OpenCL Reduction on the ZYNQ

The ZYNQ is not a GPU

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

1 Introduction

It is well known that OpenCL, while being portable, is not "performance"-portable[2, 3]. In other words, code written in OpenCL can be expected to "work" on any OpenCL platform but there are no guarantees that the performance characteristics transfer from one system to the next.

This document presents the results of applying a set of optimisations from a GPU tutorial onto a ZYNQ. The computation that is being optimized is reduction (sum) of an array of integers and the optimisation steps are taken from an NVIDIA tutorial [1].

GPUs and FPGAs are both massively parallel platforms. It does not seem impossible that a single decomposition of a computation performs well on both systems. The experiments presented in this document show that this is not the case. At least, this is not the case under the current hardware synthesis methodology as applied by Vivado onto OpenCL kernels.

1.1 Experimental setup

All reduction performance experiments are performed on a ZYNQ 7010. The hardware kernels are generated using VIVADO HLS 2016.3 and synthesized using VIVADO 2016.3.

The NVIDIA reduction tutorial [1] kernels written in CUDA have been translated to OpenCL where applicable. This resulted in six reduction kernels being synthesized and tested on the ZYNQ. The NVIDIA tutorial present more steps than six, but some of these related to warp level optimisations. It is possible that warp-level optimisations could be applied to the OpenCL code if the wavefront size is known. It is, however, unclear to me what the generated wavefront size is on a hardware synthesized OpenCL kernel (more thoughts on this in section 4).

2 Refinement of a Reduction Kernel

In this section a reduction kernel is refined step by step. The optimisation steps mimic those performed in the NVIDIA reduction tutorial[1]. The starting point is a kernel that may appear idiomatic, at least to a GPU programmer. The refined kernels (`reduce2` - `reduce6`) are all collected in section 6.

Kernel 1

```
__kernel void __attribute((reqd_work_group_size(BLOCKSIZE,1,1)))
  reduce1(__global int *input, __global int* output) {

  __local int sdata [BLOCKSIZE];

  int lid = get_local_id(0);
  int bid = get_group_id(0);
  int i   = bid * BLOCKSIZE + lid;

  sdata[lid] = input[i];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (int s = 1; s < BLOCKSIZE; s*=2) {
    if (lid % (2*s) == 0) {
      sdata[lid] += sdata[lid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (lid == 0) output[bid] = sdata[0];
}

```

The table below shows the performance obtained when doing a 4million elements reduction on the ZYNQ compared to the G80 used in the NVIDIA reduction tutorial. Note that the purpose is not to compare the absolute levels of performance between the platforms. Rather the interesting bit is the relative increase in performance between steps. These optimisations steps clearly are more beneficial on the G80.

	ZYNQ Time (ms)	ZYNQ MB/s	ZYNQ % Bandwidth 1 HP-AXI IF	ZYNQ Speedup	G80 Time (ms)	G80 GB/s	G80 % Bandwidth	G80 speedup
Kernel1	4702.01	3.4	0.4		8.054	2.083	2.4	
Kernel2	2081.66	7.7	1	2.26	3.456	4.854	5.6	2.33
Kernel3	1447.12	11.1	1.4	3.25	1.722	9.741	11.3	4.68
Kernel4	1503.3	10.6	1.3	3.13	0.965	17.377	20.1	8.34
Kernel5	1373.96	11.7	1.5	3.42	0.536	31.289	36.2	15.01
Kernel6	804.46	19.9	2.5	5.85	0.268	62.671	73	30.04

The performance of these kernels on the ZYNQ is quite horrible. The kernel used as a starting point is only able to reach 3.4MBps of throughput over the AXI interface. The best kernel only makes use of 2.5% of the memory bandwidth of a single high performance AXI interface. However, each optimisation step does seem to improve performance, see figure 1. The final optimisation step (`reduce6`) adds sequential computation. In the next section we push sequentiality to the extreme and investigates what happens.

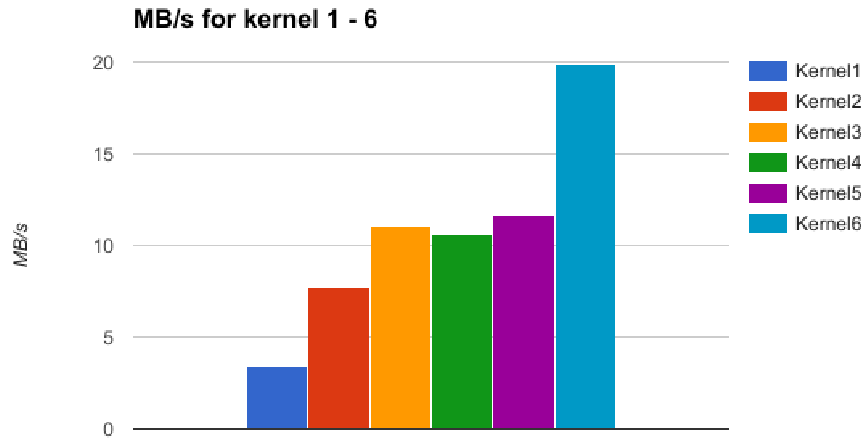

Figure 1

3 Starting over!

In the previous section, a bunch of OpenCL kernels where evaluated with very poor results on the ZYNQ. Here, degenerate OpenCL kernels are implemented and tested for performance. The first of these degenerate kernels is entirely sequentialised and uses no shared memory.

Kernel 0

```
__kernel void __attribute__ ((reqd_work_group_size(1,1,1)))
  reduce0(__global int *input, __global int *output) {

    int bid = get_group_id(0);

    int i = 0;
    int acc = 0;
    int index = index = bid * BLOCKSIZE;

    for (i = 0; i < BLOCKSIZE; i ++) {
      acc += input[index + i];
    }
    output[bid] = acc;
}
```

The performance of this kernel , again on 4million elements, is about 6x that of the best performing kernel in the section before. It achieved approximately 34x the throughput compared to the first kernel, `reduce1`. The analysis of why this kernel is performing so much better than the previous ones is saved until section 4.

The next kernel, `reduce0a`, applies unrolling to the degenerate sequential kernel.

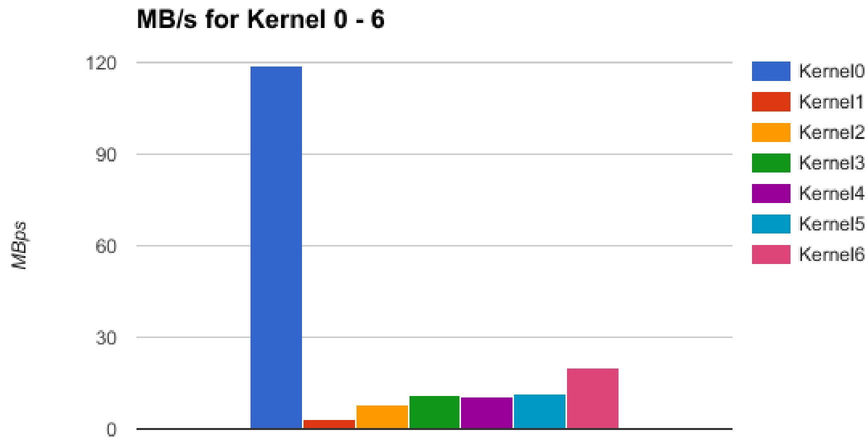

Figure 2

Kernel 0a
```
__kernel void __attribute__ ((reqd_work_group_size(1,1,1)))
  reduce0a(__global int *input, __global int *output) {

    int bid = get_group_id(0);

    int i = 0;
    int acc = 0;
    int index = bid * BLOCKSIZE;


    for (i = 0; i < BLOCKSIZE; i ++) {
    #pragma HLS UNROLL
      acc += input[index + i];
    }
    output[bid] = acc;
}
```

And the next attempt applies pipelining and some parallelism. The parallelism in this case comes from using a vector type, `int2`, for loading values and computing intermediate results. The `+=` operator is overloaded in the code below for operation of vectors of length 2.

Kernel 0b
```
__kernel void __attribute__ ((reqd_work_group_size(1,1,1)))
  reduce0b(__global int2 *input, __global int *output) {

    int bid = get_group_id(0);

    int i = 0;
    int2 acc = (int2)(0,0);
    int index = bid * BLOCKSIZE;


    for (i = 0; i < BLOCKSIZE; i ++) {
    #pragma HLS PIPELINE
      acc += input[index + i];
    }
    output[bid] = acc.s0 + acc.s1;
}
```

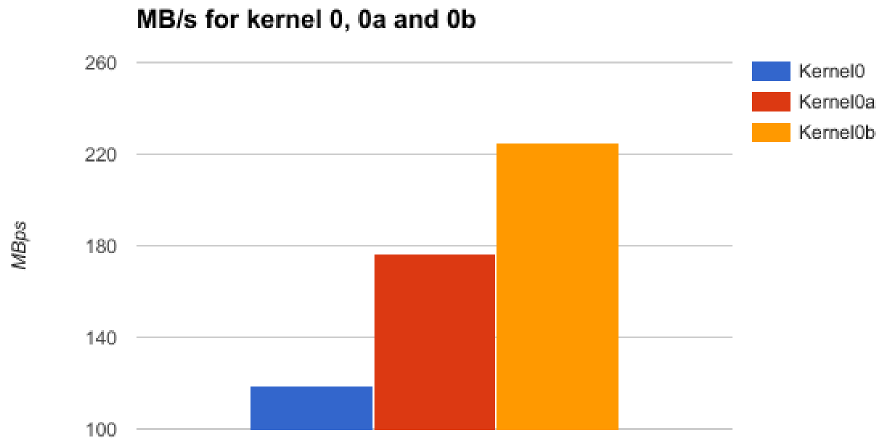

Figure 3

When writing this final kernel, `reduce0b`, different vector sizes was chosen but with no particular increase in performance over going to 2-wide vectors.

4 Conclusion

This section presents some thoughts on what possible reasons for the various performance characteristics of the reduction kernels.

4.1 The NVIDIA Reduction Tutorial Experiment

Kernels 1 - 6 were all OpenCL adaptations of CUDA code written for a GPU. All of these kernels loaded data into local memory before starting to sum the values up. The storing of data locally does only make sense if there is a way to hide the latency this loading. On an NVIDIA GPU groups of 32 threads (a CUDA Warp, OpenCL Wavefront) execute together as unit of work. The GPU has a scheduler that schedules these groups of 32 threads individually. If threads in one these groups hit a load instruction, the group can be swapped out and another take its place and thus hide the latency of the memory read.

That the reduction kernels that load data into local memory behave so badly on the ZYNQ seems to indicate that the synthesized hardware has no means to hide memory latencies by swapping other work. In other words the wavefront (warp) size is very likely as large as the workgroup in the generated hardware.

4.2 The Degenerate Sequential Kernels

The degenerate kernels are run as a workgroup (block, for CUDA programmers) of size one, containing a sequential loop that performs the reduction. The degenerate kernels are also different from the more idiomatic OpenCL kernels by not using any local memory. These kernels load a value and directly accumulate the sum in a register.

If, as hypothesised above, the generated OpenCL hardware has no means to hide the memory load latency, these degenerate kernels have an advantage. As soon as an element is loaded it can be added to the accumulator and we are done with that element. The earlier kernels would have two distinct phases. One phase for loading all data into local memory and another phase for computing the sum, both phases being O(N) work.

Making changes to the degenerate kernel (unrolling, pipelining and vectorizing) also had big impact compared to changes done to the idiomatic OpenCL kernels. This makes sense given that we do not have latency hiding capability and that we read all data through a single 64bit interface. The 64bit High performance AXI interface allows us to read one 64bit quantity each clock cycle (theoretically). This setup favors an approach that can be pipelined. Going to a vector size of 2x32bit quantities also makes sense given the 64bit AXI interface.

4.3 Architectural Differences

The comparison presented in this document is not really between a GPU and an FPGA. It would be more fair to say that the comparison is between a GPU and one out of all the possible ways to generate hardware for an OpenCL kernel for execution on an FPGA. So far we have seen that it is likely the case that:

- Hardware generated for an OpenCL kernel has no ability to hide memory read latencies.
- Hardware generated for an OpenCL kernel uses a single 64 or 32bit interface for all its reads and writes. Thus favoring pipelining over massive data parallelism.

For a GPU, the ability to hide memory read latencies is key to performance. GPUs also have wider memory interfaces and are tailored to data parallel mode operation.

GPUs also have resources for simultaneous execution of multiple workgroups (CUDA Blocks), essentially a indexed instance of a kernel, in parallel. On the FPGA (using vivado hls) this can be configured by the programmer. The single hardware unit generated for the kernel can be instantiated multiple times and hooked up to different memory interfaces giving workgroup (block) level parallelism.

5 Future Work

This document shows that if you approach the ZYNQ with the mind set of GPU programmer, you will have poor performance. The document, however, does not go deeply into figuring out how exactly to write your code for obtaining good performance on the ZYNQ. If we say that reaching 70% of the theoretical bandwidth is good (as with the G80 values presented), then on the ZYNQ platform used here a throughput of approximately 1.5GB/s is needed. Achieving this level of performance is not possible using a single high performance AXI interface that maxes out at 800MB/s. Either hardware units with more than one interface is needed or several single-interface hardware units has to be used in parallel[4]. It is left as future work to explore exactly how to implement high performance reduction on the ZYNQ using either C based or OpenCL based high level synthesis.

The conclusions reached in section 4 are guesses based on what one can see when measuring the performance. It is left as future work to figure out exactly what the generated hardware looks like when using OpenCL based high level synthesis. It is also interesting to see if there is any way to achieve asynchronous memory copy operations (there is for example an async_work_group_copy in OpenCL).

6 Kernel Code

Kernel 2

```
1  __kernel void __attribute((reqd_work_group_size(BLOCKSIZE,1,1)))
2    reduce2(__global int *input, __global int* output) {
3
4    __local int sdata [BLOCKSIZE];
5
6    int lid = get_local_id(0);
7    int bid = get_group_id(0);
8    int i   = bid * BLOCKSIZE + lid;
9
10   sdata[lid] = input[i];
11   barrier(CLK_LOCAL_MEM_FENCE);
12
13   for (int s = 1; s < BLOCKSIZE; s*=2) {
14     int index = 2 * s * lid;
15
16     if (index < BLOCKSIZE) {
17       sdata[index] += sdata[index + s];
18     }
19     barrier(CLK_LOCAL_MEM_FENCE);
20   }
21   if (lid == 0) output[bid] = sdata[0];
22
```

Kernel 3

```
__kernel void __attribute((reqd_work_group_size(BLOCKSIZE,1,1)))
  reduce3(__global int *input, __global int* output) {

  __local int sdata [BLOCKSIZE];

  int lid = get_local_id(0);
  int bid = get_group_id(0);
  int i   = bid * BLOCKSIZE + lid;

  sdata[lid] = input[i];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (int s = BLOCKSIZE / 2; s > 0; s >>=1) {
    if (lid < s) {
      sdata[lid] += sdata[lid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (lid == 0) output[bid] = sdata[0];
```

Kernel 4

```
__kernel void __attribute((reqd_work_group_size(HALF_BLOCKSIZE,1,1)))
  reduce4(__global int *input, __global int* output) {

  __local int sdata [HALF_BLOCKSIZE];

  int lid = get_local_id(0);
  int bid = get_group_id(0);
  int i   = bid * BLOCKSIZE + lid;

  sdata[lid] = input[i] + input[i + HALF_BLOCKSIZE];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (int s = HALF_BLOCKSIZE; s > 0; s >>=1) {
    if (lid < s) {
      sdata[lid] += sdata[lid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (lid == 0) output[bid] = sdata[0];
```

Kernel 5

```
__kernel void __attribute__ ((reqd_work_group_size(HALF_BLOCKSIZE,1,1)))
  reduce5(__global int *input, __global int *output) {

    __local int sdata[HALF_BLOCKSIZE];

    int lid = get_local_id(0);
    int bid = get_group_id(0);

    int i = bid * BLOCKSIZE + lid;

    sdata[lid] = input[i] + input[i + HALF_BLOCKSIZE];

    barrier(CLK_LOCAL_MEM_FENCE);

    if(lid < 64) { sdata[lid] += sdata[lid + 64];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 32) { sdata[lid] += sdata[lid + 32];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 16) { sdata[lid] += sdata[lid + 16];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 8) { sdata[lid] += sdata[lid + 8];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 4) { sdata[lid] += sdata[lid + 4];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 2) { sdata[lid] += sdata[lid + 2];} barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 1) {
      sdata[lid] += sdata[lid + 1];
      output[bid] = sdata[0];
    }
}
```

Kernel 6

```
__kernel void __attribute__ ((reqd_work_group_size(BLOCKSIZE,1,1)))
  reduce6(__global int *input, __global int *output, int n) {

        __local int sdata[BLOCKSIZE];

        int lid = get_local_id(0);
        int bid = get_group_id(0);
        int num_blocks = get_num_groups(0);
        int gridsize = BLOCKSIZE * 2 * num_blocks;
        int i = bid * (BLOCKSIZE * 2) + lid;

        sdata[lid] = 0;

        while (i < n) {
                sdata[lid] += input[i] + input[i + BLOCKSIZE];
                i += gridsize;
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        if(lid < 64) { sdata[lid] += sdata[lid + 64];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 32) { sdata[lid] += sdata[lid + 32];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 16) { sdata[lid] += sdata[lid + 16];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 8) { sdata[lid] += sdata[lid + 8];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 4) { sdata[lid] += sdata[lid + 4];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 2) { sdata[lid] += sdata[lid + 2];} barrier(CLK_LOCAL_MEM_FENCE);
        if(lid < 1) {
                sdata[lid] += sdata[lid + 1];
        output[bid] = sdata[0];
        }
}
```

References

[1] Mark Harris. Optimizing parallel reduction in cuda. http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[2] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The fifth international workshop on automatic performance tuning*, volume 66, 2010.

[3] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[4] Bo Joel Svensson. Exploring OpenCL Memory Throughput on the Zynq. Technical Report no: 2016:04, Chalmers University of Technology.