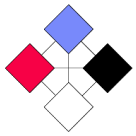
D6.4 Final prototype of managed storage tiers including automatic placement



DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.0

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

Report and final prototype of the end-to-end computation storage components. This deliverable describes prototypes based on both on-path and off-path architectures that integrate hardware acceleration as a form of near-data processing. We also compare these prototypes to the new NVMe standard on computational storage, we describe how to program computational storage and more generally discuss the lessons learnt, in DAPHNE, about computational storage.

D6.4 Final prototype of managed storage tiers including automatic placement			
WP6 – Computational Storage			
Type of document	D	Version	1.0
Dissemination level	PU	Project month	48
Lead partner	ITU		
Author(s)	Philippe Bonnet, Niclas Hedam, Alex Krause, Morten Tychsen, Piotr Ratuszniak.		
Reviewer(s)	Mark Dokter, Matthias Pohl		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Structure of the document	Philippe Bonnet
V0.2	First Complete Draft	Philippe Bonnet, Piotr Ratuszniak, Niclas Hedam, Alex Krause, Morten Tychsen.
V1.0	Feedback from reviewers incorporated	Philippe Bonnet

1. Introduction

At its core, computational storage is about increasing parallelism of data processing, by identifying functions which can be offloaded from the host's central porcessing unit (CPU) to a processing unit associated to a storage device. This frees up the CPU on the host to perform other actions.

We consider that data is staged on local solid-state drives in a HPC system equipped with computational storage processor. The data is staged, from an object server or from an archival storage system. In the context of DAPHNE, this placement of data from an archival tier to the processing tier is orthogonal to the execution of a data pipeline.

A computational storage processor is equipped with a computational storage engine that executes functions that are either statically installed or that are downloaded from the host. These functions operate on data that is located in the local memory. Portions of the local memory are shared with the host, with solid-state drives (SSDs) and possibly other devices. We have described computational storage in the following book: A.Lerner and Ph.Bonnet, The Principles of Databases and Solid-State Drives Co-Design, to be published by Springer in 2025 [1]. This book is a significant part of the dissemination effort stemming from WP6. Figure 1 below is taken from this book and illustrates the architecture of a computational storage processor.

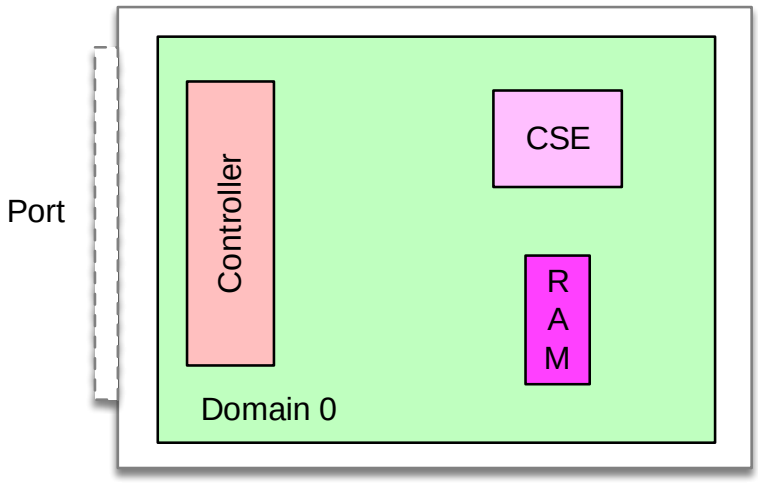


Figure 1: A computational storage processor is composed of (i) a controller that communicates with the rest of the system, (ii) a computational storage engine (CSE) that performs computation and (iii) local memory (RAM) (Figure 4.4 in [1]).

Figure 2 below further illustrates how a computational storage processor is integrated in a complete system with a host and multiple solid-state drives. We distinguish between an on-path architecture, where the computational storage processor is placed in between SSD and host, and an on-path architecture where the computational storage processor is placed next to the solid-state drives and behind a PCIe switch, so that these devices can interact with each other without involving the host.

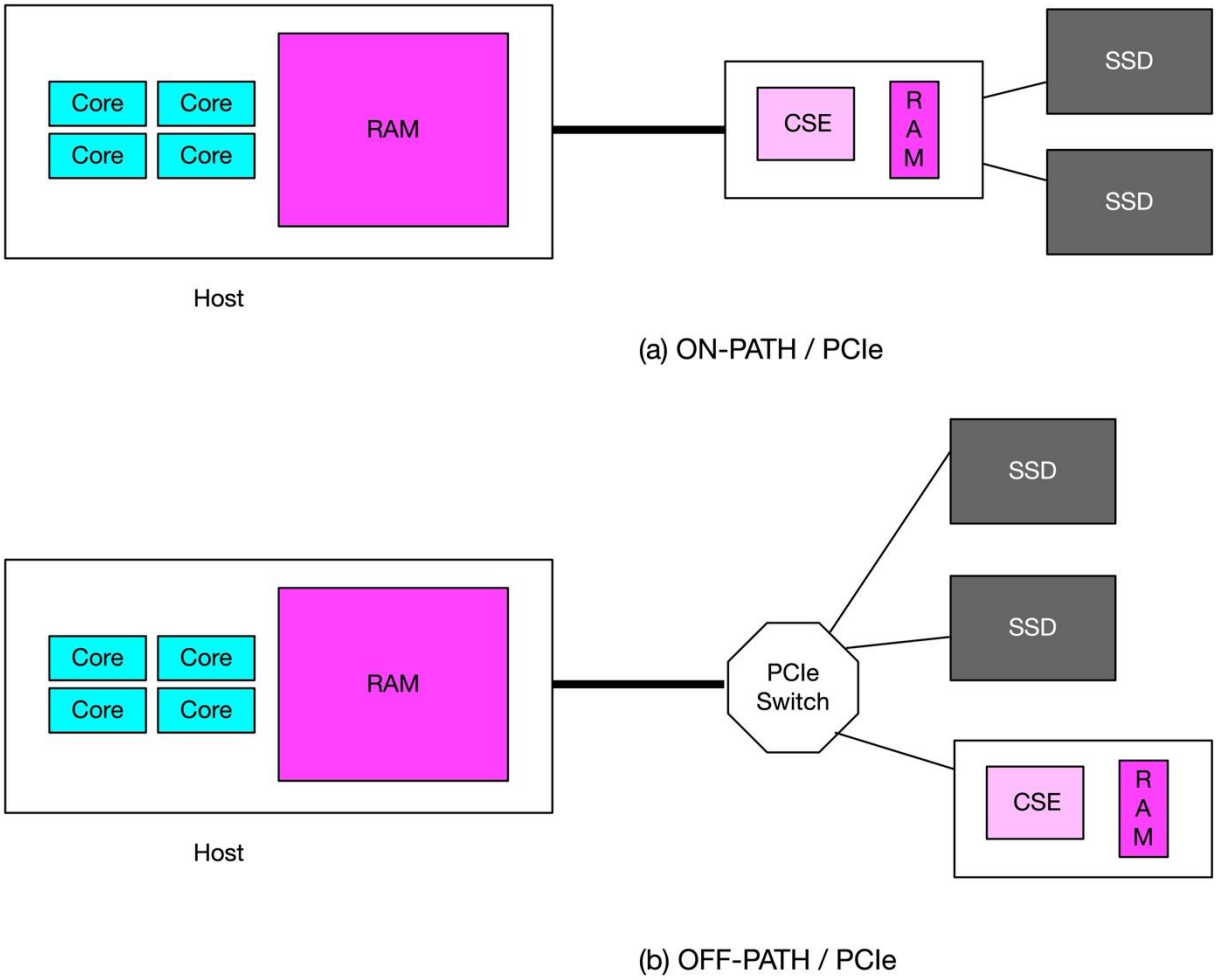


Figure 2: Computational storage architectures with a computational storage device located between the host and the solid-state drives (on-path) and next to the solid-state drives behind a PCIe switch (off-path) (Figure 4.3 in [1]).

In previous deliverables, we presented the design of Delilah, a prototype that supports the offload of eBPF functions from the host to an on-path computational storage processor. As a result, it can be used to define a novel storage interface, but it prevents the host from accessing directly the data that is stored on the SSDs. In the Section 2, we focus on the latest version of the Delilah prototype, where a hardware-accelerated function is part of the computational storage engine. With Delilah, the data path for a data processing pipeline starts with data stored on the SSD, first processed through the computational storage processor and further processed on the host.

In Section 3, we present a new prototype that implements an off-path computational storage architecture. This prototype supports a data path, where data is processed as it is transferred from a solid-state drive to the memory of a computational storage processor, where it can be accessed from the host or from other devices.

Finally, in Section 4, we reflect on the lessons we learnt about computational storage with a focus on an analysis of the NVMe standard, a discussion of how to program computational storage and integrate it with a host system.

2. On-Path Hardware-Acceleration

The Delilah Computational Storage Processor has been described in previous deliverables. Figure 3 presents the architecture of Delilah (in green) and the components it relies on (in grey), as presented in [2] and in previous deliverables.

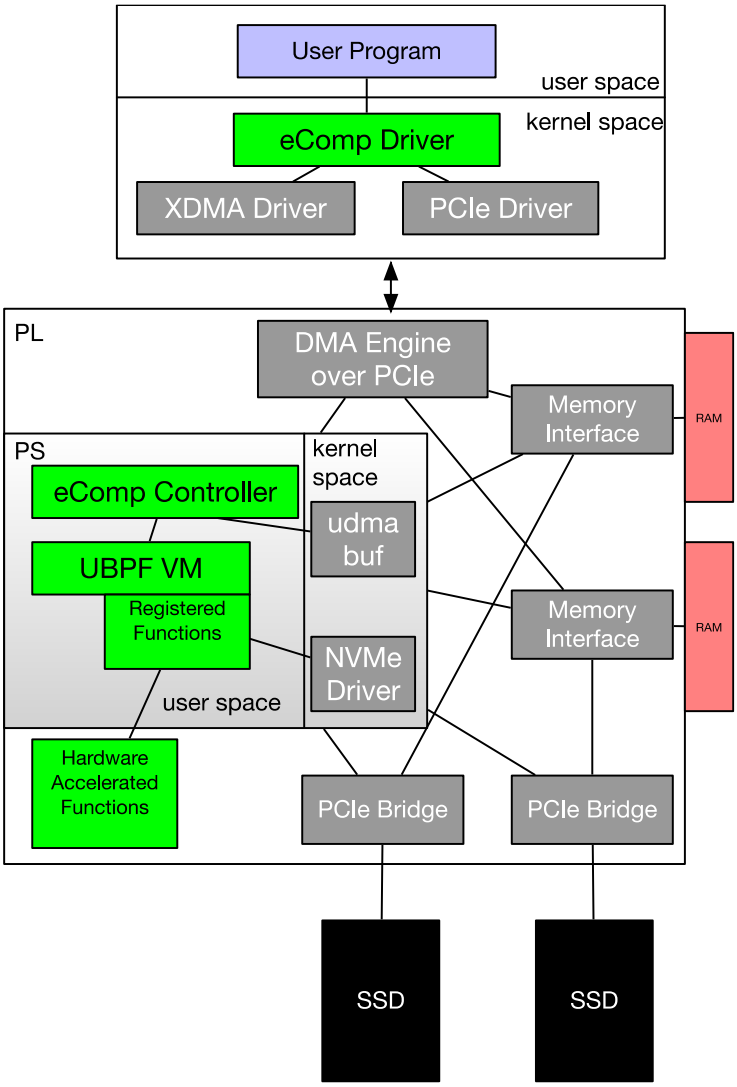


Figure 3: Delilah Architecture. The computational storage engine (in green) is composed of a host driver, a controller and a virtual machine/JIT compiler for eBPF programs as well as registered functions located in embedded software running on ARM cores on the device (PS), with hardware accelerated functions implemented in FPGA on the device (PL).

At its core, Delilah implements a host-controller transport protocol through a driver module on the host kernel and a controller on the device. Delilah relies on uBPF as an eBPF run-time environment. It makes it possible for the host to offload eBPF programs to the device, with their arguments, execute them and retrieve the results. The host driver relies on DMA to write an eBPF program on a program slot in the device memory, to write program arguments and read results from a data slot, also in the device memory. Data that is read from the SSD can be placed in a data slot or in a shared data buffer (that is available to the functions running on the

device, but not to the host). The different memory regions used by Delilah are illustrated in Figure 4.

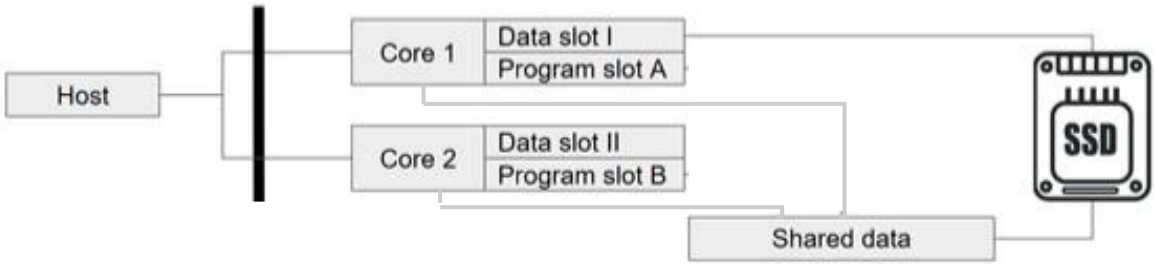


Figure 4: Program slot, data slot and shared data buffer are portions of the local memory of the Delilah computational storage processor.

Program and data slots are shared memory regions between the host and the device. The host can read and write to them via DMA. These memory regions are defined as DMA buffers in XDMA (the DMA library that we use in Delilah). We rely on user-space mappable DMA buffers to access them from functions runnning on the device.

Figure 3 refers to two distinct areas on the device: PL denotes a FPGA and PS denotes general-purpose cores. Delilah is implemented on a Zynq Ultrascale+ MPSoC that combines hardware acceleration on FPGA (PL) with the flexibility of ARM cores running embedded Linux (PS).

The Delilah controller accesses peripherals (PCIe connection to the host and to M.2 SSDs and RAM DIMMS) through the PL. This requires a block design that defines a DMA engine over PCIe (for communication with the host), memory interfaces (for accessing RAM) and PCIe bridges (for communicating with the SSDs). These components are shown on Figure 5.

Figure 5 illustrates two block designs using Xilinx IP modules (white boxes) connected with each other, external pins and the Zynq PS (grey box) with AXI point-to-point interconnects. The red design relies on external RAM DIMMs. It is well-suited for high throughput workloads. The green design relies on internal RAM within the Zynq PS. It is well-suited for low latency workloads. The red block design can be used to conduct a simple bottleneck analysis. The width of the AXI interconnect and the clock domain of the IP components involved in the red block design are indicated on the figure. Let us consider that data is read from both SSDs, placed in RAM, read and updated from the PS and then transferred to the host. Data can be read from each SSD at 4GB/s. Memory bandwidth is 12.8 GB/s for each RAM DIMM. The PS can read and write data at 4.8 GB/s from each RAM DIMM. The PCIe connection to the host can transfer data at 16 GB/s. In this example, memory is the bottleneck as it is used for writing data from the SSDs, reading and writing from the PS and reading to the host. This pattern imposes an end-to-end throughput limit within the block design of 12.8 / 4 = 3.2 GB/s.

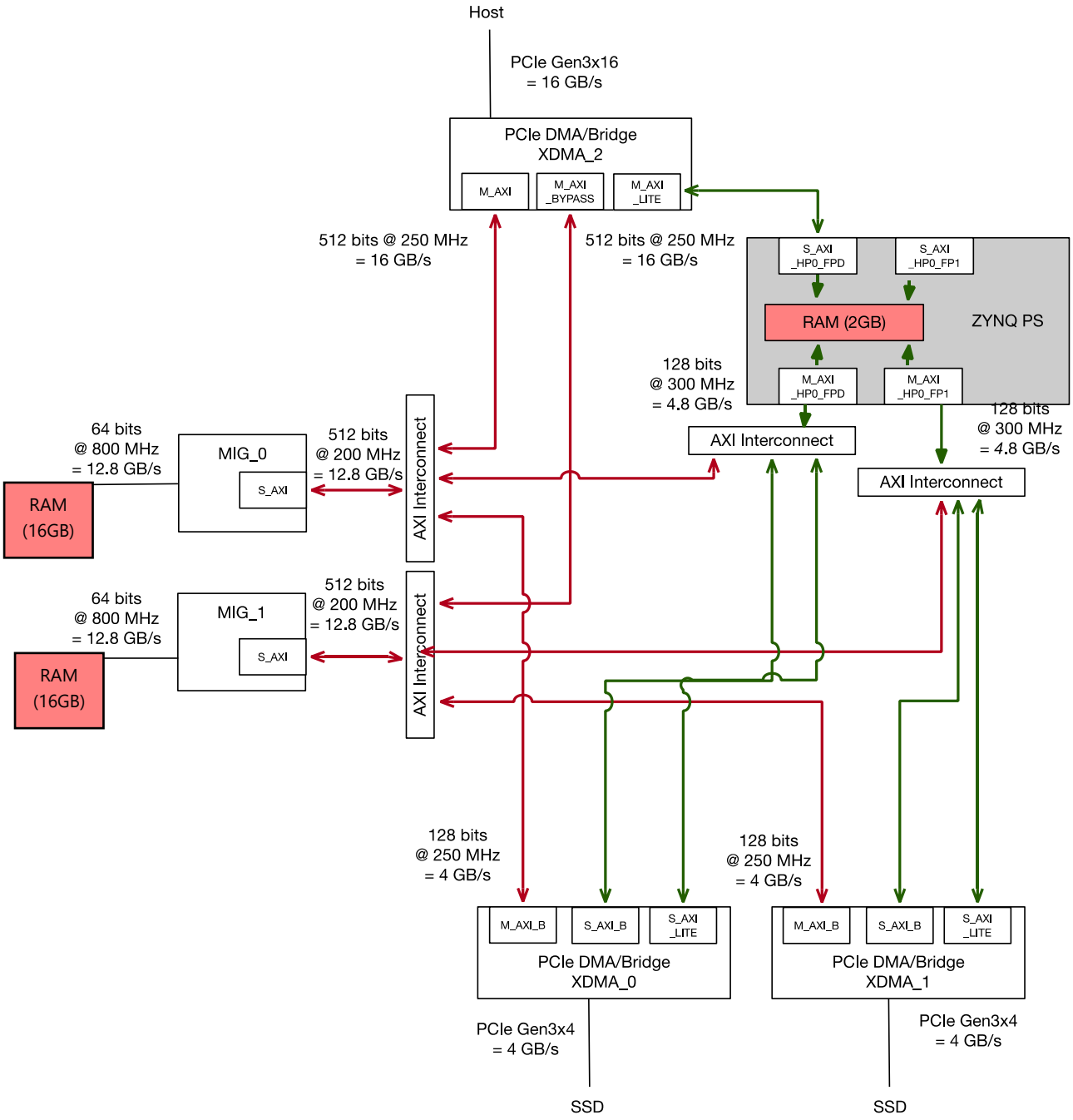


Figure 5: Delilah block design for high throughput (red) and low latency (green).

To further chracterize the performance of the different memory components, we designed a function that executes 32 million memory loads and stores with (i) reads and writes, (ii) sequential and random accesses, (iii) with and without 16-byte and 4-kilobyte alignment. The memory components we consider are:

- Internal PS memory (placed in the ARM processor), both statically allocated in kernel space on embedded Linux using UDMA (UDMA allocates contiguous memory blocks within kernelspace via the device tree), or dynamically allocated in user space with malloc.

- External PL memory (external DRAM DIMM modules) accessed via a Memory Interface Generator (MIG) block in FPGA.

The experiment yields an amortised access latency, measured in nanoseconds. The results are shown in Figure 6. As expected sequental accesses are much faster than random accesses. Our experiment shows that there is a 2 orders of magnitude difference. Also, the choice of an underlying RAM module for sequential reads and writes appears inconsequential, as the ARM cores caches hide significant performance differences. However, random accesses are much faster from PS memory. The key insight here is that processor caching and workload locality are crucial to obtain good memory performance when offloading functions on the Delilah computational storage processor.

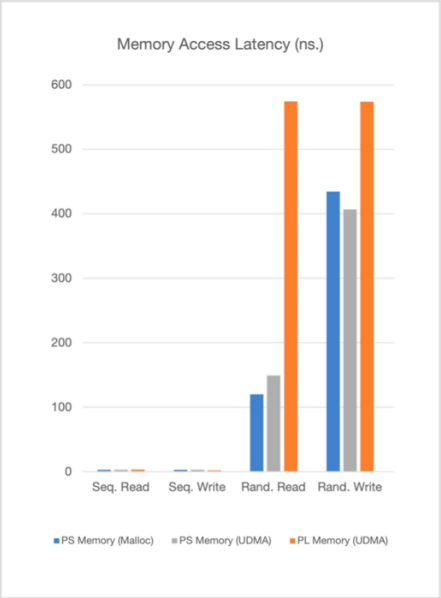


Figure 6: Average latency of a single memory access when running 32 million accesses in a row. The results are shown in linear scaling. Seq denotes sequential, rand denotes random.

Further experimental results are contained in Niclas Hedam's PhD thesis [3], that he succesfully defended on October 25th 2024.

In addition to the generic Xilinx IP showed (in white) in Figure 5, we developed a custom IP as a hardware accelerated filter function. Our filtering accelerator is implemented in C with High-Level Synthesis (HLS) pragmas. It supports five modes: equality, inequality, less than or equal, greater than or equal, and between-inclusive.

```
typedef uint32_t filter_t;

uint32_t filter(filter_t *in, filter_t *out, uint32_t num,
                uint8_t op, filter_t comp1, filter_t comp2) {
#pragma HLS INTERFACE m_axi port=in offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=num bundle=control
#pragma HLS INTERFACE s_axilite port=op bundle=control
#pragma HLS INTERFACE s_axilite port=comp1 bundle=control
#pragma HLS INTERFACE s_axilite port=comp2 bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  ...

}
```

Figure 7: Signature of our hardware accelerated filtering operator.

Figure 7 shows the signature of our hardware-accelerated filter operator. The function accepts two pointers (one for input and one for output), an opcode to select the mode, and two comparison values. The #pragma directives specify the use of two AXI ports: a high-throughput AXI port for data access (gmem) and a low-latency AXI Lite port for reading and writing function parameters (control).

```
uint8_t k = 0;  // Vector result counter
uint32_t c = 0, // Overall result counter
    i,          // Vector iterator
    j;          // Value iterator

filter_t in_buf[BUF_SIZE];
filter_t out_buf[BUF_SIZE];

switch (op) {
  case DELILAH_FILTER_EQ:
  Vector_Loop_EQ:
    for (i = 0; i < num; i++) {
      memcpy(in_buf, &in[i * BUF_SIZE], BUF_SIZE * sizeof(filter_t)
    );
    Value_Loop_EQ:
      for (j = 0; j < BUF_SIZE; j++) {
#pragma HLS UNROLL factor = 1
#pragma HLS PIPELINE
        if (in_buf[j] == comp1) {
          out_buf[k] = BUF_SIZE * i + j;
          k += 1;
        }
      }
      memcpy(out + c, out_buf, k * sizeof(filter_t));
      c += k;
      k = 0;
    }
    break;

    ...
}
```

Figure 8: Implementation of the hardware accelerated filtering operator (equality mode).

The operator utilises vectorisation, processing sets of 256 elements at once. This approach avoids individual sequential memory requests for 32-bit elements, and instead issues 1-kilobyte memory reads and writes. The vector is stored in local BRAM within the accelerator.

During execution, the operator first iterates over the number of vectors. Each vector iterates over the elements in a pipelined manner, where it begins to compare the next element before the current one finishes. The vector results counter increments on finding a match, and the index is written to the output buffer. The output buffer is then written back to the output pointer for each vector.

The operator is exposed as a registered function that can be called from eBPF functions. The hardware accelerated filter operator IP generated with the code presented above is exposed as a register function denoted *delilah_hw_filter*. This function transforms arguments to match the input format of the IP. For example, pointers provided by the eBPF program are always relative to the execution context, i.e., the logical addressing space of the data slot or the shared data slot. It is necessary to map and transform these addresses from logical to physical addresses. Also, the registered function is coupled with Xilinx-generated drivers. The Vivado HLS platform automatically generates C drivers that write to the correct offsets in the HLS registers. The registered function invokes the Xilinx driver to write any appropriate registers to set parameters, start execution, and fetch return values.

Initially, our experiments showed a marginal difference in performance between hardware acceleration using HLS and the eBPF baseline. With both experiments concurrently executing four operators, HLS shows a slightly faster runtime of 5.72 seconds, compared to 5.85 seconds for the eBPF baseline.

To better understand the limited improvement in performance, we modified the experiment by introducing serial execution for the filtering process. In the modified version, the four operators are run serially, with only one active function at any time. Under these conditions, we observe a hardware-accelerated runtime of 6.46 seconds versus 7.29 seconds for the eBPF baseline. This shift illustrates the point we made above: memory bandwidth is the scare resource on the Delilah computational storage processor. While hardware acceleration does offer performance improvements, its impact is reduced when the underlying data path to memory becomes saturated.

3. Off-Path Hardware-Acceleration

The Intel team incorporated its quantization IP in the context of an off-path computational storage architecture. The experiments were performed with an Intel Agilex® 7 FPGA I-Series FPGA Development Kit (DK-DEV-AGI027RES) (see Figure 8).

Figure 8: Agilex® 7 FPGA I-Series FPGA Development Kit

The experiments connected the device shown in Figure 8, as a computational storage processor in an off-path architecture together with two Intel® Optane™ P4800X SSDs to root complex (see Figure 9).

Figure 9: FPGA and SSD devices connected on the same server motherboard.

Here, the entire computational storage engine is implemented within a FPGA, which is associated with external DDR memory on its board. In fact, we consider a minimal engine that (i) moves data from SSD to local memory and (ii) processes data as it is being moved using the quantization IP placed between the FPGA PCIe IP and the DDR management component. The quantization IP architecture implements range-based linear quantization algorithm[1] using additional scale factor and offset parameters (see Figure 10) and converts data format from 32 bits floating point to 8 bits unsigned integer.

The p2pmem-test program[2] moves a data chank from first SSD device to the second one through p2pmem device, through the FPGA with connected DDR modules exactly. When the data chunk is read from computational storage engine memory, each data chunk is processed through the quantization IP composed of 16 pipelines. Each pipeline processes one input 32-bit floating point value for each clock cycle at each pipeline stage. Finally, 16*32-bit floating-point values are transformed into 16* 8 bits integer concatenated with 24-bit zero value on 512-bit interface.

Per-to-peer memory (p2pmem) is used for the direct communication between two PCIe devices, between the two SSDs in described scenario. FPGA device exposes to the system 16MB p2pmem memory on PCIe Base Address Register(BAR0). To make the memory visible as a peer-to-peer memory required linux kernel driver (dpdr.ko) has been implemented according to peer-to-peer DMA support requirements[3] for a peer-to-peer memory Provider. Additional linux user space driver is required() to enable/disable quantization and for used offset and scale factor definition. The code of the prototype(linux kernel/user space drivers and FPGA project) is available at the following URL: XXX

The host is responsible for setting up the peer-to-peer DMA transfer, after that computational storage engine and SSD communicate directly through p2pmem.

[1] https://intellabs.github.io/distiller/algo_quantization.html
[2] https://github.com/sbates130272/p2pmem-test
[3] https://www.kernel.org/doc/html/latest/driver-api/pci/p2pdma.html

For comparison, the quantization IP was validated in another project with FPGA as a stand-alone computing accelerator with data moved from host memory to device memory and back. Here, the data is directly obtained from a SSD (and the result of the quantization can be written to SSD) without involving the host.

Here is a terminal output showing the difference between native peer-to-peer DMA (without quantization)

```
[pratuszn@localhost p2pmem-test]$ sudo ./p2pmem-test /dev/nvme0n1 /dev/nvme0n1 /sys/devices/pci0000:80/0000:80:02.0/0000:83:00.0/p2pmem/allocate -c 1 -s 4k --check -o 0

Running p2pmem-test: reading /dev/nvme0n1 (1.5TB): writing /dev/nvme0n1 (1.5TB): p2pmem buffer /sys/devices/pci0000:80/0000:80:02.0/0000:83:00.0/p2pmem/allocate.

    chunk size = 4096 : number of chunks =  1: total = 4.096kB : thread(s) = 1 : overlap = OFF.

    skip-read = OFF : skip-write =  OFF : duration = INF sec.

    buffer = 0x7fcdefb14000 (p2pmem): mmap = 4.096kB

    PAGE_SIZE = 4096B

    checking data with seed = 1731192712

MATCH on data check, 0x947c878 = 0x947c878.

Transfer:

  4.10kB in 908.9 us
```

and peer-to-peer DMA with quantization - output quantized data are fulfilled with zero values:

```
pratuszn@localhost p2pmem-test]$ sudo ./p2pmem-test /dev/nvme0n1 /dev/nvme0n1 /sys/devices/pci0000:80/0000:80:02.0/0000:83:00.0/p2pmem/allocate -c 1 -s 4k --check -o 0

Running p2pmem-test: reading /dev/nvme0n1 (1.5TB): writing /dev/nvme0n1 (1.5TB): p2pmem buffer /sys/devices/pci0000:80/0000:80:02.0/0000:83:00.0/p2pmem/allocate.

    chunk size = 4096 : number of chunks =  1: total = 4.096kB : thread(s) = 1 : overlap = OFF.

    skip-read = OFF : skip-write =  OFF : duration = INF sec.

    buffer = 0x7efda7b02000 (p2pmem): mmap = 4.096kB

    PAGE_SIZE = 4096B

    checking data with seed = 1731192742

MISMATCH on data check, 0x733ab514 != 0x23000000.

Transfer:

  4.10kB in 922.2 us
```

Additional peer-to-peer data transfers with quantization experiments are in progress using PCIe Gen3 switch (PEX 8747) on server motherboard and using external PCIe Gen5 switch (PEX89048) on separate development board are in progress and significant bandwidth improvement is expected .

4. Lessons Learned

2.1 NVMe Standard

In December 2023, NVMe released the first iteration of the *Computational Programs Command Set Specification*[4]. This specification defines two new names- paces/command sets:

- The Compute namespace and command sets defines the mechanism for offloading and storing programs on an NVMe device. These programs, which serve specific and well-defined purposes, can be defined by the device or downloaded from the host. A program is associated with one or several memory ranges defined with the SLM namespace.
- The Subsystem Local Memory (SLM) command set exposes the device memory as a collection of memory ranges and defines operations that the host and local programs (from the compute namespace) can issue on those memory ranges.

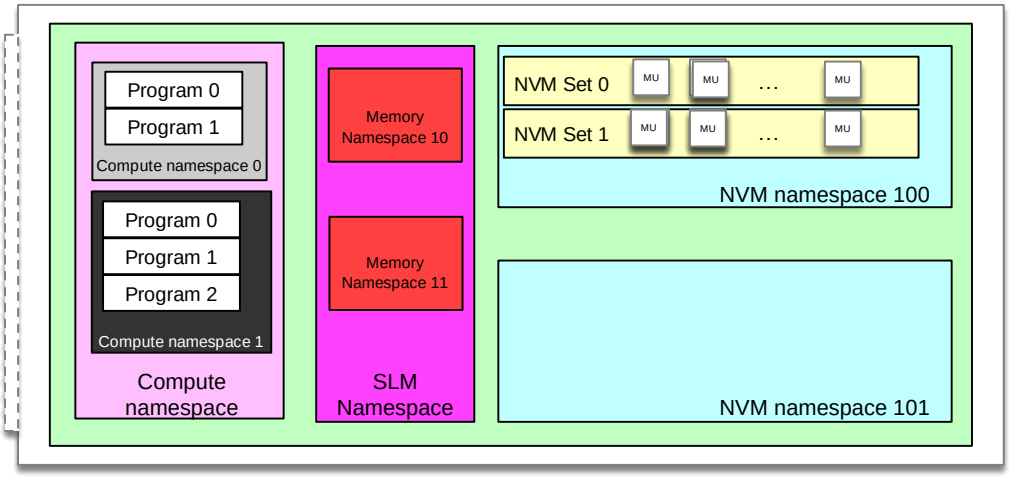


Figure 9: Compute and SLM namespaces in the NVMe standard. (Figure 4.5 in [1]).

The proposed execution model supposes that data is copied between storage (i.e., NVM namespaces) and the device subsystem local memory by the host, prior to (after) program execution, using an NVMe I/O commands. More specifically, the subsystem local memory (SLM) **copy command** is used (by the host) to copy data from an NVM namespace (i.e., flash) to a device memory namespace, from one device memory namespace to another, or from a device memory namespace to an NVM namespace. The **write command** is used to write data from the host to a device memory, i.e., to send input values to a program, while the **read command** is used to read data from the device memory onto the host, i.e., to get the output of a program.

In addition, commands on the computational program namespace make it possible to load or activate a program, associate it to a memory range, and execute it. Programs run to completion. Each program has a globally unique program identifier. A program can call another program. Device-specific programs are installed in hardware (FPGA bitstream) or software (OS image)

[4] https://nvmexpress.org/nvm-express-computational-storage-standardizing-storage-management-and-reducing-storage-and-latency-costs-in-the-enterprise/

before a namespace is attached. Downloadable programs, represented in hardware agnostic bytecode (eBPF), can be installed on a computational namespace after it is attached to a host.

There are many similarities between Delilah and the NVMe compute command set. In both cases, functions (that are denoted programs in the NMVe terminology) are loaded in program slots on the device, associated to memory ranges (where arguments are written by the host and results are read from the host, with memory ranges defined as shared buffers across functions) and executed. eBPF is used to represent downloadable programs, that may access device-specific programs (denoted registered functions in Delilah).

However, the standard does not address any issue related to (i) concurrent accesses to shared memory from multiple computational storage functions and (ii) concurrent access to data slots between host and computational storage function. We will consider these concurrency issues in the context of end-to-end programming of computational storage.

2.2 End-to-end Programming

End-to-end programming of computational storage relies on (i) a program executing on a host that issues commands triggering the execution of (ii) one or several functions on a computational storage processor. We denote CSF (computational storage functions), the functions executing on computational storage.

Here is an overview of the design space:

- CSFs are installed on computational storage statically (at deployment time) and/or dynamically (at run-time through I/Os). CSFs that are installed dynamically are denoted downloadable CSFs.
- CSFs are associated to memory regions statically (at deployment time) and/or dynamically (at run-time through I/Os).
- Downloadable CSFs are pre-defined on the host or they are automatically generated (through code generation).
- CSFs are compiled on the host into a bytecode representation that can be shipped to the device (e.g., eBPF) where it is interpreted or JIT-compiled, or into a binary representation that is well-suited for the target device (e.g., bitstream for FPGA, binary executable for general-purpose CPU).

When a host program executes a CSF, it proceeds through the following steps: (1) the program accesses the CSF (that is stored as bytecode or binary in a file when pre-defined or already in memory when automatically generated), (2) loads the CSF onto the device through an I/O command onto a program slot on the device (see Figure 4), (3) associates the CSF to one or several memory regions (including one data slot, and possibly several shared buffers), (4) writes input parameters into the data slot (or reuse an existing one), (5) executes the CSF on a given program slot and (6) reads the returned value on the data slot.

The first issue with this model is that the host program must wait until the CSF is done executing before it can read the return value produced by the CSF. There is a need for synchronization between device and host. In Delilah, we address this need with an explicit flag that is set whenever a function is done executing. This flag is located in the region of memory that is used to expose the computational storage device as a PCIe device (so-called BAR register). We

implemented a driver function that blocks until a CSF function is done executing. This way, the host can wait until a CSF function actually returns its value.

Put differently, the Delilah design and the NVMe computational storage standard are both essentially based on synchronous remote function calls between host and device. This is a direct consequence of the use of data slots to hold both the input and return value of each function call. Such synchronous function calls are a barrier to performance as they restrict the number of inflights CSF calls that can be issued by the host and managed by the device. A radically different design is needed to efficiently overlap processing between host and device.

The second issue concerns the concurrent execution of multiple programs accessing the same memory region. In cases where both programs write to the memory region, there is a need for concurrency control. This need is specially acute, when a program produces a value that is consumed by another.

In Delilah, as in the NVMe computational storage standard the necessary synchronization betwewen concurrent execution of CSFs must be managed by the host. The host must identify conflicts and schedules CSF execution to avoid them. Again, this form of coarse-grained synchronization is costly in terms of performance if at all possible.

An alternative is to introduce a resource manager on the device that mediates accesses to memory locally, see Figure 10. In order for the CSF programmer to focus on how a CSF should transform data, and not have to worry about concurrency, we want to provide the following guarantees:

1. (1) A CSF is guaranteed to eventually gain access to any region of shared memory used in the CSFs computation.
2. (2) Once a CSF has obtained access to a region of shared memory, the contents of that region will not change outside operations performed by the CSF itself.
3. (3) A CSF that has started execution will not be aborted due to concurrency conflicts.

Guarantee (1) and (3) also means that we promise no deadlocks, as this would require aborting a CSF to guarantee access. To achieve this we utilize a queue system to impose a total ordering on requests, an approach found in deterministic databases. See [4] for details.

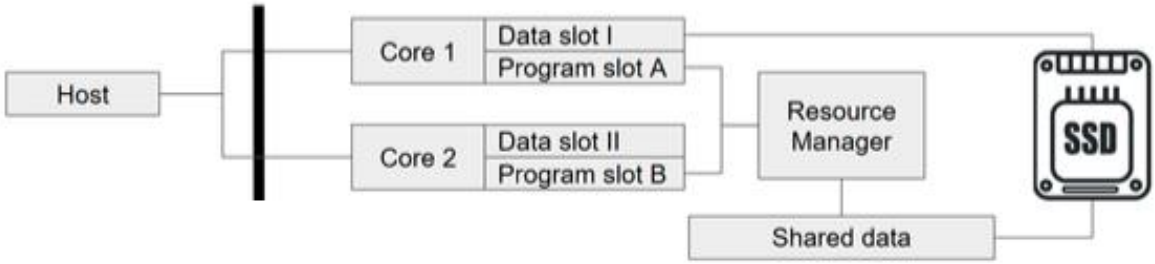


Figure 10: A resource manager is responsible for scheduling accesses to shared data to avoid conflicts.

5. Conclusion

We presented two prototypes of computational storage illustrating the on-path and off-path architectures. The Delilah prototype is on-path, with downloadable CSFs (represented in eBPF) as well as statically installed CSFs implemented both on a general-purpose processor equipped with an embedded operating system, and as hardware-accelerated functions in FPGA. CSFs are dynamically associated to memory regions at run-time. We showed that memory is the scarce resource on Delilah and is thus the main criteria that should be used to decide what programs to offload on the device. The Intel prototype is off-path, with statically installed CSF, implemented in FPGA and associated to memory regions fixed at deployment time. In the Intel prototype the CSF is incorporated as an extension of the DMA engine and it is triggered in the context of peer-to-peer DMA, without host involvement.

Finally, we reflected on the new NVMe standard for computational storage based on the lessons we learnt with Delilah. In particular, we identified synchronization and concurrency control as two key issues. We proposed to add a resource manager on computational storage as a way to deal with CSF concurrent accesses to shared memory regions.

Some of the key limitations of computational storage, in particular the synchronization issues, should be addressed before it can be incorporated into large-scale systems such as DAPHNE.

References

[1] Alberto Lerner, and Philippe Bonnet. “Principles of Database Systems and Solid-State Drives Co-Design”. Springer, 2025. https://link.springer.com/book/9783031578762

[2] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. “Delilah: eBPF-offload on Computational Storage”. In Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN '23). Association for Computing Machinery, New York, NY, USA, 70–76. https://doi.org/10.1145/3592980.3595319

[3] Niclas Hedam. “Delilah: Efficient eBPF Offload for Integrated Data Pipelines on Computational Storage”. Doctoral Thesis, ITU, 2024.

[4] Morten Tychsen, Jieung Kim, Philippe Bonnet. “Rogers: How to Write Programs for Computational Storage”. Under submission.