An NVMe-based FPGA Storage Workload Accelerator

Dr. Sean Gibb, VP Software
Eideticom
Acceleration

- Storage I/O Bandwidth rapidly increasing
- Storage workloads taxing on host CPU
- Hyperconverged storage exacerbates problem
- FPGAs provide compelling solutions for storage workloads
- NoLoad = NVMe Offload
NoLoad Accelerator

NoLoad Bitfiles

- U.2 FPGA Card
- COTS PCIe FPGA Card
- Cloud Servers i.e. Amazon F1
Why NVMe?

- **Accelerators require:**
  - Low latency
  - High throughput
  - Low CPU overhead
  - Multicore awareness
  - QoS awareness

- **NVMe provides:**
  - Low latency
  - High throughput
  - Low CPU overhead
  - Multicore awareness
  - QoS awareness

Why develop an maintain a driver when NVMe capabilities align so well with accelerator needs and you can have world-class driver writers working on your driver? Real question is “Why not NVMe?”
Host CPU communicates with accelerators via NVMe controller using NVMe commands
- NVMe controller pushes and pulls commands and data via DMA engine
- NVMe controller is in-house developed soft controller on a RISC-V
- Board has external DDR for accelerators that require large data storage
- Controller supports command queue and data CMB (using portion of DDR)
- Developed an accelerator wrapper to handle details of NVMe
NVMe for Accelerators

- Presents as NVMe 1.3 device with multiple namespaces
  - One namespace per accelerator
- Accelerators map to namespaces and are discovered using identify namespace
  - Vendor specific fields provide accelerator specific information
- Configuration using in-situ data path configuration or vendor specific command
- Input data and in-situ configuration are transferred using NVMe Writes to the namespace associated with the accelerator
- Output data and in-situ status are transferred using NVMe Reads to the namespace associated with the accelerator
NVMe for Accelerators

- Our in-house NVMe controller supports advanced features including queue and data CMB, SGL, and NVMe-oF
- We also support peer-to-peer (P2P) operation
- No customized drivers are required – all inbox drivers!
- Leverage industry-standard NVMe test tools
  - FIO and nvme-cli
  - Assist with deployment and benchmarking
- Take advantage of rich NVMe ecosystem
  - Can leverage servers and storage systems developed for NVMe
libnoload

- Developed a user API to assist with common tasks associated with accelerator over NVMe
- Provides C and C++ libraries
- Handles discovering NoLoad adapters and enumerating accelerators on the adapters
- Provides support to lock/unlock accelerators
- Provides thin wrappers over system calls for writing data to the accelerators and reading results back from the accelerators
  - Includes support for either synchronous or asynchronous transfers
- Handles seamless integration with our accelerator interface IP
- API is BSD licensed and available via our public github
Controller Performance

- Results for single RISC-V core controller implementation
- Saturating the bus for ≥ 32kB block transfer sizes for Gen3x8
- Saturating the bus for ≥ 16kB block transfer sizes for Gen3x4
- Latency for small blocks is <10μs
- Focus to date has been on accelerators with ≥16kB block size
- Working on multicore RISC-V system that drastically improves small block size performance
Case Study: ISA-L EC

- Implemented an RS(32,4) EC accelerator:
  - ISA-L compatible
  - Supports 16kB and 32kB block sizes
- Saturates PCIe Gen3x8 throughput with 1.5 cores and Gen3x16 (or Gen4x8) with 3 cores
- Modified ISA-L performance tests in less than an hour to use NoLoad NVMe accelerator
- Roadmap includes ZFS integration with accelerator
Case Study: Compression

- Implemented deflate and inflate accelerators:
  - zlib/gzip compatible
- A single deflate core is capable of > 1GB/s throughput
- Peer-to-peer offers reduced CPU usage and memory bandwidth
- Have P2P demo of deflate/inflate acceleration with Xilinx and NVMe-oF demo with Broadcom at FMS
  - Acceleration over NVMe using U.2 device
  - Has 3 deflate cores and 2 inflate cores to saturate PCIe Gen3x4 build
Case Study: Compression

<table>
<thead>
<tr>
<th>Engine</th>
<th>calgary.1G Compression Ratio</th>
<th>calgary.1G Throughput</th>
<th>cal4k.1G Compression Ratio</th>
<th>cal4k.1G Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZLIB-1 on CPU [2]</td>
<td>2.62</td>
<td>81 MB/s</td>
<td>29.56</td>
<td>340 MB/s</td>
</tr>
<tr>
<td>QAT-8955 [3]</td>
<td>2.60</td>
<td>1.46 GB/s</td>
<td>7.30</td>
<td>2.85 GB/s</td>
</tr>
<tr>
<td>Eideticom-H [2,4]</td>
<td>2.22</td>
<td>2.04 GB/s</td>
<td>35.81</td>
<td>2.97 GB/s</td>
</tr>
<tr>
<td>Eideticom-F [2,4]</td>
<td>2.12</td>
<td>2.19 GB/s</td>
<td>27.93</td>
<td>3.14 GB/s</td>
</tr>
</tbody>
</table>

2. Tests were performed on a single core of an Intel i5-6500 @3.2GHz machine running Ubuntu 16.04.
3. Intel QuickAssist 8955 with six compression cores on it’s ASIC chipset. All of the compression cores were used for this test [1].
4. FPGA test were performed on a NoLoad with three compression cores. The -H option provides higher compression while the -F option provides higher data throughput (~ same area)