Domain Specific Accelerators for Networking: Technologies and Opportunities for OCP

Tom Herbert, SiPanda

Introduction

Domain Specific Accelerators, (*DSAs*) are specialized hardware components that accelerate certain workloads within a specific domain or application. The general motivation and benefits for DSA have been well documented. In this paper, we look at Domain Specific Accelerators specifically for the domain of networking. Considerations include models, manifestations, use cases, system design, software/hardware integration, hardware interfaces, APIs, and programming models.

DSA Models

Domain Specific Accelerators for networking can be characterized by three basic operational models:

- Offloads
- Acceleration instructions
- Accelerator engines

For each of these models we consider the programming model, hardware interaction, benefits and disadvantages, as well as opportunities for development. The three models are complementary and can be combined to work in tandem in a hybrid accelerator system model as shown in Figure 1.

Offloads

Offload acceleration, or just *offloads*, is an early form of DSA. Almost every commercial NIC (Network Interface Card) in existence offers some form of offload. The basic idea is that functionality running in the host CPU is run in a hardware device instead. Use cases of offloads include checksum offload, TCP segmentation offload, receive coalescing, TLS offload, and TCP offload.

The programming model of offloads is a "tail call" from a program running in a host CPU to a device, or from the device to the host CPU. In the networking transmit path, instructions to request offloaded functions are specified in a transmit descriptor, the descriptor is then interpreted by the NIC and the requested offload processing is performed (for instance, computing and setting the TCP checksum). Receive offload works in reverse, the NIC autonomously performs offload processing on received packets and reports results in receive descriptors (for instance, the ones' complement sum over the packet for verifying checksums).

The advantage of offloads is its simple programming model-- the details of an offload are easily abstracted from the programmer by a device driver.

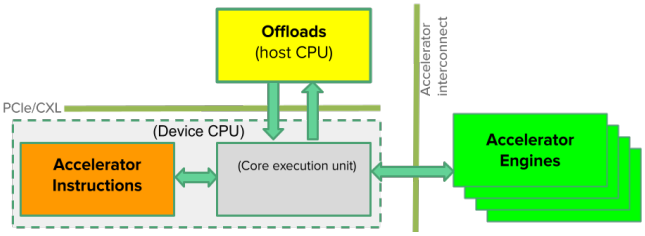

Figure 1: Example system architecture for accelerators. This diagram shows a hybrid accelerator model that employs offloads, accelerator instructions, and accelerator engines.

A disadvantage of offloads is that there's no means to leverage offload functionality outside of the networking path with any granularity. For instance, if a NIC has an encryption engine it can only be used in the context of sending a packet, a CPU program can't call the engine to encrypt an arbitrary block of host memory.

Deployment and ubiquity of offloads is underwhelming, and only a few simple offloads like checksum offload have been widely deployed. The impediment is that hardware offloads are only approximations of software functionality; even small discrepancies in functionality might lead to nondeterministic or incorrect behaviors. This motivates us to define the "fundamental requirement of offloads":

> *The functionality of a hardware offload must be **exactly** the same as that in the CPU software being offloaded*

While the fundamental requirement is easily stated, that begs the obvious question: How can this requirement be met? Our answer is:

> *Run the same code in the CPU and the offload target!*

Programmable devices are an enabler. We can infer an offload is viable if the same program code runs in both the host CPU and the device. Specifically, we would generate two images from the same base source code, one that runs in the CPU and one that runs in the offload device. There are some caveats: offload interfaces are still needed, resource limitations of the offload device must be taken in account, and a method is needed to query an offload device to see if the program to be offloaded has been loaded.

Acceleration instructions

Acceleration instructions are effectively Domain Specific Instructions. These are CPU instructions that can be output by a compiler in a CPU executable binary. Examples of acceleration instructions for networking are AES and CRC instructions. Open Instruction Set Architecture (ISA), like that of RISC-V, facilitates custom acceleration instructions.

The advantage of acceleration instructions is the simple programming model. Typically, the instructions are used in the back end of library functions such that the programmer doesn't even know they're using them. Acceleration instructions work best on data already in the CPU cache, with functions that don't require a lot of state. The disadvantage of acceleration instructions is that they need to be implemented in the CPU, and modifying a CPU may require licensing the ISA, standardizing new instructions, and compiler support for new instructions.

We have developed two sets of acceleration instructions for networking: DISC (Dynamic Instruction Set Computer) and parser instructions.

DISC defines a general acceleration instruction that can be used to support a large class of custom accelerations. The DISC opcode includes a function number as an immediate operand. The function number is dynamic and is mapped to a hardware function at runtime. As an example, the code below uses DISC instructions to compute a SipHash over sixty-four bytes. Performance is 97 cycles for RISC-V with DISC instructions, versus 793 cycles for x86.

```
// Load first thirty-two bytes into regs a0-a3
ld a0, 0(a9); ld a1, 8(a9); ld a2, 16(a9); ld a3, 24(a9)
li a4, 64 // Set length
mv a5,a10; mv a6, a11 // Set up keys
disc a8,siphash_start$7 // Hash 32 bytes
ld a0, 32(a9); ld a1, 40(a9)
disc a8,siphash_round$7 // Hash 16 bytes
ld a0, 48(a9); ld a1, 56(a9)
disc a8,siphash_end2$1 // Hash final 16 bytes
```

Parser instructions implement high performance protocol parsing in a CPU. They abstract out common functions of parsing into instructions, where each instruction can perform multiple sub-functions and leverage gate level parallelism for high performance. The code below shows an example for parsing the IPv4 header and extracting the IP addresses as metadata. Performance is 17 cycles for RISC-V with parser instructions, versus 77 cycles for x86.

```
prs.load.b paccum, pcurptr
prs.cmpi.h.stop paccum[1], 4 // Check IP version
prs.lensetmin.n pcurhdr, paccum[1], 4:20 // Hdr len
prs.load.b paccum, pcurptr+9
prs.cam.b pnext, paccum[0], 3 // Set up next proto
prs.load.h paccum, pcurptr+6
prs.load paccum, pcurptr+16 // Save addresses
prs.store.stp pframe+32, paccum
```

Accelerator Engines

Accelerator engines are external IP blocks that are accessed by the CPU over an accelerator interconnect such as BoW and AXI interfaces. Accelerator engines provide a message based interface that allows a CPU to make requests and get responses. The programming model is a type of Remote Procedure Call (RPC). Figure 2 shows an example processing flow for invoking an accelerator engine.

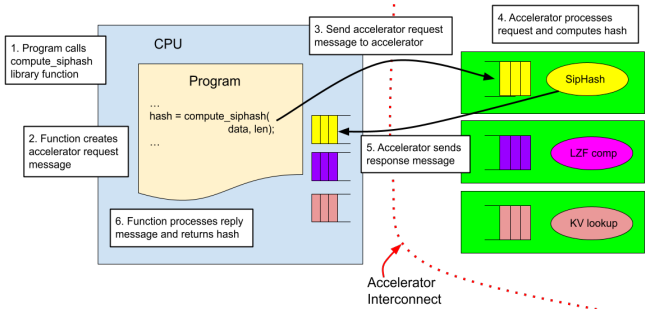

Figure 2: Example of invoking an accelerator engine. A function is called to compute the SipHash over some data block. The SipHash accelerator engine is invoked to perform the hash.

The advantage of accelerator engines is the RPC-like programming model which allows a lot of granularity in invoking functions. Ironically, this advantage is also its disadvantage-- allowing a CPU to interact directly with accelerator hardware opens the specter of isolation and security issues. These issues might be manageable in the kernel, but that would entail significant overhead in making system calls to perform accelerations. For lowest overhead and highest performance, we want to invoke accelerator engines directly from user space applications, but that requires a lot of infrastructure for security and isolation. As suggested in Figure 1, one solution is to invoke accelerator engines from Domain Specific CPUs in a closed and secure environment (like an App CPU in a SmartNIC).

A standard and open API is critical to the success of accelerator engines, and so we propose a simple message format for accelerator request and replies. This common format defines a sixty-four byte message, where the first sixty-four bits hold a control header for message delivery, and the following fifty-six bytes can be used as arguments. The message arguments can contain pointers to memory, including references to scatter/gather lists. For a hardware friendly data structure for scatter/gather lists, we propose *Packet Vector Buffers* or *Pvbufs;* these are based on iovec arrays with the twist that an entry may contain a pointer to a buffer or another iovec. We also propose *accelerator pipelining* where accelerators can be linked together in an accelerator pipeline such that output from one accelerator is the immediate input to another. Accelerator pipelines can be used to create "super accelerators" that run multiple functions over data in sequence without CPU intervention.

Opportunity

Realizing the dream of widely deployed and useful Domain Specific Accelerators for networking entails developing and innovating in the three DSA models. This is a cross disciplinary effort covering compilers, operating systems (Linux), CPU ISA, hardware devices and hardware interconnects. There are many opportunities for OCP involvement, especially in the accelerator hardware side and the development of open hardware and interfaces.