A Sorting Library for FPGA Implementation in OpenCL Programming

Ryohei Kobayashi†, Kento Miura†, Norihisa Fujita†, Taisuke Boku†, and Toshiyuki Amagasa†

*Center for Computational Sciences,
†Graduate School of Systems and Information Engineering,
University of Tsukuba, Japan

HEART 2021
International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies
Monday, 21 June, 2021
OpenCL programming model for FPGAs

- enabling implementation of FPGA applications without the HDL
  - Pros
    - hiding from the user the fundamental parts of FPGA implementation (PCIe, DDR, etc.)
      - All-in-one dev. environment
  - Cons
    - limited in their expressiveness
      - Programming is easy, but optimization is not
    - suffering from place and route problems that limit the maximum frequency

```c
__kernel void vecadd(__global float *a, __global float *b, __global float *c) {
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

```c
int main(int argc, char *argv[]) {
    init();
    clEnqueueWriteBuffer(...);
    clEnqueueNDRangeKernel(..., vecadd, ...);
    clEnqueueReadBuffer(...);
    display_result(...);
    return 0;
}
```
What we are aiming is...

- to optimize general-purpose computation kernels for practical application development for FPGAs
  and
- to make them available to users

**Related work: OpenCL implementation of matrix multiplication [2]**
- assuming that it could serve as a building block for algorithms that are built upon the base functionality of matrix multiplications.

### TABLE II

<table>
<thead>
<tr>
<th>#CUs</th>
<th>block size (d_1)</th>
<th>#DSPs (%) of avail.</th>
<th>#MK20s (%) of avail.</th>
<th>(f_{\text{max}}) [MHz]</th>
<th>effective GFLOPS</th>
<th>model GFLOPS</th>
<th>area efficiency</th>
<th>frequency efficiency</th>
<th>cycle efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1024</td>
<td>645 (14%)</td>
<td>6242 (70%)</td>
<td>363</td>
<td>344</td>
<td>349</td>
<td>0.115</td>
<td>0.955</td>
<td>0.925</td>
</tr>
<tr>
<td>2</td>
<td>720</td>
<td>1289 (29%)</td>
<td>6340 (71%)</td>
<td>354</td>
<td>648</td>
<td>665</td>
<td>0.229</td>
<td>0.931</td>
<td>0.894</td>
</tr>
<tr>
<td>3</td>
<td>512</td>
<td>1934 (43%)</td>
<td>4902 (55%)</td>
<td>319</td>
<td>863</td>
<td>869</td>
<td>0.344</td>
<td>0.839</td>
<td>0.880</td>
</tr>
<tr>
<td>4</td>
<td>512</td>
<td>2578 (58%)</td>
<td>6536 (73%)</td>
<td>315</td>
<td>1104</td>
<td>1144</td>
<td>0.458</td>
<td>0.820</td>
<td>0.856</td>
</tr>
<tr>
<td>5</td>
<td>512</td>
<td>3223 (72%)</td>
<td>8170 (91%)</td>
<td>294</td>
<td>1324</td>
<td>1335</td>
<td>0.573</td>
<td>0.774</td>
<td>0.880</td>
</tr>
<tr>
<td>6</td>
<td>360</td>
<td>3867 (86%)</td>
<td>5196 (58%)</td>
<td>224</td>
<td>1120</td>
<td>1167</td>
<td>0.688</td>
<td>0.589</td>
<td>0.814</td>
</tr>
</tbody>
</table>

Target kernel: sorting

● Sorting is everywhere
  ➢ basic arithmetic operation
    ✓ SNS
    ✓ Gene analysis
    ✓ Database
    ✓ Etc...

● Our proposal
  ➢ Providing a sorting library for FPGA implementation in OpenCL programming

```c
#include "fpga_sort.h"
__kernel void fpga_sort_test(
  __global uint *restrict tmp,
  __global uint *restrict src,
  const uint numdata,
  __global uint *restrict ret
) {
  // Do sort
  *ret = fpga_sort(tmp, src, numdata);
}
```

OpenCL kernel code with sorting function
Batcher’s Odd-Even Sorting Network

- Output sorted values in pipeline manner
  - two-stage pipelined CAE (Compare-and-exchange) units are used
    - critical path: one comparator
      - increasing the operating frequency
High-bandwidth Merge Sorter Tree [7]

- Merging sorted values with a throughput of a maximum of N data values

> In this figure, N is 8

A high-bandwidth merge sorter tree with 8 input ports (the number of input ports is parameterized)

[7]: M. Saitoh et al., “Very Massive Hardware Merge Sorter”, ICFP18, pp.86-93
Issue of High-bandwidth Merge Sorter Tree

- As the number of input ports of the tree is increased, the hardware resource usage increases linearly
  - the size of the tree that can be implemented in an FPGA is limited to a certain extent
  - the number of sorted data values in the output data stream generated by passing through the tree does not scale

Solution

connecting a merge sorter tree with wide input width to each input port of the high-bandwidth merge sorter tree
→ a merged sort tree that has both high throughput and wide input can be built
Building a merge sorter tree with wide input width

● Solution: Virtual Merge Sorter Tree [9]
  ➢ placing only one CAS (Compare-and-select) unit in each stage
  ➢ Integrating the FIFOs into a single buffer layer implemented in BlockRAM

Hardware-efficient implementation

[9]: K. Manev et al., “Large Utility Sorting on FPGAs”, ICFPT18, pp.334–337

● How it can be done?

In the traditional merge sorter tree, the CAS unit located at the root selects and outputs only one value on either side at a time.

Therefore, in each stage of the tree, only one CAS unit is active as well as one FIFO’s dequeue request and one FIFO’s enqueue request, which is highlighted in the figure.
Virtual Merge Sorter Tree [9]’s issue

- Operating frequency is too low
  - less than half that of the high-bandwidth merge sorter

  ![Sad Face] Not good...

  the performance of the high-bandwidth merge sorter tree cannot be maximized by simply connecting the virtual merge sorter tree

- Our solution: A multi-cycle virtual merge sorter tree
  - outputs N data values per N cycles (throughput is same as [9])
  - critical path: one comparator
    - increasing the operating frequency compared to [9]

[9]: K. Manev et al., “Large Utility Sorting on FPGAs”, ICFPT18, pp.334-337
Sorting Engine’s Behavior

- Example: sorting of 2,048 data values
  - the width of the sorting network, number of leaves in the virtual merge sorter tree, and number of input ports in the high-bandwidth merge sorter tree to be 8, 4, and 4

![Diagram of sorting engine](image-url)
Sorting Engine’s Behavior

merges chunks of sorted data values

FPGA

initial pass?

merges chunks generated by each virtual merge sorter tree

Virtual 0
merge sorter tree

Virtual 1
merge sorter tree

Virtual 2
merge sorter tree

Virtual 3
merge sorter tree

High-bandwidth merge sorter tree

Memory controller

Write buffer

External memory (DRAM)

read out sequentially

write back sequentially

2,048 unsorted data values
Sorting Engine’s Behavior

merges chunks of sorted data values again

FPGA

initial pass?

Sorting Network

Virtual 0; merge sorter tree

Virtual 1; merge sorter tree

Virtual 2; merge sorter tree

Virtual 3; merge sorter tree

High-bandwidth merge sorter tree

Write buffer

Memory controller

External memory (DRAM)

merged at 0

merged at 1

merged at 2

merged at 3

2,048 unsorted data values

128 sorted data values
Sorting Engine’s Behavior

FPGA

Sorting Network

Virtual merge sorter tree

Virtual merge sorter tree

Virtual merge sorter tree

Virtual merge sorter tree

High-bandwidth merge sorter tree

Memory controller

Write buffer

External memory (DRAM)

write back sequentially

2,048 unsorted data values

128 sorted data values

merges chunks generated by each virtual merge sorter tree again

initial pass?
Theoretical performance

● # of pass: \( [\log_{K \times E}(N/P)] \)
  - \( W \): the number of leaves in the virtual merge sorter tree
  - \( E \): the number of input ports in the high-bandwidth merge sorter
  - \( N \): the number of data values to be sorted
  - \( P \): the width of the sorting network

● Example: sorting 2,048 data values
  - \( W = 4, \ E = 4, \ N = 2048, \ P = 8 \)
    - \( \# \) of pass to be 2

● Throughput: \( E \times B \times F \) / (# of pass) [Byte / s ]
  - \( B \): data size of a data value
  - \( F \): Operating Frequency
Creating an OpenCL library for data sorting

```xml
<RTL_SPEC>
  <FUNCTION name="fpga_sort" module="fpga_sort">  
    <ATTRIBUTES>
      <ISSTALL_FREE value="no"/>
      <IS_FIXED_LATENCY value="no"/>
      <EXPECTED_LATENCY value="10"/>
      <CAPACITY value="1"/>
      <HAS_SIDE_EFFECTS value="yes"/>
      <ALLOW_MERGING value="no"/>
    </ATTRIBUTES>
  </FUNCTION>
  <PARAMETER name="MAXBURST_LOG" value="4"/>
  <PARAMETER name="WRITENUM_SIZE" value="5"/>
  <PARAMETER name="DRAM_ADDRSPACE" value="64"/>
  <PARAMETER name="DRAM_DATAWIDTH" value="512"/>
  <PARAMETER name="W_LOG" value="2"/>
  <PARAMETER name="P_LOG" value="3"/>
  <PARAMETER name="E_LOG" value="2"/>
</RTL_SPEC>
```

specifying sorting engine’s configuration

Overview of OpenCL library creation that contains the sorting function
#include "fpga_sort.h"

__kernel void fpga_sort_test(
    __global uint *restrict tmp,
    __global uint *restrict src,
    const uint numdata,
    __global uint *restrict ret)
{
    // Do sort
    *ret = fpga_sort(tmp, src, numdata);
}
### Evaluation testbed

**Pre-PACS version X (PPX)**

- working at Center for Computational Sciences, University of Tsukuba.

#### Hardware specification

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2690 v4 x2</td>
</tr>
<tr>
<td>Host Memory</td>
<td>DDR4-2400 8GB x8</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Tesla V100 (PCIe Gen3 x16)</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>32 GiB CoWoS HBM2 @ 900 GB/s with ECC</td>
</tr>
<tr>
<td>FPGA</td>
<td>BittWare 520N (Intel Stratix 10 1SG280HN2F43E2VG)</td>
</tr>
<tr>
<td>FPGA Memory</td>
<td>DDR4 2400MHz 32 GB (8GB × 4)</td>
</tr>
</tbody>
</table>

#### Software specification

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>CentOS 7.3</td>
</tr>
<tr>
<td>Host Compiler</td>
<td>Intel C++ Compiler 18.0.1</td>
</tr>
<tr>
<td>GPU Compiler</td>
<td>CUDA 9.2.148</td>
</tr>
<tr>
<td>OpenCL SDK</td>
<td>Intel FPGA SDK for OpenCL 19.4.0 Build 64 Pro Edition</td>
</tr>
</tbody>
</table>

A computation node of PPX
Performance comparison to OpenCL kernel for sorting

- For comparison, we restructured merge sort algorithm [12] for 2^29 data elements (data: 64 bit [32-bit key, 32-bit payload])

- Sorting engine’s configuration: W = 4, P = 8, E = 4

  Comparison result. data: 64 bit (32-bit key, 32-bit payload) three orders of magnitude greater

<table>
<thead>
<tr>
<th></th>
<th>ALMs (%)</th>
<th>Registers (%)</th>
<th>M20Ks (%)</th>
<th>DSPs (%)</th>
<th>fmax [MHz]</th>
<th>Throughput [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our sorting library</td>
<td>86,856</td>
<td>244,697</td>
<td>486</td>
<td>2</td>
<td>309.98</td>
<td>116</td>
</tr>
<tr>
<td>The merge sort [12]</td>
<td>35,455</td>
<td>73,736</td>
<td>231</td>
<td>0</td>
<td>283.60</td>
<td>0.26</td>
</tr>
</tbody>
</table>

- Operating frequency: 309.98 MHz
  - Critical path: a control logic for global memory interleave
  - can be removed by specifying -no-interleaving=DDR -global-ring at compilation
    → 378.21 MHz (380 MHz is the upper limit of the achievable operating freq.)

  the sorting performance of the library is further improved

Performance comparison between CPU and GPU

- **CPU**: OpenMP-versioned radix sort based on [14]
- **GPU**: Thrust library (CUDA 9.2.148)
- **Sorting engine’s configuration (FPGA)**: $W = 64$, $P = 32$, $E = 4$

[14]: Nadathur Satish et al., “Fast Sort on CPUs and GPUs: A case for bandwidth oblivious SIMD Sort” SIGMOD10, pp.351–362

Comparison of sorting performances based on data size
Data: 64 bit (32-bit key, 32-bit payload)
The ability to process small problem sizes

• necessary to achieve strong scaling for FPGA-centric applications
  • # of FPGAs is increased → problem size per FPGA becomes smaller

• sorting is required in nearly all algorithms, particularly in data intensive applications
  • statistics (percentile)
  • search (k-nearest neighbor query)
  • index construction (inverted index)
  • graph processing
  • etc...

Our study is worthwhile for developing a hardware logic to accelerate that fundamental process and enabling its ease of use as a library.
Conclusion

● A sorting library that can be used with the OpenCL programming model for FPGA
  ➢ which is built by combining the three hardware sorting algorithms.

● Our sorting library consumes more than twice the overall hardware resources compared to a merge sort restructured for the OpenCL programming model for FPGA, but its operating frequency is 1.09x higher and its sorting throughput is three orders of magnitude better.

● We also derived the performance model to estimate for data sorting and application developers can determine the optimal configuration of the sorting engine, taking into account the amount of target FPGA resources and the required sorting performance.