonomous Machine-Specific DAA

While traditional data flow architectures target general-purpose processing, we focus on designing and optimizing DAAs specifically for autonomous machine. The software stack of autonomous machines possesses four characteristics that can be leveraged to avoid common pitfalls of traditional data flow machines while retaining their main benefits.

Low Bookkeeping Overhead. The M-DFG has only a handful of nodes, so the bookkeeping overhead, such as tag boradcasting, matching, and storage overhead, is low. In contrast, performance of conventional data flow machines is sometimes bottlenecked by managing the tags.

Continuous Inputs Provide Abundant Parallelisms. Autonomous machine workloads provide abundant parallelisms and, thus, do not typically starve the hardware. Many conventional programs do not have sufficient intrinsic parallelisms to fully utilize a realistic data flow hardware except when processing large arrays.

The abundant parallelism comes from the fact that inputs to autonomous machines are continuous: as an autonomous machine operates, various sensors continuously pump data frames to the hardware. Since frames are largely independent, autonomous machine software exposes input-level parallelism that is unavailable in programs that conventional data flow machines target.

Flexible Dependencies. Conversely, conventional data flow machines are also limited when a program has too much parallelism, which requires throttling mechanisms that degrade performance. Autonomous machine workloads, in contrast, have flexible dependencies in that a consumer node, while dependent on a producer node, can afford to drop data and operate on the latest data from producers. As a result, parallelism explosion is generally not a concern.

The reason behind the flexible dependencies in autonomous machine workloads has to do with the real-time nature of the workload. Autonomous machines must operate in real time; thus, each node in the DFG has a prescribed output frequency. For instance, the image perception node processes images at 30 FPS; the LiDAR perception node processes point clouds at 10 FPS. Fig. 1 annotates each node with a firing frequency. As a result, engineers intentionally design algorithms such that a node does not block if its firing time is reached. For instance, the planning algorithm fetches the latest produced data, essentially dropping previously produced data; the localization algorithms consumes a sequence of frames such that missing one frame of data is not catastrophic.

No Loops. The M-DFG of autonomous machines is a directed acyclic graph (DAG) and the nodes within the DAG are stateless. The M-DFG nodes are thus naturally non-reentrant, eliminating the notorious difficulty to deal with reentracy in conventional data flow architectures [25, 26].

4.3 Programming the DAAs

An architectural model is only useful when it is easy to program (and, by extension, compile for). One main detractor that kept general-purpose data flow architectures from gaining practical adoption is that they target only functional programming languages [27], which naturally exposes parallelism, while providing poor support for imperative languages.

(a) Kalman gain. (b) Marginalization.

Figure 3: Latencies of VIO's Kalman filter and marginalization blocks are correlated with the environment an agent operates in (represented by the number of visual feature points here).

For DAA, however, programming model no longer presents a roadblock. This is because software developers for autonomous machines are already (implicitly) programming in a functional-style through the widely used Robot Operating System (ROS) [28]. ROS is a programming interface and run-time system for programming autonomous machines. Among many other features, ROS exposes a publish-subscribe programming interface: each task is implemented as a ROS node, which subscribes to a set of messages sent from the publishers (i.e., producers); each node is triggered whenever the messages are delivered. ROS thus explicitly encourages expressing an autonomous machine application in a data-driven manner. One could directly generate the M-DFG of a given ROS application.

5 TIMING-SAFE DAAS FOR AUTONOMOUS MACHINES

In addition to improving the sheer performance and energy efficiency, DAA also provides a foundation for guaranteeing timing safety, i.e., meeting the (soft) real-time requirement, for autonomous machines, which challenges contemporary SoCs. This section discusses why the DAA model is a desirable substrate, followed by mechanisms that DAAs incorporate for timing safety.

5.1 The Problem

The real-time requirement of an autonomous machine application translates to meeting the prescribed firing frequency of each M-DFG node. Today's the computing systems provide a best-effort delivery of real-time requirement without providing any guarantees. In ROS, for instance, each task is annotated with an expected firing frequency, which, however, is not guaranteed. We routinely see violations of the POF. For instance, in an early mobile robot deployment, while the localization frequency was expected to be 30 FPS, the achieved frame rate was on average only 20 FPS and could vary by as much as a factor of five [17].

The primary source of timing violation in SoCs is that the CPU and memory communication are known to present non-deterministic

latency, susceptible to many sources of variability in the system [7]. DAA naturally addresses this issue. DAA does not rely on CPU for task coordination and memory communication and, thus, is large free from these variabilities.

Taking the CPU and memory sideways mitigates the inter-task variabilities, but the intra-task latency, i.e., the execution latency of each task, could be unpredictable for two primary reasons. First, a task's latency varies naturally with its micro-architectural implementation. Second, the environment in which a vehicle operates in changes dynamically, posing changing program inputs and computation requirements. Taking visual inertial odometery (VIO), a particular localization algorithm, as an example, Fig. 3 demonstrates that the latency of VIO's building blocks, such as Kalman filter and marginalization, varies widely but strongly correlates with the environment a vehicle operates in (represented by the number of visual feature points detected, G axis) [29].

5.2 Synthesising and Dynamically Optimizing Accelerators with Timing Guarantees

We develop a framework called Archytas to tame the intra-node variability [29]. Architectures work the best when the division of labor between the static and run-time systems exploits the strengths and limitations of each. Guided by this principle, Archytas integrates two components. First, a static synthesizer generates hardware accelerators to provide clean timing specifications for each task, describing clearly the per-task latency/throughput under different input conditions. Second, the run-time system reasons about the end-to-end latency using the task-level timing specification, generating timing-safe execution policies even in dynamically changing environments. While currently targeting the localization task as a case study, we believe it generalizes to the broad autonomous machine domain. Fig. 4 shows the Archytas framework.

Static Synthesis. To reason about and guarantee timing of the entire M-DFG, we envision that each M-DFG node, when mapped to hardware, provides a latency description under different input conditions. This is achieved through a generative approach, where we synthesize an accelerator for each M-DFG node given a specific latency target (with potential power constraints). This guarantees per-task timing specification by construction.

Archytas analyzes an algorithm to identify a set of key architectural knobs that significantly influence the overall latency of the task. Archytas builds an analytical latency and power model of the entire algorithm parameterized by the architectural knobs. While the latency of traditional general-purpose processors is hard to model, it is much easier to do so for accelerators due to the simpler (e.g., shallower, software-managed memory hierarchy) and more specialized design.

Using the model, hardware generation becomes a principled constraint optimization problem. A concrete, Pareto-optimal localization accelerator (in the form of synthesizable Verilog code) can be generated in just a few seconds (compared to months or even years if the design space is to be exhaustively searched) given a timing specification and power/energy constraints. Fig. 5 shows the measured power vs. time of the Pareto-optimal accelerators generated by our constraint optimization (squares), which do indeed Pareto-dominate other designs (circles), suggesting the feasibility of generating accelerators with precise timing guarantees.

Dynamic Optimization. In dynamic environments with changing computation requirements, a purely statically-synthesized accelerator faces one of the two challenges. It either has to overprovision the hardware resources to accommodate the worst-case performance requirement, which unnecessarily wastes power in most of the time; alternatively, it could provide only an average-case guarantee without the ability to adapt to run-time dynamism.

The Archytas approach is to generate an accelerator that accommodates the average case, but couple that with a run-time system, which dynamically scales the hardware configuration to adapt to the changing workload at run time. The run-time system first detects the workload changes (e.g., using heuristics such as the number of visual features detected), and then dynamically scale up (down) the hardware capability when the workload increases (decreases). The actual mechanisms to change the hardware capability could range from light-weight clock gating to more involved techniques such as partial reconfiguration (on an FPGA platform).

More Advanced Roles. In addition to the central roles described above, the compiler and run-time systems can also assume a variety of other roles, across different stages in the development and deployment cycle, when delivering timing safety. For instance, the compiler could reject timing-unsafe M-DFGs by checking the timing requirement from the autonomous machine against the timing specifications of each M-DFG node. A similar approach is a type system for constant-time cryptographic programming [30].

In addition, the compiler and run-time system can cooperate to ensure timing portability, i.e., guaranteeing timing-safety when the same code base is ported to a wide range of hardware with dramatically different computation and memory resources. The portability is critical given the enormous range of autonomous system capabilities (e.g., from home robotic vacuums all the way to autonomous cars trucks).

Consider, for example, on-chip buffer allocation and organization, which must be done in a way that does not block accelerator execution with minimal on-chip memory utilization. The compiler could exploit the observation that, under a given a set of sensors and sensing rates, the communication volume across nodes is largely deterministic. As shown in Fig. 1, the sensing module ships about 100 MB/s of data to the perception module but the output to the vehicle is only about 5 KB/s, comprised of simple actuation commands. The compiler could leverage this pattern to allocate just enough buffer space, and the run-time system could dynamically decide when and how to drop data frames to avoid timing violations (excessive stalls).

6 CONCLUSION

DAA offers a promising architectural model as the computing substrate for autonomous machines. Guided by the classic data flow principle and embracing hardware specialization, DAA removes the central limitations of contemporary SoCs when executing autonomous machine workloads, and provides a desirable foundation for ensuring timing safety. DAA is also an extensible model, where new software components and hardware accelerators can be easily incorporated. The extensibility allows the DAA to potentially consolidate the fragmented hardware design space for autonomous machines.

Figure 4: Archytas system overview. Archytas first generates a task graph from a high-level algorithm description. The task graph is then mapped to a parameterized hardware template, which is concretized by the hardware synthesizer (in the form of synthesizable Verilog code) given the latency and power specifications. The run-time system continuously re-optimizes the hardware according to the operating environment.

Figure 5: Latency-vs-power Pareto optimal frontier of different localization accelerators generated by Archytas.

REFERENCES

[1] Statista. Size of the global market for industrial and non-industrial robots between 2018 and 2025. *https://www.statista.com/statistics/760190/worldwide-robotics-market-revenue/*, 2021.

[2] Waymo. Introducing waymo's suite of custom-built, self-driving hardware. *https://medium. com/waymo/introducing-waymos-suite-of-custom-built-self-driving-hardware-c47d1714563*, 2019.

[3] Tesla. Tesla autopilot: Full self-driving hardware on all cars. *https://www.tesla.com/autopilot*, 2021.

[4] Mobileye. True redundancy. *https://www.mobileye.com/true-redundancy/*, 2021.

[5] Nvidia. Nvidia drive sim ecosystem creates diverse proving ground for self-driving vehicles. *https://blogs.nvidia.com/blog/category/auto/*, 2021.

[6] Shaoshan Liu and Jean-Luc Gaudiot. Rise of the autonomous machines. *Computer*, 55, 2022.

[7] Liu Liu, Jie Tang, Shaoshan Liu, Bo Yu, Yuan Xie, and Jean-Luc Gaudiot. π-rt: A runtime framework to enable energy-efficient real-time robotic vision applications on heterogeneous architectures. *Computer*, 54(4):14–25, 2021.

[8] Shaoshan Liu, Jie Tang, Zhe Zhang, and Jean-Luc Gaudiot. Computer architectures for autonomous driving. *Computer*, 50(8):18–25, 2017.

[9] Qualcomm. Snapdragon mobile platform. *https://www.qualcomm.com/snapdragon*.

[10] NVIDIA. Nvidia tegra. *https://www.nvidia.com/object/tegra-features.html*.

[11] NVIDIA. Nvidia jetson tx2 module. *https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/*.

[12] Bo Yu, Wei Hu, Leimeng Xu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Building the computing system for autonomous micromobility vehicles: Design constraints and architectural optimizations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1067–1081. IEEE, 2020.

[13] Yiming Gan, Bo Yu, Boyuan Tian, Leimeng Xu, Wei Hu, Shaoshan Liu, Qiang Liu, Yanjun Zhang, Jie Tang, and Yuhao Zhu. Eudoxus: Characterizing and accelerating localization in autonomous machines industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 827–840. IEEE, 2021.

[14] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. Navion: a fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones. In *2018 IEEE Symposium on VLSI Circuits*, pages 133–134. IEEE, 2018.

[15] Shaoshan Liu, Bo Yu, Yahui Liu, Kunai Zhang, Yisong Qiao, Thomas Yuang Li, Jie Tang, and Yuhao Zhu. Brief industry paper: The matter of time—a general and efficient system for precise sensor synchronization in robotic computing. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 413–416. IEEE, 2021.

[16] Praveen Yedlapalli, Nachiappan Chidambaram Nachiappan, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. Short-circuiting memory traffic in handheld platforms. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–177. IEEE, 2014.

[17] Jie Tang, Bo Yu, Shaoshan Liu, Zhe Zhang, Weikang Fang, and Yanjun Zhang. π-soc: Heterogeneous soc architecture for visual inertial slam applications. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8302–8307. IEEE, 2018.

[18] Zhe Zhang, Shaoshan Liu, Grace Tsai, Hongbing Hu, Chen-Chi Chu, and Feng Zheng. Pirvs: An advanced visual-inertial slam system with flexible sensor fusion and hardware co-design. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–7. IEEE, 2018.

[19] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1974.

[20] Arvind, Vinod Kathail, and Keshav Pingali. A dataflow architecture with tagged tokens. *Tech. Rep.*, 1980.

[21] Yale N Patt, Wen-mei Hwu, and Michael Shebanow. Hps, a new microarchitecture: Rationale and introduction. *ACM SIGMICRO Newsletter*, 16(4):103–108, 1985.

[22] Subbarao Palacharla, Norman P Jouppi, and James E Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 206–218, 1997.

[23] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[24] Shuichi Sakai, Y Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. *ACM SIGARCH Computer Architecture News*, 17(3):46–53, 1989.

[25] Jack B Dennis. The mit data flow engineering model. In *Proc. IFIP Congress 83*, 1983.

[26] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[27] Richard S Bird and Phillip L Wadler. *Functional programming*. Prentice Hall, 1988.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[29] Weizhuang Liu, Bo Yu, Yiming Gan, Qiang Liu, Jie Tang, Shaoshan Liu, and Yuhao Zhu. Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization. In *2021 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2021.

[30] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.