Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL

Sangjin Lee[1] Alberto Lerner[1] Philippe Bonnet[2] Philippe Cudré-Mauroux[1]
[1]University of Fribourg
Switzerland
[2]University of Copenhagen
Denmark

ABSTRACT

Flash memory is the *de facto* standard for data persistence in data-intensive systems. Despite its benefits, this type of memory has at least one severe disadvantage: it is offered only as part of tightly closed Solid-State Drives (SSDs). To access an SSD, applications need to resort to one of many possible I/O frameworks, which themselves are wrappers around a block interface abstraction, the NVMe standard. These levels of indirection impact how applications are structured and prevent them from benefiting from the full power of Flash-based devices.

In this paper, we argue that SSDs should instead interact with applications via CXL. CXL is a new technology driven by an Intel-led consortium that allows systems to maintain coherence between a host's memory and memory from attached peripherals. With CXL, a device can expose a range of Flash-backed addresses through the server's memory. One implementation option is to allow applications to read and write to that range and let the device convert them to Flash operations. In our SSD, however, we pick a different option. The device exposes what we call a Database Kernel (DBK) through a CXL-backed memory range. Read/writes against a kernel would trigger database-centric computations that the kernel would perform inside the device. We show examples of DBKs to support different database functionalities and discuss their benefits. We believe that CXL and Database Kernels can support a new generation of heterogeneous database platforms with unprecedented efficiency, performance, and functionality.

1 INTRODUCTION

Storage layers API galore. Applications nowadays can access many different memory types, ranging from caches, DRAM (local and remote), Persistent Memory (ditto), and NAND Flash. This breadth of alternatives is necessary because each memory type supports a storage layer with a distinct size, speed, latency, persistence, and cost trade-off. Since no single memory type can outperform all the others in all criteria, having many storage layers gives applications some necessary flexibility.

Dealing with most of those storage layers is relatively straightforward. For instance, caches and DRAM are transparent to applications; they see these layers as a continuous memory region that can be directly read and written with simple `loads` and `stores` instructions. However, this simplicity is sacrificed when applications require persistent memory. The storage layers providing persistence come with much heavier abstractions.

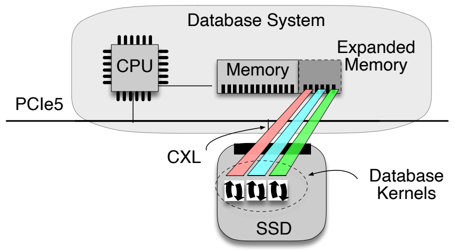

Figure 1: CXL allows a host's memory to be expanded by a peripheral device. Both host and device can update the memory, i.e., CXL guarantees coherence. Database Kernels use the memory to provide services to the database by changing the semantics of specific regions of CXL memory. Writing to a particular area may use a low-latency data path—useful for logging, for instance. Reading from another area may return the result of a table scan instead of the table itself. Reading/Writing to a third area may decompress/compress the contents along the way.

Currently, the best persistent memory option is arguably NAND Flash, the underlying storage medium of SSDs. To use fast SSDs, and thus Flash, an application must resort to NVMe and, most likely, an additional I/O framework to issue fast block reads and writes [9]. These layers force applications to be structured around asynchronous API calls instead of the much simpler `load/store` instructions. Even with these APIs, properly coupling a database system with an SSD requires much effort and may not unlock the device's entire performance [15].

CXL as a storage layer unifier. Recently, a technology called Compute Express Link (CXL) [7] emerged that can bring back simplicity. CXL promises to allow memory of any type sitting on peripheral cards or even in remote machines to be accessed directly by applications as if they were local DRAM. At its heart, CXL is a cache coherence protocol [31]. For decades, these protocols have allowed multi-socket servers to offer applications a unified view of memory. These protocols, however, have been closely guarded and have all been proprietary. CXL's most significant advantage is, arguably, that it is public. It promises to support interoperability across different manufacturers' devices.

This allows third parties to build, for instance, so-called *memory expanders*, PCIe form factor daughter cards that contribute extra memory to a host, be it Intel- or AMD-based, or any future server that would support the protocol [24, 29]. Figure 1 (top) depicts this scenario. CXL characteristics are already known [33], and major hyperscalers, such as Microsoft and Meta, have already announced potential adoption plans [21, 23].

SSD storage can join CXL but do more than memory expansion. The CXL standard, however, does not clearly address how to incorporate a persistence storage layer such as the one provided by an SSD. Flash has widely different characteristics than DRAM and requires completely different maintenance procedures. Even so, one can imagine that a CXL memory expander could simply mask these differences and use Flash to back a CXL memory address range. An I/O against a Flash-backed CXL address would be somehow converted into an I/O against Flash. Put differently, an application would not need intermediate frameworks to access this device and could thus have a more natural structure. This semantics is viable and is one of the CXL-Flash integration alternatives proposed by this paper and some prototypes [14, 28].

This paper, however, goes beyond this level of integration. We introduce a new CXL-Flash coupling option that allows parts of a database system to be implemented in the device, in what is commonly referred to as Near-Data Processing (NDP) [2]. Storage devices have been steadily gaining such capabilities [10, 11, 18, 19]. We call our new abstraction Database Kernels (DBK). DBK leverage the CXL technology to export an address space to the database system. However, instead of implementing a 1:1 mapping into Flash, I/Os against this area trigger computations inside the device. Figure 1 also depicts this possibility.

The DBK abstraction allows moving different aspects of database systems into the device. For instance, early predicate execution can be implemented as a DBK. The device could take a predicate description through an address within the CXL-backed window and present a materialized view of the result in the rest of the area. As with other kernels, we expect the execution to consume fewer resources (e.g., transfers), to be faster, and to liberate the host CPU for other tasks. We will discuss how different classes of DBKs can extend the database functionality into the device.

To execute DBKs, we proposed a new SSD architecture. This SSD is more modular than a traditional one and opens access to some of the device's internal interfaces. The interfaces are designed to shield non-specialized programmers from unnecessary intricacies while allowing them to develop NDP database functionality.

The structure of this paper follows its contributions, which are summarized below:

- We start by briefly explaining the main CXL tenets and present techniques to extend the use of CXL to storage devices (Section 2).
- We introduce the concept of Database Kernels as a means to execute data-intensive tasks in a near-data-processing fashion and show a device architecture to support such kernels (Section 3).
- We present the different types of CXL features that can be leveraged by DBKs (Section 4).
- We show examples of DBK in the context of database systems (Section 5).
- We discuss the viability of implementing Database Kernels devices in practice (Section 6).
- We enumerate the research questions that require further investigation to realize Database Kernels supporting devices (Section 7).

We discuss related work in Section 8 and conclude in Section 9.

2 BACKGROUND & MOTIVATION

Memory Coherence and Caching. Applications perceive memory as a continuous range of addresses that can be freely accessed. Providing this level of abstraction in the modern memory hierarchy is not trivial. The issue is that modern CPUs have many cores, each with a private cache. When several cores access a given memory address simultaneously, the data at the given address may be copied and possibly updated in multiple caches. The discipline that governs how cores access data copies for the same memory address is called Memory (or Cache) Coherence.

Formally, coherence can be defined in various ways through invariants [31]. A simple definition is the following: (1) writes to the **same memory location** are serialized; and (2) **every write** is eventually made visible to all cores[1]. Informally, a typical implementation of coherence allows many copies of a piece of data to exist in different caches, provided no core modifies them. If a core wants to modify its copy, it must acquire an exclusive version, invalidating all existing copies before proceeding.

Such an implementation relies on two components. The *Directory Controller* keeps track of which memory addresses are cached, by whom, and in which state. The *Cache Controller* requests cache addresses to the directory controller and receives invalidation requests from it. Figure 2 (left) depicts a simplified implementation of this architecture. Note that this implementation scales well to multi-socket systems, by associating a Directory Controller to each socket and a Cache controller to each cache. The memory addresses are assigned to Directory Controllers in such a way that each Cache Controller can tell if a request should be directed to the local or a remote Directory Controller. Figure 2 (left) depicts this scenario.

CXL versions and Device Types. The first version of CXL, called 1.1, extends Memory Coherence to caches located on local peripherals. We are starting to see the first products emerging in the market that support that CXL version, e.g., servers [12] and memory expanders [28]. Two additional versions of CXL are already ratified. CXL 2.0 enables single-level switching, i.e., Memory Coherence is supported across multiple hosts and multiple devices connected through a single switch. CXL 3.0 extends Memory Coherence to multiple switches over various interconnects and fabrics protocols.

CXL also adopts the concept of a Directory and a Cache controllers—although it uses different terminology— dividing the message types that make up the protocol into two. The messages originating from the Cache Controller form the subprotocol named `cxl.cache`. The messages originating from the Directory controller form the `cxl.mem` sub-protocol. A peripheral can implement only the `cxl.cache` protocol as a *Type 1 device*, both protocols as a *Type 2 device*, or only the `cxl.mem` protocol as a *Type 3 device.*

Figure 2 (right) depicts a Type 3 device. It is suitably called a *memory expander* because its goal is to provide additional memory to a server without caching the latter's memory. In other words, only CPU cores on the server side will cache contents of the memory the device is providing. For that reason, it only needs to implement `cxl.mem`. The figure shows that the device implements a Directory Controller—called a Device Controller here.

[1]Note the emphasis on a single memory location. The discipline that governs the order of accesses to multiple addresses is a different one: Memory Consistency [25]. Memory coherence and consistency are orthogonal concepts.

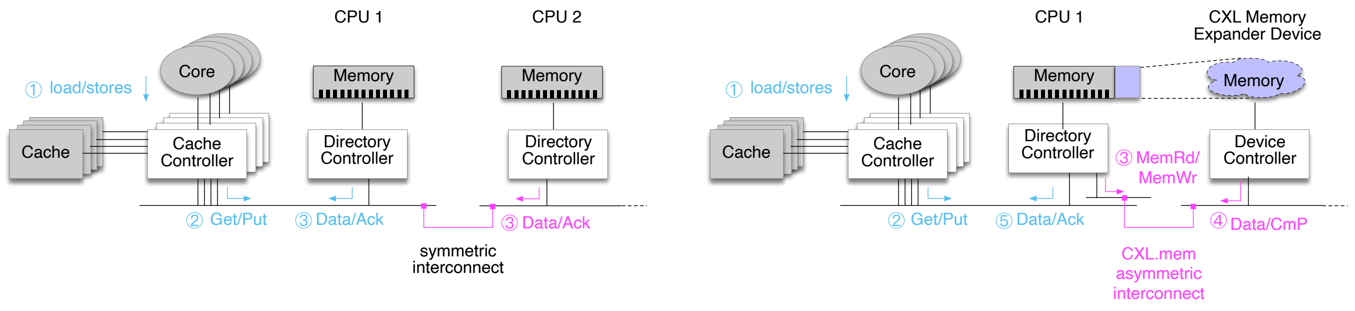

Figure 2: (Left) Coherence across sockets: To access or modify the contents of a memory address, a core brings a copy of it to its cache ①. This can be triggered by issuing a `load` or a `store` instruction. Upon receiving the instruction, the Cache Controller issues a request to either get a copy or put (write) its copy of the modified content from/back to memory ②. The Directory Controller receives this message and executes the required memory access, either sending a copy of the read data to the Cache Controller or acknowledging that the modified data was written ③. The Cache Controller can then signal to the core that the instruction is complete. Note that if the address required were held by a remote Directory Controller, the Cache Controller would have targeted it instead ③. (Right) Coherence with a memory expander device: The Cache Controller asks or sends a cache line as before but is unaware of who is backing that address. Upon noticing that the request is for the expanded memory area, the Directory Controller issues the proper command to the Device Controller ③, which in turn interacts with the local memory ④ and responds. It is the Directory Controller that sends the cache line or the acknowledgment back as if the line accessed was local ⑤.

Curiously, CXL imposes a hierarchy of Directory Controllers. A Cache controller cannot talk directly with a Device Controller because it does not know whether it is accessing a range of memory that it manages. The Directory Controller on the host mediates all communications. This type of coherence protocol is called *asymmetric*. Note that in Figure 2 (left), we portray a *symmetric* coherence protocol; there is no hierarchy between the Directory Controllers of each CPU. There have been debates on the relative merits of the approaches. CXL proponents argue that asymmetric protocols are simpler. Symmetric protocol proponents argue that more balanced systems may be built if every peripheral that offers memory to the system controls its own memory. It seems, however, that the Industry decided to proceed with asymmetric protocols.

A naive type 3 Flash-based CXL device. The standard recognizes memory expanders, but nothing in it constrains what type of memory can back the address range the expander adds to the Directory Controller. In fact, recent versions of the standard even stipulate ranges of tolerable latency for transactions (request-response message pairs) issued against the device. Some of these ranges are well within the response times of NAND Flash-based devices. Naturally, one can conceive of a Type 3 device whose address range would be backed by that kind of memory and, therefore, be persistent (e.g., Samsung's recent Memory-Semantics SSD Prototype [28]).

Although the exercise of building such a storage device is interesting, we think it would deliver subpar response times. As we will substantiate shortly, the device would only be notified of a write operation when it is about to complete, leaving very little time to hide the Flash-memory latency. Instead, we claim that a device with this functionality is better realized by a Type 2 device that is allowed to hold a cache—and implement a cache controller—of its own. Cache Coherence here is a means to give early notice of on-going write operations to the NAND-based device.

3 DEVICE ARCHITECTURE

We propose a storage device that supports simple and efficient access to expanded memory (including Flash—but not only) through memory reads and writes, and rich semantics associated to operations on a given memory range.

CXL Integration and Storage Options. Our storage device is a CXL Type 2 device. It exposes memory regions to the server through `cxl.mem`, and it caches memory from the host through `cxl.cache`, as depicted in Figure 3. We provide a simple API to map physical addresses from the device onto process memory. Once this mapping is done, applications access these memory ranges "as usual."

Internally, however, the device offers a powerful indirection mechanism. It associates a range of addresses with a kernel. *A kernel is a function that provides well-defined semantics for reads and writes.* A kernel that implements memory expansion semantics will simply redirect reads/writes to a selected memory type. We will provide such a kernel with the device that can opt between DRAM or NAND-Flash as backing memory. As Figure 3 shows, the device can still present itself to the system as an NVMe device and offer a traditional data path. Nothing prevents a legacy application from using it that way.

The device can also associate standard or application-specific kernels to a chosen memory range. We focus on kernels that are relevant for data-intensive applications that we denote as DATABASE KERNELS. These kernels can provide much more than 1:1 address mapping. They can host functionality that would otherwise be performed by the database system on the server host. DATABASE KERNELS can manipulate memory directly but can also rely on the help of local Direct Memory Access (DMA) engines, capable of efficiently transferring data in, out, and across the storage options, including fast memory, such as SRAM, in addition to DRAM and NAND-Flash.

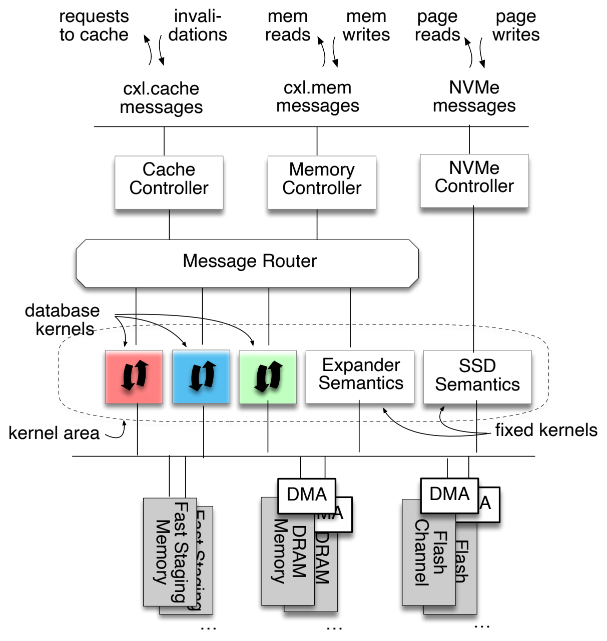

Figure 3: A Database Kernels supporting device. The device can be accessed as a conventional SSD or through CXL. In the latter case, the messages to a given memory address range will be directed to its assigned kernel. The kernel can choose which kind of storage type to use and how.

Note that a kernel does not need to map that range onto storage addresses directly. Instead, it can tie an address range to a *virtually materialized result of a given computation.* In other words, if we imagine that the device is holding a table, a kernel can expose only some of these table's rows, as if it had applied a predicate on behalf of the application. The kernel can also expose data that is not stored – if it knows how to calculate it from the data that is. We will discuss more examples in Section 5.

Why use a Type 2 device? So far, we have only discussed memory accesses in what could be perfectly accomplished by a Type 3 device. However, we propose developing Database Kernels on a Type 2 device. As explained above, this type of device can contribute memory to the system and also cache data from the system locally. To do so, it implements a Cache Controller that interacts with the system's Directory Controller via `cxl.cache`. Interestingly, Type 2 devices release control of their memory range to the Directory Controller (on the host), which forces the device to notify the Directory Controller if it wishes to cache its own memory. This is called *Host Bias* [7].

In our device, a kernel has the option to request shared cached lines on any portion of the address ranges it exposes. The benefit of this arrangement is subtle but powerful. If a core on the host wishes to access a cached memory address in exclusive mode—e.g., it wishes to write a new entry in an exposed area—the device can be notified of this intent through a cache invalidation message. Figure 4 illustrates this case This early notification of an intent to write gives the device much more time to prepare for the write than it would have if it learned about the write as it was requested.

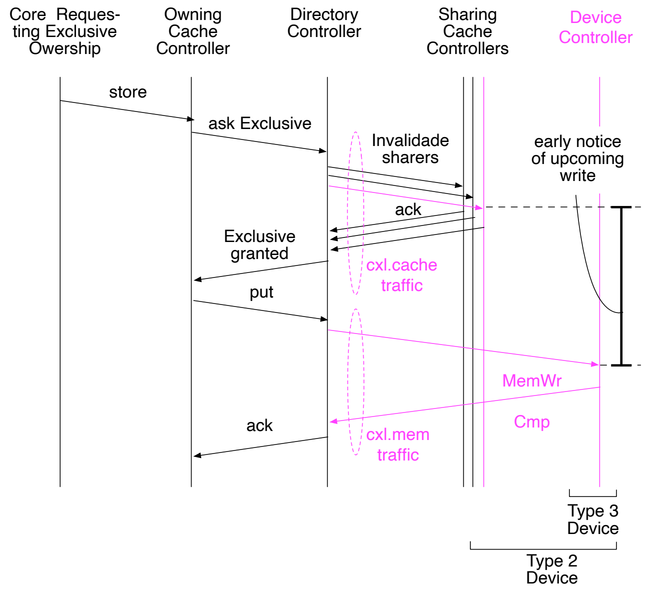

Figure 4: A Type 3 device would not learn about a write operation until the Directory Controller requested it. In contrast, if the same device could cache memory as well, i.e., if it were a Type 2 device, it would learn about the early intent to write. The reason is that, to give a core exclusive access to a memory address, the Directory Controller must invalidate all accesses given before. The invalidation is an early signal to the Flash-based Type 2 device that it should prepare to hear a write request for that address in the short future, giving it ample time to prepare.

4 DATABASE KERNEL TYPES

The discussion above about choosing a Type 2 device for its monitoring capabilities suggests that there are other CXL mechanisms than coherence that could be attractive for kernel development. We divide the kernels in categories according to the CXL features they use and discuss these categories next.

Classic Kernels. These are kernels that expose a memory address range backed by device's DRAM to the database system. There could be processes inside the device that issue `load` and `store` instructions against these addresses without differentiating whether an address sits on the host system or the device DRAM. Because these kernels rely on the coherence mechanisms of CXL, we call them *classic CXL kernels.*

In a classic kernel, the memory semantics and its backing implementation are the same as those of traditional memory. Ultimately, these kernels allow extending the database functionality into the device, simply by moving wholesale processes into it. We will discuss more specific examples in the following section, but generally speaking, any database process one would spin in a multi-socket machine—where remote access to memory occurs transparently—could technically be spun inside of the device as well. What defines these kernels is access to coherent memory.

Advanced Kernels. Database kernels, however, can leverage other aspects of the CXL protocol than coherence to build more *advanced kernels.* We comment on at least three features that enable new kernel functionality, summarized in Table 1.

The first feature pertains to how CXL is built upon an extremely low-latency messaging system. Arguably, the most important CXL component that supports this feature is the *Arbiter.* The Arbiter is a hardware component that sits on the lowest part of the CXL stack and decides which type of message—`cxl.io`, `cxl.mem`, or `cxl.cache`—will use the PCIe bus next. In practice, the Arbiter will prioritize cache coherence traffic, even, for instance, during heavy DMA operations in the PCIe bus. The very presence of the Arbiter is what differentiates PCIe Gen 5 and `cxl.io`. Another CXL aspect that makes it low latency is that its messages are small. They occur in *flits*, or 68 bytes in CXL 1.1 (although flits will grow in later CXL standard iterations) [30]. We have seen above how this combination of low latency and small messages enable the protocol to monitor the status of data regions at a cache line granularity and prepare for upcoming memory modification events before they have been concluded. Simply put, the coherence traffic can sometimes predict the memory traffic, and it does so with low latency. There could be kernels that use this prediction mechanism.

The second feature that can unlock advanced kernel possibilities is memory independence. The standard does not dictate the type of memory a device associates with exported address ranges. Servers supporting CXL on the market at the time of this writing invariably use DDR5 as the memory standard. However, some CXL-enabled devices that started appearing can offer DDR4 or even high bandwidth memory (HBM). Other CXL-enable devices are built with FPGAs, which carry some SRAM variations. Moreover, CXL devices can offer persistence by backing addresses with persistent memory or with NAND Flash memory. A DBK may use any type of memory available in the device or even a combination thereof, for instance, implementing some sort of memory layering inside the device with anti-caching semantics [8]. The kernel is free to implement a *contract* (semantics) that the application relies on when issuing reads and writes against that kernel memory. This contract may even vary across different addresses of the same kernel.

The third CXL feature that enables advanced Database Kernels is perhaps the most powerful. The standard neither dictates the kind of memory to back up addresses offered by a CXL device, nor does it mandate that there should be some memory backing the address! This allows an address to hold the results of a dynamic computation, performed only when the address is accessed. A typical usage of this kernel is data compression. Such a kernel would accept uncompressed writes to a memory region but persist them in a compressed way. In turn, reading from that region would decompress the necessary addresses only. Another interesting kernel using this feature is to allow a given data structure to be seen as column-oriented through one memory region but row-oriented through another region. Most importantly, if either gets updated, the kernel responsible for these two regions would invalidate the corresponding cached lines from both regions.

5 KERNEL EXAMPLES

The previous section discussed how certain CXL features, both centered on coherence and not, can unlock several useful Database Kernels. In this section, we provide a functional description of a few of them.

Buffer Manager Extensions. Perhaps the most intuitive kernel to add to a system is one that expands the amount of memory it can use. Technically, there is no need for a Database Kernel to achieve this. Any type 3 device can be attached to the system whose memory could be promptly used by the system's Buffer Manager simply through a larger buffer frame.

Flushing buffers in this arrangement, however, could be less than efficient. If the device offering the memory has storage capabilities, the database system may not have the means to transfer data from the expanded memory into the persistent area easily. It would, most likely, treat the CXL device as two, the volatile and the persistent storage device areas. It would stage data from the expanded memory and send it to a persistent area—when this data movement could very well be performed intra-device. The goal of a Buffer Manager Extension kernel is to enable such optimizations. In other words, the kernel would implement a Flush operation to move data from DRAM into Flash.

Note that this kernel may allow for exciting data placement possibilities. The system may know upfront that certain pages are being retrieved that may be updated, e.g., as part of a SQL `UPDATE` command, while others would not, e.g., in a SQL query. Pages not already loaded in the system may be allocated accordingly, with likely writable pages being allocated from the device memory pool. The benefit of such a data placement scheme is to reduce I/O bandwidth.

Query Execution Worker. The idea of pushing predicates down is as old as query optimization. Naturally, this type of optimization was one of the first to be attempted in-storage [16]. With kernels, we can support not only this type of scan and filter operation but also an extensive array of access paths, as the query operations that interact with the base tables/indexes storage are called. For instance, a kernel could implement an indexed lookup access. It would entail an index tree traversal and a base table page read. The kernel could implement the tree traversal in the device benefiting from cached data, saving index data transfers between the host and the device.

As with other typical query workers, the operators this worker would be executing would communicate with each other via pages pinned in the Buffer Manager. These pages may or may not reside inside the device. A traditional query worker on the system processes the rest. The coherence feature of CXL allows either type of worker to access a common set of pages.

A Fast Transaction Logging Kernel. In the two previous kernels, we took advantage of the memory coherence feature of CXL to ship some of the database functionality to the device. In this kernel, we mostly rely on CXL's fast communication messaging system in a situation that does not strictly require coherence. Transaction logging typically entails a sensitive operation in a database system, as every transaction has to be reflected in a local (and persistent) log file. Quite often, the speed with which the transaction is persisted

CXL Features	Potential Kernel Functionality	Kernel Examples
Coherence	Allows different processes to access memory coherently.	Buffer manager extensions, query execution worker, etc.
Flit-based messaging	Support low-latency communication for coherence traffic.	Fast transaction logging, etc.
Backing-memory freedom	Allows mixing memory types within the same memory region.	Data placement for LSM compression, etc.
Non-backed addressing	Enables view materialization mechanisms.	Compression/decompression, data transposition, coherent views, etc.

Table 1: CXL features and the kernel functionality they enable.

is the main bottleneck in a database. Transaction logging is an ideal operation to offload to a DBK, in the sense that it can use a type of memory with low latency and with a battery-backed option for persistence. A dedicated database kernel exposing an append-only abstraction on top of CXL could streamline this set of operations in various ways. For instance, the kernel can stage log entries on SRAM and quickly destage them asynchronously to Flash. Comparable semantics can be obtained using a dedicated storage device (such as our own X-SSD system [17]), but CXL and kernels bring many advantages in this context, including the upfront notice of an intent to write (cf. Section 3).

Data Placement Kernel. In column stores, data is traditionally stored in two very different memory regions: a write-optimized store that first receives all updates, and a compressed, read-optimized store [32]. In HTAP systems, the areas that store different data representations can be even more disparate [26]. Moreover, in LSM-tree-based key-value stores, the writing activity in the upper layers of the tree is more frequent than in the lower ones, and these interfere with compaction work [3]. As discussed in the previous section, the address ranges exported by a type 3 device may be backed by more than one memory type. Areas with more intensive write activity could be initially persisted in SLC Flash memories, for instance. Areas with more read activity could be moved to MLC, TLC, or QLC Flash memory, which sacrifice write performance in the name of more economical reads. Currently, this fine-grained data placement is not visible to an SSD user, even if it exists in some devices [34], but with Database Kernels, it can. A kernel can place different portions of their memory range under distinct Service Level Agreement, so to speak, just by assigning each portion to a different type of backing memory.

Coherent Virtual Views. This is perhaps the most powerful type of Database Kernel. It can equate memory accesses with performing computations. The idea with this kernel is that it exports a memory address range but does not back it up with memory. It associates a computation with the region and performs the computation as a side-effect of accessing that region. One example would be to use this functionality to compress data when writing and decompressing when reading, both transparently. As the name implies, we say the kernel implements a view over the data.

The views may explore coherence in a unique way. Suppose a view exports a column-oriented representation of an area that is, in fact, stored in a row-oriented format. Each virtual view be accessible through a different memory range. Updating one of the memory regions implies updating the other as well. If, however, either area is being cached anywhere in the system, these cache lines need to be invalidated. The kernel can control how the areas of the different regions are associated. When one area gets updated, besides asking for exclusive access over that data structure, the kernel asks for the same access over the virtual structure. This technique was pioneered by CCKit [27] and PLayer [5].

6 PRELIMINARY VIABILITY ANALYSIS

We seek a platform that supports CXL to develop a Database Kernels storage device. This platform inevitably needs specialized hardware because PCIe and CXL protocol messages require low latency. Moreover, memory management requires specialized hardware for the same reason. For mostly everything else, there are alternatives that can use the software in efficient ways. This combination of hardware and software makes FPGAs a particularly promising platform. We already have access to a platform that fits this profile.

One of the most challenging aspects of developing using FPGAs is predicting the necessary area (how many logic units in the FPGA fabric) a given design will have. Therefore, our first experiments were to prototype a hardware design that connects the PCIe/CXL areas of the card with the memory controller areas. The rationale behind this design is that it can approximate the *data path* we wish the card to support. The data path is certainly one of the components that will consume the larger area in our design. Table 2 shows the results of this experiment for a Type 2 and a Type 3 design. As expected, the Type 2 design uses more area since it implements a Cache Controller (cf. Figure 2).

Case	IP(s)	Logic Unit Counts			% of Total
		Ideal	Real	Total	Logic Units
1	CXL Type 2	179K	213K	251K	27.5%
	2 DDRs + User Logic	31K	38K		
2	CXL Type 3	141K	167K	204K	22.5%
	2 DDRs + User Logic	30K	37K		

Table 2: The table shows how many logic units a primary type 2 or 3 device design uses. Each design includes CXL IP, two channels of DDR 4, and user logic.

Fortunately, the FPGA is comfortably sized; our data path occupies only slightly more than 1/4th of the available area, leaving ample space for the other components.

Area, however, is not the only concern. The data path should provide adequate bandwidth to be effective. In practice, the maximum bandwidth of a PCIe card is capped at the width of the PCIe connection. In our case, this is 64GB/s (a Gen5 x16 connection). This bandwidth can roughly support 2 DDR4 memory controllers, but it is a somewhat high bandwidth for an FPGA card. For comparison, this bandwidth is equivalent to five 100Gbps network ports. Therefore, we analyze whether a preliminary design could achieve such bandwidth. Figure 5 shows the result of this experiment. The data path in that figure connects (1) the PCIe/CXL controller with (2) two memory controllers.

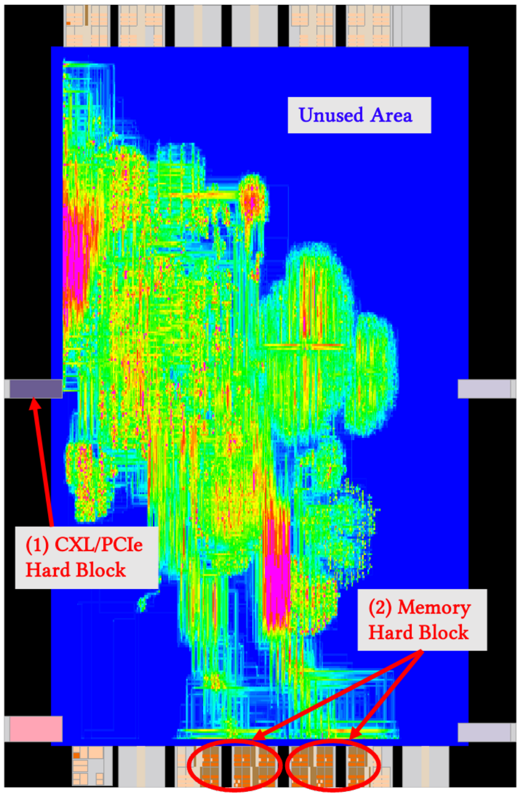

Figure 5: Floorplan of the FPGA fabric. The figure shows the FPGA fabric in the center (large blue rectangle) and the specialized hard blocks in the periphery. The PCIe/CXL block is located at the left-center of the FPGA fabric (1). The two DDR4 channels are located at the bottom-center (2). The circuit placement in the FPGA connecting these areas is shown in a green-magenta color range. That range represents the density of the circuit. In particular, the magenta areas are close to saturating the resources in that area.

The FPGA floorplan shows that the data path is viable but there are congested areas within our data path. In essence, this may mean that the FPGA synthesizer may take longer to compile circuit definitions, trying to place and route them on the FPGA. We believe the drawbacks are minor and that we have a suitable platform to develop a DATABASE KERNELS-supporting device.

7 RESEARCH AGENDA

We believe DATABASE KERNELS is a foundation for future database systems incorporating in-storage processing capabilities. The work presented here is but the start of several fundamental research directions that need to be explored, including:

Following CXL evolution. Our current proposal is based on CXL 1.1, which is the version that is about to become commercially available. However, the future versions of the CXL protocol are already specified. The additional features specified for versions 2.0 and 3.0 can unlock further possibilities for DBK. Of particular interest is the possibility of integrating a host with remote storage, which CXL 3.0 and beyond will allow.

Hardware Support for Application Logic. We would like DATABASE KERNELS to be an inviting and performant environment for software development. We can achieve so with a combination of user and pre-installed functions [11]. Examples of pre-installed functions are sorting, merging, filtering, transposition, etc. Some functions could even deal with serialization and deserialization of traditional file formats. Since these internal functions are stable and generic, they may be implemented in hardware. The user functions, in contrast, could be developed in a software environment and a general-purpose language. To support the latter, the device should dedicate one or more cores to run application software. We discuss how to develop in this environment next.

A Database Kernel Development Kit (DSK). Admittedly, the development of kernels in our current proposal requires skills that are only available to SSD and FPGA design specialists. The integration with Flash is, for now, too low level, and, for performance, some of it must be implemented via hardware. However, there is no fundamental impediment to shifting the development techniques towards a more software-centric approach. This may require implementing clearer software interfaces and wrapping hardware aspects of this integration in something akin to function calls.

Additional Memory Technologies. The current DATABASE KERNELS proposal makes SRAM, DRAM, and NAND-Flash memories available for kernel development. In the future, it should be possible to incorporate other types of memory such as HBM—and, perhaps future formats of persistent memory that replace Optane—should they become more commonly available on development platforms.

Safety and Security Aspects. With many kernels running in a device, crash safety and security issues may arise. The kernels should not interfere with one another, and despite being integrated into the device, they should be isolated in a way that does not corrupt or otherwise hamper the proper functioning of the device.

Fostering Interoperability. One factor that could significantly improve adoption is if different vendors supported DATABASE KERNELS and competed by offering different cost vs. performance tradeoffs. Similarly, it would be interesting if a database system boot process could check whether the storage over which it is running provides DATABASE KERNELS and adjust accordingly. With such interoperability features, it should be possible for something akin to a marketplace of DBK to emerge.

8 RELATED WORK

To the best of our knowledge, this is the first work that leverages CXL to provide in-storage database functionality. Storage, however, has been historically gaining processing capabilities [4, 11, 19].

Leveraging that potential for query processing was described by Do et al. [10]. More recently, Lerner & Bonnet characterized the architectural alternatives to do so [18]. There are many examples of functionalities that were pushed into storage: joins and filters [6, 13, 16], transaction log acceleration [17], LSM compaction [22], and device profiling [20], to cite a few. These are excellent kernel candidates, and if implemented so, they will benefit from the uniform and transparent interface that CXL can offer.

Some works started to speculate about how to use CXL memory expanders to integrate a host's memory either with storage [14, 28] or directly with a database system [1]—but never both, as Database Kernels do. Abishek et al. have discussed mechanisms to maintain cache coherence across virtually materialized views [27]. The mechanism is very powerful, and Database Kernels can take full advantage of it within a storage device.

9 CONCLUSION

In this paper, we introduced Database Kernels, the first platform to embed database functionalities deep into storage devices using coherence technology. The cornerstone of this database-device integration is the use of CXL, and in particular its caching capabilities. The device uses coherence traffic to monitor requests, prepare ahead of time, and ultimately answer database queries more efficiently. While realizing the full potential of DBKs will take years, we are already excited about the new database architectural possibilities that this new technology opens.

ACKNOWLEDGMENTS

This work has received funding from the Swiss State Secretariat for Education (SERI) in the context of the SmartEdge EU project (Grant agreement No. 101092908).

REFERENCES

[1] Minseon Ahn et al. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *DaMoN*. https://doi.org/10.1145/3533737.3535090

[2] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro* 34, 4 (2014), 36–42.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX*. https://www.usenix.org/conference/atc19/presentation/balmau

[4] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *CIDR*. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper29.pdf

[5] Richard Braun, Abishek Ramdas, Michal Friedman, and Gustavo Alonso. 2023. PLayer: Expanding Coherence Protocol Stack with a Persistence Layer. In *DIMES Workshop*. https://doi.org/10.1145/3609308.3625270

[6] Wei Cao et al. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in {Cloud-Native} Relational Database. In *FAST*. https://www.usenix.org/system/files/fast20-cao_wei.pdf

[7] CXL Consortium. 2023. Compute Express Link Specification. https://www.computeexpresslink.org/download-the-specification.

[8] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. 6, 14 (2013), 1942–1953. https://doi.org/10.14778/2556549.2556575

[9] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring. In *SYSTOR*. https://doi.org/10.1145/3534056.3534945

[10] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *SIGMOD*. https://doi.org/10.1145/2463676.2465295

[11] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. Delilah: EBPF-Offload on Computational Storage. In *DaMoN*. https://doi.org/10.1145/3592980.3595319

[12] Intel. 2023. Sapphire Rapids Family. https://ark.intel.com/content/www/us/en/ark/products/codename/126212/products-formerly-sapphire-rapids.html.

[13] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. In *VLDB*. https://doi.org/10.14778/2994509.2994512

[14] Myoungsoo Jung. 2022. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). In *HotStorage*. https://doi.org/10.1145/3538643.3539745

[15] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian P. Kroth. 2019. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. (2019). https://doi.org/10.14778/3372716.3372724

[16] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. In *Information Sciences*. https://doi.org/10.1016/j.ins.2015.07.056

[17] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *SIGMOD*. https://doi.org/10.1145/3514221.3526188

[18] Alberto Lerner and Philippe Bonnet. 2021. Not Your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD*. https://doi.org/10.1145/3448016.3457540

[19] Alberto Lerner, Rana Hussein, André Ryser, Sangjin Lee, and Philippe Cudré-Mauroux. 2020. Networking and Storage: The Next Computing Elements in Exascale Systems?. In *IEEE Data Engineering Bulletin*. https://exascale.info/assets/pdf/lerner20debull.pdf

[20] Alberto Lerner, Jaewook Kwak, Sangjin Lee, Kibin Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2020. It Takes Two: Instrumenting the Interaction between In-Memory Databases and Solid-State Drives. In *CIDR*. https://www.cidrdb.org/cidr2020/papers/p19-lerner-cidr20.pdf

[21] Huaicheng Li et al. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ASPLOS*. https://doi.org/10.1145/3575693.3578835

[22] Minje Lim, Jeeyoon Jung, and Dongkun Shin. 2021. LSM-tree Compaction Acceleration Using In-storage Processing. https://doi.org/10.1109/ICCE-Asia53811.2021.9641965

[23] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *ASPLOS*. https://doi.org/10.1145/3582016.3582063

[24] Micron. 2023. CZ120 memory expansion module. https://www.micron.com/solutions/server/cxl.

[25] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs*. https://doi.org/10.1007/978-3-642-03359-9_27

[26] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *SIGMOD*. 1771–1775. https://doi.org/10.1145/3035918.3054784

[27] Abishek Ramdas. 2023. *CCKit: FPGA acceleration in symmetric coherent heterogeneous platforms*. Ph. D. Dissertation. ETH Zurich.

[28] Samsung. 2023. Memory Semantics SSD. https://samsungmsl.com/ms-ssd/.

[29] Samsung. 2023. Samsung Develops Industry's First CXL DRAM Supporting CXL 2.0. https://semiconductor.samsung.com/news-events/news/samsung-develops-industrys-first-cxl-dram-supporting-cxl-2-0/.

[30] Debendra Das Sharma, Robert Blankenship, and Daniel S Berger. 2023. An Introduction to the Compute Express Link (CXL) Interconnect. *arXiv preprint arXiv:2306.11227* (2023).

[31] Daniel Sorin, Mark Hill, and David Wood. 2011. *A primer on memory consistency and cache coherence*. Morgan & Claypool Publishers. https://doi.org/10.1007/978-3-031-01764-3

[32] Michael Stonebraker et al. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. https://doi.org/10.5555/1083592.1083658

[33] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *arXiv*. https://arxiv.org/abs/2303.15375.

[34] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. 2016. Reducing Solid-State Storage Device Write Stress through Opportunistic In-place Delta Compression. In *FAST*. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-xuebin