# Rapid Development of gzip with MaxJ

Nils Voss[1,2], Tobias Becker[1], Oskar Mencer[1], and Georgi Gaydadjiev[1,2]

[1] Maxeler Technologies Ltd.
[2] Imperial College London

**Abstract.** Design productivity is essential for high-performance application development involving accelerators. Low level hardware description languages such as Verilog and VHDL are widely used to design FPGA accelerators, however, they require significant expertise and considerable design efforts. Recent advances in high-level synthesis have brought forward tools that relieve the burden of FPGA application development but the achieved performance results can not approximate designs made using low-level languages. In this paper we compare different FPGA implementations of gzip. All of them implement the same system architecture using different languages. This allows us to compare Verilog, OpenCL and MaxJ design productivity. First, we illustrate several conceptional advantages of the MaxJ language and its platform over OpenCL. Next we show on the example of our gzip implementation how an engineer without previous MaxJ experience can quickly develop and optimize a real, complex application. The gzip design in MaxJ presented here took only one man-month to develop and achieved better performance than the related work created in Verilog and OpenCL.

## 1 Introduction

Gzip is a popular utility and widely used file format for lossless data compression. In this paper, we compare different implementations of the gzip compression on FPGAs using various languages. All implementations use very similar system architectures and are inspired by previous work by IBM [1].

This study provides an opportunity to show, how choices regarding the programming language offer distinct trade offs in productivity, performance and area utilization. This is of special interest, since FPGAs provide many possibilities to accelerate tasks while reducing energy consumption at the same time.

Designer productivity, and thereby development time, is a major cost factor in system design. While we acknowledge the challenges with accurately measuring productivity, especially in a comparable and quantified way, we still draw some claims on productivity advantages in the context of gzip development.

In recent years, different high-level synthesis tools emerged, in order to overcome the high complexity of hardware description languages such as VHDL and Verilog especially when targeting FPGAs. One of these tools provided by Altera is based on the OpenCL standard [2]. The programmer writes C-like code with additional OpenCL features to guide Altera's SDK in creating FPGA bitstreams.

A different approach are new languages for hardware description, which maintain the concepts known from high-level programming languages and thereby preserve their comfort while targeting hardware. One example is MaxJ by Maxeler and OpenSPL [3]. MaxJ is a Java based language with additional features and libraries to enable the rapid creation of FPGA designs.

To emphasize the OpenCL advantages Altera published the results of their gzip implementation [4] and compared them to results published by IBM. In this paper, an implementation of the same algorithm in MaxJ is presented and compared to related work in Verilog (IBM) and OpenCL (Altera).

The main contributions of this paper are:

- the analysis of various MaxJ advantages over OpenCL;
- a high-throughput gzip compression design;
- a productivity comparison of OpenCL, Verilog and MaxJ for gzip.

The paper is structured as follows. First in Section 2, we outline the background in high-level synthesis approaches, present MaxJ including its supporting ecosystem and give a short overview of OpenCL and Altera SDK. In Section 3 we briefly explain gzip, discuss existing gzip implementations and present the design considerations on implementing gzip on an FPGA. In Section 4 we study different implementation decisions and the differences between MaxJ and OpenCL. The performance of our design is compared against state-of-the-art implementations in Section 5. In Section 6 we examine the productivity advantages of the different languages and in Section 7 we draw our final conclusions.

## 2 Background - High-Level Design

FPGA designs are typically developed in low-level hardware description languages such as Verilog and VHDL. Designing in such languages can result in fast and efficient hardware implementations, but they require considerable skill and effort, which means that their productivity is low. There have been a wide range of approaches to raise the productivity of FPGA design. A typical approach to boost productivity is IP blocks reuse. Another possibility is to automatically generate FPGA designs from domain-specific tools such as Matlab Simulink or LabView but this is naturally limited to certain application types. It has also been proposed to increase productivity by using overlay architectures [5]. These provide a number of customisable templates that can be quickly used offering a compromise in efficiency, performance and development time.

Recently, various high-level synthesis tools have become available. These typically attempt to create FPGA designs from conventional programming languages, such as C, and often require some form of manual intervention in the transformation process.

*Vivado HLS* is a tool developed by Xilinx. It accepts C, C++ and System-C as inputs and supports arbitrary precision data types. Xilinx claims a 4x speed up in development time and a 0.7x to 1.2x improvement for the Quality of Result

compared to traditional RTL design [6]. Vivado HLS is not a push-button C-to-FPGA synthesis tool and requires various manual transformations to customise the hardware architecture and achieve well performing designs.

Additionally Xilinx provides *SDAccel* which is a programming environment for OpenCL, C and C++. Additionally to the compiler, it also provides a simulator and profiling tools. Xilinx claims to achieve up to 20% better results than with hand-coded RTL designs and 3x better performance and resource efficiency compared to OpenCL solutions by competitors. SDAccel also supports partial runtime reconfiguration of FPGA regions without halting the remaining accelerators running on the chip [7].

IBM's *Liquid Metal* supports data flow and map-reduce. The *Lime* language is Java based and supports CPUs as well as FPGAs and GPUs. The hardware type is chosen at runtime based on available capacities in the datacenter [8].

*Catapult C* creates FPGA and ASIC designs from ANSI C++ and SystemC descriptions [9]. Similar to other high-level synthesis tools, it requires the designer to perform iterations on the original C-code and manually tweak the hardware architecture in order to achieve a fast implementation.

*Chisel* is a Scala based hardware description language. Unlike other approaches focusing on synthesis from a C-like language, the concept behind Chisel is to add modern programming language features to a hardware description language. Design is still low level but the goal is to improve productivity by supporting high-level abstractions in the language [10].

The next section will explain the main advantages and differences of MaxJ.

### 2.1 MaxJ Development Ecosystem

MaxJ builds upon data-flow. A conventional processor reads and decodes instructions, loads the required data, performs operations on the data, and writes the result to a memory location. This iterative process requires complex control mechanisms that manage the basic operations of the processor.

In comparison, the data-flow execution model is greatly simplified. Data flows from memory into the chip where arithmetic units are organized in a graph structure reflecting the implemented algorithm.

In contrast to the majority of high-level synthesis tools, MaxJ is not generating hardware designs from control-flow oriented, and hence sequential, languages like C or C++. The programmer is expected to describe his/hers application as an inherently parallel data-flow graph structure in 2D space.

MaxJ is based on Java to benefit from its syntax while providing additional APIs for data-flow graph generation at scale. At build time the Java code creates the data-flow graph describing the hardware structure. This means that, for example, an *if-else* statement will be evaluated at build time to add either the nodes described in the *if* block or those in the *else* part to the data-flow graph and thereby to the hardware. This enables code fine tuning to different use cases and the creation of libraries covering many use-cases without overheads.

MaxCompiler translates MaxJ code into FPGA configurations. It also provides cycle accurate software simulation. In combination with Maxeler's Max-

elerOS and the SLiC library the simulation models or hardware configurations are tightly integrated into a CPU executable written in for example C, Fortran, Matlab or Python to allow rapid development of FPGA accelerated applications. The communication between FPGA and CPU is implemented using very high-level streaming primitives and there is no need for the user to worry about any of the low level details.

Maxeler's data-flow systems are built using its proprietary PCIe data-flow engines (DFEs). The MAX4 DFEs incorporate the largest Altera Stratix-V FPGAs as a reconfigurable computing substrate. This device is connected to a large capacity parallel DRAM (24-96 GB) to facilitate large in-memory datasets. Additionally DFEs for networking are available which offer additional connectivity via a maximum of three 40 GBits ports.

### 2.2 Altera OpenCL Compiler

OpenCL is a standard that aims at providing a single API to target different heterogenous computing platforms with a special focus on parallelization and allows a programmer to target different hardware platforms and instruction sets with the same code. While OpenCL does not guarantee optimal performance for the same code on all hardware platforms, it does guarantee correct functionality (if no vendor specific extensions are used) [11].

OpenCL uses a C-like syntax and provides many custom datatypes to enable easier access to SIMD instructions as well as additional syntax which takes the memory hierarchy used in modern hardware architectures into account. The workload can be distributed between multiple devices and is executed by processing elements on the available hardware. A scheduler distributes the computing tasks to the processing elements at runtime.

The first versions of OpenCL mainly targeted multicore CPUs, GPUs and DSPs but OpenCL can also be used for FPGA programming since Altera and Xilinx published their OpenCL SDKs for FPGAs [2, 4, 7].

The Altera OpenCL compiler supports the core OpenCL 1.0 features as well as extensions, which, for example, support streaming of data from an ethernet interface to a compute kernel. Altera OpenCL also provides an emulator for functional verification of the created designs in order to speed up the development time. In addition, a detailed optimization report and a profiler is provided to allow easier development of more efficient designs.

## 3 Gzip

Gzip is a utility [12] as well as a file format for lossless data compression [13]. For data compression DEFLATE [14] is used, which is a combination of Lempel-Ziv compression [15] and Huffman encoding [16].

The idea of the Lempel-Ziv compression algorithm is to replace multiple occurrences of equivalent byte sequences with a reference to the first sequence. This reference consists of a marker, showing that this data has to be interpreted

as an index, a match length, indicating how many bytes are equal, and an offset, defining the distance to the first occurrence of the byte sequence.

Huffman coding replaces all data in a symbol stream with code words. It is an entropy encoder, which means that frequently used words will require less bits. A Huffman code is a prefix code which guarantees that no code word is a prefix of any other codeword and, as a result, unambiguous encoding.

The gzip standard knows two different forms of Huffman codes. The simpler one is the static Huffman code which is defined in the standard itself [14]. A different option is to create a customized Huffman code based on the actual input data. The Huffman code itself then needs to be encoded as well to enable the decompressor to correctly decode the data. Therefore the compressed Huffman code description is placed before the actual compressed data in the data-stream. While often providing better compression ratio this method is more complex to implement and leads to extra calculations at runtime.

Since gzip is so widely used, there are many different implementations of it. Intel published a high throughput CPU implementation achieving a throughput of 0.34 GB/s [17]. There are also many high-throughput FPGA implementations like the already mentioned implementations by Altera [4] and IBM [1] which achieve throughputs between 2.8 and 3 GiB/s. A more recent FPGA based publication by Microsoft reports a throughput of 5.6 GB/s [18]. In addition, ASIC implementations of gzip exist with throughputs of up to 10GB/s [19].

### 3.1 Gzip FPGA Implementation

The majority of gzip FPGA implementations struggle to process more than one byte per cycle, which severely limits throughput [20, 21]. The problem is that the encoding of a symbol could also influence the encoding of the next one.

The approach used in this paper (the same as in [1] and [4]) enables processing of multiple byte per cycle using hash tables. In each cycle a fixed number of bytes is loaded and for each byte a hash key is computed. This hash key is usually based on the byte itself as well as a pre-defined number of following symbols.

These hash keys are used to address the hash tables. The hash tables store possible matches for a given hash value. There are as many hash tables as bytes read per cycle. So every computed hash key is used to update one of these tables. On the other side a parallel lookup is performed on all hash tables in order to find all possible matches. The whole process is depicted in Figure 1.

The hash tables are also used to store the already seen data. If $n$ bytes are read per cycle than $n$ bytes have to be stored for each symbol in the hash table. These $n$ bytes consist of the symbol itself followed by the next $n-1$ input bytes.

This avoids a large memory structure with many read ports holding all the previous data. Instead, only the data that can be referenced by the hash tables is stored. The disadvantage of this solution is that each symbol in the input window is stored $n$ times. The hash table memory requires a wide word width and $n$ read and one write ports, which strongly increases area usage.

In order to avoid the $O(n^2)$ memory usage complexity a different hash table architecture was proposed by Microsoft [18]. Instead of $n$ hash tables with $n$ read

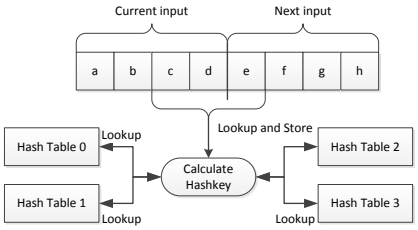



**Fig. 1.** Hash table implementation

ports they used a fixed number of hash tables with one read port each. The main idea is, that the possible hash keys are equally distributed onto different hash tables. Then if $m$ hash tables are created, the least significant $log2(m)$ bits are used to determine which hash table is used for each hash value. In order to be able to save different data items for the same hash value, each hash table can be copied. So in order to avoid hash conflicts a different copy of the hash table can be used. The hash tables run at double frequency compared to the remaining design which effectively doubles the number of read and write ports.

The biggest problem with this implementation is that for a given set of least significant bits only two writes can be accomplished in one cycle. All other matches, which hash keys have the same least significant bits, have to be dropped slightly reducing the compression ratio. With this optimizations and a few other small changes Microsoft was able to increase the throughput significantly with limited impact on the compression ratio.

Since Microsoft did not report design time we can not directly compare against their design process and will focus on those used by Altera and IBM.

The hash table lookup provides $n^2$ possible matches, since we perform $n$ lookups for each input byte. The first step is to perform the actual match search, which requires a comparison of the input data with the already processed data stored in the hash tables. The target is to find the longest match starting at each position in the input window, to allow encoding with as few bits as possible. In order to avoid complex inter-cycle dependencies the maximal match length is limited to the number of bytes read per cycle.

Since one byte may be covered by multiple matches, only a selection of all found matches has to be encoded. Decisions made here also impact the encoding in the next cycle, since a match might also cover symbols of the next input window. Since the design has to be fully pipelined, this inter cycle dependency has to be resolved within one cycle to prevent pipeline stalls.

If a match only covers a few symbols it might be cheaper to encode this as literals and not as a match. In this case the match will be ignored. A heuristic is applied on the remaining matches to resolve the inter-cycle dependencies.

This heuristics takes the match for the last symbol in the input window as the maximal match length into the input window of the next cycle. Since the maximal match length is $n$ the last symbol is never covered by a match in a previous input window and thereby we do not have to consider any other inter

cycle dependencies here. While this heuristic may decrease the compression ratio, it enables a fully pipelined design while limiting the design complexity.

In order to finally select the matches first all matches for symbols that were already covered by a match from the previous cycle are removed. Then the reach of each match is calculated, which is defined as the sum of the position of the current symbol and the match length. If two symbols have the same reach, they encode all symbols up to the same position and the match which covers more symbols in total is selected. In [4] a more detailed explanation is available.

At last, the data has to be encoded using Huffman coding. This can be done symbol-wise after the match selection. These code words then get combined using shifters and OR-gates to form the final output bitstream.

## 4 MaxJ Implementation Advantages

Our gzip implementation is similar to the implementation reported by Altera [4] to allow easier comparison between OpenCL and MaxJ implementations.

MaxJ custom datatypes offer a significant advantage. While C and OpenCL only support char (8 bit), short (16 bit), int (32 bit) and similar types, MaxJ allows programmers to define non-standard datatypes such as a 5 bit integer. Even for a byte-based algorithm like gzip many values do not need data types with power of 2 bit-widths. This applies for example for the Huffman code words, the match length, the match offset or the control signals.

The part of the architecture where the biggest number of similar modules exist is the match length calculation, since we have $n^2$ possible matches. The straight forward way of implementing this would be to byte-wise compare each byte of the input data with the data referenced by the lookup. As a result, if the bytes are equal and if all previous bytes were equal as well, the match length can be incremented as shown in Figure 2. So if we process 16 Bytes per cycle we have to use 16 comparators, adders and MUXs per match and in total 4096 units of each element. Hence, the resource usage has a complexity of $O(n^3)$.

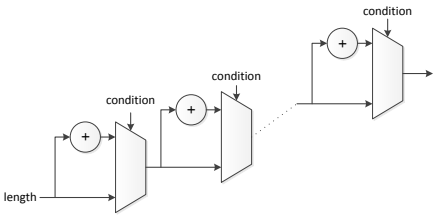



**Fig. 2.** Simple match length

Altera uses bit vectors instead so that for every similar byte a bit in the vector is set as shown in Listing 1.1 and Figure 3. The advantage is that OR operations and shifters cost less than ADDs and MUXs. It also enables the scheduler to

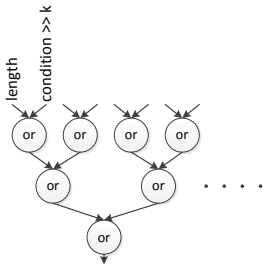



**Fig. 3.** Altera match length

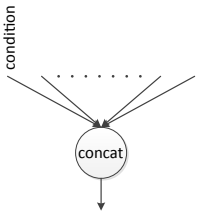



**Fig. 4.** MaxJ match length

use less FIFOs to implement this part of the algorithm, since all OR operations can be scheduled in the same clock cycle and there is no dependency between the different iterations of the unrolled loop. As a result the OR operations can be scheduled in a tree like fashion which reduces the number of required FIFOs. By using this technique a 5% reduction of logic resources is claimed.

```
1 // compare current/comparison windows
2 #pragma unroll
3 for (char k = 0; k < LEN; k++)
4 {
5   if (curr_window[j + k] == comp_window[k][i][j])
6     length_bool[i][j] |= 1 << k;
7 }
```

**Listing 1.1.** OpenCL implementation of match length calculation

Writing the same code in MaxJ would already reduce resources, since the shifts are omitted in hardware as the result of these operations would be computed at build time instead. This, as stated in [4], is not done by the OpenCL SDK.

Listing 1.2 shows an equivalent MaxJ implementation with some additional improvements. The # operator is used to concatenate bits. So in this case we concatenate all results of the comparators bit-by-bit, which does not use any additional resources. Also we do not need any registers or FIFOs because the concatenation has no latency at all. The only costs come from the comparators. The result of that is also shown in Figure 4.

```
1 // compare current/comparison windows
2 lengthBool[i][j] = currWindow[j] === compWindow[0][i][j];
3 for (int k = 1; k < LEN; k++) {
4   lengthBool[i][j] #= currWindow[j + k] === compWindow[k][i][j];
5 }
```

**Listing 1.2.** MaxJ implementation of match length calculation

Other MaxJ language features make it easier to meet timing. For example, the calculated hash keys are used at many different places and, as a result, have quite a large fanout. Since a huge chunk of the available memory resources on the FPGA are used for hash tables, the hash keys have to be routed to very distant locations of the chip. In order to compensate this and help meeting timing, an additional register was added after the hash key calculation as shown in Listing

1.3. The place and route tools can now duplicate this register, if needed, in order to distribute the signal to all hash tables, where it is used for addressing.

```
1 for (int i = 0; i < bytesPerCycle; i++) {
2   hashKey[i] = optimization.pipeline(calculateHashKey(currWindow, i));
3 }
```

**Listing 1.3.** Adding a Register to the hashKey signal, which is returned by the calculateHashKey() function. It is then passed into the optimization.pipeline() function to add the register.

On the FPGA platform used by Altera the input data gets transmitted over PCIe to DDR3 memory. The same principle applies to the encoded data which first is written into DDR3 memory before it is send back to the host via PCIe.

In the MaxJ design the data does not need to be buffered in external memory but can be send directly via PCIe to the FPGA where it is processed.

Since on-chip memory capacity is the limiting factor of the gzip design a different implementation of the Huffman encoding was used. Altera used a lookup table which can be changed by the CPU. In our design we calculate the Huffman code words on the fly and do not waste any on-chip memory.

This slightly limits the adaptability since only one fixed Huffman tree is available. This tree is optimized to all possible match lengths but could also be optimized for known payloads. While no big impact on compression ratio could be observed, this change is key in enabling our design to process 20 bytes per cycle. Both, IBM and Altera designs process only 16 Bytes per cycle.

## 5 Performance evaluation

We now compare the performance and area utilization of the different designs. The area utilization is compared in Table 1. First, we are going to only compare the 16 byte per cycle MaxJ design with the designs implemented by IBM and Altera, since all these designs process the same number of bytes per cycle. The MaxJ design uses significantly less resources as the OpenCL design. The area utilization numbers for the IBM design shown here were estimated and reported by Altera based on a chip image [4]. So while we can only work with estimations, we can still assume that the logic utilization of the MaxJ design in comparison to the Verilog design is at least on par. Only the RAM utilization is higher which is probably caused by the scheduling overhead of 443 pipeline stages in contrast to the 17 stages of the Verilog design. Despite the fact that the OpenCL design uses only 87 pipeline stages the MaxJ design uses fewer memory resources.

Throughput and compression ratio differences are depicted in Table 2. The compression ratio for all designs was evaluated using the calgary corpus [22] and the geometric mean. While the compression ratio of the Intel, IBM and Altera designs are almost identical, the MaxJ design shows a slight improvement. The reason for this is probably a different hashing function (as described in [23]) which improves the compression ratio at the cost of additional logic resources.

**Table 1.** Area utilization of the gzip compression on Stratix V FPGA

| | IBM (Verilog) [1] | Altera (OpenCL) [4] | MaxJ (16 Bytes) | MaxJ (20 Bytes) |
|---|---|---|---|---|
| Logic Utilization | 45% | 47% | 42.8% | 51.1% |
| RAM | 45% | 70% | 59.2% | 88.6% |

IBM implementation figures were estimated by Altera using a chip image [4]

**Table 2.** Compression Ratio and Throughput

| | Intel (i5 650 CPU) [17] | IBM (Verilog) [1] | Altera (OpenCL) [4] | MaxJ (16 Byte) | MaxJ (20 Byte) |
|---|---|---|---|---|---|
| Compress. Ratio | 2.18 | 2.17 | 2.17 | 2.25 | 2.27 |
| Throughput | 0.338 GB/s<br>0.315 GiB/s | 3.22 GB/s<br>3.00 GiB/s | 3.05 GB/s<br>2.84 GiB/s | 3.20 GB/s<br>2.98 GiB/s | 5.00 GB/s<br>4.66 GiB/s |

While IBM reported a frequency of "just under 200 MHz" [1], Altera claims a frequency of 193 MHz. Our MaxJ design for 16 Bytes successfully runs at 200 MHz without any optimizations aimed to help meeting timing.

When we use the available space to process 20 bytes per cycle instead of 16 and additionally perform timing optimizations, our design achieved a throughput of 5 GB/s at 250 MHz. This makes our design nearly 15x faster than Intel's high throughput CPU implementation and nearly 1.8x faster than the OpenCL implementation by Altera [1, 4].

## 6 Productivity discussion

In [4] Altera reported one month development time for their OpenCL gzip implementation. The MaxJ design presented here was performed by one intern student within a single month. The intern was novice to MaxJ and had only one week to work through the MaxJ tutorials. This clearly shows that learning MaxJ can be quick with a software development background in high-level languages.

An advantage of HLS in contrast to classical hardware description languages is, that the code is very readable and compact (the entire MaxJ gzip code is only 959 lines). This makes it easier to focus on optimizations and to make big changes in the architecture, since modern programming tools like unit tests can be used in combination with the simulator to quickly validate functionality. For example, the switch from the 16 byte per cycle design to 20 bytes was done by only changing a single constant in the code.

Because the MaxJ tools create deeply pipelined structures meeting timing is easier. While deep pipelining increases the overall memory usage it enables the designer to use more space of the chip productively.

As previously mentioned, Microsoft also reported an FPGA based gzip design using a slightly modified design architecture [18] achieving 5.6 GB/s on a Stratix V FPGA. We were able to also create a design using their architecture and again

achieve a higher throughput of 9.6 GB/s. Since we could reuse most of the already written MaxJ code, the actual implementation time went down to roughly one week. A few more weeks of not full-time effort were needed in order to fine-tune parameters like the used hash function and hash tables configuration as well as improve timing. It has to be noted that while meeting timing is time consuming it is not as costly as development time, since it mainly requires CPU time and not engineering effort.

When comparing to OpenCL, we can see that in a similar time far better results could be achieved with MaxJ. A reason for this is the more direct control over the hardware provided by MaxJ. This allows designers with good understanding of the underlying hardware to benefit from those additional improvements. For example, the option to directly insert registers in the design (as shown in Section 4) allows easier timing closure. Another good example is the direct impact that widths of the variables have on the hardware area utilization.

While it is possible to reuse existing OpenCL designs for CPUs and GPUs to target FPGAs it has to be noted, that the performance of the ported designs will be suboptimal in most cases. For example in [24] the same OpenCL code was executed on CPUs and FPGAs. The CPU versions all outperform the FPGA versions even though efficient hardware implementations for the tested algorithms exist. This shows that, similar to most other high-level synthesis frameworks (see Section 2), it is necessary to employ a series of code transformations in order to create efficient hardware designs. As a result a change of the programming language as well as the associated toolchain introduces only a limited overhead.

The above suggests that developing in MaxJ is significantly faster than in OpenCL since we had enough time to perform careful timing optimizations and compression ratio improvements. As a result this enabled us to deliver a significantly better bitstream in terms of throughput and compression ratio.

## 7 Conclusion

In this paper we presented a rapid FPGA implementation of gzip compression. We demonstrated that using MaxJ for high-level synthesis enabled us to achieve better results within the same amount of development time as compared to OpenCL. Furthermore, we showed that MaxJ and its development tools enable very competitive development times in comparison to classical hardware description approaches. Our design outperforms the OpenCL implementation by 1.8x in terms of throughput and delivers 5% better compression ratio by using only ~10% more resources. In addition, the presented design achieves a 1.7x higher throughput as compared to the Verilog implementation by IBM.

## References

1. A. Martin, D. Jamsek and K. Agarwal. FPGA-Based Application Acceleration: Case Study with GZIP Compression/Decompression Stream Engine. In *International Conference on Computer-Aided Design (ICCAD)*, Nov 2013.

2. Altera. OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity. http://www.altera.com/products/software/opencl/opencl-index.html, 2012.
3. OpenSPL. http://www.openspl.org, 2015.
4. M. S. Abdelfattah, A.Hagiescu and D. Singh. "Gzip on a Chip: High Performance Lossless Data Compression on FPGAs using OpenCL" in *International Workshop on OpenCL* ACM, pp. 1-9, 2014.
5. R. Rashid, J. G. Steffan, V. Betz. "Comparing Performance, Productivity and Scalability of the TILT Overlay Processor to OpenCL HLS". *Field-Programmable Technology (FPT).* IEEE, pp. 20-27, 2014.
6. Vivado HLS. `http://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf`, Nov. 18, 2015.
7. Xilinx. The Xilinx SDAccel Development Environment. http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf, 2014
8. Liquid Metal. www.research.ibm.com/liquidmetal/, 2015.
9. Catapult C. http://calypto.com/en/products/catapult/overview/, 2015.
10. J. Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". *Design Automation Conference (DAC).* ACM, pp. 1216–1225, 2012.
11. J. E. Stone, D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," in *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66-73, May-June 2010.
12. Gzip. http://www.gzip.org, 2015.
13. P. Deutsch. (1996) Gzip file format specification version 4.3. http://tools.ietf.org/html/rfc1952
14. P. Deutsch. (1996) RFC 1951 deflate compressed data format specification version1.3. http://tools.ietf.org/html/rfc1951
15. J. Ziv and A. Lempel. "A Universal Algorithm for Sequential Data Compression". *IEEE Transactions on Information Theory*, vol. 23, no.3, pp. 337-343, 1977.
16. D. A. Huffman. "A Method for the Construction of Minimum-Redundancy codes". *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, 1952.
17. V. Gopal, J. Guilford, W. Feghali, E. Ozturk and G. Wolrich. "High Performance DEFLATE Compression on Intel Architecture Processors". http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-deflate-compression-paper.pdf, 2011.
18. J. Fowers, J.-Y. Kim, D. Burger and S. Hauck. "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs". *The 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 52-59, 2015.
19. AHA 378. (2015). http://www.aha.com/data-compression/
20. W.-J. Huang, N. Saxena and E. J. McCluskey. "A reliable lz data compressor on reconfigurable coprocessors". in *Symposium on Field-Programmable Custom Computing Machines.* IEEE, pp. 249-258, 2000.
21. S.A. Hwang and C.-W. Wu. "Unified vlsi systolic array design for lz data compression". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 489-499, 2001.
22. Calgary Corpus. http://corpus.canterbury.ac.nz/descriptions/#calgary, 2015.
23. K. Sadakane , H. Imai. "Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting". *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences E83-A(12)*, pp. 2689 - 2698, 2000.
24. G. Ndu, J. Navaridas, and M. Lujan. 2015. CHO: towards a benchmark suite for OpenCL FPGA accelerators. In Proceedings of the 3rd International Workshop on OpenCL (IWOCL '15). ACM, New York, NY, USA, Article 10.