High Throughput Parallel Computation with High Bandwidth Memory on FPGA

Joep C. Dumont

MSc Thesis
Embedded Systems
Quantum & Computer Engineering
Q&CE-CE-MS-2020-14

TU Delft

High Throughput Parallel Computation with High Bandwidth Memory on FPGA

by

Joep C. Dumont

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday February 25, 2021 at 13:00 PM.

Student number:	4314271	
Project duration:	March 2020 – February 2021	
Thesis committee:	Dr. Ir. Z. Al-Ars,	TU Delft, supervisor
	Prof. Dr. H. P. Hofstee,	TU Delft
	Dr. Ir. J. Rellermeyer,	TU Delft
	Dr. Ir. J. Hoozemans,	TU Delft

TU Delft

Abstract

With the increase of available storage bandwidth, CPUs can not keep up with the compute throughput needed to process this amount of incoming data. GPUs and FPGAs are generally better suited for such tasks. To assist FPGAs in their functions, some boards are equipped with one or more high bandwidth memory (HBM) stacks, with a bandwidth of 230 GB/s each. This thesis presents a hardware design for the Alveo U280 FPGA board with HBM. Each HBM stack provides multiple interfaces to the full range of memory within HBM. Utilizing these multiple interfaces, a hardware decompressor for the Snappy compression algorithm is placed in parallel to achieve a higher end-to-end throughput. Additionally a design is created to perform benchmarks on HBM where varying sizes of data are transported between HBM and logic within the FPGA.

The hardware decompressor and component that interfaces to memory within HBM were found to be incompatible and required additional logic to become able to transport data between them. To ease the parallelization of the decompressor a custom Snappy framing format is implemented. Using this format a softcore processor on the FPGA is able to buffer the locations of compressed data within HBM and divide these over available decompressors. The design is successfully synthesized into a kernel that can be loaded by the FPGA.

From the moment compressed data is sitting in HBM until it is decompressed, a single decompressor reaches a maximum end-to-end throughput of 4.0 GB/s. When eight or more decompressors are activated, they reach a throughput between 20.0 to 26.2 GB/s. The hardware decompressor designs uses less than 10% of the resources of the U280 with little power usage. Compared to a software implementation, using multithreading, the hardware solution is 1.5-2.5x faster on a set of files that is used for benchmarks on decompression speed with varying compression ratios.

Preface

After many months of research, development and writing, this thesis is the final product of an amazing experience. It has been a time filled with debugging to figure out why something didn't work, and realizing 50% of the time it was my own fault and 50% of the time it was Xilinx's. At first, the technologies used for this project involved many concepts I was not yet familiar with. However, through the incredible help from the Accelerated Big Data Systems group and their feedback I was able to learn more about the field than I ever knew was possible. The knowledge and passion of the people inside the group amazed me more with each meeting I attended.

First I want to thank my supervisor Zaid Al-Ars, my daily supervisor Joost Hoozemans and Jeroen van Straten for all their suggestions and help they gladly gave whenever asked. Although for most of the time I have worked on the project the doors of the faculty were closed, they were always reachable. Their positivity and excitement over each small step forward kept on motivating me to keep progressing.

As the world is going through a difficult time I am even more grateful for the support of those close to me. Thanks to all my friends for making me laugh during every long synthesis and multiple debugging adventures. Last, but not least, this accomplishment would not have been possible without the never-ending love of my girlfriend Joyce and my family. *Thank you*.

Joep C. Dumont
Delft, February 17, 2021

Contents

List of Figures

List of Tables

1 Introduction

1.1. Context

The year 2020 changed the world drastically, forcing many people to work from home until some form of vaccination is released to the general public. This led to more digital traffic and data being generated than ever before. For example, every minute over 200.000 people are in a Zoom-meeting and 500 hours of video is uploaded to YouTube [8]. These numbers are likely to become even larger in coming years as the amount of data created and transferred is expected to grow by at least 40% between 2019 and 2024 [9].

At the same time, companies want to make decisions based on their huge amount of recorded interactions [10], which are in a raw form when obtained (Big Data). A data analyst can, after applying their tools and techniques, shape some statistic from this data on which a decision can be based. However, an analyst may have to wait long for a query on their database to complete. This long waiting period has a client wait even longer until they can make their decision based on the result, which might make them miss out of a potential business opportunity. So there may be great importance of analytic tools being able to parse huge amounts of data in an efficient way.

In addition, storing data is not free and being able to reduce the amount of data stored through compression can save a company a lot of money and energy over time. When at a later moment the stored data is required, the extra step of decompression has to be performed making the delay between query and result even longer.

Along with Moore's Law [11], stating that the number of transistors on an integrated chip double every two years, the available bandwidth for transferring data has kept growing, as can be seen in Figure 1.1. With a CPU being able to obtain data from RAM at a bandwidth of 100+ GB/s, it is not always capable of processing the data at such speeds. For example, an extremely fast decompression algorithm, LZ4 reaches at most a throughput of 5 GB/s per thread [12], meaning 20+ threads would have to be running in parallel to saturate the bandwidth. Other decompression algorithms may reach a throughput of 500 MB/s - 1 GB/s, requiring over 100 parallel threads to be active to saturate the storage bandwidth.

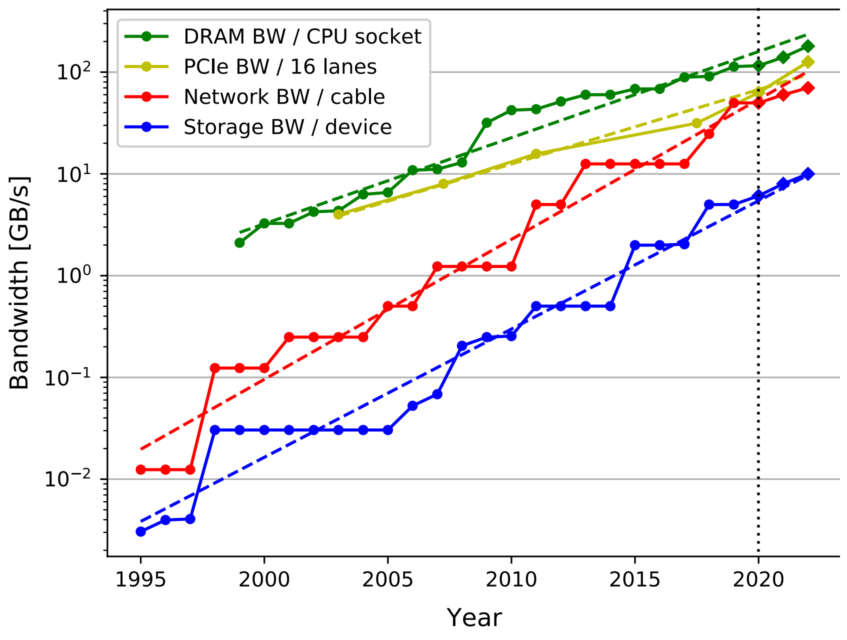

Figure 1.1: Bandwidth trend of different devices from [1].

Graphics processing units (GPUs) and field programmable gate arrays (FPGAs) can be used to relieve the CPU of the decompression task allowing it to work on the decompressed data or other tasks. FPGAs, even more so than GPUs, are able to use more of the available storage bandwidth at a lower power usage due to their low-level customization [13]. FPGAs are especially good at accelerating applications because of their capability of exploiting fine-grained parallelism for specific algorithms and their parallel kernels. As accelerators they are commonly used to stream data through and immediately perform calculations on the passing data, such as decompression using Snappy [14], image processing [15], (neural) simulation [16] and many in-memory database acceleration [1] applications.

However, a general issue for FPGAs are memory accesses, which need to be fast and with a wide bandwidth to prevent idle components. For example, a raw 1080p video consists of $1920 * 1080 * 3 = 6$ MB per frame. Attempting to apply a blur using the average of three or more frames will require some form of memory unit to buffer data as it quickly outgrows the amount of available storage using look-up tables (LUTs) on an FPGA. For applications which are memory-bound supporting platforms have to be developed [17] [18] to make benefiting from the parallelism of an FPGA easier.

To enable accelerators to temporarily store data there is a wide variety of options where on-chip memory is only a few megabytes wide and off-chip DDR memory holds multiple gigabytes. For example, the Alveo U200 has 64 GB of off-chip memory at a maximum bandwidth of 77 GB/s [19]. GPUs had a memory bandwidth one magnitude higher than the U200, the GeForce RTX 2080 Ti has 11 GB of GDDR6 at 616 GB/s and the Nvidia Tesla P100 has 16GB of High Bandwidth Memory (HBM) at 732 GB/s. To pull FPGAs into these ranges of bandwidth, vendors have started adding HBM to their FPGA which, for Xilinx, resulted in the U50 (8GB at 316 GB/s) and the U280 (8GB at 460 GB/s) for their Alveo Ultrascale+ family.

For simple applications, such as calculating the average of some raw video frames, the whole amount of bandwidth of the HBM can be easily used through parallelization, but when the frames are compressed, the decompression step can make it hard to parallelize. A way to still use this high amount of bandwidth for decompression would be to place smaller decompressing engines in parallel that perform a decompression on a part of the video stream. In much the same way this thesis will attempt to design an architecture that takes a compressed Snappy file and decompresses it directly on the FPGA and keep the result in memory for further calculations. The CPU on the host will only manage the transfer of data from storage to the FPGA so the host does not need to see the data by loading it in its own memory.

With the use of a hardware Snappy decompressor [20], the throughput of HBM will be measured when multiple engines are working on the same stream of data. As well the benefits of adding a processor to the architecture on an FPGA to aid the processing of data will be explored.

1.2. Challenges

Even a high-performance FPGA implementation for an application will often not be able to saturate the available bandwidth when using a single instance. One way of solving this would be to place multiple of these applications as independent hardware modules in parallel. This creates the problem of distributing the single input stream over the many placed modules and reconstructing their output. To manage this, the CPU of the host that contains the storage could, before starting the stream, go through the data itself and chop it up in chunks which can then be send to the individual modules. However this requires the data to pass from storage into the memory of the host before going to the FPGA. It would be more efficient, and now possible with HBM, to let the FPGA itself perform the parsing while a part of the data remains buffered in HBM.

Some FPGAs are currently capable of hosting their own processor in the reconfigurable fabric in the form of a softcore. This softcore can perform the same basic tasks as any usual processor, although at a much lower clock frequency. If data is being parsed by modules at such a high throughput that the softcore does not have enough time to manage the next packet, it may create stalls resulting in a lower overall throughput. Therefore, the program executed by the softcore should be able to process the content of such packet while the rest of the FPGA circuit processes another packet at the same time.

Finally, the addition of HBM to FPGAs is a very recent development so some benchmarks have to be performed to identify how the documented throughput can be achieved on an actual device.

1.3. Research questions

This thesis is concerned with addressing the challenges mentioned in Section 1.2. In order to do so the thesis will answer the following question:

What are the advantages of using HBM for accelerating decompression algorithms in big data applications?

To aid in the answering of the research question, it can be divided in the following sub-questions:

- What are the computational needs of incoming data streams to enable parallelization of applications?
- Is HBM able to mitigate challenges caused by limitations of on-chip memory resources?
- How can HBM aid in handling variable computation on variable sizes of data?

1.4. Thesis outline

Chapter 2 will provide all the required background information for understanding the design and implementation choices discussed in Chapter 4 and Chapter 3. The content discussed in Chapter 2 is focused on the parts that are relevant for the designs in this thesis. Chapter 3 states the requirements of the design and discusses some of the important choices made for its architecture. In the chapter also some other designs are briefly discussed on their strategy to benchmark HBM. In Chapter 4 the implementation and the challenges that had to be overcome are described. The general layout of a kernel is introduced and the configuration of its internal components is explained. Using the kernel from Chapter 4 results are obtained through benchmarks and discussed in Chapter 5. Concluding in Chapter 6 a summary of the thesis is provided and an evaluation of the research questions from Section 1.3. Also possible improvements and future work are given in this chapter.

2 Background

2.1. FPGAs as a compute platform

A field-programmable gate array (FPGA) is a device that allows a developer to program it's internal logic-circuitry after it has been manufactured into a chip. Internally, a basic FPGA mainly consists of configurable logic blocks (CLBs) that are interconnected and routed with switch matrices to other CLBs or input/output cells. A schematic overview of an FPGA is shown in Figure 2.1. Manufactures can choose to replace a number of CLBs for certain modules such as digital signal processing slices, memory blocks, transceivers for a certain external connection or integrate a whole processor.

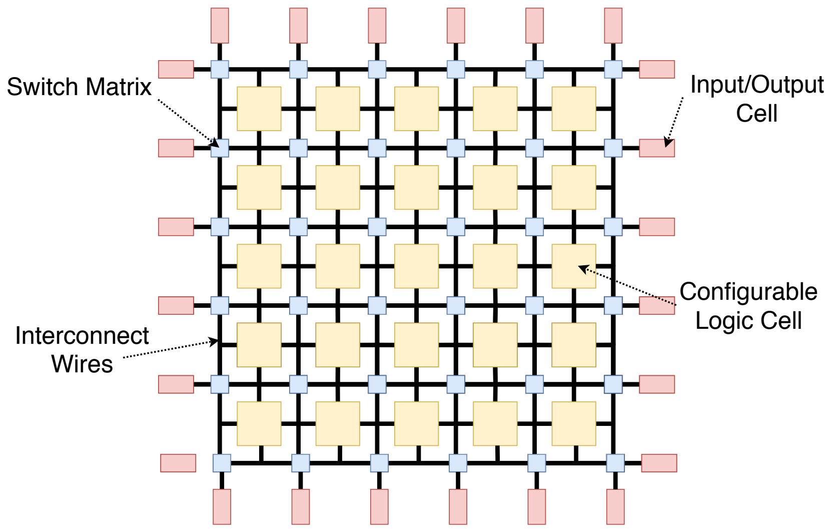

Figure 2.1: Schematic overview of a basic FPGA.

The programmed circuitry on an FPGA, once started, processes its inputs to its outputs in parallel where intermediate steps are synced by a global clock. This means that adding tasks does not affect the completion time of other tasks. If, for example, an addition of two inputs happen at one place, a division can take place on two inputs at another place at the same time. A very basic processor, with a single core, would first calculate the addition and then the division, consecutively. The FPGA, however, has a much lower clock frequency, generally in the order of a few hundred megahertz. A CPU has frequencies at a few gigahertz and some parallelism capabilities through the addition of multiple cores. So to achieve an identical throughput as the CPU, an FPGA would have to do at least 10 additions in parallel per clock cycle.

Apart from parallelism, the FPGA has another strength in its reconfigurability. The fact that FPGAs are reconfigurable is useful for prototyping circuitry that would later be baked into a chip or ASIC (application-specific integrated circuit). Their reconfigurability also makes it possible to quickly switch from one specifically applied algorithm to another. Their highly parallel execution proofs useful when many identical calculations have to be made, which occurs in fields such as image processing [15], (neural) simulation [16] and many in-memory database acceleration [1] applications.

2.2. AXI-bus interface

The Advanced eXtensible Interface (AXI) is an open-source specification [21] which is part of the Advanced Microcontroller Bus Architecture (AMBA) released by ARM. AXI was first released in AMBA 3 as AXI3 in 2003. In 2010 ARM released AMBA 4 that introduced AXI4 along with AXI4-lite and an AXI4-stream protocol. The designs used in this thesis contain many components that use these protocols for configuration or data transportation, so they are briefly discussed.

The term *packet* will be used to describe an element of data on a bus. In Figure 2.2 the single byte `0xDE` is an example of a packet. The term *transaction* describes the full transfer of data a component has requested. For example, in Figure 2.3 a transaction takes place between clockpulse 1 to 13.

2.2.1. AXI3 & AXI4

The AXI protocol describes a design for communication between any amount of master and slave components. Communication takes place using five independent channels. Each channel handles its transactions using a handshake mechanism. This handshake allows both the master and the slave to control the rate at which data moves between them. In this mechanism the source generates a `valid` signal to indicate that it has a new and stable packet for the destination. The destination asserts a `ready` signal when it can consume the next packet. A transfer then occurs during the rising edge of a clock cycle when both the `valid` and `ready` signal are asserted.

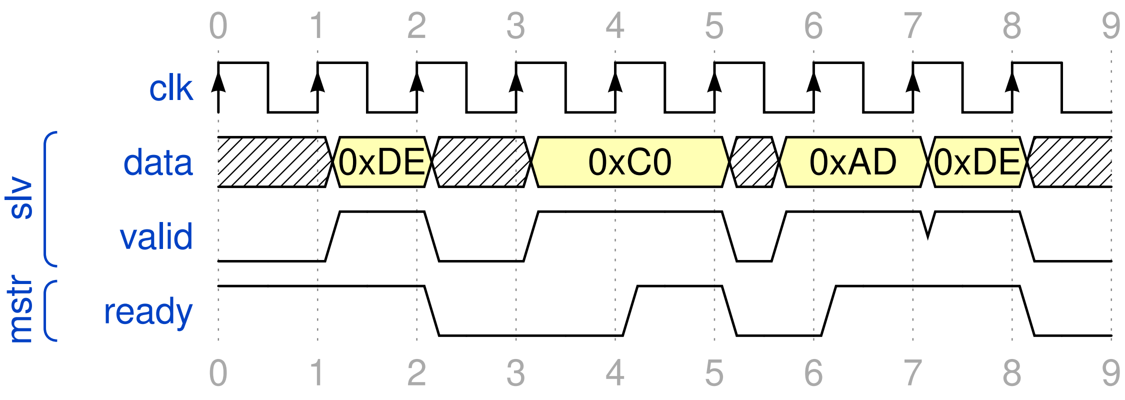

Figure 2.2: Four single byte transfers using handshaking mechanism.

In Figure 2.2 four packets containing one byte each are transferred using handshakes on a single channel. At the rising edge of clock cycle 2, indicated by the 2 above the `clk` signal, both the master is ready and the slave contains valid data so a packet is transferred. Now the master might need some time to clear a buffer and the slave might stall on retrieving some data so their lines go low. Once both the master and slave are ready again, which is the figure occurs at clock cycle 5, their next transfer takes place. The final two packet are transferred at at cycles 7 and 8.

An AXI-bus consists of the following five channels, each using this handshake protocol:

1. Read address and control
2. Read data
3. Write address and control
4. Write data
5. Write response

The address and control channels are used by the master component to tell the slave components information about the incoming data. Information such as the size of a single transfer, the amount of transfers, the address of the component it should reach and how the address changes between each transfer. The read data channel consists of signals from the slave such as the data that the master requested, a response signal indicating the status of the transfers and a signal that is asserted when the current packet is the last packet of this transaction. The write data channel consists of mostly the same signals as the read data channel, however they are generated by the master and received by the slave. The write response channel lets the slave indicate to the master that a write has completed.

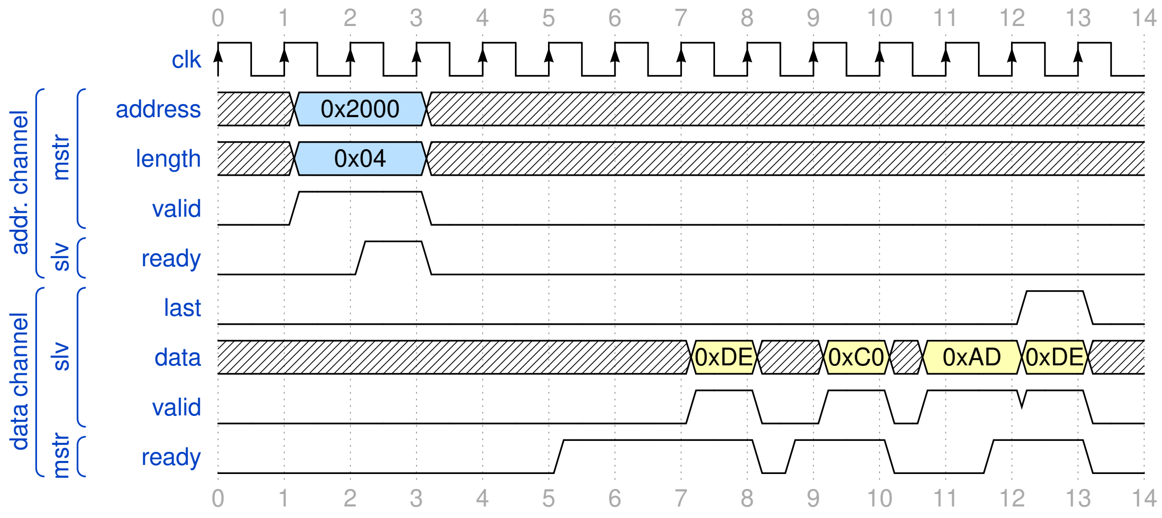

Figure 2.3: Example of AXI3/AXI4 protocol retrieving data at specified address.

Figure 2.3 shows an example of a master reading `0x04` bytes from a slave at address `0x2000` using the read address and read data channels. The master supplies the address it wants to read from, the amount of bytes it wants to read and asserts `valid`. Once the slave is ready to read the address, it asserts its `ready` signal. In the figure, at clock cycle 3, the transfer of the address packet is considered completed and the master can deassert `valid` as no extra address will be transferred. In the read data channel the slave can now start pushing the data found at the received address. Data transfers are consequently performed until the `last` signal is asserted, indicating the full transaction has completed. In a practical transaction more settings are negotiated by the master and slave, such as the size of a packet (one byte here), so the slave knows how to increase the address.

2.2.2. AXI-Lite

Not every communication between two components needs all the sophisticated options that the full AXI protocol facilitates. When only a single value has to be written or set, such as during configuration, a subset of the AXI protocol will suffice. This is implemented as AXI-Lite and consists of the 5 channels that AXI has but only implements the signals required to transfer a single packet per transaction. For example the `last` signal used in the transaction in Figure 2.3 is not present in AXI-Lite as a transaction consists of a single packet anyway.

2.2.3. AXI-Stream

When data is supposed to flow in one direction without any requirements of an address an AXI-Stream is more suitable, specified separately from AXI in [22]. For example, if a component reads data from memory this data can be transferred as a stream to a component that sums up all values that are read. The summing component does not care where the data comes from or goes and the direction of the flow of data stays the same. The AXI-Stream protocol, in contrast to AXI and AXI-Lite, consists of a single channel going in one direction. The basic signals implemented in the channel are the same as the signals of the read data channel in the example of Figure 2.3. It, like the AXI protocol, has some optional signals that can be added.

Of the optional signals the `keep` signal is used in some streams used in the designs for this thesis. This signal indicates which bytes from a packet should be processed as data and which bytes can be ignored. This is useful when a data element does not fill the width of the bus. For example on a bus that has a width of four bytes and the actual value to be transferred is three bytes, one byte can be ignored by the receiving component. The sending component indicates this by having the `keep` signal consist of three 1's and one 0 where the location of the 0 correspond to the location of the byte that can be ignored. Usually the `keep` bus is filled with 1's since only for the last packet some bytes will have to be ignored, as the full length of a transaction doesn't have to be divisible by the size of the data-bus.

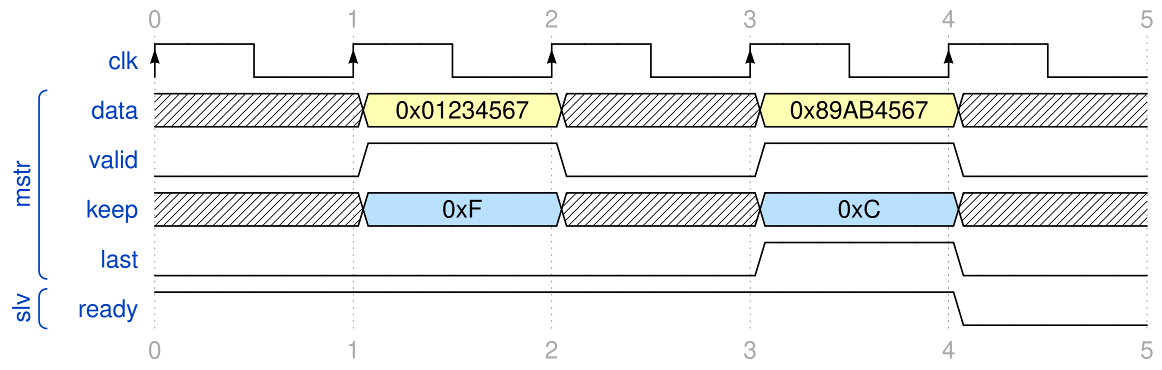

Figure 2.4: AXI-Stream example using keep signal.

In Figure 2.4 an AXI-Stream is displayed where a transaction of six bytes is completed. As six bytes cannot be equally divided by the width of the data-bus, which is four bytes, one of the packets must contain partial data. In the figure all the bytes of the first packet are to be kept as data, and as such the `keep` signal is set to `0b1111` or `0xF` in hexadecimal. The second packet however has a value of `0xC` or `0b1100`, so only the first two bytes are kept: `0x89AB`.

2.2.4. vhlib stream library

vhlib [23] is a library of components that can be used to create, manipulate and handle streams. It was created as part of Fletcher [17] but became generic enough to become its own library. The hardware Snappy decompressor used within the designs in this thesis implements vhlib to handle streams. AXI-Streams and vhlib streams are not directly compatible (yet) but under certain circumstances can be transformed to each other with some additional logic.

An important difference is that instead of a `keep` signal it uses a `cnt` signal that indicates *how many* of the bytes in a packet are part of the data, starting from the most significant byte. As vhlib does not support `NULL`-packets, where none of the bytes in a packet are to be kept as data, a `cnt` of `0x0` is not used and can instead be used to indicate all the bytes are data.

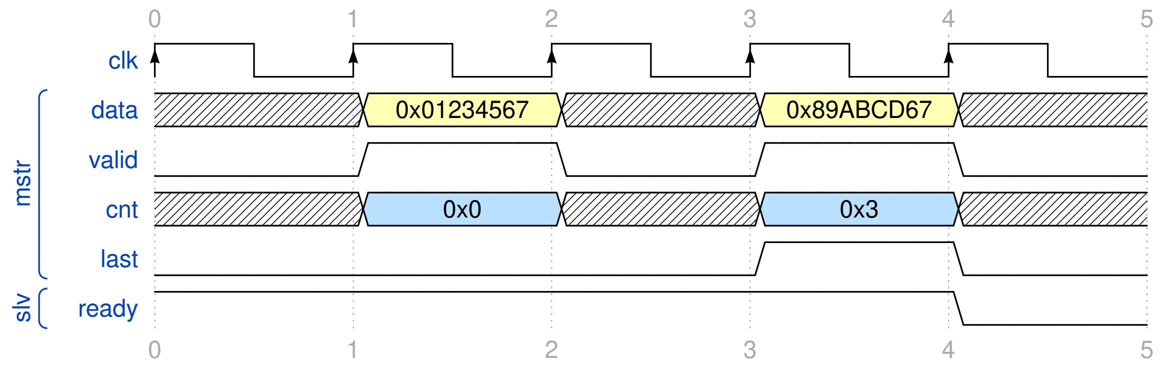

Figure 2.5: vhlib example using cnt signal.

Figure 2.5 shows the same data being transferred as in Figure 2.4, but with the vhlib protocol. As all four of the bytes in the first packet are data, the `cnt` signal has a value of `0x00`. In the second packet only the first two bytes are data, so `cnt` is set to `0x3` or `0b11`. Where an AXI-Stream needs four bits to describe which bytes to keep, vhlib only requires two. However the `cnt` signal loses the option to keep *any* byte within a packet, so the data bytes will always have to be shifted to the front of a packet.

2.3. Microprocessor on FPGA

To aid in reducing the complexity of a design on an FPGA a (micro)processor may be integrated along the reconfigurable logic. For FPGAs there are two varieties of embedded microprocessors: as a hard-core processor or soft-core processor. A hard-core processor is preconfigured on a dedicated block, which a manufacturer has placed directly inside the FPGA, replacing a few CLBs of Figure 2.1.

An example is the Xilinx Zynq 7000 series which includes ARM Cortex-A9 that can be configured to a frequency of up to 1 GHz. A soft-core processor can optionally be added to the design and will be synthesized along the rest of the design within the reconfigurable fabric. Even multiple soft-cores can be implemented to create a multiprocessor system on a chip (MPSoC) which has been explored extensively [24] [25]. Advantages of implementing soft-core processors within a design include the flexibility (many processors can be embedded, limited only by the resources on the FPGA and each can be configured independently) and the low time-to-market (programming and debugging the source code is usually faster than implementing a hardware solution). However, soft-core processors are limited to an operating frequency of 250-300 MHz.

The FPGA used for the designs of this thesis, the Xilinx Alveo U280, does not include a hard-core processor but can implement the MicroBlaze soft-core developed by Xilinx. The MicroBlaze is a reduced instruction set computer (RISC) that supports both 32-bit and 64-bit address configurations with a clock frequency limited to 250 MHz. Once `C`-code is written it is compiled using its associated compiler. These are then written to the (Block RAM) memory components of the FPGA which the MicroBlaze has access to during execution. The maximum size of the program, including the stack and heap, should be 128Kb for optimal performance.

2.4. High Bandwidth Memory

High Bandwidth Memory (HBM) is an interface to synchronous dynamic random-access memory (SDRAM) that are stacked in 3D. It's main advantages are lower power consumption, less area usage and higher bandwidth than other interfaces to SDRAM, such as DDR6. HBM however costs more in manufacturing and needs more cooling due to being densely packed. In 2013 JEDEC standardized and specified HBM as JESD235 [26] and in 2015 HBM2, as JESD235A [27], was announced. Since 2016 HBM2 is being developed by Samsung and SK Hynix and has been used for GPUs, the first being the Nvidia Tesla P100 (which wasn't proving very useful at that point [28]). In 2018 Xilinx released the first FPGA that had an HBM module added. with a 1024-bit interface over it's multiple channels and being able to reach a bandwidth of 256 GB/s per stack featuring 4GB of memory.

In Figure 2.6 a schematic cross section of HBM on a package is displayed. To reach the high bandwidth multiple memory dies are stacked on top of each other with a controller/logic die on the bottom. A vertical connection through the stacked dies is made with electrical connections named through-silicon via (TSV) and connected via microbumps. An interposer layer, without logic, connects the logic die to the processor/GPU/FPGA of the package. Further connections through the substrate of the package are available for debugging purposes. Often this configuration is named 2.5D as the memory is stacked in 3D but attaches to a processing unit horizontally.

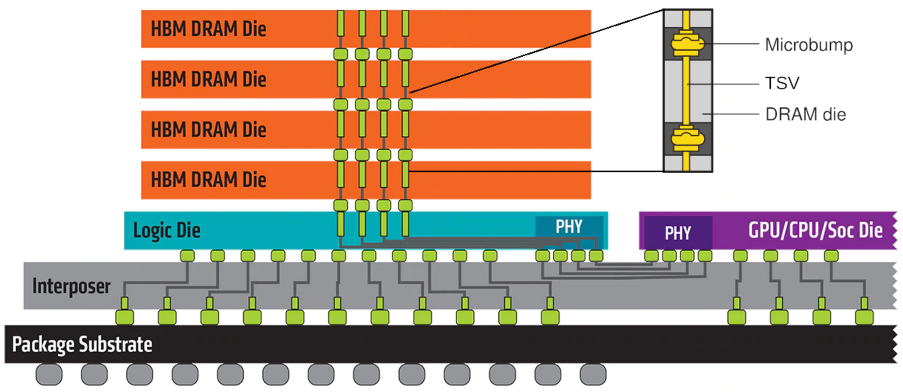

Figure 2.6: Side view of HBM on a chip, figure from [2] and TSV zoomed in figure from [3].

HBM IP

An overview of the interfaces of HBM, with two stacks, used in an architecture with an FPGA can be seen in Figure 2.7. The HBM IP is the interface Xilinx provides to support the communication between the FPGA logic and the HBM and specified in [29]. In a stack, each of its 16 banks of 256 MB is connected to the IP through its own so-called pseudo-memory channels that have a width of 64 bits. The IP features 8 memory controllers per stack that each manage two pseudo channels for a total of 16 pseudo channels. The memory controllers upscale the width of the connection to 256 bit and are connected to a crossbar. At the other end of the crossbar are an AXI3 for each pseudo-memory channel with which the rest of the logic of the FPGA can communicate. The conversion from 64 bit to 256 allows the FPGA to reach a high bandwidth while at a lower frequency. The crossbar allows any AXI3 port to reach any memory location within the HBM. The crossbar, however, does not support reading the same bank using different channels at the same time. This means that reading two values from the same bank takes twice as long as reading two values from two different banks as will be shown with later measurements.

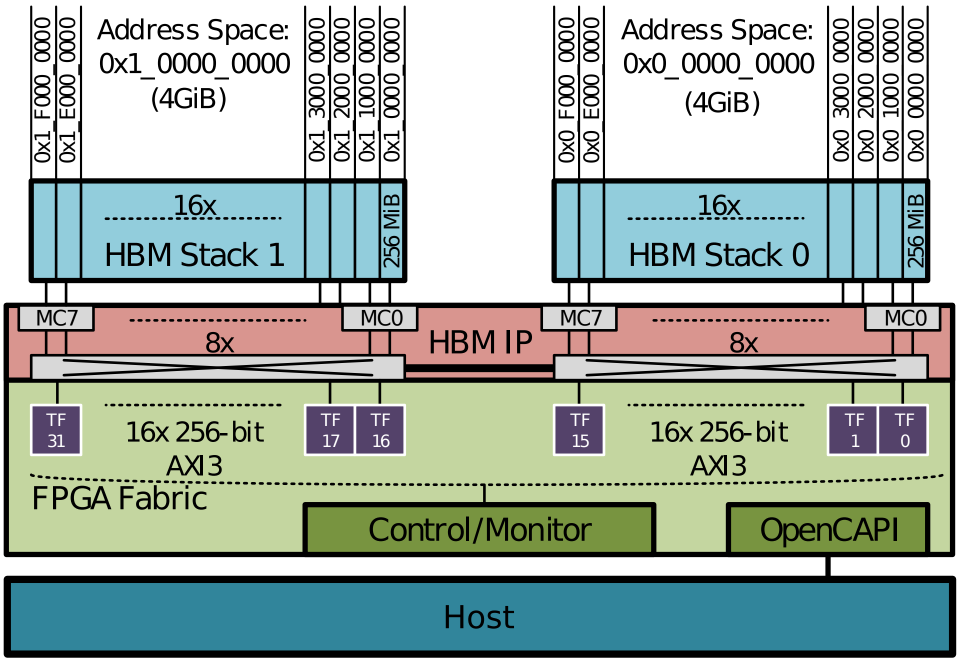

Figure 2.7: Overview of HBM on an FPGA, figure from [4].

In optimal conditions the throughput from the side of the IP can be calculated from the following considerations. A stack contains 16 pseudo-memory channels with a width of 64 bit, $16 * 64 = 1024$ bit $=$ 128 bytes. Within a channel a data bit toggles twice the rate of the HBM clock, which is maxed at 900 MHz. Per stack this means $2 * 900 * 10^6 * 128 = 230.4$ GB/s can be transferred. On a configuration with two stacks, like in Figure 2.7, the throughput is doubled to 460.8 GB/s. To match this throughput on the FPGA side the 16 AXI3 ports, with a width of 256 bit (32 bytes), have to be clocked at 450 MHz. Then in the same way for one stack $16 * 32 * 450 * 10^6 = 230.4$ GB/s.

2.5. Snappy compression

Snappy (Github: [30]) is a lossless compression and decompression algorithm based on the scheme algorithm named LZ77 [31]. LZ77 algorithm are dictionary coders, which achieve compression by placing references to data that it has encountered earlier, rather than the data itself. This algorithm therefore reaches the best ratio when a lot of identical data is compressed, however achieves barely any compression if data is completely random.

It aims for very high speeds rather than compression ratio. On a single core of an i7 processor in 64-bit mode it can compress at a rate of 250 MB/sec and decompress at around 500 MB/sec. This scheme is used to compress huge tables of data, most notably by Google. It is written in C++ but can be used by many other programming languages through bindings from third parties.

within the repository is a specification for Snappy that describes how data is compressed and decompressed. It also offers a framing format that allows the compressed data to be chunked in smaller parts and written to a file, however using this framing is optional.

2.5.1. Compression format

Since Snappy (de)compression will be used to measure performance the format is described and an example given. The first bytes indicate the total length of the original data as a little-endian `varint`. Varints are an encoding used to represent integers using one or more bytes. This allows smaller numbers to use less bytes while also supporting bigger numbers. All bytes in a varint have 7 of their 8 bits store a part of the two's complement number it is encoding, the most significant bit (MSB) is set to 1 if more parts are following and set to 0 if it was the last byte.

An example of the decimal 333 being encoded as a varint is given in Figure 2.8. First the decimal is encoded as binary number and, starting from the right, split at every 7 bits. Since the varint will be in little-endian first the lower bits of the number are evaluated. As more bits are followed by the first bits a '1' is placed as the MSB. For the second part (the `0b10`), it is the last part of the number, so a '0' is placed as MSB. Converting these numbers to their hexadecimal representation and concatenating it results in the varint `0xCD02`.

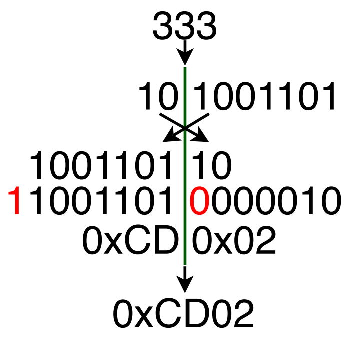

Figure 2.8: Example of decimal 333 being encoded as varint.

For a single Snappy compressed stream the maximum length is $2^{32}-1$, so a single stream consists of a maximum of 4 GB and the size of the varint is at most four bytes. After the length the rest of the bytes in the stream are encoded using four types of elements. Which type the next element has is indicated by the lower two bits of the first byte of the element. This first byte is also named the `tag byte`. The remaining, upper, six bits have an interpretation that is dependent on the element type. The four elements are indicated by the two bits as follows:

- 00: Literal
- 01: Copy with 1-byte offset
- 10: Copy with 2-byte offset
- 11: Copy with 4-byte offset

A literal element means that no compression is performed so data can be read directly. This length of the literal is stored depending on the size. If the length is 60 or less bytes the length is stored in the remaining six bits of the `tag byte`. If the length of the literal is longer than 60 bytes, the length is found in the bytes following the `tag byte`. The amount of bytes the length is written in described by the same six bits of the `tag byte` where 60, 61, 62 or 63 mean a length of 1, 2, 3 or 4 bytes respectively.

A copy points back to previous decompressed data. A copy element consists of two parts: the offset, how many bytes back the data can be found and length, how many bytes to insert. In the copy with 1-byte offset, three bits from the `tag byte` are used for the length (+4), the remaining three and the next byte together (for a length of 11 bits) describe the offset. In the other two copy elements the six remaining bits of the `tag byte` store the length (+1) of the copy and the two or four bytes after the `tag byte` indicate the offset of the copy, in little-endian.

An example of data that can benefit from Snappy's compression technique are the lyrics to the song "Around the world", made famous by the French electronic music duo Daft Punk. The lyrics are the phrase "Around the world, around the world" repeated 72 times. Even within the phrase already a repetition is present, except for the capitalization of the 'a' character. An example of the decompression of a Snappy compression applied to the first three lines of Daft Punk's "Around the world" is shown in Equation 2.1.

$$694841726F756E642074686520776F726C642C20613A1200000AFE23000923 \tag{2.1}$$

$$\begin{aligned}
0x69 &= 0b01101001 \\
&= \text{varint storing length of uncompressed data} \\
&= 105 \\
0x48 &= 0b1001000 \\
&= \text{literal follows, length} = 18 + 1 \text{ bytes} \\
0x4172 - 0x2061 &= \textit{"Around the world, a"} \\
0x3A1200 &= 0b111010,\ 0b10010,\ 0b0 \\
&= \text{copy with 2-byte offset, length} = 14 + 1, \text{offset} = 18 \\
&= \textit{"round the world"} \\
0x00 &= 0b000 \\
&= \text{literal follows, length} = 0 + 1 \text{ byte} \\
0x0A &= \text{'newline character'} \\
0xFE2300 &= 0b11111110,\ 0b100011,\ 0b0 \\
&= \text{copy with 2-byte offset, length} = 63 + 1, \text{offset} = 35 \\
&= \textit{"Around the world, around the world"} + \text{(newline)} \\
&\quad \textit{"Around the world, around the "} \\
0x0923 &= 0b1001,\ 0b100011,\ 0b0 \\
&= \text{copy with 1-byte offset, length} = 2 + 4, \text{offset} = 35 \\
&= \textit{"world"} + \text{'newline character'}
\end{aligned}$$

The first line in Equation 2.1 shows the compressed stream of bytes as hexadecimal numbers. The bytes are parsed from left to right. In red is the varint describing the length of the decompressed data is decoded as 105. Blue bytes describe literals. black bytes represent uncompressed data, indicated by the literals. The violet bytes are copy elements. Converting all the decompressed bytes to their ASCII character the first three lines of "Around the world" are retrieved. Each line, consists of 35 characters, including the newline character. The total amount of decompressed bytes is $35 * 3 = 105$, which is the same as the varint, indicating the full stream is decompressed.

2.5.2. Custom framing format

In Equation 2.1 a byte stream is given which could directly be written to storage. However when input data becomes large compression and decompression can benefit from chopping up the data into chunks of smaller sizes. These smaller chunks can be compressed or decompressed in parallel since no data dependency exists between them. Indicating the start and length of a chunk is done through a framing format. A framing format is provided in the Snappy Github at [30]. It describes a file header, checksums and how chunks lie back-to-back, each chunk starting with a chunk header followed by the raw data.

For the design in this thesis a custom framing format is created. This is done to ease complexity of parsing chunk headers inside the FPGA. As the design is more proof-of-concept then a product this suffices for the measurements. In the future the Snappy framing format as described on the Github could be implemented.

The custom framing format adds 64 bits to each chunk of a Snappy compressed byte stream. The 64 bits are two (little-endian) 32 bit unsigned integers. The first integer is the size of the compressed data, the second the size of the uncompressed data in the chunk. The end of a file is indicated by a header consisting of 64 0-bits, or 8 empty bytes.

2.5.3. Hardware Snappy decompressor

The Accelerated Big Data Systems group of the Quantum & Computer Engineering department of Delft University of Technology has created an open-source hardware version of a Snappy decompressor named vhsnunzip (VHdl SNappy unzip) at [20]. It is targeted at the Xilinx UltraScale+ FPGA family. It has two versions, a buffered and unbuffered one both with a maximum frequency of about 250 MHz. The unbuffered version can decompress Snappy compressed data at a rate of 1.5 GB/s, can take chunks of sizes larger than 64 KB and has an input width of 64 bits. The buffered version supports multiple cores of decompressors in parallel, allowing decompression at a throughput of 8.0 GB/s, when 8 cores run in parallel. The buffered version only works for chunks of at most 64 KB but allows an input width up to 256 bits.

3 Alternative solutions

The attachment of HBM to FPGAs has not gone unnoticed and the uncharted limits of HBM have sparked some studies into applying benchmarks to it. In one paper [32] an open-source tool named Shuhai is developed. This tool allows micro-benchmarks on HBM, exploring the latency of a channel under certain conditions, such as changing the used AXI-port when accessing a specific bank. Their tool also measures the impact other channels might have when used when a single bank is accessed, which is something this thesis will also be much involved with. In another paper [33] the high-level synthesis (HLS) tools, that are supposed to ease development on FPGA, are used for development and benchmark the performance and overhead of their resulting kernels on three different boards.

The design in this thesis will go one step further by combining the ideas of both papers as will be discussed in the following sections.

3.1. Requirements specification

In order to design an architecture that can find advantages and limitations of HBM on an FPGA some requirements are specified using the MoSCow method. The summary of the design requirements are shown in Table 3.1. The most important requirement is that the design should be synthesizable for the Alveo U280 FPGA as it is the FPGA that is available to perform benchmarks on. The design itself should be able to interact with HBM, via reading or writing to it and measure the time it takes to complete such an interaction. Once a design is able to perform this action the measured time can be used to find how different types of data changes the measured time. From a design that is able to perform such transactions further steps can be taken. The design that is limited to just the reading and writing of data can eventually be used as a baseline, to compare the other, more sophisticated, designs to.

The design should implement an engine in parallel that uses the data obtained from a transaction. Additionally the result of this engine should be available for further processing. With this it may show more of the limitations HBM may have when parallel operations are being applied to its memory banks. The data being available proofs that the implementation could be used as part in a bigger chain of processing elements. It also allows the result to be tested for correct and expected values through some verification unit. The design should be able to measure both the full possible throughput achievable from HBM as well as be able to apply a practical set of data that represents practical usage of the FPGA.

The host to which the FPGA is attached should not have to interact with the data from the moment the full processing of the data is started. This specifically means that the processor of the host should only be concerned with setting up, starting and monitoring the processing of the kernel(s) on the FPGA. The processor should not be required to parse headers or measure the amount of bytes of the data for the design to work. The kernel should be able to work autonomously on the data. Through this requirement the measurements and its results on HBM do not rely on the specifications of the CPU on the host. At the same time it prevents any potential overhead experienced due to communication between the FPGA and CPU.

A useful property of the design would be the ability to configure the amount of engines that are operating at one time on the HBM. Ideally the host can configure a kernel to use a selected amount of engines such that it can be scripted as part of a benchmark. To proof further processing of data is possible the design can be extended to implement an additional engine that applies a final reducing function on the result of the first engine. Although it does not show more advantages or limitations of HBM it may show the effect of total throughput when used in a practical use-case. Finally it would be useful to implement an other engine to compare their individual throughput, as the results of a single implementation may not be representative enough. Additionally, through the process of replacing the engine it may remove any issues of implementing an other engine, such that the ease of replacing the engine is increased.

Xilinx provides two types of engines for Data Base acceleration cards to perform Host to Card (H2C) and Card to Host (C2H) operations: XDMA and the newer QDMA. The main difference between the two is that QDMA uses queues and XDMA uses channels for it's transfers. For the implementation this means that QDMA allows data to be streamed in using AXI-Streams, directly into the kernel and streamed back out again. XDMA on the other hand uses a channel to memory-map data from the storage into a memory slot on the card (such as in HBM or DDR) using the standard AXI protocol. As the available Alveo U280 card is programmed for XDMA the design is adjusted to expect memory to be inside the FPGA once the kernel is activated, instead of being streamed in through a QDMA interface as an AXI-Stream.

Must have	***Should have***	***Could have***	***Wont have***
Synthesize on U280	Use parallel engines	Process data after first engine into result	AXI-Stream as input
Interact with HBM	Processed data remain available for further processing	Implement other engine	
Measure time of transaction using a engine	Host should not have to interact with data	Allow variable amount of engines at runtime	
	Able to measure full bandwidth and apply practical engine	Make it easy to replace engine by any other engine	

Table 3.1: Summary of MoSCoW method requirements

3.2. Architectural alternatives

3.2.1. Controlling engine

Within the kernel there will exist many elements that require some form of control: data has to be routed, headers of data have to be parsed, the time of a transaction has to be measured, etc. Each of these tasks can be performed by some unit that manages the single task or a central component can be used that handles everything. A central component may create more overhead than when each task has its own monitoring unit. But it allows most tunable parameters to remain in one place at which a developer may configure or the host may communicate its settings to. Through the use of AXI-Lite connections these parameters can then be passed forward to the other components in the design.

This single controlling unit can be designed with a few architectures as well. Either it could be an in logic created finite-state machine (FSM), which is what was used done to parse headers in a the hardware implementation of a parquet converter [34]. Or it can be setup using a processor. An FSM should be able to perform all the controlling tasks within a lower amount of clock cycles than when a processor is implemented. However the FSM will become very complex and may not even be possible to synthesize once many parallel engines are added. Additionally it is expected that most of the time the controlling engine will be idle. It will only be busy when parsing a header or setting up a new transaction. Once those tasks are completed it will have to wait for some engine to finish its transaction, which may take many times as much cycles to complete. So with the overhead likely not impacting the overall throughput of the full transaction, when the transaction transfers a big amount of data, the processor is the least complex option. Another advantage of using a processor is that an update on the specification of the headers or management of modules can easily be adapted in code without having to dive deep into HDL logic.

As the processor has the most advantages, it will be used as the controlling engine in the design.

The Alveo U280 does not have hard-core processor component, but the MicroBlaze is available as a soft-core processor, described in Section 2.3. The MicroBlaze can connect to other components through an AXI-Lite interconnect. The interconnect allows a 1-to-N connection that routes data from a single input to another components based on it's address, much like a router. Additionally it is connected to its own local memory bank to retrieve instructions and perform transactions with the stack and heap. Lastly it has the ability to output and read data to and from AXI-Streams.

Alternatively the CPU on a host can be used as the processor. However, as discussed in the previous section, Section 3.1, it is beneficial if the data does not have to pass the host before processing. For the implementation of the design a soft-core processor will be used as the controlling engine.

3.2.2. Functional engine

As the functional engine that is used within the design there are many options. To make use of the fact that there is a HBM the application should be memory-bound. Within the paper [33] that uses HLS with HBM, a bucket and merge sort, a matrix-vector multiplication and a stencil application is used. The other paper [32], using Shuhai, does not embed an application and measure the achieved bandwidth by queuing reads and writes.

Within this thesis a design will be used that allows measuring both the full bandwidth as performed by Shuhai and implements a application as the paper using HLS. The full bandwidth will be achieved in the same way as Shuhai, by adding a component that constantly tries to read and write from HBM. Where these values are written to in HBM will be controlled by the controlling engine.

For the functional engine that applies some computation, the TU Delft has some application already created. In order to explore the added benefit of implementing a MicroBlaze, a application that requires the parsing of headers would be the best fit. One such application is the Snappy hardware decompressor described in Section 2.5.3, in which the framing format headers are used. Another is a hardware Parquet-to-Arrow converter [34] which uses *pages* as its input that use a header to store the specific compression used for that page. Both will not immediately fit within the design as they do not use AXI-Streams as their protocol for streaming data through the engine. The specifications of the hardware Snappy decompressor are better known and the vhlib streams are only a slight deviation from AXI-Streams so it will be used as the functional engine for the first design. Once it is shown to work, it the Parquet-to-Arrow may additionally be implement to use for comparison.

3.2.3. Design environment

When creating a kernel for an FPGA there are a few alternative strategies with which a design can be described that allows synthesizing it into a kernel for an FPGA. Generally used strategies are describing the kernel using high-level synthesis (HLS) tools or register-transfer level (RTL) abstraction. Usually HLS tools let a developer use *C/C++* as their ”high-level” language which the tools then translate to an RTL abstraction. Such an abstraction is usually written using a hardware description language (HDL) such as Verilog or VHDL. Using a HDL allows a developer to tell the synthesizer exactly what kind of connections and functions should exist to other functions and peripherals.

Within the Xilinx Vivado Design Suite, that is generally used to develop kernels for FPGAs made by Xilinx, an extra strategy is provided. Vivado allows a kernel to be designed using a *block design* or *block diagram*. A block design is the visual representation of a kernel. In the block design a block, named IP (Intellectual Property) by Xilinx, can be linked to other IPs to form a kernel. Xilinx provides many pre-made IPs through its library such as adders, counters and a MicroBlaze softcore. A developer can add these IPs to their block diagram and configure them to their need within Vivado through a provided graphical user interface (GUI). If the available IPs does not suffice a developer can create their own by packaging a HDL description into an IP. The provided IPs by Xilinx can not be configured more than through the provided GUI as the HDL of the IPs are encrypted.

As mentioned in the paper [33] using HLS to benchmark HBM, currently HLS is limited in its customization of the connections an engine can make to HBM channels. Additionally the hardware Snappy decompressor discussed in the previous section, Section 3.2.2, is fully implemented in HDL. Therefore either an RTL design or block design would be the safest approach to accomplish a fully working kernel. With the block design the focus can remain on getting a working kernel rather than a fully custom made kernel. The HDL from the hardware Snappy decompressor can be packaged as an IP within the block design to attach to other IPs. During development a testbench can be created to monitor the flow of data for any errors, issues or crashes. Block designs will be created to eventually synthesize into the kernels for the Alveo U280 using the Vivado design suite.

3.2.4. Multiple bank accesses

Although 16 banks are available per HBM stack on the FPGA, a single compressed file can not be easily decompressed by reading from two banks at the same time. Before a decompression can be started, the processor has to find the location of a chunk of data by parsing a header. From the contents of that header the next header can be found. This continues until the end of the file is found. If two banks are to be used, at least one location of a chunk in the second bank should be known. However with the limitation of the host not interacting with the data the only way to find a header in the second bank would be to traverse the whole link of headers until passing the pointer to the header crosses the size of bank. Before starting this journey, it is unknown what the sizes of each header is and if the total data would even be larger than a single bank. With the size of a HBM bank being 256 MB and the default chunksize being 64 KB, it may be possible that 4000 headers will have to be parsed before the header in a new bank is found. One could argue that these headers will have to be parsed at some point anyway and some may even be stored in memory so they will not have to be parsed anymore. Doing so may create a long stall before starting decompression, and if the file is smaller than the size of a bank even a stall that does not provide any throughput as well.

Once such an implementation is created, it would be an extension of the results obtained from accessing a single bank. Therefore it is not attempted within this thesis, but may be performed in the future.

3.3. Relevant characteristics

3.3.1. Expected speedup from parallel decompressors

Shown in Section 2.4 a single HBM stack has a bandwidth of 230 GB/s divided over 16 banks, each accessible using an AXI3 port. As stated in Section 3.2.4 most of the time during a decompression a single bank will be used to read from. A different bank can be used to write to. When a single decompressor is used for decompression it will not be able to use the whole 14.4 GB/s bandwidth of the bank, as the decompressor can handle a maximum of about 1.5 GB/s [20]. Additionally the actual throughput will include the time it takes the processor on the FPGA to parse the header of each chunk. With a single decompressor, it may have to be idle while the processor is parsing the header before being able to start the next transaction.

With multiple decompressor active in parallel the actual speedup mostly depends on the ratio of time spent parsing headers and the time spent streaming data through the decompressors. This can be observed in Amdahl's law, shown in Equation 3.1. When the ratio, P, gets close to one, meaning most of the time is spent on streaming data compared to the parsing of headers, the speedup nears N, the amount of parallel processing elements. However when the parsing of a header takes long, compared to the time of a transaction and P goes to zero, no speedup will be observed when adding more decompressors.

$$\text{Speedup}(N) = \frac{1}{1 - P + \frac{P}{N}} \tag{3.1}$$

When ten decompressors are placed in parallel and P is close to one, the calculated throughput would become 1.5 GB/s * 10 = 15 GB/s. However a bank can only provide 14.4 GB/s and may not even be able to handle the many different decompressors trying to access the bank. Using the size of a chunk the ratio P can be slightly changed for different measurements. The design will try to find the actual throughput measured when multiple decompressors try to access a bank and where the optimal amount lies. These measurements will show the actual speedup obtained from the parallelization.

3.3.2. Other memory types

Xilinx provides a wide range of memory structures on most of their FPGAs: *Distributed RAM*, *Block RAM* and *UltraRAM*. Distributed RAM is created by using LUT (lookup table) resources of an FPGA to store up to 64 bits. To store bigger portions of data Block RAM (BRAM) may be used to store up to 36 Kb (4.5 KB) of data. These can be cascaded to form bigger blocks of memory. There is a limit to the amount of memory that can be obtained through cascading as there are a finite amount of them available. Additionally they are not all placed in the same location on the FPGA, requiring resources to fully cascade them together and a lower operating frequency as data is transported. For bigger memory blocks UltraRAM (URAM) is available. Their blocks are 288 Kb (36 KB) in size and also support cascading. They are dual-ported meaning two memory operations can take place during a single clock cycle.

Memory specification	
Distributed RAM size	64 bit
Distributed RAM capacity range	1.2 Mb – 48.3 Mb
Block RAM size	36 Kb
Block RAM capacity range	5.3 Mb – 94.5 Mb
UltraRAM size	288 Kb
UltraRAM capacity range	90 Mb – 360 Mb
HBM size	4 GB – 8 GB
HBM capacity range	4 GB – 16 GB

Table 3.2: Specification of different memory types available for the Ultrascale+ family as provided on [6].

In Table the available internal memory capacity ranges are shown for the Virtex Ultrascale+ FPGA cards, of which the Alveo U280 is part, with HBM added. URAM at 360 Mb or 45 MB may be enough to operate in parallel on a few chunks of 1 MB of data. However to allow bigger data files to be decompressed in parallel a bigger amount of memory will be required.

HBM lets data up to 256 MB in size be placed within a single bank and provides multiple interfaces to the whole available range of banks. As off-chip memory 32 GB of DDR, divided over two banks, is available for the Alveo U280. This can easily be used by single decompressor where is being read from one bank and the result is written back to the other bank. However when multiple decompressor are to be attached, not enough interfaces are present. This may be possible through an efficient interconnect in the same way the HBM IP allows 16 AXI interfaces to a single stack of HBM. Additionally, it will not be possible to extend to the use of multiple kernels as all available banks are in use. Finally it will also no be possible to use multiple banks within one design as described in Section 3.2.4.

4 Implementation

4.1. System level design

As specified in Section 3.1, in the end the goal is to have a host be able to perform some operation on data within a Snappy compressed file, sitting in storage. The compressed data should flow from storage directly into an FPGA, through decompressors and afterwards have the data sit ready for an operation in global memory. The host should not have to see the data or interact with it in any way for the decompression to succeed. For measurement purposes the FPGA should be timed on how long it takes to completely decompressing the data, from the moment the compressed data is in global memory until the moment the decompressed data is available in global memory. In Figure 4.1 a schematic overview of the design, is shown. The CPU on a host can control both the storage and FPGA, for starting and monitoring an operation. Through OpenCL bindings written in `C++` the host can program a kernel on the FPGA, read and write data to global memory, set arguments used by the kernel and monitor its status.

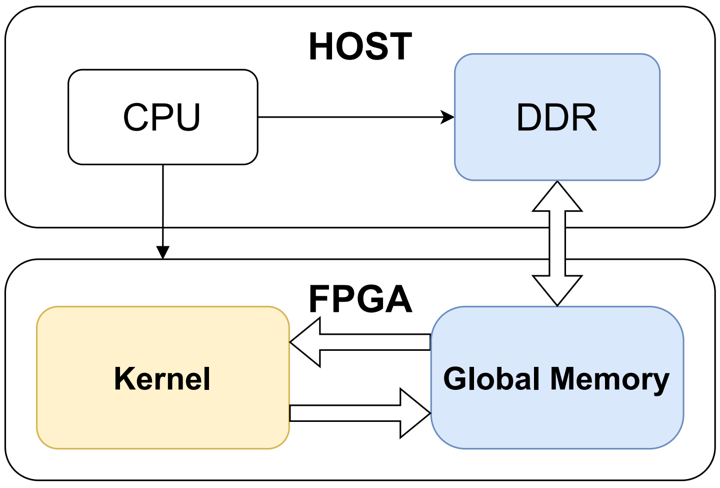

Figure 4.1: Basic overview of system design implementing an FPGA attached to a host.

In Figure 4.1 as well as following schematic designs the thick white arrows represent interfaces that perform transactions with data, like AXI-Streams and AXI4. The smaller black arrows indicate a single signal or an AXI-Lite interface.

4.1.1. Execution model

In the UML-diagram shown in Figure 4.2, the order of operations when performing a benchmark are shown. Before the start of a measurement, compressed data will be memory mapped from host memory to global memory through the available OpenCL binding. With the kernel created and the compressed data ready in HBM, the kernel can be started. The processor within the kernel starts parsing headers and decompressors are started to work on the compressed data until the end of the data is encountered. Once the kernel signals it is done decompressing, the data inside the FPGA is copied back to the host. The uncompressed (file before compression) and decompressed file (from the FPGA) are compared for equality. When everything is found to be correct, a report is setup including the configurations and amount of cycles the processor on the FPGA counted for the whole process to complete.

The execution of this script performs a single measurement and can be included in a bigger benchmark script that executes multiple measurements with different configurations. As the host does not perform any necessary operations on the data, except for issuing transfers, the data could easily come directly out of storage instead of being loaded into memory before a transfer.

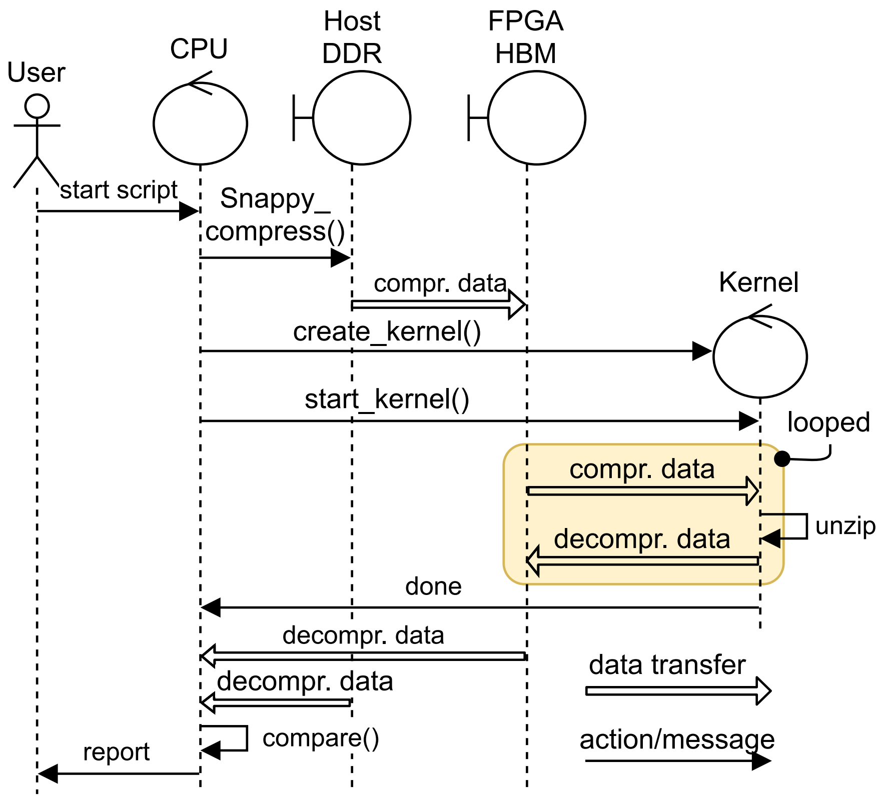

Figure 4.2: UML sequence diagram of toplevel design.

4.1.2. FPGA kernel

A kernel, in the context of FPGAs, is a piece of preconfigured logic that can be loaded on the FPGA and executed as a program. It is the result of the synthesis performed on a design that describes the logic the FPGA should implement. A kernel can be connected to other kernels or the interfaces of the FPGA, such as AXI ports or memory. Kernels can be scheduled for execution by the host and can be placed in parallel, as long as there are interfaces and resources available on the FPGA. A basic kernel would be the synthesis of an adder and a more complex kernel would be a full neural network.

A kernel can relatively easily be set up using the Vivado RTL kernel generation wizard. The wizard lets you set initialization parameters, such as scalar variables which the host might provide to the kernel and parameters that indicate how many connections and individual pointers to the global memory the host and kernel should share. It also allows you to set the type of description you will provide for the kernel, having the option between RTL and block design. As discussed in Section 3.2.3 the block design option will be selected here.

For the measurements many different kernels will have to be created. All the kernels will consist of three modules, shown in Figure 4.3:

1. One or more DMAs, for reading and writing to global memory (HBM)
2. One or more engines, performing operations on the data
3. A MicroBlaze, to monitor and control components

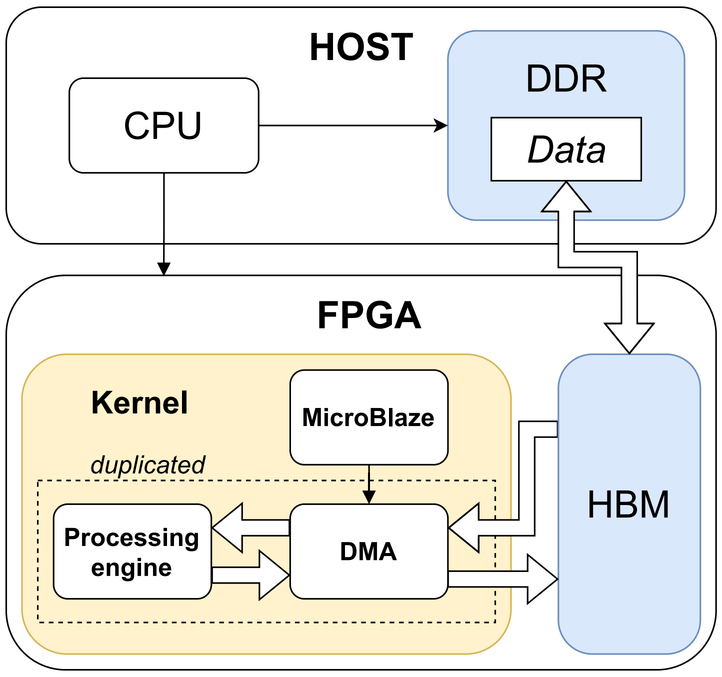

Figure 4.3: Basic overview with the toplevel used modules inside kernel.

Each component will be discussed in more detail in the next section, Section 4.2. A kernel will get at least one `reset` and `clk` (clock) signal externally. More `clk` signals can be created when components need to run asynchronous on higher or lower frequencies. However the DMA, MicroBlaze and vhsnunzip all share a maximum frequency of about 250 MHz and will therefore be connected to the same external `clk` signal.

4.2. Module design

4.2.1. Direct memory access

For any decompression to take place data has to be retrieved from memory. Once decompressed it can be directly fed to an application, like a filter, but can also be stored in memory for later use. Since the results of decompressed data has to be tested for equality on the host it is directly written back to global memory. To perform these read and write transaction without much overhead from a processor Xilinx provides a Direct Memory Access (DMA) IP.

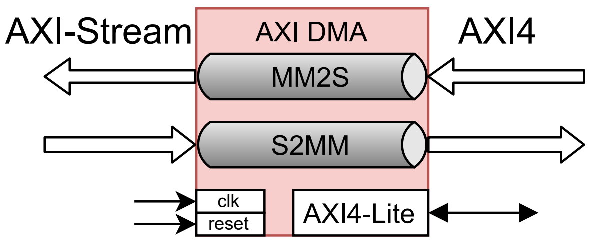

Figure 4.4: Schematic view of AXI DMA block

The DMA IP uses the different AXI protocols for its in and outputs. Through an AXI-Lite port its internal registers can be accessed to monitor the status of the DMA and set up transactions. It grabs data from memory with an AXI interface outputs data as an AXI-Stream. When the length and starting address of a block of data is provided the DMA starts reading data at consecutive addresses from the address. For a write it expects data as an AXI-Stream at its input. Again with a provided length and destination address it start writing data on consecutive address in memory. A very simple application for a DMA using both channels would be a memory-copy where the stream of data produced from a read is directly fed to the input channel of the write. In this the length of both transactions would be the same and the destination address the address the data is to be copied to.

Since essentially a read transforms the data from being memory-mapped to a stream the channel is often abbreviated to *MM2S*. In the same way a write transaction is performed through a *S2MM* channel. When the configuration of the S2MM and MM2S channels, such as the bus width, are identical the AXI interface connected to the global memory can be merged to a single AXI interface. This is because a single AXI interface has independent read and write channels in its specification, as explained in Section 2.2. In Figure 4.4 a DMA block is shown where on the right memory would be attached on the AXI interface. Figure 4.6 shows the block as an IP as it appears in Vivado. A DMA can be configured using polling or interrupt mode. As the MicroBlaze will not be computing anything intensively and most of the time will be waiting for a DMA to go idle, it will be configured for polling mode.

To perform a transfer the DMA needs to know an address to read from/write to and the amount of data that is transferred. Once the length of the transfer is written to the specific register it starts the transfer. Once a read transfer is completed the DMA asserts the `last` signal of the stream and goes idle. For the write channel it keeps writing until the `last` signal of the input stream is asserted. Once the number of bytes written exceeds the number of bytes it was configured to write without the `last` signal being asserted by the input, it goes into an error state.

The DMA can additionally be configured to allow unaligned transfers. In an unaligned transfer the address being read from does not align with the width of the data stream. Since the length of a Snappy chunk can be any number of bytes, the start of the second or later chunk, could be on any unaligned address. In the same way the decompressed data needs to be able to be written to any unaligned address. To allow these unaligned chunks to be easily read without additional logic, the DMAs in the design will be allowed to perform unaligned transfers.

The DMA IP is a wrapper of multiple components that together allow easy transportation of data, see page 5 of [35]. One of these components is the AXI Datamover. It is the main component of the DMA IP, but uses commands in the form of AXI-Streams to schedule transfers instead of reading from a register. The DMA IP allows registers to be used instead of AXI-Streams by converting between the two internally. A control unit can write to these registers through the AXI-Lite port.

4.2.2. Processor configuration

As discussed in Section 3.2.1 a softcore processor, which Xilinx provides and is named MicroBlaze, will be implemented to perform the controlling parts of the design. The code executed on the MicroBlaze is written in `C` and compiled using Xilinx's Vitis Software design platform. The fact that can easily be reprogrammed is its main advantage, at the cost of possibly taking a few more cycles to complete some operation. The reprogrammability is especially useful when, for example, the headers of some engine would be updated in their specification, the code for the MicroBlaze can be relatively fast updated accordingly, as compared to an HDL implementation. Through this platform the block design is synced to the software project by generating header files with settings such as AXI-addresses and including certain drivers for IPs.

In Listing 1 the important parts of `C`-code compiled to run on the MicroBlaze is displayed as pseudocode. The first thing the MicroBlaze should perform, once it boots up (which occurs when the `reset` signal of the kernel is deasserted), is to poll the control registry for the start bit which the host sets to signal that the kernel can start. Before giving the start signal the host will set the scalar and pointer variables for the kernel in the control registry. Once the start signal is given, the MicroBlaze can read those and perform the operations that depend on those. Before receiving the start signal, the MicroBlaze can set up some internal arguments and initialize peripheral components that do not depend on these arguments, such as a timer.

Once the host has written the necessary arguments and signaled that the kernel can be executed the MicroBlaze can read and parse the arguments, performed on Line 5. Now that the MicroBlaze knows

```
init_controlRegister_timer_AXIStreamSwitch_DMAs();

while(1) {
    wait_for_start_signal(); // blocking
    get_kernel_args();
    startTimeFull = reset_timer();

    for (loopCount = 0; loopCount < krnlArgsLoop; loopCount++) {
        reset_DMAs();
        startTimeLoop = get_Time();

        set_switch_mblaze();
        headerList = get_chunk_headers(krnlArgsBuffersize);
        set_switch_vhsnunzip();

        headerData =  get_next_header(headerList);
        while(headerData != 0) {
            DMAPtr = get_idle_dma(); // blocking
            start_DMA(DMAPtr, headerData);

            advance_header_list(headerList);
            if(headerList->tail == NULL) {
                headerList = get_chunk_headers(krnlArgsBuffersize);
            }
            headerData = get_next_header(headerList);
        }
        wait_dmas_done(); //blocking
    }

    endTimeLoop = get_time() - startTimeLoop;
    endTimeFull = get_time() - startTimeFull;
    write_times();
    signal_done();
}
```

Listing 1: Pseudo-C-code executed by MicroBlaze.

where in global memory the compressed data is located it can begin a decompression. Before it can configure the DMAs to stream data to the vhsnunzip modules it needs to know what the compressed and decompressed length of each chunk is. For the MicroBlaze to retrieve this information it has to use a DMA for itself to transfer data from global memory into its own local memory. Since the DMAs that are connected to the decompressors are not doing anything yet one of them can be ”borrowed” by inserting a switch in the stream and temporarily setting the switch to flow to the MicroBlaze. This process is further explained in Section 4.2.7. With the size of a Snappy header fixed at 64 bits, as described in Section 2.5.2, The MicroBlaze can now set up a request for a header through a DMA. From this header it parses the location of a chunk and its compressed and decompressed size. With this information it can set up a DMA to read data into a decompressor and write the decompressed data back to global memory. This process continues until it finds a header that indicates the end of a file at which point it waits until all DMAs are done transferring their data. Once completed, the MicroBlaze calculates the amount of cycles the whole process took and writes it to global memory for the host to retrieve. At the end, the done signal is asserted so the host knows it can start using the decompressed data.

4.2.3. Buffer containing parsed headers

From a parsed header the location of the next header is known. Once header is parsed a decompressor can be started. To be able to immediately start a decompressor once it completes multiple headers should be parsed. If a header is only retrieved from HBM and parsed once a compressor finishes, a decompressor would have to remain idle until a header is successfully parsed. Although it is possible that *all* headers can be read in one go and stored in the local memory of the MicroBlaze this may cause an overflow when the chunk size is small and the file itself is very big. If, for example, a parsed header would require 16 bytes of local memory and the compressed file is 64 MiB in total, consisting of chunks of 1 KiB in size this would require $16 * 64 * 1024 * 1024/1024 = 16 * 64 * 1024 = 1$ MiB of local storage. Since the MicroBlaze has a maximum of 128 KB of local storage, the kernel would crash somewhere during parsing. So instead of parsing all headers in one go a maximum buffer size (configurable by the host) is set. The MicroBlaze will parse as many headers until this buffer is full and start parsing new headers once the buffer is empty.

As this buffer has to be dynamic to support the configurable buffer size, a linked list is implemented. A node in this list stores the parsed header information. The `head` indicates the most recent chunk that has been parsed. The `tail` points to the node that should be decompressed next. Each time a node is used to configure a DMA, the `tail` can be advanced. Once the `tail` points to a node that is uninitialized (`NULL` in `C`) the buffer is known to be empty.

Now that the location and size of chunks is known the DMAs can be configured. The MicroBlaze can now enter a loop, at Line 17, where the following happens sequentially: First it waits until it can find an idle DMA. It will configure it and start using the information in the node the `tail` of the linked list points to. It will advance the `tail` pointer and check if the next `tail` pointer points to a valid (non-`NULL`) pointer. If it is `NULL` it will retrieve the next batch of headers. Otherwise it continues with the next loop until a node indicating the end of the file is retrieved. It then waits for all DMAs to finish their jobs, indicating that the decompressed data is written to the global memory.

4.2.4. Hardware Snappy decompressor

As mentioned in the background section on the vhsnunzip module, Section 2.5.3, there are two types: the unbuffered and buffered version. The main difference are the throughput, stream width and ability to parse chunks larger than 64 KB. For the purpose of measuring the effect of parallelization of an application, it is import to have the ability to parse long chunks to see how scaling from small to large data influences the throughput. The base throughput matters less since it is the effect, rather than the final throughput that is of importance. The stream width does matter for saturating the available bandwidth. However when placing multiple unbuffered modules the same stream width can be achieved as using the buffered version. As such for the measurements the decompressor is chosen to be the unbuffered version of the vhsnunzip module. Through Vivado the HDL files within the library of [23] are compiled into an IP that can be added to the block design of the kernel. The resulting block from the packaging is shown in Figure 4.5.

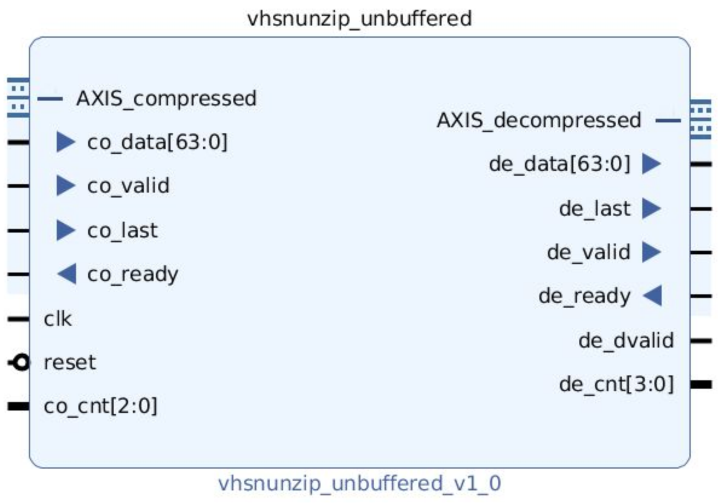

Figure 4.5: Hardware Snappy decompressor, vhsnunzip, as seen in Vivado block design.

Once a DMA is started it will output data over an AXI-Stream and expects data to flow in as an AXI-Stream as well. The vhsnunzip module however uses vhlib-streams, which are not compatible (yet). In most cases all the bytes of the compressed stream easily flow into the decompressor and come out decompressed a few cycles later, except for the bytes in the last packet. When the total amount of bytes cannot be equally divided over the width of the data stream, a packet will consist of bytes that are part of the data and bytes that should be ignored. Such a packet is named a sparse packet. The vhsnunzip module uses a stream from vhlib (using `cnt`), which handles this in a different way from the DMA that streams AXI-Streams (using `keep`), as explained in the background on vhlib in Section 2.2.4. The difference can be seen by comparing the stream signals shown in Figure 4.5 and Figure 4.6. Normally, the outgoing stream of the DMA named `M_AXIS_MM2S` would connect to the input of the vhsnunzip module at `AXIS_compressed`. However the output signal `s_axis_mm2s_tkeep[7:0]` from `M_AXIS_MM2S` does not have an input signal on `AXIS_compressed`. Instead the `co_cnt[2:0]` is available, with a different width of 3 as opposed to the width of 8 at the side of the DMA. To convert this part of the streams from `keep` to `cnt` at the input of the vhsnunzip and from `cnt` to `keep` additional logic has to be created and implemented. For this a mix between the available IPs from Xilinx and self-written VHDL converted to IPs are used. The individual components that enable the conversion between keep-to-cnt and the other way around are described in the next sections.

4.2.5. Custom keep-to-cnt conversion

In order to convert the AXI-Stream from a DMA to the vhlib steam of a decompressor, the `keep` signal has to be converted to a `cnt` signal. Under the condition that the data-bytes from the DMA are always consecutive, meaning no to-be-ignored bytes are between bytes representing data, this conversion can be achieved. As mentioned in Section 4.2.1 the AXI Datamover is the main component of the DMA IP. On page 44 of the Datamover manual [36] it says: ”... the read data is aligned such that the first byte read is the first valid byte out on the AXI4-Stream.” This means that always the first byte will be a valid byte. There is no statement about the bytes that follow it, but after examination it shows that the DMA will always produce consecutive valid bytes, meaning this condition holds.

With these consecutive data-bytes the `keep` signal should always be a sequence of 1’s followed by a sequence of 0’s. For example, in Figure 4.7, for the second packet the `keep` signal `0b1110` means the first 3 bytes (`0x89`, `0xAB` and `0x45`) are data bytes and the last (`0x67`) can be ignored. In this example the `cnt` equivalent would be `0b11` or `0x3`, to indicate the first 3 bytes are data. This encoding is named thermometer code or unary coding and is implemented in a custom IP named *keep_to_cnt*, abbreviated to *k2c* in Figure 4.7.

Since a `NULL`-packet, where the whole content of a packet should be ignored, is not supported by both the DMA (page 34 AXI Datamover manual [36]) and vhsnunzip (uses alternative signal) the `cnt` signal for `0x0` is used to indicate all bytes are a data byte. With this the width of the `cnt` bus can remain 3.

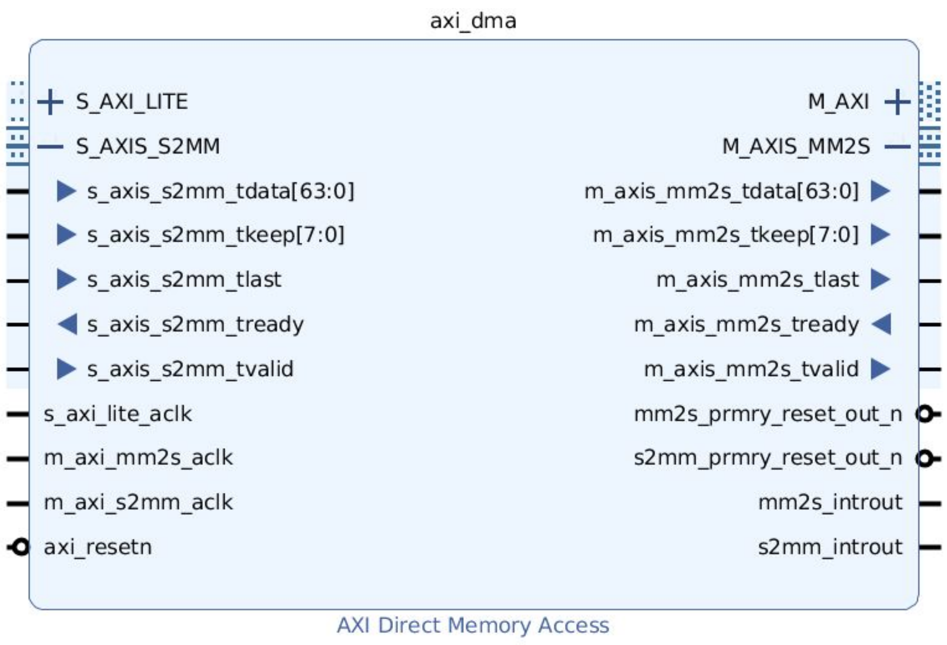

Figure 4.6: AXI DMA as seen in Vivado block design with AXI-Stream signals expanded.

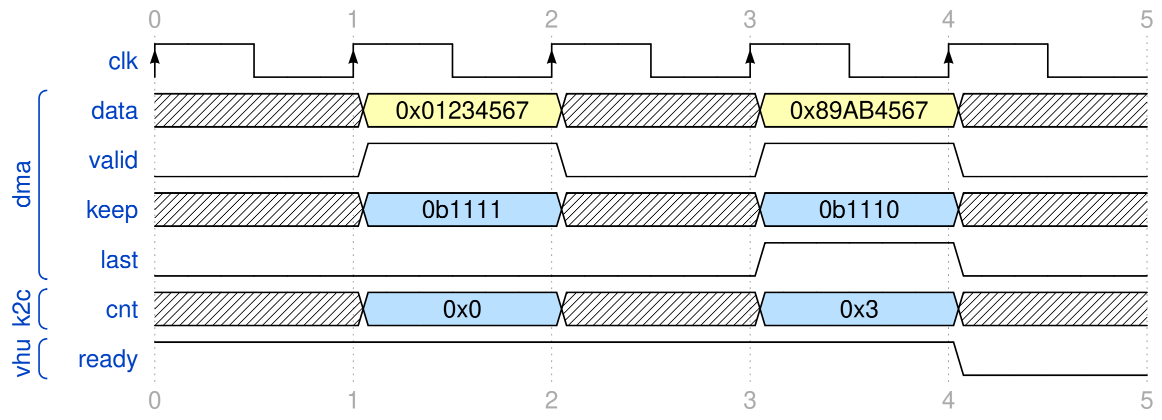

Figure 4.7: Waveform of `keep` from DMA being converted to `cnt` going to vhsnunzip.

4.2.6. Custom cnt-to-keep conversion

The conversion between the stream from a DMA to a hardware decompressor is more complex. As can be seen in Figure 4.5 the module has an additional signal named `dvalid` at the output. When `dvalid` is asserted the contents of the package should be considered data. If it is deasserted during a transaction it is a `NULL`-package. The fact that the output stream contains `NULL`-packages is an issue, since the datamover component of the DMA does not support them and will ignore the packet when encountered. From simulation it is found that vhsnunzip uses `NULL`-packages to carry the `last` signal of a transfer, since a `last` signal has to be accompanied with a packet. This is useful when the vhsnunzip realizes the transfer has completed after already having sent its last packet. An example of such a transaction is shown in Figure 4.8. When at clock-cycle `4` a packet arrives where `keep` is `0b0000` the DMA ignores it and waits for a `last` signal it will never get. So `ready` is never deasserted and the DMA never goes idle. A few solutions are available and attempted.

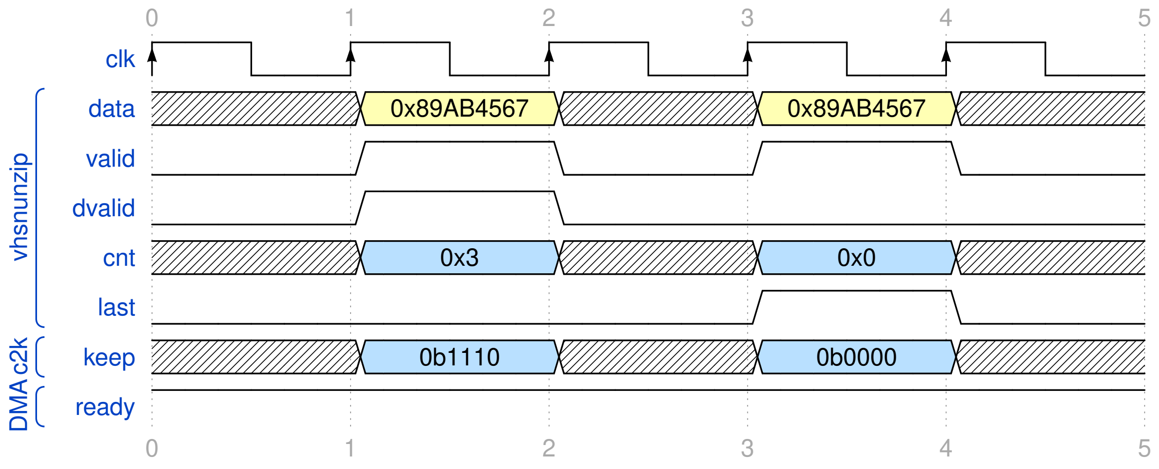

Figure 4.8: Waveform of `cnt` from vhsnunzip being converted to `cnt` going to DMA resulting in a crash.

A naive solution is to transform the `NULL`-packet into a regular packet. In the example the `0b000` of `keep` could be set to `0b1111`. This would result in an additional packet to be written to global memory with garbage data. When chunks are written sequentially this garbage data would be overwritten by the first package of the next chunk, effectively cleaning the garbage data. Unfortunately this is not the case since chunks are to be parsed and written in parallel instead of sequential. Therefore the garbage-package will overwrite the start of a chunk, corrupting it.

The implemented solution makes use of a AXI-Stream Register Slice IP as shown in Figure 4.10. This IP can be considered a mini FIFO (first-in first-out) buffer, that can hold a maximum of two stream packages. The idea is that if the `NULL`-packet, at cycle 4 in Figure 4.8 can be identified before the packet, at cycle 2, has been fed to the DMA, the `last` signal can be added to the earlier packet. The `NULL`-packet, which now serves no purpose anymore, can be discarded before it reaches the DMA. The last packet in the register slice now has the `last` signal that the DMA expects. The addition of the register slice enables this as the sparse packet and the packet that receives the `last` signal and the `NULL`-packet can both be seen before they are processed by the DMA. In the example of Figure 4.8 this would mean that when the packet at clock-cycle 4 would be at the output of the vhsnunzip while packet at clock-cycle 2 is in the register slice. The `last` signal will be added at the output of the register slice and the packet at the output of the vhsnunzip may not enter the register slice. Using the register slice makes a transaction require a few more cycles, which is negligible with the thousands of packets being sent per transaction.

Identifying a `NULL`-packet is achieved by reversing the *keep_to_cnt* module and extending it with the `dvalid` and `last` signals on the input. To make able to specifically identify the moment a transaction will take place it will additionally be extended with the `ready` and `valid` signals of the stream. Now when a `cnt` of `0x0`, a `dvalid` of 0, a `last` of 1, a `valid` of 1 and a `ready` of 1 is found, the packet is a `NULL`-packet and a `null_pkt` signal can be asserted. Figure 4.9 shows a `NULL`-packet being detected and the assertion of the `null_pkt` signal when the conditions are met. This detector, along with a conversion of `cnt` to `keep` is combined in a *cnt_to_keep* IP.

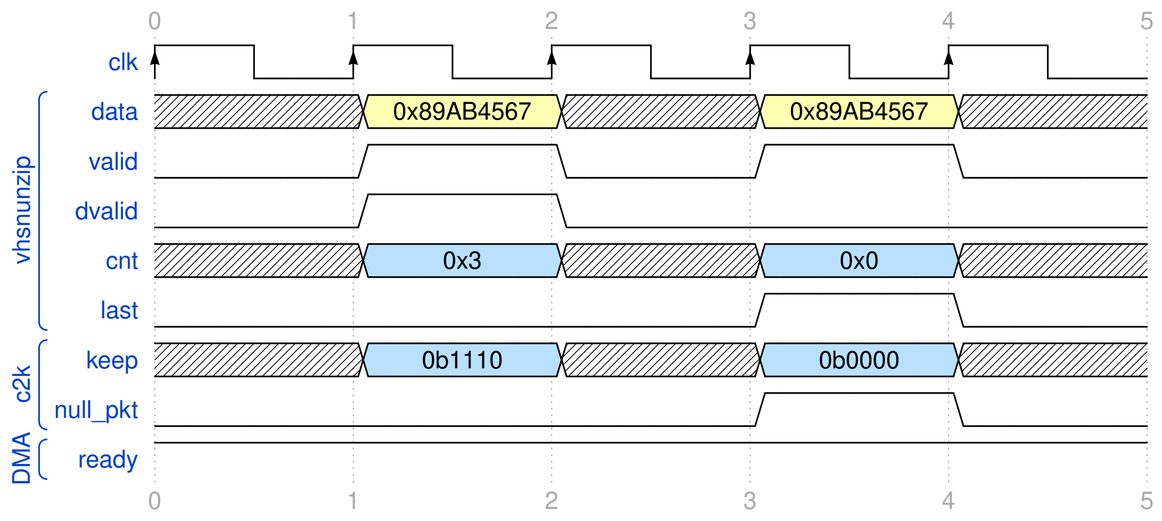

Figure 4.9: Extended version of Figure 4.8 adding the `null_pkt` signal

With the `null_pkt` signal available the `NULL`-packet can be discarded by deasserting the `valid` signal from the vhsnunzip to the register slice. When a regular packet is transferred the `valid` signal should behave as normal. This leads to the following logic, implemented with IPs in the block diagram at the input of the register slice:

$$\texttt{valid\_RegSlice} = \overline{\texttt{null\_pkt}} \land \texttt{valid\_vhsnunzip} \tag{4.1}$$

With the `NULL`-packet found and discarded, the `last` signal now has to be added to the packet that was before the `NULL`-packet. The packet that should have an asserted `last` signal is either in the register slice (waiting on an other packet) or being outputted by the register slice. The easiest way to figure out which of the two packets in the register slice is the actual last one is through counting the packets that left the vhsnunzip and the packets that left the registers slice. If the difference between these one or less, there are no packets buffered up in the register slice. Combining this with the fact `null_pkt` has been asserted, the current packet should have it's `last` asserted.

To implement this in the block diagram some logic is required. For the counting of the packets two counters of width 2 are added. For checking the requirement of the difference being one or less the difference between the two counters is calculated and the MSB is inspected. The `null_pkt`, once encountered, has to be kept asserted until the last packet has been fed to the DMA. This can be achieved by a counter with width 1. Finally, at the start of a new transaction all the counters have to be reset, this can be done using the `last` signal when it is asserted for the last packet.

Combining everything the `cnt` and `dvalid` signals have successfully been converted to a `keep` signal and removed the `NULL`-packet. The block diagram implementing the two custom IPs can be seen schematically in Figure 4.10 and implemented in Vivado in Figure A.2 in the Appendix. The contents of the `cnt_to_keep` block contain too many IPs and connections to be shown on a single page and is therefore not included but can be viewed using Vivado.

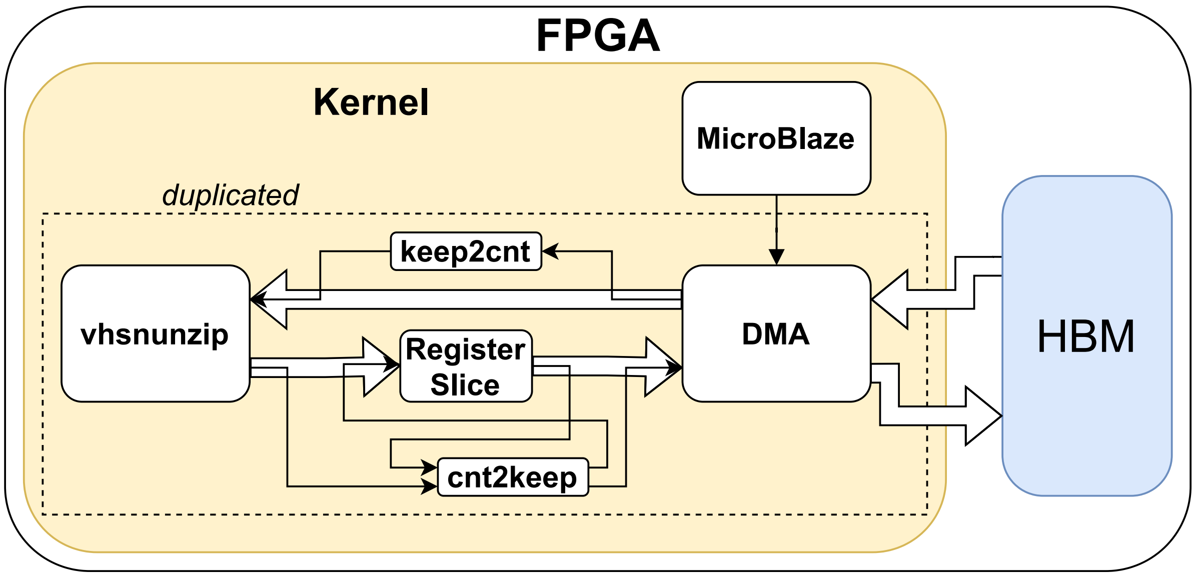

Figure 4.10: Schematic diagram of the `cnt_to_keep` and `keep_to_cnt` blocks added between a vhsnunzip and a DMA within a kernel.

4.2.7. Transferring data between HBM and MicroBlaze local memory

The MicroBlaze needs to be able to read headers from global memory and write the measured cycle amount back for the host to retrieve. The MicroBlaze can initially only interact with its own local memory within SRAM. In order to enable the MicroBlaze to interact with HBM one of the already placed DMAs can be used while it is idle. To make this possible AXI-Stream switch IPs from Xilinx's library are added to the one of the DMAs. In Figure 4.11 is the addition of switches to Figure 4.10 shown.

A switch is added from the DMA to the decompressor since an any-cast configuration using an interconnect will not work as header data would be grabbed by the decompressor because it will grab any packet it can (has `ready` asserted). Broadcasting the stream to both modules could work but then some logic is needed that would prevent the vhsnunzip from accepting a header as potential compressed data.

The AXI-Stream Switch IP can be configured with a table and logic before synthesis or using control registers during execution. The table routing uses arbitration such as Round-Robin or fixed-priority to select which slave gets a packet at which time. This is mostly useful when you want to distribute individual packets over multiple components. In the case of the MicroBlaze and vhsnunzip having to share a stream, the easiest option is to use the control register routing. This is because the MicroBlaze already knows when it wants to get headers and the vhsnunzip is not using the DMA anyway. For this an AXI-Lite connection is used between the MicroBlaze and switch.

Another case of having to share a stream occurs when the MicroBlaze wants to write the timings it measured to global memory. Unlike the previous case, the sharing of this channel concerns a 2-to-1 connection. Once decompression has completed no data will flow from the vhsnunzip anymore. The only data the DMA can now expect is that from the MicroBlaze and the only way it can flow is into the DMA. Therefore the Switch can be kept in a table-routing configuration without any specific table.

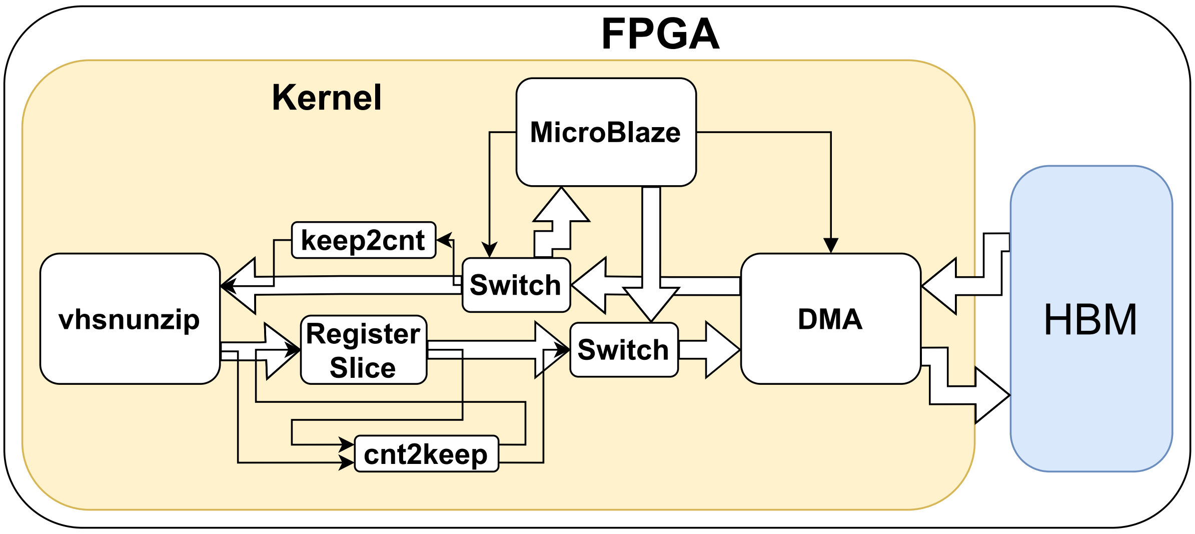

Figure 4.11: Architecture of Figure 4.10 with added switches.

In the final design, one implementation as shown in Figure 4.11 should be placed and multiple blocks containing the part within the box annotated by *duplicated* in Figure 4.10 should be included. In the Vivado block diagram shown in Figure A.1 for example, one DMA including the streams to the MicroBlaze are placed and three modules without.

4.3. Vivado block diagram

In Figure A.1 a configuration with four decompressors is shown. In this the DMA and vhsnunzip modules are combined in one package. One of the DMAs is shared with the MicroBlaze to let it read and write to a bank of HBM. When compared to the other DMA blocks, this DMA has an extra AXI-Stream flowing in and out to the control block. Between the control block and the DMAs an interconnect block, named Smartconnect, allows the MicroBlaze to use a single AXI-Lite interface to interact with each DMA.

In the configuration used for measurements sixteen DMAs are placed in parallel, instead of the shown four. Only four are shown, as a design with more blocks becomes hard to read on a single page. The design can easily be extended to include more DMAs by copying a block and adding an AXI-Lite interface to the interconnect, for each extra DMA. The address of the DMA has to be added to the code executed on the MicroBlaze as well, so it knows where to sent AXI-Lite transactions to.

In Figure A.2 the contents of a package with a DMA and vhsnunzip module are shown, without the added AXI-Stream switches. In this diagram the pink line shows the path of the `keep` and `cnt` signals. An inverting block is placed before the `reset` signal of the vhsnunzip block since its reset is triggered by an assertion whereas the rest of the IPs have their reset triggered by a deassertion. Under the inverting block the `keep_to_cnt` block is placed where its signal is placed between the output of the DMA and the input of the vhsnunzip block. The `cnt_to_keep` block is placed between the output of the vhsnunzip block and the input of the DMA.

4.4. Baseline configuration

4.4.1. Baseline kernel

To compare the resulting throughput from the implementation of the Snappy decompressor a baseline measurements has been constructed. A useful baseline measurement is the throughput of an HBM bank when one or many parallel accesses are performed on the same or different banks. In this baseline measurement, data should always be present to be written to HBM or ready to be read from HBM such that only the throughput of the combination of the HBM, DMA and MicroBlaze is measured not the generation or accepting of data. Additionally the reading of data should be independent of the writing of data so possible differences in the throughput of storing or retrieving from HBM can be observed.

The baseline kernel is almost identical to the architecture in Figure 4.3 but with replacement parts for the vhsnunzip module. The stream of data being read from HBM and outputted by the DMA will be left unconnected except for the `ready` signal, which will be kept asserted to implement a data sink. At the input of the DMA a traffic generator is placed that, at every clock cycle, has a stream of constant data available and asserts a `last` signal as configured by the MicroBlaze.

A schematic overview of the baseline implementation can be seen in Figure 4.12. The implemented Vivado block diagram can be seen in Figure A.3. In the configuration in Figure 4.12 the throughput of writing and reading to a bank within HBM is influenced by many factors such as the kernel frequency, the width of the AXI-bus, amount of parallel read/write requests and the properties of the HBM IP. Most of those factors can be configured or are known before execution.

With one DMA being connected to one AXI-ports of the HBM IP and 16 AXI-ports available per stack, at most 32 DMAs can be implemented in this configuration. However, between the two stacks a relatively small interconnect is available, as sketched in Figure 2.7. Through this smaller interconnection data can still be retrieved or stored from an other stack than the stack of which an interface to HBM is used. Within the specification of the HBM IP [29] it is stated: ”The shared connections limit the maximum throughput laterally to 50% of the full bandwidth, but enables global addressing from any AXI port to any portion of the HBM.”. Therefore a single kernel will be limited to use banks from one stack, limiting the DMAs to 16. Since the vhsnunzip module works with AXI-Streams with a width of 8 bytes the baseline will implement that as well. The kernel frequency will be left at 250 MHz as this is suitable for the DMA and MicroBlaze as well as the maximum frequency of the hardware Snappy decompressor.

Originally a traffic generator IP created by Xilinx was used. However the driver for the MicroBlaze required a lot of local memory per IP added to the design. Presumably the driver tried to allocate space on the stack to store a preconfigurable stream of data. With 16 such IPs the compiled program would not fit on the MicroBlaze anymore. As the data itself did not matter a custom traffic generator was created that outputted a single constant (`0xABCD`).

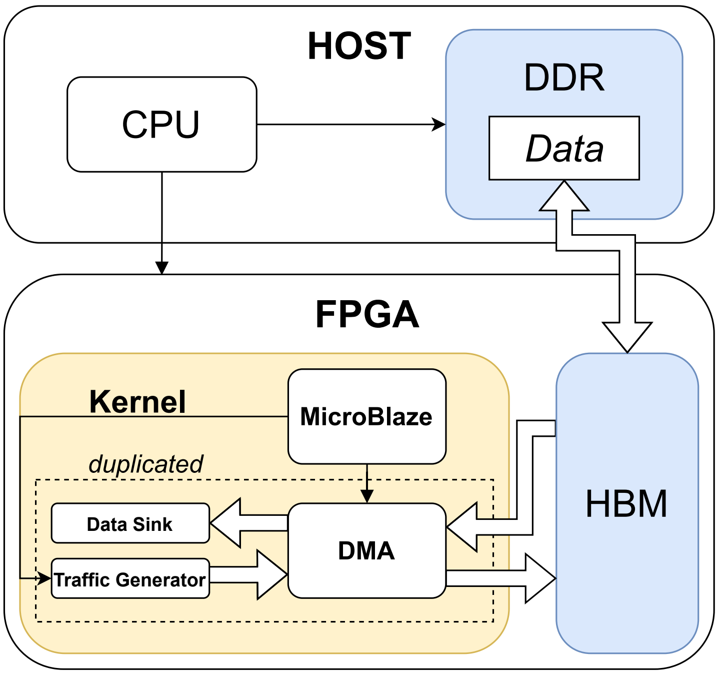

Figure 4.12: Overview of baseline configuration

4.4.2. DMA to HBM configurations

The duplication of the DMA module of Figure 4.13 means that many DMAs will be connected to HBM. With each DMA making two connection to HBM, even more distinct configurations can be created. Of all possible configurations, three configurations are used, shown in Section 4.4.1. In each configuration shown in the figure only the first few banks and DMAs are shown. In *Configuration 1* each channel of a DMA is connected to an individual bank in HBM. In *Configuration 2* both channels of a DMA are connected to the same bank, making the bank perform both read and writes. In *Configuration 3* the read and write channels of multiple DMAs are connected to a single port for reading or writing. In this configuration only two banks are used per kernel. Each configuration has its advantages and disadvantages and those will be discussed with the results of their measurements in Section 5.2.

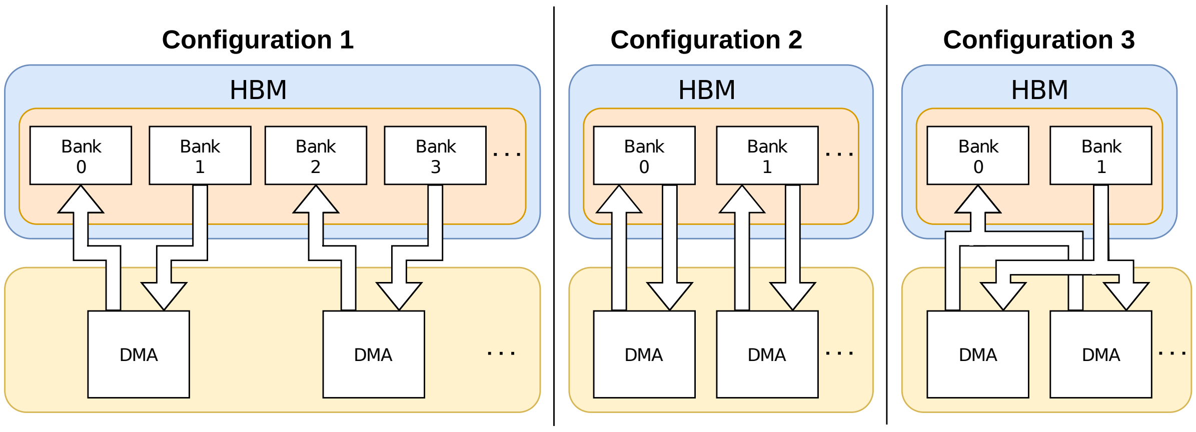

Figure 4.13: Three strategies for connecting DMA channels to HBM banks

4.4.3. MicroBlaze code

The code executed by the MicroBlaze performs almost the same steps as the code shown in Listing 1. The for-loop on line 8 of Listing 1 is replaced with the pseudo-code shown in Listing 2. It supports starting a variable amount of DMAs that can either read, write or read and write.

Before the DMAs are started a 32 bit signal is sent to all traffic generators telling it how many bytes it has to generate before asserting a `last` signal. It is also used to reset a counter that monitors how many packages it has sent.

```
for (loopCount = 0; loopCount < krnlArgsLoop; loopCount++) {
    start_TrafficGenerator();
    for (indexDMA = 0; indexDMA < krnlArgsActiveDMAs; indexDMA++) {
        switch(transferMode) {
            case READ_ONLY:
                start_DMA_channel(indexDMA, S2MM, krnlArgsTransSize);
                break;
            case READ_WRITE: //Fallthrough!
                start_DMA_channel(indexDMA, S2MM, krnlArgsTransSize);
            case WRITE_ONLY:
                start_DMA_channel(indexDMA, MM2S, krnlArgsTransSize);
                break;
        }
    }
    wait_dmas_done(); //blocking
}
```

Listing 2: Pseudo-C-code executed by MicroBlaze for baseline measurements.

4.5. Vivado block diagram

Figure A.3 shows a kernel implementing four DMAs and peripherals. It is nearly identical to Figure A.1, discussed in the previous section. The only difference is a small 32 bit wide signal flowing out of the control block into each DMA block. This signal is used to tell the custom traffic generators inside each block how many bytes they should generate before asserting the `last` signal.

In Figure A.2 the internals are shown which contain the custom traffic generator and a single constant value asserting the `ready` signal on the output of the DMA.

5 Measurement results

5.1. Experimental setup

5.1.1. Hardware

The goal is to measure the time it takes for a certain compressed file to be fully decompressed inside HBM. Additionally the decompressed file has to be compared on the host for correct decompression. To perform the hardware decompression of data in HBM the Xilinx Alveo U280 Data Center FPGA, shown in Figure 5.1 is available for testing and measurement purposes on a server provided by the TU Delft. This card is built with the 16 nm Ultrascale+ architecture and contains an HBM2 module with two stacks totaling 8GB of HBM. Additionally 32 GB of DDR is available for a total of 40 GB of global memory. As calculated in Section 2.4 the theoretical maximum bandwidth of the HBM2 global memory is 460.8 GB/s. These and more resources are shown in Table 5.1. For the implementation and measurements this card will be used while it is hooked up at a server. Consequently physical cables, such as JTAG or USB, can not be attached for debugging purposes. Vivado 2020.1 along with Vitis 2020.1 are used for designing, simulating and synthesis of FPGA components. Interaction of the host with the FPGA is performed through `C++` code using the OpenCL API to flash a kernel on the FPGA and transfer data to and from the FPGA.

The server itself uses a dual-socket configuration with two Intel Xeon Silver 4114 CPUs that have a maximum clock frequency of 3.0 GHz. Each CPU has 10 cores that provide 2 threads each for a total of 40 threads available on the server. The CPU has a 13.75 MB L3 cache. For memory the server holds 6 DIMMs of 16 GB each for a total of 96 GB. They operate at 2666 MHz and a 64 bit bus-width, making their (theoretical) maximum read/write bandwidth $\frac{64}{8} * 2666 * 10^{-9} = 21.33$ GB/s per DIMM.

The created scripts and designs are made available on Github at [37].

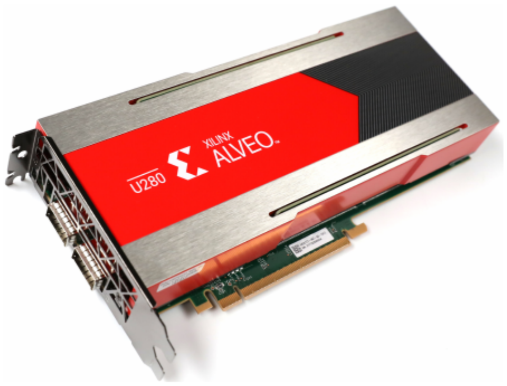

Figure 5.1: The Alveo U280 card that is plugged in the server at TU Delft. Picture from [5]

Component	Specification
HBM2 total capacity	8 GB
HBM2 total bandwidth	460 GB/s
Look-up tables (LUTs)	1,304K
Registers	2,607K
DSP slices	9,024
Block RAMs	2,016
UltraRAMs	960
DDR total capacity	32 GB
DDR maximum data rate	2400 MT/s
DDR total bandwidth	38 GB/s

Table 5.1: The hardware specification of the U280 as written in [7]

5.1.2. Compressed data

Throughput can be measured at only the output of the decompressors or as a combination of the input data and output data. As mentioned in Section 2.5, Snappy is a dictionary-based compression algorithm. When a lot of data is identical within a chunk a high compression ratio can be achieved. This results in a lower end-to-end throughput as a bottleneck will be on the output side of decompressors, as more data is flowing out than in. The maximum throughput will be reached when data with a compression ratio near 1.0 is decompressed with the size of the input data being near the size of the output data. So the compression ratio of a compressed file will matter for the measured throughput.

Another important parameter is the chosen size of uncompressed data chunks. Some data file being split in smaller chunks allows for more parallel tasks to be available. However each chunk has a header that has to be parsed, creating additional overhead.

To generate Snappy files with configurable input data such as files from storage or debugging sequences, and allow variable chunk sizes, a Python script is developed. The script generates a Snappy compressed file using custom chunk headers as described in Section 2.5.2. Additionally the uncompressed data is stored as a file, which can be used for verifying the output of a decompression.

As concluded in [38] it is difficult to reliably evaluate performance of a certain compression algorithm. To get somewhat of a good indication of the performance a set of files is proposed that should be used for compression and decompression, available at [39]. These data sets, named the Canterbury Corpus after the University they were developed at, were developed in 1997 and are not the best representation of currently compressed data. The biggest file in *”The Large Corpus”* is only 4.6 MB in size. A more recent corpus is *The Silesia Corpus*, released by Sebastian Deorowicz as part of his dissertation [40]. This corpus is used as a data set for benchmarks on modern compression algorithms [41] due to its bigger file size. A brief description of the contents of the corpus are displayed in Table 5.2 and the actual contents are available at [42]. The compressed size is from compressing the file using the custom Snappy framing format with a chunk size of 2^{16}.

For measurements both random data (with a compression ratio near 1.0) and files of varying compression ratios will be used. The random data will show the maximum throughput supported by a configuration whereas the files with varying sizes will give a more practical representation of the throughput. Both the Silesia Corpus and 128 MB data files with a custom repetition of bytes will be used to measure throughput on different compression ratios.

Filename	***Type***	***Raw size [B]***	***Compressed size [B]***	***Ratio***
dickens	English text	10.192.446	6.339.554	1,61
mozilla	exe	51.220.480	26.470.530	1,94
mr	picture	9.970.564	5.421.517	1,84
nci	database	33.553.445	6.152.480	5,45
ooffice	exe	6.152.192	4.272.188	1,44
osdb	database	10.085.684	5.331.019	1,89
reymont	Polish pdf	6.627.202	3.234.912	2,05
samba	src	21.606.400	8.012.408	2,70
sao	bin data	7.251.944	6.436.491	1,13
webster	html	41.458.703	20.213.433	2,05
xml	html	5.345.280	1.309.280	4,08
x-ray	raw data	8.474.240	8.210.614	1,03
Total		211.938.580	101.404.426	2,09
Average		17.661.548	8.450.369	2,27

Table 5.2: Contents of Silesia Corpus

In the framing format supplied in the Snappy Github [30] a default of 64 KB as the maximum size of a compressed chunk is specified, allowing for small fixed-size buffers within a decompressor. The hardware decompressor, vhsnunzip, supports bigger chunks and the only difference is a slight increase in resource usage. Therefore, as the size of a chunk can be easily adjusted and will show the ratio of overhead between the parsing of a header and the actual decompression, it will be kept variable and iterated over in measurements. With the custom framing format described in Section 2.5.2 the theoretical maximum size of a chunk is the same as the maximum value of an unsigned 32-bit integer.

Each measurement is performed on a file that has been chopped in chunks of equal size. Except for the last chunk, since a data file does not necessarily have to be divisible by the selected size of a chunk. For measurements the chunk sizes are increasingly multiplied by 2, starting at a size of 1 KB, until but not including the chunk size becomes larger than the size of the uncompressed file.

5.1.3. Measurements

To get the most reliable results, a measurement should not be performed once. In the code executed on the MicroBlaze a loop has been placed around the decompression of a file. The host communicates to the kernel the amount of loops should be performed and, with that, the amount of times the file is decompressed. It is expected that between each loop the deviation in measured time should be very small because the only non-deterministic influence will be the access time of data from HBM. So additional measurements should not result in very different amounts of measured cycles. However, as a single vhsnunzip can decompress at a speed of at least 1 GB/s a single decompression of a file of 128 MB takes less than a second. Performing 100 such measurements would take around 13 seconds to complete. These measurement will only go faster once the size of chunks increase and multiple decompressors are placed in parallel. As it is a reasonable time to wait for measurements to complete multiple amounts of loops, it is set at 100. The total amount of cycles it took to complete 100 loops is averaged to resemble the time for a single decompression to complete.

Each kernel on the FPGA is configured such that it returns the total amount of clock pulses it took the kernel to complete a certain task. In the case of the baseline measurements this is the amount of pulses it took to pull data from HBM and/or write data to HBM. In the case of Snappy decompression it is the amount of pulses it took to pull data from HBM, parse headers, decompress chunks and place decompressed data back in HBM. From the known amount of cycles, the known size of transferred data and the set clock frequency the throughput can be calculated using the equation:

$$\text{throughput [GB/s]} = \frac{\text{data size [B]} \times \text{amount of loops}}{\text{measured cycles} \times 10^9} \times \text{clock frequency [Hz]} \tag{5.1}$$

First, measurements will be performed on the HBM of the Alveo U280 FPGA using the kernels created from the configurations described in Section 4.4.1. The results are the throughput the HBM IP can support when the banks are connected using different configurations and with a different bus-width. Another set of measurements are performed using the kernel described in Section 4.1.2. Finally a similar application is measured that is executed on a processor on the host.

5.2. Baseline FPGA measurements

Using the baseline kernel described in Section 4.4.1 many measurements can be performed. Each DMA can be connected to a different HBM bank or all to the same. Each DMA can have both its reading and writing channel connected to a different bank or both to the same bank. Additionally, one DMA can be used or many DMAs in parallel can be enabled at the same time. Some of these configurations are shown in Figure 4.13. In the following sections each configuration strategy will be measured and discussed. First the kernel will use a 256 bit AXI-bus as this is the maximum width the HBM IP supports. From those measurements the maximum bandwidth of a stack of HBM can be found. However, the hardware Snappy decompressor supports a narrower bus-width of 64 bit. Therefore additional measurements will be made with a 64 bit bus as they are better comparable to the results from the parallel Snappy decompressor.

5.2.1. Results from 256-bit AXI-bus to HBM

From the specifications of the HBM IP discussed in Section 2.4 a single HBM stack is expected to have a throughput of 230.4 GB/s. In the used setup with the Alveo U280 the HBM module consists of two stacks and would be capable of having a theoretical bandwidth of 460.8 GB/s. However, as mentioned in the design of the kernel at Section 4.4.1 the kernel will limit itself to a single stack. Additionally, the design runs at 250 MHz instead of the maximum supported 450 MHz. For a single bank being connected through a single channel with a 256 bit AXI-bus the expected throughput would then become $250 * 10^6 * \frac{256}{8} * 10^{-9} = 8.0$ GB/s per bank. The pseudo channel of the bank will be idle for a few cycles during such transaction as it can support a frequency up to 450 MHz for a maximum throughput of $450 * 10^6 * \frac{256}{8} * 10^{-9} = 14.4$ GB/s per bank.

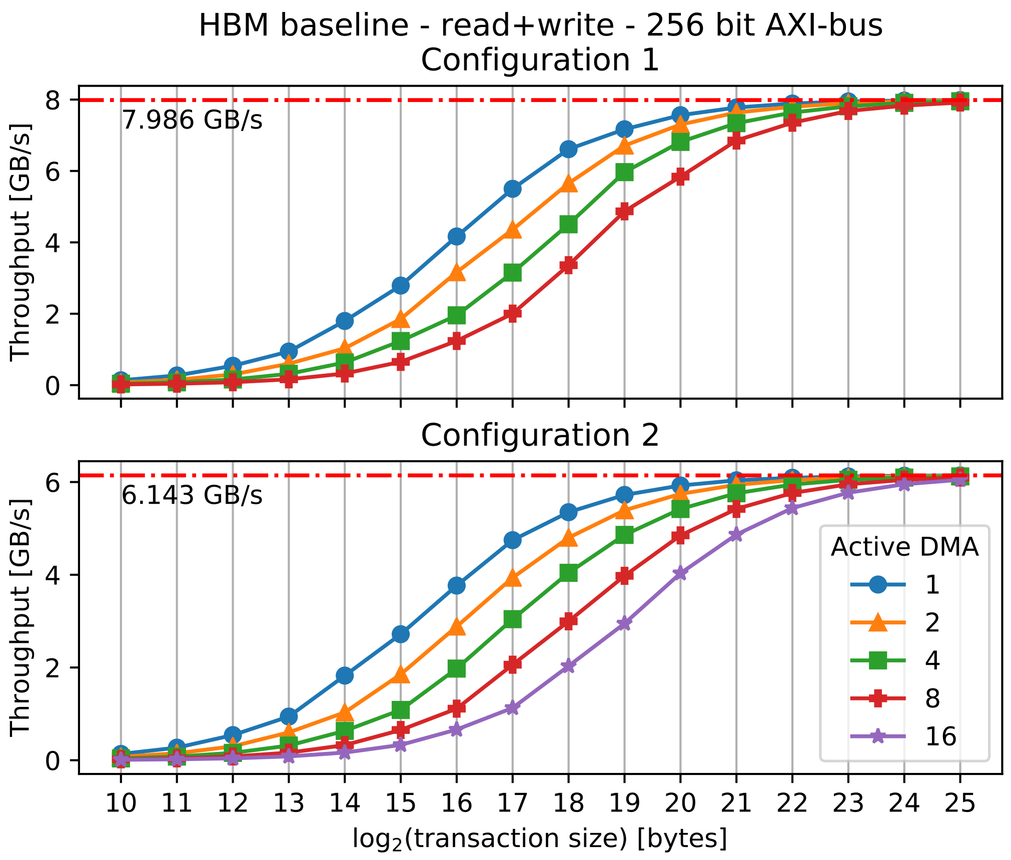

Figure 5.2: Baseline 256 bit AXI-bus results of multiple DMAs reading and writing to HBM banks. Top: each DMA *channel* own HBM bank. Bottom: Each DMA own HBM bank.

In Figure 5.2 the results of the measurements performed using *Configuration 1* and *Configuration 2* are shown. The throughput is calculated using Equation 5.1 where the *data size* is the same as the used transaction size. So it will show the throughput a single bank can provide.

In *Configuration 1*, each bank is connected to only the read or write channel from a DMA. The amount of banks is now twice the amount of DMAs. Since at most 16 banks are used, there is no measurement with 16 DMAs. In both of the graphs in Figure 5.2, three regions can be distinguished. In the bottom left, from 2^{10} to 2^{13} each transaction is small enough that it is finished in a few hundreds of cycles. In this region the number of cycles the MicroBlaze is busy starting a transaction and polling the state of a DMA is relatively big compared to the amount of cycles a transaction takes. At its worst, for one DMA and a transaction size of 2^{10} the throughput becomes 126.3 MB/s. Also, when 8 DMAs are active the overhead is almost 8 times as much and results in a throughput of 19.0 MB/s per DMA, or 151.8 MB/s in total.

In the center of the graph, between 2^{13} and 2^{20}, the difference in throughput between active number of DMAs becomes more obvious. At a transaction size of 2^{17} a single bank reaches a throughput of slightly over 5 GB/s while 8 DMAs reach less than 2 GB/s per bank. Although it looks like the ratio of overhead in the center is bigger than in the bottom left, the ratio is becoming less as the transaction size increases. It is only more distinct in this part.

At the top right part, there is nearly no difference between the measured throughput with a different number of DMAs. The MicroBlaze has more than enough time to start a transaction and wait for another DMA to finish its transaction. The dotted line represents the highest measured throughput out of the performed benchmark, which is when one DMA is connected to two banks and the transaction size is 2^{25} bytes. When 8 DMAs are connected to 16 banks, a throughput of 7.986 GB/s is measured per bank, for a total of 127.8 GB/s from the whole stack. Each of the 8 banks are very close to the expected maximum of 8.0 GB/s where the slight decrease is a result of the overhead incurred on the MicroBlaze.

Configuration 2 has both the read and write channel of a DMA attached to the same bank. The HBM IP now has to switch between reading from and writing to a bank frequently. The same characteristic overhead is seen as discussed from *Configuration 1*. The main difference is that the maximum throughput has dropped to roughly 75% of the throughput found using the setup of *Configuration 1*. Important to note about this measurement is that although the bank is now accessed twice as often as in *Configuration 1* the throughput is not reduced by half. As mentioned earlier, the pseudo-memory channel of each bank can support a frequency up to 450 MHz with a bus of 256 bit. From this measurement a throughput of $2 * 6.143 = 12.286$ GB/s per bank is found. In this configuration two connections (one reading and one writing) are made, requesting 512 bits being transferred at a rate of 250 MHz from a bank for throughput of $250 * 10^6 * \frac{512}{8} * 10^{-9} = 16.0$ GB/s which is just over the 14.4 GB/s per bank that would be supported at 450 MHz. This will result in some stalls, reducing the throughput significantly. At the same time switches have to be made between reading to a bank and writing from a bank, further decreasing performance.

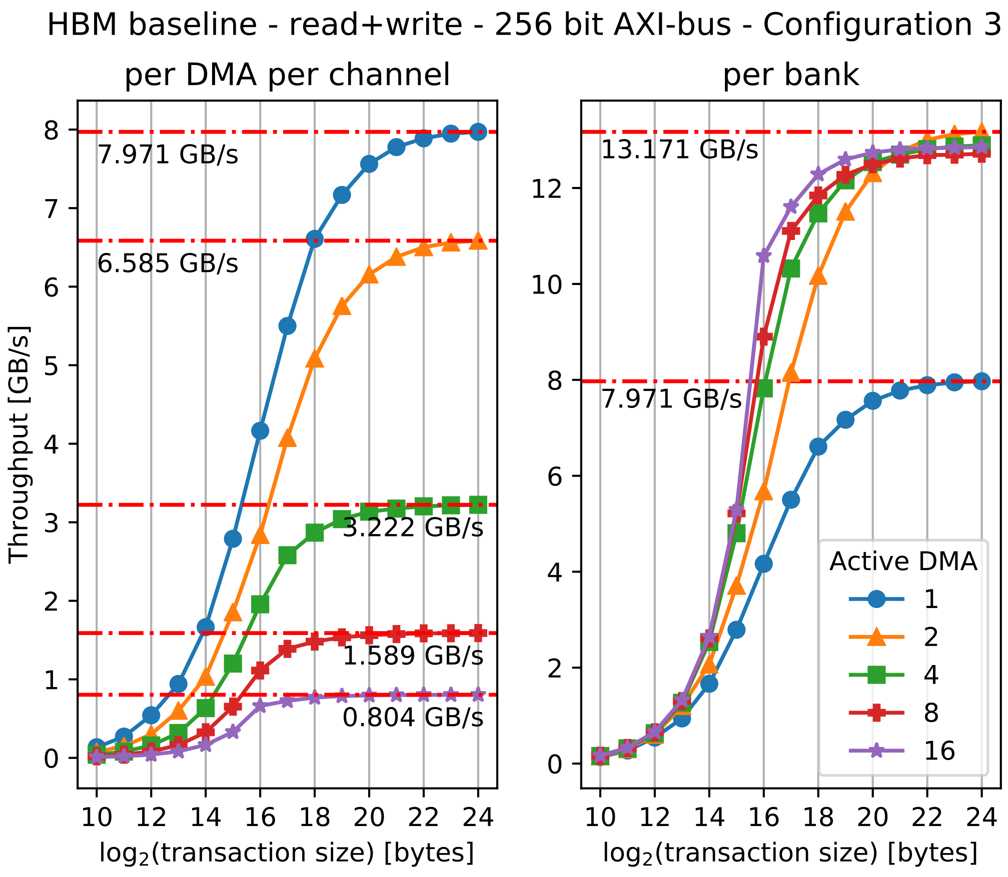

Figure 5.3: Baseline 256 bit AXI-bus results of multiple DMAs reading and writing to single HBM bank. Left: shows throughput per channel. Right shows total throughput (left graph multiplied by amount of active DMAs)

Configuration 3 uses only two of the sixteen available banks but connects all DMAs to those two. One bank is use for reading while the other is used for writing. In this configuration the amount of simultaneous operations a bank can perform is measured and the results are plotted in Figure 5.3. Because an HBM bank can only hold 256 MB of data and 16 DMAs require their own part to write data to and a small portion is required to store the amount of cycles from each measurement, the total size is limited to 2^{24} in contrast to the 2^{25} from the measurement in Figure 5.2. In the plot on the left the throughput for the read/write channel of a DMA is plotted and on the right these are multiplied by the amount of DMAs resulting in the throughput for each of the two HBM banks.

The result for a single DMA is much the same as the result from *Configuration 1*, which is expected as its configuration is identical. The maximum throughput of 7.971 GB/s from the measurement is slightly lower than the 7.986 GB/s displayed in Figure 5.2 because the transaction size for 2^{25} is not included. An important result from this measurement is that when two DMAs are connected the throughput, 6.585 GB/s, is not identical to the maximum throughput measured in *Configuration 2*, 6.143 GB/s. Although in both measurements two DMAs are connected to two banks, in *Configuration 3* a bank is now used for *only* writing or reading. This results in a slightly higher throughput as now no switching between reading and writing has to be made within HBM.

In the plot on the right it shows that the utilization of a single bank does not depend significantly on the amount of parallel accesses that are attempted. With two DMAs the throughput of each bank is nearly the same as when sixteen DMAs are connected to a bank. With 16 parallel DMAs each DMA has a throughput of 0.804 GB/s at its maximum so each bank has a throughput of $16 * 0.804 = 12.864$ GB/s. This is slightly lower than the maximum of 13.171 GB/s measured from using two DMAs, which is because of the additional overhead incurred from using multiple DMAs.

From these results it is clear that a single bank can be accessed in parallel by multiple instances without having a significant decrease in throughput. Additionally, it is slightly inefficient to perform reads and writes on the same bank. The best throughput is obtained when each bank is accessed through only one channel, however this does not allow any parallelization on the data within that bank.

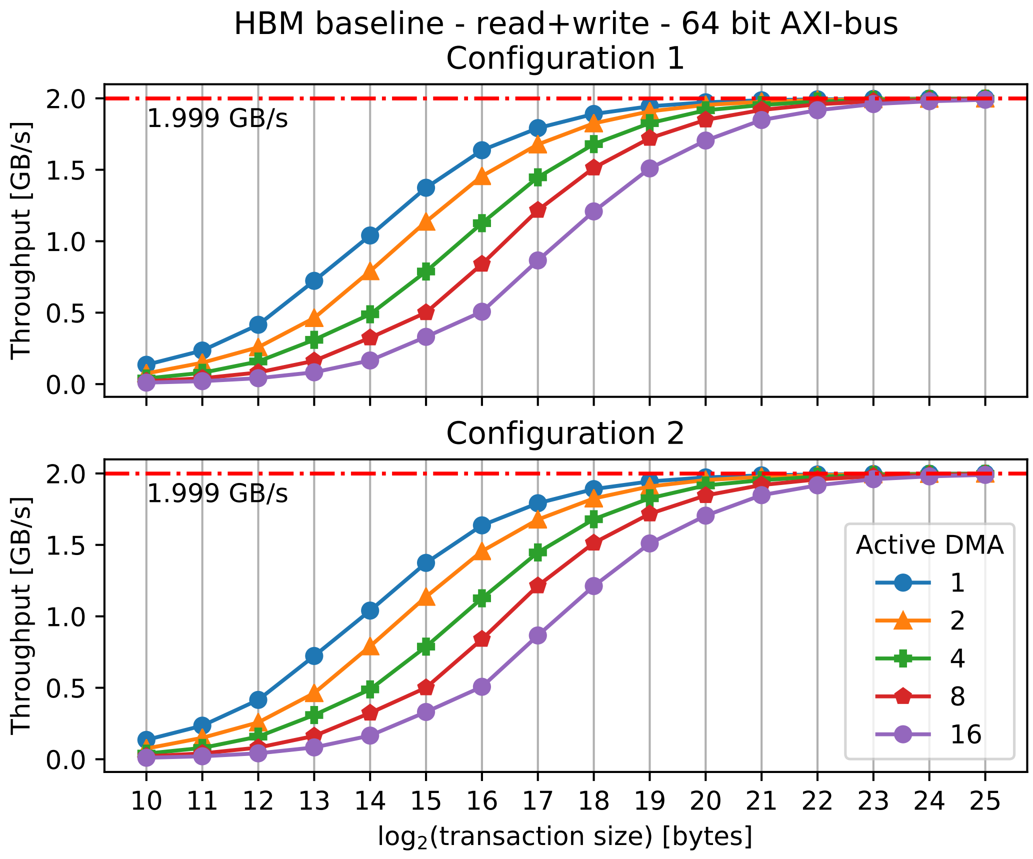

Figure 5.4: Baseline 64 bit AXI-bus results of multiple DMAs reading and writing HBM banks. Top: each DMA *channel* own HBM bank. Bottom: Each DMA own HBM bank.

5.2.2. Results from 64-bit AXI-bus to HBM

Although the 256-bit AXI-bus measurements show some of the limits and characteristics of the HBM IP, the Snappy decompressor will use a 64-bit AXI-bus. Therefore to compare those results the same measurements are performed on the same configurations as the in the previous section, with the narrower bus.

The results from the baseline measurement from *Configuration 1* and *Configuration 2* are plotted in Figure 5.4. The curves are much the same as the curves from the measurement found using the same configurations on a 256 bit AXI-bus, shown in Figure 5.2. However, these result show a much lower throughput. The results from *Configuration 1* are a fourth of the results from the results from the 256 bit measurement, which is as expected as the bus is a fourth of the width as well. The expected throughput would be $250 * 10^6 * \frac{64}{8} * 10^{-9} = 2.0$ GB/s per bank and twice as much, 4.0 GB/s, per DMA.

In this case there is no difference between the two configurations since the channel can easily support it when two channels of a DMA are both reading and writing on a single bank. The available bandwidth of a bank is still theoretically 14.4 GB/s and the reading or writing 64 bit values at 250 MHz only reaches 2.0 GB/s, which is not near this limit.

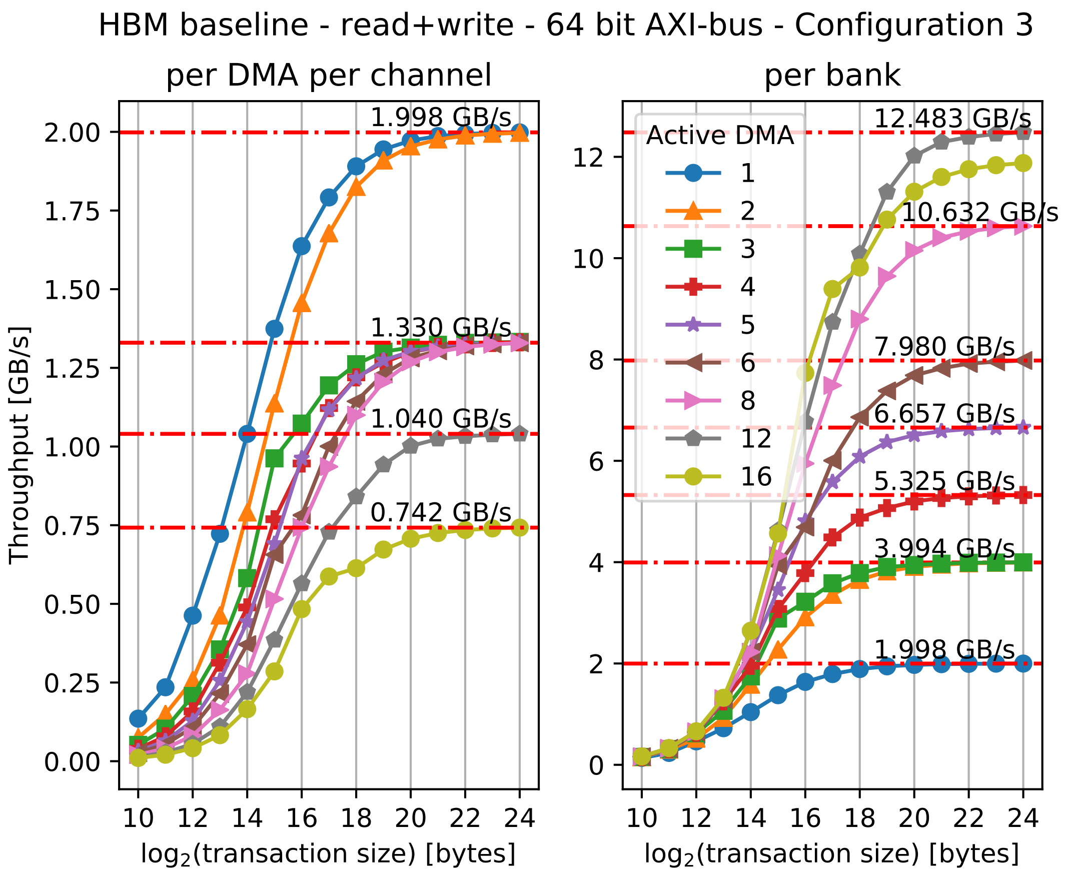

Figure 5.5: Baseline 64 bit AXI-bus results of multiple DMAs reading and writing to single HBM bank. Left: shows throughput per channel. Right shows total throughput (left graph multiplied by amount of active DMAs)

More interesting are the results from the benchmark performed on *Configuration 3*, shown in Figure 5.5. When up to two 64 bit channels are performing transfers on a bank they can reach their maximum bandwidth.

Once more are added the throughput seems to drop by nearly 33%. While a drop in throughput is expected, as it was observed in the 256-bit measurements as well, that it already occurs at three parallel DMAs is strange. What seems to be the case is that only twice every three cycles a transaction is completed. This would result in the observed $\frac{2}{3} * 250 * 10^6 * \frac{64}{8} * 10^{-9} = 1.33$ GB/s. With three DMAs having one read and write channel each an expected combined bandwidth of approximately $3 * 2 * 2.0 \approx 12.0$ GB/s should be obtained. Instead it gives $3 * 2 * 1.33 = 7.44$ GB/s for both the read and write bank. The cause of this stall can be found in the HBM IP [29]. In it, smaller blocks of 4x4 internal switch connections are named within the larger interconnect. Between these switches the following is stated: "However, for write cycles, there is a single dead cycle when switching between masters on the lateral channel." So when switching from a write on one of the connections to another on a different 4x4 block this extra cycle is stalled.

Much the same as the results from the right plot of Figure 5.3 the utilization of a bank is maximized while multiple DMAs perform transfers on it. For the highest throughput twelve DMAs seems to be optimal.

5.3. Snappy decompressor measurements

In Section 4.1.2 a kernel implementing multiple hardware Snappy decompressors in parallel is presented. As mentioned in Section 5.1 the decompressor will be benchmarked using both random data as well as the Silesia Corpus. Expected from the benchmarks performed on the vhsnunzip module [20] is a throughput of between 1 to 2 GB/s per module, where the data is the sum of the in and outgoing data.

5.3.1. Hardware Snappy decompressor results

The results of a benchmark performed when multiple decompressors are activate are shown in Figure 5.6. The throughput is calculated here with the sum of the size of data being retrieved from and stored into HBM as the *data size* for Equation 5.1. For a set of random data where the decompression ratio is near 1.0 this means that the individual channels reach half of the throughput.

From the benchmark it is clear that a single hardware decompressor reaches the near maximum throughput for a single 64 bit AXI-bus channel of 2.0 GB/s. As the maximum throughput measured is 3.988 GB/s where both the read and write data are used to calculate the throughput, in contrast to the throughput in Figure 5.5 where it shows the throughput for either read or write. So the throughput should be approximately doubled. This measurement shows that the decompressor is capable of fully saturating the bandwidth of a single AXI bus.

When multiple decompressor are activated the throughput quickly rises. Until a chunk size of 2^{14} not a lot of speedup is achieved. Once the chunk size is about 2^{17} or higher it shows that additional decompressors are able to reach higher throughput than when less decompressors are activated. With a chunk size of 2^{20} eight decompressors reach a throughput of more than 20 GB/s.

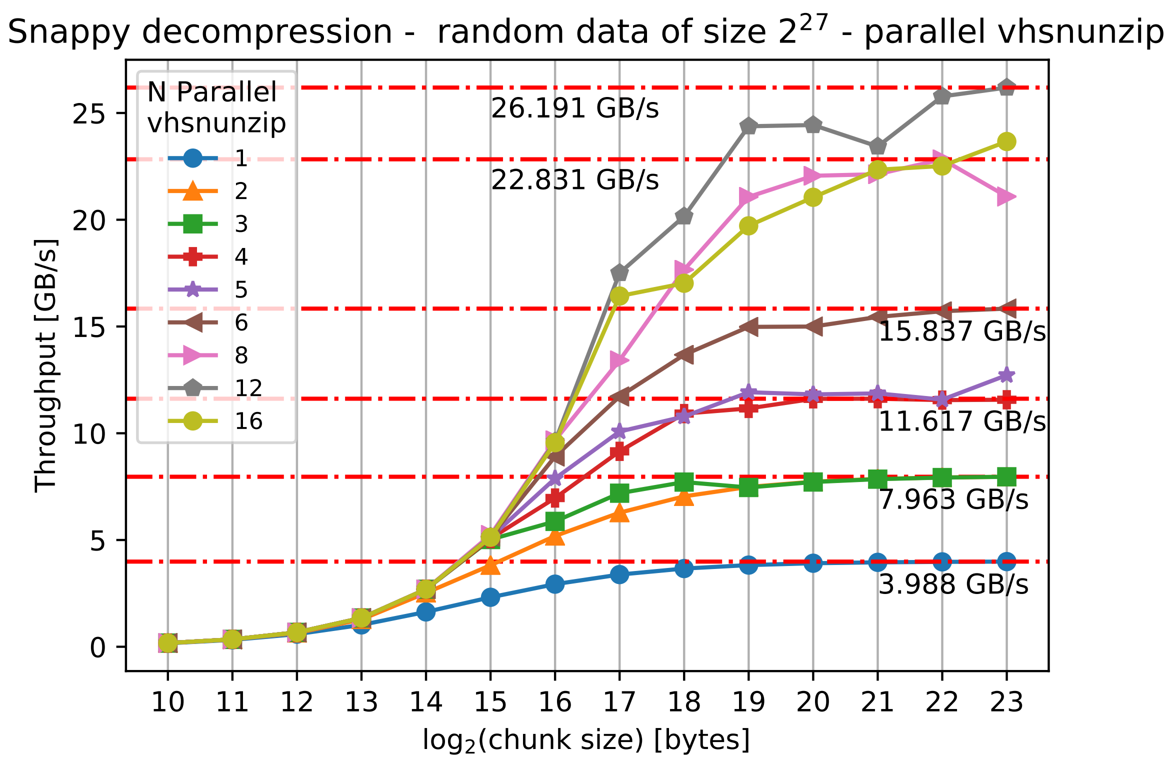

Figure 5.6: Snappy decompression performed on random data with multiple vhsnunzip modules in parallel

Although expected from Amdahl's Law, discussed in Section 3.3.1, in this measurement shows no difference for some results when an additional decompressor is added. For example the maximum obtained throughput for two and three parallel decompressors are nearly the same. This was also observed in Figure 5.5 where the maximum throughput of two and three DMAs are the same.

Curiously, some results are higher than those measured in the baseline measurement. For example when four decompressors are in parallel a maximum throughput of 11.617 GB/s is found, however in Figure 5.5 the maximum throughput of four DMAs on a single bank is 5.325 GB/s which is less than half. The same is also the case for twelve parallel decompressors with a maximum of 26.191 GB/s here and 12.483 GB/s in Figure 5.5. This is likely because of the mentioned issue where a cycle is stalled between writes. Within this design sometimes a DMA is idle as it has to wait for some headers to be parsed. During this time the stalling may be removed allowing a slightly higher throughput.

In Figure 5.7 the results are shown where each Silesia Corpus file is decompressed. One benchmark is plotted where the chunk size is set to 2^{16} which is the default for the Snappy framing format specified by Google. For the other benchmark chunk sizes of 2^{20} are used as they show the reachable throughput while still splitting the file in enough chunks of the parallel hardware decompressors. The maximum throughput measured for each file is given in Table 5.3.

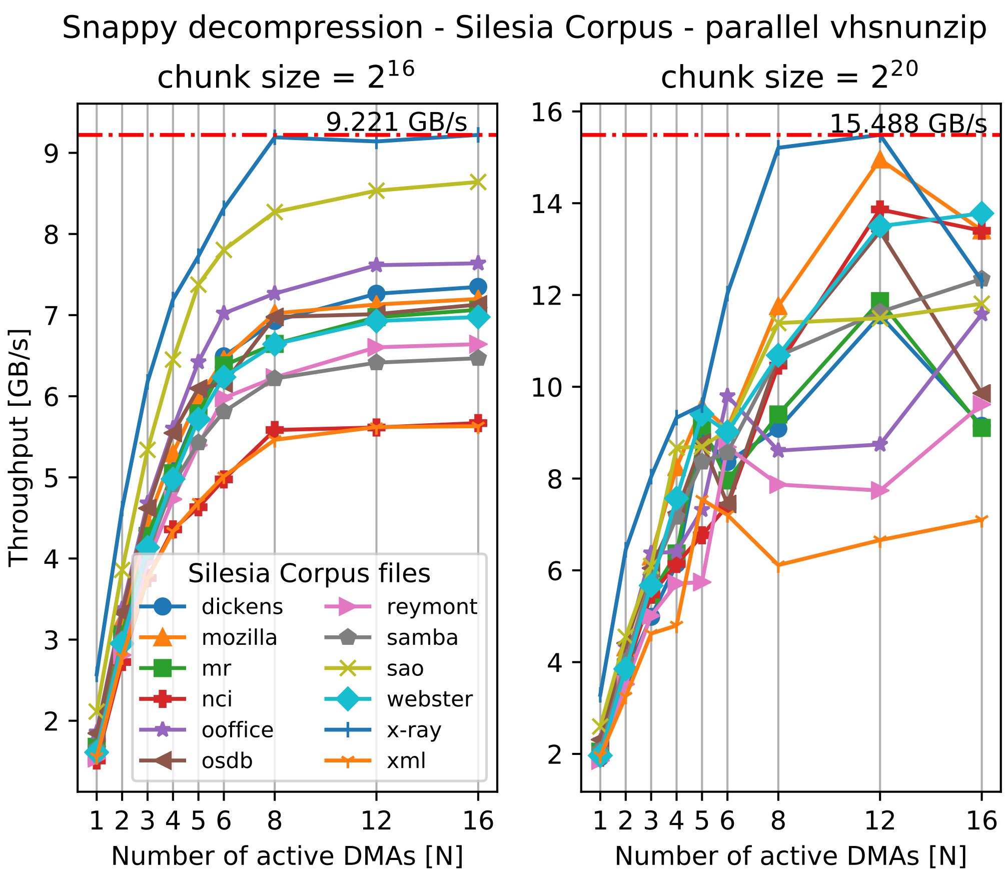

Figure 5.7: Snappy decompression performed on Silesia Corpus files with multiple vhsnunzip modules in parallel

Especially in the left plot no increase in throughput is achieved when more than eight parallel DMAs are in use. The maximum throughput of 9.221 GB/s occurs for the file named *x-ray* because its ratio is very close to 1.0 and as such shows a throughput close to the measurement with random bytes. When a file has a higher compression ratio the end-to-end throughput drops significantly as the input stream has to be stalled to let decompressed data being streamed out. In the right plot, with a larger chunk size, the results are less predictive but in most cases seem to maximize with twelve parallel decompressors. For a chunk size of 1 MiB (2^{20} bytes) some of the files are too small to really benefit from the parallelization such as the *xml* that has the lowest throughput.

Filename	***Type***	***File size [B]***		***Ratio***	***Max throughput [GB/s]***	
		Raw	***Compressed***		***chunk***=2^{16}	***chunk***=2^{20}
dickens	English text	10,192,446	6,339,554	1.61	7.35	11.56
mozilla	exe	51,220,480	26,470,530	1.94	7.20	14.95
mr	picture	9,970,564	5,421,517	1.84	7.07	11.87
nci	database	33,553,445	6,152,480	5.45	5.67	13.86
ooffice	exe	6,152,192	4,272,188	1.44	7.64	11.58
osdb	database	10,085,684	5,331,019	1.89	7.13	13.39
reymont	Polish pdf	6,627,202	3,234,912	2.05	6.64	9.62
samba	src	21,606,400	8,012,408	2.70	6.47	12.35
sao	bin data	7,251,944	6,436,491	1.13	8.64	11.81
webster	html	41,458,703	20,213,433	2.05	6.98	13.78
xml	html	5,345,280	1,309,280	4.08	5.63	7.54
x-ray	raw data	8,474,240	8,210,614	1.03	9.22	15.49
Total		211,938,580	101,404,426	2.09		
Average		17,661,548	8,450,369	2.27	7.14	12.32

Table 5.3: Throughput results of decompressing Silesia Corpus using hardware Snappy decompressor.

To show the throughput of larger files a benchmark is performed on 128 MiB files with a chunk size of 2^{20} that have a varying compression ratio. Within the benchmark ratios are used that are comparable to the range of ratios found in the Silesia Corpus. The results of this benchmark is shown in Figure 5.8. From these measurements can be concluded that for big files the parallel decompressor design reaches an end-to-end throughput in the range of 11-25 GB/s when twelve or more decompressors are active in parallel.

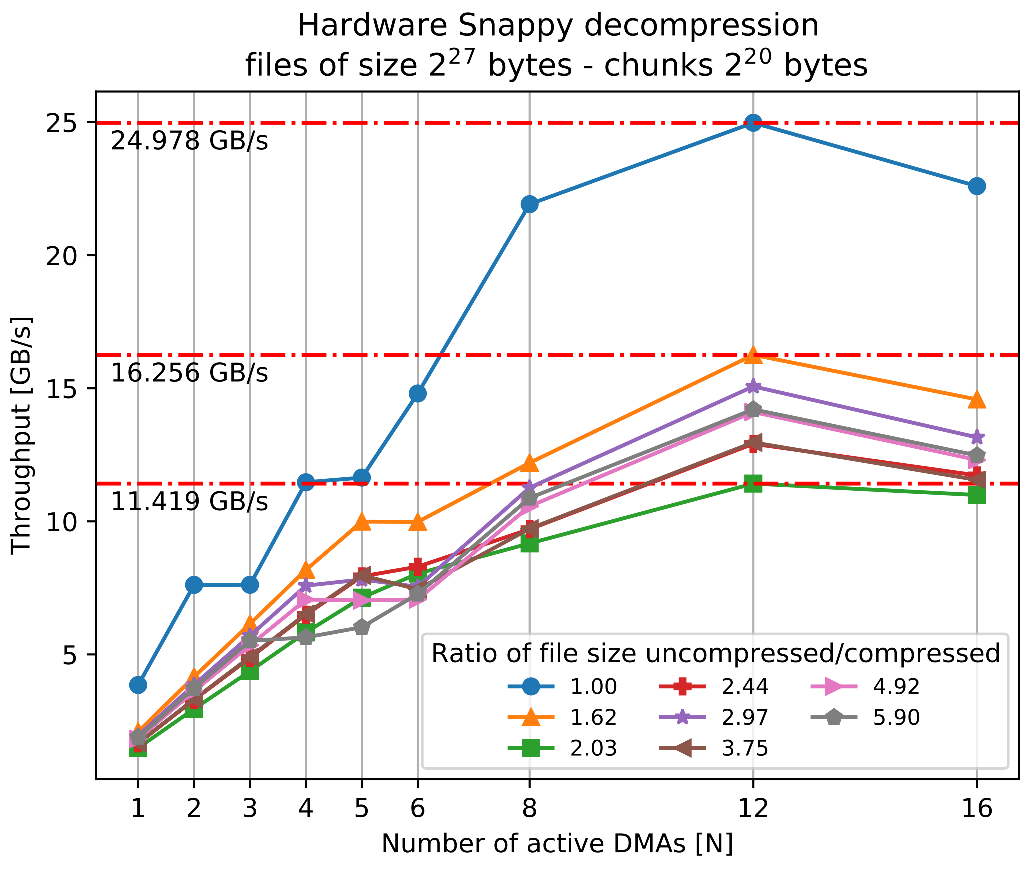

Figure 5.8: Snappy decompression performed on files of varying decompression ratios with a chunk size of 1 MiB.

5.3.2. Software Snappy decompressor results

To compare the results from the hardware design on an FPGA an implementation in software has been developed. Using `C++` as a programming language and the `std::thread` class, a multithreading application has been created where each thread on the multiple cores of a CPU can be used as a decompressor. In the same way as the hardware implementation has a pool of DMAs available to decompress chunks, the software implementation uses a pool of threads.

The application is executed on the same server that has the Alveo Card installed. As stated in Section 5.1.1, this server has 40 threads available. Measurements are performed on both random data and the Silesia Corpus, using an increasing number of threads. For the measurements on random data the chunk size is increased in by the power of 2 for each measurements. The throughput is calculated using Equation 2.1 where the processed data is the sum of the read and written data.

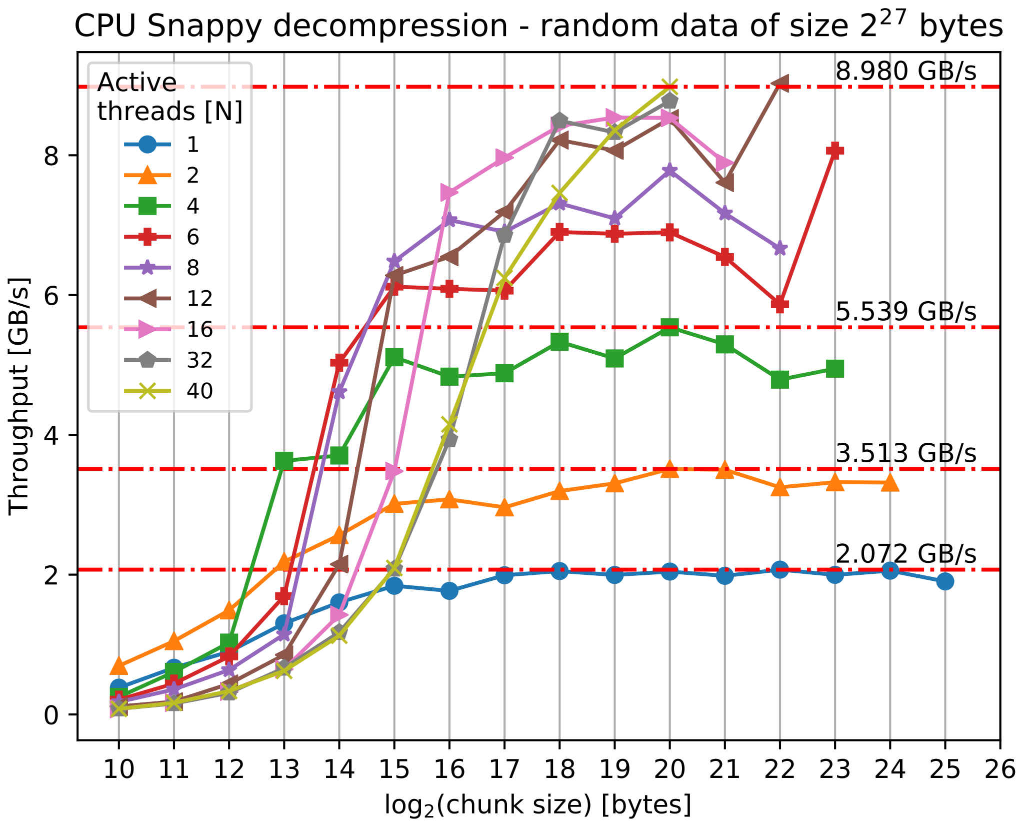

Figure 5.9: Snappy decompression performed on a CPU using random data

In Figure 5.9 the results of the benchmark on a CPU using random data of size 2^{27} bytes are shown. Because the server is used for multiple applications, a thread might be stalled for higher priority tasks, making measurements occasionally be faster or slower depending on the other applications. The effect of this can be seen in the results for a single decompressing thread going up and down when the chunk size is larger than 2^{17} bytes, although the limit of a single thread is reached. Therefore the measurements should be used as an estimation of the throughput. Some of the measurements are left out where there were less chunks in the file than threads that were being used. During multiple loops of decompression the CPU was likely caching data for a threads resulting in very high throughput as memory was not read from DDR anymore.

A single decompressing thread reaches a throughput of around 2 GB/s. Doubling the amount of threads does not linearly increase the measured maximum throughput. When six or more threads are active, the throughput is between 6 to 9 GB/s for this software application.

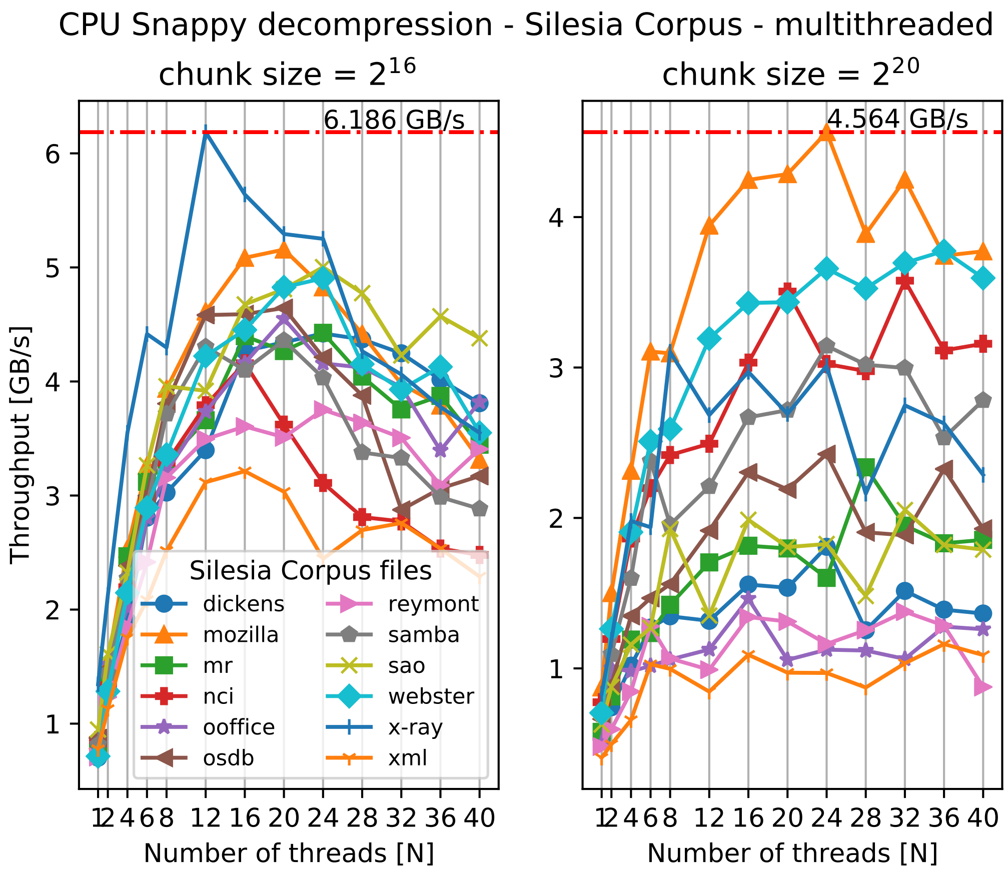

Figure 5.10: Multithreaded Snappy decompression performed on a CPU using the Silesia Corpus with different chunk sizes.

Filename	***Type***	***File size [B]***		***Ratio***	***Max throughput [GB/s]***	
		Raw	***Compressed***		***chunk***=2^{16}	***chunk***=2^{20}
dickens	English text	10,192,446	6,339,554	1.61	4.42	1.81
mozilla	exe	51,220,480	26,470,530	1.94	5.16	4.56
mr	picture	9,970,564	5,421,517	1.84	4.43	2.34
nci	database	33,553,445	6,152,480	5.45	4.16	3.58
ooffice	exe	6,152,192	4,272,188	1.44	4.55	1.46
osdb	database	10,085,684	5,331,019	1.89	4.64	2.42
reymont	Polish pdf	6,627,202	3,234,912	2.05	3.75	1.38
samba	src	21,606,400	8,012,408	2.70	4.36	3.14
sao	bin data	7,251,944	6,436,491	1.13	5.00	2.05
webster	html	41,458,703	20,213,433	2.05	4.90	3.77
xml	html	5,345,280	1,309,280	4.08	3.21	1.16
x-ray	raw data	8,474,240	8,210,614	1.03	6.19	3.10
Total		211,938,580	101,404,426	2.09		
Average		17,661,548	8,450,369	2.27	4.56	2.57

Table 5.4: Throughput results of decompressing Silesia Corpus using software Snappy decompressor

In Figure 5.10 the results of the benchmark on the files in the Silesia Corpus are plotted. In Table 5.4 the maximum measured throughput for each file are shown. An increase of threads shows that quickly the throughput rises. However, from 12-16 active threads in parallel the throughput does not seem to increase any further and for some even drop. Where the hardware decompressor showed an increase in throughput on bigger chunks, the software implementation seems to show a lower throughput. Likely this is because the file consists of less chunks that are parallizable, making additional threads not being used.

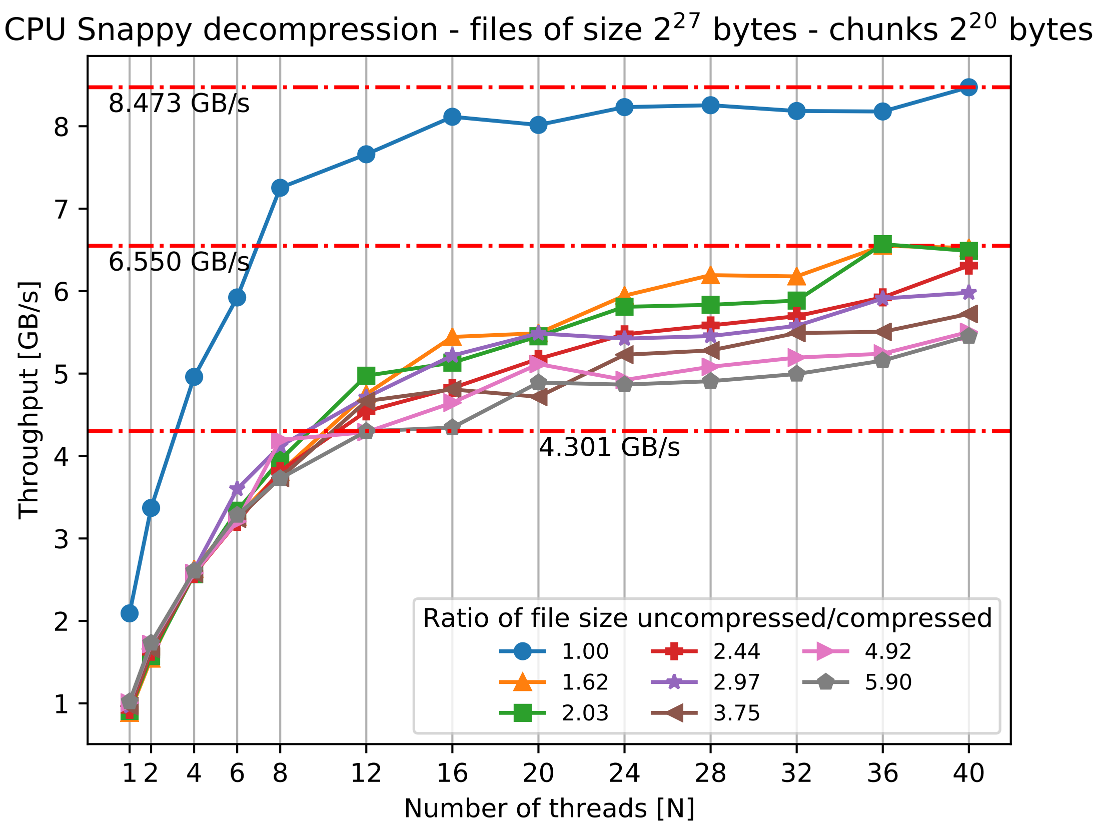

Figure 5.11: Software Snappy decompression performed on files of varying decompression ratios.

As was done for the hardware implementation an additional benchmark is performed on 128 MiB of data with varying compression ratios. These results are shown in Figure 5.11. In the figure a much more stable throughput is found than was obtained through the Silesia Corpus files. With the ratio near 1.0 the throughput is about the same as found in Figure 5.9. The other ratios show a throughput in the range of 4.3 to 6.6 GB/s when the number of threads is at least 12.

Comparing the throughput of the hardware to the software implementation where a 64 KB chunk size is used the hardware shows an average maximum throughput of 7.14 in Table 5.3 and the software shows 4.56 GB/s in Table 5.4. So for chunks of 64 KB in size a speedup of 1.5x is achieved on the FPGA.

When a chunk size of 1 MB is used the hardware shows, for more than 12 active parallel decompressors, a throughput between 11.4 and 25.0 GB/s in Figure 5.8 and for software, with 12 of more parallel threads, between 4.3 and 8.5 GB/s in Figure 5.11. Comparing these to the throughput of 4.3 to 8.5 GB/s to the 11.4 and 25 GB/s range the hardware parallel compressor architecture accomplishes an additional throughput of 2.5x for varying compression ratios with a chunk size of 1 MiB.

5.4. Resource usage

5.4.1. Hardware utilization

The resource utilization of a kernel on the Alveo U280 FPGA are shown in Table 5.5. The utilization of the vhsnunzip block are split up between the block that has additional AXI-Stream switches and the ”regular” vhsnunzip blocks. In this kernel the MicroBlaze is a very lightweight component, mostly using 17 BRAMs. Of the registers almost 25% are used by the Smartconnect IP, likely because it implements a 1-to-16 connection meaning many components are placed 16 times. The total implementation uses less than 9% of the available resources which leaves more than enough room for any possible extensions that may be added to the kernel.

From the 16 AXI-interfaces available per stack all 16 are used in this design, so no other connections will be possible to be made to the same stack. However the Alveo U280 has a second stack that may be used to make addition connections through a second kernel.

The `C`-code for the MicroBlaze is compiled using the *Debug* property (optimization flag `-O0`) in Vitis 2020.1. From compilation an `ELF` file is obtained which is almost 64 KB in size at 63200 bytes. Vitis does provide a *Release* option (with optimization flag `-O2` which should compile away redundant parts and perform optimization, however currently it "optimizes" away everything, leaving an empty executable. So for the design the results of a *Debug* compilation are used, as it fits easily in the 128 KB of the MicroBlaze anyway.

Component	**LUT**	**Registers**	**BRAM**	**URAM**
vhsnunzip single	4804	4772	3	2
vhsnunzip x15	72060	71580	45	30
vhsnunzip with switches	8357	10398	3	2
vhsnunzip total	80417	81978	48	32
MicroBlaze	7145	9570	17	0
Smartconnect	19868	25910	0	0
Total	107430	117458	65	32
Available	1303680	2607360	2016	960
Percentage used	8.24%	4.50%	3.22%	3.33%

Table 5.5: Resource utilization of kernel implementing 16 Snappy decompressors

In Table 5.6 the sizes of the different components in the executable for the MicroBlaze are shown. In this `.text` is mainly filled with the code to be executed. `.data` contains the initialized global variables. The `.bss` part contains the uninitialized parts including the stack and heap. The stack and heap are manually set to `0x1500` which are big enough for the program to successfully execute while keeping the total size just below 64 KB.

Part	**Size [Bytes]**
.text	18900
.data	2412
.bss	41888
stack	5376
heap	5376
full size	63200

Table 5.6: Size of components in .ELF file compiled for the MicroBlaze processor by Vitis 2020.1.

5.4.2. Power consumption

Using a tool provided by Xilinx named `xbutil` the kernel running on the FPGA may be monitored during operation. But the current state of the tool shows some weird behavior. When the U280 has completed a reset the *Card Power(W)* reads 29 W. When an operation is started using 16 parallel decompressors on random data with chunks of 1 KB size, this number does not change and remains at 29 W. During operation the FPGA should show at least some increase in energy consumption. Using the power estimation within Vivado an estimated static usage of 3.2 W and dynamic usage of 2.3 W is calculated for a total usage of 5.5 W. While this is an estimation, the actual usage should be within this amount.

For the processors on the host a tool named `RAPL` [43] is used to measure the energy usage. As mentioned in Section 5.1.1 the host has two processors in a dual-socket configuration. From "idle", measurements a usage of 15-20 W is observed per processor. For the decompression, using the same configuration as for the FPGA and 16 parallel threads, each processor uses between 35-40 W. This is about half the maximum power of 85 W [44] this processor would use under full load. As the 16 active threads are divided over the two processors which have 20 threads available, this usage is expected ($85 * \frac{8}{20} = 34$). This shows an increase of about 20 W per processor or 40 W for the whole configuration to perform the decompression.

6 Conclusions and recommendations

6.1. Conclusions

The main research question of this thesis was formulated in Section 1.3 as ”What are the advantages of using HBM for accelerating decompression algorithms in big data applications”. To answer this a hardware design is designed for and executed on the Alveo U280 FPGA board with HBM utilizing multiple hardware decompressors. In order for multiple decompressors to work, a parallel decompressor configuration in combination with a softcore processor for control and managing purposes is presented. Before decompression is performed, a compressed file is placed in HBM. During compression an uncompressed file is chopped in many *chunks* of data, usually between 64 KiB and 1 MiB in size, which are accompanied by a header. With chunks of compressed data stored in HBM, the softcore processor is used to find and parse headers of these chunks. Using the parsed information of the headers, the softcore processor starts decompression through assigning parallel decompressors individual chunks of compressed data. Through the many available interfaces to HBM the multiple hardware decompressors are connected and able to retrieve compressed data and store compressed data back. The conventional on-chip memory available on FPGAs would not be able to provide enough memory at high frequencies for large chunks of data to be processed in parallel. Through implementation of the design, HBM shows that it allows processing of chunks of variable size while remaining at a very high throughput by providing a high storage size at a high bandwidth.

The used hardware decompressor, *vhsnunzip*, performs decompression of a compression scheme named Snappy. To formalizes chunks of Snappy compressed data within a file a custom framing format has been created. This format specifies the contents of the headers of chunks. These chunks can be decompressed in parallel by individual decompressing engines. Using vhsnunzip as the decompressor a design has been presented where each decompressor is accompanied by a DMA that provides access to HBM. As the vhsnunzip and DMA were using different streaming specification, additional logic has been designed to make them compatible. In the design up to sixteen decompressors are placed in parallel.

A softcore processor, provided by Xilinx under the name MicroBlaze, is added to the design as a controlling unit. This processor is used to manage the many parallel decompressors and their DMAs. It additionally is used to parse the headers of chunks and measure the amount of cycles each decompression takes. By first processing a portion of the headers and storing these in a local buffer multiple decompressors can be quickly activated in succession.

Before implementing the sixteen decompressors a baseline design has been designed using an almost identical architecture. This design attempts to fully saturate the available bandwidth of the HBM interfaces. In the baseline implementation the decompressor is removed and the maximum throughput of HBM is measured using different configurations to connect to the banks of HBM. From these measurements it became clear that placing multiple parallel connections to a single bank does not result in a high loss of maximum throughput. A maximum of 12.5 GB/s is measured on a single HBM bank using twelve DMAs in parallel on a 64-bit wide AXI-interface and 13.2 GB/s on a 256-bit wide AXI-interface when at least two DMAs are used in parallel.

From benchmarks performed using sixteen Snappy decompressor in parallel comparable results are obtained. As the chunk size of a compressed file does not have to be the same for each file, benchmarks are performed iterating over different chunk sizes. Using a 128 MB file of random bytes, where the compression ratio is near one, multiple decompressors achieve a higher throughput. At larger chunk sizes larger than 64 KiB the throughput of parallel decompressors quickly rises over the throughput of a single decompressor. From a chunk size of at least 1 MB no significant rise of throughput is seen anymore. At this point eight or more decompressors can achieve a combined throughput of 20 GB/s per decompressor or 10 GB/s per HBM bank.

As a more practical data set some files with varying compression ratios are also used for two benchmarks with different chunk sizes on both hardware and software. For the benchmark a chunk size of 64 KiB is used and shows an average throughput of 7.1 GB/s. In an other benchmark using chunks of 1 MiB an average throughput of 12.3 GB/s is found. To compare these results an additional implementation in software for execution on the host is made. These measurements show a throughput of 4.56 GB/s and 2.57 GB/s for chunk sizes of 64 KiB and 1 MiB respectively. This shows that mostly for files using bigger chunks, but also for files compressed into smaller chunks a quicker decompression can be achieved on an FPGA using the HBM. When 128 MB files with chunks of 1 MiB are used, the parallel decompressor shows about a 2.5x speedup as compared to a software implementation. A speedup of 1.5x is found for chunks of 64 KiB in size.

Next to the increased throughput a big advantage of the implemented design is the ability to parse variable headers and transactions sizes. With other formats, such as parquet, where the size of a chunk is variable the processor in combination with the DMA can easily handle this.

In terms of resource usage it uses only a slight percentage of available resources on the FPGA. The most usage is seen in the amount of used LUTs, being 8.3% of the total available. This leaves more than enough room for any additional applications or engines to be placed on the FPGA that might use the decompressed data in the HBM.

From the work it can be concluded that HBM is very capable in being used to parallelize an application. Using HBM in combination with a processor and DMA within the FPGA allows for applications that are usually memory-bound to be parallelized. The processor on the host does not have to immediately interact with any of the data as the headers can be parsed directly on the FPGA. The softcore processor is fast enough to allow the end-to-end parsing of some data split in chunks with headers to reach a throughput in the range of 20 GB/s. On the FPGA four such kernels can be enabled in parallel for an even higher throughput to work on multiple files at the same time.

6.2. Recommendations

Use decompressed data: The current design ends with decompressed data remaining in HBM. While these can be pulled from the FPGA to the host it would be more useful to immediately perform some computation on the result of the decompression. As mentioned in Section 3.1 this would not immediately show an advantage of HBM but would proof that HBM is useful in an actual use-case. Once the compressed data is within HBM another kernel could be loaded that performs an operation such as a sum or map/reduce on the data to generate a result of some use-case scenario.

Use more of the available HBM banks: Currently the design uses only two of the sixteen available HBM banks on one of the two stacks. Discussed in Section 3.2.4 is the option to enable multiple banks to be used during a decompression. This would require the softcore processor to parse headers until it has reached the next bank so it can start assigning chunks located in an other bank. Files that are larger than the size of a single bank, 256 MB, would see an even bigger benefit of the parallel decompression. With this the total required bandwidth is spread out over more banks so the limit of a single bank may not be reached.

Stream in compressed data: Before starting the kernel compressed data is copied to an HBM bank on the FPGA. This is done because the Alveo U280, available on a server for testing, is not configured to use streaming input. Once the so called QDMA option has been enabled data can be streamed in. The design would need a slight modification where one or more DMAs take this stream to place the compressed data on an available place in memory. For this task the MicroBlaze is available to point these DMA(s) into the correct direction.

Optimize softcore performance: Within the software on the MicroBlaze some performance increase may be achievable as well. The limits of the local buffer for storing headers are not well defined. It may be possible to find this limit through a trial-and-error strategy or by closely inspecting the usage of the stack during a simulation. Another point of interest may be changing the double use of one of the DMAs. The one DMA that is performing both decompression and retrieving headers from HBM to the local memory of the MicroBlaze may create stalls during execution. It may be more beneficial to use this DMA to *only* retrieve headers, so the header buffer remains as full as possible.

Compare to DDR architecture: To get a better idea of how much faster HBM is than the currently attached DDR a similar application for DDR should be created. DDR however has less AXI-interfaces to connect to so some interconnect like the HBM IP contains should be added.

A
Appendix

Vivado block diagrams

In Figure A.1 four blocks each containing a DMA and Snappy decompressor are shown in parallel. From a control block on the left an AXI interface connects to an interconnect IP which routs commands to the DMAs in each block.
The internals of a block is shown in A.2. In this figure the pink signal shows the conversion from a `cnt` signal to a `keep` signal and the other way around. The contents of the `cnt_to_keep` block contain too many IPs and connections to be any useful in this document. To view them the block diagram should be opened in Vivado.
In Figure A.3 a kernel with four traffic generators and DMAs are shown. The architecture inside each block is shown in Figure A.2.

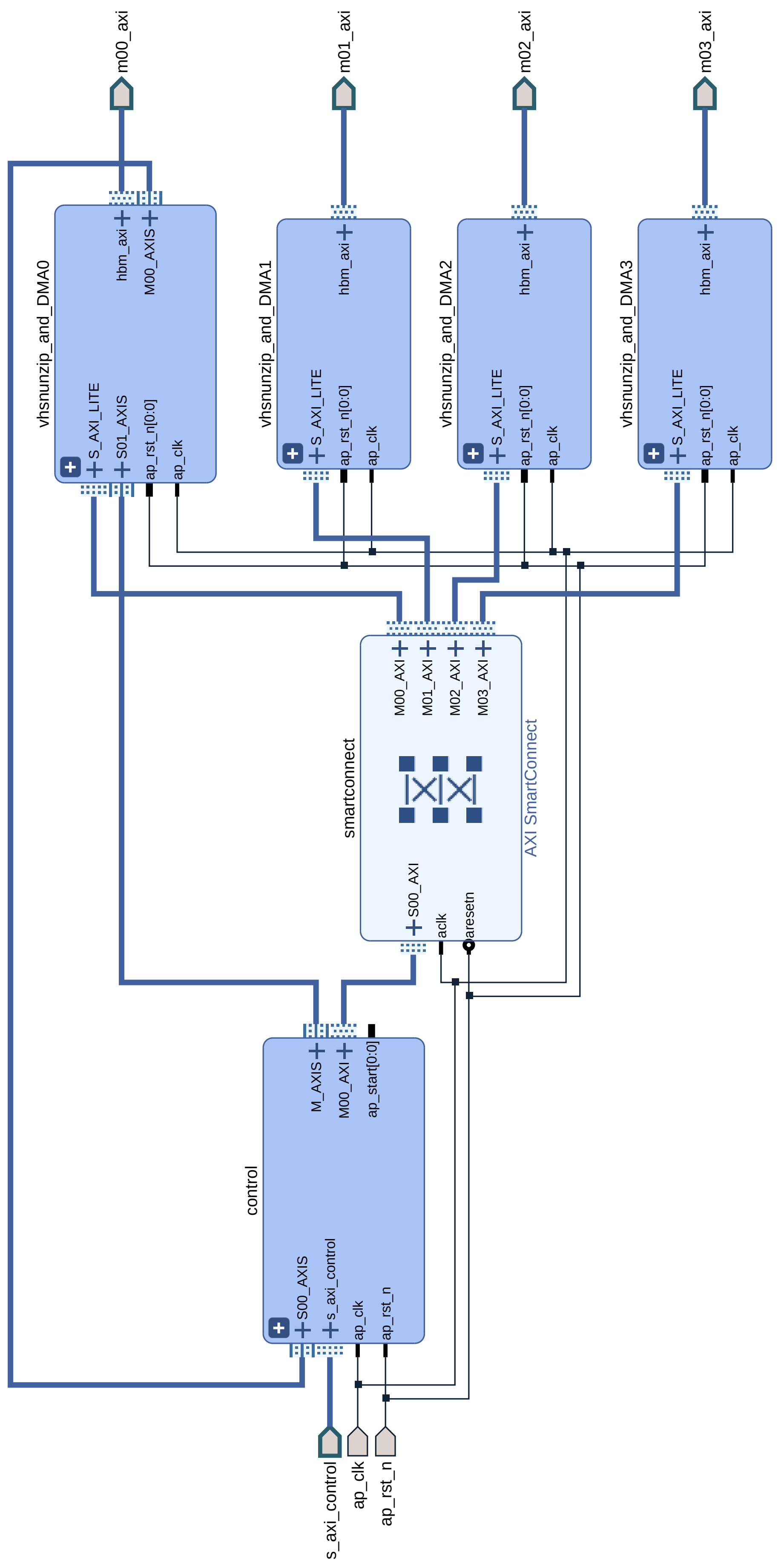

Figure A.1: Vivado block diagram showing a kernel implementing four decompressors and DMAs.

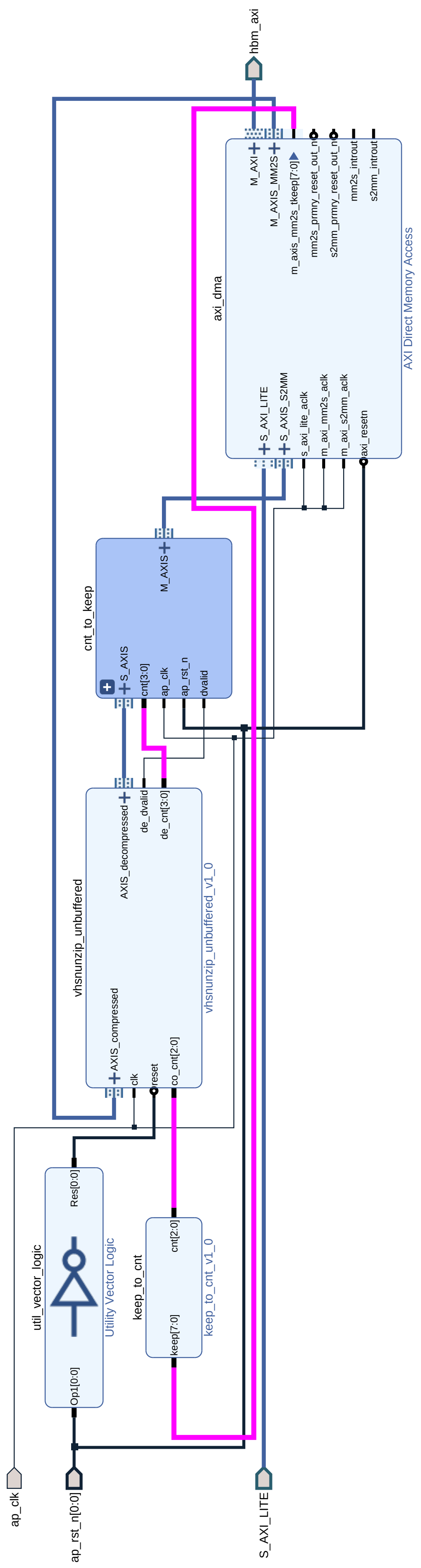

Figure A.2: Vivado block diagram internal of decompressor block.

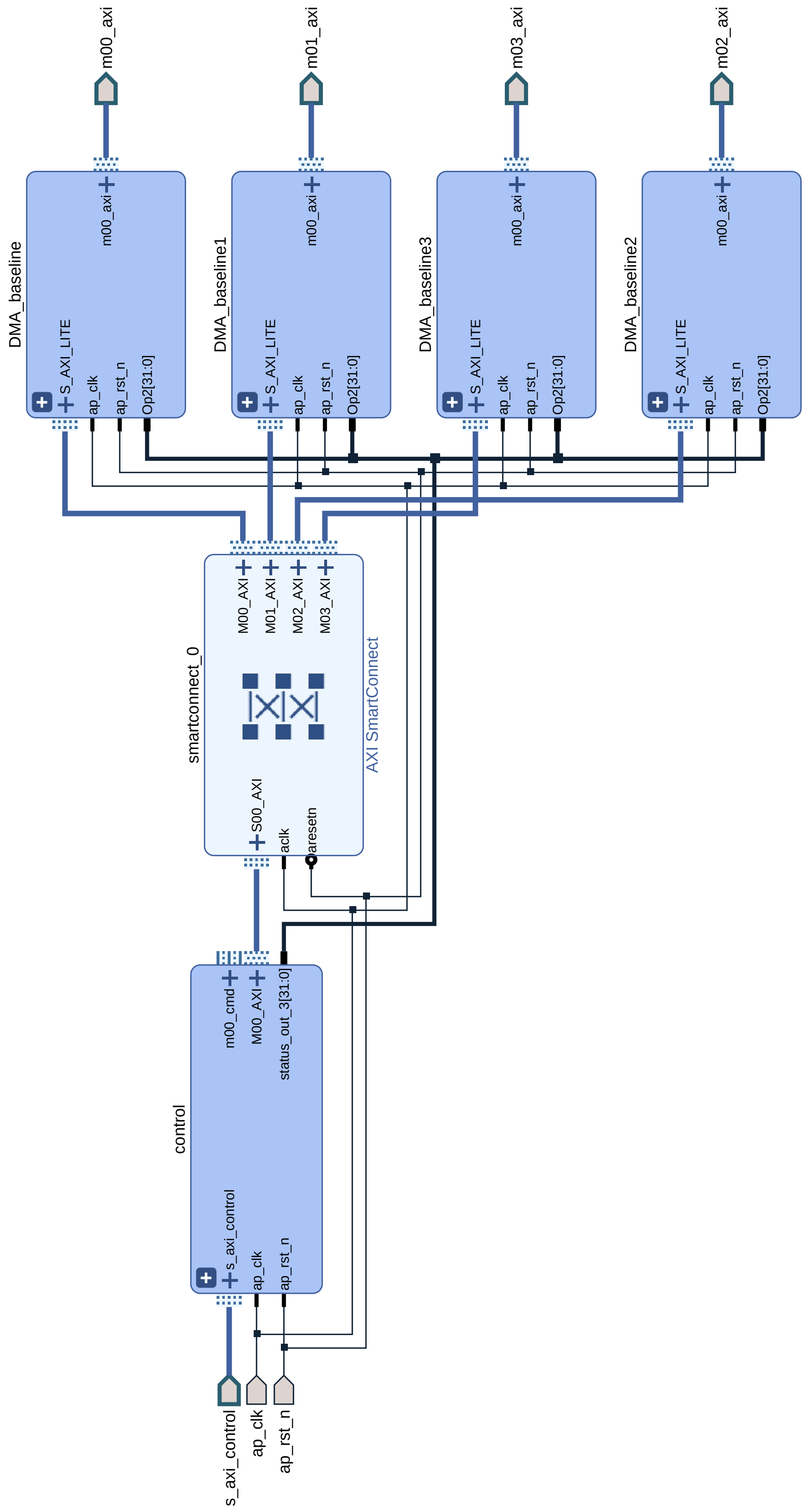

Figure A.3: Vivado block diagram showing a kernel implementing four blocks containing a traffic generator and DMA.

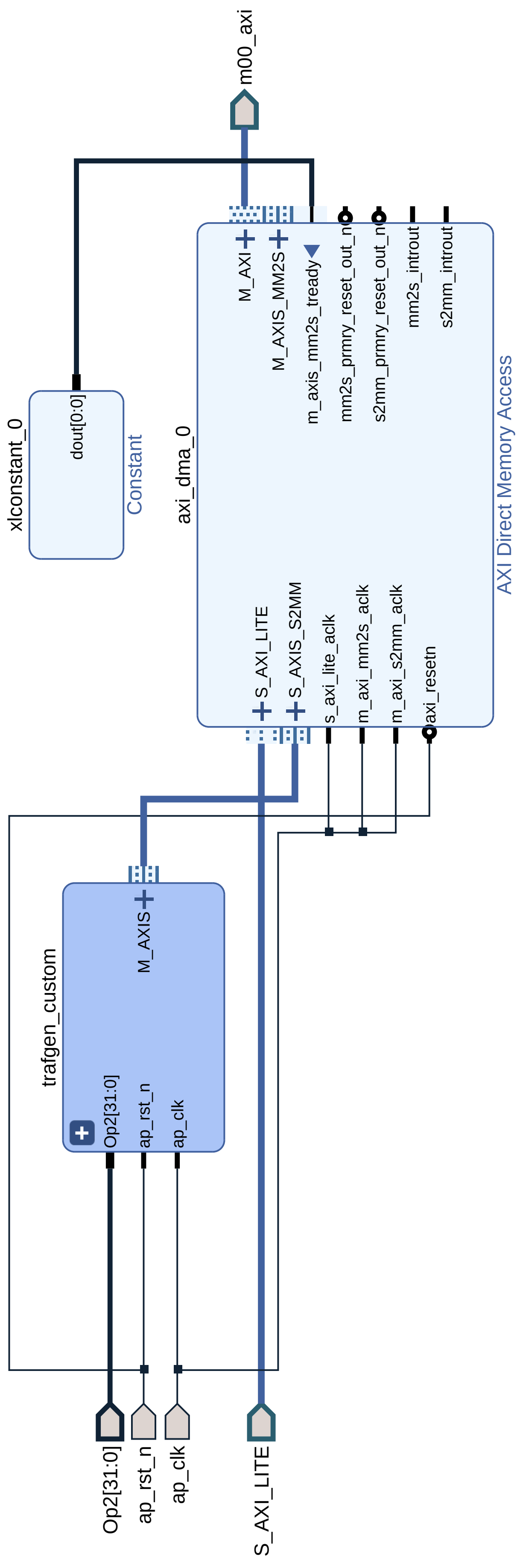

Figure A.4: Vivado block diagram internal of decompressor block.

Bibliography

[1] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59, Jan 2020. ISSN 0949-877X. doi: 10.1007/s00778-019-00581-w. URL `https://doi.org/10.1007/s00778-019-00581-w`.

[2] Amd high bandwidth memory, [url]. `https://www.amd.com/en/technologies/hbm`. Accessed: 18-12-2020.

[3] *[White paper] Supercharge Your AI and Database Applications with Xilinx's HBM-Enabled UltraScale+ Devices Featuring Samsung HBM2*. Xilinx, 7 2019. WP508 (v1.1.2).

[4] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. High bandwidth memory on fpgas: A data analytics perspective. 4 2020.

[5] *Alveo U280 Data Center Accelerator Card User Guide*. Xilinx, 2 2020. UG1314 (v1.3).

[6] Xilinx - integrated hbm and ram, [url]. `https://www.xilinx.com/products/technology/memory.html#internalMemory`. Accessed: 15-02-2021.

[7] *Alveo U280 Data Center Accelerator Card Data Sheet*. Xilinx, 5 2020. DS963 (v1.3).

[8] DOMO. Domo (11-08-2020) - data never sleeps 8.0, [url]. `https://www.domo.com/learn/data-never-sleeps-8`. Accessed: 03-12-2020.

[9] Hamlata J. Bhat. Investigate the implication of "self-service business intelligence (ssbi)", a big data trend in today's business world. *Current Trends in Information Technology*, 10:17–22, 2020.

[10] Rohit Kulkarni at Forbes. Forbes big data (07-02-2019), [url]. `https://www.forbes.com/sites/rkulkarni/2019/02/07/big-data-goes-big`. Accessed: 03-12-2020.

[11] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11 (3):33–35, 2006. doi: 10.1109/N-SSC.2006.4785860.

[12] Lz4 - extremely fast compression, [url]. `http://lz4.github.io/lz4/`. Accessed: 29-01-2021.

[13] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2019. doi: 10.1109/ICESS.2019.8782524.

[14] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H. Peter Hofstee. An efficient high-throughput lz77-based decompressor in reconfigurable logic. *Journal of Signal Processing Systems*, 92(9): 931–947, Sep 2020. ISSN 1939-8115. doi: 10.1007/s11265-020-01547-w. URL `https://doi.org/10.1007/s11265-020-01547-w`.

[15] Joost Hoozemans, Rob de Jong, Steven van der Vlugt, Jeroen Van Straten, Uttam Kumar Elango, and Zaid Al-Ars. Frame-based programming, stream-based processing for medical image processing applications. *Journal of Signal Processing Systems*, 91(1):47–59, Jan 2019. ISSN 1939-8115. doi: 10.1007/s11265-018-1422-3. URL `https://doi.org/10.1007/s11265-018-1422-3`.

[16] R. Miedema, G. Smaragdos, M. Negrello, Z. Al-Ars, M. Möller, and C. Strydis. flexhh: A flexible hardware library for hodgkin-huxley-based neural simulations. *IEEE Access*, 8:121905–121919, 2020. doi: 10.1109/ACCESS.2020.3007019.

[17] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H. Peter Hofstee, and Zaid Al-Ars. Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz, editors, *Applied Reconfigurable Computing*, pages 32–47, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17227-5.

[18] J. Peltenburg, J. Van Straten, M. Brobbel, Z. Al-Ars, and H. P. Hofstee. Tydi: An open specification for complex data structures over hardware streams. *IEEE Micro*, 40(4):120–130, 2020. doi: 10.1109/MM.2020.2996373.

[19] *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. Xilinx, 5 2020. DS962 (v1.3.1).

[20] Hardware snappy decompressor, [url]. `https://github.com/abs-tudelft/vhsnunzip`. Accessed: 18-12-2020.

[21] *AMBA AXI and ACE Protocol Specification*. ARM, 3 2020. Issue H.

[22] *AMBA 4 AXI4-Stream Protocol*. ARM, 3 2010. Issue A.

[23] vhlib: a vendor-agnostic vhdl ip library, [url]. `https://github.com/abs-tudelft/vhlib`. Accessed: 18-12-2020.

[24] Mouna Baklouti and Mohamed Abid. Multi-softcore architecture on fpga. *International Journal of Reconfigurable Computing*, 2014, 11 2014. doi: 10.1155/2014/979327.

[25] Dorta Taho, Jaime Jimenez, José Martín, Bidarte Unai, and Armando Astarloa. Reconfigurable multiprocessor systems: A review. *International Journal of Reconfigurable Computing*, 2010, 10 2010. doi: 10.1155/2010/570279.

[26] *High Bandwidth Memory (HBM) DRAM*. JEDEC, 10 2013. JESD235.

[27] *High Bandwidth Memory (HBM) DRAM*. JEDEC, 11 2015. JESD235A.

[28] Grzegorz Korpala. Data management in cuda programming for high bandwidth memory in gpu accelerators. *Computer Methods in Materials Science*, 16:121–126, 01 2016.

[29] *AXI High Bandwidth Memory Controller*. Xilinx, 7 2020. PG276 (v1.0).

[30] Snappy, a fast compressor/decompressor, [url]. `https://github.com/google/snappy`. Accessed: 18-12-2020.

[31] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

[32] Z. Wang, H. Huang, J. Zhang, and G. Alonso. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119, 2020. doi: 10.1109/FCCM48280.2020.00024.

[33] Young kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. When hls meets fpga hbm: Benchmarking and bandwidth optimization, 2020.

[34] J. Peltenburg, J. van Straten, L. Wijtemans, L. van Leeuwen, Z. Al-Ars, and P. Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277, 2019. doi: 10.1109/FPL.2019.00051.

[35] *AXI DMA v7.1*. Xilinx, 6 2019. PG021 (v7.1).

[36] *AXI DataMover v5.1*. Xilinx, 4 2017. PG022 (v5.1).

[37] Github xorjoep hbm, [url]. `https://github.com/XorJoep/HBM`, 2021. Accessed: 15-02-2021.

[38] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC '97. Data Compression Conference*, pages 201–210, 1997. doi: 10.1109/DCC.1997.582019.

[39] The canterbury corpus, [url]. `https://corpus.canterbury.ac.nz/descriptions/`, 1997. Accessed: 18-12-2020.

[40] Sebastian Deorowicz. *Universal lossless data compression algorithms*. PhD thesis, Silesian University of Technology, 2003.

[41] A. Gupta, A. Bansal, and V. Khanduja. Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–8, 2017. doi: 10.1109/ICECCT.2017.8117850.

[42] Silesia compression corpus, [url]. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`, 2013. Accessed: 18-12-2020.

[43] Reading rapl energy measurements from linux., [url]. `http://web.eece.maine.edu/~vweaver/projects/rapl/`. Accessed: 12-02-2020.

[44] Intel® xeon® silver 4114 processor specifications, [url]. `https://ark.intel.com/content/www/us/en/ark/products/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz.html`. Accessed: 16-02-2020.