Timing-Predictable Vision Processing for Autonomous Systems

Tanya Amert*, Michael Balszun†, Martin Geier†, F. Donelson Smith*, James H. Anderson*, Samarjit Chakraborty*
*University of North Carolina at Chapel Hill, USA, †Technical University of Munich, Germany
Email: {tamert, smithfd, anderson, samarjit}@cs.unc.edu, {michael.balszun, mgeier}@tum.de

***Abstract*—Vision processing for autonomous systems today involves implementing machine learning algorithms and vision processing libraries on embedded platforms consisting of CPUs, GPUs and FPGAs. Because many of these use closed-source proprietary components, it is very difficult to perform any timing analysis on them. Even measuring or tracing their timing behavior is challenging, although it is the first step towards reasoning about the impact of different algorithmic and implementation choices on the end-to-end timing of the vision processing pipeline. In this paper we discuss some recent progress in developing tracing, measurement and analysis infrastructure for determining the timing behavior of vision processing pipelines implemented on state-of-the-art FPGA and GPU platforms.**

I. Introduction

Modern autonomous systems – be autonomous vehicles or robots – consist of two major components: (a) a decision making unit, which is often made up of one or more feedback control loops, and (b) a perception unit that feeds the environmental state to the control unit and is made up of camera, radar and lidar sensors and their associated processing algorithms and infrastructure. The overall behavior and correctness of the system relies on the correct functioning of *both* these components. The notion of *correctness* is also interpreted as both *functional* as well as *timing* correctness. Both functional as well as timing verification of the decision making (or the control) unit has a rich literature. However, there is much less work on how to verify the functional and timing correctness of the perception unit. There are several reasons that can be attributed to this. First, progress in the domain of autonomous systems is fairly recent. Hence, it is only now that there is a major push towards their verification, which is a major prerequisite – and currently a major obstacle – for wide scale deployment. Second, it is only recently that architectures for vision processing algorithms have also become more complex, with accelerators like GPUs and TPUs being used in combination with multicore processors. Finally, the complexity of the algorithms for vision processing – with the use of machine learning techniques – have also increased only recently. These developments in both algorithmic and architectural complexity for vision processing have made both functional and timing verification extremely complex, and also all the more necessary.

In this paper we focus only on *timing analysis* for vision processing algorithms implemented on multicore + accelerator platforms. In particular, we outline our work on measurement and tracing for timing properties on FPGA- and GPU-based accelerators, broadly outline the underlying techniques and list some of the major open issues.

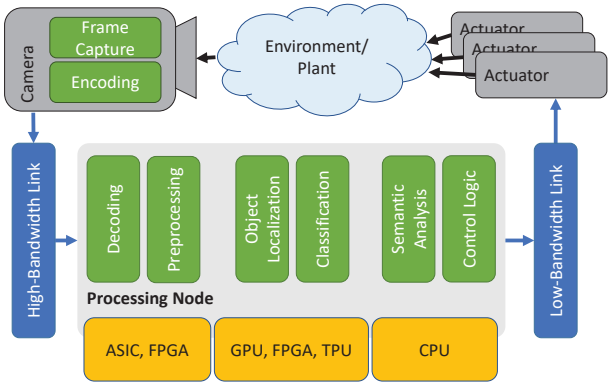


Fig. 1: Physical Plant with Sensors/Actuators (top) and Processing (bottom)

A. Real-time Image Processing Pipelines

Visual data serves two main purposes in autonomous systems: First, creating a semantic understanding of the world that the system is interacting with. This provides the basis for planning- and decision-making algorithms, and includes tasks like analyzing obstacles, recognizing street signs (to determine speed limits or analyzing surface conditions in order to adapt trajectories or movement speed), and, of course, identifying objects to be manipulated. Second, video data serves as real-time sensor input to many control algorithms while navigating through and interacting with the environment – to, e.g., determine the system's position and movement relative to its designated path. Of course, those categories may overlap, e.g., when an autonomous car needs to identify a child running towards the street in time to initiate a braking maneuver. Especially for the second class of applications, the timing behavior of each step in the processing and control pipeline must be well understood and be predictable in order to determine the worst-case end-to-end delay of the pipeline from the sensor acquisition to the corresponding actuator update – usually called the worst-case response time (WCRT) – and ultimately guarantee timely reactions to occurring events and stability of the system dynamics.

Fig. 1 gives an overview over the different stages found in a typical vision-based control loop (see [1] for more details). The physical world is captured by a camera, which creates a digital representation of the video frames according to one of different possible encoding and maybe compression schemes. This data is then transmitted via a communication link (either dedicated

978-3-9819263-5-4/DATE21/©2021 EDAA

or via a shared bus system) to the processing nodes. There, the data needs to be decoded, reassembled and preprocessed. While those operations are relatively cheap when compared to the common Deep Neural Networks (DNNs) used in the next steps, just receiving and decoding a full HD video stream sent, e.g., over automotive Ethernet can be a significant burden for a typical embedded Central Processing Unit (CPU) and is ideally already offloaded to specialized hardware (e.g., to fixed-function circuitry or FPGAs). The next step is to detect, localize and classify objects and markers that are relevant for the control- and higher-level decision making algorithms. This is the predominant domain of DNNs, which may require billions of operations to analyze a single frame. Again, this needs to be offloaded to hardware that is more optimized for this kind of task than general purpose CPUs, in particular GPUs or dedicated accelerators for neural networks like Tensor Processing Units (TPUs). Once the semantic information is extracted from the video stream, the information can then be used by the decision logic and control algorithms to compute the optimal actuator settings to drive the autonomous system towards its desired state. This information is then sent to the actuators (potentially over another network) that finally apply the updated force, valve positions, voltage levels and so on.

B. Fundamentals of Field Programmable Gate Arrays (FPGAs)

FPGAs are a particular kind of re-programmable hardware, whose functionality is not fixed at chip production time, but can be configured by the system developer and even changed later when in the hands of the end customer. They mainly consist of a large array of programmable logic blocks that usually comprise SRAM-based Lookup Tables (LUTs), and are connected via an equally configurable signal interconnect. Together with memory and I/O elements, this allows the implementation of almost any digital functionality an Application-specific Integrated Circuit (ASIC) could be used for, but at a much lower cost compared to the development and production of a new chip, and with the ability to later change the functionality without developing a new chip and replacing components already in the field (e.g., in order to support a new compression algorithm). Separate FPGAs are most commonly used in prototyping stages, or in products where developing and using a specialized chip is too expensive or not flexible enough, but a software-based solution is too slow or inefficient. Additionally, chip vendors like Xilinx offer heterogeneous solutions that pair classic CPU cores with an FPGA fabric. While the FPGA part can be used as classic accelerator similar to GPUs, it can also be used to process incoming and outgoing data streams on the fly and, e.g., also handle low-level details of communication protocols without interrupting the software on the CPUs [2].

Contrary to GPUs, FPGAs and their accompanying development tools can offer excellent, cycle-accurate insights into the behavior of a given design entity, although the level of detail varies with the abstraction of the used Hardware Description Language (HDL) and/or simulation model. On the system level of purely FPGA-based systems, there, usually, is no dynamic scheduling involved that would affect the execution times in an unpredictable manner. The challenge for timing analysis resides more in predicting and balancing chip area against latency of a particular logic function's implementation (see Sec. II-B) at design time. For heterogeneous devices with a CPU subsystem, however, each shared resource like memory or bus interconnect complicates the timing analysis, as access latencies no longer depend on the logic implemented within the FPGA fabric alone.

C. Fundamentals of Graphics Processing Units (GPUs)

Computer-graphics applications require highly parallel computations for scaling in time (low latency) and space (millions of pixels). GPUs were designed to meet these requirements and later evolved to become powerful accelerators for use in graphics-based computer-vision applications. NVIDIA strategically positioned its GPUs to also be ubiquitous general-purpose parallel computers by introducing the CUDA API and extensive supporting tool chains. AMD provides an alternative GPU ecosystem complete with open-source drivers, but it is not as widely supported in popular computer-vision libraries.

As we describe in Sec. III-B, analysis for timing predictability of GPUs can be even more difficult than analysis of multicore CPUs. The number and complexity of the processing cores and the multiple levels of execution scheduling in hardware and device drivers are the primary factors. Unfortunately, NVIDIA GPUs are effectively black boxes – their inner details are locked behind restrictive NDAs and incomplete documentation. Timing analysis of GPUs also depends on how access to the GPU is arbitrated: this can be via synchronization protocols, a server or other middleware, or through the use of the existing or modified GPU drivers. Additionally, as computer-vision algorithms are typically designed for throughput rather than timing predictability, algorithmic changes are often necessary when using multicore+GPU platforms for autonomous systems.

II. Multicore+FPGA Platforms

A. "Traditional" FPGA Fabric vs. Heterogeneous FPGA-SoCs

In contrast to the earlier Flash-based Complex Programmable Logic Devices (CPLDs), the majority of current FPGAs relies on volatile SRAM cells to implement arbitrary digital functions. During offline design, synthesis and implementation phases, the system designer describes the desired functionality by means of HDLs and Intellectual Property (IP) cores. Together with timing constraints at the level of a *single clock cycle*, the vendor's tools then break the desired overall functionality into relatively small chunks of combinatorial circuits with – if necessary – single-bit downstream registers to form a multi-stage processing pipeline.

Each of those subcircuits is then mapped to one Configurable Logic Block (CLB) within the reconfigurable FPGA fabric that consists of an architecture-dependent number of (SRAM-based) LUTs and registers. Each LUT implements one small Boolean function by mapping its n-bit input signal to a single-bit output, which, optionally, can be registered. By combining from dozens up to many thousands of such CLBs, complex digital blocks are implemented without the financial/temporal overhead of custom ASICs. The cut-out within Fig. 2 (top right) also shows various other *fabric internals* found in today's FPGA architectures. This includes not only dedicated memories and multiply-accumulate (MAC) blocks (which are more area- and energy-efficient than a

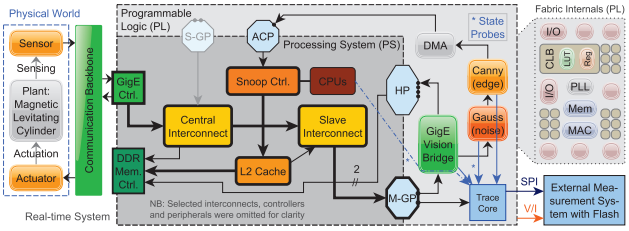


Fig. 2: Heterogeneous FPGA Architecture with a fixed-function dual-core CPU Subsystem (PS) and reconfigurable Fabric (PL)

solution entirely based on CLBs) but also crucial components to control/interface the implemented logic with the outside world. Whilst Input/Output (I/O) blocks at the edges of the device link hundreds of internal signals to physical pins, dedicated clocking resources such as Phase-locked Loops (PLLs) generate a variety of clock signals that are distributed across the entire device via carefully balanced clock trees (not shown in Fig. 2) to ensure an efficient, fully synchronous operation of fabric CLBs, memories and MACs. Similarly, regular data/control signals both between CLBs, and from/to I/O blocks and internal logic are routed via a large, programmable signal interconnect, accounting for considerable area on FPGA dies. Like LUTs and signal routing within every CLB, clock trees and signal interconnect are configured at device start (i.e., after manufacturing) by means of writing to an SRAM-based configuration memory to hold the tool-generated *bitstream* implementing the desired digital system on an FPGA.

With today's high-end devices featuring over a million CLBs, FPGAs are not only widely used to prototype the next CPU and GPU designs before tape-out, but also capable to host an entire System-on-Chip (SoC) implementing, e.g., a Real-time System (RTS). Due to its reconfigurability, however, the FPGA fabric is considerably less area- and energy-efficient than an ASIC-based implementation. As many RTSs both require the combination of parallel and sequential computation (Sec. I-A), and thus – more and more – depend on *hardware acceleration* (for, e.g., a sensor acquisition stage) *in addition to software processing*, designers increasingly instantiate a soft-core CPU subsystem in an FPGA. For this reason, selected, recent devices/series extend traditional FPGA fabric with a hard-core (i.e., fixed-function) CPU island. Similar to the integrated GPU in Sec. III-A, such heterogeneous FPGA-SoCs complement a multi-core SoC with CPUs, on-chip bus interconnects and I/O interfaces for off-chip peripherals like DDR memory and Ethernet. Such devices are available from all major vendors (e.g., Xilinx Zynq or Intel Arria) and enable the system designers to choose from a variety of SoC architectures (ranging from single- to hexa-core CPU subsystems) and FPGA fabrics [3], [4], yielding the most efficient RTS implementation.

The Xilinx Zynq shown in Fig. 2, for instance, comprises the fabric-based reconfigurable Programmable Logic (PL) drawn in light gray, and the software-driven Processing System (PS) with its various fixed-function components (dark gray). Besides two ARM Cortex-A9 CPUs with shared L2 cache (red/orange), AXI interconnects (yellow) and I/O peripherals (green), four types of bus interfaces (light blue) to the PL are crucial parts of the PS. The latter enable a tight coupling between the software (running on the PS CPUs), and any hardware I/O paths and/or processing accelerators mapped to the PL. Four General-Purpose (GP), one Accelerator Coherency Port (ACP) and four High-Performance (HP) ports are available, with only the Master (M) GP interfaces executing transfers from PS to PL. The remaining HP, ACP, and two Slave (S) GP ports enable custom IP cores in the PL to use the highly efficient PS components, e.g., for external storage to large DDR memories, or I/O via the PS's two GigE controllers.

B. Challenges in FPGA(-SoC) Implementation & Measurement

Due to the expressiveness of (at least some of) the hardware description methods and the flexibility of the underlying fabric, both the functional and temporal aspects of an individual FPGA design unit can be rather precisely defined and/or modeled on a *per-bit and clock-cycle* level. In particular using design entry on the Register-Transfer Level (RTL), the crucial "input-to-output" latency of many stream-driven (i.e., not data-dependent) blocks can be calculated formally or determined using simulations. For data-dependent processing stages, however, this problem easily becomes as intractable as a worst-case execution time (WCET) analysis in a purely software-based system [5]. With potential combinations of input data and internal states out of reach of the traditional simulation-based approaches, formal methods for the verification of hardware are widely used [6], [7], in particular for ASICs due to their high setup costs, and, more recently, also FPGAs [8]. In both worlds, however, system-level simulation-based and formal verification still can be thwarted – e.g., if used IP cores are only available in encrypted form (and without, e.g., simulation model), or if assertions or properties are insufficient.

As a preceding and ubiquitous challenge for both data in- and data dependent hardware subsystems, the fixed granularity (e.g., input/output data width) of certain FPGA resources like a fixed-function MAC unit can lead to "nonlinearities" between design-time parameters (such as numerical precision) and the efficiency of the implemented digital circuits. This not only holds true for temporal aspects such as latencies (affected by, e.g., the pipeline depth), but also resource-, area-, and thus energy-related factors like signal fan-outs or used fabric elements. Similar to the fixed-width integer units of CPUs, the MAC blocks, for instance, have upper limits w.r.t. supported data widths (e.g., 25 and 18 bits for inputs and 48 bits on the output for a Xilinx 7-Series DSP slice), which, if exceeded even by one bit, force tools and/or designers to perform the computation sequentially – requiring no longer a single, but, potentially, hundreds of clock cycles (and additional fabric resources for the control logic in the FPGA's MAC case). Small algorithmic changes thus can have a drastic impact on the implementation performance *despite* the flexibility of the fabric, which may complicate predicting the temporal properties before the design unit – or even an entire system – is actually available.

Once implemented and operational, most measurement-based approaches for CPU-only architectures (e.g., monitoring of task execution times via watermarks) can, to a certain extent, be used for an FPGA-based RTS as well. If the latency of interest is not at all associated with software, however, the measurement logic required can often be added due to the flexibility of both fabric and, in case of communication-related tasks, the I/O peripherals of current FPGA-SoCs, which range from serial to Ethernet [9]. In contrast to FPGA-specific, cycle-accurate measurement tools

(like Xilinx' Integrated Logic Analyzer) used to gain functional insights, above latency watermarks/histograms are able to cover significantly longer measurement intervals and enable statistical analysis or modeling approaches to capture the RTS's behavior.

Shared resources in both FPGA fabric and (for heterogeneous architectures like the one in Fig. 2) fixed-function SoC (i.e., PS) are an additional source of delay. More or less well-defined bus interconnects and proprietary Ethernet IPs (in PS and/or PL) are often part of time-critical control loops – spanning, for instance, from input (i.e., GigE controller) via several interconnects to the PL, and back to the PS (for software-based processing). As their delays are variable and out of reach of even the aforementioned, fabric-based solutions, additional sources for data are necessary.

C. Hybrid Power/State Tracing for FPGA(-SoC)-based Systems

To capture both partial and end-to-end delays of control loops in the heterogeneous RTS as shown in Fig. 2, hybrid power/state tracing combines functional and temporal state data from PS/PL components with a multi-rail power measurement of the FPGA-SoC and its external I/O devices (like Ethernet PHYs) [1]. Apart from guiding energy optimization, the latter also captures an architecture-dependent number of crucial temporal events (such as the arrival time of a sensor signal at the RTS boundary using, e.g., an Ethernet-based communication backbone like in Fig. 2). The cross-PS/PL latencies of used shared resources (thick lines) thus can be analyzed by combining the temporal events from PS and PL states with such timestamps for sensor and actuator I/O.

Open issues in tracing of complex, mixed-hardware/software RTSs, however, remain plenty. Current solutions either occupy a significant amount of fabric memory or require external storage, e.g., via a Flash-based measurement system (as shown in Fig. 2, bottom right). Whilst the former might interfere with the actual RTS application, the latter often cannot be deployed in the field. As further complication, the fabric signals, software sources or, in case of hybrid tracing, measurement rails might not be known to the full extent before operation, causing costly turnarounds.

III. Multicore+GPU Platforms

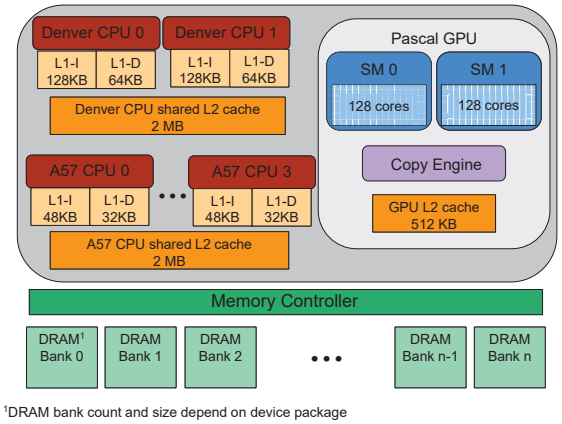
We now provide an overview of GPU hardware and the software that executes upon it. We then discuss challenges both in performing timing analysis for GPU-using workloads and when using GPU in vision applications, as well as open issues.

A. GPU Hardware and Software

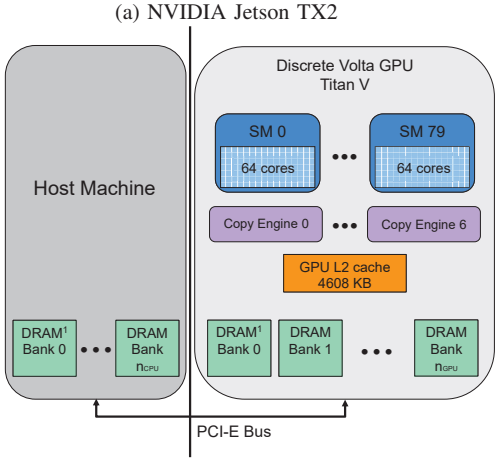
We focus our attention on NVIDIA GPUs, as they are ubiquitous in computer-vision and autonomous-driving applications.[1]

NVIDIA GPU hardware. NVIDIA GPUs consist of multiple *copy engines* (CEs), which copy data to and from the GPU, and an *execution engine* (EE) comprised of multiple *streaming multiprocessors* (SMs). An SM services up to 2048 *GPU threads* at once. Integrated GPUs (on the same chip as the CPU), such as the NVIDIA Jetson TX2, usually have an order of magnitude fewer SMs than discrete GPUs (on separate chips, typically connected to the host machine via a PCI-E bus), such as the NVIDIA Titan V.

[1] AMD GPUs have similar hardware constructs, so many issues and solutions we discuss apply to them as well.



(a) NVIDIA Jetson TX2



(b) NVIDIA Titan V

Fig. 3: Comparison of two GPUs: (a) NVIDIA TX2 (integrated) and (b) Titan V (discrete)

Details of two NVIDIA GPUs are depicted in Fig. 3. NVIDIA's TX2 SoC (inset (a)) includes a quad-core ARMv8 A57 processor, a dual-core ARMv8 Denver processor, and an integrated GPU comprised of one CE and two SMs. In contrast, the discrete Titan V (inset (b)) boasts seven CEs and 80 SMs.

The NVIDIA CUDA API. CUDA is a C/C++ extension developed by NVIDIA to allow programmers to write parallel processing GPU programs and to submit operation requests to the GPU [10]. The CUDA API includes commands to copy data between the CPU and the GPU memories, as well as between multiple GPUs, and to submit programs (called *kernels*) for execution on the GPU. The general structure of a CUDA program includes (i) allocating memory on the GPU, (ii) copying data from CPU to GPU memory, (iii) executing one or more kernels on the GPU, (iv) copying the results back to CPU memory, and (v) freeing any allocated memory on the GPU. The typical kernel program is written in a single instruction, multiple data (SIMD) form such that each GPU thread performs the same set of instructions but using data unique to the thread.

Kernel execution is scheduled by the GPU in groups of 32 threads, called *warps*. Each thread in a warp executes on a

GPU core in a SIMD fashion, using built-in CUDA variables to determine the data for that thread. Warps are combined into *blocks*, which are further grouped into *grids*; to request a kernel execution, the programmer specifies the layout of threads into grids and blocks to facilitate binding threads to data. Each SM is comprised of a number of hardware processing cores, scheduled by a small number of internal *warp schedulers*. A warp scheduler hides execution latency by dispatching a warp from its ready queue onto 32 of its cores when the previously executing warp stalls for access to a hardware resource (execution unit, memory, etc.).

NVIDIA tracing tools. Tools such as NVIDIA's Visual Profiler can be used to profile CUDA applications to visualize kernel executions, copies, and other CUDA operations, and to guide optimization steps to maximize performance. However, these tools do not give information necessary for timing analysis of such workloads (e.g., which SMs execute a given kernel, context switching between different processes), and their optimization recommendations do not consider interference effects.

B. Timing Analysis for GPU-Using Workloads

One or more GPUs in an autonomous system are typically shared among multiple vision applications. A fundamental consequence is that interference from competing demands has a significant impact on timing behavior. Interference can be both temporal (which tasks execute when) and spatial (competition for shared caches, DRAM, and buses). Accounting for GPU execution time in real-time analysis depends on how competing demands are arbitrated – by synchronization or scheduling. Synchronization can be realized through the use of locking protocols to limit interference by managing access to the GPU. Scheduling can be realized using default GPU scheduling rules to determine potential interference; alternatively, middleware can be used to enable modified scheduling policies.

Multicore+GPU synchronization. Given the complex or even black-box nature of GPUs, they are often treated as monolithic devices that must be accessed exclusively. Prior work on real-time multicore+GPU systems has treated the GPU as a mutual-exclusion resource (e.g., [11]–[13]). In multi-GPU systems, identical GPUs can be treated as replicated resources [14].

Multicore+GPU scheduling. On NVIDIA GPUs, kernels submitted to the GPU from different address spaces are multi-programmed, i.e., all resources are given to one kernel at a time, switching between the processes' kernels in a time-sliced manner. Thus, true concurrency between processes is disabled by default; to enable concurrency between multiple address spaces, NVIDIA's Multi-Process Server (MPS) must be used.[2]

For GPU kernels submitted from a single address space (or multiple address spaces if MPS is enabled), a hierarchical queue structure can be used to model the NVIDIA driver scheduling mechanisms; Amert *et al.* provided rules to describe the scheduling of GPU operations [15], specifying block-level scheduling behavior. Their benchmark suite is available online [16], and can be used to trace block-level scheduling behavior on NVIDIA GPUs. Sañudo *et al.* provided additional details on how individual blocks are mapped to SMs [17].

Middleware or driver-level changes can be used to intercept and reorder GPU operations. For example, Kato *et al.* presented TimeGraph [18], a non-preemptive fixed-priority scheduler that relies on modifications to the open-source Nouveau driver for NVIDIA GPUs (their work considered only graphical workloads, as the Nouveau driver does not support CUDA). Extending to general-purpose computations, Capodieci *et al.* demonstrated a real-time scheduler for graphical and CUDA-using tasks [19]. They implemented a software scheduler module within the NVIDIA Drive-PX2 driver, enabling preemptive earliest-deadline first (EDF) scheduling on the Drive-PX2 embedded platform.

In addition to allowing multiple processes to use a GPU concurrently, MPS also enables partitioning of the EE. However, it does not allow for partitioning of the memory hierarchy, which can lead to conservative estimation of spatial interference. To enable partitioning of both cache and DRAM for discrete NVIDIA GPUs, Jain *et al.* developed micro-benchmarking experiments to reverse engineer the NVIDIA GPU memory hierarchy [20]. Their page coloring technique enabled "fractional GPUs" with improved isolation between processes.

Response-time and WCET analysis for GPUs. The scheduling rules of NVIDIA GPUs can be directly used to determine response-time bounds for GPU-using workloads. Yang *et al.* provided response-time analysis [21] that depends on the maximum thread requirement of any block in the system, the number of blocks per kernel, and the number of SMs comprising the GPU's EE, among other parameters.

Considering WCET analysis of individual kernels, Heo *et al.* provided a WCET analysis for each layer of a DNN [22]. Their model considers both processor contention (based on eligibility of individual warps) and memory contention. Given the closed-source nature of NVIDIA's cuDNN library, Heo *et al.* used NVIDIA's profiling tools to measure counters (e.g., cycle and memory instruction counts) needed to estimate parameters for their model. They used their WCET analysis to modify an existing DNN pipeline, as described below.

C. Using GPUs in Vision Applications

Care must be taken when using NVIDIA GPUs in safety-critical applications, such as vision processing in autonomous systems. NVIDIA documentation is vague (and sometimes self contradicting), particularly around sources of implicit blocking delays on the GPU and even the host CPU. Such pitfalls were explored by Yang *et al.* [23], and include blocking on the CPU if any GPU-memory frees occur concurrently with the submission of GPU operations (contrary to the expected behavior based on the documentation).

Many popular computer-vision frameworks have support for NVIDIA GPUs, including PyTorch and TensorFlow for deep learning and OpenCV for general computer-vision processing. However, such frameworks are designed for throughput rather than predictability. Furthermore, in automotive systems, hardware resources are constrained by size, weight, and power. As a result, algorithmic changes can be necessary to enable running

[2]MPS is only available for discrete GPUs.

vision applications as real-time tasks and maximally utilizing multicore+GPU platforms.

Yang *et al.* [24] considered various parallelism and pipelining approaches for a convolutional neural network (CNN) object-detection application, including merging images from multiple camera sources into a single composite image. To enable real-time deadlines to be met for DNNs, Heo *et al.* modified a DNN-based object-detection system to be able to choose between different pre-configured network paths [22]. The different paths are chosen at runtime based on their WCET analysis for each layer of the network and timing requirements.

D. Open Issues

A key open issue with multicore+GPU platforms is the exploration of AMD GPUs. Due to the open-source nature of the AMD software stack, real-time scheduling at the driver level could enable better predictability than is available for NVIDIA GPUs. However, AMD GPUs are not currently as widely supported as NVIDIA GPUs by popular vision frameworks.

Thus far, most scheduling-based approaches have focused on middleware, negating much of the potential for concurrent execution; Capodieci *et al.* [19] explored scheduling changes within the open-source drivers available only for NVIDIA's embedded platforms such as the Drive-PX2, and details and source code were not made available due to non-disclosure agreements. For both AMD and NVIDIA GPUs, more work is needed to evaluate GPU driver implementations with different real-time scheduling policies for GPU kernels.

IV. Concluding Remarks

Both FPGAs and GPUs enable the acceleration of general-purpose computations used in vision applications by means of the parallelism offered by their architectures. However, fully capturing the timing behavior of FPGA- and GPU-enabled systems remains a complex endeavor. While latency analysis of a hand-written FPGA design entity is often relatively straightforward, the challenges lie in estimating resource requirements and temporal behavior before design completion, the integration of closed-source IP cores, and hard-to-predict shared resources.

In terms of GPUs, NVIDIA currently leads the market, but their devices and software stack target throughput rather than predictability, and their tools do not elucidate key details needed for full timing analysis. GPUs from AMD present a promising alternative, but are not as widely adopted by vision frameworks.

For systems comprised of both FPGAs and GPUs in addition to multicore CPUs, analysis such as that of Yang et al. [21] enables response times to be computed for acyclic processing graphs in which different graph nodes (computations) execute on different processor types. Such analysis could enable system designers to choose between implementing algorithm components on the CPU, FPGA, GPU, or other accelerators. However, the problem of accurately analyzing timing behavior – be it mathematically or via tracing – for real-life heterogeneous architectures and general application models, still remains a largely open problem that needs to be addressed for better design and verification of autonomous systems.

Acknowledgements

This work was supported by the NSF award #2038960.

References

[1] M. Balszun, M. Geier, and S. Chakraborty, "Predictable vision for autonomous systems," in *23rd IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2020.

[2] M. Geier, F. Pitzl, and S. Chakraborty, "GigE vision data acquisition for visual servoing using SG/DMA proxying," in *14th ACM/IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*, 2016.

[3] Xilinx Inc., "All Programmable SoCs and MPSoCs," https://www.xilinx.com/products/silicon-devices/soc.html.

[4] Intel Corporation, "Intel SoC FPGAs Programmable Devices," https://www.intel.com/content/www/us/en/products/programmable/soc.html.

[5] R. Wilhelm *et al.*, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[6] D. D. Borrione, L. V. Pierre, and A. M. Salem, "Formal verification of vhdl descriptions in the prevail environment," *IEEE Design & Test of Computers (D&T)*, vol. 9, no. 2, pp. 42–56, Jun. 1992.

[7] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, Apr. 1999.

[8] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open source framework from verilog to bitstream for commercial FPGAs," in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[9] M. Geier, T. Burghart, M. Hackl, and S. Chakraborty, "In situ latency monitoring for heterogeneous real-time systems," in *32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems (VLSID)*, 2019.

[10] NVIDIA, "CUDA toolkit documentation v11.1.1," Online at http://docs.nvidia.com/cuda/.

[11] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *32nd IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[12] U. Verner, A. Mendelson, and A. Schuster, "Scheduling periodic real-time communication in multi-GPU systems," in *23rd International Conference on Computer Communication and Networks (ICCCN)*, 2014.

[13] G. Elliott, B. Ward, and J. Anderson, "GPUSync: A framework for real-time GPU management," in *34th IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[14] C. Nemitz *et al.*, "Multiprocessor real-time locking protocols for replicated resources," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[15] T. Amert *et al.*, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *38th IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[16] N. Otterness, "Cuda scheduling viewer," https://github.com/yalue/cuda_scheduling_examiner_mirror, 2020 (accessed 29 November 2020).

[17] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective," in *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[18] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *USENIX Annual Technical Conference*, 2011.

[19] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[20] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[21] M. Yang *et al.*, "Making OpenVX really 'real time'," in *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[22] S. Heo, S. Cho, Y. Kim, and H. Kim, "Real-time object detection system with multi-path neural networks," in *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[23] M. Yang *et al.*, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[24] ——, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.