Abstract

Climate simulation are very expensive computationally, which ironically have negative consequences on carbon emissions and leads to global warming. Existing large-scale climate simulations are generally run on archaic software stacks like C and Fotran and run on large cluster computers. These implementations are great from a pure performance perspective, but other productivity metrics are also important like ease of use and development speed. Leveraging the high productivity environment of Python to run high performance computing tasks trough the use of very efficient accelerators, like graphical processing units, have proven to be a viable solution. Here we target a hardware solution not widely used for HPC: Field Programmable Gate Arrays, as these have very favorable power/performance ratios. Trationally FPGAs have been too unwieldy to program large scale programs like climate simulations, but recent developments in high level synthesis have provided an avenue for the use of these types of chips. By exploiting the flexibility of FPGAs we aim to build flexible system that can run deep pipelines of the compute tasks at hand. A FPGA specific optimization, the use of alternative data encoding formats like fixed-point or posits is explored.

Contents

1 Introduction

Performing climate simulations to gain a greater understanding of climate change[37] comes at great energy costs[27]. While these are not a main driver of climate change if we are hoping to be advocating for a greener future, we as climate scientists should also be doing our part. Reducing energy use will allow more, and better, experiments to be run as computing cost is largely dependent on power use. While for many years performance per Watt improvements was more-or-less given by the advance in computer hardware, the apparent end of Moores law[40] and Dennard scaling[12] suggest we need to find new ways to optimize our throughput while keeping energy use reasonable. This is a problem that is not only relevant to climate physicists, but the world at large: Energy use of computing at cloud-scale is a fast-growing sector[6][2], and an important contributor to climate change. In this thesis, we will explore a method to drastically reduce the power, and the associated carbon footprint, of running large-scale climate simulations.

To achieve this promise, we aim to provide a subset of Numpy[19], often used in small scale climate and weather simulations, accelerated on a Field Programmable Gate Array (FPGA). Performance per Watt in these devices is roughly an order of magnitude reduced while performance is comparable or even improved in some cases[10]. A system like this will allow the migration of many expensive climate related computations, everything from weather forecasting to ocean simulations, to lower power systems.

FPGAs are re-configurable hardware chips. Implementing a system in hardware reduces the amount of power needed to run a given computation. An example of a huge speed and power increase for a real world computational problem is seen in cryptocurrency mining where a lot of mining today happens on so called ASICs (Application Specific Integrated Circuits). AICSs are specialized hardware completing one specific task, allowing them to be much more efficient, both in power and in computation speed, than general purpose hardware like CPUs and GPUs. Performance is improved by scaling and designing the chip to match the exact needs of the computation at interest. FPGAs provide a lot of the benefits of ASICs but with an important feature namely that they are reprogrammable. A feature that greatly improves the development cycle in designing hardware, as you can now iterate on your design without fabricating silicon. This also allows FPGAs to be flexible to changing requirements. For example, in the telecommunications[31] industry standards are constantly changing and hardware, such as low-latency switches, needs to implement these changes. Importantly we also get the benefit that you can actually buy FPGAs off-the-shelf while ASICs needs to be special purpose build for you application.

Traditionally FPGAs would be "programmed" at the so-called "register transfer level" (or RTL for short), which means that you essentially control the buses and registers directly. A world very unfamiliar to the average software developer. To improve adoptability FPGA vendors like Xilinx and Intel have created what is known as High Level Synthesis[9]. A system promising to allow you to write C or C++ kernels to be executed on the FPGA, without worrying about FPGA or hardware details. In practice, this is a custom compiler translating kernels to bitstreams that can be executed on the FPGA chip. Developers with experience in writing high performance computing software targeting CPU/GPU systems can now target these systems with better performance characteristics instead. Just as writing high performance code has its quirks with regards to coding style, so does writing HLS, and while performance can be extracted it is not trivial. These systems are starting to be adopted in various high performance computing applications[43][42], but also in data center tasks by Microsoft[34]. You can even rent one on Amazon AWS elastic compute[3]

Developing applications in these kinds of environments is still not easy. Testing and writing the software is more cumbersome than compared to the very quick development cycles you can achieve in a stack like Python + Numpy. It has even been shown that comparable performance can be achieved in this more friendly environment. Veros[16] has shown that pure Python code can run at very competitive speeds compared to traditional models written in e.g. Fotran or C. A feat achieved by using accelerators to run the code without much end-user interaction required. This approach has multiple benefits. Keeping your code base accessible and readable allows for way less bugs[39] to slip trough. Running

and modifying the code becomes very easy. A property which is very attractive in teaching, and research, environments as you can work on a "real" (as in not a toy) models.

In this thesis we aim to target a specific subset of the Veros model and accelerate it on FPGA hardware. Specifically this part includes a high amount of slicing operations and strided operations on large arrays. Furthermore we also include a large "global" operation involving the whole array. This is a hard problem to accelerate and speedups gained here should be transferable to many "real world" scenarios, and should scale to running the entire Veros model. The domain of problems that could be solved with these techniques are quite large, ranging from astronomy to plasma models in nuclear reactors, as Veros is at its core a fluid dynamics model.

To achieve this we build a modular system containing some basic arithmetic functions like adders and multipliers, with some more intricate functions like a tri-diagonal matrix solver. Furthermore, we implement broadcasting operations on all our functions.

We also investigate upon speedups available by using smaller bit-widths. For example, we have seen that it is possible to run simple fluid dynamics simulations on 16 bit hardware[24]. On an FPGA we can do even better. FPGAs are extremely flexible regarding bitwidths and can use any mix and combination of these. We propose to start by building an emulator where we can freely vary bit widths in the simulation. Trough experiments we can find the right set of fixed point variables that allows us to run simulations at a satisfactory performance/accuracy trade-off.

Our approach does have some limitations: FPGAs are limited in size: How can we scale to multiple[36] FPGAs? Can our kernels be optimized[10] further? Although promises are high, actually writing HLS code is not like writing C/C++. You need to implement a very specific coding style to achieve good performance[28], and even then you will almost certainly be beat by a custom hand written RTL version. It is also possible that writing arithmetic kernels might not yield the performance we are looking for, and to actually gain any appreciable speedup we would have to re-write the algorithm from scratch. The possible upside, that we would be able to generalize to very many problems and to the entirety of Veros, makes this a worthwhile endeavour.

2 Background

2.1 Veros

Veros[16] is a fast ocean simulator written in Python. The main trick employed in writing fast Python code is that we can write vectorize operations and then accelerate these functions. Either on CPU by multithreading or by offloading to external accelerators like GPUs. The result is a Python implementation written in the Numpy API which has comparable speed to C or Fortran implementations. To scale simulations to very large sizes, which cannot fit into one machine due to memory requirements, we employ a technique called ghost lines and use technologies like Message Passing Interace (MPI) to distribute model state.

Exchanging the accelerator from a GPU to an FPGA does not matter when employing the techniques allowing for distributing to multiple hardware accelerators. Often in fluid dynamics, calculations are local in the sense that they require information only about the nearest neighbours. From a physical standpoint, this makes a lot of sense: The important forces are local. In this simple case we can employ a technique called ghost lines. Here we split up the model grid on and distribute to different machines, but with an important trick: We add copies of data on the edges.

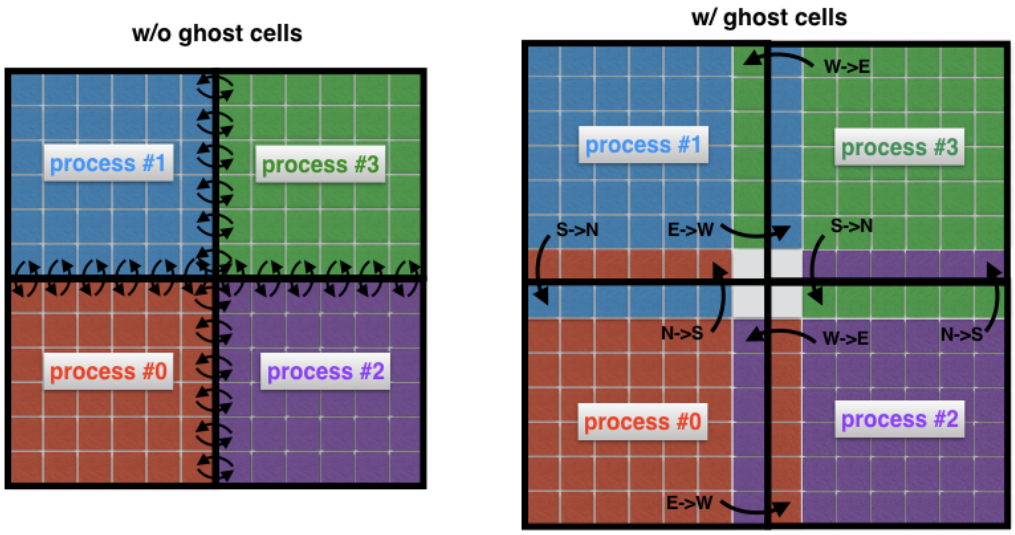

Figure 1: Each proccess is responsible for a given area $N \times N$, we send data of shape $(N+2) \times (N+2)$ so that each process has a local copy of all the data that it needs to do calculations. Then, after each iteration, we exchange the so-called ghost lines with our neighboors only. This keeps intra-proccesses communcation to a minimum.

We can now do our computations completely local. Then after finishing one iteration, we only need to inform our neighbours about what happened on the so-called ghost lines. This keeps communications to a minimum, and works fine for all explicit schemes, but in implicit schemes we can run into trouble.

2.1.1 Implicit vs Explicit time stepping scheme

When we a running fluid simulations we are interested in solving various differential equations. There exists a myriad of different numerical schemes, but we will illustrate an important difference between two classes, implicit and explicit schemes, which affects stability. If a scheme is unstable the solution will grow in time and explode to infinity.

To illustrate, lets look at a common and simple explicit scheme the "Forward in time central in space" (FTCS) scheme. We will take the heat equation as an example:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \tag{1}$$

We then take a finite difference approximation:

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = \alpha \frac{T_{i-1}^n - 2T_i^n + T_{i+1}^n}{(\Delta x)^2} \tag{2}$$

Where the subscript i denotes spatial coordinates (in this case on a 1d grid), and the superscript denotes timestep. We can rearrange this equation for T_i^{n+1}, which is then the state of the variable T for the next timestep:

$$T_i^{n+1} = \lambda(T_{i-1}^n + T_{i+1}^n) + (1 - 2\lambda)T_i^n \tag{3}$$

With $\lambda = \frac{\alpha \Delta t}{(\Delta x)^2}$. Here we neglected to analyse any errors due to the discretization.

If we had chosen another set of finite difference approximations, we could have arrived at a situation where the solution for T_i^{n+1} depended on other T_{i+1}^{n+1}, or other combinations of spatial variables. This would lead to a system of equations we then needed to solve to get to our new system state. For example, lets choose a backwards difference in time and a centered second order in space:

$$\frac{T_i^n - T_i^{n-1}}{\Delta t} = \alpha \frac{T_{i-1}^n - 2T_i^n + T_{i+1}^n}{(\Delta x)^2} \tag{4}$$

To make the idea a little clearer, by a trick of notation, make $n \to n+1$ and rearrange:

$$T_i^{n+1}(1+2\lambda) - \lambda(T_{i-1}^{n+1} + T_{i+1}^{n+1}) = T_i^n \tag{5}$$

This is a tri-diagonal system:

$$\mathbf{M}T^{n+1} = T^n \tag{6}$$

$$T^{n+1} = \mathbf{M}^{-1}T^n \tag{7}$$

Where the matrix $\mathbf{M}$ looks like (for circular boundary conditions):

$$\begin{bmatrix} 1{+}2\lambda & -\lambda & 0 & \cdots & -\lambda \\ -\lambda & 1{+}2\lambda & -\lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & -\lambda & 1{+}2\lambda & -\lambda \\ -\lambda & 0 & 0 & -\lambda & 1{+}2\lambda \end{bmatrix} \tag{8}$$

By inverting this matrix, we need (in more realistic scenarios than this, imagine α varying in space and time) information about the whole system. This is not great, but there are good reasons why we do this. It turns out that this allows us to take much greater timesteps as implicit methods are unconditionally stable.

To reason about this strong claim, we first introduce the concept of Von Neumann stability analysis[45]. Lets start by considering the stability of the heat equation we looked at before. We can write the solution T_j^n as a sum of Fourier components:

$$T_j^n = \sum_k \rho_k^n \, \mathrm{e}^{ikj} \tag{9}$$

Note that the superscript on ρ should be understood as an exponentiation. We can think of ρ as an operator that advances the solution to the next timestep, and we will refer to it as the amplification factor. For a given scheme to be stale it is then required that the norm of this amplification factor to be below unity:

$$|\rho| \leq 1 \tag{10}$$

Which should hold for the entirety of the space and time domain of our solution. By inserting (9) into our finite difference equation, we can find conditions for numerical stability of our schemes. For the explicit FTCS scheme (3) we get:

$$\sum_k \rho_k^{n+1} \, \mathrm{e}^{ijk} = \lambda \left(\sum_k \rho_k^n \, \mathrm{e}^{ik(j-1)} + \sum_k \rho_k^n \, \mathrm{e}^{ik(j+1)} \right) + (1-2\lambda) \sum_k \rho_k^n \, \mathrm{e}^{ikj} \tag{11}$$

Because this should hold everywhere we can remove the summation and require the condition hold for all k. We can also divide out a factor of e^{ikj}. This gives us:

$$\rho^{n+1} = \lambda \left(\rho^n \, \mathrm{e}^{-ik} + \rho^n \, \mathrm{e}^{ik} \right) + (1-2\lambda)\rho^n \tag{12}$$

$$\rho = 2\lambda \cos(k) + (1-2\lambda) \tag{13}$$

$$\rho = 2\lambda(\cos(k) - 1) + 1 = -4\lambda \sin^2(k/2) + 1 \tag{14}$$

Where we used a half-angle trigonometric identity in the last line. The worst case is that the $\sin^2$ term will be unity and we are left with:

$$\rho = -4\lambda + 1 \tag{15}$$

Any negative value for λ will result in ρ being larger than unity. We investigate when a postive value of λ will result in the amplification factor being smaller than -1:

$$-4\lambda + 1 \geq -1 \tag{16}$$
$$-4\lambda \geq -2 \tag{17}$$
$$\lambda \leq 1/2 \tag{18}$$

For this particular scheme we must choose λ, and in turn our time steps and grid size, to be between zero and one half.

Doing the same exercise for the explicit scheme[1] it will be evident that we have *no* requirements on λ. Our solution will be stable with arbitrarily large timesteps.

2.1.2 Accelerating Veros — Profiling to find the slow bits

Currently Veros can be GPU accelerated trough various means. Using Bohrium[25] we can essentially `import bohrium as np` and in-place replace Numpy functions with GPU accelerated versions. We can do similar things with other acceleration backends like Tensorflow[30] and JAX[8]. Here we leverage the enormous efforts that have been applied for accelerating machine learning into running ocean simulations. To get a sense for the relative performance of different backends, Veros provides a series of benchmarks to target[17].

In this thesis we are targeting the "hardest" benchmark containing an implicit scheme where we need to solve a system of equation spanning the entire 3d grid. To arrive at this system of equations we also have to perform many array operations to setup the system. All in all, this makes it a balanced benchmark that should provide insight into how a given backend would perform at the tasks of computing large-scale models.

To optimize any code, it is imperative that one performs profiling to see exactly where the "hot" parts of the program are. To profile this particular program is not trivial due to the fact that the regular Python profiling tool CProfile[35] does not register Numpy array operations, as no functions gets called. A Numpy line of code like:

```
1 A[large_index_matrix] = large_value_matrix
```

Might likely be a slow part of you program, but in CProfile it will register all its run-time inside the calling function leaving us no explicit information where our code becomes slow. As Python is an interpreted language, we can do better and perform benchmarking line-by-line. Here the de-facto standard tool is `LineProfiler`[22]. Running our target benchmark trough the profiler, we investigate which parts of our algorithm is taking up most of our run-time:

1. The tri-diagonal matrix solver.
2. All other code. This mainly comprises of Numpy array operations.

This is still a subset of a full Veros simulation but should still provide some information about what kinds of acceleration would be beneficial to implement.

[1] See apendix.

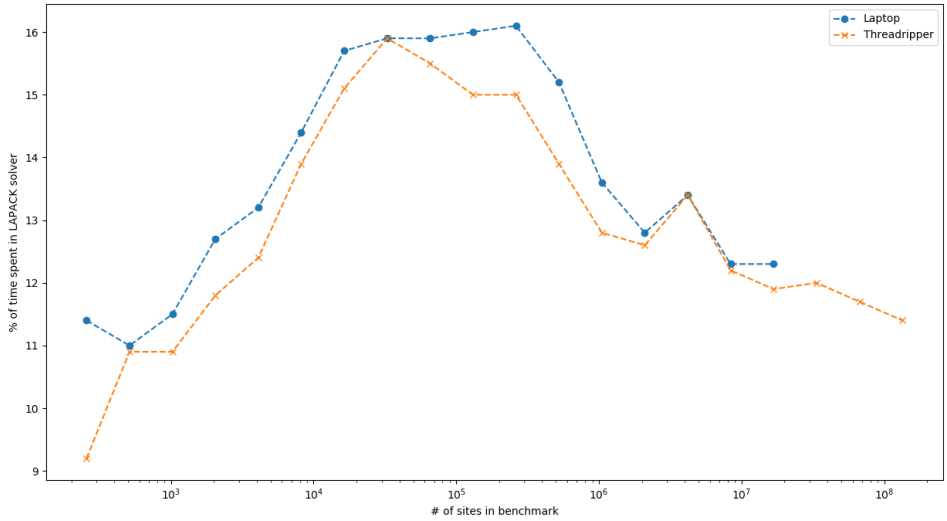

Figure 2: We investigate which parts of the Turbolent kinetic energy benchamrk takes the most relative runtime. Here we plot the ratio of time spent solving of a tri-diagonal system with LAPACK[5]. Plotted is the benchmark performed on two machines: A laptop with a Intel i5-7200U processor and 16 GB of ram, and the so-called Threadripper which includes a AMD Ryzen Threadripper 1950X 16-Core processor and 128 GB of RAM. To make the benchmark somewhat comparable we run the code using only one thread by specifying environment variables like `OPENBLAS_NUM_THREADS=1`, and enforce the same BLAS backend.

This illustrates the point that it is important to accelerate the simple array operations.

2.2 Representing numbers on computers

Data storage in computers are fundamentally done in bits of ones and zeros. A given series of ones and zeroes leaves many options for the choice of how to interpret the data. Most modern computer hardware run on 64 bits which means that most operations in the processor is done on so-called words which are 64 bits wide. A huge advantage to this is that modern programmers very seldomly have to worry about implementation details when working with numbers in computers. 64 bits are large enough that overflows, underflows and precision are not issues in day-to-day use. A disadvantage is that they are twice as large as the 32 bit standard they superseded. Usually this is a worthwhile trade-off as computer storage and memory has become very cheap[7].

High-performance computing comes with additional requirements. Often we do not worry so much about the hardware footprint but the memory throughput. Lets take as an example the aforementioned ghost-line setup. Here we wish to minimize communication latency as much as possible to keep the machines working to their full potential, and not waiting around for synchronization. If the data is half the size, the transfer speed is incrased. While this is somewhat of a trivial example, this is important in many aspects of HPC. Smaller data structures fit more information into cache lines, for a given size of a vector-computer you can compute on more numbers, etc.

Not having to worry about precision and overflow comes at a significant cost in HPC. Here we will give a short introduction to various standard methods of storing numbers.

2.2.1 Integers

As we are limited by storing zeros and ones in the computer, it naturally lends itself to a binary, or base 2, representation. In this representation the 8 bit (or 1 byte) binary number will be stored as:

$$1001\ 1101 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \tag{19}$$
$$= 128 + 0 + 0 + 16 + 8 + 4 + 0 + 1 = 157 \tag{20}$$

We *store* 1001 1101 but it represent the decimal number 157. If we wanted to store a negative number, we would interpret the binary data using a scheme called twos-complement:

$$1001\ 1101 = -1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \tag{21}$$
$$= -128 + 0 + 0 + 16 + 8 + 4 + 0 + 1 = -99 \tag{22}$$

This scheme allows us to map memory to a given natural number as long as we have enough bits.

2.2.2 Fixed point representation

If we have some real number x, we can represent it in binary by introducing a "point" like the familiar decimal point. For example, the decimal number 5.25 would be represented by:

$$101.01 \tag{23}$$

To represent negative numbers, it turns out that using twos-complement preserves the regular arithmetic operations. Lets introduce the notation `Fp(sign, wordSize, fracBits)`. This is a `sign`ed number type of `wordSize` bits in total of which `fracBits` are fractional. Lets write the numbers 8.25 and -4.5 in the fixed point variable `Fp(1, 8, 2)`:

$$8.25 : 001000.01 = 2^3 + 2^{-2} \tag{24}$$
$$-4.5 : 111011.10 = -2^5 + 2^4 + 2^3 + 2^1 + 2^0 + 2^{-1} \tag{25}$$

We can then take the sum of these numbers like a regular sum:

$$001000.01 \tag{26}$$
$$111011.10 \tag{27}$$
$$1000011.11 = 000011.11\,(\text{truncated}) = 3.75 \tag{28}$$

Which was the expected result. Throwing away the overflow bit depends on which case (two negatives, one negative of large magnitude and a negative etc) is being added, and you must implement logic to handle this. To invert a number, we take the logical (NOT) of each bit and add the smallest representable number to it:

$$5.0 = 000101.00 \tag{29}$$
$$-5.0 = 111010.11 + 000000.01 = 111011.00 = -2^5 + 2^4 + 2^3 + 2^1 + 2^0 = -5 \tag{30}$$

As should be obvious, when adding or subtracting we are not committing any arithmetic errors in the sense that what is represented is calculated exactly. This is *not* true for multiplying and divisions. Furthermore, we actually must introduce rounding in general to make this operation work. First, a multiplication algorithm on binary integers works like the regular grade-school multiplication on decimals, except that the times table for binary multiplication is extraordinarily simple. Lets illustrate

with the decimals numbers 3.25 and 4.75.

$$\underline{000100.11 \times 000011.01} \tag{31}$$
$$000011\ 01 \quad (1 \times 000011.01) \tag{32}$$
$$0000110\ 1 \quad (1 \times 000011.01) \tag{33}$$
$$00000000 \quad (0 \times 000011.01) \tag{34}$$
$$00000000 \quad (0 \times 000011.01) \tag{35}$$
$$00001101 \quad (1 \times 000011.01) \tag{36}$$
$$00000000 \quad (0 \times 000011.01) \tag{37}$$
$$00000000 \quad (0 \times 000011.01) \tag{38}$$
$$\underline{00000000} \quad (0 \times 000011.01) \tag{39}$$
$$00000001111.0111 \tag{40}$$

As can be seen here, we get a number that is wider than our input numbers. This means that we have to scale our results, if we wish to stay in the same variable type. In the simplest case, but other rounding modes exists, we simply right shift until we have the desired amount of fractional bits. This will truncate the number, and essentially always round down. Whatever method we choose, we introduce an arithmetic error. This is different from errors originating from the precision of our variable in encoding a given real number.

The trick to using fixed point in hardware is that we can reuse all the circuits for doing integer arithmetic as the algorithms are all the same. The only difference is in how we interpret the resulting bits when printing to screen.

2.2.3 Floating point numbers

To store real numbers we generally use a standard set by IEEE-754[21]. This standard defines the floating point numbers for 32 and 64 bits word-sizes. A floating point number consist of some number of bits:

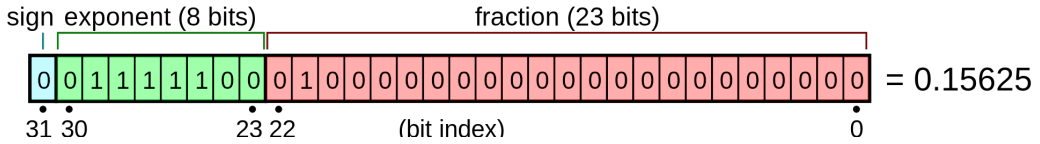

Figure 3: Bit pattern of a 32-bit IEEE-754 floating point number.

These bits represents a decimal number according to the formula:

$$x = (-1)^s \cdot (1 + f) \cdot 2^{e-b} \tag{41}$$

Where x is the decimal value of the number to represent, s is a sign bit (0 if positive, 1 if negative), e is the exponential part, where the bits the make up this number are understood as an unsigned integer, while b is a fixed bias used to effectively make the exponent a signed integer while saving us 1 bit by building it into the standard. For the fractional part we do something similar: f is essentially a fixed-point with no integer part, and the standard implements the implicit 1 bit. Lets take an example. The IEEE float is defined as having 8 exponent bits, $n_e = 8$. To properly offset that we get a bias of: $b = 2^{n_e - 1} - 1 = 2^{8-1} - 1 = 127$. Now lets decide on some values for our sign, fractional and exponential part:

$$s = 0 \tag{42}$$
$$f = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + \cdots = 0.25 \tag{43}$$
$$e = 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 124 \tag{44}$$

These bits correspond to the decimal number:

$$x = (-1)^0 \cdot (1 + 0.25) \cdot 2^{124-127} \tag{45}$$
$$x = 1.25 \cdot 2^{-3} = 0.15625 \tag{46}$$

To do arithmetic on floating points we can essentially do integer arithmetic on their bit patterns. Lets take a simple example and add the floating point number 0.15625 we just showed the bit pattern for, to the floating point number 2, which will have the pattern:

$$s = 0 \tag{47}$$
$$f = 0 \cdot 2^{-1} + 0 \cdot 2^{-1} + \cdots = 0 \tag{48}$$
$$e = 0 \cdot 2^8 + 1 \cdot 2^7 + \cdots = 128 \tag{49}$$
$$2 = (-0)^1 \cdot (1 + 0) \cdot 2^{128-127} \tag{50}$$

Our summation is then:

$$2 + 0.15625 = (1) \cdot 2^1 + (1 + 0.25) \cdot 2^{-3} \tag{51}$$

Notice how it would be really convenient if the exponents where the same so that we could factor it out. What we can do is right shift the number with the smaller exponent, throwing away the least significant bits, while incriminating the exponent until we get a match:

$$x_1 = 2 \qquad x_2 = 0.15625 \quad \text{(shifted)} \tag{52}$$
$$f_1 = 0000\,0000 \cdots = 0 \qquad f_2 = 0001\,0100 \cdots = 0.078125 \tag{53}$$
$$x_1 = 1 \cdot 2^1 \qquad x_2 = 0.078125 \cdot 2^1 \tag{54}$$

When shifting, notice the implicit leading one, which we did not actually store but kept as a part of the standard, now appears. This leaves us with adding the bits of the two fractional parts to get the sum, as we have factored out the exponent. Here we must be careful to remember the implicit leading bit on the binary pattern of x_1, and which is now not present on x_2, after shifting.

$$1.0000\,0000 \tag{55}$$
$$\underline{\quad .0001\,0100} \tag{56}$$
$$1.0001\,0100 \tag{57}$$

Here we arrive at our answer:

$$s = 0 \tag{58}$$
$$f = 0001\,0100 = 0.078125 \tag{59}$$
$$e = 0100\,0000 = 128 \tag{60}$$
$$x = (0)^s \cdot (1 + 0.078125) \cdot 2^1 = 2.15625 \tag{61}$$

To perform multiplication we can essentially add the exponents and perform the same multiplication we did for fixed-point on the fractional part. Some care must be taken to treat the hidden bit as we did with addition.

An important thing to consider is that when we did addition with floating points we had to throw away useful information when aligning the exponents. Throwing away information is what can lead to effects such as catastrophic cancelation[11], where simple arithmetic leads to wildly wrong answers.

2.2.4 Posits

Posits can purportedly "beat floats at their own game"[15]. Posits provides a larger dynamic range than equivalently sized floats while at the same to giving greater accuracy around unity. Downside to this

is, that they are less accurate for very large numbers at the edges of their range. There exists tentative developments of implementing posits in hardware[46][46], but also interesting simulation results in the fluid-dynamics world[24].

A posit words looks like:

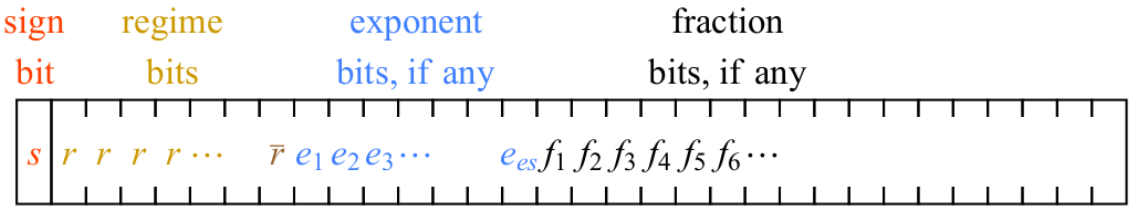

Figure 4: Bit pattern of a posit. Compared to the IEEE-754 float, we have an added `regime bits` field. This field provides variable dynamic width and dynamic length of the fractional part. Illustration taken from [24]

This represents a decimal number:

$$x = (-1)^s \cdot useed^k \cdot 2^e \cdot (1 + f) \tag{62}$$

Again we can make a choice: How many exponent bits would we like? If we choose this number, and the word size (eg. 32 or 64 bits), we get a well defined type. A keen reader will smell the familiarity with the IEEE float. We still employ a fractional and an exponential part. But a new factor $useed^k$ has been introduced. k can be read from the so-called `regime` bits following the rules that:

- The regime bits are the series of same-bits immediately following the sign bit
- If these bits are zeros: $k = -n_r$ but if they are ones, $k = n_r - 1$.

The value of $useed$ is defined to be $2^{2^{n_e}}$ where n_e is the number of exponential bits. An example posit can be shown:

$$01011001_{\text{Posit}(8,1)} = (-1)^0 \cdot 4^0 \cdot 2^1 \cdot (1 + 2^{-1} + 2^{-4}) = 3.125 \approx \pi.$$

Figure 5: Pi represented in a Posit(8, 1) (ie. 8 bits, 1 exponent bit) format. Sign bit is illustrated in red, regime bits in orange ending regime bit in brown, exponent bit in blue and finally fractional bits in black. Illustration from [24].

This format has wider range than floats while even improving accuracy while close to unity.[24]

2.3 Representing vectors and matrices on computers

Multiple possible abstractions can be made for keeping arrays of numbers in memory. We will mainly discuss contiguous arrays, as this strategy will yield the most benefits in our application of running climate simulations on FPGA hardware.

2.3.1 Striding

In computers, memory is fundamentally addressed linearly. You have an address space which maps an integer to a position in memory, where you can put information. The standard way to implement multi-dimensional arrays is to use a technique called striding. First, lets establish an indexing to make

sense of multi-dimensional arrays. Here we can take the usual notation used in linear algebra:

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \tag{63}$$

To make sense of this in a linear way, we could arrange the entries of the matrix like so:

$$\overline{A} = [A_{00}, \quad A_{01}, \quad A_{02}, \quad A_{10}, \quad A_{11}, \quad A_{12}, \quad A_{20}, \quad A_{21}, \quad A_{22}] \tag{64}$$

The correspondence between elements of A and $\overline{A}$ is then:

$$A_{ij} = \overline{A}_{3 \cdot j + i} \tag{65}$$

In generel, the strides of A will be an array consisting of the numbers that will offset us into the correct place in memory, if we increment the corresponding index by one:

$$A_{stride} = [3, 1] \tag{66}$$

This lets us store data of multiple dimensions in memory in a convenient way.

The way we selected the direction of our array was arbitrarily chosen: We could have selected the columns to be the leading dimension and have a memory representation like:

$$\overline{A} = [A_{00}, \quad A_{10}, \quad A_{20}, \quad A_{01}, \quad A_{11}, \quad A_{21}, \quad A_{02}, \quad A_{12}, \quad A_{22}] \tag{67}$$

This layout is known as column-major (the layout first presented is likewise referred to as row-major). While this choice might seem innocuous, it *can* have performance implications. An effect caused by the fact that reading from memory into the registers where the calculations is performed is generally slow. To optimize this reading operations, you will want to read large chunks at a time. CPUs will automatically make this optimization by fetching not just the data you are asking for, but also the data in the immediate neighborhood which is referred to as a cache line[38]. To illustrate how dramatic this effect can be, we can design a small program calculating a matrix-vector product, and analyze the difference in run times:

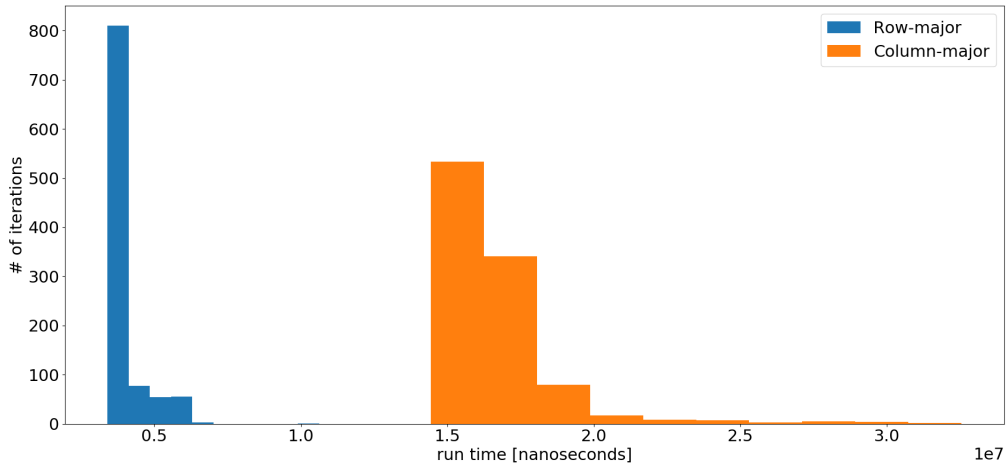

Figure 6: Difference in run times is observed for a matrix-vector product $c = \mathbf{A}b$. Matrix A has dimensions of 1024×1024 and vector b a length of 1024. The difference in run time is associated only with the exchange of the order in the nested for-loop calculating the product. Program is written in C++ and compiled with `g++` with standard flags, and run on a laptop on an Intel i-5 7200U processor.

Simply permuting the nested loop produced a significantly slower program.

2.3.2 Broadcasting

When working with array data, it can be useful to perform the same operation on all the elements. For example, let say we want to scale every entry in our array, or we want to divide every rows pairwise with the elements from a vector. These kinds of operations can be conveniently written in terms of broadcasting.

To broadcast an operation we imagine stretching it on empty, or singleton, axis. Ie.:

$$O_{ij} = A_{ij} \cdot B_i = A_{ij} \cdot B_{ii} \tag{68}$$

Here we trivially promote the vector B to a matrix that allows it to be multiplied elementwise by A. We can also imagine other situations: For example taking an outer product by forcing different indices:

$$O_{ij} = I_{ij} \cdot A_i \cdot B_j \tag{69}$$

Where I is the identity. To determine if a broadcast is possible, we can write the dimensions of the operands like so:

$$\text{Shape of } A\text{:} \quad 9 \quad 4 \quad 1 \quad 5 \tag{70}$$

$$\text{Shape of } B\text{:} \quad \quad 4 \quad 6 \quad 1 \tag{71}$$

$$\text{Shape of } O\text{:} \quad 9 \quad 4 \quad 6 \quad 5 \tag{72}$$

First, we align arrays of different number of dimensions along their *leading* dimension. Then, for each output the inputs must either be equal, or one of them has to be one. I.e:

$$O_i = \max(A_i, B_i) \quad \text{iff}(A_i == B_i \vee A_i == 1 \vee B_i == 1) \tag{73}$$

If this is not satisfied the broadcasting operation is not well-defined and should fail.

2.3.3 Why is broadcasting useful?

In many cases in scientific computing the operations involved naturally lend themselves to broadcasting. A great example of this is computing partial derivatives in an uneven grid, which is often used in the z direction[45]. Imagine we have some field $f(x, y, z)$ which we have computed on non-equidistant grid points. To get an estimate for the partial derivative we would do something like:

$$\frac{\partial f}{\partial z} = \frac{f(x, y, z + \Delta x) - f(x, y, z - \Delta z)}{2\Delta z}$$

Here Δz is a essentially a 3d field, but the symmetries in the x and y directions makes it possible to encode the information in a vector. Physically we can think of the partial derivative as the difference between "slices" of the field. Thinking in this way lets us write the equation for the partial derivative in a compact and easily readable way[2]:

```
dfdz = (f[:, :, 1:] - f[:, :, :-1]) / 2 / dz[:]
```

If this is Numpy code, we have many benefits. Firstly the code is very easy to debug If we miss an index it is obvious compared to missing an index in a thrice-nested for loop. As an ugly demonstration, here is comparable code written out, but with caveat: There has purposely been placed an indexing error. Try to see how fast you can spot it.[3]

```
dfdz = zeros(x_dim, y_dim, z_dim-2)

for i in range(x_dim):
    for j in range(y_dim):
        for k in range(1, z_dim - 1):
            dfdz[i, j, k] = (dfdz[i, j, k-1] - dfdz[i, j, k+1]) /  2 / dz[k]
```

[2] We assume that `f` is padded such that appropriate boundary conditions are applied when taking the finite difference.
[3] The `k+1` and `k-1` should be swapped. "Right" side should be postive, left side negative.

Secondly the computations are offloaded to fast C code. This can be a huge benefit in writing high-level Numpy code, but gaining the speed of low-level C. As a quick example, lets look at some code computing the pairwise distance matrix between two arrays of 3d coordinates which is an operation commonly used in astrophysics type simulations to calculate the gravitational force experience between objects:

```
def aj_noloop(A, B):
    B = np.reshape(B, (-1, 3, 1))
    A = np.reshape(A, (-1, 3, 1))
    aj = np.sqrt(np.sum((A.T - B)**2, axis=1))
    return aj

def aj(A, B):
    d = np.zeros(shape=(N, N))
    for i in range(N):
        for j in range(N):
            d[j, i] = np.sqrt(np.sum((A[i] - B[j])**2))
    return d
```

Running this code we see that for approximately 1000 objects, we get a two orders of mangitude faster execution. Some clever broadcasting to essentially take an outer product gives us a massive speed boost. This speedup is achieved solely by going to optimized C code, and not from any parallelism. There is no free lunch here: Taking the outer products produced a huge amount of unnecessary computations, costing us both memory and processing, but the raw speed up of doing C compared to Python gave us a nice speed boost. If we scaled this to a very high number of objects, at some point we expect the loop version to win, as this is running in $O(n^2)$ while the no loop vesion is running in $O(n^3)$. This suggests that while we should generally employ a coding style with no loops when trying to implement HPC code in Python so that our accelerators can take over, we should not loose our heads and be careful about unnecessary computations.

2.3.4 Alternative methods

If we have really sparse arrays we can opt to keep the data in other formats. Different implementations strategies exists, but boil down to keeping only non-zero entries along with some coordinate information around. Keeping only the non-zero entries can save you very large amounts of RAM, as this this will grow like N^d with d the dimensions of the array, while the non-zero entries of sparse matrices will grow with a smaller exponent. This obviously entails different implementations of the operations on the matrix.

Totally different methods for keeping arrays also exists which have obvious draw-backs for our purposes. An example of this could be a linked list. A linked list is a data structure where we do not reference elements by their indices, but instead keep a reference to the next element together with the data contained. This allows us to easily find the next element, but since the data could be place in-principle at a random memory address we will not get the benefits of caching and definitely not the benefits of doing broadcasting. The upside to linked list are that you can easily add elements to the beginning and ends, and that they do not need to be of a pre-defined size.

2.4 FPGA details

FPGAs promises to compute at a lower power footprint than competing technologies. As is, FPGA hardware is more expensive than comparable CPU/GPU processors, but recent approaches like Amazon F1[3][36] instances and an increased focus on usability with HLS might make them a new target for silicon manufacturers[4], and in the future be produced at a more attractive price point. From a practical standpoint, the advent of datacenter cards like the Xilinx Alveo family[47] is also very promising. This approach makes targeting FPGAs feel like running code on regular cluster computers.

2.4.1 Computational model on an FPGA

On an FPGA we have abandoned the Von Neumann architecture and are running directly on (reconfigurable) hardware circuits. This gives us the advantage that we can move registers closer to the computation, and we can create deep pipelines of operations than can process data in a staggered manner.

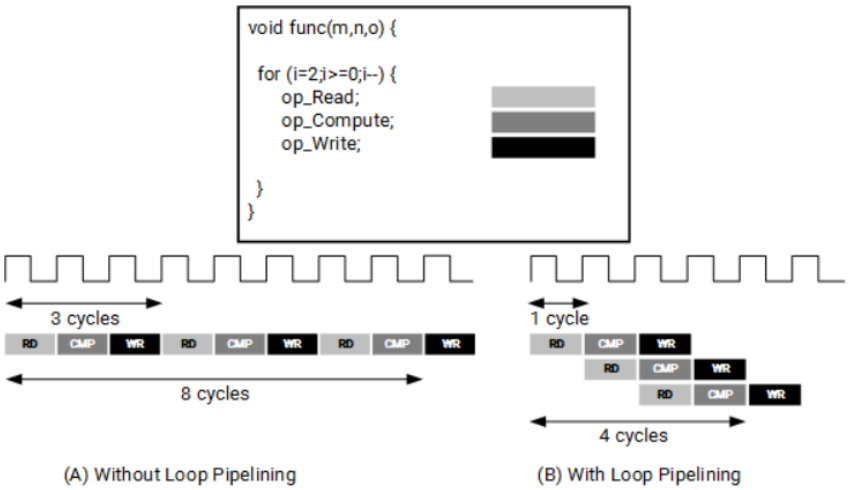

Figure 7: We can build pipelines to efficiently execute code. In this trivial example we can imagine being limited by a bus allowing us to `read` every clock cycle: To maximize our throughput, we need to saturate this bus. In the FPGA we can build systems (circuits) that does the required operations and then stream our data trough them.

FPGAs do not have cache misses, network packets, context switching etc. This means that a program executing in n cycles will always execute in n cycles. Coupling this feature with the potential for very low latency makes FPGAs an attractive choice for early-stage data processing coupled to high data throughput sensors. In this project we hope to leverage the pipelines and the data locality features of the FPGA to move the data needed for a given computation quickly trough the hardware.

FPGAs can come equipped with regular memory as we know it from our computers, which have some benefits. Implementing the RAM in the reconfigurable part of the chip costs a lot of area, in the sense that storing one bit of information reliably costs more than one transistor. As a result, if we want lots of memory we need very large chips. This is not feasible: Due to fabrication limitations the FPGAs will have random errors in their silicon. The rate (per area) of these random errors essentially limits the size of the average chip to be proportional to 1/err rate, as chips with errors needs to be discarded. The downsides of implementing the RAM separately is that it will be slower and further away from the parts of the FPGA doing the computation.

2.4.2 HLS vs RTL

Traditionally we would develop FPGAs like we would ASICs and everything would be implemented in RTL. When writing RTL code we are using an abstraction to specify the shape of the hardware. This amounts to controlling the clock and then applying transformations on our registers. Registers are analogues to variables, in the sense that they contain data. This production flow is not very suitable for programmers, who does usually not have a deep understanding of the specifics of the underlying hardware. Furthermore it is not very productive to write larger applications, if we need to manage every register "by hand". The difficulties does pay off as RTL will generally win in terms of performance when compared to HLS implementations. We have also experience other issues related to tooling. For example, when working with HLS on some systems, only a fraction of the available bandwidth could be utilized. It has been shown that the abstraction of going to HLS reduced the number of lines of

code by a factor of 10 compared to equivalent RTL code[44]. This makes HLS manageable in larger systems where RTL, implementation details notwithstanding, becomes too complex.

HLS allows more traditional software development work-flows targeting FPGAs[9]. Generally we work on three levels of developments and abstraction, with approximately exponentially increasing compile times:

1. software emulation (order of seocnds)
2. hardware emulation (order of minutes)
3. hardware (order of hours)

The first software emulation layer allows us to write unit tests for whatever application we wish to accelerate. This gives us confidence that we have written a functionally correct program. By writing our HLS code in a language like C, we are able to run it on our regular hardware. The second layer allows us to optimize the program for the FPGA as the hardware emulation is cycle accurate, although with some caveats[51], the main one being that the memory model is approximate. This means that performance estimates can be off, especially if we are close to saturating the bandwidth of our memory. The third layer allows us to actually produce a working binary, referred to as a bitstream, to run on the FPGA hardware.

2.4.3 HLS coding style

Getting performance out of an FPGA turns to be not a simple feat. Simply running existing code or naively porting to C++ will not grant performance[28]. We abandon the way of thinking adopted from programming accelerators in general, where we distribute our kernel to many instances of our kernel each responsible for subset of the program. Instead we write one kernel and optimize that with various pragmas. For example, imagine we have some vector of 1024 elements that we want to apply some function `compute` to. In the regular GPU accelerator we would writing something that roughly corresponds to:

```
host code:
    broadcast-to-kernels(data, N)
    retrieve-from-kernels(result, N)
```

With:

```
kernel code:
    k = which_kernel_am_i()
    for (i = k * 1024/N, i < (k + 1) * 1024/N, i++):
        result[i] = compute(data[i])
```

Where each kernel is responsible for some amount of the computation. Instead when targeting an FPGA we get:

```
host code:
    send-to-kernel(data)
    retrieve-from-kernel(result)
```

With a loop like:

```
kernel code:
    for (i = 0, i < 1024, i++):
        #pragma unroll factor = N
        result[i] = compute(data[i])
```

The difference is, we only enqueue 1 kernel in the FPGA example, but N kernels in the GPU example. On the FPGA we get concurrency by specifying a pragma which tells our compiler that we want to copy the loop N times. In principle this is enough. But the tricky part is that we need to recognize

that we will run into performance issues due to limited read capabilities from our memory. Instead we need to deploy other tricks, like copying the memory first into local memory, and then spreading these to different physical memory blocks. Employing an array_partition pragma allows us to create the right type of memory structure, and our kernel code ends up looking like:

```
kernel code:
#pragma array_partition local-data factor = N
local-data[]
read-data(data, local-data)

for (i = 0, i < 1024, i++):
    #pragma unroll factor = N
    result[i] = compute(local-data[i])
```

As is evident, we are not saved from writing memory optimized code if we want performance on our FPGA accelerator.

Another important pragma to know is `dataflow`.

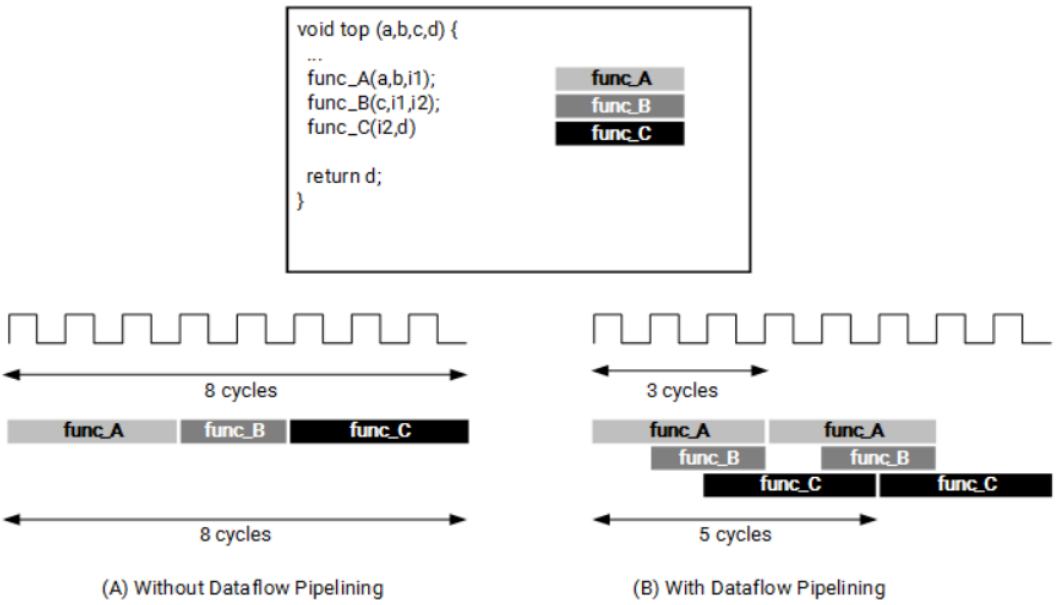

Figure 8: When using dataflow pragma the compiler will analyze your code and allow paralell execution. This parrallezation can be limited by data dependencies. This sometimes means re-writing your code to limit loop-carried dependencies if possible.

Using this pragma we can get overlapping execution of multiple functions. An almost trivial example could be a loop like:

```
for (i, 1024) {
    read(data[i], local_data[i])
    compute(local_data[i], result[i])
    write(out[i], result[i]
}
```

Here write depends on compute which depends on read. The dataflow optimized version is able to finish more iterations in less cycles. It should be noted that dataflow is similar to pipelining, but on a task level.

3 Accelerating on an FPGA — strategies

It is not obvious which way to perform the acceleration of Veros on an FPGA. It would be favourable to have an extendable system that can generalize to accelerate the entirety of Veros, and other simulations that can be written from the same powerful building blocks of Numpy array operations with broadcasting. Ideally this should not require an FPGA expert to setup, but could should be an somewhat automatic system akin to the way we can accelerate Veros on GPUs by swapping in a backend which understands the Numpy API.

In this chapter we will discuss various avenues of achieving this task.

3.1 Leveraging AI-on-FPGA

AI is a huge area of research and has seen many interesting ways to speedup the costly training and inference operations. At Google, they have built special purpose TPU[14], and for GPUs we are seeing different number encoding such as 16 bit floats[32]. But in the FPGA space we have also seen developments in this area[49]. Mostly this research have been focused on running inference on FPGAs. Here we leverage the fixed (and low) latency features coupled with good parallelism and power characteristics to build robust inference systems. These features become especially important in embedded devices needing to process high bandwidth data. In this class of devices, power requirements are often an important feature. Examples could be in drones[26] or in self-driving cars[18].

For our purposes it could be interesting to repurpose these performance optimized kernels performing neural net inference to do fluid dynamics, just like we could do for the computation backends of Veros. The problem set is not as unlike as a first glance might suggest. Common for inference in AI and fluid dynamics is the multi-dimensional arrays being operated on, while the operations performed are also mostly elementary arithmetic operations.

As it turns out, the available tools are not useful for our use case. The tools allows you to compile specific machine learning models to FPGA. These models come with important limitations. For example, you are not allowed to have any `if` statements in your code. The models also need to be pre-compiled by your machine learning framework, eg. PyTorch[33], before being processed by the FPGA tools, and we do not have access to any lower level functions that we could re-use.

Machine learning algorithms in general do not need slicing operations, which we rely on extensively in Veros. While it is possible to cast an arithmetic operation with a slicing operation to be a matrix product, this approach would be more suitable on TPU, and would require major rewrites of core Veros logic[1]. This approach proved to be a dead end.

3.2 Refactoring into HLS

3.2.1 Arithemtic kernels

We can imitate Numpy and write kernels responsible for adding, multiplying etc. This lends itself to translate Numpy code quite nicely, and it should even by possible to write a transpiler converting Numpy code to an intermediate target of HLS code and host code, which can then be compiled to target FPGA hardware. The class of problems we can solve is also large as the method should scale to all programs that can be written with broadcasting operations.

In an extreme case, we might even be able to build a Numpy translation layer that can run directly on the FPGA. While this might seem like a good idea, we argue that this will result in the FPGA essentially being converted into a (probably worse) GPU.

Writig arithemtic kernels, we need to work very hard to gain any appreciable speed-ups, as this problem is embarrassingly parallel. GPUs are essentially special purpose hardware made to solve this exact problem.

To stand a fighting chance, it is obvious that we need to keep the calculations local and exploit the pipeline capabilities of the FPGA to gain any appreciable speed ups.

The simplest way to implement this strategy would be to:

1. Write the necessary kernels
2. Create a host program calling the kernels

In this naive implementation the control flow of the code is handled by a host, which in principle could be the Python code. Here we would transfer data to and from the FPGA often and the kernels would not be able to talk to each other and produce pipelines. Most likely this will not result in a faster program.

It is possible to have kernels stream data between them, which might be a solution to this problem. Doing this requires writing a configuration file, which is essentially a call graph of the program, that is supplied in the compile stage of hardware synthesis. This seems like a promising strategy, but it is not clear how current FPGAs scale to operating on large amounts of kernels[50].

3.2.2 One single kernel

We can imagine going line-by-line in the Veros code and converting the operations into their equivalent nested loops in a HLS translation. Then, making sure we treat any loop carried dependencies and apply the proper pragmas, we should end up at a working FPGA implementation. Copying all this code by hand introduces many, many opportunities for bugs besides being very tedious to do. Furthermore, the fact the we need to carefully apply pragmas and manage memory to gain any real speedup prevents us from making any automated translation efforts.

The fact that there is no hope to have an automatic conversion from Numpy to this target also means that we would have to maintain two separate implementations of Veros and all other projects using FPGAs to accelerate their code. To make matters worse, this implementation would require substantial domain knowledge as writing HLS code is not as simple as writing C++.

3.2.3 Single kernel, arithmetic functions

A middle ground between one-kernel and multi kernel is to implement one `work` kernel and then define multiple functions that implements our operations.

Morally this should be similar to having kernels stream between them, as function calls should be instantiating the creation of a hardware block in the final design of the FPGA.

The downsides to this is that we have lesser control over the kernels. This will also affect our ability to swap kernels for hand-written RTL versions.

3.3 Refactoring into RTL

Writing huge programs in general is hard to manage. An increase in code size by and order of magnitude by switching from python to C++ seems not unreasonable, and on top of this, we have seen a 10x "penalty" in number of lines of code in going from HLS to RTL. By simply counting the lines in any `.py` file in a Veros installation we get a *very* rough feeling for the complexity of the codebase:

	numpy	C++	RTL
Approx # LOC	19441	200 000	2 000 000

Table 1: Rough sense of scale between implementations of veros.

Now consider the inevitable bugs introduced by writing this massive amount of code means that the complexity of the program makes it almost impossible for a single person to write in a year. Bugs are extra tricky to catch in this type of application. It is generally difficult to verify the correctness of

PDE solvers as the solution is not known. Unless something is obviously wrong with the output, there might be hidden bugs in the code making the result less accurate than it potentially could be. These bugs could be very innocuous indexing errors, like using the non-updated element in a finite difference calculation etc., which are very hard to catch.

This solution also suffers from maintainability problems, but to an even greater extend than the HLS version. In this case we need actual hardware experts to write our code.

While there are considerable downsides implementing the code in RTL will generally result in faster and more efficient execution. It could be worthwhile to target the most expensive operation and produce hand-written and highly optimized kernels to handle these workloads.

3.4 Chosen strategy

Due to practical concerns we can quickly reject the RTL approach. Managing such a huge project is not viable. Our chosen approach is to demote the arithmetic kernels to be functions, which are similarly translated to hardware and placed by the HLS tools. By writing one kernel that controls the compute flow, but delegates to arithmetic functions, we hope that by employing a `dataflow` pragma we can optimize the code to run as a deep pipeline.

Roughly this would correspond to a host code iniatating a `work` kernel which then calls functions:

```
host code:
    Transfer_to_work_kernel(model_state)
    retrieve_from_work_kernel(updated_model_state)

kernel code:
    u, v, w, ...rest_of_state = unpack_state(model_state)

    du = sub(u[..., 1:], u[..., :-1])
    du = div(du, dx)
    ...
    out = pack_state(u, v, w, ...)
```

We think this provides the optimal solution to our goals. Creating an automated system is still theoretically possible by analysing Python code. The resulting HLS code should be somewhat readable by anyone with programming experience by delegating to arithmetic functions.

All of this is achieved while still having a reasonable technical backing: The functions should be synthesis to hardware, and we can optimize these functions just as we could kernels.

4 Implementing striding and broadcasting on an FPGA

The choice of where to manage calculation flow has implications in the viable options for calculating strides in our array operations. We give an overview of the strategies we have examined, and then argue for our choice coupled with our above choice of using one `work` kernel delegating to functions using task-level parallelism.

4.1 Handlining broadcasting on an FPGA

We have written a series of kernels: `abs4d`, `add4d`, `gt4d` (greater than), `mult4d`, and so on, for each operation needed. We have chosen to only implement one "size" (in terms of dimensions) of each type of kernel. A smaller dimensional kernel can be emulated in a larger dimensional one by padding with singleton dimensions. Doing this, we can can reuse the hardware and the functions to be more general.

FPGAs excel in doing specific tasks while CPUs are great at doing generalized tasks, so this might seem like we are throwing away possible speed-ups by reusing parts of the hardware. We argue that any possible speedup gains by doing dimensional specific optimizations would be negligible as the smaller dimensional arrays would require less operations by the nature of being smaller slices of the bigger

arrays. This arguments should hold for the specific case of fluid dynamics, but in general as lower dimensional arrays can just scale to huge sizes.

Another way we can mitigate this problem is by shifting all the calculations over to the leading indices and then perform our pipelined optimizations on these parts of the loop. If we need to, we can still unroll the large chunks to achieve parallelization when we are working on very large arrays.

To actually compute the results of our operations we will for each kernel loop once over each output that needs to be computed. This is implemented in C as a nested for loop to be optimized (unrolling, pipelining etc) by the HLS tools. For each input/output we will calculate the relevant linear index as:

$$\begin{aligned}\text{ind} = \text{linear offset} + \sum_i \sum_j \sum_k \sum_l (i + \text{offset}_i) \cdot \text{stride}_i + (j + \text{offset}_j) \cdot \text{stride}_j + \\ (k + \text{offset}_k) \cdot \text{stride}_k + (l + \text{offset}_l) \cdot \text{stride}_l\end{aligned} \tag{74}$$

Where stride_i refers to the stride associated with the i'th dimension. Furthermore the offset is relative to the current output index, but the linear offset is relative to the start of the array.

The bounds on these sums will vary depending on which operation is being performed, but the strategy is to only touch each output index exactly once, and to only touch the output indices that we are actually interested in computing. For trivial cases the ranges are essentially the shape of the output array, but for more niche cases, eg. filling the first row of a matrix, they will not be trivial.

To exemplify how we calculate the strides and offsets we need for a given situation, we will work trough an example. In numpy notation we want to calculate the following:

```
O = np.zeros(shape=(10, 10))
A = np.random.standard_normal(size=(10, 10, 5))
B = np.random.standard_normal(size=(8,))

O[1:-3, 2:-2] = A[0:6, 4:, 2] + B[:-2, np.newaxis]
```

Here some pretty obscure indexing choices are taken to highlight the nuances in selecting the strides.

We have chosen to implement a slightly different notation than numpy where we use a simpler `slice` object. A couple of freedoms have been taken from you, and the rules are that:

1. You are only allowed to select ranges,
2. You are only allowed to specify the endpoints of these ranges by a negative number.
3. You can only select contiguous arrays.

Under these requirements the above numpy example would correspond to:

```
O[1:-3, 2:-2] = np.squeeze(A[0:-4, 4:None, 2:3] + B[0:-2, np.newaxis])
```

We had to introduce some hacks. Namely that because we selected a *range* instead of an index in the last dimension of `A`, we need to then squeeze the singleton dimension on the result for it to fit in `O`. Furthermore Python slice objects does not like the value of -0 that would logically fit with our requirements to use negative only numbers for endpoints of ranges, and we have to resort to a `None` value.

It turns out that for our calculations we can compute each arguments strides separately as long as the operation actually *can* be broadcasted. For each output element, we do not care about any relationship between the input, we just need their value. For each argument to a function we then calculate the strides.

First, we compute the "View Shapes" of one of the input objects and the output object. By view shapes, we mean the shapes that are defined by the requested index into the array. View shape is

easily computed by the range and negative-only endpoints:

$$\text{view\_shape\_A}_i = \text{A\_shape}_i + \text{A\_end\_offset}_i - \text{A\_start\_offset}_i \tag{75}$$

For array `A` this is:

$$\text{view\_shape\_A} = \{6, 6, 1\} \tag{76}$$

Next we compute the stride as if it were given by the actual shape of the array. For `A` that would be:

$$\text{A\_stride} = \{50, 5, 1\} \tag{77}$$

To do this calculation we start by setting the leading stride to a zero. Then, going backwards trough the shape we multiply our largest known stride by the corresponding dimension:

```
def stride_from_shape(shape):
    shape_len = len(shape) // how many dimensions are we calulating stride for?
    stride = [0] * shape_len // initalize as zeros.
    stride[shape_len - 1] = 1 // set leading stride to a 1

    for i in range(shape_len - 2, -1, -1):
        stride[i] = stride[i + 1] * shape[i + 1] // going backwards, fill the stride

    return stride
```

Now, sometimes the naive stride needs to be mutated. For example, the stride for array `B` will naively be computed as $\{1, 1\}$, as the `newaxis` will correspond to a dimension of 1. But to achieve the expected broadcasting we need to change the stride to be $\{1, 0\}$. To solve this problem we set any strides to zero which corresponds to a view shape of 1.

Following this approach allows us to mimic `np.newaxis` by inserting empty dimensions of size one, but also fixes issues where we selected an index from an array. Due to the restrictions in our range object selecting an indexing amounted to selecting a range with length one. But this will also produce a `view_shape` of one in the given axis.

Then we will need to collect any linear offsets. These are offsets from dimensions we are not looping over. In our example, we have "selected" (used a range with length 1) index 2 in the third dimensions of `A`. We need to take this offset into account by collecting it from every place in our view shape with a dimension of size 1.

Now, we need to do something very similar to the offsets, as with did the strides. Every time there is a view shape with a one it it, we put this to offset to zero, as the offsetting is handled by the linear offset.

4.2 Implementation strategies

There exists different specific ways we could implement the algorithm to run broadcasting. We will discuss different options, and choose one that balances ease of implementation with following the design goals of having a system that can be extendable, while still being able to run fast

4.2.1 Striding in host

Calculating the strides in the host and transferring them together with the data to the FPGA makes the most sense if we use a host controlled flow. A choice that would result in reducing the complexity of the HLS code, as we could then "trust" the computed strides and offsets and neglect bound checking in the kernel.

It is also possible to do the necessary stride calculations on the host, even tough we are following the `work` kernel pattern. By pre-compute all the strides on the host, transferring them, and then picking them out of FPGA memory when they are needed.

4.2.2 Striding on the FPGA

Alternatively we could implement the algorithm in HLS and have it run on the FPGA. This means we either transfer the ranges we wish to operate on together with the data, or if we are using a `work` kernel framework, we simple build the stride object and send that to the function responsible for computing the operation.

Adding extra computations to the FPGA will induce an overhead. But compared to the amount of operations involved in computing the results of the operations this is negligible, as the involved math is relatively simple and on very small arrays.

Doing the striding on the FPGA has the downside that we are additional new complex logic to the FPGA, which we prefer to keep small and compact to easier reason about run times and possible speed ups.

The algorithm for computing the strides is itself complicated slightly by the fact that we need to work with arrays of fixed sizes which we need to accommodate.

4.2.3 Choosing a striding strategy — pros and cons

Ideally we would calculate striding information on the host, as this simplifies the kernel code to be only "close-to-metal" operations. The FPGA only does the heavy calculations that we need, not fidgeting around with indices.

Both approaches has added code complexity. Pre-computing means we need to manage a list of strides to use at the correct operation all while transferring and unpacking correctly on the FPGA. On the flipside, performing striding calculations on the FPGA requires us to implement the striding algorithm in a somewhat more cumbersome environment.

We have opted to implement calculating strides on the FPGA when we need them. This choice was made as we argue that calculating the strides involves doing additional simple integer math on tiny arrays, and should not effect runtime significantly. The upside is that we can easier debug the program when the tools inevitably gives us problems by re-writing the kernel only and not have to worry about host code changes.

5 Optimizing for hardware using fixed point operations

As discussed in section 2.2 it would be beneficial to run simulations in a reduced precision environment. Here we want to leverage the fact that FPGAs are extremely flexible with regards to bit widths compared to the hardware we are used to in CPUs and GPUs.

It is needed to consider what errors, in terms of precision loss in the solution to our differential equations, we get by reducing the bits on our operands. To do this, we built an emulator where we can run our algorithm trough, to test the response to reduced precision schemes.

5.1 Introdocution to fixed point algorithms

There have been various pushes to convert GPUs to operate on reduced precision floating points[32], which have interesting applications for machine learning. Other projects[24][41] aim to run fluid dynamics in a reduced precision environment.

In FPGAs we can do even better than converting doubles to single precision, as we do not have any hardware limitations regarding the composition of word sizes. By freely mixing between 64, 50, n and 32 bit numbers, we can theoretically squeeze out performance by choosing an optimal set of variables.

Converting a given algorithm into fixed point operations is not trivial. By converting to fixed point we introduce a defined error, max size and minimum size representable. We need to select the minimal viable size, for our numbers to go trough the calculations while keeping the error tolerable and without

triggering any over or under flows. Lets explore an example to illustrate the ways our choices affects our algorithm. We want to calculate the mean of two numbers x_0 and x_1 both between 0 and 15:

$$y = (x_0 + x_1)/2 \tag{78}$$

To actually do this calculation, we need to store the intermediate result $x_0 + x_1$. In pseudo code this looks like:

```
tmp = x_0 + x_1
y = tmp / 2
```

Writing it this way, we need our temp variable to be a minimum of 5 integer bits to contain the dynamic range of the sum. Instead we could have written it like:

```
tmp1 = x_0 / 2 # we need only 3 integer bits here!
tmp2 = x_1 / 2
y = tmp1 + tmp
```

This required another operation but it allowed us to keep everything expressable in smaller variables. It turns out that this rearrangement does not result in the same outputs due to rounding errors. This can be seen as in the second example, we are throwing out information by dividing twice, while in the first example we only do it once. While these re-arrangement type transformations could also be an avenue for speed-ups, we neglect to analyze this type type of transformation in this discussion.

In the above example it was quite obvious what sizes we needed to contain the dynamic range of our problem without overflow. And in the same vein, we could easily have gotten constraints on the amount of fractional bits needed to match a given error tolerance. If we had a really complicated algorithm it quickly becomes a hard problem. In principle every operation in our code could be optimized to a minimum bit size. Also note that we needed an *exact* implementation to reason authoritatively about the precision, as the same mathematical expression resulted in different behaviour when implemented. To gain any confidence in our analysis we therefor opt to use emulation.

Existing simulators are quite simple[13], which does not lend them to easily be used for our purposes of emulating somewhat complicated array operations. To remedy this, we implement a subclass for the Numpy array the can handle doing fixed-point operations on arrays.

Veros also have another complication regarding fixed-point math: We have stability concerns regarding the numerical integration schemes used. If our system is already stable, in a Von Neumann[45] sense, then the errors should be dampened in time, and as long as we uphold our stability conditions we should be in an equivalent situation as had we been calculating in doubles. While this is true, we do know that fixed-point have different characteristics than doubles. As we have shown, addition is exact and multiplication has errors. These are differences, which coupled with non-linearity in the system at-large gives some concern regarding stability and accuracy.

To gain confidence in our choice of word sizes for our fixed point variables, we instead opt to simulate the system.

5.2 Fixed point emulation

To make this emulation as pain free as possible, we opt to repurpose the framework which our problem is already stated in. By subclassing a Numpy array to add functionality to turn any inputs into a given fixed point representation, we can (almost[4]) transparently emulate our existing code.

Furthermore, as the calculations are done in Numpy, they are performed in fast C code. This is relevant as we would like to see some scaling behaviour of our systems before we are confident that our selected fixed-point variables will work at real simulation scale.

[4]Some subtelty is required when assigning the result of a calculation to a `Fixed` objct. In some edge-cases we need to make sure we assign into the object `A[...] = B` and not `A = B`, if `B` needs to be converted to the `Fixed` type.

There is a strong limitation on this procedure, which is that the integers that are being operated on are fixed to be at-most 64 bits as of a hard limitation in the inner workings of Numpy. To overcome this, we can instead opt to work with regular Python integers which are protected from overflowing. The obvious downside to this is that we lose the fast execution speed of our emulator. We still get to keep the productivity benefits of being able to run and implement our algorithm in Numpy, and run emulation transparently.

The not-so-obvious downside to this hack is that internally Numpy requires unknown objects, like the inbuilt Python integer class, to have special methods implemented for each `ufunc`. Therefor we also need to extend the inbuilt integer class of Python to include methods which are expected by Numpy.

Putting these two classes together we arrive at a point where we can emulate Veros benchmarks in a fixed point environment. It turns out, that the dynamic range of the problem are quite large and even 64 bit integers are often too small to effectively capture the problems without either:

1. Over/ under flowing
2. Division by zero

The over/underflows are relatively straight forward: the bits available are not sufficient to represent the numbers. The division by zero issues arise from the fact that if we wish to divide some number with some number ε where $\varepsilon < 2^{-f_b}$ with f_b the number of fractional bits, we will get ε represented as a zero.

To alleviate these problems without resorting to huge fixed point variables, it might be possible to use mixed precision. Some parts of the code run in some collection of fixed type variable, others parts run in regular floating point. Another solution is to add additional meta-data to the fixed-point variable like offsets and scale.

It should be stressed that these considerations was found by experimenting randomly generated data in the unit interval. Real world data is distributed wildly different and especially will be much more correlated on short scales. This will have important effects when calculating derivatives. The fact that our variables are scaled to the unit interval could trivially be done to real-world data by a scale-factor to work in unitless dimensions.

5.2.1 Implemtation

Our basic strategy revolves around extending the powerful Numpy array object. In Numpy, array operations are performed trough what is known as Universal Functions (ufunc). We wish to high jack these functions, and perform our calculations on what is, to Numpy, integers contained in arrays. To us, these integers represent some real number according to some meta-data specifying the fixed-type variable, which we attach to the array. We will use the available hooks to always make sure that we do the necessary pre-processing to convert any "foregin" elements in our calculation to the required fixed-point type.

Because do not want to deal with overflows in intermediate results we can actually contain regular Python integers as the objects inside the underlying Numpy array. While this saves us from writing a big-integer implementation, it turns out that we also need to subclass the python integer and monkey patch various methods. Numpy expects the object contained in the array to have special methods needed for their ufuncs like `sqrt`.

Subclassing Numpy arrays is described in the Numpy manual[19]. There is some additional boilerplate code required compared to run of the mill subclassing. Numpy expects to call some specific dunder[5] functions on our objects. When creating a new object of our class, Numpy will call `__new__` on our object, and then run it trough `__array_finalize__`. In these methods, we can attach the necessary meta data to our object, and do the necessary conversion from input to fixed point type. The final

[5]Dunder (double underscore) methods are functions like __add__ which implements operator overload in Python.

result is an object of the `Fixed` type which contains a Numpy view filled with Python integers and has attached meta-data about its fixed point properties.

Now we need to write a `__array_ufunc__` method which handles computing ufuncs when our array is involved. Here we make sure to check any arguments for their class, and cast to the `Fixed` type if necessary. The actual result of the calculation is done trough the usual Numpy machinery, and we take a `Fixed` view of the resulting data.

5.2.2 Emulation results

We can simulate running a smaller, but still significant, part of the target benchmark in fixed-point arithmetic. Here, we set up a non-trivial implicit method to solve for derivatives on a 3d grid. This is achieved by first doing arithmetic operations on our arrays in the fixed type to find the inputs to our diagonals. We then convert our internal representation to their equivalent floating point and solve the resulting tri-diagonal system with lapack trough scipy. We choose 18 bit integer part as that have been verified experimentally to not result in overflow for this particular seed of randomly generated input data.

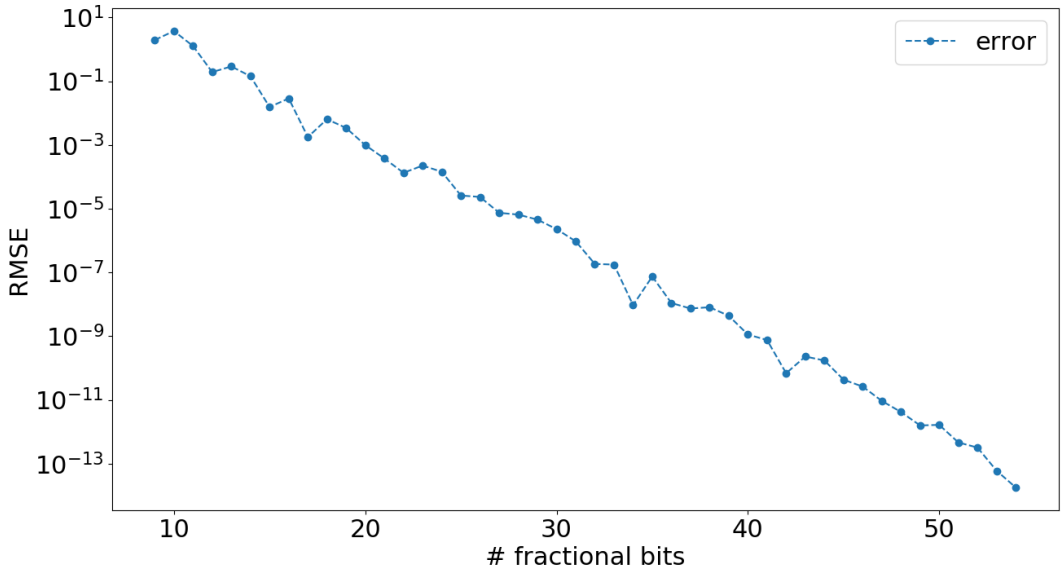

Figure 9: RMSE of the fixed point implementation in the sample benchmark compared to a reference double implementation.

The fact that we have an exponentially decreasing error is not surprising as increasing the bits in the fractional part increases the precision by $2^{-n_{frac}}$ for each individual number. It is significant that the error does not seem to accumulate after doing various arithmetic operations on the numbers.

6 Implementing hardware kernels

Available to us is various methods of transferring data into the FPGA:

1. Go trough main memory
2. Kernel to kernel streaming
3. Host to kernel streaming

These are roughly ordered in terms of complexity to implement.

In this project we have chosen to implement scheme one. We have tried to different kinds of implementation. The simplest version possible where we transfer the data associated with just a single operation and wait for the answer. In this version, the host controls execution flow of the code. And a less naive version where we start by transferring all the data we need, and then control the entire calculation from the FPGA. In this version, we create functions which does the job of our arithmetic kernels.

Controlling the flow of execution on the FPGA is going to look like a kernel-to-kernel implementation, but we avoid uncertainty in the tools ability to handle very large number of kernels[50]. The downside to having the arithmetic operations be implemented in HLS functions instead of kernels is that it would be more difficult to swap kernels for highly optimized RTL versions. This is manifested in the fact that while function arguments are synthesized to RTL as streams, we do not control the properties of these interfaces, like we do with kernels.

Doing host to kernel streaming is not implemented. To do this, we need a QDMA shell which is an experimental feature only currently available in high-end Alveo U280 boards[48]. Even then, the features we are looking to exploit in the FPGA are the deep pipelines. Being able to interact directly with our accelerator from the host is convenient if we need to take control of our computation to e.g respond to an outside signal with very low latency. This is not relevant for our use case, as we are worried about throughput not latency.

6.1 Go trough main memory

To implement this approach we must specify which "port" of our FPGA is paired with what input in our kernels in a configuration file. We have some limitations: each memory bank supports 15 ports, and the FPGAs we are targeting have 4 memory banks, for a total of 60 ports of input/output. Furthermore, we also have some low-latency/low-troughput ports we can use for control variables like strides and offsets. In practical terms, we write our kernel function signature like:

```
extern "C" void add4d(
    double* A,
    double* B,
    double* out,
    int* broadcasting_information)
    {
    ...
}
```

For the majority of our kernels, we then need 3 ports: one for each input, and one for the output. We can then map each I/O in a configuration file like so:

```
sp=add4d_1.broadcasting_information:PLRAM[0]
sp=add4d_1.A:DDR[0]
sp=add4d_1.B:DDR[1]
sp=add4d_1.out:DDR[2]
```

Scalar arguments does not need to be mapped, as these can be passed directly to the kernel without further adjustment needed in host or kernel code. Note also that we are using a different RAM block for each data array specificed by the number in the bracket. This means we can transfer data from A and B concurrently.

After this setup, we can invoke the kernel from our host code via an OpenCL[23] call.

6.2 Kernel to kernel streaming

We can implement kernel to kernel streaming, that is, data "flows" trough the calculation hardware without touching the slow main memory of the FPGA. To achieve this we must do a number of things:

1. Setup our host code OpenCL calls
2. Modify our kernel arguments to be streaming types

3. Supply a config file describing the connections during the linking stage of compiling our hardware

Obviously this puts some limits to our capabilities of accelerating "on-the-fly", as we need to re-link our program on every change. A process that takes on the order of hours. If we imagine a workflow of writing Numpy code and wishing to accelerate on an FPGA, we coin the new term NIT (Not In Time) compilation. While somewhat tounge-in-cheek, this system would still be valuable.

While step 1 and 2 are straightforward to implement step 3 has some additional worrisome features. As we are essentially routing the connections between kernels, we can not reuse kernels in a hardware sense. We must create a new adder kernel for each addition needed in our program. This process will eventually eat all the available space on the FPGA. Additionally we have limits on the amount of ports we can connect memory to. It is not clear[50] if we can bypass this limitation with the streaming arguments, or if this is a hard limitation.

6.3 Optimizing hardware kernels

Optimizing the kernels to run fast is an important step in accelerating Veros at large. Making individual kernels run fast on an FPGA, compared to similar hardware like GPUs or even CPUs, is not a trivial case. We are fighting against very optimized operations that have been studied for many years. CPUs are already very good at implementing vector operations, and GPUs are tailor made to exploit the inherent parallelism in the problem. Essentially we are fighting a losing battle by re-implementing what has already been implemented very efficiently in existing hardware.

We do have a few tricks up our sleeve: FPGAs can build custom pipelines to more efficiently carry out the needed calculations, which are not that trivial. Running a two-input one-output arithmetic kernel in four dimensions, we need to perform for each element in our array:

```
1 A_ind = (i + A_offset[0])*A_stride[0] + (j + A_offset[1])*A_stride[1] + (k +
      A_offset[2])*A_stride[2] + (l + A_offset[3])*A_stride[3] + A_lin_offset
2 B_ind = (i + B_offset[0])*B_stride[0] + (j + B_offset[1])*B_stride[1] + (k +
      B_offset[2])*B_stride[2] + (l + B_offset[3])*B_stride[3] + B_lin_offset
3 O_ind = i*out_stride[0] + j*out_stride[1] + k*out_stride[2] + l*out_stride[3] +
      out_lin_offset
```

In total we need to do:

- 10 additions (caused by offsets into the current index, 5 for each input. One on each index + a general offset)
- 12 multiplications (Multiply by strides to get linear index. Four for each input/output)
- 1 arithmetic operation to calculate the result.

For each element. In an optimized kernel all these pointer arithmetic operations should all execute in approximately one clock cycle (depending if we can meet timing), as there is no dependencies between the data. Alternatively we could reduce the number of operations needed by saving the results of the slow-moving strides between loop iterations.

The naive implementation sketched above will be labeled `baseline` so we can compare various implementations against it.

6.3.1 Pipelining the loop

Because we are doing very simple operations with no loop carried dependencies arithmetic operation kernels should be almost trivial to pipeline. We do have one problem tough: We do not know our loop bounds at compile time, as we will execute the kernel multiple times. To get our data to execute in a tight pipeline, we will need to both read and write from the same array in the same clock cycle. As arrays get implemented in hardware with one port, it is generally not possible, and the way to get around this limitation is to partition the array to create smaller memory structures which can then

be read and written to individually. We can not partition our arrays as we do not know their size. It should be possible to circumvent this limitation by writing buffers which can then be partitioned.

In effect we do not expect simply applying a pipeline to the loop to have any significant speed-ups.

6.3.2 Loop Tiling

Our goal is to process our array in blocks of known size. Doing this, we can avoid the problems we faced in pipelining the loop. Loop tiling is achieved by creating local memory and copying data into these local buffers. An important reason is that we are accessing main memory less often for an increase in throughput by requesting more data in each transaction. Potentially the tools should automatically detect this optimization if we "just" write a tiled loop, but leave no room compiler magic, we will implement the inputs as arrays of 512 bit words as recommended by Xilinx[50].

This allows us to process our arrays in blocks of 8 doubles at a time. The downside is that we also need to make sure our data fits on a 512 bit boundary and handle edge cases. Another important problem is that as we wish to use broadcasted operations, we can run into the case where we need to select from not contiguous indices. In this case, packing the data into larger tiles actually works against us if we are only interested in a single piece of information inside the block.

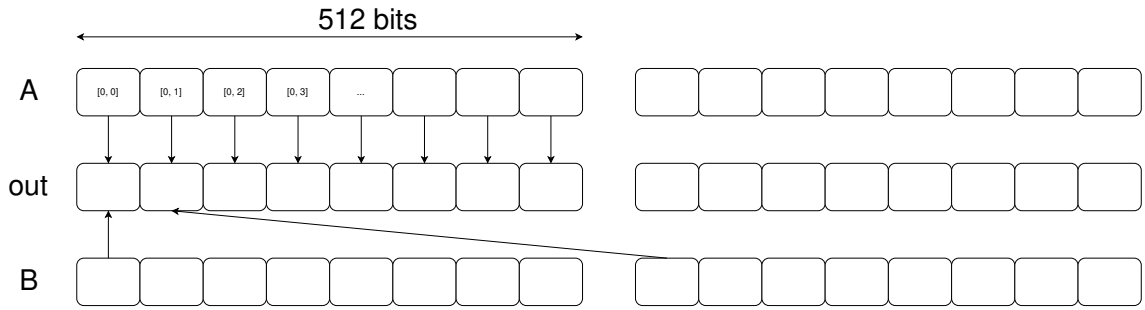

Figure 10: Demonstration of why we need to do something smart, when selecting which data to send to FPGA.

Furthermore we have another complication: We need to interpret the binary data inside a 512-bit word as 8 doubles. An operation that seems simple enough, but in most day-to-day programming situations we do not mess around with the specifics bits our or variables, as this would quickly lead to very unexpected results. For this reason these operations are very rare[6], and the C/C++ standards does not provide strong guarantees towards their behaviour. The tools we would need are `reinterpret_cast` in C++ or something like `double x = * ( double * ) &y` if we wanted to interpret the bits of `y` as a double. we opted to use an open source implementation by Falcon[10][7] computing to do these very low-level bit operations, as this is a known-good implementation for interfacing with the Xilinx FPGAs.

6.4 Results

We measure run times for the discussed implementations and compare to a reference Numpy implementation running on a laptop computer.

[6] A famous example of this type of bit hacking is seen in the 1/sqrt(x) approximation used in old computer games[29]

[7] In this paper, and in the code shared on Github, the authors attribute the code to an open source release by Falcon Computing[52]. Nowhere else can we find references to this implementation. The company itself is hard to find good references on, but apparently Xilinx has acquired them. It also appears that the CEO of Falcon computing is the author of the paper.

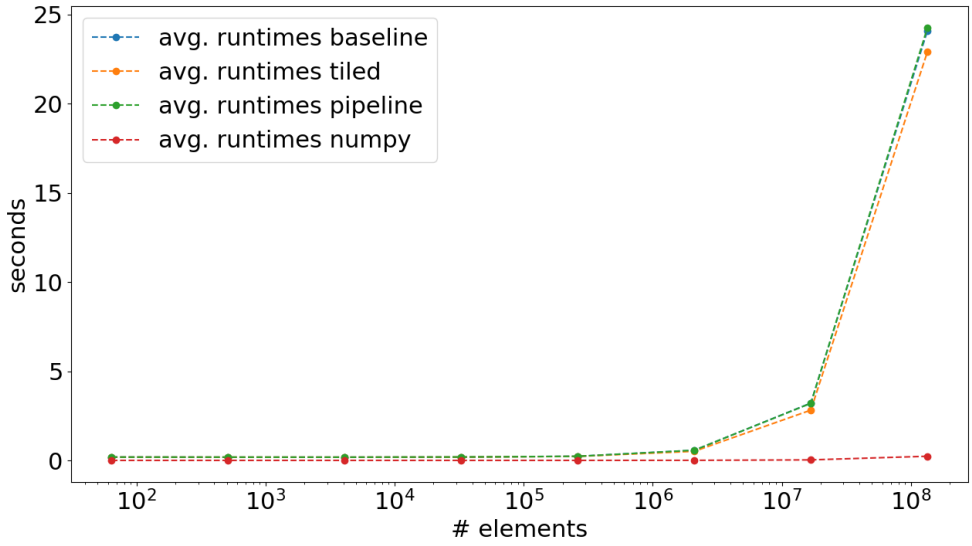

Figure 11: Run times for various implementations of a vector adder kernel. FPGA benchmarks are run on an U250 Xilinx Alveo board. Kernels are implemented in HLS. Numpy benchmark is run on my laptop which includes an Intel i5-7200U processor.

Here we see that the FPGA implementation is significantly slower than the reference Numpy implementation even tough we are running on much cheaper hardware. We notice that the tiled implementation is a slight improvement over the others. To get a better sense for this relationship, we plot the speedups compared to the baseline reference. Here we neglect to plot the speedup of Numpy as this would make the plot unreadable, but the speedups measured here is roughly 100x compared to the FPGA baseline. We also produced a naive C implementation compiled with `-O0` (turning off a lot of optimizations) in `gcc`. The performance of this optimized code was roughly half of Numpys execution speed giving an approximate 50 times speedup over the FPGA solution.

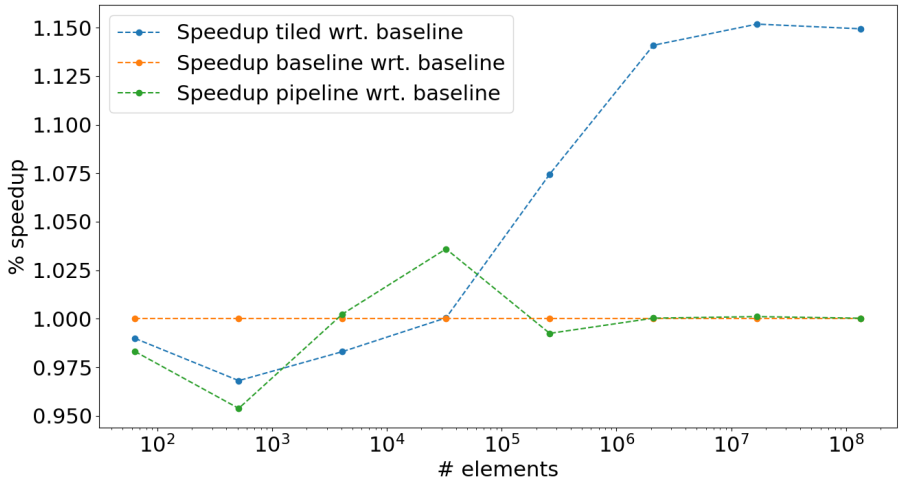

Figure 12: Speedups for various implementations of a vector adder kernel. FPGA benchmarks are run on an U250 Xilinx Alveo board. Kernels are implemented in HLS. Speedups for the reference Numpy implementation is roughly 100x.

We see that for large arrays the overhead induced by unpacking the 512 tiled words are offset by the increase in throughput. The trend does seem to flatten out, which suggests that we can not beat the

```
================================================================
OpenCL Binary:     add4d
Kernels mapped to: clc_region

Timing Information (MHz)
Compute Unit  Kernel Name  Module Name  Target Frequency  Estimated Frequency
------------  -----------  -----------  ----------------  -------------------
add4d_1       add4d        read_1       300.300293        411.015198
add4d_1       add4d        compute      300.300293        430.848755
add4d_1       add4d        write_1      300.300293        411.015198
add4d_1       add4d        add4d        300.300293        238.322205

Latency Information
Compute Unit  Kernel Name  Module Name  Start Interval  Best (cycles)  Avg (cycles)  Worst (cycles)  Best (absolute)  Avg (absolute)  Worst (absolute)
------------  -----------  -----------  --------------  -------------  ------------  --------------  ---------------  --------------  ----------------
add4d_1       add4d        read_1       1               75             75            75              0.250 us         0.250 us        0.250 us
add4d_1       add4d        compute      1               8              8             8               26.664 ns        26.664 ns       26.664 ns
add4d_1       add4d        write_1      1               73             73            73              0.243 us         0.243 us        0.243 us
add4d_1       add4d        add4d        undef           undef          undef         undef           undef            undef           undef

Area Information
Compute Unit  Kernel Name  Module Name  FF     LUT    DSP  BRAM  URAM
------------  -----------  -----------  -----  -----  ---  ----  ----
add4d_1       add4d        read_1       2532   1206   0    0     0
add4d_1       add4d        compute      7919   6546   0    0     0
add4d_1       add4d        write_1      1969   690    0    0     0
add4d_1       add4d        add4d        25976  20986  0    190   0
```

Figure 13: Analysis by the hardware emulation tool. These are suggested run times by the hardware emulation stage of the tools.

laptop by going to even larger scales. It should also be noted that the speedup achieved is very modest at around 15%, which does suggest that there could be further optimizations which have not been found, as optimizations on the order of thousands have been seen in the literature[28][10]. These kinds of speedups have been achieved on more complicated tasks, which might have a larger optimization space. It is important to stress that this benchmark was a "best-case" for the tiled implementation, as we performed the benchmark on a trivial operation without any broadcasting handling of edge cases.

Looking at results from hardware emulation, we see that we are spending most our of clock cycles reading and writing: It seems that we can initiate the functions every clock cycle, and since the total runtime is 156 cycles for computing 8 doubles operating at approximately 240Mhz we expect to have a throughput of:

$$r = 8 \cdot \frac{240\text{Mhz}}{156\,\text{cycles}} \approx 1.2 \cdot 10^7 \frac{\text{ops}}{\text{second}} \tag{79}$$

In practice, we achieved around half that:

$$r_{exp} = \frac{512^3\,\text{ops}}{23\,\text{second}} \approx 5.8 \cdot 10^6 \frac{\text{ops}}{\text{second}} \tag{80}$$

This corresponds to a rough bandwidth of:

$$b = 5.8 \cdot 10^6 \cdot 2 \cdot 8\,\text{bytes} \approx 100\,\text{MBs} \tag{81}$$

A bandwidth on the order of hundreds of megabytes per second is very slow compared to the interfaces we are running on, which have a combined read-write bandwidth of 77 GB/s according to the specifications[47]. An obvious solution to this speed problem is to unroll some of the loops to run in parallel until we can saturate our bandwidth. This comes at the cost of area in our FPGA which we will need to tune, when we are implementing many kernels.

6.5 Unrolling the loop

We try to unroll one of the loops to hopefully get closer to saturating the bandwidth of the FPGA. A rough order of magnitude calculation for the bandwidth on each channel gives:

$$\frac{77\,\text{GB/s}/4/2}{50\,50\ \text{MB/s}} \approx 200 \tag{82}$$

That is, if we have around 200 copies of the loop, we should be close to saturating the bandwidth.

In compiling this code we have chosen to unroll 32 times which should give plenty of speedup while not pushing the hardware to its absolute limits. To land on the 32 times unroll figure we must consider the

maximum transaction size of the AXI-4 interface[**hlsmanual**] of 4KB. As our words are a package of 8 doubles totaling 64 bytes, an unroll factor of 64 would land us at exactly the limit of the maximum transaction size. The reason for then choosing 32 was that we did not consider the fact that the interfaces for our inputs was actually connected to two separate RAM blocks 6.1.

We notice that the area information given by the tool does not correspond to around 32 times more area used as expected. A warning is also given:

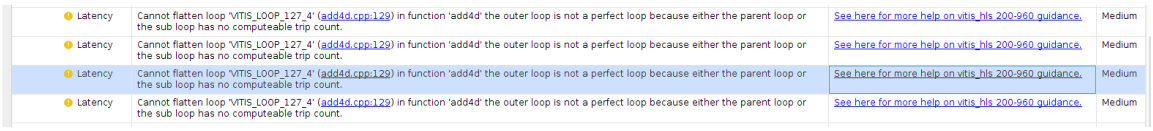

Figure 14: Warning thrown by Vitis HLS hardware emulation. We get 32 copies of the warning as we tried to unroll this loop 32 times. The loop bounds are in-principle not known at compile time as we wish to make the system dynamic for different sizes, but also to reuse to kernel for multiple operations throughout the computation.

It seems that the tool has trouble unrolling the loop as it does not know its bounds. While this is sensible if we tried to completely unroll the loop, as we would then *need* to know how many copies to create, we should be able to unroll by a factor of N[**hlsmanual**].

Running the test on real hardware we do see an improved run time, but not an ideal-case 32 times speedup:

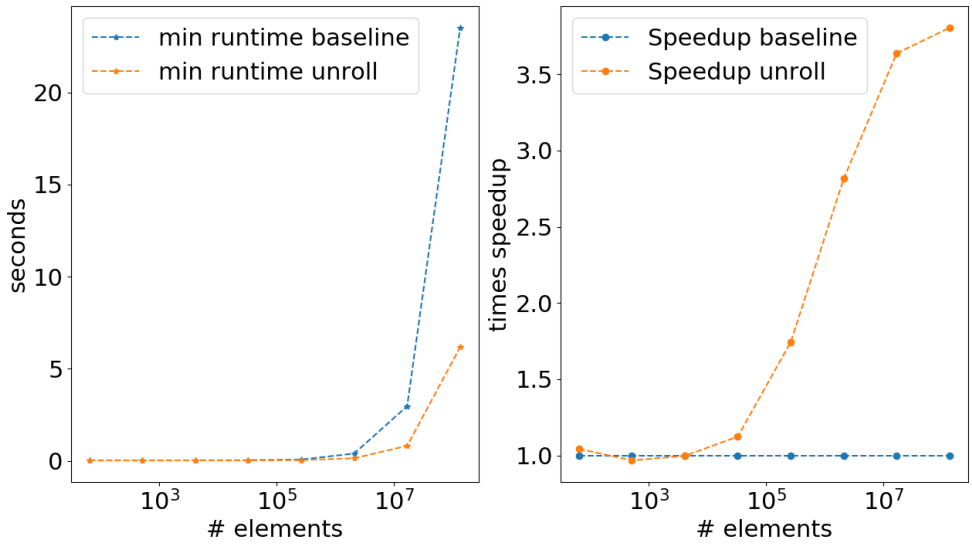

Figure 15: Run time and speedup of the 32-times unrolled version of the tiled benchmark. We see that we get an approximate 3.8 times speedup compared to the naive baseline implementation.

Even tough this is a speedup compared to the tiled version, it is still slow compared to much inferior hardware. The fact that we did not see a speed up matching the unrolling factor suggests that the design is stalling. By changing nothing in the compute part of our kernel it is likely happening in relation to the reading and writing from the memory ports.

This could be caused by the compiler not recognizing that it is possible to do larger transactions when requesting memory to the kernel. Writing this part more explicitly is possible, by requesting all 4KB of data and then delegating this to an unrolled loop that calls the compute function. What this amounts to is essentially to create a larger tiling wrapping the smaller tiling.

7 Running the full benchmark

7.1 Emulation

We could without any major issues[8] write kernels and host code that was functionally correct as per testing in software emulation. The correctness was verified by examining sums of the arrays (and at various sizes) after going trough the computations which strongly suggests the correctness.

Hardware emulation proved to be more troublesome. While the kernels individually could be run in hardware emulation mode and providing valuable information about the produced hardware, when combining all kernels into one binary, we ran into trouble. We could successfully compile the binary, but running hardware emulation produced random segfaults. The nature of these segfaults were hard to debug and related to standard low-level libraries like `libcc` and `pthread`. To make matters worse, they got thrown at different places in the code on each rerun.

The hardware emulation works by producing some kernel binary which is then ran on a simulator. The core mechanism for running this simulator is that Vitis will write-to-disk a `simulate.sh` which is then executed by the run-time libraries when executing code with the `XCL_EMULATION_MODe=hw_emu` environment variable. This shell script also modifies the `PATH` variable to point to the shared versions of the aforementioned libraries. It turns out that Xilinx distributes (multiple) copies of these libraries in their packaging of Vitis, but no combination we tried yielded any results in fixing the segfaults.

It turns out, that even tough software emulation did not work, compiling, and running, on hardware did.

7.2 Hardware — Control flow on host

We first try to run the simplest version of the benchmark where the execution is controlled in the host. This requires a lot of memory transfer and large overheads.

When interacting with the FPGA, we would like to have aligned our memory to specific memory address offsets. If the requested memory is not aligned to a page, the run-time will automatically copy the memory to a new place in memory where the alignment is fixed. This will obviously slow the program down as we are now doing unnecessary reads and writes.

[8] Minor issues ranged from finding that the tri-diagonal solver supplied in Vitis solver library is broken for all inputs not sized as a power of two, to magic environment variables before compilation will run.

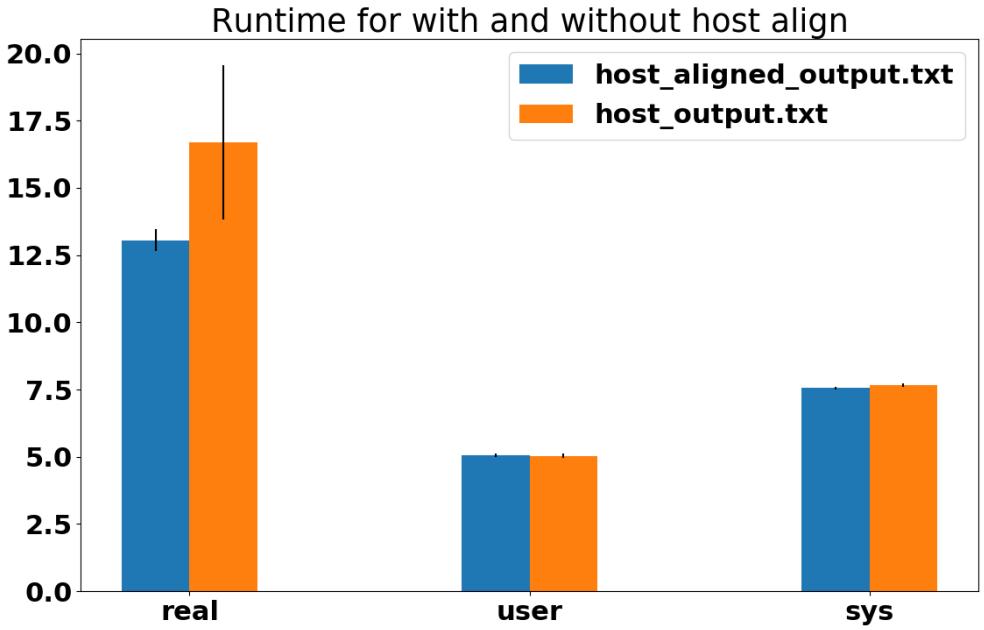

Figure 16: Running with and without page alignment shows a speedup. These are timed with the unix command `time`. Here a very small experiment is run with a grid size of approx. 800 elements. Run times compared to a laptop with an Intel i5-7200U processor running the reference Numpy implementation is in the order of 10 000 times slower.

7.3 Hardware — Control flow on FPGA

This version could not compile. The error messages were not helpful. Even tough we are not able to run the full benchmark, we can try to test the *idea*. We build a (much) smaller and less complicated system that still utilizes multiple kernels and test to see if the runtime is significantly faster for the combination of them, than if ran individually.

8 Discussion and conclusions

To have a general system that can accelerate large scale climate simulations, like Veros, on FPGA is not realised in this work. In section 6.3 we have developed the necessary kernels and broadcasting framework for accelerating mostly any Numpy functions to run on an FPGA, but this system does not stand on its own. Currently the code is mostly still manually transcribed. This transcription amounts to rewriting every arithmetic operation. Either in host code calls or in a kernel using pre-supplied functions. While slightly annoying there is also the problem of manually having to manage temporary memory in cases where temporary results are not explicitly stored in the original Numpy implementation. This means a person transcribing a Numpy line like:

```
1 A = (B[0:1] + B[:-1]) + C[:] * 2
```

Must make choices of where to keep one of the intermediate results.

It might seem like an obvious idea to call the host code from a transparent layer build ontop of the Numpy arrays somewhat like we build the fixed-point emulator. But this is also problematic, as this locks you into doing only run-time optimized calls. Essentially forever locking you into the version where the host controls the execution flow. While this is fine in principle, we have then demoted our FPGA to be essentially a GPU. A solution which is not very wise, as GPUs are already a very optimized ASIC build to solve essentially the problem of working on arrays. It should be possible to construct a transpiler that takes Numpy code and produces a kernel which makes calls to our HLS

functions and automatically writes the program saving the user some time. The worthwhileness of this effort depends strongly on the expected reward: How much faster, or cheaper (in terms of power used) can we run or code on FPGAs?

Benchmarking simple kernels in section 6.4 operating on arrays we saw expected disappointing result for the naive implementations. This is very much in-line with known results from the literature[28][10]. Many transformations to the code are needed to gain performance. In our case we have an somewhat more difficult task of optimizing embarrassingly (simple and) parallel problems which are tailor made to run fast on GPUs. Another exacerbating fact is that we have a lot of variation in our loop bounds which the tools do *not* like.

Using fixed point, or other representations, does seem to provide some benefits in theory. Lots of work would need to go into emulation to figure out the optimal word sizes. Achieving a speedup trough this avenue would further complicate any efforts to build a general way to accelerate Numpy codes, as you would now have an extra layer in you analysis of the code and what the hardware would correspond to. This problem could potentially be solved by posits, but this is not inherently an FPGA specific solution, as silicon manufactures could easily start building posit-capable hardware if the demand was large enough. FPGAs would be an obvious avenue for proof-of-concept hardware implementation of this new number format in large scale experiments. And if successful, FPGA manufactures could implement hard logic blocks akin to the floating processing units we are starting to see implemented[20].

The naive version of the benchmark proved to be *extremely* slow, but does prove that it is possible in-principle as shown in section 7. The improved version disappointingly did not compile, but produces functionally correct results in software emulation. Here we are fighting a losing battle with the tools, as we can never quite get the information we need. A problem shared in all stages of the project. Here we would really like to test if we could gain any benefits out of the deep pipelining ability of the FPGA.

Alternatively one could write the program "by-hand" in HLS. Here we would not write functions handling arithmetic but rewrite the program using explicit loops. This approach have other downsides besides the inherent difficulty of writing optimized HLS code. It requires manual transcription of large code bases and also requires maintaining two separate code bases. While this solution might provide larger performance increases it fails by design in any practical scientific setting.

9 Further work

9.1 More general accelerator

Going from what has been shown here to a more general acceleration platform should be possible. Parsing Numpy code to produce either host code calling the FPGA or producing a work kernel should not be too difficult. We imagine the process being:

- Parse Numpy algorithm into correct order of operations.
- Extract strides from the `range` objects
- Produce equivalent code. (HLS code + host code to call it)
- Compile
- Run

This process can obviously not be run "in-time" as the compilation stage is very long.

For this to be worthwhile we must gain some benefits of running on an FPGA compared to running on CPU or GPU. Accelerating on GPU has some clear workflow improvements in a lot of computing areas, where you want to experiment with large amounts of data in a quick development cycle. FPGAs obviously fail in this regard, as the compile time introduces a huge computational cost which have to

be paid up-front. It seems then that the Numpy-HLS transpiler must then produce good results to be worth the investment, compared to running on GPU.

9.2 Running fixed-point hardware

FPGA hardware gives you the very unique opportunity to execute variable width fixed-point code fast. To actually figure out the width of each variable is not trivial as your algorithm grows to any considerable size.

To remedy this, we can run emulations of the system. Here we can select a unique fixed point representation for each operation. Writing this as an equation we can

$$V_{fxp}(S_i, w_1, w_2, w_3, ..., w_n) \approx S_{i+1} = V_{double}(S_i) \tag{83}$$

Where V is one timestep of Veros, w_i is a fixed-point variable (bit width, num. fixed, num. exp, scaling, offsets etc) and S_i is the Veros state variable for a given time step. We now want to minimize the difference between the reference and fixed point version, but also weighting the total bit-length of our fixed-point variables w_i. The simplest cost function looks something like:

$$C = \alpha \cdot |V_{fxp} - V_{double}| + \beta \cdot \sum_i w_{ibitlength} \tag{84}$$

The choice of minimizing the bit-length is made as this seems to be the most direct measure of performance. Smaller bit-lengths means higher throughput and faster calculations.

We should now be able to minimize this cost function trough standard means like gradient descent.

An obvious problem to this approach is that we inherently cannot scale this to any real-world size because of the emulation needed to calculate V_{fxp}. Here we must rely on the selected variables generalizing and not suffering from any numeric defects at large scale.

References

[1] Albert Alonso. "Porting of exisitng simulations". unpublished. 2021.

[2] Cambridge centre for Alternative Finance. *Cambridge Bitcoin Electricity Consumption Index*. 2021. URL: https://cbeci.org/ (visited on 03/29/2021).

[3] Amazon. *Amazon EC2 F1 Instances*. 2021. URL: https://aws.amazon.com/ec2/instance-types/f1/ (visited on 03/31/2021).

[4] AMD. *AMD to Acquire Xilinx*. 2021. URL: https://www.amd.com/en/corporate/xilinx-acquisition (visited on 03/29/2021).

[5] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[6] Anders S. G. Andrae and Tomas Edler. "On Global Electricity Usage of Communication Technology: Trends to 2030". In: *Challenges* 6.1 (2015), pp. 117–157. ISSN: 2078-1547. DOI: 10.3390/challe6010117. URL: https://www.mdpi.com/2078-1547/6/1/117.

[7] Havard Block. *Historical Cost of Computer Memory and Storage*. 2017. URL: https://hblok.net/blog/storage/ (visited on 05/20/2021).

[8] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[9] Jason Cong et al. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 30 (May 2011), pp. 473–491. DOI: 10.1109/TCAD.2011.2110592.

[10] Jason Cong et al. "Understanding Performance Differences of FPGAs and GPUs". In: Apr. 2018, pp. 93–96. DOI: 10.1109/FCCM.2018.00023.

[11] Annie Cuyt et al. "A Remarkable Example of Catastrophic Cancellation Unraveled". In: *Computing* 66 (May 2001), pp. 309–320. DOI: 10.1007/s006070170028.

[12] R. Dennard et al. "Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions". In: *Proceedings of the IEEE* 87 (1999), pp. 668–678.

[13] francof2a. *fxpmath*. 2020. URL: https://github.com/francof2a/fxpmath (visited on 05/12/2021).

[14] Google. *Cloud TPU*. 2021. URL: https://cloud.google.com/tpu (visited on 05/20/2021).

[15] Gustafson and Yonemoto. "Beating Floating Point at Its Own Game: Posit Arithmetic". In: *Supercomput. Front. Innov.: Int. J.* 4.2 (June 2017), pp. 71–86. ISSN: 2409-6008. DOI: 10.14529/jsfi170206. URL: https://doi.org/10.14529/jsfi170206.

[16] D. Häfner et al. "Veros v0.1 – a fast and versatile ocean simulator in pure Python". In: *Geoscientific Model Development* 11.8 (2018), pp. 3299–3312. DOI: 10.5194/gmd-11-3299-2018. URL: https://gmd.copernicus.org/articles/11/3299/2018/.

[17] Dion Häfner. *HPC benchmarks for Python*. 2019. URL: https://github.com/dionhaefner/pyhpc-benchmarks (visited on 05/12/2021).

[18] Cong Hao et al. "A Hybrid GPU + FPGA System Design for Autonomous Driving Cars". In: Oct. 2019, pp. 121–126. DOI: 10.1109/SiPS47522.2019.9020540.

[19] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[20] Don Lahiru Hettiarachchi, Venkata Salini Priyamvada Davuluru, and Eric Balster. "Integer vs. Floating-Point Processing on Modern FPGA Technology". In: Jan. 2020, pp. 0606–0612. DOI: 10.1109/CCWC47524.2020.9031118.

[21] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[22] Robert Kern. *line_profiler and kernprof*. 2021. URL: https://github.com/pyutils/line_profiler (visited on 05/16/2021).

[23] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1*. Ed. by Aaftab Munshi. 2011. URL: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[24] M. Klöwer, P. D. Düben, and T. N. Palmer. "Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model". In: *Journal of Advances in Modeling Earth Systems* 12.10 (2020). e2020MS002246 10.1029/2020MS002246, e2020MS002246. DOI: https://doi.org/10.1029/2020MS002246. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2020MS002246. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2020MS002246.

[25] Mads Ruben Burgdorff Kristensen et al. "Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster". English. In: *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*. 2013.

[26] Banwari Lal, Narendra Khatri, and Abhishek Sharma. "An Analytical Review on FPGA Based Autonomous Flight Control System for Small UAVs". In: Mar. 2016. DOI: 10.1109/ICEEOT.2016.7754907.

[27] Jonathan Leake. *Met Office forecasts a supercomputer embarrassment*. 2009. URL: https://www.thetimes.co.uk/article/met-office-forecasts-a-supercomputer-embarrassment-5kzcxq8n76w (visited on 04/05/2021).

[28] Johannes Licht, Simon Meierhans, and Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In: (May 2018).

[29] Chris Lomont. *Fast inverse square root*. Tech. rep. 2003.

[30] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[31] O. Mencer et al. "The history, status, and future of FPGAs". In: *Communications of the ACM* 63 (2020), pp. 36–39.

[32] Paulius Micikevicius et al. "Mixed Precision Training". In: (Oct. 2017).

[33] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[34] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Selected as an IEEE Micro TopPick. IEEE Press, June 2014, pp. 13–24. ISBN: 978-1-4799-4394-4. URL: https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/.

[35] Python. *The Python Profilers*. 2021. URL: https://docs.python.org/3/library/profile.html (visited on 05/16/2021).

[36] Junnan Shan et al. "Exact and Heuristic Allocation of Multi-kernel Applications to Multi-FPGA Platforms". In: June 2019, pp. 1–6. DOI: 10.1145/3316781.3317821.

[37] P.R. Shukla et al. "IPCC, 2019: Climate Change and Land: an IPCC special report on climate change, desertification, land degradation, sustainable land management, food security, and greenhouse gas fluxes in terrestrial ecosystems". In: (2019).

[38] Alan Jay Smith. "Cache memories". In: *ACM Computing Surveys* 14 (1982), pp. 473–530.

[39] David Soergel. "Rampant software errors undermine scientific results". In: *F1000Research* 3 (Nov. 2015), p. 303. DOI: 10.12688/f1000research.5930.1.

[40] T.N. Theis and H.-S. Philip Wong. "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science Engineering* 19 (Mar. 2017), pp. 41–50. DOI: 10.1109/MCSE.2017.29.

[41] Filip Váňa et al. "Single Precision in Weather Forecasting Models: An Evaluation with the IFS". In: *Monthly Weather Review* 145.2 (2017), pp. 495–502. DOI: 10.1175/MWR-D-16-0228.1. URL: https://journals.ametsoc.org/view/journals/mwre/145/2/mwr-d-16-0228.1.xml.

[42] Wim Vanderbauwhede and Khaled Benkrid. *High-Performance Computing Using FPGAs*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1493943103.

[43] Mário Véstias and Horácio Neto. "Trends of CPU, GPU and FPGA for high-performance computing". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–6. DOI: 10.1109/FPL.2014.6927483.

[44] Kazutoshi Wakabayashi. "C-based behavioral synthesis and verification - Analysis on industrial design examples". In: Jan. 2004, pp. 344–348. DOI: 10.1145/1015090.1015177.

[45] Thomas Tomkins Warner. *Numerical Weather and Climate Prediction*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511763243.

[46] Feibao Xiao et al. "Posit Arithmetic Hardware Implementations with The Minimum Cost Divider and SquareRoot". In: *Electronics* 9.10 (2020). ISSN: 2079-9292. DOI: 10.3390/electronics9101622. URL: https://www.mdpi.com/2079-9292/9/10/1622.

[47] Xilinx. *Alveo U200 and U250 Data Center Accelerator Cards Data Sheet*. 2021. URL: https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf (visited on 05/16/2021).

[48] Xilinx. *Alveo U280 Data Center Accelerator Card Data Sheet*. 2021. URL: https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pd (visited on 05/16/2021).

[49] Xilinx. *Vitis AI User Guide*. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_3/ug1414-vitis-ai.pdf (visited on 05/16/2021).

[50] Xilinx. *Vitis HLS*. 2021. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/gnq1597858079367.html (visited on 05/20/2021).

[51] Xilinx. *Vitis Unified Software Development Platform 2020.2 Documentation*. 2020. URL: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/runemulation1.html#ariaid-title7 (visited on 04/05/2021).

[52] Xilinx. *Xilinx acquires Falcon Computing*. 2021. URL: https://www.xilinx.com/about/xilinx-ventures/falcon-computing.html (visited on 05/20/2021).

A Stability of implicit scheme

We wish to show that the FDE

$$T_j^{n+1}(1+2\lambda) - \lambda\left(T_{j-1}^{n+1} + T_{j+1}^{n+1}\right) = T_i^n \tag{85}$$

is unconditionally stable. Inserting the usual Fourier assumption on the solution and demanding this holds for all wavenumbers we arrive at:

$$\rho^{n+1}(1+2\lambda) - \lambda(\rho^{n+1}\,\mathrm{e}^{-ik} + \rho^{n+1}\,\mathrm{e}^{ik}) = \rho^n \tag{86}$$

$$\rho(1+2\lambda) - \lambda\rho 2\cos(k) = 1 \tag{87}$$

$$\frac{1}{1+2\lambda-2\lambda\rho\cos(k)} = \rho \tag{88}$$

$$\frac{1}{1+4\lambda\sin^2(k/2)} = \rho \tag{89}$$

Where we used the identity that $1-\cos(k) = \sin^2(k/2)$. As the denominator is always positive ρ will always be smaller than one which means that the stability is unconditional.