Implementation of heapsort in programmable logic with High-Level Synthesis

Wojciech M. Zabołotny[a]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

The paper presents the implementation of the streaming heapsort algorithm using the High-Level Synthesis (HLS). Results of synthesis in different configurations are compared with the reference implementation based on heavily optimized HDL code. Obtained results are used for evaluation of High-Level Synthesis as a method of implementation of data processing algorithms.

Keywords: HLS, FPGA, Heapsort, Sorting in hardware, VHDL

1. INTRODUCTION

Implementation of signal or data processing algorithms in FPGA is a difficult task. To fully utilize the potential of the hardware, it is usually necessary to decompose the problem into smaller tasks, that may be executed in parallel. To minimize the storage requirements and to maximize the throughput, it is desired that different processing subsystems are implemented in pipelined architectures, and further proper synchronization of results may be a time-consuming and error-prone job.[1] Therefore, implementing algorithms in FPGAs is considered to be a more demanding task, than implementing them in software.

The High-Level Synthesis is a method of automated translation of the algorithms written in common programming languages, like C or C++ into the HDL description ready for FPGA implementation. If successful, that methodology offers much faster and simpler testing of algorithms in C programs. The aim of this paper is to check the applicability of the High-Level Synthesis offered by Xilinx in the Vivado suite[2,3] for conversion of data-processing algorithms into HDL implementation. As the test case, the streaming heapsort algorithm was selected, because of its simple description in C language and availability of open source highly optimized and parametrized HDL version.[4,5]

2. STREAMING HEAPSORT ALGORITHM

In data acquisition systems, it often happens that the timestamped data arriving from different sources are not perfectly time-sorted. The data concentrating node is then responsible for time-sorting of such stream. Let us consider the data stream consisting of a series of records. Let us denote the timestamp of the k^{th} record as t_k. The stream of length K is sorted if $\forall n \in \mathbb{Z} \cap [1, K-1] : t_{k+1} \geq t_k$. That definition creates certain problems for the streams of arbitrary length. The timestamp is always stored in a finite number of bits. Therefore for long streams, the timestamp periodically wraps around. It forces us to use the special “local” definitions of the comparison function. If the timestamp is encoded in N bits, then it may store values between 0 and $2^N - 1$, and to calculate the differences between the timestamps, we must use the modular arithmetic with modulus 2^N. Then we may define our own "greater than" and "less than" operators:

$$a > b \pmod{2^N} \overset{\text{def}}{\iff} a - b \pmod{2^N} < 2^{N-1} \tag{1}$$

$$a < b \pmod{2^N} \overset{\text{def}}{\iff} a - b \pmod{2^N} \leq 2^{N-1} \tag{2}$$

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 6693

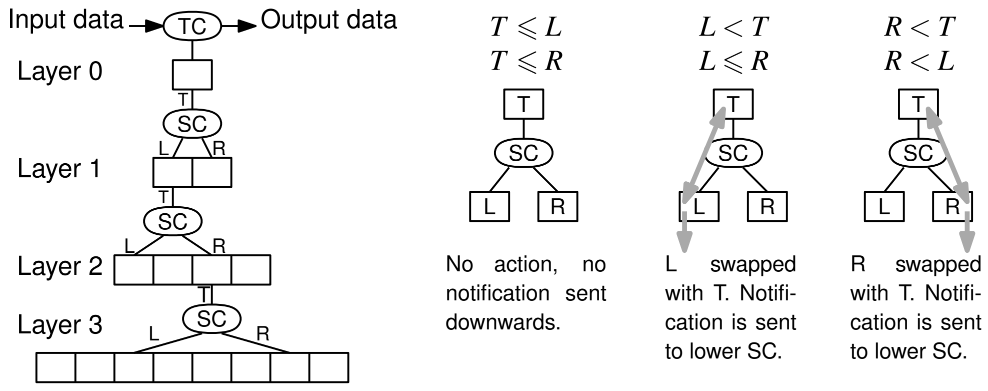

Figure 1: Implementation of the streaming heap sorter.[4] The top node controller (TC) compares the input data with the data at the top of the heap. If the input data is "older", it is transferred directly to the output. Otherwise, it replaces the data at the top. Each sorter node controller (SC) is notified if its top data (T) is changed. If the new T data is "newer" than L or R data, those data are swapped, and the next SC is notified of the change.

The disorder in the stream may be characterized with the "maximum distance between unsorted data records", defined as follows:[4]

$$D_{max} \overset{\text{def}}{=} \max\{d \in \mathbb{Z} \cap [1, K-1] : \exists k \in \mathbb{Z} \cap [1, K-d] : t_{k+d} < t_k\} \tag{3}$$

To sort the stream with certain D_{max}, the sorter must remember at least D_{max} previous records, because that's the maximum number of "later" records that may precede the "delayed" newly received record in the unsorted data stream. The heapsort algorithm in the streaming version is optimal in that sense that the heap sorter with the storage capacity of M words is able to sort the stream with $D_{max} = M$. The heapsort algorithm may also be efficiently parallelized. The article [4] presents the concept and the implementation of heapsort in VHDL, where the new data record may be handled every two clock periods[*]. A simplified version of that sorter[†] was created and used as a reference for the results produced by the HLS-based approach.

3. TEST PLATFORM

The tests were performed using Vivado 2017.4 and Vivado HLS 2017.4. As the target platform, the Zynq 7 chip XC7Z020-CLG400-1C was selected (e.g., used in the Z-Turn board[6]). The analysis was performed for the records containing 16-bit timestamps and 32-bit payload. It was assumed, that the sorter will contain 11 layers, and therefore it should be able to sort the streams with $D_{max} = 2^{12} - 1 = 4095$. However, the code is parametrized and may be easily modified for other formats of data.

4. IMPLEMENTATION OF HEAP SORTER WITH HLS

The initial simplest implementation was prepared for short streams, so timestamp (the "key" field in the "sort_data" structure) was coded as a signed short integer, and standard comparisons were used. The heap is implemented in a single vector, where layer N occupies the locations between 2^N and $2^{N+1} - 1$. The code implementing that version of heapsort is shown in Figure 2.

Unfortunately, HLS synthesis of that code gives very poor results. Both the "initialization interval" (II - the number of clock periods between the new records) and the latency (the number of clock periods between reception of data and availability of results) varies between 4 and 62 clock periods (see Table 1, row 1). That is caused by the fact, that HLS by default uses the "DATAFLOW" implementation, where the processing time may depend on data. For the pipelined systems another version of the algorithm, with constant initialization interval and constant latency is needed. Additionally, the input and output interfaces should be defined so that the records are transmitted as bit vectors with minimalistic handshake

[*]The sources of that sorter are available at [5].

[†]The special flags "init" and "valid" were removed from the data records.

```
>>>> wz_hsort.h <<<<

typedef struct {
  short int key;
  char payload[4];
} sort_data;

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*10)
extern sort_data sort_mem[SORT_LEN];
sort_data heap_sort(sort_data val);

>>>> wz_hsort.cc <<<<

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data sort_mem[SORT_LEN];
sort_data heap_sort(sort_data val)
{
  sort_data res;
  sort_data cur = val;
  int lev, offs, top, left, right;
  cur = sort_mem[1];
  if (val.key <= cur.key)
    return val;   //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    for (lev = 1; lev <= NM; lev++) {
      top = (1 << (lev - 1)) + offs;
      left = (2 << (lev - 1)) + offs;
      right = (3 << (lev - 1)) + offs;
      if ((lev == NM) ||
          ((cur.key <= sort_mem[left].key) &&
           (cur.key <= sort_mem[right].key))) {
        sort_mem[top] = cur;
        break;
      } else if ((sort_mem[left].key < cur.key) &&
                 (sort_mem[left].key <=
                  sort_mem[right].key)) {
        sort_mem[top] = sort_mem[left];
      } else if (sort_mem[right].key < cur.key) {
        sort_mem[top] = sort_mem[right];
        offs += (1 << (lev - 1));
      } else {
        printf("impossible!!!\n");
      }
    }
  }
  return res;
}
```

Figure 2: The simplest implementation of the heap sorter. The "for" loop implements the SC nodes from Figure 1. The "cur" variable transfers the address of the updated data. The timestamp is stored in the "key" field in the "sort_data" structure. To simplify the implementation, the timestamp is of "short signed integer" type, and standard comparisons are used. Such solution works correctly for short streams, where timestamp does not wrap.

signals. Therefore, the code must be supplemented with special HLS pragmas at the beginning of the heap_sort function body:

```
#pragma HLS PIPELINE II=2
#pragma HLS INTERFACE ap_hs port=val
#pragma HLS DATA_PACK variable=val
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS DATA_PACK variable=return
```

Such modification results in constant initialization interval, and constant latency, but their values are unacceptable: L=59, II=60 (see Table 1, row 2). The bottleneck is a single memory used to store the data records. To increase the data bandwidth, it is desirable to divide that memory into smaller independent memories for individual layers of the sorter. The required size of memory depends on the layer. The N^{th} layer needs to store 2^N data records. Unfortunately, HLS does not support such non-uniform partitioning of memories [7,8]. In VHDL the "for-generate" statement may be used to generate a set of memories with different sizes, and that approach was used in the implementation described in [4]. In the subset of C and C++ used by HLS, unfortunately, it is not possible to build such structure. Therefore it is necessary to declare the two-dimensional array with the number of rows equal to the number of layers and the number of columns equal to the size of memory required by the last layer[‡].

The resulting code is shown in Figure 3. That implementation resulted in improved II=12 (see Table 1, row 3), but it is still not satisfactory. The thorough analysis of the results, has shown that the loops were not properly parallelized ("unrolled"), even though the appropriate message was produced in the log file. The proper parallelization of loops requires that the number of iterations is constant. Therefore, leaving the loop with conditional "break" is not acceptable. To work it around, the special "enable" variable had to be added, that allows using the constant number of iterations, but disables any actions in iterations after the former break condition was true. The modified code is shown in Figure 4. The obtained II was equal to 4 (see Table 1, row 4), which is still higher than achieved in the reference HDL code.

[‡]The non-used memory location should be later on detected by the synthesis tools and eliminated.

```
>>>> wz_hsort.h <<<<

typedef struct {
  short int key;
  char payload[4];
} sort_data;

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*10)
extern sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val);

>>>> wz_hsort.cc <<<<

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val)
{
#pragma HLS PIPELINE II=2
#pragma HLS INTERFACE ap_hs port=val
#pragma HLS DATA_PACK variable=val
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS DATA_PACK variable=return
#pragma HLS ARRAY_PARTITION variable=sort_mem dim=1
  sort_data res;
  sort_data cur = val;
  int lev, offs, top, left, right;
  cur = sort_mem[0][0];
  if (val.key <= cur.key)
    return val;   //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    for (lev = 1; lev <= NM; lev++) {
      top = offs;
      left = offs;
      right = (1 << (lev - 1)) + offs;
      if ((lev == NM) ||
          ((cur.key <= sort_mem[lev][left].key) &&
           (cur.key <= sort_mem[lev][right].key))) {
        sort_mem[lev - 1][top] = cur;
        break;
      } else if ((sort_mem[lev][left].key < cur.key) &&
                 (sort_mem[lev][left].key <=
                  sort_mem[lev][right].key)) {
        sort_mem[lev - 1][top] = sort_mem[lev][left];
      } else if (sort_mem[lev][right].key < cur.key) {
        sort_mem[lev - 1][top] = sort_mem[lev][right];
        offs += (1 << (lev - 1));
      } else {
        printf("impossible!!!\n");
      }
    }
  }
  return res;
}
```

Figure 3: The HLS implementation of the heap sorter with partitioned memory. The two-dimensional array is split into parts used as memories for individual layers.

```
>>>> wz_hsort.h <<<<

typedef struct {
  short int key;
  char payload[4];
} sort_data;

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*10)
extern sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val);

>>>> wz_hsort.cc <<<<

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val)
{
#pragma HLS PIPELINE II=2
#pragma HLS INTERFACE ap_hs port=val
#pragma HLS DATA_PACK variable=val
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS DATA_PACK variable=return
#pragma HLS ARRAY_PARTITION variable=sort_mem dim=1
  sort_data res;
  sort_data cur = val;
  int lev, offs, top, left, right, enable;
  cur = sort_mem[0][0];
  if (val.key <= cur.key)
    return val;   //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    enable = 1;
    for (lev = 1; lev <= NM; lev++) {
      top = offs;
      left = offs;
      right = (1 << (lev - 1)) + offs;
      if (enable) {
        if ((lev == NM) ||
            ((cur.key <= sort_mem[lev][left].key) &&
             (cur.key <= sort_mem[lev][right].key))) {
          sort_mem[lev - 1][top] = cur;
          enable = 0;
        } else if ((sort_mem[lev][left].key < cur.key) &&
                   (sort_mem[lev][left].key <=
                    sort_mem[lev][right].key)) {
          sort_mem[lev - 1][top] = sort_mem[lev][left];
        } else if (sort_mem[lev][right].key < cur.key) {
          sort_mem[lev - 1][top] = sort_mem[lev][right];
          offs += (1 << (lev - 1));
        } else {
          printf("impossible!!!\n");
        }
      }
    }
  }
  return res;
}
```

Figure 4: The HLS implementation of the heap sorter with properly unrolled loops.

```
>>>> wz_hsort.h <<<<

typedef struct {
  short unsigned int key;
  char payload[4];
} sort_data;

int inline kcmp(short unsigned int v1,
                short unsigned int v2)
{
#pragma HLS LATENCY min=0 max=0
  if (v1 == v2)
    return 0;
  if ((v1 - v2) & (1 << 15))
    return -1;
  return 1;
}

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*100)
extern sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val);

>>>> wz_hsort.cc <<<<

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val)
{
#pragma HLS PIPELINE II=2
#pragma HLS INTERFACE ap_hs port=val
#pragma HLS DATA_PACK variable=val
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS DATA_PACK variable=return
#pragma HLS ARRAY_PARTITION variable=sort_mem dim=1
  sort_data res;
  sort_data cur = val;
  int lev, offs, top, left, right, enable;
  cur = sort_mem[0][0];
  if (kcmp(val.key, cur.key) <= 0)
    return val;   //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    enable = 1;
    for (lev = 1; lev <= NM; lev++) {
      top = offs;
      left = offs;
      right = (1 << (lev - 1)) + offs;
      if (enable) {
        if ((lev == NM) ||
            ((kcmp(cur.key, sort_mem[lev][left].key) <= 0)
             && (kcmp(cur.key, sort_mem[lev][right].key) <=
                 0))) {
          sort_mem[lev - 1][top] = cur;
          enable = 0;
        } else
            if ((kcmp(sort_mem[lev][left].key, cur.key) < 0)
                && (kcmp(sort_mem[lev][left].key,
                   sort_mem[lev][right].key) <= 0)) {
          sort_mem[lev - 1][top] = sort_mem[lev][left];
        } else
            if (kcmp(sort_mem[lev][right].key, cur.key) < 0) {
          sort_mem[lev - 1][top] = sort_mem[lev][right];
          offs += (1 << (lev - 1));
        } else {
          printf("impossible!!!\n");
        }
      }
    }
  }
  return res;
}
```

Figure 5: The HLS implementation of the heap sorter supporting continuous streams with timestamp implemented as "unsigned short integer", and the separate function for timestamp comparison.

The obtained results were good enough to introduce the support for continuous streams, as described in section 2. To achieve that, the type of the timestamp field ("key") was changed to "short unsigned int", and the appropriate comparison function was created (see Figure 5). Unfortunately, these simple changes resulted in significant increase of II - to 9 cycles (see Table 1, row 5). Even when the function was declared as inline (with "#pragma HLS INLINE") or having zero latency (with "#pragma HLS LATENCY min=0 max=0"), the II remained equal to 9.

The proper solution was found after multiple further tests and experiments. Replacement of the function with the two macros (the first one for "$<$" and the second one for "$\leqslant$") provided the required functionality without increasing of the initialization interval. The corrected code, working with II=3 clock periods (see Table 1, row 6) is shown in Figure 6.

It is interesting that HLS can achieve II=3 with a single memory used in each layer. The reference implementation uses two separate memories in each layer, to minimize the time needed to access "L" and "R" values. The same approach was tried in the HLS-based solution (see Figure 7), to check if further improvement of II may be achieved. Unfortunately, the above modification didn't reduce II to the desired level of 2 clock periods (see Table 1, row 7).

5. TESTS OF HLS-SYNTHESIZED DESIGNS IN SIMULATION

All HLS-synthesized designs were tested in simulations. Unfortunately, the original "RTL/C cosimulation" in Vivado HLS didn't work, but the produced VHDL RTL code was compatible with typical VHDL simulators. The dedicated testbench in VHDL was written that read the input data records from the file and wrote the output data to another file. The input data were generated with a Python script, and the output data were checked with another Python script to verify the correctness of the sorting. The simulations were performed with GHDL simulator,[9,10] and they have proven the correct operation of the generated sorters. In particular, it was verified, that sorting is correct for data streams with D_{max} below the sorter

```
>>>> wz_hsort.h <<<<

typedef struct {
  short unsigned int key;
  char payload[4];
} sort_data;

#define klt(v1,v2) ((v1-v2) & 0x8000)
#define kle(v1,v2) (((v2-v1) & 0x8000)==0)

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*100)
extern sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val);

>>>> wz_hsort.cc <<<<

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data sort_mem[NM][SORT_LEN];
sort_data heap_sort(sort_data val)
{
#pragma HLS PIPELINE II=2
#pragma HLS INTERFACE ap_hs port=val
#pragma HLS DATA_PACK variable=val
#pragma HLS INTERFACE ap_ctrl_hs port=return
#pragma HLS DATA_PACK variable=return
#pragma HLS ARRAY_PARTITION variable=sort_mem dim=1
#pragma HLS DATA_PACK variable=sort_mem
  sort_data res;
  sort_data cur = val;
  int lev, offs, top, left, right, enable;
  cur = sort_mem[0][0];
  if (kle(val.key, cur.key))
    return val;    //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    enable = 1;
    for (lev = 1; lev <= NM; lev++) {
      top = offs;
      left = offs;
      right = (1 << (lev - 1)) + offs;
      if (enable) {
        if ((lev == NM) ||
            (kle(cur.key, sort_mem[lev][left].key)
             && kle(cur.key, sort_mem[lev][right].key))) {
          sort_mem[lev - 1][top] = cur;
          enable = 0;
        } else if (klt(sort_mem[lev][left].key, cur.key)
                   && kle(sort_mem[lev][left].key,
                          sort_mem[lev][right].key)) {
          sort_mem[lev - 1][top] = sort_mem[lev][left];
        } else if (klt(sort_mem[lev][right].key, cur.key)) {
          sort_mem[lev - 1][top] = sort_mem[lev][right];
          offs += (1 << (lev - 1));
        } else {
          printf("impossible!!!\n");
        }
      }
    }
  }
  return res;
}
```

Figure 6: The HLS implementation of the heap sorter with macros used to implement comparisons in modular arithmetics.

Version	Description	Latency		Initialization Interval		Predicted resource usage		
		min	max	min	max	BRAM	FF	LUT
1	Initial version with “DATAFLOW” architecture	4	62	4	62	6	339	1105
2	Version with “PIPELINE” architecture	59	59	60	60	6	966	3145
3	Version with separate memory for each layer	15	15	12	12	66	881	4222
4	Version with properly unrolled loops	14	14	4	4	66	1285	4572
5	Initial version with support for continuous stream	44	44	9	9	68	1669	5048
6	Version with comparison implemented in macros	11	11	3	3	70	731	2562
7	Version with separate “L” and “R” memories	11	11	3	3	67	730	3504

Table 1: Results of HLS-synthesis of the streaming heap sorter. The reported resource usage is only the preliminary value predicted by the HLS tool.

```
>>>> wz_hsort.h <<<<

typedef struct {
  short unsigned int key;
  char payload[4];
} sort_data;

#define klt(v1,v2) ((v1-v2) & 0x8000)
#define kle(v1,v2) (((v2-v1) & 0x8000)==0)

#define NM 11
#define SORT_LEN (1<<NM)
#define MAX_DEL (SORT_LEN)
#define TEST_LEN (MAX_DEL*100)
extern sort_data l_smem[NM][SORT_LEN / 2];
extern sort_data r_smem[NM][SORT_LEN / 2];
sort_data heap_sort(sort_data);

>>>> wz_hsort.cc <<<<

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "wz_hsort.h"

sort_data l_smem[NM][SORT_LEN / 2] = { 0, };
sort_data r_smem[NM][SORT_LEN / 2] = { 0, };

sort_data heap_sort(sort_data val)
{
#pragma HLS INTERFACE ap_hs port=val
#pragma HSL INTERFACE ap_hs port=return
#pragma HLS DATA_PACK variable=val
#pragma HLS DATA_PACK variable=return
#pragma HLS DATA_PACK variable=l_smem
#pragma HLS DATA_PACK variable=r_smem
#pragma HLS PIPELINE II=2
#pragma HLS ARRAY_PARTITION variable=l_smem complete dim=1
#pragma HLS ARRAY_PARTITION variable=r_smem complete dim=1
#pragma HLS UNROLL
  sort_data res;
  sort_data cur = val;
  int tmp;
  int lev;
  int shift;
  int offs = 0;
  int enable;
  cur = r_smem[0][0];
  if (kle(val.key, cur.key))
    return val; //No need to update the heap
  else {
    res = cur;
    cur = val;
    offs = 0;
    enable = 1;
    for (lev = 1; lev <= NM; lev++) {
      shift = 0;
      if (lev >= 2)
        shift = 1 << (lev - 2);
      if (enable) {
        if ((lev == NM)
            || (kle(cur.key, l_smem[lev][offs].key)
                && kle(cur.key, r_smem[lev][offs].key))) {
          if (offs < shift)
            l_smem[lev - 1][offs] = cur;
          else
            r_smem[lev - 1][offs - shift] = cur;
          enable = 0;
        } else if (enable
                   && klt(l_smem[lev][offs].key, cur.key)
                   && kle(l_smem[lev][offs].key,
                          r_smem[lev][offs].key)) {
          if (offs < shift)
            l_smem[lev - 1][offs] = l_smem[lev][offs];
          else
            r_smem[lev - 1][offs - shift] =
                l_smem[lev][offs];
        } else if (klt(r_smem[lev][offs].key, cur.key)) {
          if (offs < shift)
            l_smem[lev - 1][offs] = r_smem[lev][offs];
          else
            r_smem[lev - 1][offs - shift] =
                r_smem[lev][offs];
          offs += (1 << (lev - 1));
        } else {
          printf("impossible!!!\n");
        }
      }
    }
  }
  return res;
}
```

Figure 7: The HLS implementation of the heap sorter with split memories.

Resource	Reference design with “bypass” channels (II=2)	Reference design without “bypass” channels (II=3)	HLS-generated sorter without split memories	HLS-generated sorter with split memories
Slice LUTs	4235 (7.96%)	3528 (6.63%)	1259 (2.37%)	1803 (3.39%)
Slice Registers	1420 (1.33%)	875 (0.82%)	649 (0.61%)	691 (0.65%)
RAMB36	20 (14.29%)	20 (14.29%)	30 (21.43%)	2 (1.43%)
RAMB18	20 (7.14%)	20 (7.14%)	2 (0.71%)	40 (14.29%)

Table 2: Results of synthesis of the streaming heap sorter basing on the handwritten VHDL implementation and the HDL code generated by Vivado HLS from the C/C++ source.

capacity and incorrect for D_{max} above the sorter capacity. The sorters capable of processing the continuous streams were successfully tested with streams of length significantly longer than the wrap-around period of the timestamp.

6. RESULTS OF THE FULL COMPILATION OF HLS-GENERATED SORTER

The last two versions of the HLS-synthesized sorter (rows 6 and 7 in Table 1) were compiled to the FPGA bitstream using the Vivado toolkit. The resource consumption in the XC7Z020-CLG400-1C FPGA is shown in Table 2. For comparison also two versions of the simplified reference design were compiled for the same FPGA. The first one uses a special “bypass” channels[4] to avoid conflicts at simultaneous access to the memories and allows to achieve II=2 (unfortunately, HLS was not able to generate such solution). The second one was compiled with disabled “bypass” channels and was similar with the HLS-generated version with separate memories. All compiled designs were able to work at the clock frequency of 80 MHz on the selected FPGA chip. It can be seen that the final resource consumption of the HLS-synthesized code is significantly smaller than the predicted values from Table 1. It also appears that the HLS-synthesized code uses even fewer resources than the handwritten VHDL code.

7. RESULTS AND CONCLUSIONS

The HLS methodology allowed implementing the streaming heap sorter with reasonable parameters. The source code in C/C++ describing the algorithm for the HLS-based implementation is much simpler than the handwritten VHDL source code. It is also easier to understand for somebody who has less experience with programming FPGAs.

On the other hand, however, the HDL code generated by the HLS is almost illegible for an engineer. Therefore, in case of problems, it is difficult to investigate possible bugs in the generated implementation. However, similarly, we could complain that the assembly code generated by the optimizing C/C++ compiler is difficult to analyze and compare with the original high-level code.

A significant advantage of the HLS is that it was able to automatically implement the solution with initialization interval equal to 3 clock periods without the manual splitting the memories into “left” and “right” parts, that was necessary in the reference solution. However, it must be emphasized, that the carefully optimized HDL code still provides better performance (II=2 instead of II=3) than the HLS-generated one. Attempts to describe the “bypass channels” used in [4] in an HLS-compatible way to decrease the II to 2 clock periods up to know were unsuccessful.

Another finding is that the HLS tools are very sensitive to the constructions used in the source code. Functionally equivalent solutions used for timestamp comparison resulted in the significant difference of performance (achievable II equal to either 9 or 3 clock periods). The HLS tools not always correctly report problems detected during the synthesis. The problem found in the tests was that certain situations disabling the required optimizations (usage of “break” inside the “for” loop) do not generate the appropriate warning.

HLS seems to be a promising technology for quick implementation of data processing algorithms in FPGA. In case of the heapsort algorithm, it produced solutions in certain aspects (resource usage) better than the handwritten VHDL code. However, efficient usage of the HLS still requires an engineer who understands possibilities and limitations of the FPGA technology. To achieve good results, it was necessary to discover the need of partitioning of the heap memory and to implement it reasonably.

Application of HLS to the problems that already have the HDL-coded solutions may help to discover new optimization possibilities. The results obtained in the paper show that the existing VHDL implementation of the heap sorter should be reviewed regarding its resource consumption.

ACKNOWLEDGMENTS

Work supported by statutory funds of Institute of Electronic Systems.

REFERENCES

[1] Zabołotny, W. M., "Automatic latency equalization in VHDL-implemented complex pipelined systems," *Proc.SPIE* **10031**, 10031 – 10031 – 12 (2016).

[2] Xilinx, "Vivado design suite user guide, high-level synthesis." https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html .

[3] Xilinx, "Vivado design suite tutorial," (2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug871-vivado-high-level-synthesis-tutorial.pdf .

[4] Zabołotny, W. M., "Dual port memory based Heapsort implementation for FPGA," *Proc.SPIE* **8008**, 8008 – 8008 – 9 (2011).

[5] Zabołotny, W. M., "Heap sorter for fpga," (2012). https://opencores.org/project/heap_sorter .

[6] MYIR Tech, "Z-turn board." http://www.myirtech.com/list.asp?id=502 .

[7] Xilinx, "Vivado HLS optimization methodology guide," (2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf .

[8] Xilinx, "Vivado design suite user guide, high-level synthesis," (2017). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug871-vivado-high-level-synthesis-tutorial.pdf .

[9] Gingold, T., "GHDL main/home page," (2017). http://ghdl.free.fr/ .

[10] Gingold, T. and Lehmann, P., "GHDL: VHDL 2008/93/87 simulator." https://github.com/ghdl/ghdl .