Cohort: Software-Oriented Acceleration for Heterogeneous SoCs

Tianrui Wei
UC Berkeley, USA
tianruiwei@berkeley.edu

Nazerke Turtayeva
UC Santa Barbara, USA
nturtayeva@ucsb.edu

Marcelo Orenes-Vera
Princeton University, USA
movera@princeton.edu

Omkar Lonkar
UC Santa Barbara, USA
omkarlonkar@ucsb.edu

Jonathan Balkind
UC Santa Barbara, USA
jbalkind@ucsb.edu

ABSTRACT

Philosophically, our approaches to acceleration focus on the extreme. We must optimise accelerators to the maximum, leaving software to fix any hardware-software mismatches. Today's software abstractions for programming accelerators leak hardware details, requiring changes to data formats and manual memory and coherence management, among other issues. This harms generality and requires deep hardware knowledge to efficiently program accelerators, a state which we consider hardware-oriented.

This paper proposes Software-Oriented Acceleration (SOA), where software uses existing abstractions, like software shared-memory queues, to interact with accelerators. We introduce the Cohort engine which exploits these queues' standard semantics to efficiently connect producers and consumers in software with accelerators with minimal application changes. Accelerators are even usable in chains which can be runtime reconfigured by software. Cohort significantly reduces the burden to add new accelerators while maintaining system-level guarantees. We implement a Cohort FPGA prototype which supports SOA applications running on multicore Linux. Our evaluation shows speedups for Cohort over traditional approaches ranging from 1.83× to 8.38× over MMIO, and from 1.69× to 11.24× for DMA baselines. Our software-oriented batching optimisations within Cohort also improve performance from 2.32× to 8.10×, demonstrating the power of SOA.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; *Reconfigurable computing*; Heterogeneous (hybrid) systems.

KEYWORDS

accelerators, heterogeneous systems, programming models, shared memory

ACM Reference Format:
Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. 2023. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,*

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582059

Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3582016.3582059

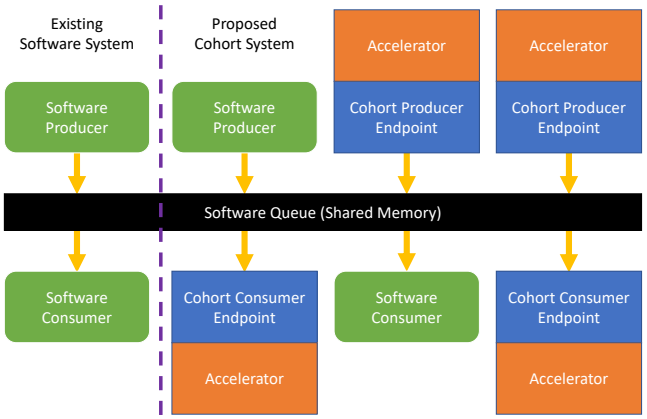

Figure 1: Proposed Cohort System Design

1 INTRODUCTION

To meet users' efficiency and performance demands, systems-on-chip (SoCs) feature many efficient, specialised accelerators alongside general-purpose cores [16, 21, 37, 39, 62, 81]. The architecture of these SoCs suggests the accelerators are subservient to the general-purpose cores, which manage their memory allocation, scheduling, and communication. One could thus view this as a software-oriented SoC rather than an accelerator- or hardware-oriented one. However, we argue that the system is driven by the demands of accelerators, which have special needs and frustratingly bespoke programming paradigms, lacking commonality [4, 23, 31, 46, 52, 61, 73] and requiring more code with each additional accelerator. Further, the hardware accelerators cannot be easily composed together since layers of privileged software must mediate communication. If heterogeneity is to further grow, we must find more efficient means for management of and communication with hardware accelerators.

Due to accelerators' piecemeal adoption, their software and hardware interfaces are routinely afterthoughts. There is little consideration for hardware or software approaches which are common across accelerators and smooth adoption. An ideal approach would feature:

- A common, flexible hardware interface to efficiently communicate between software and hardware elements.
- A software paradigm using existing abstractions to minimise software modification to add new accelerators.

We introduce an approach, termed **Software-Oriented Acceleration**, to emphasise that accelerator communication, isolation, and memory management respect existing software paradigms. Our chosen software abstraction is shared-memory queues, with which we reduce hardware and software overhead and ease accelerator adoption. Our system design (Figure 1) has cores and accelerators communicating arbitrarily as peers, forming a **Cohort**. This enables accelerator composition without software imposition, a pattern which has been underutilised to date.

The Cohort SOA model focuses on accelerators which follow a stream/buffer in and stream/buffer out (SBIO) communication pattern. Many accelerators follow this model, using memory-mapped I/O (MMIO) or direct memory access (DMA) to consume and produce buffers or streams of data [26, 43, 47, 52, 75]. For accelerators with broader communication patterns than SBIO (e.g., GPGPUs or sparse graph and neural network accelerators [7, 39, 59, 62, 72]), Cohort can enable more efficient communication for the use cases which are SBIO.

Cohort provides a common, standard queue interface both to software and to hardware, enabling reuse both of existing multithreaded software and of existing hardware accelerators with little to no modification. To bridge between software-friendly shared-memory queues and hardware-friendly interfaces, we created the Cohort engine. With it, existing software produces data into queues for consumption by hardware or software without special memory allocation routines, cache coherence management, etc. Likewise, accelerators are connected unmodified with existing, high-performance, latency-insensitive [12, 13, 71] hardware interfaces including AXI-Stream [3].

We implement Cohort on an SoC using the native coherence protocol to enable efficient operation. Cohort's primitives could also be implemented atop emerging cache coherent interconnects beyond the SoC scale, e.g. CXL, CCIX, and CAPI [15, 70, 74]. We have implemented a software-oriented SoC prototype on FPGA with RISC-V cores booting Linux and accelerators connected via Cohort engines (shown in Figure 2. Our evaluation tests lightly modified software running on Linux, communicating between accelerators and software threads with no consideration for the nature of the producer or consumer on the other side of the queues.

Our paper makes the following contributions:

- The software-oriented acceleration approach, using existing communication, isolation, and memory management from software to manage hardware accelerators.
- The identification of queue coherence semantics to enable efficient hardware support for software shared-memory queues.
- Implementation of a Cohort-enabled SoC with Linux support and multiple integrated accelerators.
- SHA benchmark speedup from Cohort's Engine and API range from 5.44× to 8.38× over MMIO and from 7.27× to 11.24× over coherent DMA baselines.
- Open-source hardware and software of Cohort available at https://github.com/cohort-project

2 MOTIVATION

Having built and programmed many heterogeneous SoCs, we have found issues across the stack. Here we describe how they motivated us to create a more software-oriented SoC design approach. These tie to the two key features mentioned in Section 1: a common, flexible hardware interface, and a software-friendly programming paradigm.

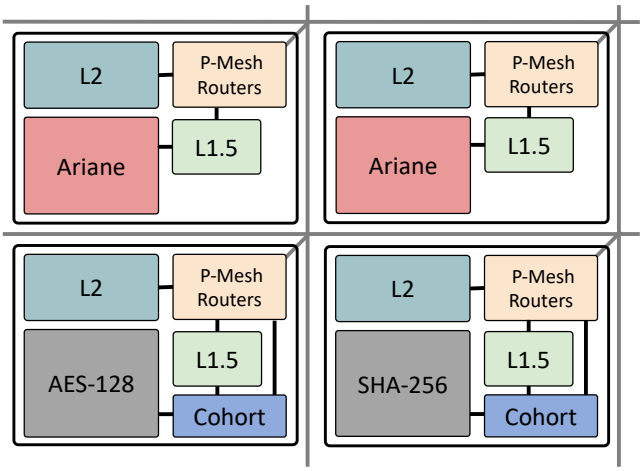

Figure 2: Cohort FPGA prototype with AES and SHA accelerators

2.1 Software Programmability Issues

Accelerator vendors provide varying software interfaces [29, 30, 36, 45, 49]. Some provide compiler passes or domain specific languages, some call into privileged kernel modules, and others adopt vendor-specific libraries, making for a fragmented software environment. To exploit multiple vendors' accelerators becomes difficult, requiring a variety of bespoke expertise. This results in fragile software, as vendors' requirements conflict, if composition is at all possible.

We have also been challenged by memory allocation and management for accelerators. It is routine for new accelerators to require special memory allocations or the addition of manual cache flushing [52, 59, 74]. These requirements come for a variety of reasons, including accelerators' use of IOMMUs, differing coherence models, and special addressing and alignment constraints. As a result, the task of complexity management falls to software.

In existing SoCs which use MMIO to configure and interact with accelerators and I/O devices, MMIO loads and stores have special semantics which hurt performance. MMIO operations have side effects, meaning that they cannot be performed speculatively and that interrupts generally should not be taken while they are outstanding in the memory system. Many cores, particularly slim in-order ones, will simply stall on these until a load or store returns from the accelerator or device. This behaviour is often relied on for synchronisation, with the return guaranteeing that an operation has completed. In terms of programmability, performing these MMIO operations from user mode is fraught with problems to ensure that operations are not optimised out or reordered by the compiler. We argue that MMIO is undesirable for performance and programmability and believe this is backed up by other recent approaches in, for example, high-performance networking, where shared-memory queues have replaced such operations [57, 80].

2.2 Hardware Interfacing Issues

It is now essential to optimise the data-movement interface, both between cores and accelerators and among accelerators. Coarse-grained accelerators in SoCs today are connected in various ways [14, 39, 59]. Some act as MMIO devices, with private registers or memories that are filled prior to invocation, and read out upon completion. Others use DMA engines, programmed prior to invocation, to fetch data from and store data to memory.

Accelerators also vary in their coherence models, often operating partially- or non-coherently, which complicates software. While fully-coherent accelerators are convenient, they have costs from participating in cache-coherence. Balancing these models is sufficiently complex that research has even been done on machine learning techniques to choose between them [83].

In desktop- and server-class settings with PCIe, protocols like CCIX, CXL, and CAPI [15, 70, 74] enable accelerator cache coherence. However, building on PCIe brings overhead and latency, meaning applications with strict latency requirements usually perform poorly. Neugebauer et al. [58] characterised end host networking and found that for a 128B payload, 90% of the observed 1000ns round-trip latency comes from PCIe itself. In addition, coherence systems like CCIX can offload coherence management to accelerators, meaning coherent access to accelerator-managed memory regions is PCIe topology dependent, which further complicates the system [15]. Many mobile-class SoCs limit use of PCIe due to its power and area, relying instead on native (shared) memory for many of their accelerators and peripherals [54, 55, 60, 65, 66].

3 COHORT APPROACH

3.1 Software-Oriented Acceleration Philosophy

At a high level, SOA reuses existing software mechanisms for communication and application design. Software is routinely optimised in unforeseen ways to drive performance and efficiency. Using programming languages' and libraries' existing abstractions, we enable new avenues for optimisations across many hardware accelerators. We need not adopt special I/O operations, memory and cache management, or complex accelerator-specific drivers, unlike existing approaches [52, 59].

SOA is user mode-oriented to give applications and language runtimes greater flexibility, isolation, and security. Mechanisms which provide kernel interposition are preferred as they enable enforcing system-level policy, profiling, etc. However, encouraging user-level operation should not fundamentally require context switches and kernel interaction for operation, which hurt performance.

3.2 Queue Coherence

In Cohort, we adopt shared-memory queues as the lingua franca of our heterogeneous SoC, as shown in Figure 1. In high-performance parallel software, shared-memory queues enable the decoupling of threads to gain parallelism [53, 76]. We work to provide a common producer-consumer communication abstraction in order to improve programmability and composition, and to enable new performance tuning opportunities. Cohort users replace existing queue-decoupled software threads with accelerators while maintaining their existing code and its chains of queues connecting producers and consumers.

```
fifo_t *fifo = fifo_init(...);
pid_t pid = fork();
if (pid == 0) { // producer
  push(element, fifo);
} else { // consumer
  element = pop();
  printf("element: %d\n", element);
}
```

Figure 3: Generic producer-consumer code snippet

Table 1: Cohort API Listing

Existing generic SPSC queue API calls
`int fifo_init(int element_size, int queue_length);`
`void push(int element, fifo_t *q);`
`int pop(fifo_t *q);`
`int fifo_deinit(fifo_t *q);`
Additional Cohort-specific API calls
`int cohort_register(int acc_id, fifo_t *acc_in, fifo_t *acc_out);`
`int cohort_unregister(int acc_id, fifo_t *acc_in, fifo_t *acc_out);`

Many parallelisation techniques rely on the queue-based producer-consumer design pattern for communication and decoupling [33, 37, 38, 51, 61, 69, 78]. Producers and consumers maintain a common queue for communication of data and synchronisation. When available, the producer hands data to the consumer by pushing it into the queue. When a consumer is able to process more data, it pops it from the queue. The producer-consumer design pattern is used in both shared-memory multithreaded applications and for inter-process communication by sharing a small memory region between the processes for the queue. Figure 3 shows a simple example code snippet following the generic queue API shown in Table 1.

Much effort has gone into developing high-performance lock-free queue libraries. These libraries provide performance by minimising coherence effects and taking advantage of the memory-level parallelism of modern cores. With their widespread use, these queues' semantics are well understood and ready to be exploited in Cohort. We focus here on lock-free single producer-single consumer (SPSC) queues, which are supported in widely-adopted software libraries [11] and further boost performance.

We introduce the term **Queue Coherence** to refer to the semantic behaviour of SPSC queues. In existing libraries, there is an agreed meaning to enqueue/push and dequeue/pop with respect to both cache coherence and memory consistency. When the producer completes an enqueue and the consumer observes a change in the queue's write pointer, there is a guarantee that the consumer will also observe the updated data items in the queue. Cohort exploits queue coherence by implementing queue operations in an engine connected to the cache coherence system. This enables software to operate as designed while feeding accelerators with the highest performance and efficiency that general-purpose cores offer.

```
fifo_t compute_fifo = fifo_init(...);
fifo_t result_fifo = fifo_init(...);
cohort_register(acc, compute_fifo, result_fifo);
push(acc_in_elem, compute_fifo);
int acc_out_elem = pop(result_fifo);
printf("element is %d\n", acc_out_elem);
```

Figure 4: Producer-consumer code snippet for Cohort with a single accelerator

```
fifo_t encrypt_fifo = fifo_init(...);
fifo_t hash_fifo = fifo_init(...);
fifo_t result_fifo = fifo_init(...);
cohort_register(encrypt_acc, encrypt_fifo, hash_fifo);
cohort_register(hash_acc, hash_fifo, result_fifo);
push(data, encrypt_fifo);
int chain_result = pop(result_fifo);
```

Figure 5: Producer-consumer code snippet for Cohort accelerator chaining

3.3 Software-Hardware Composition

Cohort enables replacement of a software thread with a Cohort engine to enable transparent acceleration, as Figure 4 shows. This "accelerator thread" communicates with our software threads in the same way that another software thread would. Instead of leaning on complex accelerator management code and driver support, offloading computation to a Cohort-enabled accelerator is as transparent as pushing data into the software queue connected to the accelerator's input. To receive results back, the software thread simply pops data from another software queue connected to the accelerator's output.

In addition, the interoperability brings the benefit of transparent accelerator chaining, enabling chaining of a series of computations through several accelerators, whilst being transparent to software. If we have a hashing and an encryption accelerator, then to perform an encryption followed by a hash, we only need the code snippet in Figure 5.

Our approach, introducing Cohort, provides this exact ease of programming. With a minor addition in the form of the queue registration routine (which does not modify the underlying memory allocation and could be incorporated into the queue library), software queues are connected to Cohort units on-chip, ready for production and consumption by accelerators. This enables runtime reconfiguration of the hardware, with accelerator chains created dynamically by software. The programmer can maintain their high-level producer-consumer abstraction while incrementally moving functionality to hardware accelerators.

4 COHORT IMPLEMENTATION

This section explains the implementation of Cohort and its integration into the SoC as well as the library and operating system support we developed to enable booting SMP Linux on the Cohort SoC on FPGA.

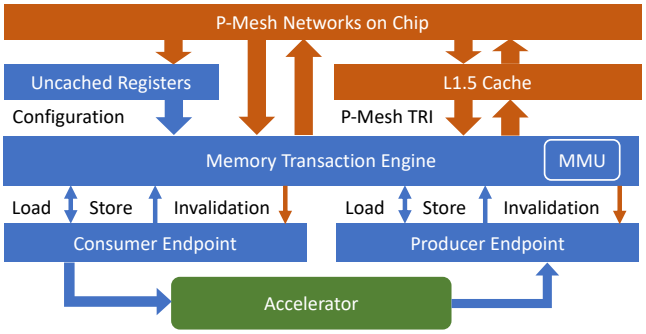

Figure 6: Cohort Engine Architecture

4.1 Programming Model

Cohort is designed for use from user mode with minimal operating system involvement. There is a single Cohort driver to support all Cohort-enabled accelerators, regardless of the variety of accelerators available. This is in contrast to many existing accelerator drivers which grow the size and complexity of our trusted software over time.

To support maximally transparent use of SPSC queues, the Cohort engine features an MMU compatible with the cores' ISA's MMU. In a modern SoC, such an MMU is low cost and its benefits are significant, as we show in Section 6. Queues are virtually contiguous and are accessed sequentially, which significantly improves TLB hit rates. Beyond this, Cohort benefits from the very same optimisations as software programmers adopt for multicore performance. E.g. If the programmer adopts huge pages in their queue library, the Cohort MMU will take advantage of them and see performance improvements just as cores would.

4.1.1 Queue Descriptors. A common goal for high-performance software is to choose a queue structure that minimises coherence traffic. Software queues are thus organised in a variety of ways that Cohort must support. To register a queue with Cohort, its structure must be described to properly configure the Cohort engine. For Cohort to achieve broad compatibility, desirable features of SPSC queues include configurable element size, configurable queue size, and use of read and write indices versus pointers.

To support a range of queue formats, we develop a queue descriptor struct which the queue library developer uses to describe their queue. The descriptor also contains (virtually addressed) pointers to the queue elements in question, such as the read or write index. At present, we support the following attributes for the queue.

- `write_pointer/index`
- `read_pointer/index`
- `fifo_base_address`
- `fifo_element_size`
- `fifo_length`

4.1.2 User-Mode API. Figure 7 shows a code snippet demonstrating the Cohort user-mode programming interface. The programmer starts by allocating two cohort-enabled queues with queue descriptors in their `fifo_t` structures from their queue library. The programmer then registers the two queues connected to a

```
fifo_t *sw_to_cohort = fifo_init(...);
fifo_t *cohort_to_sw = fifo_init(...);
cohort_register(acc_id, sw_to_cohort, cohort_to_sw);
push(0xcafedeed, sw_to_cohort);
uint64_t result = pop(cohort_to_sw);
cohort_unregister(acc_id, sw_to_cohort, cohort_to_sw);
```

Figure 7: Cohort API usage example

Cohort-enhanced accelerator by calling the `cohort_register` system call with the ID of the accelerator to be used plus its input and output queues. From that point, the standard push and pop functions from their queue library are used to feed data to and from their accelerator. At the end of the application, they then call `cohort_unregister` to end their use of the queues with the accelerator by again providing its ID and input and output queues.

From the programmer's perspective, the programming experience is extremely close to programming for an everyday multi-threaded environment. Unlike existing approaches which either expose complexity to the programmer or try to hide it in allocation routines and syscalls, Cohort-enabled library code continues nearly unmodified. The queue library developer simply describes their queue via a Cohort queue descriptor to make the queue usable by the programmer, with no need for specialised memory allocation or deallocation routines. The only addition are the `cohort_register` and `cohort_unregister` syscalls provided by the Cohort kernel driver. A generic representation of the existing queue APIs on which Cohort can be built is shown in Table 1 alongside the two new API calls required for Cohort.

Queue Library Support. Thanks to Cohort's flexible queue descriptors, it is reusable in a variety of queue settings. As noted in Section 3.2, SPSC queues are commonly adopted in software libraries. We designed Cohort's hardware based on established queue implementations in software and thus have seen remarkably straightforward development and integration. While our evaluation uses a hand-rolled C implementation, we have also demonstrated Cohort's cohesive integration with a high-level software library by implementing support in the C++ Boost Lockfree library to communicate with Cohort-enabled accelerators [11].

4.2 Cohort Engine

Thanks to queue coherence, the Cohort engine bridges from software-level thread-safe queue operations to simple hardware queues with latency insensitive interfaces, including standard valid/ready FIFOs and AXI Stream [3, 12, 13, 71]. The designer chooses where to place the Cohort engine in their cache coherence system, how to optimise it for their specific needs, and uses it to hide coherence system details from the accelerators. For SBIO accelerators, Cohort does not require the accelerator to have any knowledge of cache coherence. Instead, the accelerator connects to two latency-insensitive endpoints provided by the Cohort engine, for consumption of input and production of results. When data is available, it is provided by the Cohort engine and otherwise the accelerator avoids polling.

As shown in Figure 6, Cohort is separated into several functional units, each handling a specific task. Our prototype Cohort engine is integrated into the P-Mesh cache coherence system of OpenPiton [8, 10], which is an open-source tile-based SoC framework. The components of P-Mesh that Cohort directly interacts with are shown in orange at the top of the figure. Cohort itself is rendered in blue in the centre of the figure, while the connected accelerator is shown at the bottom in green. Cohort is integrated on its own tile within the SoC: a zoomed out view of a complete SoC is shown in Figure 2.

The Cohort Engine has a few main components: uncached configuration registers, the Memory Transaction Engine (MTE) which contains the MMU, a Consumer endpoint, and a Producer endpoint. CPU cores may configure Cohort through its uncached configuration registers, which are the only MMIO component of Cohort. The producer and consumer endpoints are responsible for generating queue operations. When the endpoints need to communicate with memory, the MTE translates from higher level memory operations and invalidations to physically addressed coherence operations. The endpoints effectively translate the software queue structure into a data stream for the connected accelerator to handle, with a hardware-friendly interface.

4.2.1 Consumer Endpoint. The consumer endpoint is the point of ingress for data into the accelerator. Following registration of a queue with the Cohort engine, the consumer endpoint performs the coherence operations needed to track the read and write pointers of the SPSC queue. When the write pointer is updated by the queue's producer (from software or hardware), the consumer endpoint receives an invalidation, which is its signal to fetch new data for the accelerator. To reduce coherence effects, our implementation is optimised to wait a configurable period. Once the endpoint has the data, it sets its valid signal to the accelerator high and waits for the accelerator to be ready. Once the accelerator receives the data, the consumer endpoint updates the queue's read pointer accordingly, which is communicated to the producer via the coherence system.

4.2.2 Producer Endpoint. The producer endpoint performs the reciprocal operations of the consumer endpoint. When the accelerator produces data (its valid signal goes high) and the queue is not full, the accelerator will take the data, store it into the queue, and then update the write pointer. Both operations are performed with appropriate consistency guarantees and the updates are communicated to the consumer via the coherence system.

4.2.3 Semantics and Coherence. As discussed in section 3.2, SPSC queues signal elements being pushed or popped by changing the write or read pointer, respectively. When a Cohort or software endpoint alters a pointer in the queue structure via the MTE, it means that elements are being pushed to or popped from the queue, following software semantics. In the SoC's microarchitecture, we assume that the coherence system will cause the line to be invalidated in other caches, which acts as a signal to Cohort endpoints to look for an update.

We exploit this behaviour in two ways for Cohort to transparently observe the changes made by software. The Reader Coherency Manager (RCM) monitors an address passed from the uncached registers for incoming invalidations. Whenever an invalidation matches the address the RCM monitors, it enters the backoff state. The length of the backoff is controlled by the backoff unit. After the

backoff ends, the RCM issues a coherent read to bring the most up to date copy of the data. Consistency is guaranteed by the writer ordering the data write before the pointer write using appropriate fences or store semantics, which is the standard behaviour in lock-free SPSC queue libraries. Just as the software library does, the Write Coherency Manager (WCM) carefully orders its write operations to ensure that a reader will see the update to the write pointer before seeing the data update.

4.2.4 Cohort MMU. For our implementation of Cohort in OpenPiton+Ariane, we include a RISC-V Sv39-compliant MMU, based on the MMU in the Ariane core itself. This MMU enables Cohort's various components to make use of virtual addresses, significantly simplifying the programming of Cohort-enhanced accelerators. As a result of the ISA-native MMU, queues are allocatable with `malloc` and the Cohort endpoints seamlessly translate VAs to PAs.

The Cohort MMU features a TLB and page table walker to maximise its independence from the cores in the SoC. The operating system manages the Cohort MMU coherently with the others in the SoC, as we describe in Section 4.4. Each MMU has the page base pointer pointing to the root address of the page table of the corresponding process. Upon a TLB miss, the page table walker traverses the page table and refills the entry transparently. When there is a page fault due to a miss in the TLB and an unsuccessful page table walk, the Cohort MMU raises an interrupt to a core to resolve the page fault. Once resolved, the core writes to one of two MMU registers: the first simply resolves the fault and requires the page table walker to complete its own page table walk, while the second enables the core to write the page table entry directly into the TLB. Besides these registers, the Cohort MMU's TLB is flushed via a write to another register, in order to maintain TLB coherence.

4.3 Accelerator Interface

Figure 6 shows the accelerator connecting to the consumer and producer endpoints. Our prototype supports both simple valid-ready handshakes [12, 13, 71] and AXI-Stream [3] as latency insensitive interfaces to and from the accelerator. As the consumer endpoint retrieves data from the queue feeding into the accelerator, it sets its valid signal to the accelerator and waits for the accelerator to be ready to consume the data. Similarly, the accelerator sets its valid signal high to produce data to the producer endpoint, which makes the corresponding updates to the queue feeding out of the accelerator. Many SBIO accelerators are usable unmodified with the Cohort accelerator interface, while others could have their LSUs straightforwardly replaced with the Cohort engine.

Accelerators need not produce data with the same interface width as it consumes data. Cohort produces and consumes data to/from the accelerator using blocks of parameterised size, with appropriate ratchet logic to resize data to the accelerator's required sizes. As an optimisation, the producer and consumer endpoints reduce coherence traffic commensurate with the accelerator's data block input or output size, updating the read or write pointers by the data block size.

Upon queue registration to initiate execution, the user is also able to provide a data buffer akin to a control and status register (CSR) bank to configure the accelerator. The user simply points Cohort to the virtually contiguous data block, which the programmer formats as an accelerator-specific struct. The Cohort-enhanced accelerator directly receives this configuration data at registration time before data is passed. For accelerators with CSRs configured by AXI-Lite, Cohort also supports writing the data over AXI-Lite.

We have so far connected four accelerators into our Cohort SoC. Section 5.2 describes the integration of three of these: SHA-256, AES, and an H264 encoder; we have also connected a short-time Fourier transform accelerator which we do not describe here. Each accelerator uses valid-ready handshakes for input and output data. We have demonstrated AXI-Stream functionality using an AXI-Stream FIFO as a "null" accelerator, which is easily replaceable with a more complete accelerator.

4.4 OS Support

Cohort provides a kernel driver to establish a complete and safe data flow and to abstract hardware details from the user. It shrinks the user space API to Queue Coherent semantics only and passes the low-level details to kernel via two simple system calls. Unlike many embedded accelerator environments, user space may not touch Cohort's configuration registers and need not be aware of physical addressing. This, in turn, *justifies the usage* of Cohort further, as functionality is solely, but safely implemented via the Cohort driver, without the need of wrappers. This also simplifies the usage of multiple Cohort engines using established OS abstractions. Our driver supports the following features:

- *MMU notifier for TLB flushes.* To maintain TLB coherence, Cohort's ISA-native MMU is also flushed alongside cores' TLBs. To enable this, MMU notifiers in Linux, normally used by platforms with IOMMUs, unified virtual memory, or hypervisors. The Cohort kernel driver registers its TLB flush function with an MMU notifier for processes at queue registration.
- *Page fault resolution via interrupt.* As noted in Section 4.2.4, when the Cohort MMU sees a page fault, it triggers an interrupt and invokes a handler registered by the Cohort kernel driver.
- *Registering queues with Cohort.* When a user mode application calls `cohort_register`, the driver virtually maps the Cohort engine's configuration register bank and registers the MMU notifier and page fault handler. Then, the driver writes the queue descriptor information to Cohort. After this simple syscall, the requested Cohort engine is ready to use.
- *Unregistering queues from Cohort.* Calling `cohort_unregister` deallocates and unmaps the prior resources.

The driver is first probed at kernel boot time to request interrupts and MMU notifier resources. When applications launch, they call `cohort_register` to register queues. The applications then push and pop data to and from Cohort via the registered queues. User space applications invoke `cohort_unregister` at exit to clear the allocated memory and MMU notifier service.

4.5 Cohort Capabilities and Extensions

Inter-thread and Inter-process Communication. Cohort is able to accelerate inter process communication with the same queue abstraction, just as two software threads can communicate via a

shared-memory queue. Such communication is commonly done by allocating the queue once and sharing its memory across two processes. This enables Cohort to receive input data from one process and produce output to another using the same push and pop methods as within a single process.

Accelerator Chaining. As more workloads necessitate hardware acceleration, accelerator chaining is a perfect way to accelerate complex computations. Cohort supports chaining together multiple accelerators using SPSC queues, bringing acceleration to more diverse workloads with minimal overhead and great flexibility. As an SoC designer moves to add new accelerators to their system, they can take existing decoupled software threads and directly replace them with Cohort-enhanced accelerators. The queues that are already used to communicate with the software thread are maintained, but registered with the relevant Cohort engine instead. This accelerator chaining capability also enables runtime reconfiguration of accelerator chains based on applications' needs.

Multi Producer/Consumer. Cohort sticks with the SPSC model, essentially a restricted dataflow where queues are not split or merged. Enabling queues supporting multiple producers or multiple consumers would provide value for a broader set of multithreaded use cases and for multiple accelerators to process data to/from a single queue. Generally these queues require atomic memory operations to guarantee correct operation, for which there is somewhat less standardisation of queue organisation. As a result, we leave support for these queues and design of their queue descriptors to future work.

5 EVALUATION METHODOLOGY

This section details our methodology, accelerator configurations, and benchmarks. We integrate Cohort into OpenPiton [8–10] and boot Linux (v5.6-rc4) on a Xilinx Alveo U200 FPGA running at 100 MHz. We use a four tile design with two 64-bit 6-stage Ariane RISC-V RV64GC cores and two accelerators (shown in Figure 2). We use the default OpenPiton configuration of 8KiB L1D, 16KiB L1I, 8KiB L1.5, and 64KiB 4-way L2 caches. The Cohort TLB has 16 entries and the producer and consumer endpoint accelerator interfaces are 64-bit wide. We use a minimal user-mode driver for our experiments.

5.1 Baselines

For our baselines, we repurposed a MAPLE decoupling unit [61] to connect with accelerators. In its original setting, MAPLE is an out-of-core, highly memory parallel load-store unit. It is designed to enable efficient data movement in manycores with slim in-order cores. Its particular focus is decoupled access-execute multithreading and prefetching models for applications with many indirect memory accesses, like graph processing. We modified MAPLE to instead host accelerators and provide MMIO-based and coherent DMA-based invocation (two common approaches). Performance counter data comes from each Cohort Engine, Ariane core or MAPLE unit.

MMIO Baseline. For some workloads, an MMIO-based invocation can make sense, as queue contents are not saved in cache and are hence disconnected from the coherence system. However, as noted in Section 3.2, MMIO often requires non-speculative round-trips from core to accelerator for each data word, a fact that was highlighted in the original presentation of MAPLE [61]. We use MAPLE's MMIO queue interface to provide data to and collect data from the accelerator with no coherence effects.

Coherent DMA Baseline. DMA enables bulk data movement between memory and the device without host interference. The DMA must be enabled and programmed by a core, and accelerators must still wait for sufficient data to operate. We use MAPLE's coherent LLC data prefetching feature to provide data to the accelerator and use the P-Mesh TRI [9] to coherently store results. Note that MAPLE's coherent DMA is more software efficient than typical DMA approaches as it uses a RISC-V MMU rather than requiring an IOMMU.

5.2 Accelerators

We adopt SHA-256, AES-128, and H264 accelerators for their streaming yet computationally-intensive nature. Their continuous data flow makes good use of the Cohort interface, while their computational complexity justifies acceleration. Figure 2 shows our Cohort SoC with two accelerators connected: SHA-256 and AES.

SHA-256. The first accelerator is an open-source SHA-256 cryptographic core [24], which is usable from Linux. Our accelerator accepts input in 512 bit data blocks and hashes them to a 256 bit irrevertible hash digest [27]. The accelerator accepts incoming 64 bit data blocks from the consumer endpoint and uses a ratchet to build a 512 bit input block. The hash digest is fed back using a ratchet to the producer unit in four chunks of 64 bits.

AES-128. Advanced Encryption Standard (AES) is a symmetric encryption algorithm [28]. In our prototype, we connected an open-source accelerator for AES encryption [2] which generates a ciphertext in blocks of 128 bits with a key of the same width. With Cohort we added a ratchet to consume 128 bits of data from 64 bit chunks from the consumer endpoint and the reverse for sending encrypted ciphertext to the producer endpoint. The encryption key is passed via a coherent CSR struct as described in Section 4.3.

H264 Encoder. H264 is a video compression standard widely used in the video industry. It provides good video quality at significantly reduced bitrates. We integrated an H264 encoder from Zexia [79] (using Context-adaptive variable-length coding or CAVLC) with the Cohort engine and confirmed its correct operation. This example also illustrates how Cohort can handle variable input size. The existing instance of the accelerator that we adopted accepts the number of frames at the start of its input to enable variable input length. The integration code includes a ratchet to prepare each frame for H264 in a similar manner to AES and SHA.

5.3 Benchmarks

To characterise the Cohort Engine, we run benchmark applications and test the entire system. Benchmark parameters are illustrated in Table 2.

Here batch size refers to an optimisation that updates the read and write pointers in batches instead of incrementally. This helps to reduce the coherency traffic in the system and improve performance.

Table 2: Benchmark Tuning Parameters.

Accelerators of Interest	AES, SHA
Communication Modes	Cohort, MMIO, DMA
Min/Max Queue Size	64/8192 elements
Min/Max Batching Factor	2/64 elements
Baseline DMA Granularity	256 Bytes

This is an optimisation that would be applied to multithreaded software but is also exploitable by Cohort, demonstrating Software-Oriented Acceleration in action. Below, we briefly describe our benchmark applications.

Benchmark Implementation in Cohort. Benchmarks in Cohort initialise the SPSC queues then push and pop the data in sequence. To hash 1 block of text we push 64 bits of data 8 times and fetch the corresponding hash with 4 pops. For AES, there are 2 pushes and 2 pops. As mentioned earlier, we encapsulate these movements into batches and run applications until queue size is reached.

Benchmark Implementation in Baselines. Benchmarks for our baselines are logically similar to Cohort's, but more complicated in their implementation. With MMIO, the core cannot achieve memory-level parallelism and so must receive the accelerator's output word by word before passing the next input word. This affects performance versus Cohort which can interleave input/output and expose MLP within batches. With Coherent DMA, special API functions (also containing MMIO writes) are called for each data block to be copied to/from the modified MAPLE unit, which matches common DMA programming mechanisms.

6 RESULTS

The goal of our evaluation is to demonstrate Cohort in a realistic heterogeneous SoC with multicore Linux support while providing a combination of programmability and performance improvements over the state-of-the-art (SOTA).

6.1 Latency Analysis

Figure 8 illustrates the latency to queue size relationship for our SHA benchmark running with different batch sizes and communication APIs. "Cohort batch=N" indicates the Cohort application optimisation of batching its write/read pointer updates to occur only after N elements have been copied into the queue. The same goes for AES in Figure 9.

Across the board, Cohort outperforms the MMIO and Coherent DMA baselines for the SHA application. Over all queue sizes tested, Cohort performs the best with larger batches and the batching optimisation works robustly. Cohort starts at a batch size of 8 elements to reflect one SHA input of 512 bits. Table 3 shows the speedup for Cohort AES and SHA with 64 element batches over the baselines as well as the improvement within Cohort brought by batching. The speedup brought by Cohort on SHA versus MMIO and DMA ranges from 5.44× to 8.38× for MMIO and from 7.27× to 11.24× for DMA. For SHA, batching increases performance by 2.32× to 3.33×.

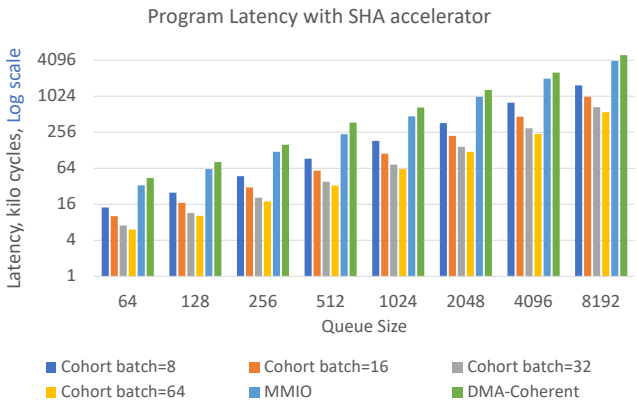

Figure 8: Latency in kilocycles to execute the SHA benchmark. X axis shows total number of queue elements, Y axis shows time in kilocycles (lower is better). Cohort batches queue elements in groups of 8 to 64.

Table 3: Summary of peak speedup for Cohort with AES and SHA (Cohort batch=64). Cohort shows consistent speedup over both baselines.

Queue size	4	128	256	512	1024	2048	4096	8192
SHA Speedup								
Vs MMIO	5.44	6.05	6.75	7.22	7.62	8.30	**8.38**	7.16
Vs DMA	7.27	7.94	8.85	**11.24**	10.70	10.83	10.62	8.97
W/ Batching	2.32	2.45	2.65	2.79	2.96	3.01	**3.33**	2.81
AES Speedup								
Vs MMIO	2.0	1.89	1.84	1.83	**2.07**	2.03	2.03	1.86
Vs DMA	1.9	1.83	1.74	1.71	1.75	**2.03**	1.94	1.69
W/ Batching	5.3	6.05	7.11	7.16	8.02	7.99	**8.10**	7.42

For AES, Cohort improves over the baselines as batch size increases. Batch sizes larger than 16 elements always perform equal or better than both MMIO and DMA baselines. Table 3 shows a range in speedup from Cohort on AES versus MMIO and DMA ranging from 1.83× to 2.07× for MMIO and from 1.69× to 2.03× for DMA. For AES, batching increases performance by 5.30× to 8.10×. The lower performance for AES comes from two factors: the first is its symmetric data movement where AES produces output of the same size as the input, which increases false sharing on the read and write pointers with the Ariane core as it waits to acquire the output data. The second factor is the accelerator's lower latency of 41 cycles versus SHA's latency of 66 cycles.

These numbers demonstrate the importance of how, with Cohort, optimisations applied in software can bring valuable speedups at the accelerator interface. These results come in line with our expectations in Section 3 and prove the importance of Software Oriented Acceleration and the benefits of the SPSC queue abstraction exploited by Cohort.

6.2 IPC Analysis

Beyond seeing an overall improvement in latency, we argued that Cohort would make more efficient use of the core as it provides data to and reads data from accelerators. Figure 10 and Figure 11

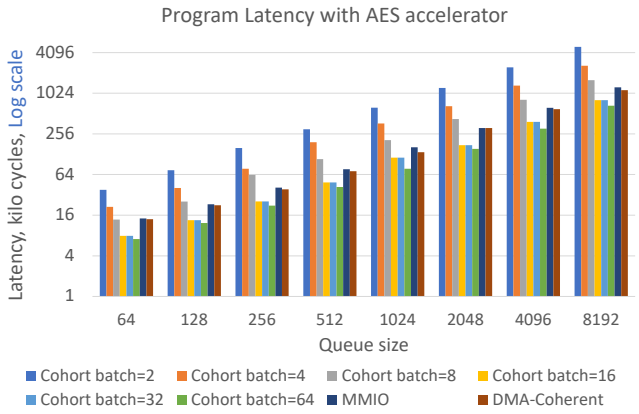

Figure 9: Latency in kilocycles to execute the AES benchmark. X axis shows total number of queue elements, Y axis shows time in kilocycles (lower is better). Cohort batches queue elements in groups of 2 to 64.

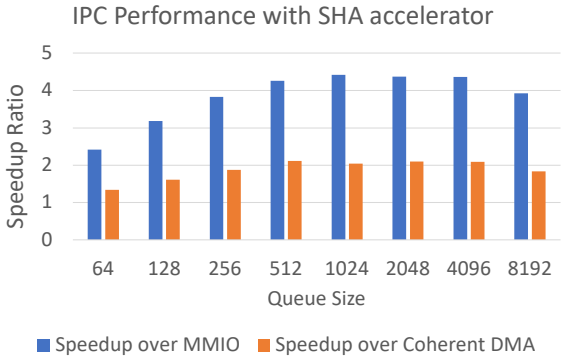

Figure 10: Instructions Per Cycle (IPC) speedup of Cohort over the baselines achieved while executing the SHA benchmark. X axis shows total number of queue elements, Y axis shows IPC speedup (higher is better). Cohort uses a batching factor of 64.

show significant improvements in IPC for Cohort over the MMIO and DMA approaches, where cores must make MMIO round trips to communicate with the accelerators. Cohort provides a peak IPC speedup for Cohort SHA over MMIO and DMA of 4.42× and 2.11×. For AES, Cohort achieves a peak speedup in IPC over MMIO and DMA of 2.83× and 1.77×, respectively. These numbers validate that Cohort better utilises the core while data transfer is occurring.

6.3 Area Overhead

We perform FPGA implementation with Xilinx Vivado 2022.1 and report post-synthesis resource utilisation in Table 4. The empty Cohort engine comprises around 10% of the LUTs and 20% of the registers of a Cohort tile, or less than 4% of the LUTs and 10% of the registers of an Ariane tile. A tile with an empty Cohort Engine is about 39% and 46.6% of the Ariane tile by LUTs and registers (both tiles feature OpenPiton's NoC routers and L1.5 and

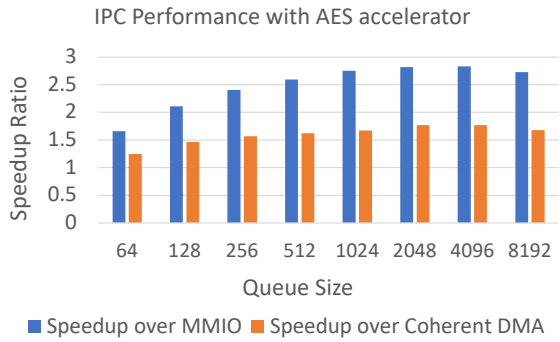

Figure 11: Instructions Per Cycle (IPC) speedup of Cohort over the baselines achieved while executing the AES benchmark. X axis shows total number of queue elements, Y axis shows IPC speedup (higher is better). Cohort uses a batching factor of 64.

L2 caches). Despite this small area, Cohort packs a punch in terms of the functionality and features it provides.

The Cohort engine has roughly 68% the LUTs and 45% the registers of the AES accelerator while consuming no BRAM, compared to the large memory needed for AES which would consume considerable SRAM in an ASIC process (note that the AES BRAM is larger than that of an Ariane tile with its previously stated cache configuration totalling around 100KiB). The Cohort engine consumes roughly 27% more LUTs and 57% more registers than the relatively small SHA accelerator. Compared to the H264 encoder, the empty Cohort engine consumes around 37% of the LUTs and 71% of the registers. H264 consumes 4 BRAM slices (tens of kilobytes of memory) and unlike Cohort or the other accelerators, it also consumes 6 DSP slices. Full tiles including Cohort plus AES, SHA, or H264 are significantly smaller than the size of an Ariane tile.

The MMU itself is area efficient, consuming 1081 LUTs, 1206 registers, and no BRAM. Of that, the TLB makes up 911 LUTs and 1029 registers, while the page table walker makes up 168 LUTs and 109 registers. This small MMU leads to savings in memory (no need for separate I/O page tables), runtime (less page table management in software), and software/OS porting. While our demonstrated accelerators are relatively small, the same MMU and Cohort engine are usable with larger accelerators.

7 RELATED WORK

As Moore's Law has slowed and Dennard Scaling has ended, system designs are increasingly exploiting heterogeneous parallelism and domain-specific accelerators to scale performance at acceptable power [23, 64]. Using accelerators often relies on the programmer configuring them manually [32, 63] or through a domain specific language, for the class of applications that are being executed [44, 82].

Decoupling of data-access and compute was originally proposed by J. Smith [69] to mitigate latency of data-supply to processor cores, by having a core accessing the data and feeding it to another core, which handles the computation. After that, several hardware implementations have been proposed, where data communication

Table 4: FPGA Synthesis results for resource utilisation: Cohort with or compared to Ariane, MAPLE, and accelerators

Resource Type	Ariane Tile	Empty Cohort Tile	Empty Cohort Engine	Cohort + AES	Cohort + SHA	MAPLE + AES + SHA	AES Only	SHA Only	H264 Only
LUTs	67083	26390	2594	6679	4524	21066	3837	2041	6851
Registers	39879	18591	3799	12176	6064	28276	8531	2420	5341
BRAM	41.5	9.5	0	47.5	0	47.5	47.5	0	4

occurs through architectural queues [33, 51, 78]. These works aim to hide memory latency as a simpler alternative to superscalar processors. DeSC [37, 38] repurposed decoupling to supply computation-exclusive units like accelerators. DeSC's programming model offers a producer-consumer relationship between heterogeneous processing units. However, DeSC uses architectural queues that are exposed to software through new ISA instructions, which requires software changes. Furthermore, there is a strict one-to-one relation between supply and compute elements, determined at tapeout time.

Data streaming proposals also have this restriction. MAD [41] and HWPEs [19, 22] have a data-access engine optimised for dataflow computation, which is integrated with cores or accelerators to perform the memory-access portion of programs.

Cohort offers similar producer-consumer relationships without restricting them in hardware, and making them transparent to the ISA. In Cohort, accelerators can be configured at runtime to be consuming from or producing to any core or accelerator on-chip. This creates a more rich space of possible heterogeneous communication patterns.

Although MAPLE [61] offers a flexible decoupling mechanism where cores communicate via a network-connected memory-fetching engine, it still requires software changes with explicit produce and consume operations. Moreover, MAPLE's communication scheme resorts to side-effectful, non-idempotent memory-mapped I/O instructions, which sets a significant minimal latency threshold between devices. Cohort communicates using a simple software queue, which enables greater speculation and parallelism of produce and consume operations.

Loosely coupled and tightly coupled accelerators. Accelerators come in different flavors and sizes, but we classify them regarding their hardware integration into tighly-coupled and loosely-coupled [20]. Often, the classification is in terms of where the accelerator sits in the architecture, namely, close to the CPU or not.

In the tightly-coupled category, we can find accelerators as a functional (computing) unit physically attached to the CPU pipeline (e.g. FPU, or inline accelerators [73]), and accelerators that share the private cache with the CPU (e.g. RocketChip [4], Hwacha [48], and Gemmini [31]). In the loosely-coupled category, we find accelerators placed on the main chip interconnect [14, 39, 59], or placed on a separate chip which communicates to the chip interconnect through an I/O interface (e.g. PCIe).

Comparison with interconnects CXL and UCIe are emerging interconnect specifications that tackle host-to-host and chiplet-to-chiplet connections respectively. Although CXL enables inter-host coherent communication at a low latency and high throughput, it has a large area and power overhead associated with underlying PCIe protocol stack. By contrast, our implementation of Cohort tackles efficient coherent communication within a single SoC. Cohort would also be implementable on top of the CXL coherency system with the correct semantics. UCIe takes a layered approach to accommodate challenges in die-to-die communication for emerging chiplets. Currently the protocol layer supports four standards: PCIe 6.0, CXL3.0, CXL2.0 and raw data streaming. We envision Cohort would be implemented on top of emerging coherency protocols in new UCIe protocol layers, or atop CXL as previously described. However, the control and physical layers of UCIe are out of scope for this paper.

Area-wise, in a state of the art design such as Intel Alder Lake fabricated with 7nm process, an 8 lane PCIe 5 PHY takes $2.4mm^2$ [50]. By contrast, a high performance CPU core on the same chip with L1 cache but without L2 cache or power gating logic takes around $5.37mm^2$, only a little over twice the PCIe PHY block size. Note that this does not take into account the area of the PCIe controller which would also be required.

System-level considerations. The advantage of loosely-coupled accelerators is that they do not modify the CPU or the ISA. They can be integrated in a relatively plug-and-play manner, through the interconnect, overcoming the verification overhead of tightly-coupled integration. This is aligned with trends in fast chip prototyping [1, 52, 77], where an SoC is made of reusable, third-party IP blocks. However, tight-coupled accelerators have the advantage of having the CPU dealing with VM translation. Loosely-coupled, driver-triggered accelerators like those of ESP [52] have the strategy to pre-fill a specialised TLB and pin memory so that no page-faults can occur, while NVDLA [59] uses a reserved, non-cacheable memory region, that the OS cannot use, and only the accelerator and its device driver have access to it. Cohort enjoys the advantages of loosely-coupled accelerators while supporting VM in a more flexible manner by incorporating a fully-capable MMU that handles page faults and TLB shootdowns.

Programmability. While tightly-coupled accelerators are triggered by ISA instructions, and thus the result goes to the register file [4, 73], loosely-coupled ones often rely on interrupts or spin-polling for the CPU to know that they finished. Several works have enhanced the I/O software stacks, by reducing interrupt overheads [42, 67], or combining them with spin-polling as a hybrid notification mechanism [25]. Recently, HyperPlane [57] proposed a hardware mechanism to accelerate these notifications. However, these techniques focus on I/O accelerators, while Cohort leverages the coherence protocol to get notified that another processing element has produced or consumed data, thus also supporting on-chip accelerators. Cong et al. [17, 18] proposed an allocation protocol to avoid OS overhead in scheduling tasks to on-chip accelerators.

However, this still requires large workloads to be efficient, while Cohort is also suited for finer-grain tasks.

M3 [6], M3X [5], Rackscale Microkernel [40], Solros [56] bake hardware semantics into OS mechanisms for tight integration. All target simple, efficient, and scalable heterogeneous communication, but implement brand new (micro)kernels (and new DTU hardware [6, 40]). Cohort exploits existing OS and multi-threaded application best practices, via well known queue semantics and enables accelerators and cores to share queues in coherent memory for communication. We improve performance over the baselines with low cost and a more intuitive API. Similarly to our MMIO baseline, DTU queues are either kept in (limited-size) scratchpads and modified by MMIO or, similarly to our DMA baseline, are kept in DRAM with pointers updated by MMIO and NoC messages. Both DTU mechanisms require software changes, unlike Cohort.

Acceleration standards A number of accelerator standardisation efforts are taking place, particularly to standardise on programming models for GPGPUs and similar programmable accelerators. Vulkan [35] and SPIR-V [34], with features like pipelined barriers, expose deeper hardware details in a traditional, hardware-oriented fashion, implying a deep asymmetry between the accelerators and cores. Cohort on the other hand effectively embeds common software semantics into the accelerator integration environment (note: not the accelerator itself). Cohort-enabled accelerators can be decoupled from memory operations and act as peers to cores in a software-oriented manner. By leveraging software semantic awareness to provide a high performance API that is implementation agnostic, we see Cohort as additional, rather than a competitor to these solutions. For accelerators designed or programmed with Vulkan/SPIR-V, any communication of a SBIO nature (whether through explicit SPSC queues or other data movement behaviours) could target a specialised Cohort engine instead of the usual LSU for a potential uplift in performance.

Queue Libraries and Language Support Beyond adoption for Boost, we are investigating using Cohort with Unix pipes. Of further interest for high-performance, asynchronous acceleration use cases is the Linux `io_uring` subsystem [68]. The `io_uring` API has enabled a variety of new high-performance I/O use cases in Linux, particularly for networking and file I/O. Integrating Cohort with `io_uring` would enable a rich runtime for managing accelerators. More intriguingly, Cohort accelerators could also use `io_uring` to request services from the kernel via their native queue interfaces.

With simple runtime support (comparable to the library support we added for Boost), languages could automatically retarget their queues to make use of Cohort. We leave automatic identification of queues to future work.

8 CONCLUSION

In this paper we argue for the importance of software-oriented acceleration, bridging the gap between hardware-centric SoCs and software programmability. We propose Software-Oriented Acceleration (SOA), an elegant and pragmatic solution for orchestrating the ever-increasing heterogeneity of modern SoCs. To back the idea up, we identify Queue Coherence semantics for communication between accelerators and present the Cohort engine and API support to enhance common producer-consumer software queues in a full-system heterogeneous environment. This concept makes SoCs more easily programmable while remaining performant. Cohort's implementation spans the application to architecture stack and shows powerful performance improvements ranging from 1.83× to 8.38× over MMIO, and from 1.69× to 11.24× for DMA baselines. The hardware and software of Cohort, plus a subset of the accelerators, are available open-source at https://github.com/cohort-project.

ACKNOWLEDGMENTS

This work was partially funded by a grant from Western Digital Technologies, Inc. We thank Daniel Jiménez Mazure, Katie Lim, Guillem López-Paradís, Brian Li, Arjun Vinod, and Guy Wilks for providing code and other useful assistance needed to realise Cohort.

REFERENCES

[1] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21.

[2] Salah Amr. 2013. AES-128 Encryption Core. OpenCores AES-128 Pipelined Encryption Implementation in Verilog , https://opencores.org/projects/aes-128_pipelined_encryption.

[3] Arm Ltd. [n. d.]. AMBA 4 AXI4-Stream Protocol Specification. https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol.

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[5] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. 2019. M3X: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 617–631.

[6] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 189–203. https://doi.org/10.1145/2872362.2872371

[7] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. 2015. MIAOW - An open source RTL implementation of a GPGPU. In *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. 1–3. https://doi.org/10.1109/CoolChips.2015.7158663

[8] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Third Workshop on Computer Architecture Research with RISC-V, CARRV*, Vol. 19.

[9] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. 2020. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *ASPLOS* (Lausanne, Switzerland) *(ASPLOS '20)*. ACM, 699–714. https://doi.org/10.1145/3373376.3378479

[10] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *ASPLOS*. ACM, 217–232.

[11] Tim Blechmann. [n. d.]. Boost.Lockfree. https://www.boost.org/doc/libs/1_79_0/doc/html/lockfree.html.

[12] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (2001), 1059–1076. https://doi.org/10.1109/43.945302

[13] Luca P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. IEEE* 103, 11 (2015), 2133–2151. https://doi.org/10.1109/JPROC.2015.2480849

[14] Luca P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proceedings of the 53rd Design Automation Conference (DAC)*. 17:1–17:6.

[15] CCIX Consortium. [n. d.]. An Introduction to CCIX. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf.

[16] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[17] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-Rich Architectures: Opportunities and Progresses. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*. Article 180. http://doi.acm.org/10.1145/2593069.2596667

[18] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. Architecture support for accelerator-rich CMPs. In *Proceedings of the 49th Annual Design Automation Conference*.

[19] F. Conti, P. D. Schiavone, and L. Benini. 2018. XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018). https://doi.org/10.1109/TCAD.2018.2857019

[20] Emilio Cota, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca Carloni. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proceedings of the 52nd Design Automation Conference (DAC)*.

[21] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[22] Masoud Dehyadegari, Andrea Marongiu, Mohammad Reza Kakoee, Siamak Mohammadi, Naser Yazdani, and Luca Benini. 2014. Architecture support for tightly-coupled multi-core clusters with shared-memory HW accelerators. *IEEE Trans. Comput.* 64, 8 (2014), 2132–2144.

[23] Allison McCarn Deiana, Nhan Tran, Joshua Agar, Michaela Blott, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Scott Hauck, Mia Liu, Mark S Neubauer, et al. 2021. Applications and Techniques for Fast Machine Learning in Science. *arXiv preprint arXiv:2110.13041* (2021).

[24] Jonny Doin. 2016. SHA256 Hash Core. OpenCores SHA256 implementation in VHDL, https://opencores.org/projects/sha256_hash_core.

[25] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. 2001. HIP: hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review* 35, 4 (2001), 50–60.

[26] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (jul 2018), P07027. https://doi.org/10.1088/1748-0221/13/07/P07027

[27] Morris Dworkin. 2017. Hash Functions. NIST Information Technology Laboratory, Overview on Hash Functions, https://csrc.nist.gov/projects/hash-functions.

[28] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. 2001. Advanced Encryption Standard (AES). https://doi.org/10.6028/NIST.FIPS.197

[29] Murali Emani, Venkatram Vishwanath, Corey Adams, Michael E Papka, Rick Stevens, Laura Florescu, Sumti Jairath, William Liu, Tejas Nama, and Arvind Sujeeth. 2021. Accelerating scientific applications with SambaNova reconfigurable dataflow architecture. *Computing in Science & Engineering* 23, 2 (2021), 114–119.

[30] Esperanto Technologies. [n. d.]. Esperanto's ET-Minion on-chip RISC-V cores. https://www.esperanto.ai/technology/.

[31] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925* 3 (2019).

[32] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. 2020. ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning. In *DATE*. IEEE Press.

[33] James R Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R Pleszkun, PB Schechter, and Honesty C Young. 1985. PIPE: a VLSI decoupled architecture. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 20–27.

[34] The Khronos Group. [n. d.]. SPIR-V Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

[35] The Khronos Vulkan Working Group. 2022. Vulkan 1.3.232 - A Specification (with all registered Vulkan extensions). https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.

[36] Linley Gwennap. 2020. Groq rocks neural networks. *Microprocessor Report, Tech. Rep., jan* (2020).

[37] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled Supply-compute Communication Management for Heterogeneous Architectures. In *MICRO*. ACM.

[38] Tae Jun Ham, Juan L. Aragon, and Margaret Martonosi. 2019. Efficient Data Supply for Parallel Heterogeneous Architectures. *ACM TACO* 16, 2, Article 9 (2019), 23 pages. https://doi.org/10.1145/3310332

[39] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2016.7783759

[40] Matthias Hille, Nils Asmussen, Hermann Härtig, and Pramod Bhatotia. 2020. A Heterogeneous Microkernel OS for Rack-Scale Systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 50–58. https://doi.org/10.1145/3409963.3410487

[41] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 118–130. https://doi.org/10.1145/2749469.2750390

[42] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2011. Sleepy Sloth: Threads as interrupts as threads. In *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 67–77.

[43] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[44] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[45] Simon Knowles. 2021. Graphcore. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–25.

[46] Venkata Krishnan, Olivier Serres, and Michael Blocksome. 2020. Configurable Network Protocol Accelerator (COPA). *IEEE Micro* 41, 1 (2020), 8–14.

[47] Aki Kuusela and Clint Smullen. 2021. Video Coding Unit (VCU). In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–30. https://doi.org/10.1109/HCS52781.2021.9567040

[48] Yunsup Lee. 2016. *Decoupled Vector-Fetch Architecture with a Scalarizing Compiler*. Ph. D. Dissertation. University of California, Berkeley.

[49] Sean Lie. 2021. Multi-Million Core, Multi-Wafer AI Cluster. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE Computer Society, 1–41.

[50] Locuza. [n. d.]. Die walkthrough: Alder Lake-S/P and a touch of Zen 3. https://locuza.substack.com/p/die-walkthrough-alder-lake-sp-and.

[51] William Mangione-Smith, Santosh G Abraham, and Edward S Davidson. 1990. The effects of memory latency and fine-grain parallelism on astronautics ZS-1 performance. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, Vol. 1. IEEE, 288–296.

[52] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC Development with Open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 96, 9 pages. https://doi.org/10.1145/3400302.3415753

[53] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a Microkernel Approach to Host Networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*. New York, NY, USA.

[54] Mediatek. 2022. Mediatek 8195 Linux device tree. https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/mediatek/mt8195.dtsi.

[55] Mediatek. 2022. Qualcomm Snapdragon 8 Gen 1 SM8450 Linux device tree. https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/qcom/sm8450.dtsi.

[56] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. 2018. Solros: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 36, 15 pages. https://doi.org/10.1145/3190508.3190523

[57] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. 2020. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 852–867. https://doi.org/10.1109/MICRO50266.2020.00074

[58] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[59] NVIDIA. [n. d.]. NVIDIA Deep Learning Accelerator. http://nvdla.org/.
[60] Nvidia. 2022. Nvidia Tegra 234 Linux device tree. https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/nvidia/tegra234.dtsi.
[61] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but Mighty: Designing and Realizing Scalable Latency Tolerance for Manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 817—-830. https://doi.org/10.1145/3470496.3527400
[62] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 27–40.
[63] L. Piccolboni, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2017. Broadening the exploration of the accelerator design space in embedded scalable platforms. In *HPEC*. IEEE Press.
[64] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.
[65] Ltd. Samsung Electronics Co. 2022. Samsung Exynos 5433 Linux device tree. https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/exynos/exynos5433.dtsi.
[66] Ltd. Samsung Electronics Co. 2022. Samsung Exynos 7 Linux device tree. https://github.com/torvalds/linux/blob/master/arch/arm64/boot/dts/exynos/exynos7.dtsi.
[67] Robert T Short, John M Parchem, and David N Cutler. 1998. Method and apparatus for reducing the rate of interrupts by generating a single interrupt for a group of events. US Patent 5,708,814.
[68] Jakub Sitnicki. 2022. Missing Manuals - io_uring worker pool. https://blog.cloudflare.com/missing-manuals-io_uring-worker-pool/.
[69] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Press.
[70] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal of Research and Development* 59, 1 (2015), 7–1.
[71] Michael Bedford Taylor. 2018. Basejump STL: Systemverilog Needs a Standard Template Library for Hardware Design. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 73, 6 pages. https://doi.org/10.1145/3195970.3199848
[72] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. 2021. Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 754–766. https://doi.org/10.1145/3466752.3480128
[73] David Trilla, John-David Wellman, Alper Buyuktosunoglu, and Pradip Bose. 2021. NOVIA: A Framework for Discovering Non-Conventional Inline Accelerators. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 507–521.
[74] S. Van Doren. 2019. Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. 18–18. https://doi.org/10.1109/HOTI.2019.00017
[75] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, Yulong Li, Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Marcel Schaal, Mauricio Serrano, Jungwook Choi, Xiao Sun, Naigang Wang, Chia-Yu Chen, Allison Allain, James Bonano, Nianzheng Cao, Robert Casatuta, Matthew Cohen, Bruce Fleischer, Michael Guillorn, Howard Haynie, Jinwook Jung, Mingu Kang, Kyu-hyoun Kim, Siyu Koswatta, Saekyu Lee, Martin Lutz, Silvia Mueller, Jinwook Oh, Ashish Ranjan, Zhibin Ren, Scot Rider, Kerstin Schelm, Michael Scheuermann, Joel Silberman, Jie Yang, Vidhi Zalani, Xin Zhang, Ching Zhou, Matt Ziegler, Vinay Shah, Moriyoshi Ohara, Pong-Fei Lu, Brian Curran, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. 2021. RaPiD: AI Accelerator for Ultra-low Precision Training and Inference. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 153–166. https://doi.org/10.1109/ISCA52012.2021.00021
[76] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/502034.502057
[77] P. N. Whatmough, M. Donato, G. G. Ko, S. K. Lee, D. Brooks, and G. Wei. 2020. CHIPKIT: An Agile, Reusable Open-Source Framework for Rapid Test Chip Development. *IEEE Micro* 40, 4 (2020), 32–40.
[78] Wm A Wulf. 1992. Evaluation of the WM Architecture. In *Proceedings of the 19th annual international symposium on Computer architecture*. 382–390.
[79] Zexia. [n. d.]. H.264 Hardware Encoder in VHDL. https://hardh264.sourceforge.net/H264-encoder-manual.html.
[80] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569
[81] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.
[82] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.
[83] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. 2021. Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 350–365. https://doi.org/10.1145/3466752.3480065

Received 2022-10-20; accepted 2023-01-19