Hardware Accelerator

Pinar Ezgi Çöl & Darko Lukić

April 2019

Contents

1 Introduction **3**

2 Experiments **3**
2.1 Instruction Cache Disabled - Data Cache Disabled 3
2.2 Instruction Cache Disabled - Data Cache Enabled 5
2.3 Instruction Cache Enabled - Data Cache Enabled 6

3 Discussion **7**
3.1 Performance Measurement with and without Interrupts 7
3.2 Influence of Caches on C Swap Implementation 7
3.3 General Conclusion 7

4 Appendix **7**
4.1 Source Code 7
4.2 Function for Performance Measurements 8

1 Introduction

The aim of this laboratory is to perform Swap operation by using different agents. Swap operation is demonstrated in Fig. 1. It works as follows:

- Exchange the first byte with the last byte and the last byte with the first byte
- Take mirror of the middle 2 bytes

For the purpose of performing the Swap operation, we investigate the use of 3 different methods: Custom Instruction, Custom C Code and Custom Hardware Accelerator. We measure the time elapsed during the calculations of these methods with different cache and input configurations.

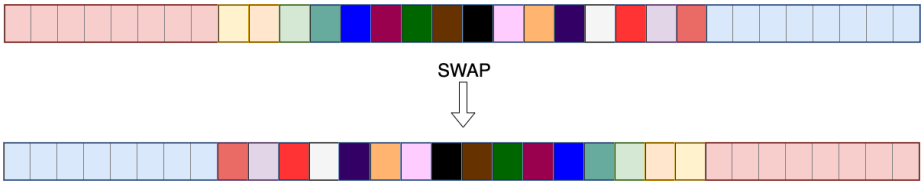

Figure 1: Swap Function

2 Experiments

In this section, we evaluate the performance of 3 different ways of performing swap operation. Different measurements are taken with various cache options and different batch sizes.

In real world application we very often read and use obtained results. Therefore, in order to simulate real world application we read all results and incorporated that time in our results.

Accelerator needs to be configured before it is started (which takes many cycles) and we expect to be slower than custom instruction for a single swap operation. Therefore, we varied batch sizes in range of 1 to 50K.

In order to obtain more precise performance measurements IRQ signal is added to SwapAccelerator. IRQ signal is raised as soon as the accelerator is done with processing. Additional commands are also added to support usage of the IRQ signal, e.g. *ClearFinish*.

2.1 Instruction Cache Disabled - Data Cache Disabled

First measurement is conducted by disabling both caches. According to the results we obtained, using hardware accelerator is not feasible when the input size is small. Custom instruction is the fastest in small number of inputs. However, the custom hardware accelerator outperforms both custom instruction and custom C code as the input size increases.

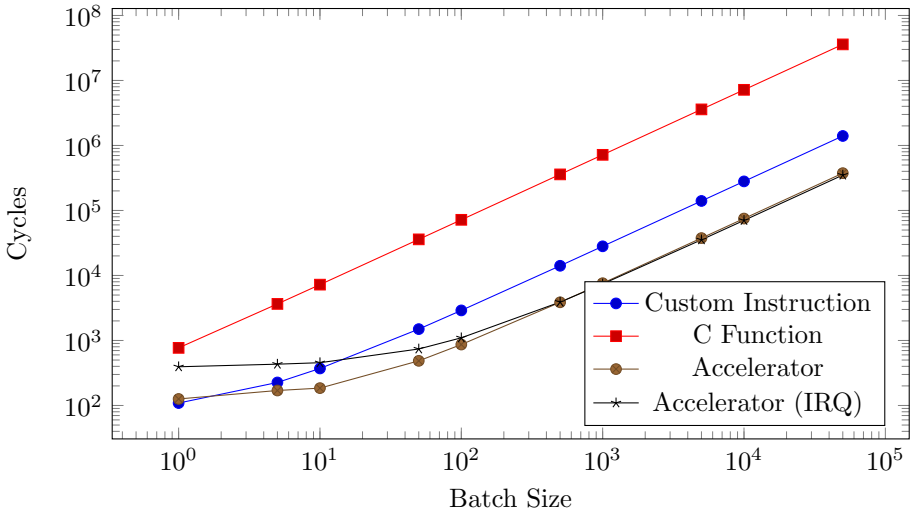

Figure 2: Number of cycles needed to calculate required swap operations on different batch sizes. The SoC is configured to use both, data and instruction cache.

The results are visualised in Fig. 1, but keep in mind that both axis have logarithmic scale. The logarithmic scale is chosen to better compare results of small and big batch size. Therefore, even though distance between e.g. custom instruction and accelerator at 50K inputs is a small in the plot, accelerator is actually about 100 times faster.

# of inputs	Instruction	C Code	Accelerator	Accelerator (IRQ)
50 000	1 400 096	35 900 044	375 106	350 399
10 000	280 096	7 180 044	75 106	70 402
5 000	140 096	3 590 044	37 606	35 399
1 000	28 096	718 044	7 606	7 399
500	14 096	359 044	3 866	3 895
100	2 901	71 840	866	1 099
50	1 496	35 944	486	740
10	371	7 229	185	455
5	226	3 651	170	432
1	109	770	126	394

Table 1: Measurements with Both Caches Disabled

More detailed results are represented in Tab. 1. From the table we can precisely measure a difference between accelerator and custom instruction performing the same number of swaps.

2.2 Instruction Cache Disabled - Data Cache Enabled

In this experiment we have enabled data cache. We observed the same trend as in the previous section (where data cache was disabled), for small batches custom instruction faster whereas accelerator achieves better performance with bigger batch sizes.

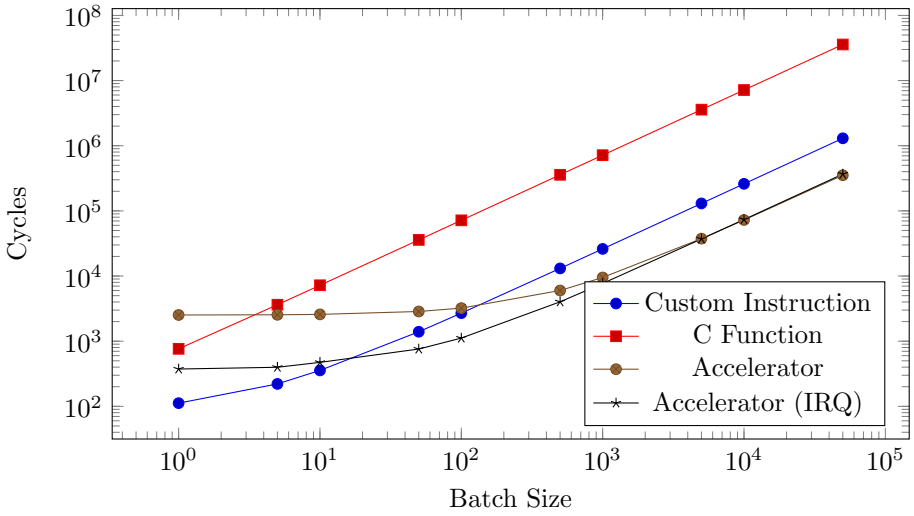

Figure 3: Number of cycles needed to calculate required swap operations on different batch sizes. The SoC is configured to use data caches, but not instruction cache.

# of inputs	Instruction	C Code	Accelerator	Accelerator (IRQ)
50 000	1 300 096	35 800 052	352 519	367 029
10 000	260 096	7 160 056	72 518	73 702
5 000	130 096	3 580 056	37 511	37 029
1 000	26 096	716 056	9 502	7 702
500	13 096	358 052	5 999	4 021
100	2 696	71 652	3 218	1 116
50	1 396	35 856	2 857	758
10	356	7 216	2 579	473
5	221	3 628	2 542	398
1	112	764	2 525	374

Table 2: Measurements with Data Cache Enabled - Instruction Cache Disabled

2.3 Instruction Cache Enabled - Data Cache Enabled

In the end we enabled both caches, data and instruction cache.

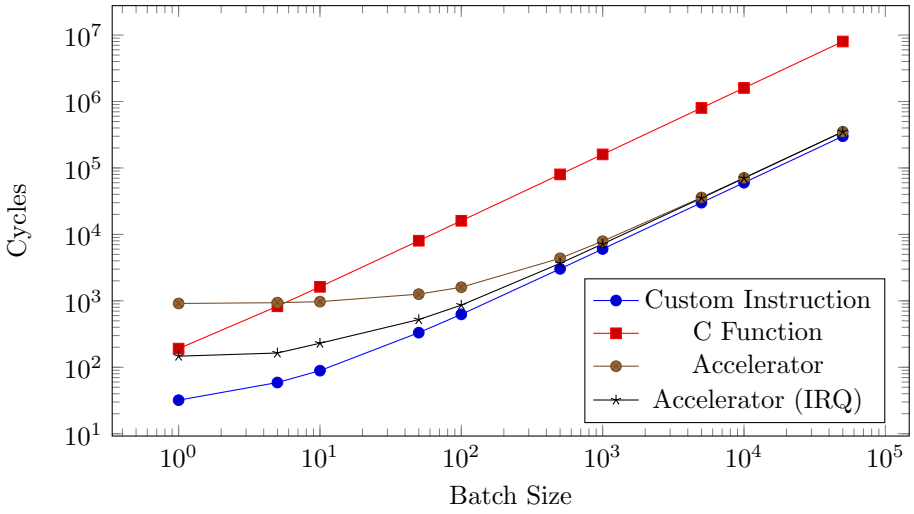

Figure 4: Number of cycles needed to calculate required swap operations on different batch sizes. The SoC is configured to use both, data and instruction cache.

# of inputs	Instruction	C Code	Accelerator	Accelerator (IRQ)
50 000	300 035	8 000 034	350 907	350 245
10 000	60 029	1 600 034	70 902	70 180
5 000	30 029	800 031	35 902	35 177
1 000	6 029	160 029	7 896	7 180
500	3 029	80 028	4 373	3 664
100	626	16 034	1 602	862
50	332	8 032	1 261	522
10	89	1 626	972	230
5	59	829	938	164
1	32	191	913	147

Table 3: Measurements with Data Cache Enabled - Instruction Cache Enabled

3 Discussion

3.1 Performance Measurement with and without Interrupts

Accelerator which uses interrupt to notify when the calculations are done supposed to be faster. The behaviour is nicely observed in SoC without data and instruction caches. However, in specific configuration it is not the case:

- SoC with data and instruction caches applied on small batch sizes (Tab. 3) and
- SoC with data cache applied on big batch sizes (Tab. 2).

Unfortunately, the irregularities are not further investigated but overloaded Avalon bus may be the cause.

3.2 Influence of Caches on C Swap Implementation

It was interesting to observe influence of caches on C swap implementation. Since the C implementation consist of many instructions it is obvious it would be slower than custom instruction. However, if we compare results from Tab. 3 and Tab. 2 we can see a significant performance improvement after instruction cache is enabled.

On the other hand, there is no significant difference between the C swap implementation running on SoC with and without data cache. The reason lies in implementation of the C function which doesn't access the memory, therefore data cache is not needed.

3.3 General Conclusion

In this exercise we learned how to implemented custom instruction and accelerator. Also, we measured performance improvements and depending on application we know whether to use accelerator or a custom instruction.

Since we measured performance with different cache types, we have discovered the impact of an instruction/data cache on performance. More precisely, we have discovered how much acceleration for custom instruction, C implementation and custom accelerator we can expect by using instruction/data cache.

By adding interrupt to the custom accelerator lower number of cycles is measured by a performance timer. The measured number of cycles is closer to actual number cycles taken by the custom accelerator to perform the operations.

4 Appendix

4.1 Source Code

`https://gitlab.com/lukicdarkoo/hardware-acceleration`

4.2 Function for Performance Measurements

```
void csv_export() {
  unsigned int time_custom, time_c, time_accelerator;
  int sample_batches[] = { 50000, 10000, 5000, 1000, 500, 100, 50, 10, 5, 1 };

  printf("BatchSize,Custom,CFunction,Accelerator\n");
  for (int ni = 0; ni < sizeof(sample_batches) / 4; ni++) {
    int n_samples = sample_batches[ni];

    // Custom shift
    TIC
    for (int i = 0; i < n_samples; i++) {
      volatile int res = ALT_CI_SWAP_0(i, 0);
    }
    TOC
    time_custom = perf_get_total_time(PERFORMANCE_COUNTER_0_BASE);

    // Accelerator
    volatile int buffer[n_samples];
    TIC
    accelerated_shift(buffer, n_samples);
    #if CONFIG_IRQ_ACC
      while (acclerator_done == 0);
      acclerator_done = 0;
    #else
      while ((IORD_32DIRECT(SWAP_ACCELERATOR_0_BASE, 8) & 0x0002) == 0);
      TOC
    #endif
    alt_dcache_flush_all();
    IOWR_32DIRECT(SWAP_ACCELERATOR_0_BASE, 12, 2);  // Clear `Finish` flag
    while (IORD_32DIRECT(SWAP_ACCELERATOR_0_BASE, 8) & 0x02 == 0x02);
    IOWR_32DIRECT(SWAP_ACCELERATOR_0_BASE, 12, 1);
    time_accelerator = perf_get_total_time(PERFORMANCE_COUNTER_0_BASE);

    // C shift
    TIC
    for (int i = 0; i < n_samples; i++) {
      volatile int res = shift(i);
    }
    TOC
    time_c = perf_get_total_time(PERFORMANCE_COUNTER_0_BASE);

    printf("%d,%d,%d,%d\n", n_samples, time_custom, time_c, time_accelerator);
  }
}
```