Maxime Germano | 17.06.2022

SmartSSD Benchmarks

We are working with the U.2 SmartSSD from Xilinx. These benchmarks are made to test the throughput capabilities of the SmartSSD with and without the use of the FPGA. You can find the Github repo of the benchmarks here with the source files and binaries, as well as a Github repo of the whole setup here with different overviews and guides.

I. Without the FPGA

The first benchmark is done with fio (Flexible I/O tester), it tests the strict capabilities of the SSD without usage of the FPGA.

Two benchmarks were done, sequential read and write. Below are the commands, they can be found inside **script/benchmark_without_fpga.sh** in the repo.

```
sudo fio --name=seq-write --ioengine=libaio --iodepth=64 --rw=write --bs=1024k --direct=1 --
size=100% --numjobs=12 --runtime=900 --filename=/dev/nvme0n1 --group_reporting=1

sudo fio --name=seq-read --ioengine=libaio --iodepth=64 --rw=read --bs=1024k --direct=1 --size=100%
--numjobs=12 --runtime=900 --filename=/dev/nvme0n1 --group_reporting=1
```

We use `libaio`, Linux native asynchronous ioengine, with an `iodepth` of 64, and set `direct` to true to use non-buffered I/O. Setting `direct` to true also assures us that the desired depth is achieved, see documentation. We set the the number of jobs to 12 with group reporting and a runtime of 15min per benchmark on 100% of the SSD. Finally, we set the block size to 1024k. These values are the recommended ones by the Xilinx userguide for the SmartSSD.

We will look at the results later on in the report.

II. With the FPGA

Here we are conducting a throughput benchmark using the FPGA as DMA. We test the throughput of transfering 2GB of data both ways, reading and writing, from the CPU, to the global memory of the FPGA, and finally to the SSD.

Host Memory Access

Unfortunatly, the U.2 platform on the SmartSSD does not support host memory access from the kernel. All data needs to be copied to the FPGA's global memory before use or transfer. More information can be found here and here on the Xilinx documentation page.

Code

The code can be found inside **src/benchmark.cpp** in the repo and can be run with :

```
bin/benchmark -x bin/empty_kernel.xclbin -p <file path on the smartssd> -i <number of iterations>
```

An empty kernel is used because the data doesn't need to be modified by the fpga logic, but a kernel is still needed because the buffers are defined with it.

The SSD has an ext4 filesystem with an empty file on it.

I used the Vitis Accel examples to learn how to use the XRT APIs and understand how communication between the CPU, FPGA and SSD works.

1. XRT Native APIs

We use the XRT Native APIs to facilitate the usage of the FPGA on the SmartSSD. Here, we use three libraries from `/opt/xilinx/xrt/include/` :

```
#include "experimental/xrt_bo.h"
#include "experimental/xrt_device.h"
#include "experimental/xrt_kernel.h"
```

The first is for creating buffers, the second for handling the device, and the third for handling kernels for the FPGA. We also use the user-defined `cmdlineparser.h` from the Vitis Accel examples which handles the parsing of the command line arguments.

2. Timing

We use a simple user-defined class for timing everything :

```
class Timer {
        std::chrono::high_resolution_clock::time_point mTimeStart;

public:
        Timer() { reset(); }

        long long stop() {
                std::chrono::high_resolution_clock::time_point timeEnd =
std::chrono::high_resolution_clock::now();
                return std::chrono::duration_cast<std::chrono::microseconds>(timeEnd -
mTimeStart).count();
        }

        void reset() { mTimeStart = std::chrono::high_resolution_clock::now(); }
};
```

A timer can be started by initialising it : `Timer timer = Timer()`. Its current value can be retrieved with `stop()` and reset with `reset()`.

3. Initialisation

After having parsed the command line arguments, we initialise the needed structures :

```
auto device = xrt::device(device_index);
auto uuid = device.load_xclbin(binaryFile); // loads the kernel on the FPGA
auto krnl = xrt::kernel(device, uuid, "dummy_kernel");
```

We also define and fill the buffer used for writing :

```
size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
```

```
xrt::bo::flags flags = xrt::bo::flags::p2p;
auto bo = xrt::bo(device, vector_size_bytes, flags, krnl.group_id(1));
auto bo_map = bo.map<int*>();

std::fill(bo_map, bo_map + DATA_SIZE, 1); // takes approximately 30s
```

`DATA_SIZE` is defined as `(500000000)` to make `vector_syze_bytes` equal to 2GB. We define the buffer with the p2p flags, p2p flags are used for data transfer between an FPGA and an NVMe device. We define a buffer map to map the contents of the buffer object into host memory. The map can then be used to modify the content of the buffer.

We use the function `std::fill()` to fill the buffer with the value 1. This is a long operation (~30s) since there is 2GB to write. We do it in the main so that it is done only once. It cannot be skipped as `pwrite()` (see below) will not work without it. The read buffer does not need to be filled as it takes it values from the written file on the SSD.

4. Iterations

We repeat the next two operations by the number of iterations asked by the user in the command line.

WRITE

```
nvmeFd = open(filepath.c_str(), O_RDWR | O_DIRECT);
if (nvmeFd < 0) {
        std::cerr << "ERROR: open " << filepath << "failed: " << std::endl;
        return EXIT_FAILURE;
}
auto p1 = p2p_host_to_ssd(nvmeFd, krnl, bo, bo_map);
sum_write_throughput_from_fpga += p1.first;
sum_write_throughput_from_cpu += p1.second;
(void)close(nvmeFd);
```

We start by opening the file located on the SSD in R/W direct access. Then we call `p2p_host_to_ssd()` with the buffer and buffer map as arguments to transfer 2GB of data to the SSD. The function returns a pair of throughputs which we add to their respective sum to later compute the averages.

Here we see code from `p2p_host_to_ssd()` :

```
timer_from_cpu = Timer();

bo.sync(XCL_BO_SYNC_BO_TO_DEVICE);

timer_from_fpga = Timer();

ret = pwrite(nvmeFd, (void*)bo_map, vector_size_bytes, 0);
if (ret == -1) std::cout << "P2P: write() failed, err: " << ret << ", line: " << __LINE__ <<
std::endl;
```

The member function `sync()` of the buffer synchronizes the buffer data from the host to the FPGA's global memory. We define two timers, one before the `sync` operation and one after, to compute the throughputs host/SSD and FPGA/SSD.

Then we use `pwrite()` from `unistd.h` to write the data inside the buffer on the SSD file from the buffer map.

```
long long duration_from_cpu = timer_from_cpu.stop();
long long duration_from_fpga = timer_from_fpga.stop();
```

```
double throughput = vector_size_bytes;
throughput *= 1000000; // convert us to s;
throughput /= 1024 * 1024; // convert to MB

double throughput_from_fpga = throughput / duration_from_fpga;
double throughput_from_cpu = throughput / duration_from_cpu;

if (throughput_from_fpga > throughput_from_fpga_max_host_to_ssd) {
        throughput_from_fpga_max_host_to_ssd = throughput_from_fpga;
}

if (throughput_from_cpu > throughput_from_cpu_max_host_to_ssd) {
        throughput_from_cpu_max_host_to_ssd = throughput_from_cpu;
}

return std::make_pair(throughput_from_fpga, throughput_from_cpu);
```

We compute the troughputs as seen above using `vector_size_bytes` and the two durations. We modify the max throughputs if needed and return a pair composed of the two.

READ

```
nvmeFd = open(filepath.c_str(), O_RDWR | O_DIRECT);
if (nvmeFd < 0) {
        std::cerr << "ERROR: open " << filepath << "failed: " << std::endl;
        return EXIT_FAILURE;
}
auto p2 = p2p_ssd_to_host(nvmeFd, device, krnl);
sum_read_throughput_from_fpga += p2.first;
sum_read_throughput_from_cpu += p2.second;
(void)close(nvmeFd);
```

The read operation is done very similarly to the write operation seen above, except the buffer is instantiated inside the function, `pread` is used instead of `pwrite`, the `sync` operation is done after `pread`, and the timers are started at the same time and stopped before and after the `sync`. The function also returns a pair of throughputs.

Output

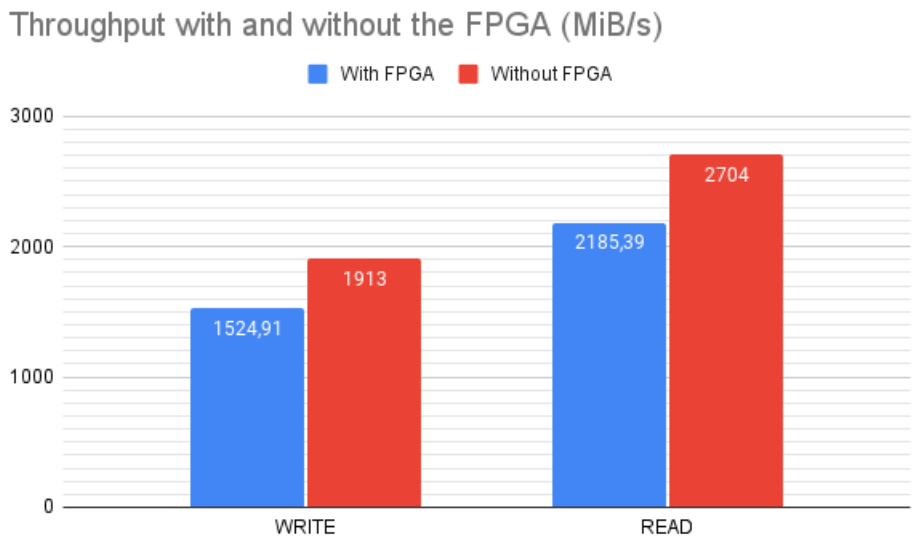
In the end, we compute the average throughputs for read and write and output everything to the console. Here is an example output :

```
Write bandwidth achieved :
        Max throughput from cpu: 1633.66 MB/s
        Average throughput from cpu: 1524.91 MB/s

        Max throughput from fpga: 1633.7 MB/s
        Average throughput from fpga: 1524.97 MB/s

Read bandwidth achieved :
        Max throughput from cpu: 2937.57 MB/s
        Average throughput from cpu: 2185.39 MB/s

        Max throughput from fpga: 2937.69 MB/s
        Average throughput from fpga: 2185.46 MB/s
```

```
Total time: 2h16m56s
```

III. Results

The throughputs CPU/SSD and FPGA/SSD were practically equal, this could mean that the `sync` operation only makes sure that everything has been written to global memory but that the writing happens at the start of the `fill` operation and therefore is not counted in the throughput. Or, it could mean that writing then reading from the global memory is really fast and creates almost as a link between the two PCIe bridges which would mean the difference in time is only the time it takes for a single packet to traverse the PCIe bridge from CPU to FPGA.



We can see a diminution of 25.5% for the WRITE operation and 24% for the READ operation when using the FPGA. The benchmark with the FPGA was run for 3000 iterations and took 2h17mins to finish.