Exploring OpenCL Memory Throughput on the Zynq

Technical Report no. 2016:04, ISSN 1652-926X

Chalmers University of Technology

Bo Joel Svensson
`bo.joel.svensson@gmail.com`

Abstract

The Zynq platform combines a general purpose processor and a Field Programmable Gate Array (FPGA), all within one single chip. Zynq-like systems are likely to become commonplace in various future computing system, from HPC compute nodes to embedded systems. In an HPC setting, the reconfigurable FPGA will be used to implement application specific accelerator units.

Much like a GPU, the FPGA in the Zynq can be programmed using OpenCL but unlike a GPU the flexibility of the Zynq is great. One example of this flexibility is in the interfacing of hardware accelerators with the processing system and memory interface. The programmer is free to choose between various different interfaces, combinations of interfaces and to explore configurations thereof.

This flexibility, however, makes even a simple task such as measuring the memory throughput capabilities of the accelerator much more involved and interesting. On a GPU, one would design a kernel that does nothing but move around data and then time its execution. The interfacing is fixed. On the Zynq we need to explore many different interface configurations and obtain different throughput result for each such setting.

This document explored a subset of the possible interface configurations in the hope of gaining insights on how to organize accelerators and interfaces for best performance.

1 Introduction

In this document we examine how fast data can be pumped through hardware accelerator units implemented on the reconfigurable logic within a Zynq device. The accelerator units of interest for this study, are generated using OpenCL based high level synthesis and the Vivado[6] tool chain.

The Zynq system consists of a multicore processor (2 ARM Cortex A9 cores) coupled with a Field Programmable Gate Array (FPGA). The processor and its associated fixed logic is called the *processing system* (PS) and the FPGA part is called the *programmable logic* (PL). The ARM cores run at 667MHz while the accelerator units in the PL run at speeds ranging from 50 - 250Mhz (typically 100Mhz).

The Zynq system has a number of interfaces between the PS and the PL, see Table 1. This flexibility in interfacing makes exploring the memory throughput on this device a lot more involved (and interesting) compared to on a GPU where the interfaces are fixed. On the Zynq platform we are free to use one of the interfaces (AXI GP/HP) to feed one or multiple, via a crossbar interconnect, accelerator units or we can feed multiple cooperating or independent accelerator units in parallel, using a separate interface for each.

Eventually all memory access must pass through the DDR controller out to DRAM. Note that no single PS - PL interface alone has a theoretical bandwidth as high as the DDR bandwidth. This means that in order to achieve as high throughput as possible from the PL, more than one AXI interface must be used.

Understanding how to best move data through an accelerator device is often the key knowledge to have at hand when trying to optimize for performance. With these experiments we hope to gain insights that form a basis for future exploration of best-practices for performance on a Zynq (or Zynq-like) system.

Table 1 is a reproduction of a similar table in "Zynq-7000 technical reference manual" [7]. This table lists theoretical maximum bandwidths of the various interfaces within the Zynq system. There are however a range of Zynq devices, some with slightly different specification regarding these interfaces. Here the table has been expanded with the actual numbers for the experimental platforms used. The theoretical maximum bandwidths listed in the table serves as a guideline and a sanity check for the experiments presented here.

2 OpenCL and Vivado

OpenCL is a programming model for heterogeneous systems[5]. OpenCL can be used to program systems consisting of general purpose processors and accelerator devices. An example is a system consisting of a CPU and a GPU, in which case the CPU is called the *host* and the GPU is called the *device*. OpenCL also

Interface (IF)	#	Type	Bits	IF MHz	Read (MB/s)	Write (MB/s)	Read + Write	Total (MB/s)
GP AXI	2	Slave	32	150	600	600	1200	2400
GP AXI	2	Master	32	150	600	600	1200	2400
HP AXI	4	Slave	64	150	1200	1200	2400	9600
AXI ACP	1	Slave	64	150	1200	1200	2400	2400
DDR	1	External	32	1066	4264	4264	4264	4264
OCM	1	Internal	64	222	1779	1779	3557	3557
Zynqberry								
GP AXI	2	Slave	32	100	400	400	800	1600
GP AXI	2	Master	32	100	400	400	800	1600
HP AXI	4	Slave	64	100	800	800	1600	6400
DDR	1	External	16	1066	2132	2132	2132	2132
ZedBoard								
GP AXI	2	Slave	32	100	400	400	800	1600
GP AXI	2	Master	32	100	400	400	800	1600
HP AXI	4	Slave	64	100	800	800	1600	6400
DDR	1	External	32	1066	4264	4264	4264	4264

Table 1: This table shows the theoretical bandwidths for the various memories and interconnects present on the Zynq platform.The top part of this table is reproduced from [7] and shows numbers applicable to a wide range of Zynq devices. The bottom part of the table presents values scaled given characteristics of the hardware platforms and experimental setup used in in this document. The Zynqberry and the ZedBoard sections of the table, list only information on the interfaces that are explored in this document.

differentiates between programs that are executed on the host and device. Programs written for the device are usually called *kernels* and are developed in a dialect of C or C++ that has been extended for massively parallel execution. There is an OpenCL runtime library for execution on the host. This host part of OpenCL is responsible for launching tasks onto attached devices.

When using OpenCL for hardware synthesis in Vivado, we are only concerned with kernels. The Vivado tool does not provide any OpenCL runtime for launching these kernels from the PS part of the Zynq system. This, however, makes sense given the freedom we have in interfacing the generated hardware with the PS system. A specialized host side RTS would be needed for each choice.

The OpenCL kernel programming model is based on a hierarchy of parallel computational resources. At the bottom of this hierarchy are *work items*,single threads of control. Multiple such work items that cooperate to solve a subproblem (possibly using local memory) are called a *work group*. Work items within a work group can be synchronized using a `barrier` statement. Each work item and each work group have an identity, an integer value that identifies a work item within a work group and a work group within a collection of workgroups. In OpenCL the collection of work groups, and the highest level of the hierarchy, is called the NDRange. The NDRange specifies dimensions, of a large number, of independent work groups. Work groups must be completely independent, and cannot be synchronized except by designing the computation as separate NDRange launches.

When using Vivado for OpenCL based hardware synthesis, the result is a hardware implementation of a single work group. Since these workgroups are independent, we are free to instantiate more than one such hardware unit within the FPGA fabric and run them in parallel. The experiments in this paper investigates what happens the total memory throughput when multiple hardware workgroups are instantiated and connected to one or more of the interfaces available. Figure 1 illustrates how more than one hardware work group unit share an interface and Figure 2 show how two interfaces are shared.

The hardware work group unit generated by Vivado has two interfaces. The first, the *control interface*, is used for starting, passing arguments and checking the status of the device and the second, *memory interface* is used to service the memory accesses of the work group. The control interface consists of a set of memory mapped registers. The basic information that needs to be passed to an OpenCL work group via the control registers is the following:

- Work group id: X,Y and Z identity of the work group within the NDRange.
- (Optional) Global offsets: X,Y and Z offset values. These are usually zero but if non-zero, the given amount will be added to each work item id within the work group.
- (Optional) Work group dimension: extents in X,Y and Z direction of the work group. This is only required for work groups that have not been implemented for a fixed size (using the `reqd_work_group_size` attribute).
- Pointers to I/O arrays: One register will be present for each input and output as defined in the OpenCL source code.

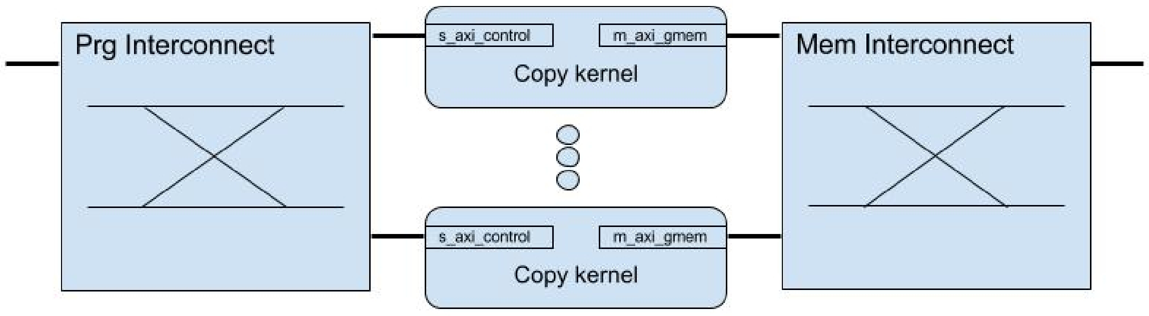


Figure 1: Multiple hardware work group units connected to the PS. The interconnect on the left, goes to an AXI Master port and is used for control of the hardware work groups (starting, checking status). The interconnect on the right, connects the work groups to memory via an AXI Slave port.

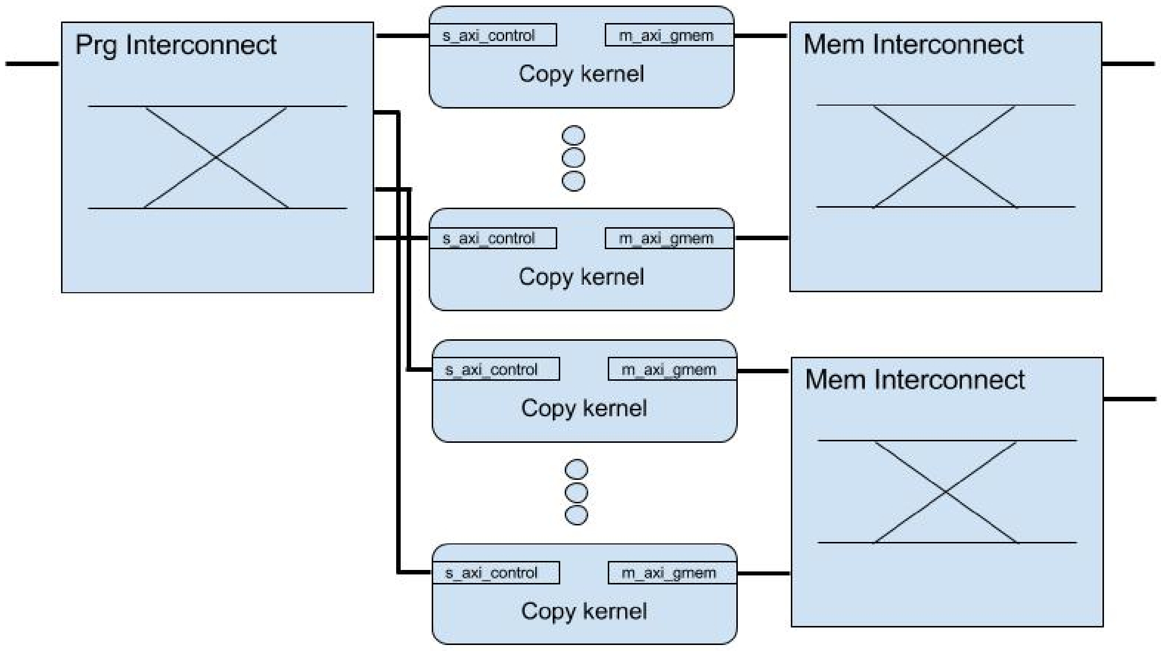


Figure 2: Here the memory interconnect has been duplicated and two AXI Slave ports are used, in parallel, to supply the work groups with data.

3 OpenCL Kernel Bandwidth Experiments

Here we present the results of a series of experiments that are intended to show what happens to the throughput as parallel workgroups are executing and performing I/O over different configurations of interfaces.

Four different kernels are used in the experiments, called “Kernel 1” to “Kernel 4”. Kernel one is very simplistic, each work item executes a for loop that copies data from the input to the output array. The high level synthesis tool is able to infer that these memory accesses should be done as bursts. Kernels 2,3 and 4 are all very similar to each other, they all use local memory but differ in how much shared memory they use. Kernels 2 to 4 use the `async_work_group_copy` for the data movement. These kernels also differ in how large chunks the copy function moves per call.

3.1 Hardware Platforms

Two different Zynq based boards are used in the bandwidth exploration presented here. One is a small Zynq 7010 based board called the Zynqberry[1] with 128MB of DRAM. The other is the ZedBoard which is based on the slightly larger Zynq 7020 and it has 512MB of DRAM. As the 7020 device is larger there is room for more instances of the hardware work groups. The larger FPGA on the ZedBoard together with the larger DRAM means that more memory will be shuffled around in the experiments run on that device. The other important characteristics of these two platforms are listed in Table 1.

3.2 Rudimentary OpenCL Work Group Scheduling

As previously mentioned, the Vivado system does not provide us with a host side OpenCL library. That is, there is no provided `enqueueNDRangeKernel` function that executes a set of workgroups on the FPGA. We need to write a basic version of this function ourselves.

Each hardware OpenCL work group instantiated into the FPGA has a control register. Through this register we can start the work group operation and poll to check if it has completed. Each work group is started more than once to complete the total task of moving an amount of data from one point to another.

```
int acc_device = 0;
for (int = 0; i < NUM_WORK_GROUPS; i ++) {

  while(1) {
    if (isFree(devices[acc_device]){
      set_wg_id(devices[acc_device],i);
      start(devices[acc_device]);

      /* prepare to start the next device */
      acc_device = (acc_device + 1) % NUM_HW_WORK_GROUPS;
      break;
    }
    /* check the next device */
    acc_device = (acc_device + 1) % NUM_HW_WORK_GROUPS;
  }
}
```

The program above is a basic work group scheduler. It loops over all the work that needs to be done. In each iteration of the outer loop the hardware units are polled until a free one is found which is then started.

3.3 (Kernel 1) No Local Storage

The first OpenCL kernel, copies data from one input array to an output. Each work item executes a loop that copies 1024 * 4Bytes integer values. The Vivado tool is able to infer that bursts should be used when reading and writing data using this kernel. The burst size inferred is 1024.

```
__kernel void wg_cpy_128(__global int *out, __global int *in) {
#pragma HLS UNROLL factor 4

  unsigned int gid = get_global_id(0);
  int i;

  for (i = 0; i < 1024; i ++){
    out[gid * 1024 + i] = in[gid * 1024 + i];
  }
}
```

3.3.1 General Purpose AXI Slave Interface

Figure 3 shows how the achieved throughput changes as an increasing number of hardware work group units are sharing a general purpose AXI port for its memory traffic.

Zynqberry When using one hardware work group unit a throughput of 301MB/s is achieved. The general purpose AXI port has a theoretical max bandwidth of 800MB/s, 400MB/s read and 400MB/s write. When using a single work group in this configuration, 38% of the theoretical maximum is achieved.

The highest throughput values using this kernel is found when running 3 or 4 work groups in parallel. The maximum throughput measured is 593MB/s or 74% of the ports theoretical max bandwidth. The average throughput when using 3 or 4 hardware work groups was 582MB/s and the median was 586MB/s.

As more hardware work groups are added beyond 3 or 4, there is no increase in throughput. Rather there is a slight decrease. This is possibly because of congestion in the memory interface increasing.

The left chart in Figure 3, shows the achieved throughput as the number of hardware work groups, connected to a single general purpose AXI port, is varied between 1 and 6.

ZedBoard The slightly larger FPGA allows for more hardware work groups. Here, between 1 and 8 hardware work groups are used in parallel. The maximum measured throughput is, however, again found when using 3 parallel work groups connected to a single interface.

The highest throughput measured is 598MB/s, which is 75% of the theoretical maximum of a single general purpose AXI port. The right side of Figure 3 shows the throughput achieved at varying number of work groups for one general purpose port.

3.3.2 One High Performance AXI Slave Interfaces

Figure 4 shows how the achieved throughput increases as hardware work groups run in parallel sharing a single high performance AXI port. The high performance port has a theoretical maximum bandwidth of 1600MB/s.

ZynqBerry The maximum throughput measured using this configuration is 1283MB/s or 80% of the theoretical maximum of a single high performance AXI port. This throughput is achieved using four parallel work groups sharing a single interface.

ZedBoard The maximum throughput measured on the ZedBoard using this configuration is 1285MB/s or 80% of the theoretical maximum. Again this value is obtained at four parallel work groups sharing a port.

3.3.3 two High Performance AXI Slave Interfaces

Figure 5 shows measurements obtained when using two high performance AXI ports in parallel. Two high performance ports in parallel, have a maximum bandwidth of 3200MB/s.

ZynqBerry On the ZynqBerry, the maximum measured throughput using two high performance ports is achieved when six work groups are used in parallel. Six work groups is the maximum amount that was placed onto the smaller FPGA of the Zynq 7010. The throughput achieved is 1624MB/s or 51% of the maximum. However, two high performance AXI ports together have a bandwidth that is higher than the the bandwidth to DRAM via the DDR controlled which is 2132MB/s. The achieved memory throughput here is 76% of the maximum bandwidth to memory.

ZedBoard Here the maximum measured throughput is 2530MB/s or 79% of maximum. This value is very close to two times the throughput of using a single port, see above.

3.3.4 Four High Performance AXI Slave Interfaces

Four high performance ports together has a maximum bandwidth of 6400MB/s. This surpasses the bandwidth to DRAM on both the ZedBoard and the ZynqBerry. This means that it is in this case more interesting to compare to the DRAM bandwidth which is 2132MB/s on the ZynqBerry and 4264MB/s on the ZedBoard.

ZynqBerry Using six parallel work groups achieves a throughput of at most 1711MB/s on this device. This is 80% of the bandwidth to DRAM.

ZedBoard A maximum throughput of 3333MB/s is measured when using 12 parallel work groups. This is 78% of the DRAM bandwidth.

3.3.5 Summary

The following tables shows the configuration that achieved the highest throughput for each interface. It also shows the obtained throughput as a percentage of theoretical DRAM bandwidth.

ZynqBerry			
Interfaces	Work groups	Max throughput MB/s	% of DRAM bandwidth
1 GP	3	593	18
1 HP	4	1282	60
2 HP	8	1624	76
4 HP	6	1711	80

ZedBoard			
Interfaces	Work groups	Max throughput MB/s	% of DRAM bandwidth
1 GP	3	598	14
1 HP	4	1285	30
2 HP	8	2530	59
4 HP	12	3333	78

3.4 (Kernel 2) Using Local Storage

The second OpenCL kernel copies data using the `asynq_work_group_copy` function which moves chunks of data from global memory (DRAM) into local memory (BRAMs) and then back again to global memory.

```
#define BUF_SIZE 32768

__kernel void wg_cpy(__global int *out, __global int *in) {

  __local int buf[BUF_SIZE];

  unsigned int group_id = get_group_id(0);

  __global int* p_r = in + (group_id * BUF_SIZE);
  __global int* p_w = out + (group_id * BUF_SIZE);

  async_work_group_copy(buf,p_r,BUF_SIZE,0);

  barrier(CLK_LOCAL_MEM_FENCE);

  async_work_group_copy(p_w,buf,BUF_SIZE,0);

}
```

The left side of Figure 7, shows the achieved throughput using one or two work groups connected one general purpose port. The maximum throughput obtained is 600MB/s which is 75% of theoretical maximum of one GP AXI port.

The right part of Figure 7 shows that a throughput of at most 1514MB/s which is 36% of the DRAM bandwidth in the system, when using high performance ports.

Due to the large resource usage of these kernels (mainly the BRAMs), not as many could be fit onto the FPGA.

3.5 (Kernel 3) Reusing Local Storage

```
#define ROUNDS   4
#define BUF_SIZE 32768

__kernel void wg_cpy2(__global int *out, __global int *in) {

  int i = 0;

  __local int buf[BUF_SIZE];

  for (i = 0; i < ROUNDS; i ++) {
    unsigned int group_id = get_group_id(0);

    __global int* p_r = in + (((group_id * 4)+ i) * BUF_SIZE);
    __global int* p_w = out + (((group_id * 4) + i) * BUF_SIZE);

    async_work_group_copy(buf,p_r,BUF_SIZE,0);

    barrier(CLK_LOCAL_MEM_FENCE);

    async_work_group_copy(p_w,buf,BUF_SIZE,0);

    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Figure 8, shows that this kernel is performing no better compared to “Kernel 2”. The maximum throughput achieved is 1524MB/s or 35% of the DRAM bandwidth.

3.6 (Kernel 4) Using More Local Storage

Kernel 4 is a simple modification of Kernel 1. The `BUF_SIZE` is redefined to be 65536.

Kernel 4 uses additional BRAM resources and at most 2 work groups can be instantiated on the FPGA. Figure 8 shows that this is reflected in the throughput. A throughput of at most 762MB/s is obtained.

4 Conclusion

The following table lists the best port configurations found for each of the kernels (1-4). The values in the table are measurements from the ZedBoard with the larger Zynq 7020. At most 78% of DRAM bandwidth was achieved when using the simplest of the kernels (Kernel 1).

Kernel	Port Config	#Work groups	Throughput MB/s	% DRAM
1	4 HP	12	3333	78
2	4 HP	4	1514	36
3	4 HP	4	1524	36
4	2 HP	2	762	18

The next table shows the total work group latency and the resource utilization for each of the kernels used in this document. The utilization numbers are in relation to the larger Zynq 7020 FPGA.

Kernel	Work group latency	BRAM %	DSP %	FF %	LUT %
1	263329	0	0	1	3
2	65552	22	0	0	2
3	262213	22	0	0	2
4	131088	45	0	0	2

Given that Kernel 1 consumes less of the FPGA resources, we can utilize more instances of it in parallel. This may be the answer to why the most naive of the kernels manages to move the most data. The graphs using Kernel 1 all plateau after adding a certain number of parallel workgroups. It is likely that that plateau occurs because of congestion in the interface. On a larger device with more resources we would likely have seen the same plateau in the charts for Kernels 2 - 4. Unfortunately we could not, with the given hardware, investigate at what throughput this plateau would happen.

Most realistic kernels will have a data-transfer to compute ratio that is different from the ones used in these experiments. The experiments show that even for kernels that mostly just read and write data, there is a benefit to add parallel work groups. Kernels with less data transfer per unit of compute should see similar benefits.

5 Future Work

None of the kernels used were optimized to any degree. That is, no pipelining directives and such were used. It would be interesting to see what throughput can be achieved using a single kernel which has been aggressively optimized.

The Zynq also has an ACP (Accelerator Coherency Port), that has not been explored here. This port has been studied in reference [3]. The ACP port is for maintaining a coherent view of memory between the CPU and the accelerator (work group). Traditionally the OpenCL programming model isolates the accelerator work, by copying the data to a separate memory area (usually device owned dram) and the accelerator then has exclusive access to this data during its operation. On the Zynq systems used here, there is however only one DRAM shared by CPU and FPGA. The synchronization between CPU and OpenCL work group is, however, coarse grained making it hard to see how to set up CPU/FPGA cooperations using OpenCL (except of course to run totally independent work groups on both CPU and and FPGA, which would still require no coherency in the general case).

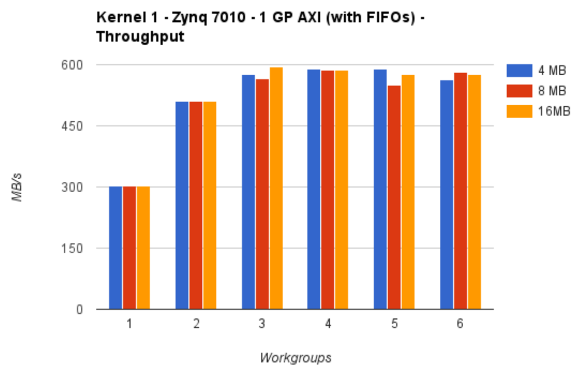
The OCM (On Chip Memory) is another interesting feature of the Zynq device that has not been explored here. Finding ways to make use of this memory is future work.

6 Related Work

The scheduling overhead and cost of interrupts when launching many short latency accelerator units is studied in [2]. This is not directly related to the work performed here, except for the potential overhead by scheduling of our OpenCL work groups. The work groups used here are however not short latency.

Reference [3] explores the performance of the ACP port. This port has been completely left out of the experiments performed in this document. So for a treatment of the ACP port we refer to [3].

In reference [4] a comparison of all the interfaces within the Zynq is presented. The exploration presented in this document is different by exploring these interfaces (some of them) from an OpenCL perspective specifically. We also explore the effects of using more than one hardware accelerator per port, which is not part of the comparison in the reference.



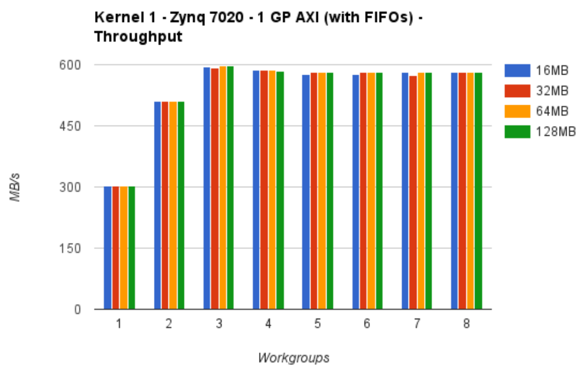
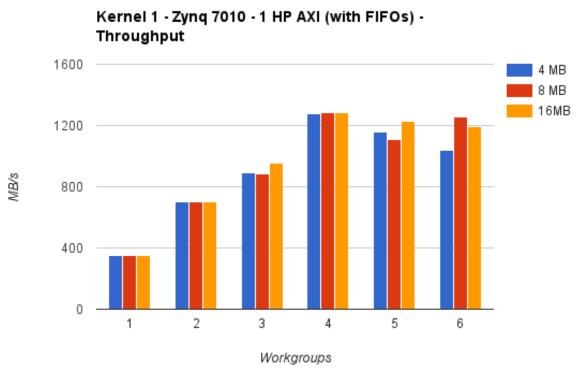


Figure 3: Obtained throughput using one of the general purpose AXI Slave ports. Left hand side shows results for the ZynqBerry and the right shows results for the ZedBoard.



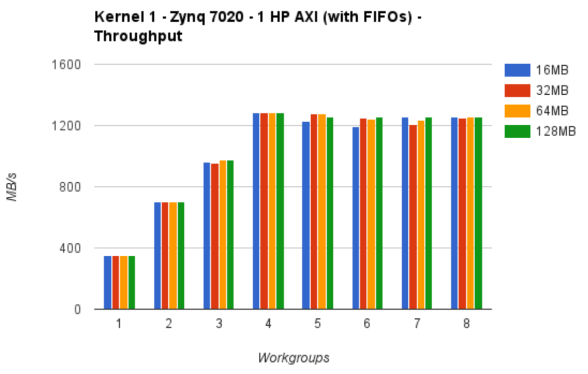
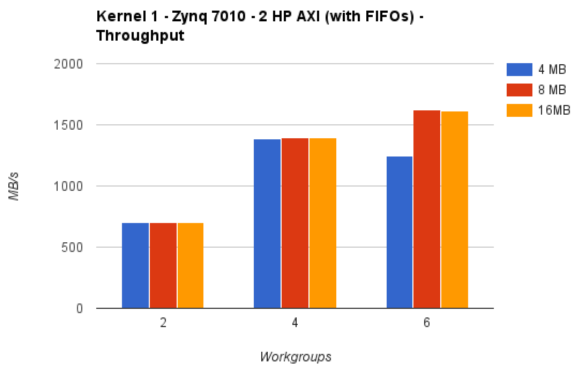


Figure 4: Obtained throughput using just one of the high performance AXI Slave ports. Left: ZynqBerry. Right: ZedBoard



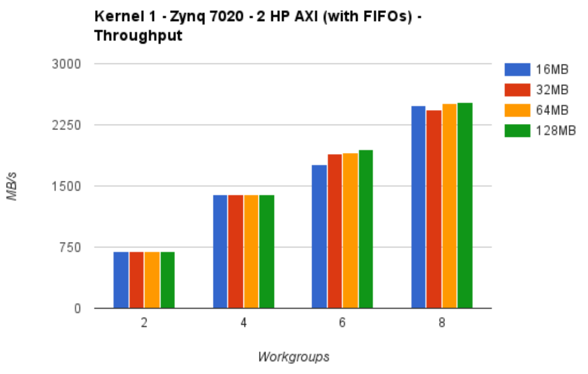
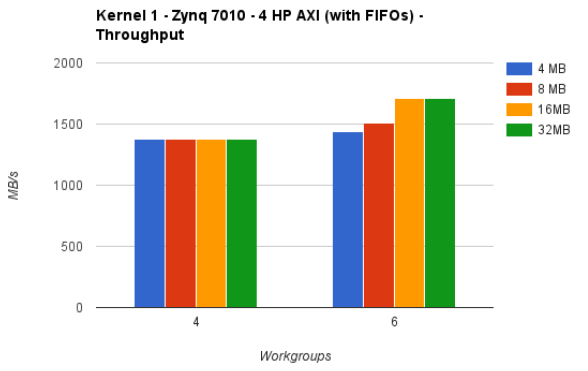


Figure 5: Obtained throughput using two of the high performance AXI Slave ports. Left: ZynqBerry. Right: ZedBoard.



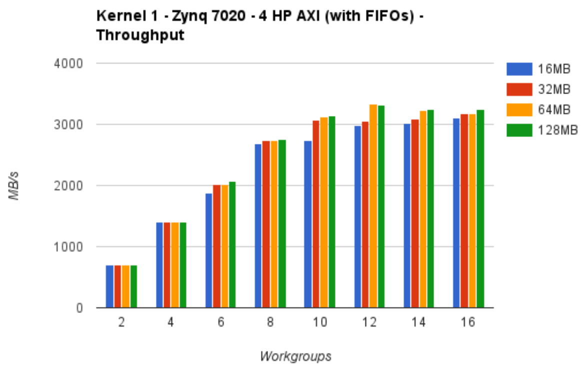
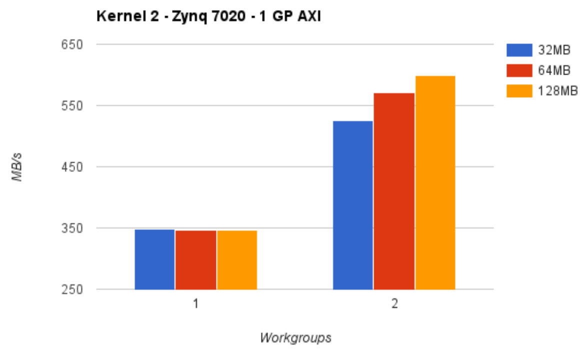


Figure 6: Obtained throughput using all four high performance AXI Slave ports. Left: ZynqBerry. Right: ZedBoard.



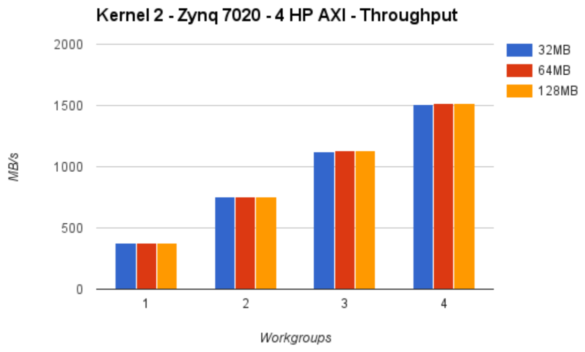
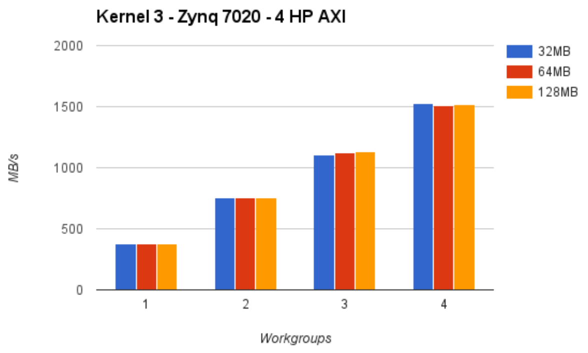


Figure 7: Obtained throughput using Kernel 2. Left: Results when using one GP port. Right: Results when using four HP ports.



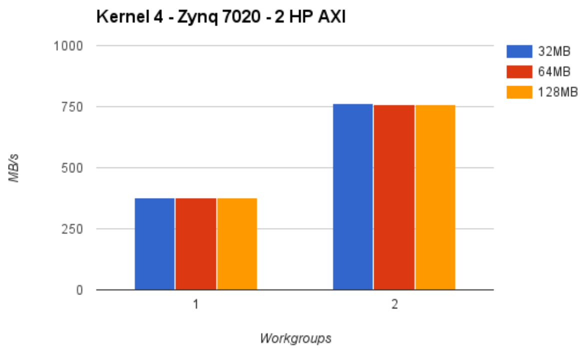


Figure 8: **ZedBoard**:

References

[1] Trenz electronic. Zynqberry. `https://wiki.trenz-electronic.de/pages/viewpage.action?pageId=21921924`.

[2] Sébastien Lafond and Johan Lilius. Interrupt costs in embedded system with short latency hardware accelerators. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 317–325. IEEE, 2008.

[3] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. Energy and performance exploration of accelerator coherency port using xilinx zynq. In *Proceedings of the 10th FPGAworld Conference*, page 5. ACM, 2013.

[4] João Silva, Valery Sklyarov, and Iouliia Skliarova. Comparison of on-chip communications in zynq-7000 all programmable systems-on-chip. *IEEE Embedded Systems Letters*, 7(1):31–34, 2015.

[5] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[6] Xilinx. Vivado. `http://www.xilinx.com/products/design-tools/vivado.html`.

[7] Xilinx. Zynq-7000 All Programmable SoC: Technical Reference Manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`.