High-Performance Computing with Xilinx Accelerator Cards

Faculty of Physics
University of Regensburg

Thomas M. Karl

2021

Contents

List of Figures **II**

List of Code Examples **1**

I. High-Level Synthesis with Xilinx Accelerators **2**

I 1. Technical Details of Alveo Accelerators 2
I 2. Build and Execution Model 4
I 3. The OpenCL Runtime API 5
I 4. C++ Kernels 12
I 5. Optimization Strategies 13
I 6. Bandwidth of PCIe 19

Bibliography **22**

List of Figures

I.1. Schematic of the *Xilinx U250* accelerator card 4
I.2. *OpenCL* programming model 6
I.3. Interaction of *OpenCL* classes 7
I.4. Throughput of PCIe, upstream 20
I.5. Throughput of PCIe, upstream and downstream concurrently 21

List of Code Examples

I.1 Platforms . . . 7
I.2 Devices . . . 8
I.3 Binaries . . . 8
I.4 OpenCL program handles . . . 9
I.5 Buffer creation and memory copies . . . 9
I.6 Kernel execution . . . 10
I.7 Event synchronization . . . 10
I.8 Time measurement . . . 11
I.9 Kernel function . . . 12
I.10 Page-aligned memory allocation . . . 14
I.11 Sub-buffer creation . . . 15
I.12 Arbitrary data types . . . 16
I.13 Function inlining . . . 16
I.14 Loop pipelining . . . 17
I.15 (Partially) unrolled loops . . . 17
I.16 Nested loops . . . 18

I. High-Level Synthesis with Xilinx Accelerators

In this chapter, we explain in detail, how high-level synthesis (HLS) [10] is used to develop hardware-accelerated software and discuss the fundamentals of the implementation of our *Gzip* decoder. The general process varies depending on the manufacturer of the board. We focus on accelerator cards developed by *Xilinx Incorporated*, one of the largest manufacturers for programmable logic. Although the *Gzip* decoder was tested using an *Alveo U250* board, the shown process works for any device of the *U-series*.

We start by introducing some technical details of the *U250* card and explaining the general process of building and executing an FPGA-accelerated application. Thereafter, we go more into detail and show, how the *OpenCL* API is used to program the host side in order to exchange memory between CPU and FPGA RAM. We point out, how FPGA device binaries are built using *C++* as a kernel language.

We introduce some optimizations for hardware and software and show how to increase the bandwidth of *PCI Express* (PCIe).

I 1. Technical Details of Alveo Accelerators

An *Alveo U250* board is a PCIe accelerator card that contains several hardware resources. The most important part is a *Xilinx XCU250 UltraScale+* FPGA running exclusively on the *Alveo* architecture. The FPGA utilizes *Xilinx Stacked Silicon Interconnect* (SSI) technology. The chip consists of four programmable *Super Logic Regions* (SLR) and a unconfigurable static region. The latter stores the *Alveo* shell, a memory layer that manages the hardware resources on the chip.

A shell is provided by *Xilinx*, which has to be flashed onto the card as an initial setup and serves as a kind of an operating system for the board. The shell ships as two usual *Linux* packages: The deployment shell is only for executing programs on the device, whereas the development shell contains additional software that is needed to create binaries for the specific device.

According to the official data sheet [4, 1], each of the SLR are connected to 16 GByte of DDR4 memory with a maximum transfer rate of 64 Bit 2400 MTransfers per second and error correcting code (ECC) DIMM for a total of 64 GBytes. A single SLR, its memory and its interface form a so-called *memory bank*. Data transfers from the host side to board and vice versa are passed through these global memory banks.

The device connects to 16 lanes of PCIe that can operate up to 8 GTransfers per second (generation 3). In addition, the device connects to two QSFP28 (*Quad Amall Form-factor Pluggables*) connectors with associated clocks generated on the board.

Some additional technical details are presented in table I.1. An overview of the *Alveo U250* board is shown in figure I.1.

Total electrical card load	225 Watt
Network interface	2× QSFP28
PCIe Interface	Gen3 ×16
Look-up tables (LUT)	1728 K
Registers	3456 K
DSP slices	12 288
UltraRAM	1280
DDR total capacity	64 GBytes
DDR maximum data rate	2400 MTransfers/s
DDR total bandwidth	77 GBytes/s

Table I.1.: Technical details of an *Alveo U250*.

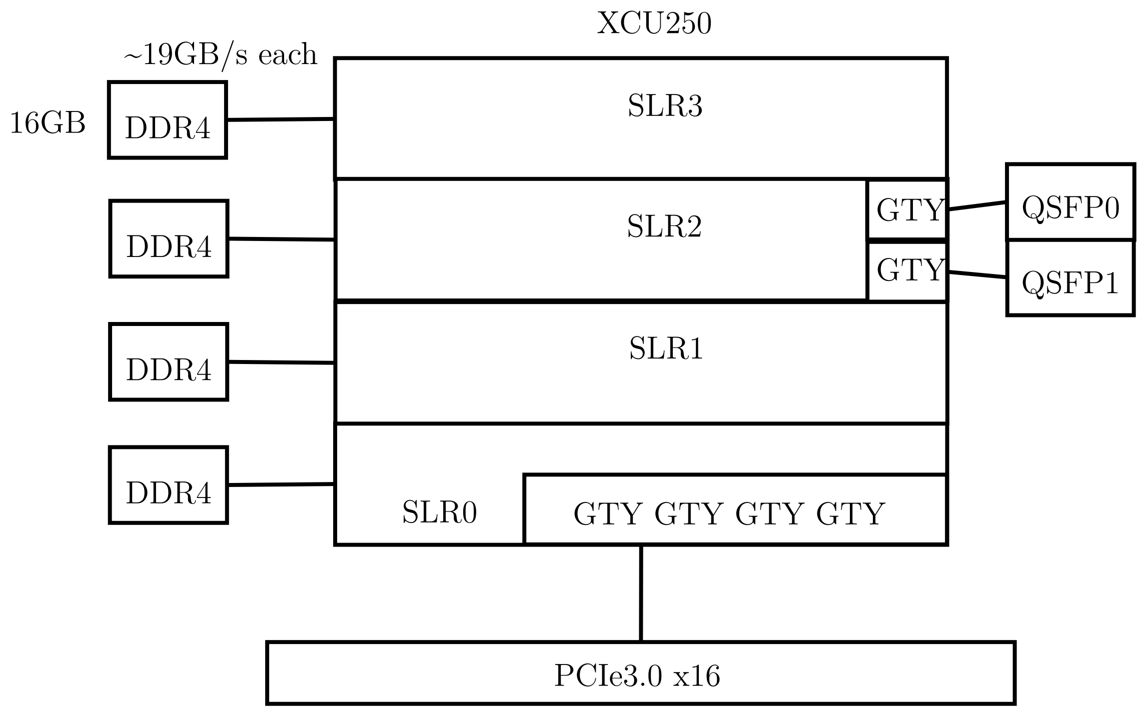

Figure I.1.: Schematic of the *Xilinx U250* accelerator card. The FPGA is a *Xilinx XCU250 UltraScale+*. It consists of four similar *Super Logic Regions* (SLR) and a static region. The latter stores the *Alveo* shell. Each SLR is attached to 16 GBytes DDR4 memory. The SL Region 0 contains four GTY controllers, which are connected to up to 16 lanes of PCIe (generation 3). Two additional GTY controller in SLR2 are attached to QSFP, which lead to a network interface.

I 2. Build and Execution Model

In order to build an FPGA application for *Xilinx* devices two crucial components are needed in addition to the already mentioned shells.

An FPGA binary has to be compiled with the *Xilinx* specific *v++* compiler, which ships as part of the *Vitis Unified Software Environment*. This environment includes a large set of tools, an *eclipse* based IDE and analyzers. Since these tools are only used to simplify programming and have nothing to do with the build process itself, we will not go into detail here. For more information see appendix **??** and **??**.

The second component is the *Xilinx Runtime* (XRT). It consists of a vendor specific *OpenCL v1.2* library [3] and an associated platform. It has to be installed like a common Linux package, since it loads a kernel module that manages communication between host application and Alveo board over PCIe. Therefore, it is highly recommended to use only operating systems that are supported by *Vitis*.

The host side of the application manages FPGA memory and executes device binaries. It is handled as any *OpenCL* application and can be compiled with any *C* or *C++* compiler. The only requirement for the compiler is to be able to link with the *Xilinx Runtime* ("xilinxopencl").

The *v++* compiler offers three modes:

- **software emulation:** The entire build process is completely emulated in software. The program behaves no different from host-sided *C++* code. The purpose is only to test functional correctness.

- **hardware emulation:** A virtual FPGA is emulated in software with help of the *Alveo* shell. The FPGA binary is executed inside that virtual environment. The purpose is to test, if specific FPGA related features work as expected (see sec. I 4). Since this process is extremely slow, it is not recommended to use this feature with large amount of transferred data. Standard output cannot be used here, which makes debugging difficult.

- **deployment:** The FPGA binary is flashed onto the selected *Alveo* board. Only in this case the FPGA is actually required. Even the compilation can be done entirely without *Xilinx* hardware. Note that compiling device code can take several hours.

When the application is executed, an environment variable controls in which of the three modes the *OpenCL* platform has to be loaded (see I 3). The variable XCL_EMULATION_MODE must be set to *hw_emu* for hardware emulation or to *sw_emu* for software emulation. The variable is also used in some *Xilinx* software to distinguish between the three modes at runtime.

I 3. The OpenCL Runtime API

The following explanation of the *OpenCL* API is strongly influenced by the *SDAccel Programmers Guide* [6]. Despite the fact that *SDAccel* was the predecessor of *Vitis*, the given instructions are still valid. The main differences to a standard *OpenCL* introduction are the focus on the FPGA perspective and the usage of the *C++ Wrapper* API [2]. The code examples are real samples of our *Gzip* decompressor. Figure I.2 shows the general idea.

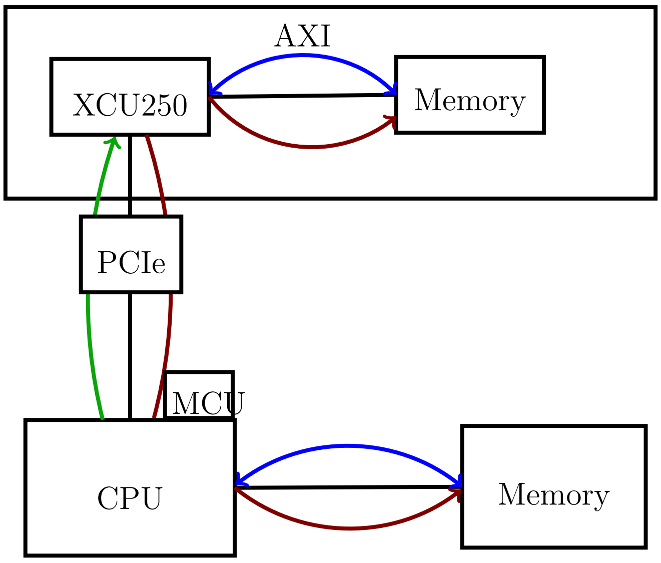

Figure I.2.: The *OpenCL* programming model.
blue: read from and write to memory
red: FPGA allocations, read from FPGA and store into CPU memory (and vice versa)
green: flash binary, execute function

The host program is compiled and executed as usual on the CPU. CPU and FPGA can access their own memory respectively (blue), but only the CPU can instruct the memory control unit (MCU) to allocate buffers in the FPGA memory. It is not possible to allocate memory in device code. The CPU writes specific data from its own memory via PCIe to the board where it is passed via AXI interface from FPGA to memory. Vice versa, the CPU can also read specific data form device memory via PCIe and store it into host memory. The CPU instructs the *Xilinx Runtime* to flash a separate compiled device binary onto the FPGA and execute a specific function with certain input parameters (green). When specifying a buffer as input, the FPGA can operate on it as if it were a normal *C* array.

The rest of the section explains in detail the necessary requirements of a working *OpenCL* application. The interaction between the different *OpenCL* classes is shown in figure I.3 in UML notation [5].

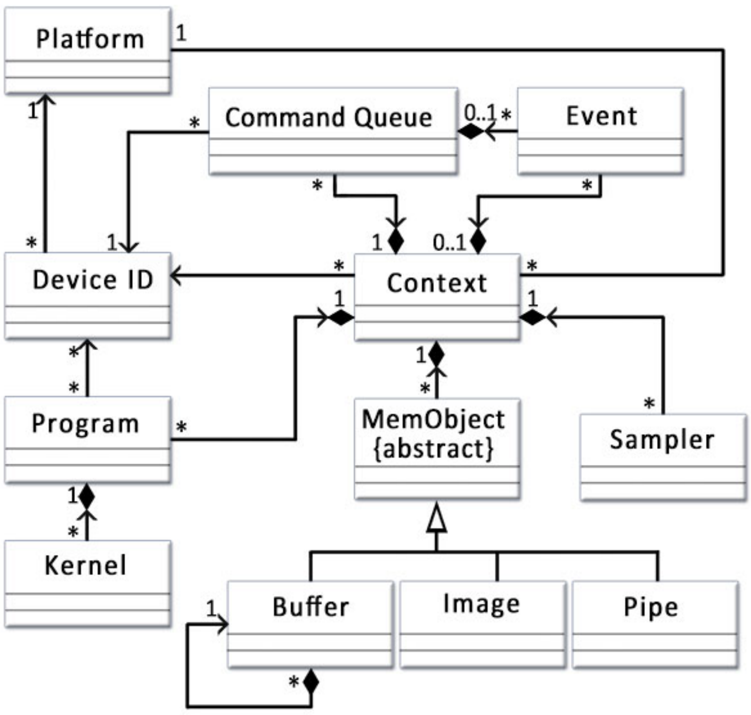

Figure I.3.: Interaction of *OpenCL* classes in UML notation [3][5]

OpenCL is a specified interface for heterogeneous computing. A manufacturer has to implement the details of the API and provide additional software that communicates with OS kernel modules. An *OpenCL* implementation is called *platform*. The first step is to load the specific platform that is associated with the desired device:

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

size_t i;
for(i = 0; i < platforms.size(); i++)
{
    std::string platformName = platform[i].getInfo<CL_PLATFORM_NAME>(&err);
    if(platformName == "Xilinx") break;
}
if(i == platforms.size())
    std::cerr << "Error: Failed to find Xilinx platform" << std::endl;

```

Listing I.1: Platforms

The first function call instructs the *OpenCL* loader to query all platforms on the system at runtime. Each platform has to provide certain meta information. Within the loop the platform with the name “Xilinx” is searched. It is technically possible to have more platforms associated with the same device. Since there is only one *Xilinx* platform, a second occurrence is probably an artifact of an incomplete update. The *Xilinx Runtime* should be reinstalled to solve the problem. Note that most of the API functions return an error value in form of an tabulated negative integer (zero if success). In the following code examples this value is named as “err”. If a function already has a return value, the last value is a pointer to an error code, which points to the desired information after the function call.

Each platform object provides a function that queries all devices associated with it:

```
std::vector<cl::Device> devices;
platform[i].getDevices(CL_DEVICE_TYPE_ACCELERATOR, &devices);
cl::Device device = devices[0];

```

Listing I.2: Devices

An FPGA usually shows up as an “accelerator”. The function call searches for all *Xilinx* devices and stores them in a standard vector. Since we assume that only one device exists, we can use the first entry of the vector as the desired device.

The next step assumes that a specific binary compiled for the desired device is available at the path “binaryFile”:

```
unsigned fileBufSize;
char* fileBuf = read_binary_file(binaryFile, fileBufSize);
cl::Program::Binaries bins{{fileBuf, fileBufSize}};

```

Listing I.3: Binaries

This step reads a file from disk as *C* string. Note that the file has to be read once in order to get the full size of a file. Thereafter, a character array can be dynamically allocated an filled. The file size cannot be deduced from the array without further computation and has to be stored as an extra variable. An object is created out of that binary. Note that “bins” is actually created with a list of pairs of strings an corresponding size.

The device has to be linked with the *OpenCL* context and a program object has to be created within it. The context is used as a handle for the entire application, while the program object is used only for the part that is executed on the device:

```
cl::Context context(devices, NULL, NULL, NULL, &err);
cl::Program program(context, devices, bins, NULL, &err);
cl::CommandQueue q(context, device[0], CL_QUEUE_PROFILING_ENABLE, &err));

```

Listing I.4: OpenCL program handles

Note that both constructors accept a list of devices. Any device from the previous step could be provided here, but for each of them one separate binary in "bins" is needed respectively. Since we do not need extra properties or callbacks for the context, the three corresponding input parameters are set to NULL. Optionally, the binary sizes can be resized to match the actual length. Since we already ensured matching sizes in the previous step, the corresponding parameter in the program constructor is also set to NULL. The third API call creates a command queue associated with exactly one device. Such a queue can be seen as a wait list of device-related instructions. A new instruction, like reading/writing data or executing device functions, has to be lined up in a command queue.

Before data can be moved, a buffer on the device has to be created. The size in bytes and the access rights have to be specified. In this case, the buffer can only be read on the device, but never written. The additional input that is set to NULL can be ignored for the moment:

```
cl::Buffer buffer_input(context, CL_MEM_READ_ONLY,
    cl::size_type(<size_in_bytes>), NULL, &err);
...

err = q.enqueueWriteBuffer(buffer_input, CL_FALSE, 0,
    input_length, input_pointer);
err = q.enqueueReadBuffer(buffer_output, CL_FALSE, 0,
    output_length, output_pointer);
...

```

Listing I.5: Buffer creation and memory copies

After the creation of the buffer object, memory located at "input_pointer" of size "input_length" in bytes is written to the global memory on the board. The input 0 denotes an offset in bytes. This is the first command of the API that can be executed asynchronously. This means, that the CPU only submits the instruction to the queue and immediately continues with the subsequent commands (non-blocking). The advantage here is, that the CPU does not waste time on waiting for the runtime to signal completion. Thus, the buffers can be read and written concurrently via PCIe (see I 6). If the command has to be executed synchronously, the CL_FLASE has to be replaced with CL_TRUE. A buffer can be read in the same manner. Since read and write command in the example are called subsequently, the user has to take care of the synchronization on the host side with the help of *OpenCL* events (more at the end of the section) if a data race needs to be avoided. Buffers are consistent over multiple (different) function calls on the device.

Finally, the kernel object is created in order to execute a function on the device. From the program object, which can consist of an arbitrary number of device functions, a specific one has to be selected by its name:

```
std::string function_name = "...";
cl::Kernel kernel_inflate(program, function_name.c_str(), &err);
size_t narg = 0;
err = kernel_inflate.setArg(narg++, buffer_input);
...
err = q.enqueueTask(kernel_inflate);
```

Listing I.6: Kernel execution

The naming conventions of kernels are explained in detail in the documentation of the *v++* compiler [9]. The arguments of the kernel have to be set according to the signature of the device function. The execution of the kernel is queued into the command queue. The *OpenCL* API usually expects device code to be written in the *OpenCL* kernel language, in which each command is automatically executed in parallel over a specified number of threads. We are writing kernels in *C* and the code runs in parallel on hardware level. The kernel must be executed by using the "enqueueTask" function. This function is usually used to execute an *OpenCL* kernel with exactly one thread.

A kernel execution is always non-blocking. At the latest by now manual synchronization is needed. Otherwise, the output is possibly read before the computation is completed. In the following example two buffers are written to the device:

```
for (...)
{
    cl::Event write1, write2, exec;
    err = q.enqueueWriteBuffer(..., NULL, write1);
    err = q.enqueueWriteBuffer(..., NULL, write2);
```

```
        //some host code

        err = q.enqueueTask(kernel_inflate, {write1,write2}, exec);
        err = q.enqueueReadBuffer(..., {exec}, NULL);

        //some host code
    }
    q.finish();
```

Listing I.7: Event synchronization

A queued command can be associated with a specific *OpenCL* event and delayed until a certain number of events are triggered. In this example, the write commands are connected to the events “write1” and “write2” respectively. When the kernel is queued, the execution on the device waits until these events are completed. Therefore, the device waits until both buffers are ready. The read command is immediately queued, but the copy instruction is delayed until the computation on the device is done. This approach can be used to maintain functional correctness despite using asynchronous commands. Between these commands some additional commands can be computed concurrently on the host side, while the device is occupied.

Also, the commands are called rapidly in a loop. This is often used when a large problem has to be divided in smaller (independent) portions. Because of the asynchronous calls, read and write operations may overlap possibly utilizing the full bandwidth of PCIe.

Additionally, *OpenCL* events can be used to get specific profiling information if the command queue was created with the CL_QUEUE_PROFILING_ENABLE option. This is useful for investigating the performance of the application:

```
unsigned long long int time_start, time_end;
err = exec.getProfilingInfo(CL_PROFILING_COMMAND_START, &time_start);
err = exec.getProfilingInfo(CL_PROFILING_COMMAND_END,   &time_end   );

double nanoSeconds = time_end - time_start;
std::cout << "OpenCl kernel execution time is: "
          << nanoSeconds / 1000000.0 << " milliseconds\n";
```

Listing I.8: Time measurement

In this example, the recorded CPU times in nanoseconds between the start and the return of the command associated with the event “exec” are retrieved from the event. The difference yields the computation time. The times for queuing (CL_PROFILING_COMMAND_ENQUEUE) and submission (CL_PROFILING_COMMAND_SUBMIT) to the queue can also be queried to compute the *OpenCL* overhead.

I 4. C++ Kernels

A kernel is a compute-intensive part of the algorithm that is to be accelerated on the FPGA. Kernels can be written in hardware description language[1], *OpenCL*, *C* or a subset of *C++*. *OpenCL* as a kernel language is mainly used for GPU computation and does not yield a real benefit here. Since the code is compiled into hardware description either way, we focus on high-level synthesis with *C++*. Therefore, we need to write kernels as usual *C++* functions, which have to be put in a separate source file, since they are compiled independently from the host code. A name mangling issue will occur if the host code is written in *C* and device code in *C++*. To avoid this issue, the “extern ”C”” linkage is wrapped around the kernel function declaration. The functions can be declared in a header file. Large data processed by the kernel is transferred through the global memory banks on the board. The host machine copies data to one global memory bank or more. Thereafter, the kernel can access the data from these memory banks. The resulting data is transferred back also through the global memory banks. Compiler pragma statements are used to declare the interfaces connecting to the memory banks in both directions inside the kernel function.

```
extern "C" {
  void fpga_uncompress(unsigned char *source, unsigned char *dest,
                       unsigned int scalar, ...)
  {
      #pragma HLS INTERFACE m_axi port=source offset=slave bundle=gmem0
      #pragma HLS INTERFACE m_axi port=dest   offset=slave bundle=gmem1
      ...
      #pragma HLS INTERFACE m_axilite     port=scalar bundle=control
      #pragma HLS INTERFACE ap_ctrl_chain port=return bundle=control
  }}
```

Listing I.9: Kernel function

[1] VHDL, Verilog or a mixture of both

The kernels running on the FPGA can have one or more memory interfaces. The connection from the global memory banks to those memory interfaces are configurable. There are three data interfaces in the example above. The inputs “source” and “dest” are connected to the global memory bank by using the pragma “HLS INTERFACE m_axi”. The “bundle” parameter specifies the name of the port. The compiler will create a port for each unique bundle name.

The bandwidth and throughput of the kernel can be increased by creating multiple ports using different bundle names. In the example from above, the bundle attribute is used to create the ports “gmem0” and “gmem1”. Since both inputs will be accessed through different ports, the kernel is able to accesses them in parallel, potentially improving the throughput of the kernel. In order to increase memory bandwidth, the ports have to be connected to two different memory banks. This can be achieved during the *v++* linking stage using the “–sp” switch. The exact procedure with all available options is described in the *Vitis Compiler Command* documentation [9].

Scalar inputs are directly loaded from the host machine and do not need to be copied by command queue instructions. These inputs will not change on the host side if modified by the kernel. The input “scalar” is specified using the “s_axilite” interface. These data inputs do not use global memory banks. Note that the return type of a kernel is always “void”. When a writable scalar input is needed that can be retrieved from the host after a kernel call, *e. g.* an error code, a one-component array on the host can be attached to a global memory bank.

By connecting the return value to the “ap_ctrl_chain” interface we allow for pipelining the kernel execution on the host side. This will lead in conjunction with page migration to a much higher throughput if the kernel is pipelined at loop level (see I 5).

I 5. Optimization Strategies

The *Xilinx Runtime* allocates the memory space in 4K boundary for internal memory management. If the host memory pointer is not aligned to a page boundary, the *Xilinx Runtime* performs an extra memory copy to make it aligned. Therefore, the host memory pointer should be aligned with the 4K boundary to avoid unnecessary copies. The simplest way in *C++* to allocate aligned memory is with help of a standard vector and a custom allocator.

```
template <typename T>
  struct aligned_allocator
  {
      using value_type = T;
      T* allocate(std::size_t num)
      {
          void* ptr = nullptr;
          if (posix_memalign(&ptr,4096,num*sizeof(T))) throw std::bad_alloc();
          return reinterpret_cast<T*>(ptr);
      }


      void deallocate(T* p, std::size_t num)
      {
          free(p);
      }
  };

  ...

    std::vector<unsigned int,aligned_allocator<unsigned int>> input(size);
    input.fill(...);

    cl::Buffer buffer_input(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_WRITE,
        cl::size_type(size*sizeof(unsigned int)), size.data(), &err)

    err = q.enqueueMigrateMemObjects({buffer_tag}, 0);

```

Listing I.10: Page-aligned memory allocation

A standard vector can be created and used as usual with help of this specific allocator.

Another optimization in this example is page migration. With the option

CL_MEM_USE_HOST_PTR

a buffer can be wrapped around the data of a vector. The “data()” member function of the vector class returns the pointer to the underlying array. The command “enqueueMigrateMem-

Objects" takes a list of buffers and migrates them to the first[2] device associated with the queue object. This is useful for software pipelining if the host is executing the same kernel multiple times. The memory copy for the next kernel call can happen when the device is still operating on the given data. The kernel mus have been compiled with the return port connected to the "ap_ctrl_chain" interface (see I 4).

When implementing data decompression, the following problem will occur: A block has to be decompressed on the device, but its size is not known *a priori*. Therefore, a far too large buffer has to be allocated on the device. The host can still read the output buffer, but by using "enqueueMigrateMemObjects" the entire buffer and not only the uncompressed data is copied. The solution is to write the size as an additional output of the kernel. Afterwards, "enqueueReadBuffer" can be used twice. First, the size of the data is read, then the exact amount of data can be retrieved from the buffer. Since "enqueueMigrateMemObject" is recommended over "enqueueReadBuffer", a sub-buffer can be used to further increase the throughput. In the following example, a sub-buffer of the specific size is created in order to migrate a certain subset of the buffer "buffer_output" to the host:

```
    err = q.enqueueMigrateMemObjects({output_length},
                                     CL_MIGRATE_MEM_OBJECT_HOST);

  _cl_buffer_region sub_buffer_output_region{0, output_length};
  cl::Buffer sub_buffer_output =
      buffer_output.createSubBuffer(CL_MEM_WRITE_ONLY,
                                    CL_BUFFER_CREATE_TYPE_REGION,
                                    &sub_buffer_output_region,
                                    &err);

  err = q.enqueueMigrateMemObjects({sub_buffer_output},
                                   CL_MIGRATE_MEM_OBJECT_HOST);
```

Listing I.11: Sub-buffer creation

The creation of the sub-buffer is a member function of the buffer object and needs a pointer to a sub-buffer region as input. The sub-buffer region is a data structure that consists of an offset and a length. The option "CL_BUFFER_CREATE_TYPE_REGION" is the only valid input and is an artifact of the API specification.

[2]The numbering starts with 0.

One advantage of an FPGA is its adaptability. The *Vitis* environment provides device type definitions for arbitrary integer and fixed point numbers.

```
#include <ap_int.h>
#include <ap_fixed.h>

...

    ap_int<9>    var1 // 9  bit   signed integer
    ap_uint<10>  var2 // 10 bit unsigned integer

    ap_fixed<18,6,AP_RND> my_type; // signed 18-bit variable with 6 bits
                                   // representing the integer value above
                                   // the binary point, rounding to plus
                                   // infinity

```

Listing I.12: Arbitrary data types

Most optimizations can be achieved with help of specific compiler directives. These statements are documented in the *Xilinx HLS Pragma Guide* [7]. The *v++* compiler generates device code for the specified kernels and for all functions that are called inside them. Inlining a function instructs the compiler to embed the resulting device code directly into the upper function for each call. This will result in a higher throughput, since concurrent calls of the function do not need to be serialized. On the other hand, this will increase the hardware requirements. The compiler automatically inlines functions that are expected to consume few hardware resources. Inlining can be enforced by applying the “INLINE” pragma to the body of the function:

```
void add(int a, int b)
{
    #pragma HLS INLINE
    ...
}

```

Listing I.13: Function inlining

This approach is advisable if a relatively large function will be called more times concurrently.

The most important optimazation is loop pipelining. Pipelining means that subsequent loop iterations overlap and run concurrently. By default, a iteration can only start when the previous iteration has finished. The pragma statement instructs the compiler to optimize the loop for an initiation interval (II) of 1. The initiation interval is the number of cycles it takes to start the next iteration:

```
vadd: for(int i = 0; i < 10; i++)
{
    #pragma HLS PIPELINE II=1
    c[i] = a[i] + b[i];
}

```

Listing I.14: Loop pipelining

Assuming that the addition of two vector elements takes 3 cycles, the entire loop would take 30 cycles to finish if no optimizations are applied. With pipelining the number reduces to 12. This is possible, since there are no data dependencies inside the loop. The most promising approach when handling loops is to pipeline large loops first and then unroll nested loops with small loop bodies and limited iterations. Nested loops are automatically unrolled by default.

Unrolling a loop instructs the compiler to create hardware for each particular iteration allowing them to run in parallel. Naturally, a fully unrolled loop can only be achieved if the exact number of iterations is specified at compile time. A loop with dynamic bound can still be unrolled partially:

```
sum: for(int i = 0; i < 4;i++)
{
    #pragma HLS UNROLL factor=2
    sum += arr[i];
}

```

Listing I.15: (Partially) unrolled loops

The additional “factor=2” is equivalent of running the loop body twice concurrently for half as many iterations. This approach requires excessive amounts of logic resources. Therefore, it is advisable to unroll loops that have a small body or a low number of iterations. In this example, a data dependency, *i. e.* a data race when reading and writing “sum” concurrently, occurs. Fortunately, the compiler can handle this simple case. Resolving loop dependencies requires not only understanding of the logic, but also of how loops are synthesized in hardware. Therefore, a general approach cannot be given here. Resolving dependencies requires always a case-by-case analysis. For example, a loop in which an conditional statement splits the body into two can neither be pipelined nor unrolled.

If the pipeline pragma is applied to a nested loop, the compiler attempts to flatten the loops, *i. e.* creating a single loop. This can only be achieved if the following three requirements are met:

- Only the inner loop has a loop body.
- There is no logic or operations specified between the loop declarations.
- The inner loop bound must be constant.

When the outer bound is also constant, the loop is said to be perfectly nested, otherwise semi-perfectly. The following example shows a iteration over each element of a two dimensional object such as a matrix or an image. The loop is perfectly nested. If the outer bound is replaced with a variable, the loop would be semi-perfectly nested:

```
ROW: for(int i = 0; i< 10; i++)
{
    COL: for(int j = 0; j< 20; j++)
    {

        #pragma HLS PIPELINE
        image[i][j] = ...
    }
}

```

Listing I.16: Nested loops

Nesting loops helps the compiler to increase the level of parallelism.

Note that each of the preceding loops where given names right in front of the declaration like "ROW". When the compilation is finished, a report is created that documents how each of them were optimized. The name helps identifying the loops in the report when using the *Vitis Analyzer*. More information is given in appendix **??**.

I 6. Bandwidth of PCIe

PCIe Gen3 $\times 16$ has a maximum physical bandwidth of 15 754 GBytes per second [8]. The overhead induced by the 128b/130b coding (130 bits are needed to transfer 128 bits of data) is already excluded. However, due to protocol instructions and addressing, which occupy some of the bandwidth, the resulting throughput is much smaller. PCIe utilizes dual simplex technology. This allows signals to pass in both directions simultaneously. In contrast to full duplex technology, dual simplex provides two distinct channels for both directions. Therefore, the throughput can only be maximized by reading and writing data concurrently.

The actual bandwidth of PCIe when data is transferred to a *Xilinx* device using the *Xilinx Runtime* must be evaluated. We introduce three simple benchmarks. In the first one we copy some data to the device and increase the size in bytes exponentially. Additionally, we divide the data in different buffers of the same size and transport them concurrently to the device. We measure the time it takes for the command queue to finish, excluding the time for setups and buffer creation. We do so 50 times each and plot the mean of the evaluated throughput against the data size (figure I.4). We use the standard deviation as error bars. From this benchmark we can conduct three important things. Large buffers are in general better than small buffers and dividing a given amount of data into fewer buffers increases the throughput. A very small number of buffers may perform good on average, but the throughput is far less predictable. The average throughput is approximately 6.5 GBytes per second.

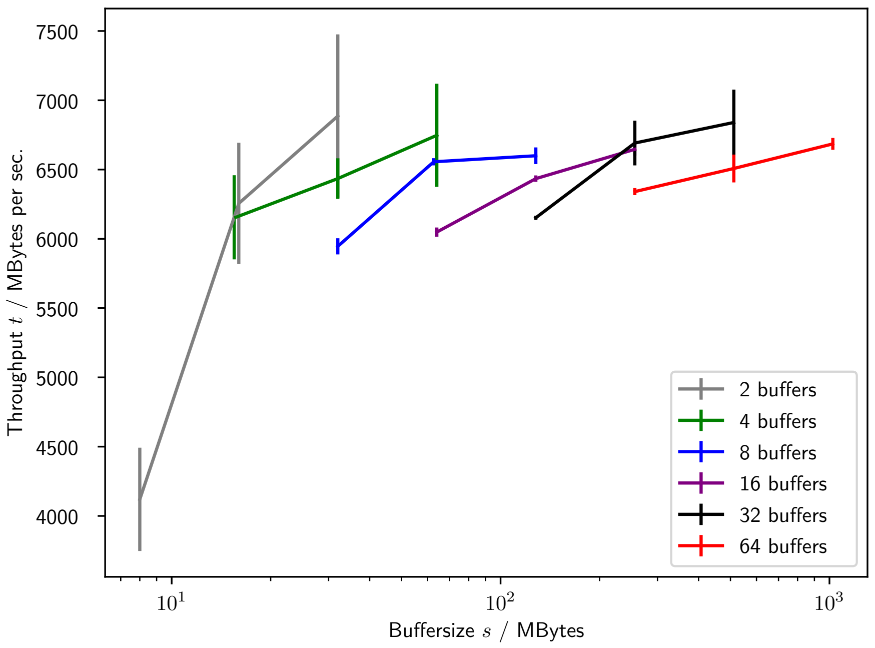

Figure I.4.: Throughput t when data of size s is transferred to the device. The data is divided in different buffers of the same size. Each point is the mean of 50 independent measurements with the standard deviation as errors. Large buffers are better than small buffers. Dividing a given amount of data into fewer buffers increases the throughput. A very small number of buffers performs good on average, but the throughput is far less predictable. The average throughput is $\approx 6.5\,\mathrm{GBytes}$ per second.

In the second benchmark we copy the buffers to the device and measure the time it takes to be transported back to the host. Since upstream and downstream behaved very similar, we do not provide the data here.

In the third benchmark we make use of the asynchronous behavior of the command queues to transfer data from and to the host concurrently, *i. e.* we perform the first two benchmarks at the same time and double the amount of data on the x-axis (figure I.5). We could actually proof, that upstream and downstream can happen concurrently, thus almost doubling the throughput to approximately 13.5 GBytes per second on average, but also increasing the errors.

Since even in the best case the overall throughput is far below the maximum bandwidth of a memory lane in one memory bank (19 GBytes) per second, it is not needed to evaluate additional cases in which the buffers are connected to more than one bank, unless the data size is larger than its maximum capacity of 16 GBytes .

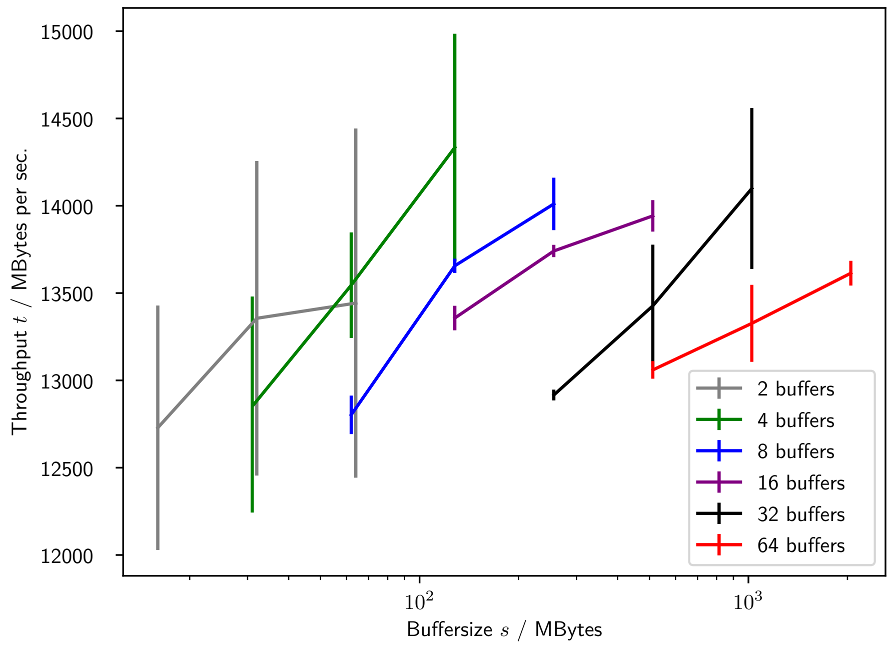

Figure I.5.: Throughput t when data of size $s/2$ is transferred to and from the device concurrently. Upstream and downstream can happen concurrently, but the errors increase. The average throughput is $\approx 13.5\,\mathrm{GBytes}$ per second.

Bibliography

[1] *Accelerator Cards Data Sheet*. Tech. rep. URL: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf.

[2] Benedict R. Gaster and Lee Howes. *The OpenCL C++ Wrapper API*. Khronos OpenCL Working Group. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.

[3] Aaftab Munshi. *The OpenCL Specification*. 1.2. Khronos OpenCL Working Group. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[4] *Product Selection Guide*. Tech. rep. https://www.xilinx.com/support/documentation/selection-guides/alveo-product-selection-guide.pdf.

[5] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[6] *SDAccel Programmers Guide*. 2019.1. Xilinx. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1277-sdaccel-programmers-guide.pdf.

[7] *SDx Pragma Reference Guide*. 2019.1. Xilinx. URL: https://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx2019%5C_1/ug1253-sdx-pragma-reference.pdf.

[8] *Specifications | PCI-SIG*. Accessed: 2021-01-26. URL: https://pcisig.com/specifications.

[9] *Vitis Compiler Command*. Accessed: 2021-01-26. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitiscommandcompiler.html.

[10] *Vivado Design Suite User Guide*. High-Level Synthesis 2019.1. Xilinx. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.